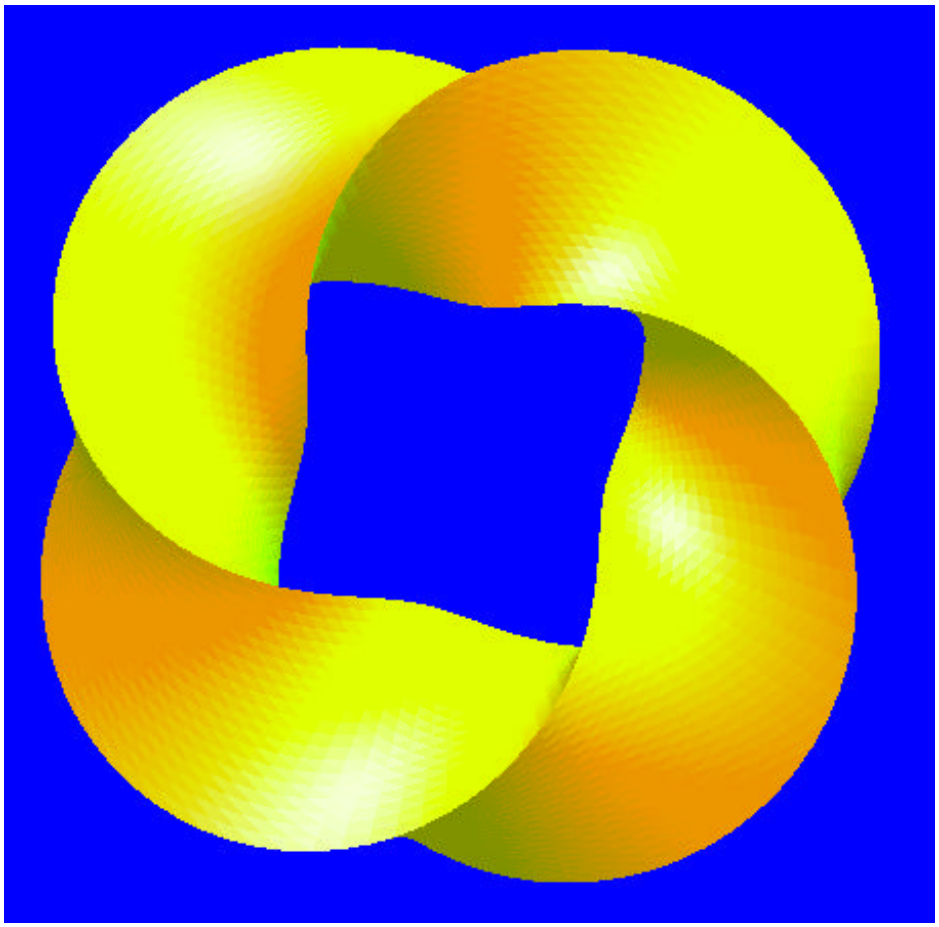
# Notes for a Computer Graphics Programming Course



Dr. Steve Cunningham
Computer Science Department
California State University Stanislaus
Turlock, CA 95382

These notes cover topics in an introductory computer graphics course that emphasizes graphics programming, and is intended for undergraduate students who have a sound background in programming. Its goal is to introduce fundamental concepts and processes for computer graphics, as well as giving students experience in computer graphics programming using the OpenGL application programming interface (API). It also includes discussions of visual communication and of computer graphics in the sciences.

The contents below represent a relatively early draft of these notes. Most of the elements of these contents are in place with the first version of the notes, but not quite all; the contents in this form will give the reader the concept of a fuller organization of the material. Additional changes in the elements and the contents should be expected with later releases.

## *CONTENTS:*

Principles of Modeling
- Introduction

Simple Geometric Modeling
- Introduction
- Definitions
- Some examples
  - Point and points
  - Line segments
  - Connected lines
  - Triangle
  - Sequence of triangles
  - Quadrilateral
  - Sequence of quads
  - General polygon
  - Normals
  - Data structures to hold objects
  - Additional sources of graphic objects
  - A word to the wise

Transformations and modeling
- Introduction
- Definitions
  - Transformations
  - Composite transformations
  - Transformation stacks and their manipulation
- Compiling geometry

Scene graphs and modeling graphs
- Introduction
- A brief summary of scene graphs
  - An example of modeling with a scene graph
- The viewing transformation
- Using the modeling graph for coding
  - Example
  - Using standard objects to create more complex scenes
  - Compiling geometry
- A word to the wise

Modeling in OpenGL
- The OpenGL model for specifying geometry
  - Point and points mode
  - Line segments
  - Line strips
  - Triangle
  - Sequence of triangles
  - Quads
  - Quad strips
  - General polygon
  - The cube we will use in many examples
- Additional objects with the OpenGL toolkits
  - GLU quadric objects
    - GLU cylinder
    - GLU disk
    - GLU sphere
  - The GLUT objects

    - An example
- A word to the wise
- Transformations in OpenGL
- Code examples for transformations
    - Simple transformations
    - Transformation stacks
    - Creating display lists

Mathematics for Modeling
    - Coordinate systems and points
    - Line segments and curves
    - Dot and cross products
    - Planes and half-spaces
    - Polygons and convexity
    - Line intersections
    - Polar, cylindrical, and spherical coordinates
    - Higher dimensions?

Color and Blending
- Introduction
- Definitions
    - The RGB cube
    - Luminance
    - Other color models
    - Color depth
    - Color gamut
    - Color blending with the alpha channel
- Challenges in blending
- Color in OpenGL
    - Enabling blending
    - Modeling transparency with blending
- Some examples
    - An object with partially transparent faces
- A word to the wise
- Code examples
    - A model with parts having a full spectrum of colors
    - The HSV cone
    - The HLS double cone
    - An object with partially transparent faces

Visual Communication
- Introduction
- General issues in visual communication
- Some examples
    - Different ways encode information
    - Different color encodings for information
    - Geometric encoding of information
    - Other encodings
    - Higher dimensions
    - Choosing an appropriate view
    - Moving a viewpoint
    - Setting a particular viewpoint
    - Seeing motion
    - Legends to help communicate your encodings

- Creating effective interaction
- Implementing legends and labels in OpenGL
- Using the accumulation buffer

• A word to the wise

Science Examples I

- Modeling diffusion of a quantity in a region
  > Temperature in a metal bar
  > Spread of disease in a region
- Simple graph of a function of two variables
- Mathematical functions
  > Electrostatic potential function
- Simulating a scientific process
  > Gas laws
  > Diffusion through a semipermeable membrane

The OpenGL Pipeline

• Introduction
• The Pipeline
• Implementation in Graphics Cards

Lights and Lighting

• Introduction
• Definitions
  - Ambient, diffuse, and specular light
  - Use of materials
• Light properties
  - Positional lights
  - Spotlights
  - Attenuation
  - Directional lights
  - Positional and moving lights
• Lights and materials in OpenGL
  - Specifying and defining lights
  - Defining materials
  - Setting up a scene to use lighting
  - Using GLU quadric objects
  - Lights of all three primary colors applied to a white surface
• A word to the wise

Shading Models

• Introduction
• Definitions
  - Flat shading
  - Smooth shading
• Some examples
• Calculating per-vertex normals
• Other shading models
• Some examples
• Code examples

Event Handling

• Introduction
• Definitions

- Some examples of events
  - Keypress events
  - Mouse events
  - system events
  - software events
- Callback registering
- The vocabulary of interaction
- A word to the wise
- Some details
- Code examples
  - Idle event callback
  - Keyboard callback
  - Menu callback
  - Mouse callback for object selection
  - Mouse callback for mouse motion

The MUI (Micro User Interface) Facility

- Introduction
- Definitions
  - Menu bars
  - Buttons
  - Radio buttons
  - Text boxes
  - Horizontal sliders
  - Vertical sliders
  - Text labels
- Using the MUI functionality
- Some examples
- A word to the wise

Science Examples II

- Examples
  - Displaying scientific objects
    - > Simple molecule display
    - > Displaying the conic sections
  - Representing a function of two variables
    - > Mathematical functions
    - > Surfaces for special functions
    - > Electrostatic potential function
    - > Interacting waves
  - Representing more complicated functions
    - > Implicit surfaces
    - > Cross-sections of volumes
    - > Vector displays
    - > Parametric curves
    - > Parametric surfaces
  - Illustrating dynamic systems
    - > The Lorenz attractor
    - > The Sierpinski attractor
- Some enhancements to the displays
  - Stereo pairs

Texture Mapping

- Introduction
- Definitions

    - 1D texture maps
    - 2D texture maps
    - 3D texture maps
    - The relation between the color of the object and the color of the texture map
    - Texture mapping and billboards
- Creating a texture map
    - Getting an image as a texture map
    - Generating a synthetic texture map
- Antialiasing in texturing
- Texture mapping in OpenGL
    - Capturing a texture from the screen
    - Texture environment
    - Texture parameters
    - Getting and defining a texture map
    - Texture coordinate control
    - Texture mapping and GLU quadrics
- Some examples
    - The Chromadepth™ process
    - Using 2D texture maps to add interest to a surface
    - Environment maps
- A word to the wise
- Code examples
    - A 1D color ramp
    - An image on a surface
    - An environment map
- Resources

Dynamics and Animation

- Introduction
- Definitions
- Keyframe animation
    - Building an animation
- Some examples
    - Moving objects in your model
    - Moving parts of objects in your model
    - Moving the eye point or the view frame in your model
    - Changing features of your models
- Some points to consider when doing animations with OpenGL
- Code examples

High-Performance Graphics Techniques and Games Graphics

- Definitions
- Techniques
    - Hardware avoidance
    - Designing out visible polygons
    - Culling polygons
    - Avoiding depth comparisons
    - Front-to-back drawing
    - Binary space partitioning
    - Clever use of textures
    - System speedups
    - LOD
    - Reducing lighting computation
    - Fog

Because this is an early draft of the notes for an introductory, API-based computer graphics course, the author apologizes for any inaccuracies, incompleteness, or clumsiness in the presentation. Further development of these materials, as well as source code for many projects and additional examples, is ongoing continuously. All such materials will be posted as they are ready on the author's Web site:

`http://www.cs.csustan.edu/~rsc/NSF/`

Your comments and suggesions will be very helpful in making these materials as useful as possible and are solicited; please contact

Steve Cunningham
California State University Stanislaus
`rsc@cs.csustan.edu`

This work was supported by National Science Foundation grant DUE-9950121. All opinions, findings, conclusions, and recommendations in this work are those of the author and do not necessarily reflect the views of the National Science Foundation. The author also gratefully acknowledges sabbatical support from California State University Stanislaus and thanks the San Diego Supercomputer Center, most particularly Dr. Michael J. Bailey, for hosting this work and for providing significant assistance with both visualization and science content. The author also thanks a number of others for valuable conversations and suggestions on these notes.

# Getting Started

These notes are intended for an introductory course in computer graphics with a few features that are not found in most beginning courses:

- The focus is on computer graphics programming with the OpenGL graphics API, and many of the algorithms and techniques that are used in computer graphics are covered only at the level they are needed to understand questions of graphics programming. This differs from most computer graphics textbooks that place a great deal of emphasis on understanding these algorithms and techniques. We recognize the importance of these for persons who want to develop a deep knowledge of the subject and suggest that a second graphics course built on the ideas of these notes can provide that knowledge. Moreover, we believe that students who become used to working with these concepts at a programming level will be equipped to work with these algorithms and techniques more fluently than students who meet them with no previous background.
- We focus on 3D graphics to the almost complete exclusion of 2D techniques. It has been traditional to start with 2D graphics and move up to 3D because some of the algorithms and techniques have been easier to grasp at the 2D level, but without that concern it seems easier simply to start with 3D and discuss 2D as a special case.
- Because we focus on graphics programming rather than algorithms and techniques, we have fewer instances of data structures and other computer science techniques. This means that these notes can be used for a computer graphics course that can be taken earlier in a student's computer science studies than the traditional graphics course. Our basic premise is that this course should be quite accessible to a student with a sound background in programming a sequential imperative language, particularly C.
- These notes include an emphasis on the scene graph as a fundamental tool in organizing the modeling needed to create a graphics scene. The concept of scene graph allows the student to design the transformations, geometry, and appearance of a number of complex components in a way that they can be implemented quite readily in code, even if the graphics API itself does not support the scene graph directly. This is particularly important for hierarchical modeling, but it provides a unified design approach to modeling and has some very useful applications for placing the eye point in the scene and for managing motion and animation.
- These notes include an emphasis on visual communication and interaction through computer graphics that is usually missing from textbooks, though we expect that most instructors include this somehow in their courses. We believe that a systematic discussion of this subject will help prepare students for more effective use of computer graphics in their future professional lives, whether this is in technical areas in computing or is in areas where there are significant applications of computer graphics.
- Many, if not most, of the examples in these notes are taken from sources in the sciences, and they include two chapters on scientific and mathematical applications of computer graphics. This makes the notes useable for courses that include science students as well as making graphics students aware of the breadth of areas in the sciences where graphics can be used.

This set of emphases makes these notes appropriate for courses in computer science programs that want to develop ties with other programs on campus, particularly programs that want to provide science students with a background that will support development of computational science or scientific visualization work.

What is a graphics API?

The short answer is than an API is an *Application Programming Interface* — a set of tools that allow a programmer to work in an application area. Thus a **graphics** API is a set of tools that allow a programmer to write applications that use computer graphics. These materials are intended to introduce you to the OpenGL graphics API and to give you a number of examples that will help you understand the capabilities that OpenGL provides and will allow you to learn how to integrate graphics programming into your other work.

*Overview of these notes*

In these notes we describe some general principles in computer graphics, emphasizing 3D graphics and interactive graphical techniques, and show how OpenGL provides the graphics programming tools that implement these principles. We do not spend time describing in depth the way the techniques are implemented or the algorithms behind the techniques; these will be provided by the lectures if the instructor believes it necessary. Instead, we focus on giving some concepts behind the graphics and on using a graphics API (application programming interface) to carry out graphics operations and create images.

These notes will give beginning computer graphics students a good introduction to the range of functionality available in a modern computer graphics API. They are based on the OpenGL API, but we have organized the general outline so that they could be adapted to fit another API as these are developed.

The key concept in these notes, and in the computer graphics programming course, is the use of computer graphics to communicate information to an audience. We usually assume that the information under discussion comes from the sciences, and include a significant amount of material on models in the sciences and how they can be presented visually through computer graphics. It is tempting to use the word "visualization" somewhere in the title of this document, but we would reserve that word for material that is fully focused on the science with only a sidelight on the graphics; because we reverse that emphasis, the role of visualization is in the application of the graphics.

We have tried to match the sequence of these modules to the sequence we would expect to be used in an introductory course, and in some cases, the presentation of one module will depend on the student knowing the content of an earlier module. However, in other cases it will not be critical that earlier modules have been covered. It should be pretty obvious if other modules are assumed, and we may make that assumption explicit in some modules.

*What is Computer Graphics?*

We view computer graphics as the art and science of creating synthetic images by programming the geometry and appearance of the contents of the images, and by displaying the results of that programming on appropriate display devices that support graphical output. The programming may be done (and in these notes, is assumed to be done) with the support of a graphics API that does most of the detailed work of rendering the scene that the programming defines.

The work of the programmer is to develop representations for the geometric entities that are to make up the images, to assemble these entities into an appropriate geometric space where they can have the proper relationships with each other as needed for the image, to define and present the look of each of the entities as part of that scene, to specify how the scene is to be viewed, and to specify how the scene as viewed is to be displayed on the graphic device. These processes are supported by the 3D graphics pipeline, as described below, which will be one of our primary tools in understanding how graphics processes work.

In addition to the work mentioned so far, there are two other important parts of the task for the programmer. Because a static image does not present as much information as a moving image, the programmer may want to design some motion into the scene, that is, may want to define some animation for the image. And because a user may want to have the opportunity to control the nature of the image or the way the image is seen, the programmer may want to design ways for the user to interact with the scene as it is presented.

All of these topics will be covered in the notes, using the OpenGL graphics API as the basis for implementing the actual graphics programming.

*The 3D Graphics Pipeline*

The 3D computer graphics pipeline is simply a process for converting coordinates from what is most convenient for the application programmer into what is most convenient for the display hardware. We will explore the details of the steps for the pipeline in the chapters below, but here we outline the pipeline to help you understand how it operates. The pipeline is diagrammed in Figure 0.9, and we will start to sketch the various stages in the pipeline here, with more detail given in subsequent chapters.

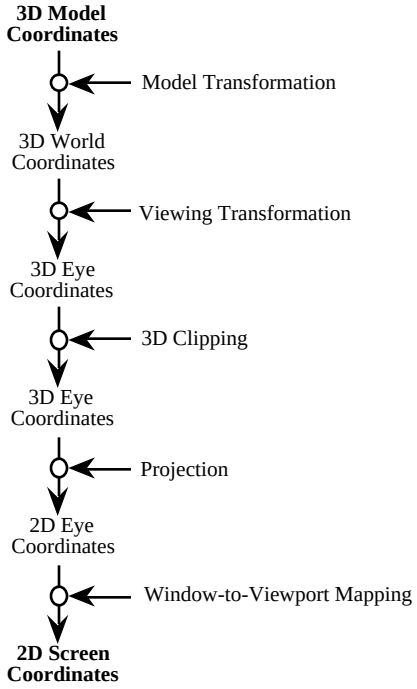



Figure 0.9: The graphics pipeline's stages and mappings

3D model coordinate systems

The application programmer starts by defining a particular object about a local origin, somewhere in or around the object. This is what would naturally happen if the object was exported from a CAD system or was defined by a mathematical function. Modeling something about its local origin involves defining it in terms of *model coordinates*, a coordinate system that is used specifically to define a particular graphical object. Note that the modeling coordinate system may be different for every part of a scene. If the object uses its own coordinates as it is defined, it must be placed in the 3D world space by using appropriate transformations.

Transformations are functions that move objects while preserving their geometric properties. The transformations that are available to us in a graphics system are rotations, translations, and scaling. Rotations hold the origin of a coordinate system fixed and move all the other points by a fixed angle around the origin, translations add a fixed value to each of the coordinates of each point in a scene, and scaling multiplies each coordinate of a point by a fixed value. These will be discussed in much more detail in the chapter on modeling below.

3D world coordinate system

After a graphics object is defined in its own modeling coordinate system, the object is transformed to where it belongs in the scene. This is called the *model transformation*, and the single coordinate system that describes the position of every object in the scene is called the *world coordinate system*. In practice, graphics programmers use a relatively small set of simple, built-in transformations and build up the model transformations through a sequence of these simple transformations. Because each transformation works on the geometry it sees, we see the effect of the associative law for functions; in a piece of code represented by metacode such as

```
transformOne(...);
transformTwo(...);
transformThree(...);
geometry(...);
```

we see that transformThree is applied to the original geometry, transformTwo to the results of that transformation, and transformOne to the results of the second transformation. Letting `t1, t2,` and `t3` be the three transformations, respectively, we see by the application of the associative law for function application that

```
t1(t2(t3(geometry))) = (t1*t2*t3)(geometry)
```

This shows us that in a product of transformations, applied by multiplying on the left, the transformation nearest the geometry is applied first, and that this principle extends across multiple transformations. This will be very important in the overall understanding of the overall order in which we operate on scenes, as we describe at the end of this section.

The model transformation for an object in a scene can change over time to create motion in a scene. For example, in a rigid-body animation, an object can be moved through the scene just by changing its model transformation between frames. This change can be made through standard built-in facilities in most graphics APIs, including OpenGL; we will discuss how this is done later.

3D eye coordinate system

Once the 3D world has been created, an application programmer would like the freedom to be able to view it from any location. But graphics viewing models typically require a specific orientation and/or position for the eye at this stage. For example, the system might require that the eye position be at the origin, looking in –Z (or sometimes +Z). So the next step in the pipeline is the *viewing transformation*, in which the coordinate system for the scene is changed to satisfy this requirement. The result is the 3D eye coordinate system. One can think of this process as grabbing the arbitrary eye location and all the 3D world objects and sliding them around together so that the eye ends up at the proper place and looking in the proper direction. The relative positions between the eye and the other objects have not been changed; all the parts of the scene are simply anchored in a different spot in 3D space. This is just a transformation, although it can be asked for in a variety of ways depending on the graphics API. Because the viewing transformation transforms the entire world space in order to move the eye to the standard position and orientation, we can consider the viewing transformation to be the inverse of whatever transformation placed the eye point in the position and orientation defined for the view. We will take advantage of this observation in the modeling chapter when we consider how to place the eye in the scene's geometry.

At this point, we are ready to clip the object against the *3D viewing volume*. The viewing volume is the 3D volume that is determined by the projection to be used (see below) and that declares what portion of the 3D universe the viewer wants to be able to see. This happens by defining how for the scene should be visible to the left, right, bottom, top, near, and far. Any portions of the scene that are outside the defined viewing volume are clipped and discarded. All portions that are inside are retained and passed along to the projection step. In Figure 0.10, note how the front of the

image of the ground in the figure is clipped — is made invisible — because it is too close to the viewer's eye.

Figure 0.10: Clipping on the Left, Bottom, and Right

2D screen coordinates

The 3D eye coordinate system still must be converted into a 2D coordinate system before it can be placed on a graphic device, so the next stage of the pipeline performs this operation, called a *projection*. Before the actual projection is done, we must think about what we will actually see in the graphic device. Imagine your eye placed somewhere in the scene, looking in a particular direction. You do not see the entire scene; you only see what lies in front of your eye and within your field of view. This space is called the viewing volume for your scene, and it includes a bit more than the eye point, direction, and field of view; it also includes a front plane, with the concept that you cannot see anything closer than this plane, and a back plane, with the concept that you cannot see anything farther than that plane.

There are two kinds of projections commonly used in computer graphics. One maps all the points in the eye space to the viewing plane by simply ignoring the value of the z-coordinate, and as a result all points on a line parallel to the direction of the eye are mapped to the same point on the viewing plane. Such a projection is called a *parallel* projection. The other projection acts as if the eye were a single point and each point in the scene is mapped, along a line from the eye to that point, to a point on a plane in front of the eye, which is the classical technique of artists when drawing with perspective. Such a projection is called a *perspective* projection. And just as there are parallel and perspective projections, there are *parallel* (also called *orthographic*) and *perspective* viewing volumes. In a parallel projection, objects stay the same size as they get farther away. In a perspective projection, objects get smaller as they get farther away. Perspective projections tend to look more realistic, while parallel projections tend to make objects easier to line up. Each projection will display the geometry within the region of 3-space that is bounded by the right, left, top, bottom, back, and front planes described above. The region that is visible with each projection is often called its *view volume*. As seen in Figure 0.11 below, the viewing volume of a parallel projection is a rectangular region (here shown as a solid), while the viewing volume of a perspective projection has the shape of a pyramid that is truncated at the top. This kind of shape is sometimes called a *frustum* (also shown here as a solid).

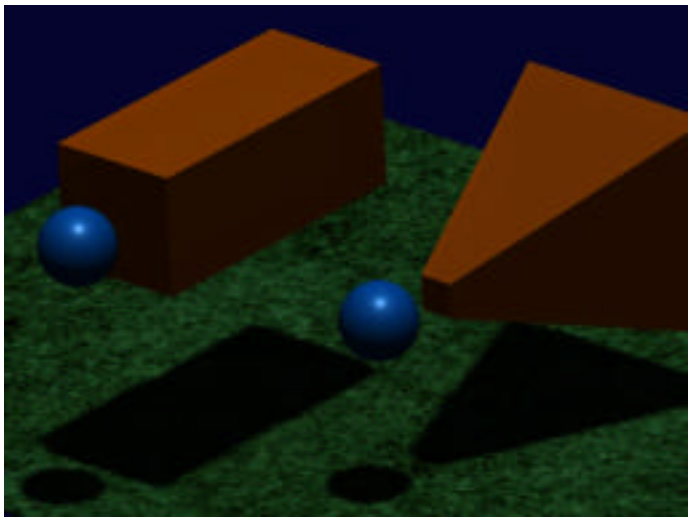


Figure 0.11: Parallel and Perspective Viewing Volumes, with Eyeballs

Figure 0.12 presents a scene with both parallel and perspective projections; in this example, you will have to look carefully to see the differences!

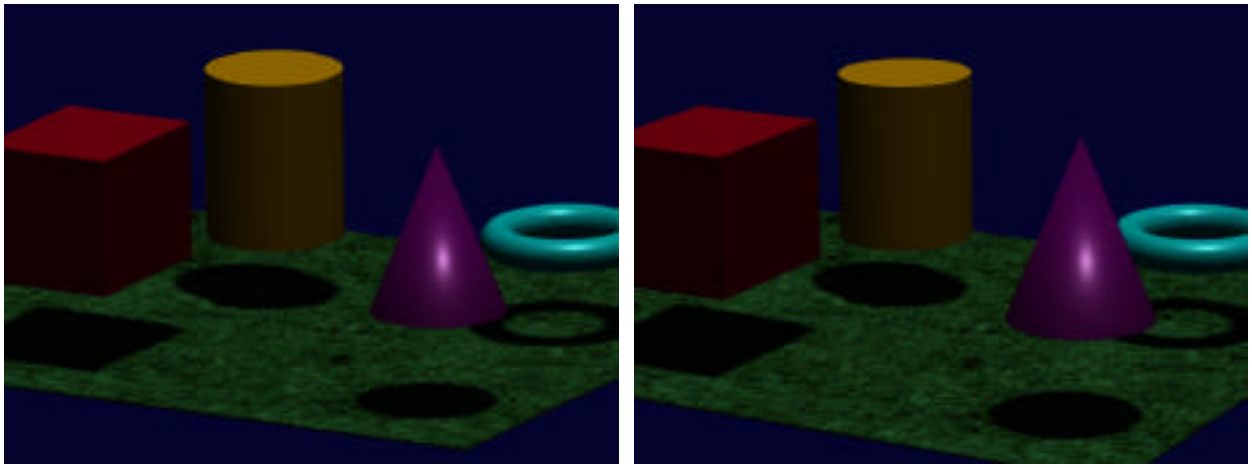


Figure 0.12: the same scene as presented by a parallel projection (left)
and by a perspective projection (right)

2D screen coordinates

The final step in the pipeline is to change units so that the object is in a coordinate system appropriate for the display device. Because the screen is a digital device, this requires that the real numbers in the 2D eye coordinate system be converted to integer numbers that represent screen coordinate. This is done with a proportional mapping followed by a truncation of the coordinate values. It is called the *window-to-viewport mapping*, and the new coordinate space is referred to as *screen coordinates*, or *display coordinates*. When this step is done, the entire scene is now represented by integer screen coordinates and can be drawn on the 2D display device.

Note that this entire pipeline process converts vertices, or *geometry*, from one form to another by means of several different transformations. These transformations ensure that the vertex geometry of the scene is consistent among the different representations as the scene is developed, but

computer graphics also assumes that the *topology* of the scene stays the same. For instance, if two points are connected by a line in 3D model space, then those converted points are assumed to likewise be connected by a line in 2D screen space. Thus the geometric relationships (points, lines, polygons, ...) that were specified in the original model space are all maintained until we get to screen space, and are only actually implemented there.

Overall viewing process

Let's look at the overall operations on the geometry you define for a scene as the graphics system works on that scene and eventually displays it to your user. Referring again to Figure 0.8 and omitting the clipping and window-to-viewport process, we see that we start with geometry, apply the modeling transformation(s), apply the viewing transformation, and apply the projection to the screen. This can be expressed in terms of function composition as the sequence

```
projection(viewing(transformation(geometry)))
```

or, as we noted above with the associative law for functions and writing function composition as multiplication,

```
(projection * viewing * transformation) (geometry).
```

In the same way we saw that the operations nearest the geometry were performed before operations further from the geometry, then, we will want to define the projection first, the viewing next, and the transformations last before we define the geometry they are to operate on. We will see this sequence as a key factor in the way we structure a scene through the scene graph in the modeling chapter later in these notes.

Different implementation, same result

*Warning!* This discussion has shown the *concept* of how a vertex travels through the graphics pipeline. There are several ways of implementing this travel, any of which will produce a correct display. Do not be disturbed if you find out a graphics system does not manage the overall graphics pipeline process exactly as shown here. The basic principles and stages of the operation are still the same.

For example, OpenGL combines the modeling and viewing transformations into a single transformation known as the *modelview matrix*. This will force us to take a little different approach to the modeling and viewing process that integrates these two steps. Also, graphics hardware systems typically perform a window-to-normalized-coordinates operation prior to clipping so that hardware can be optimized around a particular coordinate system. In this case, everything else stays the same except that the final step would be normalized-coordinate-to-viewport mapping.

In many cases, we simply will not be concerned about the details of how the stages are carried out. Our goal will be to represent the geometry correctly at the modeling and world coordinate stages, to specify the eye position appropriately so the transformation to eye coordinates will be correct, and to define our window and projections correctly so the transformations down to 2D and to screen space will be correct. Other details will be left to a more advanced graphics course.

Summary of viewing advantages

One of the classic questions beginners have about viewing a computer graphics image is whether to use perspective or orthographic projections. Each of these has its strengths and its weaknesses. As a quick guide to start with, here are some thoughts on the two approaches:

*Orthographic* projections are at their best when:

- Items in the scene need to be checked to see if they line up or are the same size
- Lines need to be checked to see if they are parallel

- We do not care that distance is handled unrealistically
- We are not trying to move through the scene

*Perspective* projections are at their best when:

- Realism counts
- We want to move through the scene and have a view like a human viewer would have
- We do not care that it is difficult to measure or align things

In fact, when you have some experience with each, and when you know the expectations of the audience for which you're preparing your images, you will find that the choice is quite natural and will have no problem knowing which is better for a given image.

*A basic OpenGL program*

Our example programs that use OpenGL have some strong similarities. Each is based on the GLUT utility toolkit that usually accompanies OpenGL systems, so all the sample codes have this fundamental similarity. (If your version of OpenGL does not include GLUT, its source code is available online; check the page at

`http://www.reality.sgi.com/opengl/glut3/glut3.h`

and you can find out where to get it. You will need to download the code, compile it, and install it in your system.) Similarly, when we get to the section on event handling, we will use the MUI (micro user interface) toolkit, although this is not yet developed or included in this first draft release.

Like most worthwhile APIs, OpenGL is complex and offers you many different ways to express a solution to a graphical problem in code. Our examples use a rather limited approach that works well for interactive programs, because we believe strongly that graphics and interaction should be learned together. When you want to focus on making highly realistic graphics, of the sort that takes a long time to create a single image, then you can readily give up the notion of interactive work.

So what is the typical structure of a program that would use OpenGL to make interactive images? We will display this example in C, as we will with all our examples in these notes. OpenGL is not really compatible with the concept of object-oriented programming because it maintains an extensive set of state information that cannot be encapsulated in graphics classes. Indeed, as you will see when you look at the example programs, many functions such as event callbacks cannot even deal with parameters and must work with global variables, so the usual practice is to create a global application environment through global variables and use these variables instead of parameters to pass information in and out of functions. (Typically, OpenGL programs use side effects — passing information through external variables instead of through parameters — because graphics environments are complex and parameter lists can become unmanageable.) So the skeleton of a typical GLUT-based OpenGL program would look something like this:

```
// include section
   #include <GL/glut.h>    // alternately "glut.h" for Macintosh
   // other includes as needed

// typedef section
   // as needed

// global data section
   // as needed

// function template section
   void doMyInit(void);
```

```
   void display(void);
   void reshape(int,int);
   void idle(void);
   // others as defined

// initialization function
   void doMyInit(void) {
      set up basic OpenGL parameters and environment
      set up projection transformation (ortho or perspective)
   }

// reshape function
   void reshape(int w, int h) {
      set up projection transformation with new window
         dimensions w and h
      post redisplay
   }

// display function
   void display(void){
      set up viewing transformation as described in later chapters
      define whatever transformations, appearance, and geometry you need
      post redisplay
   }

// idle function
   void idle(void) {
      update anything that changes from one step of the program to another
      post redisplay
   }

// other graphics and application functions
   // as needed
// main function -- set up the system and then turn it over to events
   void main(int argc, char** argv) {
   // initialize system through GLUT and your own initialization
      glutInit(&argc,argv);
      glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
      glutInitWindowSize(windW,windH);
      glutInitWindowPosition(topLeftX,topLeftY);
      glutCreateWindow("A Sample Program");
      doMyInit();

   // define callbacks for events
      glutDisplayFunc(display);
      glutReshapeFunc(reshape);
      glutIdleFunc(idle);

   // go into main event loop
      glutMainLoop();
   }
```

The viewing transformation is specified in OpenGL with the `gluLookAt()` call:

```
gluLookAt( ex, ey, ez, lx, ly, lz, ux, uy, uz );
```

The parameters for this transformation include the coordinates of eye position (`ex, ey, ez`), the coordinates of the point at which the eye is looking (`lx, ly, lz`), and the coordinates of a

vector that defines the "up" direction for the view (`ux, uy, uz`). This would most often be called from the `display()` function above and is discussed in more detail in the chapter below on viewing.

Projections are specified fairly easily in the OpenGL system. An orthographic (or parallel) projection is defined with the function call:

```
glOrtho( left, right,  bottom, top,  near, far );
```

where `left` and `right` are the x-coordinates of the left and right sides of the orthographic view volume, `bottom` and `top` are the y-coordinates of the bottom and top of the view volume, and `near` and `far` are the z-coordinates of the front and back of the view volume. A perspective projection is defined with the function call:

```
glFrustum( left, right,  bottom, top,  near, far );
```

or:

```
gluPerspective( fovy, aspect,  near, far );
```

In the `glFrustum(...)` call, the values `left`, `right`, `bottom`, and `top` are the coordinates of the left, right, bottom, and top clipping planes as they intersect the near plane; the other coordinate of all these four clipping planes is the eye point. In the `gluPerspective(...)` call, the first parameter is the field of view in degrees, the second is the aspect ratio for the window, and the near and far parameters are as above. In this projection, it is assumed that your eye is at the origin so there is no need to specify the other four clipping planes; they are determined by the field of view and the aspect ratio.

In OpenGL, the modeling transformation and viewing transformation are merged into a single modelview transformation, which we will discuss in much more detail in the modeling chapter below. This means that we cannot manage the viewing transformation separately from the rest of the transformations we must use to do the detailed modeling of our scene.

There are some specific things about this code that we need to mention here and that we will explain in much more detail later, such as callbacks and events. But for now, we can simply view the main event loop as passing control at the appropriate time to the following functions specified in the main function:

```
void doMyInit(void)
void display(void)
void reshape(int,int)
void idle(void)
```

The task of the function `doMyInit()` is to set up the environment for the program so that the scene's fundamental environment is set up. This is a good place to compute values for arrays that define the geometry, to define specific named colors, and the like. At the end of this function you should set up the initial projection specifications.

The task of the function `display()` is to do everything needed to create the image. This can involve manipulating a significant amount of data, but the function does not allow any parameters. Here is the first place where the data for graphics problems must be managed through global variables. As we noted above, we treat the global data as a programmer-created environment, with some functions manipulating the data and the graphical functions using that data (thae graphics environment) to define and present the display. In most cases, the global data is changed only through well-documented side effects, so this use of the data is reasonably clean. (Note that this argues strongly for a great deal of emphasis on documentation in your projects, which most people believe is not a bad thing.) Of course, some functions can create or receive control parameters, and it is up to you to decide whether these parameters should be managed globally or locally, but even in this case the declarations are likely to be global because of the wide number of functions that

may use them. You will also find that your graphics API maintains its own environment, called its system state, and that some of your functions will also manipulate that environment, so it is important to consider the overall environment effect of your work.

The task of the function `reshape(int, int)` is to respond to user manipulation of the window in which the graphics are displayed. The two parameters are the width and height of the window in screen space (or in pixels) as it is resized by the user's manipulation, and should be used to reset the projection information for the scene. GLUT interacts with the window manager of the system and allows a window to be moved or resized very flexibly without the programmer having to manage any system-dependent operations directly. Surely this kind of system independence is one of the very good reasons to use the GLUT toolkit!

The task of the function `idle()` is to respond to the "idle" event — the event that nothing has happened. This function defines what the program is to do without any user activity, and is the way we can get animation in our programs. Without going into detail that should wait for our general discussion of events, the process is that the `idle()` function makes any desired changes in the global environment, and then requests that the program make a new display (with these changes) by invoking the function `glutPostRedisplay()` that simply requests the display function when the system can next do it by posting a "redisplay" event to the system.

The execution sequence of a simple program with no other events would then look something like is shown in Figure 0.13. Note that `main()` does not call the `display()` function directly; instead `main()` calls the event handling function `glutMainLoop()` which in turn makes the first call to `display()` and then waits for events to be posted to the system event queue. We will describe event handling in more detail in a later chapter.

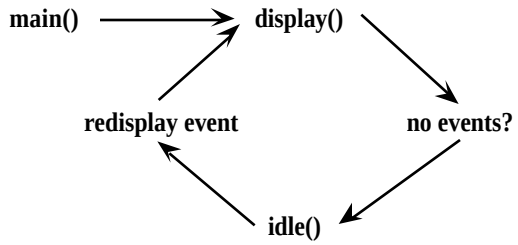



Figure 0.13: the event loop for the idle event

So we see that in the absence of any other event activity, the program will continues to apply the activity of the `idle()` function as time progresses, leading to an image that changes over time — that is, to an animated image.

Now that we have an idea of the graphics pipeline and know what a program can look like, we can move on to discuss how we specify the viewing and projection environment, how we define the fundamental geometry for our image, and how we create the image in the `display()` function with the environment that we define through the viewing and projection.

# Viewing and Projection

*Prerequisites*

An understanding of 2D and 3D geometry and familiarity with simple linear mappings.

*Introduction*

We emphasize 3D computer graphics consistently in these notes, because we believe that computer graphics should be encountered through 3D processes and that 2D graphics can be considered effectively as a special case of 3D graphics. But almost all of the viewing technologies that are readily available to us are 2D — certainly monitors, printers, video, and film — and eventually even the active visual retina of our eyes presents a 2D environment. So in order to present the images of the scenes we define with our modeling, we must create a 2D representation of the 3D scenes. As we saw in the graphics pipeline in the previous chapter, you begin by developing a set of models that make up the elements of your scene and set up the way the models are placed in the scene, resulting in a set of objects in a common world space. You then define the way the scene will be viewed and the way that view is presented on the screen. In this early chapter, we are concerned with the way we move from the world space to a 2D image with the tools of viewing and projection.

We set the scene for this process in the last chapter, when we defined the graphics pipeline. If we begin at the point where we have the 3D world coordinates—that is, where we have a complete scene fully defined in a 3D world—then this chapter is about creating a view of that scene in our display space of a computer monitor, a piece of film or video, or a printed page, whatever we want. To remind ourselves of the steps in this process, they are shown in Figure 1.1:

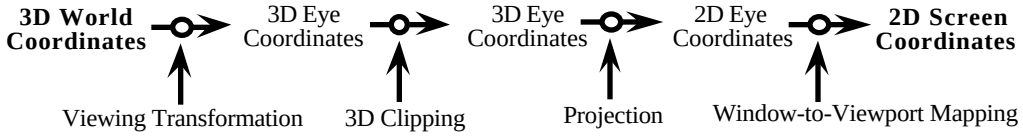



Figure 1.1: the graphics pipeline for creating an image of a scene

Let's consider an example of a world space and look at just what it means to have a view and a presentation of that space. One of the author's favorite places is Yosemite National Park, which is a wonderful example of a 3D world. If you go to Glacier Point on the south side of Yosemite Valley you can see up the valley towards the Merced River falls and Half Dome. The photographs in Figure 1.2 below give you an idea of the views from this point.

Figure 1.2: two photographs of the upper Merced River area from Glacier Point

If you think about this area shown in these photographs, you notice that your view depends first on where you are standing. If you were standing on the valley floor, or at the top of Nevada Falls (the higher falls in the photos), you could not have this view; the first because you would be below this terrain instead of above it, and the second because you would be looking away from the terrain instead of towards it. So your view depends on your position, which we call your eye point. The view also depends on the direction in which you are looking. The two photographs in the figure above are taken from the same point, but show slightly different views because one is looking at the overall scene and the other is looking specifically at the falls. So your scene depends on the direction of your view. The view also depends on whether you are looking at a wide part of the scene or a narrow part; again, one photograph is a panoramic view and one is a focused view. So your image depends on the breadth of field of your view. Finally, although this may not be obvious at first because our minds process images in context, the view depends on whether you are standing with your head upright or tilted (this might be easier to grasp if you think of the view as being defined by a camera instead of by your vision; it's clear that if you tilt a camera at a 45° angle you get a very different photo than one that's taken by a horizontal or vertical camera.) The world is the same in any case, but the four facts of where your eye is, the direction you are facing, the breadth of your attention, and the way your view is tilted, determine the scene that is presented of the world.

But the view, once determined, must now be translated into an image that can be presented on your computer monitor. You may think of this in terms of recording an image on a digital camera, because the result is the same: each point of the view space (each pixel in the image) must be given a specific color. Doing that with the digital camera involves only capturing the light that comes through the lens to that point in the camera's sensing device, but doing it with computer graphics requires that we calculate exactly what will be seen at that particular point when the view is presented. We must define the way the scene is transformed into a two-dimensional space, which involves a number of steps: taking into account all the questions of what parts are in front of what other parts, what parts are out of view from the camera's lens, and how the lens gathers light from the scene to bring it into the camera. The best way to think about the lens is to compare two very different kinds of lenses: one is a wide-angle lens that gathers light in a very wide cone, and the other is a high-altitude photography lens that gathers light only in a very tight cylinder and processes light rays that are essentially parallel as they are transferred to the sensor. Finally, once the light from the continuous world comes into the camera, it is recorded on a digital sensor that only captures a discrete set of points.

This model of viewing is paralleled quite closely by a computer graphics system. You begin your work by modeling your scene in an overall world space (you may actually start in several modeling spaces, because you may model the geometry of each part of your scene in its own modeling space where it can be defined easily, then place each part within a single consistent world space to define the scene). This is very different from the viewing we discuss here but is covered in detail in the next chapter. The fundamental operation of viewing is to define an eye within your world space that represents the view you want to take of your modeling space. Defining the eye implies that you are defining a coordinate system relative to that eye position, and you must then transform your modeling space into a standard form relative to this coordinate system by defining, and applying, a viewing transformation. The fundamental operation of projection, in turn, is to define a plane within 3-space, define a mapping that projects the model into that plane, and displays that plane in a given space on the viewing surface (we will usually think of a screen, but it could be a page, a video frame, or a number of other spaces).

We will think of the 3D space we work in as the traditional X-Y-Z Cartesian coordinate space, usually with the X- and Y-axes in their familiar positions and with the Z-axis coming toward the viewer from the X-Y plane. This orientation is used because most graphics APIs define the plane onto which the image is projected for viewing as the X-Y plane, and project the model onto this

plane in some fashion along the Z-axis. The mechanics of the modeling transformations, viewing transformation, and projection are managed by the graphics API, and the task of the graphics programmer is to provide the API with the correct information and call the API functionality in the correct order to make these operations work. We will describe the general concepts of viewing and projection below and will then tell you how to specify the various parts of this process to OpenGL.

Finally, it is sometimes useful to "cut away" part of an image so you can see things that would otherwise be hidden behind some objects in a scene. We include a brief discussion of clipping planes, a technique for accomplishing this action.

*Fundamental model of viewing*

As a physical model, we can think of the viewing process in terms of looking through a rectangular hole cut out of a piece of cardboard and held in front of your eye. You can move yourself around in the world, setting your eye into whatever position and orientation from you wish to see the scene. This defines your view. Once you have set your position in the world, you can hold up the cardboard to your eye and this will set your projection; by changing the distance of the cardboard from the eye you change the viewing angle for the projection. Between these two operations you define how you see the world in perspective through the hole. And finally, if you put a piece of paper that is ruled in very small squares behind the cardboard (instead of your eye) and you fill in each square to match the brightness you see in the square, you will create a copy of the image that you can take away from the Of course, you only have a perspective projection instead of an orthogonal projection, but this model of viewing is a good place to start in understanding how viewing and projection work.

As we noted above, the goal of the viewing process is to rearrange the world so it looks as it would if the viewer's eye were in a standard position, depending on the API's basic model. When we define the eye location, we give the API the information it needs to do this rearrangement. In the next chapter on modeling, we will introduce the important concept of the *scene graph*, which will integrate viewing and modeling. Here we give an overview of the viewing part of the scene graph.

The key point is that your view is defined by the location, direction, orientation, and field of view of the eye as we noted above. There are many ways to create this definition, but the effect of each is to give the transformation needed to place the eye at its desired location and orientation, which we will assume to be at the origin, looking in the negative direction down the Z-axis. To put the eye into this standard position we compute a new coordinate system for the world by applying what is called the *viewing transformation*. The viewing transformation is created by computing the inverse of the transformation that placed the eye into the world. (If the concept of computing the inverse seems difficult, simply think of undoing each of the pieces of the transformation; we will discuss this more in the chapter on modeling). Once the eye is in standard position, and all your geometry is adjusted in the same way, the system can easily move on to project the geometry onto the viewing plane so the view can be presented to the user.

Once you have organized the view in this way, you must organize the information you send to the graphics system to create your scene. The graphics system provides some assistance with this by providing tools that determine just what will be visible in your scene and that allow you to develop a scene but only present it to your viewer when it is completed. These will also be discussed in this chapter.

*Definitions*

There are a small number of things that you must consider when thinking of how you will view your scene. These are independent of the particular API or other graphics tools you are using, but

later in the chapter we will couple our discussion of these points with a discussion of how they are handled in OpenGL. The things are:

- Your world must be seen, so you need to say how the view is defined in your model including the eye position, view direction, field of view, and orientation.
- In general, your world must be seen on a 2D surface such as a screen or a sheet of paper, so you must define how the 3D world is projected into a 2D space
- When your world is seen on the 2D surface, it must be seen at a particular place, so you must define the location where it will be seen.

These three things are called setting up your viewing environment, defining your projection, and defining your window and viewport, respectively.

Setting up the viewing environment: in order to set up a view, you have to put your eye in the geometric world where you do your modeling. This world is defined by the coordinate space you assumed when you modeled your scene as discussed earlier. Within that world, you define four critical components for your eye setup: where your eye is located, what point your eye is looking towards, how wide your field of view is, and what direction is vertical with respect to your eye. When these are defined to your graphics API, the geometry in your modeling is adjusted to create the view as it would be seen with the environment that you defined. This is discussed in the section below on the fundamental model of viewing.

Projections: When you define a scene, you will want to do your work in the most natural world that would contain the scene, which we called the model space in the graphics pipeline discussion of the previous chapter. For most of these notes, that will mean a three-dimensional world that fits the objects you are developing. But you will probably want to display that world on a two-dimensional space such as a computer monitor, a video screen, or a sheet of paper. In order to move from the three-dimensional world to a two-dimensional world we use a projection operation.

When you (or a camera) view something in the real world, everything you see is the result of light that comes to the retina (or the film) through a lens that focuses the light rays onto that viewing surface. This process is a projection of the natural (3D) world onto a two-dimensional space. These projections in the natural world operate when light passes through the lens of the eye (or camera), essentially a single point, and have the property that parallel lines going off to infinity seem to converge at the horizon so things in the distance are seen as smaller than the same things when they are close to the viewer. This kind of projection, where everything is seen by being projected onto a viewing plane through or towards a single point, is called a perspective projection. Standard graphics references show diagrams that illustrate objects projected to the viewing plane through the center of view; the effect is that an object farther from the eye are seen as smaller in the projection than the same object closer to the eye.

On the other hand, there are sometimes situations where you want to have everything of the same size show up as the same size on the image. This is most common where you need to take careful measurements from the image, as in engineering drawings. Parallel projections accomplish this by projecting all the objects in the scene to the viewing plane by parallel lines. For parallel projections, objects that are the same size are seen in the projection with the same size, no matter how far they are from the eye. Standard graphics texts contain diagrams showing how objects are projected by parallel lines to the viewing plane.

In Figure 1.3 we show two images of a wireframe house from the same viewpoint. The left-hand image of the figure is presented with a perspective projection, as shown by the difference in the apparent sizes of the front and back ends of the building, and by the way that the lines outlining the sides and roof of the building get closer as they recede from the viewer. The right-hand image of the figure is shown with a parallel or orthogonal projection, as shown by the equal sizes of the front and back ends of the building and the parallel lines outlining the sides and roof of the building. The differences between these two images is admittedly small, but you should use both

projections on some of your scenes and compare the results to see how the differences work in different views.

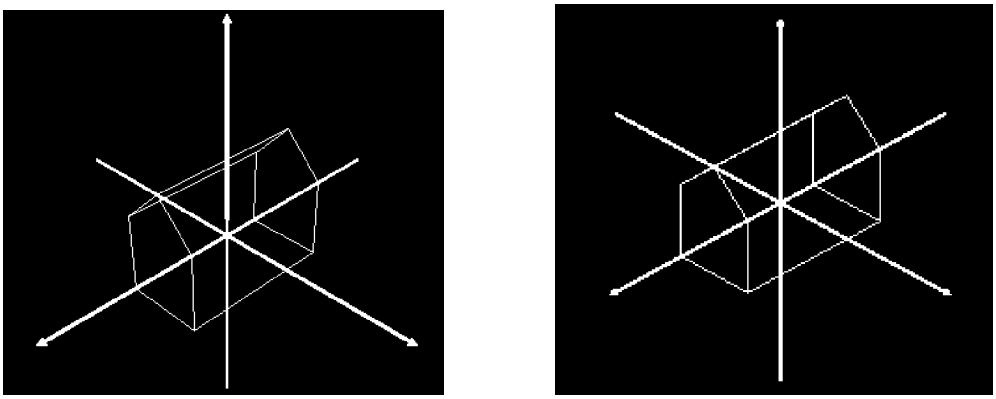


Figure 1.3: perspective image (left) and orthographic image (right)

A projection is often thought of in terms of its view volume, the region of space that is visible in the projection. With either perspective or parallel projection, the definition of the projection implicitly defines a set of boundaries for the left and right sides, top and bottom sides, and front and back sides of a region in three-dimensional space that is called the *viewing volume* for the projection. The viewing volumes for the perspective and orthogonal projections are shown in Figure 1.4 below. Only objects that are inside this space will be displayed; anything else in the scene will be clipped and be invisible.

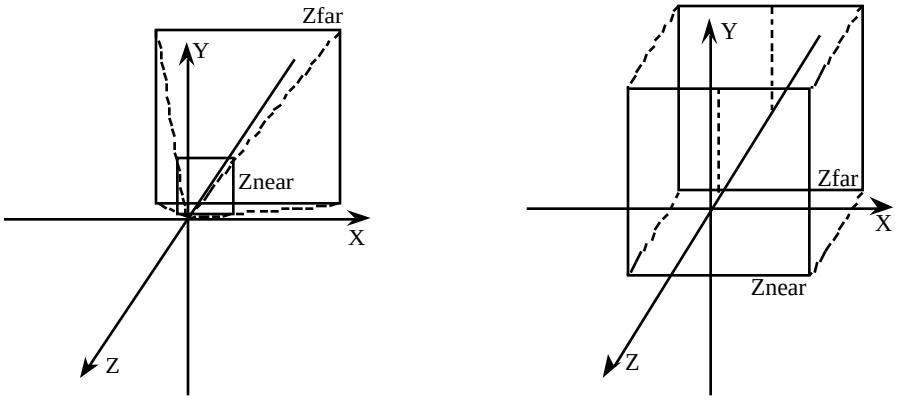



Figure 1.4: the viewing volumes for the perspective (left) and orthogonal (right) projections

While the parallel view volume is defined only in a specified place in your model space, the orthogonal view volume may be defined wherever you need it because, being independent of the calculation that makes the world appear from a particular point of view, an orthogonal view can take in any part of space. This allows you to set up an orthogonal view of any part of your space, or to move your view volume around to view any part of your model.

Defining the window and viewport: We usually think first of a window when we do graphics on a screen. A window in the graphics sense is a rectangular region in your viewing space in which all of the drawing from your program will be done, usually defined in terms of the physical units of the drawing space. The space in which you define and manage your graphics windows will be called *screen space* here for convenience, and is identified with integer coordinates. The smallest displayed unit in this space will be called a *pixel*, a shorthand for picture element. Note that the

window for drawing is a distinct concept from the window in a desktop display window system, although the drawing window may in fact occupy a window on the desktop; we will be consistently careful to reserve the term window for the graphic display.

The scene as presented by the projection is still in 2D real space (the objects are all defined by real numbers) but the screen space is discrete, so the next step is a conversion of the geometry in 2D eye coordinates to screen coordinates. This required identifying discrete screen points to replace the real-number eye geometry points, and introduces some sampling issues that must be handled carefully, but graphics APIs do this for you. The actual screen space used depends on the viewport you have defined for your image.

In order to consider the screen point that replaces the eye geometry point, you will want to understand the relation between points in two corresponding rectangular spaces. In this case, the rectangle that describes the scene to the eye is viewed as one space, and the rectangle on the screen where the scene is to be viewed is presented as another. The same processes apply to other situations that are particular cases of corresponding points in two rectangular spaces, such as the relation between the position on the screen where the cursor is when a mouse button is pressed, and the point that corresponds to this in the viewing space, or points in the world space and points in a texture space.

In Figure 1.5 below, we consider two rectangles with boundaries and points named as shown. In this example, we assume that the lower left corner of each rectangle has the smallest values of the X and Y coordinates in the rectangle. With the names of the figures, we have the proportions

```
X : XMIN :: XMAX : XMIN = u : L :: R : L
Y : YMIN :: YMAX : YMIN = v : B :: T : B
```

from which we can derive the equations:

```
(x - XMIN)/(XMAX - XMIN) = (u - L)/(R - L)
(y - YMIN)/(YMAX - YMIN) = (v - B)/(T - B)
```

and finally these two equations can be solved for the variables of either point in terms of the other:

```
x = XMIN + (u - L)*(XMAX - XMIN)/(R - L)
y = YMIN + (v - B)*(YMAX - YMIN)/(T - B)
```

or the dual equations that solve for (u,v) in terms of (x,y).

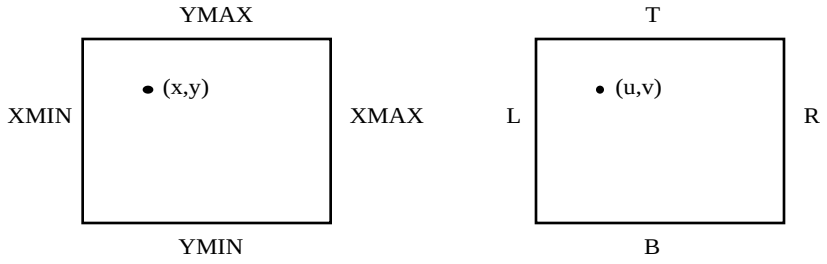



Figure 1.5: correspondences between points in two rectangles

In cases that involve the screen coordinates of a point in a window, there is an additional issue because the upper left, not the lower left, corner of the rectangle contains the smallest values, and the largest value of Y, YMAX, is at the bottom of the rectangle. In this case, however, we can make a simple change of variable as `Y' = YMAX - Y` and we see that using the Y' values instead of Y will put us back into the situation described in the figure. We can also see that the question of rectangles in 2D extends easily into rectangular spaces in 3D, and we leave that to the student.

Within the window, you can choose the part where your image is presented, and this part is called a viewport. A viewport is a rectangular region within that window to which you can restrict your image drawing. In any window or viewport, the ratio of its width to its height is called its *aspect*

*ratio.* A window can have many viewports, even overlapping if needed to manage the effect you need, and each viewport can have its own image. The default behavior of most graphics systems is to use the entire window for the viewport. A viewport is usually defined in the same terms as the window it occupies, so if the window is specified in terms of physical units, the viewport probably will be also. However, a viewport can also be defined in terms of its size relative to the window.

If your graphics window is presented in a windowed desktop system, you may want to be able to manipulate your graphics window in the same way you would any other window on the desktop. You may want to move it, change its size, and click on it to bring it to the front if another window has been previously chosen as the top window. This kind of window management is provided by the graphics API in order to make the graphics window compatible with all the other kinds of windows available.

When you manipulate the desktop window containing the graphics window, the contents of the window need to be managed to maintain a consistent view. The graphics API tools will give you the ability to manage the aspect ratio of your viewports and to place your viewports appropriately within your window when that window is changed. If you allow the aspect ratio of a new viewport to be different than it was when defined, you will see that the image in the viewport seems distorted, because the program is trying to draw to the originally-defined viewport.

A single program can manage several different windows at once, drawing to each as needed for the task at hand. Window management can be a significant problem, but most graphics APIs have tools to manage this with little effort on the programmer's part, producing the kind of window you are accustomed to seeing in a current computing system — a rectangular space that carries a title bar and can be moved around on the screen and reshaped. This is the space in which all your graphical image will be seen. Of course, other graphical outputs such as video will handle windows differently, usually treating the entire output frame as a single window without any title or border.

What this means: Any graphics system will have its approach to defining the computations that transform your geometric model as if it were defined in a standard position and then project it to compute the points to set on the viewing plane to make your image. Each graphics API has its basic concept of this standard position and its tools to create the transformation of your geometry so it can be viewed correctly. For example, OpenGL defines its viewing to take place in a left-handed coordinate system (while all its modeling is done in a right-handed system) and transforms all the geometry in your scene (and we do mean **all** the geometry, including lights and directions, as we will see in later chapters) to place your eye point at the origin, looking in the negative direction along the Z-axis. The eye-space orientation is illustrated in Figure 1.6. The projection then determines how the transformed geometry will be mapped to the X-Y plane, and these processes are illustrated later in this chapter. Finally, the viewing plane is mapped to the viewport you have defined in your window, and you have the image you defined.

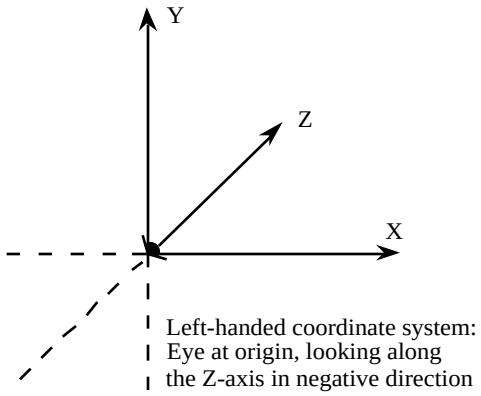



Figure 1.6: the standard OpenGL viewing model

Of course, no graphics API assumes that you can only look at your scenes with this standard view definition. Instead, you are given a way to specify your view very generally, and the API will convert the geometry of the scene so it is presented with your eyepoint in this standard position. This conversion is accomplished through a *viewing transformation* that is defined from your view definition.

The information needed to define your view includes your eye position (its (x, y, z) coordinates), the direction your eye is facing or the coordinates of a point toward which it is facing, and the direction your eye perceives as "up" in the world space. For example, the default view that we mention above has the position at the origin, or (0, 0, 0), the view direction or the "look-at" point coordinates as (0, 0, -1), and the up direction as (0, 1, 0). You will probably want to identify a different eye position for most of your viewing, because this is very restrictive and you aren't likely to want to define your whole viewable world as lying somewhere behind the X-Y plane, and so your graphics API will give you a function that allows you to set your eye point as you desire.

The viewing transformation, then, is the transformation that takes the scene as you define it in world space and aligns the eye position with the standard model, giving you the eye space we discussed in the previous chapter. The key actions that the viewing transformation accomplishes are to rotate the world to align your personal up direction with the direction of the Y-axis, to rotate it again to put the look-at direction in the direction of the negative Z-axis (or to put the look-at point in space so it has the same X- and Y-coordinates as the eye point and a Z-coordinate less than the Z-coordinate of the eye point), to translate the world so that the eye point lies at the origin, and finally to scale the world so that the look-at point or look-at vector has the value (0, 0, -1). This is a very interesting transformation because what it *really* does is to invert the set of transformations that would move the eye point from its standard position to the position you define with your API function as above. This is discussed in some depth later in this chapter in terms of defining the view environment for the OpenGL API.

*Some aspects of managing the view*

Once you have defined the basic features for viewing your model, there are a number of other things you can consider that affect how the image is created and presented. We will talk about many of these over the next few chapters, but here we talk about hidden surfaces, clipping planes, and double buffering.

Hidden surfaces: Most of the things in our world are opaque, so we only see the things that are nearest to us as we look in any direction. This obvious observation can prove challenging for computer-generated images, however, because a graphics system simply draws what we tell it to

draw in the order we tell it to draw them. In order to create images that have the simple "only show me what is nearest" property we must use appropriate tools in viewing our scene.

Most graphics systems have a technique that uses the geometry of the scene in order to decide what objects are in front of other objects, and can use this to draw only the part of the objects that are in front as the scene is developed. This technique is generally called Z-buffering because it uses information on the z-coordinates in the scene, as shown in Figure 1.4. In some systems it goes by other names; for example, in OpenGL this is called the *depth buffer*. This buffer holds the z-value of the nearest item in the scene for each pixel in the scene, where the z-values are computed from the eye point in eye coordinates. This z-value is the depth value after the viewing transformation has been applied to the original model geometry.

This depth value is not merely computed for each vertex defined in the geometry of a scene. When a polygon is processed by the graphics pipeline, an interpolation process is applied as described in the interpolation discussion in the chapter on the pipeline. This process will define a z-value, which is also the distance of that point from the eye in the z-direction, for each pixel in the polygon as it is processed. This allows a comparison of the z-value of the pixel to be plotted with the z-value that is currently held in the depth buffer. When a new point is to be plotted, the system first makes this comparison to check whether the new pixel is closer to the viewer than the current pixel in the image buffer and if it is, replaces the current point by the new point. This is a straightforward technique that can be managed in hardware by a graphics board or in software by simple data structures. There is a subtlety in this process that should be understood, however. Because it is more efficient to compare integers than floating-point numbers, the depth values in the buffer are kept as unsigned integers, scaled to fit the range between the near and far planes of the viewing volume with 0 as the front plane. If the near and far planes are far apart you may experience a phenomenon called "Z-fighting" in which roundoff errors when floating-point numbers are converted to integers causes the depth buffer shows inconsistent values for things that are supposed to be at equal distances from the eye. This problem is best controlled by trying to fit the near and far planes of the view as closely as possible to the actual items being displayed.

There are other techniques for ensuring that only the genuinely visible parts of a scene are presented to the viewer, however. If you can determine the depth (the distance from the eye) of each object in your model, then you may be able to sort a list of the objects so that you can draw them from back to front — that is, draw the farthest first and the nearest last. In doing this, you will replace anything that is hidden by other objects that are nearer, resulting in a scene that shows just the visible content. This is a classical technique called the *painter's algorithm* (because it mimics the way a painter could create an image using opaque paints) that was widely used in more limited graphics systems, but it sometimes has real advantages over Z-buffering because it is faster (it doesn't require the pixel depth comparison for every pixel that is drawn) and because sometimes Z-buffering will give incorrect images, as we discuss when we discuss modeling transparency with blending in the color chapter.

Double buffering: As you specify geometry in your program, the geometry is modified by the modeling and projection transformations and the piece of the image as you specified it is written into the color buffer. It is the color buffer that actually is written to the screen to create the image seen by the viewer. Most graphics systems offer you the capability of having two color buffers — one that is being displayed (called the front buffer) and one into which current graphics content is being written (called the back buffer). Using these two buffers is called double buffering.

Because it can take some time to do all the work to create an image, if you are using only the front buffer you may end up actually watching the pixels changing as the image is created. If you were trying to create an animated image by drawing one image and then another, it would be disconcerting to use only one buffer because you would constantly see your image being drawn and then destroyed and re-drawn. Thus double buffering is essential to animated images and, in

fact, is used quite frequently for other graphics because it is more satisfactory to present a completed image instead of a developing image to a user. You must remember, however, that when an image is completed you must specify that the buffers are to be swapped, or the user will never see the new image!

Clipping planes: Clipping is the process of drawing with the portion of an image on one side of a plane drawn and the portion on the other side omitted. Recall from the discussion of geometric fundamentals that a plane is defined by a linear equation

```
Ax + By + Cz + D = 0
```

so it can be represented by the 4-tuple of real numbers `(A, B, C, D)`. The plane divides the space into two parts: that for which `Ax+By+Cz+D` is positive and that for which it is negative. When you define the clipping plane for your graphics API with the functions it provides, you will probably use the four coefficients of the equation above. The operation of the clipping process is that any points for which this value is negative will not be displayed; any points for which it is positive or zero will be displayed.

Clipping defines parts of the scene that you do *not* want to display — parts that are to be left out for any reason. Any projection operation automatically includes clipping, because it must leave out objects in the space to the left, right, above, below, in front, and behind the viewing volume. In effect, each of the planes bounding the viewing volume for the projection is also a clipping plane for the image. You may also want to define other clipping planes for an image. One important reason to include clipping might be to see what is inside an object instead of just seeing the object's surface; you can define clipping planes that go through the object and display only the part of the object on one side or another of the plane. Your graphics API will probably allow you to define other clipping planes as well.

While the clipping process is handled for you by the graphics API, you should know something of the processes it uses. Because we generally think of graphics objects as built of polygons, the key point in clipping is to clip line segments (the boundaries of polygons) against the clipping plane. As we noted above, you can tell what side of a plane contains a point `(x, y, z)` by testing the algebraic sign of the expression `Ax+By+Cz+D`. If this expression is negative for both endpoints of a line segment, the entire line must lie on the "wrong" side of the clipping plane and so is simply not drawn at all. If the expression is positive for both endpoints, the entire line must lie on the "right" side and is drawn. If the expression is positive for one endpoint and negative for the other, then you must find the point for which the equation `Ax+By+Cz+D=0` is satisfied and then draw the line segment from that point to the point whose value in the expression is positive. If the line segment is defined by a linear parametric equation, the equation becomes a linear equation in one variable and so is easy to solve.

In actual practice, there are often techniques for handling clipping that are even simpler than that described above. For example, you might make only one set of comparisons to establish the relationship between a vertex of an object and a set of clipping planes such as the boundaries of a standard viewing volume. You can then use these tests to drive a set of clipping operations. We leave the details to the standard literature on graphics techniques.

*Stereo viewing*

Stereo viewing gives us an opportunity to see some of these viewing processes in action. Let us say quickly that this should not be your first goal in creating images; it requires a bit of experience with the basics of viewing before it makes sense. Here we describe binocular viewing — viewing that requires you to converge your eyes beyond the computer screen or printed image, but that gives you the full effect of 3D when the images are converged. Other techniques are described in later chapters.

Stereo viewing is a matter of developing two views of a model from two viewpoints that represent the positions of a person's eyes, and then presenting those views in a way that the eyes can see individually and resolve into a single image. This may be done in many ways, including creating two individual printed or photographed images that are assembled into a single image for a viewing system such as a stereopticon or a stereo slide viewer. (If you have a stereopticon, it can be very interesting to use modern technology to create the images for this antique viewing system!) Later in this chapter we describe how to present these as two viewports in a single window on the screen with OpenGL.

When you set up two viewpoints in this fashion, you need to identify two eye points that are offset by a suitable value in a plane perpendicular to the up direction of your view. It is probably simplest is you define your up direction to be one axis (perhaps the z-axis) and your overall view to be aligned with one of the axes perpendicular to that (perhaps the x-axis). You can then define an offset that is about the distance between the eyes of the observer (or perhaps a bit less, to help the viewer's eyes converge), and move each eyepoint from the overall viewpoint by half that offset. This makes it easier for each eye to focus on its individual image and let the brain's convergence create the merged stereo image. The result can be quite startling if the eye offset is large so the pair exaggerates the front-to-back differences in the view, or it can be more subtle if you use modest offsets to represent realistic views. Figure 1.7 shows the effect of such stereo viewing with a full-color shaded model. Later we will consider how to set the stereo eyepoints in a more systematic fashion.

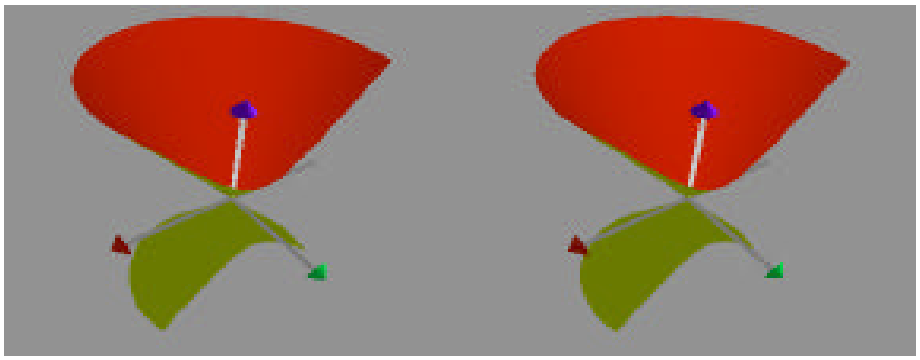


Figure 1.7: A stereo pair, including a clipping plane

Many people have physical limitations to their eyes and cannot perform the kind of eye convergence that this kind of stereo viewing requires. Some people have general convergence problems which do not allow the eyes to focus together to create a merged image, and some simply cannot seem to see beyond the screen to the point where convergence would occur. In addition, if you do not get the spacing of the stereo pair right, or have the sides misaligned, or allow the two sides to refresh at different times, or ... well, it can be difficult to get this to work well for users. If some of your users can see the converged image and some cannot, that's probably as good as it's going to be.

There are other techniques for doing 3D viewing. When we discuss texture maps later, we will describe a technique that colors 3D images more red in the near part and more blue in the distant part. This makes the images self-converge when you view them through a pair of ChromaDepth™ glasses, as we will describe there, so more people can see the spatial properties of the image, and it can be seen from anywhere in a room. There are also more specialized techniques such as creating alternating-eye views of the image on a screen with a overscreen that can be given alternating polarization and viewing them through polarized glasses that allow each eye to see only one screen at a time, or using dual-screen technologies such as head-mounted displays. The extension of the

techniques above to these more specialized technologies is straightforward and is left to your instructor if such technologies are available.

*Implementation of viewing and projection in OpenGL*

The OpenGL code below captures much of the code needed in the discussion that follows in this section. It could be taken froma single function or could be assembled from several functions; in the sample structure of an OpenGL program in the previous chapter we suggested that the viewing and projection operations be separated, with the first part being at the top of the `display()` function and the latter part being at the end of the `init()` and `reshape()` functions.

```
// Define the projection for the scene
   glViewport(0,0,(GLsizei)w,(GLsizei)h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0,(GLsizei)w/(GLsizei)h,1.0,30.0);

// Define the viewing environment for the scene
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   //           eye point     center of view       up
   gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

Defining a window and viewport: The window was defined in the previous chapter by a set of functions that initialize the window size and location and create the window. The details of window management are intentionally hidden from the programmer so that an API can work across many different platforms. In OpenGL, it is easiest to delegate the window setup to the GLUT toolkit where much of the system-dependent parts of OpenGL are defined; the functions to do this are:

```
glutInitWindowSize(width,height);
glutInitWindowPosition(topleftX,topleftY);
glutCreateWindow("Your window name here");
```

The viewport is defined by the `glViewport` function that specifies the lower left coordinates and the upper right coordinates for the portion of the window that will be used by the display. This function will normally be used in your initialization function for the program.

```
glViewport(VPLowerLeftX,VPLowerLeftY,VPUpperRightX,VPUpperRightY);
```

You can see the use of the viewport in the stereo viewing example below to create two separate images within one window.

Reshaping the window: The window is reshaped when it initially created or whenever is moved it to another place or made larger or smaller in any of its dimenstions. These reshape operations are handled easily by OpenGL because the computer generates an event whenever any of these window reshapes happens, and there is an event callback for window rehaping. We will discuss events and event callbacks in more detail later, but the reshape callback is registered by the function `glutReshapeFunc(reshape)` which identifies a function `reshape(GLint w,GLint h)` that is to be executed whenever the window reshape event occurs and that is to do whatever is necessary to regenerate the image in the window.

The work that is done when a window is reshaped can involve defining the projection and the viewing environment, updating the definition of the viewport(s) in the window, or can delegate some of these to the display function. Any viewport needs either to be defined inside the reshape callback function so it can be redefined for resized windows or to be defined in the display function where the changed window dimensions can be taken into account when it is defined. The viewport probably should be designed directly in terms relative to the size or dimensions of the window, so the parameters of the reshape function should be used. For example, if the window is defined to

have dimensions (width, height) as in the definition above, and if the viewport is to comprise the right-hand side of the window, then the viewport's coordinates are

```
(width/2, 0, width, height)
```

and the aspect ratio of the window is width/(2*height). If the window is resized, you will probably want to make the width of the viewport no larger than the larger of half the new window width (to preserve the concept of occupying only half of the window) or the new window height times the original aspect ratio. This kind of calculation will preserve the basic look of your images, even when the window is resized in ways that distort it far from its original shape.

Defining a viewing environment: To define what is usually called the viewing projection, you must first ensure that you are working with the GL_MODELVIEW matrix, then setting that matrix to be the identity, and finally define the viewing environment by specifying two points and one vector. The points are the eye point, the center of view (the point you are looking at), and the vector is the up vector — a vector that will be projected to define the vertical direction in your image. The only restrictions are that the eye point and center of view must be different, and the up vector must not be parallel to the vector from the eye point to the center of view. As we saw earlier, sample code to do this is:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
//            eye point     center of view       up
gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

The gluLookAt function may be invoked from the reshape function, or it may be put inside the display function and variables may be used as needed to define the environment. In general, we will lean towards including the gluLookAt operation at the start of the display function, as we will discuss below. See the stereo view discussion below for an idea of what that can do.

The effect of the gluLookAt(...) function is to define a transformation that moves the eye point from its default position and orientation. That default position and orientation has the eye at the origin and looking in the negative z-direction, and oriented with the y-axis pointing upwards. This is the same as if we invoked the gluLookAt function with the parameters

```
gluLookAt(0., 0., 0., 0., 0., -1., 0., 1., 0.).
```

When we change from the default value to the general eye position and orientation, we define a set of transformations that give the eye point the position and orientation we define. The overall set of transformations supported by graphics APIs will be discussed in the modeling chapter, but those used for defining the eyepoint are:

1. a rotation about the Z-axis that aligns the Y-axis with the up vector,
2. a scaling to place the center of view at the correct distance along the negative Z-axis,
3. a translation that moves the center of view to the origin,
4. two rotations, about the X- and Y-axes, that position the eye point at the right point relative to the center of view, and
5. a translation that puts the center of view at the right position.

In order to get the effect you want on your overall scene, then, the viewing transformation must be the inverse of the transformation that placed the eye at the position you define, because it must act on all the geometry in your scene to return the eye to the default position and orientation. Because function inverses act by

$$(F*G)^{-1} = G^{-1}*F^{-1}$$

the viewing transformation is built by inverting each of these five transformations in the revierse order. And because this must be done on all the geometry in the scene, it must be applied last — so it must be specified before any of the geometry is defined. We will thus usually see the gluLookAt(...) function as one of the first things to appear in the display() function, and its operation is the same as applying the transformations

1. translate the center of view to the origin,

2. rotate about the X- and Y-axes to put the eye point on the positive Z-axis,
3. translate to put the eye point at the origin,
4. scale to put the center of view at the point (0.,0.,-1.), and
5. rotate around the Z-axis to restore the up vector to the Y-axis.

You may wonder why we are discussing at this point how the `gluLookAt(...)` function defines the viewing transformation that goes into the modelview matrix, but we will need to know about this when we need to control the eye point as part of our modeling in more advanced kinds of scenes.

Defining perspective projection: a perspective projection is defined by first specifying that you want to work on the GL_PROJECTION matrix, and then setting that matrix to be the identity. You then specify the properties that will define the perspective transformation. In order, these are the field of view (an angle, in degrees, that defines the width of your viewing area), the aspect ratio (a ratio of width to height in the view; if the window is square this will probably be 1.0 but if it is not square, the aspect ratio will probably be the same as the ratio of the window width to height), the zNear value (the distance from the viewer to the plane that will contain the nearest points that can be displayed), and the zFar value (the distance from the viewer to the plane that will contain the farthest points that can be displayed). This sounds a little complicated, but once you've set it up a couple of times you'll find that it's very simple. It can be interesting to vary the field of view, though, to see the effect on the image.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0,1.0,1.0,30.0);
```

It is also possible to define your perspective projection by using the `glFrustum` function that defines the projection in terms of the viewing volume containing the visible items, as was shown in Figure 1.4 above. However, the `gluPerspective` function is so natural that we'll leave the other approach to the student who wants it.

Defining an orthogonal projection: an orthogonal projection is defined much like a perspective projection except that the parameters of the projection itself are different. As you can see in the illustration of a parallel projection in Figure 1.3, the visible objects lie in a box whose sides are parallel to the X-, Y-, and Z-axes in the viewing space. Thus to define the viewing box for an orthogonal projection, we simply define the boundaries of the box as shown in Figure 1.3 and the OpenGL system does the rest.

```
glOrtho(xLow,xHigh,yLow,yHigh,zNear,zFar);
```

The viewing space is still the same left-handed space as noted earlier, so the `zNear` and `zFar` values are the distance from the X-Y plane in the negative direction, so that negative values of `zNear` and `zFar` refer to positions behind the eye (that is, in positive Z-space). There is no alternate to this function in the way that the `glFrustum(...)` is an alternative to the `gluLookAt(...)` function for parallel projections.

Managing hidden surface viewing: in the Getting Started module when we introduced the structure of a program that uses OpenGL, we saw the `glutInitDisplayMode` function, called from `main`, as a way to define properties of the display. This function also allows the use of hidden surfaces if you specify `GLUT_DEPTH` as one of its parameters.

```
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

You must also enable the depth test. Enabling is a standard property of OpenGL; many capabilities of the system are only available after they are enabled through the `glEnable` function, as shown below.

```
glEnable(GL_DEPTH_TEST);
```

From that point the depth buffer is in use and you need not be concerned about hidden surfaces. If you want to turn off the depth test, there is a `glDisable` function as well. Note the use of these two functions in enabling and disabling the clipping plane in the `stereoView.c` example code.

Setting double buffering: double buffering is a standard facility, and you will note that the function above that initializes the display mode includes a parameter `GLUT_DOUBLE` to set up double buffering. In your `display()` function, you will call `glutSwapBuffers()` when you have finished creating the image, and that will cause the background buffer to be swapped with the foreground buffer and your new image will be displayed.

Defining clipping planes: In addition to the clipping OpenGL performs on the standard view volume in the projection operation, OpenGL allows you to define at least six clipping planes of your own, named `GL_CLIP_PLANE0` through `GL_CLIP_PLANE5`. The clipping planes are defined by the function `glClipPlane(plane, equation)` where `plane` is one of the pre-defined clipping planes above and `equation` is a vector of four `GLfloat` values. Once you have defined a clipping plane, it is enabled or disabled by a `glEnable(GL_CLIP_PLANEn)` function or equivalent `glDisable(...)` function. Clipping is performed when any modeling primitive is called when a clip plane is enabled; it is not performed when the clip plane is disabled. They are then enabled or disabled as needed to take effect in the scene. Specifically, some example code looks like

```
GLfloat myClipPlane[] = { 1.0, 1.0, 0.0, -1.0 };
glClipPlane(GL_CLIP_PLANE0, myClipPlane);
glEnable(GL_CLIP_PLANE0);
...
glDisable(GL_CLIP_PLANE0);
```

The stereo viewing example at the end of this chapter includes the definition and use of clipping planes.

*Implementing a stereo view*

In this section we describe the implementation of binocular viewing as described earlier in this chapter. The technique we will use is to generate two veiws of a single model as if they were seen from the viewer's separate eyes, and present these in two viewports in a single window on the screen. These two images are then manipulated together by manipulating the model as a whole, while viewer resolves these into a single image by focusing each eye on a separate image.

This latter process is fairly simple. First, create a window that is twice as wide as it is high, and whose overall width is twice the distance between your eyes. Then when you display your model, do so twice, with two different viewports that occupy the left and right half of the window. Each display is identical except that the eye points in the left and right halves represent the position of the left and right eyes, respectively. This can be done by creating a window with space for both viewports with the window initialization function

```
#define W 600
#define H 300
width = W; height = H;
glutInitWindowSize(width,height);
```

Here the initial values set the width to twice the height, allowing each of the two viewports to be initially square. We set up the view with the overall view at a distance of `ep` from the origin in the x-direction and looking at the origin with the z-axis pointing up, and set the eyes to be at a given offset distance from the overall viewpoint in the y-direction. We then define the left- and right-hand viewports in the `display()` function as follows

```
//    left-hand viewport
      glViewport(0,0,width/2,height);
      ...
```

```
//                eye point      center of view       up
     gluLookAt(ep, -offset, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
     ... code for the actual image goes here
     ...
//   right-hand viewport
     glViewport(width/2,0,width/2,height);
     ...
//                eye point      center of view       up
     gluLookAt(ep,  offset, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
     ... the same code as above for the actual image goes here
     ...
```

This particular code example responds to a `reshape(width,height)` operation because it uses the window dimensions to set the viewport sizes, but it is susceptible to distortion problems if the user does not maintain the 2:1 aspect ratio as he or she reshapes the window. It is left to the student to work out how to create square viewports within the window if the window aspect ratio is changed.

# Modeling

*Prerequisites*

This chapter requires an understanding of simple 3-dimensional geometry, knowledge of how to represent points in 3-space, enough programming experience to be comfortable writing code that calls API functions to do required tasks, ability to design a program in terms of simple data structures such as stacks, and an ability to organize things in 3D space.

*Introduction*

Modeling is the process of defining the geometry that makes up a scene and implementing that definition with the tools of your graphics API. This chapter is critical in developing your ability to create graphical images and takes us from quite simple modeling to fairly complex modeling based on hierarchical structures, and discusses how to implement each of these different stages of modeling in OpenGL. It is fairly comprehensive for the kinds of modeling one would want to do with a basic graphics API, but there are other kinds of modeling used in advanced API work and some areas of computer graphics that involve more sophisticated kinds of constructions than we include here, so we cannot call this a genuinely comprehensive discussion. It is, however, a good enough introduction to give you the tools to start creating interesting images.

The chapter has four distinct parts because there are four distinct levels of modeling that you can use to create images. We begin with simple geometric modeling: modeling where you define the coordinates of each vertex of each component you will use at the point where that component will reside in the final scene. This is straightforward but can be very time-consuming to do for complex scenes, so we will also discuss importing models from various kinds of modeling tools that can allow you to create parts of a scene more easily.

The second section describes the next step in modeling. Here we extend the utility of your simple modeling by defining the primitive transformations you can use for computer graphics operations and by discussing how you can start with simple modeling and use transformations to create more general model components in your scene. This is a very important part of the modeling process because it allows you to build standard templates for many different graphic objects and then place them in your scene with the appropriate transformations. These transformations are also critical to the ability to define and implement motion in your scenes because it is typical to move objects, lights, and the eyepoint with transformations that are controlled by parameters that change with time. This can allow you to extend your modeling to define animations that can represent such concepts as changes over time.

In the third section of the chapter we introduce the concept of the scene graph, a modeling tool that gives you a unified approach to defining all the objects and transformations that are to make up a scene and to specifying how they are related and presented. We then describe how you work from the scene graph to write the code that implements your model. This concept is new to the introductory graphics course but has been used in some more advanced graphics tools, and we believe you will find it to make the modeling process much more straightforward for anything beyond a very simple scene. In the second level of modeling discussed in this section, we introduce hierarchical modeling in which objects are designed by assembling other objects to make more complex structures. These structures can allow you to simulate actual physical assemblies and develop models of structures like physical machines. Here we develop the basic ideas of scene graphs introduced earlier to get a structure that allows individual components to move relative to each other in ways that would be difficult to define from first principles.

Finally, the fourth section of the chapter covers the implementation of modeling in the OpenGL API. This includes the set of operations that implement polygons, as well as those that provide the

geometry compression that we describe in the first section. This section also describes the use of OpenGL's pre-defined geometric components that you can use directly in your images to let you use more complex objects without having to determine all the vertices directly, but that are defined only in standard positions so you must use transformations to place them correctly in your scenes. It also includes a discussion of transformations and how they are used in OpenGL, and describes how to implement a scene graph with this API.

Following these discussions, this chapter concludes with an appendix on the mathematical background that you will find useful in doing your modeling. This may be a review for you, or it may be new; if it is new and unfamiliar, you might want to look at some more detailed reference material on 3D analytic geometry.

# Simple Geometric Modeling

*Introduction*

Computer graphics deals with geometry and its representation in ways that allow it to be manipulated and displayed by a computer. Because these notes are intended for a first course in the subject, you will find that the geometry will be simple and will use familiar representations of 3-dimensional space. When you work with a graphics API, you will need to work with the kinds of object representations that API understands, so you must design your image or scene in ways that fit the API's tools. For most APIs, this means using only a few simple graphics primitives, such as points, line segments, and polygons.

The space we will use for our modeling is simple Euclidean 3-space with standard coordinates, which we will call the X-, Y-, and Z-coordinates. Figure 2.1 below illustrates a point, a line segment, a polygon, and a polyhedron—the basic elements of the computer graphics world that you will use for most of your graphics. In this space a *point* is simply a single location in 3-space, specified by its coordinates and often seen as a triple of real numbers such as `(px, py, pz)`. A point is drawn on the screen by lighting a single pixel at the screen location that best represents the location of that point in space. To draw the point you will specify that you want to draw points and specify the point's coordinates, usually in 3-space, and the graphics API will calculate the coordinates of the point on the screen that best represents that point and will light that pixel. Note that a point is usually presented as a square, not a dot, as indicated in the figure. A *line segment* is determined by its two specified endpoints, so to draw the line you indicate that you want to draw lines and define the points that are the two endpoints. Again, these endpoints are specified in 3-space and the graphics API calculates their representations on the screen, and draws the line segment between them. A *polygon* is a region of space that lies in a plane and is bounded in the plane by a collection of line segments. It is determined by a sequence of points (called the *vertices* of the polygon) that specify a set of line segments that form its boundary, so to draw the polygon you indicate that you want to draw polygons and specify the sequence of vertex points. A *polyhedron* is a region of 3-space bounded by polygons, called the faces of the polyhedron. A polyhedron is defined by specifying a sequence of faces, each of which is a polygon. Because figures in 3-space determined by more than three vertices cannot be guaranteed to line in a plane, polyhedra are often defined to have triangular faces; a triangle always lies in a plane (because three points in 3-space determine a plane. As we will see when we discuss lighting and shading in subsequent chapters, the direction in which we go around the vertices of each face of a polygon is very important, and whenever you design a polyhedron, you should plan your polygons so that their vertices are ordered in a sequence that is counterclockwise as seen from outside the polyhedron (or, to put it another way, that the angle to each vertex as seen from a point inside the face is increasing rather than decreasing as you go around each face).

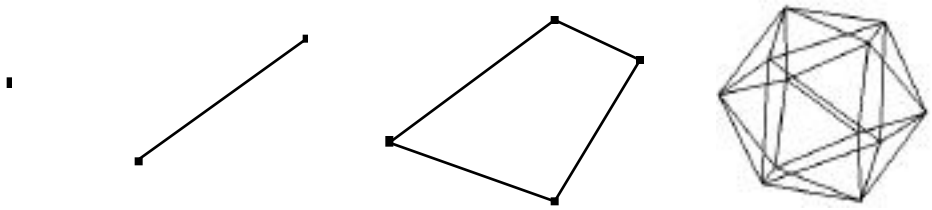


Figure 2.1: a point, a line segment, a polygon, and a polyhedron

Before you can create an image, you must define the objects that are to appear in that image through some kind of modeling process. Perhaps the most difficult—or at least the most time-consuming—part of beginning graphics programming is creating the models that are part of the image you want to create. Part of the difficulty is in designing the objects themselves, which may

require you to sketch parts of your image by hand so you can determine the correct values for the points used in defining it, for example, or it may be possible to determine the values for points from some other technique. Another part of the difficulty is actually entering the data for the points in an appropriate kind of data structure and writing the code that will interpret this data as points, line segments, and polygons for the model. But until you get the points and their relationships right, you will not be able to get the image right.

*Definitions*

We need to have some common terminology as we talk about modeling. We will think of modeling as the process of defining the objects that are part of the scene you want to view in an image. There are many ways to model a scene for an image; in fact, there are a number of commercial programs you can buy that let you model scenes with very high-level tools. However, for much graphics programming, and certainly as you are beginning to learn about this field, you will probably want to do your modeling by defining your geometry in terms of relatively simple primitive terms so you may be fully in control of the modeling process.

Besides defining a single point, line segment, or polygon, graphics APIs provide modeling support for defining larger objects that are made up of several simple objects. These can involve disconnected sets of objects such as points, line segments, quads, or triangles, or can involve connected sets of points, such as line segments, quad strips, triangle strips, or triangle fans. This allows you to assemble simpler components into more complex groupings and is often the only way you can define polyhedra for your scene. Some of these modeling techniques involve a concept called *geometry compression*, which allow you to define a geometric object using fewer vertices than would normally be needed. The OpenGL support for geometry compression will be discussed as part of the general discussion of OpenGL modeling processes. The discussions and examples below will show you how to build your repertoire of techniques you can use for your modeling.

Before going forward, however, we need to mention another way to specify points for your models. In some cases, it can be helpful to think of your 3-dimensional space as embedded as an affine subspace of 4-dimensional space. If we think of 4-dimensional space as having X, Y, Z, and W components, this embedding identifies the three-dimensional space with the subspace `W=1` of the four-dimensional space, so the point `(x,y,z)` is identified with the four-dimensional point `(x,y,z,1)`. Conversely, the four-dimensional point `(x,y,z,w)` is identified with the three-dimensional point `(x/w,y/w,z/w)` whenever `w≠0`. The four-dimensional representation of points with a non-zero `w` component is called *homogeneous coordinates*, and calculating the three-dimensional equivalent for a homogeneous representation by dividing by `w` is called *homogenizing* the point. When we discuss transformations, we will sometimes think of them as 4x4 matrices because we will need them to operate on points in homogeneous coordinates.

Not all points in 4-dimensional space can be identified with points in 3-space, however. The point `(x,y,z,0)` is not identified with a point in 3-space because it cannot be homogenized, but it is identified with the direction defined by the vector `<x,y,z>`. This can be thought of as a “point at infinity” in a certain direction. This has an application in the chapter below on lighting when we discuss directional instead of positional lights, but in general we will not encounter homogeneous coordinates often in these notes.

*Some examples*

We will begin with very simple objects and proceed to more useful ones. With each kind of primitive object, we will describe how that object is specified, and in later examples, we will create a set of points and will then show the function call that draws the object we have defined.

Point and points

To draw a single point, we will simply define the coordinates of the point and give them to the graphics API function that draws points. Such a function can typically handle one point or a number of points, so if we want to draw only one point, we provide only one vertex; if we want to draw more points, we provide more vertices. Points are extremely fast to draw, and it is not unreasonable to draw tens of thousands of points if a problem merits that kind of modeling. On a very modest-speed machine without any significant graphics acceleration, a 50,000 point model can be re-drawn in a small fraction of a second.

Line segments

To draw a single line segment, we must simply supply two vertices to the graphics API function that draws lines. Again, this function will probably allow you to specify a number of line segments and will draw them all; for each segment you simply need to provide the two endpoints of the segment. Thus you will need to specify twice as many vertices as the number of line segments you wish to produce.

Connected lines

Connected lines—collections of line segments that are joined "head to tail" to form a longer connected group—are shown in Figure 2.2. These are often called line strips, and your graphics API will probably provide a function for drawing them. The vertex list you use will define the line segments by using the first two vertices for the first line segment, and then by using each new vertex and its predecessor to define each additional segment. Thus the number of line segments drawn by the function will be one fewer than the number of vertices in the vertex list. This is a geometry compression technique because to define a line strip with N segments you only specify N+1 vertices instead of 2N vertices; instead of needing to define two points per line segment, each segment after the first only needs one vertex to be defined.

Figure 2.2: a line strip

Triangle

To draw one or more unconnected triangles, your graphics API will provide a simple triangle-drawing function. With this function, each set of three vertices will define an individual triangle so that the number of triangles defined by a vertex list is one third the number of vertices in the list. The humble triangle may seem to be the most simple of the polygons, but as we noted earlier, it is probably the most important because no matter how you use it, and no matter what points form its vertices, it always lies in a plane. Because of this, most polygon-based modeling really comes down to triangle-based modeling in the end, and almost every kind of graphics tool knows how to manage objects defined by triangles. So treat this humblest of polygons well and learn how to think about polygons and polyhedra in terms of the triangles that make them up.

Sequence of triangles

Triangles are the foundation of most truly useful polygon-based graphics, and they have some very useful capabilities. Graphics APIs often provide two different geometry-compression techniques to assemble sequences of triangles into your image: triangle strips and triangle fans. These

techniques can be very helpful if you are defining a large graphic object in terms of the triangles that make up its boundaries, when you can often find ways to include large parts of the object in triangle strips and/or fans. The behavior of each is shown in Figure 2.3 below. Note that this figure and similar figures that show simple geometric primitives are presented as if they were drawn in 2D space. In fact they are not, but in order to make them look three-dimensional we would need to use some kind of shading, which is a separate concept discussed in a later chapter (and which is used to present the triangle fan of Figure 2.18). We thus ask you to think of these as three-dimensional, even though they look flat.

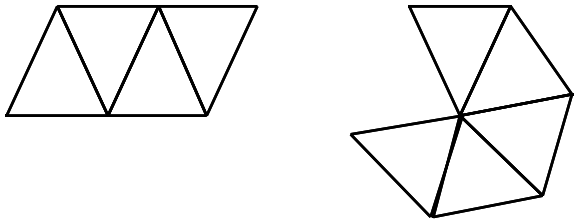


Figure 2.3: triangle strip and triangle fan

Most graphics APIs support both techniques by interpreting the vertex list in different ways. To create a triangle strip, the first three vertices in the vertex list create the first triangle, and each vertex after that creates a new triangle with the two vertices immediately before it. We will see in later chapters that the order of points around a polygon is important, and we must point out that these two techniques behave quite differently with respect to polygon order; for triangle fans, the orientation of all the triangles is the same (clockwise or counterclockwise), while for triangle strips, the orientation of alternate triangles is reversed. This may require some careful coding when lighting models are used. To create a triangle fan, the first three vertices create the first triangle and each vertex after that creates a new triangle with the point immediately before it and the first point in the list. In each case, the number of triangles defined by the vertex list is two less than the number of vertices in the list, so these are very efficient ways to specify triangles.

Quadrilateral

A convex quadrilateral, often called a “quad” to distinguish it from a general quadrilateral because the general quadrilateral need not be convex, is any convex 4-sided figure. The function in your graphics API that draws quads will probably allow you to draw a number of them. Each quadrilateral requires four vertices in the vertex list, so the first four vertices define the first quadrilateral, the next four the second quadrilateral, and so on, so your vertex list will have four times as many points as there are quads. The sequence of vertices is that of the points as you go around the perimeter of the quadrilateral. In an example later in this chapter, we will use six quadrilaterals to define a cube that will be used in later examples.

Sequence of quads

You can frequently find large objects that contain a number of connected quads. Most graphics APIs have functions that allow you to define a sequence of quads. The vertices in the vertex list are taken as vertices of a sequence of quads that share common sides. For example, the first four vertices can define the first quad; the last two of these, together with the next two, define the next quad; and so on The order in which the vertices are presented is shown in Figure 2.4. Note the order of the vertices; instead of the expected sequence around the quads, the points in each pair have the same order. Thus the sequence 3-4 is the opposite order than would be expected, and this same sequence goes on in each additional pair of extra points. This difference is critical to note when you are implementing quad strip constructions. It might be helpful to think of this in terms

of triangles, because a quad strip acts as though its vertices were specified as if it were really a triangle strip — vertices 1/2/3 followed by 2/3/4 followed by 3/4/5 etc.

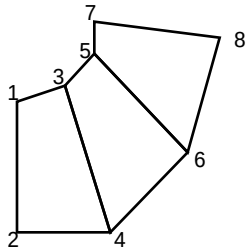



Figure 2.4: sequence of points in a quad strip

A good example of the use of quad strips and triangle fans would be creating your own model of a sphere. As we will see later in this chapter, there are pre-built sphere models from both the GLU and GLUT toolkits in OpenGL, but the sphere is a familiar object and it can be helpful to see how to create familiar things with new tools. There may also be times when you need to do things with a sphere that are difficult with the pre-built objects, so it is useful to have this example in your "bag of tricks."

Recall the discussion of spherical coordinates in the discussion of mathematical fundamentals. We can use spherical coordinates to model our object at first, and then convert to Cartesian coordinates to present the model to the graphics system for actual drawing. Let's think of creating a model of the sphere with N divisions around the equator and N/2 divisions along the prime meridian. In each case, then, the angular division will be theta = 360/N degrees. Let's also think of the sphere as having a unit radius, so it will be easier to work with later when we have transformations. Then the basic structure would be:

```
// create the two polar caps with triangle fans
doTriangleFan()      // north pole
   set vertex at (1, 0, 90)
   for i = 0 to N
      set vertex at (1, 360/i, 90-180/N)
endTriangleFan()
doTriangleFan()      // south pole
   set vertex at (1, 0, -90)
   for i = 0 to N
      set vertex at (1, 360/i, -90+180/N)
endTriangleFan()
// create the body of the sphere with quad strips
for j = -90+180/N to 90 - 180/2N
   // one quad strip per band around the sphere at a given latitude
   doQuadStrip()
      for i = 0 to 360
         set vertex at (1, i, j)
         set vertex at (1, i, j+180/N)
         set vertex at (1, i+360/N, j)
         set vertex at (1, i+360/N, j+180/N)
   endQuadStrip()
```

Note the order in which we set the points in the triangle fans and in the quad strips, as we noted when we introduced these concepts; this is not immediately an obvious order and you may want to think about it a bit. Because we're working with a sphere, the quad strips as we have defined them are planar, so there is no need to divide each quad into two triangles to get planar surfaces.

General polygon

Some images need to include more general kinds of polygons. While these can be created by constructing them manually as collections of triangles and/or quads, it might be easier to define and display a single polygon. A graphics API will allow you to define and display a single polygon by specifying its vertices, and the vertices in the vertex list are taken as the vertices of the polygon in sequence order. As we noted in the earlier chapter on mathematical fundamentals, many APIs can only handle *convex* polygons — polygons for which any two points in the polygon also have the entire line segment between them in the polygon. We refer you to that earlier discussion for more details.

Normals

When you define the geometry of an object, you may also want or need to define the direction the object faces as well as the xxx. This is done by defining a normal for the object. Normals are often fairly easy to obtain. In the appendix to this chapter you will see ways to calculate normals for plane polygons fairly easily; for many of the kinds of objects that are available with a graphics API, normals are built into the object definition; and if an object is defined by mathematical formulas, you can often get normals by doing some straightforward calculations.

The sphere described above is a good example of getting normals by calculation. For a sphere, the normal to the sphere at a given point is the radius vector at that point. For a unit sphere with center at the origin, the radius vector to a point has the same components as the coordinates of the point. So if you know the coordinates of the point, you know the normal at that point.

To add the normal information to the modeling definition, then, you can simply use functions that set the normal for a geometric primitive, as you would expect to have from your graphics API, and you would get code that looks something like the following excerpt from the example above:

```
for j = -90+180/N to 90 - 180/2N
   // one quad strip per band around the sphere at a given latitude
   doQuadStrip()
      for i = 0 to 360
         set normal to (1, i, j)
         set vertex at (1, i, j)
         set vertex at (1, i, j+180/N)
         set vertex at (1, i+360/N, j)
         set vertex at (1, i+360/N, j+180/N)
   endQuadStrip()
```

Data structures to hold objects

There are many ways to hold the information that describes a graphic object. One of the simplest is the *triangle list* — an array of triples, and each set of three triples represents a separate triangle. Drawing the object is then a simple matter of reading three triples from the list and drawing the triangle. A good example of this kind of list is the STL graphics file format discussed in the chapter below on graphics hardcopy.

A more effective, though a bit more complex, approach is to create three lists. The first is a *vertex list*, and it is simply an array of triples that contains all the vertices that would appear in the object. If the object is a polygon or contains polygons, the second list is an edge list that contains an entry for each edge of the polygon; the entry is an ordered pair of numbers, each of which is an index of a point in the vertex list. If the object is a polyhedron, the third is a *face list*, containing information on each of the faces in the polyhedron. Each face is indicated by listing the indices of

all the edges that make up the face, in the order needed by the orientation of the face. You can then draw the face by using the indices as an indirect reference to the actual vertices. So to draw the object, you loop across the face list to draw each face; for each face you loop across the edge list to determine each edge, and for each edge you get the vertices that determine the actual geometry.

As an example, let's consider the classic cube, centered at the origin and with each side of length two. For the cube let's define the vertex array, edge array, and face array that define the cube, and let's outline how we could organize the actual drawing of the cube. We will return to this example later in this chapter and from time to time as we discuss other examples throughout the notes.

We begin by defining the data and data types for the cube. The vertices are points, which are arrays of three points, while the edges are pairs of indices of points in the point list and the faces are quadruples of indices of faces in the face list. In C, these would be given as follows:

```
typedef float point3[3];
typedef int   edge[2];
typedef int   face[4];    // assumes a face has four edges for this example

     point3 vertices[8] = {{-1.0, -1.0, -1.0},
                           {-1.0, -1.0,  1.0},
                           {-1.0,  1.0, -1.0},
                           {-1.0,  1.0,  1.0},
                           { 1.0, -1.0, -1.0},
                           { 1.0, -1.0,  1.0},
                           { 1.0,  1.0, -1.0},
                           { 1.0,  1.0,  1.0} };

     edge   edges[24]   = {{ 0, 1 }, { 1, 3 }, { 3, 2 }, { 2, 0 },
                           { 0, 4 }, { 1, 5 }, { 3, 7 }, { 2, 6 },
                           { 4, 5 }, { 5, 7 }, { 7, 6 }, { 6, 4 },
                           { 1, 0 }, { 3, 1 }, { 2, 3 }, { 0, 2 },
                           { 4, 0 }, { 5, 1 }, { 7, 3 }, { 6, 2 },
                           { 5, 4 }, { 7, 5 }, { 6, 7 }, { 4, 6 }};

     face   cube[6]     = {{  0,  1,  2,  3 }, {  5,  9, 18, 13 },
                           { 14,  6, 10, 19 }, {  7, 11, 16, 15 },
                           {  4,  8, 17, 12 }, { 22, 21, 20, 23 }};
```

Notice that in our edge list, each edge is actually listed twice—once for each direction the in which the edge can be drawn. We need this distinction to allow us to be sure our faces are oriented properly, as we will describe in the discussion on lighting and shading in later chapters. For now, we simply ensure that each face is drawn with edges in a counterclockwise direction as seen from outside that face of the cube. Drawing the cube, then, proceeds by working our way through the face list and determining the actual points that make up the cube so they may be sent to the generic (and fictitious) `setVertex(...)` function. In a real application we would have to work with the details of a graphics API, but here we sketch how this would work in a pseudocode approach. In this pseudocode, we assume that there is no automatic closure of the edges of a polygon so we must list both the vertex at both the beginning and the end of the face when we define the face; if this is not neede by your API, then you may omit the first `setVertex` call in the pseudocode for the function `cube()` below.

