# 3-D Graphics and Visualisation (Part II)

EE563

Dr. Derek Molloy and Dr. Robert Sadleir

# Contents

**1 OpenGL - The Graphics Pipeline** **5**

1.1 Introduction . . . 5
1.2 What is the OpenGL API? . . . 5
1.3 Introduction to OpenGL Programming . . . 9
1.3.1 OpenGL Primitives and Attributes . . . 9
1.3.2 OpenGL Full Source Code Example . . . 14
1.3.3 OpenGL Colour . . . 16
1.3.4 Exercise: A Rotating Colour Solid . . . 17
1.3.5 OpenGL Viewing . . . 18
1.3.6 OpenGL Display Lists . . . 24
1.3.7 OpenGL Stack . . . 28
1.3.8 Input Events . . . 29
1.3.9 Double Buffering . . . 31
1.4 Some Math . . . 32
1.4.1 3-D Math Notation . . . 32
1.4.2 Vectors - Math summary . . . 32
1.4.3 A Vector Class . . . 37
1.4.4 Matrices . . . 41
1.4.5 A Matrix Class . . . 42
1.5 Transformations . . . 47
1.5.1 Translation . . . 47
1.5.2 Rotation . . . 48
1.5.3 Scaling . . . 49
1.5.4 Shearing . . . 49
1.5.5 Inverse Operations . . . 50
1.5.6 Combination and Other Operations . . . 50
1.5.7 The OpenGL Current Transformation Matrix (CTM) . . . 50
1.5.8 Coordinate Spaces in the Graphics Pipeline . . . 55
1.6 OpenGL Shading . . . 57
1.6.1 The Phong Model . . . 58
1.6.2 Lambertian Surfaces . . . 60
1.6.3 The Normal Vector . . . 61
1.6.4 Shading . . . 63
1.6.5 Lighting . . . 64
1.6.6 OpenGL code for Shading . . . 65

**2 Scene Graph Theory** **67**

2.1 Introduction . . . 67
2.2 A Simple Scene Graph Implementation . . . 68
2.2.1 Traversing the Scene Graph . . . 70
2.2.2 Putting it Together: The Scene Graph with OpenGL . . . 74
2.2.3 Making it Better: The Scene Graph with OpenGL . . . 79
2.3 Constructive Solid Geometry (CSG) . . . 82

2.4 Scene Graphs and Bounding Volume Hierarchies . . . 84
2.5 Space Subdivision Structures . . . 85
2.5.1 Binary Space Partitioning Trees . . . 85
2.6 Hidden Surface Removal . . . 88
2.6.1 The Painter's Algorithm and BSP . . . 89

**3 Real-Time 3D Computer Graphics Techniques 93**
3.1 Introduction . . . 93
3.2 Texture Mapping in OpenGL . . . 93
3.2.1 Mipmapping . . . 98
3.3 Level of Detail (LOD) . . . 98
3.4 Billboarding . . . 101
3.5 Mappings . . . 102
3.5.1 Bump Mapping . . . 102
3.5.2 Displacement Mapping . . . 105
3.6 Shadows . . . 105
3.6.1 Algorithms for Computing Shadows . . . 105

**4 Appendices 109**
4.1 Installing Dev C++ . . . 109
4.2 Exercise Solutions . . . 109
4.2.1 Solution: A Rotating Colour Solid . . . 109

# Chapter 1

# OpenGL - The Graphics Pipeline

## 1.1 Introduction

The Graphics Pipeline is responsible for taking the description of a scene in 3-D space and mapping it to the view plane in a raster form, i.e. generally to the computer monitor. The most popular models of a graphics pipeline with an extensive API are Java3D and OpenGL.

## 1.2 What is the OpenGL API?

OpenGL is defined simply as 'a software interface to graphics hardware", but this does not do justice to the true capability of this technology; it is an extremely fast graphics library that provides with all the functionality we require to model in 3-D. OpenGL is intended for use with graphics hardware, but there are software versions of OpenGL available - in particular the Microsoft version; however, these days almost all personal computers are supplied with a 3-D graphics card that contains a hardware implementation of OpenGL. Indeed, OpenGL works well in hiding the complexities involved with different 3D graphic accelerators and their processing capabilities, which are supplied from different vendors[1]. The processing capability of OpenGL is supplied by a graphics pipeline, known as the OpenGL State Machine, which is illustrated in 1.1 and 1.2.

There are several 3-D graphic Application Programming Interfaces (APIs) available, such as OpenGL, Direct3D and Java3D. We have discussed Java3D earlier in this module, and now we are discussing the lower-level OpenGL API: We could have examined the Direct3D API, but it is a Microsoft Technology, tied to the Windows OS. OpenGL is maintained by Silicon Graphics Ltd. (SGI) and provides a standard specification defining a cross-platform API for writing 3-D graphic applications. OpenGL is more commonly used in 3-D Visualisation applications, whereas Direct3D is more commonly used in computer gaming applications. As mentioned, OpenGL is a low-level API, which requires the programmer to supply the exact steps to render a scene. This makes the generation of scenes much more difficult than Java3D where we only had to describe the scene, but it makes it possible for us to create novel rendering algorithms or to configure how the pipeline processes the primitives that we specify.

OpenGL provides us with the functionality to specify the objects, cameras, light sources and materials that we need to create 3-D scenes. OpenGL allows us to create objects by using points, line segments, triangles and polygons... in particular, we create objects through lists of vertices. The following listing allows us to create a simple triangle in the $xy$ plane (i.e. with $z = 0$ for every vertex):

```
0 glBegin(GL_POLYGON);
```

[1] The best known vendors are probably nVidia (www.nvidia.com) and ATI technologies (www.ati.com) - now owned by AMD

```
        glVertex3f(0.0f, 0.0f, 0.0f);
        glVertex3f(0.0f, 1.0f, 0.0f);
        glVertex3f(1.0f, 0.0f, 0.0f);
glEnd();
```

The function `glBegin()` specifies the type of shape that the vertices define. Each call to the `glVertex()` function gives the $x$, $y$ and $z$ position of the vertex. In this case we are creating a `GL_POLYGON` but we could change this to `GL_POINTS` to specify 3 points, or `GL_LINE_STRIP` to specify two connected lines. So, in the example above, we could have used `glVertex2f()` instead, as the $z$-component is 0 in each case. This is typical of the syntax used in OpenGL.

The `glVertex*()` function allows us many different functions to specify a vertex in 3-D space; we can use two, three and even four dimensional spaces in OpenGL. In this function the `*` can be replaced by any of the numbers 2, 3 or 4 and the particular type of the value of the vertex, e.g. `float(f)`, `double(d)`, `integer(i)` or `pointer(v)`: So, `glVertex2i(10,20)` would specify that we are creating a vertex using two integer values at $(10, 20)$. We generally use the OpenGL types that are defined in the OpenGL header files, such as: `#define GLfloat float`. This allows flexibility for implementations where we might want to change floats into doubles without affecting the code already written. Note that OpenGL functions all begin with the letters `gl`.

We can use arrays of points, using the following syntax:

```
0   GLfloat vertices[3][3] = {{0.0f, 1.0f, 0.0f},
                {0.87f, -0.5f, 0.0f}, {-0.87f, -0.5f, 0.0f}};
    // ...
    glBegin(GL_TRIANGLES);
      for(int i=0; i<3; i++)
5     {
        glVertex3fv(vertices[i]);
      }
    glEnd();
    // ...
```

For a full source example see: `Example1a`

The OpenGL user API provides us with a set of resources to help us in the development of our applications:

- *OpenGL Utility Library (GLU)* - This library provides us with a set of functions for drawing objects (such as spheres), manipulating textures, creation of standard views, NURBS surfaces etc.; functions that are common across many applications. The prefix for functions of this library is `glu`, not `gl`.

- *OpenGL Utility Toolkit (GLUT)* - This library provides us with functionality related to the windowing system on the current platform, for creating windows, create menus, interaction with the mouse, draw objects, display text etc. The prefix for functions of this library is `glut`. The GLUT library is available from Nate Robins' web site: `http://www.xmission.com/~nate/glut.html`.

To include these libraries, we can use `#include<GL/glut.h>` for example. Figure 1.3 illustrates a simplified OpenGL Windows pipeline.

### OpenGL Resources

The following are a list of the most important OpenGL references for this module:

- *The OpenGL Red Book* - This book is the programming guide for OpenGL. It is designed to explain OpenGL functionality and to give examples of its use. The Red Book is currently in its fifth edition, which covers OpenGL Version 2, but here is an

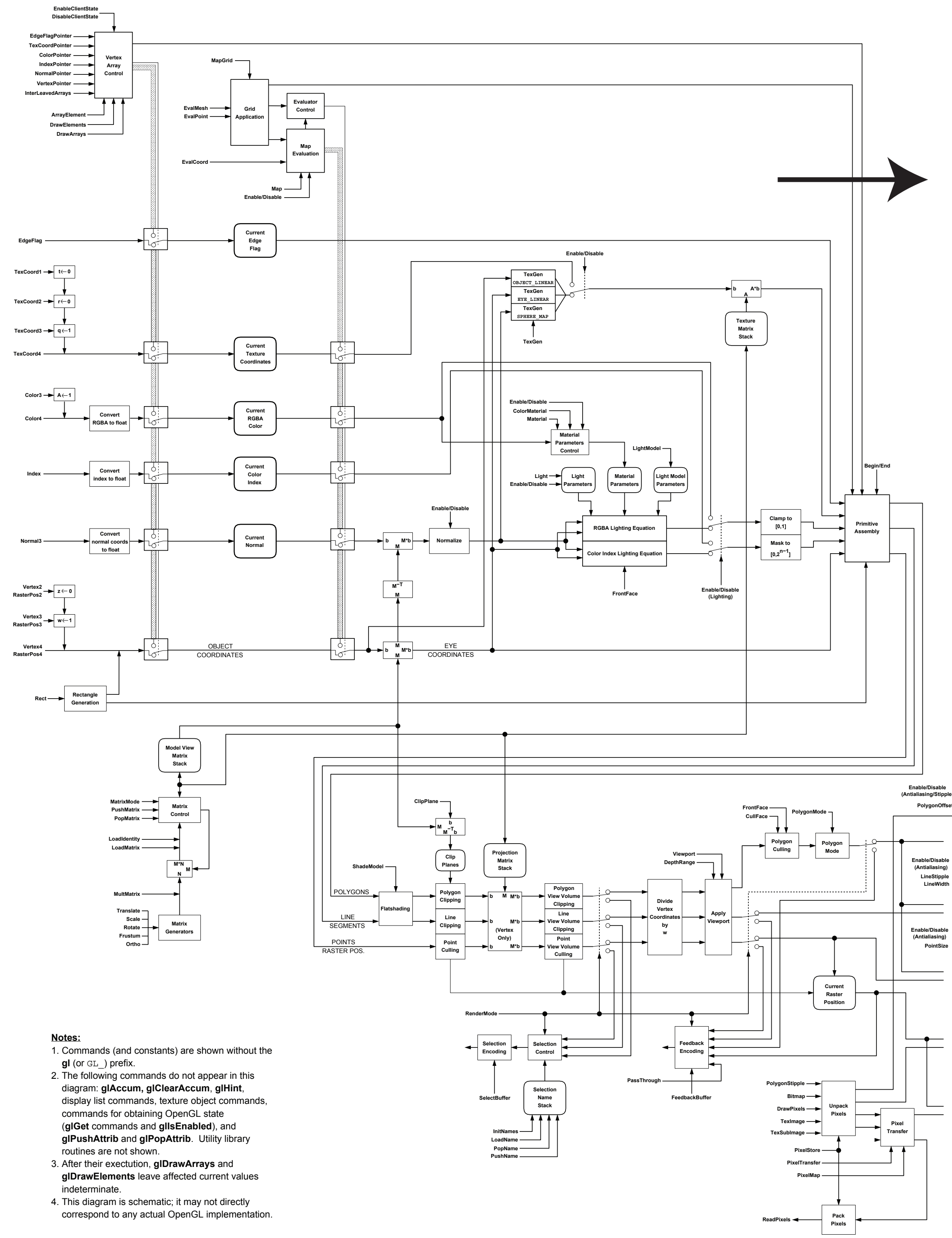



Figure 1.1: A schematic diagram of the OpenGL State Machine LHS ©Silicon Graphics Ltd.

# The OpenGL® Machine

The OpenGL ? graphics system diagram, Version 1.1. Copyright ? 1996 Silicon Graphics, Inc. All rights reserved.

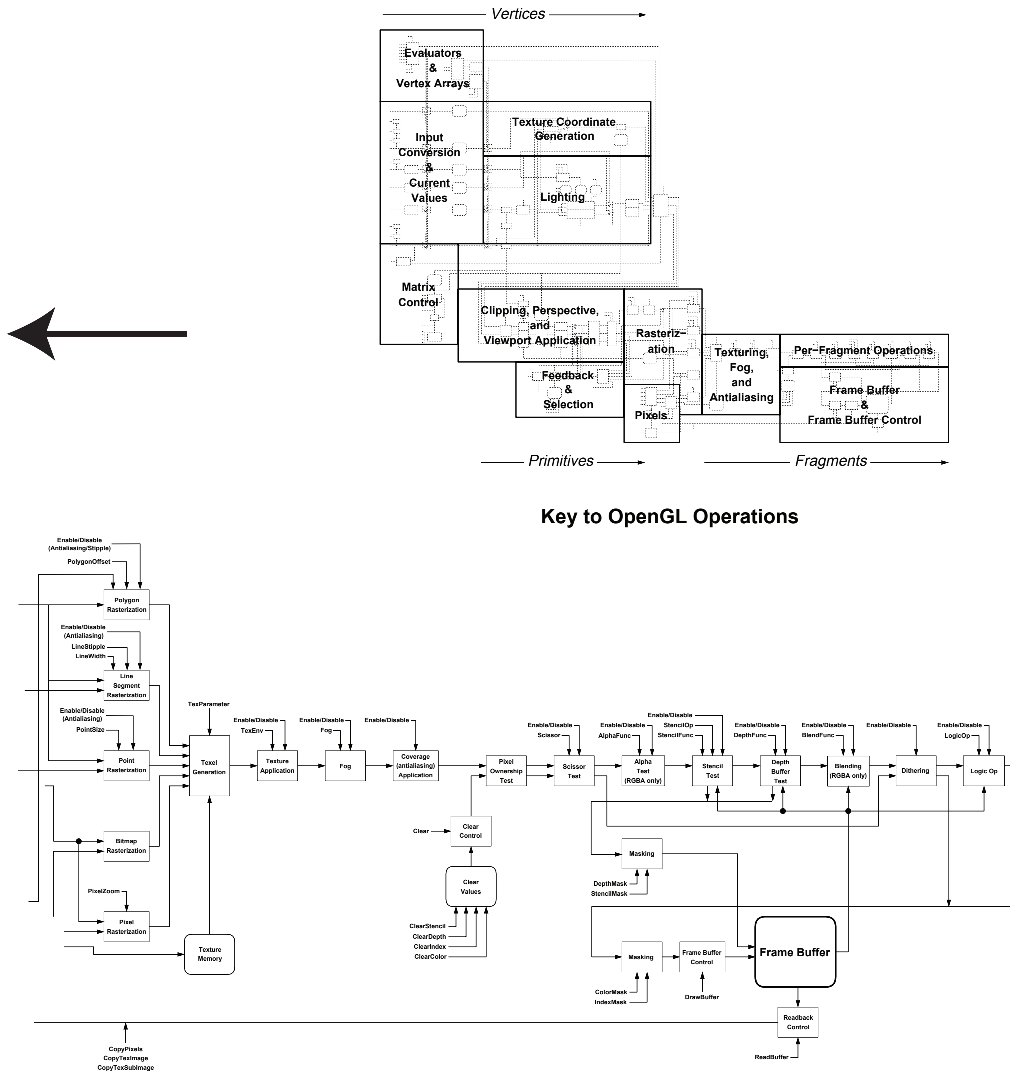



Figure 1.2: A schematic diagram of the OpenGL State Machine RHS ©Silicon Graphics Ltd.

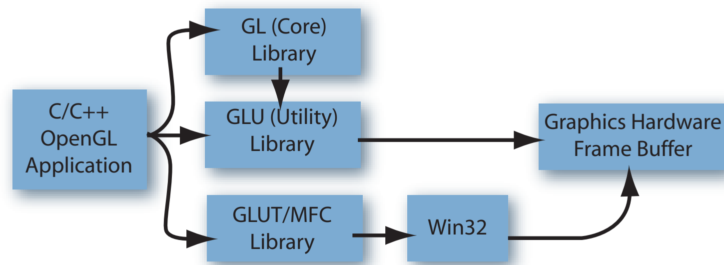



Figure 1.3: The Simplified OpenGL Windows Pipeline.

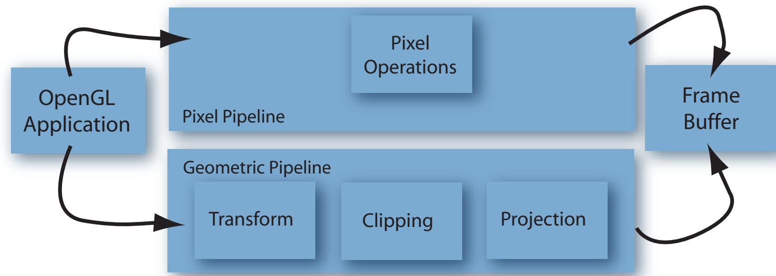



Figure 1.4: A simplified OpenGL pipeline.

older version of the book in `resources` `redbook.pdf` and associated examples in `resources` `redbookexamples.zip`.

- *The OpenGL Blue Book* - This book is the reference guide for OpenGL. It is designed to document all the functionality of OpenGL. It currently covers version 1.4 of OpenGL, but an old version is available here in `resources` `bluebook.pdf`, which covers OpenGL version 1.0.

- *OpenGL Specification Version 2.1* - The OpenGL specification 2.1 was released in August 2006 and it describes the latest features of OpenGL. The most up-to-date specification is available at `resources` `OpenGLSpecificationVersion2_1.pdf`.

## 1.3 Introduction to OpenGL Programming

### 1.3.1 OpenGL Primitives and Attributes

The core OpenGL library supports a relatively small set of geometric primitives (such as points, lines, polygons, curves and surfaces) and raster primitives (such as arrays of pixels). Figure 1.4 (see 1.1 and 1.2 for the full diagram) illustrates the series of geometric operations which are used to see if a primitive appears on the screen (frame buffer). The transform block allows 2-D/3-D geometric primitives to be rotated or translated. Because raster objects do not have geometric properties, they cannot be manipulated in the same way and so follow a parallel path through the pipeline.

As discussed previously, basic OpenGL geometric primitives are created using the following code:

```
0 glBegin(<type>);
        glVertex *(...);
        ...
```

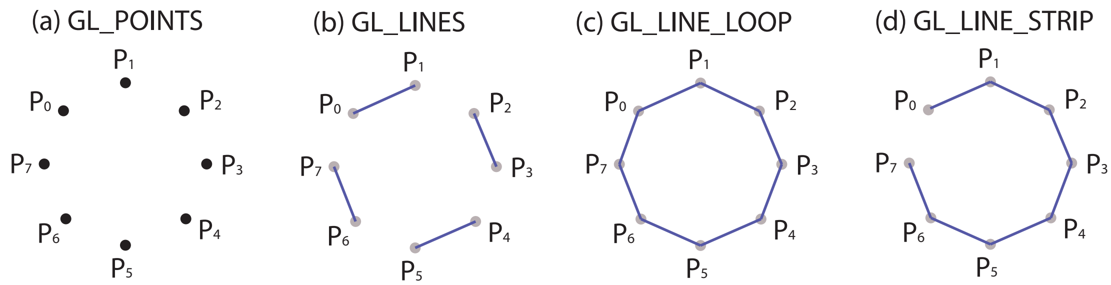



Figure 1.5: OpenGL Line Based Primitives.

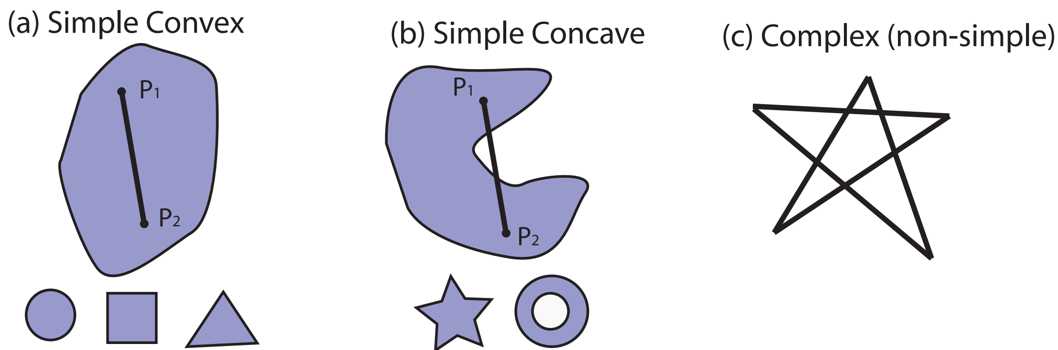



Figure 1.6: Displaying a polygon correctly (a) simple convex shapes (b) concave shapes (c) a non-simple polygon

```
        glVertex *(...);
glEnd();
```

where $*$ defines the particular `glVertex` function to call, and where `<type>` is one of the following (see figure 1.5(a)-(d)) in the case of line based primitives:

- `GL_POINTS` - Each vertex is displayed as a point, with a size of at least one pixel.
- `GL_LINES` - Each vertex pair are used as the start and end point, thus creating a line.
- `GL_LINE_LOOP` - Each vertex is used as the end point to create a line, using the previous vertex as the start point (except $P_0$). The line loop connects the last point to the first point, thus creating a loop.
- `GL_LINE_STRIP` - Each vertex is used as the end point to create a line, using the previous vertex as the start point (except $P_0$).

When we are creating objects that are constructed like line loops (see figure 1.5(c)), but are filled in some way, we use the term *polygon*. Polygons are the fundamental building blocks of complex models, in that these models are constructed from thousands of polygons.

For a polygon to be displayed correctly it must be (see figure 1.6):

- *simple* - the polygon should have a well-defined interior region. Lines connecting the vertices of the polygon should not cross.
- *convex* - an object is convex if all points on the object, or its boundary may be connected without leaving the boundary of the object[2].

[2]A set in Euclidean space **R** is a convex set if it contains all the line segments connecting any pair of

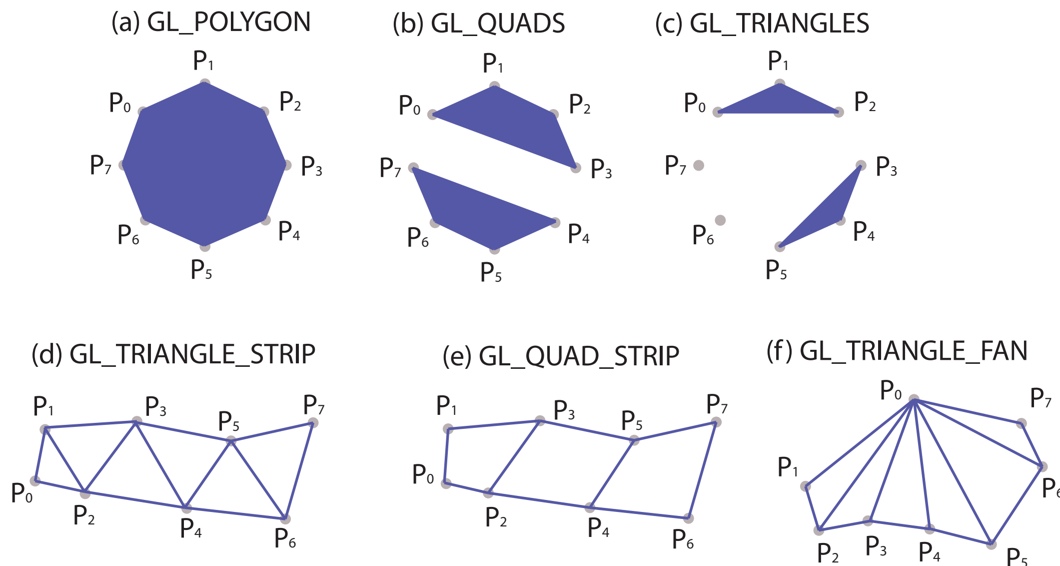



Figure 1.7: OpenGL Polygon Based Primitives.

- *flat* - for a 3-D polygon all points of the polygon must lie on the same plane. If we are using triangles to represent polygons then this is not an issue.

We can use the same segment of code to create polygon objects:

```
glBegin(<type>)
        glVertex ∗(...);
         ...
        glVertex ∗(...);
glEnd();
```

where $*$ defines the particular `glVertex` function to call, and where `<type>` is one of the following (see figure 1.7(a)-(f)) in the case of polygon based primitives:

- `GL_POLYGON` - Each successive vertex defines a line segment, with the final vertex connected to the first vertex. The interior of the polygon is filled according to the current OpenGL state. If we wish we can use the `glPolygonMode()` function to display the edges or points of the vertices, instead of filling the polygon.

- `GL_TRIANGLES` and `GL_QUADS` - For the purpose of efficiency, it is possible to use successive groups of either 3 or 4 vertices to create complex objects.

- `GL_TRIANGLE_STRIP`, `GL_QUAD_STRIP` and `GL_TRIANGLE_FAN` - When complex objects are being constructed that share vertices and edges, these types are particularly efficient at where each additional vertex is combined with the two previous vertices in the case of a triangle strip array; each new pair of vertices combine with the previous two vertices for the quad strip array; and using one fixed point and each subsequent vertex in the case of a triangle fan array.

Here is an example of using these basic primitives to create more complex objects. The easy way to do this is to use the fact that GLUT already provides a set of complex primitives such as a sphere. Example1b provides the source code project for an example of drawing a sphere using GLUT, and a screen capture of this application is shown in figure 1.8.

