CHAPTER 1

INTRODUCTION

The computing world of the late 20th and early 21st centuries have been dominated by Microsoft's Windows operating systems (95, 98, NT, 2000, XP), taking control of around 90% of the consumer marketplace. Alternative operating systems such as Linux or Mac OS often have similarly matched software offerings which Windows holds, whether it be web browsers, e-mail clients, or productivity programs. However, there is one area where the Windows-based realm reigns – games. Generally, only the most successful games get ported to Linux or Mac. Even then, there is no guarantee of this happening. When the top games cost millions of dollars to produce, there is little incentive to spend further money and time on minority platforms.

Much of the difficulty of porting games from Windows to other systems is caused by developers using Microsoft-branded tools such as DirectX or the Windows Application Programming Interface (API). When porting, a developer essentially has to translate the game from one language to another operating system's APIs. PIGE is intended to greatly reduce the struggle involved when porting by removing as many system-dependent APIs as possible and relying on open technologies such as OpenGL (Open Graphics Library), OpenAL (Open Audio Library), and the C language, one of the most popular programming languages of the past 30 years.

PIGE follows after the Java methodology: write once, run anywhere. The intended purpose of PIGE is to provide the necessary platform dependent tools for

various operating systems (Windows, Linux, Mac), but to allow the core code to remain virtually untouched, so the only modification when compiling for another platform is to change a library name or file header in the main source file. It will not be true platform independence, but the game engine is supported on the top three current operating systems, which are Mac OS, Linux, and Windows. Ideally, all that would be necessary to set up a program using PIGE code would be to add the platform specific header in the main file. Table 1 demonstrates the naming convention used for the header libraries to link to a program's main function.

Operating System	Header
Linux	<pige_linux.h>
Macintosh	<pige_mac.h>
Windows	<pige_windows.h>

Table 1. Header examples.

The 1980s saw computer gaming become a serious form of business and entertainment. Companies such as Sierra On-line and LucasArts created quite a few games in the late 1980s, many with a similar look and feel. This is because they modeled many of their games from the same game engines, which allowed the game developers to work more on the game design, and not on the supporting technology. Like these 1980s game engines, PIGE strives to follow the often touted practice of code reuse.

Since PIGE is available in the public domain, its contents are freely available to anyone with no restrictions. PIGE's free availability is a benefit to the small-scale game developer, or for a person who is interested in learning more about game programming.

Computer	Specifications
Apple Power Mac G4	400 MHz G4 2 Hard drives 20 GB running Mac OS 9 15 GB running Mac OS 10.2 896 MB RAM 16 MB ATI Rage 128 Pro Project Builder
Apple iBook	500 MHz G3 10 GB hard drive, two partitions, running Mac OS 9 and Mac OS 10.1 320 MB RAM 8 MB ATI Rage Mobility Project Builder
PC	1 GHz Intel Pentium III 2 hard drives 3 GB running Windows 98 60 GB running RedHat Linux 8.0 320 MB RAM 32 MB ATI Rage Pro CodeWarrior and gcc

Table 2. Test computer systems.

PIGE was developed on three different machines, two Macintosh computers, and a generic PC running both Windows 98 and RedHat Linux 8.0. Table 2 lists the further specifications of these machines.

This paper is divided into several parts: graphics, sound, porting, and integration. Chapter 2 will discuss more complicated elements of graphics such as collision detection, picking and selection, and loading textures using OpenGL. Chapter 3 will go over the fundamentals of OpenAL and integrating it with OpenGL. Chapter 4 will review the difficulties involved when porting these open technologies across several platforms. Integrating PIGE's graphics and audio components together will be explained in Chapter 5. The appendices hold example code which was used in creating PIGE.

Related Work

If an open source, platform independent, game engine already existed, there would be little purpose for the creation of PIGE. There are quite a few web sites dedicated to the creation of games, such as NeHe [10], Gametutorials.com, idevgames.com, Gamasutra.com, and several other sites committed to gaming technologies for OpenGL and OpenAL.

The most prominent game developer to support multiple platforms is id Software. Many of id's games, especially those in the Quake series, were released for Windows, Linux, and Macintosh. In 1999, id Software released the source code for Quake under the GNU General Public License [5], allowing people to download, view, and use the source code for free as long as they adhered to the GPL guidelines. Two years later, the source code for Quake II was released. Their latest release in the Quake series, Quake III, has not been released under the GPL, but is licensed for $250,000 against a 5% royalty of the wholesale for a single license [8]. The first two Quake

games are essentially free under the GPL, but they contain dated technology. Especially in the gaming realm, any technology more than two or three years old becomes recognizably dated. The Quake III engine displays id's most current offerings, but it comes at an expensive price which does not make it a feasible solution to smaller developers.

One of the most inexpensive game engines is the Torque Engine by Garage Games, which was formed from the remnants of Dynamix, after it closed in September 2001. The game engine costs $100 per programmer [6], but it has the restriction that the game needs to be published by Garage Games. The Torque Engine is based on the technology which was used to develop the Tribes games (Tribes, Starsiege, Tribes2). The Torque Engine has been made available for Windows, Linux, and Mac OS 9/X, making it one of the most versatile game engines available.

A trend over the last several years is for companies such as Loki or Westlake Interactive to be solely focused on porting games from Windows to Linux or Macintosh. The disadvantage of this method is that only the most popular games are worth porting, and additional resources of time and money are necessary to fund the port.

Yet, for all of these solutions, none of them have tried to combine OpenGL and OpenAL into an open source game engine, which is the focus of PIGE.

CHAPTER 2

OPENGL

OpenGL was introduced in 1992 as an Application Programming Interface (API) to support the drawing of 2D and 3D graphics [12]. Since then, it has become an industry standard for graphics, extending its capabilities across a wide variety of applications such as medical imaging, virtual reality, mathematical visualization, and computer games. Various technologies have vied with OpenGL for prominence, but none have had its success.

However, OpenGL is somewhat limited in its capabilities, limited mostly to the rendering and drawing of images. GLUT (OpenGL Utility Toolkit) is a window system independent toolkit often used in conjunction with OpenGL to provide extra functionality for windows and I/O communication via the mouse or keyboard. What OpenGL does not take care of, GLUT handles.

OpenGL and GLUT are supported on nearly every popular operating system and computer architecture available. These are optimal choices for PIGE, providing for graphics and user input capabilities.

Documented below are several of the more complicated graphics techniques involved in PIGE. The more rudimentary and basic elements of OpenGL will not be covered in this paper. Refer to either OpenGL Programming Guide [15] or OpenGL : A Primer [1] for an introduction to programming OpenGL.

Collision Detection

With the advent of 3D technologies in the past several years, programmers have made radical changes in how they program applications, especially when regarding computer games. Collision detection is an essential part in 3D games. It ensures that the game physics are relatively realistic, so that an object does not cut through other objects or hovers when it should fall. How well a game can detect collisions is an integral part of the believability and enjoyment of the game. A poorly implemented collision detection system can be a bane to a product, whereas an excellent implementation can produce amazing results.

The two main parts in collision detection are detecting whether or not a collision has happened, and if so, responding to the collision. Discovering if a collision has occurred is the basis of this problem.

While responding to the collision is computationally much easier than discovering a collision, it can still pose several problems in how objects are going to react to each other. In modern computer games, if the character runs into a wall, then the character will either stop or will continue 'sliding' along the wall. However, if this character comes up to a movable box, then the character might start pushing the box instead. Or consider a ball bouncing around in a room. The ball is going to behave quite differently than a person walking around in a room.

The first step is to see if the viewer, or the 'camera', will move through any polygons or planes on its next move. Instead of just calculating if the camera will hit

any particular polygon, all polygons are extended indefinitely along their plane. This makes calculations easier to initially perform. If the camera does not intersect the plane, then no calculations are necessary to see if the camera will cross through the polygon itself. This saves some computation by using an easier calculation (with less processing). When a collision with the plane is detected, further processing is done to check if the camera intersects the polygon itself.

2D applications can easily detect collisions by determining if two objects are trying to occupy the same area. If the circle in Figure 1 is trying to get to the triangle, it checks in all available directions. The circle cannot move to the right, since it is blocked by the gray wall, so its only option is to move down. Such a world can easily be represented by a 2D array.

Many older 2D games use a static drawing for the scene, which allows for only a

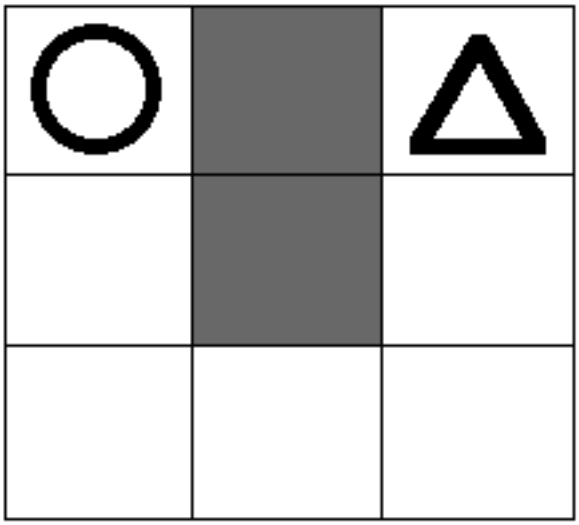

Figure 1. 2D grid.

single perspective, but it simplifies the process of drawing the scene. In comparison, 3D graphics require a large amount of mathematical calculations to render each scene.

Collision Detection Mathematics

A vector is essentially a directed line segment which has direction and magnitude. With graphics, this can be translated as the angle and length or distance. But the magnitude can also represent other factors such as the force or speed of an object. A scalar, as opposed to a vector, only has magnitude, but not direction.

Vectors differ from a point on a Cartesian plane, because the point represents only one spot on the plane, whereas a vector is the difference between two points on a plane. Vectors can be represented in a variety of ways, as shown in equations 1-1 through 1-3. A vector $\underline{\mathbf{V}}$ can be represented as the difference of two points P_1 and P_2 (1-1), the difference of the components of those two points (1-2), or by the constituent vector components (1-3). A 2D world has only x and y components, whereas a 3D world adds the z axis and another component to each vector. Figure 2 gives a visual representation of these equations on a Cartesian plane.

$$\underline{\mathbf{V}} = P_2 - P_1 \qquad (1\text{-}1)$$

$$\underline{\mathbf{V}} = (x_2 - x1, y_2 - y_1) \qquad (1\text{-}2)$$

$$\underline{\mathbf{V}} = (V_x, V_y) \qquad (1\text{-}3)$$

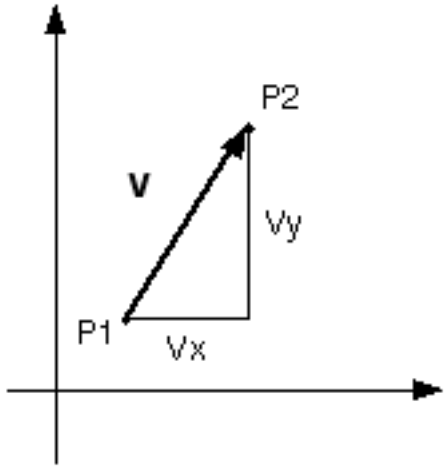

Figure 2. Vector representation.

Vectors play an important role in collision detection by determining how far apart objects are from each other. The magnitude or length of a vector can be found by taking the square root of the sum of the squares of each of the vector's components.

2D Vector: $|\underline{\mathbf{V}}| = \text{sqrt}(V_x^2 + V_y^2)$ (1-4)

3D Vector: $|\underline{\mathbf{V}}| = \text{sqrt}(V_x^2 + V_y^2 + V_Z^2)$ (1-5)

To normalize a vector, each component of the vector is divided by the magnitude of the vector. When all of the vector components are added together, they will equal 1. A plane's normal is important to provide realistic lighting and collision detection.

$V_x = V_x/|V|$ (1-6)

$V_y = V_y/|V|$ (1-7)

$V_z = V_z/|V|$ (1-8)

$V_x + V_y + V_z = 1$ (1-9)

Planar Equation: $Ax + By + Cz + D = 0$ (1-10)

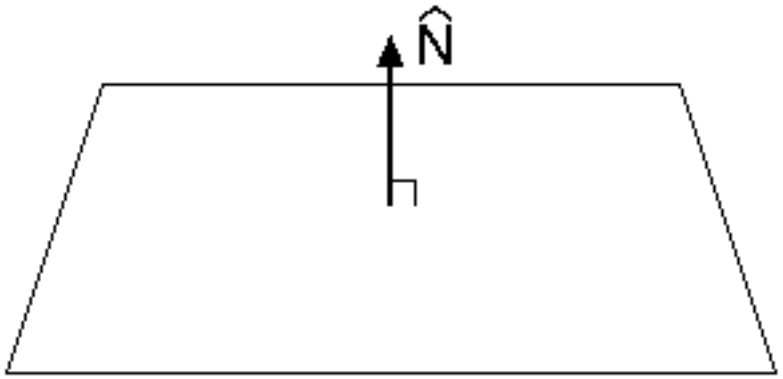

Figure 3. Plane normal.

A plane is defined by three non-collinear points. (x, y, z) from the planar equation are the coordinates of a point on the plane, and the coefficients A, B, C, and D are constants which describe the spatial properties of the plane. A, B, and C can also represent a vector (a, b, c) which is a normal of the plane.

Back-face Culling

Back-face culling is the process of not rendering a polygon if its face is turned away from the viewer. A point can be identified as being on the inside or outside of a plane surface according to the sign of the plane equation. If $Ax + By + Cz + D < 0$, then the point (x, y, z) is on one side of the plane surface. If $Ax + By + Cz + D > 0$, then the point (x, y, z) is on the other side of the plane surface. Otherwise, if the planar equation

is equal to zero, the point (x, y, z) is on the plane.

On the average, about half of the polygons in a scene will not be visible by the viewer. Eliminating these unseen polygons can help the performance of viewing the scene since not nearly as many polygons need to be rendered or be involved in the collision detection. This can save an enormous amount of time in calculating where an item may collide. One technique which can save time is the use of Binary Space Partitioning (BSP) trees, which divide up a world into convex hulls to remove unnecessary polygons [9].

To determine whether a plane is facing the viewer or not, the dot product is used to get the angle between the plane's normal and a vector from the viewer's position to a point on the plane. If the angle is between 90 and 270 degrees, then the polygon is facing the viewer. Otherwise, the polygon or plane is not facing the viewer.

Sphere – Plane Collision

One way to detect a collision in a 3D world is the sphere – plane detection method. The demonstration program which was created with PIGE used this technique to detect if the user had bumped into an object. This demo is explained further in chapter 5, and the source code is available in Appendix A. The sphere-plane method is relatively easy to compute since not every polygon of a more complex model has to be compared to the environment to see if a collision has occurred. The viewer or camera can be thought of as one solid entity, such as a ball, instead of a human with several limbs.

Detecting collisions with a sphere tends to be easier to calculate because of the symmetry of the object. The entire surface on a sphere is the same distance from the center, so it is easy to determine whether or not an object has intersected with a sphere. If the distance from the center of the sphere to an object is less than or equal to the sphere's radius, then a collision has occurred.

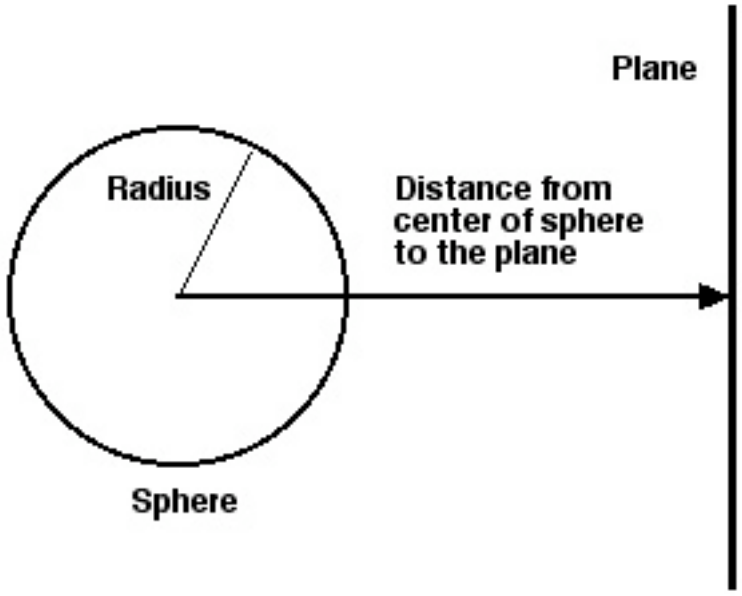

Figure 4. Sphere – plane collision.

