Chapter 0: Getting Started

This chapter is intended to give you a basic overview of the concepts of computer graphics so that you can move forward into the rest of the book with some idea of what the field is about. It gives some general discussion of the basis of the field, and then has two key content areas.

The first key area is the discussion of three-dimensional geometry, managed by the 3D geometry pipeline, and the concept of appearance for computer graphics objects, managed by the rendering pipeline. The geometry pipeline shows you the key information that you must specify to create an image and the kind of computation a graphics system must do in order to present that image. We will also discuss some of the ways appearance can be specified, but we will wait until a later chapter to discuss the rendering pipeline.

The second key area is a presentation of the way a graphics program is laid out for the OpenGL graphics API, the key API we will use in this book. In this presentation you will see both the general structure of an OpenGL program and a complete example of a program that models a particular problem and produces a particular animated image. In that example you will see how the information for the geometry pipeline and the appearance information are defined for the program and will be able to try out various changes to the program as part of the chapter exercises.

3D Geometry and the Geometry Pipeline

Computer graphics, at least as we will treat it in this book, is fundamentally three-dimensional, so a graphics system must deal with three-dimensional geometry. Computer graphics systems do this by creating the 3D geometry pipeline, a collection of processes that convert 3D points, the basic building blocks of 3D geometry, from those that are most convenient for the application programmer into those that are most convenient for the display hardware. We will explore the details of the steps for the geometry pipeline in chapter on viewing and projection, but here we outline the steps of the geometry pipeline to help you understand how it operates. This pipeline is diagrammed in Figure 0.1, and we will start to sketch some stages in the pipeline in this chapter. A great deal more detail will be given in the next few chapters.

Figure 0.1: The geometry pipeline’s stages and mappings

3D model coordinate systems

In order to create an image, we must define the geometry that represents each part of the image. The process of creating and defining this geometry is called modeling, and is described in the chapters below on principles of modeling and on modeling in OpenGL. This is usually done by defining each object in terms of a coordinate system that makes sense for that particular object, and then using a set of modeling transformations that places that object in a single world coordinate system that represents the common space in which all the objects will live. Modeling then creates the 3D model coordinates for each object, and the modeling transformations place the objects in the world coordinate system that contains the entire scene.
3D world coordinate system

The 3D coordinate system that is shared by all the objects in the scene is called the world coordinate system. By placing every component of the scene in this single shared world, we can treat the scene uniformly as we develop the presentation of the scene through the graphics display device to the user. The scene is a master design element that contains both the geometry of the objects placed in it and the geometry of lights that illuminate it. Note that the world coordinate system often is considered to represent the actual dimensions of a scene because it may be used to model some real-world environment. This coordinate system exists without any reference to a viewer, as is the case with any real-world scene. In order to create an image from the scene, the viewer is added at the next stage.

3D eye coordinate system

Once the 3D world has been created, an application programmer would like the freedom to allow an audience 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 geometry 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. We can think of this process as grabbing the arbitrary eye location and all the 3D world objects and sliding them around to realign the spaces 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. Because standard viewing models may also specify a standard distance from the eyepoint to some fixed “look-at” point in the scene, there may also be some scaling involved in the viewing transformation. The viewing transformation is just a transformation in the same sense as modeling transformations, although it can be specified in a variety of ways depending on the graphics API. Because the viewing transformation changes the coordinates of 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.

Clipping

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 much of the scene should be visible, and includes defining the left, right, bottom, top, near, and far boundaries of that space. 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 0.2, it is clear that some of the world and some of the helicopter lie outside the viewable space to the left, right, bottom, top, or bottom, but note how the front of the image of the ground in the figure is clipped—is made invisible in the scene—because it is too close to the viewer’s eye. This is a bit difficult to see, but if you look at the cliffs at the upper left of the scene you will see a clipped edge.

Clipping is done as the scene is projected to the 2D eye coordinates in projections, as described next. Besides ensuring that the view includes only the things that should be visible, clipping also increases the efficiency of image creation because it eliminates some parts of the geometry from the rest of the display process.
Figure 0.2: Clipping on the Left, Bottom, and Right

Projections

The 3D eye coordinate system still must be converted into a 2D coordinate system before it can be mapped onto a graphics display device. The next stage of the geometry pipeline performs this operation, called a projection. Before discussing the actual projection, 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. In Figure 0.3 we see two viewing volumes for the two kinds of projections that we will discuss in a moment.