To use this example you must complete the following steps:

its points. If the set does not contain all the line segments, it is called concave. The convex hull is the set of points that we get by stretching a tight-fitting surface over the object (like an elastic band in the 2-D case). It is the smallest convex object that includes the the set of points.

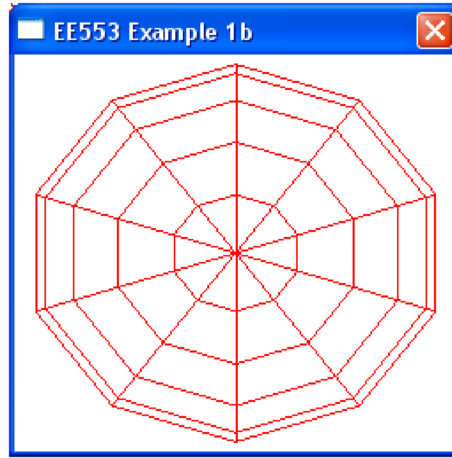



Figure 1.8: Using OpenGL GLUT to draw a sphere.

- Download the GLUT library from `http://www.xmission.com/~nate/glut.html`
- Place the GLUT.h file in the DevC++ header file "GL" directory (i.e. where GL.h is located)
- Place the .lib file in the DevC++ library directory.
- Place the GLUT32.dll file in the windows SYSTEM32 directory.
- In DevC++ Project ⟶ Project Options ⟶ press on "Parameters" and add the following libraries to your Linker: `-lglut32 -lwinmm -lgdi32`
- In DevC++ you seem to have to add a line `#define GLUT_DISABLE_ATEXIT_HACK` to your code to get it to work correctly.

We can then very simply create a red sphere by using the code:

```
0   // Set the current OpenGL color to red
    glColor4f(1.0, 0.0, 0.0, 1.0);
    // Draw a wire sphere
    glutWireSphere(0.95, 10, 10);
```

If we wish to do this the hard way! In this example we will approximate a sphere. We can do this very efficiently by using a set of polygons defined by lines of longitude, using either triangle strips of quad strips. A sphere can be described by the following equations:

$$\begin{aligned} x(\theta,\phi) &= r\sin\theta\cos\phi \\ y(\theta,\phi) &= r\cos\theta\cos\phi \\ z(\theta,\phi) &= r\sin\phi \end{aligned}$$

where $-180 \leq \theta \leq +180$ and $-90 \leq \phi \leq +90$. In this case when $\phi$ is fixed, we can draw circles of constant latitude by varying $\theta$ (the Earth's equator is a line of latitude). In our code we will only vary $-80 \leq \phi \leq +80$, so that the top and bottom (poles) of the sphere can be closed more accurately using a triangle fan. Figure 1.9(a) illustrates the approach to be taken where we will use `GL_QUAD_STRIP`s to fill in the largest area of the sphere and `GL_TRIANGLE_FAN`s to fill in the top and bottom of the sphere, ensuring that our sphere begins and ends with a single point. To get OpenGL to display all of the lines in the sphere it is necessary to set the mode to: `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);` The source code for this example can be found in the Example1c directory.

Here is the implementation of the code for figure 1.9(b), where both ends of the sphere have not yet been closed. For this example we call the method by the call:

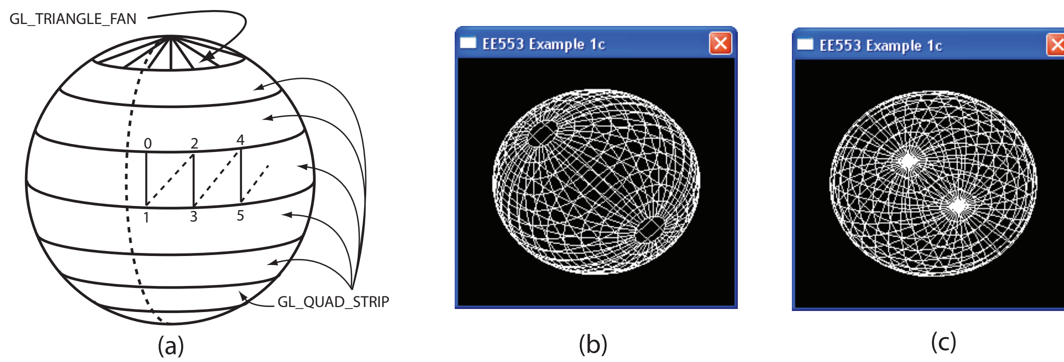



Figure 1.9: Drawing a sphere manually using the OpenGL core functionality. (a) illustrates the approach taken (b) shows a screen grab with the ends not closed and (c) shows a screen grab with the ends closed

`drawGLSphere(0.75f, 10.0f);` where the first arguments specifies the radius of the sphere and the second specifies the number of degrees to use for each component polygon, i.e. a larger number gives a lower resolution sphere mesh. Remember that the standard trigonometric functions in C++ work in radians, so while we can specify the behaviour of our sphere in degrees, we must do the calculations in radians.

```
0   void drawGLSphere(GLfloat radius, GLfloat step)
    {
        GLfloat x,y,z, c = 3.14159f/180.0f;
        for (GLfloat phi=-80.0f; phi<80.0; phi+=step)
        {
5           GLfloat phi_rad = c * phi;
            GLfloat phi_rad_end = c * (phi+step);
            glBegin(GL_QUAD_STRIP);
            for (GLfloat theta=-180.0f; theta<=180.0f; theta+=step)
            {
10              GLfloat theta_rad = c * theta;
                x = radius * sin(theta_rad)*cos(phi_rad);
                y = radius * cos(theta_rad)*cos(phi_rad);
                z = radius * sin(phi_rad);
                glVertex3f(x,y,z);
15              x = radius * sin(theta_rad)*cos(phi_rad_end);
                y = radius * cos(theta_rad)*cos(phi_rad_end);
                z = radius * sin(phi_rad_end);
                glVertex3f(x,y,z);
            }
20          glEnd();
        }
    }
```

To close the sphere at the top and bottom as in figure 1.9(b) it is necessary to add some more code within the same function:

```
0       // Close one end
        GLfloat closeRing_rad = c * 80.0;
        glBegin(GL_TRIANGLE_FAN);
        glVertex3f(0.0f, 0.0f, -radius);
        z = radius * -sin(closeRing_rad);
5       for(GLfloat theta=-180.0f; theta<=180.0f; theta+=step)
        {
            GLfloat theta_rad = c * theta;
            x = radius * sin(theta_rad) * cos(closeRing_rad);
            y = radius * cos(theta_rad) * cos(closeRing_rad);
```

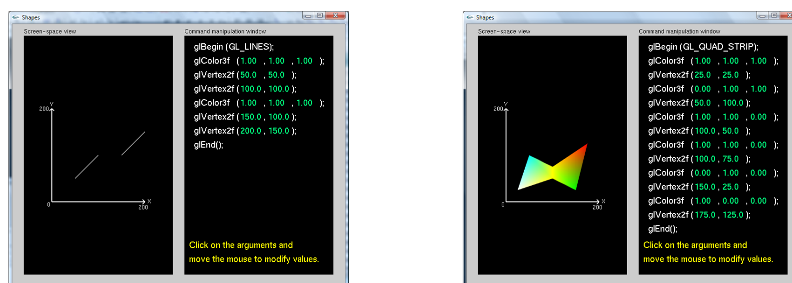



Figure 1.10: The Nate Robins' Tutorial Example on Shapes (a) demonstrates the use of `GL_LINES`, and (b) demonstrates the use of `GL_QUAD_STRIP`

.

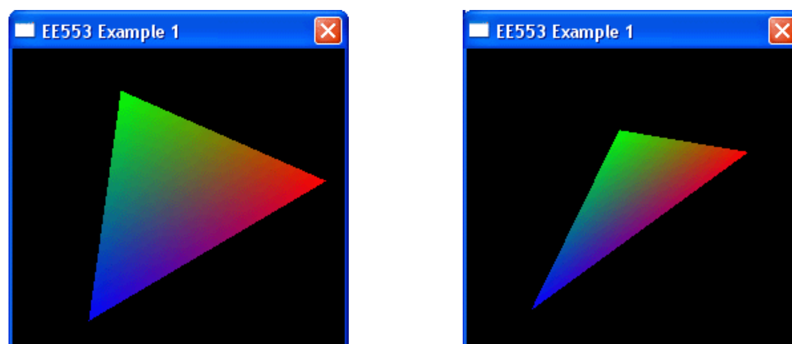



Figure 1.11: Two screen captures of the OpenGL Full Example

```
10            glVertex3f(x,y,z);
        }
        glEnd();

        // Close the other end
15      glBegin(GL_TRIANGLE_FAN);
        glVertex3f(0.0f, 0.0f, radius);
        z = radius * sin(closeRing_rad);
        for(GLfloat theta=-180.0f; theta<=180.0f; theta+=step)
        {
20            GLfloat theta_rad = c * theta;
              x = radius * sin(theta_rad) * cos(closeRing_rad);
              y = radius * cos(theta_rad) * cos(closeRing_rad);
              glVertex3f(x,y,z);
        }
25      glEnd();
```

1.10 demonstrates an example of drawing shapes when using the OpenGL API. In all of these tutorial examples you can change the parameters by left-clicking the mouse on their values. In this example you can also right-click to change from drawing `GL_LINES`, as in 1.10(a) to drawing `GL_QUAD_STRIPs` as in 1.10(b).

### 1.3.2 OpenGL Full Source Code Example

This example demonstrates the use of Dev C++ for the creation of a first OpenGL windows application. It is written for the MS Windows environment and should work on all PCs with and without 3D graphics cards. 1.11 shows two screen captures of this example running. The triangle is coloured red, green and blue at each vertex of the triangle, with the apex red.

The source code for this example is listed as:

```
0    /****************************************
      * EE563 Example Project 1
      * by: Derek Molloy
      * based on the DevC++ OpenGL template
      *
5     ****************************************/

     #include <windows.h> // Header File For Windows
     #include <gl\gl.h>    // Header File For The OpenGL32 Library
     #include <gl\glu.h>   // Header File For The GLu32 Library
10
     HDC          hDC=NULL; // Private GDI Device Context
     HGLRC        hRC=NULL; // Permanent Rendering Context
     HWND         hWnd=NULL; // Holds Our Window Handle
     HINSTANCE    hInstance; // Holds The Instance Of The Application
15
     // Function Declarations

     LRESULT CALLBACK WndProc (HWND hWnd, UINT message, WPARAM wParam,
             LPARAM lParam);
20   void enableOpenGL (HWND hWnd, HDC *hDC, HGLRC *hRC);
     int initGL(GLvoid);
     int drawGLScene(float theta);
     void disableOpenGL (HWND hWnd, HDC hDC, HGLRC hRC);

25   // WinMain - the starting point of our application

     int WINAPI WinMain (HINSTANCE hInst, HINSTANCE hPrevInstance,
         LPSTR lpCmdLine, int iCmdShow)
     {
30       WNDCLASS wc;
         HGLRC hRC;
         hInstance = hInst;
         MSG msg;
         BOOL bQuit = FALSE;
35       float theta = 0.0f;

         // register window class
         wc.style = CS_OWNDC;
         wc.lpfnWndProc = WndProc;
40       wc.cbClsExtra = 0;
         wc.cbWndExtra = 0;
         wc.hInstance = hInstance;
         wc.hIcon = LoadIcon (NULL, IDI_APPLICATION);
         wc.hCursor = LoadCursor (NULL, IDC_ARROW);
45       wc.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH);
         wc.lpszMenuName = NULL;
         wc.lpszClassName = "EE553GLExample";
         RegisterClass (&wc);

50       // create main window
         hWnd = CreateWindow ("EE553GLExample", "EE553_Example_1",
           WS_CAPTION | WS_POPUPWINDOW | WS_VISIBLE,
           0, 0, 256, 256,
           NULL, NULL, hInstance, NULL);
55
         // enable OpenGL for the window
         enableOpenGL (hWnd, &hDC, &hRC);
         initGL();

60       // program main loop
         while (!bQuit)
         {
             // check for messages
             if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
65           {
                 // handle or dispatch messages
                 if (msg.message == WM_QUIT)
                 {
                     bQuit = TRUE;
70               }
                 else
                 {
                     TranslateMessage (&msg);
                     DispatchMessage (&msg);
75               }
             }
             else
             {
                 drawGLScene(theta);
80               SwapBuffers (hDC);
                 theta += 1.0f;
                 Sleep (1);
             }
         }
85       // shutdown OpenGL
         disableOpenGL (hWnd, hDC, hRC);
         // destroy the window explicitly
         DestroyWindow (hWnd);
         return msg.wParam;
90   }


     // Window Callback Process

95   LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
     {
         switch (message)
         {
             case WM_CREATE:
100              return 0;
             case WM_CLOSE:
                 PostQuitMessage (0);
```

```
            return 0;
        case WM_DESTROY:
            return 0;
        case WM_KEYDOWN:
            switch (wParam)
            {
            case VK_ESCAPE:
                PostQuitMessage(0);
                return 0;
            }
            return 0;
        default:
            return DefWindowProc (hWnd, message, wParam, lParam);
    }
}


//Enable OpenGL

void enableOpenGL (HWND hWnd, HDC *hDC, HGLRC *hRC)
{
    PIXELFORMATDESCRIPTOR pfd;
    int iFormat;

    // get the device context (DC)
    *hDC = GetDC (hWnd);

    // set the pixel format for the DC
    ZeroMemory (&pfd, sizeof (pfd));
    pfd.nSize = sizeof (pfd);
    pfd.nVersion = 1;
    pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
    pfd.iPixelType = PFD_TYPE_RGBA;
    pfd.cColorBits = 24;
    pfd.cDepthBits = 16;
    pfd.iLayerType = PFD_MAIN_PLANE;
    iFormat = ChoosePixelFormat (*hDC, &pfd);
    SetPixelFormat (*hDC, iFormat, &pfd);

    // create and enable the render context (RC)
    *hRC = wglCreateContext( *hDC );
    wglMakeCurrent( *hDC, *hRC );
}

// Setup our GL Scene
int initGL(GLvoid)
{
        glShadeModel(GL_SMOOTH);        // Enable Smooth Shading
        glClearColor(0.0f, 0.0f, 0.0f, 0.5f);// Black Background
        glClearDepth(1.0f);                                     // Depth Buffer Setup
        glEnable(GL_DEPTH_TEST);                        // Enables Depth Testing
        glDepthFunc(GL_LEQUAL);                         // The Type Of Depth Testing To Do
        glHint(GL_PERSPECTIVE_CORRECTION_HINT,
         GL_NICEST); // Really Nice Perspective Calculations
        return TRUE;                                            //  Initialization  Went OK
}

// Called to update the scene - give us the animation
int drawGLScene(float theta)                    // Draw the scene;
{                                       // theta is amount to rotate

    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen
    glLoadIdentity();
    glPushMatrix ();
    glRotatef (theta, 1.0f, 1.0f, 0.0f);     // Rotate theta around the xy-axis
    glBegin (GL_TRIANGLES);                  // Drawing Using Triangles
            glColor3f (1.0f, 0.0f, 0.0f);    glVertex2f (0.0f, 1.0f);
            glColor3f (0.0f, 1.0f, 0.0f);    glVertex2f (0.87f, -0.5f);
            glColor3f (0.0f, 0.0f, 1.0f);    glVertex2f (-0.87f, -0.5f);
    glEnd ();
    glPopMatrix ();
        return TRUE; // Keep Going
}

// Disable OpenGL
void disableOpenGL (HWND hWnd, HDC hDC, HGLRC hRC)
{
    wglMakeCurrent (NULL, NULL);
    wglDeleteContext (hRC);
    ReleaseDC (hWnd, hDC);
}
```

### 1.3.3 OpenGL Colour

There are two colour models used in OpenGL; the *RGB Colour Model* and the *Indexed Colour Model*. OpenGL provides us with a fairly simple mechanism for specifying colour directly by providing the red, green and blue values directly. When using 24-bit colour ($2^{24}$ = 16.7M colours) we specify our colour components as values between 0.0 and 1.0, using functions such as:

```
        glColor3f(0.0f, 0.0f, 1.0f); // (R, G, B)
```

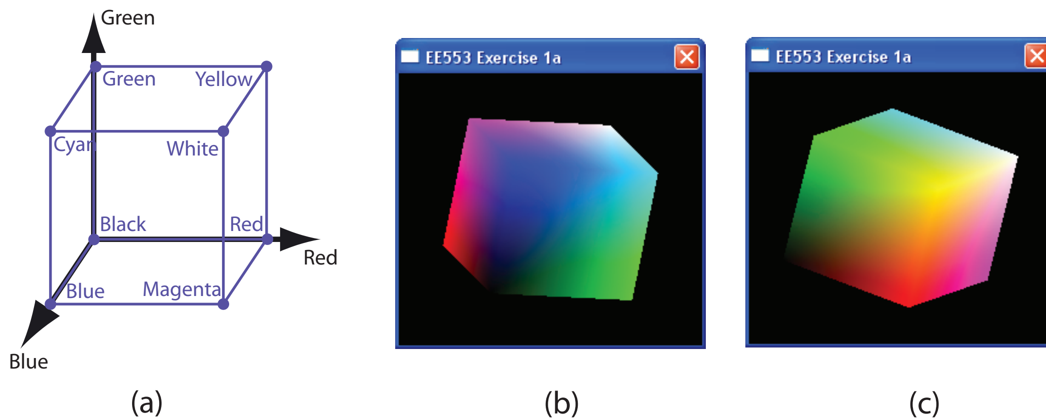



Figure 1.12: The Colour Cube Exercise (a) Colour Cube ;(b),(c) Screen Grabs of my solution.

which sets the current drawing colour to blue. The drawing colour will remain as blue until we change the colour again. OpenGL also provides support for transparency (opacity), which can be used for transparent models, image blending or for effects such as fog. The function takes a fourth parameter which specifies the level of opacity, which can vary from 0.0 (fully transparent) to 1.0 (fully opaque).

```
0    glColor4f(0.0f, 0.0f, 1.0f, 1.0f); // (R, G, B, A)
```

which will give us a fully opaque blue. If we wish to change the background colour we can use the function:

```
0    glClearColor(1.0f, 1.0f, 1.0f, 1.0f); // (R, G, B, A)
```

which will result in a white clearing colour. OpenGL also has support for indexed colour, by providing a user-defined colour-lookup table that is of size $2^n$, which contains 3 columns, one for each of R,G, and B. If the bit level for each of R,G and B is 8-bits then the user can choose $2^n$ individual colours out of the 16.7M possibilities, creating a palette. When you are in indexed colour mode you can select the current colour from the palette, by using:

```
0    glIndexi();
```

The indexed colour model is not so important any more due to the vast capabilities of todays' graphics cards.

### 1.3.4 Exercise: A Rotating Colour Solid

Write the code to generate a rotating colour solid which demonstrates the range of RGB colours that has a variable diameter. Please use the structure as illustrated in figure 1.12, where (a) illustrates the colour cube, and (b),(c) show screen grabs of my solution.

Please note that OpenGL uses variants of bilinear interpolation to generate the colours between the vertices. Therefore, as in the previous triangle example it is possible to specify colours on a vertex-by-vertex basis and allow OpenGL to do the colour mixing over the polygons.

You should try to write this code as efficiently as possible. You should be able to modify previous examples to generate the cube, but you should look in particular at the `glVertex3fv()` and `glColor3fv()` functions, which take an array of values, rather than individual values. My solution is given in the appendices.

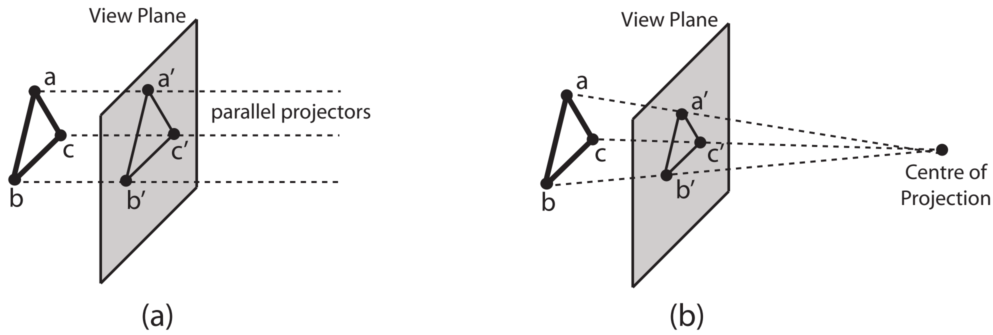



Figure 1.13: The projection of a triangle onto the image plane using (a) parallel projection and (b) perspective projection.

### 1.3.5 OpenGL Viewing

There are two ways to view an OpenGL scene; one way is by viewing objects in an orthographic manner, and the other is viewing objects in a perspective view. Looking down a long straight road you will see that the road width appears narrower the further away the section of road is - this is a perspective view. In the same example, with an orthographic view, the road would remain the same width for as far as you can see. Figure 1.13 illustrates the projection of a triangle onto the image plane using both parallel projection and perspective projection. It is very important that we understand what is happening in viewing when using OpenGL as changes to the viewing volume can have dramatic effects on the appearance of our scene.

The view volume is also very important for real-time visualisation as we can use *clipping* against the view volume to remove objects which are not within the view volume, where typically polygons outside of the view volume are discarded and polygons that intersect the view volume boundary are ‘clipped’. *Culling* is a similar operation that takes account of the viewer’s position to determine if part of an object cannot be seen; for example, in 3-D (the real world!) most objects are only 50% visible from any one point, as the back side of the object is occluded by the front side. A very simple test to see if we are looking at the front-side or back-side of a polygon is to use the view vector $\vec{n}$ and the polygon’s normal vector $\vec{n}_p$, where a particular polygon is visible if: $\vec{n}_p \cdot \vec{n} < 0$ (we will discuss this later).

#### Orthographic Viewing

Mathematically speaking, an orthographic view is what we would get if a camera had an infinitely long telephoto lens and we could place the camera infinitely far from the object - in 3-D it is an affine[3], parallel projection of an object onto a perpendicular plane. Orthographic projections are very common in CAD and 3-D modelling applications, where the user is presented with a top, front and side orthographic view. Objects in these orthographic views appear to be the exact same size, no matter how far they are from the viewer.

In OpenGL an orthographic projection can be specified as:

```
0  void glOrtho(GLdouble left, GLdouble right, GLdouble bottom,
                GLdouble top , GLdouble near, GLdouble far)
```

where distances are measured from the camera. Objects outside of this viewing volume are not displayed. If we do not specify a viewing volume then OpenGL uses its default 2x2x2 cube, with the origin as the centre.

[3]affine transformation between two vector spaces consists of a linear transformation followed by a translation $x \mapsto Ax + b$

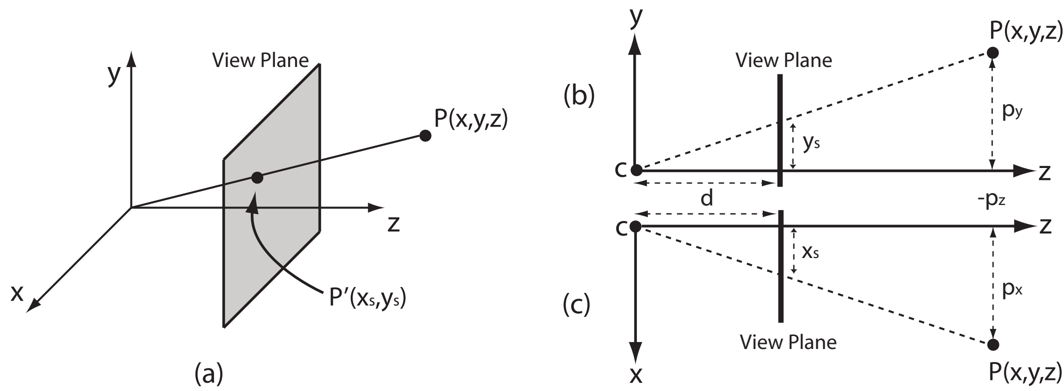



Figure 1.14: Illustration of the projection of a single point onto the view plane when using perspective transformation (a) illustrates the project, (b) shows the same figure when looking along the x-axis, and (c) shows the same figure when looking along the y-axis.

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

### Perspective Viewing

Figure 1.14 illustrates the perspective projection of a point onto the view plane, with the cross-sections of this projection illustrated in (b) and (c), where (b) illustrates the view when looking along the x-axis and (c) illustrates the view when looking along the y-axis. In OpenGL the centre of projection is always at the origin and pointing in the negative z-axis, hence the z-axis in this figure is pointing in the negative direction. The point $P(x, y, z)$ is a point in the view coordinate system, and $P'(x_s, y_s)$ is the projection of this point onto the view plane, which is normal to the z-axis. Assuming that the image plane is a distance $d$ in front of the centre of projection, the point $P(x, y, z)$ is projected onto $(x, y, -d)$. From similar triangles [4], we have:

$$\frac{x_s}{-d} = \frac{p_x}{p_z} \quad , \quad \frac{y_s}{-d} = \frac{p_y}{p_z} \quad , \quad z = -d \tag{1.1}$$

which gives:

$$x_s = \frac{-dp_x}{p_z} \quad , \quad y_s = \frac{-dp_y}{p_z} \quad , \quad z = -d \tag{1.2}$$

where all $z$ values are reduced to $-d$. Later we will see that OpenGL uses a technique called z-buffering for hidden surface removal, which does not reduce $z$ values to a single value.

### OpenGL Frustum

The mechanism for creating a perspective projection in OpenGL is through the definition of a pyramidal frustum[5], using the `glFrustum` function call, which has the following form:

[4] Two triangles are similar if their triples of vertex angles are the same. If two triangles are similar, then their sides will be proportional; meaning that every side of one triangle will be in a fixed ratio with the corresponding side of the other triangle

[5] Note: Frustum, not "Fustrum" - A frustum is the portion of a solid (normally a cone or pyramid), which lies between two parallel planes cutting the solid.