```
   void cube(void) {
      for faces 1 to 6
         start face
            setVertex(vertices[edges[cube[face][0]][0]);
            for each edge in the face
```

```
            setVertex(vertices[edges[cube[face][edge]][1]);
        end face
    }
```

In addition to the vertex list, you may want to add a structure for a list of normals. In many kinds of modeling, each vertex will have a normal representing the perpendicular to the object at that vertex. In this case, you often need to specify the normal each time you specify a vertex, and the normal list would allow you to do that easily. For the code above, for example, each `setVertex` operation could be replaced by the pair of operations

```
setNormal(normals[edges[cube[face][0]][0]);
setVertex(vertices[edges[cube[face][0]][0]);
```

Neither the simple triangle list nor the more complex structure of vertex, normal, edge, and face lists takes into account the very significant savings in memory you can get by using geometry compression techniques. There are a number of these techniques, but we only talked about line strips, triangle strips, triangle fans, and quad strips above because these are more often supported by a graphics API. Geometry compression approaches not only save space, but are also more effective for the graphics system as well because they allow the system to retain some of the information it generates in rendering one triangle or quad when it goes to generate the next one.

Additional sources of graphic objects

Interesting and complex graphic objects can be difficult to create, because it can take a lot of work to measure or calculate the detailed coordinates of each vertex needed. There are more automatic techniques being developed, including 3D scanning techniques and detailed laser rangefinding to measure careful distances and angles to points on an object that is being measured, but they are out of the reach of most college classrooms. So what do we do to get interesting objects? There are four approaches.

The first way to get models is to buy them: to go is to the commercial providers of 3D models. There is a serious market for some kinds of models, such as medical models of human structures, from the medical and legal worlds. This can be expensive, but it avoids having to develop the expertise to do professional modeling and then putting in the time to create the actual models. If you are interested, an excellent source is viewpoint.com; they can be found on the Web.

A second way to get models is to find them in places where people make them available to the public. If you have friends in some area of graphics, you can ask them about any models they know of. If you are interested in molecular models, the protein data bank (with URL `http://www.pdb.bnl.gov`) has a wide range of structure models available at no charge. If you want models of all kinds of different things, try the site `avalon.viewpoint.com`; this contains a large number of public-domain models contributed to the community by generous people over the years.

A third way to get models is to digitize them yourself with appropriate kinds of digitizing devices. There are a number of these available with their accuracy often depending on their cost, so if you need to digitize some physical objects you can compare the cost and accuracy of a number of possible kinds of equipment. The digitizing equipment will probably come with tools that capture the points and store the geometry in a standard format, which may or may not be easy to use for your particular graphics API. If it happens to be one that your API does not support, you may need to convert that format to one you use or to find a tool that does that conversion.

A fourth way to get models is to create them yourself. There are a number of tools that support high-quality interactive 3D modeling, and it is no shame to create your models with such tools. This has the same issue as digitizing models in terms of the format of the file that the tools

produce, but a good tool should be able to save the models in several formats, one of which you could use fairly easily with your graphics API. It is also possible to create interesting models analytically, using mathematical approaches to generate the vertices. This is perhaps slower than getting them from other sources, but you have final control over the form and quality of the model, so perhaps it might be worth the effort. This will be discussed in the chapter on interpolation and spline modeling, for example.

If you get models from various sources, you will probably find that they come in a number of different kinds of data format. There are a large number of widely used formats for storing graphics information, and it sometimes seems as though every graphics tool uses a file format of its own. Some available tools will open models with many formats and allow you to save them in a different format, essentially serving as format converters as well as modeling tools. In any case, you are likely to end up needing to understand some model file formats and writing your own tools to read these formats and produce the kind of internal data that you need for your models, and it may take some work to write filters that will read these formats into the kind of data structures you want for your program. Perhaps things that are "free" might cost more than things you buy if you can save the work of the conversion — but that's up to you to decide. An excellent resource on file formats is the *Encyclopedia of Graphics File Formats*, published by O'Reilly Associates, and we refer you to that book for details on particular formats.

A word to the wise...

As we said above, modeling can be the most time-consuming part of creating an image, but you simply aren't going to create a useful or interesting image unless the modeling is done carefully and well. If you are concerned about the programming part of the modeling for your image, it might be best to create a simple version of your model and get the programming (or other parts that we haven't talked about yet) done for that simple version. Once you are satisfied that the programming works and that you have gotten the other parts right, you can replace the simple model — the one with just a few polygons in it — with the one that represents what you really want to present.

# Transformations and Modeling

This section requires some understanding of 3D geometry, particularly a sense of how objects can be moved around in 3-space. You should also have some sense of how the general concept of stacks works.

*Introduction*

Transformations are probably the key point in creating significant images in any graphics system. It is extremely difficult to model everything in a scene in the place where it is to be placed, and it is even worse if you want to move things around in real time through animation and user control. Transformations let you define each object in a scene in any space that makes sense for that object, and then place it in the world space of a scene as the scene is actually viewed. Transformations can also allow you to place your eyepoint and move it around in the scene.

There are several kinds of transformations in computer graphics: projection transformations, viewing transformations, and modeling transformations. Your graphics API should support all of these, because all will be needed to create your images. Projection transformations are those that specify how your scene in 3-space is mapped to the 2D screen space, and are defined by the system when you choose perspective or orthogonal projections; viewing transformations are those that allow you to view your scene from any point in space, and are set up when you define your view environment, and modeling transformations are those you use to create the items in your scene and are set up as you define the position and relationships of those items. Together these make up the graphics pipeline that we discussed in the first chapter of these notes.

Among the modeling transformations, there are three fundamental kinds: rotations, translations, and scaling. These all maintain the basic geometry of any object to which they may be applied, and are fundamental tools to build more general models than you can create with only simple modeling techniques. Later in this chapter we will describe the relationship between objects in a scene and how you can build and maintain these relationships in your programs.

The real power of modeling transformation, though, does not come from using these simple transformations on their own. It comes from combining them to achieve complete control over your modeled objects. The individual simple transformations are combined into a composite modeling transformation that is applied to your geometry at any point where the geometry is specified. The modeling transformation can be saved at any state and later restored to that state to allow you to build up transformations that locate groups of objects consistently. As we go through the chapter we will see several examples of modeling through composite transformations.

Finally, the use of simple modeling and transformations together allows you to generate more complex graphical objects, but these objects can take significant time to display. You may want to store these objects in pre-compiled display lists that can execute much more quickly.

*Definitions*

In this section we outline the concept of a geometric transformation and describe the fundamental transformations used in computer graphics, and describe how these can be used to build very general graphical object models for your scenes.

Transformations

A transformation is a function that takes geometry and produces new geometry. The geometry can be anything a computer graphics systems works with—a projection, a view, a light, a direction, or

an object to be displayed. We have already talked about projections and views, so in this section we will talk about projections as modeling tools. In this case, the transformation needs to preserve the geometry of the objects we apply them to, so the basic transformations we work with are those that maintain geometry, which are the three we mentioned earlier: rotations, translations, and scaling. Below we look at each of these transformations individually and together to see how we can use transformations to create the images we need.

Our vehicle for looking at transformations will be the creation and movement of a rugby ball. This ball is basically an ellipsoid (an object that is formed by rotating an ellipse around its major axis), so it is easy to create from a sphere using scaling. Because the ellipsoid is different along one axis from its shape on the other axes, it will also be easy to see its rotations, and of course it will be easy to see it move around with translations. So we will first discuss scaling and show how it is used to create the ball, then discuss rotation and show how the ball can be rotated around one of its short axes, then discuss translations and show how the ball can be moved to any location we wish, and finally will show how the transformations can work together to create a rotating, moving ball like we might see if the ball were kicked. The ball is shown with some simple lighting and shading as described in the chapters below on these topics.

*Scaling* changes the entire coordinate system in space by multiplying each of the coordinates of each point by a fixed value. Each time it is applied, this changes each dimension of everything in the space. A scaling transformation requires three values, each of which controls the amount by which one of the three coordinates is changed, and a graphics API function to apply a scaling transformation will take three real values as its parameters. Thus if we have a point `(x, y, z)` and specify the three scaling values as `Sx`, `Sy`, and `Sz`, then the point is changed to `(x*Sx, y*Sy, z*Sz)` when the scaling transformation is applied. If we take a simple sphere that is centered at the origin and scale it by 2.0 in one direction (in our case, the y-coordinate or vertical direction), we get the rugby ball that is shown in Figure 2.4 next to the original sphere. It is important to note that this scaling operates on everything in the space, so if we happen to also have a unit sphere at position farther out along the axis, scaling will move the sphere farther away from the origin and will also multiply each of its coordinates by the scaling amount, possibly distorting its shape. This shows that it is most useful to apply scaling to an object defined at the origin so only the dimensions of the object will be changed.

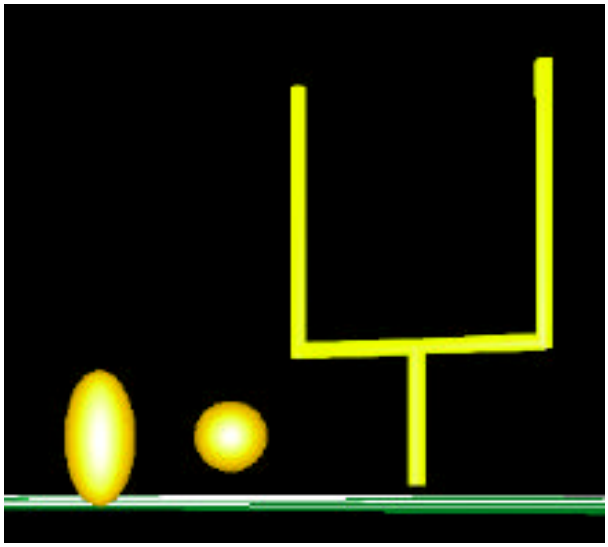


Figure 2.4: a sphere a scaled by 2.0 in the y-direction to make a rugby ball (left) and the same sphere is shown unscaled (right)

*Rotation* takes everything in your space and changes each coordinate by rotating it around the origin of the geometry in which the object is defined. The rotation will always leave a line through the origin in the space fixed, that is, will not change the coordinates of any point on that line. To define a rotation transformation, you need to specify the amount of the rotation (in degrees or radians, as needed) and the line about which the rotation is done. A graphics API function to apply a rotation transformation, then, will take the angle and the line as its parameters; remember that a line through the origin can be specified by three real numbers that are the coordinates of the direction vector for that line. It is most useful to apply rotations to objects centered at the origin in order to change only the orientation with the transformation.

*Translation* takes everything in your space and changes each point's coordinates by adding a fixed value to each coordinate. The effect is to move everything that is defined in the space by the same amount. To define a translation transformation, you need to specify the three values that are to be added to the three coordinates of each point. A graphics API function to apply a translation, then, will take these three values as its parameters. A translation shows a very consistent treatment of everything in the space, so a translation is usually applied after any scaling or rotation in order to take an object with the right size and right orientation and place it correctly in space.

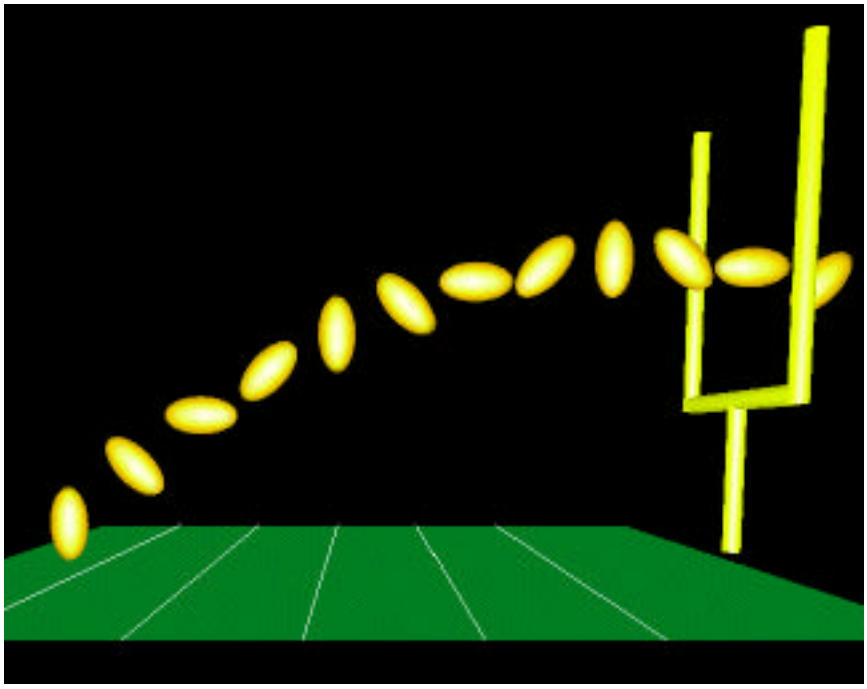


Figure 2.5: a sequence of images of the rugby ball as transformations move it through space

Finally, we put these three kinds of transformations together to create a sequence of images of the rugby ball as it moves through space, rotating as it goes, shown in Figure 2.5. This sequence was created by first defining the rugby ball with a scaling transformation and a translation putting it on the ground appropriately, creating a composite transformation as discussed in the next section. Then rotation and translation values were computed for several times in the flight of the ball, allowing us to rotate the ball by slowly-increasing amounts and placing it as if it were in a standard gravity field. Each separate image was created with a set of transformations that can be generically described by

```
translate( Tx, Ty, Tz )
rotate( angle, x-axis )
scale( 1., 2., 1. )
drawBall()
```

where the operation `drawBall()` was defined as

```
translate( Tx, Ty, Tz )
scale( 1., 2., 1. )
drawSphere()
```

Notice that the ball rotates in a slow counterclockwise direction as it travel from left to right, while the position of the ball describes a parabola as it moves, modeling the effect of gravity on the ball's flight. This kind of composite transformation constructions is described in the next section, and as we point out there, the order of these transformation calls is critical in order to achieve the effect we need.

Composite transformations

In order to achieve the image you want, you may need to apply more than one simple transformation to achieve what is called a composite transformation. For example, if you want to create a rectangular box with height A, width B, and depth C, with center at (C1,C2,C3), and oriented at an angle A relative to the Z-axis, you could start with a cube one unit on a side and with center at the origin, and get the box you want by applying the following sequence of operations:

first, scale the cube to the right size to create the rectangular box with dimensions A, B, C,
second, rotate the cube by the amount A to the right orientation, and
third, translate the cube to the position C1, C2, C3.

This sequence is critical because of the way transformations work in the whole space. For example, if we rotated first and then scaled with different scale factors in each dimension, we would introduce distortions in the box. If we translated first and then rotated, the rotation would move the box to an entirely different place. Because the order is very important, we find that there are certain sequences of operations that give predictable, workable results, and the order above is the one that works best: apply scaling first, apply rotation second, and apply translation last.

The order of transformations is important in ways that go well beyond the translation and rotation example above. In general, transformations are an example of *noncommutative* operations, operations for which $f*g \neq g*f$ (that is, $f(g(x)) \neq g(f(x))$ ). Most students have little experience with noncommutative operations until you get to a linear algebra course, so this may be a new idea. But let's look at the operations we described above: if we take the point (1, 1, 0) and apply a rotation by 90° around the Z-axis, we get the point (-1, 1, 0). If we then apply a translation by (2, 0, 0) we get the point (1, 1, 0) again. However, if we start with (1, 1, 0) and first apply the translation, we get (3, 1, 0) and if then apply the rotation, we get the point (-1, 3, 0) which is certainly not the same as (1, 1, 0). That is, using some pseudocode for rotations, translations, and point setting, the two code sequences

```
rotate(90, 0, 0, 1)          translate(2, 0, 0)
translate (2, 0, 0)          rotate(90, 0, 0, 1)
setPoint(1, 1, 0)            setPoint(1, 1, 0)
```

produce very different results; that is, the rotate and translate operations are not commutative.

This behavior is not limited to different kinds of transformations. Different sequences of rotations can result in different images as well. Again, if you consider the sequence of rotations

(sequence here)

and the same rotations in a different sequence

(different sequence here)

then the results are quite different, as is shown in Figure 2.7 below.

Figure 2.7: the results from two different orderings of the same rotations

Mathematical notation can be applied in many ways, so your previous mathematical experience may not help you very much in deciding how you want to approach this problem. However, we want to define the sequence of transformations as *last-specified, first-applied*, or in another way of thinking about it, we want to apply our functions so that the function nearest to the geometry is applied first. Another way to think about this is in terms of building composite functions by multiplying the pieces, and in this case we want to compose each new function by multiplying it on the right of the previous functions. So the standard operation sequence we see above would be achieved by the following algebraic sequence of operations:

```
translate * rotate * scale * geometry
```

or, thinking of multiplication as function composition, as

```
translate(rotate(scale(geometry)))
```

This might be implemented by a sequence of function calls like that below that is not intended to represent any particular API:

```
translate(C1, C2, C3);  // translate to the desired point
rotate(A, Z);           // rotate by A around the Z-axis
scale(A, B, C);         // scale by the desired amounts
cube();                 // define the geometry of the cube
```

At first glance, this sequence looks to be exactly the opposite of the sequence noted above. In fact, however, we readily see that the scaling operation is the function closest to the geometry (which is expressed in the function `cube()`) because of the last-specified, first-applied nature of transformations. In Figure 2.8 we see the sequence of operations as we proceed from the plain cube (at the left), to the scaled cube next, then to the scaled and rotated cube, and finally to the cube that uses all the transformations (at the right). The application is to create a long, thin, rectangular bar that is oriented at a 45° angle upwards and lies above the definition plane.

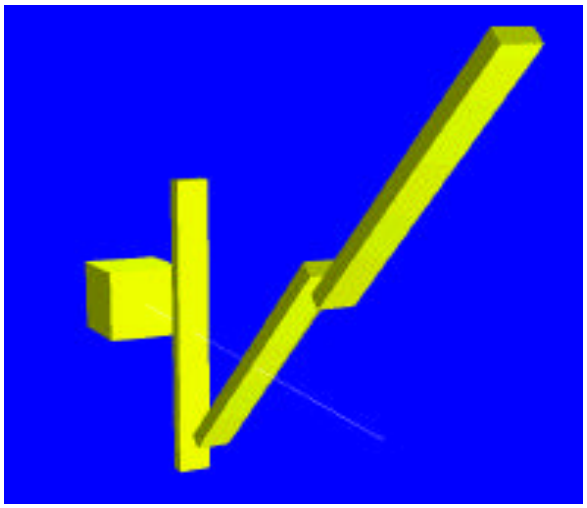


Figure 2.8: the sequence of figures as a cube is transformed

In general, the overall sequence of transformations that are applied to a model by considering the total sequence of transformations in the order in which they are specified, as well as the geometry on which they work:

```
P  V  T0 T1 T2 …  Tn Tn+1  …  Tlast  …  geometry
```

Here, `P` is the projection transformation, `V` is the viewing transformation, and `T0`, `T1`, … `Tlast` are the transformations specified in the program to model the scene, in order (`T1` is first, `Tlast` is last). The projection transformation is defined in the `reshape` function; the viewing transformation is defined in the `init` function or at the beginning of the `display` function so it is defined at the beginning of the modeling process. But the sequence is actually applied in reverse: `Tlast` is actually applied first, and `V` and finally `P` are applied last. The code would then have the definition of `P` first, the definition of `V` second, the definitions of `T0`, `T1`, … `Tlast` next in order, and the definition of the geometry last. You need to understand this sequence very well, because it's critical to understand how you build complex heirarchical models.

Transformation stacks and their manipulation

In defining a scene, we often want to define some standard pieces and then assemble them in standard ways, and then use the combined pieces to create additional parts, and go on to use these parts in additional ways. To do this, we need to create individual parts through functions that do not pay any attention to ways the parts will be used later, and then be able to assemble them into a whole. Eventually, we can see that the entire image will be a single whole that is composed of its various parts.

The key issue is that there is some kind of transformation in place when you start to define the object. When we begin to put the simple parts of a composite object in place, we will use some transformations but we need to undo the effect of those transformations when we put the next part in place. In effect, we need to save the state of the transformations when we begin to place a new part, and then to return to that transformation state (discarding any transformations we may have added past that mark) to begin to place the next part. Note that we are always adding and discarding at the end of the list; this tells us that this operation has the computational properties of a stack. We may define a stack of transformations and use it to manage this process as follows:

- as transformations are defined, they are multiplied into the current transformation in the order noted in the discussion of composite transformations above, and
- when we want to save the state of the transformation, we make a copy of the current version of the transformation and push that copy onto the stack, and apply all the subsequent transformations to the copy at the top of the stack. When we want to return to the original transformation state, we can pop the stack and throw away the copy that was removed, which gives us the original transformation so we can begin to work again at that point. Because all transformations are applied to the one at the top of the stack, when we pop the stack we return to the original context.

Designing a scene that has a large number of pieces of geometry as well as the transformations that define them can be difficult. In the next section we introduce the concept of the scene graph as a design tool to help you create complex and dynamic models both efficiently and effectively.

Compiling geometry

It can take a fair amount of time to calculate the various components of a piece of an image when that piece involves vertex lists and transformations. If an object is used frequently, and if it must be re-calculated each time it is drawn, it can make a scene quite slow to display. As a way to save time in displaying the image, many graphics APIs allow you to "compile" your geometry in a way that will allow it to be displayed much more quickly. Geometry that is to be compiled should be carefully chosen so that it is not changed between displays. If changes are needed, you will need to re-compile the object. Once you have seen what parts you can compile, you can compile them

and use the compiled versions to make the display faster. We will discuss how OpenGL compiles geometry later in this chapter. If you use another API, look for details in its documentation.

*A word to the wise...*

As we noted above, you must take a great deal of care with transformation order. It can be very difficult to look at an image that has been created with mis-ordered transformations and understand just how that erroneous example happened. In fact, there is a skill in what we might call "visual debugging" — looking at an image and seeing that it is not correct, and figuring out what errors might have caused the image as it is seen. It is important that anyone working with images become skilled in this kind of debugging. However, obviously you cannot tell than an image is wrong unless you know what a correct image should be, so you must know in general what you should be seeing. As an obvious example, if you are doing scientific images, you must know the science well enough to know when an image makes no sense.

# Scene Graphs and Modeling Graphs

*Introduction*

In this chapter, we define modeling as the process of defining and organizing a set of geometry that represents a particular scene. While modern graphics APIs can provide you with a great deal of assistance in rendering your images, modeling is usually supported less well and causes programmers considerable difficulty when they begin to work in computer graphics. Organizing a scene with transformations, particularly when that scene involves hierarchies of components and when some of those components are moving, involves relatively complex concepts that need to be organized very systematically to create a successful scene. Hierarchical modeling has long been done by using trees or tree-like structures to organize the components of the model.

Recent graphics systems, such as Java3D and VRML 2, have formalized the concept of a scene graph as a powerful tool for both modeling scenes and organizing the rendering process for those scenes. By understanding and adapting the structure of the scene graph, we can organize a careful and formal tree approach to both the design and the implementation of hierarchical models. This can give us tools to manage not only modeling the geometry of such models, but also animation and interactive control of these models and their components.

In this section of the chapter we will describe the scene graph structure and will adapt it to a modeling graph that you can use to design scenes, and we will identify how this modeling graph gives us the three key transformations that go into creating a scene: the projection transformation, the viewing transformation, and the modeling transformation(s) for the scene's content. This structure is very general and lets us manage all the fundamental principles in defining a scene and translating it into a graphics API. Our terminology is based on with the scene graph of Java3D and should help anyone who uses that system understand the way scene graphs work there.

*A brief summary of scene graphs*

The fully-developed scene graph of the Java3D API has many different aspects and can be complex to understand fully, but we can abstract it somewhat to get an excellent model to help us think about scenes that we can use in developing the code to implement our modeling. A brief outline of the Java3D scene graph in Figure 2.9 will give us a basis to discuss the general approach to graph-structured modeling as it can be applied to beginning computer graphics.

A *virtual universe* holds one or more (usually one) locales, which are essentially positions in the universe to put scene graphs. Each scene graph has two kinds of branches: content branches, which are to contain shapes, lights, and other content, and view branches, which are to contain viewing information. This division is somewhat flexible, but we will use this standard approach to build a framework to support our modeling work.

The *content branch* of the scene graph is organized as a collection of nodes that contains group nodes, transform groups, and shape nodes. A *group node* is a grouping structure that can have any number of children; besides simply organizing its children, a group can include a switch that selects which children to present in a scene. A *transform group* is a collection of modeling transformations that affect all the geometry that lies below it. The transformations will be applied to any of the transform group's children with the convention that transforms "closer" to the geometry (geometry that is defined in shape nodes lower in the graph) are applied first. A *shape node* includes both geometry and appearance data for an individual graphic unit. The geometry data includes standard 3D coordinates, normals, and texture coordinates, and can include points, lines, triangles, and quadrilaterals, as well as triangle strips, triangle fans, and quadrilateral strips. The appearance data includes color, shading, or texture information. Lights and eye points are

included in the content branch as a particular kind of geometry, having position, direction, and other appropriate parameters. Scene graphs also include shared groups, or groups that are included in more than one branch of the graph, which are groups of shapes that are included indirectly in the graph, and any change to a shared group affects all references to that group. This allows scene graphs to include the kind of template-based modeling that is common in graphics applications.

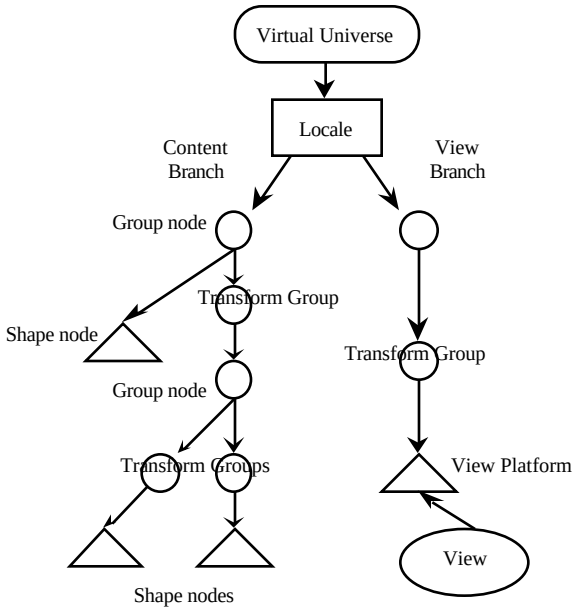



Figure 2.9: the structure of the scene graph as defined in Java3D

The *view branch* of the scene graph includes the specification of the display device, and thus the projection appropriate for that device. It also specifies the user's position and orientation in the scene and includes a wide range of abstractions of the different kinds of viewing devices that can be used by the viewer. It is intended to permit viewing the same scene on any kind of display device, including sophisticated virtual reality devices. This is a much more sophisticated approach than we need for our relatively simple modeling. We will simply consider the eye point as part of the geometry of the scene, so we set the view by including the eye point in the content branch and get the transformation information for the eye point in order to create the view transformations in the view branch.

In addition to the modeling aspect of the scene graph, Java3D also uses it to organize the processing as the scene is rendered. Because the scene graph is processed from the bottom up, the content branch is processed first, followed by the viewing transformation and then the projection transformation. However, the system does not guarantee any particular sequence in processing the node's branches, so it can optimize processing by selecting a processing order for efficiency, or can distribute the computations over a networked or multiprocessor system. Thus the Java3D programmer must be careful to make no assumptions about the state of the system when any shape node is processed. We will not ask the system to process the scene graph itself, however, because we will only use the scene graph to develop our modeling code.

An example of modeling with a scene graph

We will develop a scene graph to design the modeling for an example scene to show how this process can work. To begin, we present an already-completed scene so we can analyze how it can be created, and we will take that analysis and show how the scene graph can give us other ways to

present the scene. Consider the scene as shown in Figure 2.10, where a helicopter is flying above a landscape and the scene is viewed from a fixed eye point. (The helicopter is the small green object toward the top of the scene, about 3/4 of the way across the scene toward the right.)

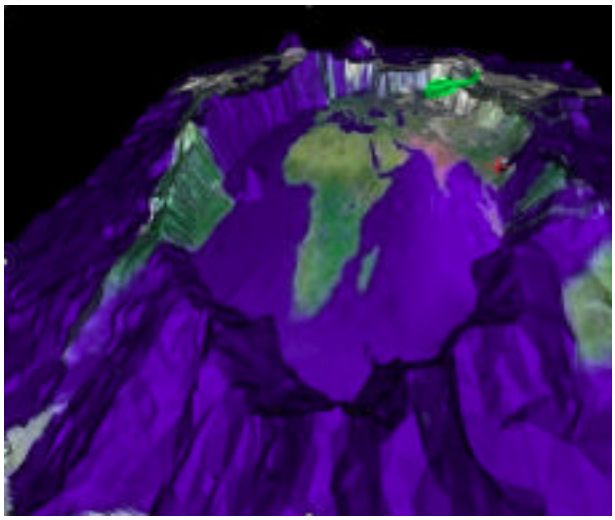


Figure 2.10: a scene that we will describe with a scene graph

This scene contains two principal objects: a helicopter and a ground plane. The helicopter is made up of a body and two rotors, and the ground plane is modeled as a single set of geometry with a texture map. There is some hierarchy to the scene because the helicopter is made up of smaller components, and the scene graph can help us identify this hierarchy so we can work with it in rendering the scene. In addition, the scene contains a light and an eye point, both at fixed locations. The first task in modeling such a scene is now complete: to identify all the parts of the scene, organize the parts into a hierarchical set of objects, and put this set of objects into a viewing context. We must next identify the relationship among the parts of the landscape way so we may create the tree that represents the scene. Here we note the relationship among the ground and the parts of the helicopter. Finally, we must put this information into a graph form.

The initial analysis of the scene in Figure 2.10, organized along the lines of view and content branches, leads to an initial (and partial) graph structure shown in Figure 2.11. The content branch of this graph captures the organization of the components for the modeling process. This describes how content is assembled to form the image, and the hierarchical structure of this branch helps us organize our modeling components. The view branch of this graph corresponds roughly to projection and viewing. It specifies the projection to be used and develops the projection transformation, as well as the eye position and orientation to develop the viewing transformation.

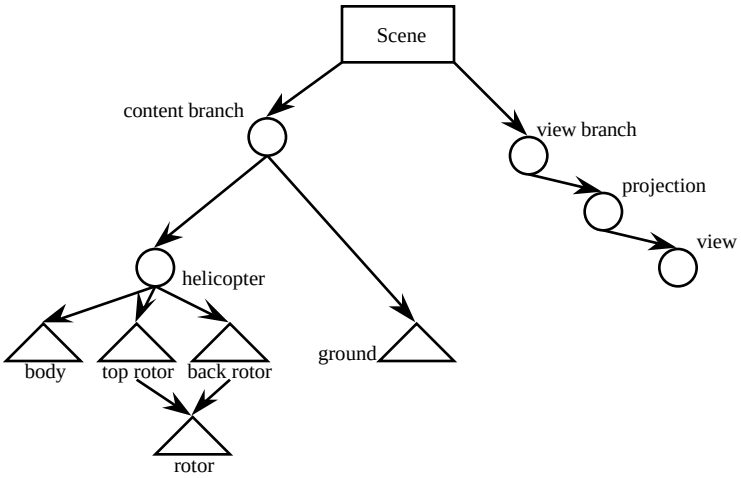



Figure 2.11: a scene graph that organizes the modeling of our simple scene

This initial structure is compatible with the simple OpenGL viewing approach we discussed in the previous chapter and the modeling approach earlier in this chapter, where the view is implemented by using built-in function that sets the viewpoint, and the modeling is built from relatively simple primitives. This approach only takes us so far, however, because it does not integrate the eye into the scene graph. It can be difficult to compute the parameters of the viewing function if the eye point is embedded in the scene and moves with the other content, and later we will address that part of the question of rendering the scene.

While we may have started to define our scene graph, we are not nearly finished. The initial scene graph of Figure 2.11 is incomplete because it merely includes the parts of the scene and describes which parts are associated with what other parts. To expand this first approximation to a more complete graph, we must add several things to the graph:

- the transformation information that describes the relationship among items in a group node, to be applied separately on each branch as indicated,
- the appearance information for each shape node, indicated by the shaded portion of those nodes,
- the light and eye position, either absolute (as used in Figure 2.10 and shown Figure 2.12) or relative to other components of the model, and
- the specification of the projection and view in the view branch.

These are all included in the expanded version of the scene graph with transformations, appearance, eyepoint, and light shown in Figure 2.121.

The content branch of this graph handles all the scene modeling and is very much like the content branch of the scene graph. It includes all the geometry nodes of the graph in Figure 2.11 as well as appearance information; includes explicit transformation nodes to place the geometry into correct sizes, positions, and orientations; includes group nodes to assemble content into logical groupings; and includes lights and the eye point, shown here in fixed positions without excluding the possibility that a light or the eye might be attached to a group instead of being positioned independently. In the example above, it identifies the geometry of the shape nodes such as the rotors or individual trees as shared. This might be implemented, for example, by defining the geometry of the shared shape node in a function and calling that from each of the rotor or tree nodes that uses it.

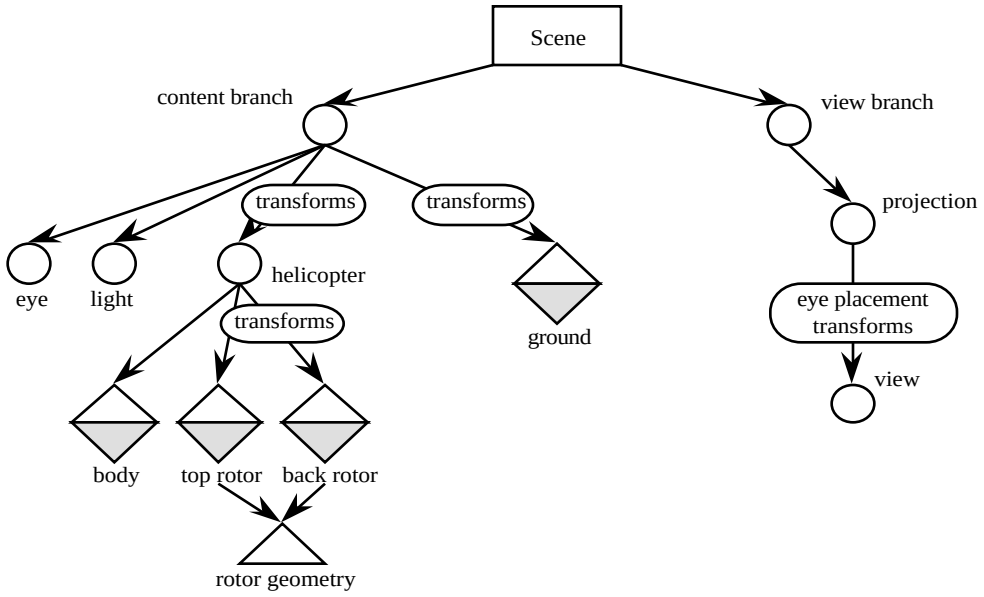



Figure 2.12: the more complete graph including transformations and appearance

The view branch of this graph is similar to the view branch of the scene graph but is treated much more simply, containing only projection and view components. The projection component includes the definition of the projection (orthogonal or perspective) for the scene and the definition of the window and viewport for the viewing. The view component includes the information needed to create the viewing transformation, and because the eye point is placed in the content branch, this is simply a copy of the set of transformations that position the eye point in the scene as represented in the content branch.

The appearance part of the shape node is built from color, lighting, shading, texture mapping, and several other kinds of operations. Eventually each vertex of the geometry will have not only geometry, in terms of its coordinates, but also normal components, texture coordinates, and several other properties. Here, however, we are primarily concerned with the geometry content of the shape node; much of the rest of these notes is devoted to building the appearance properties of the shape node, because the appearance content is perhaps the most important part of graphics for building high-quality images.

The scene graph for a particular image is not unique because there are many ways to organize a scene. When you have a well-defined set of transformation that place the eye point in a scene, we saw in the earlier chapter on viewing how you can take advantage of that information to organize the scene graph in a way that can define the viewing transformation explicitly and simply use the default view for the scene. As we noted there, the real effect of the viewing transformation is to be the inverse of the transformation that placed the eye. So we can explicitly compute the viewing transformation as the inverse of the placement transformation ourselves and place that at the top of the scene graph. Thus we can restructure the scene graph of Figure 2.12 as shown below in Figure 2.13 so it may take any arbitrary eye position. This will be the key point below as we discuss how to manage the eyepoint when it is a dynamic part of a scene.

It is very important to note that the scene graph need not describe a static geometry. Callbacks for user interaction and other events can affect the graph by controlling parameters of its components, as noted in the re-write guidelines in the next section. This can permit a single graph to describe an animated scene or even alternate views of the scene. The graph may thus be seen as having some components with external controllers, and the controllers are the event callback functions.

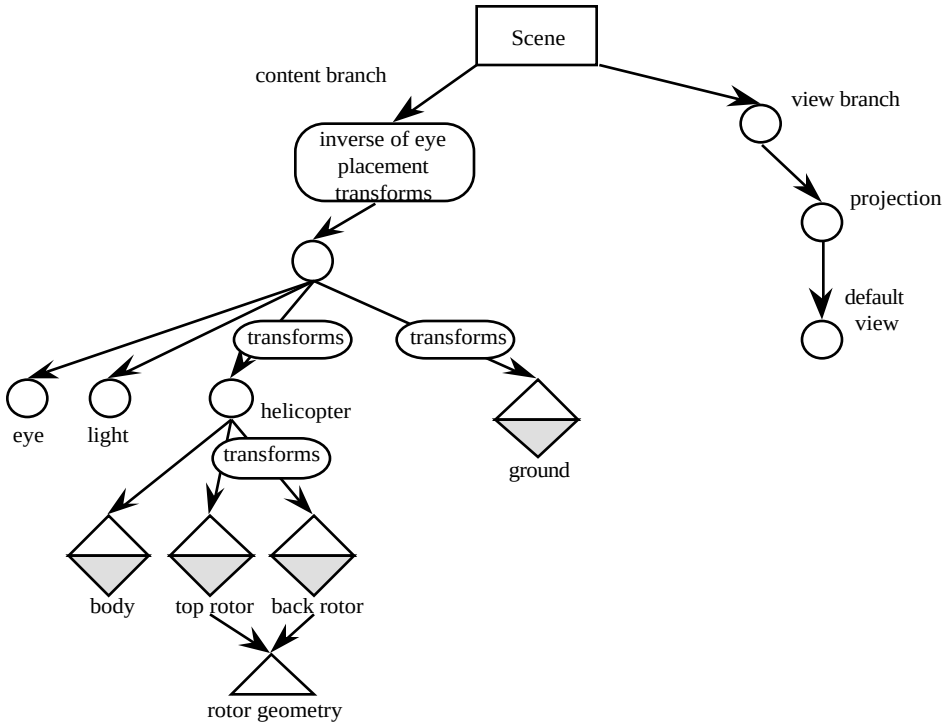



Figure 2.13: the scene graph after integrating the viewing transformation into the content branch

We need to extract the three key transformations from this graph in order to create the code that implements our modeling work. The projection transformation is straightforward and is built from the projection information in the view branch, and this is easily managed from tools in the graphics API. The viewing transformation is readily created from the transformation information in the view by analyzing the eye placement transformations as we saw above, and the modeling transformations for the various components are built by working with the various transformations in the content branch as the components are drawn. These operations are all straightforward; we begin with the viewing transformation and move on to coding the modeling transformations.

*The viewing transformation*

In a scene graph with no view specified, we assume that the default view puts the eye at the origin looking in the negative z-direction with the y-axis upward. If we use a set of transformations to position the eye differently, then the viewing transformation is built by inverting those transformations to restore the eye to the default position. This inversion takes the sequence of transformations that positioned the eye and inverts the primitive transformations in reverse order, so if $T_1T_2T_3\ldots T_K$ is the original transformation sequence, the inverse is $T_K^u\ldots T_3^uT_2^uT_1^u$ where the superscript u indicates inversion, or "undo" as we might think of it.

Each of the primitive scaling, rotation, and translation transformations is easily inverted. For the scaling transformation `scale(Sx, Sy, Sz)`, we note that the three scale factors are used to multiply the values of the three coordinates when this is applied. So to invert this transformation, we must divide the values of the coordinates by the same scale factors, getting the inverse as `scale(1/Sx, 1/Sy, 1/Sz)`. Of course, this tells us quickly that the scaling function can only be inverted if none of the scaling factors are zero.

For the rotation transformation `rotate(angle, line)` that rotates space by the value `angle` around the fixed line `line`, we must simply rotate the space by the same angle in the reverse direction. Thus the inverse of the rotation transformation is `rotate(-angle, line)`.

For the translation transformation `translate(Tx, Ty, Tz)` that adds the three translation values to the three coordinates of any point, we must simply subtract those same three translation values when we invert the transformation. Thus the inverse of the translation transformation is `translate(-Tx, -Ty, -Tz)`.

Putting this together with the information on the order of operations for the inverse of a composite transformation above, we can see that, for example, the inverse of the set of operations (written as if they were in your code)

```
translate(Tx, Ty, Tz)
rotate(angle, line)
scale(Sx, Sy, Sz)
```

is the set of operations

```
scale(1/Sx, 1/Sy, 1/Sz)
rotate(-angle, line)
translate(-Tx, -Ty, -Tz)
```

Now let us apply this process to the viewing transformation. Deriving the eye transformations from the tree is straightforward. Because we suggest that the eye be considered one of the content components of the scene, we can place the eye at any position relative to other components of the scene. When we do so, we can follow the path from the root of the content branch to the eye to obtain the sequence of transformations that lead to the eye point. That sequence of transformations is the eye transformation that we may record in the view branch.

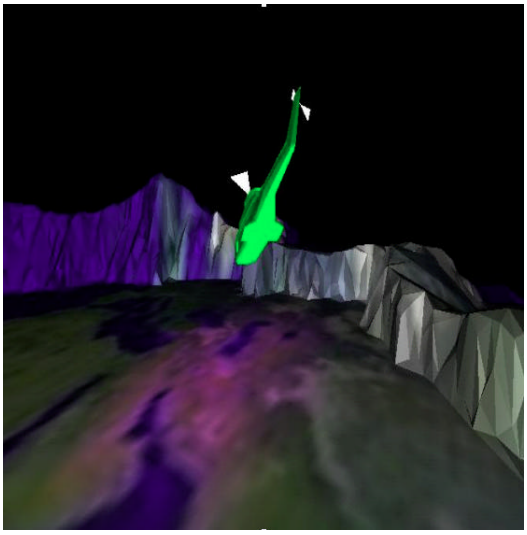


Figure 2.14: the same scene as in Figure 2.10 but with the eye point following directly behind the helicopter

In Figure 2.14 we show the change that results in the view of Figure 2.10 when we define the eye to be immediately behind the helicopter, and in Figure 2.15 we show the change in the scene graph of Figure 2.12 that implements the changed eye point. The eye transform consists of the transforms that places the helicopter in the scene, followed by the transforms that place the eye relative to the helicopter. Then as we noted earlier, the viewing transformation is the inverse of the

eye positioning transformation, which in this case is the inverse of the transformations that placed the eye relative to the helicopter, followed by the inverse of the transformations that placed the helicopter in the scene.

This change in the position of the eye means that the set of transformations that lead to the eye point in the view branch must be changed, but the mechanism of writing the inverse of these transformations before beginning to write the definition of the scene graph still applies; only the actual transformations to be inverted will change. This is how the scene graph will help you to organize the viewing process that was described in the earlier chapter on viewing.

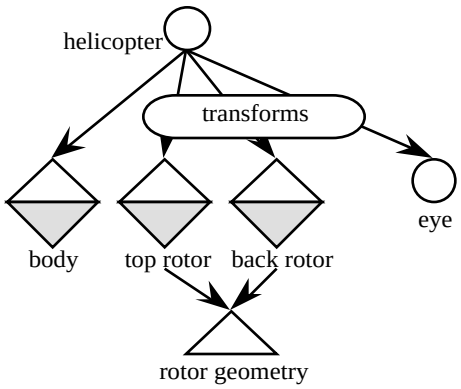



Figure 2.15: the change in the scene graph of Figure 2.10
to implement the view in Figure 2.14

The process of placing the eye point can readily be generalized. For example, if you should want to design a scene with several possible eye points and allow a user to choose among them, you can design the view branch by creating one view for each eye point and using the set of transformations leading to each eye point as the transformation for the corresponding view. You can then invert each of these sets of transformations to create the viewing transformation for each of the eye points. The choice of eye point will then create a choice of view, and the viewing transformation for that view can then be chosen to implement the user choice.

Because the viewing transformation is performed before the modeling transformations, we see from Figure 2.13 that the inverse transformations for the eye must be applied before the content branch is analyzed and its operations are placed in the code. This means that the display operation must begin with the inverse of the eye placement transformations, which has the effect of moving the eye to the top of the content branch and placing the inverse of the eye path at the front of each set of transformations for each shape node.

*Using the modeling graph for coding*

Let us use the name "modeling graph" for the analogue of the scene graph we illustrated in the previous section. Because the modeling graph is intended as a learning tool, we will resist the temptation to formalize its definition beyond the terms we used there:

- shape node containing two components
  - geometry content
  - appearance content
- transformation node
- group node
- projection node
- view node

Because we do not want to look at any kind of automatic parsing of the modeling graph to create the scene, we will merely use the graph to help organize the structure and the relationships in the model to help you organize your code to implement your simple or hierarchical modeling. This is quite straightforward and is described in detail below.

Once you know how to organize all the components of the model in the modeling graph, you next need to write the code to implement the model. This turns out to be straightforward, and you can use a simple set of re-write guidelines that allow you to re-write the graph as code. In this set of rules, we assume that transformations are applied in the reverse of the order they are declared, as they are in OpenGL, for example. This is consistent with your experience with tree handling in your programming courses, because you have usually discussed an expression tree which is parsed in leaf-first order. It is also consistent with the Java3D convention that transforms that are " closer" to the geometry (nested more deeply in the scene graph) are applied first.

The informal re-write guidelines are as follows, including the re-writes for the view branch as well as the content branch:

- Nodes in the view branch involve only the window, viewport, projection, and viewing transformations. The window, viewport, and projection are handled by simple functions in the API and should be at the top of the display function.
- The viewing transformation is built from the transformations of the eye point within the content branch by copying those transformations and undoing them to place the eye effectively at the top of the content branch. This sequence should be next in the display function.
- The content branch of the modeling graph is usually maintained fully within the display function, but parts of it may be handled by other functions called from within the display, depending on the design of the scene. A function that defines the geometry of an object may be used by one or more shape nodes. The modeling may be affected by parameters set by event callbacks, including selections of the eye point, lights, or objects to be displayed in the view.
- Group nodes are points where several elements are assembled into a single object. Each separate object is a different branch from the group node. Before writing the code for a branch that includes a transformation group, the student should push the modelview matrix; when returning from the branch, the student should pop the modelview matrix.
- Transformation nodes include the familiar translations, rotations, and scaling that are used in the normal ways, including any transformations that are part of animation or user control. In writing code from the modeling graph, students can write the transformations in the same sequence as they appear in the tree, because the bottom-up nature of the design work corresponds to the last-defined, first-used order of transformations.
- As you work your way through the modeling graph, you will need to save the state of the modeling transformation before you go down any branch of the graph from which you will need to return as the graph is traversed. Because of the simple nature of each transformation primitive, it is straightforward to undo each as needed to create the viewing transformation. This can be handled through a transformation stack that allows you to save the current transformation by pushing it onto the stack, and then restore that transformation by popping the stack.
- Shape nodes involve both geometry and appearance, and the appearance must be done first because the current appearance is applied when geometry is defined.
    - An appearance node can contain texture, color, blending, or material information that will make control how the geometry is rendered and thus how it will appear in the scene.
    - A geometry node will contain vertex information, normal information, and geometry structure information such as strip or fan organization.
- Most of the nodes in the content branch can be affected by any interaction or other event-driven activity. This can be done by defining the content by parameters that are modified

by the event callbacks. These parameters can control location (by parametrizing rotations or translations), size (by parametrizing scaling), appearance (by parametrizing appearance details), or even content (by parametrizing switches in the group nodes).

We will give some examples of writing graphics code from a modeling graph in the sections below, so look for these principles as they are applied there.

In the example for Figure 2.14 above, we would use the tree to write code as shown in skeleton form in Figure 2.16. Most of the details, such as the inversion of the eye placement transformation, the parameters for the modeling transformations, and the details of the appearance of individual objects, have been omitted, but we have used indentation to show the pushing and popping of the modeling transformation stack so we can see the operations between these pairs easily. This is straightforward to understand and to organize.

```
display()
   set the viewport and projection as needed
   initialize modelview matrix to identity
   define viewing transformation
      invert the transformations that set the eye location
   set eye through gluLookAt with default values
   define light position         // note absolute location
   push the transformation stack       // ground
      translate
      rotate
      scale
      define ground appearance (texture)
      draw ground
   pop the transformation stack
   push the transformation stack       // helicopter
      translate
      rotate
      scale
      push the transformation stack    // top rotor
         translate
         rotate
         scale
         define top rotor appearance
         draw top rotor
      pop the transformation stack
      push the transformation stack    // back rotor
         translate
         rotate
         scale
         define back rotor appearance
         draw back rotor
      pop the transformation stack
      // assume no transformation for the body
      define body appearance
      draw body
   pop the transformation stack
   swap buffers
```

Figure 2.16: code sketch to implement the modeling in Figure 2.15

Animation is simple to add to this example. The rotors can be animated by adding an extra rotation in their definition plane immediately after they are scaled and before the transformations that orient them to be placed on the helicopter body, and by updating angle of the extra rotation each time the

idle event callback executes. The helicopter's behavior itself can be animated by updating the parameters of transformations that are used to position it, again with the updates coming from the idle callback. The helicopter's behavior may be controlled by the user if the positioning transformation parameters are updated by callbacks of user interaction events. So there are ample opportunities to have this graph represent a dynamic environment and to include the dynamics in creating the model from the beginning.

Other variations in this scene could by developed by changing the position of the light from its current absolute position to a position relative to the ground (by placing the light as a part of the branch group containing the ground) or to a position relative to the helicopter (by placing the light as a part of the branch group containing the helicopter). The eye point could similarly be placed relative to another part of the scene, or either or both could be placed with transformations that are controlled by user interaction with the interaction event callbacks setting the transformation parameters.

We emphasize that you should include appearance content with each shape node. Many of the appearance parameters involve a saved state in APIs such as OpenGL and so parameters set for one shape will be retained unless they are re-set for the new shape. It is possible to design your scene so that shared appearances will be generated consecutively in order to increase the efficiency of rendering the scene, but this is a specialized organization that is inconsistent with more advanced APIs such as Java3D. Thus it is very important to re-set the appearance with each shape to avoid accidentally retaining an appearance that you do not want for objects presented in in later parts of your scene.

Example

We want to further emphasize the transformation behavior in writing the code for a model from the modeling graph by considering another small example. Let us consider a very simple rabbit's head as shown in Figure 2.17. This would have a large ellipsoidal head, two small spherical eyes, and two middle-sized ellipsoidal ears. So we will use the ellipsoid (actually a scaled sphere, as we saw earlier) as our basic part and will put it in various places with various orientations as needed.

The modeling graph for the rabbit's head is shown in Figure 2.18. This figure includes all the transformations needed to assemble the various parts (eyes, ears, main part) into a unit. The fundamental geometry for all these parts is the sphere, as we suggested above. Note that the transformations for the left and right ears include rotations; these can easily be designed to use a parameter for the angle of the rotation so that you could make the rabbit's ears wiggle back and forth.

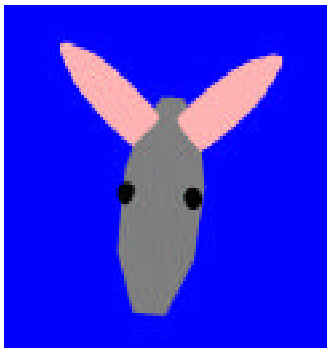


Figure 2.17: the rabbit's head

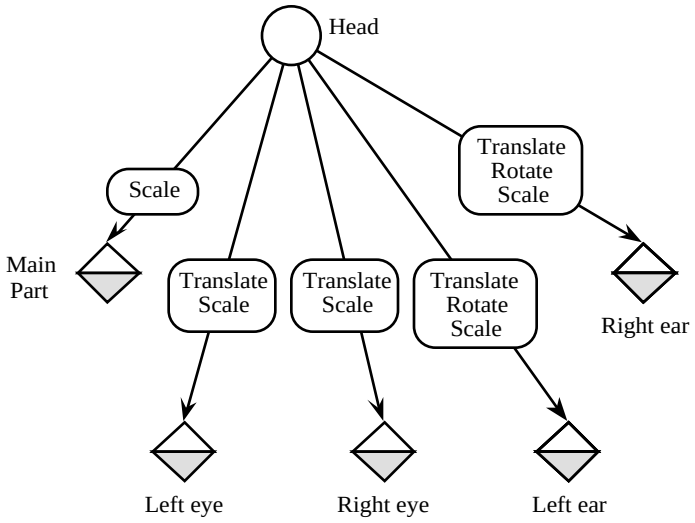



Figure 2.18: the modeling graph for the rabbit's head

To write the code to implement the modeling graph for the rabbit's head, then, we would apply the following sequence of actions on the modeling transformation stack:

- push the modeling transformation stack
- apply the transformations to create the head, and define the head:
  scale
  draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the left eye relative to the head, and define the eye:
  translate
  scale
  draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the right eye relative to the head, and define the eye:
  translate
  scale
  draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the left ear relative to the head, and define the ear:
  translate
  rotate
  scale
  draw sphere
- pop the modeling transformation stack
- push the modeling transformation stack
- apply the transformations that position the right ear relative to the head, and define the ear:
  translate
  rotate
  scale
  draw sphere
- pop the modeling transformation stack

You should trace this sequence of operations carefully and watch how the head is drawn. Note that if you were to want to put the rabbit's head on a body, you could treat this whole set of

operations as a single function `rabbitHead()` that is called between operations push and pop the transformation stack, with the code to place the head and move it around lying above the function call. This is the fundamental principle of hierarchical modeling — to create objects that are built of other objects, finally reducing the model to simple geometry at the lowest level. In the case of the modeling graph, that lowest level is the leaves of the tree, in the shape nodes.

The transformation stack we have used informally above is a very important consideration in using a scene graph structure. It may be provided by your graphics API or it may be something you need to create yourself; even if it provided by the API, there may be limits on the depth of the stack that will be inadequate for some projects and you may need to create your own. We will discuss this in terms of the OpenGL API later in this chapter.

Using standard objects to create more complex scenes

The example of transformation stacks is, in fact, a larger example — an example of using standard objects to define a larger object. In a program that defined a scene that needed rabbits, we would create the rabbit head with a function `rabbitHead()` that has the content of the code we used (and that is given below) and would apply whatever transformations would be needed to place a rabbit head properly on each rabbit body. The rabbits themselves could be part of a larger scene, and you could proceed in this way to create however complex a scene as you wish.

# Implementing Modeling in OpenGL

This chapter discusses the way OpenGL implements the general modeling discussion of the last chapter. It includes specifying geometry, specifying points for that geometry in model space, specifying normals for these vertices, and specifying and managing transformations that move these objects from model space into the world coordinate system. It also discusses some pre-built models that are provided in the OpenGL and GLUT environments to help you create your scenes more easily.

*The OpenGL model for specifying geometry*

In defining your model for your program, you will use a single function to specify the geometry of your model to OpenGL. This function specifies that geometry is to follow, and its parameter defines the way in which that geometry is to be interpreted for display:

```
glBegin(mode);
// vertex list: point data to create a primitive object in
// the drawing mode you have indicated
// normals may also be specified here
glEnd();
```

The vertex list is interpreted as needed for each drawing mode, and both the drawing modes and the interpretation of the vertex list are described in the discussions below. This pattern of `glBegin(mode) - vertex list - glEnd` uses different values of the `mode` to establish the way the vertex list is used in creating the image. Because you may use a number of different kinds of components in an image, you may use this pattern several times for different kinds of drawing. We will see a number of examples of this pattern in this module.

In OpenGL, point (or vertex) information is presented to the computer through a set of functions that go under the general name of `glVertex*(…)`. These functions enter the numeric value of the vertex coordinates into the OpenGL pipeline for the processing to convert them into image information. We say that `glVertex*(…)` is a *set* of functions because there are many functions that differ only in the way they define their vertex coordinate data. You may want or need to specify your coordinate data in any standard numeric type, and these functions allow the system to respond to your needs.

- If you want to specify your vertex data as three separate real numbers, or floats (we'll use the variable names `x`, `y`, and `z`, though they could also be float constants), you can use `glVertex3f(x,y,z)`. Here the character `f` in the name indicates that the arguments are floating-point; we will see below that other kinds of data formats may also be specified for vertices.
- If you want to define your coordinate data in an array, you could declare your data in a form such as `glFloat x[3]` and then use `glVertex3fv(x)` to specify the vertex. Adding the letter `v` to the function name specifies that the data is in vector form (actually a pointer to the memory that contains the data, but an array's name is really such a pointer). Other dimensions besides 3 are also possible, as noted below.

Additional versions of the functions allow you to specify the coordinates of your point in two dimensions (`glVertex2*`); in three dimensions specified as integers (`glVertex3i`), doubles (`glVertex3d`), or shorts (`glVertex3s`); or as four-dimensional points (`glVertex4*`). The four-dimensional version uses homogeneous coordinates, as described earlier in this chapter. You will see some of these used in the code examples later in this chapter.

One of the most important things to realize about modeling in OpenGL is that you can call your own functions between a `glBegin(mode)` and `glEnd()` pair to determine vertices for your vertex list. Any vertices these functions define by making a `glVertex*(…)` function call will be added to the vertex list for this drawing mode. This allows you to do whatever computation you

need to calculate vertex coordinates instead of creating them by hand, saving yourself significant effort and possibly allowing you to create images that you could not generate by hand. For example, you may include various kind of loops to calculate a sequence of vertices, or you may include logic to decide which vertices to generate. An example of this way to generate vertices is given among the first of the code examples toward the end of this module.

Another important point about modeling is that a great deal of other information can go between a `glBegin(mode)` and `glEnd()` pair. We will see the importance of including information about vertex normals in the chapters on lighting and shading, and of including information on texture coordinates in the chapter on texture mapping. So this simple construct can be used to do much more than just specify vertices. Although you may carry out whatever processing you need within the `glBegin(mode)` and `glEnd()` pair, there are a limited number of OpenGL operations that are permitted here. In general, the available OpenGL operations here are `glVertex`, `glColor`, `glNormal`, `glTexCoord`, `glEvalCoord`, `glEvalPoint`, `glMaterial`, `glCallList`, and `glCallLists`, although this is not a complete list. Your OpenGL manual will give you additional information if needed.

Point and points mode

The mode for drawing points with the `glBegin` function is named `GL_POINTS`, and any vertex data between `glBegin` and `glEnd` is interpreted as the coordinates of a point we wish to draw. If we want to draw only one point, we provide only one vertex between `glBegin` and `glEnd`; if we want to draw more points, we provide more vertices between them. If you use points and want to make each point more visible, the function `glPointSize(float size)` allows you to set the size of each point, where `size` is any nonnegative real value and the default size is 1.0.

The code below draws a sequence of points in a straight line. This code takes advantage of fact that we can use ordinary programming processes to define our models, showing we need not hand-calculate points when we can determine them by an algorithmic approach. We specify the vertices of a point through a function `pointAt()` that calculates the coordinates and calls the `glVertex*()` function itself, and then we call that function within the `glBegin/glEnd` pair. The function calculates points on a spiral along the z-axis with x- and y-coordinates determined by functions of the parameter `t` that drives the entire spiral.

```
void pointAt(int i) {
  glVertex3f(fx(t)*cos(g(t)),fy(t)*sin(g(t)),0.2*(float)(5-i));
}

void pointSet( void ) {
  int i;

  glBegin(GL_POINTS);
    for ( i=0; i<10; i++ )
      pointAt(i);
  glEnd();
}
```

Some functions that drive the x- and y-coordinates may be familiar to you through studies of functions of polar coordinates in previous mathematics classes, and you are encouraged to try out some possibilities on your own.

Line segments

To draw line segments, we use the `GL_LINES` mode for `glBegin/glEnd`. For each segment we wish to draw, we define the vertices for the two endpoints of the segment. Thus between `glBegin` and `glEnd` each pair of vertices in the vertex list defines a separate line segment.

Line strips

Connected lines are called *line strips* in OpenGL, and you can specify them by using the mode `GL_LINE_STRIP` for `glBegin/glEnd`. The vertex list defines the line segments as noted in the general discussion of connected lines above, so if you have N vertices, you will have N-1 line segments. With either line segments or connected lines, we can set the line width to emphasize (or de-emphasize) a line. Heavier line widths tend to attract more attention and give more emphasis than lighter line widths. The line width is set with the `glLineWidth(float width)` function. The default value of `width` is 1.0 but any nonnegative width can be used.

As an example of a line strip, let's consider a parametric curve. Such curves in 3-space are often interesting objects of study. The code below define a rough spiral in 3-space that is a good (though simple) example of using a single parameter to define points on a parametric curve so it can be drawn for study.

```
glBegin(GL_LINE_STRIP);
  for ( i=0; i<=10; i++ )
    glVertex3f(2.0*cos(3.14159*(float)i/5.0),
        2.0*sin(3.14159*(float)i/5.0),0.5*(float)(i-5));
glEnd();
```

This can be made much more sophisticated by increasing the number of line segments, and the code can be cleaned up a bit as described in the code fragment below. Simple experiments with the `step` and `zstep` variables will let you create other versions of the spiral as experiments.

```
#define PI 3.14159
#define N 100
step = 2.0*PI/(float)N;
zstep = 2.0/(float)N;
glBegin(GL_LINE_STRIP);
  for ( i=0; i<=N; i++)
    glVertex3f(2.0*sin(step*(float)i),2.0*cos(step*(float)i),
        -1.0+zstep*(float)i);
glEnd();
```

If this spiral is presented in a program that includes some simple rotations, you can see the spiral from many points in 3-space. Among the things you will be able to see are the simple sine and cosine curves, as well as one period of the generic shifted sine curve.

Triangle

To draw unconnected triangles, you use `glBegin/glEnd` with the mode `GL_TRIANGLES`. This is treated exactly as discussed in the previous chapter and produces a collection of triangles, one for each three vertices specified.

Sequence of triangles

OpenGL provides both of the standard geometry-compression techniques to assemble sequences of triangles: triangle strips and triangle fans. Each has its own mode for `glBegin/glEnd`:

GL_TRIANGLE_STRIP and GL_TRIANGLE_FAN respectively. These behave exactly as described in the general section above.

Because there are two different modes for drawing sequences of triangles, we'll consider two examples in this section. The first is a triangle fan, used to define an object whose vertices can be seen as radiating from a central point. An example of this might be the top and bottom of a sphere, where a triangle fan can be created whose first point is the north or south pole of the sphere. The second is a triangle strip, which is often used to define very general kinds of surfaces, because most surfaces seem to have the kind of curvature that keeps rectangles of points on the surface from being planar. In this case, triangle strips are much better than quad strips as a basis for creating curved surfaces that will show their surface properties when lighted.

The triangle fan (that defines a cone, in this case) is organized with its vertex at point (0.0,1.0,0.0) and with a circular base of radius 0.5 in the XZ-plane. Thus the cone is oriented towards the y-direction and is centered on the y-axis. This provides a surface with unit diameter and height, as shown in Figure 3.1. When the cone is used in creating a scene, it can easily be defined to have whatever size, orientation, and location you need by applying appropriate modeling transformations in an appropriate sequence. Here we have also added normals and flat shading to emphasize the geometry of the triangle fan, although the code does not reflect this.

```
glBegin(GL_TRIANGLE_FAN);
  glVertex3f(0., 1.0, 0.);      // the point of the cone
  for (i=0; i < numStrips; i++) {
    angle = 2. * (float)i * PI / (float)numStrips;
    glVertex3f(0.5*cos(angle), 0.0, 0.5*sin(angle));
    // code to calculate normals would go here
  }
glEnd();
```

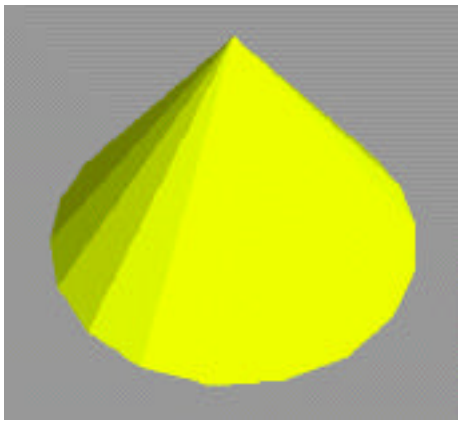


Figure 3.1: the cone produced by the triangle fan

The triangle strip example is based on an example of a function surface defined on a grid. Here we describe a function whose domain is in the X-Z plane and whose values are shown as the Y-value of each vertex. The grid points in the X-Z domain are given by functions XX(i) and ZZ(j), and the values of the function are held in an array, with vertices[i][j] giving the value of the function at the grid point (XX(i),ZZ(j)) as defined in the short example code fragment below.

```
for ( i=0; i<XSIZE; i++ )
   for ( j=0; j<ZSIZE; j++ )
   {
      x = XX(i);
      z = ZZ(j);
      vertices[i][j] = (x*x+2.0*z*z)/exp(x*x+2.0*z*z+t);
   }
```

The surface rendering can then be organized as a nested loop, where each iteration of the loop draws a triangle strip that presents one section of the surface. Each section is one unit in the X-direction that extends across the domain in the Z-direction. The code for such a strip is shown below, and the resulting surface is shown in Figure 3.2. Again, the code that calculates the normals is omitted; this example is discussed further and the normals are developed in the later chapter on shading. This kind of surface is explored in more detail in the chapters on scientific applications of graphics.

```
for ( i=0; i<XSIZE-1; i++ )
   for ( j=0; j<ZSIZE-1; j++ )
   {
      glBegin(GL_TRIANGLE_STRIP);
        glVertex3f(XX(i),vertices[i][j],ZZ(j));
        glVertex3f(XX(i+1),vertices[i+1][j],ZZ(j));
        glVertex3f(XX(i),vertices[i][j+1],ZZ(j+1));
        glVertex3f(XX(i+1),vertices[i+1][j+1],ZZ(j+1));
      glEnd();
   }
```

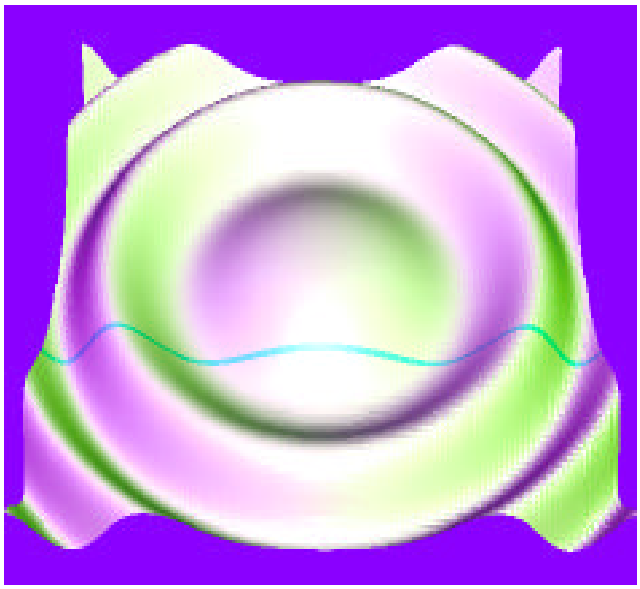


Figure 3.2: the full surface created by triangle strips, with a single strip highlighted in cyan

This example is a white surface lighted by three lights of different colors, a technique we describe in the chapter on lighting. This surface example is also briefly revisited in the quads discussion below. Note that the sequence of points here is slightly different here than it is in the example below because of the way quads are specified. In this example instead of one quad, we will have two triangles—and if you rework the example below to use quad strips instead of simple quads to display the mathematical surface, it is simple to make the change noted here and do the surface with extended triangle strips.

Quads

To create a set of one or more distinct quads you use `glBegin/glEnd` with the `GL_QUADS` mode. As described earlier, this will take four vertices for each quad. An example of an object based on quadrilaterals would be the function surface discussed in the triangle strip above. For quads, the code for the surface looks like this:

```
    for ( i=0; i<XSIZE-1; i++ )
       for ( j=0; j<ZSIZE-1; j++ )
       {
          // quad sequence: points (i,j),(i+1,j),(i+1,j+1),(i,j+1)
          glBegin(GL_QUADS);
             glVertex3f(XX(i),vertices[i][j],ZZ(j));
             glVertex3f(XX(i+1),vertices[i+1][j],ZZ(j));
             glVertex3f(XX(i+1),vertices[i+1][j+1],ZZ(j+1));
             glVertex3f(XX(i),vertices[i][j+1],ZZ(j+1));
          glEnd();
       }
}
```

Note that neither this surface nor the one composed from triangles is going to look very good yet because it does not yet contain any lighting or color information. These will be added in later chapters as this concept of function surfaces is re-visited when we discuss lighting and color.

Quad strips

To create a sequence of quads, the mode for `glBegin/glEnd` is `GL_QUAD_STRIP`. This operates in the way we described at the beginning of the chapter, and as we noted there, the order in which the vertices are presented is different from that in the GL_QUADS mode. Be careful of this when you define your geometry or you may get a very unusual kind of display!

In a fairly common application, we can create long, narrow tubes with square cross-section. This can be used as the basis for drawing 3-D coordinate axes or for any other application where you might want to have, say, a beam in a structure. The quad strip defined below creates the tube oriented along the Z-axis with the cross-section centered on that axis. The dimensions given make a unit tube—a tube that is one unit in each dimension, making it actually a cube. These dimensions will make it easy to scale to fit any particular use.

```
    #define RAD 0.5
    #define LEN 1.0
    glBegin(GL_QUAD_STRIP);
      glVertex3f( RAD, RAD, LEN ); // start of first side
      glVertex3f( RAD, RAD, 0.0 );
      glVertex3f(-RAD, RAD, LEN );
      glVertex3f(-RAD, RAD, 0.0 );
      glVertex3f(-RAD,-RAD, LEN ); // start of second side
      glVertex3f(-RAD,-RAD, 0.0 );
      glVertex3f( RAD,-RAD, LEN ); // start of third side
      glVertex3f( RAD,-RAD, 0.0 );
      glVertex3f( RAD, RAD, LEN ); // start of fourth side
      glVertex3f( RAD, RAD, 0.0 );
    glEnd();
```

You can also get the same object by using the GLUT cube that is discussed below and applying appropriate transformations to center it on the Z-axis.

General polygon

The GL_POLYGON mode for glBegin/glEnd is used to allow you to display a single convex polygon. The vertices in the vertex list are taken as the vertices of the polygon in sequence order, and we remind you that the polygon needs to be convex. It is not possible to display more than one polygon with this operation because the function will always assume that whatever points it receives go in the same polygon.

Probably the simplest kind of multi-sided convex polygon is the regular N-gon, an N-sided figure with all edges of equal length and all interior angles between edges of equal size. This is simply created, again using trigonometric functions to determine the vertices.

```
#define PI 3.14159
#define N 7
step = 2.0*PI/(float)N;
glBegin(GL_POLYGON);
  for ( i=0; i<=N; i++)
    glVertex3f(2.0*sin(step*(float)i),2.0*cos(step*(float)i),0.0);;
glEnd();
```

Note that this polygon lives in the XY-plane; all the Z-values are zero. This polygon is also in the default color (white) for simple models. This is an example of a "canonical" object — an object defined not primarily for its own sake, but as a template that can be used as the basis of building another object as noted later, when transformations and object color are available. An interesting application of regular polygons is to create regular polyhedra — closed solids whose faces are all regular N-gons. These polyhedra are created by writing a function to draw a simple N-gon and then using transformations to place these properly in 3-space to be the boundaries of the polyhedron.

The cube we will use in many examples

Because a cube is made up of six square faces, it is very tempting to try to make the cube from a single quad strip. Looking at the geometry, though, it is impossible to make a single quad strip go around the cube; in fact, the largest quad strip you can create from a cube's faces has only four quads. It is possible to create two quad strips of three faces each for the cube (think of how a baseball is stitched together), but here we will only use a set of six quads whose vertices are the eight vertex points of the cube. Below we repeat the declarations of the vertices, normals, edges, and faces of the cube from the previous chapter. We will use the glVertex3fv(…) vertex specification function within the specification of the quads for the faces.

```
typedef float point3[3];
typedef int   edge[2];
typedef int   face[4];    // each face of a cube has four edges

     point3 vertices[8] = {{-1.0, -1.0, -1.0},
                           {-1.0, -1.0,  1.0},
                           {-1.0,  1.0, -1.0},
                           {-1.0,  1.0,  1.0},
                           { 1.0, -1.0, -1.0},
                           { 1.0, -1.0,  1.0},
                           { 1.0,  1.0, -1.0},
                           { 1.0,  1.0,  1.0} };
```

```
point3 normals[6]  = {{ 0.0, 0.0, 1.0},
                      {-1.0, 0.0, 0.0},
                      { 0.0, 0.0,-1.0},
                      { 1.0, 0.0, 0.0},
                      { 0.0,-1.0, 0.0},
                      { 0.0, 1.0, 0.0} };

edge   edges[24]   = {{ 0, 1 }, { 1, 3 }, { 3, 2 }, { 2, 0 },
                      { 0, 4 }, { 1, 5 }, { 3, 7 }, { 2, 6 },
                      { 4, 5 }, { 5, 7 }, { 7, 6 }, { 6, 4 },
                      { 1, 0 }, { 3, 1 }, { 2, 3 }, { 0, 2 },
                      { 4, 0 }, { 5, 1 }, { 7, 3 }, { 6, 2 },
                      { 5, 4 }, { 7, 5 }, { 6, 7 }, { 4, 6 }};

face   cube[6]     = {{  0,  1,  2,  3 }, {  5,  9, 18, 13 },
                      { 14,  6, 10, 19 }, {  7, 11, 16, 15 },
                      {  4,  8, 17, 12 }, { 22, 21, 20, 23 }};
```

As we said before, drawing the cube proceeds by working our way through the face list and determining the actual points that make up the cube. We will expand the function we gave earlier to write the actual OpenGL code below. Each face is presented individually in a loop within the `glBegin-glEnd` pair, and with each face we include the normal for that face. Note that only the first vertex of the first edge of each face is identified, because the GL_QUADS drawing mode takes each set of four vertices as the vertices of a quad; it is not necessary to close the quad by including the first point twice.

```
void cube(void) {
   int face, edge;
   glBegin(GL_QUADS);
      for (face = 0; face < 6; face++) {
         glNormal3fv(normals[face];
         for (edge = 0; edge < 4; edge++)
            glVertex3fv(vertices[edges[cube[face][edge]][0]]);
      }
   glEnd();
}
```

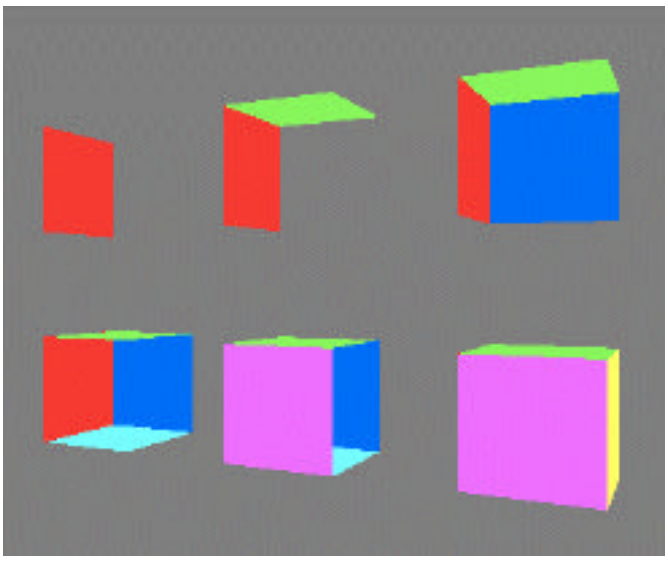


Figure 3.3: the cube as a sequence of quads

This cube is shown in Figure 3.3, presented through the six steps of adding individual faces (the faces are colored in the typical RGBCMY sequence so you may see each added in turn). This approach to defining the geometry is actually a fairly elegant way to define a cube, and takes very little coding to carry out. However, this is not the only approach we could take to defining a cube. Because the cube is a regular polyhedron with six faces that are squares, it is possible to define the cube by defining a standard square and then using transformations to create the faces from this master square. Carrying this out is left as an exercise for the student.

This approach to modeling an object includes the important feature of specifying the normals (the vectors perpendicular to each face) for the object. We will see in the chapters on lighting and shading that in order to get the added realism of lighting on an object, we must provide information on the object's normals, and it was straightforward to define an array that contains a normal for each face. Another approach would be to provide an array that contains a normal for each vertex if you would want smooth shading for your model; see the chapter on shading for more details. We will not pursue these ideas here, but you should be thinking about them when you consider modeling issues with lighting.

*Additional objects with the OpenGL toolkits*

Modeling with polygons alone would require you to write many standard graphics elements that are so common, any reasonable graphics system should include them. OpenGL includes the OpenGL Utility Library, GLU, with many useful functions, and most releases of OpenGL also include the OpenGL Utility Toolkit, GLUT. We saw in the first chapter that GLUT includes window management functions, and both GLU and GLUT include a number of built-in graphical elements that you can use. This chapter describes a number of these elements.

The objects that these toolkits provide are defined with several parameters that define the details, such as the resolution in each dimension of the object with which the object is to be presented. Many of these details are specific to the particular object and will be described in more detail when we describe each of these.

GLU quadric objects

The GLU toolkit provides several general quadric objects, which are objects defined by quadric equations (polynomial equations in three variables with degree no higher than two in any term), including Spheres (`gluSphere`), cylinders (`gluCylinder`), and disks (`gluDisk`). Each GLU primitive is declared as a GLUqadric and is allocated with the function

```
GLUquadric* gluNewQuadric( void )
```

Each quadric object is a surface of revolution around the z-axis. Each is modeled in terms of subdivisions around the z-axis, called slices, and subdivisions along the z-axis, called stacks. Figure 3.4 shows an example of a typical pre-built quadric object, a GLUT wireframe sphere, modeled with a small number of slices and stacks so you can see the basis of this definition.

The GLU quadrics are very useful in many modeling circumstances because you can use scaling and other transformations to create many common objects from them. The GLU quadrics are also useful because they have capabilities that support many of the OpenGL rendering capabilities that support creating interesting images. You can determine the drawing style with the `gluQuadricDrawStyle()` function that lets you select whether you want the object filled, wireframe, silhouette, or drawn as points. You can get normal vectors to the surface for lighting models and smooth shading with the `gluQuadricNormals()` function that lets you choose whether you want no normals, or normals for flat or smooth shading. Finally, with the `gluQuadricTexture()` function you can specify whether you want to apply texture maps to

the GLU quadrics in order to create objects with visual interest. See later chapters on lighting and on texture mapping for the details.

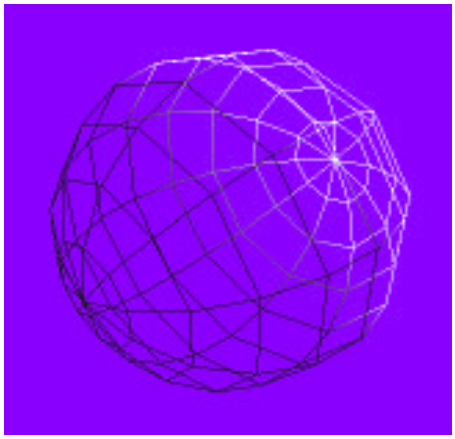


Figure 3.4: A GLUT wireframe sphere with 10 slices and 10 stacks

Below we describe each of the GLU primitives by listing its function prototype; more details may be found in the GLU section of your OpenGL manual.

GLU cylinder:

```
void gluCylinder(GLUquadric* quad, GLdouble base, GLdouble top,
GLdouble height, GLint slices, GLint stacks)
```

*quad* identifies the quadrics object you previously created with `gluNewQuadric`
*base* is the radius of the cylinder at z = 0, the base of the cylinder
*top* is the radius of the cylinder at z = height, and
*height* is the height of the cylinder.

GLU disk:

The GLU disk is different from the other GLU primitives because it is only two-dimensional, lying entirely within the X-Y plane. Thus instead of being defined in terms of stacks, the second granularity parameter is loops, the number of concentric rings that define the disk.