Figure 0.3: Parallel and Perspective Viewing Volumes, with Eyeballs

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, and it has the effect that the viewer can read accurate dimensions in the x- and y-coordinates. It is common for engineering drawings to present two parallel projections with the second including a 90° rotation of the world space so accurate z-coordinates can also be seen. 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 we see in Figure 0.3, 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 (also shown as a solid). This kind of shape is a truncated pyramid and is sometimes called a frustum.

While the viewing volume describes the region in space that is included in the view, the actual view is what is displayed on the front clipping space of the viewing volume. This is a 2D space and is essentially the 2D eye space discussed below. Figure 0.4 presents a scene with both parallel and perspective projections; in this example, you will have to look carefully to see the differences!

![Figure 0.4: the same scene as presented by a parallel projection (left) and by a perspective projection (right)](image)

2D eye coordinates

The space that projection maps to is a two-dimensional real-coordinate space that contains the geometry of the original scene after the projection is applied. Because a single point in 2D eye coordinates corresponds to an entire line segment in the 3D eye space, depth information is lost in the projection and it can be difficult to perceive depth, particularly if a parallel projection was used. Even in that case, however, if we display the scene with a hidden-surface technique, object occlusion will help us order the content in the scene. Hidden-surface techniques are discussed in a later chapter.
The final step in the geometry pipeline is to change the coordinates of objects in the 2D eye space so that the object is in a coordinate system appropriate for the 2D 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.

Appearance

Along with geometry, computer graphics is built on the ability to define the appearance of objects, so you can make them appear naturalistic or can give them colors that can communicate something to the user.

In the discussion so far, we have only talked about the coordinates of the vertices of a model. There are many other properties of vertices, though, that are used in rendering the scene, that is, in creating the actual image defined by the scene. These are discussed in many of the later chapters, but it is worth noting here that these properties are present when the vertex is defined and are preserved as the vertex is processed through the geometry pipeline. Some of these properties involve concepts that we have not yet covered, but these will be defined below. These properties include:

• a depth value for the vertex, defined as the distance of the vertex from the eye point in the direction of the view reference point,
• a color for the vertex,
• a normal vector at the vertex,
• material properties for the vertex, and
• texture coordinates for the vertex.

These properties are used in the development of the appearance of each of the objects in the image. They allow the graphics system to calculate the color of each pixel in the screen representation of the image after the vertices are converted to 2D screen coordinates. For the details of the process, see the chapter below on the rendering pipeline.

Appearance is handled by operations that are applied after the geometry is mapped to screen space. In order to do this, the geometric primitives described above are broken down into very simple primitives and these are processed by identifying the parts of the window raster that make up each one. This is done by processing the vertex information described in the previous paragraph into scanline information, as described in a later chapter. Appearance information is associated with each vertex, and as the vertex information is processed into scanlines, and as the pixels on each scanline are processed, appearance information is also processed to create the colors that are used in filling each primitive. Processes such as depth buffering are also handled at this stage, creating the appropriate visible surface view of a scene. So the appearance information follows the
geometry information, and the chapters of this book that discuss appearance issues will follow most of the geometry chapters.

Color

Color can be set directly by the program or can be computed from a lighting model in case your scene is defined in terms of lights and materials. Most graphics APIs now support what is called RGBA color: color defined in terms of the emissive primaries red, green, and blue, and with an alpha channel that allows you to blend items with the background when they are drawn. These systems also allow a very large number of colors, typically on the order of 16 million. So there are a large number of possibilities for color use, as described in later chapters on color and on lighting.

Texture mapping

Among the most powerful ways to add visual interest to a scene is texture mapping, a capability that allows you to add information to objects in a scene from either natural or synthetic complex images. With texture mapping you can achieve photographic surface effects or other kinds of images that will make your images much more interesting and realistic. This is discussed in a later chapter and should be an important facility for you.

Depth buffering

As your scene is developed, you want only the objects nearest the eye to be seen; anything that is behind these will be hidden by the nearer objects. This is managed in the rendering stage by keeping track of the distance of each vertex from the eye. If an object is nearer than the previously drawn part of the scene for the same pixels, then the object will replace the previous part; otherwise the previous part is retained. This is a straightforward computation that is supported by essentially all modern graphics systems.