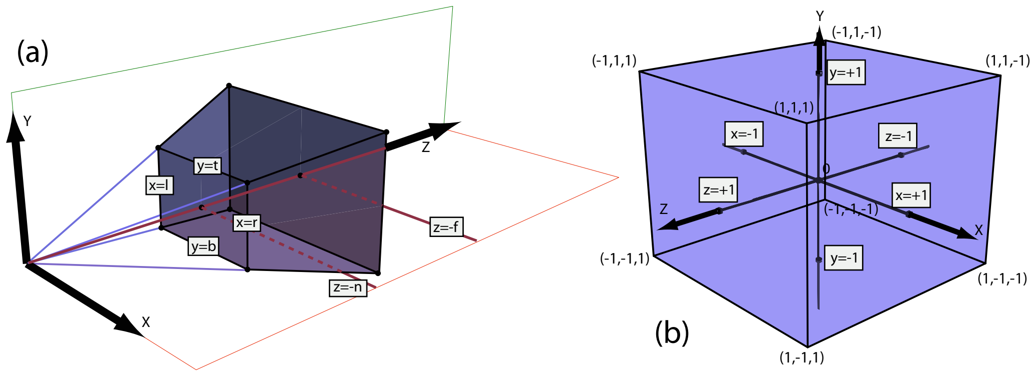



Figure 1.15: The OpenGL Frustum (a) illustrates the perspective view frustum volume and (b) illustrates its mapping to a homogeneous clip space cube.

```
0  void glFrustum(GLdouble left, GLdouble right, GLdouble bottom,
                  GLdouble top, GLdouble near, GLdouble far)
```

The perspective view frustum volume that is defined by `glFrustum` is illustrated in figure 1.15(a), where $x$ spans from $left$ to $right$, $y$ spans from $top$ to $bottom$, and $z$ spans from $near$ to $far$. Objects outside of this view frustum are clipped and are not displayed. OpenGL maps the frustum as illustrated in 1.15(b), where $left \mapsto -1$, $right \mapsto +1$, $bottom \mapsto -1$, $top \mapsto +1$, $near \mapsto -1$ and $far \mapsto +1$, reversing the direction of the $z$-axis.

From figure 1.15 and from equations 1.2 we can write:

$$x_s = \frac{-np_x}{p_z} \quad , \quad y_s = \frac{-np_y}{p_z} \quad , \quad z = -n \tag{1.3}$$

For an affine mapping of $u \mapsto v$ based on $u_0 \mapsto v_0$ and $u_1 \mapsto v_1$ we define the mapping as a scaling $a$ with a shift $b$ as:

$$v = au + b \tag{1.4}$$

It can be shown that this can be represented as:

$$v = (u - u_0)\frac{v_1 - v_0}{u_1 - u_0} + v_0 \tag{1.5}$$

which maps $u_0 \mapsto v_0$ and $u_1 \mapsto v_1$. Taking our present case where as illustrated in figure 1.15 OpenGL maps $x = left \mapsto x' = -1$, $x = right \mapsto x' = +1$, $y = bottom \mapsto y' = -1$ and $y = top \mapsto y' = +1$. In our case, where $u = x$, $v = x'$, $(u_0 = left) \mapsto (v_0 = -1)$ and $(u_1 = right) \mapsto (v_1 = +1)$, we can write equation this as:

$$\begin{aligned} x' &= (x - left)\frac{+1 - (-1)}{right - left} + (-1) && (1.6) \\ &= (x - l)\frac{2}{r - l} - 1 && (1.7) \\ &= \frac{2(x - l) - r + l}{r - l} \\ &= \frac{2x - 2l - r + l}{r - l} \end{aligned}$$

So,

$$x' = \frac{2x - (r + l)}{r - l} \tag{1.8}$$

When we substitute this into equation $(x = x_s)$ 1.3, we get:

$$x' = \frac{2(-n\frac{p_x}{p_z}) - (r + l)}{r - l} \tag{1.9}$$

Rewriting in the form that we will require for our projection matrix, we get:

$$-x'p_z = \frac{2np_x + (r+l)p_z}{r-l} \tag{1.10}$$

If we solve similarly for $y'$ we get:

$$-y'p_z = \frac{2np_y + (t+b)p_z}{t-b} \tag{1.11}$$

Unfortunately we also have to solve for $z'$, which is made a bit more difficult by the fact that the rasterisation state needs the reciprocal of $p_z$, so our affine mapping $v = au+b$ is $z' = a(\frac{1}{p_z}) + b$, where $s = -\frac{1}{n} \mapsto z' = -1$ and $s = -\frac{1}{f} \mapsto z' = +1$, so from equation 1.5 we get:

$$\begin{aligned}
z' &= \frac{(\frac{1}{p_z} + \frac{1}{n})(1+1)}{(-\frac{1}{f} + \frac{1}{n})} - 1 \\
&= \frac{(\frac{2}{p_z} + \frac{2}{n})}{(\frac{-n+f}{fn})} - 1 \\
&= \frac{2\frac{fn}{p_z} + 2f}{-n+f} - 1 \\
&= \frac{2\frac{fn}{p_z} + 2f}{f-n} - 1 \\
&= \frac{2\frac{fn}{p_z} + 2f - f + n}{f-n} \\
-z'p_z &= \frac{-2fn - fp_z - np_z}{f-n}
\end{aligned}$$

Therefore:

$$-z'p_z = \frac{-p_z(f+n) - 2fn}{f-n} \tag{1.12}$$

From a projected point $P'(x, y, z)$, the mapped from $P(x, y, z)$, we can now write from equations 1.10, 1.11 and 1.12 as:

$$-x'p_z = \frac{2n}{r-l}p_x + \frac{r+l}{r-l}p_z \tag{1.13}$$

$$-y'p_z = \frac{2n}{t-b}p_y + \frac{t+b}{t-b}p_z \tag{1.14}$$

$$-z'p_z = -\frac{f+n}{f-n}p_z - \frac{2fn}{f-n} \tag{1.15}$$

`glFrustum` describes a perspective matrix that produces a perspective projection. The current matrix (see `glMatrixMode`) is multiplied by the matrix and the result replaces the current matrix, just as if `glMultMatrix` was called with the following matrix as its argument:

The final projective transformation can now be written in terms of a matrix multiplication and homogeneous coordinates as:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

where $w = 1$. Remember, that if you create a 3-D vertex using `glVertex3d(x,y,z)`, OpenGL will create a 4-D homogeneous vectors by augmenting $(x, y, z)$ with $w = 1$, giving $(x, y, z, w = 1)$. In OpenGL all vertices are vectors and are represented by four homogeneous coordinates $(x, y, z, w)$ and all transformations are $4 \times 4$ matrices.

While we are at it we can also derive a matrix for the orthographic projection matrix. In the parallel projection the rays are parallel to one another and the lack of perspective distortion means that in OpenGL true $z$ coordinates can be interpolated directly, rather than using the reciprocal $(1/z)$ values. From equation 1.8, we have:

$$x' = \frac{2x - (r + l)}{r - l} \tag{1.16}$$

We get:

$$x' = \frac{2x}{r - l} - \frac{r + l}{r - l} \tag{1.17}$$

and, the same for $y$ gives us:

$$y' = \frac{2x}{t - b} - \frac{t + b}{t - b} \tag{1.18}$$

Because the z coordinate mapping is $-far \mapsto -1$ and $-near \mapsto +1$, therefore,

$$z' = \frac{-2z}{f - n} - \frac{f + n}{f - n} \tag{1.19}$$

Giving the final OpenGL orthographic matrix representation:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

### Matrix Modes

The OpenGL pipeline architecture requires that we apply a number of transformation matrices to manipulate our scene. As we have seen OpenGL involves placing and moving vertices in 3-D space, creating primitives using `GL_TRIANGLES`, `GL_QUAD_STRIPs` etc. and transforming them into a set of coordinates to make them have a 3-D appearance on the screen. The two matrices that are most important for this task are the *projection* and *model view* matrices. OpenGL is a state machine and so these variables are part of the state and remain there until changed.

The `GL_PROJECTION` is a matrix transformation that is applied to every vertex that comes after it and `GL_MODELVIEW` is a matrix transformation that is applied to every vertex on a particular model. The `GL_PROJECTION` matrix should contain only the projection transformation calls it needs to transform eye space coordinates into clip coordinates - i.e. think of the projection matrix as describing the attributes of the camera, such as the focal length, field of view etc.

The `GL_MODELVIEW` matrix, as its name implies, should contain modelling and viewing transformations, which transform object space coordinates into eye space coordinates. Remember to place the camera transformations on the `GL_MODELVIEW` matrix and never on the `GL_PROJECTION` matrix. Think of the model view matrix as the location in the 3-D world where you stand your camera and the direction you point it.

The only functions that should be called when the matrix mode is in `GL_PROJECTION` mode are: `glLoadIdentity()`, `glFrustum()`, `gluPerspective()`, `glOrtho()` and `glOrtho2()` - That is it[6].

To set up the view in OpenGL we will carry out the following steps:

[6] The one weak exception is that you could use `glLoadMatrix()` to set up your own projection matrices

- Position the camera - using the Model-View matrix
- Select a lens - set up the projection matrix
- Describe clipping - set up the view volume

By default, the object and camera frames are both the same (the model-view matrix is an identity matrix), where the camera is positioned at the origin, pointing in the -ve z direction. The default clipping view volume is a cube with all sides of length 2, centered at the origin. The default project is orthogonal.

So, if we begin with an object that straddles the z axis (i.e. contains +ve and -ve z values) and we wish to visualise the object we can either:

- Translate the camera in the +ve z direction, translating the camera frame, or,
- Move the object in the -ve z direction, translating the world frame.

Both of these operations are equivalent and determined by the model-view matrix, and performed by a call such as `glTranslatef(0.0f, 0.0f,` $-d_z$`)`, where $d_z > 0$.

We can set the matrix transformation by changing the *matrix mode*, using the `glMatrixMode()` function, as follows:

```
0 glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-1.0f, 2.0f, -1.5f, 1.5f, -1.0f, 1.0f);
      // l, r, b, t, zNear, zFar
```

By default, the matrix mode is `GL_MODELVIEW`. Once you set the matrix mode each following operation is applied to that particular matrix mode (matrix level) and below it. In this example `glLoadIdentity()` replaces the current matrix space with the *identity matrix*[7]. The output of this is applied to the sphere example from earlier and we can see that the `glOrtho()` function has adjusted the view. This is illustrated in figure 1.16(a). The next segment of code applies the `glTranslatef()` function to the view (see figure 1.16(b)).

```
0 glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-1.0f, 2.0f, -1.5f, 1.5f, -1.0f, 1.0f);
    // l, r, b, t, zNear, zFar
  glTranslatef( 0.5f, 0.5f, 0.0f ); // x, y, z
```

This code means current GL_PROJECTION Matrix = Identity * ORTHOGRAPHIC Matrix * TRANSLATION Matrix. In the example Example1d, we then multiply this by a ROTATION matrix `glRotatef(theta, 1.0f, 1.0f, 0.0f);`, where $\theta$ changes to give the impression of rotation in our scene. Interestingly, in OpenGL we often talk about changing the projection matrix as moving a camera in the 3-D world; however, this is not the case - we really move the entire world!

Figure 1.16 demonstrates the code segment:

```
0 glMatrixMode(GL_PROJECTION); // set up the camera attributes
  glLoadIdentity();
  gluPerspective(60.0, 1.0, 2.0, 10.0); // 60 degrees fov
  glMatrixMode(GL_MODELVIEW); // set up the camera location
  glLoadIdentity();
5 gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

where it is important to note how the projection and model view transforms work together. In this example, the projection transform sets up a 60.0-degree field of view, with an aspect ratio of 1.0. The near clipping plane is 2.0 units in front of the eye, and the far clipping plane is 10.0 units in front of the eye. This leaves a $z$ volume distance of

[7]This is semantically equivalent to calling `glLoadMatrix()` with a 4x4 identity matrix.

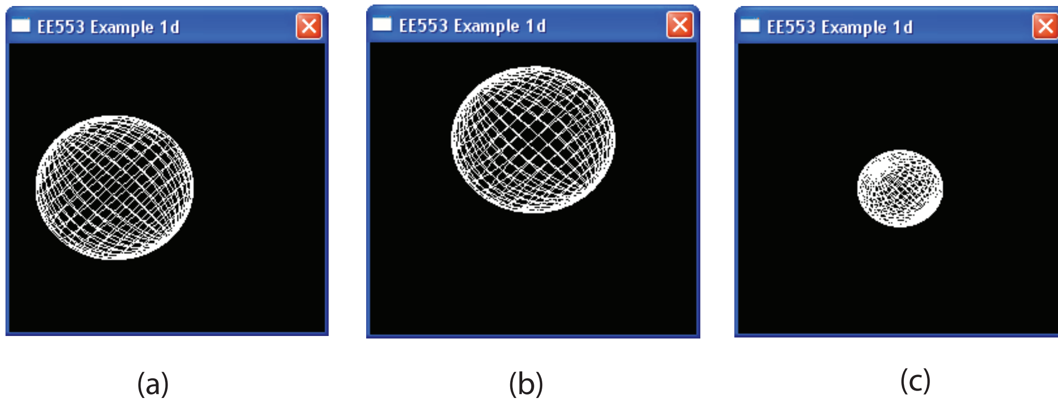



Figure 1.16: The OpenGL matrix mode screen capture example (a) demonstrates with `glOrtho(-1.0f, 2.0f, -1.5f, 1.5f, -1.0f, 1.0f);` (b) applying `glTranslatef( 0.5f, 0.5f, 0.0f ); // x, y, z` and (c) using `gluPerspective(50.0, 1.0, 3.0, 7.0);` instead.

8.0 units, which is plenty of room for our sphere example. The `GL_MODELVIEW` transform sets the eye position at (0.0, 0.0, 5.0), and the look-at point is the origin in the center of the sphere. So the eye is 5.0 units away from the "look at" point, which is within the $z$ volume distance of 8.0 units. Again, remember that the model-view matrix is used to position the camera (such as by using `gluLookAt()`) and to build models of objects; and the projection matrix is used to define the view volume and to set up the camera lens properties.

Rotating the model-view and projection matrices by the same matrix are not the same operations; At the matrix level, post-multiplication of the model-view matrix is the same as pre-multiplication of the projection matrix.

1.17 demonstrates the setup of a view using the `gluLookAt()` function. You can right click on the code view to change the scene from a `gluPerspective()` view setup, to either a `glOrtho()` or a `glFrustum()` view setup.

### 1.3.6 OpenGL Display Lists

Display lists allow us to improve the performance of OpenGL as we can cache OpenGL commands for later execution. This is particularly important if we plan to redraw the same object many times. Using display lists we can define the geometry (and OpenGL state changes) only once and execute it many times. If we think about a car being represented in 3-D, we would have a wheel model that we would have to draw four times. An efficient way to draw the car's wheels would be to store the geometry for one wheel in a display list and then execute the list four times, adjusting the model view matrix between each draw wheel operation.

The OpenGL model is a client/server model, where OpenGL is the server and your PC is the client. Because of the processing power of modern 3-D graphics cards the bottleneck in performance is due to the amount of traffic being passed between the client and the server. The fundamental mode of operation in OpenGL is *immediate mode* where as soon as our C++ program executes a statement that defines a primitive (or indeed vertices, attributes, viewing information etc.), the primitive is sent immediately to the graphics card server for display. When the scene needs to be redrawn, as in our rotating sphere example, then the vertices defining the sphere must be resent to the server. Clearly, this will involve sending large amounts of data between your C++ client application and the 3-D graphics server.

OpenGL also provides *retained mode* graphics, which provides us with display lists.

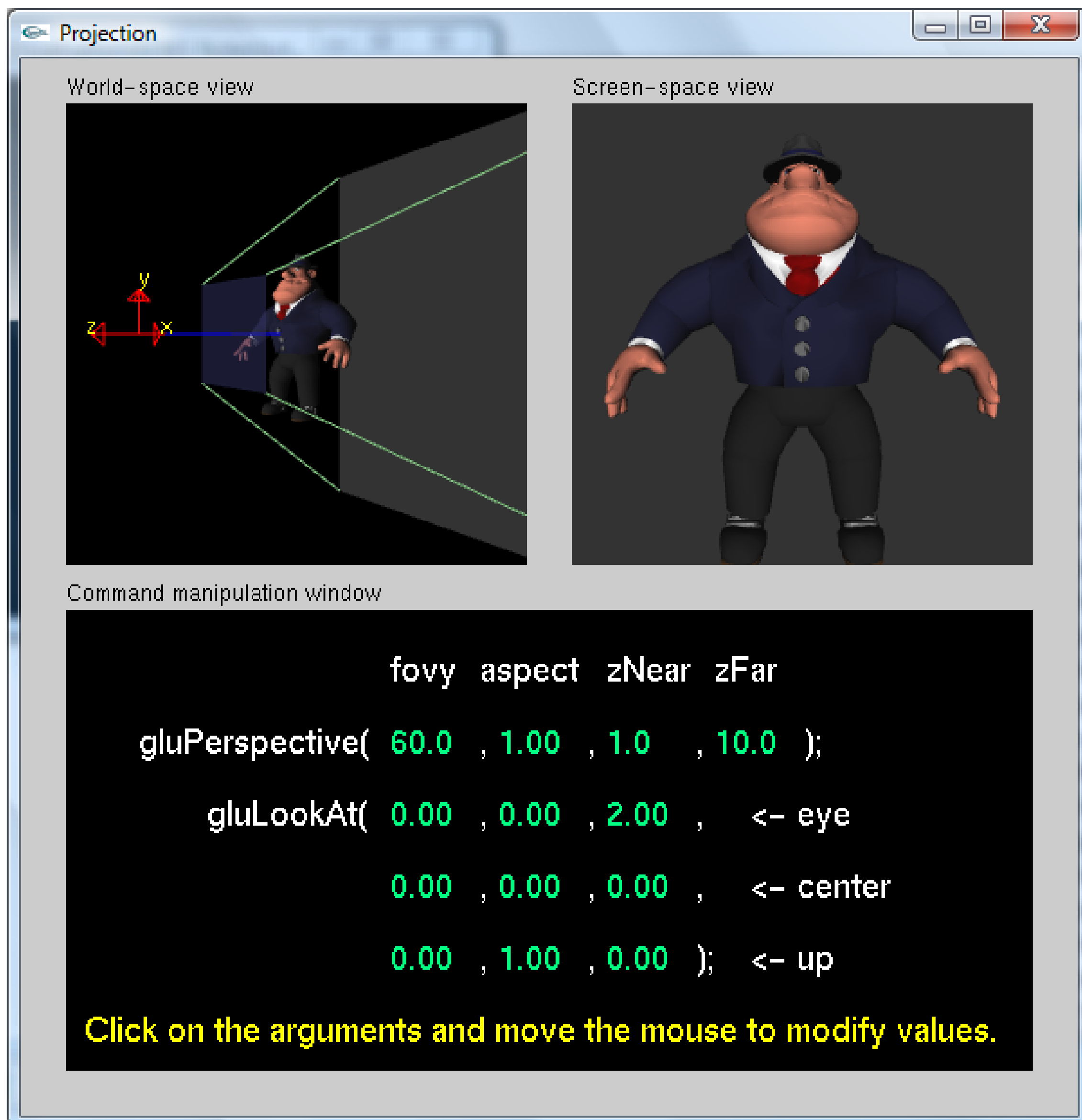



Figure 1.17: The Nate Robins' Tutorial Example on Projection

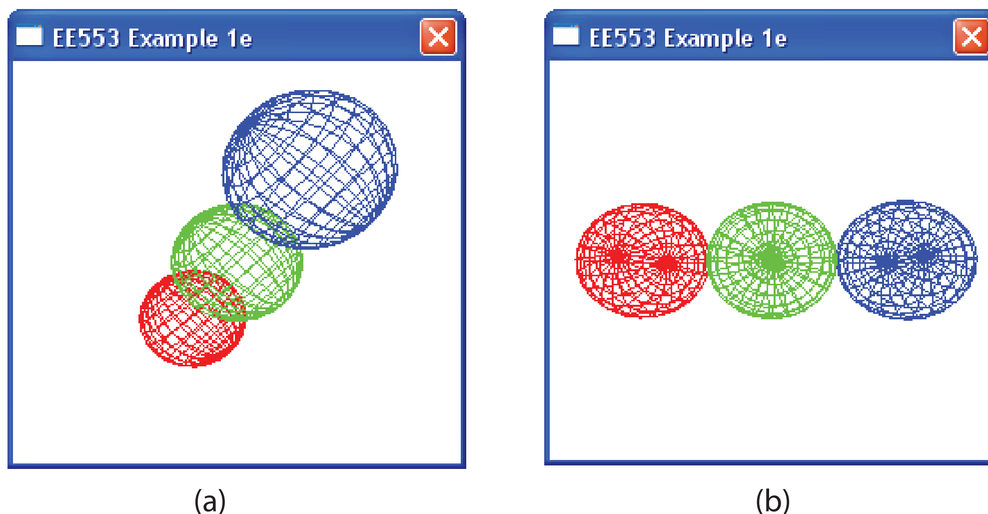



Figure 1.18: The Example 1e screen capture, illustrating the use of the display lists to display three spheres (a) and (b) capture two different views (note that in (a) the view is clearly perspective and not orthographic as the spheres are actually the same size, but display differently)

As discussed, we define the object once and place it in a display lists. Since display lists are part of the server state, therefore residing on the 3-D graphics server, the cost of repeatedly sending vertex information is dramatically reduced. Some graphics hardware may store display lists in dedicated memory or may store the data in an optimised form that is more compatible with the graphics hardware. There are some disadvantages with display lists; display lists require memory on the server and there is a small overhead in creating the display lists.

Here is an example, which builds on the previous sphere example to use display lists for retained mode in place of immediate mode graphics (The full source code is in example 1e):

```
0   #define SPHERE 1 // An identifier for our sphere

    void defineGLSphere(GLfloat radius, GLfloat step)
    {
        glNewList(SPHERE, GL_COMPILE); // define the sphere
5           ...
            for (GLfloat phi=−80.0f; phi<80.0; phi+=step)
            {
                ...
                glBegin(GL_QUAD_STRIP);
10              ...
                glEnd();
            }

            // Close one end
15          glBegin(GL_TRIANGLE_FAN);
              ...
            glEnd();

            // Close the other
20          glBegin(GL_TRIANGLE_FAN);
            ...
            glEnd();
        glEndList(); // end sphere definition
```

```
}

// Called to update the scene - give us the animation
int drawGLScene(float theta)
{
    ...
    glCallList (SPHERE); // actually draw the sphere
    ...
}
```

This example demonstrates how the `drawGLSphere()` function has been changed to a `defineGLSphere()` function, in which we only define the geometrical structure of a sphere, and do not actually draw it. The main change here is that the definition is surrounded by the lines of code `glNewList(SPHERE, GL_COMPILE);` and `glEndList();`. The very first line of code `#define SPHERE 1` gives a simple identifier for our defined list, which we then use in the `glNewList()` function. There is an alternative to using this `#define` style identifier, by using the `GLuint glGenLists(range)` method, which creates a contiguous set of empty display lists. It returns an unsigned integer which is the first number from the range that is unused.

The second parameter is a flag `GL_COMPILE`, which sends the list to the graphics server, but does not display its contents; if we wished to display the contents also, we could have used the parameter `GL_COMPILE_AND_EXECUTE` instead. To draw the sphere on the screen we use the line of code `glCallList(SPHERE);` which calls the display list. In the initial example of a car with four wheels, we can apply any transformations we wish to the current state, call the `glCallList()` function; apply more transformations and call it again, displaying the list in different locations each time. Figure 1.18 displays two screen captures of the same example, which demonstrates the use of the display lists.