```
void gluDisk(GLUquadric* quad, GLdouble inner, GLdouble outer,
GLint slices, GLint loops)
```

*quad* identifies the quadrics object you previously created with `gluNewQuadric`
*inner* is the inner radius of the disk (may be 0).
*outer* is the outer radius of the disk.

GLU sphere:

```
void gluSphere(GLUquadric* quad, GLdouble radius, GLint slices,
GLint stacks)
```

*quad* identifies the quadrics object you previously created with `gluNewQuadric`
*radius* is the radius of the sphere.

The GLUT objects

Models provided by GLUT are more oriented to geometric solids, except for the teapot object. They do not have as wide a usage in general situations because they are of fixed shape and many

cannot be modeled with varying degrees of complexity. They also do not include shapes that can readily be adapted to general modeling situations. Finally, there is no general way to create a texture map for these objects, so it is more difficult to make scenes using them have stronger visual interest. The GLUT models include a cone (`glutSolidCone`), cube (`glutSolidCube`), dodecahedron (12-sided regular polyhedron, `glutSolidDodecahedron`), icosahedron (20-sided regular polyhedron, `glutSolidIcosahedron`), octahedron (8-sided regular polyhedron, `glutSolidOctahedron`), a sphere (`glutSolidSphere`), a teapot (the Utah teapot, an icon of computer graphics sometimes called the "teapotahedron", `glutSolidTeapot`), a tetrahedron (4-sided regular polyhedron, `glutSolidTetrahedron`), and a torus (`glutSolidTorus`). There are also wireframe versions of each of the GLUT solid objects.

The GLUT primitives include both solid and wireframe versions. Each object has a canonical position and orientation, typically being centered at the origin and lying within a standard volume and, if it has an axis of symmetry, that axis is aligned with the z-axis. As with the GLU standard primitives, the GLUT cone, sphere, and torus allow you to specify the granularity of the primitive's modeling, but the others do not.

If you have GLUT with your OpenGL, you should check the GLUT manuals for the details on these solids and on many other important capabilities that GLUT will add to your OpenGL system. If you do not already have it, you can download the GLUT code from the OpenGL Web site for many different systems and install it in your OpenGL area so you may use it readily with your system.

Selections from the overall collection of GLU and GLUT objects are shown in Figure 3.5 to show the range of items you can create with these tools. From top left and moving clockwise, we see a gluCylinder, a gluDisk, a glutSolidCone, a glutSolidIcosahedron, a glutSolidTorus, and a glutSolidTeapot. You should think about how you might use various transformations to create other figures from these basic parts.

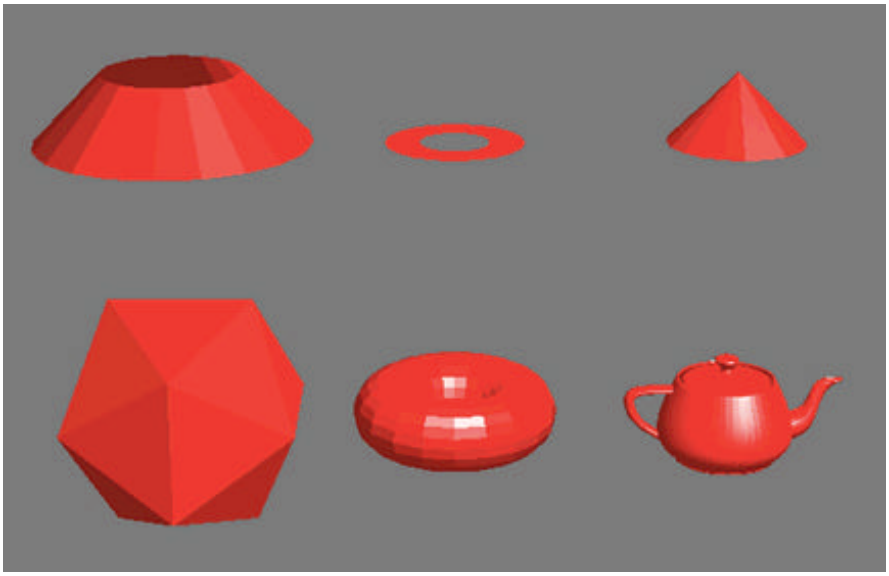


Figure 3.5: several GLU and GLUT objects as described above

An example

Our example for this module is quite simple. It is the heart of the `display()` function for a simple application that displays the built-in sphere, cylinder, dodecahedron, torus, and teapot provided by OpenGL and the GLU and GLUT toolkits. In the full example, there are operations that allow the user to choose the object and to control its display in several ways, but for this example we will only focus on the models themselves, as provided through a switch() statement such as might be used to implement a menu selection. This function is not complete, but would need the addition of viewing and similar functionality that is described in the chapter on viewing and projection.

```
void display( void )
{
      GLUquadric *myQuad;
      GLdouble radius = 1.0;
      GLint slices, stacks;
      GLint nsides, rings;

      ...

      switch (selectedObject) {
            case (1): {
                  myQuad=gluNewQuadric();
                  slices = stacks = resolution;
                  gluSphere( myQuad , radius , slices , stacks  );
                  break;
                  }
            case (2): {
                  myQuad=gluNewQuadric();
                  slices = stacks = resolution;
                  gluCylinder( myQuad, 1.0, 1.0, 1.0, slices, stacks );
                  break;
                  }
            case (3): {
                  glutSolidDodecahedron(); break;
                  }
            case (4): {
                  nsides = rings = resolution;
                  glutSolidTorus( 1.0, 2.0, nsides, rings);
                  break;
                  }
            case (5): {
                  glutSolidTeapot(2.0); break;
                  }
            }
      ...
}
```

*A word to the wise...*

One of the differences between student programming and professional programming is that students are often asked to create applications or tools for the sake of learning creation, not for the sake of creating working, useful things. The graphics primitives that are the subject of the first section of this module are the kind of tools that students are often asked to use, because they require more analysis of fundamental geometry and are good learning tools. However, working programmers developing real applications will often find it useful to use pre-constructed templates

and tools such as the GLU or GLUT graphics primitives. You are encouraged to use the GLU and GLUT primitives whenever they can save you time and effort in your work, and when you cannot use them, you are encouraged to create your own primitives in a way that will let you re-use them as your own library and will let you share them with others.

*Transformations in OpenGL*

In OpenGL, there are only two kinds of transformations: projection transformations and modelview transformations. The latter includes both the viewing and modeling transformations. We have already discussed projections and viewing, so here we will focus on the transformations used in modeling.

Among the modeling transformations, there are three fundamental kinds: rotations, translations, and scaling. In OpenGL, these are applied with the built-in functions (actually function sets) `glRotate`, `glTranslate`, and `glScale`, respectively. As we have found with other OpenGL function sets, there are different versions of each of these, varying only in the kind of parameters they take.

The `glRotate` function is defined as

```
glRotatef(angle, x, y, z)
```

where `angle` specifies the angle of rotation, in degrees, and `x`, `y`, and `z` specify the coordinates of a vector, all as floats (`f`). There is another rotation function `glRotated` that operates in exactly the same way but the arguments must all be doubles (`d`). The vector specified in the parameters defines the fixed line for the rotation. This function can be applied to any matrix set in `glMatrixMode`, allowing you to define a rotated projection if you are in projection mode or to rotate objects in model space if you are in modelview mode. You can use `glPushMatrix` and `glPopMatrix` to save and restore the unrotated coordinate system.

This rotation follows the right-hand rule, so the rotation will be counterclockwise as viewed from the direction of the vector `(x,y,z)`. The simplest rotations are those around the three coordinate axes, so that `glRotate(angle, 1., 0., 0.)` will rotate the model space around the X-axis.

The `glTranslate` function is defined as

```
glTranslatef(Tx, Ty, Tz)
```

where `Tx`, `Ty`, and `Tz` specify the coordinates of a translation vector as floats (`f`). Again, there is a translation function `glTranslated` that operates exactly the same but has doubles (`d`) as arguments. As with `glRotate`, this function can be applied to any matrix set in `glMatrixMode`, so you may define a translated projection if you are in projection mode or translated objects in model space if you are in modelview mode. You can again use `glPushMatrix` and `glPopMatrix` to save and restore the untranslated coordinate system.

The `glScale` function is defined as

```
glScalef(Sx, Sy, Sz)
```

where `Sx`, `Sy`, and `Sz` specify the coordinates of a scaling vector as floats (`f`). Again, there is a translation function `glScaled` that operates exactly the same but has doubles (`d`) as arguments. As above, this function can be applied to any matrix set in `glMatrixMode`, so you may define a scaled projection if you are in projection mode or scaled objects in model space if you are in modelview mode. You can again use `glPushMatrix` and `glPopMatrix` to save and restore the unscaled coordinate system. Because scaling changes geometry in non-uniform ways, a scaling transformation may change the normals of an object. If scale factors other than 1.0 are

applied in modelview mode and lighting is enabled, automatic normalization of normals should probably also be enabled. See the chapter on lighting for details.

As we saw earlier in the chapter, there are many transformations that go into defining exactly how a piece of geometry is presented in a graphics scene. When we consider the overall order of transformations for the entire model, we must consider not only the modeling transformations but also the projection and viewing transformations. If we consider the total sequence of transformations in the order in which they are specified, we will have the sequence:

```
P   V   T0  T1  …   Tn  Tn+1    …   Tlast
```

with `P` being the projection transformation, `V` the viewing transformation, and `T0`, `T1`, … `Tlast` the transformations specified in the program to model the scene, in order (`T1` is first, `Tlast` is last and is closest to the actual geometry). The projection transformation is defined in the `reshape` function; the viewing transformation is defined in the `init` function, in the `reshape` function, or at the beginning of the `display` function so it is defined at the beginning of the modeling process. But the sequence in which the transformations are applied is actually the reverse of the sequence above: `Tlast` is actually applied first, and `V` and finally `P` are applied last. You need to understand this sequence very well, because it's critical to understand how you build complex, heirarchical models.

*Code examples for transformations*

Simple transformations:
All the code examples use a standard set of axes, which are not included here, and the following definition of the simple square:

```
void square (void)
{
   typedef GLfloat point [3];
   point v[8] = {{12.0, -1.0, -1.0},
                 {12.0, -1.0,  1.0},
                 {12.0,  1.0,  1.0},
                 {12.0,  1.0, -1.0} };

   glBegin (GL_QUADS);
      glVertex3fv(v[0]);
      glVertex3fv(v[1]);
      glVertex3fv(v[2]);
      glVertex3fv(v[3]);
   glEnd();
}
```

To display the simple rotations example, we use the following display function:

```
void display( void )
{  int i;
   float theta = 0.0;

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   axes(10.0);
   for (i=0; i<8; i++) {
      glPushMatrix();
      glRotatef(theta, 0.0, 0.0, 1.0);
      if (i==0) glColor3f(1.0, 0.0, 0.0);
         else  glColor3f(1.0, 1.0, 1.0);
      square();
      theta += 45.0;
```

```
      glPopMatrix();
   }
   glutSwapBuffers();
}
```

To display the simple translations example, we use the following display function:

```
void display( void )
{  int i;

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   axes(10.0);
   for (i=0; i<=12; i++) {
      glPushMatrix();
      glTranslatef(-2.0*(float)i, 0.0, 0.0);
      if (i==0) glColor3f(1.0, 0.0, 0.0);
         else  glColor3f(1.0, 1.0, 1.0);
      square();
      glPopMatrix();
   }
   glutSwapBuffers();
}
```

To display the simple scaling example, we use the following display function:

```
void display( void )
{  int i;
   float s;

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   axes(10.0);
   for (i=0; i<6; i++) {
      glPushMatrix();
      s = (6.0-(float)i)/6.0;
      glScalef( s, s, s );
      if (i==0) glColor3f(1.0, 0.0, 0.0);
         else  glColor3f(1.0, 1.0, 1.0);
      square();
      glPopMatrix();
   }
   glutSwapBuffers();
}
```

Transformation stacks: The OpenGL functions that are used to manage the transformation stack are `glPushMatrix()` and `glPopMatrix()`. Technically, they apply to the stack of whatever transformation is the current matrix mode, and the `glMatrixMode` function with parameters `GL_PROJECTION` and `GL_MODELVIEW` sets that mode. We only rarely want to use a stack of projection transformations (and in fact the stack of projections can only hold two transformations) so we will almost always work with the stack of modeling/viewing transformation. The rabbit head example was created with the display function given below. This function makes the stack operations more visible by using indentations; this is intended for emphasis in the example only and is not standard programming practice in graphics. Note that we have defined only very simple display properties (just a simple color) for each of the parts; we could in fact have defined a much more complex set of properties and have made the parts much more visually interesting. We could also have used a much more complex object than a simple `gluSphere` to make the parts much more structurally interesting. The sky's the limit…

```
void display( void )
{
// Indentation level shows the level of the transformation stack
// The basis for this example is the unit gluSphere; everything else
// is done by explicit transformations

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glPushMatrix();
      // model the head
      glColor3f(0.4, 0.4, 0.4);  //    dark gray head
      glScalef(3.0, 1.0, 1.0);
      myQuad = gluNewQuadric();
      gluSphere(myQuad, 1.0, 10, 10);
   glPopMatrix();
   glPushMatrix();
      // model the left eye
      glColor3f(0.0, 0.0, 0.0);  //    black eyes
      glTranslatef(1.0, -0.7, 0.7);
      glScalef(0.2, 0.2, 0.2);
      myQuad = gluNewQuadric();
      gluSphere(myQuad, 1.0, 10, 10);
   glPopMatrix();
   glPushMatrix();
      // model the right eye
      glTranslatef(1.0, 0.7, 0.7);
      glScalef(0.2, 0.2, 0.2);
      myQuad = gluNewQuadric();
      gluSphere(myQuad, 1.0, 10, 10);
   glPopMatrix();
   glPushMatrix();
      // model the left ear
      glColor3f(1.0, 0.6, 0.6);  //    pink ears
      glTranslatef(-1.0, -1.0, 1.0);
      glRotatef(-45.0, 1.0, 0.0, 0.0);
      glScalef(0.5, 2.0, 0.5);
      myQuad = gluNewQuadric();
      gluSphere(myQuad, 1.0, 10, 10);
   glPopMatrix();
   glPushMatrix();
      // model the right ear
      glColor3f(1.0, 0.6, 0.6);  //    pink ears
      glTranslatef(-1.0, 1.0, 1.0);
      glRotatef(45.0, 1.0, 0.0, 0.0);
      glScalef(0.5, 2.0, 0.5);
      myQuad = gluNewQuadric();
      gluSphere(myQuad, 1.0, 10, 10);
   glPopMatrix();
   glutSwapBuffers();
}
```

In OpenGL, the stack for the modelview matrix is to be at least 32 deep, but this can be inadequate to handle some complex models if the hierarchy is more than 32 layers deep. In this case, as we mentioned in the previous chapter, you need to know that a transformation is a 4x4 matrix of `GLfloat` values that is stored in a single array of 16 elements. You can create your own stack of these arrays that can have any depth you want, and then push and pop transformations as you wish on that stack. To deal with the modelview transformation itself, there are functions that allow you to save and to set the modelview transformation as you wish. You can capture the current value of the transformation with the function

```
glGetFloatv(GL_MODELVIEW_MATRIX, viewProj);
```
(here we have declared `GLfloat viewProj[16]`), and you can use the functions
```
glLoadIdentity();
glMultMatrixf( viewProj );
```
to set the current modelview matrix to the value of the matrix `viewProj`, assuming that you were in modelview mode when you execute these functions.

Creating display lists

In OpenGL, graphics objects can be compiled into what is called a *display list*, which will contain the final geometry of the object as it is ready for display. OpenGL display lists are named by nonzero unsigned integer values (technically, GLuint values) and there are several tools available in OpenGL to manage these name values. We will assume in a first graphics course that you will not need many display lists and that you can manage a small number of list names yourself, but if you begin to use a number of display lists in a project, you should look into the `glGenLists`, `glIsList`, and `glDeleteLists` functions to help you manage the lists properly. Sample code and a more complete explanation is given below.

Display lists are relatively easy to create in OpenGL. First, choose an unsigned integer (often you will just use small integer constants, such as 1, 2, ...) to serve as the name of your list. Then before you create the geometry for your list, call the function `glNewList`. Code whatever geometry you want into the list, and at the end, call the function `glEndList`. Everything between the new list and the end list functions will be executed whenever you call `glCallList` with a valid list name as parameter. All the operations between `glNewList` and `glEndList`will be carried out, and only the actual set of instructions to the drawing portion of the OpenGL system will be saved. When the display list is executed, then, those instructions are simply sent to the drawing system; any operations needed to generate these instructions are omitted.

Because display lists are often defined only once, it is common to create them in the `init()` function or in a function called from within `init()`. Some sample code is given below, with most of the content taken out and only the display list operations left.

```
void Build_lists(void) {
   glNewList(1, GL_COMPILE);
      glBegin(GL_TRIANGLE_STRIP);
         glNormal3fv(...); glVertex3fv(...);
         ...
      glEnd();
   glEndList();
}

static void Init(void) {
   ...
   Build_lists();
   ...
}

void Display(void) {
   ...
   glCallList(1);
   ...
}
```

You will note that the display list was created in `GL_COMPILE` mode, and it was not executed (the object was not displayed) until the list was called. It is also possible to have the list displayed as it is created if you create the list in `GL_COMPILE_AND_EXECUTE` mode.

# Mathematics for Modeling

The primary mathematical background needed for computer graphics programming is 3D analytic geometry. It is unusual to see a course with this title, however, so most students pick up bits and pieces of mathematics background that fill this in. One of the common sources of the background is introductory physics; another is multivariate calculus. Neither of these is a common requirement for computer graphics, however, so here we will outline the general concepts we will use in these notes.

Coordinate systems and points

The set of real numbers — often thought of as the set of all possible distances — is a mathematical abstraction that is effectively modeled as a Euclidean straight line with two uniquely-identified points. One point is identified with the number 0.0 (we write all real numbers with decimals, to meet the expectations of programming languages), called the origin, and the other is identified with the number 1.0, which we call the unit point. The direction of the line from 0.0 to 1.0 is called the positive direction; the opposite direction of the line is called the negative direction. These directions identify the parts of the lines associated with positive and negative numbers, respectively.

In this model, any real number is identified with the unique point on the line that is

- at the distance from the origin which is that number times the distance from 0.0 to 1.0, and
- in the direction of the number's sign.

We have heard that a line is determined by two points; let's see how that can work. Let the first point be `P0=(X0,Y0,Z0)` and the second point be `P1=(X1,Y1,Z1)`. Let's call `P0` the origin and `P1` the unit point. Points on the segment are obtained by starting at the "first" point `P0` offset by a fraction of the difference vector `P1-P0`. Then any point `P=(X,Y,Z)` on the line can be expressed in vector terms by

```
P = P0 + t*(P1-P0) = (1-t)*P0 + t*P1
```

for a single value of a real variable `t`. This computation is actually done on a per-coordinate basis, with one equation each for X, Y, and Z as follows:

```
X = X0 + t*(X1-X0) = (1-t)*X0 + t*X1
Y = Y0 + t*(Y1-Y0) = (1-t)*Y0 + t*Y1
Z = Z0 + t*(Z1-Z0) = (1-t)*Z0 + t*Z1
```

Thus any line segment can be determined by a single parameter, and so is called a 1–dimensional object. This is illustrated in Figure 4.1 below that represents a way to calculate the coordinates of the points along a line segment by incrementing the value of `t` from 0 to 1.

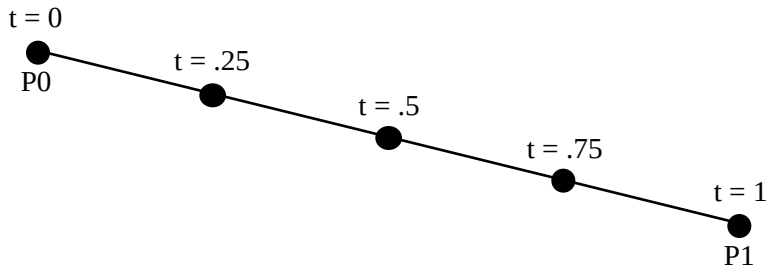



Figure 4.1: a parametric line segment with some values of the parameter

This representation for a line segment (or an entire line, if you place no restrictions on the value of `t`) also allows you to compute intersections involving lines. The reverse concept is also useful, so if you have a known point on the line, you can calculate the value of the parameter `t` that would produce that point. For example, if a line intersects an object at a point `Q`, a vector calculation of

the form `P0+t*(P1-P0)=Q` would allow you to fine the parameter `t` that gives the intersection point on the line. This calculation is not usually applied for points, however, because the calculation for any of the single variables `X`, `Y`, or `Z` would yield the same value of `t`. This is often the basis for geometric computations such as the intersection of a line and a plane.

If we have two straight lines that are perpendicular to each other and meet in a point, we can define that point to be the origin for both lines, and choose two points the same distance from the origin on each line as the unit points. A distance unit is defined to be used by each of the two lines, and the points at this distance from the intersection point are marked, one to the right of the intersection and one above it. This gives us the classical 2D coordinate system, often called the Cartesian coordinate system. The vectors from the intersection point to the right-hand point (respectively the point above the intersection) are called the X- and Y-direction vectors and are indicated by `i` and `j` respectively. Points in this system are represented by an ordered pair of real numbers, `(X,Y)`, and this is probably the most familiar coordinate system to most people. These points may also be represented by a vector `<X,Y>` from the origin to the point, and this vector may be expressed in terms of the direction vectors as `Xi+Yj`.

In 2D Cartesian coordinates, any two lines that are not parallel will meet in a point. The lines make four angles when they meet, and the acute angle is called the angle between the lines. If two line segments begin at the same point, they make a single angle that is called the angle between the line segments. These angles are measured with the usual trigonometric functions, and we assume that the reader will have a modest familiarity with trigonometry. Some of the reasons for this assumption can be found in the discussions below on polar and spherical coordinates, and in the description of the dot product and cross product. We will discuss more about the trigonometric aspects of graphics when we get to that point in modeling or lighting.

The 3D world in which we will do most of our computer graphics work is based on 3D Cartesian coordinates that extend the ideas of 2D coordinates above. This is usually presented in terms of three lines that meet at a single point, which is identified as the origin for all three lines and is called the *origin*, that have their unit points the same distance from that point, and that are mutually perpendicular. Each point represented by an ordered triple of real numbers `(x,y,z)`. The three lines correspond to three unit direction vectors, each from the origin to the unit point of its respective line; these are named **`i`**, **`j`**, and **`k`** for the X-, Y-, and Z-axis, respectively, and are called the canonical basis for the space, and the point can be represented as `xi+yj+zk`. Any triple of numbers is identified with the point in the space that lies an appropriate distance from the two-axis planes. This is all illustrated in Figure 4.2 below.

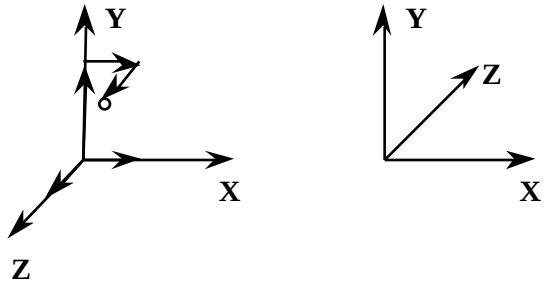



Figure 4.2: right-hand coordinate system with origin (left) and with a point identified by its coordinates; left-hand coordinate system (right)

3D coordinate systems can be either right-handed or left-handed: the third axis can be the cross product of the first two axes, or it can be the negative of that cross product, respectively. (We will talk about cross products a little later in this chapter.) The "handed-ness" comes from a simple technique: if you hold your hand in space with your fingers along the first axis and curl your

fingers towards the second axis, your thumb will point in a direction perpendicular to the first two axis. If you do this with the right hand, the thumb points in the direction of the third axis in a right-handed system. If you do it with the left hand, the thumb points in the direction of the third axis in a left-handed system.

Some computer graphics systems use right-handed coordinates, and this is probably the most natural coordinate system for most uses. For example, this is the coordinate system that naturally fits electromagnetic theory, because the relationship between a moving current in a wire and the magnetic field it generates is a right-hand coordinate relationship. The modeling in Open GL is based on a right-hand coordinate system.

On the other hand, there are other places where a left-handed coordinate system is natural. If you consider a space with a standard X-Y plane as the front of the space and define Z as the distance back from that plane, then the values of Z naturally increase as you move back into the space. This is a left-hand relationship.

Line segments and curves

In standard Euclidean geometry, two points determine a line as we noted above. In fact, in the same way we talked about any line having unique origin identified with 0.0 and unit point identified with 1.0, a line segment — the points on the line between these two particular points — can be identified as the points corresponding to values between 0 and 1. It is done by much the same process as we used to illustrate the 1-dimensional nature of a line above. That is, just as in the discussion of lines above, if the two points are `P0` and `P1`, we can identify any point between them as `P = (1-t)*P0 + t*P1` for a unique value of `t` between 0 and 1. This is called the parametric form for a line segment

The line segment gives us an example of determining a continuous set of points by functions from the interval [0.1] to 3-space. In general, if we consider any set of functions `x(t)`, `y(t)`, and `z(t)` that are defined on [0,1] and are continuous, the set of points they generate is called a *curve* in 3-space. There are some very useful examples of such curves, which can display the locations of a moving point in space, the positions from which you will view a scene in a fly-through, or the behavior of a function of two variables if the values two variables lie on a curve in 2-space.

Dot and cross products

There are two computations that we will need to understand, and sometimes to perform, in developing the geometry for our graphic images. The first is the *dot product* of two vectors. This produces a single real value that represents the projection of one vector on the other and its value is the product of the lengths of the two vectors times the cosine of the angle between them. The dot product computation is quite simple: it is simply the sum of the componentwise products of the vectors. If the two vectors A and B are

```
A = (X1,Y1,Z1)
B = (X2,Y2,Z2),
```

the dot product is computed as

```
A•B = X1*X2+Y1*Y2+Z1*Z2.
```

The second computation is the *cross product* of two vectors. This produces a vector that is perpendicular to each of the original vectors and whose length is the product of the two vector lengths times the sin of the angle between them. Thus if two vectors are parallel, the cross product is zero; if they are orthogonal, the cross product has length equal to the product of the two lengths; if they are both unit vectors, the cross product is the sine of the included angle. The computation of the cross product can be expressed as the determinant of a matrix whose first row is the three

standard unit vectors, whose second row is the first vector of the product, and whose third row is the second vector of the product. Denoting the unit direction vectors in the X, Y, and Z directions as `i`, `j`, and `k` , we can express the cross product of two vectors `<a,b,c>` and `<u,v,w>` in terms of a determinant:

$$\langle a,b,c\rangle \times \langle u,v,w\rangle = \det \begin{vmatrix} i & j & k \\ a & b & c \\ u & v & w \end{vmatrix}$$

The cross product has a "handedness" property and is said to be a right-handed operation. That is, if you align the fingers of your right hand with the direction of the first vector and curl your fingers towards the second vector, your right thumb will point in the direction of the cross product. Thus the order of the vectors is important; if you reverse the order, you reverse the sign of the product (recall that interchanging two rows of a determinant will change its sign), so the cross product operation is not commutative. As a simple example, with `i`, `j`, and `k` as above, we see that `i x j = k` but that `j x i = -k`. In general, if you consider the arrangement of Figure 4.3, if you think of the three direction vectors as being wrapped around as if they were visible from the first octant of 3-space, the product of any two is the third direction vector if the letters are in counterclockwise order, and the negative of the third if the order is clockwise. Note also that the cross product of two collinear vectors (one of the vectors is a constant multiple of the other) will always be zero, so the geometric interpretation of the cross product does not apply in this case.

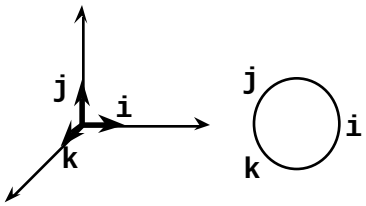



Figure 4.3: the direction vectors in order

The cross product can be very useful when you need to define a vector perpendicular to two given vectors; the most common application of this is defining a normal vector to a polygon by computing the cross product of two edge vectors. For a triangle as shown in Figure 4.4 below with vertices `A`, `B`, and `C` in order counterclockwise from the "front" side of the triangle, the normal vector can be computed by creating the two difference vectors `P = C - B` and `Q = A - C`, and computing the cross product as `P x Q` to yield a vector `N` normal to the plane of the triangle. In fact, we can say more than this; the cross product of two vectors is not only perpendicular to the plane defined by those vectors, but its length is the product of their lengths times the sine of the angle between them. As we shall see in the next section, this normal vector, and any point on the triangle, allow us to generate the equation of the plane that contains the triangle. When we need to use this normal for lighting, we will need to normalize it, or make it a unit vector, but that can easily be done by calculating its length and dividing each component by that length.

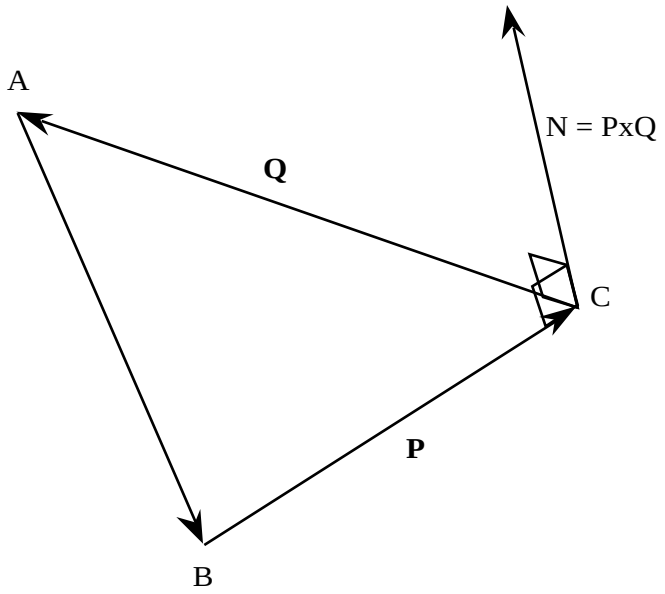



Figure 4.4: the normal to a triangle as the cross product of two edges

Planes and half-spaces

We saw above that a line could be defined in terms of a single parameter, so it is often called a one-dimensional space. A plane, on the other hand, is a two-dimensional space, determined by two parameters. If we have any two non-parallel lines that meet in a single point, we recall that they determine a plane that can be thought of as all points that are translations of the given point by vectors that are linear combinations of the direction vectors of the two lines. Thus any plane in space is seen as two-dimensional where each of the two lines contributes one of the dimensional components. In more general terms, let's consider the vector `N=<A,B,C>` defined as the cross product of the two vectors determined by the two lines. Then `N` is perpendicular to each of the two vectors and hence to any line in the plane. In fact, this can be taken as defining the plane: the plane is defined by all lines through the fixed point perpendicular to `N`. If we take a fixed point in the plane, `(U,V,W)`, and a variable point in the plane, `(x,y,z)`, we can write the perpendicular relationship as

```
<A,B,C>•<x-U,y-V,z-W>=0.
```

When we expand this dot product we see

```
A(x-U)+b(y-V)+C(z-W)=Ax+By+Cz+(-AU-BV-CW)=0.
```

This allows us to give an equation for the plane:

```
Ax+By+Cz+D=0.
```

Thus the coefficients of the variables in the plane equation exactly match the components of the vector normal to the plane — a very useful fact from time to time.

Any line divides a plane into two parts. If we know the equation of the line in the traditional form

```
ax + by + c = 0,
```

then we can determine whether a point lies on, above, or below the line by evaluating the function `f(x,y) = ax + by + c` and determining whether the result is zero, positive, or negative. In a similar way, the equation for the plane as defined above does more than just identify the plane; it allows us to determine on which side of the plane any point lies. If we create a function of three variables from the plane equation

```
f(x,y,z)=Ax+By+Cz+D,
```

then the plane consists of all points where `f(x,y,z)=0`. All points `(x,y,z)` with `f(x,y,z)>0` lie on one side of the plane, called the positive half-space for the plane, while all points with `f(x,y,z)<0` lie on the other, called the negative half-space for the plane. We will find that OpenGL uses the four coordinates A,B,C,D to identify a plane and uses the half-space concept to choose displayable points when the plane is used for clipping.

Polygons and convexity

Most graphics systems, including OpenGL, are based on modeling and rendering based on polygons and polyhedra. A *polygon* is a plane region bounded by a sequence of directed line segments with the property that the end of one segment is the same as the start of the next segment, and the end of the last line segment is the start of the first segment. A *polyhedron* is a region of 3–space that is bounded by a set of polygons. Because polyhedra are composed of polygons, we will focus on modeling with polygons, and this will be a large part of the basis for the modeling chapter below.

The reason for modeling based on polygons is that many of the fundamental algorithms of graphics have been designed for polygon operations. In particular, many of these algorithms operate by interpolating values across the polygon; you will see this below in depth buffering, shading, and other areas. In order to interpolate across a polygon, the polygon must be *convex*. Informally, a polygon is complex if it has no indentations; formally, a polygon is complex if for any two points in the polygon (either the interior or the boundary), the line segment between them lies entirely within the polygon.

Because a polygon or polyhedron bounds a region of space, we can talk about the interior or exterior of the figure. In a convex polygon or polyhedron, this is straightforward because the figure is defined by its bounding planes or lines, and we can simply determine which side of each is "inside" the figure. For a non-convex figure this is less simple, so we look to convex figures for a starting point and notice that if a point is inside the figure, any ray from an interior point (line extending in only one direction from the point) must exit the figure in precisely one point, while if a point is outside the figure, if the ray hits the polygon it must both enter and exit, and so crosses the boundary of the figure in either 0 or 2 points. We extend this idea to general polygons by saying that a point is inside the polygon if a ray from the point crosses the boundary of the polygon an odd number of times, and is outside the polygon if a ray from the point crosses the boundary of the polygon an even number of times. This is illustrated in Figure 4.5. In this figure, points A, D, E, and G are outside the polygons and points B, D, and F are inside. Note carefully the case of point G; our definition of inside and outside might not be intuitive in some cases.

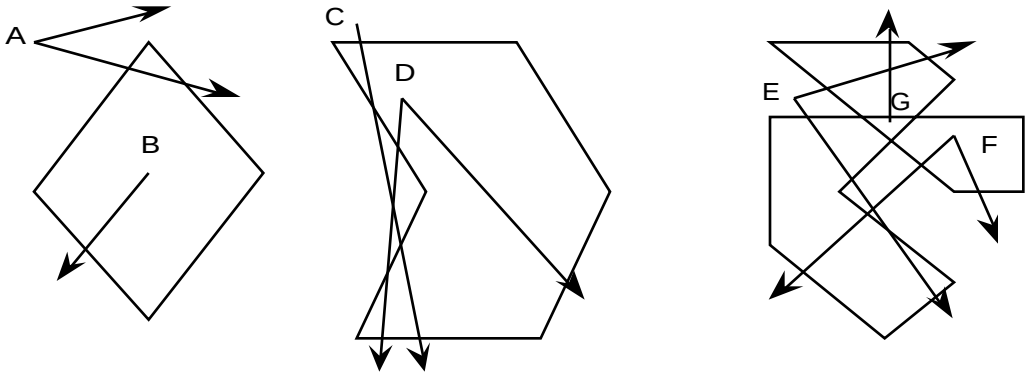



Figure 4.5: Interior and exterior points of a convex polygon (left) and two general polygons (center and right)

Another way to think about convexity is in terms of linear combinations of points. We can define a *convex sum* of points `P0`, `P1`, … `Pn` as a sum $\sum c_i Pi$ where each of the coefficients $c_i$ is non-negative and the sum of the coefficients is exactly 1. If we recall the parametric definition of a line segment, `(1-t)*P0+t*P1`, we note that this is a convex sum. So a polygon is convex if certain convex sums of points in the polygon must also lie in the polygon. However, it is straightforward to see that this can be generalized to say that all convex sums of points in a convex polygon must lie in the polygon, so this gives us an alternate definition of convex polygons that can sometimes be useful.

As we suggested above, most graphics systems, and certainly OpenGL, require that all polygons be convex in order to render them correctly. If you have a polygon that is not convex, you may always subdivide it into triangles or other convex polygons and work with them instead of the original polygon. As an alternative, OpenGL provides a facility to tesselate a polygon — divide it into convex polygons — automatically, but this is a complex operation that we do not cover in these notes.

Line intersections

There are times when we need to know whether two objects meet in order to understand the logic of a particular scene. Calculating whether there are intersections is relatively straightforward and we outline it here. The fundamental question is whether a line segment that is part of one object meets a triangle that is part of the second object.

There are two levels at which we might be able to determine whether an intersection occurs. The first is to see whether the line containing the segment can even come close enough to meet the triangle, and the second is whether the segment actually meets the triangle. The reason for this two-stage question is that most of the time there will be few segments that could even come close to intersecting, so we will ask the first question because it is fastest and will only ask the second question when the first indicates it can be useful.

To consider the question of whether a line can come close enough to meet the triangle, look at the situation outlined on the left in Figure 2.29. We first compute the incenter of the triangle and then define the bounding circle to lie in the plane of the triangle, to have that point as center, and to have as its radius the distance from that point to each vertex:

center = `(P0 + P1 + P2)/3`

radius = `distance (center, P0)`

Then we can compute the point where the line meets the plane of the triangle. Let the line segment be given by the parametric equation `Q0 + t*(Q1 - Q0)`; the entire line is given by considering all values of `t`, and if you consider only values of `t` between 0 and 1 you get the segment. Next compute the cross product of two edges of the triangle in order and call the result N. Then the equation of the plane is given by the processes described earlier in this chapter, and when the parametric equation for the line meets the plane we can solve for a single value of `t`. If that value of `t` does not lie between 0 and 1 we can immediately conclude that there is no possible intersection because the line segment does not meet the plane at all. If the value does lie between 0 and 1, we calculate the point where the line segment meets the plane and compute the distance from that point to the center of the triangle. The line cannot meet the triangle unless this distance is less than the radius of the circle.

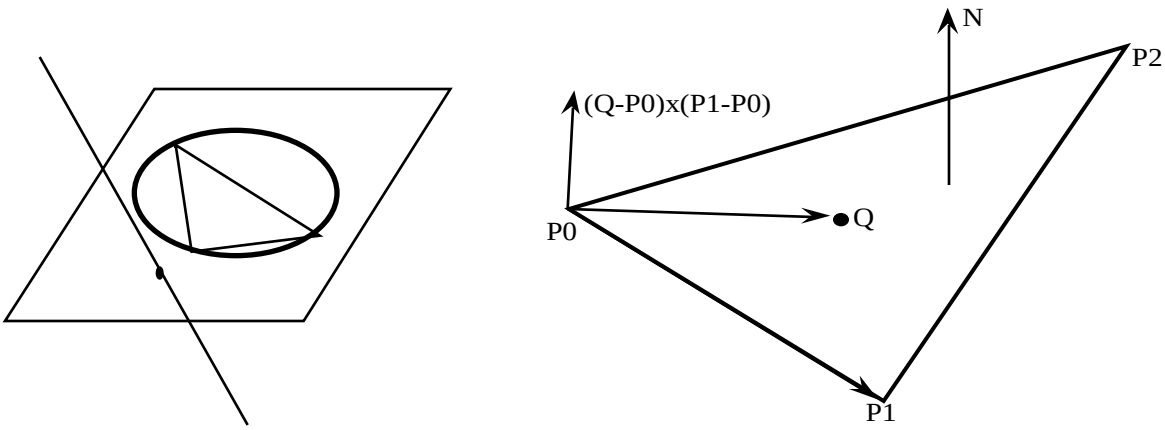



Figure 4.6: a line and the bounding circle (left) and a line and triangle (right)

Once we know that the line is close enough to have a potential intersection, we move on to look at the exact computation of whether the point where the line meets the plane is inside the triangle, as shown in the right-hand part of Figure 4.6. We note that a point on the inside of the triangle is characterized by being to the left of the oriented edge for each edge of the triangle. This is further characterized by the cross product of the edge vector and the vector from the vertex to the point; if this cross product has the same orientation as the normal vector to the triangle for each vertex, then the point is inside the triangle. If the intersection of the line segment and the triangle's plane is `Q`, this means that we must have `N•((Q-P0)x(P1-P0))>0` for the first edge, and similar relations for each subsequent edge.

Polar, cylindrical, and spherical coordinates

Up to this point we have emphasized Cartesian, or rectangular, coordinates for describing 2D and 3D geometry, but there are times when other kinds of coordinate systems are most useful. The coordinate systems we discuss here are based on angles, not distances, in at least one of their terms. Because OpenGL does not handle these coordinate systems directly, when you want to use them you will need to translate points between these forms and rectangular coordinates.

In 2D coordinates, we can identify any point `(X,Y)` with the line segment from the origin to that point. This identification allows us to write the point in terms of the angle `Θ` the line segment makes with the positive X-axis and the distance R of the point from the origin as:

`X = R*cos(Θ), Y = R*sin(Θ)` or, inversely,

`R = sqrt(X`$^2$ `+ Y`$^2$`), Θ = arccos(X/R)` where `Θ` is the value that matches the signs of X and Y.

This representation `(R,Θ)` is known as the *polar form* for the point, and the use of the polar form for all points is called the *polar coordinates* for 2D space.

There are two alternatives to Cartesian coordinates for 3D space. *Cylindrical coordinates* add a third linear dimension to 2D polar coordinates, giving the angle between the X-Z plane and the plane through the Z-axis and the point, along with the distance from the Z-axis and the Z-value of the point. Points in cylindrical coordinates are represented as `(R,Θ,Z)` with R and `Θ` as above and with the Z-value as in rectangular coordinates. Figure 4.7 shows the structure of the 2D and 3D spaces with polar and cylindrical coordinates, respectively.

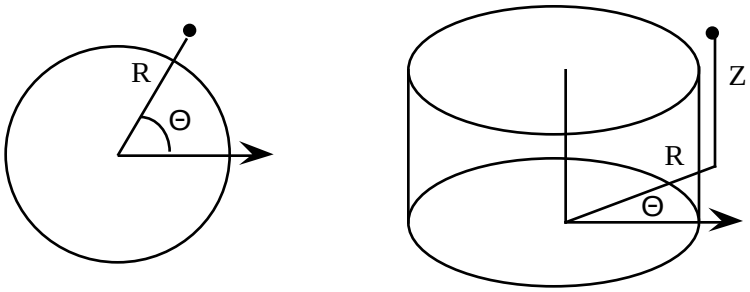



Figure 4.7: polar coordinates (left) and cylindrical coordinates (right)

Cylindrical coordinates are a useful extension of a 2D polar coordinate model to 3D space. They not particularly common in graphics modeling, but can be very helpful when appropriate. For example, if you have a planar object that has to remain upright with respect to a vertical direction, but the object has to rotate to face the viewer in a scene as the viewer moves around, then it would be appropriate to model the object's rotation using cylindrical coordinates. An example of such an object is a billboard, as discussed later in the chapter on high-efficiency graphics techniques.

*Spherical coordinates* represent 3D points in terms much like the latitude and longitude on the surface of the earth. The latitude of a point is the angle from the equator to the point, and ranges from 90° south to 90° north. The longitude of a point is the angle from the "prime meridian" to the point, where the prime meridian is determined by the half-plane that runs from the center of the earth through the Greenwich Observatory just east of London, England. The latitude and longitude valued uniquely determine any point on the surface of the earth, and any point in space can be represented relative to the earth by determining what point on the earth's surface meets a line from the center of the earth to the point, and then identifying the point by the latitude and longitude of the point on the earth's surface and the distance to the point from the center of the earth. Spherical coordinates are based on the same principle: given a point and a unit sphere centered at that point, with the sphere having a polar axis, determine the coordinates of a point P in space by the latitude Φ (angle north or south from the equatorial plane) and longitude Θ (angle from a particular half-plane through the diameter of the sphere perpendicular to the equatorial plane) of the point where the half-line from the center of the sphere, and determine the distance from the center to that point. Then the spherical coordinates of P are (R, Θ, Φ).

Spherical coordinates can be very useful when you want to control motion to achieve smooth changes in angles or distances around a point. They can also be useful if you have an object in space that must constantly show the same face to the viewer as the viewer moves around; again, this is another kind of billboard application and will be described later in these notes.

It is straightforward to convert spherical coordinates to 3D Cartesian coordinates. Noting the relationship between spherical and rectangular coordinates shown in Figure 2.31 below, and noting that this figure shows the Z-coordinate as the vertical axis, we see the following conversion equations from polar to rectangular coordinates.

$$x = R \cos(\Phi) \sin(\Theta)$$
$$y = R \cos(\Phi) \cos(\Theta)$$
$$z = R \sin(\Phi)$$

Converting from rectangular to spherical coordinates is not much more difficult. Again referring to Figure 2.31, we see that R is the diagonal of a rectangle and that the angles can be described in terms of the trigonometric functions based on the sides. So we have the equations

$$R = \sqrt{X^2 + Y^2 + Z^2}$$
$$\Phi = \arcsin(Z/R)$$
$$\Theta = \arctan(X/\sqrt{X^2 + Y^2})$$

Note that the inverse trigonometric function is the principle value for the longitude ($\Phi$), and the angle for the latitude ($\Theta$) is chosen between 0° and 360° so that the sine and cosine of $\Theta$ match the algebraic sign (+ or -) of the X and Y coordinates.

Figure 4.8 shows a sphere showing latitude and longitude lines and containing an inscribed rectangular coordinate system, as well as the figure needed to make the conversion between spherical and rectangular coordinates.

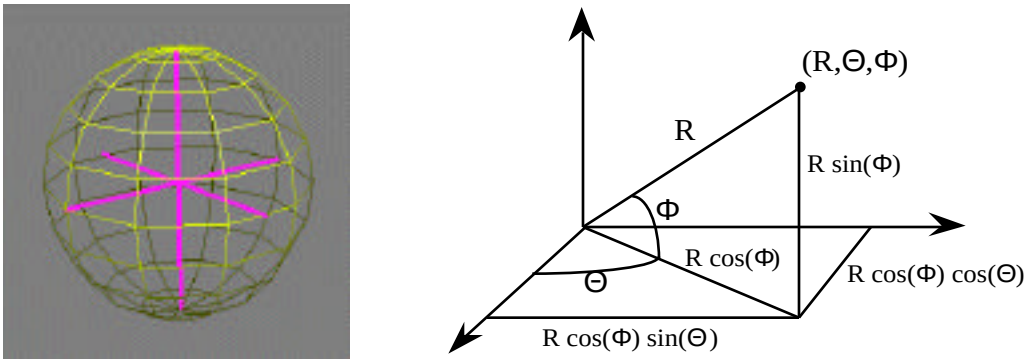



Figure 4.8: spherical coordinates (left);
the conversion from spherical to rectangular coordinates (right)

Higher dimensions?

While our perceptions and experience are limited to three dimensions, there is no such limit to the kind of information we may want to display with our graphics system. Of course, we cannot deal with these higher dimensions directly, so we will have other techniques to display higher-dimensional information. There are some techniques for developing three-dimensional information by projecting or combining higher-dimensional data, and some techniques for adding extra non-spatial information to 3D information in order to represent higher dimensions. We will discuss some ideas for higher-dimensional representations in later chapters in terms of visual communications and science applications.

# Color and Blending

*Prerequisites*

No background in color is required; this chapter discusses color issues from first principles.

*Introduction*

Color is a fundamental concept for computer graphics. We need to be able to define colors for our graphics that represent good approximations of real-world colors, and we need to be able to manipulate colors as we develop our applications.

There are many ways to specify colors, but all depend principally on the fact that the human visual system generally responds to colors through the use of three kinds of cells in the retina of the eye. This response is complex and includes both physical and psychological processes, but the fundamental fact of three kinds of stimulus is maintained by all the color models in computer graphics. For most work, the usual model is the RGB (Red, Green, Blue) color model that matches in software the physical design of computer monitors, which are made with a pattern of three kinds of phosphor which emit red, green, and blue light when they are excited by an electron beam. This RGB model is used for color specification in almost all computer graphics APIs, and it is the basis for the discussion here. There are a number of other models of color, and we refer you to textbooks and other sources, especially Foley et al. [FvD], for additional discussions on color models and for more complete information on converting color representations from one model to another.

Because the computer monitor uses three kinds of phosphor, and each phosphor emits light levels based on the energy of the electron beam that is directed at it, a common approach is to specify a color by the level of each of the three primaries. These levels are a proportion of the maximum light energy that is available for that primary, so an RGB color is specified by a triple (r, g, b) where each of the three components represents the amount of that particular component in the color and where the ordering is the red-green-blue that is implicit in the name RGB. This proportion for each primary is represented by a real number between 0.0 and 1.0, inclusive. There are other ways to represent colors, of course. In an integer-based system that is also often used, each color component can be represented by an integer that depends on the color depth available for the system; if you have eight bits of color for each component, which is a common property, the integer values are in the range 0 to 255. The real-number approach is used more commonly in graphics APIs because it is more device-independent. In either case, the number represents the proportion of the available color of that primary hue that is desired for the pixel. Thus the higher the number for a component, the brighter is the light in that color, so with the real-number representation, black is represented by (0.0, 0.0, 0.0) and white by (1.0, 1.0, 1.0). The RGB primaries are represented respectively by red (1.0, 0.0, 0.0), green (0.0, 1.0, 0.0), and blue (0.0, 0.0, 1.0); that is, colors that are fully bright in a single primary component and totally dark in the other primaries. Other colors are a mix of the three primaries as needed.

While we say that the real-number representation for color is more device-independent, most graphics hardware deals with colors using integers. Floating-point values are converted to integers to save space and to speed operations, with the exact representation and storage of the integers depending on the number of bits per color per pixel and on other hardware design issues. This distinction sometimes comes up in considering details of color operations in your API, but is generally something that you can ignore. The color-generation process itself is surprisingly complex because the monitor or other viewing device must generate perceptually-linear values, but most hardware generates color with exponential, not linear, properties. All these color issues are hidden from the API programmer, however, and are managed after being translated from the API

representations of the colors, allowing API-based programs to work relatively the same across a wide range of platforms.

In addition to dealing with the color of light, modern graphics systems add a fourth component to the question of color. This fourth component is called “the alpha channel” because that was its original notation [POR], and it represents the opacity of the material that is being modeled. As is the case with color, this is represented by a real number between 0.0 (no opacity — completely transparent) and 1.0 (completely opaque — no transparency). This is used to allow you to create objects that you can see through at some level, and can be a very valuable tool when you want to be able to see more than just the things at the front of a scene. However, transparency is not determined globally by the graphics API; it is determined by compositing the new object with whatever is already present in the Z-buffer. Thus if you want to create an image that contains many levels of transparency, you will need to pay careful attention to the sequence in which you draw your objects, drawing the furthest first in order to get correct attenuation of the colors of background objects.

*Definitions*

The RGB cube

The RGB color model is associated with a geometric presentation of a color space. That space is a cube consisting of all points (r, g, b) with each of r, g, and b having a value that is a real number between 0 and 1. Because of the easy analogy between color triples and space triples, every point in the unit cube can be easily identified with a RGB triple representation of a color. This gives rise to the notion of the RGB color cube that is seen in every graphics text (and thus that we won't repeat here).

To illustrate the numeric properties of the RGB color system, we will create the edges of the color cube as shown in Figure 5.1 below, which has been rotated to illustrate the colors more fully. To do this, we create a small cube with a single color, and then draw a number of these cubes around the edge of the geometric unit cube, with each small cube having a color that matches its location. We see the origin (0,0,0) corner, farthest from the viewer, mostly by its absence because of the black background, and the (1,1,1) corner nearest the viewer as white. The three axis directions are the pure red, green, and blue corners. Creating this figure is discussed below in the section on creating a model with a full spectrum of colors, and it would be useful to add an interior cube within the figure shown that could be moved around the space interactively and would change color to illustrate the color at its current position in the cube.

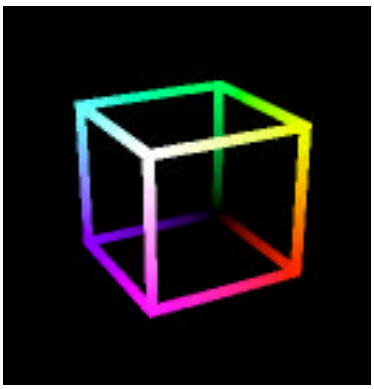


Figure 5.1: tracing the colors of the edges of the RGB cube

This figure suggests the nature of the RGB cube, but a the entire RGB cube is shown from two points of view in Figure 5.2, from the white vertex and from the black vertex, so you can see the full range of colors on the surface of the cube. Note that the three vertices closest to the white vertex are the cyan, magenta, and yellow vertices, while the three vertices closest to the black vertex are the red, green, and blue vertices. This illustrates the additive nature of the RGB color model, with the colors getting lighter as the amounts of the primary colors increase, as well as the subtractive nature of the CMY color model, where the colors get darker as the amounts of color increase. This will be explored later and will be contrasted to the subtractive nature of other color models. Not shown is the center diagonal of the RGB cube from (0, 0, 0) to (1, 1, 1) that corresponds to the colors with equal amounts of each primary; these are the gray colors that provide the neutral backgrounds that are very useful in presenting colorful images.

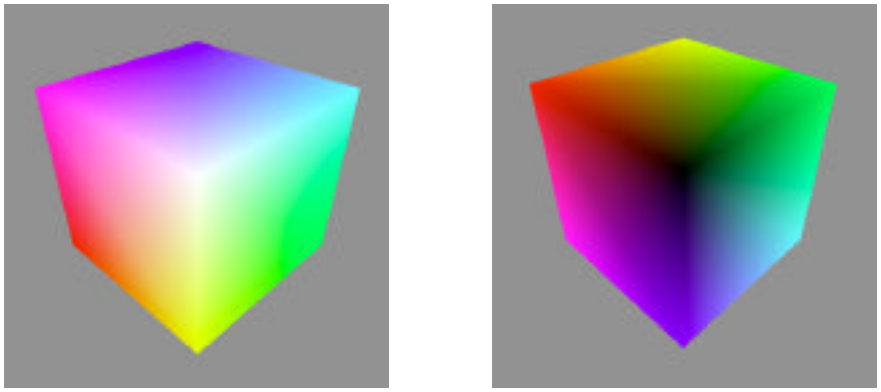


Figure 5.2: two views of the RGB cube — from the white (left) and black (right) corners

Color is ubiquitous in computer graphics, and we can specify color in two ways: by directly setting the color for the objects we are drawing, or by defining properties of object surfaces and lights and having the color generated by a lighting model. In this module we only think about the colors of objects, and save the color of lights and the way light interacts with surfaces for a later module on lighting. In general, the behavior of a scene will reflect both these attributes—if you have a red object and illuminate it with a blue light, your object will seem to be essentially black, because a red object reflects no blue light and the light contains no other color than blue.

Luminance

Luminance of a color is the color's brightness, or the intensity of the light it represents, without regard for its actual color. This concept is particularly meaningful for emissive colors on the screen, because these actually correspond to the amount of light that is emitted from the screen. The concept of luminance is important for several reasons. One is that a number of members of any population have deficiencies in the ability to distinguish different colors, the family of so-called color blindness problems, but are able to distinguish differences in luminance. You need to take luminance into account when you design your displays so that these persons can make sense of them Luminance is also important because part of the interpretation of an image deals with the brightness of its parts, and you need to understand how to be sure that you use colors with the right kind of luminance for your communication. Fore example, in the chapter on visual communication we will see how we can use luminance information to get color scales that are approximately uniform in terms of having the luminance of the color represent the numerical value that the color is to represent.

For RGB images, luminance is quite easy to compute. Of the three primaries, green is the brightest and so contributes most to the luminance of a color. Red is the next brightest, and blue is the least bright. The actual luminance will vary from system to system and even from display

device to display device because of differences in the way color numbers are translated into voltages and because of the way the phosphors respond. In general, though, we are relatively accurate if we assume that luminance is calculated by the formula

```
0.59 * green + 0.30*red + 0.11*blue
```

so the overall brightness ratios are approximately 6:3:1 for green:red:blue.

Other color models

There are times when the RGB model is not easy to use. Few of us think of a particular color in terms of the proportions of red, green, and blue that are needed to create it, so there are other ways to think about color that make this more intuitive. And there are some processes for which the RGB approach does not model the reality of color production. So we need to have a wider range of ways to model color to accomodate these realities.

A more intuitive approach to color is found with either of the HSV (Hue-Saturation-Value) or HLS (Hue-Lightness-Saturation) models. These models represent color as a hue (intuitively, a descriptive variation on a standard color such as red, or magenta, or blue, or cyan, or green, or yellow) that is modified by setting its value (a property of darkness or lightness) and its saturation (a property of brightness). This lets us find numerical ways to say "the color should be a dark, vivid reddish-orange" by using a hue that is to the red side of yellow, has a relatively low value, and has a high saturation.

Just as there is a geometric model for RGB color space, there is one for HSV color space: a cone with a flat top, as shown in Figure 5.3 below. The distance around the circle in degrees represents the hue, starting with red at 0, moving to green at 120, and blue at 240. The distance from the vertical axis to the outside edge represents the saturation, or the amount of the primary colors in the particular color. This varies from 0 at the center (no saturation, which makes no real coloring) to 1 at the edge (fully saturated colors). The vertical axis represents the value, from 0 at the bottom (no color, or black) to 1 at the top. So a HSV color is a triple representing a point in or on the cone, and the "dark, vivid reddish-orange" color would be something like (40.0, 1.0, 0.7). Code to display this geometry interactively is discussed at the end of this chapter, and writing an interactive display program gives a much better view of the space.

The shape of the HSV model space can be a bit confusing. The top surface represents all the lighter colors based on the primaries, because colors getting lighter have the same behavior as colors getting less saturated. The reason the geometric model tapers to a point at the bottom is that there is no real color variation near black. In this model, the gray colors are the colors with a saturation of 0, which form the vertical center line of the cone. For such colors, the hue is meaningless, but it still must be included.

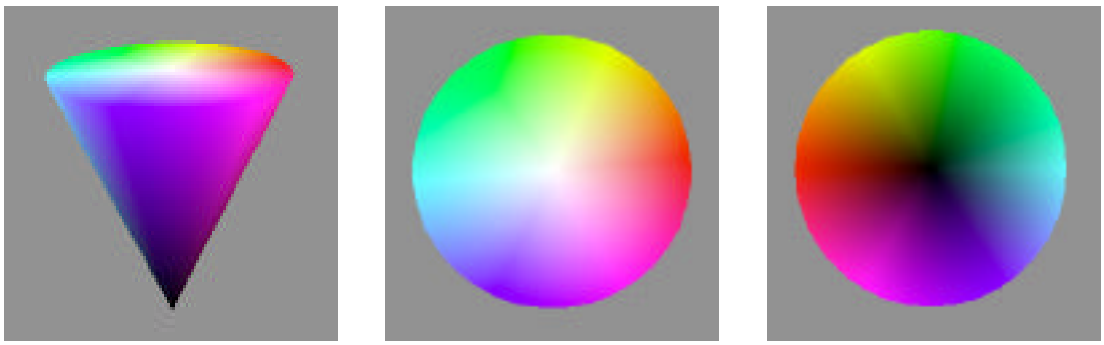


Figure 5.3: three views of HSV color space: side (left), top (middle), bottom (right)

In the HLS color model, shown in Figure 5.4, the geometry is much the same as the HSV model but the top surface is stretched into a second cone. Hue and saturation have the same meaning as HSV but lightness replaces value, and lightness corresponds to the brightest colors at a value of 0.5. The rationale for the dual cone that tapers to a point at the top as well as the bottom is that as colors get lighter, they lose their distinctions of hue and saturation in a way that is very analogous with the way colors behave as they get darker. In some ways, the HLS model seems to come closer to the way people talk about "tints" and "tones" when they talk about paints, with the strongest colors at lightness 0.5 and becoming lighter (tints) as the lightness is increased towards 1.0, and becoming darker (tones) as the lightness is decreased towards 0.0. Just as in the HSV case above, the grays form the center line of the cone with saturation 0, and the hue is meaningless.

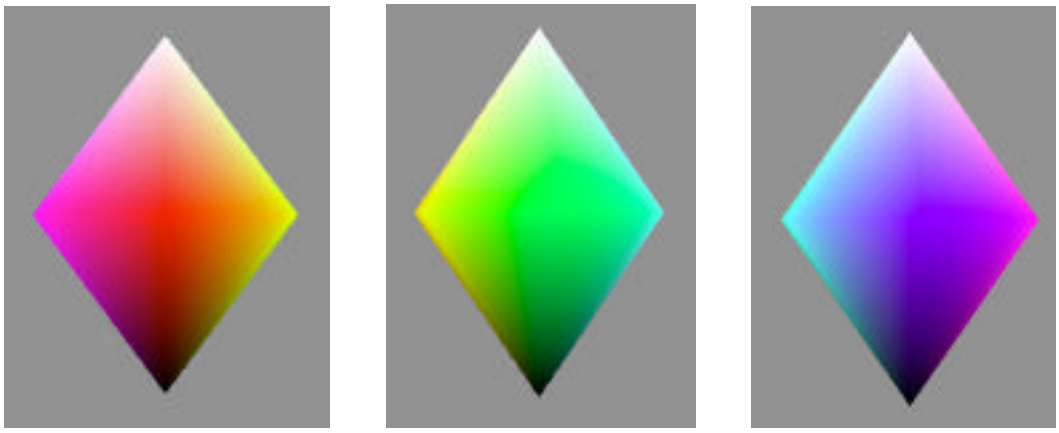


Figure 5.4: the HLS double cone from the red (left), green (middle), and blue(right) directions.

The top and bottom views of the HLS double cone look just like those of the HSV single cone, but the side views of the HLS double cone are quite different. Figure 5.4 shows the HLS double cone from the three primary-color sides: red, green, and blue respectively. The views from the top or bottom are exactly those of the HSV cone and so are now shown here. The images in the figure do not show the geometric shape very well; the discussion of this model in the code section below will show you how this can be presented, and an interactive program to display this space will allow you to interact with the model and see it more effectively in 3-space.

There are relatively simple functions that convert a color defined in one space into the same color as defined in another space. We do not include all these functions in these notes, but they are covered in [FvD], and the functions to convert HSV to RGB and to convert HLS to RGB are included in the code discussions below about producing these figures.

All the color models above are based on colors presented on a computer monitor or other device where light is emitted to the eye. Such colors are called *emissive* colors, and operate by adding light at different wavelengths as different screen cells emit light. The fact that most color presented by programming comes from a screen makes this the primary way we think about color in computer graphics systems. This is not the only way that color is presented to us, however. When you read these pages in print, and not on a screen, the colors you see are generated by light that is reflected from the paper through the inks on the page. Such colors can be called *transmissive* colors and operate by subtracting colors from the light being reflected from the page. This is a totally different process and needs separate treatment. Figure 5.5 illustrates this principle. The way the RGB add to produce CMY and eventually white shows why emissive colors are sometimes called additive colors, while the way CMY produce RGB and eventually black shows why transmissive colors are sometimes called subtractive colors.

Figure 3.5: emissive colors (left) and transmissive colors (right)

Transmissive color processes use inks or films that transmit only certain colors while filtering out all others. Two examples are the primary inks for printing and the films for theater lights; the primary values for transmissive color are cyan (which transmits both blue and green), magenta (which transmits both blue and red), and yellow (which transmits both red and green). In principle, if you use all three inks or filters (cyan, magenta, and yellow), you should have no light transmitted and so you should see only black. In practice, actual materials are not perfect and allow a little off-color light to pass, so this would produce a dark and muddy gray (the thing that printers call "process black") so you need to add an extra "real" black to the parts that are intended to be really black. This cyan-magenta-yellow-black model is called CMYK color and is the basis for printing and other transmissive processes. It is used to create color separations that combine to form full-color images as shown in Figure 5.6, which shows a full-color image (left) and the sets of yellow, cyan, black, and magenta separations (right-hand side, clockwise from top left) that are used to create plates to print the color image. We will not consider the CMYK model further in this discussion because its use is in printing and similar technologies, but not in graphics programming. We will meet this approach to color again when we discuss graphics hardcopy, however.

Figure 5.6: color separations for printing

Color depth

The numerical color models we have discussed above are device-independent; they assume that colors are represented by real numbers and thus that there are an infinite number of colors available to be displayed. This is, of course, an incorrect assumption, because computers lack the capability of any kind of infinite storage. Instead, computers use color capabilities based on the amount of memory allocated to holding color information.

The basic model we usually adopt for computer graphics is based on screen displays and can be called direct color. For each pixel on the screen we store the color information directly in the screen buffer. The number of bits of storage we use for each color primary is called the color depth for our graphics. At the time of this writing, it is probably most common to use eight bits of color for each of the R, G, and B primaries, so we often talk about 24-bit color. In fact, it is not uncommon to include the Z-buffer depth in the color discussion, with the model of RGBA color, and if the system uses an 8-bit Z-buffer we might hear that it has 32-bit color. This is not universal, however; some systems use fewer bits to store colors, and not all systems use an equal number of bits for each color, while some systems use more bits per color. The very highest-end professional graphics systems, for example, often use 36-bit or 48-bit color.

One important effect of color depth is that the exact color determined by your color computations will not be displayed on the screen. Instead, the color is aliased by rounding it to a value that can be represented with the color depth of your system. This can lead to serious effects called *Mach bands*, in which very small differences between adjacent color representations are perceived visually to be significant. Because the human visual system is extremely good at detecting edges, these differences are interpreted as strong edges and disrupt the perception of a smooth image. You should be careful to look for Mach banding in your work, and when you see it, you should try to modify your image to make it less visible. Figure 5.7 shows a small image that contains some Mach bands.

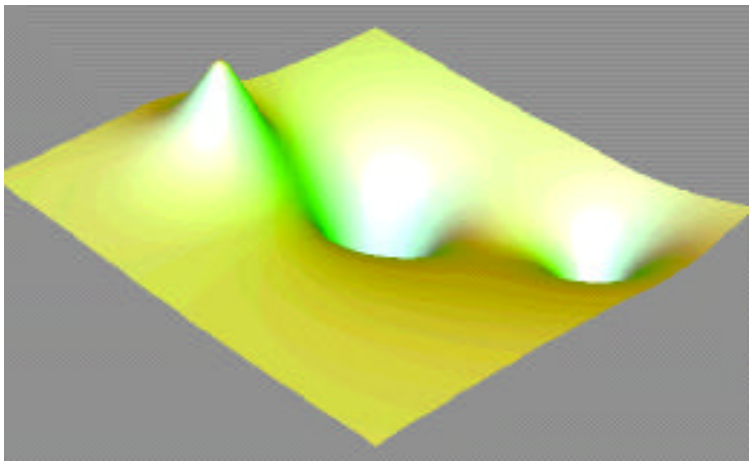


Figure 5.7: an image showing Mach banding

Color gamut

Color is not only limited by the number of colors that can be displayed, but also by limitations in the technology of display devices. No matter what display technology you use—phosphates on a video-based screen, ink on paper, or LCD cells on a flat panel—there is a limited range of color that the technology can present. This is also true of more traditional color technologies, such as color film. The range of a device is called its *color gamut*, and we must realize that the gamut of

our devices limits our ability to represent certain kinds of images. A significant discussion of the color gamut of different devices is beyond the scope of the content we want to include for the first course in computer graphics, but it is important to realize that there are serious limitations on the colors you can produce on most devices.

Color blending with the alpha channel

In most graphics APIs, color can be represented as more than just a RGB triple; it can also include a blending level (sometimes thought of as a transparency level) so that anything with this color will have a color blending property. Thus color is represented by a quadruple (r,g,b,a) and the color model that includes blending is called the RGBA model. The transparency level *a* for a color is called the *alpha value*, and its value is a number between 0.0 and 1.0 that is actually a measure of opacity instead of transparency. That is, if you use standard kinds of blending functions and if the alpha value is 1.0, the color is completely opaque, but in the same situation if the alpha value is 0.0, the color is completely transparent. However, we are using the term "transparent" loosely here, because the real property represented by the alpha channel is *blending*, not transparency. The alpha channel was invented to permit image image compositing [POR] in which an image could be laid over another image and have part of the underlying image show through. So while we may say "transparent" we really mean blended.

This difference between blended and transparent colors can be very significant. If we think of transparent colors, we are modeling the logical equivalent of colored glass. This kind of material embodies transmissive, not emissive, colors — only certain wavelengths are passed through, while the rest are absorbed. But this is not the model that is used for the alpha value; blended colors operate by averaging emissive RGB colors, which is the opposite of the transmissive model implied by transparency. The difference can be important in creating the effects you need in an image. There is an additional issue to blending because averaging colors in RGB space may not result in the intermediate colors you would expect; the RGB color model is one of the worse color models for perceptual blending but we have no real choice in most graphics APIs.

Challenges in blending

The concept of blending is based on the premise that the color of a pixel in a scene can reasonably be represented by a linear combination of the colors of partially-transparent objects that lie under that pixel. For many kinds of objects this is not unreasonable, but the premise is vulnerable to the effects of our RGB color model.

If the RGB color model were uniform, then we would perceive uniform changes in color as we move from one color to another in the visual space. In Figure 5.8 below, we create some color strips with well-known endpoints and ask ourselves if we believe that the color in the middle of the strips really is the average of the colors at the end.

Figure 5.8: the color strips between red and cyan (top) and green and magenta (bottom)

There is no real solution for this problem in the RGB color space or in any system based on it, but the problem can be addressed by doing color interpolation in another color space: converting the two RGB colors to their representations in the other space, interpolating in the other space, and converting the result back to RGB space. Sometimes the HSV or HLS spaces are used for this kind of interpolation, but what is really needed is a perceptually-uniform color space, and these are both difficult to describe simply and usually not available with a simple graphics API.

When we look at each of these color strips, we should consider the luminance of the colors along the strip. In the top strip, the red end has luminance 0.3 while the cyan end has luminance 0.7. but the middle gray point has luminance 0.5. In the bottom strip, the green end has luminance 0.59 while the magenta end has luminance 0.41, and again the middle gray point has luminance 0.5. Thus these strips seems to have a fairly uniform change in luminance. A similar analysis could be made for other kinds of color interpolation.

Another way to consider these color strips, however, is in terms of the amount of color (the chrominance or chroma) they represent. The chroma can be seen as the saturation in the HLS or HSV color space, and in these terms both the endpoints of the color strips have chroma 1 while the middle points have chroma 0.5, so the strips do not interpolate chroma well. Interpolating colors while preserving chroma would require interpolations in another color space, such as HSV or HLS.

Modeling transparency with blending

Blending creates some significant challenges if we want to create the impression of transparency. To begin, we make the simple observation that is something is intended to seem transparent to some degree, you must be able to see things behind it. This suggests a simple first step: if you are working with objects having their alpha color component less than 1.0, it is useful and probably important to allow the drawing of things that might be behind these objects. To do that, you should draw all solid objects (objects with alpha component equal to 1.0) before drawing the things you want to seem transparent, turn off the depth test while drawing items with blended colors, and turn the depth test back on again again after drawing them. This at least allows the possibility that some concept of transparency is allowed.

But it may not be enough to do this, and in fact this attempt at transparency may lead to more confusing images than leaving the depth test intact. Let us consider the case that that you have three objects to draw, and that you will be drawing them in a particular order. For the discussion, let's assume that the objects are numbered 1, 2, and 3, that they have colors C1, C2, and C3, that you draw them in the sequence 1, 2, and 3, that they line up from the eye but have a totally white background behind them, and that each color has alpha = 0.5. Let's assume further that we are not using the depth buffer so that the physical ordering of the objects is not important. The layout is shown in Figure 5.9. And finally, let's further assume that we've specified the blend functions as suggested above, and consider the color that will be drawn to the screen where these objects lie.

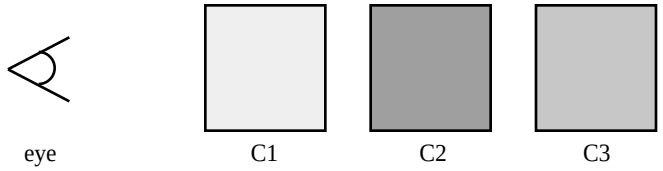



Figure 5.9: the sequence for drawing the objects

When we draw the first object, the frame buffer will have color C1; no other coloring is involved. When we draw the second object on top of the first, the frame buffer will have color

`0.5*C1+0.5*C2`, because the foreground (C2) has alpha 0.5 and the background (C1) is included with weight `0.5 = 1 - 0.5`. Finally, when the third object is drawn on top of the others, the color will be

`0.5*C3+0.5*(0.5*C1+0.5*C2)`, or `0.5*C3+0.25*C2+0.25*C1`.

That is, the color of the most recent object drawn is emphasized much more than the color of the other objects. This shows up clearly in the right-hand part of Figure 5.10 below, where the red square is drawn after the other two squares. On the other hand, if you had drawn object three before object 2, and object 2 before object 1, the color would have been

`0.5*C1+0.25*C2+0.25*C3`,

so the order in which you draw things, not the order in which they are placed in space, determines the color.

But this again emphasizes a difference between blending and transparency. If we were genuinely modeling transparency, it would not make any difference which object were placed first and which last; each would subtract light in a way that is independent of its order. So this represents another challenge if you would want to create an illusion of transparency with more than one non-solid object.

The problem with the approaches above, and with the results shown in Figure 5.10 below, is that the most recently drawn object is not necessarily the object that is nearest the eye. Our model of blending actually works fairly well if the order of drawing is back-to-front in the scene. If we consider the effect of actual partial transparency, we see that the colors of objects farther away from the eye really are of less importance in the final scene than nearer colors. So if we draw the objects in back-to-front order, our blending will model transparency much better. We will address this with an example later in this chapter.

*Some examples*

Example: An object with partially transparent faces

If you were to draw a piece of the standard coordinate planes and to use colors with alpha less than 1.0 for the planes, you would be able to see through each coordinate plane to the other planes as though the planes were made of some partially-transparent plastic. We have modeled a set of three squares, each lying in a coordinate plane and centered at the origin, and each defined as having a rather low alpha value of 0.5 so that the other squares are supposed to show through. In this section we consider the effects of a few different drawing options on this view.

The left-hand side of Figure 5.10 below shows the image we get with these colors when we leave the depth test active. Here what you can actually "see through" depends on the order in which you draw the objects. With the depth test enabled, the presence of a transparent object close to your eye prevents the writing of its blend with an object farther away. Because of this, the first coordinate plane you draw is completely opaque to the other planes, even though we specified it as being partly transparent, and a second coordinate plane allows you to see through it to the first plane but is fully opaque to the second plane. We drew the blue plane first, and it is transparent only to the background (that is, it is darker than it would be because the black background shows through). The green plane was drawn second, and that only allows the blue plane to show through. The red plane was drawn third, and it appears to be fully transparent and show through to both other planes. In the actual working example, you can use keypresses to rotate the planes in space; note that as you do, the squares you see have the same transparency properties in any position.

However, in the image in the center of Figure 5.10, we have disabled the depth test, and this presents a more problematic situation. In this case, the result is something much more like

transparent planes, but the transparency is very confusing because the last plane drawn, the red plane, always seems to be on top because its color is the brightest. This figure shows that the OpenGL attempt at transparency is not necessarily a desirable property; it is quite difficult to get information about the relationship of the planes from this image. Thus one would want to be careful with the images one would create whenever one chose to work with transparent or blended images. This figure is actually created by exactly the same code as the one above with blending disabled instead of enabled.

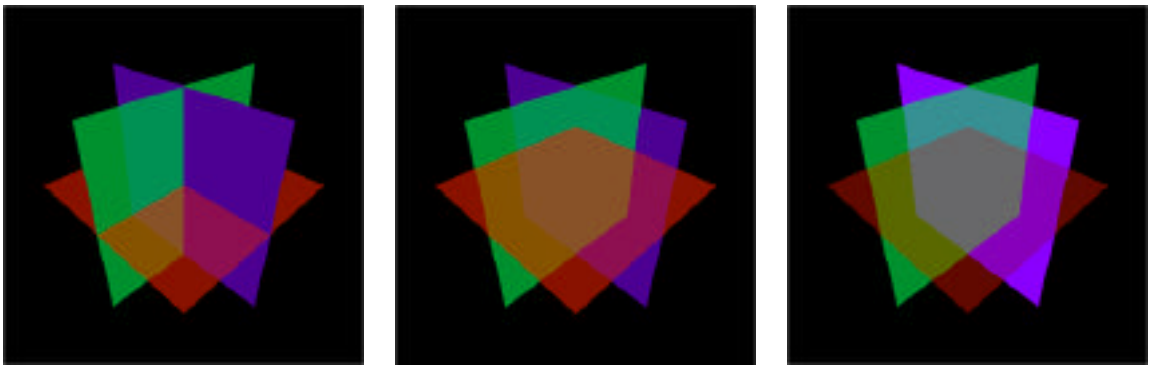


Figure 5.10: the partially transparent coordinates planes (left); the same coordinate planes fully transparent but with same alpha (center); the same coordinate planes with adjusted alpha (right)

Finally, we change the alpha values of the three squares to account for the difference between the weights in the final three-color section. Here we use 1.0 for the first color (blue), 0.5 for the second color (green) but only 0.33 for red, and we see that this final image, the right-hand image in Figure 3.8, has the following color weights in its various regions:

- 0.33 for each of the colors in the shared region,
- 0.5 for each of blue and green in the region they share,
- 0.33 each for red and green in the region they share,
- 0.33 for red and 0.67 for blue in the region they share,
- the original alpha values for the regions where there is only one color.

Note that the "original alpha values" gives us a solid blue, a fairly strong green, and a weak read as stand-alone colors. This gives us a closer approximation to the appearance actual transparency for these three colors, with a particular attention to the clear gray in the area they all cover, but there are still some areas that don't quite work. To get even this close, however, we must analyze the rendering carefully and we still cannot quite get a perfect appearance.

Let's look at this example again from the point of view of depth-sorting the things we will draw. In this case, the three planes intersect each other and must be subdivided into four pieces each so that there is no overlap. Because there is no overlap of the parts, we can sort them so that the pieces farther from the eye will be drawn first. This allows us to draw in back-to-front order, where the blending provides a better model of how transparency operates. Figure 5.11 shows how this would work. The technique of adjusting your model is not always as easy as this, because it can be difficult to subdivide parts of a figure, but this shows its effectiveness.

There is another issue with depth-first drawing, however. If you are creating a scene that permits the user either to rotate the eye point around your model or to rotate parts of your model, then the model will not always have the same parts closest to the eye. In this case, you will need to use a feature of your graphics API to identify the distance of each part from the eyepoint. This is usually done by rendering a point in each part in the background and getting its Z-value with the current eye point. This is a more advanced operation than we are now ready to discuss, so we refer you to the manuals for your API to see if it is supported and, if so, how it works.

Figure 5.11: the partially transparent planes broken into quadrants and drawn back-to-front

As you examine this figure, note that although each of the three planes has the same alpha value of 0.5, the difference in luminance between the green and blue colors is apparent in the way the plane with the green in front looks different from the plane with the blue (or the red, for that matter) in front. This goes back to the difference in luminance between colors that we discussed earlier in the chapter.

*Color in OpenGL*

OpenGL uses the RGB and RGBA color models with real-valued components. These colors follow the RGB discussion above very closely, so there is little need for any special comments on color itself in OpenGL. Instead, we will discuss blending in OpenGL and then will give some examples of code that uses color for its effects.

Enabling blending

In order to use colors from the RGBA model, you must specify that you want the blending enabled and you must identify the way the color of the object you are drawing will be blended with the color that has already been defined. This is done with two simple function calls:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

The first is a case of the general enabling concept for OpenGL; the system has many possible capabilities and you can select those you want by enabling them. This allows your program to be more efficient by keeping it from having to carry out all the possible operations in the rendering pipeline. The second allows you to specify how you want the color of the object you are drawing to be blended with the color that has already been specified for each pixel. If you use this blending function and your object has an alpha value of 0.7, for example, then the color of a pixel after it has been drawn for your object would be 70% the color of the new object and 30% the color of whatever had been drawn up to that point.

There are many options for the OpenGL blending function. The one above is the most commonly used and simply computes a weighted average of the foreground and background colors, where the weight is the alpha value of the foreground color. In general, the format for the blending specification is

```
glBlendFunc(src, dest)
```

and there are many symbolic options for the source (src) and destination (dest) blending values; the OpenGL manual covers them all.

*A word to the wise...*

You must always keep in mind that the alpha value represents a blending proportion, not transparency. This blending is applied by comparing the color of an object with the current color of the image at a pixel, and coloring the pixel by blending the current color and the new color according to one of several rules that you can choose with `glBlendFunc(...)` as noted above. This capability allows you to build up an image by blending the colors of parts in the order in which they are rendered, and we saw that the results can be quite different if objects are received in a different order. Blending does not treat parts as being transparent, and so there are some images where OpenGL simply does not blend colors in the way you expect from the concept.

If you really do want to try to achieve the illusion of full transparency, you are going to have to do some extra work. You will need to be sure that you draw the items in your image starting with the item at the very back and proceeding to the frontmost item. This process was described in the second example above and is sometimes called Z-sorting. It can be very tricky because objects can overlap or the sequence of objects in space can change as you apply various transformations to the scene. You may have to re-structure your modeling in order to make Z-sorting work. In the example above, the squares actually intersect each other and could only be sorted if each were broken down into four separate sub-squares. And even if you can get the objects sorted once, the order would change if you rotated the overall image in 3-space, so you would possibly have to re-sort the objects after each rotation. In other words, this would be difficult.

As always, when you use color you must consider carefully the information it is to convey. Color is critical to convey the relation between a synthetic image and the real thing the image is to portray, of course, but it can be used in many more ways. One of the most important is to convey the value of some property associated with the image itself. As an example, the image can be of some kind of space (such as interstellar space) and the color can be the value of something that occupies that space or happens in that space (such as jets of gas emitted from astronomical objects where the value is the speed or temperature of that gas). Or the image can be a surface such as an airfoil (an airplane wing) and the color can be the air pressure at each point on that airfoil. Color can even be used for displays in a way that carries no meaning in itself but is used to support the presentation, as in the Chromadepth™ display we will discuss below in the texture mapping module. But never use color without understanding the way it will further the message you intend in your image.

*Code examples*

A model with parts having a full spectrum of colors

The code that draws the edges of the RGB cube uses translation techniques to create a number of small cubes that make up the edges. In this code, we use only a simple cube we defined ourselves (not that it was too difficult!) to draw each cube, setting its color by its location in the space:

```
typedef GLfloat color [4];
color cubecolor;

cubecolor[0] = r; cubecolor[1] = g; cubecolor[2] = b;
cubecolor[3] = 1.0;
glColor4fv(cubecolor);
```

We only use the cube we defined, which is a cube whose sides all have length two and which is centered on the origin. However, we don't change the geometry of the cube before we draw it. Our technique is to use transformations to define the size and location of each of the cubes, with scaling to define the size of the cube and translation to define the position of the cube, as follows:

```
glPushMatrix();
glScalef(scale,scale,scale);
glTranslatef(-SIZE+(float)i*2.0*scale*SIZE,SIZE,SIZE);
```

```
      cube((float)i/(float)NUMSTEPS,1.0,1.0);
      glPopMatrix();
```

Note that we include the transformation stack technique of pushing the current modeling transformation onto the current transformation stack, applying the translation and scaling transformations to the transformation in use, drawing the cube, and then popping the current transformation stack to restore the previous modeling transformation. This was discussed earlier in the chapter on modeling.

The HSV cone

There are two functions of interest here. The first is the conversion from HSV colors to RGB colors; this is taken from [FvD] as indicated, and is based upon a geometric relationship between the cone and the cube, which is much clearer if you look at the cube along a diagonal between two opposite vertices. The second function does the actual drawing of the cone with colors generally defined in HSV and converted to RGB for display, and with color smoothing handling most of the problem of shading the cone. For each vertex, the color of the vertex is specified before the vertex coordinates, allowing smooth shading to give the effect in Figure 5.3. For more on smooth shading, see the later chapter on the topic.

```
void convertHSV2RGB(float h,float s,float v,float *r,float *g,float *b)
{
// conversion from Foley et.al., fig. 13.34, p. 593
   float f, p, q, t;
   int   k;

   if (s == 0.0) {   // achromatic case
      *r = *g = *b = v;
   }
   else {   // chromatic case
      if (h == 360.0) h=0.0;
      h = h/60.0;
      k = (int)h;
      f = h - (float)k;
      p = v * (1.0 - s);
      q = v * (1.0 - (s * f));
      t = v * (1.0 - (s * (1.0 - f)));
      switch (k) {
         case 0: *r = v; *g = t; *b = p; break;
         case 1: *r = q; *g = v; *b = p; break;
         case 2: *r = p; *g = v; *b = t; break;
         case 3: *r = p; *g = q; *b = v; break;
         case 4: *r = t; *g = p; *b = v; break;
         case 5: *r = v; *g = p; *b = q; break;
      }
   }
}

void HSV(void)
{
#define NSTEPS 36
#define steps (float)NSTEPS
#define TWOPI 6.28318

   int i;
   float r, g, b;

   glBegin(GL_TRIANGLE_FAN);     //    cone of the HSV space
```

```
      glColor3f(0.0, 0.0, 0.0);
      glVertex3f(0.0, 0.0, -2.0);
      for (i=0; i<=NSTEPS; i++) {
         convert(360.0*(float)i/steps, 1.0, 1.0, &r, &g, &b);
         glColor3f(r, g, b);
         glVertex3f(2.0*cos(TWOPI*(float)i/steps),
                    2.0*sin(TWOPI*(float)i/steps),2.0);
      }
   glEnd();
   glBegin(GL_TRIANGLE_FAN);     //    top plane of the HSV space
      glColor3f(1.0, 1.0, 1.0);
      glVertex3f(0.0, 0.0, 2.0);
      for (i=0; i<=NSTEPS; i++) {
         convert(360.0*(float)i/steps, 1.0, 1.0, &r, &g, &b);
         glColor3f(r, g, b);
         glVertex3f(2.0*cos(TWOPI*(float)i/steps),
                    2.0*sin(TWOPI*(float)i/steps),2.0);
      }
   glEnd();
}
```

The HLS double cone

The conversion itself takes two functions, while the function to display the double cone is so close to that for the HSV model that we do not include it here. The source of the conversion functions is again Foley et al. This code was used to produce the images in Figure 5.4.