The 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 0.1 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 finally apply the projection to the screen. This can be expressed in terms of function composition as the sequence

\[ \text{projection}(\text{viewing}(\text{transformation}(\text{geometry}))) \]

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

\[ (\text{projection} \ast \text{viewing} \ast \text{transformation})(\text{geometry}). \]

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. This is independent of whether we want to use a perspective or parallel projection. 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! To this point, our discussion has only shown the concept of how a vertex travels through the geometry pipeline, but we not given any details on how this actually is done. There are several ways of implementing this travel, any of which will produce a correct display. Do not be surprised if you find out a graphics system does not manage the overall geometry 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 need to measure or align parts of the image

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 structure-only example in C, as we will with all our examples in these notes. We have chosen not to present examples in C++ because 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, while object-oriented design usually calls for objects to maintain their own state. 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.)

In the code below, you will see that the main function involves mostly setting up the system. This is done in two ways: first, setting up GLUT to create and place the system window in which your work will be displayed, and second, setting up the event-handling system by defining the callbacks to be used when events occur. After this is done, main calls the main event loop that will drive all the program operations, as described in the chapter below on event handling.

The full code example that follows this outline also discusses many of the details of these functions and of the callbacks, so we will not go into much detail here. For now, the things to note are that the reshape callback sets up the window parameters for the system, including the size, shape, and location of the window, and defines the projection to be used in the view. This is called first when the main event loop is entered as well as when any window activity happens (such as resizing or dragging). The reshape callback requests a redisplay when it finishes, which calls the display callback function, whose task is to set up the view and define the geometry for the scene. When this is finished, OpenGL is finished and goes back to your computer system to see if there has been any other graphics-related event. If there has, your program should have a callback to manage it; if there has not, then the idle event is generated and the idle callback function is called; this may change some of the geometry parameters and then a redisplay is again called.

```c
#include <GL/glut.h>   // Windows; other includes for other systems
// other includes as needed

// typedef and 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
}
```

3/15/03 Page 0.8
// display function
void display(void){
    set up viewing transformation as in later chapters
    define the geometry, transformations, appearance you need
    post redisplay
}

// idle function
void idle(void) {
    update anything that changes between steps of the program
    post redisplay
}

// other graphics and application functions
// as needed

// main function -- set up the system, 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 as needed; this is pretty minimal
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutIdleFunc(idle);
    // go into main event loop
    glutMainLoop();
}

Now that we have seen a basic structure for an OpenGL program, we will present a complete, working program and will analyze the way it represents the geometry pipeline described earlier in this chapter, while describing the details of OpenGL that it uses. The program is the simple simulation of temperatures in a uniform metal bar that is described in the later chapter on graphical problem-solving in science, and we will only analyze the program structure, not its function. It creates the image shown in Figure 0.5.

![Figure 0.5: heat distribution in a bar](image)

3/15/03
The code we will discuss is given below. We will segment it into components so you may see the different ways the individual pieces contribute to the overall graphics operations, and then we will discuss the pieces after the code listing.

// Example – heat flow in a thin rectangular body
// declarations and initialization of variables and system
#include <GL/glut.h>  // for windows; changes for other systems
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
define ROWS 10 // body is ROWSxCOLS (unitless) squares
#define COLS 30
#define AMBIENT 25.0; // ambient temperature, degrees Celsius
#define HOT 50.0  // hot temperature of heat-source cell
#define COLD 0.0  // cold temperature of cold-sink cell
#define NHOTS 4 // number of hot cells
#define NCOLDS 5 // number of cold cells
GLfloat angle = 0.0;
GLfloat temps[ROWS][COLS], back[ROWS+2][COLS+2];
GLfloat theta = 0.0, vp = 30.0;
int hotspots[NHOTS][2] =
      {{ROWS/2,0},{ROWS/2-1,0},{ROWS/2-2,0},{0,3*COLS/4}};
int coldspots[NCOLDS][2] =
      {{ROWS-1,COLS/3}, {ROWS-1,1+COLS/3}, {ROWS-1,2+COLS/3},
        {ROWS-1,3+COLS/3}, {ROWS-1,4+COLS/3}};

myWin;
void myinit(void);
void cube(void);
void display(void);
void setColor(float);
void reshape(int, int);
void animate(void);
void iterationStep(void);

void myinit(void) {
    int i, j;
    glEnable (GL_DEPTH_TEST);
    glClearColor(0.6, 0.6, 0.6, 1.0);