The main point is not to let the sphere get too close to the plane. Before doing so, every plane needs to have its own normal vector and D value, which are taken from the planar equation (1-10).

The distance between a vertex, which is the sphere's center point in this case, and the plane is calculated by taking the dot product of the plane's normal and the sphere's position.

distance = plane.normal • sphere.position (1-11)

Depending on which side of the plane the sphere is on, the distance value can be either positive or negative. If the distance becomes zero, then the sphere is intersecting the plane, which is generally not a desirable effect when detecting a collision. This can be corrected by subtracting the sphere's radius from the distance.

Waiting for an object's distance to reach zero before detecting a collision will not always work. If the sphere's velocity is high, it might pass entirely through the plane on its next move. The way to check for this situation is to see if the distance to the plane has either turned negative or positive. If the sphere passed through the plane, the distance's numeric sign will change. If the numeric sign changes, then a collision occurred [2].

When checking for a collision, two points are important: the distance between the sphere and a plane should not become zero, and the numeric sign of the distance also should not change. If it does change, then the sphere has moved through the wall.

The game program will first check if a collision will result when the object moves in the desired direction. If there is a collision, then the program will respond appropriately, such as refusing to move in the desired direction.

Extensive source code implementing collision detection can be found on pages 66 through 68 in Appendix A.

Picking and Selection

With the creation of the Graphical User Interface came a new input device -- the mouse. The mouse would be rendered useless if the computer could not detect the user pressing a button and reacting appropriately to where the mouse cursor is on the screen.

GLUT makes it simple to check when a mouse button has been pressed. In the main function, the glutMouseFunc function specifies which function will respond and handle mouse events.

```
int main(int argc, char *argv[]))
{
    .
    .
    glutMouseFunc(mouse_function);
    .
    .
}
```

The mouse_function checks when the mouse button is pressed down and then calls the retrieveObjectID function to determine if a selectable object is underneath the mouse cursor.

```
if (state == GLUT_DOWN)
{
    objectID = retrieveObjectID(x, y);

    switch (objectID)
    {
        case PYRAMID: printf("Saw the pyramid\n"); break;
        case CUBE: printf("Saw the cube\n"); break;
        case TORUS: printf("Saw the torus\n"); break;
        default: printf("I didn't see anything\n"); break;
    }
}
```

The retrieveObjectID function returns the number of the closest item which was selected. If no item was selected, then 0 is returned.

```
int retrieveObjectID(int x, int y)
{
    int objectsFound = 0;
    GLint viewportCoords[4] = {0};
    GLuint selectBuffer[32] = {0};
```

The glSelectBuffer function registers the selection buffer, which is an array of size 32. Then the view port coordinates are retrieved with glGetIntegerv. Changing the matrix mode to projection allows the program to check the x and y coordinates of the mouse's position against the rendered 3D scene.

```
    glSelectBuffer(32, selectBuffer);
    glGetIntegerv(GL_VIEWPORT, viewportCoords);
    glMatrixMode(GL_PROJECTION);
```

A new combined transformation matrix is pushed on the model view stack so the 3D projection is not affected. The rendering mode is then changed to GL_SELECT. Any changes which are made will not affect the original copy of the scene since it is stored in the selectBuffer. The gluPickMatrix function creates a projection matrix which is around the cursor. This allows for rendering in only the region which is specified by the included parameters.

```
    glPushMatrix();
        glRenderMode(GL_SELECT);
        glLoadIdentity();
        gluPickMatrix(x, viewportCoords[3] - y, 2, 2, viewportCoords);
```

The gluPerspective function was originally called in the OpenGL initialization

function (see InitGL in Appendix A, page 60), but now it is called again so the perspective matrix is multiplied by the newly created pick matrix.

```
gluPerspective(45.0f, (float)kWindowWidth
               / (float)kWindowHeight, 0.1f, 150.0f);
```

Go back into the model view matrix, then render into the selective mode.

```
glMatrixMode(GL_MODELVIEW);
drawShapes(GL_SELECT);
```

When the render mode returns to GL_RENDER from the select mode, it returns the number of objects which were found under the mouse cursor. If no objects were found, then glRenderMode will return 0. If there was more than one object found, then the selectBuffer will need to be searched to find the closest object. The projection matrix is then returned back to normal.

```
objectsFound = glRenderMode(GL_RENDER);
glMatrixMode(GL_PROJECTION);
```

The current matrix is popped off to stop affecting the projection matrix, and then the normal model view matrix is reloaded.

```
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
```

If at least one object was found, then the function iterates through the selected items to find which one is closest to the viewer. If only one object was selected, then it

skips past the for loop and returns the selected object. Otherwise, zero is returned to indicate that nothing was found beneath the cursor.

```
if (objectsFound > 0)
{
    GLuint lowestDepth = selectBuffer[1];
    int selectedObject = selectBuffer[3];

    for (int i = 1; i < objectsFound; i++)
    {
        if (selectBuffer[(i*4)+1] < lowestDepth)
        {
            lowestDepth = selectBuffer[(i*4)+1];
            selectedObject = selectBuffer[(i*4)+3];
        }
    }
    return selectedObject;
}
```

If an object has been selected, it is up to the program to determine how to respond. For example, PIGE has four options on how to respond, depending on the state of the cursor: LOOK, TALK, DO, INV. If the cursor is over a lamb chop, LOOK would give a description of the lamb chop; TALK might either try and speak with the meat, or try and eat it; DO would likely pick it up; INV would use another inventory item, such as a knife, with the lamb chop.

Loading Textures

The ability to use textures brings amazing detail to what would otherwise be a bland, single-color polygon. Some of the first computer programs which experimented with 3D graphics were limited to wire frame or flat colored polygon models, as shown in

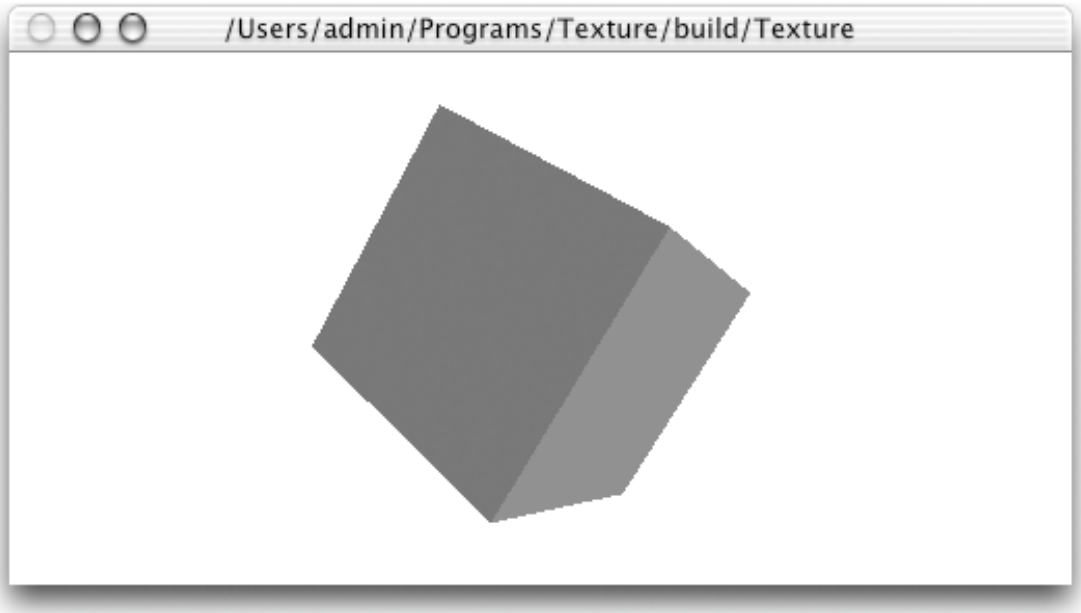

Figure 5. Cube without texture.

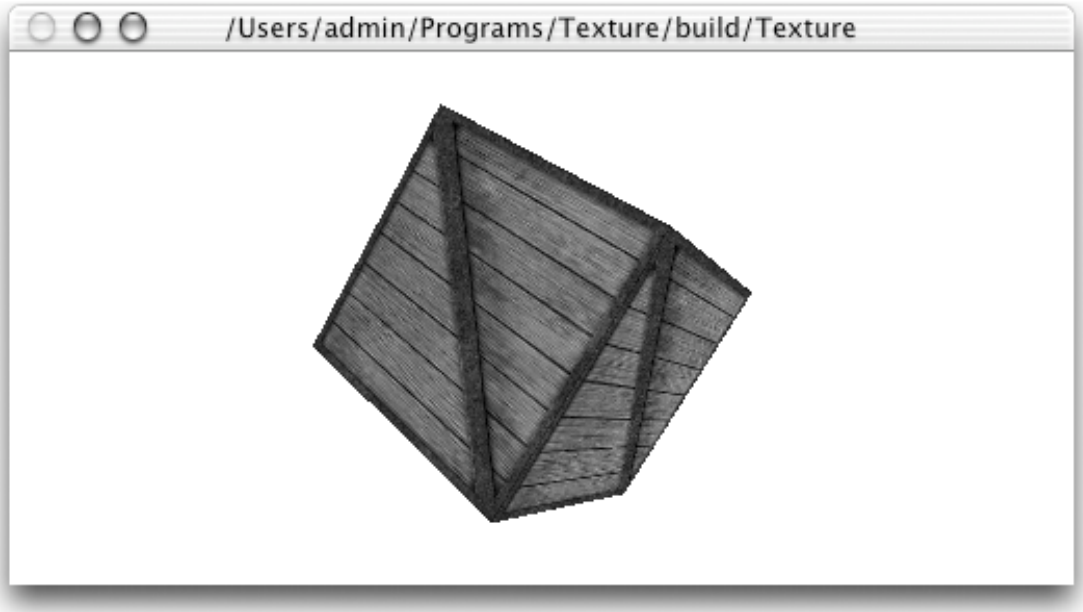

Figure 6. Cube with texture.

Figure 5. In retrospect, the cube in Figure 6 appears to be more realistic with the wood texture applied.

PIGE supports the use of Targa (TGA) images because of their high resolution and capability for transparency. GIF images also allow transparency, but have a limit of 256 colors, a severe limitation when monitors can display nearly 17 million colors. Transparency is a critical element, especially when displaying the icons, which are non-rectangular shapes such as a hand, lips, or an eye. It is because of these capabilities that PIGE loads TGA files instead of other image formats (GIF, BMP, JPG).

All images must have a height and width which is a power of 2 (2, 4, 8 ... 64, 128, 256) [12]. Every image in PIGE is 128 x 128 pixels in size. If an image does not fit these criteria, it will not load or display correctly.

To begin adding a texture into a program, a global array is declared, which will contain the image(s). If more than one texture is being loaded, then the array size needs to be set appropriately. For some systems, such as Windows and Linux, this variable is declared as an unsigned int instead of an unsigned long.

```
unsigned long     texture[1];
```

The TGAImageRec structure contains the data for the texture which will be loaded. The data variable holds the actual data, bpp is for the image color depth in bits per pixel, and the two size variables record the width and height of the image.

```
typedef struct TGAImageRec
{
    GLubyte       *data;      // Image Data (Up To 32 Bits)
    GLuint        bpp;        // Image Color Depth In Bits Per Pixel.
```

```
    GLuint        sizeX;      // Width of image
    GLuint        sizeY;      // Height of image
} TGAImageRec;
```

LoadGLTextures() is one of the two important functions used to load textures into a program. It is often called by the initializing function. The type variable is initialized to GL_RGBA, which allows for standard red, green, and blue colors, plus the option for an alpha transparency layer. A new TGAImageRec structure is then allocated to contain the new image.

```
void LoadGLTextures(unsigned long texture[])
{
    GLuint type = GL_RGBA;
    TGAImageRec *texture0;
```

Next, load an image using the LoadTGA function, which returns a TGAImageRec to be stored in texture0. LoadTGA will later be explained in further detail.

```
    texture0 = LoadTGA("crate.tga");
```

Now that the image has been loaded, the texture needs to be built. The glGenTextures call tells OpenGL to generate one texture. glBindTexture binds the named texture texture[0] to a target. Keep in mind that the first array entry starts at 0, not 1, in the C programming language.

```
    glGenTextures(1, &texture[0]);
    glBindTexture(GL_TEXTURE_2D, texture[0]);
```

The two glTexParameteri calls are OpenGL filters which specify how to

manipulate a texture when it is stretched or shrunk to proportions which are larger or smaller than the original texture.

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
```

If the bits per pixel is set to 24, then there is no specified alpha channel, so the type is set to RGB. However, if the bpp is 32, then the type is set to RGBA, which has an alpha channel that allows for transparency.

```
if ( texture0->bpp == 24 )
{
  type = GL_RGB;
}
else
{
  type = GL_RGBA;
}
```

Now the texture is finally created with glTexImage2D. Since this texture is a 2D image, it is set with GL_TEXTURE_2D. The second parameter is the detail, which tends to be set to 0. Type specifies the number of components in the image. For a standard RGB image, there are three components: Red, Blue, and Green. If there is an Alpha element, then there are four components. The next two parameters refer to the width and height of the image. After that is the border of the texture, which is generally set to 0. The type variable is used again to tell OpenGL that the image is composed of red, blue, green, and sometimes an alpha component. GL_UNSIGNED_BYTE reveals that the data is composed of unsigned bytes. The final parameter, texture0->data refers to where to retrieve the image data.

```
    glTexImage2D(GL_TEXTURE_2D, 0, type, texture0->sizeX,
    texture0->sizeY, 0, type, GL_UNSIGNED_BYTE, texture0->data);

}
```

LoadTGA is the second half to loading images. The LoadGLTextures function varies slightly when loading in other file formats, such as bitmaps, but the LoadTGA function is specific to only TGA files. Below are the variables used for the function. The variables of type GLuint can sometimes be replaced with a standard int, but that specification can vary from system to system. For consistency, all variables are declared as GLuint, which compiles well across all systems.

```
TGAImageRec*    LoadTGA( char *filename )
{
        Glubyte         TGAheader[12]={0,0,2,0,0,0,0,0,0,0,0,0};
        // Uncompressed TGA Header
        GLubyte         TGAcompare[12];
        GLubyte         header[6];
        GLuint          bytesPerPixel;
        GLuint          imageSize;
        GLuint          temp;
        GLuint          type = GL_RGBA;
        GLuint          i;
        TGAImageRec     *texture;
        FILE            *file;
```

The file is opened and checked to see if it was opened successfully. If not, the program returns an error to let the programmer know what went wrong. Some IDEs require that the images are included into the project, or the compiler will not be able to locate the images.

```
        file = fopen( filename, "rb" );

        if( ( file == NULL ) || // Does File Even Exist?
          ( fread( TGAcompare, 1, sizeof( TGAcompare ), file ) !=
```

```
    sizeof( TGAcompare ) ) ||   // Are There 12 Bytes To Read?
  ( memcmp(TGAheader, TGAcompare, sizeof( TGAheader )) != 0) ||
  ( fread( header, 1, sizeof( header ), file ) !=
    sizeof( header ) ) ) // If So Read Next 6 Header Bytes
{
        printf("Couldn't open file %s\n", filename);
        fclose( file );
        return NULL;
}

texture = ( TGAImageRec* )malloc( sizeof( TGAImageRec ) );

texture->sizeX  = header[1] * 256 + header[0];
texture->sizeY = header[3] * 256 + header[2];
```

By using careful error checking, if anything failed when checking the height, width, and bit depth to be valid, then the file is closed and the memory is freed.