```
int drawGLScene(float theta)
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE); // display all mesh lines of the sphere

    glMatrixMode(GL_PROJECTION); // Set up the camera properties
    glLoadIdentity();                                          // clear
    gluPerspective(60.0, 1.0, 1.0, 20.0);
      //60 degrees fov, aspect ratio 1.0, zNear 1.0, zFar 20.0 units
    glMatrixMode(GL_MODELVIEW); // move the camera location
    glLoadIdentity();
    gluLookAt(0.0, 0.0, 3.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
      // Sphere at (x,y,z)=(0,0,3), looking at (0,0,0), up vector
      // is (0,1,0) i.e. the up direction is up the y-axis
    glColor3f(0.0f, 1.0f, 0.0f);  // green - middle sphere
    glRotatef (theta, 1.0f, 1.0f, 0.0f); // Rotate theta around the xy-axis
    glCallList (SPHERE);          // draw the first sphere
    glColor3f(0.0f, 0.0f, 1.0f);  // blue - right sphere
    glTranslatef(1.0f, 0.0f, 0.0f);  //move to the right
    glCallList (SPHERE);                                // draw second sphere
    glColor3f(1.0f, 0.0f, 0.0f);  // red - left sphere
    glTranslatef(-2.0f, 0.0f, 0.0f); // move to the left
    glCallList (SPHERE);                                // draw third sphere
    return TRUE;
}
```

This source code example shows how we can use the same list three times, while changing the OpenGL states; in this example we change the colour of each sphere and the location at which it is drawn. The use of the `gluPerspective()` and `gluLookAt()` functions set up the view of our scene, placing a camera that has a field of view of 60 degrees, aspect ratio of 1.0 and zNear of 1.0 and zFar of 20.0 at the location $(0, 0, 3)$,

looking at the origin $(0, 0, 0)$ which the up direction of the camera $(0, 1, 0)$ - i.e. the +ve y-axis is up. The fact that this is a perspective view and not an orthographic view is clearly visible in figure 1.18(a) as the spheres have been created with the exact same radius, but the as the spheres rotate the one that is closest to the camera on rotation is clearly perceived to be larger.

### 1.3.7 OpenGL Stack

Because we need to apply many transformations to different objects and vertices in using OpenGL, we have to be careful that we only apply these transformations to the correct objects and vertices. This can be a difficult problem, but fortunately OpenGL has provided us with the matrix and attribute stacks. These stacks allow us to store the current state, by pushing it onto the stack; change the state to some other value, perform some operations and finally to restore the original state. We can do this by pushing and popping them from the stack.

It is good practice to push both the current matrices and attributes onto the their correct stacks when we enter a display list, and to pop them off when we are exiting the list. For example, we could use:

```
void defineGLSphere(GLfloat radius, GLfloat step)
{
    glNewList(SPHERE, GL_COMPILE); // define the sphere
        glPushMatrix();
        glPushAttrib(GL_ALL_ATTRIB_BITS);
       ...
      // Define sphere here.
       ...
      glPopMatrix();
      glPopAttrib();
    glEndList();  // end sphere definition
}
```

We have also altered Example1e to Example1f to give the exact same output, but by using the OpenGL stack to reset our drawing location of each sphere. The updated source code is:

```
   glPushMatrix();
   glColor3f(0.0f, 1.0f, 0.0f);   // green
   glCallList(SPHERE);
   glPopMatrix();

   glPushMatrix();
   glColor3f(0.0f, 0.0f, 1.0f);   // blue
   glTranslatef(1.0f, 0.0f, 0.0f);
   glCallList(SPHERE);
   glPopMatrix();

   glPushMatrix();
   glColor3f(1.0f, 0.0f, 0.0f);   // red
   glTranslatef(-1.0f, 0.0f, 0.0f);   //NB - note change to -1 from -2
   glCallList(SPHERE);
   glPopMatrix();
```

where the first sphere is drawn at $(0, 0, 0)$. The second sphere (blue) is translated by $(1, 0, 0)$, i.e. 1 unit in the +ve x-direction. Because we have done this encapsulated in a call to `glPushMatrix()` and `glPopMatrix()`, when we pop the matrix we have reset our universe to where it was originally, i.e. $(0, 0, 0)$. Now for the third sphere this is where we have a difference; we are translating 1 unit in the -ve x-direction. When we did not use

the matrix stack, we previously had to translate 2 units in the -ve x-direction, i.e. where the origin was the centre of the second sphere.

### 1.3.8 Input Events

We now wish to look at interacting with our OpenGL application. There are two main approaches that we can take; the first is to use the GLUT library and the second is to use the Windows API. Both approaches are very similar in that they both use events to drive the interaction with the application. This is very similar to the event approach that you would have seen when developing Java applications. Mouse events and keyboard events are the primary way of allowing a user to interact with our application.

We will use the Windows API for this module for handling our events as it will allow us to embed our OpenGL code within MFC applications, taking advantage of its components; however, the GLUT library is very similar in the way that it operates and it should not be difficult to transition between the two approaches.

The first events we will look at are keyboard input events. Example1g has implemented a key event handler that you can use as the basis of your own event driven application. Our application receives keyboard input in the form of keystroke messages and character messages. We need to map these messages into useful functionality within our application. For example:

```
0  LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                                    WPARAM wParam, LPARAM lParam)
   {
       switch (message)
       {
5          case WM_CREATE:
               ...
           case WM_KEYDOWN:
               switch (wParam)
               {
10             case VK_ESCAPE:
                   PostQuitMessage(0);
                   return 0;
               case VK_LEFT:
                   decreaseFOV();
15                 return 0;
               case VK_RIGHT:
                   increaseFOV();
                   return 0;
               ...
20             }
               return 0;
           case WM_CHAR:
               switch(wParam)
               {
25               case 'a':
                     decreaseFOV();
                     return 0;
                 case 's':
                     increaseFOV();
30                   return 0;
               }
               return 0;
           ...
       }
35 }
```

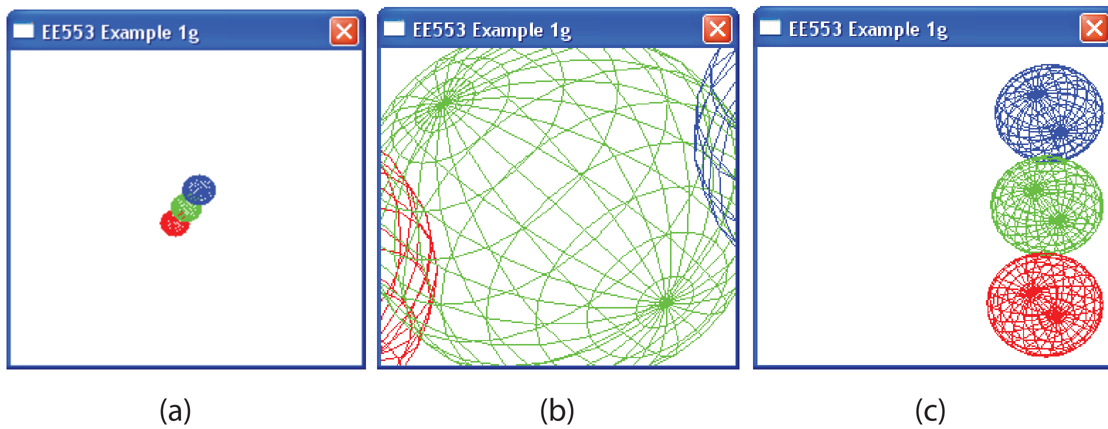



Figure 1.19: Example 1g which illustrates input events (a) is a screen capture of the initial view, (b) is the use of zoom (with the left/right arrows), and (c) is obtained by holding the left mouse button and dragging the mouse.

This segment of code shows that for calling our own functions `increaseFOV()` and `decreaseFOV()`, which are designed to increase and decrease the field of view of our application. The `WndProc()` function is being called when a key on the keyboard is pressed. Here we have two different types of messages being received, a `WM_KEYDOWN` message, which we must compare against the virtual-key code in the message's wParam parameter. This is used for dealing with the non-character keys, such as the arrows, function keys, escape keys and edit keys (such as INS, DEL, HOME, END etc.). For character keys (such as 'a', 'b' etc.) the window will receive a `WM_CHAR` message, and we can examine its wParam parameter to find out the character code of the key that was pressed.

Now we will look at the mouse:

```
GLfloat     fov  = 60.0f;
int         downX = 0, downY = 0;
float       xOffset  = 0.0f,  yOffset  = 0.0f;

LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
                WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_RBUTTONDOWN: // if the right button is down
             setFOV(60);
             xOffset=0.0f;
             yOffset=0.0f;
             return 0;
        case WM_LBUTTONDOWN:
             downX = (int)(short)LOWORD(lParam);
             downY = (int)(short)HIWORD(lParam);
        case WM_MOUSEMOVE: // if the mouse is moved
             if  (wParam == MK_LBUTTON) // and the left mouse button is down
             {
                int xPos = (int)(short)LOWORD(lParam);
                int yPos = (int)(short)HIWORD(lParam);
                int  diffx  = xPos − downX;
                int  diffy  = yPos − downY;
                // move by 1/10th of the difference
                xOffset+= ((float)diffx)/10;
                yOffset+= ((float)diffy)/10;
             }
             return 0;
```

```
            ...
30      }
    }

    // Other Windows Messages available − and their meanings
    //
35  // WM_LBUTTONDBLCLK − The left mouse button was double−clicked.
    // WM_LBUTTONDOWN − The left mouse button was pressed.
    // WM_LBUTTONUP − The left mouse button was released.
    // WM_MBUTTONDBLCLK − The middle mouse button was double−clicked.
    // WM_MBUTTONDOWN − The middle mouse button was pressed.
40  // WM_MBUTTONUP − The middle mouse button was released.
    // WM_RBUTTONDBLCLK − The right mouse button was double−clicked.
    // WM_RBUTTONDOWN − The right mouse button was pressed.
    // WM_RBUTTONUP − The right mouse button was released.
```

When the right button on the mouse is pressed a `WM_RBUTTONDOWN` message is generated, and we reset the view to the original position. When the left button is pressed for the first time a `WM_LBUTTONDOWN` message is generated and we record the current $(x, y)$ position of the mouse. This is important as this is the location of the mouse that we will use as the origin. The third mouse message that we will handle is `WM_MOUSEMOVE` message is called, we check to see if the left mouse button is also down with the `if (wParam == MK_LBUTTON)` call. If this is the case then we will extract the new $(x, y)$ position and use this to translate our view. In this example the `(int)(short)LOWORD(lParam);` call takes the low word out of the parameter, converts it to a short and then to an int; allowing lParam to contain both the $x$ and $y$ values in one value.

### 1.3.9 Double Buffering

Computers constantly redraw the visible video page (at around 70 times a second), and so it is difficult to make changes to the video page (such as creation or movement of complex 3-D objects) without the monitor showing the results before the graphics operation is complete. This results in ugly artifacts such as flickering, tearing and shearing.

The hardware method uses two graphics pages in VRAM. At any one time, one page is actively being displayed by the monitor, while the other, background page is being drawn. When drawing is complete, the roles of the two pages are switched, so that the previously shown page is now being modified, and the previously drawn page is now being shown. The page-flip is typically accomplished by modifying the value of a pointer to the beginning of the display data in the video memory. The hardware method means that artifacts will not be seen as long as the pages are switched over during the monitor's vertical blank period when no video data is being drawn. This method requires twice the amount of VRAM that is required for a single video page. The currently active and visible buffer is called the front buffer, while the background page is called the back buffer.

Since we are using a double buffered DC in Microsoft Windows, we set up double buffering through our call to:

```
0   ...
    pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL |
                              PFD_DOUBLEBUFFER;
    ...
    SetPixelFormat (*hDC, iFormat, &pfd);
```

anything we draw to the device context actually goes to a non-visible space in memory. This happens when you specify the `PFD_DOUBLEBUFFER` flag in the pixel format above. It is not displayed in the window until we allow it. This is very useful for rendering the scene off-screen and displaying the final image in one call. This can be performed in one simple function call to `SwapBuffers()`, to which we pass the GDI device context.

```
SwapBuffers( hDC );
```

in the `winMain()` function.

It is possible to setup double buffering in non-Microsoft Windows environments by using the GLUT library. It provides functions to set up a double buffered display for the platform on which we are compiling.

## 1.4 Some Math

### 1.4.1 3-D Math Notation

It is important to be able to clearly distinguish between the following geometric objects:

- *Point* - A point in 3-D graphics is simply a position in space. It specifies a location in space and has no volume, area or length. In OpenGL it is represented by a dot, and allow us to specify geometric objects. In a Carthesian 3-D space we can specify a point with three real number coordinates $P = (5.2, 2.4, -1.2)$
- *Scalar* - In linear algebra, real-numbers are called scalars. A vector space is defined as a set of vectors, a set of scalars and a scalar multiplication that takes a scalar $s$ and a vector $\vec{v}$ to give a new vector $s\vec{v}$.
- *Vector* - Vectors in 3-D space are ordered triples of real-numbers. The description of a vector is very similar to the description of a point; however, a vector indicates a direction and magnitude. For example, velocity is a good example of a vector. It is important to remember that a vector does not have a fixed position in space.

The linear vector space can contain both vectors and scalars; we can combine scalars and vectors to create new vectors, and we can add vectors to create new vectors. A Euclidean space is an extension to a vector space that adds a measure of size/distance, which allows us to define the length of our line segments. An affine space is an extension of the vector space that includes the concept of a point. In an affine space we can subtract two points to get a vector, or add a vector to a point to get another point (you cannot add points, as there is no origin).

Object-oriented programming with C++ allows us to use data abstraction to apply a set of operations to different data, independent of its type. We can use features of the language, in particular operator overloading to give us the ability to apply $x = a * (b + c)$; to real-number, matrices or vectors. However, while in theory this is a very powerful feature that should simplify our coding, in practice it will introduce some complexity; for example, what does the $*$ mean when we are operating on vectors, is it the dot-product or the cross-product? Well, we will have to decide.

### 1.4.2 Vectors - Math summary

A vector is characterised by a magnitude and a direction. To extract the magnitude of a 3-D vector (denoted using two vertical bars) we can use:

$$\|\vec{v}\| = \sqrt{v_x^2 + v_y^2 + v_z^2} \tag{1.20}$$

The magnitude is a real number. The unit vector is a method of finding out the simple direction of a vector (1.20 summarises the rules). A unit vector, also known as a normalised vector (or normal), has a magnitude of 1. To normalise a vector we divide it by its magnitude e.g.

$$\vec{v}_{norm} = \frac{\vec{v}}{\|\vec{v}\|}\ ,\ \vec{v} \neq 0 \tag{1.21}$$

We can multiply two vectors together using either the dot product or the cross product. The dot product (inner product) is the sum of the products of corresponding components, which results in a scalar value. In the 3-D case:

$$\vec{a} \cdot \vec{b} = a_x b_x + a_y b_y + a_z b_z \tag{1.22}$$

This tells us how similar two vectors are, where the larger the dot product, the more similar the two vectors are. The dot product is equal to the product of the magnitudes of the vector and the cos of the angle between the vectors:

$$\vec{a} \cdot \vec{b} = \left\|\vec{a}\right\|\left\|\vec{b}\right\| \cos(\theta) \tag{1.23}$$

Therefore we can solve for $\theta$ to give:

$$\theta = \arccos\left(\frac{\vec{a} \cdot \vec{b}}{\left\|\vec{a}\right\|\left\|\vec{b}\right\|}\right) \tag{1.24}$$

If we just examine the sign of the dot product.

$$\vec{a} \cdot \vec{b} = \begin{cases} > 0 & 0^\circ < \theta < 90^\circ & \text{acute, pointing in the same direction} \\ 0 & \theta = 90^\circ & \text{orthogonal (perpendicular)} \\ < 0 & 90^\circ < \theta < 180^\circ & \text{obtuse, pointing in the opposite direction} \end{cases}$$

If either $\vec{a}$ or $\vec{b}$ is equal to zero then $\vec{a} \cdot \vec{b} = 0$, a vector parallel to every vector.

The cross product (outer/vector product) gives a vector that is perpendicular to the original two vectors, allowing us to derive three mutually orthogonal vectors in a 3-D space from any two non-parallel vectors. It is given by:

$$\vec{a} \times \vec{b} = \tag{1.25}$$

$$\begin{bmatrix} a_x \\ a_y \\ a_z \end{bmatrix} \times \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} = \begin{bmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{bmatrix} \tag{1.26}$$

If these two vectors $\vec{a}$ and $\vec{b}$ are lying in the same plane, then the vector $\vec{a} \times \vec{b}$ pointing straight up out of the plane, perpendicular to $\vec{a}$ and $\vec{b}$. The coordinate system is given by the right-hand rule. So :

$$\begin{aligned} \vec{a} \times \vec{b} &= \vec{n}\left\|\vec{a}\right\|\left\|\vec{b}\right\| \sin\theta \quad \text{, or} \\ \left\|\vec{a} \times \vec{b}\right\| &= \left\|\vec{a}\right\|\left\|\vec{b}\right\| \sin\theta \end{aligned} \tag{1.27}$$

Where $\theta$ is the angle between $\vec{a}$ and $\vec{b}$ on the plane defined by the span of the vector and $\vec{n}$ is the unit vector perpendicular to both $\vec{a}$ and $\vec{b}$.

Vectors need a frame of reference, so that we can relate points and objects to the physical world. For example, I could ask you, where exactly are you standing now? You could not answer without some frame of reference (e.g. in a particular room in DCU with reference to its map, at a particular longitude/latitude on the earth - my office is at: West 6°15’22” North 53°23’08”, See: `http://maps.google.com/maps?q=++53+23.14+-+6+15.37`)- but West and North of where?[8]). In OpenGL we can relate this to world co-ordinates or view co-ordinates.

To create such a coordinate system, we consider a basis $\vec{v_1}, \vec{v_2}, \ldots, \vec{v_n}$ (a subset of vectors in vector space $V$ ), where a vector can be written as $\vec{v} = \alpha_1\vec{v_1} + \alpha_2\vec{v_2} + \ldots + \alpha_n\vec{v_n}$.

[8]This system still needs an origin - Latitude is the angle formed by a line from the centre of the earth to the equator and a line from the centre of the earth to your location, and Longitude is the angle formed by a line from the centre of the earth to the prime meridian at Greenwich England and a line from the centre of the earth to your location

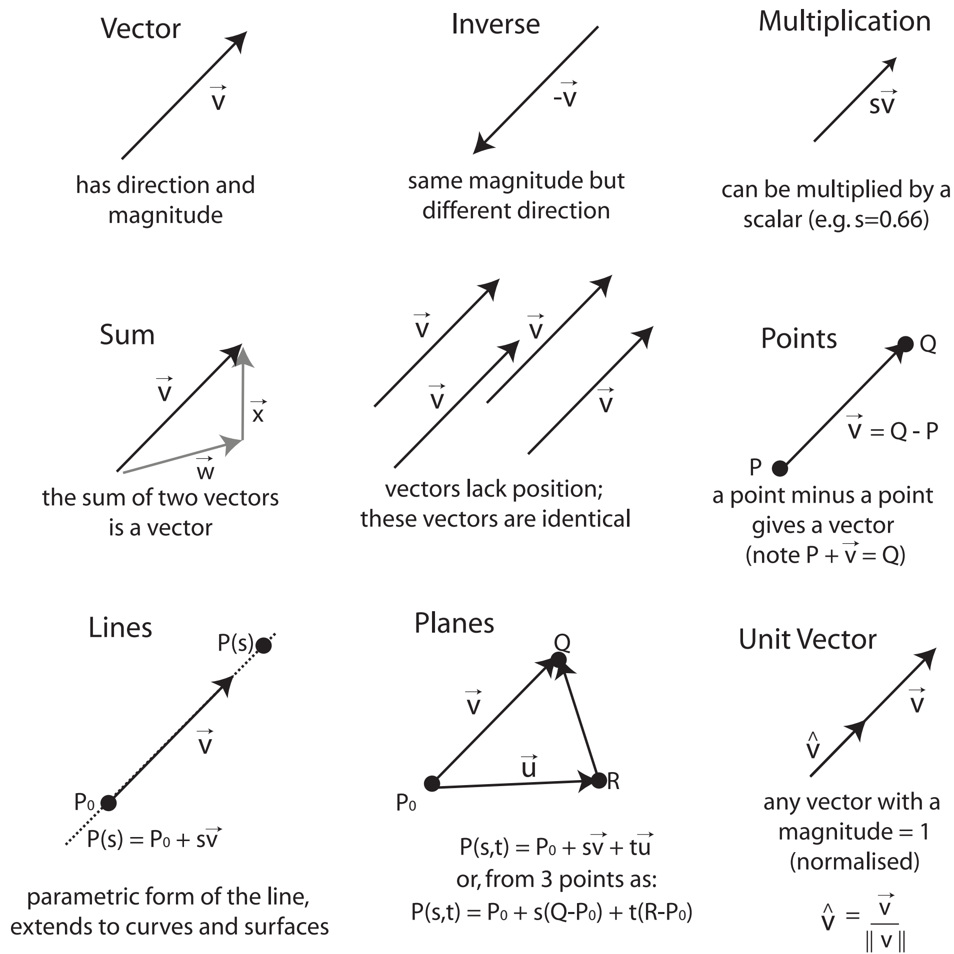



Figure 1.20: Summary of the rules when working with vectors

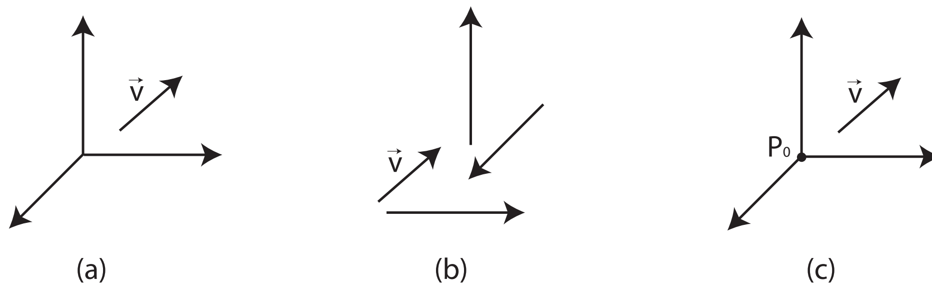



Figure 1.21: (a) An example of three basis vectors forming a co-ordinate system (b) The same example (c) A frame which includes an origin.

The list of scalars $\alpha_1, \alpha_2, \ldots, \alpha_n$ is the representation of $\vec{v}$ with respect to this basis, and we can write it as $\begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \end{bmatrix}^T$. A basis vector for a vector space is a set of linearly independent vectors where every object in the vector space can be described as a linear combination of the basis vectors. For instance any position vector in a three-dimensional coordinate system with axis $x$, $y$ and $z$ can be described as a weighted sum of a vector of unit length in the $x$-axis, unit length in the $y$-axis and unit length in the $z$-axis. In this example the weights in the weighted sum are the $x$, $y$ and $z$ coordinate of the point which is described by the position vector. e.g. $\vec{a} = \begin{bmatrix} 1 & -2 & 5 \end{bmatrix}^T$. For instance, any position vector in a three-dimensional coordinate system with axis $x$, $y$ and $z$ can be described as a weighted sum of a vector of unit length in the $x$-axis, unit length in the $y$-axis and unit length in the $z$-axis. In this example the weights in the weighted sum are the $x$, $y$ and $z$ co-ordinate of the point which is described by the position vector. $\vec{a} = \begin{bmatrix} 1 & -2 & 5 \end{bmatrix}^T$.

Referring to 1.21(a) illustrates an example of three basis vectors forming a co-ordinate system in which to place our vector $\vec{v}$. However, since vectors have no fixed location, (b) is just as valid a co-ordinate system as (a). Think of this as longitude and latitude without the intersection of the prime meridian and equator providing us with an origin; we would have a co-ordinate system which would allow us to use a vector to describe a movement from one point in the world to another, e.g. a movement $\vec{a} = \begin{bmatrix} 0°00' & -3°00' \end{bmatrix}^T$, might describe a trip from Dublin to Galway, but it might also be a trip from London to Cardiff.

In other words, a co-ordinate system is insufficient to represent points. If we work in an affine space, we can add an origin to our basis vectors to form a frame, as in figure 1.21(c).In the longitude and latitude example, this origin is the intersection of the prime meridian and the equator, thus providing us with a reference by which we can describe any point on the Earth. Mathematically, a frame is determined by $(P_0, \vec{v_1}, \vec{v_2}, \vec{v_3})$, where within this frame every vector can be written as $\vec{v} = \alpha_1\vec{v_1} + \alpha_2\vec{v_2} + \ldots + \alpha_n\vec{v_n}$ and every point can be written as $P = P_0 + \beta_1\vec{v_1} + \beta_2\vec{v_2} + \ldots + \beta_n\vec{v_n}$. It is important not to confuse points and vectors; In this example we could write $\vec{v} = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \end{bmatrix}^T$ and $P = \begin{bmatrix} \beta_1 & \beta_2 & \beta_3 \end{bmatrix}^T$, but remember that a vector has no position, so the point is at a fixed location $(\beta_1, \beta_2, \beta_3)$,but the vector could be anywhere.

Homogeneous co-ordinates allow affine transformations to be easily represented by a matrix. Homogeneous co-ordinate systems are key to computer graphic systems as all standard transformations can be implemented with matrix multiplications using 4x4 matrices, taking advantage of an accelerated hardware pipeline.

We can write a vector $\vec{v}$ and point $P$ as:

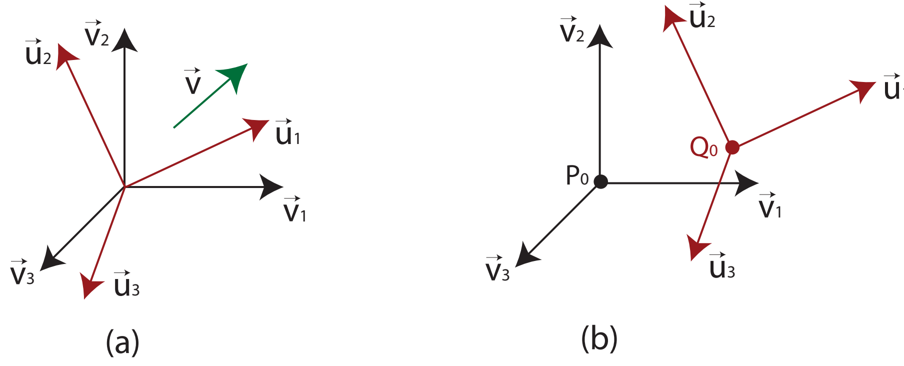



Figure 1.22: (a) Two different bases with the one vector $\vec{v}$, and (b) the change of frames.

$$\vec{v} = \alpha_1\vec{v_1} + \alpha_2\vec{v_2} + \alpha_3\vec{v_3} = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & 0 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 & P_0 \end{bmatrix}^T \tag{1.28}$$

$$P = P_0 + \beta_1\vec{v_1} + \beta_2\vec{v_2} + \beta_3\vec{v_3} = \begin{bmatrix} \beta_1 & \beta_2 & \beta_3 & 1 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 & P_0 \end{bmatrix}^T \tag{1.29}$$

Therefore, a four-dimensional homogeneous representation is $\vec{v} = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & 0 \end{bmatrix}^T$ and $P = \begin{bmatrix} \beta_1 & \beta_2 & \beta_3 & 1 \end{bmatrix}^T$. If we consider two representations $a = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \end{bmatrix}$ and $b = \begin{bmatrix} \beta_1 & \beta_2 & \beta_3 \end{bmatrix}$of the same vector $\vec{v}$ with respect to two different bases (as in figure 1.22), where:

$$\vec{v} = \alpha_1\vec{v_1} + \alpha_2\vec{v_2} + \alpha_3\vec{v_3} = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \end{bmatrix} \begin{bmatrix} v_1 & v_2 & v_3 \end{bmatrix}^T \tag{1.30}$$

$$\vec{v} = \alpha_1\vec{u_1} + \alpha_2\vec{u_2} + \alpha_3\vec{u_3} = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \end{bmatrix} \begin{bmatrix} u_1 & u_2 & u_3 \end{bmatrix}^T \tag{1.31}$$

Figure 1.22 illustrates the change of basis for a vector $\vec{v}$. It is possible to represent each of the second basis vectors $\vec{u_1}$, $\vec{u_2}$ and $\vec{u_3}$ in terms of the first basis as:

$$\vec{u_1} = \gamma_{11}\vec{v_1} + \gamma_{12}\vec{v_2} + \gamma_{13}\vec{v_3} \tag{1.32}$$

$$\vec{u_2} = \gamma_{21}\vec{v_1} + \gamma_{22}\vec{v_2} + \gamma_{23}\vec{v_3} \tag{1.33}$$

$$\vec{u_3} = \gamma_{31}\vec{v_1} + \gamma_{32}\vec{v_2} + \gamma_{33}\vec{v_3} \tag{1.34}$$

We can write this in matrix form $a = M^T b$, where:

$$M = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{32} & \gamma_{33} \end{bmatrix} \tag{1.35}$$

We can also apply this process in homogeneous coordinate representation to extend the process to both vectors and points. 1.22(b) illustrates two different frames $(P_0, v_1, v_2, v_3)$ and $(Q_0, u_1, u_2, u_3)$, where any point or vector can be represented in either frame. So, if we wish to represent the second frame in terms of the first frame, we can use:

$$\vec{u_1} = \gamma_{11}\vec{v_1} + \gamma_{12}\vec{v_2} + \gamma_{13}\vec{v_3} \tag{1.36}$$

$$\vec{u_1} = \gamma_{21}\vec{v_1} + \gamma_{22}\vec{v_2} + \gamma_{23}\vec{v_3} \tag{1.37}$$

$$\vec{u_1} = \gamma_{31}\vec{v_1} + \gamma_{32}\vec{v_2} + \gamma_{33}\vec{v_3} \tag{1.38}$$

$$\vec{Q_0} = \gamma_{41}\vec{v_1} + \gamma_{42}\vec{v_2} + \gamma_{43}\vec{v_3} + \gamma_{44}P_0 \tag{1.39}$$

So, within two frames any vector or point has a representation $a = \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 \end{bmatrix}$ in the first frame and $b = \begin{bmatrix} \beta_1 & \beta_2 & \beta_3 & \beta_4 \end{bmatrix}$ in the second frame, where $\alpha_4 = \beta_4 = 0$ for vectors and $\alpha_4 = \beta_4 = 1$ for a point, then we can write $a = M^T b$, as before, where:

$$M = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix} \tag{1.40}$$

The matrix $M$ is a 4x4 matrix that specifies an affine transformation in homogeneous co-ordinates, which has 12 degrees of freedom, as 4 of the elements in the matrix are fixed. Affine transformations preserve line structure, and allow us to apply rotations, translations, scaling and shear to our scene objects. For 3-D graphics applications these transforms are very efficient as we only need to transform the two end points of a line, and we can let the hardware implementation draw the line segment between these transformed points.

### 1.4.3 A Vector Class

Here is an example 3-D Vector class.