    // set up initial temperatures in cells
    for (i=0; i<ROWS; i++) {
        for (j=0; j < COLS; j++) {
            temps[i][j] = AMBIENT;
        }
    }
    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;
    }
}
// create a unit cube in first octant in model coordinates

void cube (void) {
    typedef GLfloat point [3];
    
    point v[8] = {
        {0.0, 0.0, 0.0}, {0.0, 0.0, 1.0},
        {0.0, 1.0, 0.0}, {0.0, 1.0, 1.0},
        {1.0, 0.0, 0.0}, {1.0, 0.0, 1.0},
        {1.0, 1.0, 0.0}, {1.0, 1.0, 1.0} 
    };
    
    glBegin (GL_QUAD_STRIP);
    glVertex3fv(v[4]);
    glVertex3fv(v[5]);
    glVertex3fv(v[0]);
    glVertex3fv(v[1]);
    glVertex3fv(v[2]);
    glVertex3fv(v[3]);
    glVertex3fv(v[6]);
    glVertex3fv(v[7]);
    glEnd();
    
    glBegin (GL_QUAD_STRIP);
    glVertex3fv(v[1]);
    glVertex3fv(v[3]);
    glVertex3fv(v[5]);
    glVertex3fv(v[7]);
    glVertex3fv(v[4]);
    glVertex3fv(v[6]);
    glVertex3fv(v[0]);
    glVertex3fv(v[2]);
    glEnd();
}

void display( void ) {
    #define SCALE 10.0
    int i,j;
    
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    
    // This short section defines the viewing transformation
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    //           eye point      center of view       up
    gluLookAt(vp, vp/2., vp/4., 0.0, 0.0, 0.0,  0.0, 0.0, 1.0);
    
    // Set up a rotation for the entire scene
    glPushMatrix();
    glRotatef(alpha, 0., 0., 1.0);
    
    // Draw the bars
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            setColor( temps[i][j] );
            glPushMatrix();
            glTranslatef((float)i-(float)ROWS/2.0,
            (float)j-(float)COLS/2.0, 0.0);
            // Here is the modeling transformation for each item in the display
            glPushMatrix();
            glTranslatef((float)i-(float)ROWS/2.0,
            (float)j-(float)COLS/2.0, 0.0);
        }
    }
}
// 0.1 cold, 4.0 hot
glScalef(1.0, 1.0, 0.1+3.9*temps[i][j]/HOT);
cube();
glPopMatrix();
}
}
glPopMatrix(); // pop the scene rotation
glutSwapBuffers();
}
void reshape(int w, int h) {
// This defines the projection transformation
   glViewport(0,0,(GLsizei)w,(GLsizei)h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0, (float)w/(float)h, 1.0, 300.0);
}
void setColor(float t) {
   float r, g, b;
   r = t/HOT; g = 0.0; b = 1.0 - t/HOT;
   glColor3f(r, g, b);
}
void animate(void) {
// This function is called whenever the system is idle; it calls
// iterationStep() to change the data so the next image is changed
   iterationStep();
   glutPostRedisplay();
}
void iterationStep(void) {
   int i, j, m, n;

   float filter[3][3]=
   {{ 0., 0.125, 0. },
    { 0.125, 0.5, 0.125 },
    { 0., 0.125, 0. } };

   // increment temperatures throughout the material
   for (i=0; i<ROWS; i++) // backup temps up to recreate it
      for (j=0; j<COLS; j++)
         back[i+1][j+1] = temps[i][j]; // leave boundaries on back
   // fill boundaries with adjacent values from original temps[][]
   for (i=1; i<ROWS+2; i++)
      for (j=0; j<COLS+2; j++)
         {
            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];
         }
   for (i=0; i<ROWS; i++) // diffusion based on back values
      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];
}  
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;  }

// update the angle for the rotation
angle += 1.0;)

void main(int argc, char** argv) {
  // Initialize the GLUT system and define the window
  glutInit(&argc,argv);
  glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(500,500);
  glutInitWindowPosition(50,50);
  myWin = glutCreateWindow("Temperature in bar");
  myinit();

  // define the event callbacks and enter main event loop
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutIdleFunc/animate();
  glutMainLoop(); /* enter event loop */
}

The structure of the main() program using OpenGL

The main() program in an OpenGL-based application looks somewhat different from the programs we probably have seen before. This function has several key operations: it sets up the display mode, defines the window in which the display will be presented, and does whatever initialization is needed by the program. It then does something that may not be familiar to you: it defines a set of event callbacks, which are functions that are called by the system when an event occurs.