```
if( ( texture->sizeX <= 0 ) || ( texture->sizeY <= 0 ) ||
  ( ( header[4] != 24 ) && ( header[4] != 32 ) ) )
{
        fclose( file );
        free( texture );
        return NULL;
}

texture->bpp = header[4];
bytesPerPixel = texture->bpp/8;

imageSize = texture->sizeX * texture->sizeY * bytesPerPixel;

texture->data = ( GLubyte* )malloc( imageSize );
```

If there is no data present, or it was misread, the function frees up the memory taken up by the texture. NULL is returned to indicate that an error occurred and the image could not be opened.

```
// Make sure the right amount of memory was allocated
if( ( texture->data == NULL ) ||
  ( fread(texture->data, 1, imageSize, file ) != imageSize) )
{
        // Free up the image data if there was any
        if( texture->data != NULL )
                free( texture->data );
```

```
                fclose( file );
                free( texture );
                return NULL;
        }
```

To avoid conflicts, imageSize is casted as an int from a Gluint. Mac OS X has no problems with imageSize being either a Gluint or an int, but Windows compilers will return this as an error. The Linux compiler gcc gives a warning, but it will still run with no noticeable problems. This loop swaps the red and blue elements so the colors will appear correctly.

```
        // Loop Through The Image Data
        for( i = 0; i < (int) imageSize; i += bytesPerPixel )
        {
                // Swaps The 1st And 3rd Bytes ('R'ed and 'B'lue)
                temp = texture->data[i];
                // Temporarily Store The Value At Image Data 'i'
                texture->data[i] = texture->data[i + 2];
                // Set The 1st Byte To The Value Of The 3rd Byte
                texture->data[i + 2] = temp;
                // Set The 3rd Byte To The Value In 'temp'
        }

        fclose( file ); // Close The File

        return texture;
}
```

Once the texture has been loaded into memory, it needs to be mapped onto a polygon inside the display function. The glBindTexture call decides which image, if there is more than one, to use for this particular polygon. glBindTexture cannot be declared between the glBegin and glEnd functions. The image needs to be mapped according to the polygon, or it might appear upside down, reversed, side ways, or not at all. When mapping, map in a counterclockwise fashion. This example starts with the

bottom left corner and rotates from there. Refer to Figure 7 for a diagram about mapping.

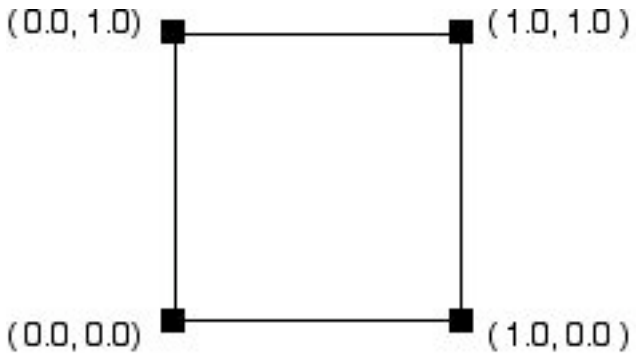

Figure 7. Texture mapping.

```
glBindTexture(GL_TEXTURE_2D, texture[0]);

glBegin(GL_QUADS);
      glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
      // Bottom Left Of The Texture and Quad
      glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
      // Bottom Right Of The Texture and Quad
      glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
      // Top Right Of The Texture and Quad
      glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
      // Top Left Of The Texture and Quad
glEnd();
```

Full source code for loading textures is available in Appendix C, starting on page 94.

CHAPTER 3

OPENAL

When deciding how to integrate audio capabilities with PIGE, there were two choices. A different audio API could be used for each operating system, or OpenAL could be used with each of the tested operating systems.

OpenAL was the logical choice, complimenting the PIGE methodologies, by being a platform-neutral, open-source, cross-platform API. OpenAL (Open Audio Library) [11] is a sister technology to OpenGL, striving to provide a vendor-neutral, cross-platform API for audio capabilities. This API is currently in the early stages of development, but it has effectively been used in several game ported to Linux and is supported by Creative Labs, a major sound board manufacturer.

As part of the effort to develop a game engine which utilizes as many platform compliant technologies as possible, the fairly young OpenAL was chosen to produce sounds. OpenGL has been around for several years and has had a large number of books and other references based upon it, but OpenAL is still in its adoption phase, and has not built up the following and resources that its graphics counterpart has. The only decent OpenAL tutorial was found at Dev-gallery.com [3], and even the examples provided by the site did not work properly. After extensive experimentation, the OpenAL demo worked on Windows, Linux, and Mac OS X. PIGE has produced documentation of the set up and construction of OpenAL on different platforms.

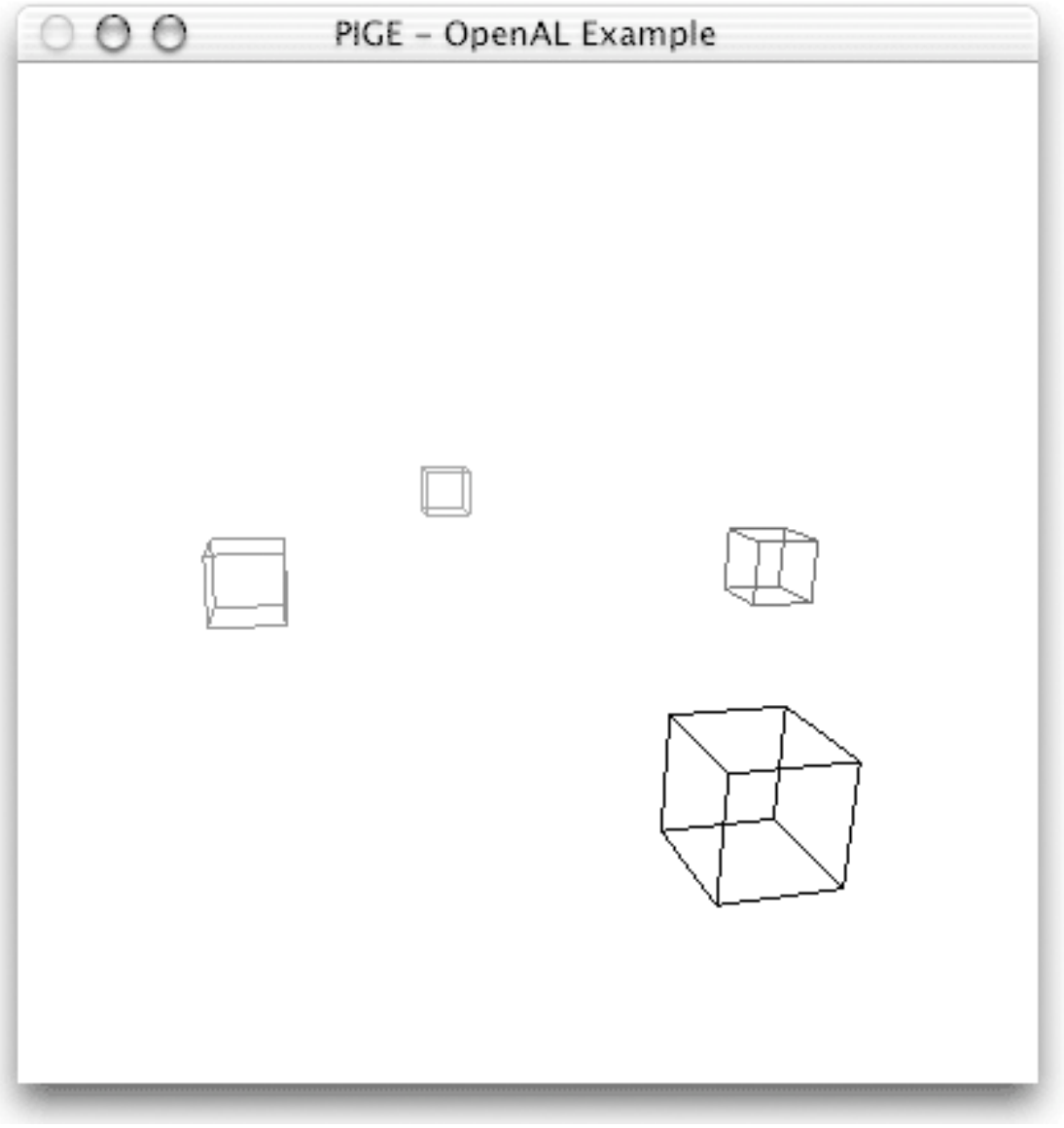

Figure 8. OpenAL demo screenshot.

The remainder of this chapter highlights the important sections of Appendix B to set up an OpenAL program.

The standard set of headers are these:

```
#include <stdio.h>
#include <stdlib.h>
#include <GL/glut.h>
#include <AL/al.h>
#include <AL/alut.h>
```

However, for Mac OS X, there are slight changes necessary for the GLUT and OpenAL libraries.

```
#include <stdio.h>
#include <stdlib.h>
#include <GLUT/glut.h>
#include <OpenAL/alut.h>
```

A standard has been developed to avoid the use of global variables, because they are generally considered bad programming style. This is to help avoid odd programming errors which can occur if global variables are used. When used with proper care, they can be useful and make programming an easier process, especially with OpenGL and OpenAL. Good practice dictates that most variables should remain local to several functions, and if the variables need to be shared, they are then passed as parameters. Such careful programming style can create unnecessary complications when working with OpenGL and OpenAL, so several global variables are made available to the program.

In this example, only one sound is being loaded, which explains why only one buffer and source is being declared.

```
#define NUM_BUFFERS 1
#define NUM_SOURCES 1
#define NUM_ENVIRONMENTS 1
```

There is only one listener, and these arrays define the initial set up of the listener for its position, direction, and velocity. The listener is not moving at the start, but can move around any other sound source.

```
ALfloat listenerPos[]={0.0,0.0,4.0};
ALfloat listenerVel[]={0.0,0.0,0.0};
Alfloat listenerOri[]={0.0,0.0,1.0, 0.0,1.0,0.0};
```

Each sound source has similar properties that the listener has. The source0Pos and source0Vel arrays show the position and velocity of the sound source.

```
ALfloat source0Pos[]={ -2.0, 0.0, 0.0};
ALfloat source0Vel[]={ 0.0, 0.0, 0.0};
```

Sounds need to be stored in an array, similar to textures. Several buffers are needed to contain the sounds and other necessary information. The size, freq, format, and data variables are used when loading the sound files.

```
Aluint  buffer[NUM_BUFFERS];
Aluint  source[NUM_SOURCES];
ALuint  environment[NUM_ENVIRONMENTS];

ALsizei size,freq;
ALenum  format;
ALvoid  *data;
```

The init function is called from the program's main function. It begins by setting the initial positions for the listener position, velocity, and orientation.

```
void init(void)
{
      alListenerfv(AL_POSITION,listenerPos);
      alListenerfv(AL_VELOCITY,listenerVel);
      alListenerfv(AL_ORIENTATION,listenerOri);
```

Next, the program creates the buffers and checks for any problems in the process.

```
alGetError(); // clear any error messages

// Generate buffers, or else no sound will happen!
alGenBuffers(NUM_BUFFERS, buffer);

if(alGetError() != AL_NO_ERROR)
{
      printf("- Error creating buffers !!\n");
      exit(1);
}
else
{
      printf("init() - No errors yet.");
}
```

The next three lines are the most important in loading the wave file. If more than one sound is loaded, then these three lines of code are repeated, but they reference different sound files and buffer positions. Refer to Table 3 on page 35 to view how the alutLoadWAVFile call differs between platforms.

```
alutLoadWAVFile("a.wav",&format,&data,&size,&freq);
alBufferData(buffer[0],format,data,size,freq);
alutUnloadWAV(format,data,size,freq);
```

Similar to how the buffers were created, the sources are now created and checked for any errors.

```
alGetError(); /* clear error */
alGenSources(NUM_SOURCES, source);

if(alGetError() != AL_NO_ERROR)
{
      printf("- Error creating sources !!\n");
      exit(2);
}
else
```

```
{
      printf("init - no errors after alGenSources\n");
}
```

Finally, the properties of the sound are set. First, the sound pitch and gain are set, then the source's position and orientation are defined, and finally the source is associated with its buffer to make it loop through the wave file. Setting the AL_LOOPING value to AL_TRUE determines whether or not the sound will play continuously, or if it will just play once and then stop. For ambient, background music, one might consider setting this to AL_TRUE, but for a sporadic sound such as a gun shot or a foot stomp, then AL_LOOPING should be AL_FALSE.

```
      alSourcef(source[0], AL_PITCH, 1.0f);
      alSourcef(source[0], AL_GAIN, 1.0f);
      alSourcefv(source[0], AL_POSITION, source0Pos);
      alSourcefv(source[0], AL_VELOCITY, source0Vel);
      alSourcei(source[0], AL_BUFFER,buffer[0]);
      alSourcei(source[0], AL_LOOPING, AL_TRUE);

}
```

CHAPTER 4

PORTING

The key to PIGE is the ease of porting a project from one operating system to another.

In a computing era where roughly 9 out of 10 of all consumer machines are running some version of Windows, there seems to be little incentive for the original developers to cater to the remaining 10% of the population. In response, most of the porting efforts come from dedicated Macintosh and Linux developers to convert Windows-native programs to other platforms.

When researching for examples to include into PIGE, difficulties were faced in translating other people's code into a more platform-neutral form. This required weeding through Windows and SDL[7] API code, extracting the parts which could be left untouched, and then effectively 'translating' the rest of the code into standard OpenGL, GLUT, and OpenAL commands.

This operation was relatively minor in comparison to what programmers have to endure when porting a large application. Well-written, modular code can be ported in a relatively short amount of time. The biggest difficulties arise when the original application was written in a non-modular or unportable manner, which makes the porting increasingly more difficult. It takes a lot of time to convert large portions of code to another machine's native API, instead of just having to deal with small, modular sections. As intended with PIGE, if a particular section needs to be rewritten, that small

section can be manipulated instead of having to replace large portions of the program to ensure that everything will continue to work correctly.

Despite all attempts to keep this program as non-platform specific as possible, there were small areas that required specific code features. One of the most difficult parts in porting was the initial set up of the systems and compilers to work with OpenGL and OpenAL. Mac OS X, with Apple's Project Builder IDE, was by far the easiest to configure. The only configuration which was necessary was to add several frameworks to the project (Foundation, GLUT, OpenGL, and OpenAL) and making sure the proper libraries were included in the source code.

Linux, alongside with the gcc compiler, took several hours of investigation to determine the proper paths and libraries to include, but once this matter was corrected, the example programs were relatively simple to create, compile, and run.

Windows, along with CodeWarrior, proved to be the most challenging to meet PIGE's goals. Microsoft’s Visual C++ tends to be the dominant compiler for Windows, and such, more references and helpful resources are available for VC++. The limited help found for CodeWarrior generally referred to the Macintosh version. Corresponding documentation for CodeWarrior and Windows98 was difficult to obtain, and while Linux documentation was plentiful, it tended to be quite scattered.

OpenGL has been around for over a decade, so there is a plethora of documentation available. OpenAL, however, is still a nascent technology, so help documents and references, both in print and on-line, are nearly nonexistent for all platforms.

OpenGL proved its ease of portability by offering fairly little resistance. The most common conflict was a difference of type declaration, which was easily fixed by changing a GLuint to a standard int, or an unsigned long to an unsigned int.

The picking and selection demo, which was detailed in the second chapter, had absolutely no problems in porting. It compiled and ran on all systems with the only modification being the header libraries.

For Windows and Linux, an unsigned int was needed for the texture in the texture demonstration. Linux would compile and run with an unsigned long, but it gave a warning. For Mac OS X, an unsigned long, not an unsigned int, was needed. This difference extended into the LoadTGA function which needed different parameters to accommodate the various operating systems.

OpenAL is a little more limited in its scope of what it accomplishes, so the range of problems is considerably smaller than that of OpenGL. The only major difference which was encountered was the way the alutLoadWAVFile() function needed to be called. All three platforms had a different set of parameters when calling alutLoadWAVFile, as shown in Table 3.

Operating System	Source Code
Linux	`alutLoadWAVFile((Albyte *) "c.wav", &format, &data, size,&freq, &al_bool);`
Macintosh	`alutLoadWAVFile("c.wav",&format,&data,&size,&freq);`
Windows	`alutLoadWAVFile("c.wav",&format,&data,&size,&freq, &al_bool);`

Table 3. alutLoadWAVFile differences.

The al_bool variable is a different type for Linux and Windows. Linux declares the al_bool as an ALboolean, but Windows declares it as a char. If the last parameter in the alutLoadWAVFile call for Windows is the number 0, it will return an unhandled exception, as shown in Figure 9.

Figure 9. OpenAL unhandled exception.

Another problem illustrated how forgiving or merciless some compilers decide to be with code. At the beginning of the init() function, alutInit(0, NULL) was declared and Mac OS X produced no errors. The Linux version compiled, but it would not play the wave files. Instead, it would return this error:

```
open /dev/[sound/]dsp: Resource temporarily unavailable
fcntl: Bad file descriptor
```

Once the alutInit(0, NULL) declaration was removed, the Linux warning disappeared, and the Mac OS X compilation returned no complications after the change.

One of the most unexpected problems to arise came from moving PIGE from Mac OS 10.1 to 10.2. There were enough internal changes to 10.2 that the Mac OS X

build of PIGE would crash. The problem with the PIGE demonstration dealt with entering the glutGameMode and full-screen mode. Downloading the latest version of GLUT 3.0.1 for Mac OS X solved this problem. Mac OS 10.2 was not the only operating system to have this problem, as RedHat 8.0 also crashed with the same problems. This is an area where the most current technology needs to be maintained, or an alternative route needs to be taken to force PIGE to go into full screen mode. System specific APIs such as Cocoa for Mac OS X or SDL for Linux can also be used to activate the full screen mode.

There were three other problems between OS 10.1 and 10.2. The glutSetCursor(GLUT_CURSOR_NONE) call makes the default cursor disappear, but this did not work under OS 10.2, yet it worked fine under OS 10.1. Another difference was OS 10.1 did not recognize the Boolean values FALSE and false to be the same. The last conflict between these two OS versions was the executable files were not backwards compatible. For a demonstration project to run on OS 10.1, it needed to be recompiled, and then it would run fine after that. Yet, it was unexpected to see such a problem occur between the minor point releases of an operating system.

PIGE adheres to the modularity paradigm by being able to take individual parts and seamlessly integrate them together. Only one problem resulted when the major graphics and audio components of PIGE were combined. The following code, which is responsible for drawing the cursor on top of the rendered 3D scene, needed to be outside of the drawShapes function, or the picking and selection would not work properly. When this code was in the drawShapes function, and the mouse was clicked, the

retrieveObjectID function would always return the last loaded name identification number, even if another object was selected.

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

  draw_cursor(mouse_x, mouse_y);

glDepthMask( GL_TRUE );
glDisable(GL_BLEND);
glDisable(GL_TEXTURE_2D);
```

In the end, the problems and inconsistencies in using PIGE between the platforms are still much less than the problems which can occur when porting from one system's native API to another. The largest concern is the amount of support for OpenGL, GLUT, and OpenAL on each of these operating systems.

CHAPTER 5

IMPLEMENTATION

In the end, none of the discussed technologies are of much practical use if they cannot be combined into a coherent product. The first project which PIGE is being used for is for the game Psychomancer.

Psychomancer was initially developed as a text-based adventure/role-playing game during the summer of 1999. It has expanded and improved since its creation, but it has also remained text-based, which is no longer an acceptable presentation medium.

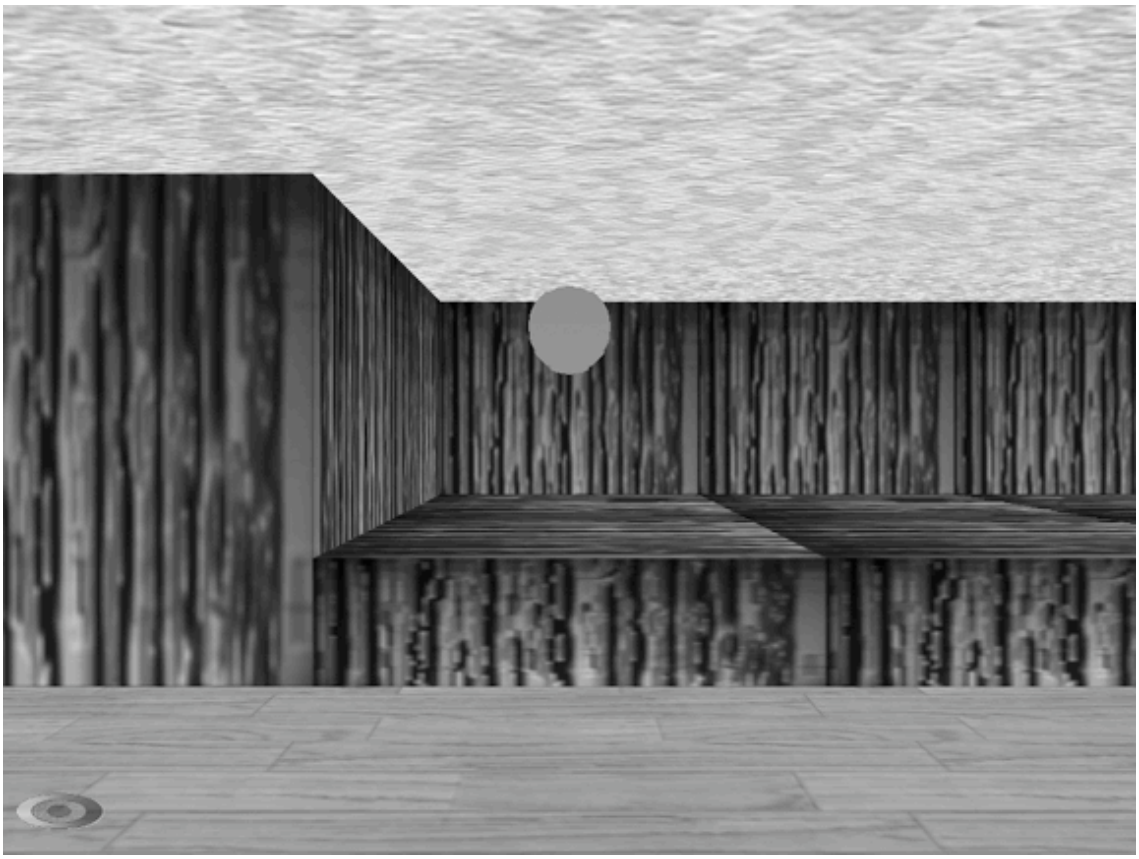

Figure 10. Psychomancer screenshot.

Figure 10 shows an example screenshot of the game, implemented with PIGE.

On the base level, this scene looks fairly simple, but there is an enormous amount of work required beforehand to reach even this point. It was discovered during the creation of Psychomancer that the core game engine needs to be constructed first before any application can be created. Once the core technologies are constructed and in place, then the design of the product can finally begin.

This has proven to be no different with game companies who have spent an enormous amount of time in just constructing the technologies. One of the key roles PIGE is trying to provide is to reduce the amount of time spent on creating gaming technologies, thus avoiding recreating the wheel, and allowing the developer to concentrate further on the actual design of the game.

Figure 11 on page 41 illustrates the connection between the game engine and its components. PIGE provides for the graphics and sound aspects of a game engine. PIGE was initially created as several smaller programs to concentrate individually on elements of graphics or sound. Each program focused on getting some smaller element of PIGE working before the pieces were to be combined together. Appendices B and C are two of these demonstrations which were created. Appendix A is the combined result of these many smaller programs to form the Psychomancer demo.

It is up to the discretion of the programmer to manipulate the tools provided by PIGE to suit their needs. PIGE provides the core foundation for graphics and sound capabilities, but the game play element is dependent upon the style of the game being created. An example presented in the Psychomancer demo is that it has four different

icons (LOOK, TALK, DO, INV) to interact with selectable items. A first person shooter (FPS) game could likely reduce this to one or two icons since the interaction in a FPS tends to be more limited and direct than the more complex interaction system provided by an adventure or role-playing game.

Another difference between game genres is the inventory system. A FPS may only provide for a simple inventory (types of weapons and ammunition), whereas an adventure game would tend to have a more complex inventory system. Other games,

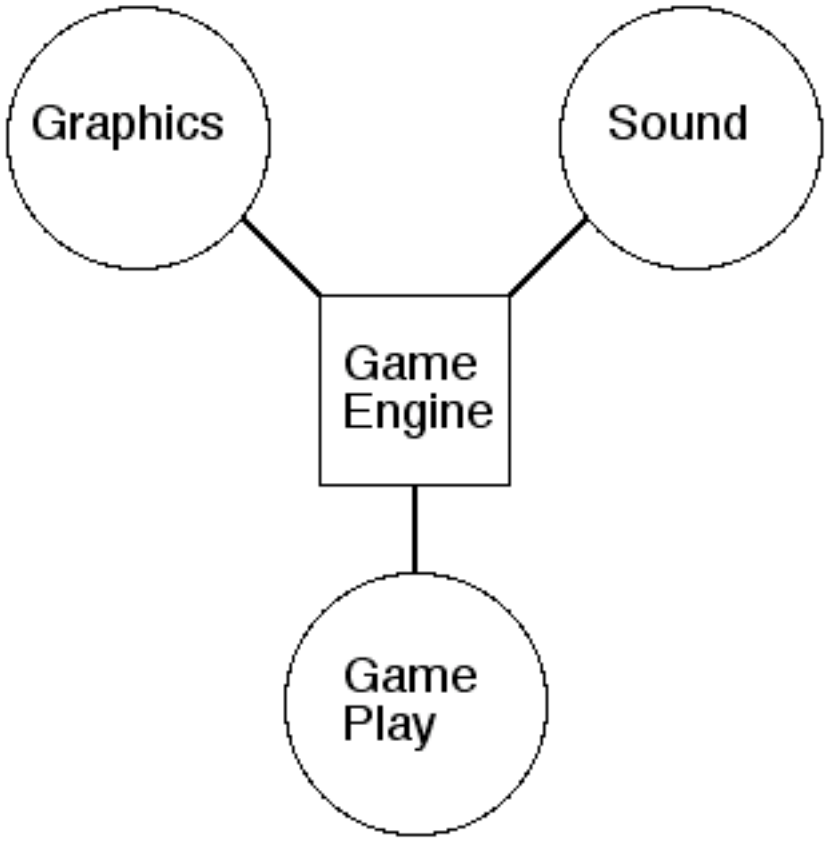

Figure 11. Game engine diagram.

like strategy or racing, may only maintain a few statistics for an inventory, such as money, lumber, or type of vehicle.

PIGE is similar to how many other APIs provide the tools to ease the process of creating an application. Once these tools are available, it is up to the programmer to decide how to make use of them.

CHAPTER 6

CONCLUSION

Evaluation of PIGE

The goal of PIGE was to provide a set of tools that would be useful in designing a computer game engine which would run on the Windows, Linux, and Macintosh operating systems. On most levels, this has been a successful endeavor. The construction of a full-featured game engine would take the effort of multiple man-years to reach a professional level of quality. PIGE is a demonstration that a platform neutral game engine is possible.

Not only did PIGE prove the feasibility of a platform independent game engine, but it also created a source of documentation to teach others about setting up and using OpenGL and OpenAL. This is particularly an important area for OpenAL since there has been so little documentation created for that API.

The largest problem facing the feasibility of PIGE is the variable amount of support for OpenGL, GLUT, and OpenAL on each platform. The support ranges from system to system, so particular features, such as glutEnterGameMode() or glutSetCursor(), may not have consistent results across all systems. For such problems, alternate solutions need to be found, which may result in resorting to some system-specific API code. This is an undesirable answer to the problem, but it may be necessary if there are no other suitable solutions.

During the creation of PIGE, the project and its tutorials received some attention from people who were also trying to use OpenGL or OpenAL on Mac OS X. Since OpenAL is still a fairly young technology, it has not garnered a significant amount of attention and documentation from developers. As of this writing, the PIGE web site has become the only centralized location to discuss how to set up projects for both OpenGL and OpenAL on Windows, Linux, and Macintosh.

Future

The gaming industry is always evolving, continually pushing the mantra "Bigger, Better, Faster, More" in the attempt to be the first to bring to market the latest and greatest in gaming technology. This current incarnation of PIGE is in no small way limited to just what is currently available. PIGE has been designed as a starting platform from where to launch and extend projects.

Future additions to PIGE are, but not limited to, the ability to read in models (OBJ, Milkshape, Quake), creating a world editor, advanced lighting techniques, further game physics, fonts, binary space partitions (BSP), shadowing effects, importing other image types, networking, and optimizations for specific computer hardware.

One of the most time consuming parts of creating an OpenGL scene with PIGE is that it needs to be created by hand. A more efficient way to circumvent this problem is to follow the same technique many game developers use to design their game worlds – by using a world editor, which easily allows the level designer to piece together a scene in far less time than it would take to code by hand.

Even though the creation of PIGE was enhanced through tools such as OpenGL, GLUT, and OpenAL, they are not always as powerful as the system APIs, and if these OS specific APIs are used, they should conform to PIGE's standards to remain modular and easy to swap out with other tools when necessary.

PIGE was developed primarily to create games. PIGE will be used to continue the development for the graphical front end of Psychomancer.

The PIGE website is located at http://www.edenwaith.com/products/pige/ [4], containing all of the tutorials, source code, and example projects. Further work on PIGE will be archived at this website.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Angel, Edward. OpenGL : A Primer. Addison-Wesley, 2002.

[2] Cfxweb.net. “3D Graphics Programming.” http://www.cfxweb.net/~cfxamir/tutorials.html.

[3] Dev-gallery.com. “Using OpenAL and OpenGL to create a 3D application with 3D sound”. http://www.dev-gallery.com/programming/openAL/basic/basicOpenAL_1.htm.

[4] Edenwaith.com. "PIGE". http://www.edenwaith.com/products/pige/.

[5] fsf.org. "GNU General Public License." http://www.fsf.org/copyleft/gpl.html.

[6] Garagegames.com. "Torque Game Engine SDK." http://www.garagegames.com/pg/product/view.php?id=1.

[7] Hall, John R. Programming Linux Games. Linux Journal Press, 2001.

[8] idsoftware.com. “id Software’s Technology Licensing Program”. http://www.idsoftware.com/business/home/technology/techlicense.php.

[9] Lengyle, Eric. Mathematics for 3D Game Programming & Computer Graphics. Charles River Media, Inc., 2002.

[10] NeHe Productions. “NeHe Productions”. http://nehe.gamedev.net.

[11] OpenAL.org. “OpenAL home page”. http://www.openal.org.

[12] OpenGL.org. “OpenGL home page”. http://www.opengl.org.

[13] PyOpenGL. “OpenGL.GLUT.glutSpecialUpFunc” http://pyopengl.sourceforge.net/documentation/ref/glut/glutspecialupfunc.html.

[14] Szymczyk, Mark. Mac Game Programming. Premier Press, 2002.

[15] Woo, Mason, Jackie Neider, Tom Davis, Dave Shreiner. OpenGL Programming Guide, Third Edition. Addison-Wesley, 1999.