```
void convertHLS2RGB(float h,float l,float s,float *r,float *g,float *b) {
// conversion from Foley et.al., Figure 13.37, page 596
   float m1, m2;

   if (l <= 0.5) m2 = l*(1.0+s);
   else          m2 = l + s - l*s;
   m1 = 2.0*l - m2;
   if (s == 0.0) {   // achromatic cast
      *r = *g = *b = l;
   }
   else {   // chromatic case
      *r = value(m1, m2, h+120.0);
      *g = value(m1, m2, h);
      *b = value(m1, m2, h-120.0);
   }
}

float value( float n1, float n2, float hue) {
// helper function for the HLS->RGB conversion
   if (hue > 360.0) hue -= 360.0;
   if (hue < 0.0)   hue += 360.0;
   if (hue < 60.0)  return( n1 + (n2 - n1)*hue/60.0 );
   if (hue < 180.0) return( n2 );
   if (hue < 240.0) return( n1 + (n2 - n1)*(240.0 - hue)/60.0 );
   return( n1 );
}
```

An object with partially transparent faces

The code that draws three squares in space, each centered at the origin and lying within one of the coordinate planes, has a few points that are worth noting. These three squares are colored according to the declaration:

```
GLfloat color0[]={1.0, 0.0, 0.0, 0.5}, // R
        color1[]={0.0, 1.0, 0.0, 0.5}, // G
        color2[]={0.0, 0.0, 1.0, 0.5}; // B
```

These colors are the full red, green, and blue colors with a 0.5 alpha value, so when each square is drawn it uses 50% of the background color and 50% of the square color. You will see that blending in Figure 5.10 for this example.

The geometry for each of the planes is defined as an array of points, each of which is, in turn, an array of real numbers:

```
typedef GLfloat point3[3];
point3 plane0[4]={{-1.0, 0.0, -1.0},   // X-Z plane
                  {-1.0, 0.0,  1.0},
                  { 1.0, 0.0,  1.0},
                  { 1.0, 0.0, -1.0} };
```

As we saw in the example above, the color of each part is specified just as the part is drawn:

```
glColor4fv(color0);           // red
glBegin(GL_QUADS);            // X-Z plane
      glVertex3fv(plane0[0]);
      glVertex3fv(plane0[1]);
      glVertex3fv(plane0[2]);
      glVertex3fv(plane0[3]);
glEnd();
```

This is not necessary if many of the parts had the same color; once a color is specified, it is used for anything that is drawn until the color is changed.

# Visual Communication

*Prerequisites*

Understanding of basic concepts of communication, of shaping information with the knowledge of the audience and with certain goals

*Introduction*

Computer graphics has achieved remarkable things in communicating information to specialists, to informed communities, and to the public at large. This is different from the entertainment areas where computer graphics gets a lot of press because it has the goal of helping the user of the interactive system or the viewer of a well-developed presentation to have a deeper understanding of a complex topic. The range of subjects for this communication include cosmology, in showing how fundamentatal structures in the universe work; archaeology and anthropology, in showing the ways earlier human groups laid out their structures and cultures; biology and chemistry, in seeing the way electrostatic forces and molecular structures lead to molecular bonding; mathematics, in considering the behavior of highly unstable differential equations; or meteorology, in examining the way global forces such as the temperatures of ocean currents or the depth of the ozone layer affect the weather.

While the importance of visual communication has been known for a long time, its role in the use of computing in the sciences was highlighted in the 1987 report on Visualization in Scientific Computing [ViSC]. That report noted the importance of computer graphics in engaging the human brain's extraordinary ability to create insight from images. That report noted that Richard Hamming's 1962 quote, "The purpose if computing is insight, not numbers," is particularly apropos when the computing can create images that lead to a deeper and more subtle insight than is possible without images. Indeed, for these notes we can parahprase Hamming and say that the purpose of computer graphics is information, not images.

Making images — in particular, making images with computer graphics using powerful machines and a capable graphics API — is relatively easy. The difficult part of effective computer graphics is the task of understanding your problem and developing ways to present the information that describes your problem so you can make images that communicate with your audience. This short section talks about this task and gives some principles and examples that we hope can start you thinking about this question, but it is a significant task to develop real skill in communicating by means of images. This chapter is relatively early in the overall presentation of graphics primarily to remind you that the main reason we use graphics is to communicate with others, and to help you keep that communication in mind as you learn about making images with computing. Some of the techniques we talk about here will not be covered until later in the notes, but they are not terribly complex, so you should be able to make sense of what the techniques mean even before you have learned how to make them work.

There are several key concepts in the area of communicating effectively with your images. In this chapter we will discuss several techniques and will consider their effect upon communication, but you must realize that this is only an introduction to the topic. Highly-skilled communicators are constantly inventing new ways to present specific information to specific audiences, so we do not try to give you the last answer in visual communication; instead, we are trying to get you to think about the information content of your images and about how you can communicate that to your particular audience. Only a great deal of experience and observation will make you genuinely skilled in this area.

The particular points we will make in this chapter cover several areas. These include:

- The important thing in an image is the information it conveys, not the beauty or sophistication of the image itself.

- Focus your image on the content you want so you can provide the information that is needed for your audience to understand the ideas you are trying to communicate.
- Use appropriate representation for your information so that your audience will be able to get the most meaning from your images.
- Use appropriate forms for your information to reach your audience at the right impact level.
- Be very careful to be accurate and not to suggest things with your images that are not supported by the information you are working with.
- If you use pseudocolor to carry information to your audience, use appropriate color ramps for the image and always include a legend so your audience can understand the values that are represented by the colors and other textual material to help your audience understand the content of your display.
- If you use shapes or geometry to carry your information, be sure the geometry supports the understanding you want to create for your information.

These points will be presented in several sections that will focus on creating images that include these communications concepts.

*General issues in visual communication*

Some of the points above are so important that we want to expand them and show some of the issues that are involved in carrying them out.

Use appropriate representation for your information so that your audience will be able to get the most meaning from your images. Sometimes this representation can use color, or sometimes it can use geometry or shapes. Sometimes it will use highly symbolic or synthetic images while sometimes it will use highly naturalistic images. Sometimes it will present the relationships between things instead of the things themselves. Sometimes it will use purely two-dimensional representations, sometimes three-dimensional images but with the third dimension used only for impact, and sometimes three-dimensional images with the third dimension a critical part of the presentation. In fact, there are an enormous number of ways to create representations of information, and the best way to know what works for your audience is probably to to observe the way they are used to seeing things and ask them what makes sense for them, probably by showing them many examples of options and alternatives. Do not assume that you can know what they should use, however, because you probably think differently from people in their field and are probably not the one who needs to get the information from the images.

Keep your images focused on just the information that is needed to understand the things you are trying to communicate. In creating the focus you need, remember that simple images create focus by eliminating extraneous or distracting content. Don't create images that are "eye candy" and simply look good; don't create images that suggest relationships or information that are not in the information. For example, when you represent experimental data with geometric figures, use flat shading instead of smooth shading and use only the resolution your data supports because creating higher resolution with smooth interpolation processes, because using smooth shading or smooth interpolation suggests that you know more than your data supports. The fundamental principle is to be very careful not to distort the truth of your information in order to create a more attractive image.

Use appropriate forms for your information. There is a wonderful concept of three levels of information polishing: for yourself (personal), for your colleagues or collaborators (peer), and for an audience when you want to make an impression (presentation). Most of the time when you're trying to understand something yourself, you can use very simple images because you know what you are trying to show with them. When you are sharing your work with your colleagues who have an idea of what you're working on but who don't have the depth of knowledge in the particular problem you're addressing, you might want a bit higher quality to help them see your point, but you don't need to spend a lot of time polishing your work. But when you are creating a public presentation such as a scientific paper or a grant proposal (think of how you would get a point across to a Congressional committee, for example!) you will need to make your work as

highly-polished as you can. So your work will sometimes be simple and low-resolution, with very sketchy images; sometimes smoother and with a little thought to how your look at things, perhaps with a little simple animation or with some interaction to let people play with your ideas; and sometimes fully developed, with very smooth animation and high-resolution images, with great care taken to make the maximum impact in the minimum time.

Be very careful to be accurate with your display. I you have only scattered data, show only the data you have; do not use smooth geometry or smooth coloring to suggest that information is known between the data points. If you use simple numerical techniques, such as difference equations, in determining your geometry, say this in a legend or title instead of implying that a more exact solution is presented. In general, try very hard not to lie with your presentation, whether that lie is an accident or is an attempt to spin your data to support a particular point of view.

This chapter will describe a small set of techniques for visual communication, including the use of color in communicating values as well as shapes to the viewer, the use of text in labels and legends to set the context and convey detailed information on the image, and the uses of slices, contours, and other techniques in communicating higher-dimensional information. This is only a small sample of the kinds of techniques that have been used for visual communication, and the reader is referred to the literature on scientific visualization for a more extensive discussion of the topic.

*Some examples*

Different ways to encode of information

Among all the ways we can encode information visually, two examples are *geometry* and *color*. We can make the size of something in an image vary with the value associated with that thing or we may make its shape represent a particular concept. We can also make the color of something vary with a value we want to present. Colors that are used as encoded information are often called *pseudocolors*. The example presented below, showing how heat flows through a metallic bar, compares two very different ways of encoding information in images.

The three-part Figure 4.1 below illustrates some different color encodings for a single problem. It has both geometric and pseudocolor encoding (center), only geometric coding (left) and only pseudocolor encoding (right). Note that each of the encodings has its value, but that they

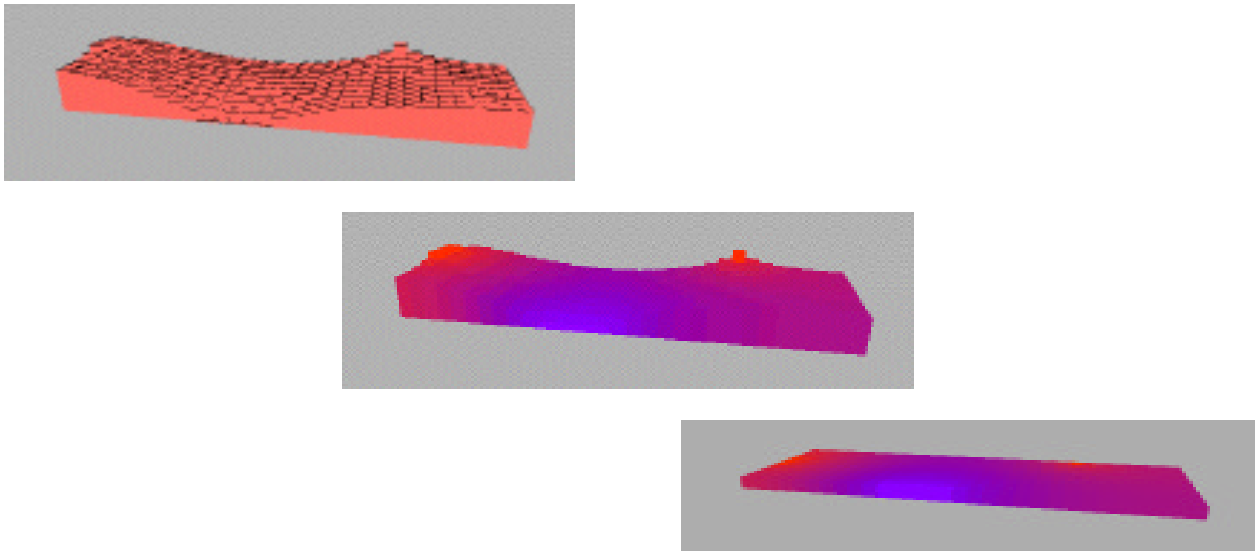


Figure 4.1: three encodings of the same information: temperature in a bar, encoded only as geometry (top left), only as color (bottom right), and as both (center)

emphasize different things. In particular, the height encoding tends to imply that the bar itself actually has a different geometry than a simple rectangular parallelpiped, so it might confuse someone who is not used to the substitution of geometry for numeric values. The color-only encoding, however, seems easier for a novice to understand because we are used to pseudocolor coding for heat (think of hot and cold water faucets) and metal changes colors when it is heated. Thus the way we encode information may depend on the experience of our users and on the conventions they are accustomed to in understanding information. The pseudocolor encoding in this case follows a traditional blue for cold and red for hot that is common in western cultures (think of the coding on hot and cold water faucets). Other encodings are found in other groups and could be used here; for example, engineers often use a full rainbow code for colors from magenta for cold, through blue to green and yellow, finally ending with red for hot.

Different color encodings for information

If you use color to carry information to your audience, use color that is appropriate for the information in the image. For pseudo-color images, make the colors fit the content. Note the use of the blue-to-red scale to represent temperature in the first example below and the several color ramps that can represent numeric values in the second example. The effective and informative choice of color representations for numeric values is a skill that can require a great deal of experience and a good knowledge of your users. In particular, you should realize that colors are always seen in a cultural context, so that engineers and life scientists are likely to read a set of colors differently, and this is particularly important to understand when looking at an international audience for your work. In some fields, some information has a standard color encoding (for example, the colors of atoms in a molecular display) so that you cannot change it without risking losing information your audience will expect to see.

The key concept for this kind of color encoding is that of *pseudocolor* — color that is determined by some property of the object instead of from geometric lighting models. Such color can be used to show a number of different kinds of properties, such as temperatures in the example above. It is common to think of pseudocolors in terms of color ramps, or colors that vary with a linear value. We will show some examples of different color ramps in this section.

Let's consider an example that we will use to look at a number of color and pseudocolor options. We want to create the graph of a function of two variables over a domain of the real plane by presenting a surface in three dimensions. First, as shown in Figure 4.2, we could present the function by presenting its 3D surface graph purely as a surface in a fairly traditional way, with an emphasis of the shape of the surface itself. If the emphasis is on the surface itself, this might be a good way to present the graph, because as you can see, the lighting shows the shape well.

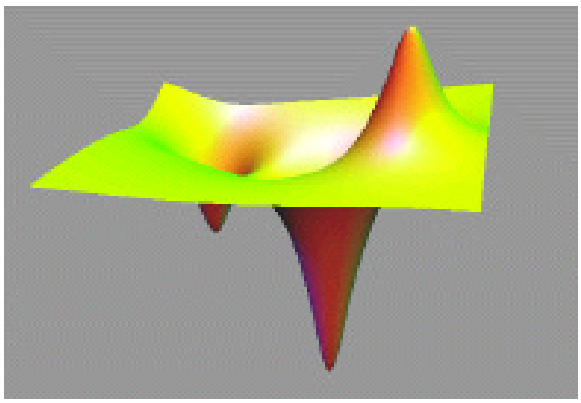


Figure 4.2: traditional surface model presented with three lights to show its shape

On the other hand, if the emphasis is on the actual values of the function, we might use a pseudocolor encoding with the colors chosen based on the value of the function, as shown in Figure 4.3. The particular pseudocolor used is the "rainbow ramp" that runs from purple (usually used for the lowest values) to red (usually used for the highest). This ramp is actually built from the HLS color model to ensure fully-saturated colors. The geometry-only encoding presents a yellow surface with lights of different colors that help illustrate the shape, and the color ramp adds color information to geometric information, creating a color encoding that reinforces the geometric encoding. In this case it might be important to allow the audience to move around the surface so they can see exactly where the various values are achieved, and it is usually very important to include a legend that interprets the colors to the audience. We will discuss legends later in this chapter.

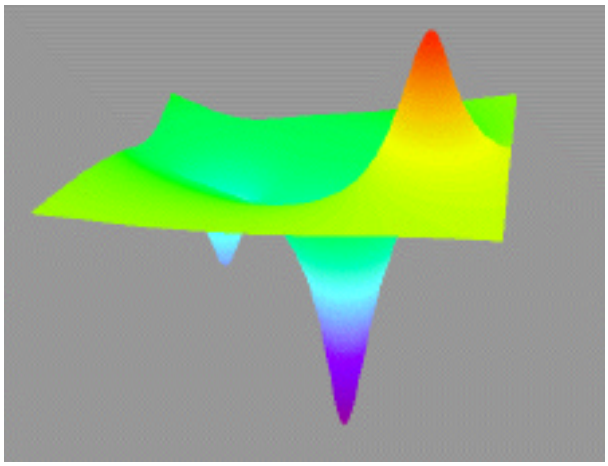


Figure 4.3: surface model presented with "rainbow" pseudocolor ramp to emphasize its values

Of course, the rainbow pseudocolor ramp is not our only choice of a pseudocolor presentation, and Figure 4.4 shows another presentation based on a ramp from black to white through red and yellow. The colors in the ramp could be evenly distributed through the relative heights from zero (the smallest value, in black) to one (the largest value, in white), by moving along the edges of the RGB cube with the lowest third of the values lying from black to red, the next third of the values

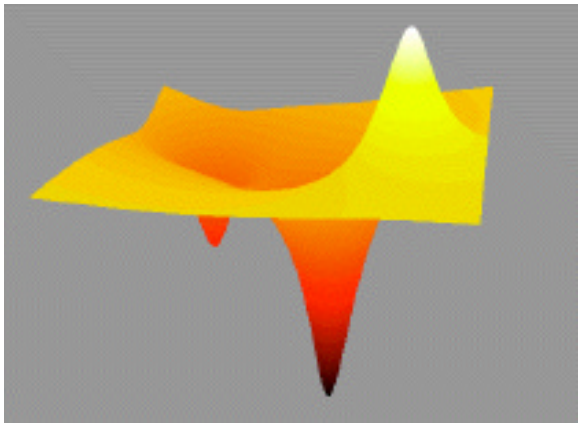


Figure 4.4: the function surface with a uniform luminance distribution of pseudocolors

lying from red to yellow, and the highest third of the values lying from yellow to white. But if we introduce a new idea, that of creating a color ramp that varies the luminance of the colors uniformly

across our range, we have a different approach. Recalling from our earlier discussion of the RGB color system that the luminance of a color is given by the luminance equation:

```
luminance = 0.30*red + 0.59*green + 0.11*blue
```

(where red, green, and blue have the usual convention of lying between zero and one) we can set up our ramp to give the lowest 30% of the values between black and red, the next 59% of the colors between red and yellow, and the hightest 11% of the values between yellow and white. The actual color ramp functions for these latter three are given as code fragments below. This particular pseudocolor ramp is sometimes called a "heat" ramp because it mimics the way a metal will change colors as it is heated and seems to be particularly good in showing the shape in this example.

Each of these images presents the same concepts, but each does so in a way that might be meaningful to a different group of viewers. The pure shape with lighting emphasizes the shape and shows values such as curvature that emphasizes the way the graph changes in different areas. The rainbow pseudocolor ramp emphasizes the individual values by providing as wide a range of different colors as possible to show values and would be commonly seen in science and engineering areas. The heat pseudocolor ramp emphasizes the relative values of the function by reflecting them in the relative luminance of the colors. The reader might find it interesting to modify the code of the heat ramp to take other paths from the black vertex of the RGB cube to the white vertex, while adapting the points where the paths along the cube change to the luminance values associated with the R/G/B/C/M/Y points on the cube.

It is important to note that color may not be a good choice of representation for some audiences. There are significant numbers of persons who have color perception problems, so information that is encoded through color may not be understood by these people. Almost all color deficiencies do not affect the ability to perceive relative luminance, however, so using a uniform luminance ramp will at least allow this part of your audience to understand your information.

There are, of course many alternatives to the even pseudocolor ramps above. One might be a multi-ramp color set, where the color range consists of a set of ramps from black to the rainbow colors, as shown in the left-hand side of Figure 4.5 below, or you may entirely omit color by using a grayscale pseudocolor ramp as in the right-hand side of the figure, although you should note that reduced bit depth gives us some Mach banding. The reason we pay so much attention to the way information can be encoded in color is that this is a critical issue in allowing your audience to understand the information you are presenting with this color.

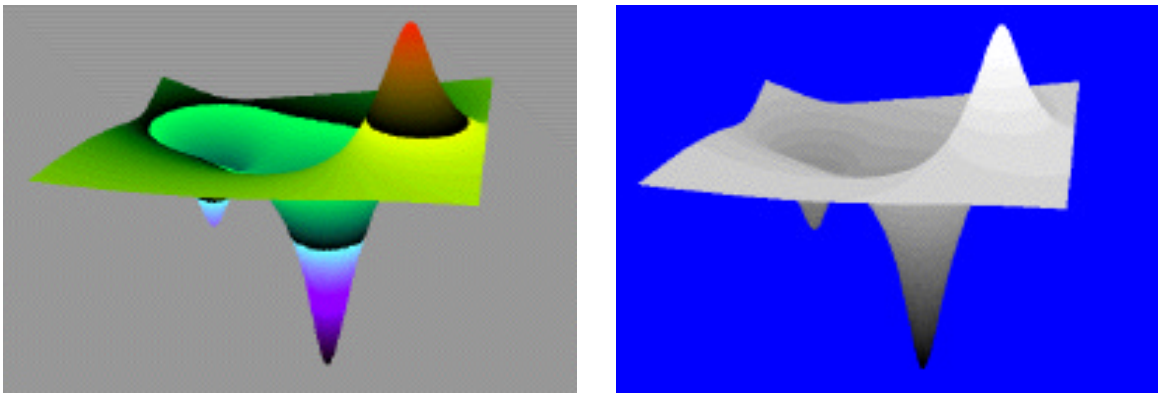


Figure 4.5: the same surface with a multi-ramp color model (left)
and with a pure grayscale color ramp (right)

Code to implement the various color ramps is fairly straightforward but is included here so you can see how these examples were done and adapt these ramps (or similar ones) to your own projects. We present code for the rainbow and heat color ramps below. Each color ramp takes a value between 0 and 1 (it is assumed that the numerical values have been scaled to this range) and returns an array of three numbers that represents the RGB color that corresponds to that value according to

the particular representation is uses. This code is independent of the graphics API you are using, so long as the API uses the RGB color model.

```
void calcHeat(float yval)
{  if (yval < 0.30)
      {myColor[0]=yval/0.3;myColor[1]=0.0;myColor[2]=0.0;return;}
   if ((yval>=0.30) && (yval < 0.89))
      {myColor[0]=1.0;myColor[1]=(yval-0.3)/0.59;myColor[2]=0.0;return;}
   if (yval>=0.89)
      {myColor[0]=1.0;myColor[1]=1.0;myColor[2]=(yval-0.89)/0.11;}
   return;
   }

void calcRainbow(float yval)
{  if (yval < 0.2)                          // purple to blue ramp
      {myColor[0]=0.5*(1.0-yval/0.2);myColor[1]=0.0;
       myColor[2]=0.5+(0.5*yval/0.2);return;}
   if ((yval >= 0.2) && (yval < 0.40))     // blue to cyan ramp
      {myColor[0]=0.0;myColor[1]=(yval-0.2)*5.0;myColor[2]=1.0;return;}
   if ((yval >= 0.40) && (yval < 0.6))     // cyan to green ramp
      {myColor[0]=0.0;myColor[1]=1.0;myColor[2]=(0.6-yval)*5.0;return;}
   if ((yval >= 0.6) && (yval < 0.8 )      // green to yellow ramp
      {myColor[0]=(yval-0.6)*5.0;myColor[1]=1.0;myColor[2]=0.0;return;}
   if (yval >= 0.8)                        // yellow to red ramp^
      {myColor[0]=1.0;myColor[1]=(1.0-yval)*5.0;myColor[2]=0.0;}
   return;
}
```

Geometric encoding of information

If you use shapes or geometry to carry your information, be sure the geometry represents the information in a way that supports the understanding you want to create. Changing sizes of objects can illustrate different values of a quantity, but sizes may be perceived in one dimension (such as height), in two dimensions (such as area), or in three dimensions (such as volume). If you double each of the dimensions of an object, then, your audience may perceive that the change represents a doubling of a value, multiplying the value by four, or multiplying the value by eight, depending on whether they see the difference in one, two, or three dimensions. The shapes can also be presented through pure color or with lighting, the lighting can include flat shading or smooth shading, and the shapes can be presented with meaningful colors or with scene enhancements such as texture mapping; these all affect the way the shapes are perceived, with more sophisticated presentation techniques moving the audience away from abstract perceptions towards a perception that somehow the shapes reflect a meaningful reality.

For example, we should not assume that a 3D presentation is necessarily best for a problem such as the surface above. In fact, real-valued functions of two variables have been presented in 2D for years with color representing the value of the function at each point of its domain. You can see this later in this chapter where we discuss the representation of complex-valued or vector-valued functions. In the present example, in Figure 4.6 we show the same surface we have been considering as a simple surface with the addition of a plane on which we provide a rainbow pseudocolor representation of the height of the function. We also include a set of coordinate axes so that the geometric representation has a value context. As we did in Figure 4.1, we should consider the differences between the color and the height encodings to see which really conveys information better.

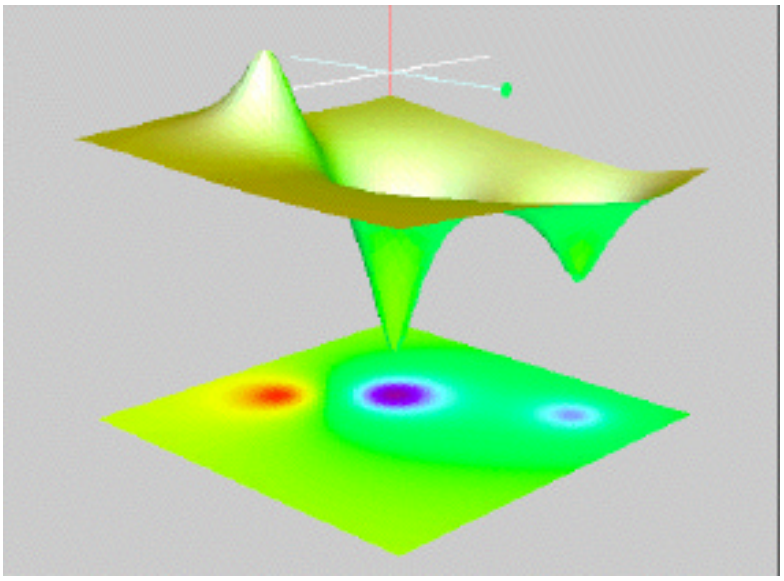


Figure 4.6: a pseudocolor plane with the lighted surface

Other encodings

Surfaces and colorings as described above work well when you are thinking of processes or functions that operate in 2D space. Here you can associate the information at each point with a third dimension or with a color at the point. However, when you get into processes in 3D space, when you think of processes that produce 2D information in 2D space, or when you get into any other areas where you exceed the ability to illustrate information in 3D space, you must find other ways to describe your information.

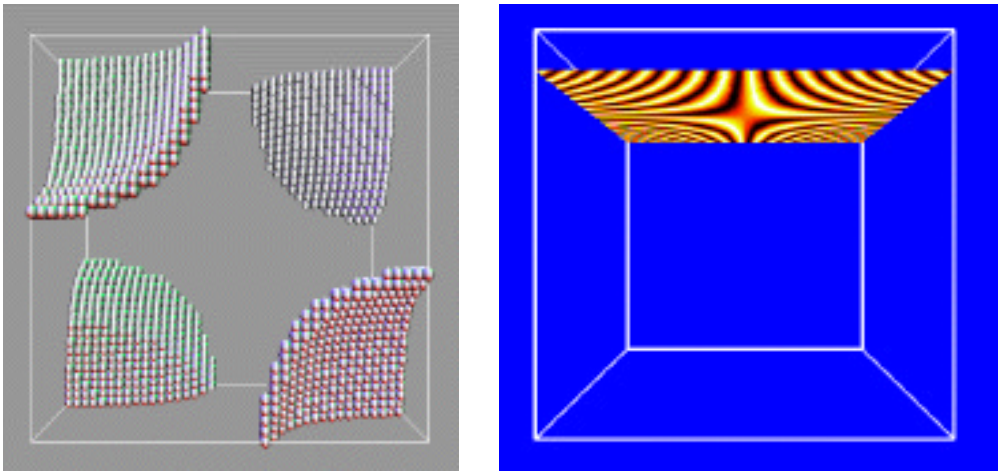


Figure 4.7: a fairly simple isosurface of a function of three variables (left); values of a function in 3D space viewed along a 2D plane in the space (right)

Perhaps the simplest higher-dimensional situation is to consider a process or function that operates in 3D space and has a simple real value. This could be a process that produces a value at each point in space, such as temperature. There are two simple ways to look at such a situation. The first asks "for what points in space does this function have a constant value?" This leads to what are called *isosurfaces* in the space, and there are complex algorithms for finding isosurfaces or volume data or of functions of three variables. The left-hand partof Figure 4.7 shows a simple approach to the problem, where the space is divided into a number of small cubic cells and the

function is evaluated at each vertex on each cell. If the cell has some vertices where the value of the function is larger than the constant value and some vertices where the function is smaller, the continuity of the function assures that the function assumes the constant value somewhere in that cell and a sphere is drawn in each such cell. The second way to look at the situation asks for the values of the function in some 2D subset of the 3D space, typically a plane. For this, we can pass a plane through the 3D space, measure the values of the function in that plane, and plot those values as colors on the plane displayed in space. The right-hand part of Figure 4.7 below, from the code in `planeVolume.c`, shows an example of such a plane-in-space display for a function that is hyperbolic in all three of the x, y, and z components in space. The pseudocolor coding is the uniform ramp illustrated above.

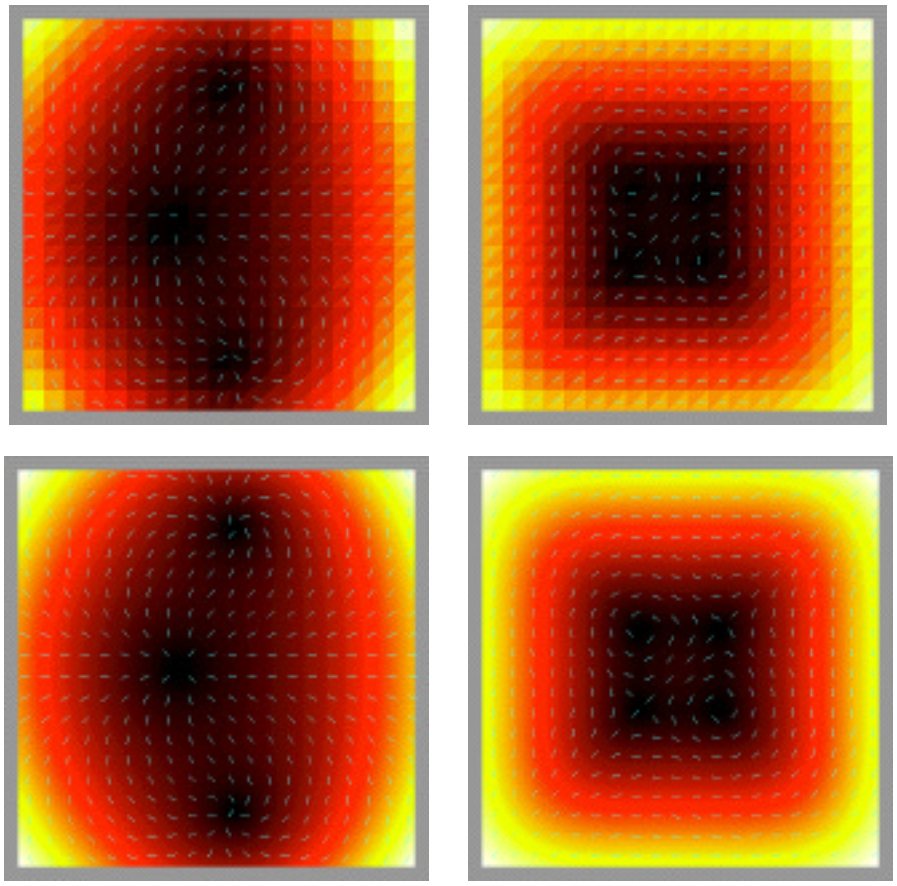


Figure 4.8: two visualizations: a function of a complex variable (L) and a differential equation (R)
The top row is relatively low resolution (20x20) and the bottom row is high resolution (200x200)

A different approach is to consider functions with a two-dimensional domain and with a two-dimensional range, and to try to find ways to display this information, which is essentially four-dimensional, to your audience. Two examples of this higher-dimension situation are vector-valued functions on a rectangular real space, or complex-valued functions of a single complex variable. Figure 4.8 below presents these two examples: a system of two first-order differential equations of two variables (left) and a complex-valued function of a complex variable (right). The domain is the standard rectangular region of two-dimensional space, and we have taken the approach of encoding the range in two parts based on considering each value as a vector with a length and a direction. We encode the magnitude of the vector or complex number as a pseudocolor with the uniform color ramp as described above, and the direction of the vector or complex number as a fixed-length vector in the appropriate direction. In the top row we use a relatively coarse resolution of the domain space, while in the bottom row we use a much finer resolution. Note that even as we increase the resolution of the mesh on which we evaluate the functions, we keep the resolution of

the vector display about the same. This 20x20 vector display mesh is about as fine a resolution as a user can understand on a standard screeen.

Higher dimensions

The displays in Figure 4.8 are fundamentally 2D images, with the domain of the functions given by the display window and the range of the functions represented by the color of the domain and the direction of the vector. There have been similar visualizations where the range had dimension higher than two, and the technique for these is often to replace the vector by an object having more information [NCSA work reference]. Such objects, called *glyphs*, need to be designed carefully, but they can be effective in carrying a great deal of information, particularly when the entire process being visualized is dynamic and is presented as an animation with the glyphs changing with time.

Of course, there are other techniques for working with higher-dimensional concepts. One of these is to extend concept of projection. We understand the projection from three-dimensional eye space to two-dimensional viewing space that we associate with standard 3D graphics, but it is possible to think about projections from spaces of four or more dimensions into three-dimensional space, where they can be manipulated in familiar ways. An example of this is the image of Figure 4.9, a image of a hypercube (four-dimensional cube). This particular image comes from an example where the four-dimensional cube is rotating in four-dimensional space and is then projected into three-space.

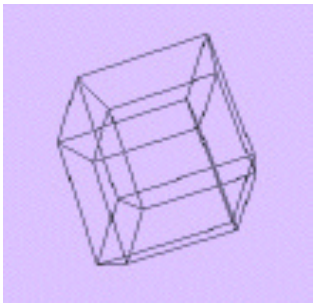


Figure 4.9: a hypercube projected into three-space

Choosing an appropriate view

When you create a representation of information for an audience, you must focus their attention on the content that you want them to see. If you want them to see some detail in context, you might want to start with a broad image and then zoom into the image to see the detail. If you want them to see how a particular portion of the image works, you might want to have that part fixed in the audience's view while the rest of your model can move around. If you want them to see the entire model from all possible viewpoints, you might want to move the eye around the model, either under user control or through an animated viewpoint. If you want the audience to follow a particular path or object that moves through the model, then you can create a moving viewpoint in the model. If you want them to see internal structure of your model, you can create clipping planes that move through the model and allow the audience to see internal details, or you can vary the way the colors blend to make the areas in front of your structure more and more transparent so the audience can see through them. But you should be very conscious of how your audience will see the images so you can be sure that they see what you need them to see.

Moving a viewpoint

We have already discussed the modeling issues involved in defining a viewpoint as part of the geometry in the scene graph. If we want to move the viewpoint, either under user control or as

part of the definition of a moving scene, we will need to include that viewpoint motion in the model design and account for it in the code that renders each version of the scene. If the viewpoint acts as a top-level item in the scene graph, you can simply use parameters to define the viewpoint with whatever tools your graphics API gives you, and the changing view will reflect your modeling. This could be the case if you were defining a path the eye is to follow through the model; you need only encode the paramters for the eye based on the path.

On the other hand, if your viewpoint is associated with other parts of the model, such as being part of the model (for example, looking through the windshield of a racing car driving around a track) or following an object in the model (for example, always looking from the position of one object in the model towards another object in the model, as one might find in a target tracking situation) then you will need to work out the transformations that place the eye as desired and then write code that inverts the eye-placement code before the rest of the scene graph is handled. This was described in the modeling chapter and you should consult that in more detail.

Setting a particular viewpoint

A common technique in an animation involving multiple moving bodies is to "ground" or freeze one of the moving bodies and then let the other bodies continue their motions, showing all these motions with respect to the grounded body. This kind of view maintains the relative relationships among all the moving bodies, but the chosen part is seen as being stationary. This is a useful technique if a user wants to zoom in on one of the bodies and examine its relationship to the system around it in more detail, because it is difficult to zoom in on something that is moving. We will outline the way this mechanism is organized based on the scene graph, and we could call this mechanism an "AimAt" mechanism, because we aim the view at the part being grounded.

In the context of the scene graph, it is straightforward to see that what we really want to do is to set the viewpoint in the model to be fixed relative to the part we want to freeze. Let us show a simplified view of the relationship among the parts as the scene graph fragment in Figure 4.10, which shows a hierarchy of parts with Part 3 attached to Part 2 and Part 2 attached to Part 1, all located relative to the world space (the Ground) of the scene. We assume that the transformations are implicit in the graph and we understand that the transformations will change as the animated scene is presented.

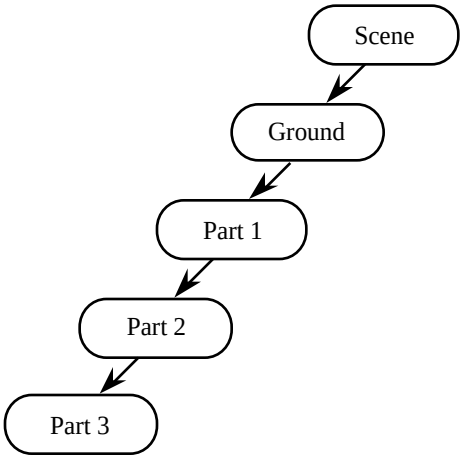



Figure 4.10: a hierarchy of parts in a mechanism

If we choose to ground Part 2 at a given time, the viewpoint is attached to that part, and the tree would now look like Figure 4.11 with right-hand branch showing the location of the eye point in the graph. Here the superscript * on a part names in the right-hand branch means that the branch includes the specific values of the transformations at the moment the part is frozen. The additional

node for the eyepoint indicates the relation of the eyepoint to the part that is to be frozen (for example, a certain number of units away in a certain direction from the part).

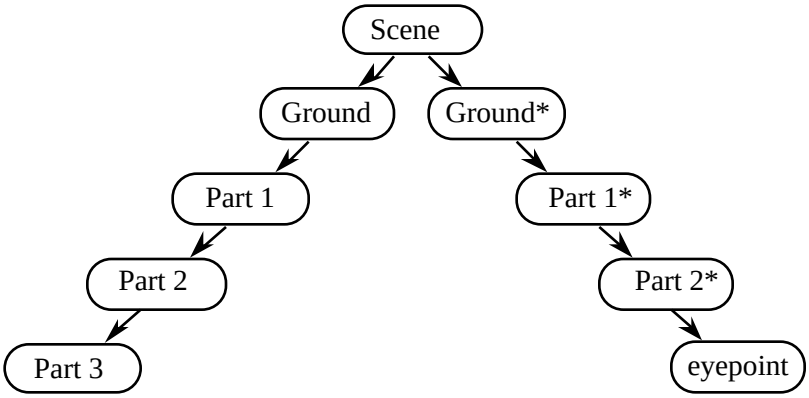



Figure 4.11: the scene graph with the transformations for Part 2* captured

Then to create the scene with this part grounded, we carry out the inversion of the eye point branch of the graph just as described in the modeling chapter. Note that anything else that was part of the original scene graph would be attached to the original Ground node, because it is not affected by the grounding of the particular part.

In Figure 4.12 we show this process at work. This figure shows time-exposures of two views of a mechanical four-bar linkage. The left-hand image of the figure shows how the mechanism was originally intended to function, with the bottom piece being fixed (grounded) and the loop of points showing the motion of the top vertex of the green piece. The right-hand image in the figure shows the same mechanism in motion with the top piece grounded and all motions shown relative to that piece.

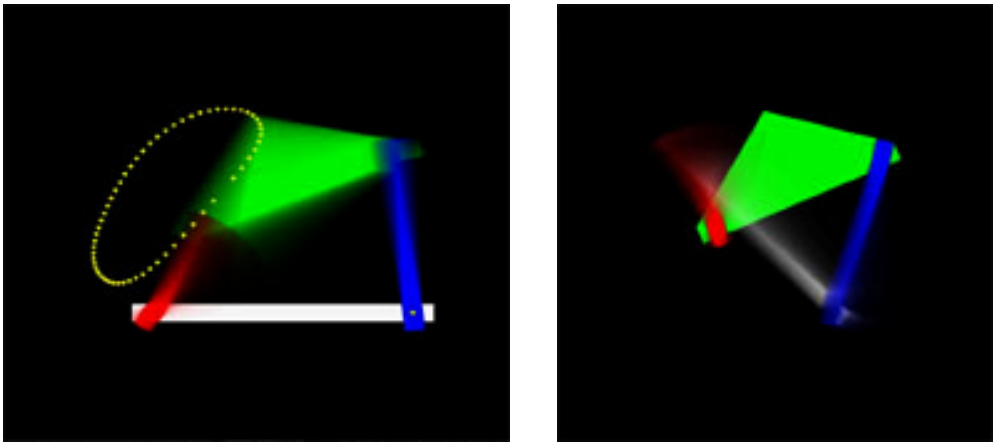


Figure 4.12: animated mechanisms with different parts fixed

Seeing motion

When you are conveying information about a moving geometry to your audience, you are likely to want to use an animation. However, sometimes you need to show more detail than a viewer can get from moving images, while you still want to show the motion. Or perhaps you want each frame of your image to show something of the motion itself. We could say that you would want to show your viewer a *trace* of the motion.

There are two standard ways you can show motion traces. The first is to show some sort of trail of previous positions of your objects. This can be handled rather easily by creating a set of lines or similar geometric objects that show previous positions for each object that is being traced. This trace should have limited length (unless you want to show a global history, which is really a different visualization) and can use techniques such as reduced alpha values to show the history of the object's position. Figure 4.13 shows two examples of such traces; the left-hand image uses a sequence of cylinders connecting the previous positions with the cylinders colored by the objecd color with reducing alpha values, while the right-hand image shows a simple line trace of a single particle illustrating a random walk situation.

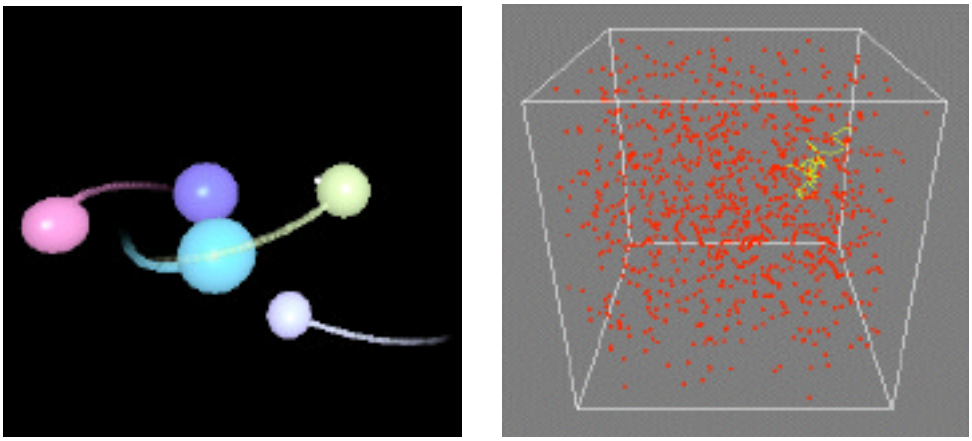


Figure 4.13: two kinds of traces of moving objects

Another approach to showing motion is to use images that show previous positions of the objects themselves. Many APIs allow you to accumulate the results of several different renderings in a single image, so if you compute the image of an object at several times around the current time you can create a single image that shows all these positions. This can be called motion blur as well as image accumulation. The images of Figure 4.12 show good examples of this kind of accumulated motion, and the technique for creating it in OpenGL is discussed later in this chapter.

Legends to help communicate your encodings

Always be careful to help your audience understand the information you are presenting with your images. Always provide appropriate legends and other textual material to help your audience understand the content of your displays. If you use pseudocolor, present scales that can help a viewer interpret the color information. This allows people to understand the relationships provided by your color information and to understand the context of your problem, and is an important part of the distinction between pretty pictures and genuine information. Creating images without scales or legends is one of the key ways to create misleading visualizations.

The particular example we present here is discussed at more length in the first science applications chapter. It models the spread of a contagious disease through a diffusion process, and our primary interest is the color ramp that is used to represent the numbers. This color ramp is, in fact, the uniform heat ramp introduced earlier in this chapter, with evenly-changing luminance that gets higher (so the colors get lighter) as the values gets higher.

So far we have primarily presented only images in the examples in this chapter, but the image alone only makes up part of the idea of using images to present information. Information needs to be put into context to help create real understanding, so we must give our audience a context to help them understand the concept being presented in the image and to see how to decode any use of color or other symbolism we use to represent content.

Figure 4.14 shows an image with a label in the main viewport (a note that this image is about the spread of disease) and a legend in a separate viewport to the right of the main display (a note that says what the color means and how to interpret the color as a number). The label puts the image in a general context, and as the results of this simulation (a simulation of the spread of a disease in a geographic region with a barrier) are presented in the main viewport, the legend to the right of the screen helps the viewer understand the meaning of the rising and falling bars in the main figure as the figure is animated and the disease spreads from a single initial infection point.

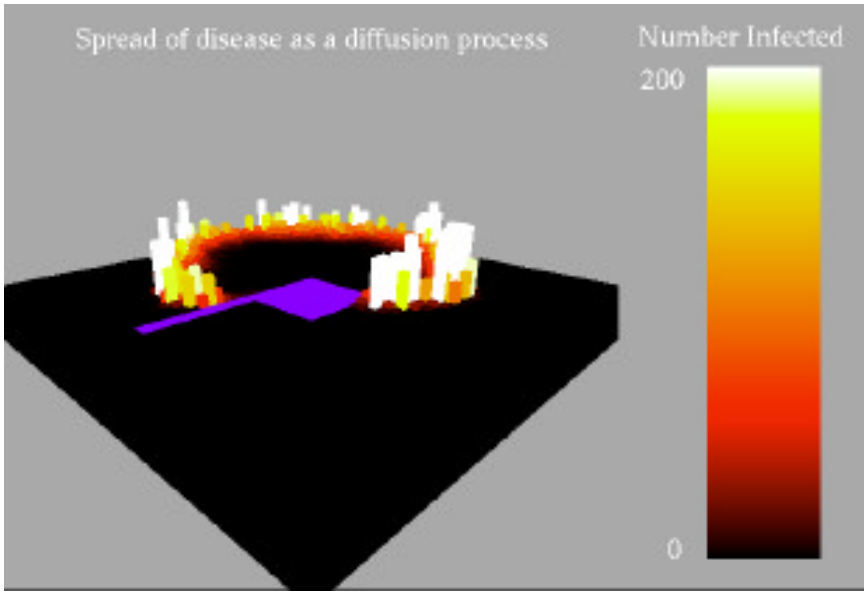



Figure 4.14: an example of figure with a label and a legend to allow the figure to be interpreted

*Implementing some of these techniques in OpenGL*

Legends and labels

Each graphics API will likely have its own ways of handling text, and in this short section we will describe how this can be done in OpenGL. We will also show how to handle the color legend in a separate viewport, which is probably the simplest way to deal with the legend's graphic.

The text in the legend is handled by creating a handy function, `doRasterString(...)` that displays bitmapped characters, implemented with the GLUT `glutBitmapCharacter()` function. Note that we choose a 24-point Times Roman bitmapped font, but there are probably other sizes and styles of fonts available to you through your own version of GLUT, so you should check your system for other options.

```
void doRasterString( float x, float y, float z, char *s)
{
   char c;

   glRasterPos3f(x,y,z);
   for ( ; (c = *s) != '\0'; s++ )
      glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, c);
}
```

The rest of the code used to produce this legend is straightforward and is given below. Note that the sprintf function in C needs a character array as its target instead of a character pointer. This code could be part of the display callback function where it would be re-drawn

```
// draw the legend in its own viewport
   glViewport((int)(5.*(float)winwide/7.),0,
      (int)(2.*(float)winwide/7.),winheight);
   glClear( GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT );
   ... // set viewing parameters for the viewport
   glPushMatrix();
   glEnable (GL_SMOOTH);
   glColor3f(1.,1.,1.);
   doRasterString(0.1, 4.8, 0., "Number Infected");
   sprintf(s,"%5.0f",MAXINFECT/MULTIPLIER);
   doRasterString(0.,4.4,0.,s);
// color is with the heat ramp, with cutoffs at 0.3 and 0.89
   glBegin(GL_QUADS);
      glColor3f(0.,0.,0.);
      glVertex3f(0.7, 0.1, 0.);
      glVertex3f(1.7, 0.1, 0.);
      colorRamp(0.3, &r, &g, &b);
      glColor3f(r,g,b);
      glVertex3f(1.7, 1.36, 0.);
      glVertex3f(0.7, 1.36, 0.);

      glVertex3f(0.7, 1.36, 0.);
      glVertex3f(1.7, 1.36, 0.);
      colorRamp(0.89, &r, &g, &b);
      glColor3f(r,g,b);
      glVertex3f(1.7, 4.105, 0.);
      glVertex3f(0.7, 4.105, 0.);

      glVertex3f(0.7, 4.105, 0.);
      glVertex3f(1.7, 4.105, 0.);
      glColor3f(1.,1.,1.);
      glVertex3f(1.7, 4.6, 0.);
      glVertex3f(0.7, 4.6, 0.);
   glEnd();
   sprintf(s,"%5.0f",0.0);
   doRasterString(.1,.1,0.,s);
   glPopMatrix();
   glDisable(GL_SMOOTH);
// now return to the main window to display the actual model
```

Using the accumulation buffer

The accumulation buffer is one of the buffers available in OpenGL to use with your rendering. This buffer holds floating-point values for RGBA colors and corresponds pixel-for-pixel with the frame buffer. The accumulation buffer holds values in the range [-1.0, 1.0], and if any operation on the buffer results in a value outside this range, its results are undefined (that is, the result may differ from system to system and is not reliable) so you should be careful when you define your operations. It is intended to be used to accumulate the weighted results of a number of display operations and has many applications that are beyond the scope of this chapter; anyone interested in advanced applications should consult the manuals and the literature on advanced OpenGL techniques.

As is the case with other buffers, the accumulation buffer must be chosen when the OpenGL system is initialized, as in

```
glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE|GLUT_ACCUM|GLUT_DEPTH);
```

The accumulation buffer is used with the function `glAccum(mode, value)` that takes one of several possible symbolic constants for its mode, and with a floating-point number as its value. The available modes are

| | |
|---|---|
| GL_ACCUM | Gets RGBA values from the current read buffer (by default the FRONT buffer if you are using single buffering or the BACK buffer if double buffering, so you will probably not need to choose which buffer to use), converts them from integer to floating-point values, multiplies them by the value parameter, and adds the values to the content of the accumulation buffer. If the buffer has bit depth n, then the integer conversion is accomplished by dividing each value from the read buffer by $2^n$-1. |
| GL_LOAD | Operates similarly to GL_ACCUM, except that after the values are obtained from the read buffer, converted to floating point, and multiplied by value, they are written to the accumulation buffer, replacing any values already present. |
| GL_ADD | Adds the value of value to each of the R, G, B, and A components of each pixel in the accumulation buffer and returns the result to its original location. |
| GL_MULT | Multiplies each of the R, G, B, and A components of each pixel in the buffer by the value of value and returns the result to its original location. |
| GL_RETURN | Returns the contents of the accumulation buffer to the read buffer after multiplying each of the RGBA components by value and scaling the result back to the appropriate integer value for the read buffer. If the buffer has bit depth n, then the scaling is accomplished by multiplying the result by $2^n$-1 and clamped to the range [0, $2^n$-1]. |

You will probably not need to use some of these operations to show the motion trace. If we want to accumulate the images of (say) 10 positions, we can draw the scene 10 times and accumulate the results of these multiple renderings with weights $2^{-i}$ for scene i, where scene 1 corresponds to the most recent position shown and scene 10 to the oldest position. This takes advantage of the fact that the sum

$$\sum_{i=1 \text{ to } 10}(2^{-i})$$

is essentially 1, so we keep the maximum value of the accumulated results below 1.0 and create almost exactly the single-frame image if we have no motion at all. Some code to accomplish this is:

```
// we assume that we have a time parameter for the drawObjects(t)
// function and that we have defined an array times[10] that holds
// the times for which the objects are to be drawn.  This is an example
// of what the manuals call time jittering; another example might be to
// choose a set of random times, but this would not give us the time
// trail we want for this example.
   drawObjects(times[9]);
   glAccum(GL_LOAD, 0.5)
   for (i = 9; i > 0; i--) {
      glAccum(GL_MULT, 0.5);
      drawObjects(times[i-1]);
      glAccum(GL_ACCUM, 0.5);
   }
   glAccum(GL_RETURN, 1.0);
```

A few things to note here are that we save a little time by loading the oldest image into the accumulation buffer instead of clearing the buffer before we draw it, we draw from the oldest to the newest image, we multiply the value of the accumulation buffer by 0.5 before we draw the next image, and we multiply the value of the new image by 0.5 as we accumulate it into the buffer. This accomplishes the successive reduction of the older images automatically.

There are other techniques one could find here, of course. One would be simply to take whatever image you had computed to date, bring it into the accumulation buffer with value 0.5, draw the new scene and accumulate it with weight 0.5, and return the scene with weight 1.0. This would be faster and would likely not show much difference from the approach above, but it does not show the possibilities of drawing a scene with various kinds of jittering, a useful advanced technique.

*A word to the wise...*

It is very easy to use the background we have developed in modeling to create geometric shapes to represent many kinds of data. However, in many cases the data will not really have a geometric or spatial context and it can be misleading to use all the geometric structures we might find. Instead of automatically assuming that we should present interesting 3D shapes, we need to ask carefully just what content we have in the data and how we can make that content visible without adding any suggestions of other kinds of content. Often we will find that color can carry more accurate meaning than geometry.

When you use color to carry information in an image, you need to be aware that there are many different meanings to colors in different cultural contexts. Some of these contexts are national: in European cultures, white means purity or brightness; in Japan, white means death. Other contexts are professional: red means heat to scientists, danger to engineers, losses to bankers, or health to physicians — at least to a significant extent. This is discussed at more length in [BRO] and you are referred there for more details. So be careful about the cultural context of your images when you choose colors.

Other serious issues with color include a range of particular considerations: people's ability to see individual colors in color environments, the way pairs or sets of colors interact in your audience's visual systems, how to communicate with people who have various color perception impairments, or how to choose colors so that your images will still communicate well in a black-and-white reproduction. You need to become aware of such considerations before you begin to do serious work in visual communication.

# Science Examples I

*Prerequisites:*

A knowledge of computer graphics through modeling, viewing, and color, together with enough programming experience to implement the images defined by the science that will be discussed in this section.

*Graphics to be learned from these projects:*

Implementing sound modeling and visual communication for various science topics.

This chapter contains a varied collection of science-based examples that you can understand with a basic knowledge of computer graphics, including viewing, modeling, and color. These examples are not very sophisticated, but are a sound start towards understanding the role of computer graphics in working with scientific concepts. The examples are grouped based on type of problem so that similar kinds of graphics can be brought to bear on the problems. This shows some basic similarities between problems in the different areas of science and perhaps can help the student see that one can readily adapt solutions to one problem in creating solutions to a similar problem in a totally different area.

Each example described below will describe a science problem and the graphic image or images that address it, and will include the following kinds of information:

- A short description of the science in the problem
- A short description of the modeling of the problem in terms of the sciences
- A short description of the computational modeling of the problem, including any assumptions that we make that could simplify the problem and the tradeoffs implicit in those assumptions
- A description of the computer graphics modeling that implements the computational modeling
- A description of the visual communication in the display, including any dynamic components that enhance the communication
- An image from an implementation of the model in OpenGL
- A short set of code fragments that make up that implementation

There topics in this chapter cover only a few scientific applications, but are chosen to use only the limited graphics tools we have at this point. Additional science examples are found in a later chapter. They are a critical part of these notes because an understanding of the science and of the scientific modeling is at the heart of any good computational representation of the problem and thus at the heart of the computer graphics that will be presented.

*Examples:*

<u>Modeling diffusion of a quantity in a region</u>

1. Temperature in a metal bar

A classical physics problem is the heat distribution in a metallic bar with fixed heat sources and cold sinks. That is, if some parts of the bar are held at constant temperatures, we ask for the way the rest of the bar responds to these inputs to achieve a steady-state temperature. We model the heat distribution with a diffusion model, where the heat in any spot at time t+1 is determined by the heat in that spot and in neighboring spots at time t. We model the bar as a grid of small rectangular regions and assume that the heat flows from warmer grid regions into cooler grid regions, so the temperature in one cell at a given time is a weighted sum of the temperatures in neighboring cells at the previous time. The weights are given by a pattern such as the following, where the current cell is at row m and column n:

| row\col | n-1 | n | n+1 |
|---|---|---|---|
| m+1 | 0.05 | 0.1 | 0.05 |
| m | 0.1 | 0.4 | 0.1 |
| m-1 | 0.05 | 0.1 | 0.05 |

That is, the temperature at time t+1 is the weighted sum of the temperatures in adjacent cells with the weights given in the table. Thus the heat at any grid point at any time step depends on the heat at previous steps through a function that weights the grid point and all the neighboring grid points. See the code sample below for the implementation of this kind of weighted sum. In this sample, we copy the original grid, `temps[][]`, into a backup grid, `back[][]`, and then compute the next set of values in `temps[][]` from the backup grid and the filter. This code is found in the idle callback and represents the changes from one time step to the next.

```
      float filter[3][3]={{ 0.0625, 0.125, 0.0625 },
                          { 0.125,  0.25,  0.125  },
                          { 0.0625, 0.125, 0.0625 } };

   // first back temps up so you can recreate temps
      for (i=0; i<ROWS; i++)
         for (j=0; j<COLS; j++)
            back[i+1][j+1] = temps[i][j];
      for (i=1; i<ROWS+2; i++) {
         back[i][0]=back[i][1];
         back[i][COLS+1]=back[i][COLS];
         }
      for (j=0; j<COLS+2; j++) {
         back[0][j] = back[1][j];
         back[ROWS+1][j]=back[ROWS][j];
         }
      // use diffusion based on back values to compute temp
      for (i=0; i<ROWS; i++)
         for (j=0; j<COLS; j++) {
            temps[i][j]=0.0;
            for (m=-1; m<=1; m++)
               for (n=-1; n<=1; n++)
                  temps[i][j]+=back[i+1+m][j+1+n]*filter[m+1][n+1];
         }
      // reset the temperatures of the hot and cold spots
      for (i=0; i<NHOTS; i++) {
         temps[hotspots[i][0]][hotspots[i][1]]=HOT; }
      for (i=0; i<NCOLDS; i++) {
         temps[coldspots[i][0]][coldspots[i][1]]=COLD; }

      // finished updating temps; now do the display
      glutPostRedisplay(); /* perform display again */
```

The behavior of the model is to reach a steady state in which the hot and cold spots stay fixed and all the other points in the region reach an intermediate temperature with the overall inflow and outflow of heat are the same. The display at one step in the process is shown in Figure 5.1.

The display was discussed earlier in the module on visual communication, but it bears reviewing here. Each grid element is displayed as a bar (a vertically-scaled cube, translated to the position of the element) whose height and color are determined by the temperature. This provides a dual encoding of the temperature information, and the image is rotated slowly around its center so the viewer can see the bar from all directions. Variations on the display would include color-only,

height-only, and varying the amount of the rotations (including the possibility of having no rotation so the user sees only the same view and can focus only on the color changes).

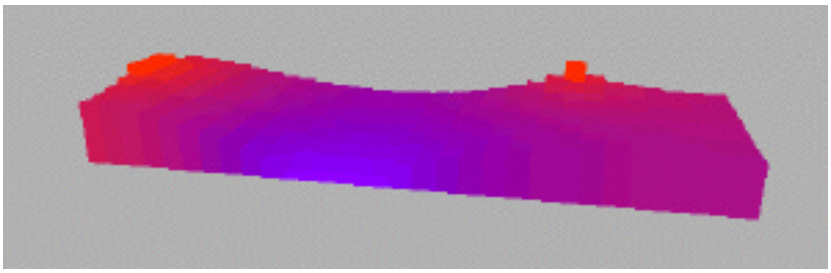


Figure 5.1: state of the heat in the bar as it moves towards stabilization

This process is readily modeled by modeling the problem in terms of changes between time steps, and updating the temperatures at step t+1 from those at step t using the process described above. As a first graphics project, we need to deal with three things: defining a grid of temperatures and modeling how the temperatures change from step to step, defining how the temperatures can be modeled as a height field on the grid, and displaying the height field as a collection of rectangular boxes that are scaled cubes whose color depends on the temperature. We thus have two ways to encode temperature information, which allows a discussion of which encoding is more informative. In addition to a simple display of the image with changing colors and heights, we suggest adding an automatic rotation of the image in order to allow the student to see how the height field looks from all directions. The example in the figure shows this dual encoding as well as the effect of different hot and cold spots in the bar.

2. Spread of disease in a region

In this example, we have an infectious disease (the original model was malaria) that arises in one location and spreads both within and between regions. The model is based on a few simple premises: that disease is spread by contact between infected and susceptible persons, that contact is approximated by the product of the number of infected persons and the number of susceptible persons, and that a certain proportion of infected persons recover and become immune at any given time. If we represent the numbers of susceptible, infected, and recovered persons by the functions S, I, and R of time, respectively, then a single-population infection is modeled by the three difference equations

```
S(n+1) = S(n) - a*I(n)*S(n)
I(n+1) = I(n) + a*I(n)*S(n) - b*S(n)
R(n+1) = R(n) + b*S(n)
```

with initial conditions `S(0) = P - e; I(0) = e; R(0) = 0` and with constants `a` and `b` representing the rates of infection and recovery. However, if we include the concept of multiple regions with separate populations in each region, this model is complicated by the interaction of infected populations in one region with the susceptible populations in adjacent regions. This changes the model of new infections from the simple `a*I(n)*S(n)` to a more complex version that includes the contact rate between persons from adjacent regions. In the real world, we can imagine that there are some regions with no contact between populations, so the full model includes some regions in which the disease cannot spread because of lack of contact. To avoid all the complexities of how this could happen, we assume in this model that there is no contact across lakes or rivers.

The computer implementation of this model illustrates the more complex contact situation. Here we model the regions as an array of populations, with each having an internal population dynamic. The regions are very homogeneous in our model, but could be made more complex. We also include the lake and river concept by making some regions have populations of zero, which is an

effective way to ensure that no contact, and hence no spread of disease, can happen. In our model, these regions are colored blue to distinguish them from the other regions. The primary code for the model is in the idle callback, and we show this and the initialization code for the regions below. The code to back up the working array is the same as that in the model for heat transfer and is omitted here to save space. The code for the legend is not presented here because that is presented in the visual communication module above.

```
      // set up initial states
      for (i=0; i<ROWS; i++) {
         for (j=0; j < COLS; j++) {
            regions[i][j].s = P;
            regions[i][j].i = regions[i][j].r = 0.;
            regions[i][j].aN = regions[i][j].aE =
               regions[i][j].aS = regions[i][j].aW = 0.01;
         }
      }
      // exceptions: lake and other barriers; initial infection in one region
      for (i = 25; i<36; i++ )
         for (j=25; j<36; j++ ) {
            regions[i][j].s = 0.;
            regions[i][j].aN = regions[i][j].aE =
               regions[i][j].aS = regions[i][j].aW = 0.0;
         }
      for (i = 36; i<50; i++ )
         for (j=25; j<27; j++ ) {
            regions[i][j].s = 0.;
            regions[i][j].aN = regions[i][j].aE =
               regions[i][j].aS = regions[i][j].aW = 0.0;
         }
      regions[20][20].i = 200.;
      regions[20][20].s = P - 200.;

   void animate(void)
   {  ...
      // use diffusion based on model using back values to compute temp
      for (ii=0; ii<ROWS; ii++)
         for (jj=0; jj<COLS; jj++) {
            i = ii+1; j = jj+1;
            newi = alpha*back[i][j].i*back[i][j].s;
            if (i!=0)
               newi  += alpha*(back[i][j].aN*back[i-1][j].i*back[i][j].s);
            if (j!=COLS)
               newi  += alpha*(back[i][j].aE*back[i][j+1].i*back[i][j].s);
            if (i != ROWS)
               newi += alpha*(back[i][j].aS*back[i+1][j].i*back[i][j].s);
            if (j != 0)
               newi += alpha*(back[i][j].aW*back[i][j-1].i*back[i][j].s);
            regions[ii][jj].r += beta * back[i][j].i;
            regions[ii][jj].i = (1.-beta)*back[i][j].i + newi;
            regions[ii][jj].s = back[i][j].s - newi;
         }
      glutPostRedisplay();
   }
```

The display in Figure 5.2 is presented as a rotation region on the left and a stationary legend on the right. As the region on the left rotates, each rotation step corresponds to a time step for the infection simulation, showing how the infection spreads and goes around the barrier regions. This

allows a user the chance to examine the nature of the disease spread and, by watching the behavior of the infection in adjacent regions, to see what amounts to the history of the infection in a single population. One could vary the display by adjusting the rate of rotation (down to zero), by randomizing some of the parameters of the simulation, or by using color alone for the display.

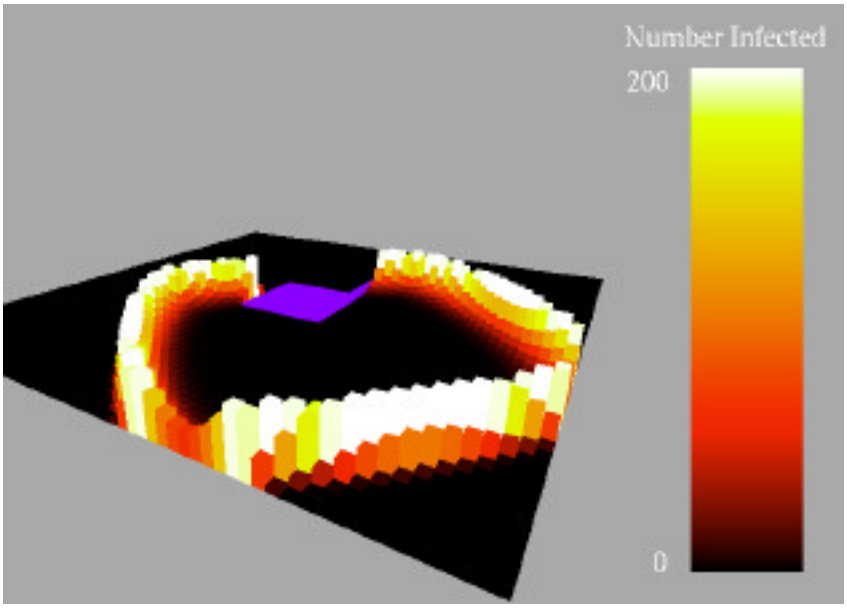



Figure 5.2: the malaria spread model showing the behavior around the barrier regions

An interesting extension of the simulation would be to experiment with the barriers that force the disease spread to take on interesting directions, and to consider more sophisticated models of heterogeneous populations or differing population interactions. The most critical thing to notice is probably that as long as there is any population interaction at all, the disease will spread to the new population; any attempt to control the spread of the infection will fail unless the separation is complete.

Simple graph of a real function of two variables

Surfaces are plotted as we illustrate in Figure 5.3. We create a grid on a rectangular domain in two-space and apply a function or functions to each of the points in the grid to determine a two-dimensional array of points in three-space. This figure illustrates the fundamental principal for

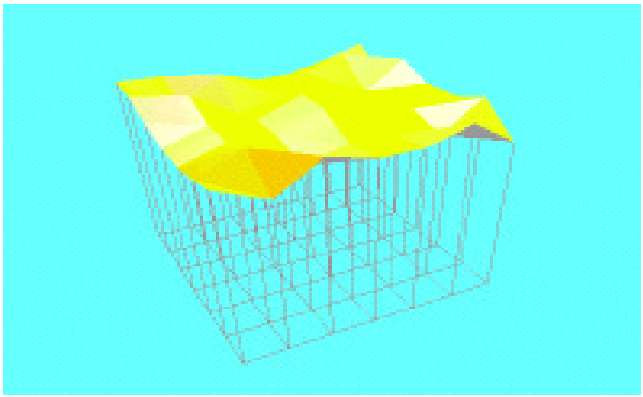


Figure 5.3: mapping a domain rectangle to a surface rectangle

surfaces that are given by a simple function of two variables; it shows a coarse grid on the domain (only a 6x6 grid for a function surface so you may see the underlying grid) and the relationship of the function's value to the surface. Those points are used to determine rectangles in three-space that can be displayed with standard OpenGL functions. Each rectangle is actually displayed as two triangles so that we can display objects that are guaranteed to be planar. The display in the figure uses standard lighting and flat shading with a good deal of specularity to show the shape of the surface, but we will explore many presentation options for creating the display. The gridded surfaces we create in this way are only approximations of the real surfaces, of course, and special cases of functions such as points of discontinuity must be considered separately.

3. Mathematical functions

If we consider a function of two variables, `z=f(x,y)`, acting on a contiguous region of real two-space, then the set of points `(x,y,f(x,y))` forms a surface in real three-space. This project explores such surfaces through processes that are described briefly in the figure above.

The goal of the example is to allow you to see the fundamental principles of surfaces by creating a rectangular domain in X-Y space, evaluating the function at a regular grid of points in that domain, and creating and displaying small rectangles on the surface that correspond to a small rectangle in the underlying grid. This is not quite so simple, however, because the rectangle in the surface may not be planar. We solve that problem by dividing the surface rectangle into two triangles, each of which is planar.

The first step in the project is to create a single view of a surface. The challenges are to create the view environment and to understand what is meant by the surface triangles and rectangles, and how they are generated. Because we do not yet have the background to manage lights and lighting models, we will use color to represent the value of the function when we display the surface, so we will set the color for each triangle to represent the height of that triangle in the surface.

The code to implement this display uses simple functions to map array indices to domain values and uses an array to hold the function values at each grid point. It includes declaring the array, defining the simple functions, loading the array, and rendering the surface.

```
// Parameters of the surface grid; NxM grid needs XSIZE = N+1, YSIZE = M+1
#define XSIZE 100
#define YSIZE XSIZE
#define MINX  -6.0
#define MAXX   6.0
#define MINY  -6.0
#define MAXY   6.0

static GLfloat vertices[XSIZE][YSIZE];

// functions for X and Y values for array indices i and j respectively
GLfloat XX(int i) {
   return (MINX+((MAXX-MINX)/(float)(XSIZE-1))*(float)(i));
}

GLfloat YY(int j) {
   return (MINY+((MAXY-MINY)/(float)(YSIZE-1))*(float)(j));
}

...
   for ( i=0; i<XSIZE; i++ )
```

```
         for ( j=0; j<YSIZE; j++ ) {
            x = XX(i);
            y = YY(j);
            vertices[i][j] = 0.3*cos(x*x+y*y+t); break;
         }

      // actually draw the surface */
      for ( i=0; i<XSIZE-1; i++ )
         for ( j=0; j<YSIZE-1; j++ ) {
            // first triangle in the quad, front face
            glBegin(GL_POLYGON);
               // set color of polygon by the z-value
               ...
            glEnd();

            // second triangle in the quad, front face
            glBegin(GL_POLYGON);
               // set color of polygon by the z-value
               ... //   similar to above
            glEnd();
         }
...
```

Note that the function `0.3*cos(x*x+y*y+t)` above includes a parameter `t` that can be incremented to provide the surface animation described in Figure 5.4 below. The display is very simple because the goal of the example is simply to understand the nature of the function by the shape of its function surface.

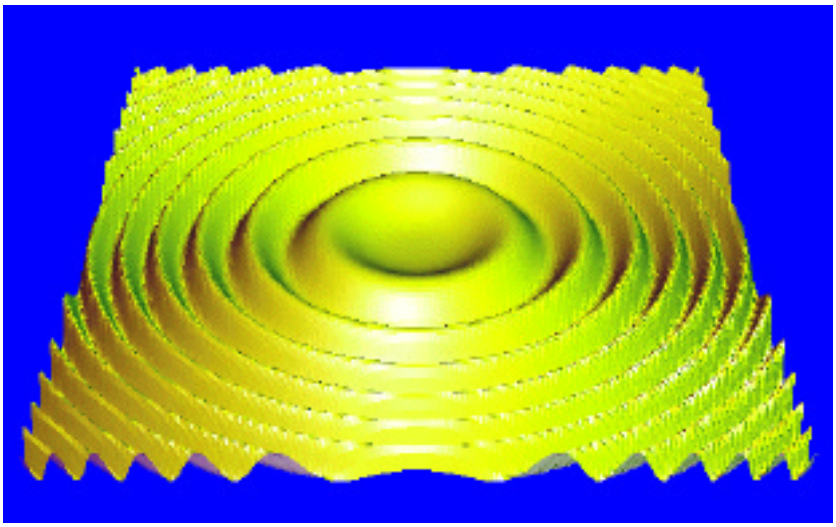


Figure 5.4: an example of a function surface display

In order to have a good set of functions to work with, we encourage you to look at your courses, including courses in physics or chemistry, or in references such as the CRC tables of mathematical functions for curious and interesting formulas that you do not immediately understand, with a goal of having your images help with this understanding.

To avoid having to change the code and recompile when you want to look at a different function, we suggest creating a menu of surfaces and defining the project to include menu selection for the function. Some interesting functions to consider for this are:

```
1. z = c * (x^3 - 3*x*y^2)
2. z = c * (x^4/a^4 - y^4/b^4)
3. z = (a*x^2 + b*y^2)/(x^2+y^2)
4. z = c*cos(2*_*a*x*y)
5. z = c*cos(2*_*a*x)*(2*_*a*y)
6. z = c*log(a*x^2 + b*y^2)
```

In this project it is fairly easy to pose some questions about the meaning of what you see, particularly if the you have chosen a good set of functions that have various kinds of discontinuities within a usual domain. Note that function 3 includes an essential singularity at the origin, so you will be faced with having to interpret this surface's inaccuracies.

4. Electrostatic potential function

The electrostatic potential `P(x,y)` at a given point `(x,y)` in a plane that contains several point charges `Qi` is given by Coulomb's law:

$$P(x,y) = \sum \frac{Qi}{\sqrt{(x-xi)^2+(y-yi)^2}}$$

where each charge `Qi` is positioned at point `(xi,yi)`. This function can readily be presented through a surface plot using standard graphics techniques as described above. This is a real-valued function of two real variables defined on the real plane, so the only issue is defining the domain on which the function is to be graphed and the grid on the domain. Again, the color of each triangle in the surface is set by the height, or z-value, of the triangle. There is one problem with the function, however — it has singularities at the points where the point charges lie. The sample code with this example avoids the problem by adding a small amount to the distance between a point in the plane and each point charge. You can see this easily in the code below:

```
#define MAXCHARGES 3
typedef struct { float charge, xpos, ypos; } charges;
charges Q[MAXCHARGES] = { {  5.0, 3.0, 1.0},
                          { -5.0, 1.0, 4.0},
                          {-10.0 ,2.0, 2.0} };

float coulSurf(float x, float y)
{
   float retVal, dist;
   int i;

   retVal = 0.0;
   for (i=0; i<MAXCHARGES; i++) {
      dist=sqrt((x-Q[i].xpos)*(x-Q[i].xpos)+(y-Q[i].ypos)*(y-Q[i].ypos)+0.1);
      retVal += Q[i].charge/dist;
   }
   retVal = retVal / 6.0;  // scale in vertical direction
   return (retVal);
}
```

The example produces the result shown in Figure 5.5 below, with both a lighted surface and pseudocolor presentation, as shown in the earlier discussion of visual communication. As we will see later in these notes, when you have introduced interactive controls to your programs, you will be able to select one of the point charges (using either a menu or a mouse selection) and can experiment with this surface by moving the charged point around the plane or by changing the amount of charge at the point, or you may add or delete point charges. This can make an interesting example for interaction.

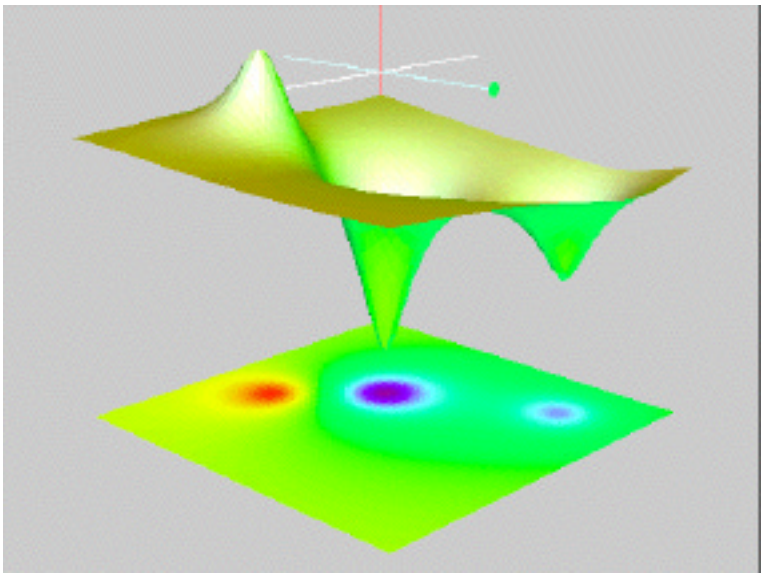


Figure 5.5: the coulombic surface from three point charges (one positive, two negative) in a plane, with both a surface and a pseudocolor presentation

Simulating a scientific process

5. Gas laws

It is fairly simple to program a simulation of randomly-moving molecules in an enclosed space. This simulation models the behavior of gases in a container and can allow a student to consider the relationship between pressure and volume or pressure and temperature in the standard gas laws. These laws say that when temperature is held constant, pressure and volume vary inversely. The simulation can display the principles involved and can allow a user to query the system for the analogs of temperature and pressure, getting back data that can be used in a statistical test of the law above.

The simulation is straightforward: the student defines an initial space and creates a large number of individual points in that space. The idle callback gives each point a random motion with a known average distance (simulating a fixed temperature) in a random direction. The space can be enlarged or shrunk. If the volume is shrunk, any points that might have been outside the new smaller space are moved back into the space; if the volume is expanded, points are allowed to move freely into the new space. If the motion would take the point outside the space, that motion is reflected back into the space and a "hit" is recorded to simulate pressure in the space. This count serves to model the pressure. So the student can use interactive controls to change the volume and observe the number of counts, testing the hypothesis that the number of counts increases inversely with the surface area. The student could also change the average distance of a random motion, modeling temperature changes, and test the hypothesis that the number of counts increases directly with the distance (that is, the temperature). Information on the volume, collision count, or distance traveled can be retrieved at any point by a simple keystroke. In fact, the small number of molecules in this simulation give a very large variability of the product of surface area and collisions, so the student will need to do some sample statistics to test the hypothesis.

The computational model for this process is straightforward. The points are positioned at random initial locations, and the idle callback handles motion of each particle in turn by generating a random offset that changes the particle's position. One particle has its motion tracked and a record

of its recent positions drawn to illustrate how the particles travel; the resulting track (the yellow track in the figure below) is a very good example of a random walk. The code for the trail for one point, for the idle callback, and for tallying the hits is:

```
// code for the trail of one point
      glBegin(GL_POINTS);
         for ( i=0; i<=NMOLS; i++ ) {
            glVertex3fv(mols[i]);
            if (i == 0) {
               if (npoints < TRAILSIZE) npoints++;
               for (j=TRAILSIZE-1; j>0; j--)
                  for (k=0; k<3; k++)
                     trail[j][k]=trail[j-1][k];
               for (k=0; k<3; k++){
                  trail[0][k] = mols[i][k];
               }
            }
         }

// the idle() callback
void animate(void)
{
   int i,j,sign;

   bounce = 0;
   for(i=0; i<NMOLS; i++) {
      for (j=0; j<3; j++) {
         sign = -1+2*(rand()%2);
         mols[i][j] +=
               (float)sign*distance*((float)(rand()%GRAN)/(float)(GRAN));
         if (mols[i][j] > bound)
            {mols[i][j] = 2.0*bound - mols[i][j]; bounce++;}
         if (mols[i][j] <-bound)
            {mols[i][j] = 2.0*(-bound) - mols[i][j]; bounce++;}
      }
   }
   glutPostRedisplay();
}

// tally the hits on the surface of the volume
void tally(void)
{
   float pressure, volume;

   pressure = (float)bounce/(bound*bound);   // hits per unit area
   volume   = bound*bound*bound;             // dimension cubed
   printf("pressure %f, volume %f, product %f\n",
            pressure,volume,pressure*volume);
}
```

A display of the volume and retrieved information is shown in Figure 5.6 below. This example naturally uses simple color and no lights because its goal is simply to show position and motion of the particles at any time step.

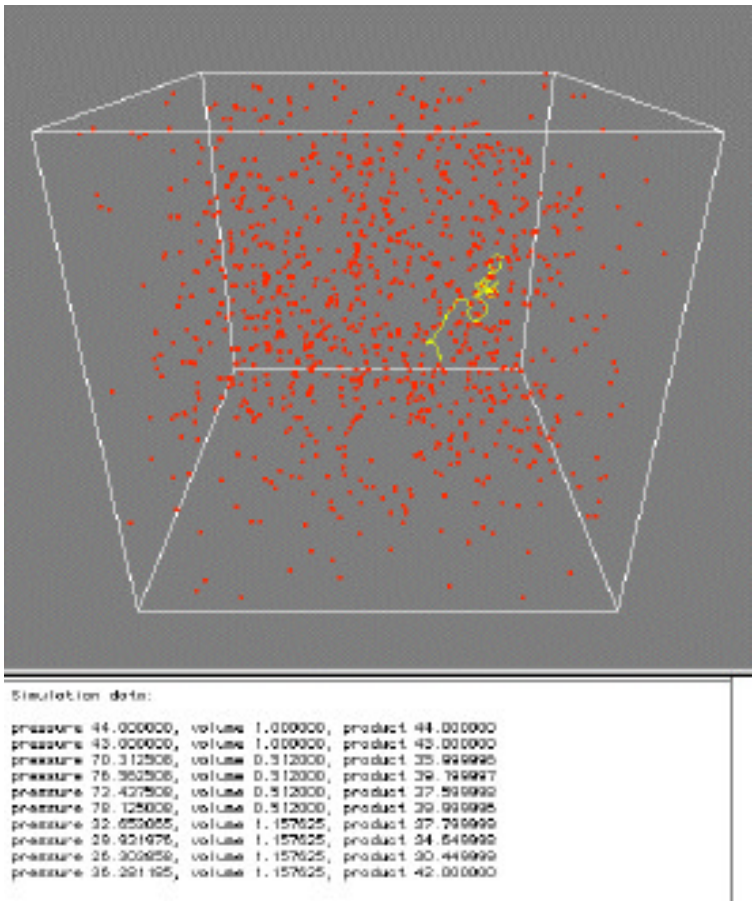



Figure 5.6: Displaying the gas as particles in the fixed space, with simulation printouts included

The display operates by providing a very rapid animation of the particles, so the user gets the strong impression of molecules moving quickly within the space. It is possible to move the view around with the keyboard, and every time the user presses the 't' key the program produces a tally such as that shown at the bottom of the figure. This the visual communication takes the form of an actual experiment that is probed at any time to read the current state. Of course, there are some possible problems with the simulation; if the user expands the space quickly, it may take some time for the particles to move out to the boundaries so the tallies are likely to be low for a while, and if the user contracts the space too quickly, the particles are seen outside the box for a short period until they can be brought back inside.

One may properly ask why this simulation does not demonstrate the relationship between pressure and temperature in the gas laws. This is primarily because the simulation does not include collisions between molecules. Adding this capability would require so much collision testing that it would slow down the simulation and take away the real-time nature of the display.

6. Diffusion through a semipermeable membrane

One of the common processes in biological processes involves molecules transporting across semi-permeable membranes. These membranes will allow molecules to pass at varying rates, depending on many factors. One of these factors is the molecular weight, where lighter molecules will pass through a membrane more easily than heavier molecules. (A more realistic factor is the physical dimension of the molecule, and this project can be re-phrased in terms of molecule size instead of weight).

This project involves processes very much like the gas law simulation above, including random motion of the molecules (represented as points), tracing the recent motion of molecules via rolling arrays, and the inclusion of a tally function. However, this simulation adds a plane between two regions and simulates the behavior of a semipermeable membrane for molecules of two different weights (here called "light" and "heavy" without any attempt to match the behaviors with those of any real molecules and membranes).

The code we present for this concentrates on simulating the motion of the particles with special attention to the boundaries of the space and the nature of the membrane, all contained in the idle callback:

```
void animate(void)
{
   #define HEAVYLEFT  0.8
   #define HEAVYRIGHT 0.9
   #define LIGHTLEFT  0.3
   #define LIGHTRIGHT 0.1

   #define LEFT  0
   #define RIGHT 1
   int i,sign,whichside;

   for(i=0; i<NHEAVY; i++) {
      if (heavy[i][0] < 0.0) whichside = LEFT; else whichside = RIGHT;
      sign = rand()%2; if (sign == 0) sign = -1;
      heavy[i][0] += sign*(float)(rand()%GRAN)/(float)(16*GRAN);
      sign = rand()%2; if (sign == 0) sign = -1;
      heavy[i][1] += sign*(float)(rand()%GRAN)/(float)(16*GRAN);
      sign = rand()%2; if (sign == 0) sign = -1;
      heavy[i][2] += sign*(float)(rand()%GRAN)/(float)(16*GRAN);
      if (heavy[i][0] > 1.0) heavy[i][0] = 2.0 - heavy[i][0];
      if ((whichside==RIGHT)&&(heavy[i][0] < 0.0)) // cross right to left?
         if ( (float)(rand()%GRAN)/(float)(GRAN) >= HEAVYLEFT)
            heavy[i][0] = -heavy[i][0];
      if ((whichside==LEFT)&&(heavy[i][0] > 0.0))  // cross left to right?
         if ( (float)(rand()%GRAN)/(float)(GRAN) >= HEAVYRIGHT)
            heavy[i][0] = -heavy[i][0];
      if (heavy[i][0] > 1.0) heavy[i][0] = 2.0 - heavy[i][0];
      if (heavy[i][0] <-1.0) heavy[i][0] =-2.0 - heavy[i][0];
      if (heavy[i][1] > 1.0) heavy[i][1] = 2.0 - heavy[i][1];
      if (heavy[i][1] < 0.0) heavy[i][1] = -heavy[i][1];
      if (heavy[i][2] > 1.0) heavy[i][2] = 2.0 - heavy[i][2];
      if (heavy[i][2] < 0.0) heavy[i][2] = -heavy[i][2];
   }
   for(i=0; i<NLIGHT; i++) {
      if (light[i][0] < 0.0) whichside = LEFT; else whichside = RIGHT;
      sign = rand()%2; if (sign == 0) sign = -1;
      light[i][0] += sign*(float)(rand()%GRAN)/(float)(16*GRAN);
      sign = rand()%2; if (sign == 0) sign = -1;
      light[i][1] += sign*(float)(rand()%GRAN)/(float)(16*GRAN);
      sign = rand()%2; if (sign == 0) sign = -1;
      light[i][2] += sign*(float)(rand()%GRAN)/(float)(16*GRAN);
      if (light[i][0] > 1.0) light[i][0] = 2.0 - light[i][0];
      if ((whichside == RIGHT)&&(light[i][0] < 0.0))
         if ( (float)(rand()%GRAN)/(float)(GRAN) >= LIGHTLEFT)
            light[i][0] = -light[i][0];
```