```
/*  Vec3.h: General Purpose 3-D Vector class.
/*  EE563 - 3-D Graphics and Visualisation.
/*        General purpose float vector for use as vertices, vectors and normals.
*/

#if !defined _VEC3
#define _VEC3

#include<iostream>
using std::ostream;
#include<cfloat>
#include<cmath> // for the isNaN (is not a number) function

class Vec3
{
        public:
                float v[3]; // states are public for ease of access

        Vec3() { v[0]=0.0f; v[1]=0.0f; v[2]=0.0f;}        // constructor
        Vec3(float x,float y,float z) { v[0]=x; v[1]=y; v[2]=z; }

                // overloaded operators
                inline bool operator == (const Vec3& v) const
                  { return v[0]==v.v[0] && v[1]==v.v[1] && v[2]==v.v[2]; }

        inline bool operator != (const Vec3& v) const
                  { return v[0]!=v.v[0] || v[1]!=v.v[1] || v[2]!=v.v[2]; }

                inline bool operator < (const Vec3& v) const
        {
            if (v[0] < v.v[0]) return true;
            else if (v[0] > v.v[0]) return false;
            else if (v[1] < v.v[1]) return true;
            else if (v[1] > v.v[1]) return false;
            else return (v[2] < v.v[2]);
        }
```

```
        inline float* ptr() { return v; }
        inline const float* ptr() const { return v; }

40      inline void set( float x, float y, float z)
        {
            v[0]=x; v[1]=y; v[2]=z;
        }

45      inline float& operator [] (int i) { return v[i]; }
        inline float operator [] (int i) const { return v[i]; }

        inline float& x() { return v[0]; }
        inline float& y() { return v[1]; }
50      inline float& z() { return v[2]; }

        inline float x() const { return v[0]; }
        inline float y() const { return v[1]; }
        inline float z() const { return v[2]; }
55
        inline bool valid() const { return !isNaN(); }
        inline bool isNaN() const { return _isnan(v[0]) || _isnan((double)v[1])
                || _isnan((double)v[2]); }

60
        inline float operator * (const Vec3& rhs) const // dot product
        {
            return v[0]*rhs.v[0]+v[1]*rhs.v[1]+v[2]*rhs.v[2];
        }
65
        inline const Vec3 crossProduct(const Vec3& rhs) const // cross product
        {
            return Vec3(v[1]*rhs.v[2]-v[2]*rhs.v[1],
                v[2]*rhs.v[0]-v[0]*rhs.v[2] ,
70              v[0]*rhs.v[1]-v[1]*rhs.v[0]);
        }
        inline const Vec3 operator ^ (const Vec3& rhs) const // cross product
        {
            return this->crossProduct(rhs); // call method just above
75      }

        inline const Vec3 operator * (float rhs) const // multiply by scalar
        {
            return Vec3(v[0]*rhs, v[1]*rhs, v[2]*rhs);
80      }

        inline Vec3& operator *= (float rhs)
        {
            v[0]*=rhs; v[1]*=rhs; v[2]*=rhs; return *this;
85      }

        inline const Vec3 operator / (float rhs) const // divide by scalar
        {
            return Vec3(v[0]/rhs, v[1]/rhs, v[2]/rhs);
90      }

        inline Vec3& operator /= (float rhs) // divide by scalar
        {
            v[0]/=rhs; v[1]/=rhs; v[2]/=rhs; return *this;
95      }

        inline const Vec3 operator + (const Vec3& rhs) const // vector add
```

```
        {
            return Vec3(v[0]+rhs.v[0], v[1]+rhs.v [1],  v[2]+rhs.v [2]);
        }

        inline Vec3& operator += (const Vec3& rhs)
        {
            v [0]  += rhs.v[0]; v [1]  += rhs.v[1];
            v [2]  += rhs.v[2]; return *this;
        }

        inline const Vec3 operator − (const Vec3& rhs) const // vector subtract
        {
            return Vec3(v[0]−rhs.v[0], v[1]−rhs.v [1],  v[2]−rhs.v [2]);
        }

        inline Vec3& operator −= (const Vec3& rhs) // vector subtract
        {
            v[0]−=rhs.v[0]; v[1]−=rhs.v[1]; v[2]−=rhs.v[2]; return *this;
        }

        inline const Vec3 operator − () const //negate
        {
            return Vec3 (−v[0], −v[1], −v[2]);
        }

        inline float length() const
        {
            return sqrtf( v[0]*v [0]  + v[1]*v [1]  + v[2]*v [2]  );
        }

        inline float normalize()
        {
            float norm = Vec3::length();
            if (norm>0.0f)
            {
                v [0]  /= norm; v[1] /= norm; v[2] /= norm;
            }
            return( norm );
        }

        inline void zero() { v [0]  = v[1]  = v[2]  = 0; }

        // to allow us to send our vector to the standard output stream cout
        friend inline ostream& operator << (ostream& output, const Vec3& vec)
        {
          output << "(" << vec.v[0] << "," << vec.v[1] << "," << vec.v[2] << ")";
          return output;
        }
};

#endif // _VEC3
```

This class has been designed to be as efficient as possible, which is very important for the real-time visualisation applications that we will be developing. This is especially important for the vector class as a single frame could involve hundreds of vector operations. To this end, we have made several design decisions:

- *Operator Overloading* - For both ease of use and efficiency we have allowed the user to apply unary and binary operations on vectors and indeed vectors with scalars, eg. $\vec{a} \cdot \vec{b}$ can be called using “a*b” and the cross product $\vec{a} \times \vec{b}$ can be called using $a\hat{}b$ (It

is not possible to treat the character 'x' as an operator in C++). The use of the ˆ for the cross-product is not an ideal choice and sometimes draws criticism when used - because of this we have included a crossProduct() method.

- *The `inline` keyword* - In C++ inline methods (identified by the `inline` keyword) are inserted by the compiler directly at the location where they are called. This provides significant performance improvement over standard methods, as there is no overhead in jumping the program counter in the compiled code. You should reserve this for short methods, otherwise your program size will be significant.

- *Why the `float` type?* - It depends on the precision of our application as to whether we should use `float` or `double` types[9]; floats will provide the level of accuracy that we will require in our systems. It is possible to use C++ templates to remove the type from the `Vec3` class developed here, to allow us to substitute doubles if required.

- *Using `const` methods* - By placing the keyword `const` after a method name, we are undertaking that this method will not modify the object, allowing us to pass a constant vector to the method.

- *Passing by `const` reference* - Passing an object by value copies the data at that memory location to create a new copy of that value, allowing us to protect the original value from changing. However, this is inefficient, especially when we are passing an object, as the constructor must be called to create the object copy. Passing by constant reference passes the address directly using the reference (like a pointer), but the `const` keyword prevents the value from being modified - providing all the benefits of pass-by-value, but with the efficiency of pass-by-reference.

- *No `virtual` methods?* - Virtual methods are useful when our solution involves inheritance and dynamic binding; however, as a design decision the `Vec3` class will not involve the creation of child classes. Virtual methods require that additional instructions are inserted and that checks are put in place at runtime, slowing performance, which would be problematic for this class. In particular, the use of the `inline` keyword would be affected by the compiler optimisation.

- *No encapsulation?* - Yes, if you completed the EE533 object-oriented module then you would know that it is generally poor practice to expose the states of the class (i.e. make them public). However, in this case the use of accessor and mutator methods would detrimentally affect performance. It is not clear that there would be any benefit either in checking the three values of the vector - as they are just floats of any value.

Here is an example of the use of this Vector.

```
0   /****************************************
     * EE563 Example Project 2
     * by: Derek Molloy
     ****************************************/

5   #include <windows.h> // Header File For Windows
    #include <gl\gl.h>    // Header File For The OpenGL32 Library
    #include <gl\glu.h>   // Header File For The GLu32 Library
    #include "vec3.h"
    #include <iostream>
10  using namespace std;
```

[9]typically `float` values are 32-bit numbers using 24-bits for the mantissa (including a sign bit) and 8-bits for the exponent; the `double` type provides 64-bit numbers that use 11-bits for the exponent and 53-bits for the mantissa (including a sign bit), allowing us to represent larger or smaller, more accurate numbers

```
int main()
{
    Vec3 a(1.0f,2.0f,3.0f), b(1.0f,2.0f,4.0f), c(1.0f,2.0f,1.0f);

    cout << "Vector a = " << a << endl;
    cout << "Vector b = " << b << endl;
    cout << "Vector c = " << c << endl;
    cout << "a has magnitude = " << a.length() << endl;
    cout << "The cross product of axb = " << (a^b) << endl;
    cout << "The dot product of a.b = " << a*b << endl;

    Vec3 d = a + b + (c * 2.0f);  // float  must be second
    cout << "Vector d = a + b + (c * 2.0f) gives: " << d << endl;
    cout << "Normalise d gives " << d.normalize() << endl;

    system("pause");
    return 0;
}
```

### 1.4.4 Matrices

Matrices are a fundamental building block of 3-D graphics and visualisation applications. They allow us to move objects around the 3-D world; scaling, rotating or translating them. Linear Algebra describes the type of mathematics used in the manipulation of matrices. We can write the linear simultaneous equations:

$$\begin{aligned} v_1 &= 6u_1 + 1v_2 \\ v_2 &= 4u_1 + 8v_2 \end{aligned} \tag{1.41}$$

As $\mathbf{v} = A\mathbf{u}$, or:

$$\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} 6 & 1 \\ 4 & 8 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{1.42}$$

Where $\mathbf{u}$ and $\mathbf{v}$ are tuples, represented by a $2 \times 1$ matrix.

So, a matrix is a 2-D array of scalars arranged into rows and columns. A vector is a one-dimensional array of scalars. A square matrix is one with the same number of rows and columns - we are most interested in 2x2, 3x3 and 4x4 matrices. We are also interested in the column $n \times 1$ matrix or row $1 \times n$ matrix as we can write a vector in this form; for example, $\vec{v} = (v_x, v_y, v_z)$ can be represented as a row vector (written horizontally) as $V = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix}$ or as a column vector (written vertically) as:

$$\begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

The transpose of a matrix $M$ with dimensions $r \times c$ is denoted as $M^T$ is the $c \times r$ matrix, where the matrix is flipped diagonally. It is easier to represent a column vector in text as the transpose of a row vector, i.e. $\begin{bmatrix} v_x & v_y & v_z \end{bmatrix}^T$.

A diagonal matrix is one in which all non-diagonal elements in the matrix are zero. For example, here is a diagonal matrix and a special diagonal matrix, known as the identity matrix:

$$\begin{bmatrix} 21 & 0 & 0 \\ 0 & -45 & 0 \\ 0 & 0 & 1.5 \end{bmatrix} \qquad I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The identity matrix produces no transformation effect, so $AI = A$.

### Multiplication

To multiply a matrix by a scalar we use:

$$kA = k\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} ka_{11} & ka_{12} & ka_{13} \\ ka_{21} & ka_{22} & ka_{23} \\ ka_{31} & ka_{32} & ka_{33} \end{bmatrix}$$

To multiply two matrices $A$, with dimensions $m_1 \times n_1$ and $B$, with dimensions $m_2 \times n_2$; we can, provided that $n_1 = m_2$. This would result in a matrix with dimensions $m_1 \times n_2$, such that:

$$\underbrace{C}_{m_1 \times n_2} = \underbrace{A}_{m_1 \times x} \underbrace{B}_{x \times n_2} \tag{1.43}$$

where $x = n_1 = m_2$.

To multiply two matrices where the number of columns in $A$ matches the number of rows in $B$ then $AB$ is as follows for a $3 \times 3$ matrix (otherwise it is undefined):

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix} =$$

$$\begin{bmatrix} a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} & a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} & a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} \\ a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} & a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} & a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \\ a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} & a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} & a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} \end{bmatrix}$$

### Transpose

The transpose of a matrix $A$ is denoted as $A^T$, so if:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \quad A^T = \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ a_{13} & a_{23} & a_{33} \end{bmatrix} \tag{1.44}$$

### Inverse

To find the inverse of a matrix $A$, denoted as $A^{-1}$ and is given by:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad A^{-1} = \frac{1}{|A|}\begin{bmatrix} a_{22} & -a_{21} \\ -a_{12} & a_{11} \end{bmatrix} \tag{1.45}$$

Where $|A| = a_{11}a_{22} - a_{12}a_{21}$.

## 1.4.5 A Matrix Class

Good 3-D engines use 4x4 matrices. That form is convenient for the translations and rotations because it allows transforming points with respect to a centre point. However, it is possible to use 3x3 matrices, but all the transformations are performed with assumption that object's origin is (0,0,0). It is impossible to perform translations when using 3x3 matrices.

```
0   // Matrix.h: interface for the Matrix class.
    //
    //////////////////////////////////////////////////////////////////////

    #if !defined MATRIX_H
5   #define MATRIX_H

    #include "Vec3.h"
    #include "Vec4.h"

10  class Quat; //must define Quat as Matrix uses Quat and Quat uses Matrix

    class Matrix
```

```
{
        protected:

        float mat[4][4];            //matrix state - using a 4x4 storage element

        public:
        Matrix();
        Matrix( const Matrix& other);

                explicit Matrix( float const * const def );

                Matrix( float a00, float a01, float a02, float a03,
                                float a10, float a11, float a12, float a13,
                                float a20, float a21, float a22, float a23,
                                float a30, float a31, float a32, float a33);

        ~Matrix() {}

        int compare(const Matrix& m) const { return memcmp(mat,m.mat,sizeof(mat)); }

        bool operator < (const Matrix& m) const { return compare(m)< 0; }
        bool operator == (const Matrix& m) const { return compare(m)==0; }
        bool operator != (const Matrix& m) const { return compare(m)!=0; }

        inline float& operator()(int row, int col) { return mat[row][col]; }
        inline float  operator()(int row, int col) const { return mat[row][col]; }

        inline bool valid() const { return !isNaN(); }
        inline bool isNaN() const { return _isnan(mat[0][0]) ||
                _isnan(mat[0][1])  ||  _isnan(mat[0][2])  ||  _isnan(mat[0][3])  ||
                _isnan(mat[1][0])  ||  _isnan(mat[1][1])  ||  _isnan(mat[1][2])  ||
                _isnan(mat[1][3])  ||  _isnan(mat[2][0])  ||  _isnan(mat[2][1])  ||
                _isnan(mat[2][2])  ||  _isnan(mat[2][3])  ||  _isnan(mat[3][0])  ||
                _isnan(mat[3][1])  ||  _isnan(mat[3][2])  ||  _isnan(mat[3][3]);  }

        inline Matrix& operator = (const Matrix& other)
        {
            if( &other == this ) return *this;
                        std::copy((float*)other.mat,(float*)other.mat+16,(float*)(mat));
            return *this;
        }

        inline void set(const Matrix& other)
        {
            std::copy((float*)other.mat,(float*)other.mat+16,(float*)(mat));
        }

        inline void set(float const * const ptr)
        {
            std::copy(ptr,ptr+16,(float*)(mat));
        }

        void set( float a00, float a01, float a02, float a03,
                  float a10, float a11, float a12, float a13,
                  float a20, float a21, float a22, float a23,
                  float a30, float a31, float a32, float a33);

        float * ptr() { return (float *)mat; }
        float * ptr() const { return (float *)mat; }

        void makeIdentity();

        void makeScale( const Vec3& );
        void makeScale( float, float, float );

        void makeTranslate( const Vec3& );
        void makeTranslate( float, float, float );

        void makeRotate( const Vec3& from, const Vec3& to );
        void makeRotate( float angle, const Vec3& axis );
        void makeRotate( float angle, float x, float y, float z );
                void makeRotate( const Quat& q);
        void makeRotate( float angle1, const Vec3& axis1,
                         float angle2, const Vec3& axis2,
                         float angle3, const Vec3& axis3);

        // Set to a orthographic projection. See glOrtho for further details.
        void makeOrtho(double left, double right,
                       double bottom, double top,
                       double zNear, double zFar);

        // Set to a 2D orthographic projection. See glOrtho2D for further details.
        inline void makeOrtho2D(double left, double right, double bottom, double top)
        {
            makeOrtho(left,right,bottom,top,-1.0,1.0);
        }

        // Set to a perspective projection. See glFrustum for further details.
        void makeFrustum(double left, double right,
                         double bottom, double top,
                         double zNear, double zFar);

        // Set to a symmetrical perspective projection, See gluPerspective for
        // further details. Aspect ratio is defined as width/height.
        void makePerspective(double fovy,double aspectRatio,
                             double zNear, double zFar);

        // Set to the position and orientation as per a camera, using the same
        //  convention as gluLookAt.
        void makeLookAt(const Vec3& eye,const Vec3& center,const Vec3& up);


        bool invert( const Matrix& );
```

```
        //basic utility functions to create new matrices
                inline static Matrix identity( void );
        inline static Matrix scale( const Vec3& sv);
        inline static Matrix scale( float sx, float sy, float sz);
        inline static Matrix translate( const Vec3& dv);
        inline static Matrix translate( float x, float y, float z);
        inline static Matrix rotate( const Vec3& from, const Vec3& to);
        inline static Matrix rotate( float angle, float x, float y, float z);
        inline static Matrix rotate( float angle, const Vec3& axis);
        inline static Matrix rotate( float angle1, const Vec3& axis1,
                                     float angle2, const Vec3& axis2,
                                     float angle3, const Vec3& axis3);
                inline static Matrix rotate( const Quat& quat);
        inline static Matrix inverse( const Matrix& matrix);

        // Create a orthographic projection. See glOrtho for further details.
        inline static Matrix ortho(double left, double right,
                                   double bottom, double top,
                                   double zNear, double zFar);

        // Create a 2D orthographic projection. See glOrtho for further details.
        inline static Matrix ortho2D(double left, double right,
                                     double bottom, double top);

        // Create a perspective projection. See glFrustum for further details.
        inline static Matrix frustum(double left, double right,
                                     double bottom, double top,
                                     double zNear, double zFar);

        // Create a symmetrical perspective projection, See gluPerspective.
        // Aspect ratio is defined as width/height.
        inline static Matrix perspective(double fovy,double aspectRatio,
                                         double zNear, double zFar);

        // Create the position and orientation as per a camera, using the
        //  same convention as gluLookAt.
        inline static Matrix lookAt(const Vec3& eye,const Vec3& center,const Vec3& up);

        inline Vec3 preMult( const Vec3& v ) const;
        inline Vec3 postMult( const Vec3& v ) const;
        inline Vec3 operator* ( const Vec3& v ) const;
        inline Vec4 preMult( const Vec4& v ) const;
        inline Vec4 postMult( const Vec4& v ) const;
        inline Vec4 operator* ( const Vec4& v ) const;
        void setTrans( float tx, float ty, float tz );
                void setTrans( const Vec3& v );
        inline Vec3 getTrans() const { return Vec3(mat[3][0],mat[3][1],mat[3][2]); }
        inline Vec3 getScale() const { return Vec3(mat[0][0],mat[1][1],mat[2][2]); }

        // apply apply an 3x3 transform of v*M[0..2,0..2]
        inline static Vec3 transform3x3(const Vec3& v,const Matrix& m);

        // apply apply an 3x3 transform of M[0..2,0..2]*v
        inline static Vec3 transform3x3(const Matrix& m,const Vec3& v);

        // basic Matrix multiplication, our workhorse methods.
        void mult( const Matrix&, const Matrix& );
        void preMult( const Matrix& );
        void postMult( const Matrix& );

        inline void operator *= ( const Matrix& other )
        {    if( this == &other ) {
                Matrix temp(other);
                postMult( temp );
            }
            else postMult( other );
        }

        inline Matrix operator * ( const Matrix &m ) const
                {
                        Matrix r;
            r.mult(*this,m);
                        return r;
                }
};


//static inline utility methods
inline Matrix Matrix::identity(void)
{
    Matrix m;
    m.makeIdentity();
    return m;
}

inline Matrix Matrix::scale(float sx, float sy, float sz)
{
    Matrix m;
    m.makeScale(sx,sy,sz);
    return m;
}

inline Matrix Matrix::scale(const Vec3& v )
{
    return scale(v.x(), v.y(), v.z() );
}

inline Matrix Matrix::translate(float tx, float ty, float tz)
{
    Matrix m;
    m.makeTranslate(tx,ty,tz);
    return m;
```