APPENDICES

APPENDIX A

PIGE SOURCE CODE

```
// ========================================================================
// main.cpp
// ========================================================================

#include <pige_mac.h>

// #include <pige_linux.h>
// #include <pige_win.h>

// ========================================================================
// Global Variables
// ========================================================================
#define NUM_BUFFERS 2
#define NUM_SOURCES 2
#define NUM_ENVIRONMENTS 1

ALfloat listenerPos[]           = {0.0,0.0,4.0};
ALfloat listenerVel[]           = {0.0,0.0,0.0};
ALfloat listenerOri[]           = {0.0,0.0,1.0, 0.0,1.0,0.0};

ALfloat source0Pos[]            = { -2.0, 0.0, 0.0};
ALfloat source0Vel[]            = { 0.0, 0.0, 0.0};

ALfloat source1Pos[]            = { 2.0, 0.0, 0.0};
ALfloat source1Vel[]            = { 0.0, 0.0, 0.0};

ALuint  buffer[NUM_BUFFERS];
ALuint  source[NUM_SOURCES];
ALuint  environment[NUM_ENVIRONMENTS];

ALsizei size,freq;
ALenum  format;
ALvoid  *data;

TGAImageRec textures[32];       // for the cursor icon

enum cursor_types {DO, LOOK, TALK, INVENTORY};
enum cursor_types cursor_type = LOOK;

int             light1_on       = 0; // 0 is on, 1 is off
int             window;
int             mouse_entry     = 1;
int             kWindowWidth    = 1024;
int             kWindowHeight   = 768;
static int      mouse_x         = 0;
static int      mouse_y         = kWindowHeight;
int             fullscreen      = 0;
int             isDirectionalKeyDown = false;
int             directional_key;

float           cursor_x        = 0;
float           cursor_y        = 0;
float           cursor_z        = 0.0;
```

```
float           old_x           = 0;
float           old_y           = kWindowHeight;
float           mouse_sensitivity= 0.0009799849;
float           mouse_sensitivity_y= 0.0013;
float           cursor_size     = 0.10;

float           x_camera        = 0.0;
float           y_camera        = -0.5;
float           z_camera        = -3.0;

int             x_angle         = 0;
int             y_angle         = 0;
int             z_angle         = 0;

int             lights_on       = false;
int             is_light_on     = true;

int             isSphereNoisy   = false;

int             objectID        = 0;

int             random_red      = get_random(0, 255);
int             random_green    = get_random(0, 255);
int             random_blue     = get_random(0, 255);

int             sphere_red      = 255;  // The three RGB components of
int             sphere_green    = 255;  // the floating sphere which
int             sphere_blue     = 255;  // will change colors randomly
int             sphere_direct   = 1;    // 1 the sphere is going up
                                        // 0 the sphere is going down
float           sphere_height   = 0.0;

float           l1z_pos         = 3.5;

GLfloat         specular[]      = { 1.0, 1.0, 1.0, 1.0 };
GLfloat         shininess[]     = { 100.0 };
GLfloat         LightAmbient[]  = { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat         LightDiffuse[]  = { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat         LightPosition[] = { 0.0f, 0.5f, l1z_pos, 1.0f };

GLfloat         xrot;                   // X Rotation
GLfloat         yrot;                   // Y Rotation
GLfloat         xspeed;                 // X Rotation Speed
GLfloat         yspeed;                 // Y Rotation Speed

GLfloat         walkbias        = 0;
GLfloat         walkbiasangle   = 0;

GLfloat         lookupdown      = 0.0f;
const float     piover180       = 0.0174532925f;

float           heading         = 0.0;
float           xpos            = 0.0;
float           zpos            = 3.0;
```

```
Vector3D        egoPosition(0.0, 0.0, 3.0);
Vector3D        egoVelocity(0.0, 0.0, 0.05);
float           egoRadius       = 1.5;


// ======================================================================
// Prototypes
// ======================================================================

// display, draw, & initialize
void draw_cursor(int, int);
void display();
GLvoid drawShapes(GLenum mode);
void InitGL();
void reshape(int w, int h);
void Idle();
// input functions
void processMouse(int button, int state, int x, int y) ;
int retrieveObjectID(int x, int y);
void PassiveMouseFunc(int, int);
void EntryFunc(int state);
void keyboard(unsigned char key, int x, int y);
void specialKeys(int key, int x, int y);
void specialKeysUp(int key, int x, int y);


// ======================================================================
// int main(int, char *[])
// ======================================================================
int main(int argc, char *argv[])
{

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowSize(kWindowWidth, kWindowHeight);

    alutInit(&argc, argv) ;

    window = glutCreateWindow("Psychomancer");

    glutGameModeString("1024x768:24@72");
    glutEnterGameMode();

    glutSetCursor(GLUT_CURSOR_NONE); // make the cursor disappear

    InitGL();
    openal_init();

    glutReshapeFunc(reshape);
    glutIdleFunc(Idle);
    glutDisplayFunc(display);
    glutMouseFunc(processMouse);
    glutPassiveMotionFunc(PassiveMouseFunc);
```

```
    glutMotionFunc(MovingMouseFunc);
    glutEntryFunc(EntryFunc);
    glutKeyboardFunc(keyboard);
    glutSpecialFunc(specialKeys);
    glutSpecialUpFunc(specialKeysUp);
    glClearColor(0, 0, 0, 0);
    glutMainLoop();

    return 0;

}



// ====================================================================
// void draw_cursor(int x, int y)
// ====================================================================
void draw_cursor(int x, int y)
{
    if (cursor_type == DO)
    {
        glBindTexture(GL_TEXTURE_2D, texture[0]);
    }
    else if (cursor_type == LOOK)
    {
        glBindTexture(GL_TEXTURE_2D, texture[1]);
    }
    else if (cursor_type == TALK)
    {
        glBindTexture(GL_TEXTURE_2D, texture[2]);
    }
    else // INVENTORY ITEM
    {
        glBindTexture(GL_TEXTURE_2D, texture[1]); // 3
    }


    glMatrixMode(GL_PROJECTION);        // Select Projection
    glPushMatrix();                     // Push The Matrix
    glLoadIdentity();                   // Reset The Matrix
    glOrtho( 0, 1 , 0 , 1, -1, 1 );    // Select Ortho Mode (640x480)
    glMatrixMode(GL_MODELVIEW);         // Select Modelview Matrix
    glPushMatrix();                     // Push The Matrix
    glLoadIdentity();

    glBegin (GL_QUADS);

        glColor4f(1.0, 1.0, 1.0, 0.5);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3f(cursor_x, cursor_y, 1.0);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3f(cursor_x + cursor_size, cursor_y, 1.0);
        glTexCoord2f(1.0f, 1.0f);
        glVertex3f(cursor_x + cursor_size, cursor_y+cursor_size, 1.0);
```

```
        glTexCoord2f(0.0f, 1.0f);
        glVertex3f(cursor_x, cursor_y + cursor_size, 1.0);

    glEnd();

    glMatrixMode( GL_PROJECTION );      // Select Projection
    glPopMatrix();                      // Pop The Matrix
    glMatrixMode( GL_MODELVIEW );       // Select Modelview
    glPopMatrix();

    glutPostRedisplay();
}


// ====================================================================
// void check_colors()
// ====================================================================
void check_colors()
{
    int correct_color = 0;

    if (sphere_red < random_red)
    {
        sphere_red++;
    }
    else if (sphere_red > random_red)
    {
        sphere_red--;
    }
    else
    {
        correct_color++;
    }

    if (sphere_green < random_green)
    {
        sphere_green++;
    }
    else if (sphere_green > random_green)
    {
        sphere_green--;
    }
    else
    {
        correct_color++;
    }

    if (sphere_blue < random_blue)
    {
        sphere_blue++;
    }
    else if (sphere_blue > random_blue)
    {
        sphere_blue--;
```

```
    }
    else
    {
        correct_color++;
    }

    // new color as been met.  Make a new random color
    if (correct_color >= 3)
    {
        random_red   = get_random(0, 255);
        random_blue  = get_random(0, 255);
        random_green = get_random(0, 255);
    }

}

// ====================================================================
// void display()
// Call non-transparent items first, then set GL_BLEND, then call
// the transparent/translucent items
// ====================================================================
void display(void)
{
    // Clear The Screen And The Depth Buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    drawShapes(GL_RENDER);

    // The following part cannot be in drawShapes() or the objectID
    // function will not work properly and return the properly
    // selected item
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glDepthMask(GL_FALSE);

      draw_cursor(mouse_x, mouse_y);

    glDepthMask( GL_TRUE );
    glDisable(GL_BLEND);
    glDisable(GL_TEXTURE_2D);

    glutSwapBuffers();
    glFlush();
}


// ====================================================================
// drawShapes(GLenum mode)
// ====================================================================
GLvoid drawShapes(GLenum mode)
{
    glInitNames();
    glPushName(0);
```

```
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);

glLoadIdentity();
    glTranslatef(0.5f,0.0f, 0.5f);

    glTranslatef(-xpos, y_camera, -zpos);

    if (GL_SELECT == mode)
    {
        glLoadName(3);
    }

    glColor3f(0.0f,0.0f,1.0f);
    glutSolidTorus(0.5, 3.0, 20, 20);

glLoadIdentity();

glPushMatrix();

    glRotatef((float)x_angle, 1.0, 0.0, 0.0);
    glRotatef(360.0f - yrot, 0.0, 1.0, 0.0);
    glRotatef((float)z_angle, 0.0, 0.0, 1.0);

    glTranslatef(-xpos, y_camera, -zpos);

// draw objects which do not utilize textures

glPushMatrix();

    if (GL_SELECT == mode)
    {
        glLoadName(1);
    }

    check_colors();
    glColor3f((float)sphere_red/255.0, (float)sphere_green/255.0,
              (float)sphere_blue/255.0);

    glTranslatef(0.0, 0.5 + sphere_height, 0.5);

    glutSolidSphere(0.1, 20, 20);

    if (1 == sphere_direct)
    {
        if (sphere_height < 0.25)
        {
            sphere_height += 0.005;
        }
        else
        {
            sphere_direct = 0;
        }
    }
    else
```

```
    {
        if (sphere_height > 0.0)
        {
            sphere_height -= 0.005;
        }
        else
        {
            sphere_direct = 1;
        }
    }

glPopMatrix();

if (lights_on == FALSE)
{
    glColor3f(1.0f, 1.0f, 1.0f);
}

if (GL_SELECT == mode)
{
    glLoadName(2);
}

// now draw objects which utilize textures
glEnable(GL_TEXTURE_2D);

glBindTexture(GL_TEXTURE_2D, texture[4]);


    glBegin (GL_QUADS);

    // (0,1)--------(1,1)
    //   |            |
    // (0,0)--------(1,0)

    // floor
    for (int i = 0; i < 5; i++)
    {
        for (int j = 0; j < 5; j++)
        {
        glNormal3d(0.0, 1.0, 0.0);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-2.5+i, 0.0, 0.0+j);
        glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.5+i, 0.0, 0.0+j );
        glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.5+i, 0.0, -1.0+j);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-2.5+i, 0.0, -1.0+j);

        }
    }



    glEnd();

    // bind the ceiling texture
```

```
 glBindTexture(GL_TEXTURE_2D, texture[5]);

 glBegin(GL_QUADS);
 // ceiling
 for (int i = 0; i < 5; i++)
 {
     for (int j = 0; j < 5; j++)
     {
         glNormal3d(0.0, -1.0, 0.0);
         glTexCoord2f(0.0f, 0.0f);
         glVertex3f(-2.5 + i, 1.0, 0.0 + j);
         glTexCoord2f(1.0f, 0.0f);
         glVertex3f(-1.5 + i, 1.0, 0.0 + j );
         glTexCoord2f(1.0f, 1.0f);
         glVertex3f(-1.5 + i, 1.0, -1.0 + j);
         glTexCoord2f(0.0f, 1.0f);
         glVertex3f(-2.5 + i, 1.0, -1.0 + j);

     }
 }
 glEnd();

// Remember: this call needs to be outside the glBegin - glEnd
glBindTexture(GL_TEXTURE_2D, texture[3]);

 glBegin(GL_QUADS);
 // left & right walls
 for (int i = 0; i < 2; i++)
 {
     for (int j = 0; j < 5; j++)
     {
         if (0 == i)
         {
             glNormal3d(-1.0, 0.0, 0.0);
         }
         else
         {
             glNormal3d(1.0, 0.0, 0.0);
         }

         glTexCoord2f(0.0f, 0.0f);
         glVertex3f(-2.5 + i*5, 1.0, 0.0 + j);
         glTexCoord2f(1.0f, 0.0f);
         glVertex3f(-2.5 + i*5, 0.0, 0.0 + j);
         glTexCoord2f(1.0f, 1.0f);
         glVertex3f(-2.5 + i*5, 0.0, -1.0 + j);
         glTexCoord2f(0.0f, 1.0f);
         glVertex3f(-2.5 + i*5, 1.0, -1.0 + j);
     }
 }

 // front and back walls
 for (int i = 0; i < 2; i++)
 {
```

```
    for (int j = 0; j < 5; j++)
    {
        if (0 == i)
        {
            glNormal3d(0.0, 0.0, -1.0);
        }
        else
        {
            glNormal3d(0.0, 0.0, 1.0);
        }

        glTexCoord2f(0.0f, 0.0f);
        glVertex3f(-1.5 + j, 1.0, -1.0 + i*5);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3f(-1.5 + j, 0.0, -1.0 + i*5);
        glTexCoord2f(1.0f, 1.0f);
        glVertex3f(-2.5 + j, 0.0, -1.0 + i*5);
        glTexCoord2f(0.0f, 1.0f);
        glVertex3f(-2.5 + j, 1.0, -1.0 + i*5);
    }
}


// raised floor
for (int i = 0; i < 5; i++)
{
    for (int j = 0; j < 2; j++)
    {
        glNormal3d(0.0, 1.0, 0.0);
        glTexCoord2f(0.0f, 0.0f);
        glVertex3f(-2.5 + i, 0.25, 0.0 + j);
        glTexCoord2f(1.0f, 0.0f);
        glVertex3f(-1.5 + i, 0.25, 0.0 + j );
        glTexCoord2f(1.0f, 1.0f);
        glVertex3f(-1.5 + i, 0.25, -1.0 + j);
        glTexCoord2f(0.0f, 1.0f);
        glVertex3f(-2.5 + i, 0.25, -1.0 + j);
    }
}

// front panels to raised area
for (int i = 2; i < 5; i++)
{
    glNormal3d(0.0, 0.0, -1.0);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.5 + i, 0.25, 1.0);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.5 + i, 0.0, 1.0 );
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-2.5 + i, 0.0, 1.0 );
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-2.5 + i, 0.25, 1.0);

}

// front part of bedroom wall
for (int i = 0; i < 2; i++)
{
```

```
            glNormal3d(0.0, 0.0, -1.0);
            glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.5 + i, 1.0, 1.0);
            glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.5 + i, 0.0, 1.0 );
            glTexCoord2f(1.0f, 1.0f); glVertex3f(-2.5 + i, 0.0, 1.0 );
            glTexCoord2f(0.0f, 1.0f); glVertex3f(-2.5 + i, 1.0, 1.0);

        }

        // side of bedroom wall
        for (int i = 0; i < 2; i++)
        {
            glNormal3d(-1.0, 0.0, 0.0);
            glTexCoord2f(0.0f, 0.0f); glVertex3f(-0.5, 1.0, 0.0 + i);
            glTexCoord2f(1.0f, 0.0f); glVertex3f(-0.5, 0.0, 0.0 + i );
            glTexCoord2f(1.0f, 1.0f); glVertex3f(-0.5, 0.0, -1.0 + i);
            glTexCoord2f(0.0f, 1.0f); glVertex3f(-0.5, 1.0, -1.0 + i);
        }

        glEnd();

    glPopMatrix();


}

// ========================================================================
// void InitGL()
// ========================================================================
void InitGL()
{

    LoadGLTextures(texture);

    // move the cursor to the set (x,y) coordinates
    glutWarpPointer(kWindowWidth/2, kWindowHeight/2);

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClearDepth(1.0);
    glDepthFunc(GL_LESS);
    glEnable(GL_DEPTH_TEST);    // Enables Depth Testing
    glShadeModel(GL_SMOOTH);

    glViewport(0, 0, kWindowWidth, kWindowHeight);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    gluPerspective(45.0f, (GLfloat) kWindowWidth /
                   (GLfloat) kWindowHeight, 0.1f, 100.0f);

    // Calculate the Aspect Ratio of the window
    glMatrixMode(GL_MODELVIEW);