```
            if ((whichside == LEFT)&&(light[i][0] > 0.0))
               if ( (float)(rand()%GRAN)/(float)(GRAN) >= LIGHTRIGHT)
                  light[i][0] = -light[i][0];
            if (light[i][0] <-1.0) light[i][0] =-2.0 - light[i][0];
            if (light[i][1] > 1.0) light[i][1] = 2.0 - light[i][1];
            if (light[i][1] < 0.0) light[i][1] = -light[i][1];
            if (light[i][2] > 1.0) light[i][2] = 2.0 - light[i][2];
            if (light[i][2] < 0.0) light[i][2] = -light[i][2];
         }
         glutPostRedisplay();
      }
```

This display, shown in Figure 5.7, has with pretty effective animation of the particles and allows the user to rotate the simulation region to view the behavior from any direction. The program presents the tally results in a separate text-output region which is included with the figure at left below. The tallys were taken at approximately one-minute intervals to illustrate that the system settles into a fairly consistent steady-state behavior over time, certainly something that one would want in a simulation. The membrane is presented with partial transparency so that the user gets a sense of seeing into the whole space when viewing the simulation with arbitrary rotations.

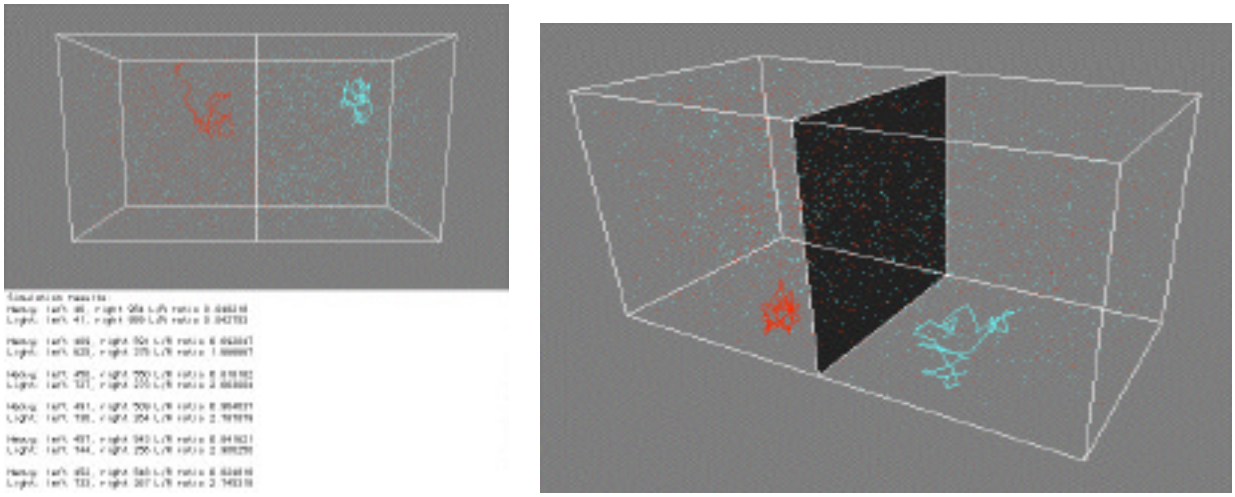


Figure 5.7: display of the diffusion simulation, directly across the membrane at left (including data output from the simulation), and in a view oriented to show the membrane itself at right.

# The OpenGL Pipeline

*Prerequisites*

An understanding of graphics primitives that will enable the student to see how the primitives are handled by the pipeline operations discussed in this chapter.

*Introduction*

In an earlier chapter we saw the outline of the graphics pipeline at a rather high level. In this chapter we will look at the pipeline in more detail, focusing on the implementation of the pipeline in the OpenGL system.

*The Pipeline*

The OpenGL system is defined in terms of the processing described by the first figure below. In this figure, system input comes from the OpenGL information handled by the processor and the output is finished pixels in the frame buffer. The input information consists of geometric vertex information that goes into the display list, polynomial information from evaluators that goes to the evaluator, and texture information that goes through pixel operations into the texture memory. We will outline the various stages of the system's operations to understand how your geometry specification is turned into the image in the frame buffer.

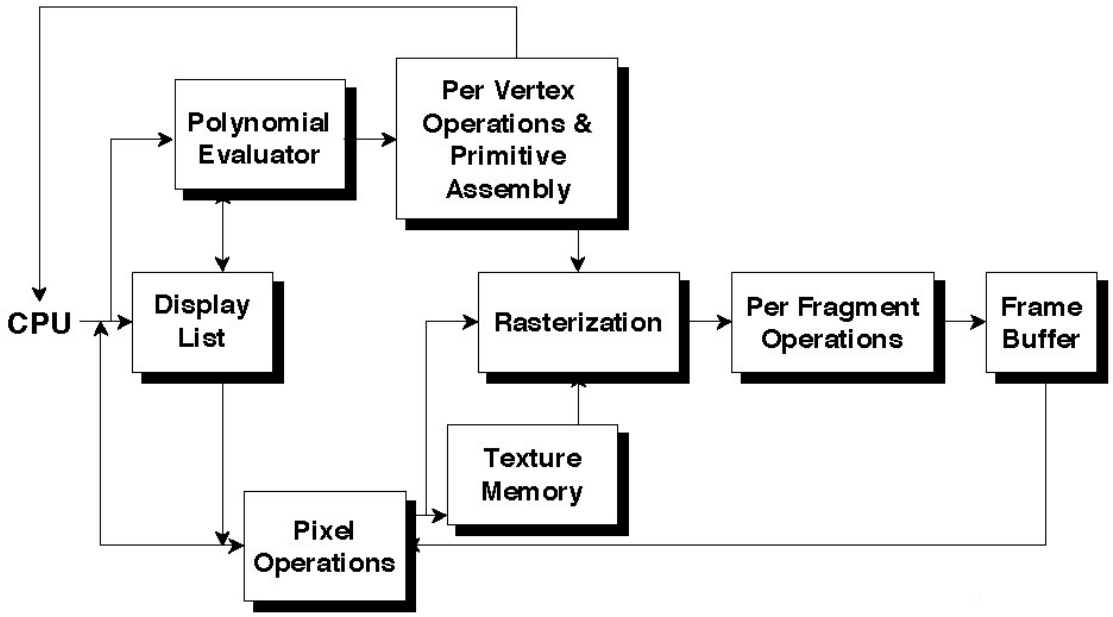



Figure: the OpenGL system model

Let us begin with a simple polygon. The 3D vertex geometry that you specify is passed into the display list and is then forwarded to the per-vertex operations. Here the modelview matrix, which includes both the model transformation and the viewing transformation in OpenGL, is applied to the vertices in modeling coordinates to transform them into 3D eye coordinates, and clipping is done against the edges of the view volume or against other planes if enabled. Next the projection transformation is applied to transform them into 2D eye coordinates. If texture and lighting have been enabled, further operations compute texture coordinates for the vertex and apply the lighting model to calculate the color for the vertex. When this step is finished, each vertex in 2D eye space

is completely prepared for rasterization, including having its color, depth, and optionally texture coordinates calculated.

In the rasterization step, the polygon itself is filled in by an interpolation process based on the vertex coordinates. For each scan line of the frame buffer that meets the polygon, the endpoints of the scan are interpolated from appropriate vertices and each pixel between them has its color, depth, and optionally texture coordinates computed to create what is called a fragment. These fragments are then passed to the per-fragment operations.

Much of the power of the OpenGL system lies in its treatment of these fragments. A fragment first has any texturing done, where the texture coordinates for the fragment are used to read the texture memory and determine what texture color to apply to the fragment. Fog calculations are done next, and then a series of tests are applied as described in the following figure:

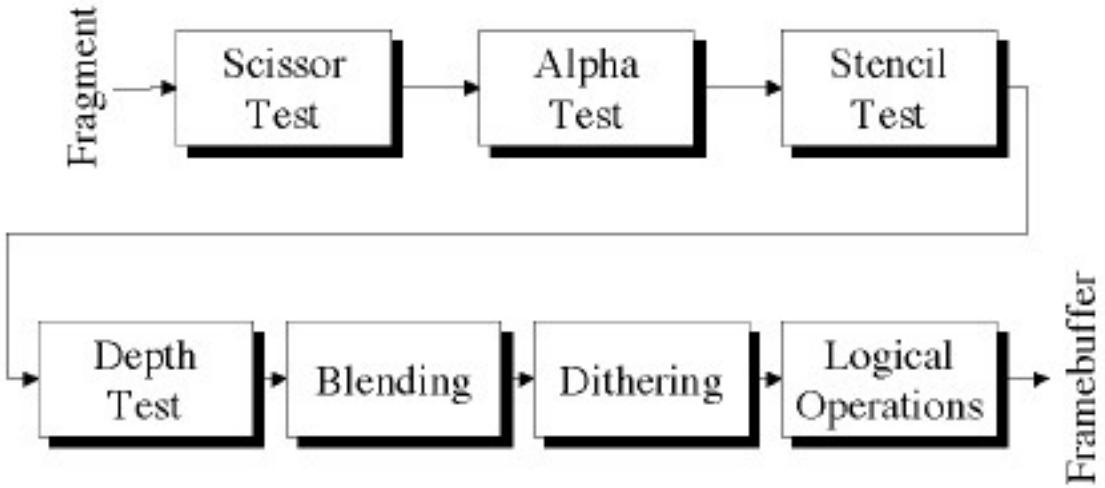



Figure: details of fragment processing

This series of tests determines whether a fragment will be visible and, if it is, how it will be treated as it moves to determine a pixel in the frame buffer. Details on most of these various operations will be covered in the next few chapters.

While we have been discussing the actions on vertex points, there are other operations in the OpenGL system. We briefly mentioned the use of the polynomial evaluator; this comes into play when we are dealing with splines and define evaluators based on a set of control points. These evaluators may be used for geometry or for a number of other graphic components, and here is where the polynomial produced by the evaluator is handled and its results are made available to the system for use. This is discussed in the chapter on spline modeling below.

Another area we have glossed over is the per-pixel operations. In this area information from a texture map (an array in memory) is translated into information that can be used for texture mapping. The arrow from the frame buffer back to the pixel operations indicates that we can take information from the frame buffer and write it into another part of the frame buffer or even make it into a texture map itself.

*Implementation in Graphics Cards*

The system described above is very general and describes the behavior required to implement the OpenGL processes. In practice, the system is implemented in many ways, and the diagram below shows the implementation in a typical graphics card ...

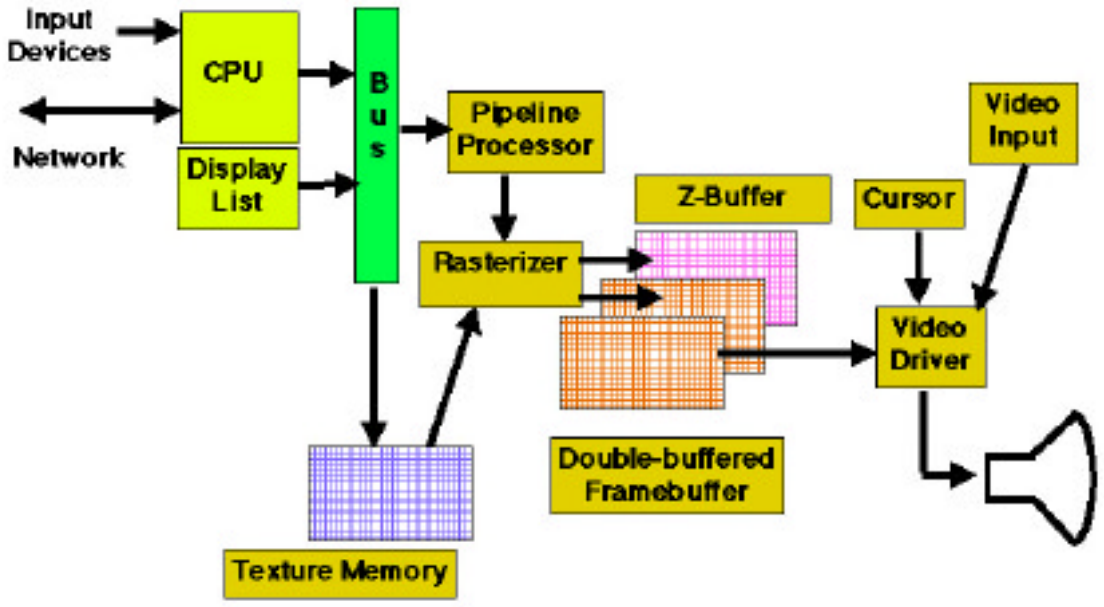



Figure: an implementation of the OpenGL system in a typical graphics card

…

# Lights and Lighting

*Prerequisites*

An understanding of color at the level of the discussion of the chapter on color in these notes, and some observation of the way lights work in creating the images of the world around you.

*Introduction*

There are two ways to think of how we see things. The first is that things have an intrinsic color and we simply see the color that they are. The color is set by defining a color in RGB space and simply instructing the graphics system to draw everything with that color until a new color is chosen. This approach is synthetic and somewhat simpleminded, because it's clear that things in the real world don't behave like this; they have a color, but the color they have is strongly influenced by the light that illuminates them, so it's somehow wrong not to take the light into account in presenting a scene.

The second way to think of how we see things is to realize that seeing involves light that reaches us because of physical interaction between light and the objects we see. Light sources emit energy that reaches the objects and is then re-sent to us in ways that involve the color of the light and the physical properties of the objects. In this chapter we will discuss how this way of thinking about light allows us to model how we see the world, and how computer graphics distills that into fairly simple definitions and relatively straightforward computations to produce images that seem to have a relationship to the lights and materials present in the scene.

A first step in developing graphics that depends on lights and materials is to create a model of the nature of light. In the simplified but workable model of light that we will use, there are three fundamental components of light: the ambient, diffuse, and specular components. We can think of each of these as follows:

- ambient: light that is present in the scene because of the overall illumination in the space. This can include light that has bounced off objects in the space and that is thus independent of any particular light.
- diffuse: light that comes directly from a particular light source to an object, where it is then sent directly to the viewer. Normal diffuse light comes from an object that reflects a subset of the wavelengths it receives and that depends on the material that makes up an object, with the effect of creating the color of the object.
- specular: light that comes directly from a particular light source to an object, where it is then reflected directly to the viewer because the object reflects the light without interacting with it and giving the light the color of the object. This light is generally the color of the light source, not the object that reflects the light.

All three of these components contribute to the light at every wavelength, and the graphics system models this by applying them to the RGB components separately. The sum of these light components is the light that is actually seen from an object.

Just as the light is modeled to create an image, the materials of the objects that make up the scene are modeled in terms of how they contribute to the light. Each object will have a set of properties that defines how it responds to each of the three components of light. The ambient property will define the behavior (essentially, the color) of the object in ambient light, the diffuse property in diffuse light, and the specular property in specular light. As we noted above, realistic lighting tends to assume that objects behave the same in ambient and diffuse light, and that objects simply take on the light color in specular light, but because of the kind of calculations that are done to display a lighted image, it is as easy to treat each of the light and material properties separately.

Thus in order to use lights in a scene, you must define your lights in terms of the three kinds of color they provide, and you must define objects not in simple terms of their color, but in terms of their material properties. This will be different from the process we saw in the earlier module on color but the changes will be something you can handle without difficulty. The OpenGL API has its own way to specify the three components of light and the three components of materials, and we will also discuss this below when we talk about implementing lighting for your work.

*Definitions*

Ambient, diffuse, and specular light

*Ambient* light is light that comes from no apparent source but is simply present in a scene. This is the light you would find in portions of a scene that are not in direct light from any of the lights in the scene, such as the light on the underside of an object, for example. Ambient light can come from each individual light source, as well as from an overall ambient light value, and you should plan for each individual light to contribute to the overall brightness of a scene by contributing something to the ambient portion of the light. The amount of diffuse light reflected by an object is given simply by $A=L_A*C_A$ for a constant $C_A$ that depends on the material of the object and the ambient light $L_A$ present in the scene, where the light $L_A$ and constant $C_A$ are to be thought of as RGB triples, not simple constants, and the calculation is to yield another RGB value. Ambient light is usually fairly low-level if you want to emphasize the effect of the lights (you might think of this as a night effect) or fairly high-level if you want to see everything in the scene with a fairly uniform light (this would be a brightly-lit effect). If you want to emphasize shapes, use a fairly low ambient light.

*Diffuse* light comes from specific light sources and is reflected by the surface of the object at a particular wavelength depending on properties of the object's material. The OpenGL model for diffuse light is based on the concept that the intensity of light seen on a surface is proportional to the amount of light falling on a unit of the surface area. This is proportional to the cosine of the angle between the surface normal and the light direction as long as that cosine is positive, and zero otherwise, as illustrated in the diagram in Figure 9.1. As the angle of incidence of the light on the surface decreases, the amount of light reflected from the surface becomes dimmer, going to zero when the light is parallel to the surface. Because it is impossible to talk about "negative light," we replace any negative value of the cosine with zero, which eliminates diffuse light on surfaces facing away from the light. The diffuse lighting calculation computes the amount of diffuse light as

$$D = L_D*C_D*\cos(\Theta) = L_D*C_D*(L \bullet N)$$

for the value of the diffuse light $L_D$ from each light source and the ambient property of the material $C_D$, which shows why we must have surface normals in order to calculate diffuse light. This computation is done separately for each light source and each object, because it depends on the angle from the object to the light. It is important to note that diffuse light is independent of the point from which we view the material, and thus the location of the eye point does not participate in

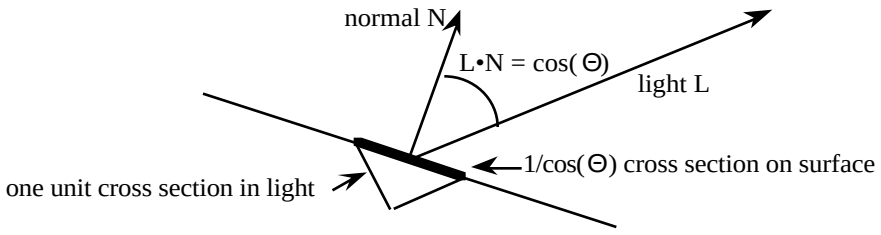



Figure 9.1: diffuse lighting

the diffuse light computation. This is obvious from looking at the world around us, because things do not change color as the angle between their normals and our eye direction does not change as we move around. The lighting model supports this behavior, as is seen by noting that for the unit area in Figure 9.1, the geometric size of that area as seen from any point decreases in exactly the same way the diffuse light diminishes as the angle $\Theta$ grows from 0 to $\pi/2$, so the brightness of the surface (the light per unit area) remains the same.

*Specular* light is a surface phenomenon that provides shiny highlights. These depend on the smoothness and electromagnetic properties of the surface, so smooth metallic objects (for example) reflect light well. The energy in specular light is not absorbed by the object and re-radiated, but is reflected with the angle of incidence equal to the angle of reflection, as illustrated in the left-hand diagram of Figure 9.2. Such light may have a small amount of "spread" as it leaves the object, depending on the shininess of the object, so the standard model for specular light allows you to define the shininess of an object to control that spread. Shininess is controlled by a parameter

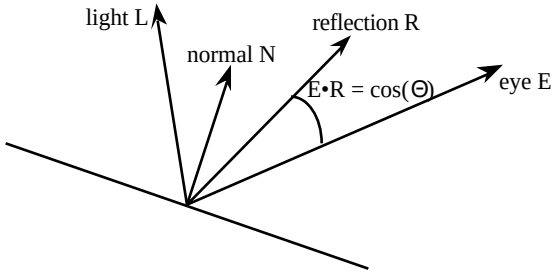



Figure 9.2: specular lighting

which gives smaller, brighter highlights as it increases, as shown in the three successive figures of Figure 9.3.

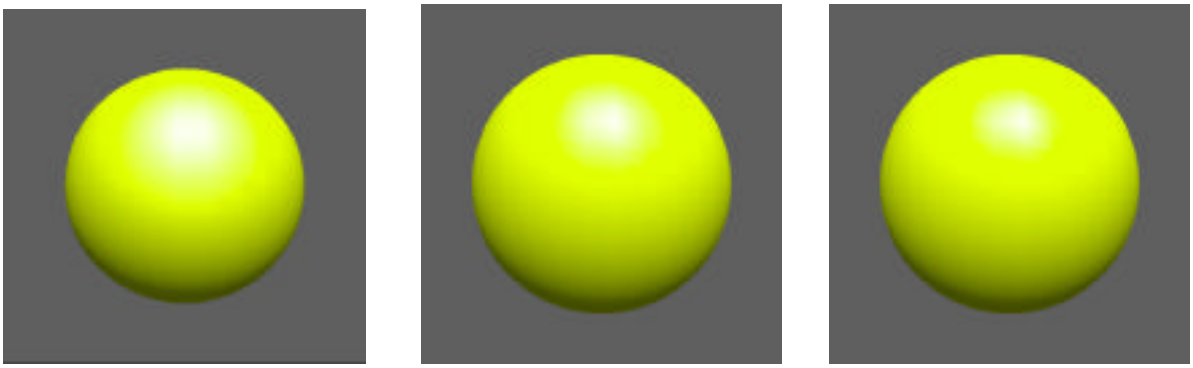


Figure 9.3: specular highlights with shininess coefficients 20, 50, and 80 (left, center, and right), respectively

The specular light seen on an object in the image is given by

$$S = L_S * C_S * \cos^N(\Theta) = L_S * C_S * (E \bullet R)^N$$

for a light's specularity $L_S$ and the object's specular coefficient $C_S$. Note that this depends on the angle between the eye and the light reflection, because specular light is light that is reflected directly by reflection from the surface of an object. In addition, the computation involves the value of the *shininess coefficient* $N$ that depends on the light and on the material. Note the visual effect of increasing the shininess coefficient: the highlight gets smaller and more focused—that is, the

sphere looks shinier and more polished. This produces a fairly good model of shininess, because the function $\cos^N(\Theta)$ has value very near one if the angle $\Theta$ is small, and drops off quickly as the angle increases, with the speed of the dropoff increasing as the power is increased. The specular light computation is done separately for each light and each object, because it depends on the angle from the object to the light (as well as the angle to the eye point). This calculation depends fundamentally on both the direction from the object to the light and the direction of the object to the eye, so you should expect to see specular light move as objects, lights, or your eye point moves.

Because both diffuse and specular lighting need to have normals to the surface at each vertex, we need to remind ourselves how we get normals to a surface. One way to do this is analytically; we know that the normal to a sphere at any given point is in the direction from the center of the sphere to the point, so we need only know the center and the point to calculate the normal. In other cases, such as surfaces that are the graph of a function, it is possible to calculate the directional derivatives for the function at a point and take the cross product of these derivatives, because the derivatives define the tangent plant to the surface at the point. But sometimes we must calculate the normal from the coordinates of the polygon, and this calculation was described in the discussion of mathematical fundamentals by taking the cross product of two adjacent edges of the polygon in the direction the edges are oriented.

So with the mechanics of computing these three light values in hand, we consider the constants that appeared in the calculations above. The ambient constant is the product of the ambient light component and the ambient material component, each calculated for the red, green, and blue parts respectively. Similarly the diffuse and specular constants are the products of their respective light and material components. Thus a white light and any color of material will produce the color of the material; a red light and a red material will produce a red color; but a red light and a blue material will produce a black color, because there is no blue light to go with the blue material and there is no red material to go with the red light. The final light at any point is the sum of these three parts: the ambient, diffuse, and specular values, each computed for all three RGB components. If any component has a final value larger then one, it is clamped to have value 1.

When you have multiple lights, they are treated additively — the ambient light in the scene is the sum of any overall ambient light for the entire scene plus the ambient lights of the individual lights, the diffuse light in the scene is the sum of the diffuse lights of the individual lights, and the specular light in the scene is the sum of the diffuse lights of the individual lights. As above, if these sums exceed one in any one component, the value is clamped to unity.

As we saw above, you need to calculate normals to the surface in order to compute diffuse and specular light. This is often done by defining normal vectors to the surface in the specifications of the geometry of a scene to allow the lighting computation to be carried out. Processes for computing normals were described in the early chapter on mathematical fundamentals. These can involve analysis of the nature of the object, so you can sometimes compute exact normals (for example, if you are displaying a sphere, the normal at any point has the same direction as the radius vector). If an analytic calculation is not available, normals to a polygonal face of an object can be computed by calculating cross products of the edges of the polygon. However, it is not enough merely to specify a normal; you need to have unit normals, normal vectors that are exactly one unit long (usually called normalized vectors). It can be awkward to scale the normals yourself, and doing this when you define your geometry may not even be enough because scaling or other computations can change the length of the normals. In many cases, your graphics API may provide a way to define that all normals are to be normalized before they are used.

In the next chapter we will discuss shading models, but here we need to note that all our lighting computations assume that we are calculating the light at a single vertex on a model. If we choose to do this calculation at only one point on each polygon, we can only get a single color for the

polygon, which leads to the kind of lighting called flat shading. If we wanted to do smooth shading, which can give a much more realistic kind of image, we would need to determine a separate normal for each vertex so that the lighting computation could give us a color for each vertex. If the vertex is part of several polygons and we want to calculate a normal for the vertex that we can use for all the polygons, we can calculate a separate normal based on each of the polygons and then average them to get the normal for the vertex. The individual colors for the vertices are then used to calculate colors for all the points in the polygon, as is discussed in more detail in the next chapter.

Note that none of our light computation handles shadows, however, because shadows depend on the light that reaches the surface, which is a very different question from the way light is reflected from the surface. Shadows are difficult and are handled in OpenGL with very specialized programming which we will not cover in these notes.

Use of materials

As we said earlier, lighting involves two parts: both the specification of the lighting properties of the objects in the scene, and the specification of the lights in a scene. If you want to use lighting in creating a scene, you must specify both of these. Here we discuss material specifications, and we follow this by discussing light properties, but implementing lighting involves putting these all together as is discussed in the example at the end of this chapter.

As we saw above, each object participates in determining the reflected light that makes up its color when it is displayed. In the discussion of the three components of light, we saw four constants $C_A$, $C_D$, $C_S$, and $N$ that are part of the computations of the light. The first three of these constants have separate RGB components and together the four constants identify the way a material interacts with light, so they are often called the set of definitions of the material. They need to be defined for each object in your scene in order to allow the lighting calculations to be carried out. Your graphics API will allow you to see these as part of your modeling work; they should be considered as part of the appearance information you would include in a shape node in your scene graph.

All the discussion of lighting above assumed that an object is reflective, but an object can also be *emissive* — that is, send out light of its own. Such a light simply adds to the light of the object but does not add extra light to the scene, allowing you to define a bright spot to present something like an actual light in the scene. This is managed by defining a material to have an emissive light property, and the final lighting calculations for this material adds the components of the light emission to the other lighting components when the object's color is computed.

*Light properties*

Lights are critical components of your modeling work in defining an image, as we saw in the discussion of lights in the scene graph in the modeling chapter. Along with the location of each light, which is directly supported by the scene graph, you will want to define other aspects of the light, and these are discussed in this section.

Your graphics API allows you to define a number of properties for a light. Typically, these can include its position or its direction, its color, how it is attenuated (diminished) over distance, and whether it is an omnidirectional light or a spotlight. We will cover these properties lightly here but will not go into depth on them all, but the properties of position and color are critical. The other properties are primarily useful is you are trying to achieve a particular kind of effect in your scene. The position and color properties are illustrated in the example at the end of this chapter.

Positional lights

When we want a light that works as if it were located within your scene, you will want your light to have an actual position in the scene. To define a light that has position, you will set the position as a four-tuple of values whose fourth component is non-zero (typically, you will set this to be 1.0). The first three values are then the position of the light and all lighting calculations are done with the light direction from an object set to the vector from the light position to the object.

Spotlights

Unless you specify otherwise, a positional light will shine in all directions. If you want a light that shines only in a specific direction, you can define the light to be a spotlight that has not only a position, but also other properties such as a direction, a cutoff, and a dropoff exponent, as you will see from the basic model for a spotlight shown in Figure 9.4. The direction is simply a 3D vector that is taken to be parallel to the light direction, the cutoff is assumed to be a value between 0.0 and 90.0 that represents half the spread of the spotlight and determines whether the light is focused tightly or spread broadly (a smaller cutoff represents a more focused light), and the dropoff exponent controls how much the intensity drops off between the centerline of the spotlight and the intensity at the edge.

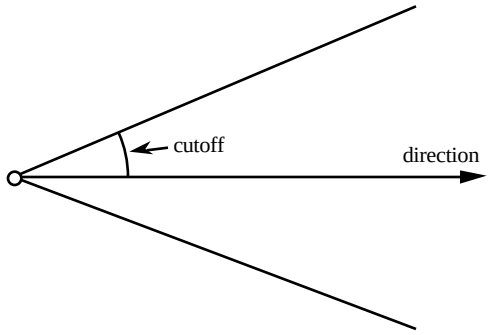



Figure 9.4: spotlight direction and cutoff

Attenuation

The physics of light tells us that the energy from a light source on a unit surface diminishes as the square of the distance from the light source from the surface. This diminishing is called attenuation, and computer graphics can model that behavior in a number of ways. An accurate model would deal with the way energy diffuses as light spreads out from a source which would lead to a light that diminishes as the square of the distance from the light, and the graphics system would diminish the intensity of the light accordingly. However, the human perceptual system is more nearly logarithmic than linear in the way we see light, so we do not recognize this kind of diminishing light as realistic, and we probably would need to use an attenuation that drops off more slowly. Your graphics API will probably give you some options in modeling attenuation.

Directional lights

Up to now, we have talked about lights as being in the scene at a specific position. When such lights are used, the lighting model takes the light direction at any point as the direction from the light position to that point. However, if we were looking for an effect like sunlight, we want light that comes from the same direction at all points in the scene. In effect, we want to have a light at infinity. If your graphics API supports directional lights, there will be a way to specify that the

light is directional instead of positional and that defines the direction from which the light will be received.

Positioning and moving lights

Positional lights can be critical components of a scene, because they determine how shapes and contours can be seen. As we noted in the chapter on modeling, lights are simply another part of the model of your scene and affected by all the transformations present in the modelview matrix when the light position is defined. A summary of the concepts from the scene graph will help remind us of the issues here.

- If the light is to be at a fixed place in the scene, then it is at the top level of the scene graph and you can define its position immediately after you set the eye point. This will create a position for the light that is independent of the eye position or of any other modeling in the scene.
- If the light is to be at a fixed place relative to the eye point, then you need to define the light position and other properties before you define the eye position. The light position and properties are then modified by the transformations that set the eye point, but not by any subsequent modeling transformations in the scene.
- If the light is to be at a fixed place relative to an object in the scene, then you define the light position as a branch of the group node in the scene graph that defines the object. Anything that affects the object will then be done above that group node, and will affect the light in the same way as it does the rest of the object.
- If the light is to move around in the scene on its own, then the light is a content node of the scene graph and the various properties of the light are defined as that node is set.

The summary of this modeling is that a positional light is treated simply as another part of the modeling process and is managed in the same way as any other object would be.

*Lights and materials in OpenGL*

Several times above we suggested that a graphics API would have facilities to support several of the lighting issues we discussed. Here we will outline the OpenGL support for lighting and materials so you can use these capabilities in your work. In some of these we will use the form of the function that takes separate R, G, and B parameters (or separate X, Y, and Z coordinates), such as `glLightf(light, name, set_of_values)`, while in others we will use the vector form that takes 3-dimensional vectors for colors and points, but in come cases we will use the vector form such as `glLightfv(light, name, vector_values)`, and you may use whichever form fits your particular design and code best.

Specifying and defining lights

When you begin to plan your scene and are designing your lighting, you may need to define your light model with the `glLightModel(...)` function. This will allow you to define some fundamental properties of your lighting. Perhaps the most important use of this function is defining is whether your scene will use one-sided or two-sided lighting, which is chosen with the function

```
glLightModel[f|i](GL_LIGHT_MODEL_TWO_SIDE, value).
```

where `[f|i]` means that you use either the letter `f` or the letter `i` to indicate whether the parameter value is real or integer. If the (real or integer) value of the numeric parameter is 0, one-sided lighting is used and only the front side of your material is lighted; if the value is non-zero, both front and back sides of your material are lighted. Other uses of the function include setting a global ambient light, discussed below, and choosing whether specular calculations are done by assuming the view direction is parallel to the Z-axis or the view direction is towards the eye point This is determined by the function

```
glLightModel[f|i](GL_LIGHT_MODEL_LOCAL_VIEWER, value),
```

with a value of 0 meaning that the view direction is parallel to the Z-axis and non-zero that it is toward the origin. The default value is 0.

OpenGL allows you to define up to eight lights for any scene. These lights have the symbolic names `GL_LIGHT0` … `GL_LIGHT7`, and you create them by defining their properties with the `glLight*(...)` functions before they are available for use. You define the position and color of your lights (including their ambient, specular, and diffuse contributions) as illustrated for the light `GL_LIGHT0` by the following position definition and definition of the first of the three lights in the three-light example

```
glLightfv(GL_LIGHT0, GL_POSITION, light_pos0 ); // light 0
glLightfv(GL_LIGHT0, GL_AMBIENT,  amb_color0 );
glLightfv(GL_LIGHT0, GL_DIFFUSE,  diff_col0 );
glLightfv(GL_LIGHT0, GL_SPECULAR, spec_col0 );
```

Here we use a light position and specific light colors for the specular, diffuse, and ambient colors that we must define in separate statements such as those below.

```
GLfloat light_pos0 = { ..., ..., ... };
GLfloat diff_col0 = { ..., ..., ... };
```

In principle, both of these vectors are four-dimensional, with the fourth value in the position vector being a homogeneous coordinate value and with the fourth value of the color vector being the alpha value for the light. We have not used homogeneous coordinates to describe our modeling, but they are not critical for us. We have used alpha values for colors, of course, but the default value for alpha in a color is 1.0 and unless you want your light to interact with your blending design somehow, we suggest that you use that value for the alpha component of light colors, which you can do by simply using RGB-only light definitions as we do in the example at the end of this chapter.

As we noted earlier in this chapter, you must define normals to your objects' surfaces for lighting to work successfully. Because the lighting calculations involve cosines that are calculated with dot products with the normal vector, however, you must make sure that your normal vectors are all of unit length. You can ensure that this is the case by enabling automatic normalization with the function call `glEnable(GL_NORMALIZE)` before any geometry is specified in your display function.

Before any light is available to your scene, the overall lighting operation must be enabled and then each of the individual lights to be used must also be enabled. This is an easy process in OpenGL. First, you must specify that you will be using lighting models by invoking the standard enable function

```
glEnable(GL_LIGHTING);   // so lighting models are used
```

Then you must identify the lights you will be using by invoking an enable function for each light, as illustrated by the following setup of all three lights for the three-light case of the example below:

```
glEnable(GL_LIGHT0);     // use LIGHT0
glEnable(GL_LIGHT1);     // and LIGHT1
glEnable(GL_LIGHT2);     // and LIGHT2
```

Lights may also be disabled with the `glDisable(...)` function, so you may choose when to have a particular light active and when to have it inactive in an animation or when carrying out a particular display that may be chosen, say, by a user interaction.

In addition to the ambient light that is contributed to your scene from each of the individual lights' ambient components, you may define an overall ambient light for the scene that is independent of any particular light. This is done with the function:

```
glLightModelf(GL_LIGHT_MODEL_AMBIENT, r, g, b, a)
```

and the value of this light is added into the overall ambient lighting computation.

The remaining properties of lights that we discussed earlier in this chapter are also straightforward to set in OpenGL. If you want a particular light to be a spotlight, you will need to set the direction, cutoff, and dropoff properties that we described earlier in this chapter, as well as the standard position property. These additional properties are set with the `glLightf*(...)` functions as follows:

```
glLightf(light, GL_SPOT_DIRECTION, -1.0, -1.0, -1.0);
glLightf(light, GL_SPOT_CUTOFF, 30.0);
glLightf(light, GL_SPOT_EXPONENT, 2.0);
```

If you do not specify the spotlight cutoff and exponent, these are 180 degrees (which means that the light really isn't a spotlight at all) and the exponent is 0. If you do set the spotlight cutoff, the value is limited to lie between 0 and 90, as we described earlier.

Attenuation is not modeled realistically by OpenGL, but is set up in a way that can make it useful. There are three components to attenuation: constant, linear, and quadratic. The value of each is set separately as noted above with the symbolic constants `GL_CONSTANT_ATTENUATION`, `GL_LINEAR_ATTENUATION`, and `GL_QUADRATIC_ATTENUATION`. If these three attenuation coefficients are $A_C$, $A_L$, and $A_Q$, respectively, and the distance of the light from the surface is `D`, then the light value is multiplied by the attenuation factor

$$A = 1/( A_C + A_L * D + A_Q*D^2 )$$

where D is the distance between the light position and the vertex where the light is calculated. The default values for $A_C$, $A_L$, and $A_Q$ are 1.0, 0.0, and 0.0 respectively. The actual values of the attenuation constants can be set by the `glLightf(GL_*_ATTENUATION, value)` functions, where the wildcard is to be replaced by one of the three symbolic constants mentioned above.

A directional light is specified by setting the fourth component in its position to be zero. The direction of the light is set by the first three components, and these are transformed by the modelview matrix. Such lights cannot have any attenuation properties but otherwise work just like any other light: its direction is used in any diffuse and specular light computations but no distance is ever calculated. An example of the way a directional light is defined would be

```
glLightf(light, GL_POSITION, 10.0, 10.0, 10.0, 0.);
```

Defining materials

In order for OpenGL to model the way a light interacts with an object, the object must be defined in terms of the way it handles ambient, diffuse, and specular light. This means that you must define the color of the object in ambient light and the color in diffuse light. (No, we can't think of any cases where these would be different, but we can't rule out the possibility that this might be used somehow.) You do not define the color of the object in specular light, because specular light is the color of the light instead of the color of the object, but you must define the way the material handles the specular light, which really means how shiny the object is and what color the shininess will be. All these definitions are handled by the `GL_MATERIAL*` function.

Recall that any polygon has two sides, which we will call the *front* side and *back* side. The difference between these is the direction of the normal to the polygon, with the front side being the side toward which the normal points. Because the normal can represent the direction of the cross product of two polygon edges in the order in which edges go around the polygon, and because of the right-hand rule for determining the direction of the cross product, we can avoid the reference to the polygon normal and simply note that the front side is the side from which the edges of the polyon are in counterclockwise order (or the side for which the angles from an interior point of the polygon to the vertices are in increasing order).

If you use two-sided lighting, when you specify the properties for your material, you must specify them for both the front side and the back side of the material. You can choose to make these properties the same by defining your material with the parameter `GL_FRONT_AND_BACK` instead of defining `GL_FRONT` and `GL_BACK` separately. This will allow you to use separate colors for the front side and back side of an object, for example, and make it clear which side is being seen in case the object is not closed.

To allow us to define an object's material properties we have the `glMaterial*(...)` function family. These functions have the general form

```
glMaterial[i|f][v](face, parametername, value)
```

and can take either integer or real parameter values (`[i|f]`) in either individual or vector (`[v]`) form. The parameter `face` is a symbolic name that must be one of `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`. The value of `parametername` is a symbolic name whose values can include `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`, `GL_EMISSION`, `GL_SHININESS`, or `GL_AMBIENT_AND_DIFFUSE`. Finally, the `value` parameter is either a single number, a set of numbers, or a vector that sets the value the symbolic parameter is to have in the OpenGL system. Below is a short example of setting these values, taken from the example at the end of the chapter.

```
GLfloat shininess[]={ 50.0 };
GLfloat white[] = { 1.0, 1.0, 1.0, 1.0};
glMaterialfv(GL_FRONT, GL_AMBIENT,   white );
glMaterialfv(GL_FRONT, GL_DIFFUSE,   white );
glMaterialfv(GL_FRONT, GL_SPECULAR,  white );
glMaterialfv(GL_FRONT, GL_SHININESS, shininess );
```

This gives the material a very neutral property that can pick up whatever colors the light should provide for its display.

Most of the parameters and values are familiar from the earlier discussion of the different aspects of the lighting model, but the `GL_AMBIENT_AND_DIFFUSE` parameter is worth pointing out because it is very common to assume that a material has the same properties in both ambient and diffuse light. (Recall that in both cases, the light energy is absorbed by the material and is then re-radiated with the color of the material itself.) This parameter allows you to define both properties to be the same, which supports this assumption.

Setting up a scene to use lighting

To define a triangle with vertex points `P[0]`, `P[1]`, and `P[2]`, compute its normal, and use the calculated normal, we would see code something like this:

```
glBegin(GL_POLYGON);
     // calculate the normal Norm to the triangle
     calcTriangleNorm(p[0],P[1],P[2],Norm);
     glNormal3fv(Norm);
     glVertex3fv(P[0]);
     glVertex3fv(P[1]);
     glVertex3fv(P[2]);
glEnd();
```

Using GLU quadric objects

As we discussed when we introduced the GLU quadric objects in the modeling chapter, the OpenGL system can generate automatic normal vectors for these objects. This is done with the function `gluQuadricNormals(GLUquadric* quad, GLenum normal)` that allows you to set `normal` to either `GLU_FLAT` or `GLU_SMOOTH`, depending on the shading model you want to use for the object.

An example: lights of all three primary colors applied to a white surface

Some lighting situations are easy to see — when you put a white light on a colored surface, you see the color of the surface, because the white light contains all the light components and the surface has the color it reflects among them. Similarly, if you shine a colored light on a white surface, you see the color of the light because only that color is available. When you use a colored light on a colored surface, however, it gets much more complex because a surface can only reflect colors that come to it. So if you shine a (pure) red light on a (pure) green surface you get no reflection at all, and the surface seems black. You don't see this in the real world because you don't see lights of pure colors, but it can readily happen in a synthetic scene.

Considering the effect of shining colored lights on a white surface, let's look at an example. A white surface will reflect all the light that it gets, so if it gets only a red light, it should be able to reflect only red. So if we take a simple shape (say, a cube) in a space with three colored lights (that are red, green, and blue, naturally), we should see it reflect these different colors. In the threelightcube example we discuss below, we define three lights that shine from three different directions on a white cube. If you add code that lets you rotate the cube around to expose each face to one or more of the three lights, you will be able to see all the lights on various faces and to experiment with the reflection properties they have. This may let you see the effect of having two or three lights on one of the faces, as well as seeing a single light. You may also want to move the lights around and re-compile the code to achieve other lighting effects.

There is a significant difference between the cube used in this example and the cube used in the simple lighting example in a previous module. This cube includes not only the vertices of its faces but also information on the normals to each face. (A normal is a vector perpendicular to a surface; we are careful to make all surface normals point away from the object the surface belongs to.) This normal is used for many parts of the lighting computations — to determine whether you're looking at a front or back face, for example, and to compute both the diffuse light and the specular light for a polygon. We refer you to any standard graphics text for more details.

*Code for the example*

Defining the light colors and positions in the initialization function:

```
GLfloat light_pos0[]={  0.0, 10.0,  2.0, 1.0 }; // light 1: up y-axis
GLfloat light_col0[]={  1.0,  0.0,  0.0, 1.0 }; // light is red
GLfloat amb_color0[]={  0.3,  0.0,  0.0, 1.0 }; // even ambiently

GLfloat light_pos1[]={  5.0, -5.0,  2.0, 1.0 }; // light 2: lower right
GLfloat light_col1[]={  0.0,  1.0,  0.0, 1.0 }; // light is green
GLfloat amb_color1[]={  0.0,  0.3,  0.0, 1.0 }; // even ambiently

GLfloat light_pos2[]={ -5.0,  5.0,  2.0, 1.0 }; // light 3: lower left
GLfloat light_col2[]={  0.0,  0.0,  1.0, 1.0 }; // light is blue
GLfloat amb_color2[]={  0.0,  0.0,  0.3, 1.0 }; // even ambiently
```

Defining the light properties and the lighting model in the initialization function:

```
glLightfv(GL_LIGHT0, GL_POSITION, light_pos0 ); // light 0
glLightfv(GL_LIGHT0, GL_AMBIENT, amb_color0 );
glLightfv(GL_LIGHT0, GL_SPECULAR, light_col0 );
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_col0 );

glLightfv(GL_LIGHT1, GL_POSITION, light_pos1 ); // light 1
glLightfv(GL_LIGHT1, GL_AMBIENT, amb_color1 );
glLightfv(GL_LIGHT1, GL_SPECULAR, light_col1 );
```

```
glLightfv(GL_LIGHT1, GL_DIFFUSE, light_col1 );

glLightfv(GL_LIGHT2, GL_POSITION, light_pos2 ); // light 2
glLightfv(GL_LIGHT2, GL_AMBIENT, amb_color2 );
glLightfv(GL_LIGHT2, GL_SPECULAR, light_col2 );
glLightfv(GL_LIGHT2, GL_DIFFUSE, light_col2 );

glLightModeliv(GL_LIGHT_MODEL_TWO_SIDE, &i ); // two-sided lighting
```

Enabling the lights in the initialization function:

```
glEnable(GL_LIGHTING);   // so lighting models are used
glEnable(GL_LIGHT0);     // we'll use LIGHT0
glEnable(GL_LIGHT1);     // ...   and LIGHT1
glEnable(GL_LIGHT2);     // ...   and LIGHT2
```

Defining the material color in the function that draws the surface: we must define the ambient and diffuse parts of the object's material specification, as shown below; note that the shininess value must be an array. Recall that higher values of shininess will create more focused and smaller specular highlights on the object. That this example doesn't specify the properties of the material's back side because the object is closed and all the back side of the material is invisible.

```
GLfloat shininess[]={ 50.0 };
glMaterialfv(GL_FRONT, GL_AMBIENT,   white );
glMaterialfv(GL_FRONT, GL_DIFFUSE,   white );
glMaterialfv(GL_FRONT, GL_SHININESS, shininess );
```

Figure 9.5 below shows the cube when it is rotated so one corner points toward the viewer. Here the ambient light contributed by all three of the lights keeps the colors somewhat muted, but clearly the red light is above, the green light is below and to the right, and the blue light is below and to the left of the viewer's eyepoint. The lights seem to be pastels because each face still gets some of the other two colors; to change this you would need to change the positions of the lights.

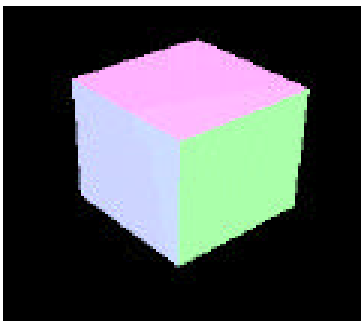


Figure 9.5: the white cube viewed with three colored lights

*A word to the wise...*

The OpenGL lighting model is essentially the same as the basic lighting model of all standard graphics APIs, but it lacks some very important things that might let you achieve some particular effects you would want if you were to try to get genuine realism in your scenes. One of the most important things lacking in the simple lighting model here is shadows; while OpenGL has techniques that can allow you to create shadows, they are tricky and require some special effort. Another important missing part is the kind of "hot" colors that seem to radiate more of a particular color than they could possibly get in the light they receive, and there is no way to fix this because of the limited gamut of the phosphors in any computer screen, as described in many textbooks.

Finally, as we discuss in the next chapter, OpenGL does not allow the kind of directional (anisotropic) reflection that you would need to model materials such as brushed aluminum, which can be created on the computer with special programming. So do not take the OpenGL lighting model as the correct way to do color; take it as a way that works pretty well and that would take much more effort to do better.

Lighting is a seductive effect because it engages our perceptual system to identify shapes of things. This can be very effective, but beware of applying lighting where your shapes or colors are purely arbitrary and represent abstract concepts. It can be dangerous to infer shapes by lighting where there is no physical reality to the things being displayed.

# Shading Models

*Prerequisites*

An understanding of the concept of color, of polygons, and of intepolation across a polygon.

*Introduction*

Shading is the process of computing the color for the components of a scene. It is usually done by calculating the effect of light on each object in a scene to create an effective lighted presentation. The shading process is thus based on the physics of light, and the most detailed kinds of shading computation can involve deep subtleties of the behavior of light, including the way light scatters from various kinds of materials with various details of surface treatments. Considerable research has been done in those areas and any genuinely realistic rendering must take a number of surface details into account.

Most graphics APIs do not have the capability to do these detailed kinds of computation. The usual beginning API such as OpenGL supports two shading models for polygons: flat shading and smooth shading. You may choose either, but smooth shading is usually more pleasing and can be somewhat more realistic. Unless there is a sound reason to use flat shading in order to represent data or other communication concepts more accurately, you will probably want to use smooth shading for your images. We will briefly discuss just a bit more sophisticated kinds of shading, even though the beginning API cannot directly support them.

*Definitions*

Flat shading of a polygon presents each polygon with a single color. This effect is computed by assuming that each polygon is strictly planar and all the points on the polygon have exactly the same kind of lighting treatment. The term flat can be taken to mean that the color is flat (does not vary) across the polygon, or that the polygon is colored as though it is flat (planar) and thus does not change color as it is lighted. This is the effect you will get if you simply set a color for the polygon and do not use a lighting model (the color is flat), or if you use lighting and materials models and then display the polygon with a single normal vector (the polygon is flat). This single normal allows you only a single lighting computation for the entire polygon, so the polygon is presented with only one color.

Smooth shading of a polygon displays the pixels in the polygon with smoothly-changing colors across the surface of the polygon. This requires that you provide information that defines a separate color for each vertex of your polygon, because the smooth color change is computed by interpolating the vertex colors across the interior of the triangle with the standard kind of interpolation we saw in the graphics pipeline discussion. The interpolation is done in screen space after the vertices' position has been set by the projection, so the purely linear calculations can easily be done in graphics cards. This per-vertex color can be provided by your model directly, but it is often produced by per-vertex lighting computations. In order to compute the color for each vertex separately you must define a separate normal vector for each vertex of the polygon so that the lighting model will produce different colors at each vertex.

Each graphic API will treat shading somewhat differently, so it is important for you to understand how your particular API handles this. The default shading behavior of OpenGL is smooth, for example, but you will not get the visual effect of smooth shading unless you specify the appropriate normals for your model, as described below. OpenGL allows you to select the shading model with the `glShadeModel` function, and the only values of its single parameter are

the symbolic parameters `GL_SMOOTH` and `GL_FLAT`. You may use the `glShadeModel` function to switch back and forth between smooth and flat shading any time you wish.

*Some examples*

We have seen many examples of polygons earlier in these notes, but we have not been careful to distinguish between whether they were presented with flat and smooth shading. Figure 10.1 shows see two different images of the same function surface, one with flat shading (left) and one with smooth shading (right), to illustrate the difference. Clearly the smooth-shaded image is much cleaner, but there are still some areas where the triangles change direction very quickly and the boundaries between the triangles still show in the smoothly-shaded image. Smooth shading is very nice—probably nicer than flat shading in many applications—but it isn't perfect.

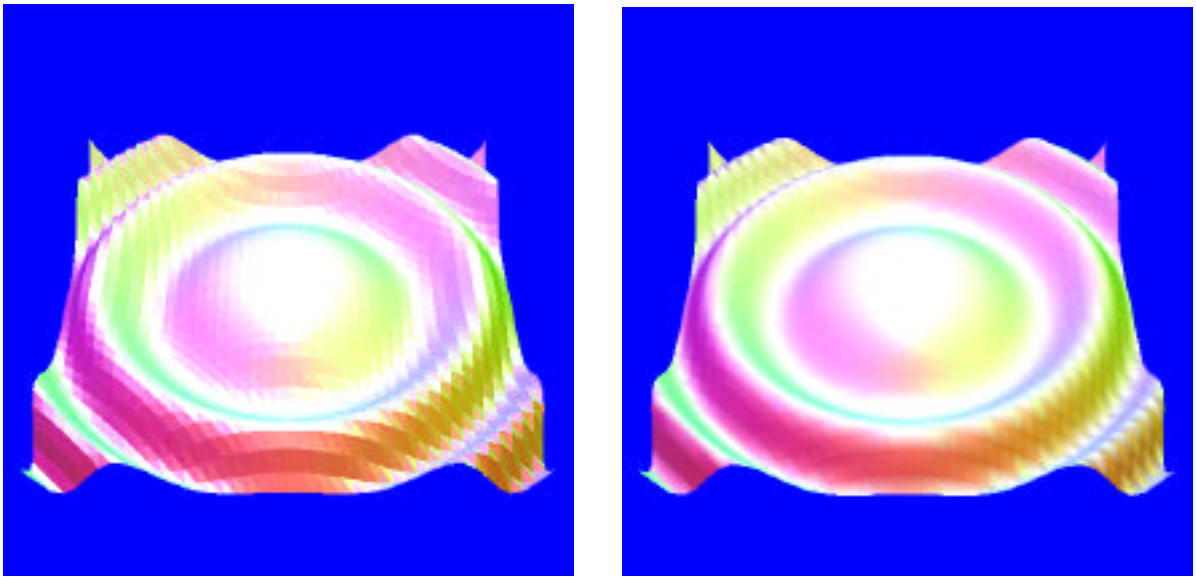


Figure 10.1: a surface with flat shading (left) and the same surface with smooth shading (right)

The computation for smooth shading uses simple polygon interpolation in screen space. Because each vertex has its own normal, the lighting model computes a different color for each vertex. The interpolation then calculates colors for each pixel in the polygon that vary smoothly across the polygon interior, providing a smooth color graduation across the polygon. This interpolation is called *Gouraud shading* and is one of the standard techniques for creating images. It is quick to compute but because it only depends on colors at the polygon vertices, it can miss lighting effects within polygons. Visually, it is susceptible to showing the color of a vertex more strongly along an edge of a polygon than a genuinely smooth shading would suggest, as you can see in the right-hand image in Figure 10.1. Other kinds of interpolation are possible that do not show some of these problems, though they are not often provided by a graphics API, and one of these is discussed below.

An interesting experiment to help you understand the properties of shaded surfaces is to consider the relationship between smooth shading and the resolution of the display grid. In principle, you should be able to use fairly fine grid with flat shading or a much coarser grid with smooth shading to achieve similar results. You should define a particular grid size and flat shading, and try to find the smaller grid that would give a similar image with smooth shading. Figure 10.2 is an example of this experiment; this surface still shows a small amount of the faceting of flat shading but avoids much of the problem with quickly-varying surface directions of a coarse smooth shading. It is probably superior in many ways to the smooth-shaded polygon of Figure 10.1. It may be either

faster or slower than the original smooth shading, depending on the efficiency of the polygon interpolation in the graphics pipeline. This is an example of a very useful experimental approach to computer graphics: if you have several different ways to approximate an effect, it can be very useful to try all of them that make sense and see which works better, both for effect and for speed, in a particular application!

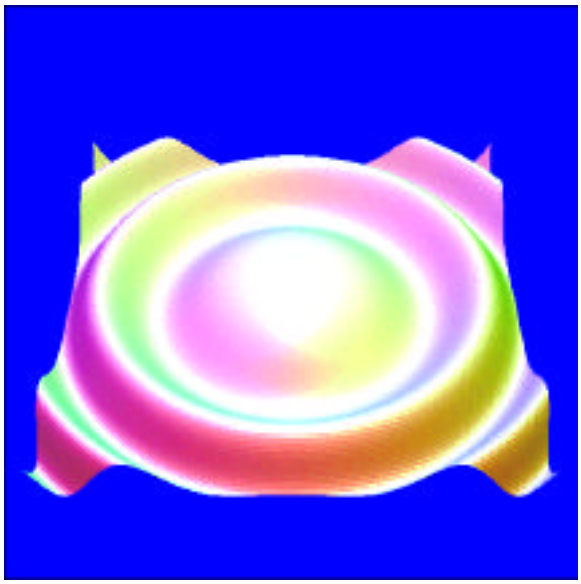


Figure 10.2: a flat-shaded image with resolution three times as great as the previous figure

*Calculating per-vertex normals*

The difference between the programming for these two parts of Figure 10.1 is that the flat-shaded model uses only one normal per polygon (calculated by computing the cross product of two edges of each triangle), while the smooth-shaded model uses a separate normal per vertex (calculated by doing some analytic work to determine the exact value of the normal). It can take a bit more work to compute the normal at each vertex instead of only once per polygon, but that is the price for smoothing.

There are a number of ways you may calculate the normal for a particular vertex of a model. You may use an interpolation technique, in which you compute a weighted average of the normals of each of the polygons that includes the vertex, or you may use an analytic computation. The choice of technique will depend on the information you have for your model.

In the interpolation technique, you can calculate the normal at a vertex by computing the average

$$N = (\Sigma a_i N_i)/(\Sigma a_i)$$

with the sum taken over all polygons $P_i$ that include this vertex, where each polygon $P_i$ has a normal $N_i$ and has angle $a_i$ at the vertex in question. Each angle $a_i$ can be calculated easily as the inverse cosine of the dot product of the two edges of the polygon $P_i$ that meet at the vertex.

In the example of Figure 10.1, an analytic approach was possible because the surface was defined by a clean, closed-form equation: `0.3*cos(x*x+y*y+t)`. In the smooth-shaded example, we were able to calculate the vertex normals by using the analytic directional derivatives at each vertex: `-0.6*x*sin(x*x+y*y+t)` and `-0.6*y*sin(x*x+y*y+t)` for `x` and `y`, respectively. These were used to calculate the tangent vectors in these directions, and those cross products were

computed to get the vertex normal. This is shown in the code sample at the end of this chapter. It can also be possible to get exact normals from other kinds of models; we saw in an early chapter in these notes that the normals to a sphere are simply the radius vectors for the sphere, so a purely geometric model may also have exactly-defined normals. In general, when models permit you to carry out analytic or geometric calculations for normals, these will be more exact and will give you better results than using an interpolation technique.

*Other shading models*

You cannot and must not assume that the smooth shading model of a simply API such as OpenGL is an accurate representation of smooth surfaces. It assumes that the surface of the polygon varies uniformly, it only includes per-vertex information in calculating colors across the polygon, and it relies on a linear behavior of the RGB color space that is not accurate, as you saw when we talked about colors. Like many of the features of any computer graphics system, it approximates a reality, but there are better ways to achieve the effect of smooth surfaces. For example, there is a shading model called *Phong shading* that requires the computation of one normal per vertex and uses the interpolated values of the normals themselves to compute the color at each pixel in the polygon, instead of simply interpolating the vertex colors. Interpolating the normals is much more complicated than interpolating colors, because the uniformly-spaced pixels in screen space do not come from uniformly-spaced points in 3D eye space or 3D model space; the perspective projection involves a division by the Z-coordinate of the point in eye space. This makes normal interpolation more complex—and much slower—than color interpolation and takes it out of the range of simple graphics APIs. However, the Phong shading model behaves like a genuinely smooth surface across the polygon, including picking up specular highlights within the polygon and behaving smoothly along the edges of the polygon. The details of how Gouraud and Phong shading operate are discussed in any graphics textbook. We encourage you to read them as an excellent example of the use of interpolation as a basis for many computer graphics processes.

The Phong shading model assumes that normals change smoothly across the polygon, but another shading model is based on controlling the normals across the polygon. Like the texture map that we describe later and that creates effects that change across a surface and are independent of the colors at each vertex, we may create a mapping that alters the normals in the polygon so the shading model can create the effect of a bumpy surface. This is called a *bump map*, and like Phong shading the normal for each individual pixel is computed as the normal from Phong shading plus the normal from the bump map. The color of each individual pixel is then computed from the lighting model. Figure 10.3 shows an example of the effect of a particular bump map. Note that the bump map itself is defined simply a 2D image where the height of each point is defined by the color; this is called a height field. Heights are sometimes presented in this way, for example in terrain modeling.

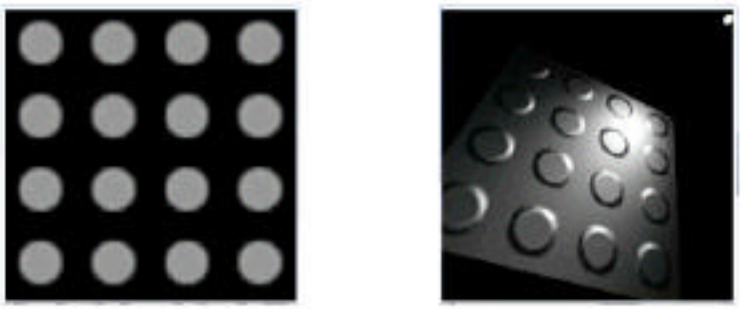


Figure 10.3: a bump map defined as a height field, left, and the bump map applied to a specular surface

The shading models that have been described so far are all based on the simple lighting model of the previous chapter, which assumes that light behavior is uniform in all directions from the surface normal (that is, the lighting is *isotropic*). However, there are some materials where the lighting parameters differ depending on the angle around the normal. Such materials include brushed metals and the surface of a CD, for example, and the shading for these materials is called *anisotropic*. Here the simple role of the angle from the normal of the diffuse reflection, and the angle from the reflected light in the specular reflection, are replaced by a more complex function called the bidirectional reflection distribution function (or BRDF) that depends typically on both the latitude $\Theta$ and longitude $\Phi$ angle of the eye and of the light from the point being lighted: $\rho(\Theta_e, \Phi_e, \Theta_l, \Phi_l)$. The BRDF may also take into account behaviors that differ for different wavelengths of light. The lighting calculations for such materials, then, may involve much more complex kinds of computation than the standard isotropic model and are beyond the scope of simple graphics APIs, but you will find this kind of shading in some professional graphics tools. Figure 10.4 shows the effect on a red sphere of applying flat, smooth, and Phong shading, and an anisotropic shading.

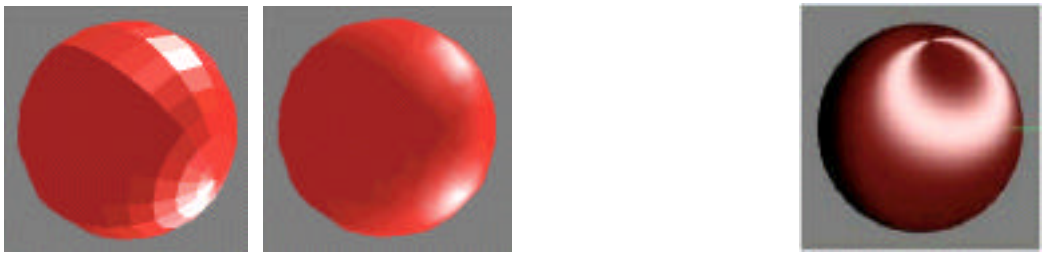


Figure 10.4: a sphere presented with flat shading (left), smooth shading (second), Phong shading (third) and an anisotropic shading (right).

Note that the smooth-shaded sphere shows some facet edges and the specular reflection is not quite smoothly distributed over the surface, while the facet edges and the specular reflection in the Phong shaded sphere are quite a bit smoother and less broken up.

*Code examples*

The two issues in using OpenGL shading are the selection of the shading model and the specification of a color at each vertex, either explicitly with the `glColor*(...)` function or by setting a normal per vertex with the `glNormal*(...)` function. Sample code to set up smooth shading by the latter approach, from the code in `flatSmooth.c` that generated the figures in this example, is shown below. To begin, note that we will have to generate one normal per vertex with smooth shading, so we define the two partial derivatives for the function in order to get tangent vectors at each vertex:

```
#define f(x,y) 0.3*cos(x*x+y*y+t)          // original function
#define fx(x,y) -0.6*x*sin(x*x+y*y+t)      // partial derivative in x
#define fy(x,y) -0.6*y*sin(x*x+y*y+t)      // partial derivative in y
```

We then use the following function call in the `init()` fuction to ensure that we automatically normalize all our normals in order to avoid having to do this computation ourselves:

```
glEnable(GL_NORMALIZE); //make normals one unit long after transform
```

In the display function, we first compute the values of `x` and `y` with the functions `XX(i)` and `YY(j)` that compute the grid points in our domain, and then we do the following (fairly long) computation for each triangle in the surface, using an inline cross product operation. We are careful to compute the triangle surface normal as (X–partial cross Y–partial), in that order, so we get the correct direction for it.

```
glBegin(GL_POLYGON);
      x = XX(i);
      y = YY(j);
      vec1[0] = 1.0;
      vec1[1] = 0.0;
      vec1[2] = fx(x,y); // partial in X-Z plane
      vec2[0] = 0.0;
      vec2[1] = 1.0;
      vec2[2] = fy(x,y); // partial in Y-Z plane
      Normal[0] = vec1[1] * vec2[2] - vec1[2] * vec2[1];
      Normal[1] = vec1[2] * vec2[0] - vec1[0] * vec2[2];
      Normal[2] = vec1[0] * vec2[1] - vec1[1] * vec2[0];
      glNormal3fv(Normal);
      glVertex3f(XX(i  ),YY(j  ),vertices[i  ][j  ]);
      ... // do similar code two more times for each vertex of the
      ... // triangle
glEnd();
```

Of course, there are many other ways to deal with normals, but you cannot simply compute a normal as a cross product of two edges because this cross product will be the same for each vertex of a triangle—any triangle is planar, after all.

# Events and Event Handling for Computer Graphics

*Introduction*

Graphics programming can focus entirely on creating one single image based on a set of data, but more and more we are seeing the value of writing programs that allow the user to interact with the world through graphical presentations, or that allow the user to control the way an image is created. These are called interactive computer graphics programs, and the ability to interact with information through an image is critically important to the success of this field.

Our emphasis in this chapter is on graphical interaction, not on user interfaces. Certainly many user interfaces use graphical presentations that give information to the user, take graphical actions, and interpret the results for program control, but we simply view these as applications of our graphics. Late in the chapter we introduce the MUI (Micro User Interface) system that allows you to add a primitive interface to your OpenGL programs, and we believe that you should try to understand the nature of the communication about images that can be supported by an external user interface, but a genuine discussion of user interfaces is much too deep for us to undertake here.

Interactive programming in computer graphics generally takes advantage of the event-handling capabilities of modern systems, so we must understand something of what events are and how to use them in order to write interactive graphics programs. Events are fairly abstract and come in several varieties, so we will need to go into some details as we develop this idea below. But modern graphics APIs handle events pretty cleanly, and you will find that once you are used to the idea, it is not particularly difficult to write event-driven programs. You should realize that some basic APIs do not include event handling, so it is sometimes necessary to use an extension to the API for this.

*Definitions*

An *event* is a transition in the control state of a computer system. Events can come from many sources and can cause any of a number of actions to take place as the system responds to the transition. In general, we will treat an event as an abstraction, a concept that we use to design interactive applications, that provides a concrete piece of data to the computer system. An *event record* is a formal record of some system activity, often an activity from a device such as a keyboard or mouse. An event record contains information that identifies the event and any data corresponding to the event. A keyboard event record contains the identity of the key that was pressed and the location of the cursor when it was pressed, for example; a mouse event record contains the mouse key that was pressed, if any, and the cursor's location on the screen when the event took place. Event records are stored in the *event queue*, which is managed by the operating system; this keeps track of the sequence in which events happen and serves as a resource to processes that deal with events. When an event occurs, its event record is inserted in the event queue and we say that the event is *posted* to the queue. The operating system manages the event queue and as each event gets to the front of the queue and a process requests an event record, the operating system passes the record to the process that should handle it. In general, events that involve a screen location get passed to whatever program owns that location, so if the event happens outside a program's window, that program will not get the event.

Let's consider a straightforward example of a user action that causes an event, and think about what is actually done to get and manage that event. .....

Programs that use events for control—and most interactive programs do this—manage that control through functions that are called *event handlers*. While these can gain access to the event queue in a number of ways, most APIs use functions called *callbacks* to handle events. Associating a callback function with an event is called *registering* the callback for the event. When the system

passes an event record to the program, the program determines what kind of event it is and if any callback function has been registered for the event, passes control to that function. In fact, most interactive programs contain initialization and action functions, callback functions, and a *main event loop*. The main event loop invokes an event handler whose function is to get an event, determine the callback needed to handle the event, and pass control to that function. When that function finishes its operation, control is returned to the event handler.

What happens in the main event loop is straightforward—the program gives up direct control of the flow of execution and places it in the hands of the user. From here on, the user will cause events that the program will respond to through the callbacks that have been created. We will see many examples of this approach in this, and later, sections of these notes.

A callback is a function that is executed when a particular event is recognized by the program. This recognition happens when the event handler takes an event off the event queue and the program has *expressed an interest* in the event. The key to being able to use a certain event in a program, then, is to express an interest in the event and to indicate what function is to be executed when the event happens—the function that has been registered for the event.

*Some examples of events*

keypress events, such as keyDown, keyUp, keyStillDown, ... Note that there may be two different kinds of keypress events: those that use the regular keyboard and those that use the so-called "special keys" such as the function keys or the cursor control keys. There may be different event handlers for these different kinds of keypresses. You should be careful when you use special keys, because different computers may have different special keys, and those that are the same may be laid out in different ways.

mouse events, such as leftButtonDown, leftButtonUp, leftButtonStillDown, … Note that different "species" of mice have different numbers of buttons, so for some kinds of mice some of these events are collapsed.

system events, such as idle and timer, that are generated by the system based on the state of the event queue or the system clock, respectively.

software events, which are posted by programs themselves in order to get a specific kind of processing to occur next.

These events are very detailed, and many of them are not used in the APIs or API extensions commonly found with graphics. However, all could be used by going deeply enough into the system on which programs are being developed.

Note that event-driven actions are fundamentally different from actions that are driven by polling—that is, by querying a device or some other part of the system on some schedule and basing system activity on the results. There are certainly systems that operate by polling various kinds of input and interaction devices, but these are outside our current approach.

*The vocabulary of interaction*

When users are working with your application, they are focusing on the content of their work, not on how you designed the application. They want to be able to communicate with the program and their data in ways that feel natural to them, and it is the task of the interface designer to create an interface that feels very natural and that doesn't interfere with their work. Interface design is the subject of a different course from computer graphics, but it is useful to have a little understanding of the vocabulary of interaction.

We have been focusing on how to program interaction with the kind of devices that are commonly found in current computers: keyboards or mice. These devices have distinctly different kinds of behaviors in users' minds. When you get away from text operations, keyboards give discrete input that can be interpreted in different ways depending on the keys that are pressed. They are basically devices that make abstract selections, with the ability select actions as well as objects. The keyboard input that navigates through simple text games is an example of action selection. The mouse buttons are also selection devices, although they are primarily used to select graphical objects on the screen, including control buttons as well as displayed objects. The keyboard and mouse buttons both are discrete devices, providing only a finite number of well-defined actions.

The mouse itself has a different kind of meaning. It provides a more continuous input, and can be used to control continuous motion on the screen. This can be the motion of a selected object as it is moved into a desired position, or it can be an input that will cause motion in an object. The motion that the mouse controls can be of various kinds as well — it can be a linear motion, such as moving the eye point across a scene, or it can be a rotational motion, such as moving an object by changing the angles defining the object in spherical coordinates.

When you plan the interaction for your application, then, you should decide whether a user will see the interaction as a discrete selection or as a continuous control, and then you should implement the interaction with the keyboard or mouse, as determined by the user's expected vocabulary.

*A word to the wise...*

This section discusses the mechanics of interaction through event handling, but it does not cover the critical questions of how a user would naturally control an interactive application. There are many deep and subtle issues involved in designing the user interface for such an application, and this module does not begin to cover them. The extensive literature in user interfaces will help you get a start in this area, but a professional application needs a professional interface — one designed, tested, and evolved by persons who focus in this area. When thinking of a real application, heed the old cliché: Kids, don't try this at home!

The examples below do their best to present user controls that are not impossibly clumsy, but they are designed much more to focus on the event and callback than on a clever or smooth way for a user to work. When you write your own interactive projects, think carefully about how a user might perceive the task, not just about an approach that might be easiest for you to program.

*Events in OpenGL*

The OpenGL API generally uses the Graphics Library Utility Toolkit GLUT (or a similar extension) for event and window handling. GLUT defines a number of kinds of events and gives the programmer a means of associating a callback function with each event that the program will use. In OpenGL with the GLUT extension, this main event loop is quite explicit as a call to the function `glutMainLoop()` as the last action in the main program.

*Callback registering*

Below we will list some kinds of events and will then indicate the function that is used to register the callback for each event. Following that, we will give some code examples that register and use these events for some programming effects. This now includes only examples from OpenGL, but it should be extensible to other APIs fairly easily.

| Event | Callback Registration Function |
|---|---|

idle — `glutIdleFunc(functionname)`
requires a function with template `void functionname(void)` as a parameter. This function is the event handler that determines what is to be done at each idle cycle. Often this function will end with a call to `glutPostRedisplay()` as described below. This function is used to define what action the program is to take when there has been no other event to be handled, and is often the function that drives real-time animations.

display — `glutDisplayFunc(functionname)`
requires a function with template `void functionname(void)` as a parameter. This function is the event handler that generates a new display whenever the display event is received. Note that the display function is invoked by the event handler whenever a display event is reached; this event is posted by the `glutPostRedisplay()` function and whenever a window is moved or reshaped.

reshape — `glutReshapeFunc(functionname)`
requires a function with template `void functionname(int, int)` as a parameter. This function manages any changes needed in the view setup to accomodate the reshaped window, which may include a fresh definition of the projection. The parameters of the `reshape` function are the width and height of the window after it has been changed.

keyboard — `glutKeyboardFunc(keybd)`
requires a function with template

```
void functionname(unsigned char, int, int)
```

as a parameter. This parameter function is the event handler that receives the character and the location of the cursor `(int x, int y)` when a key is pressed. As is the case for all callbacks that involve a screen location, the location on the screen will be converted to coordinates relative to the window. Again, this function will often end with a call to `glutPostRedisplay()` to re-display the scene with the changes caused by the particular keyboard event.

special — `glutSpecialFunc(special)`
requires a function with template

```
void functionname(int key, int x, int y)
```

as a parameter. This event is generated when one of the "special keys" is pressed; these keys are the function keys, directional keys, and a few others. The first parameter is the key that was pressed; the second and third are the integer window coordinates of the cursor when the keypress occurred as described above. The usual approach is to use a special symbolic name for the key, and these are described in the discussion below. The only difference between the special and keyboard callbacks is that the events come from different kinds of keys.

menu — `glutCreateMenu(functionname)`
requires a function with template `void functionname(int)` as a parameter. This creates a menu that is brought up by a mouse button down event, specified by

```
glutAttachMenu(event),
```

and the function

```
glutAddMenuEntry(string, int)
```

identifies each of the choices in the menu and defines the value to be returned by each one. That is, when the user selects the menu item labeled with the string, the value is passed as the parameter to the menu callback function. The menu choices are identified before the menu itself is attached, as illustrated in the lines below:

```
glutAddMenuEntry("text", VALUE);
...
glutAttachMenu(GLUT_RIGHT_BUTTON)
```

Note that the Macintosh uses a slightly different menu attachment with the same parameters,

`glutAttachMenuName(event, string)`,

that attaches the menu to a name on the system menu bar. The Macintosh menu is activated by selecting the menu name from the menu bar, while the windows for Unix and Windows are popup windows that appear where the mouse is clicked and that do not have names attached.

Along with menus one can have sub-menus — items in a menu that cause a cascaded sub-menu to be displayed when they are selected. Sub-menus are created by the use of the function

`glutAddSubMenu(string, int)`

where the string is the text displayed in the original menu and the int is the identifier of the menu to cascade from that menu item. For more details, see the GLUT manuals.

mouse

`glutMouseFunc(functionname)`
requires a function with a template such as

```
void functionname(int button, int state,
int mouseX, int mouseY)
```

as a parameter, where `button` indicates which button was pressed (an integer typically made up of one bit per button, so that a three-button mouse can indicate any value from one to seven), the `state` of the mouse (symbolic values such as `GLUT_DOWN` to indicate what is happening with the mouse) — and both raising and releasing buttons causes events — and integer values `xPos` and `yPos` for the window-relative location of the cursor in the window when the event occurred.

The mouse event does not use this function if it includes a key that has been defined to trigger a menu.

mouse active motion

`glutMotionFunc(functionname)`
requires a function with template `void functionname(int, int)` as a parameter. The two integer parameters are the window-relative coordinates of the cursor in the window when the event occurred. This event occurs when the mouse is moved with one or more buttons pressed.

mouse passive motion

`glutPassiveMotionFunc(functionname)`
requires a function with template `void functionname(int, int)` as a parameter. The two integer parameters are the window-relative coordinates of the cursor in the window when the event occurred. This event occurs when the mouse if moved with no buttons pressed.

timer

`glutTimerFunc(msec, timer, value)`

requires an integer parameter, here called `msec`, that is to be the number of milliseconds that pass before the callback is triggered; a function, here called `timer`, with a template such as `void timer(int)` that takes an integer parameter; and an integer parameter, here called `value`, that is to be passed to the `timer` function when it is called.

Note that in any of these cases, the function NULL is an acceptable option. Thus you can create a template for your code that includes registrations for all the events your system can support, and simply register the NULL function for any event that you want to ignore.

Besides the kind of device events we generally think of, there are also software events such as the display event, created by a call to `glutPostRedisplay()`. There are also device events for devices that are probably not found around most undergraduate laboratories: the spaceball, a six-degree-of-freedom deviceused in high-end applications, and the graphics tablet, a device familiar to the computer-aided design world and still valuable in many applications. If you want to know more about handling these devices, you should check the GLUT manual.

*Some details*

For most of these callbacks, the meaning of the parameters of the event callback is pretty clear. Most are either standard characters or integers such as window dimensions or cursor locations. However, for the special event, the callback must handle the special characters by symbolic names. Many of the names are straightforward, but some are not; the full table is:

| | |
|---|---|
| Function keys F1 through F12: | GLUT_KEY_F1 through GLUT_KEY_F12 |
| Directional keys: | GLUT_KEY_LEFT, GLUT_KEY_UP, GLUT_KEY_RIGHT, GLUT_KEY_DOWN |
| Other special keys: | GLUT_KEY_PAGE_UP (Page up) |
| | GLUT_KEY_PAGE_DOWN (Page down) |
| | GLUT_KEY_HOME (Home) |
| | GLUT_KEY_END (End) |
| | GLUT_KEY_INSERT (Insert ) |

So to use the special keys, use these symbolic names to process the keypress that was returned to the callback function.

*Code examples*

This section presents four examples. This first is a simple animation that uses an idle event callback and moves a cube around a circle, in and out of the circle's radius, and up and down. The user has no control over this motion. When you compile and run this piece of code, see if you can imagine the volume in 3-space inside which the cube moves.

The second example uses keyboard callbacks to move a cube up/down, left/right, and front/back by using a simple keypad on the keyboard. This uses keys within the standard keyboard instead of using special keys such as a numeric keypad or the cursor control keys. A numeric keypad is not used because some keyboards do not have them; the cursor control keys are not used because we need six directions, not just four.

The third example uses a mouse callback to pop up a menu and make a menu selection, in order to set the color of a cube. This is a somewhat trivial action, but it introduces the use of pop-up menus, which are a very standard and useful tool.

Finally, the fourth example uses a mouse callback with object selection to identify one of two cubes that are being displayed and to change the color of that cube. Again, this is not a difficult action, but it calls upon the entire selection buffer process that is the subject of another later module in this set. For now, we suggest that you focus on the event and callback concepts and postpone a full understanding of this example until you have read the material on selection.

Idle event callback

In this example, we assume we have a function named `cube()` that will draw a simple cube at the origin `(0,0,0)`. We want to move the cube around by changing its position with time, so we will let the idle event handler set the position of the cube and the display function draw the cube using the positions determined by the idle event handler. Much of the code for a complete program has been left out, but this illustrates the relation between the display function, the event handler, and the callback registration.

```
GLfloat cubex = 0.0;
GLfloat cubey = 0.0;
GLfloat cubez = 0.0;
GLfloat time  = 0.0;

void display( void )
{
    glPushMatrix();
    glTranslatef( cubex, cubey, cubez );
    cube();
    glPopMatrix();
}

void animate(void)
{
    #define deltaTime 0.05

// Position for the cube is set by modeling time-based behavior.
// Try multiplying the time by different constants to see how that
// behavior changes.

    time += deltaTime; if (time > 2.0*M_PI) time -= 2*0*M_PI;
    cubex = sin(time);
    cubey = cos(time);
    cubez = cos(time);
    glutPostRedisplay();
}

void main(int argc, char** argv)
{

/* Standard GLUT initialization precedes the functions below*/
    ...
    glutDisplayFunc(display);
    glutIdleFunc(animate);

    myinit();
    glutMainLoop();
}
```

Keyboard callback

Again we start with the familiar `cube()` funcntion. This time we want to let the user move the cube up/down, left/right, or backward/forward by means of simple keypresses. We will use two virtual keypads:

```
Q  W            I  O
 A  S          J  K
  Z  X        N  M
```

with the top row controlling up/down, the middle row controlling left/right, and the bottom row controlling backward/forward. So, for example, if the user presses either `Q` or `I`, the cube will move up; pressing `W` or `O` will move it down. The other rows will work similarly.

Again, much of the code has been omitted, but the display function works just as it did in the example above: the event handler sets global positioning variables and the display function performs a translation as chosen by the user. Note that in this example, these translations operate in the direction of faces of the cube, not in the directions relative to the window.

```
GLfloat cubex = 0.0;
GLfloat cubey = 0.0;
GLfloat cubez = 0.0;
GLfloat time  = 0.0;

void display( void )
{
    glPushMatrix();
    glTranslatef( cubex, cubey, cubez );
    cube();
    glPopMatrix();
}

void keyboard(unsigned char key, int x, int y)
{
    ch = ' ';
    switch (key)
    {
        case 'q' : case 'Q' :
        case 'i' : case 'I' :
           ch = key; cubey -= 0.1; break;
        case 'w' : case 'W' :
        case 'o' : case 'O' :
           ch = key; cubey += 0.1; break;
        case 'a' : case 'A' :
        case 'j' : case 'J' :
           ch = key; cubex -= 0.1; break;
        case 's' : case 'S' :
        case 'k' : case 'K' :
           ch = key; cubex += 0.1; break;
        case 'z' : case 'Z' :
        case 'n' : case 'N' :
           ch = key; cubez -= 0.1; break;
        case 'x' : case 'X' :
        case 'm' : case 'M' :
           ch = key; cubez += 0.1; break;
    }
    glutPostRedisplay();
}
```

```
void main(int argc, char** argv)
{
/* Standard GLUT initialization */
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);

    myinit();
    glutMainLoop();
}
```

The similar function, `glutSpecialFunc(...)`, can be used in a very similar way to read input from the special keys (function keys, cursor control keys, ...) on the keyboard.

Menu callback

Again we start with the familiar `cube()` function, but this time we have no motion of the cube. Instead we define a menu that allows us to choose the color of the cube, and after we make our choice the new color is applied.

```
#define RED     1
#define GREEN   2
#define BLUE    3
#define WHITE   4
#define YELLOW  5

void cube(void)
{
    ...

    GLfloat color[4];

//  set the color based on the menu choice

    switch (colorName) {
        case RED:
            color[0] = 1.0; color[1] = 0.0;
            color[2] = 0.0; color[3] = 1.0; break;
        case GREEN:
            color[0] = 0.0; color[1] = 1.0;
            color[2] = 0.0; color[3] = 1.0; break;
        case BLUE:
            color[0] = 0.0; color[1] = 0.0;
            color[2] = 1.0; color[3] = 1.0; break;
        case WHITE:
            color[0] = 1.0; color[1] = 1.0;
            color[2] = 1.0; color[3] = 1.0; break;
        case YELLOW:
            color[0] = 1.0; color[1] = 1.0;
            color[2] = 0.0; color[3] = 1.0; break;
    }

//  draw the cube

    ...
 }

void display( void )
{
```

```
    cube();
}

void options_menu(int input)
{
    colorName = input;
    glutPostRedisplay();
}

void main(int argc, char** argv)
{
    ...

        glutCreateMenu(options_menu);        // create options menu
        glutAddMenuEntry("Red", RED);        // 1 add menu entries
        glutAddMenuEntry("Green", GREEN);    // 2
        glutAddMenuEntry("Blue", BLUE);      // 3
        glutAddMenuEntry("White", WHITE);    // 4
        glutAddMenuEntry("Yellow", YELLOW);  // 5
        glutAttachMenu(GLUT_RIGHT_BUTTON, "Colors");

        myinit();
        glutMainLoop();
}
```

Mouse callback for object selection

This example is more complex because it illustrates the use of a mouse event in object selection. This subject is covered in more detail in the later chapter on object selection, and the full code example for this example will also be included there. We will create two cubes with the familiar `cube()` function, and we will select one with the mouse. When we select one of the cubes, the cubes will exchange colors.

In this example, we start with a full `Mouse(...)` callback function, the `render(...)` function that registers the two cubes in the object name list, and the `DoSelect(...)` function that manages drawing the scene in `GL_SELECT` mode and identifying the object(s) selected by the position of the mouse when the event happened. Finally, we include the statement in the `main()` function that registers the mouse callback function.

```
glutMouseFunc(Mouse);

...

void Mouse(int button, int state, int mouseX, int mouseY)
{
   if (state == GLUT_DOWN) { /* find which object was selected */
         hit = DoSelect((GLint) mouseX, (GLint) mouseY);
   }
   glutPostRedisplay();
}

...

void render( GLenum mode )
{
// Always draw the two cubes, even if we are in GL_SELECT mode,
// because an object is selectable iff it is identified in the name
```

```
// list and is drawn in GL_SELECT mode
   if (mode == GL_SELECT)
         glLoadName(0);
   glPushMatrix();
   glTranslatef(  1.0,  1.0, -2.0 );
   cube(cubeColor2);
   glPopMatrix();
   if (mode == GL_SELECT)
         glLoadName(1);
   glPushMatrix();
    glTranslatef( -1.0, -2.0,  1.0 );
   cube(cubeColor1);
   glPopMatrix();
   glFlush();
   glutSwapBuffers();
}

...

GLint DoSelect(GLint x, GLint y)
{
   GLint hits, temp;

   glSelectBuffer(MAXHITS, selectBuf);
   glRenderMode(GL_SELECT);
   glInitNames();
   glPushName(0);

// set up the viewing model
   glPushMatrix();
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
// set up the matrix that identifies the picked object(s), based on
// the  x  and  y  values of the selection and the information on the
// viewport
   gluPickMatrix(x, windH - y, 4, 4, vp);
   glClearColor(0.0, 0.0, 1.0, 0.0);
   glClear(GL_COLOR_BUFFER_BIT);
   gluPerspective(60.0,1.0,1.0,30.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
//               eye point     center of view      up
   gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

   render(GL_SELECT);  // draw the scene for selection

   glPopMatrix();
// find the number of hits recorded and reset mode of render
   hits = glRenderMode(GL_RENDER);
// reset viewing model into GL_MODELVIEW mode
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0,1.0,1.0,30.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
//               eye point     center of view      up
   gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
// return the label of the object selected, if any
   if (hits <= 0) {
```

```
            return -1;
        }
    // carry out the color changes that will be the effect of a selection
        temp = cubeColor1; cubeColor1 = cubeColor2; cubeColor2 = temp;
        return selectBuf[3];
    }

    void main(int argc, char** argv)
    {
        ...
        glutMouseFunc(Mouse);

        myinit();
        glutMainLoop();
    }
```

Mouse callback for mouse motion

This example shows the callback for the motion event. This event can be used for anything that uses the position of a moving mouse with button pressed as control. It is fairly common to see a graphics program that lets the user hold down the mouse and drag the cursor around in the window, and the program responds by moving or rotating the scene around the window. The program this code fragment is from uses the integer coordinates to control spin, but they could be used for many purposes and the application code itself is omitted.

```
    void motion(int xPos, int yPos)
    {
        spinX = (GLfloat)xPos;
        spinY = (GLfloat)yPos;
    }

    int main(int argc, char** argv)
    {
        ...
        glutMotionFunc(motion);

        myinit();
        glutMainLoop();
    }
```

# The MUI (Micro User Interface) facility

*Prerequisites*

An understanding of event-driven programming and some experience using the simple events and callbacks from the GLUT toolkit in OpenGL, and some review of interface capabilities in standard applications.

*Introduction*

There are many kinds of interface tools that we are used to seeing in applications but that we cannot readily code in OpenGL, even with the GLUT toolkit. Some of these are provided by the MUI facility that is a universal extension of GLUT for OpenGL. With MUI you can use sliders, buttons, text boxes, and other tools that may be more natural for many applications than the standard GLUT capabilities. Of course, you may choose to write your own tools as well, but you may choose to use your time on the problem at hand instead of writing an interface, so the MUI tools may be just what you want.

MUI has a good bit of the look and feel of the X-Motif interface, so do not expect applications you write with this to look like they are from either the Windows or Macintosh world. Instead, focus on the functionality you need your application to have, and find a way to get this functionality from the MUI tools. The visible representation of these tools are called *widgets*, just as they are in the X Window System, so you will see this term throughout thest notes.

This chapter is built on Steve Baker's "A Brief MUI User Guide," and it shares similar properties: it is based on a small number of examples and some modest experimental work. It is intended as a guide, not as a manual, though it is hoped that it will contribute to the literature on this useful tool.

*Definitions*

The capabilities of MUI include pulldown menus, buttons, radio buttons, text labels, text boxes, and vertical and horizontal sliders. We will outline how each of these work below and will include some general code to show how each is invoked.

The main thing you must realize in working with MUI is that MUI takes over the event handling from GLUT, so you cannot mix MUI and GLUT event-handling capabilities in the same window. This means that you will have to create separate windows for your MUI controls and for your display, which can feel somewhat clumsy. This is a tradeoff you must make when you design your application — are you willing to create a different kind of interface than you might expect in a traditional application in order to use the extra MUI functionality? Only you can say. But before you can make that choice, you need to know what each of the MUI facilities can do.

Menu bars: A MUI menu bar is essentially a GLUT menu that is bound to a MUI object and then that object is added to a UIlist. Assuming you have defined an array of GLUT menus named `myMenus[...]`, you can use the function to create a new pulldown menu and then use the function to add new menus to the pulldown menu list:

```
muiObject *muiNewPulldown();
muiAddPulldownEntry(muiObject *obj,char *title,int glut_menu,
                    int is_help);
```

An example of the latter function would be

```
myMenubar = muiNewPulldown();
muiAddPulldownEntry(myMenubar, "File", myMenu, 0);
```

where the is_help value would be 1 for the last menu in the menu bar, because traditionally the help menu is the rightmost menu in a menu bar.

According to Baker [Bak], there is apparently a problem with the pulldown menus when the GLUT window is moved or resized. The reader is cautioned to be careful in handling windows when the MUI facility is being used.

Buttons: a button is presented as a rectangular region which, when pressed, sets a value or carries out a particular operation. Whenever the cursor is in the region, the button is highlighted to show that it is then selectable. A button is created by the function

```
muiNewButton(int xmin, int xmax, int ymin, int ymax)
```

that has a `muiObject *` return value. The parameters define the rectangle for the button and are defined in window (pixel) coordinates, with `(0,0)` at the lower left corner of the window. In general, any layout in the MUI window will be based on such coordinates.

Radio buttons: radio buttons are similar to standard buttons, but they come in only two fixed sizes (either a standard size or a mini size). The buttons can be designed so that more than one can be pressed (to allow a user to select any subset of a set of options) or they can be linked so that when one is pressed, all the others are un-pressed (to allow a user to select only one of a set of options). Like regular buttons, they are highlighted when the cursor is scrolled over them.

You create radio buttons with the functions

```
muiObject *muiNewRadioButton(int xmin, int ymin)
muiObject *muiNewTinyRadioButton(int xmin, int ymin)
```

where the `xmin` and `ymin` are the window coordinates of the lower left corner of the button. The buttons are linked with the function

```
void muiLinkButtons(button1, button2)
```

where `button1` and `button2` are the names of the button objects; to link more buttons, call the function with overlapping pairs of button names as shown in the example below. In order to clear all the buttons in a group, call the function below with any of the buttons as a parameter:

```
void muiClearRadio(muiObject *button)
```

Text boxes: a text box is a facility to allow a user to enter text to the program. The text can then be used in any way the application wishes. The text box has some limitations; for example, you cannot enter a string longer than the text box's length. However, it also gives your user the ability to enter text and use backspace or delete to correct errors. A text box is created with the function

```
muiObject *muiNewTextbox(xmin, xmax, ymin)
```

whose parameters are window coordinates, and there are functions to set the string:

```
muiSetTBString(obj, string)
```

to clear the string:

```
muiClearTBString(obj)
```

and to get the value of the string:

```
char *muiGetTBString (muiObject *obj).
```

Horizontal sliders: in general, sliders are widgets that return a single value when they are used. The value is between zero and one, and you must manipulate that value into whatever range your application needs. A slider is created by the function

```
muiNewHSlider(int xmin,int ymin,int xmax,int scenter,int shalf)
```

where `xmin` and `ymin` are the screen coordinates of the lower left corner of the slider, `xmax` is the screen coordinate of the right-hand side of the slider, `scenter` is the screen coordinate of the center of the slider's middle bar, and `shalf` is the half-size of the middle bar itself. In the event callback for the slider, the function `muiGetHSVal(muiObject *obj)` is used to return the value (as a float) from the slider to be used in the application. In order to reverse the process — to make the slider represent a particular value, use the function

```
muiSetHSValue(muiObject *obj, float value)
```

Vertical sliders: vertical sliders have the same functionality as horizontal sliders, but they are aligned vertically in the control window instead of horizontally. They are managed by functions that are almost identical to those of horizontal sliders:

```
muiNewVSlider(int xmin,int ymin,int ymax,int scenter,int shalf)
muiGetVSValue(muiObject *obj, float value)
muiSetVSValue(muiObject *obj, float value)
```

Text labels: a text label is a piece of text on the MUI control window. This allows the program to communicate with the user, and can be either a fixed or variable string. To set a fixed string, use

```
muiNewLabel(int xmin, int ymin, string)
```

with xmin and ymin setting the lower left corner of the space where the string will be displayed. To define a variable string, you give the string a `muiObject` name via the variation

```
muiObject *muiNewLabel(int xmin, int ymin, string)
```

to attach a name to the label, and use the `muiChangeLabel(muiObject *, string)` function to change the value of the string in the label.

*Using the MUI functionality*

Before you can use any of MUI's capabilities, you must initialize the MUI system with the function `muiInit()`, probably called from the `main()` function as described in the sample code below.

MUI widgets are managed in UI lists. You create a UI list with the `muiNewUIList(int)` function, giving it an integer name with the parameter, and add widgets to it as you wish with the function `muiAddToUIList(listid, object)`. You may create multiple lists and can choose which list will be active, allowing you to make your interface context sensitive. However, UI lists are essentially static, not dynamic, because you cannot remove items from a list or delete a list.

All MUI capabilities can be made visible or invisible, active or inactive, or enabled or disabled. This adds some flexibility to your program by letting you customize the interface based on a particular context in the program. The functions for this are:

```
void muiSetVisible(muiObject *obj, int state);
void muiSetActive(muiObject *obj, int state);
void muiSetEnable(muiObject *obj, int state);
int muiGetVisible(muiObject *obj);
int muiGetActive(muiObject *obj);
int muiGetEnable(muiObject *obj);
```

Figure 11.1 shows most of the MUI capabilities: labels, horizontal and vertical sliders, regular and radio buttons (one radio button is selected and the button is highlighted by the cursor as shown), and a text box. Some text has been written into the text box. This gives you an idea of what the standard MUI widgets look like, but because the MUI source is available, you have the opportunity to customize the widgets if you want, though this is beyond the scope of this discussion. Layout is facilitated by the ability to get the size of a MUI object with the function

```
void muiGetObjectSize(muiObject *obj, int *xmin, int *ymin,
                      int *xmax, int *ymax);
```


Figure 11.1: the set of MUI facilities on a single window

MUI object callbacks are optional (you would probably not want to register a callback for a fixed text string, for example, but you would with an active item such as a button). In order to register a callback, you must name the object when it is created and must link that object to its callback function with

```
void muiSetCallback(muiObject *obj, callbackFn)
```

where a callback function has the structure

```
void callbackFn(muiObject *obj, enum muiReturnValue)
```

Note that this callback function need not be unique to the object; in the example below we define a single callback function that is registered for three different sliders and another to handle three different radio buttons, because the action we need from each is the same; when we need to know which object handled the event, this information is available to us as the first parameter of the callback.

If you want to work with the callback return value, the declaration of the `muiReturnValue` is:

```
enum muiReturnValue {
     MUI_NO_ACTION,
     MUI_SLIDER_MOVE,
     MUI_SLIDER_RETURN,
     MUI_SLIDER_SCROLLDOWN,
     MUI_SLIDER_SCROLLUP,
     MUI_SLIDER_THUMB,
     MUI_BUTTON_PRESS,
     MUI_TEXTBOX_RETURN,
     MUI_TEXTLIST_RETURN,
     MUI_TEXTLIST_RETURN_CONFIRM
};
```

so you can look at these values explicitly. For the example below, the button press is assumed because it is the only return value associated with a button, and the slider is queried for its value instead of handling the actual MUI action.

*Some examples*

Let's consider a simple application and see how we can create the controls for it using the MUI facility. The application is color choice, commonly handled with three sliders (for R/G/B) or four sliders (for R/G/B/A) depending on the need of the user. This application typically provides a way

to display the color that is chosen in a region large enough to reduce the interference of nearby colors in perceiving the chosen color. The application we have in mind is a variant on this that not only shows the color but also shows the three fixed-component planes in the RGB cube and draws a sphere of the selected color (with lighting) in the cube.

The design of this application is built on an example in the Science Examples chapter that shows three cross-sections of a real function of three variables. In order to determine the position of the cross sections, we use a control built on MUI sliders. We also add radio buttons to allow the user to define the size of the sphere at the intersection of the cross-section slices.

Selected code for this application includes declarations of muiObjects, callback functions for sliders and buttons, and the code in the main program that defines the MUI objects for the program, links them to their callback functions, and adds them to the single MUI list we identify. The main issue is that MUI callbacks, like the GLUT callbacks we met earlier, have few parameters and do most of their work by modifying global variables that are used in the other modeling and rendering operations.