```
}

inline Matrix Matrix::translate(const Vec3& v )
{
    return translate(v.x(), v.y(), v.z() );
}


inline Matrix Matrix::rotate( const Quat& q )
{
    Matrix m;
    m.makeRotate( q );
    return m;
}

inline Matrix Matrix::rotate(float angle, float x, float y, float z )
{
    Matrix m;
    m.makeRotate(angle,x,y,z);
    return m;
}

inline Matrix Matrix::rotate(float angle, const Vec3& axis )
{
    Matrix m;
    m.makeRotate(angle,axis);
    return m;
}

inline Matrix Matrix::rotate( float angle1, const Vec3& axis1,
                              float angle2, const Vec3& axis2,
                              float angle3, const Vec3& axis3)
{
    Matrix m;
    m.makeRotate(angle1,axis1,angle2,axis2,angle3,axis3);
    return m;
}
inline Matrix Matrix::rotate(const Vec3& from, const Vec3& to )
{
    Matrix m;
    m.makeRotate(from,to);
    return m;
}

inline Matrix Matrix::inverse( const Matrix& matrix)
{
    Matrix m;
    m.invert(matrix);
    return m;
}

inline Matrix Matrix::ortho(double left, double right,
                            double bottom, double top,
                            double zNear, double zFar)
{
    Matrix m;
    m.makeOrtho(left,right,bottom,top,zNear,zFar);
    return m;
}

inline Matrix Matrix::ortho2D(double left, double right,
                              double bottom, double top)
{
    Matrix m;
    m.makeOrtho2D(left,right,bottom,top);
    return m;
}

inline Matrix Matrix::frustum(double left, double right,
                              double bottom, double top,
                              double zNear, double zFar)
{
    Matrix m;
    m.makeFrustum(left,right,bottom,top,zNear,zFar);
    return m;
}

inline Matrix Matrix::perspective(double fovy,double aspectRatio,
                                  double zNear, double zFar)
{
    Matrix m;
    m.makePerspective(fovy,aspectRatio,zNear,zFar);
    return m;
}

inline Matrix Matrix::lookAt(const Vec3& eye,const Vec3& center,const Vec3& up)
{
    Matrix m;
    m.makeLookAt(eye,center,up);
    return m;
}


inline Vec3 Matrix::postMult( const Vec3& v ) const
{
    float d = 1.0f/(mat[3][0]*v.x()+mat[3][1]*v.y()+mat[3][2]*v.z()+mat[3][3]) ;
    return Vec3( (mat[0][0]*v.x() + mat[0][1]*v.y() + mat[0][2]*v.z() + mat[0][3])*d,
        (mat[1][0]*v.x() + mat[1][1]*v.y() + mat[1][2]*v.z() + mat[1][3])*d,
        (mat[2][0]*v.x() + mat[2][1]*v.y() + mat[2][2]*v.z() + mat[2][3])*d) ;
}

inline Vec3 Matrix::preMult( const Vec3& v ) const
{
```

```
    float d = 1.0f/(mat[0][3]*v.x()+mat[1][3]*v.y()+mat[2][3]*v.z()+mat[3][3]) ;
    return Vec3( (mat[0][0]*v.x() + mat[1][0]*v.y() + mat[2][0]*v.z() + mat[3][0])*d,
        (mat[0][1]*v.x() + mat[1][1]*v.y() + mat[2][1]*v.z() + mat[3][1])*d,
        (mat[0][2]*v.x() + mat[1][2]*v.y() + mat[2][2]*v.z() + mat[3][2])*d);
}

inline Vec4 Matrix::postMult( const Vec4& v ) const
{
    return Vec4( (mat[0][0]*v.x() + mat[0][1]*v.y() + mat[0][2]*v.z() + mat[0][3]*v.w()),
        (mat[1][0]*v.x() + mat[1][1]*v.y() + mat[1][2]*v.z() + mat[1][3]*v.w()),
        (mat[2][0]*v.x() + mat[2][1]*v.y() + mat[2][2]*v.z() + mat[2][3]*v.w()),
        (mat[3][0]*v.x() + mat[3][1]*v.y() + mat[3][2]*v.z() + mat[3][3]*v.w())) ;
}

inline Vec4 Matrix::preMult( const Vec4& v ) const
{
    return Vec4( (mat[0][0]*v.x() + mat[1][0]*v.y() + mat[2][0]*v.z() + mat[3][0]*v.w()),
        (mat[0][1]*v.x() + mat[1][1]*v.y() + mat[2][1]*v.z() + mat[3][1]*v.w()),
        (mat[0][2]*v.x() + mat[1][2]*v.y() + mat[2][2]*v.z() + mat[3][2]*v.w()),
        (mat[0][3]*v.x() + mat[1][3]*v.y() + mat[2][3]*v.z() + mat[3][3]*v.w()));
}
inline Vec3 Matrix::transform3x3(const Vec3& v,const Matrix& m)
{
    return Vec3( (m.mat[0][0]*v.x() + m.mat[1][0]*v.y() + m.mat[2][0]*v.z()),
                 (m.mat[0][1]*v.x() + m.mat[1][1]*v.y() + m.mat[2][1]*v.z()),
                 (m.mat[0][2]*v.x() + m.mat[1][2]*v.y() + m.mat[2][2]*v.z()));
}

inline Vec3 Matrix::transform3x3(const Matrix& m,const Vec3& v)
{
    return Vec3( (m.mat[0][0]*v.x() + m.mat[0][1]*v.y() + m.mat[0][2]*v.z()),
                 (m.mat[1][0]*v.x() + m.mat[1][1]*v.y() + m.mat[1][2]*v.z()),
                 (m.mat[2][0]*v.x() + m.mat[2][1]*v.y() + m.mat[2][2]*v.z()) ) ;
}


inline Vec3 operator* (const Vec3& v, const Matrix& m )
{
        return m.preMult(v);
}
inline Vec4 operator* (const Vec4& v, const Matrix& m )
{
        return m.preMult(v);
}

inline Vec3 Matrix::operator* (const Vec3& v) const
{
        return postMult(v);
}
inline Vec4 Matrix::operator* (const Vec4& v) const
{
        return postMult(v);
}

inline std::ostream& operator<< (std::ostream& os, const Matrix& m )
{
    os << "{"<<std::endl;
    for(int row=0; row<4; ++row) {
        os << "\t";
        for(int col=0; col<4; ++col)
            os << m(row,col) << " ";
        os << std::endl;
    }
    os << "}" << std::endl;
    return os;
}

#endif // MATRIX_H
```

The `explicit` keyword allows us prevent a constructor from acting as an implicit conversion operatior. For example if we had a class A that had the form:

```
class A{
 ...
   public:
                C(int x); // this constructor will allow implicit conversions
}
```

We can use it to construct an object by typing `A a(1);` or we could write `a = 1;` in an implicit form. There are cases where this implicit conversion does not work correctly and we can prevent the assignment `a = 1;` by adding in the explicit keyword; so, writing for example: `explicit C(int x);` in the class `A` above.

An example of the use of this class is in:

```
/******************************************
 * EE563 Example Project 3
 * by: Derek Molloy
 ******************************************/

#include <windows.h> // Header File For Windows
#include <gl\gl.h>       // Header File For The OpenGL32 Library
#include <gl\glu.h>      // Header File For The GLu32 Library
#include "vec3.h"
#include "Matrix.h"
#include "vec4.h"
#include "Quat.h"
#include <iostream>
```

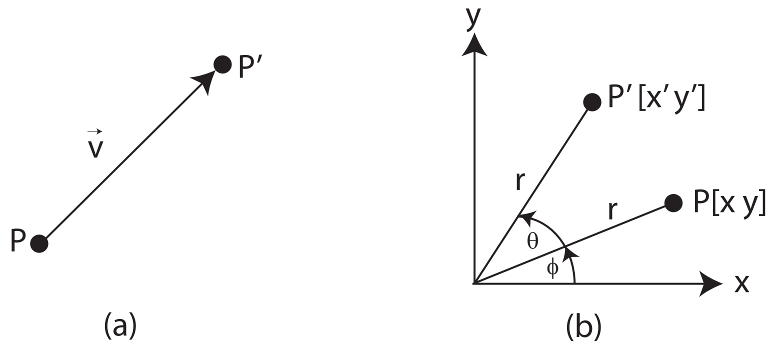



Figure 1.23: (a) illustrates the transformation of a point $P$ to a point $P'$ by a vector $\vec{v}$.

```
    using namespace std;

15  int main()
    {
        Matrix m(11, 12, 13, 14, 21, 22, 23, 24, 31, 32, 33, 34, 41, 42, 43, 44);
        cout << "Matrix M: " << m << endl;

20      Vec4 e (1,2,3,4);
        cout << "Vector e: " << e << endl;
        Vec4 f = m * e;
        cout << "Vector f = Me: " << f << endl;

25      Matrix r;
        r.makeScale(0.5, 0.5, 0.5);
        cout << "Matrix R (scaling matrix): " << r << endl;
        Matrix n = m*r;
        cout << "Matrix N = MR:  " << n << endl;
30
        system("pause");
        return 0;
    }
```

## 1.5 Transformations

### 1.5.1 Translation

At this stage you will have examined transformations in Java3D. We will cover some of that material again here, just to provide the basis for transformations in OpenGL. 1.23(a) illustrates the transformation of a point $P = \begin{bmatrix} x & y & z & 1 \end{bmatrix}^T$ to a new location $P' = \begin{bmatrix} x' & y' & z' & 1 \end{bmatrix}^T$ by a vector $\vec{v} = \begin{bmatrix} v_x & v_y & v_z & 0 \end{bmatrix}^T$ when using homogeneous coordinate representation. We can write $P' = P + \vec{v}$ (where $x' = x + v_x$, $y' = y + v_y$ and $z' = z + v_z$), or in matrix form $P' = TP$, where:

$$T = \begin{bmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.46}$$

This matrix notation is much better as it can represent all affine transformations and even allow multiple transformations at the same time (e.g. rotation and translation).

See Example3a for a numerical example. It demonstrates that $P = \begin{bmatrix} 1 & 2 & 3 & 1 \end{bmatrix}^T$ (i.e. a point $P = (1, 2, 3)$) translated by a homogeneous vector $\vec{v} = \begin{bmatrix} 3 & 4 & 5 & 0 \end{bmatrix}^T$ can

be written as $P' = TP$ where:

$$T = \begin{bmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 4 \\ 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.47}$$

and when applied will result in a point $P' = MP = \begin{bmatrix} 4 & 6 & 8 & 1 \end{bmatrix}^T$ Using the 4x4 matrix class that we just created, the code for this example is:

```
Matrix m(1, 0, 0, 3,
   0, 1, 0, 4,
   0, 0, 1, 5,
   0, 0, 0, 1);
cout << Matrix M:  << m << endl;

Vec4 p (1,2,3,1);
cout << "Point␣P:␣␣<<␣p␣<<␣endl;
Vec4␣pp␣=␣m␣*␣p;
cout␣<<␣"Point P\ = MP:  << pp << endl;
```

### 1.5.2 Rotation

We may also wish to apply a rotation to our point (we will consider 2-D rotation for simplicity). Figure 1.23 illustrates the rotation of a point $P$ about the origin by $\theta$ degrees. We can describe this as $x = r\cos\phi$, $y = r\sin\phi$ and:

$$x' = r\cos(\phi + \theta) \tag{1.48}$$
$$y' = r\sin(\phi + \theta) \tag{1.49}$$

Now, the standard cosine identity rule allows us to write:

$$x' = r\cos\phi\cos\theta - r\sin\phi\sin\theta \tag{1.50}$$
$$y' = r\sin\phi\cos\theta + r\cos\phi\sin\theta \tag{1.51}$$

Since $x = r\cos\phi$ and $y = r\sin\phi$:

$$r = x/\cos\phi \tag{1.52}$$
$$r = y/\sin\phi \tag{1.53}$$

Substituting into above to remove $\phi$, we get:

$$x' = x\cos\theta - y\sin\theta \tag{1.54}$$
$$y' = x\cos\theta + x\sin\theta \tag{1.55}$$

So, if we take it that rotating about the $z$-axis leaves the $z$ value unchanged, we can write:

$$x' = x\cos\theta - y\sin\theta \tag{1.56}$$
$$y' = x\sin\theta + y\cos\theta \tag{1.57}$$
$$z' = z \tag{1.58}$$

We can then write this in homogeneous co-ordinates as $P' = R_z(\theta)P$, where:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.59)$$

We can apply the same technique as for rotation about the $z$-axis to derive rotation matrices for rotations about the $x$-axis (where $x$ is unchanged) and $y$-axes(where $y$ is unchanged), giving:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.60)$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.61)$$

### 1.5.3 Scaling

We can also apply these transformations to the scaling of objects, allowing the expansion or contraction of the object along each axis. We can write this as:

$$x' = s_x x \quad (1.62)$$
$$y' = s_y y \quad (1.63)$$
$$z' = s_z z \quad (1.64)$$

We can rewrite this as $P' = SP$ where:

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.65)$$

### 1.5.4 Shearing

We will look at one last transformation, a shear operation, which will result in the distortion of the object as illustrated in 1.24(a). This can be described mathematically as in figure 1.24(b) and can be described as:

$$\begin{aligned} x' &= x + y\cot\theta \\ y' &= y \\ z' &= z \end{aligned}$$

This can be represented in homogeneous form as:

$$H(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.66)$$

Which can be extended to shearing operations along other axes. Figure 1.24 demonstrates the execution of various affine transforms on an object in 3-D space.

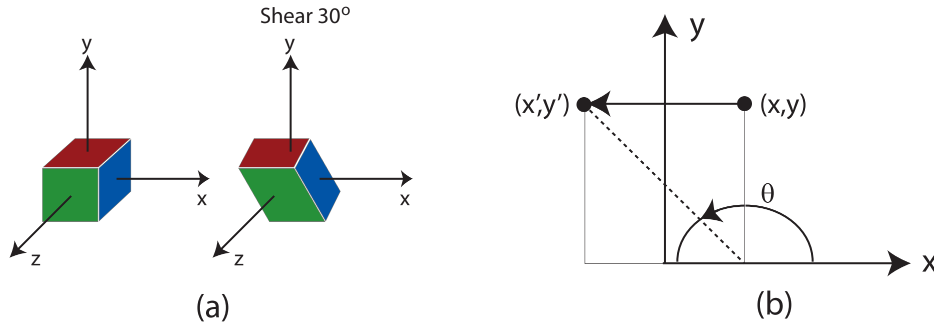



Figure 1.24: (a) illustrates the shearing of an object along the $x$-axis, and (b) shows the mathematical representation of this shearing.

### 1.5.5 Inverse Operations

We can compute inverse matrices by general formula, but we can also provide simple geometric observations, such as:

- Translation: $T^{-1}(v_x, v_y, v_z) = T(-v_x, -v_y, -v_z)$
- Rotation: $R^{-1}(\theta) = R(-\theta)$ where $\cos(-\theta) = \cos(\theta)$ and $\sin(-\theta) = -\sin(\theta)$
- Scaling: $S^{-1}(s_x, s_y, s_z) = S(\frac{1}{s_x}, \frac{1}{s_y}, \frac{1}{s_z})$

### 1.5.6 Combination and Other Operations

We can create an arbitrary affine transformation by combining rotation, translation and scaling matrices by multiplication. The same transformation can then be applied to many vertices. This is quite efficient, as the computational cost of combining the matrices is insignificant when it is then applied to hundreds of vertices in a complex object. If we have a vector $\vec{v'} = TSR\vec{v}$, where we wish to apply a translation, rotation and scaling, then the order by which these operations will be carried out in this case is: $\vec{v'} = T(S(R\vec{v}))$.

We can carry out about a rotation about an arbitrary axis $\vec{v}$ (as in figure 1.25(a)), by decomposing the rotation into a combination of rotations about the $x$, $y$ and $z$-axes in the form: $R(\theta) = R_x(\theta_x)R_y(\theta_y)R_z(\theta_z)$. One important note is that these rotations do not commute, i.e. if we apply the rotations in different orders it will result in a different location; we can achieve the same location by using different angles. If we wish to apply a rotation about a point P other than the origin then we must carry out the following steps:

1. identify our perceived rotation point $P$
2. translate our object to the origin
3. rotate the object about the origin
4. translate the object back to the point $P$

This matrix will have the form $M = T(P)R(\theta)T(-P)$

### 1.5.7 The OpenGL Current Transformation Matrix (CTM)

The Current Transformation Matrix (CTM) defines a 4x4 homogeneous co-ordinate matrix that is part of the current OpenGL state (loaded into the transformation unit) and is

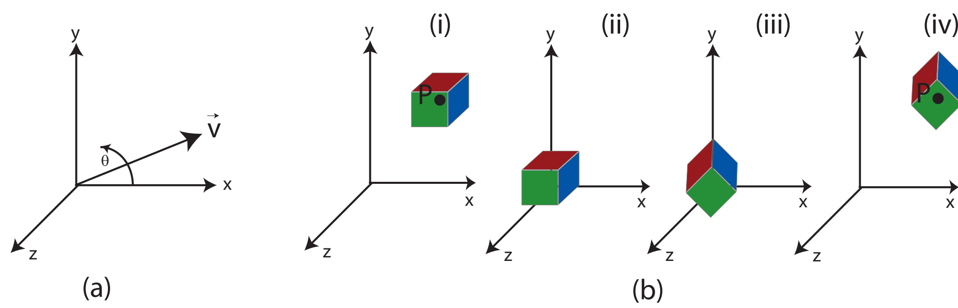



Figure 1.25: (a) illustrates the rotation of a vector about the $x$, $y$ and $z$-axes (b) illustrates the steps involved in the rotation of an object about a point $P$.

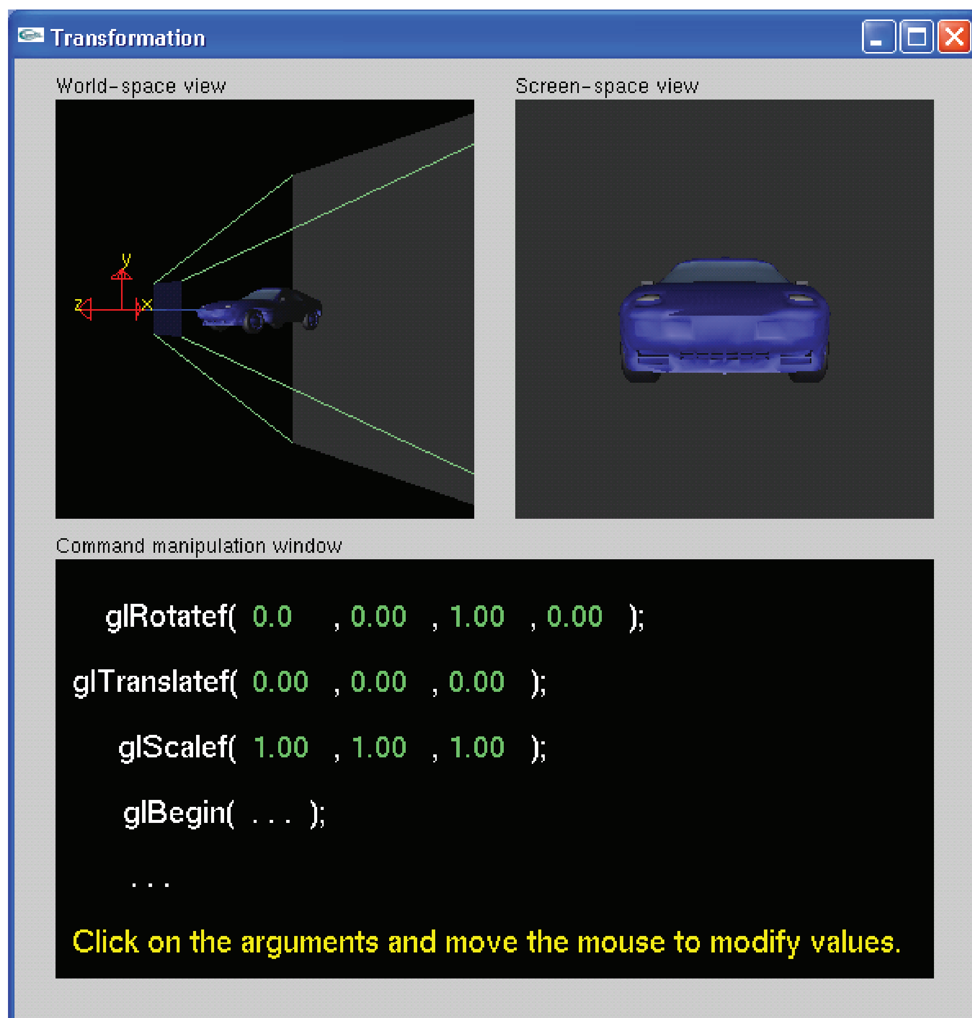



Figure 1.26: The Nate Robins Tutorial Example on Transformation.

applied to all vertices that are passed down the OpenGL pipeline. Where vertices $v$ are passed into the transformation unit, where they are operated on by a matrix $C$ to provide the output $p' = Cp$.

We can change the CTM by loading a new matrix, or by post-multiplication by another matrix. We can do some of the following, basic loads:

- Load an Identity matrix $I \mapsto C$
- Load a new matrix $M \mapsto C$

Or, we could load a particular matrix, such as:

- Load a translation matrix $T \mapsto C$
- Load a scaling matrix $S \mapsto C$
- Load a rotation matrix $R \mapsto C$

Or, we could post-multiply the current $C$, such as:

- Post-multiply by some arbitrary matrix $CM \mapsto C$
- Post-multiply by some translation matrix $CT \mapsto C$
- Post-multiply by some scaling matrix $CS \mapsto C$
- Post-multiply by some rotation matrix $CR \mapsto C$

Remembering back to figure 1.25(b), where we wish to rotate about a fixed point $P$; if we wish to apply this in OpenGL we would:

- Start with an identity matrix $I \mapsto C$
- Move the fixed point $P$ to the origin $CT \mapsto C$
- Rotate the object $CR \mapsto C$
- Move the fixed point $P$ back $CT^{-1} \mapsto C$

If we were to apply these as post-multiplications, then this will result in an operation of $C = TRT^{-1}$ which would be applied in the wrong order! (backwards). To do this correctly we have to perform the operations in the order $C = T^{-1}RT$ (i.e. $I \mapsto C$, $CT^{-1} \mapsto C$, $CR \mapsto C$, and then $CT \mapsto C$). Each one of these operations corresponds to an OpenGL function call, remembering the the last operation specified is the first executed in the program.

OpenGL has a pipeline as illustrated in figure 1.27 where the model-view (`GL_MODELVIEW`) and projection (`GL_PROJECTION`) matrix in the pipeline is concatenated to create the CTM. Again, we use the `glMatrixMode()` function to state which part of the CTM we are changing. If we use the same example of rotating an object about a fixed point, we could pick a numerical example of rotating 45 degrees about the $y$-axis at a fixed point $P = (1.5, 2.5, 3.5)$. You will remember that mathematically we defined this (intuitively backwards) as: $C = T^{-1}RT$. Therefore, the correct OpenGL code for this operation is:

```
0 glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef (1.5 f , 2.5 f , 3.5 f );
  glRotatef(0.0 f , 45.0f , 0.0 f , 1.0 f );
  glTranslatef(−1.5f, −2.5f, −3.5f);
```

Where the last operation is the first one applied. We could also use OpenGLs ability to load an arbitrary matrix to do the same Example3b

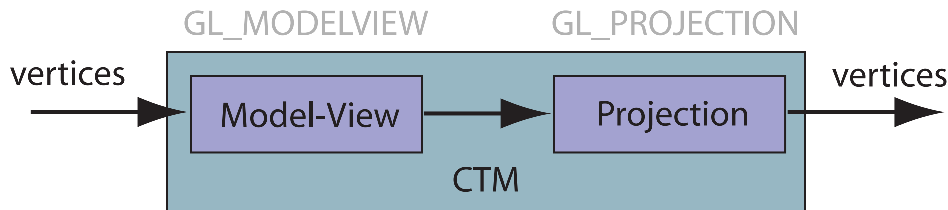



Figure 1.27: An illustration of the OpenGL model-view and projection in the pipeline.

```
Matrix C: {
  0.71  0.00  -0.71 0.0
  0.00  1.00   0.00 0.0
  0.71  0.00   0.71 0.0
 -2.74  0.00  -7.21 1.0
}
```

One additional useful feature is that we can query the CTM in OpenGL to find out what the current CTM matrix values are by using the query functions `glGetIntegerv()`, `glGetFloatv()`, `glGetDoublev()`, `glGetBooleanv()` and `glIsEnabled()`. So, we could write:

```
float matrix_arr[] = new float[16];
glGetFloatv(GL_MODELVIEW_MATRIX, matrix_arr);
Matrix m(matrix_arr);
```

These calls allow us to query the hundreds of states of the current OpenGL state machine; in this case we are using `GL_MODELVIEW_MATRIX` as our parameter name, but there are many other states that can be queried.

```
void glGetBooleanv(GLenum pname, GLboolean *params);
void glGetDoublev(GLenum pname, GLdouble *params);
void glGetFloatv(GLenum pname, GLfloat *params);
void glGetIntegerv(GLenum pname, GLint *params);
```

See: `http://www.xfree86.org/current/glGet.3.html` for a full list of the states. Example3c demonstrates the display of the `GL_MODELVIEW_MATRIX` state, using the code:

```
case VK_F1: //F1 key pressed
  float matrix_arr[16];
  glGetFloatv(GL_MODELVIEW_MATRIX, matrix_arr);
  Matrix m(matrix_arr);
  std::string s = m.toString();
  MessageBox (NULL,s.data(),GL_MODELVIEW Matrix,MB_OK|MB_ICONINFORMATION);
  return 0;
```

So, by pressing the key 'F1' key in Example3c the model-view matrix will be displayed in a window message box.

This segment of code from Example3c has a boolean switch `isRawMatrixMode` which allows us to switch from a manual calculation of the transformation matrix to the use of the standard OpenGL functions for rotation and translation. It is important to note here that the order of the matrix multiplication $C = T^{-1}RT$ is in the reverse order to the calls to the functions `glTranslatef()`.

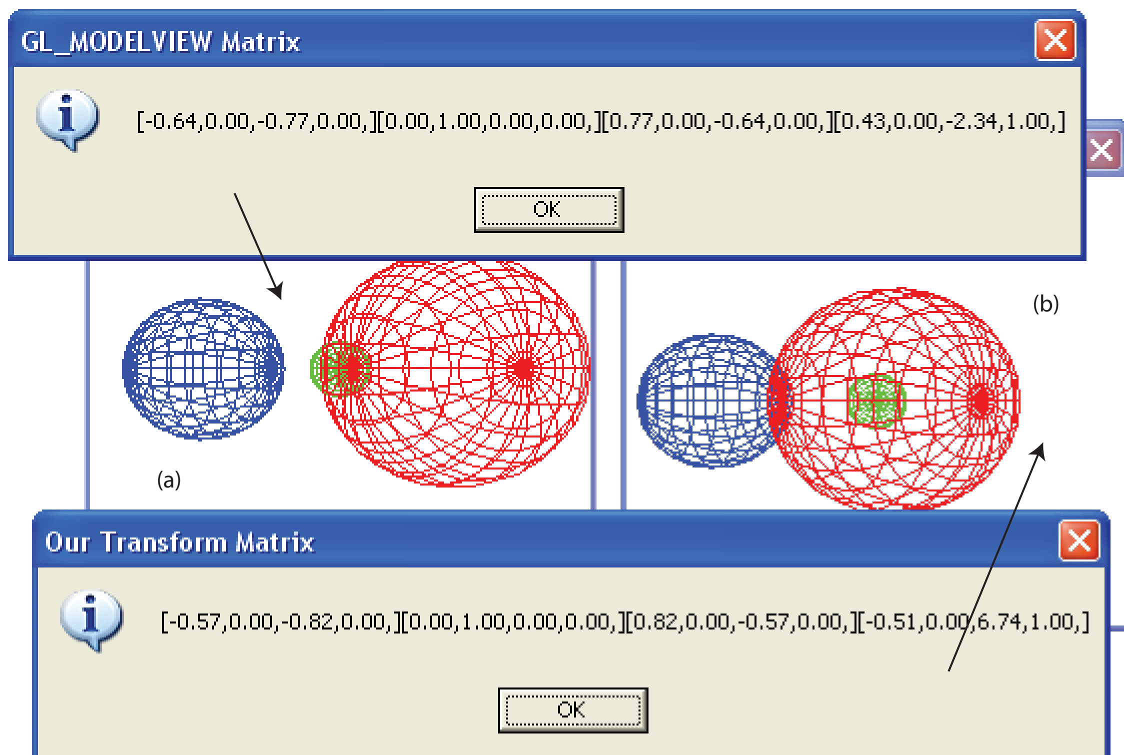



Figure 1.28: Screen captures of Example 3c, displaying the CTM Matrix and the matrix transform that we have calculated manually