    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specular);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, shininess);
```

```
    glColorMaterial(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
    glEnable(GL_COLOR_MATERIAL);

    glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse);
    glLightfv(GL_LIGHT1, GL_POSITION, LightPosition);

    if (true == is_light_on)
    {
        glEnable(GL_LIGHTING);
    }
    glEnable(GL_LIGHT1);

}

// ========================================================================
// void reshape(int w, int h)
// ========================================================================
void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();

    gluPerspective(45.0, (GLfloat) w / (GLfloat) h, 0.1, 100.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}


// ========================================================================
// void Idle()
// ========================================================================
void Idle()
{
    glutPostRedisplay();
}


// ========================================================================
// void processMouse(int button, int state, int x, int y)
// ========================================================================
void processMouse(int button, int state, int x, int y)
{

    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {

        objectID = retrieveObjectID(x, y);

        if ( (1 == objectID) && (FALSE == isSphereNoisy) &&
             (DO == cursor_type) )
        {
```

```
            alSourcePlay(source[1]);
            isSphereNoisy = TRUE;
        }
        else if ( (1 == objectID) && (TRUE == isSphereNoisy) &&
                  (DO == cursor_type) )
        {
            alSourceStop(source[1]);
            isSphereNoisy = FALSE;
        }


        glutPostRedisplay();
    }
    else if (button == GLUT_MIDDLE_BUTTON)
    {
        // Do nothing for now
    }
    else if (button == GLUT_RIGHT_BUTTON)
    {
        if (state == GLUT_DOWN)
        {
            switch (cursor_type)
            {
                case DO:        cursor_type = LOOK;      break;
                case LOOK:      cursor_type = TALK;      break;
                case TALK:      cursor_type = INVENTORY; break;
                case INVENTORY: cursor_type = DO;        break;
                default:        cursor_type = DO;
            }

            glutPostRedisplay();
        }
    }
}


// ========================================================================
// int retrieveObjectID(int x, int y)
// ========================================================================
int retrieveObjectID(int x, int y)
{
    int objectsFound = 0;
    GLint viewportCoords[4] = {0};
    GLuint selectBuffer[32] = {0};

    glSelectBuffer(32, selectBuffer);
    glGetIntegerv(GL_VIEWPORT, viewportCoords);
    glMatrixMode(GL_PROJECTION);

    glPushMatrix();
        glRenderMode(GL_SELECT);
        glLoadIdentity();
        gluPickMatrix(x, viewportCoords[3] - y, 2, 2, viewportCoords);
```

```
        gluPerspective(45.0f, (float)kWindowWidth/
                       (float)kWindowHeight, 0.1f, 150.0f);
        glMatrixMode(GL_MODELVIEW);

        drawShapes(GL_SELECT);

        objectsFound = glRenderMode(GL_RENDER);

        glMatrixMode(GL_PROJECTION);

    glPopMatrix();

    glMatrixMode(GL_MODELVIEW);

    if (objectsFound > 0)
    {
        GLuint lowestDepth = selectBuffer[1];

        int selectedObject = selectBuffer[3];

        for (int i = 1; i < objectsFound; i++)
        {
            if (selectBuffer[(i*4)+1] < lowestDepth)
            {
                lowestDepth = selectBuffer[(i*4)+1];
                selectedObject = selectBuffer[(i*4)+3];
            }
        }

        return selectedObject;

    }

    return 0;
}


// ======================================================================
// void PassiveMouseFunc(int x, int y)
// Find out where the mouse cursor is when buttons aren't pressed
// ======================================================================
void PassiveMouseFunc(int x, int y)
{
    // global information to know where the cursor is
    mouse_x = x;
    mouse_y = y;

    // Between each 'move', need to calculate how far the cursor has
    // moved from the previous position to the new position, and then
    // calculate how far the cursor icon is to be moved.
    if(mouse_entry == 1) // if cursor is in game screen
    {
        cursor_x +=  mouse_sensitivity * (x - old_x);
        cursor_y += -mouse_sensitivity_y * (y - old_y);
```

```
        old_x = x;
        old_y = y;

    }

}


// ========================================================================
// void EntryFunc(int x, int y)
// ========================================================================
void EntryFunc(int state)
{
    if (state == GLUT_LEFT)
    {
        mouse_entry = 0;
    }
    else // if (state == GLUT_ENTERED)
    {
        mouse_entry = 1;
    }
}


// ========================================================================
// void keyboard(unsigned char key, int x, int y)
// ========================================================================
void keyboard(unsigned char key, int x, int y)
{
    switch (key)
    {
        case 27:
            glutLeaveGameMode();
            glutDestroyWindow(window);
            exit(0);
            break;
        case 'l':
        case 'L':
            if (is_light_on == true)
            {
                is_light_on = false;
                glDisable(GL_LIGHTING);
            }
            else
            {
                is_light_on = true;
                glEnable(GL_LIGHTING);
            }
            break;
       case 'q':
       case 'Q':
            glutDestroyWindow(window);
            glutLeaveGameMode(); exit(0); break;
```

```
        case 'x':
             x_angle = (x_angle + 5) % 360;
             glutPostRedisplay();
             break;
        case 'X':
             x_angle = (x_angle - 5) % 360;
             glutPostRedisplay();
             break;
        case '\t':
             switch (cursor_type)
             {
                 case DO:        cursor_type = LOOK;      break;
                 case LOOK:      cursor_type = TALK;      break;
                 case TALK:      cursor_type = INVENTORY; break;
                 case INVENTORY: cursor_type = DO;        break;
                 default:        cursor_type = DO;
             }

             cout << "Cursor_type: " << cursor_type << endl;
             glutPostRedisplay();
             break;
        case '\n':
             cout << "Saw ENTER" << endl;
             break;
        case '\r':
             cout << "Saw Mac ENTER" << endl;
             // '\r' is the Mac version of ENTER
             break;


    }
}


// ========================================================================
// void specialKeys(int key, int x, int y)
// ========================================================================
void specialKeys(int key, int x, int y)
{
    Vector3D egoPosAfter;
    bool is_collision = false;

    switch(key)
    {
        case GLUT_KEY_F3:
             if (lights_on == TRUE)
             {
                 lights_on = FALSE;
             }
             else
             {
                 lights_on = TRUE;
             }
```

```
    break;
case GLUT_KEY_UP:
    egoPosAfter = egoPosition - egoVelocity;

    // check for collision with all available planes
    for (int i = 0; i < 2; i++)
    {
        float D0 = 0.0, D1 = 0.0;
        Vector3D world_normals[2] = {
          Vector3D(0.0, 0.0, -1.0), Vector3D(0.0, 0.0, 1.0)};

        D0 = egoPosition.dotProduct(world_normals[i]);
        // before movement
        D1 = egoPosAfter.dotProduct(world_normals[i]);
        // after movement

        // check distance to plane

        if (fabs(D1) < egoRadius)
        {
            // the sphere is too close to plane
            is_collision = true;
        }

        // collision check
        if ( (D0 > 0 && D1 < 0) || (D0 < 0 && D1 > 0) )
        {
            // ego has through the plane
            is_collision = true;
        }
    }

    if (false == is_collision)
    {
        // move the sphere
        egoPosition = egoPosAfter;

        z_camera += 1.0;
        xpos -= (float)sin(heading*piover180) * 0.05f;
        zpos -= (float)cos(heading*piover180) * 0.05f;

        egoVelocity.setVector((float)sin(heading*piover180) *
        0.05f, 0.0, (float)cos(heading*piover180) * 0.05f);

        if (walkbiasangle >= 359.0f)
                walkbiasangle = 0.0f;
        else
                walkbiasangle+= 10;

        walkbias = (float)sin(walkbiasangle *
                   piover180)/20.0f;

        // Play the footsteps wave
        if (isDirectionalKeyDown == false)
```

```
        {
            alSourcePlay(source[0]);
            isDirectionalKeyDown = true;
        }
    }
    glutPostRedisplay();
    break;
case GLUT_KEY_DOWN:
    egoPosAfter = egoPosition + egoVelocity;

    // check for collision with all available planes
    for (int i = 0; i < 2; i++)
    {
        float D0 = 0.0, D1 = 0.0;
        Vector3D world_normals[2] = {
          Vector3D(0.0, 0.0, -1.0), Vector3D(0.0, 0.0, 1.0)};

        D0 = egoPosition.dotProduct(world_normals[i]);
        // before movement
        D1 = egoPosAfter.dotProduct(world_normals[i]);
        // after movement

        // check distance to plane

        if (fabs(D1) < egoRadius)
        {
            // the sphere is too close to plane
            is_collision = true;
        }

        // collision check
        if ( (D0 > 0 && D1 < 0) || (D0 < 0 && D1 > 0) )
        {
            // ego has through the plane
            is_collision = true;
        }
    }

    if (false == is_collision)
    {
        // move the sphere
        egoPosition = egoPosAfter;

        z_camera -= 1.0;
        xpos += (float)sin(heading*piover180) * 0.05f;
        zpos += (float)cos(heading*piover180) * 0.05f;

        egoVelocity.setVector((float)sin(heading*piover180) *
        0.05f, 0.0, (float)cos(heading*piover180) * 0.05f);

        if (walkbiasangle <= 1.0f)
                walkbiasangle = 359.0f;
        else
                walkbiasangle-= 10;
```

```
                walkbias = (float)sin(walkbiasangle *
                           piover180)/20.0f;

                // Play the footsteps wave
                if (isDirectionalKeyDown == false)
                {
                    alSourcePlay(source[0]);
                    isDirectionalKeyDown = true;
                }
            }
            glutPostRedisplay();
            break;
        case GLUT_KEY_RIGHT:
            y_angle = (y_angle + 5) % 360;
            heading -= 2.0f;
            yrot = heading;
            egoVelocity.setVector((float)sin(heading*piover180) *
            0.05f, 0.0, (float)cos(heading*piover180) * 0.05f);
            glutPostRedisplay();
            break;
        case GLUT_KEY_LEFT:
            y_angle = (y_angle - 5) % 360;
            heading += 2.0f;
            yrot = heading;
            egoVelocity.setVector((float)sin(heading*piover180) *
            0.05f, 0.0, (float)cos(heading*piover180) * 0.05f);
            glutPostRedisplay();
            break;
        case GLUT_KEY_PAGE_DOWN:
            x_angle = (x_angle + 5) % 360;
            glutPostRedisplay();
            break;
        case GLUT_KEY_PAGE_UP:
            x_angle = (x_angle - 5) % 360;
            glutPostRedisplay();
            break;
        default:
            break;
    }
}


// ========================================================================
// void specialKeysUp(int key, int x, int y)
// set the isDirectionalKeyUp to false if necessary
// Also it is possible to create an array of the special keys, so
// info can be contained in an array instead of just relying on
// one or two boolean type values
// ========================================================================
void specialKeysUp(int key, int x, int y)
{
    switch (key)
    {
```

```
        case GLUT_KEY_RIGHT :
                glutPostRedisplay();
                break;
        case GLUT_KEY_LEFT :
                glutPostRedisplay();
                break;
        case GLUT_KEY_UP :
                alSourceStop(source[0]);
                isDirectionalKeyDown = false;
                glutPostRedisplay();
                break;
        case GLUT_KEY_DOWN :
                alSourceStop(source[0]);
                isDirectionalKeyDown = false;
                glutPostRedisplay();
                break;
    }
}
```

```
// ========================================================================
// pige_linux.h
// Linux main header
// ========================================================================


#define TRUE 0
#define FALSE 1

// include files
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>

#include <GL/glut.h>

#include <AL/al.h>
#include <AL/alut.h>

#include "LoadTGA.h"
#include "LoadGLTextures_win_linux.h"

#include "openal_linux.h"

#include "get_random.h"

#include "Vector3D.h"

unsigned int texture[32];
```

```
// ====================================================================
// pige_mac.h
// Macintosh main header
// ====================================================================


// include files
#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>

#include <GLUT/glut.h>
#include <OpenAL/alut.h>

#include "LoadTGA.h"
#include "LoadGLTextures_mac.h"

#include "openal_mac.h"

#include "get_random.h"

#include "Vector3D.h"

unsigned long texture[32];
```

```
// ========================================================================
// pige_win.h
// Windows main header
// ========================================================================

#define TRUE 0
#define FALSE 1

// include files
 #include <iostream.h>
 #include <stdio.h>
 #include <stdarg.h>
 #include <string.h>
 #include <stdlib.h>
 #include <math.h>


#include <GL/glut.h>

 // In CodeWarrior, add the path (Target Settings -> Access Path)
 // c:\Program Files\Creative Labs\OpenAL 1.0 SDK\Include\
 #include "altypes.h"
 #include "al.h"
 #include "alu.h"
 #include "alut.h"

#include "LoadTGA.h"
#include "LoadGLTextures_win_linux.h"

#include "openal_win.h"

#include "get_random.h"

#include "Vector3D.h"

unsigned int texture[32];
```

```
// ========================================================================
// LoadTGA.h
// load in textures
// ========================================================================


// ========================================================================
// TGAImageRec structure
// ========================================================================
typedef struct TGAImageRec
{
    GLubyte     *data;          // Image data (Up to 32 bits)
    GLuint      bpp;            // Image color depth in bits per pixel
    GLuint      sizeX;          // Image width
    GLuint      sizeY;          // Image height
}TGAImageRec;


// ========================================================================
// TGAImageRec* LoadTGA( char *filename )
// ========================================================================
TGAImageRec* LoadTGA( char *filename )
{
    int i = 0;
    GLubyte     TGAheader[12]= {0,0,2,0,0,0,0,0,0,0,0,0};
    GLubyte     TGAcompare[12]; // Used to compare TGA header
    GLubyte     header[6];      // First 6 Useful bytes from header
    GLuint      bytesPerPixel;  // Holds number of bytes per pixel used
    GLuint      imageSize;      // Used to store image size
    GLuint      temp;           // Temporary variable
    // int      type = GL_RGBA; // Set default GL mode to RGBA (32 BPP)
    TGAImageRec *texture;       // Texture structure to return