When you set up the display mode, you indicate to the system all the special features that your program will use at some point. In the example here,

  glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);

you tell the system that you will be working in double-buffered mode, will use the RGB color model, and will be using depth testing. Some of these have to be enabled before they are actually used, as the depth testing is in the myInit() function with

  glEnable(GL_DEPTH_TEST).

Details on depth testing and a discussion of how this is managed in OpenGL are found in the next chapter.

Setting up the window (or windows—OpenGL will let you have multiple windows open and active) is handled by a set of GLUT function calls that position the window, define the size of the window, and give a title to the window. As the program runs, an active window may be reshaped by the user using the standard techniques of whatever window system is being used, and this is handled by the reshape() function.

The way OpenGL handles event-driven programming is described in much more detail in a later chapter, but for now you need to realize that GLUT-based OpenGL (which is all we will describe in this book) operates entirely from events. For each event that the program is to handle, you need to define a callback function here in main(). When the main event loop is started, a reshape
event is generated to create the window and a display event is created to draw an image in the window. If any other events have callbacks defined, then those callback functions are invoked when the events happen. The reshape callback allows you to move the window or change its size, and is called whenever you do any window manipulation. The idle callback allows the program to create a sequence of images by doing recomputations whenever the system is idle (is not creating an image or responding to another event), and then redisplaying the changed image.

**Model space**

The function `cube()` above defines a unit cube with sides parallel to the coordinate axes, one vertex at the origin, and one vertex at (1,1,1). This cube is created by defining an array of points that are the eight vertices of such a cube, and then using the `glBegin()`...`glEnd()` construction to draw the six squares that make up the cube through two quad strips. This is discussed in the chapter on modeling with OpenGL; for now, note that the cube uses its own set of coordinates that may or may not have anything to do with the space in which we will define our metallic strip to simulate heat transfer.

**Modeling transformation**

Modeling transformations are found in the `display()` function or functions called from it, and are quite simple: they define the fundamental transformations that are to be applied to the basic geometry units as they are placed into the world. In our example, the basic geometry unit is a unit cube, and the cube is scaled in Z (but not in X or Y) to define the height of each cell and is then translated by X and Y (but not Z) to place the cell in the right place. The order of the transformations, the way each is defined, and the `pushMatrix()`/`popMatrix()` operations you see in the code are described in the later chapter on modeling in OpenGL. For now it suffices to see that the transformations are defined in order to make a rectangular object of the proper height to represent the temperature.

**3D world space**

The 3D world space for this program is the space in which the graphical objects live after they have been placed by the modeling transformations. The translations give us one hint as to this space; we see that the x-coordinates of the translated cubes will lie between $-\text{ROWS}/2$ and $\text{ROWS}/2$, while the y-coordinates of these cubes will lie between $-\text{COLS}/2$ and $\text{COLS}/2$. Because $\text{ROWS}$ and $\text{COLS}$ are 30 and 10, respectively, the x-coordinates will lie between -15 and 15 and the y-coordinates will lie between -5 and 5. The low z-coordinate is 0 because that is never changed when the cubes are scaled, while the high z-coordinate is never larger then 4. Thus the entire bar lies in the region between -15 and 15 in x, -5 and 5 in y, and 0 and 4 in z. (Actually, this is not quite correct, but it is adequate for now; you are encouraged to find the small error.)

**Viewing transformation**

The viewing transformation is defined at the beginning of the `display()` function above. The code identifies that it is setting up the modelview matrix, sets that matrix to the identity (a transformation that makes no changes to the world), and then specifies the view. A view is specified in OpenGL with the `gluLookAt()` call:

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

with parameters that 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 is discussed in more detail in the chapter below on viewing.
3D eye space

There is no specific representation of the 3D eye space in the program, because this is simply an intermediate stage in the production of the image. We can see, however, that we had set the center of view to the origin, which is the center of our image, and we had set our eye point to look at the origin from a point somewhat above and to the right of the center, so after the viewing transformation the object seems to be tilted up and to the side. This is the representation in the final 3D space that will be used to project the scene to the eye.