```
// Please note that these matrices are in transpose form
if (isRawMatrixMode) // Allow us to switch between the following sets of code
{
    Matrix Tti(1.0f , 0, 0, 0,
      0, 1.0 f , 0, 0,
      0, 0, 1.0 f , 0,
      −1.5f, −2.5f, −3.5f, 1.0 f );
    float thetaRads = DegreesToRadians(theta);
    Matrix Rt ( cos(thetaRads), 0, −sin(thetaRads), 0,
      0, 1.0 f , 0, 0,
      sin(thetaRads), 0, cos(thetaRads), 0,
      0, 0, 0, 1.0 f );
    Matrix Tt (1.0f, 0, 0, 0,
      0, 1.0 f , 0, 0,
      0, 0, 1.0 f , 0,
      1.5 f , 2.5 f , 3.5 f , 1.0f );
    C = Tti ∗ Rt ∗ Tt;
    glMultMatrixf(C.ptr());
}
else
{
    glTranslatef (1.5 f , 2.5 f , 3.5 f );
    glRotatef (theta, 0.0f , 1.0 f , 0.0 f );
    glTranslatef(−1.5f, −2.5f, −3.5f);
}
```

### 1.5.8 Coordinate Spaces in the Graphics Pipeline

#### Local coordinate system (Object space)

When we create models it is easiest if we have a local coordinate system centered on or close to the geometry of the model. If for example we created a sphere, it would probably make the most sense to have the sphere centered on the origin of our coordinate system. For example, when we think of an airplane we think of a coordinate system local to the airplane, with a $z$-axis representing our height above the ground and an $xy$ plane aligned parallel to the Earth's surface. We can then consider rotations around the $x$ or $y$ axis as describing the pitch and roll of the airplane. Object space can be split into many subspaces at many levels, depending on the complexity of the object; for example, a car may have nested subspaces for the individual wheels etc.

#### World coordinate system (World space)

Many models with individual local coordinate systems can be placed in a single scene. We place the models in the world coordinate system by transforming their locations from the local coordinate system, taking account of any spatial relationships between the various models. As well as models, other graphics objects such as lights, cameras and special effects may also be added to the World coordinate system. To continue the airplane analogy, the world coordinate system could be defined as spherical in nature with the centre of the coordinate system at the centre of the earth (using a polar coordinate system); or more possibly it could be considered to be a flat surface[10], going to infinity in all directions, with the centre of the coordinate system at the "centre" of an $xy$ plane, with $z$ representing height into the sky (a Cartesian coordinate system). The OpenGL API doesn't really have a World space.

#### View space (Camera space)

The View space or Camera space is the virtual camera view of the World space. In camera space, the camera is at the origin with the $+x$ pointing to the right, $+y$ pointing up and the $+z$ pointing forward in the direction that the camera is facing (this can differ with implementation, sometimes pointing the opposite direction). View space has several properties, such as a viewing point, a viewing direction and a view volume. The viewing point defines the location of the observer in the world space; the viewing direction describes the orientation of the viewer; and the view volume is the pyramid of space that is in front of the camera in the viewing direction - also called the view frustum. Figure 1.29 illustrates several different representations of the view space. Lighting, culling and back-face culling can be performed in view space.

If we describe a view as being the combination of a view coordinate system the minimal system required is illustrated in Figure 1.29 (a-d), where the view coordinate system, $UVN$, has $N$ coincident with the viewing direction and $U$, $V$ in a plane parallel to the view plane. The virtual camera can be placed at any location $C$ and can be pointed in any direction using the viewing direction $\vec{n}$. To transform points in world coordinate space to points in view space we have to apply a change of coordinate systems, using a translation and rotation. Therefore, to get the view point equivalent of a point in world space we can

[10] In the good old days when the world was assumed to be flat, OpenGL programming was mathematically much more straightforward; this was particularly the case when developing flight simulators.

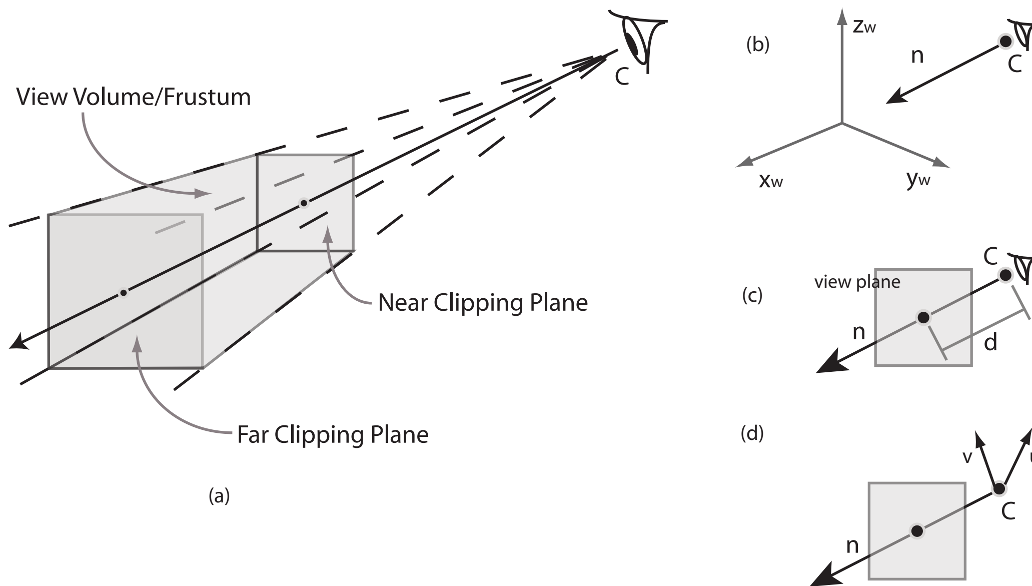



Figure 1.29: Several different representations of the View plane illustrating (a) the view frustum with the near and far clipping planes formed by the view point $C$ and the view plane window; (b) the view point $C$ and viewing direction $\vec{n}$; (c) the view plane normal to the view direction $\vec{n}$ a distance $d$ from $C$; and, (d) the View point $C$ and $UV$ axes parallel to the view plane.

use[11]:

$$\begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = T_{view} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}, \tag{1.67}$$

where $T_{view} = RT$ describes the rotation and translation components, where:

$$T = \begin{bmatrix} 1 & 0 & 0 & -C_x \\ 0 & 1 & 0 & -C_y \\ 0 & 0 & 1 & -C_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad R = \begin{bmatrix} U_x & U_y & U_z & 0 \\ V_x & V_y & V_z & 0 \\ N_x & N_y & N_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.68}$$

Choosing the position of $C$ in the world coordinate system is relatively straightforward, but choosing the viewing direction is much more intuitive if we use two angles in a spherical coordinate system $(\rho, \phi, \theta)$ [12] (3D form of the polar coordinate system) - I suppose this is similar to the human head, looking up and down, or left to right. This is more formally know as:

- $\theta$ the azimuth angle (looking left and right, using the $+x$ axis as the reference)
- $\phi$ the elevation angle (looking up and down, using the $+z$ axis as the reference)

[11]The notation used in this document is: Scalar variables are represented by lowercase letters in italics, e.g. $a, b, x, y$; Vector variables are lowercase letters with an arrow, e.g. $\vec{a}, \vec{b}, \vec{x}, \vec{y}$; and Matrices are represented using uppercase letters e.g. $A, B, I, M$

[12]The spherical coordinate system is a coordinate system for representing geometric figures in three dimensions using three coordinates, $(\rho, \phi, \theta)$, where $\rho$ represents the radial distance of a point from the origin, $\phi$ represents the angle from the positive z-axis and $\theta$ represents the azimuth angle from the positive x-axis. To plot a point from its spherical coordinates, go $\rho$ units from the origin along the positive z-axis, rotate $\phi$ about the y-axis in the direction of the positive x-axis and rotate $\theta$ about the z-axis in the direction of the positive y-axis.

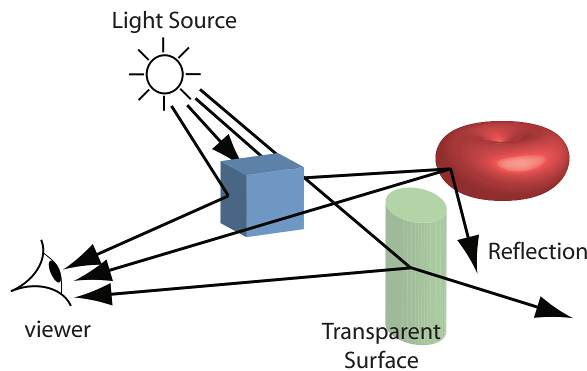



Figure 1.30: An example of the global effects that can occur when there are several objects with different shading properties in a scene. There can be multiple scattering from object to object and we can have other effects such as shadowing of one object on another (not illustrated).

To convert from the spherical coordinate system to the Carthesian coordinate system we can use:

$$\begin{aligned} x &= \rho \sin\phi \cos\theta \\ y &= \rho \sin\phi \sin\theta \\ z &= \rho \cos\phi \end{aligned}$$

Therefore, $\vec{n}$ can be described as:

$$\begin{aligned} \vec{n}_x &= \sin\phi \cos\theta \\ \vec{n}_y &= \sin\phi \sin\theta \\ \vec{n}_z &= \cos\phi \end{aligned}$$

## 1.6 OpenGL Shading

At this stage we have shown how OpenGL can be used to create primitive objectives from arrays of vertices. We have also shown how to add colour to an object; however, the colour that we have added so far has been relatively simple as it assumes that the object is lit in such a way that the colour appears uniform from all directions. This is not that realistic, as in the real-world objects will appear shaded, depending on the direction of light and the position of the observer. It is possible for an object to emit light (self-emission), such as a glowing object, or for an object to be highly reflective, like a mirror - Now, if two such objects are close together, there will be recursive effects when the light source emits light rays on the mirror, which will then be reflected back onto the light source itself, or indeed any other object in the scene. We can also describe this as having some light scattered and some absorbed, as in 1.30.

Light that strikes an object is partially absorbed and partially scattered; the amount of light reflected determines the colour and brightness of the object. A surface with a red material appears red under a white light because the red component of the light is reflected and the rest is absorbed. The reflected light is scattered in a manner that depends on the smoothness and orientation of the surface.

We shall begin by defining different terms to describe shaded surfaces:

- *Specular surfaces* - These surfaces appear shiny as the light that is reflected is maintained within a narrow range of angles, close to the angle of reflection. Mirrors are perfect specular surfaces.
- *Translucent surfaces* - These surfaces allow some of the light to penetrate the surface and to emerge from some other location on the object. For example, refraction in glass or water would cause the light to emerge from another location on the object.
- *Diffuse surfaces* - These surfaces are characterised by having light scattered in all directions; for example, walls painted with a matte paint are diffuse reflectors.

All of these shaded surfaces will appear differently depending on the wavelength of the light that strikes the surface and the orientation of the surfaces of the objects, which is characterised by the normal vector at each point. We can characterise a light source by a 6-variable illumination function: $I(x, y, z, \theta, \phi, \lambda)$, where each point on the surface $(x, y, z)$ can emit light by the direction of emission $(\theta, \phi)$ and intensity of energy at each wavelength $\lambda$. For this module, we will consider four types of source: ambient light, point sources, spotlights and distant lights. These types are sufficient for most applications.

- *Ambient light* - In a large room, such as a classroom the lights are placed on the roof in such a way as to try to uniformly distribute the light across every surface, creating a uniform light level. Ambient illumination can be characterised by an intensity $I_a$ that is identical at every point in the scene - Our ambient source will have three colour components (red, green, blue) and can be given as: $I_a = \begin{bmatrix} I_a^r & I_a^g & I_a^b \end{bmatrix}^T$
- *Point sources* - A point source emits light equally in all directions and can be characterised by a three component colour matrix: $I(P_0) = \begin{bmatrix} I_a^r(P_0) & I_a^g(P_0) & I_a^b(P_0). \end{bmatrix}^T$ The intensity of light received from the point source is proportional to the inverse square of the distance between the light source and the surface. Therefore, at a point $P$ the intensity of light received from the point source is given by the matrix: $i(P, P_0) = \frac{1}{|P-P_0|^2} I(P_0)$. Point sources are quite simple sources of light and tend to result in very high-contrast harsh renderings; however, the combination of point sources and ambient light reduces these effects.
- *Spotlights* - Spotlights are characterised by a narrow range of angles through which light is emitted. Typically we represent a spotlight as a cone with apex at $P_s$ which points in the direction $\vec{l_s}$ and whose cone radius is determined by the angle $\theta$ (a spotlight with $\theta = 180$ is a point source). More realistic spotlights have a more complex distribution of light within the cone, usually with light concentrated at the centre of the cone, often given by a function $\cos^e \phi$, where $0 \leq \phi \leq \theta$ and where the exponent $e$ determines how quickly the light intensity falls off.
- *Distant lights* - If a light source is far from the surface that it is illuminating then the light vector does not change much as we move across the surface (like the sun illuminating objects on the earth). This distant light in effect replaces a point source with a parallel source of light. A distant light source can be described by a direction vector as follows $P = \begin{bmatrix} x & y & z & 0 \end{bmatrix}^T$. OpenGL can carry out rendering calculations for distant light sources more efficiently that it can for near ones.

### 1.6.1 The Phong Model

The reflection model that we present here was introduced by Phong and later modified by Blinn. It provides a good approximation to physical reality, producing good renderings under varying lighting conditions and materials. The Phong model uses four vectors to calculate the colour at a particular point $P$ on a surface; these are $\vec{n}$, the normal vector

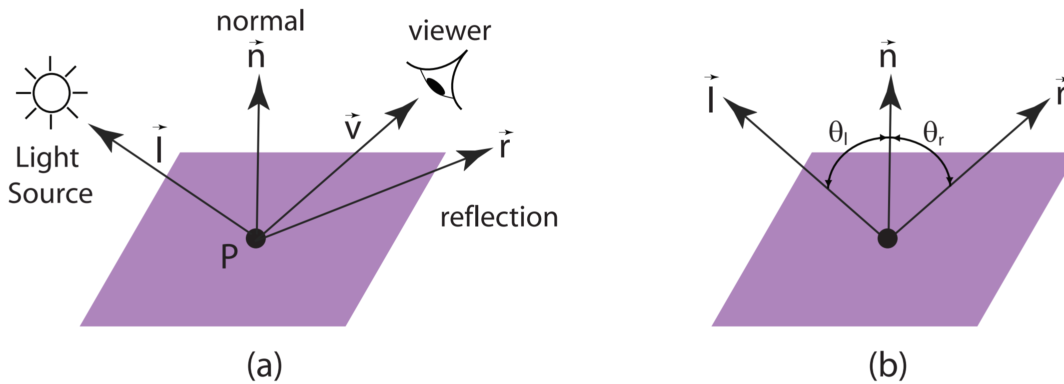



Figure 1.31: (a) illustrates the Phong model and (b) illustrates an ideal reflector.

at that point on the surface; $\vec{v}$, which is in the direction from point P to the viewer (or centre of projection);$\vec{l}$, the direction of a line from P to a point light source; and $\vec{r}$ is the direction that a perfectly reflected ray from $\vec{l}$ would take. 1.31(a) illustrates the Phong Model. The Phong model is a simple model that can be computed rapidly.

1.31(b) illustrates an ideal reflector, where the normal is determined by the orientation of the local polygon, and the angle of incidence $\theta_l$ equals the angle of reflection $\theta_r$. The ideal reflector can be described as $\vec{r} = 2(\vec{l} \cdot \vec{n})\vec{n} - \vec{l}$.

The Phong model supports the three types of material-light interaction of ambient, diffuse and specular. OpenGL works by assuming that if there is a set of point sources that each source can have separate red, green and blue ambient, diffuse and specular components. Therefore, at an point on a surface $P$ for the $i^{th}$ light source we can use:

$$L_i = \begin{bmatrix} L^r_{ia} & L^g_{ia} & L^b_{ia} \\ L^r_{id} & L^g_{id} & L^b_{id} \\ L^r_{is} & L^g_{is} & L^b_{is} \end{bmatrix} \tag{1.69}$$

which has the first row for the ambient intensities, the second row for the diffuse intensities and the last row for the specular intensities. We assume that we can calculate how much incident ligh t is reflected at $P$ based on the material properties, the orientation of the surface, the direction of the light source and the distance between the point on the surface and the viewer. Therefore, we can give the reflection in the form:

$$R_i = \begin{bmatrix} R^r_{ia} & R^g_{ia} & R^b_{ia} \\ R^r_{id} & R^g_{id} & R^b_{id} \\ R^r_{is} & R^g_{is} & R^b_{is} \end{bmatrix} \tag{1.70}$$

and therefore, we can obtain the total intensity by adding the contribution of all the sources, giving for the red intensity:

$$I^r_i = L^r_{ia}R^r_{ia} + L^r_{id}R^r_{id} + L^r_{is}R^r_{is}$$

If we simplify this to assume that the computations are the same for each primary colour, but differ depending if we are considering ambient, diffuse or specular terms, we can write:

$$I = L_aR_a + L_dR_d + L_sR_s$$

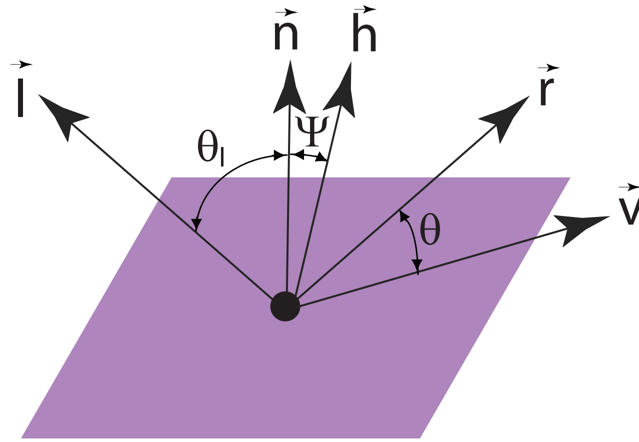



Figure 1.32: $\vec{h}$ is the halfway vector, half way between light vector $\vec{l}$ and the view vector $\vec{v}$

The intensity of ambient light is the same at every point on the surface, where some of the light is absorbed and some is reflected depending on some coefficient $k_a$, where $0 \leq k_a \leq 1$ as only a positive fraction of the light can be reflected, so $I_a = k_a L_a$.

We can add up all of the components, allowing the Phong model to be written without the distance terms as (referring to 1.31(a)):

$$I = k_d I_d (\vec{l} \cdot \vec{n}) + k_s I_s (\vec{v} \cdot \vec{r})^{\alpha} + k_a I_a \tag{1.71}$$

(1.72)

For each colour component we can add contributions from all sources.

The specular term in the Phong model causes problems because it requires the calculation of a new reflection vector and view vector for each vertex. Blinn suggested an approximation which uses the halfway vector that is more efficient. Figure 1.32 illustrates the halfway vector $\vec{h}$, which is half way between the light vector $\vec{l}$ and the view vector $\vec{v}$. This can be described as: $\vec{h} = (\vec{l} + \vec{v}) / \left|\vec{l} + \vec{v}\right|$. We can then replace the $(\vec{v} \cdot \vec{r})^{\alpha}$ component in our Phong model by $(\vec{n} \cdot \vec{h})^{\beta}$ to give:

$$I = k_d I_d (\vec{l} \cdot \vec{n}) + k_s I_s (\vec{n} \cdot \vec{h})^{\beta} + k_a I_a \tag{1.73}$$

(1.74)

Where $\beta$ is chosen to match the shininess of the surface. This is known as the modified Phong or Blinn lighting model. The vectors $\vec{l}$ and $\vec{v}$ are easy to calculate as they are specified by the application; we can compute $\vec{r}$ using $\vec{l}$ and $\vec{n}$, but it is not so easy to determine $\vec{n}$ - we will discuss this slightly later.

### 1.6.2 Lambertian Surfaces

As previously mentioned, diffuse reflections are characterised by rough surfaces, where rays of light that strike the surface are reflected back at quite different angles. Perfectly diffuse surfaces are called Lambertian Surfaces and can be modelled by Lambert's law, which states that:

$$R_d \propto \cos\theta \tag{1.75}$$

where $\theta$ is the angle between the normal at the point of interest $\vec{n}$ and the direction of the light source $\vec{l}$. If only a fraction of incoming light is reflected we can add in a reflection coefficient $k_d$ (where $0 \leq k_d \leq 1$) we can write:

$$R_d = k_d L_d \cos\theta \tag{1.76}$$

The amount of light is proportional to the vertical component of the incoming light. If we used only the ambient and diffuse properties our materials would appear dull as we are missing the highlights on the surface. The highlights that we see on a reflective surface are usually different from the colour of the ambient and diffuse light, for example a silver shiny metal ball viewed under white light will reflect a white highlight in the direction of the viewer. This is the specular property of the surface and is associated with a smooth surface, rather than the rough surface associated with a diffuse model. Modelling realistic specular surfaces is complex because the pattern by which light is scattered is not symmetric and depends on the reflection angle. Phong proposed an approximate model for specular surfaces, which assumes that the surface is smooth for the specular reflection. The Phong model uses the equation:

$$I_s = k_s L_s cos^{\alpha}\phi \tag{1.77}$$

which gives the specular light that a viewer sees at a particular point on the surface, where the coefficient $k_s$ (where $0 \leq k_s \leq 1$) is the faction of incoming specular light that is reflected, the exponent $\alpha$ is the shininess coefficient (where when $\alpha$ is increased the light is concentrated on a narrower region centered on the angle of a perfect reflector, i.e. as $\alpha \to \infty$ (typically values of 200 to 500) we get a perfect mirror), and $\alpha$ is the angle between $\vec{r}$ a perfect reflector and $\vec{v}$, the direction of the viewer.

### 1.6.3 The Normal Vector

With smooth surfaces the vector normal to the surface exists at every point and gives the local orientation of the surface. A surface normal, or just normal to a flat surface is a three-dimensional vector which is perpendicular to that surface (orthogonal). For a polygon, the surface normal can be calculated as the vector cross product of two (non-parallel) vector edges of the polygon. For a plane given by the equation $ax+by+cz+d=0$; this equation can be rewritten in terms of the normal to the plane as:

$$\vec{n} \cdot (P - P_0) = 0 \tag{1.78}$$

where $P$ is any point on the plane. If we have three non-colinear points $P_0$, $P_1$ and $P_2$ (and therefore a plane) then the vectors $P_1 - P_0$ and $P_2 - P_0$ can be used to calculate the normal as:

$$\vec{n} = (P_2 - P_0) \times (P_1 - P_0) \tag{1.79}$$

Figure 1.33 illustrates the calculation of the normal vector for a plane.

In OpenGL a normal can be associated with a vertex through:

```
0  glNormal3f(Nx, Ny, Nz);
   glNormal3fv(pointerToNormal);
```

Normals are state variables that can be associated with all the vertices and can be used for the lighting calculations at all of the vertices. Unfortunately, we are expected to calculate these normals ourselves.

The OpenGL Red Book gives an example of the use of normals with the creation of a icosahedron.

```
0   void drawGLIcosahedron() // From the red book
    {
        GLfloat x,y,z, c = 3.14159f/180.0f;
        #define X .525731112119133606
        #define Z .850650808352039932
5
        static GLfloat vdata[12][3] = {
                {−X, 0.0, Z}, {X, 0.0, Z}, {−X, 0.0, −Z}, {X, 0.0, −Z},
                {0.0, Z, X}, {0.0, Z, −X}, {0.0, −Z, X}, {0.0, −Z, −X},
                {Z, X, 0.0}, {−Z, X, 0.0}, {Z, −X, 0.0}, {−Z, −X, 0.0}
10          };
        static GLuint tindices[20][3] = {
                {0,4,1}, {0,9,4}, {9,5,4}, {4,5,8}, {4,8,1},
                {8,10,1}, {8,3,10}, {5,3,8}, {5,2,3}, {2,7,3},
```

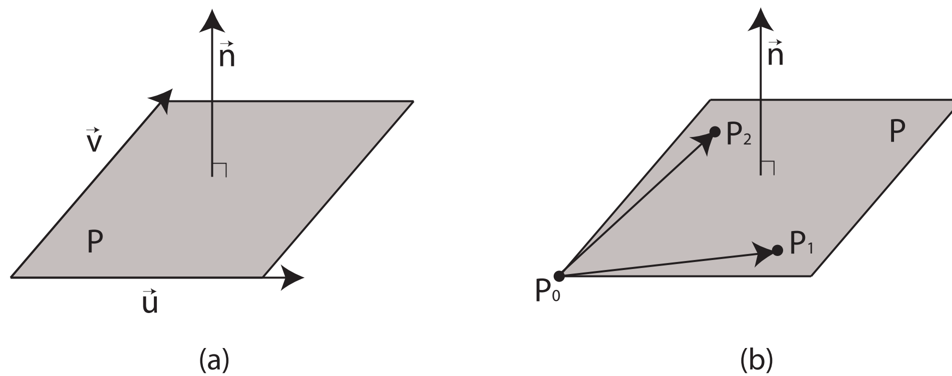



Figure 1.33: (a) Illustrates the calculation of the normal $\vec{n}$ from two vectors $\vec{u}$ and $\vec{v}$, using $\vec{n} = \vec{u} \times \vec{v}$ and (b) illustrates the calculation of the normal using three non-colinear points as $\vec{n} = (P_2 - P_0) \times (P_1 - P_0)$.