    FILE        *file;          // Image file to be opened

    // bytesPerPixel, imageSize, and temp can be 'int' instead of
    // GLUint for Mac OS X.  But for consistency among the various
     // platforms, just keep as GLuint.

     file = fopen( filename, "rb" );

    if( ( file == NULL ) ||
        ( fread( TGAcompare, 1, sizeof( TGAcompare ), file ) !=
        sizeof( TGAcompare ) ) ||
        ( memcmp(TGAheader, TGAcompare, sizeof( TGAheader )) != 0 ) ||
        ( fread( header, 1, sizeof( header ), file )
        != sizeof( header ) ) )
    {
        printf("Error: could not find or open file %s\n", filename);
        fclose( file );
        return NULL;
    }

    texture = ( TGAImageRec* )malloc( sizeof( TGAImageRec ) );
```

```
    texture->sizeX = header[1] * 256 + header[0];
    texture->sizeY = header[3] * 256 + header[2];

    if( ( texture->sizeX <= 0 ) || ( texture->sizeY <= 0 ) ||
        ( ( header[4] != 24 ) && ( header[4] != 32 ) ) )
    {
        fclose( file );
        free( texture );
        return NULL;
     }

    texture->bpp = header[4];
    bytesPerPixel = texture->bpp/8;

    imageSize = texture->sizeX * texture->sizeY * bytesPerPixel;

    texture->data = ( GLubyte* )malloc( imageSize );

    if( ( texture->data == NULL ) ||
        ( fread(texture->data, 1, imageSize, file ) != imageSize ))
    {

        if( texture->data != NULL )
        {
            free( texture->data );
        }

        fclose( file );
        free( texture );
        return NULL;
    }

    // For Linux and Windows, imageSize may need to be
    // casted as an int.
    for ( i = 0; i < (int) imageSize ; i += bytesPerPixel )
    {

        temp = texture->data[i];
        texture->data[i] = texture->data[i + 2];
        texture->data[i + 2] = temp;

     }

    fclose( file );

    return texture;
}
```

```
// ====================================================================
// LoadGLTextures_mac.h
// Macintosh version to load in textures
// ====================================================================


// ====================================================================
// void LoadTextures(unsigned long texture[])
// ====================================================================
void LoadGLTextures(unsigned long texture[])
{
    GLuint type = GL_RGBA;
    TGAImageRec *local_tex[7];

    // alpha images should be 32 bit or it won't work!
    local_tex[0] = LoadTGA("do_alpha32.tga");      // DO icon
    local_tex[1] = LoadTGA("eye_alpha.tga");       // LOOK icon
    local_tex[2] = LoadTGA("talk_alpha32.tga");    // TALK icon
    local_tex[3] = LoadTGA("wood-1.tga");          // wall texture
    local_tex[4] = LoadTGA("hardwoodfloor32.tga"); // floor texture
    local_tex[5] = LoadTGA("ceiling.tga");         // ceiling texture

    for (int i = 0; i < 6; i++)
    {

        glGenTextures(1, &texture[i]);
        glBindTexture(GL_TEXTURE_2D, texture[i]);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                        GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                        GL_LINEAR);

        if ( local_tex[i]->bpp == 24 )
        {
            type = GL_RGB;
        }
        else
        {
            type = GL_RGBA;
        }

        glTexImage2D(GL_TEXTURE_2D, 0, type, local_tex[i]->sizeX,
local_tex[i]->sizeY, 0, type, GL_UNSIGNED_BYTE, local_tex[i]->data);

    }

}
```

```
// ========================================================================
// LoadGLTextures_win_linux.h
// Windows & Linux version to load in textures
// ========================================================================


// ========================================================================
// void LoadTextures(unsigned int texture[])
// ========================================================================
void LoadGLTextures(unsigned int texture[])
{
    GLuint type = GL_RGBA;
    TGAImageRec *local_tex[7];

    // alpha images should be 32 bit or it won't work!
    local_tex[0] = LoadTGA("do_alpha32.tga");      // DO icon
    local_tex[1] = LoadTGA("eye_alpha.tga");       // LOOK icon
    local_tex[2] = LoadTGA("talk_alpha32.tga");    // TALK icon
    local_tex[3] = LoadTGA("wood-1.tga");          // wall texture
    local_tex[4] = LoadTGA("hardwoodfloor32.tga"); // floor texture
    local_tex[5] = LoadTGA("ceiling.tga");         // ceiling texture

    for (int i = 0; i < 6; i++)
    {

        glGenTextures(1, &texture[i]);
        glBindTexture(GL_TEXTURE_2D, texture[i]);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                        GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                        GL_LINEAR);

        if ( local_tex[i]->bpp == 24 )
        {
            type = GL_RGB;
        }
        else
        {
            type = GL_RGBA;
        }


        glTexImage2D(GL_TEXTURE_2D, 0, type, local_tex[i]->sizeX,
local_tex[i]->sizeY, 0, type, GL_UNSIGNED_BYTE, local_tex[i]->data);

    }

}
```

```
// ========================================================================
// openal_linux.h
// Linux version to load in sound
// ========================================================================


// ========================================================================
// void openal_init()
// ========================================================================
void openal_init(void)
{
    ALboolean al_bool = 0;

    alListenerfv(AL_POSITION,listenerPos);
    alListenerfv(AL_VELOCITY,listenerVel);
    alListenerfv(AL_ORIENTATION,listenerOri);

    alGetError(); // clear any error messages

    if(alGetError() != AL_NO_ERROR)
    {
        printf("- Error creating buffers !!\n");
        exit(1);
    }
    else
    {
        printf("init() - No errors yet.\n");
    }

    // Generate buffers, or else no sound will happen
    alGenBuffers(NUM_BUFFERS, buffer);

    alutLoadWAVFile((ALbyte *)"Footsteps.wav", &format, &data, &size,
                    &freq, &al_bool);
    alBufferData(buffer[0],format,data,size,freq);
    alutUnloadWAV(format,data,size,freq);

    alutLoadWAVFile((ALbyte *)"a.wav", &format, &data, &size, &freq,
                    &al_bool);
    alBufferData(buffer[1],format,data,size,freq);
    alutUnloadWAV(format,data,size,freq);

    alGetError(); /* clear error */
    alGenSources(NUM_SOURCES, source);

    if(alGetError() != AL_NO_ERROR)
    {
        printf("- Error creating sources !!\n");
        exit(2);
    }
    else
    {
        printf("init - no errors after alGenSources\n");
```

```
    }

    alSourcef(source[0],AL_PITCH,1.0f);
    alSourcef(source[0],AL_GAIN,1.0f);
    alSourcefv(source[0],AL_POSITION,source0Pos);
    alSourcefv(source[0],AL_VELOCITY,source0Vel);
    alSourcei(source[0],AL_BUFFER,buffer[0]);
    alSourcei(source[0],AL_LOOPING,AL_TRUE);


    alSourcef(source[1],AL_PITCH,1.0f);
    alSourcef(source[1],AL_GAIN,1.0f);
    alSourcefv(source[1],AL_POSITION,source1Pos);
    alSourcefv(source[1],AL_VELOCITY,source1Vel);
    alSourcei(source[1],AL_BUFFER,buffer[1]);
    alSourcei(source[1],AL_LOOPING,AL_TRUE);

}
```

```
// ========================================================================
// openal_mac.h
// Macintosh version to load in sound
// ========================================================================


// ========================================================================
// void openal_init()
// ========================================================================
void openal_init(void)
{
    alListenerfv(AL_POSITION,listenerPos);
    alListenerfv(AL_VELOCITY,listenerVel);
    alListenerfv(AL_ORIENTATION,listenerOri);

    alGetError(); // clear any error messages

    if(alGetError() != AL_NO_ERROR)
    {
        printf("- Error creating buffers !!\n");
        exit(1);
    }
    else
    {
        printf("init() - No errors yet.\n");
    }

    // Generate buffers, or else no sound will happen
    alGenBuffers(NUM_BUFFERS, buffer);

    alutLoadWAVFile("Footsteps.wav", &format, &data, &size, &freq);
    alBufferData(buffer[0],format,data,size,freq);
    alutUnloadWAV(format,data,size,freq);

    alutLoadWAVFile("a.wav", &format, &data, &size, &freq);
    alBufferData(buffer[1],format,data,size,freq);
    alutUnloadWAV(format,data,size,freq);

    alGetError(); /* clear error */
    alGenSources(NUM_SOURCES, source);

    if(alGetError() != AL_NO_ERROR)
    {
        printf("- Error creating sources !!\n");
        exit(2);
    }
    else
    {
        printf("init - no errors after alGenSources\n");
    }

    alSourcef(source[0],AL_PITCH,1.0f);
    alSourcef(source[0],AL_GAIN,1.0f);
```

```
    alSourcefv(source[0],AL_POSITION,source0Pos);
    alSourcefv(source[0],AL_VELOCITY,source0Vel);
    alSourcei(source[0],AL_BUFFER,buffer[0]);
    alSourcei(source[0],AL_LOOPING,AL_TRUE);


    alSourcef(source[1],AL_PITCH,1.0f);
    alSourcef(source[1],AL_GAIN,1.0f);
    alSourcefv(source[1],AL_POSITION,source1Pos);
    alSourcefv(source[1],AL_VELOCITY,source1Vel);
    alSourcei(source[1],AL_BUFFER,buffer[1]);
    alSourcei(source[1],AL_LOOPING,AL_TRUE);

}
```

```
// ========================================================================
// openal_win.h
// Windows version to load in sound
// ========================================================================


// ========================================================================
// void openal_init()
// ========================================================================
void openal_init(void)
{
    char al_bool;

    alListenerfv(AL_POSITION,listenerPos);
    alListenerfv(AL_VELOCITY,listenerVel);
    alListenerfv(AL_ORIENTATION,listenerOri);

    alGetError(); // clear any error messages

    if(alGetError() != AL_NO_ERROR)
    {
        printf("- Error creating buffers !!\n");
        exit(1);
    }
    else
    {
        printf("init() - No errors yet.\n");
    }

    // Generate buffers, or else no sound will happen
    alGenBuffers(NUM_BUFFERS, buffer);

    alutLoadWAVFile("Footsteps.wav", &format, &data, &size, &freq,
                    &al_bool);
    alBufferData(buffer[0],format,data,size,freq);
    alutUnloadWAV(format,data,size,freq);

    alutLoadWAVFile("a.wav", &format, &data, &size, &freq, &al_bool);
    alBufferData(buffer[1],format,data,size,freq);
    alutUnloadWAV(format,data,size,freq);

    alGetError(); /* clear error */
    alGenSources(NUM_SOURCES, source);

    if(alGetError() != AL_NO_ERROR)
    {
        printf("- Error creating sources !!\n");
        exit(2);
    }
    else
    {
        printf("init - no errors after alGenSources\n");
    }
```

```
    alSourcef(source[0],AL_PITCH,1.0f);
    alSourcef(source[0],AL_GAIN,1.0f);
    alSourcefv(source[0],AL_POSITION,source0Pos);
    alSourcefv(source[0],AL_VELOCITY,source0Vel);
    alSourcei(source[0],AL_BUFFER,buffer[0]);
    alSourcei(source[0],AL_LOOPING,AL_TRUE);


    alSourcef(source[1],AL_PITCH,1.0f);
    alSourcef(source[1],AL_GAIN,1.0f);
    alSourcefv(source[1],AL_POSITION,source1Pos);
    alSourcefv(source[1],AL_VELOCITY,source1Vel);
    alSourcei(source[1],AL_BUFFER,buffer[1]);
    alSourcei(source[1],AL_LOOPING,AL_TRUE);

}
```

```
// ========================================================================
// get_random.h
// Random number generator
// ========================================================================


#include <stdlib.h>
#include <time.h>

int get_random(int low, int high)
{
    static int seeded = 0;
    int value;

    if (seeded == 0)
    {
        srand((unsigned)time((time_t *)NULL));
        seeded = 1;
    }

    value = rand()%(high - low + 1) + low;

    return (value);
}
```

```
// ========================================================================
// Vector3D.h
// 3-dimensional vector class
// ========================================================================

#include <math.h>

class Vector3D
{
    public:

        float x, y, z;

        // constructors and destructors
        Vector3D() {}
        Vector3D(const float fx,const float fy,const float fz)
                { x=fx; y=fy; z=fz; }
        ~Vector3D() {}

        // set and get methods
        void setVector(float, float, float);
        Vector3D vector();

        // vector operations
        Vector3D operator+(const Vector3D & v)
                  { return Vector3D(x+v.x,y+v.y,z+v.z); }
        Vector3D operator-(const Vector3D & v)
                  { return Vector3D(x-v.x,y-v.y,z-v.z); }
        Vector3D operator*(const Vector3D & v)
                  { return Vector3D(x*v.x,y*v.y,z*v.z); }
        Vector3D operator/(const Vector3D & v)
                  { return Vector3D(x/v.x,y/v.y,z/v.z); }
        Vector3D operator*(const float f)
                  { return Vector3D(x*f, y*f, z*f);   }
        Vector3D operator/(const float f)
                  { return Vector3D(x/f, y/f, z/f); }

        Vector3D operator-() { return Vector3D(-x,-y,-z); }

        Vector3D operator+=(const Vector3D & v)
                  { *this = *this + v; return *this; }

        // vector mathematical functions
        void crossProduct(Vector3D, Vector3D);
        float dotProduct(Vector3D &, Vector3D &);
        float dotProduct(Vector3D &);
        void normalize();

        float vectorMagnitude()
                  { return (float)sqrt(x*x + y*y + z*z); }

        void printComponents();

};
```

```
// ========================================================================
// void setVector(float new_x, float new_y, float new_z)
// ========================================================================
void Vector3D :: setVector(float new_x, float new_y, float new_z)
{
    x = new_x;
    y = new_y;
    z = new_z;
}


// ========================================================================
// Vector3D vector()
// ========================================================================
Vector3D Vector3D :: vector()
{
    return (Vector3D(x, y, z));
}


// ========================================================================
// void crossProduct(Vector3D v1, Vector3D v2)
// ========================================================================
void Vector3D :: crossProduct(Vector3D v1, Vector3D v2)
{
      x = ( v1.y * v2.z ) - ( v1.z * v2.y );
      y = ( v1.z * v2.x ) - ( v1.x * v2.z );
      z = ( v1.x * v2.y ) - ( v1.y * v2.x );
}


// ========================================================================
// float dotProduct(Vector3D & v1, Vector3D & v2)
// ========================================================================
float Vector3D :: dotProduct(Vector3D & v1, Vector3D & v2)
{
      return ( v1.x*v2.x  +  v1.y*v2.y  +  v1.z*v2.z );
}


// ========================================================================
// float dotProduct(Vector3D & v2)
// ========================================================================
float Vector3D :: dotProduct(Vector3D & v2)
{
    return ( x*v2.x + y*v2.y + z*v2.z );
}


// ========================================================================
// void normalize
// ========================================================================
void Vector3D :: normalize()
```

```
{
      float len = vectorMagnitude();

      // if the vector has no magnitude (0), then
      // return since normalizing a 0 vector will
      // result in an error by trying to divide by 0.
      if (len == 0) return;

      len = 1.0f/len;

      x *= len;
      y *= len;
      z *= len;
}


// ====================================================================
// void printComponents()
// ====================================================================
void Vector3D :: printComponents()
{
    cout << "x: " << x << " y: " << y << " z: " << z << endl;
}
```