```
// selected declarations of muiObjects and window identifiers
muiObject *Rslider, *Gslider, *Bslider;
muiObject *Rlabel, *Glabel, *Blabel;
muiObject *noSphereB, *smallSphereB, *largeSphereB;
int muiWin, glWin;

// callbacks for buttons and sliders
void readButton(muiObject *obj, enum muiReturnValue rv) {
   if ( obj == noSphereB )
      sphereControl = 0;
   if ( obj == smallSphereB )
      sphereControl = 1;
   if (obj == largeSphereB )
      sphereControl = 2;
  glutSetWindow( glWin );
  glutPostRedisplay();
}

void readSliders(muiObject *obj, enum muiReturnValue rv) {
   char rs[32], gs[32], bs[32];
   glutPostRedisplay();

   rr = muiGetHSVal(Rslider);
   gg = muiGetHSVal(Gslider);
   bb = muiGetHSVal(Bslider);

   sprintf(rs,"%6.2f",rr);
   muiChangeLabel(Rlabel, rs);
   sprintf(gs,"%6.2f",gg);
   muiChangeLabel(Glabel, gs);
   sprintf(bs,"%6.2f",bb);
   muiChangeLabel(Blabel, bs);

   DX = -4.0 + rr*8.0;
   DY = -4.0 + gg*8.0;
   DZ = -4.0 + bb*8.0;

   glutSetWindow(glWin);
   glutPostRedisplay();
}
```

```
void main(int argc, char** argv){
   char rs[32], gs[32], bs[32];
// Create MUI control window and its callbacks
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGBA);
   glutInitWindowSize(270,350);
   glutInitWindowPosition(600,70);
   muiWin = glutCreateWindow("Control Panel");
   glutSetWindow(muiWin);
   muiInit();
   muiNewUIList(1);
   muiSetActiveUIList(1);

// Define color control sliders
   muiNewLabel(90, 330, "Color controls");

   muiNewLabel(5, 310, "Red");
   sprintf(rs,"%6.2f",rr);
   Rlabel = muiNewLabel(35, 310, rs);
   Rslider = muiNewHSlider(5, 280, 265, 130, 10);
   muiSetCallback(Rslider, readSliders);

   muiNewLabel(5, 255, "Green");
   sprintf(gs,"%6.2f",gg);
   Glabel = muiNewLabel(35, 255, gs);
   Gslider = muiNewHSlider(5, 225, 265, 130, 10);
   muiSetCallback(Gslider, readSliders);

   muiNewLabel(5, 205, "Blue");
   sprintf(bs,"%6.2f",bb);
   Blabel = muiNewLabel(35, 205, bs);
   Bslider = muiNewHSlider(5, 175, 265, 130, 10);
   muiSetCallback(Bslider, readSliders);

// define radio buttons
   muiNewLabel(100, 150, "Sphere size");
   noSphereB    = muiNewRadioButton(10, 110);
   smallSphereB = muiNewRadioButton(100, 110);
   largeSphereB = muiNewRadioButton(190, 110);
   muiLinkButtons(noSphereB, smallSphereB);
   muiLinkButtons(smallSphereB, largeSphereB);
   muiLoadButton(noSphereB,    "None");
   muiLoadButton(smallSphereB, "Small");
   muiLoadButton(largeSphereB, "Large");
   muiSetCallback(noSphereB,    readButton);
   muiSetCallback(smallSphereB, readButton);
   muiSetCallback(largeSphereB, readButton);
   muiClearRadio(noSphereB);

// add sliders and radio buttons to UI list 1
   muiAddToUIList(1, Rslider);
   muiAddToUIList(1, Gslider);
   muiAddToUIList(1, Bslider);
   muiAddToUIList(1, noSphereB);
   muiAddToUIList(1, smallSphereB);
   muiAddToUIList(1, largeSphereB);

// Create display window and its callbacks
   ...
}
```

The presentation and communication for this application are shown in Figure 11.2 below. As the sliders set the R, G, and B values for the color, the numerical values are shown above the sliders and the three planes of constant R, G, and B are shown in the RGB cube. At the intersection of the three planes is drawn a sphere of the selected color in the size indicated by the radio buttons. The RGB cube itself can be rotated by the usual keyboard controls so the user can compare the selected color with nearby colors in those planes, but you have the usual issues of active windows: you must make the display window active to rotate the cube, but you must make the control window active to use the controls.

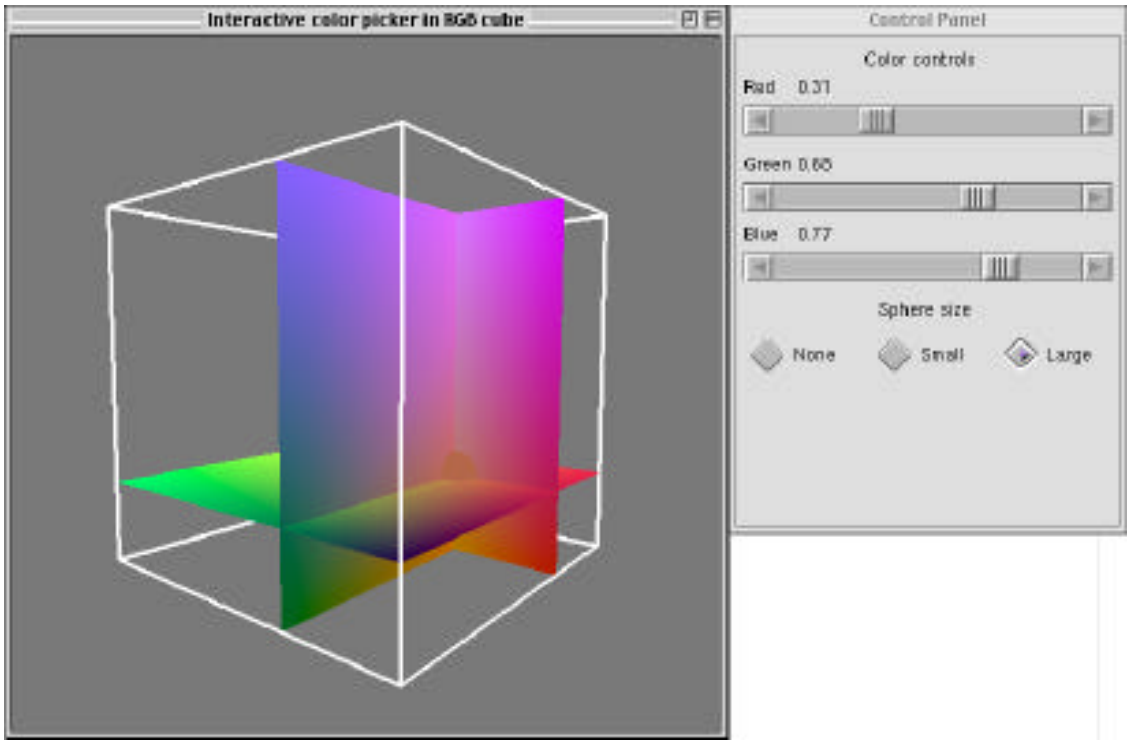



Figure 11.2: the color selector in context, with both the display and control windows shown

*A word to the wise...*

The MUI control window has behaviors that are outside the programmer's control, so you must be aware of some of these in order to avoid some surprises. The primary behavior to watch for is that many of the MUI elements include a stream of events (and their associated redisplays) whenever the cursor is within the element's region of the window. If your application was not careful to insulate itself against changes caused by redisplays, you may suddenly find the application window showing changes when you are not aware of requesting them or of creating any events at all. So if you use MUI, you should be particularly conscious of the structure of your application on redisplay and ensure that (for example) you clear any global variable that causes changes in your display before you leave the display function.

# Science Examples II

*Prerequisites:*

A knowledge of computer graphics through color, lighting, shading, and event handling, together with some knowledge of the science areas that will be discussed in this section.

This chapter contains a varied collection of science-based examples that students can understand with the more advanced knowledge of graphics that they have at this point, including lighting, shading, and event handling. These examples are not as sophisticated as one would see in professional scientific visualization, but are a sound beginning towards that level of sophistication. They cover additional topics in the sciences beyond those in the previous chapter on science examples, and again are grouped so that similar kinds of graphics can be brought to bear on the problems.

As before, each example will describe a science problem and the graphic image or images that address it, and will include the following kinds of information:

- A short description of the science in the problem
- A short description of the modeling of the problem in terms of the sciences
- A short description of the computational modeling of the problem, including any assumptions that we make that could simplify the problem and the tradeoffs implicit in those assumptions
- A description of the computer graphics modeling that implements the computational modeling
- A description of the visual communication in the display, including any dynamic components that enhance the communication
- An image from an implementation of the model in OpenGL
- A short set of code fragments that make up that implementation

There are enough topics that this is a fairly long chapter, but it is very important for the student to look at this in depth, because an understanding of the science and of the scientific modeling is at the heart of any good computational representation of the problem and thus at the heart of the computer graphics that will be presented.

*Examples:*

<u>Displaying scientific objects</u>

1. Simple molecule display

These projects ask a student to read the description of a molecule in a standard format (see the Appendices for two standard molecule file formats) and display the resulting molecule in a way that supports simple manipulations such as rotation, zooming, transparency, and clipping. These are straightforward projects that require the student to extract information from a file and use that information to determine the geometry of the molecule to display it. They cover most of the topics one would want to include in an introductory computer graphics course, and the sequence of the topics is fairly standard. The instructor should be aware, however, that the project sequence for students from different disciplines may not draw on graphics topics in exactly the same way, so there may need to be some adjustment of the project sequencing that makes allowance for these differences.

The information with this project set includes source code for an extensive project implementation. This source includes essentially the full set of project functionality, including reading the file and handling the keyboard and menu implementation. The author has tried to create good practice in design and code, but others might find better ways to carry out these operations. Other instructors are encouraged to look at this code critically in order to determine whether it meets their standards

for examples and to share any improvements with the author so they can be incorporated in the example for others.

An important part of making a project such as this accessible to the student is to provide information that describes how the atoms in the molecules should be displayed. This information is in the file `molmodel.h` that is provided with this project. The file includes atom names, colors, and sizes so that a student can pick up information directly from the input file and match each atom name with the appropriate sizes and colors for the display.

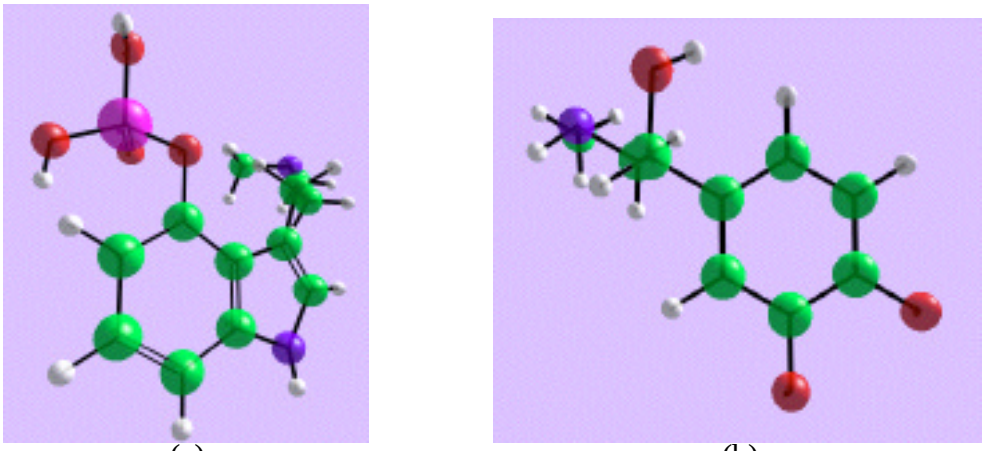
The display of a program that satisfies this project will be something like the images in the figure below from two different molecule description files. These figures use atom names and positions from the atom-position information in the molecule description files and bond linkages and types from those files, along with colors and sizes from the `molmodel.h` file, to achieve a standard look for the molecule. In order to get students to look at the chemistry, however, and not just the images, it probably is necessary to have them work with several different molecule files and try to relate the images with specific chemical questions such as the kind of reactions in which the molecules participate. We hope that the set of sample molecule files included with this module will provide instructors enough examples that students can make these connections, but the author is not a chemist and it is will certainly be useful to talk with chemists at your institution to find out how to make your projects relate directly to their individual course content.

The molecular model file will give you the positions and identities of each atom and the linkages between individual atoms in the molecule. These are read into arrays by two functions that are available at the online site, and the contents of the arrays are parsed to determine the color, size, and position of each atom and the linkages between them. Declarations for the arrays and code to produce the images is:

```
// data for storing molecular description
int      natoms, nbonds;
typedef struct atomdata {
               float x, y, z;
               char  name[5];
               int   colindex;
               } atomdata;
atomdata atoms[AMAX];
typedef struct bonddata{
               int first, second, bondtype;
               } bonddata;
bonddata bonds[BMAX];

void molecule(void) {
#define ANGTOAU(ang)  ((ang)*0.529177)
#define DBGAP 0.05

   int i, j, startindex, endindex, bondtype;
   GLUquadric *atomSphere;

   GLfloat color1[]={1.0, 0.0, 0.0, 0.7};
   GLfloat color3[]={0.0, 1.0, 1.0, 1.0};
   GLfloat mat_shininess[]={ 50.0 };

   glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, mat_shininess );

// use the location and link material that was read from the file
// to display the molecule
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, bondBlack );
```

```
      glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, bondBlack );
      for (i=0; i<nbonds; i++)
         {  // draw the bonds - note that the file is 1-origin indexed
            // while our tables are 0-origin indexed (language is C)
            startindex = bonds[i].first-1;      //so we decrease index by 1
            endindex   = bonds[i].second-1;
            bondtype   = bonds[i].bondtype;
            if (bondtype == 1) {
               glLineWidth(5.0);
               glBegin(GL_LINE_STRIP);
                  glVertex3f(atoms[startindex].x,atoms[startindex].y,
                     atoms[startindex].z);
                  glVertex3f(atoms[endindex].x,atoms[endindex].y,
                     atoms[endindex].z);
               glEnd();
            }
            if (bondtype == 2) {
               glLineWidth(3.0);
               glBegin(GL_LINE_STRIP);
                  glVertex3f(atoms[startindex].x-DBGAP,
                     atoms[startindex].y-DBGAP, atoms[startindex].z-DBGAP);
                  glVertex3f(atoms[endindex].x-DBGAP,atoms[endindex].y-DBGAP,
                     atoms[endindex].z-DBGAP);
               glEnd();
               glBegin(GL_LINE_STRIP);
                  glVertex3f(atoms[startindex].x+DBGAP,
                     atoms[startindex].y+DBGAP, atoms[startindex].z+DBGAP);
                  glVertex3f(atoms[endindex].x+DBGAP,atoms[endindex].y+DBGAP,
                     atoms[endindex].z+DBGAP);
               glEnd();
            }
         }

      for (i=0; i<natoms; i++)
         {  // draw the atoms
            glPushMatrix();
            atomSphere=gluNewQuadric();
            j = atoms[i].colindex;  // index of color for atom i
            glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, atomColors[j] );
            glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, atomColors[j] );
            glTranslatef(atoms[i].x, atoms[i].y, atoms[i].z);
            gluSphere(atomSphere, ANGTOAU(atomSizes[j]), GRAIN, GRAIN);
            glPopMatrix();
         }
      glutSwapBuffers();
   }
```

The presentation of the molecules is simply a display of the geometry that you interpret from the arrays. The examples shown in Figure 10.1 below use some blending to show the bond structure from the center of each atom and use a single white light to display the shape of the spheres. It uses the standard sizes of the atoms, although one could also use the Bohr radius of atoms to create a view without exposed bonds.



(a) (b)

Figure 10.1: (a) Example image from the file psilocybin.mol,
(b) Example image from the file adrenaline.pdb

This example can include a great deal of user interaction, using the keyboard and menu callbacks and event handlers to view the molecule in several different ways. The keyboard callback could control rotation of the molecule (in the three standard axes) and zooming the view in or out. A menu callback could control other parts of the display such as the size and transparency of atoms, and could provide an alternate to keyboard zoom in/out control so students can compare keyboard and menu functionality for detailed control.

This example could easily be extended would be to present two molecules in side-by-side viewports and allow the user to select one and move it around to compare it to the other. This has application in understanding similarity of structures. Another extension could present two molecules in a single window and allow one to be manipulated in trying to dock it with the other. This is a more complex issue because it could involve computing collisions and might need things like cutaway or partially transparent molecules, but it has very important applications and might be workable for simple molecules.

2. Displaying the conic sections

One of the common things students hear in algebra and calculus is that the graphs of quadratic expressions are all curves produced by conic sections, by intersections of a plane with a cone. This project allows students to generate the various conic sections by drawing a cone and clipping that cone with a plane, and to observe the shapes of the resulting curves.

The computer modeling is fairly straightforward. It involves creating a dual cone through simple triangle strips, defining the axes, and adding a clipping plane. The event handling and callbacks for the menu and keyboard control are straightforward and are omitted, allowing us to focus on creating the conic section and the axes, and about seeing the clipping plane work.

```
void cone(void)
{
   ... // declarations and material definitions here

   aStep = 3.14159/(float)GRAIN;
   glBegin(GL_TRIANGLE_FAN);     // top half of double cone
   glVertex3f(0.0, 0.0, 0.0);
      for (i=0; i<=GRAIN; i++) {
         myAngle = (float)i*aStep*2.0;
```

```
            glVertex3f(RADIUS*cos(myAngle), RADIUS*sin(myAngle), HEIGHT);
         }
      glEnd();
      glBegin(GL_TRIANGLE_FAN);     // bottom half of double cone
         glVertex3f(0.0, 0.0, 0.0);
         for (i=0; i<=GRAIN; i++) {
            myAngle = -(float)i*aStep*2.0;
            glVertex3f(RADIUS*cos(myAngle), RADIUS*sin(myAngle), -HEIGHT);
         }
      glEnd();
   }

   void display( void )
   {
      ... // viewing and other operations here
      // note that the axes are drawn with the clip plane *OFF*
      glDisable(GL_CLIP_PLANE1);
      drawAxes(3.0);
      glEnable(GL_CLIP_PLANE1);
      glClipPlane(GL_CLIP_PLANE1, myClipPlane);
      cone();
      glutSwapBuffers();
   }
```

Figure 10.2 below shows two screen captures from the sample that illustrate the example's operation, and it is straightforward experiment and to try out the interaction. Because the object of the work is to see the conic sections in the context of the cone itself, the visual communication in the example is simply presenting the geometry in the interactive setting where the user can see the section in context. A menu callback controls the orientation of the clipping plane itself. This is a particularly good project for getting students to think about how interface options work for this kind of geometric process.

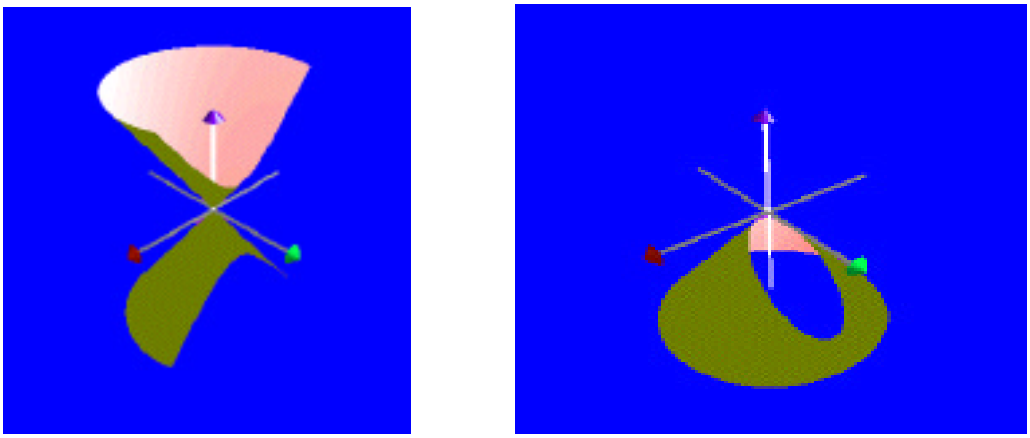


Figure 10.2: examples of two conic section displays (left: hyperbola; right: ellipse)

Representing a real function of two variables

The graphing of surfaces was discussed in the previous chapter on science examples, and the basic modeling layout for this graphing is given again in Figure 10.3 below. We will consider these kinds of surfaces from a more sophisticated point of view here, because we have the tools of lighting and shading to show the shape of the surfaces in more realistic ways. This "realism" is, of course, only apparent, and we will consider whether or not it actually adds to the science content of

these graphs. We would remind you that the surfaces we create in this way are only approximations of the real surfaces, and special cases of functions such as points of discontinuity must be considered separately.

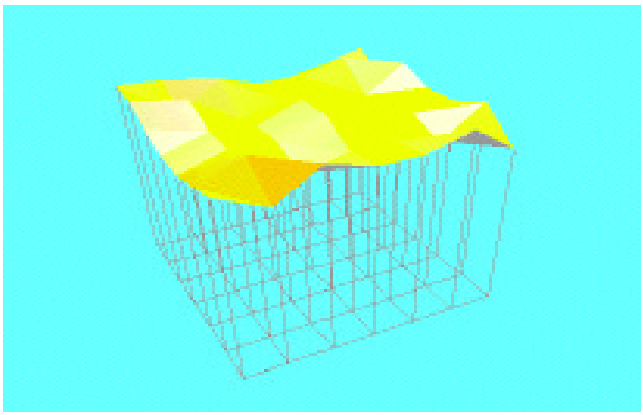


Figure 10.3: mapping a domain rectangle to a surface rectangle

3. Mathematical functions

If we consider a function of two variables, `z=f(x,y)`, acting on a contiguous region of real two-space, then the set of points `(x,y,f(x,y))` forms a surface in real three-space. This project explores such surfaces through processes that are described briefly in the figure above.

The goal of the example is the same as it was when this example was first seen in the earlier chapter on science examples: to see the fundamental principles of surfaces by creating a rectangular domain in X-Y space, evaluating the function at a regular grid of points in that domain, and creating and displaying small rectangles on the surface that correspond to a small rectangle in the underlying grid. This time, however, we shall view the graph using lighting and shading models to treat it as if it were an actual surface. More precisely, we will use two-sided surfaces with different colors above and below, so you can clearly see how the view goes from one side to the other when the surface is rotated. This will also also allow you to distinguish the two sides when some other behaviors occur, such as surfaces that show their underside at the edge of the domain. Figure 10.4 below shows such a surface with a yellow top-surface color and with three lights, red, green, and blue, evenly spaced around the surface. This example can use keyboard or mouse rotation controls and is a good introduction to event-driven programming with callbacks.

The code to implement this display is much the same as it was for the earlier version of the example, of course, but it adds the work to define the lighting and shading information for rendering the surface. This rendering involves a straightforward computation of the normal to a triangle as the cross product of two edge vectors, each of which is computed as the difference between two triangle vertices. Only the code for these computations is included here; the code to set up the example is the same as in the earlier chapter, and the code to handle interactions, materials, lights, and the like is omitted.

```
...
   for ( i=0; i<XSIZE; i++ )
      for ( j=0; j<YSIZE; j++ ) {
         x = XX(i);
         y = YY(j);
         vertices[i][j] = 0.3*cos(x*x+y*y+t); break;
      }
```

```
      // actually draw the surface */
      for ( i=0; i<XSIZE-1; i++ )
         for ( j=0; j<YSIZE-1; j++ ) {
            // first triangle in the quad, front face
            glBegin(GL_POLYGON);
               vec1[0] = XX(i+1)-XX(i);
               vec1[1] = YY(j)-YY(j);
               vec1[2] = vertices[i+1][j]-vertices[i][j];
               vec2[0] = XX(i+1)-XX(i+1);
               vec2[1] = YY(j+1)-YY(j);
               vec2[2] = vertices[i+1][j+1]-vertices[i+1][j];
               triNormal[0] = vec1[1] * vec2[2] - vec1[2] * vec2[1];
               triNormal[1] = vec1[2] * vec2[0] - vec1[0] * vec2[2];
               triNormal[2] = vec1[0] * vec2[1] - vec1[1] * vec2[0];
               glNormal3fv(triNormal); // hack together the normal vector...
               glVertex3f(XX(i),  YY(j),  vertices[i  ][j  ]);
               glVertex3f(XX(i+1),YY(j),  vertices[i+1][j  ]);
               glVertex3f(XX(i+1),YY(j+1),vertices[i+1][j+1]);
            glEnd();

            // second triangle in the quad, front face
            glBegin(GL_POLYGON);
               ... //   similar to above
               glNormal3fv(triNormal);
               glVertex3f(XX(i),  YY(j),  vertices[i  ][j  ]);
               glVertex3f(XX(i+1),YY(j+1),vertices[i+1][j+1]);
               glVertex3f(XX(i),  YY(j+1),vertices[i  ][j+1]);
            glEnd();
         }
   ...
```

The display is very simple because the goal of the project is to understand the shape of the function surface, but it is worth comparing this plastic-like surface with the display from the earlier version of this example to see how much smoother and more “realistic” this graph is.

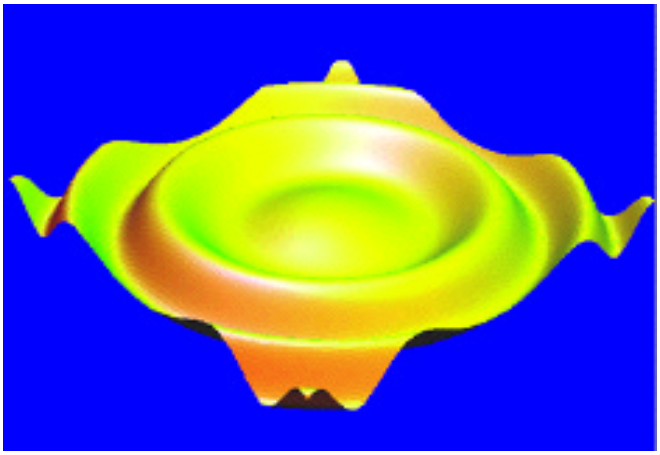


Figure 10.4: an example of a function surface display

We suggest that you not only think about the example here, but also work with a small number of functions whose shape is well understood. There are a number of sources of such functions. We encourage you to look at courses in physics or chemistry, or in references such as the CRC tables of mathematical functions for curious and interesting formulas whose surfaces are not immediatly

obvious, with a goal of having the surface graphs help you with this understanding. As we noted for the first version of this example, some interesting functions to consider for this are:

```
1. z = c * (x^3 - 3*x*y^2)
2. z = c * (x^4/a^4 - y^4/b^4)
3. z = (a*x^2 + b*y^2)/(x^2+y^2)
4. z = c*cos(2*_*a*x*y)
5. z = c*cos(2*_*a*x)*(2*_*a*y)
6. z = c*log(a*x^2 + b*y^2)
```

Later development of this example can also include animation by including a single parameter in the function definition and changing the value of the single parameter in the surface function. In effect, this examines the relationships in a one-parameter family of surfaces as that parameter is changed smoothly. The parameter's values can be stepped along by the idle callback function to allow the student to work with a more complex set of surfaces and in particular to see how the value of the function parameter affects the shape of the surface. This animation can be combined with rotation and even clipping controls so that the student can move the surface around as it is animating, though desktop systems do not seem to have enough speed to do this easily. Animations such as this are very interesting, so you are encouraged to explore them whenever possible.

This example not only allows you to use menu and keyboard controls for function selection and world rotation; it also allows you to see the difference between the speeds of different kinds of systems and graphics cards. If this is of interest and enough resources are available, then some speed comparisons can be a good thing, and this can be expanded to compare how quickly different kinds of programs execute.

4. Surfaces for special functions

Functions that are given by formulas in closed form are almost always analytic at almost all points in the domain. Special cases such as zero denominators or undefined transcendental functions usually disturb that analytic nature only at discrete points. However, there are other kinds of functions that exhibit more unusual behavior. One such kind of function is everywhere continuous but nowhere differentiable. Computer graphics can allow students to experiment with such functions and get a better understanding of their behavior.

As an example of a function of a single variable that is everywhere continuous but nowhere differentiable, consider the Weierstrass function defined by the convergent infinite series

$$f(x) = \sum_i \sin(x*2^i)/2^i$$

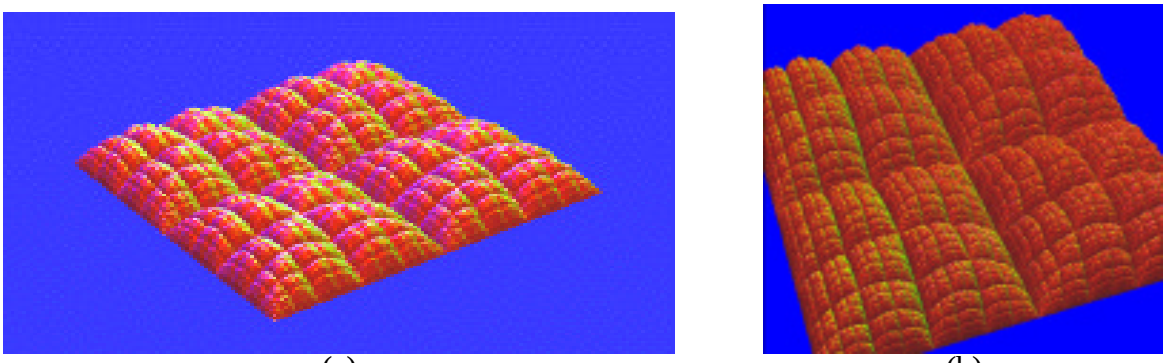
over all positive integers i. This can easily be extended to a function of two variables with the same property by defining `F(x,y) = f(x)*f(y)`, and the surface would be useful for the student to see. For computational purposes, however, it is probably better (for speed purposes) to use an algebraic function instead of the transcendental sine function, so we have developed an example that uses the function `x*(1-x)` instead of sin(x). This kind of example has been called a *blancmange* function (after a traditional British holiday pudding whose surface is extremely wrinkled) and the surface for this example is shown in the figure below, both at modest and high resolutions.

The computer modeling for this is straightforward, with the array `vertices[][]` now created through the use of the blancmange() function instead of the function in the last example.

```
// Calculate the points of the surface on the grid
   for ( i=0; i<XSIZE; i++ )
      for ( j=0; j<YSIZE; j++ ) {
```

```
            x = XX(i); y = YY(j);
            vertices[i][j] = blancmange(x,y,ITER);
         }
      ...

float blancmange(float x, float y, int n)
{
   float retVal, multiplier;
   int i;

   retVal = 0.0;
   multiplier = 0.5;
   for (i=0; i<n; i++) {
      multiplier = multiplier * 2.0;
      retVal = retVal + zigzag(x*multiplier,y*multiplier)/multiplier;
   }
   return (retVal);
}

float zigzag(float x, float y)
{
   float smallx, smally;
   int intx, inty;

   smallx = fabs(x); intx = (int)smallx;
   smally = fabs(y); inty = (int)smally;
   smallx = smallx - intx;             // move x and y between 0 and 1
   smally = smally - inty;
   smallx = smallx * (1.0 - smallx);   // maximum if value was 1/2
   smally = smally * (1.0 - smally);   // minimum if value was 0 or 1
   return (4.0*smallx*smally);         // scale to height 1
}
```

The surface is presented in Figure 10.5 just as in the previous example, but in addition to the rotation control there is also an iterate control that allows the user to take finer and finer approximations to the surface up to the limit of the raster display.



(a) (b)

Figure 10.5: (a) the blancmange surface, (b) zoomed in with more iterations

5. Electrostatic potential function

The electrostatic potential function `P(x,y)` was discussed in the earlier chapter on science examples. As we did with the general mathematical function above, it is worth revisiting in order

to see whether the more sophisticated graphics we now have available will help us understand the nature of this function better than the simple height-as-color approach. The example produces the result shown in Figure 10.6 below, with a standard lighting model and standard material definitions, including a set of coordinate axes to show the position of the domain in the plane. This presentation of the example should be compared with the earlier one to see which better helps you understand the nature of the "curved space" created by the electrostatic charge.

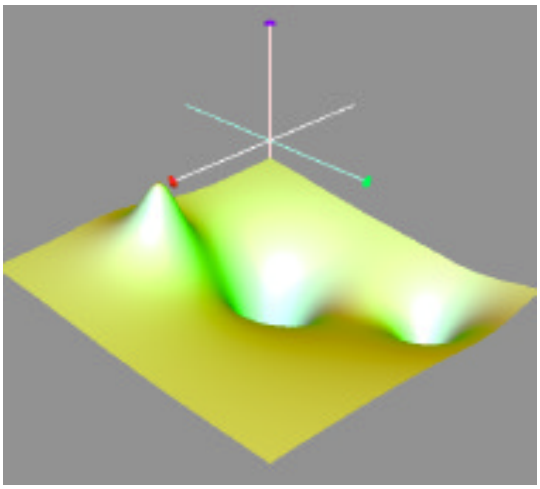


Figure 10.6: the coulombic surface from three point charges (one positive, two negative) in a plane

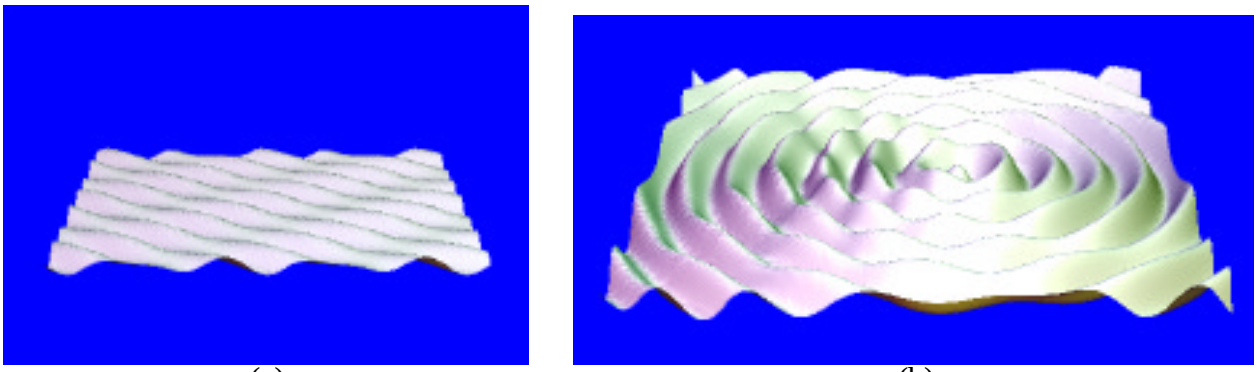
With the use of callbacks to build interactive controls in your programs, you can select a charge (using either a menu or a mouse selection) and can experiment with this surface by moving the charged point around the plane, by changing the amount of charge at the point, or by adding or deleting point charges. This can make an interesting example for interaction.

6. Interacting waves

There are many places in physics where important topics are modeled in terms of wave phenomena. It is useful to visualize behavior of waves, but single wave functions are very simple to model and display. It is more interesting to consider the behavior of multiple wave functions acting simultaneously, giving rise to different kinds of interaction phenomena. We might consider wavefronts that travel in parallel, or wavefronts that travel circularly from a given points. The basic modeling for these wave simulations treats the combined wave as a function that is the sum of the two basic wave functions.

Modeling the interactive waves, then, operates in exactly the same way as the function surfaces above. You pick an appropriate domain, divide the domain into a grid, compute the value of the function at each point on the grid, and display each of the grid rectangles as two triangles. The code for the train and circular wave functions shown in Figure 10.7 below is:

```
// some sample circular and train wave functions
#ifdef CIRCULAR
#define f1(x,y) 0.2*cos(sqrt((3.0*(x-3.14))*(3.0*(x-3.14))+(3.0*y)*(3.0*y)+t))
#define f2(x,y) 0.5*cos(sqrt((4.0*(x+1.57))*(4.0*(x+1.57))+(4.0*y)*(4.0*y)+t))
#endif

#ifdef TRAIN
#define f1(x,y)  0.1*sin(3.0*x+2.0*y+t)
#define f2(x,y)  0.2*sin(2.0*x+3.0*y+t)
#endif
```



(a) (b)

Figure 10.7: (a) two wave trains intersecting at a shallow angle,
(b) two circular waves whose origins are offset by 3π/2

These figures are presented as animations with the `idle()` function updating the parameter `t` in the definition of the wave functions. This provides the student with a picture of the changes in the waveforms as they move along, something that no static wave simulation can hope to provide. The faster the computer, of course, the faster the wave will move, and if the wave motion is too fast one can always reduce the step in `t` in the idle event handler shown below.

```
void animate(void) {
   t += 0.7;
   glutPostRedisplay();
}
```

The images in the figure provide examples of these interactions where the amplitudes and frequencies of the waves vary and the wave propagation parameters are different. Students may want to examine questions about the behavior of waves; two frequencies that are very nearly the same lead to beat phenomena, for example. This would lead to interactive projects extensions of this approach, allowing students to vary the different parameters of the waves through menus, keystrokes, or other techniques. In addition to this simple two-wave problems, it is possible to model waves in the open ocean by adding up a number of simple waveforms of different directions, different frequencies, and different amplitudes. So this area is a rich source of questions that students can examine visually.

Representing more complicated functions

7. Implicit surfaces

There are many places where one finds a real-valued function of three variables. As one example, consider the Coulomb's law case in the previous project. This law operates in three space as well as two space — the potential at any point is the sum over all the charged particles of the charge divided by the distance from the point to the particle. Other examples include gravitational forces and the density of material in the universe. Note that the function may be known from theoretical sources, may be inferred from experimental data, or may even be only known numerically through interpolation of sample values in a spatial volume. We focus on the function with a known mathematical expression because that is easiest to program, but the surface-finding process we describe here is very important in many scientific areas.

The difference between the 3-space situation and the 2-space situation is significant, though. We cannot display the graph of a function of three variables, because that graph only lives in 4-space.

Thus we must look for other ways to examine the behavior of these functions. A standard technique for this problem is to identify where the function has a constant value, often called an *implicit* surface. Creating implicit surfaces, or giving a user a way to identify the shape of these surfaces, is an important tool in understanding these functions. The critical question in this examination is to find those points in 3-space for which the function has a particular value.

One technique is to create an approximation to the actual implicit surface by creating a grid of points in the space and asking ourselves whether the function takes on the value we seek within each of the regions defined by the grid. If the function is (or may be assumed to be) continuous it is easy enough to answer that question; simply calculate the function's value at each of the eight vertices of the region and see whether the function's value is larger than the critical value at some vertices and smaller at others. If this is the case, the intermediate value theorem from calculus tells us that the function must pass through the critical value somewhere in the region.

Once we know that the function takes on the desired value in the region, we can take two approaches to the problem. The first, and easiest, is simply to identify the region by drawing something there. In the figure below, we draw a simple sphere in the region and use standard light and material properties to give shape to the spheres. (The function is `f(x,y,z)=x*y*z`, and for any given value `c` the shape of the surface `x*y*z=c` is basically hyperbolic.) The basic code for this includes functions to identify the point in the domain that corresponds to the indices in the grid, and the code to scan the 3D grid and draw a sphere in each grid space the surface goes through.

```
GLfloat XX(int i) {
   return (MINX+((MAXX-MINX)/(float)(XSIZE-1))*(float)(i));
   }

GLfloat YY(int i) {
   return (MINY+((MAXY-MINY)/(float)(YSIZE-1))*(float)(i));
   }

GLfloat ZZ(int i) {
   return (MINZ+((MAXZ-MINZ)/(float)(ZSIZE-1))*(float)(i));
   }

...
// Identify grid points where the value of the function equals the constant
   rad = 0.7*(MAXX-MINX)/(float)XSIZE;
   for ( i=0; i<XSIZE; i++ )
         for (j=0; j<YSIZE; j++)
               for ( k=0; k<ZSIZE; k++ ) {
                     x = XX(i); x1 = XX(i+1);
                     y = YY(j); y1 = YY(j+1);
                     z = ZZ(k); z1 = ZZ(k+1);
                     p1 = f( x, y, z);
                     p2 = f( x, y,z1);
                     p3 = f(x1, y,z1);
                     p4 = f(x1, y, z);
                     p5 = f( x,y1, z);
                     p6 = f( x,y1,z1);
                     p7 = f(x1,y1,z1);
                     p8 = f(x1,y1, z);
                     if (  ((p1-C)*(p2-C)<0.0) || ((p2-C)*(p3-C)<0.0) ||
                           ((p3-C)*(p4-C)<0.0) || ((p1-C)*(p4-C)<0.0) ||
                           ((p1-C)*(p5-C)<0.0) || ((p2-C)*(p6-C)<0.0) ||
                           ((p3-C)*(p7-C)<0.0) || ((p4-C)*(p8-C)<0.0) ||
```

```
                        ((p5-C)*(p6-C)<0.0) || ((p6-C)*(p7-C)<0.0) ||
                        ((p7-C)*(p8-C)<0.0) || ((p5-C)*(p8-C)<0.0)  ) {
                              mySphere(x,y,z,rad);
                        }
            }
```

This works fairly well, because the smoothness of the spheres allows us to see the shape of the surface as shown in Figure 10.8 below. A second and more difficult process is to find all the points on the edges of the region where the function takes the critical value and use them as the vertices of a polygon, and then draw the resulting polygon. This would give better results, but the techniques used to identify the geometry of the polygon make this beyond the scope of an introductory course.

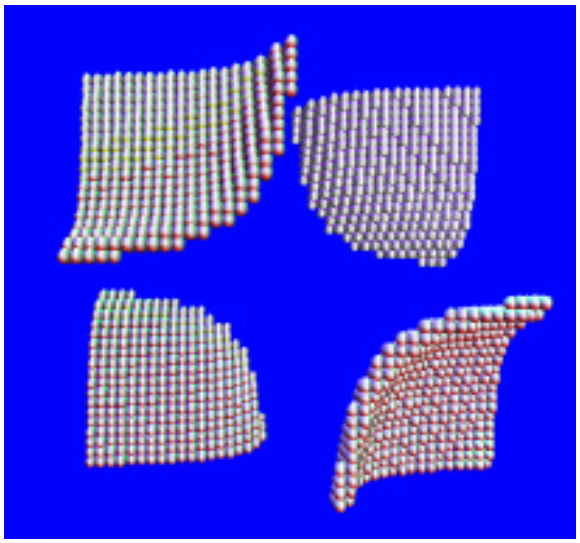


Figure 10.8: an implicit surface approximation that uses spheres to indicate surface location

8. Cross-sections of volumes

Another technique for seeing the behavior of a function of three variables is to give the user a way to see the function values by displaying the values in a planar cross-section of the region. We could take the cross-section as a 2-dimensional space and present the graph of the function as a 3D surface, but this would be confusing; each cross-section would have its own 3-D graph whose behavior is would be confusing in the 3D region. Instead, we use the 2D technique of associating colors with numerical values and drawing the cross-section in colors that reflect the values of the function. The figure below shows an example of three cross-sections, one parallel to each coordinate plane through the origin, for the function `sin(x*y*z)`.

The code for this process is shown below, where we only show the function being presented, the function that defines the cutting plane with constant X (the function functionX) and the surface rendering for that plane. The operations for the other two cutting planes are essentially identical to this.

```
// the function of three variables
float f(float x, float y, float z) {
   return sin(x*y*z);
}

// function for value of x with global constants AX, BX, CX, DX
```

```
// that determine the plane
float functionX( float y, float z) {
   return (-DX-BX*y-CX*z)/AX;
}

void surfaceX(void)
{
// define a point data type and general variables
   point3 myColor;
   int  i, j;
   float x, y, z, c;

// Calculate the points of the surface on the grid
   for ( i=0; i<XSIZE; i++ )
      for ( j=0; j<YSIZE; j++ ) {
         vertices[i][j][1]=y=MINY+(MAXY-MINY)*((float)j/(float)(YSIZE-1));
         vertices[i][j][2]=z=MINZ+(MAXZ-MINZ)*((float)i/(float)(ZSIZE-1));
         vertices[i][j][0]=x=functionX(y,z);
      }

// draw the surface with quads because surface is planar
   for ( i=0; i<XSIZE-1; i++ )
      for ( j=0; j<YSIZE-1; j++ )      // for each quad in the domain
      {  // compute "average" point in the quad
         x = (vertices[i][j][0] + vertices[i+1][j][0] +
              vertices[i][j+1][0] + vertices[i+1][j+1][0])/4.0;
         y = (vertices[i][j][1] + vertices[i+1][j][1] +
              vertices[i][j+1][1] + vertices[i+1][j+1][1])/4.0;
         z = (vertices[i][j][2] + vertices[i+1][j][2] +
              vertices[i][j+1][2] + vertices[i+1][j+1][2])/4.0;
         c = f(x,y,z);  // compute function at the "average" point
         getColor(c, &r, &g, &b);
         glColor3f(r, g, b);     // color determined by value of function
         if ((i==0)&&(j==0)) glColor3f(1.0,0.0,0.0);
         glBegin(GL_POLYGON);
            glVertex3fv(vertices[i  ][j  ]);
            glVertex3fv(vertices[i+1][j  ]);
            glVertex3fv(vertices[i+1][j+1]);
            glVertex3fv(vertices[i  ][j+1]);
         glEnd();
      }
}
```

The display itself is shown in Figure 10.9 below; note how the cross-sections match at the lines where they meet, and how the function changes depending on how large the value of the fixed x, y, or z is for the plane. The display includes user-controlled rotations so the user can see the figure from any angle, and also includes keyboard control to move each of the cutting planes forward and backward to examine the behavior of the function throughout the space.

Usually we use a standard smooth color ramp so the smoothness of the function can be seen, but we can use an exceptional color for a single value (or very narrow range of values) so that unique value can be seen in a sort of contour band. The color ramp function is not included here because it is one of the standard ramps discussed in the visual communication section.

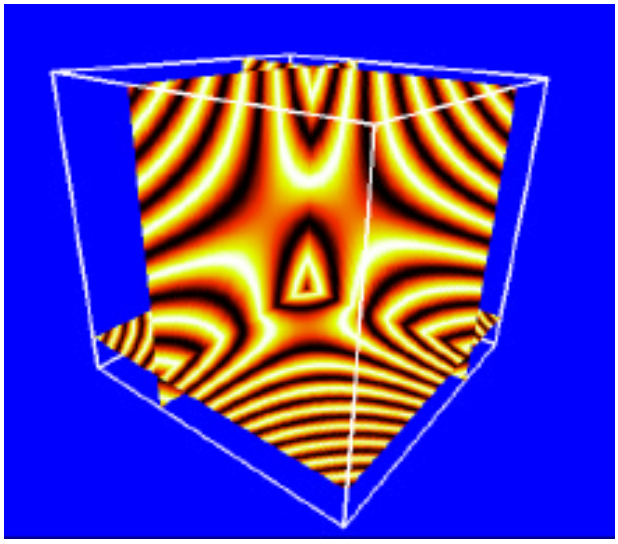


Figure 10.9: cross-sections of a function's values

9. Vector displays

A different problem considers displaying functions of two variables with two-variable range, where you must display this essentially four-dimensional information to your audience. Two examples of this are vector-valued functions in 2-space, or complex-valued functions of a single complex variable. In particular, Figure 10.10 below presents two examples: a system of two first-order differential equations of two variables (left) and a complex-valued function of a complex variable (right). The domain is the standard rectangular region of two-dimensional space, and we have taken the approach of encoding the range in two parts based on considering each value as a vector with a length and a direction. We encode the length as pseudocolor with the uniform color ramp as described above, and the direction as a fixed-length vector in the appropriate direction.

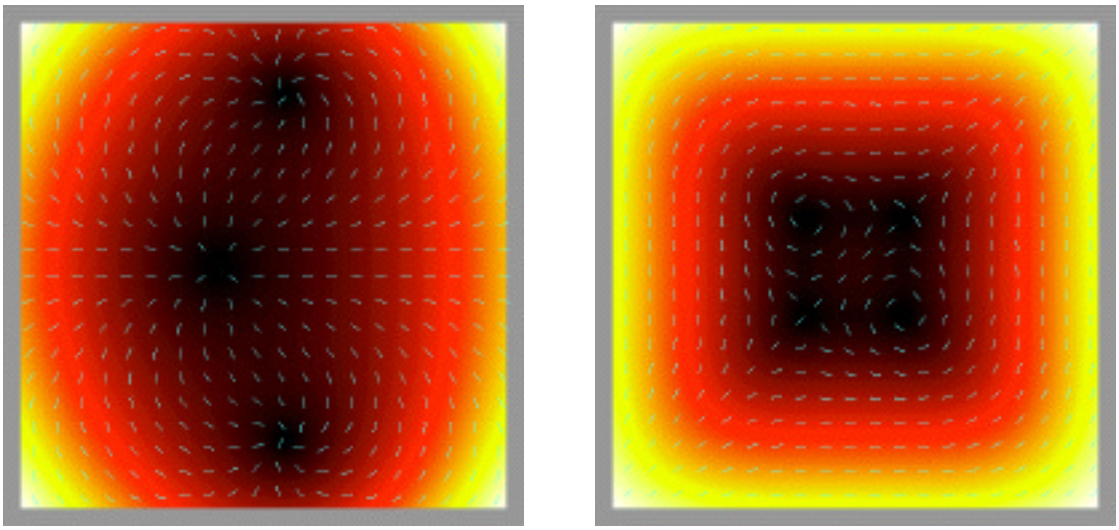


Figure 10.10: two visualizations: a function of a complex variable (left)
and a differential equation (right)

The leftmost figure is based on the complex-valued function `f(z)=z^3+12z+12` for complex numbers `z`. This function is evaluated at each point of a grid, and the result is a complex number.

This number, like every complex number, can be viewed as a vector and as such has a magnitude and a direction. We set the color at the vertex by the magnitude of the vector and draw a vector at the grid point with the direction of the vector, allowing the user to see both these properties of the function simultaneously.

Code for this process is listed below, with the color ramp treated a little differently than in other examples: the `calcEven(...)` function is assumed to take a value between 0 and 1 and return a value in a global variable `myColor[3]` instead of as three real numbers. The colors are also calculated based on triangles instead of quads because this was created by adapting a surface plotting function; quads would have worked equally well.

```
// Calculate the points of the surface on the grid and log min and max
   for ( i=0; i<XSIZE; i++ )
      for ( j=0; j<YSIZE; j++ ) {
         x = 0.5*(XX(i)+XX(i+1));
         y = 0.5*(YY(j)+YY(j+1));;
         getVector(x, y, &u, &v);
         vectors[i][j][0] = u; vectors[i][j][1] = v;
         length[i][j] = getLength(u,v);
         if (length[i][j] > YMAX) YMAX = length[i][j];
         if (length[i][j] < YMIN) YMIN = length[i][j];
      }
...

float getLength(float a, float b)
{  return (sqrt(a*a+b*b)); }

void getVector(float x, float y, float *u, float *v) {
// w = z^3+12z+12; value is complex number w
   *u = x*x*x - 3.0*x*y*y +12.0*x + 12.0;
   *v = 3.0*x*x*y - y*y*y +12.0*y;
}

void surface(void) {
   int i, j;
   float yavg, len, x, y;

   YRANGE = YMAX - YMIN;
// draw the vector and the surface
   for ( i=0; i<XSIZE-1; i++ )
      for ( j=0; j<YSIZE-1; j++ ) {
         // draw the direction vector
         if ((i%10 == 5) && (j%10 == 5)) {
         glColor4f(0.0, 1.0, 1.0, 1.0);
         x = 0.5*(XX(i)+XX(i+1));
         y = 0.5*(YY(j)+YY(j+1));
         glBegin(GL_LINE_STRIP);
            glVertex3f(x,y,0.0);
            glVertex3f(x,y,EPSILON);
            len = 5.0 * sqrt(vectors[i][j][0]*vectors[i][j][0]+
                        vectors[i][j][1]*vectors[i][j][1]);
            glVertex3f(x+vectors[i][j][0]/len,
                        y+vectors[i][j][1]/len, EPSILON);
         glEnd();
         }
// first triangle in the quad
         glBegin(GL_POLYGON);
```

```
            yavg = (length[i][j]+length[i+1][j]+length[i+1][j+1])/3.0;
            calcEven((yavg-YMIN)/YRANGE);
            glColor3f(myColor[0],myColor[1],myColor[2]);
            glVertex3f(XX(i),YY(j), 0.0);    // colors in the plane only
            glVertex3f(XX(i+1),YY(j), 0.0);
            glVertex3f(XX(i+1),YY(j+1), 0.0);
         glEnd();

// second triangle in the quad
         glBegin(GL_POLYGON);
            yavg = (length[i][j]+length[i][j+1]+length[i+1][j+1])/3.0;
            calcEven((yavg-YMIN)/YRANGE);
            glColor3f(myColor[0],myColor[1],myColor[2]);
            glVertex3f(XX(i),YY(j), 0.0);
            glVertex3f(XX(i+1),YY(j+1), 0.0);
            glVertex3f(XX(i),YY(j+1), 0.0);
         glEnd();
      }
}
```

Another use of this approach is to present the system of two differential equations

```
x' = y^2-1
y' = x^2-1
```

Here the vector `<x',y'>` is the result of the differential equation process, and we can present the vector directly using the magnitude and direction components as described above. This was the source of the right-hand display in the figure above.

10. Parametric curves

A curve may be defined mathematically by any function (usually a continuous function) from real one-space into real three-space. The function may be expressed through a function or through parametric equations. Function curves look like standard graphs, while parametric curves can have loops or other complex behavior. It is also possible to have curves defined by other processes, such as differential equations. A simple example of curves defined by functions is given by a standard helix:

x=a*sin(t)
y=a*cos(t)
z=t

Other curves can be more interesting and complex. Some can be derived by taking the parametric surfaces described above and making one of the u or v variables a constant; we will not write any of these explicitly. Others may come from different sources. A couple of interesting examples are the rotating sine wave:

x=sin(a*t)*cos(b*t)
y=sin(a*t)*sin(b*t)
z=c*t/(2*π)

or the toroidal spiral:

x=(a*sin(c*t)+b)*cos(t)
y=(a*sin(c*t)+b)*sin(t)
z=a*cos(c*t).

The modeling for this is done by dividing a real interval into a number of subintervals and calculating the point on the curve corresponding to each division point. These are connected in sequence by line segments. This is shown in the code fragment below:

```
void spiral(void)
   {
```

```
        int i;
        float a=2.0, b=3.0, c=18.0, t;

        glBegin(GL_LINE_STRIP);
        for ( i=0, t=0.0; i<=1000; i++ ) {
           glVertex3f((a*sin(c*t)+b)*cos(t),(a*sin(c*t)+b)*sin(t),a*cos(c*t));
           t = t + .00628318;      //    2*PI/1000
        }
        glEnd();
    }
```

This example above is also shown in Figure 10.11 below. Here the visual communication is quite simple, with the main question being to show the shape of the curve. However, to let the user get the best feel for the shape, it is useful to use keyboard-controlled rotations to allow the user to see it from any viewpoint. Curves such as these are sufficiently complex that students should feel some satisfaction in seeing the results of their programs.

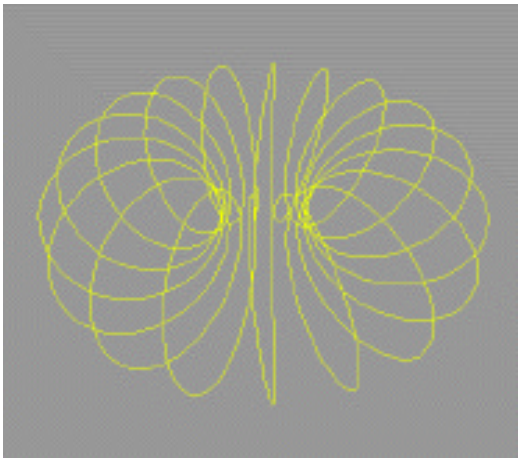


Figure 10.11: The toroidal spiral curve

11. Parametric surfaces

In the function surface projects above, the student was asked to start with a grid in the X-Y domain and compute a value of Z for each point in the grid. The resulting points (x,y,z) were then used to determine rectangles in real three-space that are each graphed as a pair of triangles. The grid need not be so directly involved in the surface, however. In a parametric surface project, we start with a grid in parameter space (which we will call U-V space). For each point (u,v) in the grid, three functions will give three real values for each of these points; these values are the x-, y-, and z-coordinates of the points that are to be graphed. So the difference between function surfaces and parametric surfaces is relatively small from the programming point of view.

The sample we present for parametric surfaces is the (3,4)-torus. This is a closed, genus-1 surface with triangular cross section that twists 4/3 times as it goes around the torus. This is an interesting shape, because it has a Möbius-like property that if you put your finger on one side of the triangle cross-section and hold it there, you will eventually return to the same place on the surface. The domain for the parameter space is $[-2\pi,2\pi]$ in both the u and v directions. Modeling the surface depends first on considering the domain as having three parts in one direction and many parts in the other direction. The three parts are mapped to the sides of a triangle, and the triangle is slowly rotated as you proceed along the many-sided direction, making a total twist of 480 degrees by the time it gets to the end. In practice, the space bounded by two of these triangles cannot readily be

managed with three quadrilaterals, but needs to be subdivided a number of times to handle the twist in the space appropriately. Code for the modeling includes several parts: the definition of the parametric functions that define the torus, and the functions to manage the various stages of the actual surface definition as described above.

```
//torus; note that this includes a parameter t for animating the surface
#define RNGRAD 4.0  // radius of torus ring
#define TUBRAD 2.0  // radius of torus tube
#define X(u,v) (RNGRAD+TUBRAD*cos(1.3333333*u+v+t))*cos(u)
#define Y(u,v) (RNGRAD+TUBRAD*cos(1.3333333*u+v+t))*sin(u)
#define Z(u,v) TUBRAD*sin(1.3333333*u+v+t)

float   umin=-3.14159, umax=3.14159, vmin=-3.14159, vmax=3.14159;

// functions to return u and v values for indices i and j respectively
GLfloat UU(int i) {
  return (umin+((umax-umin)/(float)(usteps-1))*(float)(i)); }
GLfloat VV(int j) {
  return (vmin+((vmax-vmin)/(float)(vsteps-1))*(float)(j)); }

void doSurface(void)
{
    int  i, j;
    float u, v;
    GLfloat yellow[]= {1.0, 1.0, 0.0, 1.0};
    GLfloat mat_shininess[]={ 30.0 };

//  Calculate the points of the surface boundaries on the grid
  for ( i=0; i<usteps; i++ )
     for ( j=0; j<vsteps; j++ )
     {
        u = UU(i);    v = VV(j);
        surface[i][j].x = X(u,v);
        surface[i][j].y = Y(u,v);
        surface[i][j].z = Z(u,v);
     }

// actually draw the surface
  for ( i=0; i<usteps-1; i++ )  // along torus -- larger
     for ( j=0; j<vsteps-1; j++ ) {   // around torus -- smaller
        glMaterialfv(GL_FRONT, GL_AMBIENT, yellow );
        glMaterialfv(GL_FRONT, GL_DIFFUSE, yellow );
        glMaterialfv(GL_BACK, GL_AMBIENT, yellow );
        glMaterialfv(GL_BACK, GL_DIFFUSE, yellow );
        glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, mat_shininess );
        doQuad(20,1,surface[i][j],surface[i+1][j],
                    surface[i][j+1], surface[i+1][j+1]);
     }
}

//   Divide quad into m strips vertically and images each separately.
void doQuad(int n,int m,surfpoint p0,surfpoint p1,surfpoint p2,surfpoint p3)
{
  int i;
  surfpoint A, B, C, D;   //  A and B are the top points of each separate
                          //  strip; C and D are the bottom points.

  for (i=0; i<m; i++) {
```

```
      A.x = (p0.x*(float)(m-i)   + p1.x*(float)i)/(float)m;
      A.y = (p0.y*(float)(m-i)   + p1.y*(float)i)/(float)m;
      A.z = (p0.z*(float)(m-i)   + p1.z*(float)i)/(float)m;
      B.x = (p0.x*(float)(m-i-1) + p1.x*(float)(i+1))/(float)m;
      B.y = (p0.y*(float)(m-i-1) + p1.y*(float)(i+1))/(float)m;
      B.z = (p0.z*(float)(m-i-1) + p1.z*(float)(i+1))/(float)m;

      C.x = (p2.x*(float)(m-i)   + p3.x*(float)i)/(float)m;
      C.y = (p2.y*(float)(m-i)   + p3.y*(float)i)/(float)m;
      C.z = (p2.z*(float)(m-i)   + p3.z*(float)i)/(float)m;
      D.x = (p2.x*(float)(m-i-1) + p3.x*(float)(i+1))/(float)m;
      D.y = (p2.y*(float)(m-i-1) + p3.y*(float)(i+1))/(float)m;
      D.z = (p2.z*(float)(m-i-1) + p3.z*(float)(i+1))/(float)m;
      doStrip(n, A, B, C, D);
   }
}

/*  Now we have one vertical strip that we must subdivide into n pieces,
    each of which will be two triangles.  We actually create a nx2 array
    of surfpoints, and then divide each quad going down the array into two
    triangles, each of which is "emitted" with its own function call.
*/

void doStrip(int n,surfpoint p0,surfpoint p1,surfpoint p2,surfpoint p3)
{
   int i, j;
   surfpoint A, B, buffer[3];

   for (i=0; i<=n; i++) {
      A.x = (p0.x*(float)(n-i) + p2.x*(float)i)/(float)n;
      A.y = (p0.y*(float)(n-i) + p2.y*(float)i)/(float)n;
      A.z = (p0.z*(float)(n-i) + p2.z*(float)i)/(float)n;
      B.x = (p1.x*(float)(n-i) + p3.x*(float)i)/(float)n;
      B.y = (p1.y*(float)(n-i) + p3.y*(float)i)/(float)n;
      B.z = (p1.z*(float)(n-i) + p3.z*(float)i)/(float)n;
      theStrip[i][0] = A;
      theStrip[i][1] = B;
   }
   // now manipulate the strip to send out the triangles one at a time
   // to the actual output function, using a rolling buffer for points.
   buffer[0] = theStrip[0][0];
   buffer[1] = theStrip[0][1];
   for (i=1; i<=n; i++)
      for (j=0; j<2; j++) {
         buffer[2] = theStrip[i][j];
         emit(buffer);
         buffer[0] = buffer[1];
         buffer[1] = buffer[2];
      }
}

// Handle one triangle as an array of three surfpoints.
void emit( surfpoint *buffer ) {
   surfpoint Normal, diff1, diff2;

   diff1.x = buffer[1].x - buffer[0].x;
   diff1.y = buffer[1].y - buffer[0].y;
   diff1.z = buffer[1].z - buffer[0].z;
```

```
    diff2.x = buffer[2].x - buffer[1].x;
    diff2.y = buffer[2].y - buffer[1].y;
    diff2.z = buffer[2].z - buffer[1].z;
    Normal.x = diff1.y*diff2.z - diff2.y*diff1.z;
    Normal.y = diff1.z*diff2.x - diff1.x*diff2.z;
    Normal.z = diff1.x*diff2.y - diff1.y*diff2.x;
    glBegin(GL_POLYGON);
       glNormal3f(Normal.x,Normal.y,Normal.z);
       glVertex3f(buffer[0].x,buffer[0].y,buffer[0].z);
       glVertex3f(buffer[1].x,buffer[1].y,buffer[1].z);
       glVertex3f(buffer[2].x,buffer[2].y,buffer[2].z);
    glEnd();
}
```

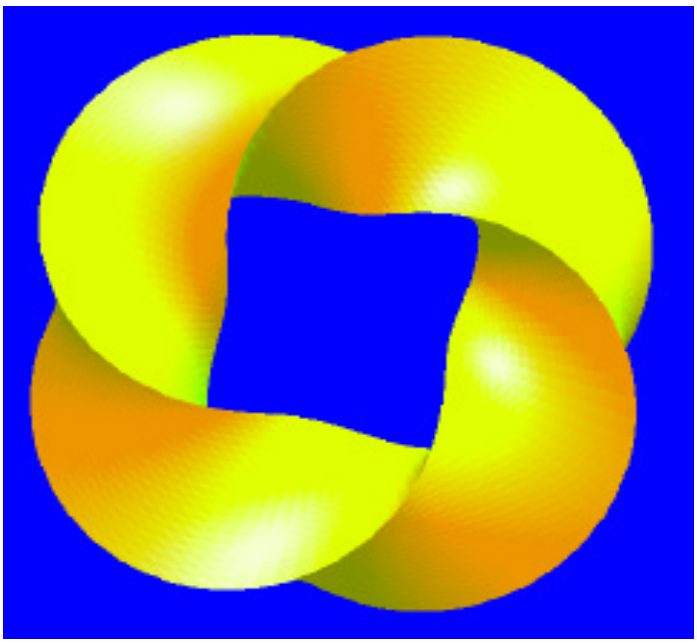


Figure 10.12: a parametric surface

The display of this surface in Figure 10.12 focuses on the surface shapes, which can be quite complex from certain viewpoints. Thus it is important to allow the student to have full rotation control in all three axes via the keyboard or the mouse. In addition, the surface rotates slowly in the plane of the figure, emphasizing that the display is dynamic. Code for the animation is quite routine and is omitted.

The differences between function surfaces and parametric surfaces can be immense. Function surfaces are always single-sheet: they always look like a horizontal sheet that has been distorted upwards and downwards, but never wraps around on itself. Parametric surfaces can be much more complex. For example, a sphere can be seen as a parametric surface whose coordinates are computed from latitude and longitude, a two-dimensional region; a torus can be seen as a parametric surface whose coordinates are computed from the angular displacement around the torus and the angular displacement in the torus ring, another pair of dimensions. The figure above shows an example of a parametric surface that encloses a volume, surely something a function surface could never do.

Many simpler surfaces can be created using parametric definitions, and these make good student projects. It is easy to define a simple torus in this way, for example; the functions are

x(u,v)=(A+B*cos(v))*cos(u);
y(u,v)=(A+B*cos(v))*sin(u); and
z(u,v)=B*sin(v) for real constants A and B

for values of u and v between 0 and 2*π, where A is the radius of the torus body and B is the radius of the torus ring. Compare these equations with the ones used to define the figure above and note the similarity. Note also how defining a grid on the (u,v) domain corresponds to the definition of the torus in the GLUT torus model. Other examples may be considered as well; a couple of these that you may wish to experiment with are:

Helicoid:

x(u,v)=a*u*cos(v),
y(u,v)=b*u*sin(v);
z(u,v)=c*v for real constants a, b, c

Conical spiral:

x(u,v)=a*[1-(v/2*π)]*cos(n*v)*[1+cos(u)]+c*cos(n*v);

y(u,v)=a*[1-(v/2*π)]*sin(n*v)*[1+cos(u)]+c*sin(n*v);

z(u,v)=(b*v/2*π)+a*[1-v/(2*π)]*sin*u) for real constants a, b, c and integer n

This area of surface exploration offers some very interesting opportunities for advanced studies in functions. For example, the parametric surface may not exist in 3-space but in higher-dimensional space, and then may be projected down into 3-space to be seen. For example, there may be four or more functions of u and v representing four or more dimensions. Familiar surfaces such as Klein bottles may be viewed this way, and it can be interesting to look at options in projections to 3-space as another way for the user to control the view.

Illustrating dynamic systems

12. The Lorenz attractor

Not all curves are given by simple algebraic equations. An excellent example of a curve given by a differential equation is given below; this describes a complex phenomenon having no closed-form solution is given by the system of differential equations:

```
dx/dt = s * (y - x)
dy/dt = r * x - y - x * z
dz/dt = x * y - b * z
```

for constants s, r, and b. These are the Lorenz equations and the curve they define, called the Lorenz attractor, is often described in discussions of strange attractors. Under certain circumstances (in particular, s=10, r=28, and b=8/3, used in the example in Figure 6.17 below) chaotic behavior occurs, and the set of curves given by the differential equations (with parameter t) are very interesting, but to present these curves well you need high-quality numerical integration. This may require tools you do not have in your programming environment, and the example that is included here only uses difference equation approximations.

This process is modeled by replacing the differential equations above by difference equations. This can be a questionable decision if we want to provide a fully accurate result, but we do not assume that the computer graphics student will necessarily have yet learned how to code numerical integration techniques. For now, we can only suggest that the user might want to reduce the size of the step in the animate() function below and iterate that several times in the course of making one step. Code for the simple difference equation approach is given below, followed by an illustration how the curve looks in Figure 10.13.

```
float  sigma = 10.0, r = 28.0, b = 2.66667;
```

```
void animate(void) {
   point3 delta;
   int i;
   float x, y, z;

   x = location[NumberSoFar][0];
   y = location[NumberSoFar][1];
   z = location[NumberSoFar][2];
   delta[0] = sigma*(y-x);
   delta[1] = r*x - y - x*x;
   delta[2] = x*y - b*z;
   for (i=0; i<3; i++)
      location[NumberSoFar+1][i] = location[NumberSoFar][i]+0.01*delta[i];
   NumberSoFar++;
   glutPostRedisplay();
}
```

The display presents the growing set of points so the user can watch the shape of the curve evolve. Because this curve lives in an interesting region of space, it is important that the user be able to move the curve around and look at its regions, so the program should allow the user to do this with either keyboard or mouse control.

Figure 10.13: the Lorenz curve

13. The Sierpinski attractor

Sometimes an attractor is a more complex object than the curve of the Lorenz attractor. Let us consider a process in which some bouned space contains four designated points that are the vertices of a tetrahedron (regular or not), and let us put a large number of other points into that space. Now define a process in which for each non-designated point, a random designated point is chosen and the non-designated point is moved to the point precisely halfway between its original position and the position of the designated point. Apply that process repeatedly. It may not seem that any order would arise from the original chaos, but it does — and in fact there is a specific region of space into which each of the points tends, and from which it can never escape. This is called the Sierpinski attractor, and it is shown in Figure 10.14.

The Sierpinski attractor is also interesting because it can be defined in a totally different way. If we recursively define a tetrahedron to be displayed as a collection of four tetrahedra, each of half the height of the original tetrahedron, each touching all the others at precisely one point, and each having one of the vertices of the original tetrahedron, then the limit of this definition is the attractor as defined and presented above.

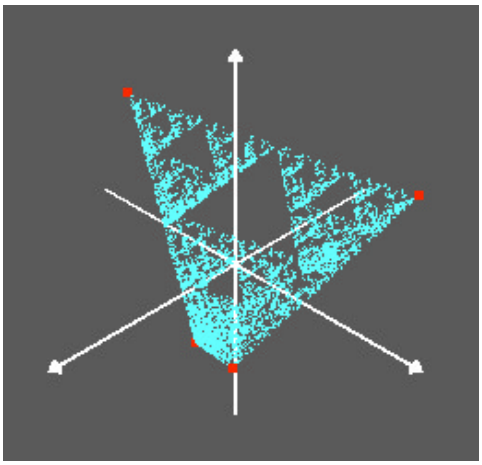


Figure 10.14: the Sierpinski attractor

*Some enhancements to the display*

Stereo pairs

If you create a window that is twice as wide as it is high, and if you divide it into left and right viewports, you can display two images in the window simultaneously. If these two images are created with the same model and same center of view, but with two eye points that simulate the location of two eyes, then the images simulate those seen by a person's two eyes. Finally, if the window is relatively small and the distance between the centers of the two viewports is reasonably close to the distance between a person's eyes, then the viewer can probably resolve the two images into a single image and see a genuine 3D view. Such a view is shown in Figure 10.15 below: a pair of views of `psilocybin.mol`, one of the molecules included in thise materials that has some 3D interest. None of these processes are difficult, so it would add some extra interest to include 3D viewing in the molecule display project or almost any other of these projects.

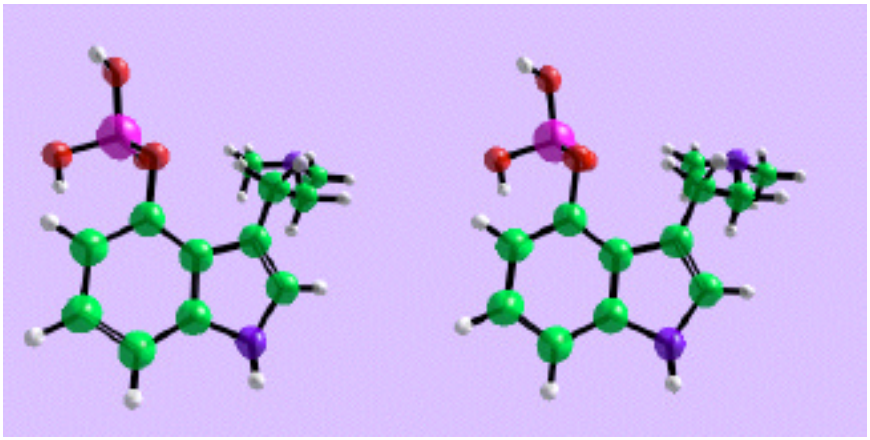


Figure 10.15: A stereo pair that the viewer may be able to resolve.

The code below manages the two viewports and performs the display in each; the display portion is not presented in order to keep the code short and to focus on the dual-viewport approach. This creates two viewports, defines them to have different viewing projections, performs the same

actual display operations (indicated by the ellipsis) on each, and then presents the completed display of both viewports at once to avoid the flicker of updating each separately.

```
void display( void )
{
// eye offset from center
   float offset = 1.0;

// left-hand viewport
   glViewport(0,0,300,300);
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt(-offset, 0.0, ep, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
   ...

// right-hand viewport
   glViewport(300,0,300,300);
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt(offset, 0.0, ep, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
   ...
   glutSwapBuffers();
}
```

A more complex kind of display — Chromadepth™ display — depends on texture mapping and will be described later.

*Credits:*

A number of colleagues have helped with graphical concepts, with scientific principles, or with models of the scientific issues. We apologize in advance for those we may miss, but we want to thank Michael J. Bailey of the San Diego Supercomputer Center (SDSC) for many fruitful discussions across all of these topics, and with Kris Stewart of San Diego State University (SDSU), Angela Shiflet of Wofford College, Rozeanne Steckler and Kim Baldridge of SDSU, and Ian Littlewood of California State University Stanislaus for their contributions in one or more of these areas.

*References:*

- Zimmermann, Walter and Steve Cunningham, *Visualization in Teaching and Learning Mathematics*, MAA Notes Number 19, Mathematical Association of America, 1991
- Banchoff, Tom et al., "Student-Generated Software for Differential Geometry," in Zimmermann and Cunningham, above, pp. 165-171
- von Seggern, David, *CRC Standard Curves and Surfaces*, CRC Press, 1993
- Textbook for multivariate calculus ...
- Textbook for differential geometry? ...

# Texture Mapping

*Prerequisites*

An understanding of the geometry of polygons in 3-space, a concept of interpolation across a polygon, and the concept of how values in an array can map linearly to values in another space.

*Introduction*

Texturing — applying textures to a graphical object to achieve a more interesting image or add information to the image without computing additional geometry — is a significant addition to the capabilities of computer graphics. Texturing is a rich topic and we will not try to cover it in depth, but we will discuss something of what is possible as we develop the subject in a way that is compatible with current graphics APIs and that you can implement in your work.

The key idea is to apply additional information to your images as your geometry is computed and displayed. In our API environment that primarily works with polygons, as the pixels of the polygon are computed, the color of each pixel is not calculated from a simple lighting model but from a texture map, and this chapter will focus on how that texture mapping works. Most of the time we think of the texture as an image, so that when you render your objects they will be colored with the color values in the texture map. This approach is called *texture mapping* and allows you to use many tools to create visually-interesting things to be displayed on your objects. There are also ways to use texture maps to determine the luminance, intensity, or alpha values of your objects, adding significantly to the breadth of effects you can achieve.

Texture mapping is not, however, the only approach that can be used in texturing. It is also possible to compute the texture data for each pixel of an object by procedural processes. This approach is more complex than we want to include in a first graphics programming course, but we will illustrate some procedural methods as we create texture maps for some of our examples. This will allow us to approximate procedural texturing and to give you an idea of the value of this kind of approach, and you can go on to look at these techniques yourself in more detail.

Texture maps are arrays of colors that represent information (for example, an image) that you want to display on an object in your scene. These maps can be 1D, 2D, or 3D arrays, though we will focus on 1D and 2D arrays here. Texture mapping is the process of identifying points on objects you define with points in a texture map to achieve images that can include strong visual interest while using simpler geometry.

The key point to be mastered is that you are dealing with two different spaces in texture mapping. The first is your modeling space, the space in which you define your objects to be displayed. The second is a space in which you create information that will be mapped to your objects. This information is in discrete pieces that correspond to cells in the texture array, often called *texels.* In order to use texture maps effectively, you must carefully consider how these two spaces will be linked when your image is created — you must include this relationship as part of your design for the final image.

There are many ways to create your texture maps. For 1D textures you may define a linear color function through various associations of color along a line segment. For 2D textures you may use scanned images, digital photos, digital paintings, or screen captures to create the images, and you may use image tools such as Photoshop to manipulate the images to achieve precisely the effects you want. Your graphics API may have tools that allow you to capture the contents of your frame buffer in an array where it can be read to a file or used as a texture map. This 2D texture world is the richest texture environment we will meet in these notes, and is the most common texture

context for most graphics work. For 3D textures you may again define your texture by associating colors with points in space, but this is more difficult because there are few tools for scanning or painting 3D objects. However, you may compute the values of a 3D texture from a 3D model, and various kinds of medical scanning will produce 3D data, so 3D textures have many appropriate applications.

Most graphics APIs are quite flexible in accepting texture maps in many different formats. You can use one to four components for the texture map colors, and you can select RGB, RGBA, or any single one of these four components of color for the texture map. Many of these look like they have very specialized uses for unique effects, but an excellent general approach is to use straightforward 24-bit RGB color (8 bits per color per pixel) without any compression or special file formats — what Photoshop calls "raw RGB."

Finally, texture mapping is much richer than simply applying colors to an object. Depending on the capabilities of your graphics API, you may be able to apply texture to a number of different kinds of properties, such as transparency or luminance. In the most sophisticated kinds of graphics, texturing is applied to such issues as the directions of normals to achieve special lighting effects such as bump mapping and anisotropic reflection.

*Definitions*

In defining texture maps below, we describe them as one-, two-, or three-dimensional arrays of colors. These are the correct definitions technically, but we usually conceptualize them a little more intuitively as one-, two-, or three-dimensional spaces that contain colors. When texture maps are applied, the vertices in the texture map may not correspond to the pixels you are filling in for the polygon, so the system must find a way to choose colors from the texture arrays. There are ways to filter the colors from the texture array to compute the value for the pixel, ranging from choosing the nearest point in the texture array to averaging values of the colors for the pixel. However, this is usually not a problem when one first starts using textures, so we note this for future reference and will discuss how to do it for the OpenGL API later in this chapter.

1D texture maps: A 1D texture map is a one-dimensional array of colors that can be applied along any direction of an object — essentially as though it were extended to a 2D texture map by being replicated into a 2D array. It thus allows you to apply textures that emphasize the direction you choose, and in our example below it allows us to apply a texture that varies only according to the distance of an object from the plane containing the eye point.

2D texture maps: A 2D texture map is a two-dimensional array of colors that can be applied to any 2D surface in a scene. This is probably the most natural and easy-to-understand kind of texture mapping, because it models the concept of "pasting" an image onto a surface. By associating points on a polygon with points in the texture space, which are actually coordinates in the texture array, we allow the system to compute the association of any point on the polygon with a point in the texture space so the polygon point may be colored appropriately. When the polygon is drawn, then, the color from the texture space is used as directed in the texture map definition, as noted below.

3D texture maps: A 3D texture map is a three-dimensional array of colors. 3D textures are not supported in OpenGL 1.0 or 1.1, but were added in version 1.2. Because we assume that you will not yet have this advanced version of OpenGL, this is not covered here, but it is described in the OpenGL Reference Manual and the OpenGL Programmers Guide for version 1.2. A useful visual examination of 3D textures is found in Rosalee Wolfe's book noted in the references. The 3D texture capability could be very useful in scientific work when the 3D texture is defined by an array of colors from data or theoretical work and the user can examine surfaces in 3-space, colored by the texture, to understand the information in the space.

The relation between the color of the object and the color of the texture map

In a texture-mapping situation, we have an object and a texture. The object may be assumed to have color properties, and the texture also has color properties. Defining the color or colors of the texture-mapped object involves considering the colors both the object and the texture map.

Perhaps the most common concept of texture mapping involves replacing any color on the original object by the color of the texture map. This is certainly one of the options that a graphics API will give you. But there are other options as well for many APIs. If the texture map has an alpha channel, you can blend the texture map onto the object, using the kind of color blending we discuss in the color chapter. You may also be able to apply other operations to the combination of object and texture color to achieve other effects. So don't assume simply that the only way to use texture maps is to replace the color of the object by the color of the texture; the options are much more interesting than merely that.

Texture mapping and billboards

In the chapter on high-performance graphics techniques we introduce the concept of a *billboard* — a two-dimensional polygon in three-space that has an image texture-mapped onto it so that the image on the polygon seems to be a three-dimensional object in the scene. This is a straightforward application of texture mapping but requires that the color of the polygon come entirely from the texture map and that some portions of the texture map have a zero alpha value so they will seem transparent when the polygon is displayed.

*Creating texture maps*

Any texture you use must be created somehow before it is loaded into the texture array. This may be done by using an image as your texture or by creating your texture through a computational process. In this section we will consider these two options and will outline how you can create a texture map through each.

Getting an image as a texture map

Using images as texture maps is very popular, especially when you want to give a naturalistic feel to a graphical object. Thus textures of sand, concrete, brick, grass, and ivy — to name only a few possible naturalistic textures — are often based on scanned or digital photographs of these materials. Other kinds of textures, such as flames or smoke, can be created with a digital paint system and used in your work. All the image-based textures are handled in the same way: the image is created and saved in a file with an appropriate format, and the file is read by the graphics program into a texture array to be used by the API's texture process. And we must note that such textures are 2D textures.

The main problem with using images is that there is an enormous number of graphics file formats. Entire books are devoted to cataloging these formats, and some formats include compression techniques that require a great deal of computation when you re-create the image from the file. We suggest that you avoid file formats such as JPEG, GIF, PICT, or even BMP, and use only formats that store a simple sequence of RGB values. Using compressed images directly requires you to use a tool called an RIP — a raster image processor — to get the pixels from your image, and this would be a complex tool to write yourself. If you want to use an image that you have in a compressed file forma, probably the simplest approach is to open your image in a highly-capable image manipulation tool such as Photoshop, which can perform the image re-creation from most formats, and then re-save it in a simplified form such as raw RGB. Graphics APIs are likely to have restrictions on the dimensions of texture maps (OpenGL requires all dimensions to be a

power of 2, for example) so even if the format is so low-level that it does not include the dimensions, they can be recalled easily. We suggest that you include the dimension as part of the file name, such as `ivy.128x64.rgb` so that the size will not be something that must be recorded. The process of using an image file as a texture map is described in the second code example in this chapter.

Generating a synthetic texture map

Because a texture map is simply an array of color, luminance, intensity, or alpha values, it is possible to generate the values of the array by applying a computational process instead of reading a file. Generating a texture computationally is a very powerful technique that can be very simple, or it may be relatively complex. Here we'll describe a few techniques that you might find helpful as a starting point in creating your own computed textures.

One of the simplest textures is the checkerboard tablecloth. Here, we will assume a 64x64 texture array and can define the color of an element `tex[i][j]` as red if `((i%4)+(j%4))%2` has value zero and white if the value is one. This will put a 4x4 red square at the top left of the texture and will alternate white and red 4x4 squares from there, thus creating a traditional checkerboard pattern.

A particularly useful instance of a computed texture involves the use of a *noise function*. A noise function is a single-valued function of one, two, or three variables that has no statistical correlation to any rotation (that is, does not seem to vary systematically in any direction) or translation (does not seem to vary systematically across the domain) and that has a relatively limited amount of change in the value across a limited change in the domain. When one does procedural texturing, one can calculate the value of a noise function for each pixel in a textured object and use that value to determine the pixel color in several different ways There are a number of ways to create such functions, and we will not begin to explore them all, but we will take one approach to defining a noise function and use it to generate a couple of texture maps.

The usual technique to define a noise function is to use (pseudo)random numbers to determine values at a mesh of points in the domain of the function, and then use interpolation techniques to define the values of the function between these. The interpolation techniques give you a smoother function than you would have if you simply used random values, producing a more effective texture. We will focus on 2D texture maps here by defining functions that calculate values for a 2D domain array, but this is all readily extended to 1D or 3D noise. You may compute a single noise function and use it to generate a noise texture map so you can work with the same texture map to have a consistent environment as you develop your image, or you may calculate a new noise function and new texture map each time you execute your program and achieve a constantly-new effect. We suggest that you use the consistent texture map for development, however, so you may distinguish effects caused by the texture map from effects caused by other graphics issues.

The approach we will take to develop a noise function is the gradient interpolation discussed by Peachy in [Ebert et al] and is chosen to give us a simple interpolation process. This is also the kind of noise function used in the Renderman™ shader system, which you may want to investigate but which we do not cover in these notes. For each point of the 3D mesh, whose indices we will treat as x-, y-, and z-components of the noise function domain, we will compute a unit vector of three components that represents the gradient at that point. We will then assume a height of 0 at each grid point and use the gradients to define a smooth function to be our basic noise function. Other functions will be derived from that. A 3D noise is a real-valued function of three variables, and we can obtain 2D or even 1D noise functions by taking 2D or 1D subspaces of the 3D noise space. But we will retain the basic 3D approach because 3D textures are difficult to get through non-computational means. We will also re-phrase the code in terms of multi-dimensional arrays to

make it easier to read, but you should note that writing in terms of arrays will create slower execution, which is a critical problem for real procedural textures. You are encouraged to read the discussions of efficiency in [Ebert] to understand these issues.

Generating the gradients for each grid point depends on generating random numbers between zero and one. We assume that you have available, or can easily create, such a function, which we will call `random()`, and that your local laboratory support will help you work with any issues such as initializing the random number environment. Note that the computation `z = 1.–2.*random()` will produce a random number between -1 and 1. Then the function below, to be executed in the initialization section of your program, sets values in the table that will be interpolated to produce the noise function:

```
float gradTab[TABSIZE][TABSIZE][TABSIZE][3];

void gradTabInit(void) {
   float z, r, theta;
   int i;

   for (i = 0; i < TABSIZE; i++) {
      z = 1. - 2.*random();
      r = sqrt((1. - z*z));
      theta = 2.0*PI*random();
      gradTab[i][0] = r*cos(theta);
      gradTab[i][1] = r*sin(theta);
      gradTab[i][2] = z;
   }
}
```

In addition to the table, we build an array of permutations of [0..255] so that we can interpolate the gradient data from distributed points, giving us the uncorrelated information we need for the noise function.

```
// the table is a permutation of 0..255; it could as easily be generated
// with a permutation function if one wished
static unsigned char perm[TABSIZE] = {
   225,155,210,108,175,199,221,144,203,116, 70,213, 69,158, 33,252,
     5, 82,173,133,222,139,174, 27,  9, 71, 90,246, 75,130, 91,191,
   169,138,  2,151,194,235, 81,  7, 25,113,228,159,205,253,134,142,
   248, 65,224,217, 22,121,229, 63, 89,103, 96,104,156, 17,201,129,
    36,  8,165,110,237,117,231, 56,132,211,152, 20,181,111,239,218,
   170,163, 51,172,157, 47, 80,212,176,250, 87, 49, 99,242,136,189,
   162,115, 44, 43,124, 94,150, 16,141,247, 32, 10,198,223,255, 72,
    53,131, 84, 57,220,197, 58, 50,208, 11,241, 28,  3,192, 62,202,
    18,215,153, 24, 76, 41, 15,179, 39, 46, 55,  6,128,167, 23,188,
   106, 34,187,140,164, 73,112,182,244,195,227, 13, 35, 77,196,185,
    26,200,226,119, 31,123,168,125,249, 68,183,230,177,135,160,180,
    12,  1,243,148,102,166, 38,238,251, 37,240,126, 64, 74,161, 40,
   184,149,171,178,101, 66, 29, 59,146, 61,254,107, 42, 86,154,  4,
   236,232,120, 21,233,209, 45, 98,193,114, 78, 19,206, 14,118,127,
    48, 79,147, 85, 30,207,219, 54, 88,234,190,122, 95, 67,143,109,
   137,214,145, 93, 92,100,245,  0,216,186, 60, 83,105, 97,204, 52
   };
```

Once the gradient and permutation tables have been computed, a smoothed linear interpolation of the nearest mesh points is computed and is returned as the value of the function. The indices in the gradient table are taken as values in the 2D domain, and this interpolation uses the gradient values

to determine the function's values between the integer points in the grid. Note that in this code we introduce an alternative to the standard C `floor()` function, an interpolation function that calculates a value that lies a proportion `p` between `x0` and `x1` with `0 <= p < =1`, a smoothstep function of a value between 0 and 1 that provides a smooth transition between those values rather than a simple linear transition, and a function that calculates an address in the gradient table based on values from the permutation table.