Projections

The projection operation is defined here in the reshape() function. It may be done in other places, but this is a good location and clearly separates the operation of projection from the operation of viewing.

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

```c
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 can be defined with the function call:

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

In this function, 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.

When the window is reshaped, it is useful to take the width and height from the reshape event and define your projection to have the same aspect ratio (ratio of width to height) that the window has. That way there is no distortion introduced into the scene as it is seen through the newly-shaped window. If you use a fixed aspect ratio and change the window’s shape, the original scene will be distorted to be seen through the new window, which can be confusing to the user.

2D eye space

This is the real 2D space to which the 3D world is projected, and it corresponds to the forward plane of the view volume. In order to provide uniform dimensions for mapping to the screen, the eye space is scaled so it has dimension -1 to 1 in each coordinate.

2D screen space

When the system was initialized, the window for this program was defined to be 500x500 pixels in size with a top corner at (50, 50), or 50 pixels down and 50 pixels over from the upper-left corner of the screen. Thus the screen space for the window is the set of pixels in that area of the screen. In fact, though, the window maintains its coordinate system independently of its location, so the point that had been (0, 0, 0) in 3D eye space is now (249, 249) in screen space. Note that screen space has integer coordinates that represent individual pixels and is discrete, not continuous, and its coordinates start at 0.
Appearance

The appearance of the objects in this program is defined by the function `setColor()`, called from the `display()` function. If you recall that `display()` is also the place where modeling is defined, you will see that appearance is really part of modeling—you model both the geometry of an object and its appearance. The value of the temperature in each cell is used to compute a color for the cell’s object as it is displayed, using the OpenGL `glColor3f()` function. This is about the simplest way to define the color for an object’s appearance, but it is quite effective.

Another way to see the program

Another way to see how this program works is to consider the code function-by-function instead of by the properties of the geometry pipeline. We will do this briefly here.

The task of the function `myinit()` 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 (the 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()` is to respond to user manipulation of the window in which the graphics are displayed. The function takes two parameters, which 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 `animate()` 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 0.7. 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.

```
main() ----} display()

redisplays event      no events?

|------|        | idle() |
```

Figure 0.7: the event loop for the idle event

So we see that in the absence of any other event activity, the program will continue 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.

A few words on the details of the `idle()` function might help in seeing what it does. The whole program presents the behavior of heat in a bar, and the transfer of heat from one place to another is described by the heat equation. In this program we model heat transfer by a diffusion process. This uses a filter that sets the current heat at a position to a weighted average of the heat of the cell’s neighbors, modeled by the filter array in this function. A full description of this is in the chapter on science applications. At each time step—that is, at each time when the program becomes idle—this diffusion process is applied to compute a new set of temperatures, and the angle of rotation of the display is updated. The call to `glutPostRedisplay()` at the end of this function then generates a call to the `display()` function that draws the image with the new temperatures and new angle.

In looking at the execution sequence for the functions in this simple program, it can be useful to consider a graph that shows which functions are called by which other functions. Bearing in mind that the program is event-driven and so the event callback functions (`animate()`, `display()`, and `reshape()`) are not called directly by the program, we have the function caller-callee graph in Figure 0.8.

Note that the graph is really a tree: functions are called only by event callbacks or the `init()` function, the `init()` function is called only once from `main()`, and all the event callbacks are called from the event handler. For most OpenGL programs, this is the general shape of the graph: a callback function may use several functions, but any function except a callback will only be called as part of program initialization or from an event callback.

```
main() ----} event handler

|------|        |-----|        |

myinit()   animate()     reshape()     display()