APPENDIX B

OPENAL SOURCE CODE

```
// ============================================================================
// openal-example.cpp
// ============================================================================

// ============================================================================
// Include libraires and files
// Below are the three standard library headers for Mac OS X.  For
// Linux or Windows, change the libraries such:
// #include <GL/glut.h>
// #include <AL/alut.h>
// Depending on your system configuration, you might need to also
// include these libraries:
// #include <AL/al.h>
// #include <GL/gl.h>
// #include <GL/glu.h>
// ============================================================================
#include <stdio.h>
#include <GLUT/glut.h>
#include <OpenAL/alut.h>

// ============================================================================
// Function prototypes
// ============================================================================
void init();
void display();
void reshape(int w, int h);
void keyboard(unsigned char key, int x, int y);
void specialKeys(int key, int x, int y);

// ============================================================================
// Global variables
// ============================================================================
#define     NUM_BUFFERS 3
#define     NUM_SOURCES 3
#define     NUM_ENVIRONMENTS 1

ALfloat     listenerPos[]={0.0,0.0,4.0};
ALfloat     listenerVel[]={0.0,0.0,0.0};
ALfloat     listenerOri[]={0.0,0.0,1.0, 0.0,1.0,0.0};

ALfloat     source0Pos[]={ -2.0, 0.0, 0.0};
ALfloat     source0Vel[]={ 0.0, 0.0, 0.0};


ALfloat     source1Pos[]={ 2.0, 0.0, 0.0};
ALfloat     source1Vel[]={ 0.0, 0.0, 0.0};

ALfloat     source2Pos[]={ 0.0, 0.0, -4.0};
ALfloat     source2Vel[]={ 0.0, 0.0, 0.0};

ALuint      buffer[NUM_BUFFERS];
ALuint      source[NUM_SOURCES];
ALuint      environment[NUM_ENVIRONMENTS];
```

```
int         GLwin;

ALsizei     size,freq;
ALenum      format;
ALvoid      *data;
int         ch;

// ====================================================================
// void main(int argc, char** argv)
// ====================================================================
int main(int argc, char** argv) //finaly the main function
{
      //initialise glut
      glutInit(&argc, argv) ;
      glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH) ;
      glutInitWindowSize(400,400) ;

      //initialise openAL
      alutInit(&argc, argv) ;

      GLwin = glutCreateWindow("PIGE - OpenAL Example") ;
      init() ;
      glutDisplayFunc(display) ;
      glutKeyboardFunc(keyboard) ;
      glutSpecialFunc(specialKeys);
      glutReshapeFunc(reshape) ;

      glutMainLoop() ;

      return 0;
}

// ====================================================================
// void init()
// ====================================================================
void init(void)
{
    alListenerfv(AL_POSITION,listenerPos);
    alListenerfv(AL_VELOCITY,listenerVel);
    alListenerfv(AL_ORIENTATION,listenerOri);

    alGetError(); // clear any error messages

    if(alGetError() != AL_NO_ERROR)
    {
        printf("- Error creating buffers !!\n");
        exit(1);
    }
    else
    {
        printf("init() - No errors yet.");
    }

    // Generate buffers, or else no sound will happen!
```

```
    alGenBuffers(NUM_BUFFERS, buffer);

    alutLoadWAVFile("c.wav",&format,&data,&size,&freq);
    alBufferData(buffer[0],format,data,size,freq);
    alutUnloadWAV(format,data,size,freq);

    alutLoadWAVFile("b.wav",&format,&data,&size,&freq);
    alBufferData(buffer[1],format,data,size,freq);
    alutUnloadWAV(format,data,size,freq);

    alutLoadWAVFile("a.wav",&format,&data,&size,&freq);
    alBufferData(buffer[2],format,data,size,freq);
    alutUnloadWAV(format,data,size,freq);

    alGetError(); /* clear error */
    alGenSources(NUM_SOURCES, source);

    if(alGetError() != AL_NO_ERROR)
    {
        printf("- Error creating sources !!\n");
        exit(2);
    }
    else
    {
        printf("init - no errors after alGenSources\n");
    }

    alSourcef(source[0],AL_PITCH,1.0f);
    alSourcef(source[0],AL_GAIN,1.0f);
    alSourcefv(source[0],AL_POSITION,source0Pos);
    alSourcefv(source[0],AL_VELOCITY,source0Vel);
    alSourcei(source[0],AL_BUFFER,buffer[0]);
    alSourcei(source[0],AL_LOOPING,AL_TRUE);

    alSourcef(source[1],AL_PITCH,1.0f);
    alSourcef(source[1],AL_GAIN,1.0f);
    alSourcefv(source[1],AL_POSITION,source1Pos);
    alSourcefv(source[1],AL_VELOCITY,source1Vel);
    alSourcei(source[1],AL_BUFFER,buffer[1]);
    alSourcei(source[1],AL_LOOPING,AL_TRUE);

    alSourcef(source[2],AL_PITCH,1.0f);
    alSourcef(source[2],AL_GAIN,1.0f);
    alSourcefv(source[2],AL_POSITION,source2Pos);
    alSourcefv(source[2],AL_VELOCITY,source2Vel);
    alSourcei(source[2],AL_BUFFER,buffer[2]);
    alSourcei(source[2],AL_LOOPING,AL_TRUE);
}


// ========================================================================
// void display()
// ========================================================================
void display(void)
```

```
{
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT) ;
        glPushMatrix() ;
      glRotatef(20.0,1.0,1.0,0.0) ;

      glPushMatrix() ;
      glTranslatef(source0Pos[0],source0Pos[1],source0Pos[2]) ;
      glColor3f(1.0,0.0,0.0) ;
      glutWireCube(0.5) ;
      glPopMatrix() ;

      glPushMatrix() ;
      glTranslatef(source2Pos[0],source2Pos[1],source2Pos[2]) ;
      glColor3f(0.0,0.0,1.0) ;
      glutWireCube(0.5) ;
      glPopMatrix() ;

      glPushMatrix() ;
      glTranslatef(source1Pos[0],source0Pos[1],source0Pos[2]) ;
      glColor3f(0.0,1.0,0.0) ;
      glutWireCube(0.5) ;
      glPopMatrix() ;

      //the listener
      glPushMatrix() ;
      glTranslatef(listenerPos[0],listenerPos[1],listenerPos[2]) ;
        glColor3f(1.0,1.0,1.0) ;
      glutWireCube(0.5) ;
      glPopMatrix() ;

        glPopMatrix() ;
      glutSwapBuffers() ;
}

// ========================================================================
// void reshape(int w, int h)
// ========================================================================
void reshape(int w, int h) // the reshape function
{
   glViewport(0,0,(GLsizei)w,(GLsizei)h) ;
   glMatrixMode(GL_PROJECTION) ;
   glLoadIdentity() ;
   gluPerspective(60.0,(GLfloat)w/(GLfloat)h,1.0,30.0) ;
   glMatrixMode(GL_MODELVIEW) ;
   glLoadIdentity() ;
   glTranslatef(0.0,0.0,-6.6) ;
}


// ========================================================================
// void keyboard(int key, int x, int y)
// ========================================================================
void keyboard(unsigned char key, int x, int y)
{
```

```
switch(key)
{
	case '1':
		alSourcePlay(source[0]);
			printf("1\n");
		break;
	case '2':
		alSourcePlay(source[1]);
			printf("2\n");
		break;
	case '3':
		alSourcePlay(source[2]);
			printf("3\n");
		break;
	case '4':
		alSourceStop(source[0]);
			printf("4\n");
		break;
	case '5':
		alSourceStop(source[1]);
			printf("5\n");
		break;
	case '6':
		alSourceStop(source[2]);
			printf("6\n");
		break;
	case 'a':
	case 'A':
		listenerPos[0] -= 0.1 ;
		alListenerfv(AL_POSITION,listenerPos);

		break ;
	case 's':
	case 'S':
		listenerPos[0] += 0.1 ;
		alListenerfv(AL_POSITION,listenerPos);

		break ;
	case 'q':
	case 'Q':
		listenerPos[2] -= 0.1 ;
		alListenerfv(AL_POSITION,listenerPos);

		break ;
	case 'z':
	case 'Z':
		listenerPos[2] += 0.1 ;
		alListenerfv(AL_POSITION,listenerPos);

		break ;
	case 27:
		alSourceStop(source[2]);
		alSourceStop(source[1]);
		alSourceStop(source[0]);
```

```
                alutExit();
                glutDestroyWindow(GLwin) ;
                exit(0) ;
                break ;
              default: break;
      }
      glutPostRedisplay() ;
}

// ========================================================================
// void specialKeys(int key, int x, int y)
// ========================================================================
void specialKeys(int key, int x, int y)
{
    switch(key)
    {
        case GLUT_KEY_RIGHT:  listenerPos[0] += 0.1 ;
                              alListenerfv(AL_POSITION,listenerPos);
                              glutPostRedisplay() ;
                              break;
        case GLUT_KEY_LEFT:   listenerPos[0] -= 0.1 ;
                              alListenerfv(AL_POSITION,listenerPos);
                              glutPostRedisplay() ;
                              break;
        case GLUT_KEY_UP:     listenerPos[2] -= 0.1 ;
                              alListenerfv(AL_POSITION,listenerPos);
                              glutPostRedisplay() ;
                              break;
        case GLUT_KEY_DOWN:   listenerPos[2] += 0.1 ;
                              alListenerfv(AL_POSITION,listenerPos);
                              glutPostRedisplay() ;
                              break;
    }
}
```

APPENDIX C

TEXTURE SOURCE CODE

```
// ========================================================================
// texture.c
// ========================================================================


// ========================================================================
// Include libraires and files
// ------------------------------------------------------------------------
// For Windows & Linux systems, comment out the <GLUT/glut.h>
// header and uncomment the other three GL headers.
// ========================================================================
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "math.h"

#include <GLUT/glut.h>
//#include <GL/gl.h>
//#include <GL/glu.h>
//#include <GL/glut.h>

// ========================================================================
// Constants and global variables
// ------------------------------------------------------------------------
// For Windows & Linux systems, the texture[1] needs to be of type
// unsigned int.
// ========================================================================
#define kWindowWidth    512
#define kWindowHeight   256

unsigned long           texture[1];             // Texture storage

GLfloat                 xrot;                   // X rotation
GLfloat                 yrot;                   // Y rotation
GLfloat                 zrot;                   // Z rotation


// ========================================================================
// TGAImageRect
// ========================================================================
typedef struct TGAImageRec
{
    GLubyte     *data;  // Image data (up to 32 bits)
    GLuint      bpp;    // Image color depth in Bits Per Pixel.
    GLuint      sizeX;
    GLuint      sizeY;
}   TGAImageRec;


// ========================================================================
// Function prototypes
// ========================================================================
GLvoid InitGL(GLvoid);
GLvoid Display(GLvoid);
```

```
GLvoid ResizeGLScene(int Width, int Height);
GLvoid Keyboard(unsigned char key, int x, int y);
GLvoid Idle(GLvoid);
GLvoid LoadGLTextures(GLvoid);
TGAImageRec* LoadTGA(char *filename);


// ====================================================================
// Main
// ====================================================================
int main(int argc, char** argv)
{

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(kWindowWidth, kWindowHeight);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);

    InitGL();

    glutKeyboardFunc(Keyboard);
    glutDisplayFunc(Display);
    glutReshapeFunc(ResizeGLScene);
    glutIdleFunc(Idle);

    xrot = 0;
    yrot = 0;
    zrot = 0;

    glutMainLoop();

    return 0;
}


// ====================================================================
// InitGL
// ====================================================================
GLvoid InitGL(GLvoid)
{

    LoadGLTextures();                   // Load the texture(s)
    glEnable(GL_TEXTURE_2D);            // Enable texture mapping

    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    // This will clear the background color to black
    glClearDepth(1.0);                  // Enables Clearing Of The
                                        // Depth Buffer
    glDepthFunc(GL_LESS);               // Type of depth test to do
    glEnable(GL_DEPTH_TEST);            // Enables depth testing
    glShadeModel(GL_SMOOTH);            // Enables smooth color
                                        // shading
```

```
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();                   // Reset The Projection Matrix

    gluPerspective(45.0f, (GLfloat) kWindowWidth / (GLfloat)
    kWindowHeight, 0.1f, 100.0f);
    // Calculate The Aspect Ratio Of The Window

    glMatrixMode(GL_MODELVIEW);

}


// ========================================================================
// Keyboard
// ========================================================================
GLvoid Keyboard(unsigned char key, int x, int y)
{
    switch(key)
    {
        case 27: exit(0); break;
        default: printf("x: %d  y: %d\n", x, y);
    }
}


// ========================================================================
// Idle
// ========================================================================
GLvoid Idle(GLvoid)
{
    xrot += 0.3f;                                   // X Axis Rotation
    yrot += 0.2f;                                   // Y Axis Rotation
    zrot += 0.4f;                                   // Z Axis Rotation

    glutPostRedisplay();
}


// ========================================================================
// Display
// ========================================================================
GLvoid Display(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-5.0f);

    glRotatef(xrot,1.0f,0.0f,0.0f);     // Rotate on the X axis
    glRotatef(yrot,0.0f,1.0f,0.0f);     // Rotate on the Y axis
    glRotatef(zrot,0.0f,0.0f,1.0f);     // Rotate on the Z axis

    glBindTexture(GL_TEXTURE_2D, texture[0]);

    glBegin(GL_QUADS);
```

```
// Front Face
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
// Bottom Left Of The Texture and Quad
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
// Bottom Right Of The Texture and Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
// Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
// Top Left Of The Texture and Quad

// Back Face
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
// Bottom Right Of The Texture and Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
// Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
// Top Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
// Bottom Left Of The Texture and Quad

// Top Face
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
// Top Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
// Bottom Left Of The Texture and Quad
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
// Bottom Right Of The Texture and Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
// Top Right Of The Texture and Quad

// Bottom Face
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
// Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
// Top Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
// Bottom Left Of The Texture and Quad
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
// Bottom Right Of The Texture and Quad

// Right face
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
// Bottom Right Of The Texture and Quad
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
// Top Right Of The Texture and Quad
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
// Top Left Of The Texture and Quad
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
// Bottom Left Of The Texture and Quad

// Left Face
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
// Bottom Left Of The Texture and Quad
```

```
        glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
        // Bottom Right Of The Texture and Quad
        glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
        // Top Right Of The Texture and Quad
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
        // Top Left Of The Texture and Quad

    glEnd();

    glutSwapBuffers();
    glFlush();
}


// ======================================================================
// ResizeGLScene
// ======================================================================
GLvoid ResizeGLScene(int Width, int Height)
{
    glViewport (0, 0, (GLsizei) Width, (GLsizei) Height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    gluPerspective(45.0, (GLfloat) Width /
                         (GLfloat) Height, 0.1, 100.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}


// ======================================================================
// LoadGLTextures
// ======================================================================
GLvoid LoadGLTextures(GLvoid)
{
    TGAImageRec         *texture1;
    GLuint              type = GL_RGBA;

    texture1 = LoadTGA("crate.tga");    // load the image

    glGenTextures(1, &texture[0]);      // generate texture
    glBindTexture(GL_TEXTURE_2D, texture[0]);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);

    if ( texture1->bpp == 24 )
        type = GL_RGB;
    else
        type = GL_RGBA;

    glTexImage2D(GL_TEXTURE_2D, 0, type, texture1->sizeX,
                texture1->sizeY, 0, type, GL_UNSIGNED_BYTE,
                texture1->data);
```

```
}


// ====================================================================
// LoadTGA
// ====================================================================
TGAImageRec*    LoadTGA( char *filename )
{
    GLubyte TGAheader[12] =     {0,0,2,0,0,0,0,0,0,0,0,0};
    // Uncompressed TGA Header
    GLubyte                     TGAcompare[12];
    GLubyte                     header[6];
    GLuint                      bytesPerPixel;
    GLuint                      imageSize;
    GLuint                      temp;   // Temporary Variable
    GLuint                      type = GL_RGBA;
    GLuint                      i;
    TGAImageRec                 *texture;
    FILE                        *file;

    file = fopen( filename, "rb" );     // Open The TGA File

    if( ( file == NULL ) ||             // Does File Even Exist?
        ( fread( TGAcompare, 1, sizeof( TGAcompare ), file ) !=
          sizeof( TGAcompare ) ) ||     // Are There 12 Bytes To Read?
        ( memcmp( TGAheader, TGAcompare, sizeof( TGAheader ) ) != 0 )
        ||                      // Does The Header Match What We Want?
        ( fread( header, 1, sizeof( header ), file )
          != sizeof( header ) ) )    // If So Read Next 6 Header Bytes
    {
        // If anything failed then close the file and return false
        printf("Couldn't open file %s\n", filename);
        fclose( file );
        return NULL;
    }

    // Create a new TGAImageRec
    texture = ( TGAImageRec* )malloc( sizeof( TGAImageRec ) );

    // Determine the TGA width (highbyte*256+lowbyte) and height
    // (highbyte*256+lowbyte)
    texture->sizeX = header[1] * 256 + header[0];
    texture->sizeY = header[3] * 256 + header[2];

    // Make sure the height, width, and bit depth are valid
    if( ( texture->sizeX <= 0 ) || ( texture->sizeY <= 0 ) ||
        ( ( header[4] != 24 ) && ( header[4] != 32 ) ) )
    {
        // If anything failed then close the file, free up memory for
        // the image, and return NULL
        fclose( file );
        free( texture );
        return NULL;
```

```
    }

    // Grab The TGA's Bits Per Pixel (24 or 32)
    texture->bpp = header[4];
    bytesPerPixel = texture->bpp/8;
    // Divide By 8 To Get The Bytes Per Pixel

    // Calculate The Memory Required For The TGA Data
    imageSize = texture->sizeX * texture->sizeY * bytesPerPixel;

    // Reserve Memory To Hold The TGA Data
    texture->data = ( GLubyte* )malloc( imageSize );

    // Make sure the right amount of memory was allocated
    if( ( texture->data == NULL ) ||
        ( fread( texture->data, 1, imageSize, file ) != imageSize ) )
    {
        // Free up the image data if there was any
        if( texture->data != NULL )
                free( texture->data );

        // If anything failed then close the file, free up memory for
        // the image, and return NULL
                fclose( file );
                free( texture );
                return NULL;
    }

        // Loop Through The Image Data
    for( i = 0; i < (int) imageSize; i += bytesPerPixel )
    {
        // Swaps The 1st And 3rd Bytes ('R'ed and 'B'lue)
        temp = texture->data[i];
        // Temporarily Store The Value At Image Data 'i'
        texture->data[i] = texture->data[i + 2];
        // Set The 1st Byte To The Value Of The 3rd Byte
        texture->data[i + 2] = temp;
        // Set The 3rd Byte To The Value In 'temp' (1st Byte Value)
    }

    fclose( file );             // Close the file

    return texture;
}
```