```
#define FLOOR(x) ((int)(x) - ((x) < 0 && (x) != (int)(x)))
#define INTERP(p,x0,x1) ((x0)+(p)*((x1)-(x0)))
#define SMOOTHSTEP(x) ((x)*(x)*(3. - 2.*(x)))
#define PERM(x) perm[(x)&(TABSIZE-1)]
#define ADDR(i, j, k) (PERM((i) + PERM((j) + PERM((k)))));

float noise(float x, float y, float z) {
   int ix, iy, iz;
   float fx0, fx1, fy0, fy1, fz0, fz1,
         wx, wy, wz, vx0, vx1, vy0, vy1, vz0, vz1;

   ix = FLOOR(x);
   fx0 = x - ix;
   fx1 = fx0 - 1.;
   wx = SMOOTHSTEP(fx0);

   iy = FLOOR(y);
   fy0 = y - iy;
   fy1 = fy0 - 1.;
   wy = SMOOTHSTEP(fy0);

   iz = FLOOR(z);
   fz0 = z - iz;
   fz1 = fz0 - 1.;
   wz = SMOOTHSTEP(fz0);

   vx0=gradTab[ADDR(ix,iy,iz)][0]*fx0+
         gradTab[ADDR(ix,iy,iz)][1]*fy0+
         gradTab[ADDR(ix,iy,iz)][2]*fz0;
   vx1=gradTab[ADDR(ix+1,iy,iz)][0]*fx1+
         gradTab[ADDR(ix+1,iy,iz)][1]*fy0+
         gradTab[ADDR(ix+1,iy,iz)][2]*fz0;
   vy0=INTERP(wx, vx0, vx1);
   vx0=gradTab[ADDR(ix,iy+1,iz)][0]*fx0+
         gradTab[ADDR(ix,iy+1,iz)][1]*fy1+
         gradTab[ADDR(ix,iy+1,iz)][2]*fz0;
   vx1=gradTab[ADDR(ix+1,iy+1,iz)][0]*fx1+
         gradTab[ADDR(ix+1,iy+1,iz)][1]*fy1+
         gradTab[ADDR(ix+1,iy+1,iz)][2]*fz0;
   vy1=INTERP(wx, vx0, vx1);
   vz0=INTERP(wy, vy0, vy1);

   vx0=gradTab[ADDR(ix,iy,iz+1)][0]*fx0+
         gradTab[ADDR(ix,iy,iz+1)][1]*fy0+
         gradTab[ADDR(ix,iy,iz+1)][2]*fz1;
   vx1=gradTab[ADDR(ix+1,iy,iz+1)][0]*fx1+
         gradTab[ADDR(ix+1,iy,iz+1)][1]*fy0+
         gradTab[ADDR(ix+1,iy,iz+1)][2]*fz1;
   vy0=INTERP(wx, vx0, vx1);
   vx0=gradTab[ADDR(ix,iy+1,iz+1)][0]*fx0+
         gradTab[ADDR(ix,iy+1,iz+1)][1]*fy1+
```

```
            gradTab[ADDR(ix,iy+1,iz+1)][2]*fz1;
        vx1=gradTab[ADDR(ix+1,iy+1,iz+1)][0]*fx1+
            gradTab[ADDR(ix+1,iy+1,iz+1)][1]*fy1+
            gradTab[ADDR(ix+1,iy+1,iz+1)][2]*fz1;
        vy1=INTERP(wx, vx0, vx1);
        vz1=INTERP(wy, vy0, vy1);

        return INTERP(wz, vz0, vz1);
    }
```

The noise function will probably be defined initially on a relatively small domain with a fairly coarse set of integer vertices, and then the values in the full domain will be computed with the smoothing operations on the coarser definition. For example, if we define the original noise function $N_0$ on an 8x8x8 grid we could extend it to a function $N_1$ for a larger grid (say, 64x64x64) by defining $N_1(x,y,z) = N_0(x/8,y/8,z/8)$. We can thus calculate the values of the function for whatever size texture map we want, and either save those values in an array to be used immediately as a texture, or write those values to a file for future use. Note that this new function has a frequency 8 times that of the original, so given an original function we can easily create new functions with a larger or smaller frequency at will. And if we know that we will only want a 1D or 2D function, we can use a limited grid in the unneeded dimensions.

The noise function we have discussed so far is based on interpolating values that are set at fixed intervals, so it has little variation between these fixed points. A texture that has variations of several different sizes has more irregularity and thus more visual interest, and we can create such a texture from the original noise function in several ways. Probably the simplest is to combine noise functions with several different frequencies as noted above. However, in order to keep the high-frequency noise from overwhelming the original noise, the amplitude of the noise is decreased as the frequency is increased. If we consider the function $N_2 = A*N_0(Fx,Fy,Fz)$, we see a function whose amplitude is increased by a factor of $A$ and whose frequency is increased by a factor of $F$ in each direction. If we create a sequence of such functions with halved amplitude and doubled frequency and add them,

$$N(x,y,z) = \Sigma_k \, N_0(2^k x, 2^k y, 2^k z)/2^k$$

we get a function that has what is called 1/f noise that produces textures that behave like many natural phenomena. In fact, it is not necessary to compute many terms of the sum in order to get good textures; you can get good results of you calculate only as many terms as you need to get to the final resolution of your texture. Another approach, which is very similar but which produces a fairly different result, is the turbulence function obtained when the absolute value of the noise function is used, introducing some discontinuities to the function and table.

$$T(x,y,z) = \Sigma_k \, abs(N_0(2^k x, 2^k y, 2^k z))/2^k$$

And again, note that we are computing 3D versions of the 1/f noise and turbulence functions, and that you can take 2D or 1D subspaces of the 3D space to create lower-dimension versions. The images in Figure 11.1 show the nature of the noise functions discussed here.

So far, we have created noise and turbulence functions that produce only one value. This is fine for creating a grayscale texture and it can be used to produce colors by applying the single value in any kind of color ramp, but it is not enough to provide a full-color texture. For this we can simply treat three noise functions as the components of a 3D noise function that returns three values, and use these the RGB components of the texture. Depending on your application, either a 1D noise function with a color ramp or a 3D noise function might work best.

Figure 11.1a: a 2D graph of the original noise function (left) and of the 1/f noise derived from it (right)

Figure 11.1b: the surface representation of the original noise function

*Antialiasing in texturing*

When you apply a texture map to a polygon, you identify the vertices in the polygon with values in texture space. These values may or may not be integers (that is, actual indices in the texture map) but the interpolation process we discussed will assign a value in texture space to each pixel in the polygon. The pixel may represent only part of a texel (texture cell) if the difference between the texture-space values for adjacent pixels is less than one, or it may represent many texels if the difference between the texture space values for adjacent pixels is greater than one. This offers two kinds of aliasing — the magnification of texels if the texture is coarse relative to the object being texture mapped, or the selection of color from widely separated texels if the texture is very fine relative to the object.

Because textures may involve aliasing, it can be useful to have antialiasing techniques with texturing. In the OpenGL API, the only antialiasing tool available is the linear filtering that we discuss below, but other APIs may have other tools, and certainly sophisticated, custom-built or research graphics systems can use a full set of antialiasing techniques. This needs to be considered when considering the nature of your application and choosing your API. See [Ebert] for more details.

*Texture mapping in OpenGL*

There are many details to master before you can count yourself fully skilled at using textures in your images. The full details must be left to the manuals for OpenGL or another API, but here we will discuss many of them, certainly enough to give you a good range of skills in the subject. The details we will discuss are the texture environment, texture parameters, building a texture array, defining a texture map, and generating textures. We will have examples of many of these details to help you see how they work.

Capturing a texture from the screen

A useful approach to textures is to create an image and save the color buffer (the frame buffer) as an array that can be used as a texture map. This can allow you to create a number of different kinds of images for texture maps. This operation is supported by many graphics APIs. For example, in OpenGL, the `glReadBuffer(mode)` and `glReadPixels(...)` functions will define the buffer to be read from and then will read the values of the elements in that buffer into a target array. That array may then be written to a file for later use, or may be used immediately in the program as the texture array. We will not go into more detail here but refer the student to the manuals for the use of these functions.

Texture environment

The a graphics API, you must define your texture environment to specify how texture values are to be used when the texture is applied to a polygon. In OpenGL, the appropriate function call is

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, *)
```

The meaning of the texture is determined by the value of the last parameter. The options are `GL_BLEND`, `GL_DECAL`, `GL_MODULATE`, or `GL_REPLACE`.

If the texture represents *RGB color*, the behavior of the texture when it is applied is defined as follows. In this and the other behavior descriptions, we use C, A, I, and L for color, alpha, intensity, and luminance respectively, and subscripts f and t for the fragment and texture values.

`GL_BLEND`: the color of the pixel is $C_f(1-C_t)$.

`GL_DECAL`: the color of the pixel is $C_t$, simply replacing the color by the texture color.

`GL_MODULATE`: the color of the pixel is $C_f*C_t$,replacing the color by the subtractive computation for color.

`GL_REPLACE`: same as GL_DECAL for color.

If the texture represents *RGBA color*, then the behavior of the texture is defined as:

`GL_BLEND`: the color of the pixel is $C_f(1-C_t)$, and the alpha channel in the pixel is $A_f*A_t$.

`GL_DECAL`: the color of the pixel is $(1-A_t)C_f+A_tC_t$, and the alpha channel in the pixel is $A_f$.

`GL_MODULATE`: the color of the pixel is $C_f*C_t$, as above, and the alpha channel in the pixel is $A_f*A_t$.

`GL_REPLACE`: the color of the pixel is $C_t$ and the alpha channel in the pixel is $A_t$.

If the texture represents the *alpha channel*, the behavior of the texture is defined as:

`GL_BLEND`: the color of the pixel is $C_f$, and the alpha channel in the pixel is $A_f$.

`GL_DECAL`: the operation is undefined

`GL_MODULATE`: the color of the pixel is $C_f$, and the alpha channel in the pixel is $A_f*A_t$.

`GL_REPLACE`: the color of the pixel is $C_f$ and the alpha channel in the pixel is $A_t$.

If the texture represents *luminance*, the behavior of the texture is defined as:

`GL_BLEND`: the color of the pixel is $C_f(1-L_t)$, and the alpha channel in the pixel is $A_f$.

`GL_DECAL`: the operation is undefined.

`GL_MODULATE`: the color of the pixel is $C_f*L_t$, and the alpha channel in the pixel is $A_f$.

`GL_REPLACE`: the color of the pixel is $L_t$ and the alpha channel in the pixel is $A_f$.

If the texture represents *intensity*, the behavior of the texture is defined as:

`GL_BLEND`: the color of the pixel is $C_f(1-I_t)$, and the alpha channel in the pixel is $A_f(1-I_t)$.

`GL_DECAL`: the operation is undefined.

`GL_MODULATE`: the color of the pixel is $C_f*I_t$, and the alpha channel in the pixel is $A_f*I_t$.

`GL_REPLACE`: the color of the pixel is $I_t$ and the alpha channel in the pixel is $I_t$.

Texture parameters

The texture parameters define how the texture will be presented on a polygon in your scene. In OpenGL, the parameters you will want to understand include texture wrap and texture filtering. Texture wrap behavior, defined by the `GL_TEXTURE_WRAP_*` parameter, specifies the system behavior when you define texture coordinates outside the [0,1] range in any of the texture dimensions. The two options you have available are repeating or clamping the texture. Repeating the texture is accomplished by taking only the decimal part of any texture coordinate, so after you go beyond 1 you start over at 0. This repeats the texture across the polygon to fill out the texture space you have defined. Clamping the texture involves taking any texture coordinate outside [0,1] and translating it to the nearer of 0 or 1. This continues the color of the texture border outside the region where the texture coordinates are within [0,1]. This uses the `glTexParameter*(...)` function to repeat, or clamp, the texture respectively as follows:

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
```

Mixing the parameters with horizontal repetition and vertical clamping produces an image like that of Figure 11.2.

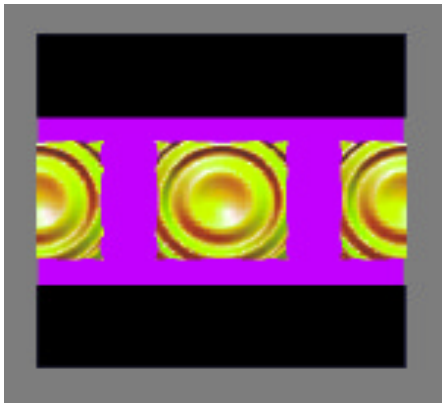


Figure 11.2: a polygon face with a texture that is wrapped in one direction and clamped in the other

Another important texture parameter controls the filtering for pixels to deal with aliasing issues. In OpenGL, this is called the minification (if there are many texture points that correspond to one pixel in the image) or magnification (if there are many pixels that correspond to one point in the texture) filter, and it controls the way an individual pixel is colored based on the texture map. For any pixel in your scene, the texture coordinate for the pixel is computed through an interpolation

across a polygon, and rarely corresponds exactly to an index in the texture array, so the system must create the color for the pixel by a computation in the texture space. You control this in OpenGL with the texture parameter `GL_TEXTURE_*_FILTER` that you set in the `glTexParameter*(...)` function. The filter you use depends on whether a pixel in your image maps to a space larger or smaller than one texture element. If a pixel is smaller than a texture element, then `GL_TEXTURE_MIN_FILTER` is used; if a pixel is larger than a texture element, then `GL_TEXTURE_MAG_FILTER` is used. An example of the usage is:

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
```

The symbolic values for these filters are `GL_NEAREST` or `GL_LINEAR`. If you choose the value `GL_NEAREST` for the filter, then the system chooses the single point in the texture space nearest the computed texture coordinate; if you choose `GL_LINEAR` then the system averages the four nearest points to the computed texture coordinate with weights depending on the distance to each point. The former is a faster approach, but has problems with aliasing; the latter is slower but produces a much smoother image. This difference is illustrated in Figure 11.3 in an extreme close-up, and it is easy to see that the `GL_NEAREST` filter gives a much coarser image than the `GL_LINEAR` filter. Your choice will depend on the relative importance of speed and image quality in your work.

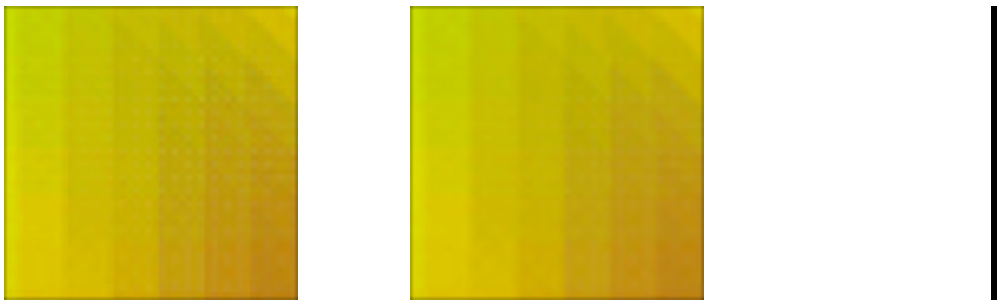


Figure 11.3: a texture zoomed in with `GL_NEAREST` (left) and `GL_LINEAR` (right) filter

Getting and defining a texture map

This set of definitions is managed by the `glTexImage*D(...)` functions. These are a complex set of functions with a number of different parameters. The functions cover 1D, 2D, and 3D textures (the dimension is the asterix in the function name) and have the same structure for the parameters.

Before you can apply the `glTexImage*D(...)` function, however, you must define and fill an array that holds your texture data. This array of unsigned integers (`GLuint`) will have the same dimension as your texture. The data in the array can be organized in many ways, as we will see when we talk about the internal format of the texture data. You may read the values of the array from a file or you may generate the values through your own programming. The examples in this chapter illustrate both options.

The parameters of the `glTexImage*D(...)` function are, in order,

- the *target*, usually `GL_TEXTURE_*D`, where * is 1, 2, or 3. Proxy textures are also possible, but are beyond the range of topics we will cover here. This target will be used in a number of places in defining texture maps.
- the *level*, an integer representing level-of-detail number. This supports multiple-level MIP-mapping.

- the *internal format* of the texture map, one of the places where an API such as OpenGL must support a large number of options to meet the needs of a wide community. For OpenGL, this internal format is a symbolic constant and can vary quite widely, but we will list only a set we believe will be most useful to the student. Most of the other options deal with other organizations that involve a different number of bits per pixel of the component. Here we deal only with formats that have eight bits per component, and we leave the others (and information on them in manuals) to applications that need specialized formats.
  - `GL_ALPHA8`
  - `GL_LUMINANCE8`
  - `GL_INTENSITY8`
  - `GL_RGB8`
  - `GL_RGBA8`
- the *dimensions* of the texture map, of type `GLsizei`, so the number of parameters here is the dimension of the texture map. If you have a 1D texture map, this parameter is the *width*; if you have a 2D texture map, the two parameters are the width and *height*; if you have a 3D texture map, the three parameters are width, height, and *depth*. Each of these must have a value of `2N+2(border)` where the value of border is either 0 or 1 as specified in the next parameter.
- the *border*, an integer that is either 0 (if no border is present) or 1 (if there is a border).
- the *format*, a symbolic constant that defines what the data type of the pixel data in the texture array is. This includes the following, as well as some other types that are more exotic:
  - `GL_ALPHA`
  - `GL_RGB`
  - `GL_RGBA`
  - `GL_LUMINANCE`

  The format indicates how the texture is to be used in creating the image. We discussed the effects of the texture modes and the texture format in the discussion of image modes above.
- the *type* of the pixel data, a symbolic constant that indicates the data type stored in the texture array per pixel. This is usually pretty simple, as shown in the examples below which use only `GL_FLOAT` and `GL_UNSIGNED_BYTE` types.
- the pixels, an address of the pixel data (texture array) in memory.

You will be creating your textures from some set of sources and probably using the same kind of tools. When you find a particular approach that works for you, you'll most likely settle on that particular approach to textures. The number of options in structuring your texture is phenomenal, as you can tell from the number of options in some of the parameters above, but you should not be daunted by this broad set of possibilities and should focus on finding an approach you can use.

Texture coordinate control

As your texture is applied to a polygon, you may specify how the texture coordinates correspond to the vertices with the `glTexture*(...)` function, as we have generally assumed above, or you may direct the OpenGL system to assign the texture coordinates for you. This is done with the `glTexGen*(...)` function, which allows you to specify the details of the texture generation operation.

The `glTexGen*(...)` function takes three parameters. The first is the texture coordinate being defined, which is one of `GL_S`, `GL_T`, `GL_R`, or `GL_Q` with S, T, R, and Q being the first, second, third, and homogeneous coordinates of the texture. The second parameter is one of three symbolic constants: `GL_TEXTURE_GEN_MODE`, `GL_OBJECT_PLANE`, or `GL_EYE_PLANE`. If the second parameter is `GL_TEXTURE_GEN_MODE`, the third parameter is a single symbolic constant with value `GL_OBJECT_LINEAR`, `GL_EYE_LINEAR`, or `GL_SPHERE_MAP`. If the second parameter is `GL_OBJECT_PLANE`, the third parameter is a vector of four values that defines the plane from which an object-linear texture is defined; if the second parameter is `GL_EYE_PLANE`, the third parameter is a vector of four values that defines the plane that contains

the eye point. In both these cases, the object-linear or eye-linear value is computed based on the coefficients. If the second parameter is GL_TEXTURE_GEN_MODE and the third parameter is GL_SPHERE_MAP, the texture is generated based on an approximation of the reflection vector from the surface to the texture map.

Applications of this texture generation include the Chromadepth™ texture, which is a 1D eye-linear texture generated with parameters that define the starting and ending points of the texture. Another example is automatic contour generation, where you use a GL_OBJECT_LINEAR mode and the GL_OBJECT_PLANE operation that defines the base plane from which contours are to be generated. Because contours are typically generated from a sea-level plane (one of the coordinates is 0), it is easy to define the coefficients for the object plane base.

Texture mapping and GLU quadrics

As we noted in the chapter on modeling, the GLU quadric objects have built-in texture mapping capabilities, and this is one of the features that makes them very attractive to use for modeling. To use these, we must carry out three tasks: load the texture to the system and bind it to a name, define the quadric to have normals and a texture, and then bind the texture to the object geometry as the object is drawn. The short code fragments for these three tasks are given below, with a generic function readTextureFile(...) specified that you will probably need to write for yourself, and with a generic GLU function to identify the quadric to be drawn.

```
readTextureFile(...);
glBindTexture(GL_TEXTURE_2D, texture[i]);
glTexImage2D(GL_TEXTURE_2D,...);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

myQuadric = gluNewQuadric();
gluQuadricNormals(myQuadric, GL_SMOOTH);
gluQuadricTexture(myQuadric, GL_TRUE);
gluQuadricDrawStyle(myQuadric, GLU_FILL);

glPushMatrix();
  // modeling transformations as needed
  glBindTexture(GL_TEXTURE_2D, texture[i]);
  gluXXX(myQuadric, ...);
glPopMatrix();
```

*Some examples*

Textures can be applied in several different ways with the function

```
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, mode )
```

One way uses a decal technique, with mode GL_DECAL, in which the content of the texture is applied as an opaque image on the surface of the polygon, showing nothing but the texture map. Another way uses a modulation technique, with mode GL_MODULATE, in which the content of the texture is displayed on the surface as though it were colored plastic. This mode allows you to show the shading of a lighted surface by defining a white surface and letting the shading show through the modulated texture. There is also a mode GL_BLEND that blends the color of the object with the color of the texture map based on the alpha values, just as other color blending is done. In the examples below, the Chromadepth image is created with a 1D modulated texture so that the underlying surface shading is displayed, while the mapped-cube image is created with a 2D decal texture so that the face of the cube is precisely the texture map. You may use several different textures with one image, so that (for example) you could take a purely geometric white terrain

model, apply a 2D texture map of an aerial photograph of the terrain with `GL_MODULATE` mode to get a realistic image of the terrain, and then apply a 1D texture map in `GL_BLEND` mode that is mostly transparent but has colors at specific levels and that is oriented to the vertical in the 3D image in order to get elevation lines on the terrain. Your only limitation is your imagination — and the time to develop all the techniques.

The Chromadepth™ process: using 1D texture maps to create the illusion of depth. If you apply a lighting model with white light to a white object, you get a pure expression of shading on the object. If you then apply a 1D texture by attaching a point near the eye to the red end of the ramp (see the code below) and a point far from the eye to the blue end of the ramp, you get a result like that shown in Figure 11.4 below.

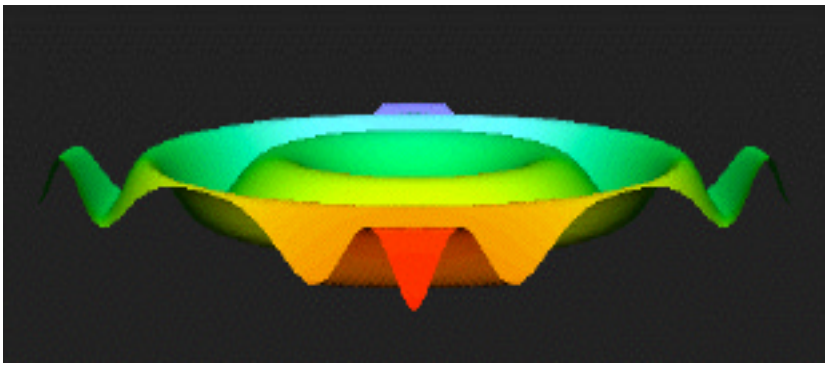


Figure 11.4: a Chromadepth-colored image of a mathematical surface

This creates a very convincing 3D image when it is viewed through Chromadepth™ glasses, because these glasses have a diffraction grating in one lens and clear plastic in the other. The diffraction grating bends red light more than blue light, so the angle between red objects as seen by both eyes is larger than the angle between blue objects. Our visual system interprets objects having larger angles between them as closer than objects having smaller angles, so with these glasses, red objects are interpreted as being closer than blue objects.

Using 2D texture maps to add interest to a surface: often we want to create relatively simple objects but have them look complex, particularly when we are trying to create models that mimic things in the real world. We can accomplish this by mapping images (for example, images of the real world) onto our simpler objects. In the very simple example shown in Figure 11.5, a screen capture and some simple Photoshop work created the left-hand image, and this image was used as

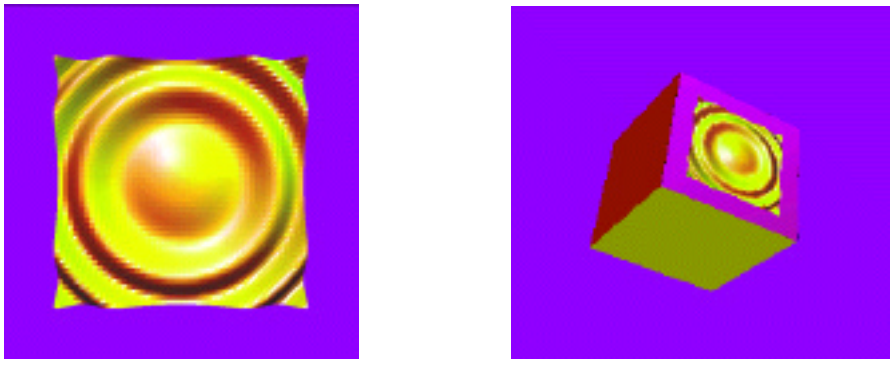


Figure 11.5: a texture map (left) and a 3D cube with the texture map placed on one face (right)

the texture map on one face of the cube in the right-hand image. (The boundary around the function surface is part of the texture.) This texture could also have been created by saving the frame buffer into a file in the program that created the mathematical surface. This created a cube that has more visual content than its geometry would suggest, and it was extremely simple to connect the square image with the square face of the cube.

Environment maps

Environment maps allow us to create the illusion that an object reflects images from a texture that we define. This can provide some very interesting effects, because realistic reflections of real-world objects is one of the visual realism clues we would expect. With environment maps, we can use photographs or synthetic images as the things we want to reflect, and we can adapt the parameters of the texture map to give us realistic effects. One of the easy effects to get is the reflection of things in a chrome-like surface. In Figure 11.6, we see an example of this as a photograph of Hong Kong that has been modified with a very wide-angle lens filter is used as a texture map on a surface. The lens effect makes the environment map much more convincing because the environment map uses the surface normals at a point to identify the texture points for the final image.

Figure 11.6: the original texture for an environment map (left) and the map on a surface (right)

*A word to the wise...*

Texture mapping is a much richer subject than these fairly simple examples have been able to show. You can use 1D textures to provide contour lines on a surface or to give you the kind of color encoding for a height value we discussed in the module on visual communication. You can use 2D textures in several sophisticated ways to give you the illusion of bumpy surfaces (use a texture on the luminance), to give the effect of looking through a varigated cloud (use a fractal texture on alpha) or of such a cloud on shadows (use the same kind of texture on luminance on a landscape image). This subject is a fruitful area for creative work.

There are several points that you must consider in order to avoid problems when you use texture mapping in your work. If you select your texture coordinates carelessly, you can create effects you might not expect because the geometry of your objects does not match the geometry of your texture map. One particular case of this is if you use a texture map that has a different aspect ratio than the space you are mapping it onto, which can change proportions in the texture that you might not have expected. More serious, perhaps, is trying to map an entire rectangular area into a quadrilateral that isn't rectangular, so that the texture is distorted nonlinearly. Imagine the effect if you were to try to map a brick texture into a non-convex polygon, for example. Another problem can arise if you texture-map two adjacent polygons with maps that do not align at the seam between

the polygons. Much like wallpaper that doesn't match at a corner, the effect can be disturbing and can ruin any attempt at creating realism in your image. Finally, if you use texture maps whose resolution is significantly different from the resolution of the polygon using the texture, you can run into problems of aliasing textures caused by selecting only portions of the texture map. We noted the use of magnification and minification filters earlier, and these allow you to address this issue.

In a different direction, the Chromadepth™ 1D texture-mapping process gives excellent 3D effects but does not allow the use of color as a way of encoding and communicating information. It should only be used when the shape alone carries the information that is important in an image, but it has proved to be particularly useful for geographic and engineering images, as well as molecular models.

*Code examples*

First example: Sample code to use texture mapping in the first example is shown below. The declaration set up the color ramp, define the integer texture name, and create the array of texture parameters.

```
float D1, D2;
float texParms[4];
static GLuint texName;
float ramp[256][3];
```

In the `init()` function we find the following function calls that define the texture map, the texture environment and parameters, and then enables the texture generation and application.

```
makeRamp();
glPixelStorei( GL_UNPACK_ALIGNMENT, 1 );
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
glTexParameterf( GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP );
glTexParameterf( GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameterf( GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
glTexImage1D( GL_TEXTURE_1D, 0, 3, 256, 0, GL_RGB, GL_FLOAT, ramp );
glEnable( GL_TEXTURE_GEN_S );
glEnable( GL_TEXTURE_1D );
```

The `makeRamp()` function is defined to create the global array `ramp[]` that holds the data of the texture map. This process works with the HSV color model in which hues are defined through angles (in degrees) around the circle which has saturation and value each equal to 1.0. The use of the number 240 in the function comes from the fact that in the HSV model, the color red is at 0 degrees and blue is at 240 degrees, with green between at 120 degrees. Thus an interpolation of fully-saturated colors between red and blue will use the angles between 0 and 240 degrees. The RGB values are calculated by a function `hsv2rgb(...)` that is a straightforward implementation of standard textbook color-model conversion processes. The Foley et al. textbook in the references is an excellent resource on color models (see Chapter 13).

```
void makeRamp(void)
{
   int i;
   float h, s, v, r, g, b;

// Make color ramp for 1D texture: starts at 0, ends at 240, 256 steps
   for (i=0; i<256; i++) {
      h = (float)i*240.0/255.0;
      s = 1.0; v = 1.0;
```

```
      hsv2rgb( h, s, v, &r, &g, &b );
      ramp[i][0] = r; ramp[i][1] = g; ramp[i][2] = b;
   }
}
```

Finally, in the `display()` function we find the code below, where `ep` is the eye point parameter used in the `gluLookAt(...)` function. This controls the generation of texture coordinates, and binds the texture to the integer name `texName`. Note that the values in the `texParms[]` array, which define where the 1D texture is applied, are defined based on the eye point, so that the image will be shaded red (in front) to blue (in back) in the space whose distance from the eye is between D1 and D2.

```
glTexGeni( GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR );
D1 = ep + 1.0; D2 = ep + 10.0;
texParms[0] = texParms[1] = 0.0;
texParms[2] = -1.0/(D2-D1);
texParms[3] = -D1/(D2-D1);
glTexGenfv( GL_S, GL_EYE_PLANE, texParms);
glBindTexture(GL_TEXTURE_1D, texName);
```

Second example: Sample code to use texture mapping in the second example is shown in several pieces below. To begin, in the data declarations we find the declarations that establish the internal texture map (`texImage`) and the set of texture names that can be used for textures (`texName`).

```
#define TEX_WIDTH 512
#define TEX_HEIGHT 512
static GLubyte texImage[TEX_WIDTH][TEX_HEIGHT][3];
static GLuint texName[1]; // parameter is the number of textures used
```

In the init function we find the `glEnable` function that allows the use of 2D textures.

```
glEnable(GL_TEXTURE_2D);  // allow 2D texture maps
```

You will need to create the texture map, either through programming or by reading the texture from a file. In this example, the texture is read from a file named `myTexture.rgb` that was simply captured and translated into a raw RGB file, and the function that reads the texture file and creates the internal texture map, called from the init function, is

```
void setTexture(void)
{
   FILE * fd;
   GLubyte ch;
   int i,j,k;

   fd = fopen("myTexture.rgb", "r");
   for (i=0; i<TEX_WIDTH; i++)     // for each row
   {
      for (j=0; j<TEX_HEIGHT; j++) // for each column
      {
         for (k=0; k<3; k++) // read RGB components of the pixel
         {
            fread(&ch, 1, 1, fd);
            texImage[i][j][k] = (GLubyte) ch;
         }
      }
   }
   fclose(fd);
```

```
}
```

Finally, in the function that actually draws the cube, called from the `display()` function, we first find code that links the texture map we read in with the texture number and defines the various parameters of the texture that will be needed to create a correct display. We then find code that draws the face of the cube, and see the use of texture coordinates along with vertex coordinates. The vertex coordinates are defined in an array `vertices[]` that need not concern us here.

```
glGenTextures(1, texName);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB8, TEX_WIDTH, TEX_HEIGHT,
             0, GL_RGB, GL_UNSIGNED_BYTE, texImage);
glBindTexture(GL_TEXTURE_2D, texName[0]);
glBegin(GL_QUADS);
   glNormal3fv(normals[1]);
   glTexCoord2f(0.0, 0.0); glVertex3fv(vertices[0]);
   glTexCoord2f(0.0, 1.0); glVertex3fv(vertices[1]);
   glTexCoord2f(1.0, 1.0); glVertex3fv(vertices[3]);
   glTexCoord2f(1.0, 0.0); glVertex3fv(vertices[2]);
glEnd();
glDeleteTextures(1,texName);
```

Third example: The third example also uses a 2D texture map, modified in Photoshop to have a fish-eye distortion to mimic the behavior of a very wide-angle lens. The primary key to setting up an environment map is in the texture parameter function, where we also include two uses of the `glHint(...)` function to show that you can define really nice perspective calculations and point smoothing — with a computation cost, of course. But the images in Figure 11.5 suggest that it might be worth the cost sometimes.

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST);
glHint(GL_POINT_SMOOTH_HINT,GL_NICEST);
...
// the two lines below generate an environment map in both the S and T
// texture coordinates
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
```

*References*


Chromadepth information is available from
Chromatek Inc
1246 Old Alpharetta Road
Alpharetta, GA 30005
888-669-8233
http://www.chromatek.com/

Chromadepth glasses may be ordered from
American Paper Optics
3080 Bartlett Corporate Drive
Bartlett, TN 38133
800-767-8427, 901-381-1515
fax 901-381-1517

Ebert, David et al., *Texturing and Modeling: a Procedural Approach*, second edition, Academic Press, 1998

Foley, James D. et al., *Computer Graphics Principles and Practice*, second edition, Addison-Wesley, 1990

Murray, James D. and William vanRyper, *Encyclopedia of Graphics File Formats*, second edition, O'Reilly & Associates, 1996

Perlin, Ken, "An Image Synthesizer," *Computer Graphics* 19(3), Proceedings of SIGGRAPH 85, July 1985, 287-296

Wolfe, R. J., *3D Graphics: A Visual Approach*, Oxford University Press, 2000

# Dynamics and Animation

*Prerequisites*

A knowledge of modeling, viewing, lighting, and on the roles of parameters in setting up modeling and viewing.

*Introduction*

This is an unusual chapter, because it includes few figures because the topic is motion, and we cannot readily capture motion in a written document. It would be possible to include movie files in a purely electronic document, of course, but these notes are intended for print. Perhaps future versions of this will include inline movies, or at least pre-compiled executables with animation, but for now you must work with the code segments that we provid and see the execution on your own systems.

Computer animation is a very large topic, and there are many books and courses on the subject. We cannot hope to cover the topic in any depth in a first course in computer graphics, and indeed the toolkits needed for a great deal of computer animation are major objects of study and skill development in themselves. Instead we will focus on relatively simple animations that illustrate something about the kind of models and images we have been creating in this course, with an emphasis on scientific areas.

Animation is thought of as presenting a sequence of frames, or individual images, rapidly enough to achieve the sense that the objects in the frames are moving smoothly. There are two kinds of animation — real-time animation, or animation in which each frame is presented by the program while it is running, and frame-at-a-time animation, or animation that is assembled by rendering the individual frames and assembling them into a viewable format (possibly through film or video in a separate production process). This chapter focuses more on frame-at-a-time animation with a goal of achieving real-time animation. The two share the problems of defining how models, lighting, and viewing change over time, but frame-at-a-time animation tends to focus on much more detailed modeling and much more sophisticated rendering while real-time animation tends to focus on simpler time-varying information in order to get refresh rates that are high enough to convey the variation that is desired. While real-time animation may not be as realistic as frame-at-a-time animation because simpler modeling and rendering are used or images may be provided at a slower rate, it can be very effective in conveying an idea and can be especially effective if the user can interact with the animation program as it is running.

As with everything else in this course, the real question is visual communication, and there are some special vocabularies and techniques in using animation for communication. This module does not try to cover this in any depth, but we suggest that you spend some time looking at successful animations and trying to discover for yourself what makes them succeed. To start, we suggest that you focus on clarity and simplicity, and work hard to create a focus on the particular ideas you want to communicate.

*Definitions*

Animation is the process of creating a sequence of images so that the viewer's eye will see them as occurring in a smooth motion sequence. The motion sequence can illustrate the relationship between things, can show processes for assembling objects, can allow you to design a sequence of ways to present information, or can allow a user to see a scene from a variety of viewpoints that you can design.

There are many ways to design an animation sequence, but a good place to start is to model your scene using parameters to control features of the model. When you use parameters — variables that you can manipulate in your program — to control positions of objects, positions or properties of lights, shapes or relationships or objects, colors, texture coordinates, or other key points in your model, you can change the values of the parameters with time to change the view you present your audience as the program runs. This allows you to emphasize special features of any of these facets of your model and to communicate those features to your audience.

In defining your modeling in terms of parameters, we need to recall that there are really only three parts to a scene as described by a scene graph. One is the geometry of the scene, where we could use parameters to define the geometry of the scene itself; one example of this could be a function surface, where the function includes a parameter that varies with time, such as $z = \cos(x^2+y^2+t)$. Another is the transformations in the scene, where we could use parameters to define the rotation, translation, or scaling of objects in the scene; one example of this could be moving an object in space by a translation with horizontal component t and vertical component $(2-t)^2$, which would give the object a parabolic path. A third is the appearance of an object in a scene, where a surface might have an alpha color component of (1-t) to change it from opaque at time 0 to transparent at time 1, allowing the user to see through the surface to whatever lies below it. These are straightforward kinds of applications of parametric modeling and should pose no problem in setting up a model.

One way to design an animation sequence is by explicitly controlling the change in the parameters above through your code. This is called *procedural* animation, and it works well for simple animations where you may have only a few parameters that define the sequence (although what defines "a few" may depend on your system and your goals in the sequence). Another way to design an animation sequence is through key frames, or particular images that you want to appear at particular times in the animation display, and this is called *keyframe* animation. Again, each key frame can be defined in terms of a set of parameters that control the display, but instead of controlling the parameters programmatically, the parameters are interpolated between the values at the key frames.

Probably the simplest approach to animation is to define your entire scene in terms of a single parameter, and to update that parameter each time you generate a new frame. You could think of the parameter as time and think of your animation in terms of a model changing with time. This is probably a natural approach when you are working with scientific problems, where time plays an active role in much of the modeling — think of how much of science deals with change per unit time. If you know how long it will take to generate your scene, you can even change a time parameter by that amount for each frame so that the viewer will see the frames at a rate that approximates the real-time behavior of the system you are modeling.

Another meaning for the parameter could be frame number, the sequence number of the particular image you are computing in the set of frames that will make up the animation sequence. If you are dealing with animation that you will record and playback at a known rate (usually 24 or 30 frames per second) then you can translate the frame number into a time parameter, but the difference in names for the parameter reflects a difference in thinking, because you will not be concerned about how long it takes to generate a frame, simply where the frame is in the sequence you are building.

A key concept in generating animations in real time is the *frame rate* — the rate at which you are able to generate new images in the animation sequence. As we noted above, the frame rate will probably be lower for highly-detailed generated frames than it would be for similar frames that were pre-computed and saved in digital or analog video, but there's one other difference: frame rates may not be constant for real-time generated images. This points out the challenge of doing your own animations and the need to be sure your animations carry the communication you need.

However, the frame rate can be controlled exactly, no matter how complex the images in the individual frames, if you create your own video "hardcopy" of your animation. See the hardcopy chapter for more details on this.

*Keyframe animation*

When you do a keyframe animation, you specify certain frames as key frames that the animation must produce and you calculate the rest of the frames so that they move smoothly from one key frame to another. The key frames are specified by frame numbers, so these are the parameter you use, as described above.

In cartoon-type animation, it is common for the key frames to be fully-developed drawings, and for the rest of the frames to be generated by a process called "tweening" — generating the frames between the keys. In that case, there are artists who generate the in-between frames by re-drawing the elements of the key frames as they would appear in the motion between key frames. However, we are creating images by programming, so we must start with models instead of drawings. Our key frames will have whatever parameters are needed to defing the images, then, and we will create our in-between frames by interpolating those parameters.

Our animation may be thought of as a collection of animation sequences (in the movies, these are thought of as *shots*), each of which uses the same basic parts, or components, of objects, transformations, lights, etc. For any sequence, then, we will have the same components and the same set of parameters for these components, and the parameters will vary as we go through the sequence. For the purposes of the animation, the components themselves are not important; we need to focus on the set of parameters and on the ways these parameters are changed. With a keyframe animation, the parameters are set when the key frames are defined, and are interpolated in some fashion in moving between the frames.

As an example of this, consider a model that is defined in a way that allows us to identify the parameters that control its behavior. Let us define ...

In order to discuss the ways we can change the parameters to achieve a smooth motion from one key frame to another, we need to introduce some notation. If we consider the set of parameters as a single vector $\boldsymbol{P} = \langle a, b, c, \ldots, n \rangle$, then we can consider the set of parameters at any given frame M as $\boldsymbol{P}_M = \langle a_M, b_M, c_M, \ldots, n_M \rangle$. In doing a segment of a keyframe animation sequence starting with frame K and going to frame L, then, we must interpolate

$\boldsymbol{P}_K = \langle a_K, b_K, c_K, \ldots, n_K \rangle$ and $\boldsymbol{P}_L = \langle a_L, b_L, c_L, \ldots, n_L \rangle$,

the values of the parameter vectors at these two frames.

A first approach to this interpolation would be to use a linear interpolation of the parameter values. So if we the number of frames between these key frames, including these frames, is $C=L-K$, we would have $p_i=(i*(p_K)+(C-i)*p_L)/C$ for each parameter $p$ and each integer $i$ between $L$ and $K$. If we let $t = i/C$, we could re-phrase this as $p_i=(t*(p_K)+(1-t)*p_L)$, a more familiar way to express pure linear interpolation. This is a straightforward calculation and would produce smoothly-changing parameter values which should translate into smooth motion between the key frames.

Key frame motion is probably more complex than this simple first approach would recognize, however. In fact, we not only want smooth motion between two key frames, but we want the motion from before a key frame to blend smoothly with the motion after that key frame. The linear interpolation discussed above will not accomplish that, however; instead, we need to use a more general interpolation approach. One approach is to start the motion from the starting point more

slowly, move more quickly in the middle, and slow down the ending part of the interpolation so that we stop changing the parameter (and hence stop any motion in the image) just as we reach the final frame of the sequence. In Figure 12.X, we see a comparison of simple linear interpolation on the left with a gradual startup/slowdown interpolation on the right. The right-hand figure shows a sinusoidal curve that we could readily describe by `t = 0.5(1-cos(f/π))`, where in both cases we use `t = i/C` as in the paragraph above, so that we are at frame `K` when t = 0 and frame `L` when t = 1.

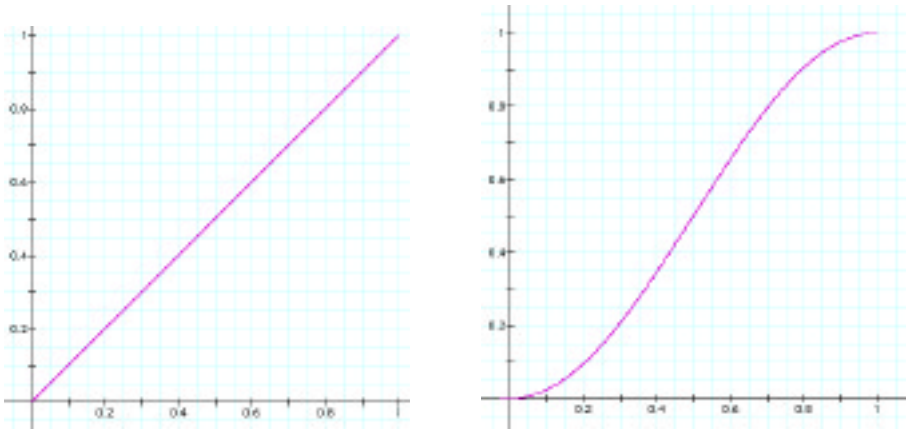


Figure 12.X: two interpolation curves; linear (left) and sinusoidal (right)

In spite of our adjustment to move through the key frames slowly, we still have the problem that a parameter can provide motion (or another effect) in one direction up to a key frame, and then that motion or effect can go off in an entirely different direction when it leaves the key frame and goes to another one. That is, our motion is not yet smooth as it goes through a key frame. To achieve this, we will need to provide a more sophisticated kind of interpolation.

If we consider the quadratic and cubic interpolations from the mathematical fundamentals in an early chapter, we see that there are interpolations among a number of points that meet some of the points. We need to find such an interpolation that allows our interpolation to meet each keyframe exactly and that moves smoothly among a set of keyframes, and in order to do this we need to be able to interpolate the parameters for the frames in a way that meets exactly the parameters for the keyframes and that moves smoothly between values of the parameters. From the various kinds of interpolations available, we would chose the Catmull-Rom interpolation described in the chapter on spline modeling, which gives us the kind of interpolation shown in the second row of Figure 12.Y and that contrasts with the point-to-point interpolation shown in the first row of that figure.

Figure 12.Y: moving in and out of a keyframe (left to right follows time).
Top row: using a two-point interpolation; bottom row: using a multi-point interpolation

Building an animation

The communication you are developing with your animation is very similar to the communication that a director might want to use in a film. Film has developed a rich vocabulary of techniques that will give particular kinds of information, and animations for scientific communication can benefit from thinking about issues in cinematic terms. If you plan to do this kind of work extensively, you should study the techniques of professional animators. Books on animations will show you many, many more things you can do to improve your planning and execution. Initially, you may want to keep your animations simple and hold a fixed eyepoint and fixed lights, allowing only the parts of the model that move with time to move in your frames. However, just as the cinema discovered the value of a moving camera in a moving scene when they invented the traveling shot, the camera boom, and the hand-held walking camera, you may find that you get the best effects by combining a moving viewpoint with moving objects. Experiment with your model and try out different combinations to see what tells your story best.

*A word to the wise...*

Designing the communication in an animation is quite different from the same task for a single scene, and it takes some extra experience to get it really right. Among the techniques used in professional animation is the storyboard — a layout of the overall animation that says what will happen when as the program executes, and what each of your shots is intended to communicate to the viewer.

*Some examples*

Moving objects in your model

Since animation involves motion, one approach to animation is to move individual things in your model. The example in the source file `movingcube.c` takes a mathematical approach to defining the position of a cube, moving it around in space. As we suggested above, the key to this is the `animate()` function that we illustrate below:

```
void animate(void)
{
   #define deltaTime 0.05

// define position for the cube by modeling time-based behavior
   aTime += deltaTime; if (aTime > 2.0*PI) aTime -= 2.0*PI;
   cubex = sin(2.0*aTime);
   cubey = cos(3.0*aTime);
   cubez = cos(aTime);
   glutPostRedisplay();
}
```

This function sets the values of three variables that are later used with a `glTranslate*(...)` function to position the cube in space. You could set variable orientation, size, or other properties of your objects as well with similar processes.

Moving parts of objects in your model

Just as we moved whole objects above, you could move individual parts of a heirarchial model. You could change the relative positions, relative angles, or relative sizes by using variables when you define your model, and then changing the values of those variables in the idle callback. You can even get more sophisticated and change colors or transparency when those help you tell the

story you are trying to get across with your images. The code below increments the parameter `t` and then uses that parameter to define a variable that is, in turn, used to set a rotation angle to wiggle the ears of the rabbit head in the file `rabbitears.c`.

```
void animate(void)
{
#define twopi 6.28318

   t += 0.1;
   if (t > twopi) t -= twopi;
   wiggle = cos(t);
   glutPostRedisplay();
}
```

Moving the eye point or the view frame in your model

Another kind of animation is provided by providing a controlled motion around a scene to get a sense of the full model and examine particular parts from particular locations. This motion can be fully scripted or it can be under user control, though of course the latter is more difficult. In this example, the eye moves from in front of a cube to behind a cube, always looking at the center of the cube, but a more complex (and interesting) effect would have been achieved if the eye path were defined through an evaluator with specified control points. This question may be revisited when we look at evaluators and splines.

```
void display( void )
{
// Use a variable for the viewpoint, and move it around ...
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt( ep.x, ep.y, ep.z, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
...
}

void animate(void)
{
   GLfloat numsteps = 100.0, direction[3] = {0.0, 0.0, -20.0};

   if (ep.z < -10.0) whichway = -1.0;
   if (ep.z > 10.0)  whichway = 1.0;
   ep.z += whichway*direction[2]/numsteps;
   glutPostRedisplay();
}
```

As you travel, you need to control not only the position of the eye point, but also the entire viewing environment — in simple terms, the entire `glutLookAt(...)` parameter list. So not only the eye point, but also the view reference point and the up vector must be considered in creating an effective moving viewpoint. Further, as you move around you will sometimes find yourself moving nearer to objects or farther from them. This means you will have the opportunity to use level-of-detail techniques to control how those objects are presented to the viewer while you keep the frame rate as high as possible. There's a lot of work here to do everything right, but you can make a good start much more easily.

Changing features of your model

There are many other special features of your models and displays that you can change with time to create the particular communication you want. Among them, you can change colors, properties of

your lights, transparency, clipping planes, fog, texture maps, granularity of your model, and so on. Almost anything that can be defined with a variable instead of a constant can be changed by changing the model.

In the particular example for this technique, we will change the size and transparency of the display of one kind of atom in a molecule, as we show in Figure 12.1. The change in the image is driven by a parameter `t` that is changed in the `idle` callback, and the parameter in turn gives a sinusoidal change in the size and transparency parameters for the image. This will allow us to put a visual emphasis on this kind of atom so that a user could see where that kind of atom fits into the molecule. This is just a small start on the overall kinds of things you could choose to animate to put an emphasis on a part of your model.

```
void molecule(void)
{
...
         j = atoms[i].colindex;  // index of color for atom i
         for (k=0; k<4; k++)
         {  // copy atomColors[j], adjust alpha by alphaMult
            myColor[k] = atomColors[j][k];
         }
         if (j==CARBON) myColor[3] += alphaAdd;
         glMaterialfv(..., myColor );
         glTranslatef(...);
         if (j==CARBON)
            gluSphere(atomSphere,sizeMult*ANGTOAU(atomSizes[j]),GRAIN,GRAIN);
         else
            gluSphere(atomSphere,ANGTOAU(atomSizes[j]),GRAIN,GRAIN);
         glPopMatrix();
...
 }
...
void animate(void)
{
   t += 0.1; if (t > 2.0*M_PI) t -= 2.0*M_PI;
   sizeMult = (1.0+0.5*sin(t));
   alphaAdd = 0.2*cos(t);
   glutPostRedisplay();
}
```

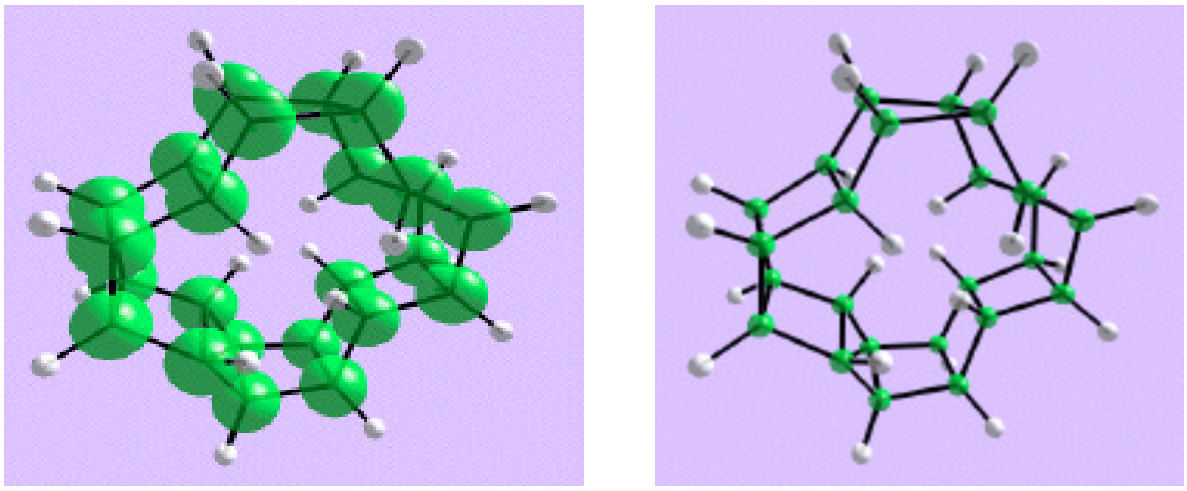


Figure 12.1: molecule with carbon expanded (left) or contracted (right)

*Some points to consider when doing animations with OpenGL*

There are some things about OpenGL that you need to understand more fully when you move your eyepoint than you when you simply create a single image. The viewing transformation is part of the overall modeling transformation, and it needs to be done at the right place if you are going to use parameters to define your view. In the `display()` function in the `viewcube.c` example, you will note that the modeling transformation is set to the identity, the `glutLookAt(...)` function is called, the resulting transformation is saved for future use, and then the rotation processes are called. This keeps the viewing transformation from changing the position of objects in the model and thus keeps the animation looking right.

Finally, be careful when you use texture maps in animations. There is always the possibility of aliasing with texture maps, and when they are animated the aliasing can cause strange-looking behavior in the texture rendering. Some effort in antialiasing textures is particularly important in animating them.

*Code examples*

As we noted above, and as the code examples show, animation control is primarily managed by changing parameters of your scene in the callback for the `idle` event. We have seen several of these examples to control several aspects of the model, the scene, and the display. You should experiment as widely as you can to see what kind of things you can control and how you can control them in order to become fluent in using animation as a communication tool.

*References*

You should look at videos of graphics work to get a fuller understanding of what you can do with this tool. In general, though, you want to avoid high-end entertainment-focused animations and look at informational animations — presentations of scientific or technical work are idea. But when you look at video, you are probably looking at *presentation-level* animations, work that is done to impress others. Most of the work you can do in a first course is more likely *personal-level* or *peer-level* animation: work that is done to explore an idea yourself or to share with a friend or colleague. So don't think you can match the quality of the videos you watch; try to find the key communication ideas and learn from them.

# High-Performance Graphics Techniques and Games Graphics

*Prerequisites*

A solid understanding of the concepts of computer graphics, and a knowledge of the details of the OpenGL graphics API in sufficient depth that you can consider alternatives to standard approaches to creating animated, interactive images.

*Definitions*

The speed with which we can generate an image is always of interest because we want to be able to get our results quickly, whether we're doing a single scene or an animation. Waiting for an image to be produced can be frustrating and can even get in the way of the image's effectiveness. This is evident in many kinds of graphics applications, but it is probably most evident in computer games, so this is the context that frames the discussion in this chapter.

Making effective computer games involves many things, including storytelling, creating characters, maintaining databases of game features, and many general programming techniques to deliver maximum speed of operation to the player. One of the critical performance areas — because it's one of the most compute-intensive bottlenecks in presenting the game to the player — is the graphics that present the game to the user. This is a different question than we've been dealing with in computer graphics to this point. Up to this point in these notes, we have focused on the quality of the images while maintaining as much performance as we can, but in this chapter we reverse this emphasis: we focus on the performance of the programs while maintaining as much quality as we can. This change makes a major difference in the kind of processes and techniques we use.

In a sense, this is not a new issue in computer graphics. For over 20 years, the computer field has included the area of "real-time graphics." This originally focused on areas such as flight simulators, real-time monitoring of safety-critical system processes, and real-time simulations, often using the highest-power computers available at the time. Some of the real-time graphics processes also were used in educational applications that are essentially simulations. But the demands that games place on graphics are both as extreme as these and are focused on personal computers with widely varying configurations, making it important to bring these real-time techniques into a graphics text.

*Techniques*

Fundamentally, high-performance computer graphics, especially as applied to games, takes advantage of a few simple principles:

- Use hardware acceleration when you can, but don't assume that everyone has it and be ready to work around it when you need to
- Do some work to determine what you don't need to display
    - Look for techniques that will support easy ways to cull objects or pieces of objects from the set of things to be displayed
- Take advantage of capabilities of texture mapping
    - Create objects as texture maps instead of as fully-rendered objects
    - Use multitextures and textures with both low and high resolution
- Use any techniques available to support the fastest display techniques you can
    - Display lists
    - Level of detail
- Avoid unnecessary lighting calculations
    - When you draw any object, only enable lights near the object

    - Use fog
- Collision detection

We will discuss these and show you as many techniques as we can to support them in helping you create high-performance graphics.

There are also some unique problems in some gaming situations that are not found in most other areas of graphics. The main one we will discuss is collision detection, because this is an area that requires some simple computation that we can streamline in ways that are similar to the techniques discussed above.

Hardware avoidance

The computer graphics pipeline includes a number of places where there are opportunities to put hardware into the process to get more performance, and graphics cards have usually been quick to take advantage of these. When you use OpenGL on a system with such a card, the graphics system will probably use the hardware features automatically. Paradoxical as it may seem, however, relying on this approach to speed may not be the best idea for high performance. Parts of your audience might not have the kind of acceleration you are programming for, for example, and even hardware has its cost. But more fundamentally, sometimes using standard techniques and relying on the hardware to get you speed will be slower than looking for alternative techniques that can avoid the processing the card accelerates.

As an example, consider the Z-buffer that is supported on almost every graphics card. When you use the Z-buffer to handle depth testing, you must carry out reading, comparing, and writing operations for each pixel you draw. If you have a fast graphics card, this is higher-speed reading, comparing, and writing, of course, but it is faster to avoid these operations than to optomize them. There are some techniques we will talk about below that allow you to do some modest computation to avoid entire polygons or, even better, to avoid making depth tests altogether.

Designing out visible polygons

As you lay out the overall design of your image, you can ensure that there is only limited visibility of the overall scene from any viewpoint. This is part of the reason why one sees many walls in games — we see large polygons with texture mapping for detail, and only a very few polygons are visible from any point. The sense of visual richness is maintained by moving quickly between places with different and clearly understood environments, so that when the player makes the transition from one place to another, they see a very different world, and even though the world is simple, the constant changing makes the game seem constantly fresh.

Other techniques involve pre-computing what objects will be visible to the player from what positions. As a very simple example, when a player moves out of one space into another, nothing in the space being vacated can be seen, so all the polygons in that space can be ignored. this kind of pre-computed design can involve maintaining lists of visible polygons from each point with each direction the player is facing, a classical tradeoff of space for speed.

Culling polygons

One of the traditional techniques for avoiding drawing is to design only objects that are made up of polyhedra (or can be made from collections of polyhedra) and then to identify those polygons in the polyhedra that are back faces from the eye point. In any polyhedron whose faces are opaque, any polygon that faces away from the eye point is invisible to the viewer, so if we draw these and use the depth buffer to manage hidden surfaces, we are doing work that cannot result in any visible effects. Thus it is more effective to decide when to avoid drawing these at all.

The decision as to whether a polygon faces toward or away from the eye point is straightforward. Remember that the normal vector for the polygon points in the direction of the front (or outside) face, so that the polygon will be a front face if the normal vector points toward the eye, and will be a back face if the normal vector points away from the eye. Front faces are potentially visible; back faces are never visible. In terms of the diagram in Figure 13.1, with the orientation of the vectors N and E as shown, a front face will have an acute angle between the normal and eye vectors, so N•E will be positive, and a back face will have an obtuse angle, so N•E will be negative. Thus a visibility test is simply the algebraic sign of the term N•E. Choosing not to display any face that does not pass the visibility test is called *backface culling*.

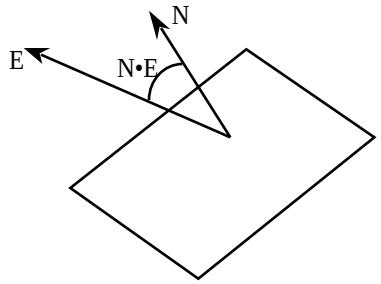



Figure 13.1: the front face test

This kind of culling can readily be done in your graphics program before any calls to the graphics API functions, but many graphics APIs support backface culling directly. In OpenGL, culling is supported by enabling an operational procedure, `GL_CULL_FACE`. Deciding what makes up a front face is done with the function

```
void glFrontFace(Glenum mode)
```

where `mode` takes on the values `GL_CCW` or `GL_CW` (counterclockwise or clockwise), depending on the orientation of the vertices of a front face as seen from the eye point. You can then choose which kind of face to cull with the function

```
void glCullFace(Glenum mode)
```

in this case, `mode` takes on the values `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK`. If culling is enabled, polygons are not drawn if they are the kind of face selected in `glCullFace`, where the concept of a front face is defined in `glFrontFace.`

Another kind of culling can take place on the viewing volume. Here you can compare each vertex of your polyhedron or polygon with the bounding planes on your view volume; if all of the vertices lie outside of the viewing volume based on comparisons with the same bounding plane, then the polyhedron or polygon cannot be seen in the defined view and need not be drawn. This calculation should be done after the viewing transformation so the boundaries of the view volume are easy to use, but before the polygons are actually rendered. Recalling that the viewing volume is a rectangular pyramid with apex at the origin and expanding in the negative Z-direction, the actual comparison calculations are given by the following:

`y > T*Z/ZNEAR` or `y < B*Z/ZNEAR`
`x > R*Z/ZNEAR` or `x < L*Z/ZNEAR`
`z > ZNEAR` or `z < ZFAR`

where `T`, `B`, `R`, and `L` are the top, bottom, right, and left coordinates of the near plane `Z = ZNEAR` as indicated by the layout in the diagram in Figure 13.2 below.

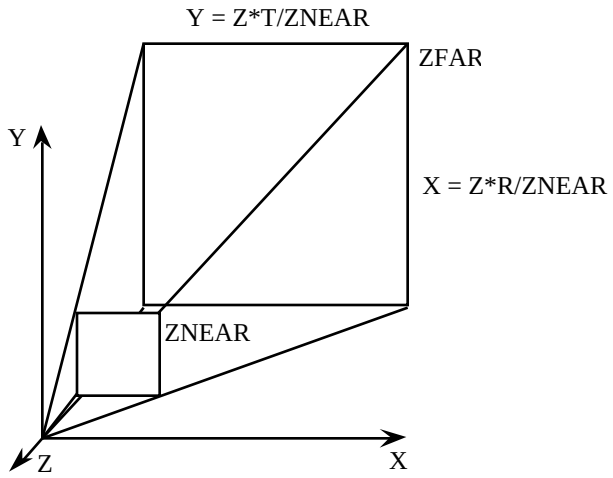



Figure 13.2: the comparisons for the bounding volume computation

Avoiding depth comparisons

One of the classic computer graphics techniques is to order your objects by depth and draw them from back to front, mimicing the way light would progress from objects to your eye. This is called the *painter's algorithm*, and it was most popular when the Z-buffer was beyond the scope of most graphics programming. This technique can be relatively simple if your model is static, had no interlocking polygons, and was intended to be seen from a single viewpoint, because these make it easy to figure out what "back" and "front" mean and which of any two polygons is in front of the other. This is not the usual design philosophy for interactive graphics, however, and particularly for games, because moving geometry and moving eye points are constantly changing which things are in front of what others. So if we were to use this approach, we would find ourselves having to calculate distances from a moving eye point in varying directions, which would be very costly to do.

It may be possible to define your scene in ways that can ensure that you will only view it from points where the depth is known, or you may need to define more complex kinds of computation to give you that capability. A relatively common approach to this problem is given by binary space partitioning, as described below.

Front-to-back drawing

Sometimes a good idea is also a good idea when it is thought of backwards. As an alternative to the painter's algorithm approach, sometimes you can arrange to draw objects only from the front to the back. This still requires a test, but you need test only whether a pixel has been written before you write it for a new polygon. When you are working with polygons that have expensive calculations per pixel, such as complex texture maps, you want to avoid calculating a pixel only to find it overwritten later, so by drawing from the front to back you can calculate only those pixels you will actually draw. You can use BSP tree techniques as discussed below to select the nearest objects, rather than the farthest, to draw first, or you can use pre-designed scenes or other approaches to know what objects are nearest.

Binary space partitioning

There are other approach to avoiding depth comparisons. It is possible to use techniques such as binary space partitioning to determine what is visible, or to determine the order of the objects as

seen from the eyepoint. Here we design the scene in a way that can be subdivided into convex sub-regions by planes through the scene space and we can easily compute which of the sub-

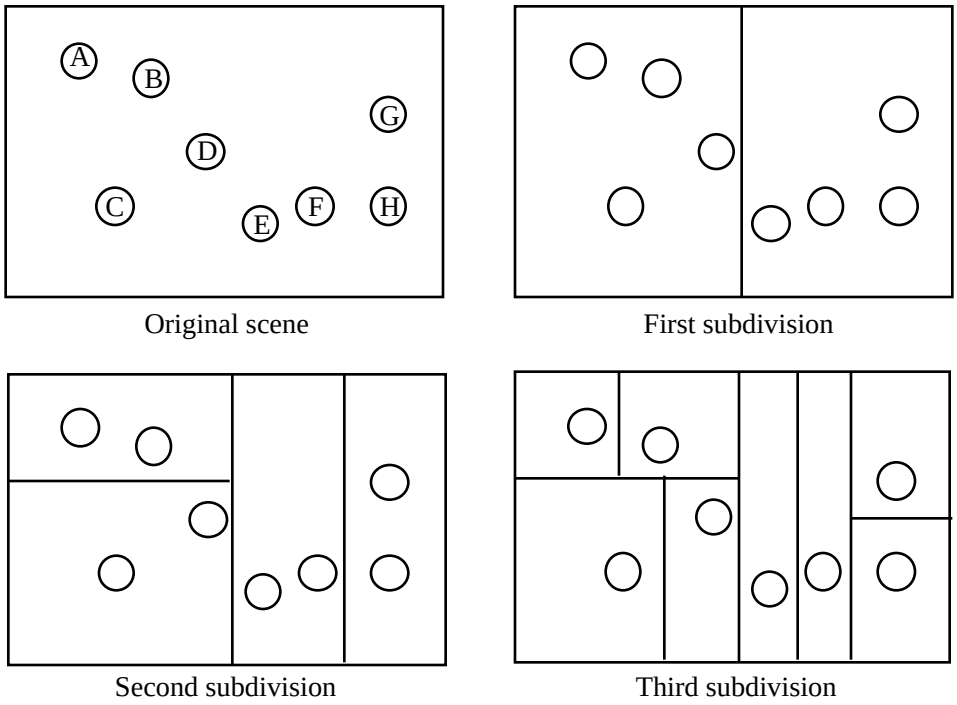



Figure 13.3: a collection of objects in a subdivided space

regions is nearer and which is farther. This subdivision can be recursive: find a plane that does not intersect any of the objects in the scene and for which half the objects are in one half-space relative to the plane and the other half are in the other half-space, and regard each of these half-spaces as a separate scene to subdivide each recursively. The planes are usually kept as simple as possible by techniques such as choosing the planes to be parallel to the coordinate planes in your space, but if your modeling will not permit this, you can use any plane at all. This technique will fail, however, if you cannot place a plane between two objects, and in this case more complex modeling may be needed. This kind of subdivision is illustrated in Figure 13.3 for the simpler 2D case that is easier to see.

This partitioning allows us to view the space of the image in terms of a binary space partitioning tree (or *BSP tree*) that has the division planes as the interior nodes and the actual drawn objects as its leaves. With each interior note you can store the equation of the plane that divides the space, and with each branch of the tree you can store a sign that says whether that side is positive or negative when its coordinates are put into the plane equation. These support the computation of which side is nearer the eye, as noted below. This tree is shown in Figure 13.4, with each interior node indicated by the letters of the objects at that point in the space. With any eye point, you can determine which parts of the space are in front of which other parts by making one test for each interior node, and re-adjusting the tree so that (for example) the farther part is on the left-hand branch and the nearer part is on the right-hand branch. This convention is used for the tree in the figure with the eye point being to the lower right and outside the space. The actual drawing then can be done by traversing the tree left-to-right and drawing the objects as you come to them.

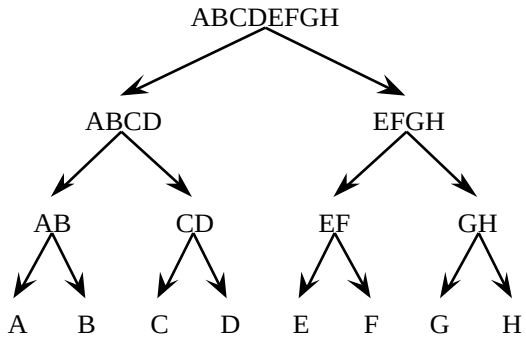



Figure 13.4: a binary space partitioning tree

The actual test for which part is nearer can be done by considering the relation of the eye point to the plane that divides the space. If you put the eye coordinates into the plane equation, you will get either a positive or negative value, and objects on the side of the plane nearer the eye will have the same relation to the plane as the eye. Further, as your eye moves, you will only need to recompute the orientation of the BSP tree when your eye point crosses one of the partitioning planes, and you may be able to conclude that some of the orientations do not need to be recomputed at all.

If you have any moving objects in your scene, you must determine their relation to the other objects and account for them in relation to the BSP tree. It is common to have moving objects only show up in front of other things, and if this is the case then you can draw the scene with the BSP tree and simply draw the moving object last. However, if the moving object is placed among the other drawn objects, you can add it into the BSP tree in particular spaces as it moves, with much the same computation of its location as you did to determine the eye location, and with the object moved from one region to another when it crosses one of the dividing planes. Details of this operation are left to the reader at this time.

Clever use of textures

We have already seen that textures can make simple scenes seem complex and can give an audience a sense of seeing realistic objects. When we take advantage of some of the capabilities of texture mapping we can also deal with graphic operations in precisely the sense that we started this chapter with: reducing the accuracy in hard-to-see ways while increasing the efficiency of the graphics.

One technique is called *billboarding*, and involves creating texture-mapped versions of complex objects that will only be seen at a distance. By taking a snapshot — either a photograph or a once-computed image — and using the alpha channel in the texture map to make all the region outside the object we want to present blend to invisible, we can put the texture onto a single rectangle that is oriented towards the eye point and get the effect of a tree, or a building, or a vehicle, on each rectangle. If we repeat this process many times we can build forests, cities, or parking lots without doing any of the complex computation needed to actually compute the complex object. Orienting each billboard to eye point involves computing the positions of the billboard and the eye (which can be readily done from the scene graph by looking for translations that affect both) and computing the cylindrical or spherical coordinates of the eye point if the billboard is regarded as the origin. The latitude and longitude of the eye point from the billboard will tell you how to rotate the billboard so it faces toward the eye. Note that there are two ways to view a billboard; if it represents an object with a fixed base (tree, building, ...) then you only want to rotate it around its fixed axis; if it represents an object with no fixed point (snowflake) then you probably want to rotate it around two axes so it faces the eye directly.

Another technique is to use techniques at several levels of resolution. OpenGL provides a capacity to do *mipmaps*, texture maps at many resolutions. If you start with the highest-resolution (and hence largest) texture map, you can automatically create texture maps with lower resolution. Recall that each dimension of any texture map must be a power of two, so you can create maps with dimensions half the original, one fourth the original, and so on, yielding a sequence of texture maps that you can use to achieve your textures without the aliasing you would get if you used the larger texture.

Yet another approach is to layer textures to achieve your desired effects. This capability, called *multitexturing*, is only available for OpenGL at level 1.2 and beyond. It allows you to apply multiple textures to a polygon in any order you want, so you can create a brick wall as a color texture map, for example, and then apply a luminance texture map to make certain parts brighter, simulating the effect of light through a window or the brightness of a torch without doing any lighting computations whatsoever.

These last two techniques are fairly advanced and the interested student is referred to the manuals for more details.

System speedups

One kind of speedup available from the OpenGL system is the display list. As we noted in Chapter 3, you can assemble a rich collection of graphics operations into a display list that executes much more quickly than the original operations. This is because the computations are done at the time the display list is created, and only the final results are sent to the final output stage of the display. If you pre-organize chunks of your image into display lists, you can execute the lists and gain time. Because you cannot change the geometry once you have entered it into the display list, however, you cannot include things like polygon culling or changed display order in such a list.

Another speedup is provided by the "geometry compression" of triangle strips, triangle fans, and quad strips. If you can ensure that you can draw your geometry using these compression techniques, even after you have done the culling and thresholding and have worked out the sequence you want to use for your polygons, these provide significant performance increases.

LOD

Level of Detail (usually just LOD) involves creating multiple versions of a graphical element and displaying a particular one of them based on the distance the element is from the viewer. This allows you to create very detailed models that will be seen when the element is near the viewer, but more simple models that will be seen when the element is far from the viewer. This saves rendering time and allows you to control the way things will be seen — or even whether the element will be seen at all.

Level of detail is not supported directly by OpenGL, so there are few definitions to be given for it. However, it is becoming an important issue in graphics systems because more and more complex models and environments are being created and it is more and more important to display them in real time. Even with faster and faster computer systems, these two goals are at odds and techniques must be found to display scenes as efficiently as possible.

The key concept here seems to be that the image of the object you're dealing with should have the same appearance at any distance. This would mean that the farther something is, the fewer details you need to provide or the coarser the approximation you can use. Certainly one key consideration is that one would not want to display any graphical element that is smaller than one pixel, or perhaps smaller than a few pixels. Making the decision on what to suppress at large distance, or

what to enhance at close distance, is probably still a heuristic process, but there is research work on coarsening meshes automatically that could eventually make this better.

LOD is a bit more difficult to illustrate than fog, because it requires us to provide multiple models of the elements we are displaying. The standard technique for this is to identify the point in your graphical element (`ObjX, ObjY, ObjZ`) that you want to use to determine the element's distance from the eye. OpenGL will let you determine the distance of any object from the eye, and you can determin the distance through code similar to that below in the function that displayed the element:

```
glRasterPos3f( ObjX, ObjY, ObjZ );
glGetFloatv( GL_CURRENT_RASTER_DISTANCE, &dist );
if (farDist(dist)) { ... // farther element definition
}
else { ...               // nearer element definition
}
```

This allows you to display one version of the element if it is far from your viewpoint (determined by the a function `float farDist(float)` that you can define), and other versions as desired as the element moves nearer to your viewpoint. You may have more than two versions of your element, and you may use the distance that

```
glGetFloatv(GL_CURRENT_RASTER_DISTANCE, &dist)
```

returns in any way you wish to modify your modeling statements for the element.

To illustrate the general LOD concept, let's display a GLU sphere with different resolutions at different distances. Recall from the early modeling discussion that the GLU sphere is defined by the function

```
void gluSphere (GLUquadricObj *qobj, GLdouble radius,
                GLint slices, GLint stacks);
```

as a sphere centered at the origin with the radius specified. The two integers *slices* and *stacks* determine the granularity of the object; small values of *slices* and *stacks* will create a coarse sphere and large values will create a smoother sphere, but small values create a sphere with fewer polygons that's faster to render. The LOD approach to a problem such as this is to define the distances at which you want to resolution to change, and to determine the number of slices and stacks that you want to display at each of these distances. Ideally you will analyze the number of pixels you want to see in each polygon in the sphere and will choose the number of slices and stacks that provides that number.

Our modeling approach is to create a function `mySphere` whose parameters are the center and radius of the desired sphere. In the function the depth of the sphere is determined by identifying the position of the center of the sphere and asking how far this position is from the eye, and using simple logic to define the values of *slices* and *stacks* that are passed to the `gluSphere` function in order to select a relatively constant granularity for these values. The essential code is

```
      myQuad=gluNewQuadric();
      glRasterPos3fv( origin );
//    howFar = distance from eye to center of sphere
      glGetFloatv( GL_CURRENT_RASTER_DISTANCE, &howFar );
      resolution = (GLint) (200.0/howFar);
      slices = stacks = resolution;
      gluSphere( myQuad , radius , slices , stacks  );
```

This example is fully worked out in the source code `multiSphere.c` included with this module. Some levels of the sphere are shown in Figure 13.5 below.

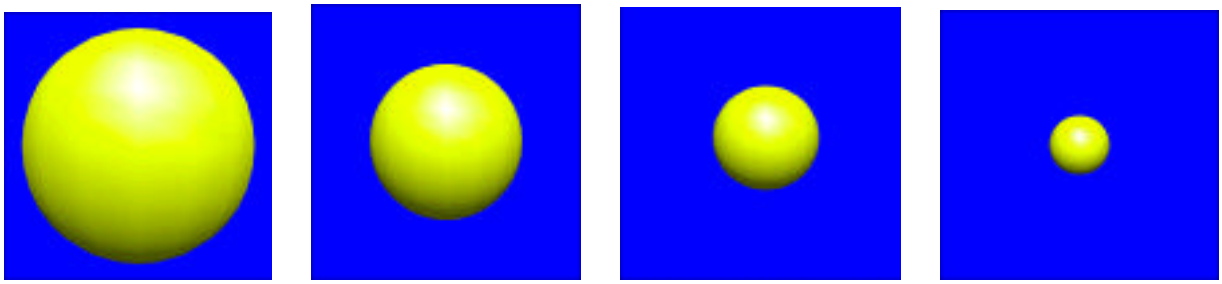


Figure 13.5: levels of detail in the sphere, from high detail level at left to lower at right

Reducing lighting computation

While we may include eight (or more) lights in a scene, each light we add takes a toll on the time it takes to render the scene. Recalling the lighting computations, you will recall that we calculate the ambient, diffuse, and specular lighting for each light and add them together to compute the light for any polygon or vertex. However, if you are using positional lights with attenuation, the amount of light a particular light adds to a vertex is pretty small when that vertex is not near the light. You may choose to simplify the light computation by disabling lights when they are not near the polygon you are working on. Again, the principle is to spend a little time on computation when it can offer the possibility of saving more time on the graphics calculation.

Fog

Fog is a technique which offers some possibility of using simpler models in a scene while hiding some of the details by reducing the visibility of the models. The tradeoff may or may not be worth doing, because the simpler models may not save as much time as it takes to calculate the effect of the fog. We include it here more because of its conceptual similarity to level-of-detail questions than for pure efficiency reasons.

When you use fog, the color of the display is modified by blending it with the fog color as the display is finally rendered from the OpenGL color buffer. Details of the blending are controlled by the contents of the depth buffer. You may specify the distance at which this blending starts, the distance at which no more blending occurs and the color is always the fog color, and the way the fog color is increased through the region between these two distances. Thus elements closer than the near distance are seen with no change, elements between the two distances are seen with a color that fades towards the fog color as the distance increases, and elements farther than the far distance are only seen with the full effect of the fog as determined by the fog density. This provides a method of depth cueing that can be very useful in some circumstances.

There are a small number of fundamental concepts needed to manage fog in OpenGL. They are all supplied through the `glFog*(`*`param`*`, value)` functions as follows, similarly to other system parameter settings, with all the capitalized terms being the specific values used for *`param`*. In this discussion we assume that color is specified in terms of RGB or RGBA; indexed color is noted briefly below.

start and end:

> fog is applied between the starting value `GL_FOG_START` and the ending value `GL_FOG_END`, with no fog applied before the starting value and no changes made in the fog after the end value. Note that these values are applied with the usual convention that the center of view is at the origin and the viewpoint is at a negative distance from the origin. The usual convention is to have fog start at 0 and end at 1.

mode:

OpenGL provides three built-in fog modes: linear, exponential, or exponential-squared. These affect the blending of element and fog color by computing the fog factor `ff` as follows:

- GL_LINEAR: `ff = density*z'` for `z'= (end-z)/(end-start)` and any `z` between `start` and `end`.
- GL_EXP: `ff = exp(-density*z')` for `z'` as above
- GL_EXP2: $ff = exp(-density*z')^2$ for `z'` as above

The fog factor is then clamped to the range [0,1] after it is computed. For all three modes, once the fog factor `ff` is computed, the final displayed color `Cd` is interpolated by the factor of ff between the element color `Ce` and the fog color `Cf` by

`Cd=ff*Ce+(1-ff)*Cf`.

density:

density may be thought of as determining the maximum attenuation of the color of a graphical element by the fog, though the way that maximum is reached will depend on which fog mode is in place. The larger the density, the more quickly things will fade out in the fog and thus the more opaque the fog will seem. Density must be between 0 and 1.

color:

while we may think of fog as gray, this is not necessary — fog may take on any color at all. This color may be defined as a four-element vector or as four individual parameters, and the elements or parameters may be integers or floats, and there are variations on the `glFog*()` function for each. The details of the individual versions of `glFog*()` are very similar to `glColor*()` and `glMaterial*()` and we refer you to the manuals for the details. Because fog is applied to graphics elements but not the background, it is a very good idea to make the fog and background colors be the same.

There are two additional options that we will skim over lightly, but that should at least be mentioned in passing. First, it is possible to use fog when you are using indexed color in place of RGB or RGBA color; in that case the color indices are interpolated instead of the color specification. (We did not cover indexed color when we talked about color models, but some older graphics systems only used this color technology and you might want to review that in your text or reference sources.) Second, fog is hintable — you may use `glHint(…)` with parameter `GL_FOG_HINT` and any of the hint levels to speed up rendering of the image with fog.

Fog is an easy process to illustrate. All of fog's effects can be defined in the initialization function, where the fog mode, color, density, and starting and ending points are defined. The actual imaging effect happens when the image is rendered, when the color of graphical elements are determined by blending the color of the element with the color of fog as determined by the fog mode. The various fog-related functions are shown in the code fragment below.

```
void myinit(void)
{     ...
      static  GLfloat fogColor[4]={0.5,0.5,0.5,1.0}; // 50% gray
      ...
// define the fog parameters
      glFogi(GL_FOG_MODE, GL_EXP);        // exponential fog increase
      glFogfv(GL_FOG_COLOR, fogColor);    // set the fog color
      glFogf(GL_FOG_START, 0.0 );         // standard start
      glFogf(GL_FOG_END, 1.0 );           // standard end
      glFogf(GL_FOG_DENSITY, 0.50);       // how dense is the fog?
      ...
      glEnable(GL_FOG);                   // enable the fog
...
}
```

An example illustrates our perennial cube in a foggy space, shown in Figure 13.6. This builds on the earlier textured cube to include fog in addition to the texture map on one face of the cube. (The texture map itself is included with this module; it is a screen capture of a graphic display, saved in Photoshop™ as a raw RGB file with no structure.) The student is encouraged to experiment with the fog mode, color, density, and starting and ending values to examine the effect of these parameters' changes on your images. This example has three different kinds of sides (red, yellow, and texture-mapped) and a fog density of only 0.15, and has a distinctly *non*-foggy background for effect.

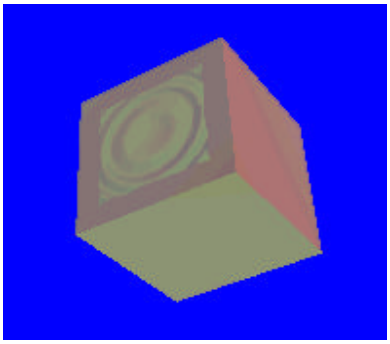


Figure 13.6: a foggy cube (including a texture map on one surface)

Collision detection

When you do polygon-based graphics, the question of collisions between objects reduces to the question of collisions between polygons. By reducing the general polygon to a triangle, that further reduces to the question of collisions between an edge and a triangle. We actually introduced this issue earlier in the mathematical background, but it boils down to extending the edge to a complete line, intersecting the line with the plane of the polygon, and then noting that the edge meets the polygon if it meets a sequence of successively more focused criteria:

- the parameter of the line where it intersects the plane must lie between 0 and 1
- the point where the line intersects the plane must lie within the smallest circle containing the triangle
- the point where the line intersects the plane must lie within the body of the triangle.

This comparison process is illustrated in Figure 13.7 below.

If you detect a collision when you are working with moving polyhedra, the presence of an intersection might require more processing because you want to find the exact moment when the moving polyhedra met. In order to find this intersection time, you must do some computations in the time interval between the previous step (before the intersection) and the current step (when the intersection exists). You might want to apply a bisection process on the time, for example, to determine whether the intersection existed or not halfway between the previous and current step, continuing that process until you get a sufficiently good estimate of the actual time the objects met. Taking a different approach, you might want to do some analytical computation to calculate the intersection time given the positions and velocities of the objects at the previous and current times so you can re-compute the positions of the objects to reflect a bounce or other kind of interaction between them.

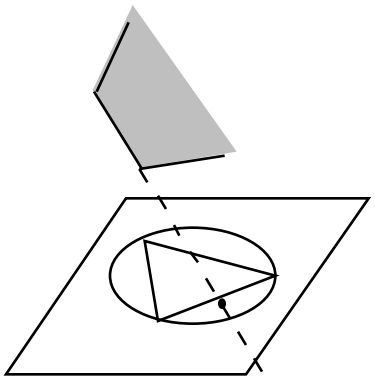


Figure 13.7: the collision detection computation

*A word to the wise...*

As LOD techniques are used in animated scenes, you must avoid sudden appearance or disappearance of objects as well as sudden jumps in appearance. These artifacts cause a break in the action that destroys the believability of the animation. It can be useful to create a fog zone deep in a scene and have things appear through the fog instead of simple jumping into place.

Fog is a tempting technique because it looks cool to have objects that aren't as sharp and “finished” looking as most objects seem to be in computer graphics. This is similar to the urge to use texture mapping to get objects that don’t seem to be made of smooth plastic, and the urge to use smooth-shaded objects so they don’t seem to be crudely faceted. In all these cases, though, using the extra techniques has a cost in extra rendering time and programming effort, and unless the technique is merited by the communication needed in the scene, it can detract from the real meaning of the graphics.

# Object Selection

*Prerequisites*

An understanding of the rendering process, an understanding of event handling, and a knowledge of list management to handle hit lists for events

*Introduction*

Object selection is a tool that permits the user to interact with a scene in a more direct way than is possible with the kind of external events, such as menu selections, mouse clicks, or key presses, that we saw in the earlier chapter on event handling. With object selection we can get the kind of direct manipulation that we are familiar with from graphical user interfaces, where the user selects a graphical object and then applies operations to it.

Conceptually, object selection allows a user to identify a particular object with the cursor and to choose it by clicking the mouse button when the cursor is on the object. The program must be able to identify what was selected, and then must have the ability to apply whatever action the user chooses to that particular selected object.

OpenGL has many facilities for identifying objects that corespond to mouse events — clicks on the screen — but many of them are quite complex and require the programmer to do significant work to identify the objects that lie between the front and back clipping planes along the line between the points in those planes that correspond to the click. However, OpenGL makes it possible for you to get the same information with much less pain with a built-in selection operation that simply keeps track of which parts of your scene involve the pixel that was selected with the mouse.

The built-in selection approach calls for the mouse event to request that you render your scene twice. In the first rendering, you work in the same mode you are used to: you simply draw the scene in `GL_RENDER` mode. In the mouse event callback, you change to `GL_SELECT` mode and re-draw the scene with each item of interest given a unique name. When the scene is rendered in `GL_SELECT` mode, nothing is actually changed in the frame buffer but the pixels that would be rendered are identified. When any named object is found that would include the pixel selected by the mouse, that object's name is added to a stack that is maintained for that name. This name stack holds the names of all the items in a hierarchy of named items that were hit. When the rendering of the scene in `GL_SELECT` modeis finished, a list of hit records is produced, with one entry for each name of an object whose rendering included the mouse click point, and the number of such records is returned when the system is returned to `GL_RENDER` mode. The structure of these hit records is described below. You can then process this list to identify the item nearest the eye that was hit, and you can proceed to do whatever work you need to do with this information.

The concept of "item of interest" is more complex than is immediately apparent. It can include a single object, a set of objects, or even a hierarchy of objects. Think creatively about your problem and you may be surprised just how powerful this kind of selection can be.

*Definitions*

The first concept we must deal with for object selection is the notion of a *selection buffer*. This is an array of unsigned integers (`GLuint`) that will hold the array of *hit records* for a mouse click. In turn, a hit record contains several items as illustrated in Figure 14.1. These include the number of items that were on the *name stack*, the nearest (`zmin`) and farthest (`zmax`) distances to objects on the stack, and the list of names on the name stack for the selection. The distances are integers because they are taken from the Z-buffer, where you may recall that distances are stored as integers

in order to make comparisons more effective. The name stack contains the names of all the objects in a hierarchy of named objects that were selected with the mouse click.

The distance to objects is given in terms of the viewing projection environment, in which the nearest points have the smallest non-negative values because this environment has the eye at the origin and distances increase as points move away from the eye. Typical processing will examine each selection record to find the record with the smallest value of `zmin` and will work with the names in that hit record to carry out the work needed by that hit. This work is fairly typical of the handling of any list of variable-length records, proceeding by accumulating the starting points of the individual records (starting with 0 and proceeding by adding the values of `(nitems+3)` from the individual records), with the zmin values being offset by 1 from this base and the list of names being offset by 3. This is not daunting, but it does require some care.

In OpenGL, choosing an object by a direct intersection of the object with the pixel identified by the mouse is called *selection*, while choosing an object by clicking near the object is called *picking*. In order to do picking, then, you must identify points near, but not necessarily exactly on, the point where the mouse clicks. This is discussed toward the end of this note.

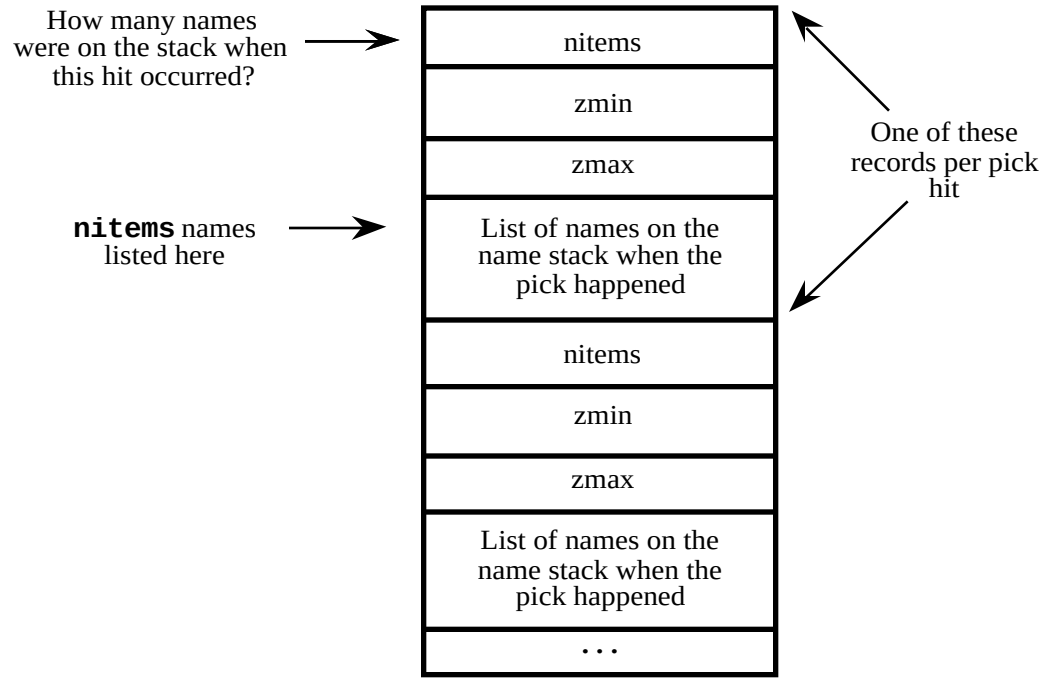



Figure 14.1: the structure of the selection buffer

*Making selection work*

The selection or picking process is fairly straightforward. The function `glRenderMode(mode)` allows you to draw in either of two modes: render mode (`GL_RENDER`) invokes the graphics pipeline and produces pixels in the frame buffer, and select mode (`GL_SELECT`) calculates the pixels that would be drawn if the graphics pipeline were to be invoked, and tests the pixels against the pixels that were identified by the mouse click. As illustrated in the example below, the mouse function can be defined to change the drawing mode to `GL_SELECT` and to post a redisplay operation. The display function can then draw the scene in select mode with selection object names defined with `glutLoadName(int)` to determine what name will be put into the selection buffer if the object includes the selected pixel, noting that the mode can be checked to decide what is to be

drawn and/or how it is to be drawn, and then the selection buffer can be examined to identify what was hit so the appropriate processing can be done. After the selection buffer is processed, the scene can be displayed again in render mode to present the effect of the selection.

In the outline above, it sounds as though the drawing in select mode will be the same as in render mode. But this is usually not the case; anything that you don't want the user to be able to select should not be drawn at all in select mode. Further, if you have a complex object that you want to make selectable, you may not want to do all the work of a full rendering in select mode; you need only design an approximation of the object and draw that. You can even select things that aren't drawn in render mode by drawing them in select mode. Think creatively and you can find that you can do interesting things with selection.

It's worth a word on the notion of selection names. You cannot load a new name inside a `glBegin(mode)-glEnd()` pair, so if you use any geometry compression in your object, it must all be within a single named object. You can, however, nest names with the *name stack*, using the `glPushName(int)` function so that while the original name is active, the new name is also active. For example, suppose we were dealing with automobiles, and suppose that we wanted someone to select parts for an automobile. We could permit the user to select parts at a number of levels; for example, to select an entire automobile, the body of the automobile, or simply one of the tires. In the code below, we create a heirarchy of selections for an automobile ("Jaguar") and for various parts of the auto ("body", "tire", etc.) In this case, the names `JAGUAR`, `BODY`, `FRONT_LEFT_TIRE`, and `FRONT_RIGHT_TIRE` are symbolic names for integers that are defined elsewhere in the code.