|------|        |-----|        |

iterationStep()   cube()     setColor()
```

Figure 0.8: the function caller/callee graph for the example program
Now that we have an idea of the geometry 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.

**OpenGL extensions**

In this chapter, and throughout these notes, we take a fairly limited view of the OpenGL graphics API. Because this is an introductory text, we focus on the basic features of computer graphics and of the graphics API, so we do not work with most of the advanced features of the system and we only consider the more straightforward uses of the parts we cover. But OpenGL is capable of very sophisticated kinds of graphics, both in its original version and in versions that are available for specific kinds of graphics, and you should know of these because as you develop your graphics skills, you may find that the original “vanilla” OpenGL that we cover here will not do everything you want.

Advanced features of OpenGL include a number of special operation to store or manipulate information on a scene. These include modeling via polygon tessellation, NURBS surfaces, and defining and applying your own special-purpose transformations; the scissor test and the more general stencil buffer and stencil test; rendering in feedback mode to get details on what is being drawn; and facilities for client/server support. Remember that this is a general text, not a detailed presentation of OpenGL, and be ready to look further (see the references) for more information.

In addition to standard OpenGL, there are a number of extensions that support more specialized kinds of operations. These include the ARB imaging subset extension for image processing, the ARB multitexturing extension, vertex shader extensions, and many others. Some of these might have just the tools you need to do the very special things you want, so it would be useful for you to keep up to date on them. You can get information on extensions at the standard OpenGL Web site, [http://www.opengl.org](http://www.opengl.org).

**Summary**

In this chapter we have discussed the geometry pipeline and have indicated what each step involves and how it contributes to creating the final image you are programming. We have also shown how appearance fits into the geometry pipeline, although it is actually implemented in a separate pipeline, and how all of this is implemented through a complete sample OpenGL program. In fact, you actually have a significant tool in this sample program, because it can be modified and adapted to serve as a basis for a great deal of other graphics programming. We do not have any programming projects in this chapter, but these will come along quickly and you will be able to use this sample program to get started on them.

**Questions**

1. There are other ways to do graphics besides API-based programming. You can use a number of different modeling, painting, and other end-user tools. Distinguish between API-based graphics and graphics done with a tool such as Photoshop™ or a commercial paint program. The sample program in this chapter can give you an idea of how API-based graphics can look, although it is only a simple program and much more complex programs are discussed in later chapters.

2. Trace the behavior of the 3D geometry in the sample program through the steps of the geometry pipeline from the point where you define a unit cube in model space, through the transformations that place that point into world space, through the viewing transformation that
places the point in 3D eye space, to the projection that places the point in 2D eye space. Without doing any of the mathematics, work out what kind of changes are made in the points’ coordinates as these operations are performed.

**Exercises**

3. Compile and execute the full sample program in the chapter so you can become familiar with the use of your compiler for graphics programming. Exercise the `reshape()` function in the code by dragging and resizing the window. As you manipulate the window, change the shape of the window (make it narrower but not shorter, for example, or make it shorter but not narrower) and see how the window and image respond.

**Experiments**

4. There are many ways you can experiment with the full sample program in this chapter. A few of them include
   (a) changing the size and upper left corner coordinates of the window [function `main()`]
   (b) changing the locations of the hot and cold spots in the bar [function `myinit()`]
   (c) changing the way the color of each bar is computed by changing the function that determines the color [function `setColor()`] (see the later chapter on color for more on this topic)
   (d) changing the rate at which the image rotates by changing the amount the angle is increased [function `animate()`]
   (e) changing the values in the filter array to change the model of how heat diffuses in the bar [function `iterationStep()`]
   (f) changing the way the edge of the bar is treated, so that instead of simply repeating the values at the edge, you get the values at the opposite edge of the bar, effectively allowing temperatures to move from one edge to the other as if the bar were a torus [function `iterationStep()`]
   (g) changing the view of the bar from a perspective view to an orthogonal view [function `reshape()`] (you will probably need to look up the details of orthogonal projections in the appropriate chapter below).
Take as many of them as you can and add appropriate code changes to the code of the previous exercise and observe the changes in the program behavior that result. Draw as many conclusions as you can about the role of these various functions in creating the final image.

5. Continuing with the earlier theme of the `reshape()` function, look at the code of the `reshape()` function and think about how you might make it respond differently. The current version uses the window dimensions `w` and `h` in the perspective projection definition to ensure that the aspect ratio of the original image are preserved, but the window may cut off part of the image if it is too narrow. You can change the projection angle to increase as the window is narrower, for example. Change the code in `reshape()` to try to change the behavior in the window.

6. There are some sample programs available for this book, and there are an enormous number of OpenGL programs available on the Web. Find several of these and create the graph of function calls described in this chapter to verify (or refute) the claim there that functions tend to operate in either program initialization or a single event callback. What does this tell you about the way you develop a graphics program with OpenGL? Where do you find most of the user-defined functions within the program?