```
glLoadName( JAGUAR );
glPushName( BODY );
      glCallList( JagBodyList );
glPopName();
glPushName( FRONT_LEFT_TIRE );
      glPushMatrix();
      glTranslatef( ??, ??, ?? );
      glCallList( TireList );
      glPopMatrix();
glPopName();
glPushName( FRONT_RIGHT_TIRE );
      glPushMatrix();
      glTranslatef( ??, ??, ?? );
      glCallList( TireList );
      glPopMatrix();
glPopName();
```

When a selection occurs, then, the selection buffer will include everything whose display involved the pixel that was chosen, including the automobile as well as the lower-level part, and your program can choose (or allow the user to choose) which of the selections you want to use.

*Picking*

Picking is almost the same operation, logically, as selection, but we present it separately because it uses a different process and allows us to define a concept of "near' and to talk about a way to identify the objects near the selection point. In the picking process, you can define a very small window in the immediate neighborhood of the point where the mouse was clicked, and then you can identify everything that is drawn in that neighborhood. The result is returned in the same selection buffer and can be processed in the same way.

This is done by creating a transformation with the function `gluPickMatrix(...)` that is applied after the projection transformation (that is, defined before the projection; recall the relation

between the sequence in which transformations are identified and the sequence in which they are applied). The full function call is

```
gluPickMatrix(GLdouble x, GLdouble y, GLdouble width, GLdouble height,
              GLint viewport[4])
```

where `x` and `y` are the coordinates of the point picked by the mouse, which is the center of the picking region; the width and height are the size of the picking region in pixels, sometimes called the pick tolerance; and the viewport is the vector of four integers returned by the function call `glGetIntegerv(GL_VIEWPORT, GLint *viewport)`.

The function of this pick matrix is to identify a small region centered at the point where the mouse was clicked and to select anything that is drawn in that region. This returns a standard selection buffer that can then be processed to identify the objects that were picked, as described above.

A code fragment to implement this picking is given below. This corresponds to the point in the code for `doSelect(...)` above labeled "set up the standard viewing model" and "standard perspective viewing":

```
int viewport[4]; /* place to retrieve the viewport numbers */
   ...
dx = glutGet( GLUT_WINDOW_WIDTH );
dy = glutGet( GLUT_WINDOW_HEIGHT );
   ...
glMatrixMode( GL_PROJECTION );
glLoadIdentity();
if( RenderMode == GL_SELECT ) {
   glGetIntegerv( GL_VIEWPORT, viewport );
   gluPickMatrix( (double)Xmouse, (double)(dy - Ymouse),
   PICK_TOL, PICK_TOL, viewport );
}
   ... the call to glOrtho(), glFrustum(), or gluPerspective() goes here
```

*A selection example*

The selection process is pretty well illustrated by some code by a student, Ben Eadington. This code sets up and renders a Bézier spline surface with a set of selectable control points. When an individual control point is selected, that point can be moved and the surface responds to the adjusted set of points. An image from this work is given in Figure 14.2, with one control point selected (shown as being a red cube instead of the default green color).

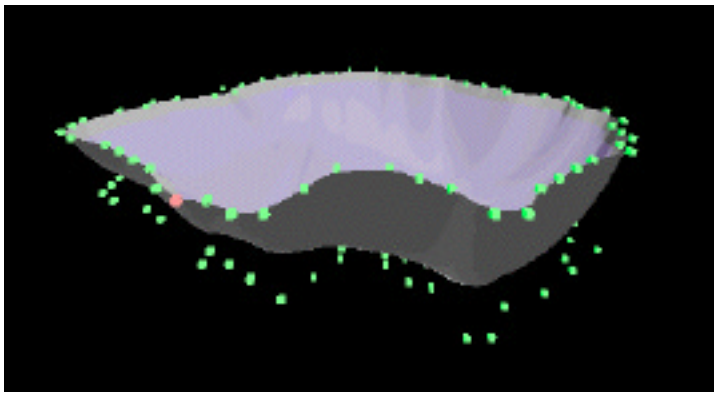


Figure 14.2: a surface with selectable control points and with one selected

Selected code fragments from this project are given below. Here all the data declarations and evaluator work are omitted, as are some standard parts of the functions that are presented, and just the important functions are given with the key points described in these notes. You will be directed to several specific points in the code to illustrate how selection works, described with interspersed text as the functions or code are presented.

In the first few lines you will see the declaration of the global selection buffer that will hold up to 200 values. This is quite large for the problem here, since there are no hierarchical models and no more than a very few control points could ever line up. The actual size would need to be no more than four `GLuints` per control point selected, and probably no more than 10 maximum points would ever line up in this problem. Each individual problem will need a similar analysis.

```
// globals initialization section
#define MAXHITS 200  // number of GLuints in hit records
// data structures for selection process
GLuint selectBuf[MAXHITS];
```

The next point is the mouse callback. This simply catches a mouse-button-down event and calls the DoSelect function, listed and discussed below, to handle the mouse selection. When the hit is handled (including the possibility that there was no hit with the cursor position) the control is passed back to the regular processes with a redisplay.

```
// mouse callback for selection
void Mouse(int button, int state, int mouseX, int mouseY)
{
   if (state == GLUT_DOWN) { // find which object, if any was selected
      hit = DoSelect((GLint) mouseX, (GLint) mouseY);
   }
   glutPostRedisplay(); /* redraw display */
}
```

The control points may be drawn in either `GL_RENDER` or `GL_SELECT` mode, so this function must handle both cases. The only difference is that names must be loaded for each control point, and if any of the points had been hit previously, it must be identified so it can be drawn in red instead of in green. But there is nothing in this function that says what is or is not hit in another mouse click; this is handled in the DoSelect function below.

```
void drawpoints(GLenum mode)
{
   int i, j;
   int name=0;
   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, green);
   // iterate through control point array
   for(i=0; i<GRIDSIZE; i++)
      for(j=0; j<GRIDSIZE; j++) {
         if (mode == GL_SELECT) {
            glLoadName(name); // assign a name to each point
            name++;           // increment name number
         }
         glPushMatrix();
         ... place point in right place with right scaling
         if(hit==i*16+j%16) { // selected point, need to draw it red
            glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, red);
            glutSolidCube(0.25);
            glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, green);
         }
```

```
            else glutSolidCube(0.25);
            glPopMatrix();
         }
   }
```

The only real issue here is to decide what you do and do not need to draw in each of the two rendering modes. Note that the surface is only drawn if the program is in GL_RENDER mode; because nothing in the surface is itself selectable, the only thing that needs to be drawn in GL_SELECT mode is the control points.

```
void render(GLenum mode) {
   ...  do appropriate transformations
   if (mode == GL_RENDER) { // don't render surface if mode is GL_SELECT
      surface(ctrlpts);
      ... some other operations that don't matter here
   }
   if(points) drawpoints(mode); // always render the control points
   ...  pop the transform stack as needed and exit gracefully
}
```

This final function is the real meat of the problem. The display environment is set up (projection and viewing transformations), the `glRenderMode` function sets the rendering mode to `GL_SELECT` and the image is drawn in that mode, the number of hits is returned from the call to the `glRenderMode` function when it returns to `GL_RENDER` mode, the display environment is rebuilt for the next drawing, and the selection buffer is scanned to find the object with the smallest `zmin` value as the selected item. That value is then returned so that the `drawpoints` function will know which control point to display in red and so other functions will know which control point to adjust.

```
GLint DoSelect(GLint x, GLint y)
{
   int i;
   GLint hits, temphit;
   GLuint zval;

   glSelectBuffer(MAXHITS, selectBuf);
   glRenderMode(GL_SELECT);
   glInitNames();
   glPushName(0);

   // set up the viewing model
   ... standard perspective viewing and viewing transformation setup

   render(GL_SELECT);  // draw the scene for selection

   // find the number of hits recorded and reset mode of render
   hits = glRenderMode(GL_RENDER);
   // reset viewing model
   ... standard perspective viewing and viewing transformation setup
   // return the label of the object selected, if any
   if (hits <= 0) return -1;
   else {
      zval =  selectBuf[1];
      temphit = selectBuf[3];
      for (i = 1; i < hits; i++) { // for each hit
         if (selectBuf[4*i+1] < zval) {
            zval = selectBuf[4*i+1];
```

```
                temphit = selectBuf[4*i+3];
            }
        }
    }
    return temphit;
}
```

*A word to the wise...*

This might be a good place to summarize the things we've seen in the discussions and code examples above:

- Define an array of unsigned integers to act as the selection buffer
- Design a mouse event callback that calls a function that does the following:
  - Sets `GL_SELECT` mode and draws selected parts of the image, having loaded names so these parts can be identified when the selection is made
  - when this rendering is completed, returns a selection buffer that can be processed
  - returns to `GL_RENDER` mode.

This design structure is straightforward to understand and can be easily implemented with a little care and planning.

Another point to recognize is that you cannot pick raster characters. For whatever reason, if you draw any raster characters in select mode, OpenGL will always think that the characters were picked no matter where you clicked. If you want to be able to pick a word that is drawn as raster characters, create a rectangle that occupies the space where the raster characters would be, and draw that rectangle in select mode.

# Interpolation and Spline Modeling

*Prerequisites*

A modest understanding of parametric functions of two variables together with an understanding of simple modeling with techniques such as triangle strips.

*Introduction*

In the discussions of mathematical fundamentals at the beginning of these notes, we talked about line segments as linear interpolations of points. Here we introduce other kinds of interpolations of points involving techniques called spline curves and surfaces. The specific spline techniques we will discuss are straightforward but we will limit them to one-dimensional spline curves. Extending these to two dimensions to model surfaces is a bit more complex and we will only cover this in terms of the evaluators that are built into the OpenGL graphics API. In general, spline techniques provide a very broad approach to creating smooth curves that approximate a number of points in a one-dimensional domain (1D interpolation) or smooth surfaces that approximate a number of points in a two-dimensional domain (2D interpolation). This interpolation is usually thought of as a way to develop geometric models, but there are a number of other uses of splines that we will mention later. Graphics APIs such as OpenGL usually provide tools that allow a graphics programmer to create spline interpolations given only the original set of points, called *control points*, that are to be interpolated.

In general, we think of an entire spline curve or spline surface as a single piece of geometry in the scene graph. These curves and surfaces are defined in a single modeling space and usually have a single set of appearance parameters, so in spite of their complexity they are naturally represented by a single shape node that is a leaf in the scene graph.

Interpolations

When we talked about the parametric form for a line segment in the early chapter on mathematical foundations for graphics, we created a correspondence between the unit line segment and an arbitrary line segment and were really interpolating between the two points by creating a line segment between them. If the points are named `P0` and `P1`, this interpolating line segment can be expressed in terms of the parametric form of the segment:

`(1-t)*P0 + t*P1`, for t in `[0.,1.]`

This form is almost trivial to use, and yet it is quite suggestive, because it hints that the set of points that interpolate the two given points can be computed by an expression such as

$f_0$`(t)*P0 +` $f_1$`(t)*P1`

for two fixed functions $f_0$ and $f_1$. This suggests a relationship between points and functions that interpolate them that would allow us to consider the nature of the functions and the kind of interpolations they provide. In this example, we have $f_0$`(t)=(1-t)` and $f_1$`(t)=t`, and there are interesting properties of these functions. We see that $f_0$`(0)=1` and $f_1$`(0)=0`, so at `t=0`, the interpolant value is `P0`, while $f_0$`(1)=0` and $f_1$`(1)=1`, so at `t=1`, the interpolant value is `P1`. This tells us that the interpolation starts at `P0` and ends at `P1`, which we had already found to be a useful property for the interpolating line segment. Note that because each of the interpolating functions is linear in the parameter `t`, the set of interpolating points forms a line.

As we move beyond line segments that interpolate two points, we want to use the term interpolation to mean determining a set of points that approximate the space between a set of given points in the order the points are given. This set of points can include three points, four points, or even more. We assume throughout this discussion that the points are in 3-space, so we will be

creating interpolating curves (and later on, interpolating surfaces) in three dimensions. If you want to do two-dimensional interpolations, simply ignore one of the three coordinates.

Finding a way to interpolate three points `P0`, `P1`, and `P2` is more interesting than interpolating only two points, because one can imagine many ways to do this. However, extending the concept of the parametric line we could consider a quadratic interpolation in t as:

$(1-t)^2*P0 + 2t*(1-t)*P1 + t^2*P2$, for t in [0.,1.]

Here we have three functions $f_0$, $f_1$, and $f_2$ that participate in the interpolation, with $f_0(t)=(1-t)^2$, $f_1(t)=2t*(1-t)$, and $f_2(t)=t^2$. These functions have by now achieved enough importance in our thinking that we will give them a name, and call them the *basis functions* for the interpolation. Further, we will call the points `P0`, `P1`, and `P2` the *control points* for the interpolation (although the formal literature on spline curves calls them *knots* and calls the endpoints of an interpolation *joints*). This particular set of functions have a similar property to the linear basis functions above, with $f_0(0)=1$, $f_1(0)=0$, and $f_2(0)=0$, as well as $f_0(1)=0$, $f_1(1)=0$, and $f_2(1)=1$, giving us a smooth quadratic interpolating function in t that has value `P0` if `t=0` and value `P1` if `t=1`, and that is a linear combination of the three points if `t=.5`. The shape of this interpolating curve is shown in Figure 15.1.

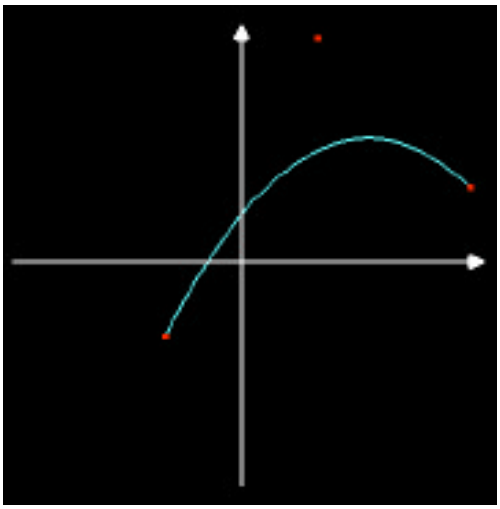


Figure 15.1: a quadratic interpolating curve for three points

The particular set of interpolating polynomials $f_0$, $f_1$, and $f_2$ in the interpolation of three points is suggestive of a general approach in which we would use components which are products of these polynomials and take their coefficients from the geometry we want to interpolate. If we follow this pattern, interpolating four points `P0`, `P1`, `P2`, and `P3` would look like:

$(1-t)^3*P0 + 3t*(1-t)^2*P1 + 3t^2*(1-t)*P2 + t^3*P3$, for t in [0.,1.]

and the shape of the curve this determines is illustrated in Figure 15.2. (We have chosen the first three of these points to be the same as the three points in the quadratic spline above to make it easier to compare the shapes of the curves). In fact, this curve is an expression of the standard Bézier spline function to interpolate four control points, and the four polynomials

$f_0(t)=(1-t)^3$,

$f_1(t)=3t(1-t)^2$,

$f_2(t)=3t^2(1-t)$, and

$f_3(t)=t^3$

are called the cubic *Bernstein basis* for the spline curve.

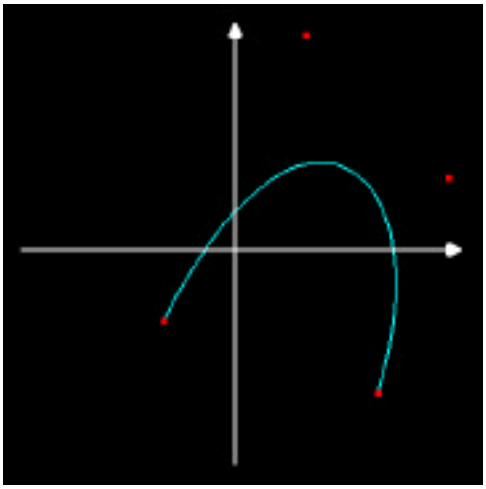


Figure 15.2: interpolating four points with the Bézier spline based on the Bernstein basis functions

When you consider this interpolation, you will note that the interpolating curve goes through the first and last control points (`P0` and `P3`) but does not go through the other two control points. This is because the set of basis functions for this curve behaves the same at the points where `t=0` and `t=1` as we saw in the quadratic spline: $f_0(0)=1$, $f_1(0)=0$, $f_2(0)=0$, and $f_3(0)=0$, as well as $f_0(1)=0$, $f_1(1)=0$, $f_2(1)=0$, and $f_3(1)=1$. You will also note that as the curve goes through the first and last control points, it is moving in the direction from the first to the second control point, and from the third to the fourth control points. Thus the two control points that are not met control the shape of the curve by determining the initial and the ending directions of the curve, and the rest of the shape is determined in order to get the necessary smoothness.

In general, curves that interpolate a given set of points need not go through those points, but the points influence and determine the nature of the curve in other ways. If you need to have the curve actually go through the control points, however, there are spline formulations for which this does happen. The Catmull-Rom cubic spline has the form

$f_0(t)*P0 + f_1(t)*P1 + f_2(t)*P2 + f_3(t)*P3$, for t in `[0.,1.]`

for basis functions

$f_0(t)=(-t^3+2t^2-t)/2$

$f_1(t)=(3t^3-5t^2+2)/2$

$f_2(t)=(-3t^3+4t^2+t)/2$

$f_3(t)=(t^3-t^2)/2$

This interpolating curve has a very different behavior from that of the Bézier curve above, because as shown in Figure 15.3. This is a different kind of interpolating behavior that is the result of a set of basis functions that have $f_0(0)=0$, $f_1(0)=1$, $f_2(0)=0$, and $f_3(0)=0$, as well as $f_0(1)=0$, $f_1(1)=0$, $f_2(1)=1$, and $f_3(1)=0$. This means that the curve interpolates the points P1 and P2 instead of P0 and P3 and actually goes through those two points. Thus the

Catmull-Rom spline curve is useful when you want your interpolated curve to include all the control points, not just some of them.

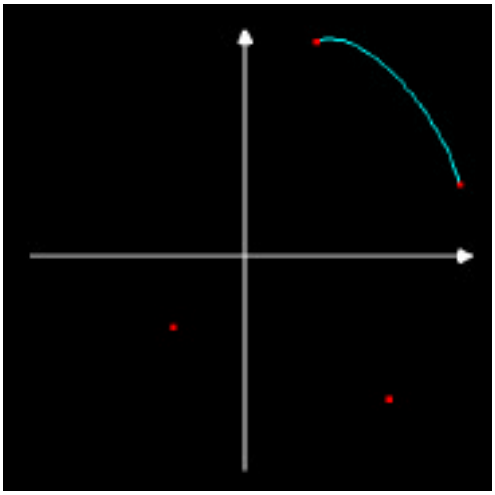


Figure 15.3: interpolating four points with the Catmull-Rom cubic spline

We will not carry the idea of spline curves beyond cubic interpolations, but we want to provide this much detailed background because it can sometimes be handy to manage cubic spline curves ourselves, even though OpenGL provides evaluators that can make spline computations easier and more efficient. Note that if the points we are interpolating lie in 3D space, each of these techniques provides a 3D curve, that is, a function from a line segment to 3D space.

While we have only shown the effect of these interpolations in the smallest possible set of points, it is straightforward to extend the interpolations to larger sets of points. The way we do this will depend on the kind of interpolation that is provided by the particular curve we are working with, however.

In the Bézier curve, we see that the curve meets the first and last control points but not the two intermediate control points. If we simply use the first four control points, then the next three (the last point of the original set plus the next three control points), and so on, then we will have a curve that is continuous, goes through every third control point (first, fourth, seventh, and so on), but that changes direction abruptly at each of the control points it meets. In order to extend these curves so that they progress smoothly along their entire length, we will need to add new control points that maintain the property that the direction into the last control point of a set is the same as the direction out of the first control point of the next set. In order to do this, we need to define new control points between each pair of points whose index is 2N and 2N+1 for N≥1 up to, but not including, the last pair of control points. We can define these new control points as the midpoint between these points, or $(P_{2N}+P_{2N+1})/2$. When we do, we get the following relation between the new and the original control point set:

```
original:   P0  P1  P2      P3  P4      P5  P6  P7
new:        P0  P1  P2  Q0  P3  P4  Q1  P5  P6  P7
```

where each point Q represents a new point calculated as an average of the two on each side of it, as above. Then the computations would use the following sequences of points: `P0–P1–P2–Q0`; `Q0–P3–P4–Q1`; and `Q1–P5–P6–P7`. Note that we must have an even number of control points for a Bézier curve, that we only need to extend the original control points if we have at least six control points, and that we always have three of the original points participating in each of the first and last segments of the curve.

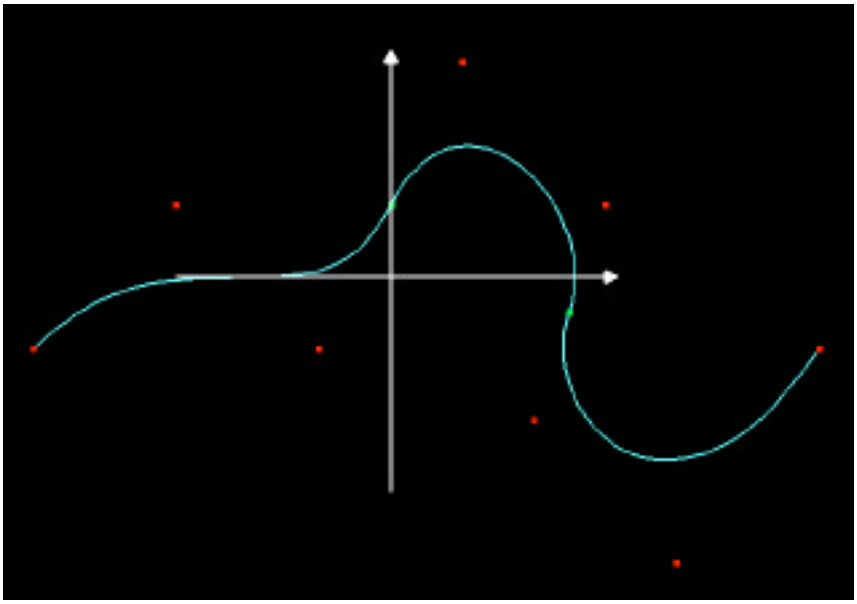


Figure 15.4: extending the Bézier curve by adding intermediate control points, shown in green

For the Catmull-Rom cubic spline, the fact that the interpolating curve only connects the control points P1 and P2 gives us a different kind of approach to extending the curve. However, it also gives us a challenge in starting the curve, because neither the starting control point P0 nor the ending control point P3 is not included in the curve that interpolates P0-P3. Hence we will need to think of the overall interpolation problem in three parts: the first segment, the intermediate segments, and the last segment.

For the first segment, the answer is simple: repeat the starting point twice. This gives us a first set of control points consisting of P0, P0, P1, and P2, and the first piece of the curve will then interpolate P0 and P1 as the middle points of these four. In the same way, to end the curve we would repeat the ending point, giving us the four control points P1, P2, P3, and P3, so the curve would interpolate the middle points, P2 and P3. If we only consider the first four control points and add this technique, we see the three-segment interpolation of the points shown in the left-hand image of Figure 15.5.

If we have a larger set of control points, and if we wish to extend the curve to cover the total set of points, we can consider a "sliding set" of control points that starts with P0, P1, P2, and P3 and, as we move along, includes the last three control points from the previous segment as the first three of the next set and adds the next control point as the last point of the set of four points. That is, the second set of points would be P1, P2, P3, and P4, and the one after that P2, P3, P4, and P5, and so on. This kind of sliding set is simple to implement (just take an array of four points, move each one down by one index so P[1] becomes P[0], P[2] becomes P[1], P[3] becomes P[2], and the new point becomes P[3]. The sequence of points used for the individual segments of the curve are then `P0–P0–P1–P2`; `P0–P1–P2–P3`; `P1–P2–P3–P4`; `P2–P3–P4–P5`; `P3–P4–P5–P6`; `P4–P5–P6–P7`; `P5–P6–P7–P8`; and `P6–P7–P8–P8`. The curve that results when we extend the computation across a larger set of control points is shown as the right-hand image of Figure 15.5, where we have taken the same setof control points that we used for the extended Bézier spline example.

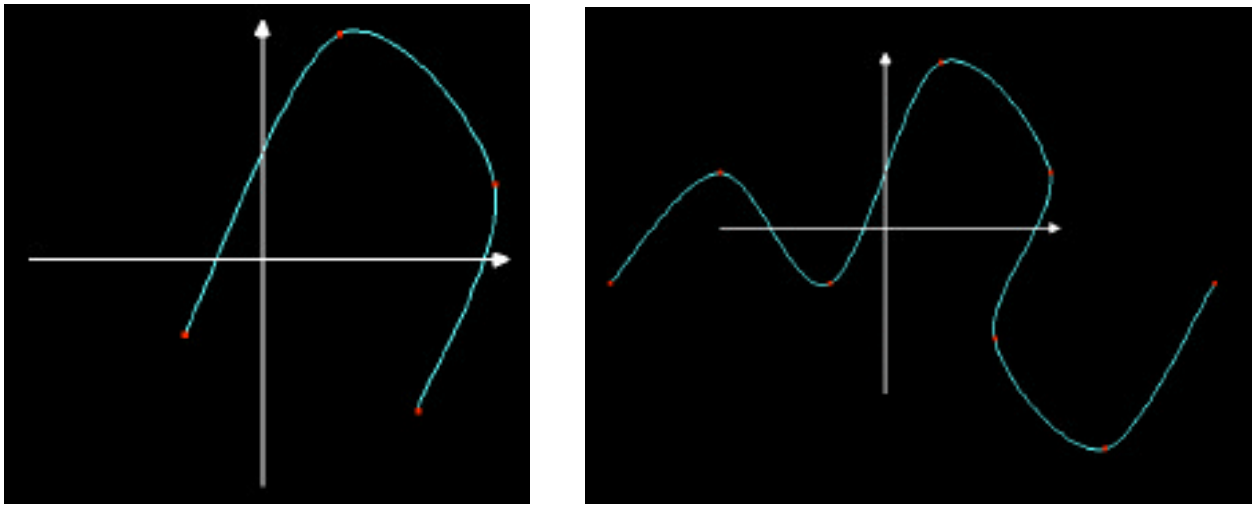


Figure 15.5: extending the Catmull-Rom curve by including the endpoints of the set (left) and by stepping along the extended set of control points (right)

*Interpolations in OpenGL*

In OpenGL, the spline capability is provided by techniques called *evaluators*, functions that take a set of control points and produce another set of points that interpolate the original control points. This allows you to model curves and surfaces by doing only the work to set the control points and set up the evaluator, and then to get much more detailed curves and surfaces as a result. There is an excellent example of spline surfaces in the Eadington example for selecting and manipulating control points in the chapter of these notes on object selection.

There are two kinds of evaluators available to you. If you want to interpolate points to produce one-parameter information (that is, curves or any other data with only one degree of freedom; think 1D textures as well as geometric curves), you can use 1D evaluators. If you want to interpolate points in a 2D array to produce two-parameter information ( that is, surfaces or any other data with two degrees of freedom; think 2D textures as well as geometric curves) you can use 2D evaluators. Both are straightforward and allow you to choose how much detail you want in the actual display of the information.

In Figures 15.6 and 15.7 below we see several images that illustrate the use of evaluators to define geometry in OpenGL. Figure 15.6 shows two views of a 1D evaluator that is used to define a curve in space showing the set of 30 control points as well as additional computed control points for smoothness; Figure 15.7 shows two uses of a 2D evaluator to define surfaces, with the top row showing a surface defined by a 4x4 set of control points and the bottom image showing a surface defined by a 16x16 set of control points with additional smoothness points not shown. These images and the techniques for creating smooth curves will be discussed further below, and some of the code that creates these is given in the Examples section.

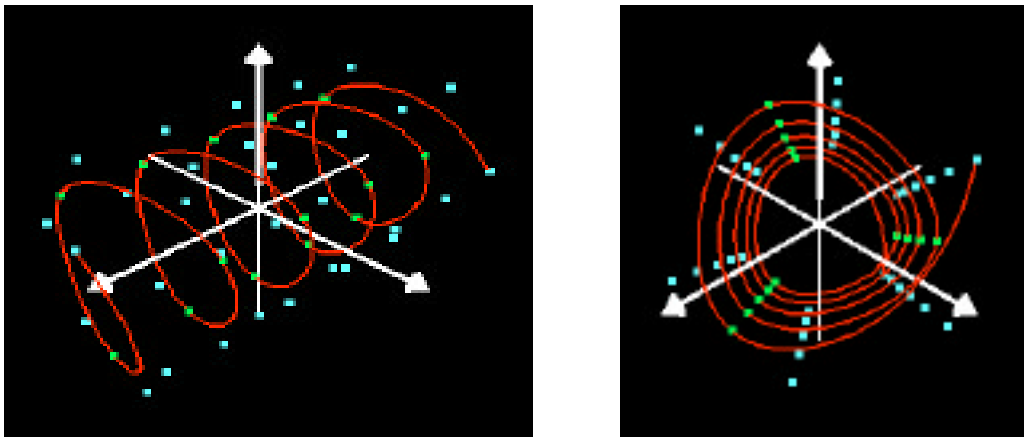


Figure 15.6: a spline curve defined via a 1D evaluator, shown from a point of view with x = y = z (left) and rotated to show the relationship between control points and the curve shape (right) The cyan control points are the originals; the green control points are added as discussed above.

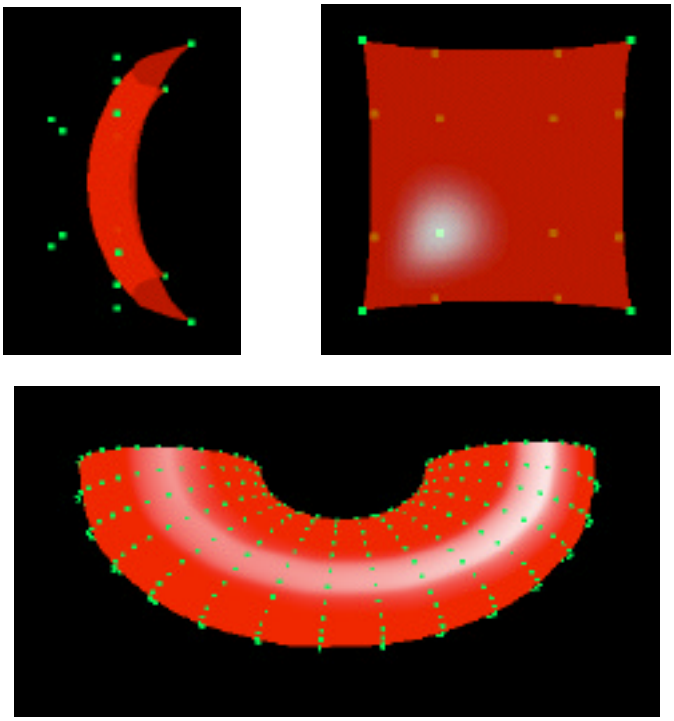


Figure 15.7: spline surfaces defined via a 2D evaluator. At top, in two views, a single patch defined by four control points; at bottom, a larger surface defined by extending the 16x16 set of control points with interpolated points as defined below

The spline surface in the top row of Figure 15.7 has only a 0.7 alpha value so the control points and other parts of the surface can be seen behind the primary surface of the patch. In this example, note the relation between the control points and the actual surface; only the four corner points actually meet the surface, while all the others lie off the surface and act only to influence the shape of the patch. Note also that the entire patch lies within the convex hull of the control points. The specular highlight on the surface should also help you see the shape of the patch from the lighting. In the larger surface at the bottom of Figure 15.7, note how the surface extends smoothly between the different sets of control points.

*Definitions*

As you see in Figures 15.6 and 15.7, an OpenGL evaluator working on an array of four control points (1D) or 4x4 control points (2D) actually fits the extreme points of the control point set but does not go through any of the other points. As the evaluator comes up to these extreme control points, the tangent to the curve becomes parallel to the line segment from the extreme point to the adjacent control point, as shown in Figure 15.8 below, and the speed with which this happens is determined by the distance between the extreme and adjacent control points.

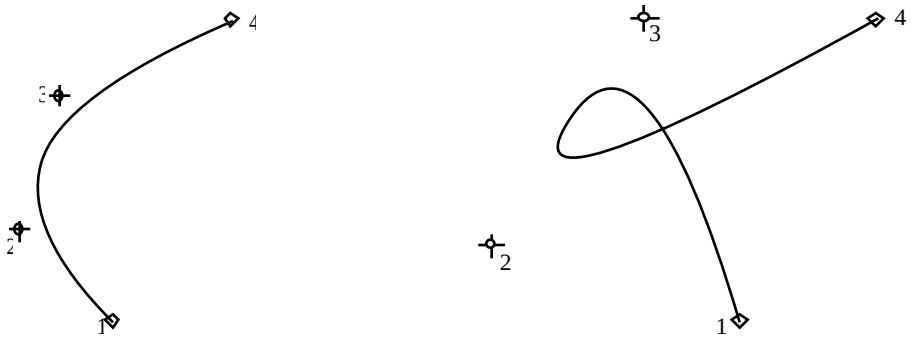



Figure 15.8: two spline curves that illustrate the shape of the curve as it goes through an extreme control point

To control the shape of an extended spline curve, you need to arrange the control points so that the direction and distance from a control point to the adjacent control points are the same. This can be accomplished by adding new control points between appropriate pairs of the original control points as indicated in the spline curve figure above. This will move the curve from the first extreme point to the first added point, from the first added point smoothly to the second added point, from the second added point smoothly to the third added point, and so on to moving smoothly through the last added point to the last extreme point.

This construction and relationship is indicated by the green (added) control points in the first figure in this section. Review that figure and note again how there is one added point after each two original points, excepting the first and last points; that the added points bisect the line segment between the two points they interpolate; and that the curve actually only meets the added points, not the original points, again excepting the two end points. If we were to define an interactive program to allow a user to manipulate control points, we would only give the user access to the original control points; the added points are not part of the definition but only of the implementation of a smooth surface.

Similarly, one can define added control points in the control mesh for a 2D evaluator, creating a richer set of patches with the transition from one patch to another following the same principle of equal length and same direction in the line segments coming to the edge of one patch and going from the edge of the other. This allows you to achieve a surface that moves smoothly from one patch to the next. Key points of this code are included in the example section below, but it does take some effort to manage all the cases that depend on the location of a particular patch in the surface. The example code in the file `fullSurface.c` included with this material will show you these details.

So how does this all work? A cubic spline curve is determined by a cubic polynomial in a parametric variable `u` as indicated by the left-hand equation in (1) below, with the single parameter `u` taking values between `0` and `1`. The four coefficients $a_i$ can be determined by

knowing four constraints on the curve. These are provided by the four control points needed to determine a single segment of a cubic spline curve. We saw ways that these four values could be represented in terms of the values of four basis polynomials, and an OpenGL 1D evaluator computes those four coefficients based on the Bézier curve definition and, as needed, evaluates the resulting polynomial to generate a point on the curve or the curve itself. A bicubic spline surface is determined by a bicubic polynomial in parametric variables `u` and `v` as indicated by the right-hand equation in (1) below, with both parameters taking values between 0 and 1. This requires computing the 16 coefficients $a_{ij}$ which can be done by using the 16 control points that define a single bicubic spline patch. Again, an OpenGL 2D evaluator takes the control points, determines those 16 coefficients based on the basis functions from the Bézier process, and evaluates the function as you specify to create your surface model.

| | | |
|---|---|---|
| (1) | $\sum_{i=0}^{3} a_i u^i$ | $\sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij} u^i v^j$ |

*Some examples*

Spline curves: the setup to generate curves is given in some detail below. This involves defining a set of control points for the evaluator to use, enabling the evaluator for your target data type, defining overall control points for the curve, stepping through the overall control points to build four-tuples of segment control points, and then invoking the evaluator to draw the actual curve. This code produced the figures shown in the figure above on spline curves. A few details have been omitted in the code below, but they are all in the sample code `splineCurve.c` that is included with this module. Note that this code returns the points on the curve using the `glEvalCoord1f(...)` function instead of the `glVertex*(...)` function within a `glBegin(...)` ... `glEnd()` pair; this is different from the more automatic approach of the 2D patch example that follows it.

Probably the key point in this sample code is the way the four-tuples of segment control points have been managed. The original points would not have given smooth curves, so as discussed above, new points were defined that interpolated some of the original points to make the transition from one segment to the other continuous and smooth.

```
   glEnable(GL_MAP1_VERTEX_3)

void makeCurve( void )
{
...
   for (i=0; i<CURVE_SIZE; i++) {
      ctrlpts[i][0]= RAD*cos(INITANGLE + i*STEPANGLE);
      ctrlpts[i][1]= RAD*sin(INITANGLE + i*STEPANGLE);
      ctrlpts[i][2]= -4.0 + i * 0.25;
   }
}

void curve(void) {
#define LAST_STEP (CURVE_SIZE/2)-1
#define NPTS 30

   int step, i, j;

   makeCurve(); // calculate the control points for the entire curve
   // copy/compute points from  ctrlpts  to  segpts to define each segment
```

```
      // of the curve.  First/last cases are different from middle cases...
      for ( step = 0; step < LAST_STEP; step++ ) {
         if (step==0) { // first case
            for (j=0; j<3; j++) {
               segpts[0][j]=ctrlpts[0][j];
               segpts[1][j]=ctrlpts[1][j];
               segpts[2][j]=ctrlpts[2][j];
               segpts[3][j]=(ctrlpts[2][j]+ctrlpts[3][j])/2.0;
            }
         }
         else if (step==LAST_STEP-1) { // last case
            for (j=0; j<3; j++) {
               segpts[0][j]=(ctrlpts[CURVE_SIZE-4][j]
                             +ctrlpts[CURVE_SIZE-3][j])/2.0;
               segpts[1][j]=ctrlpts[CURVE_SIZE-3][j];
               segpts[2][j]=ctrlpts[CURVE_SIZE-2][j];
               segpts[3][j]=ctrlpts[CURVE_SIZE-1][j];
            }
         }
         else for (j=0; j<3; j++) { // general case
            segpts[0][j]=(ctrlpts[2*step][j]+ctrlpts[2*step+1][j])/2.0;
            segpts[1][j]=ctrlpts[2*step+1][j];
            segpts[2][j]=ctrlpts[2*step+2][j];
            segpts[3][j]=(ctrlpts[2*step+2][j]+ctrlpts[2*step+3][j])/2.0;
         }

         // define the evaluator
         glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &segpts[0][0]);
         glBegin(GL_LINE_STRIP);
         for (i=0; i<=NPTS; i++)
            glEvalCoord1f( (GLfloat)i/(GLfloat)NPTS );
         glEnd();
         ...
      }
   }
```

Spline surfaces: we have two examples, the first showing drawing a simple patch (surface based on a 4x4 grid of control points) and the second showing drawing of a larger surface with more control points. Below is some simple code to generate a surface given a 4x4 array of points for a single patch, as shown in the top row of the second figure above. This code initializes a 4x4 array of points, enables auto normals (available through the `glEvalMesh(...)` function) and identifies the target of the evaluator, and carries out the evaluator operations. The data for the patch control points is deliberately over-simplified so you can see this easily, but in general the patch points act in a parametric way that is quite distinct from the indices, as is shown in the general surface code.

```
   point3 patch[4][4] = {{{-2.,-2.,0.},{-2.,-1.,1.},{-2.,1.,1.},{-2.,2.,0.}},
                         {{-1.,-2.,1.},{-1.,-1.,2.},{-1.,1.,2.},{-1.,2.,1.}},
                         {{1.,-2.,1.},{1.,-1.,2.},{1.,1.,2.},{1.,2.,1.}},
                         {{2.,-2.,0.},{2.,-1.,1.},{2.,1.,1.},{2.,2.,0.}}};

   void myinit(void)
   {
      ...
      glEnable(GL_AUTO_NORMAL);
      glEnable(GL_MAP2_VERTEX_3);
   }
```

```
void doPatch(void)
{
// draws a patch defined by a 4 x 4 array of points
#define NUM 20 //

   glMaterialfv(...);  // whatever material definitions are needed

   glMap2f(GL_MAP2_VERTEX_3,0.0,1.0,3,4,0.0,1.0,12,4,&patch[0][0][0]);
   glMapGrid2f(NUM, 0.0, 1.0, NUM, 0.0, 1.0);
   glEvalMesh2(GL_FILL, 0, NUM, 0, NUM);
}
```

The considerations for creating a complete surface with a 2D evaluator is similar to that for creating a curve with a 1D evaluator. You need to create a set of control points, to define and enable an appropriate 2D evaluator, to generate patches from the control points, and to draw the individual patches. These are covered in the sample code below.

The sample code below has two parts. The first is a function that generates a 2D set of control points procedurally; this differs from the manual definition of the points in the patch example above or in the pool example of the selection section. This kind of procedural control point generation is a useful tool for procedural surface generation. The second is a fragment from the section of code that generates a patch from the control points, illustrating how the new intermediate points between control points are built. Note that these intermediate points all have indices 0 or 3 for their locations in the patch array because they are the boundary points in the patch; the interior points are always the original control points. Drawing the actual patch is handled in just the same way as it is handled for the patch example, so it is omitted here.

```
// control point array for pool surface
point3 ctrlpts[GRIDSIZE][GRIDSIZE];

void genPoints(void)
{
#define PI 3.14159
#define R1 6.0
#define R2 3.0
   int i, j;
   GLfloat alpha, beta, step;

   alpha = -PI;
   step = PI/(GLfloat)(GRIDSIZE-1);
   for (i=0; i<GRIDSIZE; i++) {
      beta = -PI;
      for (j=0; j<GRIDSIZE; j++) {
         ctrlpts[i][j][0] = (R1 + R2*cos(beta))*cos(alpha);
         ctrlpts[i][j][1] = (R1 + R2*cos(beta))*sin(alpha);
         ctrlpts[i][j][2] = R2*sin(beta);
         beta -= step;
      }
      alpha += step;
   }
}

void surface(point3 ctrlpts[GRIDSIZE][GRIDSIZE])
{
...
      ...{ // general case (internal patch)
```

```
            for(i=1; i<3; i++)
               for(j=1; j<3; j++)
                  for(k=0; k<3; k++)
                     patch[i][j][k]=ctrlpts[2*xstep+i][2*ystep+j][k];
            for(i=1; i<3; i++)
               for(k=0; k<3; k++) {
                  patch[i][0][k]=(ctrlpts[2*xstep+i][2*ystep][k]
                                  +ctrlpts[2*xstep+i][2*ystep+1][k])/2.0;
                  patch[i][3][k]=(ctrlpts[2*xstep+i][2*ystep+2][k]
                                  +ctrlpts[2*xstep+i][2*ystep+3][k])/2.0;
                  patch[0][i][k]=(ctrlpts[2*xstep][2*ystep+i][k]
                                  +ctrlpts[2*xstep+1][2*ystep+i][k])/2.0;
                  patch[3][i][k]=(ctrlpts[2*xstep+2][2*ystep+i][k]
                                  +ctrlpts[2*xstep+3][2*ystep+i][k])/2.0;
               }
            for(k=0; k<3; k++) {
               patch[0][0][k]=(ctrlpts[2*xstep][2*ystep][k]
                              +ctrlpts[2*xstep+1][2*ystep][k]
                              +ctrlpts[2*xstep][2*ystep+1][k]
                              +ctrlpts[2*xstep+1][2*ystep+1][k])/4.0;
               patch[3][0][k]=(ctrlpts[2*xstep+2][2*ystep][k]
                              +ctrlpts[2*xstep+3][2*ystep][k]
                              +ctrlpts[2*xstep+2][2*ystep+1][k]
                              +ctrlpts[2*xstep+3][2*ystep+1][k])/4.0;
               patch[0][3][k]=(ctrlpts[2*xstep][2*ystep+2][k]
                              +ctrlpts[2*xstep+1][2*ystep+2][k]
                              +ctrlpts[2*xstep][2*ystep+3][k]
                              +ctrlpts[2*xstep+1][2*ystep+3][k])/4.0;
               patch[3][3][k]=(ctrlpts[2*xstep+2][2*ystep+2][k]
                              +ctrlpts[2*xstep+3][2*ystep+2][k]
                              +ctrlpts[2*xstep+2][2*ystep+3][k]
                              +ctrlpts[2*xstep+3][2*ystep+3][k])/4.0;
            }
         }

   ...
   }
```

*A word to the wise...*

Spline techniques may also be used for much more than simply modeling. Using them, you can generate smoothly changing sets of colors, or of normals, or of texture coordinates — or probably just about any other kind of data that one could interpolate. There aren't built-in functions that allow you to apply these points automatically as there are for creating curves and surfaces, however. For these you will need to manage the parametric functions yourself. To do this, you need to define each point in the `(u,v)` parameter space for which you need a value and get the actual interpolated points from the evaluator using the functions `glEvalCoord1f(u)` or `glEvalCoord2f(u,v)`, and then use these points in the same way you would use any points you had defined in another way. These points, then, may represent colors, or normals, or texture coordinates, depending on what you need to create your image.

Another example of spline use is in animation, where you can get a smooth curve for your eyepoint to follow by using splines. As your eyepoint moves, however, you also need to deal with the other issues in defining a view. The up vector is fairly straightforward; for simple animations, it is probably enough to keep the up vector constant. The center of view is more of a challenge, however, because it has to move to keep the motion realistic. The suggested approach is to keep

three points from the spline curve: the previous point, the current point, and the next point, and to use the previous and next points to set the direction of view; the viewpoint is then a point at a fixed distance from the current point in the direction set by the previous and next points. This should provide a reasonably good motion and viewing setup.

This discussion has only covered cubic and bicubic splines, because these are readily provided by OpenGL evaluators. OpenGL also has the capability of providing NURBS (non-uniform rational B-splines) but these are beyond the scope of this discussion. Other applications may find it more appropriate to use other kinds of splines, and there are many kinds of spline curves and surfaces available; the interested reader is encouraged to look into this subject further.

# Hardcopy

*Prerequisites*

An understanding of the nature of color and visual communication, and an appreciation of what makes an effective image.

*Introduction*

You have worked hard to analyze a problem and have developed really good models and great images or animations that communicate your solution to that problem, but those images and animations only run on your computer and are presented on your screen. Now you need to take that work and present it to a larger audience, and you don't want to lose the control over the quality of your work when you take it to a medium beyond the screen. In this chapter we talk about the issues you will face when you do this.

*Definitions*

Hardcopy images are images that can be taken away from the computer and communicated to your audience without any computer mediation. There are several ways this can be done, but the basic idea is that any kind of medium that can carry an image is a candidate for hardcopy. Each of these media has its own issues in terms of its capability and how you must prepare your images for the medium. In this chapter we will discuss some of the more common hardcopy media and give you an idea of what you must do to use each effectively.

In general, we think of hardcopy as something we output from the computer to some sort of output device. That device may be actually attached to the computer, such as a printer or a film recorder, or it may be a devide to which we communicate data by network, disk, or CD-ROM. So part of the discussion of graphics hardcopy will include a description of the way data must be organized in order to communicate with external production processes.

Print: One version of printed hardcopy is created by a standard color printer that you can use with your computer system. Because these printers put color on paper, they are usually CMYK devices, as we talked about in the color chapter, but the printer driver will usually handle the conversion from RGB to CMYK for you. In order of increasing print quality, the technologies for color output are

- inkjet, where small dots of colored ink are shot onto paper and you have to deal with dot spread and over-wetting paper as the ink is absorbed into the paper,
- wax transfer, where wax sticks of the appropriate colors are melted and a thin film of wax is put onto the paper, and
- dye sublimation, where sheets of dye-saturated material are used to transfer dyes to the paper.

These devices have various levels of resolution, but in general each has resolution somewhat less than a computer screen. All these technologies can also be used to produce overhead foils for those times when you have only an overhead projector to present your work to your audience.

Print can also mean producing documents by standard printing presses. This kind of print has some remarkably complex issues in reproducing color images. Because print is a transmissive or subtractive medium, you must convert your original RGB work to CMYK color before beginning to develop printed materials. You will also need to work with printing processes, so someone must make plates of your work for the press, and this involves creating separations as shown in Figure 16.1 (which was also shown in the chapter on color). Plate separations are created by masking the individual C, M, Y, and K color rasters with a screen that is laid across the image at a different angle for each color; the resulting print allows each of the color inks to lay on the paper

with minimal interference with the other colors. A screen is shown in Figure 16.2, which is enlarged so much that you can see the angles of the screens for the C, M, Y, and K components. (You should look at a color image in print to see the tell-tale rosettes of standard separations.) There are other separation technologies, called stochastic separations, that dither individual dots of ink to provide more colored ink on the page and sharper images without interference, but these have not caught on with much of the printing world. Creating separations for color-critical images is something of an art form, and it is strongly suggested that you insist on high-quality color proofs of your work. You must also plan for a lower resolution in print than in your original image because the technologies of platemaking and presses do not allow presses to provide a very high resolution on paper.

Figure 16.1: separations for color printing

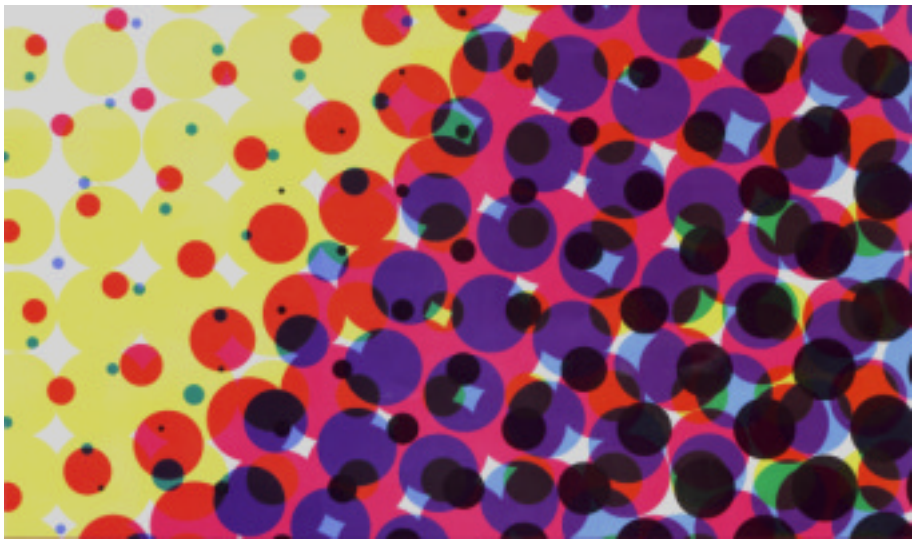


Figure 16.2: C, M, Y, and K screens in a color image, greatly enlarged

Film: sometimes you want to present the highest-quality images you can to an audience: the most saturated colors and the highest resolution. Sometimes you want to be sure you can present your work without relying on computer projection technology. In both cases, you want to consider standard photographic images from digital film recorders. These are devices that generate images using a very high-quality grayscale monitor, a color wheel, and a camera body and that work with whatever kind of film you want (usually slide film: Kodachrome, Ektachrome, or the like).

A film recorder is organized as shown in Figure 16.3. The grayscale monitor generates the images for each color separately, and that image is photographed through a color wheel that provides the color for the image. Because a grayscale monitor does not need to have a shadow mask to separate the phosphors for different colors, and because the monitor can be designed to have a long neck and small screen to allow for extremely tight control of the electron beam, it can have extraordinary resolution; 8K line resolution is pretty standard and you can get film recorders with up to 32K lines. This allows you to generate your image at resolutions that would be impossible on the screen.

Film is much less of a problem than print, because you can work directly with the image and do not need to deal with separations, and you work with the usual RGB color model. Recall that slides produce their image by having light projected through them, so they behave as if they were an emissive medium like the screen. Your only issue is to deal with the resolution of the camera or to accept the interpolations the film recorder will use if you don't provide enough resolution.

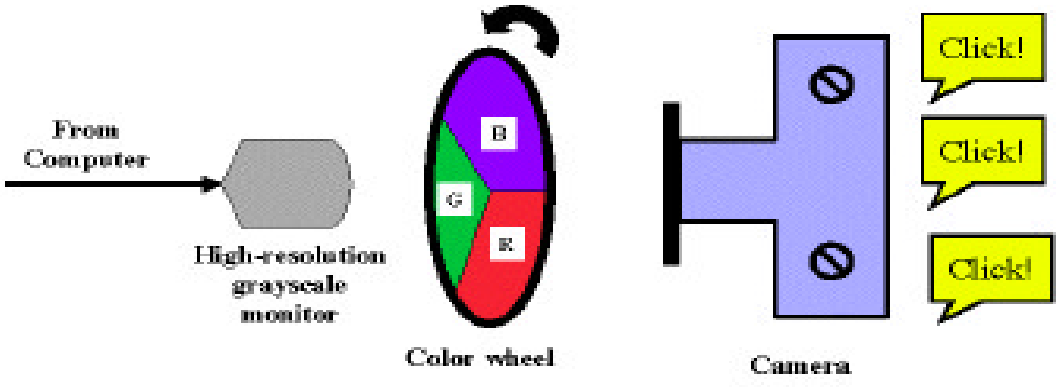



Figure 16.3: schematic of digital film recorder

Video: video is a very important medium for your work, because it is the only medium available to show the motion that is so important to communicate many of your ideas. At the same time, it can be one of the most limited media available to you — at least until video leaves the first half of the 20th century and really comes into the 21st century. We will focus on NTSC video here, but there are similar issues for PAL or SECAM video, and if you are reading this in one of the areas where PAL or SECAM are the standards, you should check to see how much the comments here apply to you.

There are some important issues in dealing with video. The first is resolution: the resolution of NTSC video is much lower than even a minimal computer resolution. NTSC standards call for 525 interlaced horizontal scan lines, of which 480 are visible, so your planned resolution should be about 640 by 480. However, many television sets have adjustment issues so you should not ever work right against the edge of this space. The interlaced scan means that only half of the horizontal lines will be displayed every 1/30 second, so you should avoid using single-pixel horizontal

elements to avoid flicker; many television sets have poorly-converged color, so you should also avoid using single-pixel vertical elements to they will not bleed into each other. In fact, you will have the best results for video if you design your work assuming that you have only half the resolution noted above.

A second issue in video is the color gamut. Instead of being composed of RGB components, the NTSC television standard is made up of significant compromises to acount for limited broadcasting bandwidth and the need to be compatible with black-and-white television (the NTSC standard dates from the late 1930s, before the wide-spread advent of color television or the advent of modern electronics and other technology). The NTSC color standard is a three-component model called the YIQ standard, but the three components are entirely focused on video issues. The Y component is the luminance (or brightness), and it gets most of the bandwidth of the signal. The I component is an orange-to-blue component, and it gets a little more than 1/3 of bandwidth of the Y component. The Q component is a purple-to-green component, and it gets a little more than 1/3 of the I component. The best color you can get in video always seems to be under-saturated, because that is part of the compromise of dealing with the technology available. To be more precise, the following table shows the bandwidth and the horizontal resolution for each of the components of the video image:

| Component | Bandwidth | Resolution/scanline |
|---|---|---|
| Y | 4.0 Mhz | 267 |
| I | 1.5 Mhz | 96 |
| Q | 0.6 Mhz | 35 |

In order to get the best possible horizontal resolution from your image, then, you need to be sure that the elements that vary across the line have differing luminance, and you should focus more on the orange-to-blue component than on the purple-to-green component. If you want to understand how your colors vary in YIQ, the following conversion matrix should help you evaluate your image for video:

$$\begin{vmatrix} Y \\ I \\ Q \end{vmatrix} = \begin{vmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.528 & 0.311 \end{vmatrix} \begin{vmatrix} R \\ G \\ B \end{vmatrix}$$

The question of video is complicated by the various digital video formats, such as QuickTime and MPEG, that require computer mediation to be played back. Digital video is RGB, so it does not have many of the problems of NTSC until it is actually played on a television screen, and there are television sets that will handle increasingly-high quality video. In fact, MPEG II is the video standard for DVD, and there are self-contained DVD players, so this provides one alternative to doing your own conversion to NTSC.

In the longer term, television will be moving to native digital formats and the HDTV standards will support direct RGB color and higher-resolution, non-interlaced images, so we look forward to this discussion becoming antiquated. For the time being, however, you may need to put up with creating images that will make your colleagues ask, “That looks terrible! Why are you doing that?” Figure 16.4 is a photograph of a video image that shows the problems with color and resolution. If they understand that you’re going to video, however, they’ll understand.

Figure 16.4: screen capture of an NTSC video image.
Note the relatively low resolution and limited colors.

Creating a digital video from an animation is straightforward with the right tools. If you are creating an animation, for example, you may generate a single frame of the animation in a window on the screen and then use a function such as the OpenGL `glReadPixels(...)` function to save the contents of the window to an array, which can then be written to an image file, possibly with a name that represents the frame number of that image in the overall animation. After you have completed running your set of animation segments, and have created the set of individual frames, you may import them into any of a number of tools that will allow you to save them as a digital movie in QuickTime or MPEG format. Many of these tools will also allow you to add a sound track, do transitions from one animation sequence to another, or add subtitles or other text information to the movie.

3D object prototyping: there are times when having an image of an object simply isn't enough, when you need to be able to run your fingers over the object to understand its shape, when you need to hold two objects together to see how they fit, or when you need to see how something is shaped so you can see how it could be manufactured. This kind of 3D object prototyping is sometimes called "3D printing" and is done by special tools. You can view the resulting object as a prototype of a later manufactured object, or you can view it as a solid representation of your graphic image. Figure 16.5 shows photographs of the (3,4)-torus as created by several of these 3D printing techniques, as noted in the figure caption. The contact information for each of the companies whose products were used for these hardcopies is given at the end of the chapter. There are, of course, other older technologies for 3D hardcopy that involve creating a tool path for a cutting tool in a numerical milling machine and similar techniques, but these go beyond the prototyping level.

There are several kinds of technologies for creating these prototype objects, but most work by building up a solid model in layers, with each layer controlled by a computation of the boundary of the solid at each horizontal cutting plane. These boundaries are computed from information on the faces that bound the object as represented in information presented to the production device. The current technologies for doing such production include the following:

- The Helisys LOM (Laminated Object Manufacturing) system lays down single sheets of adhesive-backed paper and cuts the outline of each layer with a laser. The portion of the sheets that is outside the object is scored so that the scrap to be removed (carefully!) with simple tools, and the final object is lacquered to make it stronger. It is not possible to build objects that have thin openings to the outside because the scrap cannot be removed from the internal volumes. LOM objects are vulnerable to damage on edges that are at the very top or bottom of the layers, but in general they are quite sturdy. Figure 16.5a shows the torus

created with the LOM system; note the rectangular grid on the surface made by the edges of the scrap scoring, the moiré pattern formed by the burned edges of the individual layers of paper in the object, and the shiny surface made when the object is lacquered.

Figure 16.5a: the torus created by the LOM system

- The Z-Corp Z-402 system lays down a thin layer of starch powder and puts a liquid binder (in the most recent release, the binder can have several different colors) on the part of the powder that is to be retained by the layer. The resulting object is quite fragile but is treated with a penetrating liquid such as liquid wax or a SuperGlue to stabilize it. Objects built with a wax treatment are somewhat fragile, but objects built with SuperGlue are very strong. Because the parts of the original object that are not treated with binder are a simple powder, it is possible to create objects with small openings and internal voids with this technology. Figure 16.5b shows the torus created with the LOM system; note the very matte surface that is created by the basic powder composition of the object.

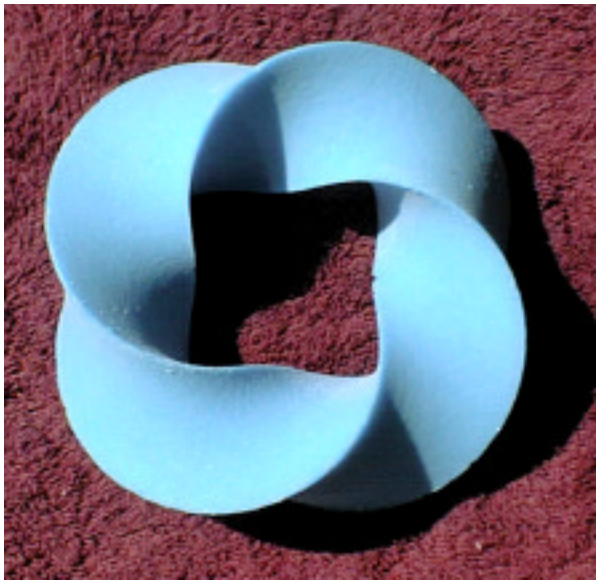


Figure 16.5b: the torus created by the Z-Corp system

- The 3D Systems ThermaJet system builds a part by injecting a layer of liquid wax for each layer of the object. Such parts must include a support structure for any regions that overhang the object's base or another part of the object, and this support can either be designed when the object is designed or provided automatically by the ThermaJet. Because the object is made of wax it is not stable in heat and is not very strong. The need for a support structure makes it difficult to include voids with small openings to the outside. Also because of the support structure, the bottom part of an object needs to be finished by removing the structure and smoothing the surface from which this was removed. Figure 16.5c shows the torus as created by the ThermaJet system; note the slightly shiny surface of the wax in the object.

Figure 16.5c: the torus created by the 3D Systems ThermaJet system

- The 3D Systems stereolithography system creates an object by building up thin layers of a polymer liquid and hardening the part of that layer that is to be retained by scanning it with a laser beam. As was the case with the ThermaJet system, this requires a very solid support structure for parts of the object, particularly because there is a small contraction of the polymer material when it is treated with the laser. The support structure must be removed from the object after it is completed, so there is some finishing work needed to get fully-developed surfaces. The polymer liquid can readily be drained from any interior spaces if there is an opening to the outside, so this technology handles this kind of object quite well. The polymer is very strong after it is hardened in a treatment after the shaping is complete, so objects created with this technology are very sturdy. Figure 16.5d shows the torus as created by the stereolithography system.

Figure 16.5d: the torus created by the 3D Systems stereolithography system

One thing all these 3d prototyping technologies have in common is that they all take data files in the STL file format in order to control their operations. This is a very simple file format that is easy to generate from your graphics program. The STL file for the (3.4)-torus is 2,459,957 bytes long and the first and last portions of the file are shown below. The file is organized by facets, and with each facet you have an optional normal and a list of the vertices of the facet; if you create your model in a way that will let you generate the explicit coordinates of your vertices, you can simply write the contents of the STL file instead of calling the graphics output functions.

```
solid
  facet normal -0.055466 0.024069 0.000000
    outer loop
      vertex -5.000010 -0.000013 -1.732045
      vertex -5.069491 -0.160129 -1.688424
      vertex -5.000009 -0.000013 -1.385635
    endloop
  endfacet
  facet normal -0.055277 0.019635 0.002301
    outer loop
      vertex -5.069491 -0.160129 -1.688424
      vertex -5.000009 -0.000013 -1.385635
      vertex -5.054917 -0.159669 -1.342321
    endloop
  endfacet
  ...
  facet normal -0.055466 -0.024069 0.000000
    outer loop
      vertex -5.000009 0.000014 1.385635
      vertex -5.069491 0.160130 1.688424
      vertex -5.000010 0.000014 1.732045
    endloop
  endfacet
endsolid
```

*A word to the wise...*

The quick summary of this chapter is to know what your eventual medium will be, and to design for that medium when you plan your image or visualization. And be prepared to experiment as you work on your design, because some of these hardcopy media simply take experience that no notes or text can ever give you.

*Contacts:*

Below are contacts for the three 3D hardcopy sources shown above:

3D Systems
26081 Avenue Hall
Valencia, CA 91355 USA
+1.888.337.9786 (USA toll-free)
moreinfo@3dsystems.com
http://www.3dsystems.com

Helisys, Inc.
1000 E. Dominguez Street
Carson, CA 90746-3608 USA
+1.310.630.8840
http://helisys.com/

z corporation
20 North Avenue
Burlington, MA 01803 USA
+1.781-852-5005
http://www.zcorp.com/

## References and Resources

### *References*


[AMES] Andrea L. Ames, David R. Nadeau, and John L. Moreland, *VRML 2.0 Sourcebook*, Wiley, 1997

[ANG] Ed Angel, *Interactive Computer Graphics with OpenGL*, second edition, Addison-Wesley, 2000

[BAK] Steve Baker, A Brief MUI User Guide, distributed with the MUI release

[BAN] Banchoff, Tom et al., "Student-Generated Software for Differential Geometry," in Zimmermann and Cunningham, eds., *Visualization in Teaching and Learning Mathematics*, MAA Notes Number 19, Mathematical Association of America, 1991, pp. 165-171

[BRO] Judith R. Brown et al., *Visualization: Using Computer Graphics to Explore Data and Present Information*, Wiley, 1995

[EB] Ebert, David et al., *Texturing and Modeling: a Procedural Approach*, second edition, Academic Press, 1998

[FvD] James D. Foley et al, *Computer Graphics Principles and Practice*, 2nd edition, Addison-Wesley, 1990

[HIL] F. S. Hill, Jr., *Computer Graphics Using OpenGL*, 2nd edition, Prentice-Hall, 2001

[MUR] Murray, James D. and William vanRyper, *Encyclopedia of Graphics File Formats*, second edition, O'Reilly & Associates, 1996

[PER] Perlin, Ken, "An Image Synthesizer," *Computer Graphics* 19(3), Proceedings of SIGGRAPH 85, July 1985, 287-296

[POR] Porter and Duff, "Compositing Digital Images," *Proceedings*, SIGGRAPH 84 Conference, ACM SIGGRAPH, 1984

[SHR] Dave Shreiner, ed., *OpenGL Reference Manual*, 3rd edition, Addison-Wesley, 2000

[SOW1] Henry Sowrizal, Kevin Rushforth, and Michael Deering, *The Java3D™ 3D API Specification*, Addison-Wesley, 1995

[SOW2] Henry A. Sowizral and David R. Nadeau, *Introduction to Programming with Java 3D*, SIGGRAPH 99 Course Notes, Course 40

[SPE] Robert Spence, Information Visualization, Addison-Wesley/ACM Press Books, 2001

[SVR] The SIGGRAPH Video Review (SVR), an excellent source of animations for anyone wanting to see how images can communicate scientific and cultural information through computer graphics, as well as how computer graphics can be used for other purposes. Information on SVR can be found at `http://www.siggraph.org/SVR/`

[ViSC] Bruce H. McCormick, Thomas A. DeFanti, and Maxine D. Brown, eds., *Visualization in Scientific Computing*, *Computer Graphics* 21(6), November 1987

[vSEG] von Seggern, David, *CRC Standard Curves and Surfaces*, CRC Press, 1993

[WAT] Alan Watt & Fabio Policarpo, 3D Games: Real-time Rendering and Software Technology, Addison-Wesley/ACM SIGGRAPH Series, 2001

[WOL] Wolfe, R. J., *3D Graphics: A Visual Approach*, Oxford University Press, 2000

[WOO] Mason Woo et al., *OpenGL Programmers Guide*, 3rd edition (version 1.2), Addison-Wesley, 1999

[ZIM] Zimmermann, Walter and Steve Cunningham, *Visualization in Teaching and Learning Mathematics*, MAA Notes Number 19, Mathematical Association of America, 1991

- Textbook for multivariate calculus ...
- Textbook for differential geometry? ...

Plus:

paper on the alpha channel

## *Resources*

Chromadepth information is available from

Chromatek Inc
1246 Old Alpharetta Road
Alpharetta, GA 30005
888-669-8233
http://www.chromatek.com/

Chromadepth glasses may be ordered from

American Paper Optics
3080 Bartlett Corporate Drive
Bartlett, TN 38133
800-767-8427, 901-381-1515
fax 901-381-1517

Below are contacts for 3D hardcopy sources:

3D Systems
26081 Avenue Hall
Valencia, CA 91355 USA
+1.888.337.9786 (USA toll-free)
moreinfo@3dsystems.com
http://www.3dsystems.com

Helisys, Inc.
1000 E. Dominguez Street
Carson, CA 90746-3608 USA
+1.310.630.8840
http://helisys.com/

Stratasys, Inc.
14950 Martin Drive
Eden Prairie, MN 55344-2020 USA
+1-888-480-3548, +1.952.937.3000
FAX: +1.952.937.0070
e-mail: info@stratasys.com
http://www.stratasys.com/

z corporation
20 North Avenue
Burlington, MA  01803  USA
+1.781-852-5005
http://www.zcorp.com/

# Appendices

This section contains the details of some file formats that have been used in examples in these notes. They are included for the student who wants to work on projects that use such file formats.

## Appendix I: PDB file format

The national Protein Data Bank (PDB) file format is extremely complex and contains much more information than we can ever hope to use for student projects. We will extract the information we need for simple molecular display from the reference document on this file format to present here. From the chemistry point of view, the student might be encouraged to look at the longer file description to see how much information is recorded in creating a full record of a molecule.

There are two kinds of records in a PDB file that are critical to us: atom location records and bond description records. These specify the atoms in the molecule and the bonds between these atoms. By reading these records we can fill in the information in the internal data structures that hold the information needed to generate the display. The information given here on the atom location (ATOM) and bond description (CONECT) records is from the reference. There is another kind of record that describes atoms, with the keyword HETATM, but we leave this description to the full PDB format manual in the references.

ATOM records: The ATOM records present the atomic coordinates for standard residues, in angstroms. They also present the occupancy and temperature factor for each atom. The element symbol is always present on each ATOM record.

Record Format:

| COLUMNS | DATA TYPE | FIELD | DEFINITION |
|---|---|---|---|
| 1 - 6 | Record name | "ATOM " | |
| 7 - 11 | Integer | serial | Atom serial number. |
| 13 - 16 | Atom | name | Atom name. |
| 17 | Character | altLoc | Alternate location indicator. |
| 18 - 20 | Residue name | resName | Residue name. |
| 22 | Character | chainID | Chain identifier. |
| 23 - 26 | Integer | resSeq | Residue sequence number. |
| 27 | AChar | iCode | Code for insertion of residues. |
| 31 - 38 | Real(8.3) | x | Orthogonal coordinates for X in Angstroms. |
| 39 - 46 | Real(8.3) | y | Orthogonal coordinates for Y in Angstroms. |
| 47 - 54 | Real(8.3) | z | Orthogonal coordinates for Z in Angstroms. |
| 55 - 60 | Real(6.2) | occupancy | Occupancy. |
| 61 - 66 | Real(6.2) | tempFactor | Temperature factor. |
| 73 - 76 | LString(4) | segID | Segment identifier, left-justified. |
| 77 - 78 | LString(2) | element | Element symbol, right-justified. |
| 79 - 80 | LString(2) | charge | Charge on the atom. |

The "Atom name" field can be complex, because there are other ways to give names than the standard atomic names. In the PDB file examples provided with this set of projects, we have been careful to avoid names that differ from the standard names in the periodic table, but that means that we have not been able to use all the PDB files from, say, the chemical data bank. If your chemistry program wants you to use a particular molecule as an example, but that example's data file uses other formats for atom names in its file, you will need to modify the `readPDBfile()` function of these examples.

Example:

```
         1       2       3       4       5       6       7       8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
ATOM      1  C           1      -2.053   2.955   3.329  1.00  0.00
ATOM      2  C           1      -1.206   3.293   2.266  1.00  0.00
ATOM      3  C           1      -0.945   2.371   1.249  1.00  0.00
ATOM      4  C           1      -1.540   1.127   1.395  1.00  0.00
ATOM      5  C           1      -2.680   1.705   3.426  1.00  0.00
ATOM      6  C           1      -2.381   0.773   2.433  1.00  0.00
ATOM      7  O           1      -3.560   1.422   4.419  1.00  0.00
ATOM      8  O           1      -2.963  -0.435   2.208  1.00  0.00
ATOM      9  C           1      -1.455  -0.012   0.432  1.00  0.00
ATOM     10  C           1      -1.293   0.575  -0.967  1.00  0.00
ATOM     11  C           1      -0.022   1.456  -0.953  1.00  0.00
ATOM     12  C           1      -0.156   2.668   0.002  1.00  0.00
ATOM     13  C           1      -2.790  -0.688   0.814  1.00  0.00
ATOM     14  C           1      -4.014  -0.102   0.081  1.00  0.00
ATOM     15  C           1      -2.532   1.317  -1.376  1.00  0.00
ATOM     16  C           1      -3.744   1.008  -0.897  1.00  0.00
ATOM     17  O           1      -4.929   0.387   1.031  1.00  0.00
ATOM     18  C           1      -0.232  -0.877   0.763  1.00  0.00
ATOM     19  C           1       1.068  -0.077   0.599  1.00  0.00
ATOM     20  N           1       1.127   0.599  -0.684  1.00  0.00
ATOM     21  C           1       2.414   1.228  -0.914  1.00  0.00
ATOM     22  H           1       2.664   1.980  -0.132  1.00  0.00
ATOM     23  H           1       3.214   0.453  -0.915  1.00  0.00
ATOM     24  H           1       2.440   1.715  -1.915  1.00  0.00
ATOM     25  H           1      -0.719   3.474  -0.525  1.00  0.00
ATOM     26  H           1       0.827   3.106   0.281  1.00  0.00
ATOM     27  H           1      -2.264   3.702   4.086  1.00  0.00
ATOM     28  H           1      -0.781   4.288   2.207  1.00  0.00
ATOM     29  H           1      -0.301  -1.274   1.804  1.00  0.00
ATOM     30  H           1      -0.218  -1.756   0.076  1.00  0.00
ATOM     31  H           1      -4.617   1.581  -1.255  1.00  0.00
ATOM     32  H           1      -2.429   2.128  -2.117  1.00  0.00
ATOM     33  H           1      -4.464   1.058   1.509  1.00  0.00
ATOM     34  H           1      -2.749  -1.794   0.681  1.00  0.00
ATOM     35  H           1       1.170   0.665   1.425  1.00  0.00
ATOM     36  H           1       1.928  -0.783   0.687  1.00  0.00
ATOM     37  H           1      -3.640   2.223   4.961  1.00  0.00
ATOM     38  H           1       0.111   1.848  -1.991  1.00  0.00
ATOM     39  H           1      -1.166  -0.251  -1.707  1.00  0.00
ATOM     40  H           1      -4.560  -0.908  -0.462  1.00  0.00
```

CONECT records: The CONECT records specify connectivity between atoms for which coordinates are supplied. The connectivity is described using the atom serial number as found in the entry.

Record Format:

| COLUMNS | DATA TYPE | FIELD | DEFINITION |
|---|---|---|---|
| 1 - 6 | Record name | "CONECT" | |
| 7 - 11 | Integer | serial | Atom serial number |
| 12 - 16 | Integer | serial | Serial number of bonded atom |
| 17 - 21 | Integer | serial | Serial number of bonded atom |
| 22 - 26 | Integer | serial | Serial number of bonded atom |
| 27 - 31 | Integer | serial | Serial number of bonded atom |
| 32 - 36 | Integer | serial | Serial number of hydrogen bonded atom |
| 37 - 41 | Integer | serial | Serial number of hydrogen bonded atom |
| 42 - 46 | Integer | serial | Serial number of salt bridged atom |

| | | | |
|---|---|---|---|
| 47 - 51 | Integer | serial | Serial number of hydrogen bonded atom |
| 52 - 56 | Integer | serial | Serial number of hydrogen bonded atom |
| 57 - 61 | Integer | serial | Serial number of salt bridged atom |

Example:

```
          1         2         3         4         5         6         7
12345678901234567890123456789012345678901234567890123456789012345678901234567890
CONECT 1179  746 1184 1195 1203
CONECT 1179 1211 1222
CONECT 1021  544 1017 1020 1022 1211 1222      1311
```

As we noted at the beginning of this Appendix, PDB files can be extremely complex, and most of the examples we have found have been fairly large. The file shown in Figure 17.2 below is among the simplest PDB files we've seen, and describes the adrenalin molecule. This is among the materials provided as `adrenaline.pdb`.

```
HEADER    NONAME 08-Apr-99                                                NONE   1
TITLE                                                                     NONE   2
AUTHOR    Frank Oellien                                                   NONE   3
REVDAT   1  08-Apr-99     0                                               NONE   4
ATOM      1  C           0      -0.017   1.378   0.010  0.00  0.00           C+0
ATOM      2  C           0       0.002  -0.004   0.002  0.00  0.00           C+0
ATOM      3  C           0       1.211  -0.680  -0.013  0.00  0.00           C+0
ATOM      4  C           0       2.405   0.035  -0.021  0.00  0.00           C+0
ATOM      5  C           0       2.379   1.420  -0.013  0.00  0.00           C+0
ATOM      6  C           0       1.169   2.089   0.002  0.00  0.00           C+0
ATOM      7  O           0       3.594  -0.625  -0.035  0.00  0.00           O+0
ATOM      8  O           0       1.232  -2.040  -0.020  0.00  0.00           O+0
ATOM      9  C           0      -1.333   2.112   0.020  0.00  0.00           C+0
ATOM     10  O           0      -1.177   3.360   0.700  0.00  0.00           O+0
ATOM     11  C           0      -1.785   2.368  -1.419  0.00  0.00           C+0
ATOM     12  N           0      -3.068   3.084  -1.409  0.00  0.00           N+0
ATOM     13  C           0      -3.443   3.297  -2.813  0.00  0.00           C+0
ATOM     14  H           0      -0.926  -0.557   0.008  0.00  0.00           H+0
ATOM     15  H           0       3.304   1.978  -0.019  0.00  0.00           H+0
ATOM     16  H           0       1.150   3.169   0.008  0.00  0.00           H+0
ATOM     17  H           0       3.830  -0.755  -0.964  0.00  0.00           H+0
ATOM     18  H           0       1.227  -2.315  -0.947  0.00  0.00           H+0
ATOM     19  H           0      -2.081   1.509   0.534  0.00  0.00           H+0
ATOM     20  H           0      -0.508   3.861   0.214  0.00  0.00           H+0
ATOM     21  H           0      -1.037   2.972  -1.933  0.00  0.00           H+0
ATOM     22  H           0      -1.904   1.417  -1.938  0.00  0.00           H+0
ATOM     23  H           0      -3.750   2.451  -1.020  0.00  0.00           H+0
ATOM     24  H           0      -3.541   2.334  -3.314  0.00  0.00           H+0
ATOM     25  H           0      -4.394   3.828  -2.859  0.00  0.00           H+0
ATOM     26  H           0      -2.674   3.888  -3.309  0.00  0.00           H+0
CONECT    1    2    6    9    0                                           NONE  31
CONECT    2    1    3   14    0                                           NONE  32
CONECT    3    2    4    8    0                                           NONE  33
CONECT    4    3    5    7    0                                           NONE  34
CONECT    5    4    6   15    0                                           NONE  35
CONECT    6    5    1   16    0                                           NONE  36
CONECT    7    4   17    0    0                                           NONE  37
CONECT    8    3   18    0    0                                           NONE  38
CONECT    9    1   10   11   19                                           NONE  39
CONECT   10    9   20    0    0                                           NONE  40
CONECT   11    9   12   21   22                                           NONE  41
CONECT   12   11   13   23    0                                           NONE  42
CONECT   13   12   24   25   26                                           NONE  43
END                                                                       NONE  44
```

Figure 17.1: Example of a simple molecule file in PDB format

# Appendix II: CTL file format

The structure of the CT file is straightforward. The file is segmented into several parts, including a header block, the counts line, the atom block, the bond block, and other information. The header block is the first three lines of the file and include the name of the molecule (line 1); the user's name, program, date, and other information (line 2); and comments (line 3). The next line of the file is the counts line and contains the number of molecules and the number of bonds as the first two entries. The next set of lines is the atom block that describes the properties of individual atoms in the molecule; each contains the X-, Y-, and Z-coordinate and the chemical symbol for an individual atom. The next set of lines is the bonds block that describes the properties of individual bonds in the molecule; each line contains the number (starting with 1) of the two atoms making up the bond and an indication of whether the bond is single, double, triple, etc. After these lines are more lines with additional descriptions of the molecule that we will not use for this project. An example of a simple CTfile-format file for a molecule (from the reference) is given in Figure 17.2 below.

Obviously there are many pieces of information in the file that are of interest to the chemist, and in fact this is an extremely simple example of a file. But for our project we are only interested in the geometry of the molecule, so the additional information in the file must be skipped when the file is read.

```
L-Alanine (13C)
GSMACCS-II10169115362D 1 0.00366 0.00000 0

6 5 0 0 1 0 3 V2000
-0.6622  0.5342 0.0000 C    0 0 2 0 0 0
 0.6220 -0.3000 0.0000 C    0 0 0 0 0 0
-0.7207  2.0817 0.0000 C    1 0 0 0 0 0
-1.8622 -0.3695 0.0000 N    0 3 0 0 0 0
 0.6220 -1.8037 0.0000 O    0 0 0 0 0 0
 1.9464  0.4244 0.0000 O    0 5 0 0 0 0
1 2 1 0 0 0
1 3 1 1 0 0
1 4 1 0 0 0
2 5 2 0 0 0
2 6 1 0 0 0
M CHG 2 4 1 6 -1
M ISO 1 3 13
M END
```

Figure 17.2: Example of a simple molecule file in CTfile format

# Appendix III: the STL file format

The STL (sometimes called StL) file format is used to describe a file that contains information for 3D hardcopy systems. The name "STL" comes from stereo lithography, one of the technologies for 3D hardcopy, but the format is used in several other hardcopy technologies as described in the hardcopy chapter.

The .stl or stereolithography format describes an ASCII or binary file used in manufacturing. It is a list of the triangular surfaces that describe a computer generated solid model. This is the standard input for most rapid prototyping machines as described in the chapter of these notes on hardcopy. The binary format for the file is the most compact, but here we describe only the ASCII format because it is easier to understand and easier to generate as the output of student projects.

The ASCII .stl file must start with the lower case keyword `solid` and end with `endsolid`. Within these keywords are listings of individual triangles that define the faces of the solid model. Each individual triangle description defines a single normal vector directed away from the solid's surface followed by the xyz components for all three of the vertices. These values are all in Cartesian coordinates and are floating point values. The triangle values should all be positive and contained within the building volume. For this project the values are 0 to 14 inches in x, 0 to 10 in the y and 0 to 12 in the z. This is the maximum volume that can be built but the models should be scaled or rotated to optimize construction time, strength and scrap removal. The normal vector is a unit vector of length one based at the origin. If the normals are not included then most software will generate them using the right hand rule. If the normal information is not included then the three values should be set to 0.0. Below is a sample ASCII description of a single triangle within an STL file.

```
solid
...
facet normal 0.00 0.00 1.00
   outer loop
     vertex  2.00  2.00  0.00
     vertex -1.00  1.00  0.00
     vertex  0.00 -1.00  0.00
   endloop
 endfacet
...
endsolid
```

When the triangle coordinates are generated by a computer program,it is not unknown for roundoff errors to accumulate to the point where points that should be the same have slightly different coordinates. For example, if you were to calculate the points on a circle by incrementing the angle as you move around the circle, you might well end up with a final point that is slightly different from the initial point. File-checking software will note any difference between points and may well tell you that your object is not closed, but that same softward will often "heal" small gaps in objects automatically.

Vertex to vertex rule

The most common error in an STL file is non-compliance with the vertex-to-vertex rule. The STL specifications require that all adjacent triangles share two common vertices. This is illustrated in Figure 17.3. The figure on the left shows a top triangle containing a total of four vertex points. The outer vertices of the top triangle are not shared with one and only one other single triangle. The lower two triangles each contain one of the points as well as the fourth invalid vertex point.

To make this valid under the vertex to vertex rule the top triangle must be subdivided as in the example on the right.

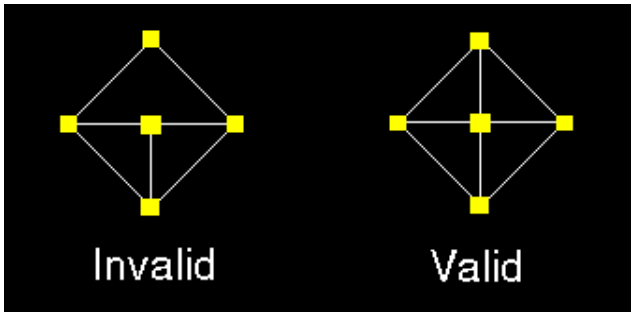



Figure 17.3

References:

*CTFile Formats*, MDL Information Systems, Inc., San Leandro, CA 94577, 1999. Available by download from `http://www.mdli.com/`

Protein Data Bank Contents Guide: Atomic Coordinate Entry Format Description, version 2.1, available online from `http://www.pdb.bnl.gov`

# Evaluation

These notes are under development, and we are very interested in hearing from anyone who uses them. Following this cover page we have added pages with questions for both instructors and students. As we work to develop the notes, your answers and comments will help us identify areas where we can improve the notes and will let us consider your ideas on how to make those improvements.

You can send us your comments by mail, by email, or by fax, and you may tell us who you are or remain anonymous (within the limits of the way you contact us, of course). If you respond by email, please send your comments to rsc@cs.csustan.edu. If you respond by post, please send your comments to

Steve Cunningham
Computer Science Department
California State University Stanislaus
801 W. Monte Vista Avenue
Turlock, CA 95382 USA

All comments will be acknowledged and will be fully considered.

## Instructor's evaluation

1. How did you use these notes? Did you review them for a course or simply to evaluate their possible use? Did you decide to use them for a course or did you decide not to use them?
2. If you chose to use them for a course, what was it about the notes that led you to that choice? If you chose not to use them, what was it about the notes that led you to that choice?
3. If you chose to use them, were they used as a course text, a course supplement, a student resource, or your personal resource? Were they used in a regular course, a special topics course, or a readings course?
4. If you chose to use them, how did your students respond to the notes?

While the notes are clearly incomplete and under development, we want your comments on the content. We would remind you of the goals of the notes as presented in the Getting Started chapter as you discuss the content.

5. Do you find the goals of the notes to represent a worthwhile approach to the introductory computer graphics course? Whether yes or no — but especially if no — we would value your feedback on these goals.
6. Were there any topics in the notes that seemed superfluous and could be omitted without any effect on your course?
7. Do you agree with the choice of OpenGL as the API for these notes, or do you suggest another API? Should the notes emphasize general concepts first in each section and then discuss the OpenGL implementation of the concepts, or should they use OpenGL as a motivator of the general discussion throughout?
8. Was the sequence of topics in the notes appropriate, or did you find that you would need to teach them in a different order to cover the material effectively?
9. Were there any topics in the notes that seemed particularly valuable to your course? Should these be emphasized, either through expanding their presence or through highlighting them in other parts of the notes?
10. Are the notes accurate? Is there any place where the notes are incorrect or misleading (not all areas have been fully tested, so this is possible)?
11. Are there areas where the discussions are difficult for students to follow?
12. Would you want to have supplementary material to accompany the notes? What kind of things would you want in such material? Should that material be on an accompanying CD-ROM or on an archival Web site?
13. Is there anything else — positive or negative — you would want to tell the author and the development group for this project?
14. Please tell us a bit about yourself: your department, your teaching and research interests, your resons for being interested in directions in computer graphics instruction, and anything else that will help us understand your comments above.

Thank you very much for your assistance.

## Student's evaluation

1. How did you find out about these notes? Was it from your instructor, from a friend, or from a general search of online computer graphics resources?

2. How did you use these notes? Was it in a class or as a personal resource?

3. If you used the notes in a class, what kind of class was it? Was it a beginning computer graphics course or did you have another computer graphics course first? Was it a regular undergraduate course, a special topics course, a readings course, or another kind of course? What department was the course offered in?

4. Do you agree with the use of OpenGL as the graphics API for the course that used these notes? Would you rather have had a different API? If so, what one and why?

5. Whak kind of system support did you use for the course (Windows, Linux, Unix, Macintosh, etc.)? Did you have to install any extra features (GLUT, MUI, etc.) yourself to use the notes? Did you need any extra instruction in the use of your local systems in order to use the ideas in the notes?

6. Without considering other important aspects of your course (laboratory, instructor, etc.), did you find the notes a helpful resource in learning computer graphics? Were you able to follow the discussions and make sense of the code fragments?

7. Were there any topics in the notes that seemed particularly valuable to your reading or your course? Should these be emphasized, either through expanding their presence or through highlighting them in other parts of the notes?

8. Were there any topics in the notes that seemed superfluous and could be removed without hurting your learning?

9. Were there any topics in computer graphics that you wanted to see in your reading or course that the notes did not cover? Why were these important to you?

10. Would you have liked to have additional materials to go along with the notes? What would you have wanted? How would you like to get these materials: on CD-ROM, on a Web site, or some other way?

11. Is there anything else — positive or negative — you would want to tell the author and the development group for this project?

12. Please tell us a bit about yourself: your major and year, your particular interests in computing and in computer graphics, your career goals, and anything else that will help us understand your comments above.

Thank you very much for your assistance.