```
               {7,10,3}, {7,6,10}, {7,11,6}, {11,0,6}, {0,1,6},
15             {6,1,10}, {9,0,11}, {9,11,2}, {9,2,5}, {7,2,11} };
          int i;
          glBegin(GL_TRIANGLES);
             for (i = 0; i < 20; i++) {

20                GLfloat d1[3], d2[3], norm[3];
                  for (int j = 0; j < 3; j++) {
                     d1[j] = vdata[tindices[i][0]][j] - vdata[tindices[i][1]][j];
                     d2[j] = vdata[tindices[i][1]][j] - vdata[tindices[i][2]][j];
                  }
25                normcrossprod(d1, d2, norm);
                  glNormal3fv(norm);

                  glVertex3fv(&vdata[tindices[i][0]][0]);
                  glVertex3fv(&vdata[tindices[i][1]][0]);
30                glVertex3fv(&vdata[tindices[i][2]][0]);
             }
          glEnd();
    }

35  void normalize(float v[3]) {
       GLfloat d = sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);
       if (d == 0.0) {
          return;
       }
40     v[0] /= d; v[1] /= d; v[2] /= d;
    }

    void normcrossprod(float v1[3], float v2[3], float out[3])
    {
45     GLint i, j;
       GLfloat length;
       out[0] = v1[1]*v2[2] - v1[2]*v2[1];
       out[1] = v1[2]*v2[0] - v1[0]*v2[2];
       out[2] = v1[0]*v2[1] - v1[1]*v2[0];
50     normalize(out);
    }
```

Example1k illustrates this example... see 1.34. As stated, Gouraud proposed the use of the normals around a mesh vertex. Gouraud shading finds the average normal at each vertex and applies the modified Phong model at each vertex. It then Interpolates vertex shades across each polygon. Phong shading finds the vertex normals; interpolates the vertex normals across the edges; interpolates edge normals across the polygon and then applies the modified Phong model at each fragment.

If the polygon mesh approximates a surface with a high curvature then Phong shading tends to look much smoother, while Gouraud shading tends to show up edges. Phong shading requires much more computational effort than Gouraud shading - It was not until recently that Phong shading was available in real-time, by using fragment shaders. Both of these shading schemes need data structures to represent the mesh, by which shading calculations can be performed.

Figure 1.34: These images give examples of the calculation of normals for an icosahedron.

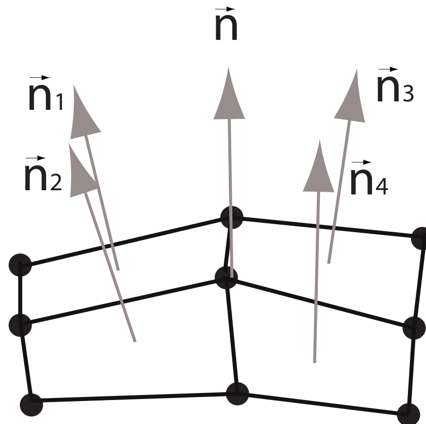



Figure 1.35: Smooth Shading - Normals at vertices

### 1.6.4 Shading

One of the problems with the calculation of these normals is that it is computationally intensive. OpenGL uses the efficiencies possible for rendering flat surfaces, by decomposing curved surfaces into many small flat polygons. We will look at the different shading models used in OpenGL.

#### Flat Shading

The three vectors $\vec{l}$ (light), $\vec{v}$ (viewer) and $\vec{n}$ (normal) vectors which describe how an object should appear. If the light source and viewer are distant (or simply far away relative to the size of a polygon) and if we assume a flat polygon, then $\vec{n}$ is constant over the polygon, then the shading calculation need only be carried out once for each polygon and each polygon is given the same shade. With OpenGL we can set flat (constant) shading through a call:

```
0  glShadeModel(GL_FLAT);
```

OpenGL will use the normal associated with the first vertex of a polygon for all vertices of that polygon if flat shading is enabled. This shading method is not very attractive as humans are very good at noticing shading intensity changes at linear boundaries (e.g. a square of different shaded paint on a wall painted with a similar shade of paint).

#### Smooth Shading

OpenGL also provides us with a smooth shading model, which can be enabled using:

```
0  glShadeModel(GL_SMOOTH);
```

Figure 1.35 illustrates the normals at four connected polygons. Mathematically speaking, the calculation of a normal at a vertex should cause concern, as it is discontinuous at this point. However, Gouraud showed that by defining a normal at a vertex that smoother shading could be achieved over an object by using interpolation. In this figure we can calculate the normal $\vec{n}$ at this vertex by using:

$$\vec{n} = \frac{\vec{n}_1 + \vec{n}_2 + \vec{n}_3 + \vec{n}_4}{|\vec{n}_1 + \vec{n}_2 + \vec{n}_3 + \vec{n}_4|} \tag{1.80}$$

This should not be very difficult from an OpenGL as all we have to do is specify the normals at the vertices and let OpenGL do the interpolation; however, it is not that simple as we do not tend to store the polygons in an order that allows us to average these polygons together. We would have to set up a data structure to store our models that puts the vertices first, i.e. stores four polygons for each vertex.

The default setup is to shade only front faces of objects, which is fine for convex objects; however, if we need to allow back faces to be seen, then we can set two sided shading to be enabled, using the `glMaterialf()` function with the properties `GL_FRONT`, `GL_BACK` and `GL_FRONT_AND_BACK`.

### 1.6.5 Lighting

As previously discussed, OpenGL supports four types of light sources, ambient, point, spotlight and distant. We can set up the position of a light using a call to:

```
glLightfv(GLenum source, GLenum parameter, GLfloat *pointer_to_array);
```

Shading calculations are enabled using a call to `glEnable(GL_LIGHTING)` we still have to enable each light source individually using `glEnable(GL_LIGHTi)`, where $i = 0, 1, \ldots n$, where $n$ is defined by your graphic card.

Allows us to create a light, where we first have to set up the required parameters, such as:

```
GLfloat light_pos[]  = {-1.0f, 1.0f, 2.0f, 1.0f};
GLfloat light_Ka[]  = {0.0f, 0.0f, 0.0f, 1.0f};
GLfloat light_Kd[]  = {1.0f, 1.0f, 1.0f, 1.0f};
GLfloat light_Ks[]  = {1.0f, 1.0f, 1.0f, 1.0f};

glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
glLightfv(GL_LIGHT0, GL_AMBIENT, light_Ka);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_Kd);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_Ks);

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

Where the light position is specified using a homogeneous co-ordinate system. The fourth parameter of $K_a$, $K_d$ and $K_s$ calls specify the alpha property of the ambient, diffuse and specular components of the light source. We then associate the arrays with the `GL_POSITION`, `GL_AMBIENT`, `GL_DIFFUSE` and `GL_SPECULAR` properties of the light source `GL_LIGHT0`. Finally, we have to enable the `GL_LIGHT0` and if necessary enable lighting on the OpenGL machine. The light position is given in homogeneous co-ordinates, where the 4th parameter is $1.0f$ if we are specifying a finite location, or $0.0f$ if we are specifying a parallel source with the given direction vector. The coefficients in the distance terms are given by default as $a = 1.0$, the constant terms; $b = c = 0.0f$, the linear and quadratic terms. If we wish to change these values we can do this by:

```
glLightfv(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 0.50f);
//Also have GL_LINEAR_ATTENUATION and GL_QUADRATIC_ATTENUATION
```

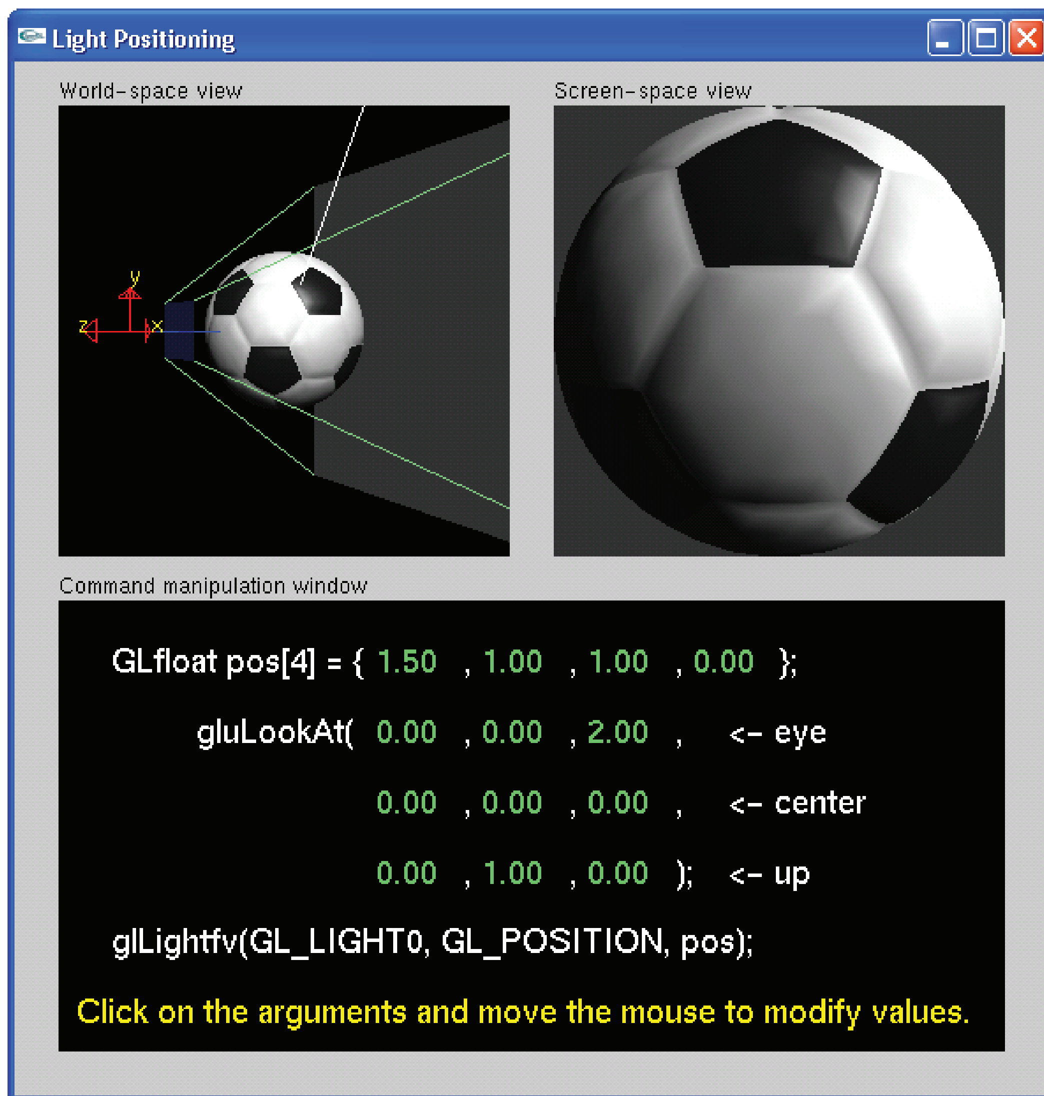



Figure 1.36: The Nate Robins Tutorial Example on Light Positioning

Figure 1.36 demonstrates the positioning of light in a scene. If we wished to have ambient light we can use:

```
GLfloat ambient_light[] = {0.1, 0.1, 0.1, 1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambient_light);
```

Example1j illustrates the use of point lights and ambient lights.

We can convert a point source to a spotlight by setting up a spotlight direction property `GL_SPOT_DIRECTION`, the exponent `GL_SPOT_EXPONENT` and the angle `GL_SPOT_CUTOFF`. We use the same `glLightfv()` method to set these properties up. Depending on the graphic card we are limited in the number of light sources that may be present.

### 1.6.6 OpenGL code for Shading

The Nate Robins' Tutorial gives an excellent example of the use of OpenGL code for specifying light and material properties. See Figures 1.36 and 1.38.

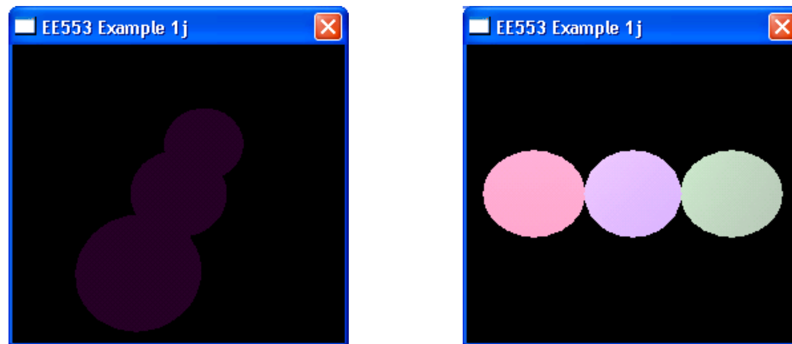



Figure 1.37: This figure illustrates the use of a point light and an ambient light. The image on the left shows the three spheres lit by the ambient light only and the image on the right shows the spheres having rotated into the beam of the light and view of observer.

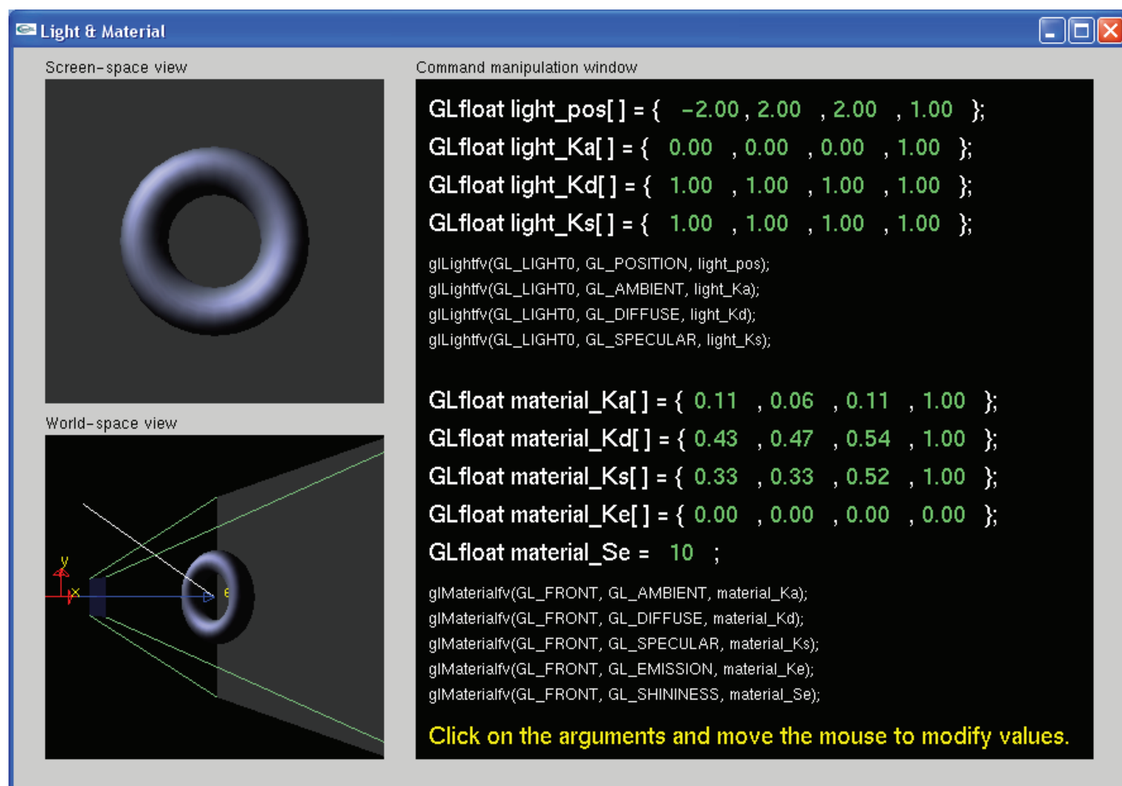



Figure 1.38: The Nate Robins Tutorial on Light and Materials in OpenGL (lightmaterial.exe). You can right click the various parameters to change their values.

# Chapter 2

# Scene Graph Theory

## 2.1 Introduction

In a previous section of this module you have experienced the use of the Java3D API and its associated scene-graph. In this section of the module we are going to examine scene graphs in more detail, in particular examining how we can build our own type of scene graph, which suits our particular application. To do this we are going to have to understand how we can organise a scene into a tree structure, how we can traverse this tree and how we can use this tree to represent complex concepts such as describing the relationship between objects, lights and cameras.

So, what is a scene graph? Well, the scene graph we are concerned with is a collection of nodes ordered into a graph or tree structure. Each node can have many children, but usually each child has only one parent. This is quite similar to the object-oriented trees that you would have seen in EE553 - object-oriented programming - in fact, more closely associated with the inheritance trees found in Java. Within a scene graph, operations that are applied to an object are propagated to the child nodes. As such, scene graphs are very useful for representing complex geometries, and operations such as transformation, selections etc. which can be applied to all components of the object.

For example, a car could have a body and four rotating wheels; the wheels are positioned and rotate relative to the body of the car, so a translation applied to the car body should have an equivalent translation effect on the four wheels. This allows us to treat the car with wheels as a single object and reduces the complexity of our application. Furthermore, as memory and performance are always concerns in 3-D graphics, a well structured scene graph would allow us to create reference nodes to geometries. In the example of the car, this would allow us to define the geometry of the car's wheel only once and use three references to position three 'copies' of the wheel that have been rotate and translated to the correct location. To do this we would use a directed acyclic graph (DAG), which is a generalisation of a tree in which certain subtrees can be shared by different parts of the tree, greatly reducing the memory footprint of our scene when there is repetition of scene objects - a Java3D Scene Graph is a DAG.

It should be clear at this stage that the concept of a scene graph is a clear candidate for an object-oriented approach. This is certainly the case, but a word of warning at this stage - OpenGL has a state-machine structure, which does not sit very well with the object-oriented structure that we are going to impose on our scenes. It is worth mentioning that scene graphs have found use in non-3D applications where there is a requirement to manage the contents of spatially oriented data; in particular, MPEG4 uses a scene graph programming model for multimedia scene composition.

## 2.2 A Simple Scene Graph Implementation

There are many scene graphs available including the Java3D API, Virtual Reality Modeling Language (VRML) and indeed Open Scene Graph and they have specific tasks in mind. For this module we are going to create our own *simple* scene graph to demonstrate how they work.

The simplest type of scene graph would use an array data structure to store the objects of the scene graph. We would iterate over the array to display the object in the scene, simply traversing from the first element in the array to the Nth element whenever we need to update the display. For small scenes this would be fine, but when we have a large scene, which may contain thousands of scene objects, this linear display will become noticeably slow. Larger scene graphs tend to use tree structures to contain the scene objects as the hierarchical relationship it provides allows significant performance efficiencies. For the car example, if we decided not to draw the car, then we could skip the scene objects which contained the four wheels etc.

A scene graph is an organised hierarchy of nodes; an n-ary tree, where a node can contain any number of children, which can inherit the transformations and states of the parent objects. To organise a scene into a tree we generally use *nodes* such as *group nodes* and *leaf nodes*. Group nodes usually have many nodes attached and can often have a transformation or switching function. Leaf nodes usually contain renderable scene objects, such as graphical geometry objects, or non-graphical scene objects such as images, lights, cameras, materials and even sounds.

The reason that we use an object-oriented programming approach is so that we can build a future-proofed extensible core engine for our visualisation systems. From EE553 object-oriented programming, you are probably aware that the features that are important to our scene graph design are *inheritance* and *polymorphism*. Inheritance allows us to define scene objects from other scene objects, and to extend or replace some of the functionality of the base class. Polymorphism allows us to use the `virtual` keyword to call methods of derived classes using pointers that have a static type of the base class. The object-oriented structure is also very useful in managing the data in our scene, in particular avoiding memory leaks as our visualisation system must be capable of running for a long period of time.

As stated, the n-ary tree is used to contain scene objects and the idea of an array was discussed. A more appropriate method would be to use the ANSI/ISO C++ Standard Template Library (STL). STL is a collection of container classes, with associated iterator mechanisms and algorithms. It provides classes for stacks, queues, linked-lists, deques (double ended queues), maps, sets and vector storage containers. All of these containers are useful in building our tree structures. We have chosen the vector class container for our implementation as it provides a simple to use random access container; in effect it is an advanced random access dynamic array. Iterators are used to move through the data structure and access its information.

### Example Scene Graph Object Class

Here is our first design of our generic Scene Object base class:

```
0  // SceneObject.h− This is the base object of all objects in
   // the scene graph.
   //////////////////////////////////////////////////////////////////////

   #if !defined SCENEOBJECT_H
5  #define SCENEOBJECT_H

   #include<vector> // Use the STL Vector as our container

   enum RTTI_OBJECT_TYPE
```

```
{
        RTTI_CAMERA, //does not exist yet
        RTTI_LIGHT,  //does not exist yet
        RTTI_DUMMY,
};

class SceneObject
{

protected:
        SceneObject*                    parentObject;
        std::vector<SceneObject*>       childrenObjects;

public:
        SceneObject();
        virtual ~SceneObject();

        // assessors/mutators
        void setParent(SceneObject* parent)     { parentObject = parent; }
        void addChild(SceneObject* child);
        std::vector<SceneObject*>* getChildrenObjects() { return &childrenObjects; }
        virtual RTTI_OBJECT_TYPE getType() const = 0;

        // Force every child to have a render and update methods
        virtual void render(float timeElapsed) = 0;
        virtual void update(float timeElapsed) = 0;
};

#endif // SCENEOBJECT_H
```

And its associated C++ file:

```
// SceneObject.cpp: implementation of the SceneObject class.
//
//////////////////////////////////////////////////////////////////////

#include <iostream>
#include "SceneObject.h"

SceneObject::SceneObject() {}

SceneObject::~SceneObject() {}

void SceneObject::addChild(SceneObject *child)
{
        if (!child) { std::cerr << "Attempt to add invalid child to scene graph."; }

        child->setParent(this);
        childrenObjects.push_back(child);
}
```

Before this class begins we can see the `RTTI_OBJECT_TYPE` enumeration. As discussed, we will use polymorphism extensively. So a pointer with a static type of SceneObject will sometimes need to know what kind of object this pointer is actually pointing to (i.e. the dynamic type). This enumeration allows us to "name"' the current object type. The list that is here is an example list for a camera, a light and a dummy object. We will extend this list later. In effect, an `enum` call simply numbers the list of constants. If you have studied Java you will be aware of a Class class, which performs the same operation. This implementation of run-time type information (RTTI) allows a constant to define exactly what kind of object we are dealing with at run-time. We can then simply use the

`getType()` method to extract the dynamic type of the object that is currently pointed to.

### Example Dummy Object Class

We can create various different scene objects. For now, we will just us a Dummy object that does not involve any 3-D graphics; rather, it will just output the name of the object to the standard output stream (`std::cout`).

```
// DummyObject.h: interface for the DummyObject class.
//
//////////////////////////////////////////////////////////////////////

#if !defined DUMMYOBJECT_H
#define DUMMYOBJECT_H

#include <iostream>
#include <string>
#include "SceneObject.h"

class DummyObject : public SceneObject
{
protected:
        std::string name;

public:
        DummyObject(std::string dummyName);
        virtual ~DummyObject();

        virtual RTTI_OBJECT_TYPE getType() const { return RTTI_DUMMY; }

        virtual void render(float timeElapsed) {}
        virtual void update(float timeElapsed)
    {
      std::cout << "Updating [" << name << "] scene object " << std::endl;
    }
};

#endif // DUMMYOBJECT_H
```

And its associated C++ file:

```
// DummyObject.cpp: implementation of the DummyObject class.
//
//////////////////////////////////////////////////////////////////////

#include <string>
#include "DummyObject.h"

DummyObject::DummyObject(std::string dummyName)
{
        name = dummyName;
}

DummyObject::~DummyObject() {}
```

### 2.2.1 Traversing the Scene Graph

Once we have our scene graph structure of choice in place it is important that we are able to interact with its information. The most basic operation is traversing the scene graph; that is, moving from one scene graph item to the next scene graph item. In the case of a

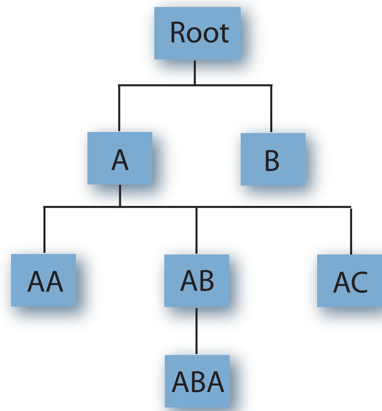



Figure 2.1: The example scene graph that we are creating.

basic array scene graph structure, this would involve iterating over the array one element at a time, probably from the $0^{th}$ to the $n^{th}$ element in the array.

In the case of a tree structure, a traversal usually involves starting at some arbitrary node of the graph, usually the root node and rendering the scene or applying some other form of operation to the nodes. For the tree, this usually involves visiting a node, then recursively moving down the tree to the child nodes until a leaf node is met. After this we traverse back up the tree until we reach the next node that provides another downward path. With the tree structure it is important to realise that we have the facility to rebuild the scene graph into a more efficient order, perhaps a spatial ordering or an application specific oder. For example, in 3-D systems it is more efficient to draw the closest objects first, since objects that are further away may not need to be rendered as they may be occluded by the closest objects.

**Example Scene Graph Class with Traversal**

Now we will carry out an implementation of traversing our scene graph. To do this we will have to use recursion. Recursion is the much feared technique whereby a method calls itself. The classic example of a recursive function is given by the factorial function (i.e. a call to `factorial(5)` will give $5 * 4 * 3 * 2 * 1 = 120$)

```
0 int factorial (int n)
  {
    if (n <= 1) return 1;
    else return (n * factorial(n-1));
  }
```

This function will call itself until the number 1 is reached. Often recursion is misunderstood; for example, the not-so-funny joke where you look up recursion in the dictionary to find: "*recursion: see recursion*". This brings up an important point; there must be an exit condition in our recursive function, otherwise it is a simple infinite loop. In the code segment above, when we reach a value of `n<=1` (it would have been sufficient to say `n==1`) the return value of 1 means that we are no longer calling the recursive function; therefore, the evaluation of `factorial(n-1)` will return the value of 1, the previous call will return n times 1 etc. until we get back to the original call to `factorial(5)` and the cycle terminates. Recursion for our scene graph structure is straightforward and necessary.

We can traverse a tree in two ways, breadth first or depth first. Breadth first traversal moves from left to right on the same level, before going to the next lower level on the tree. Depth first traversal moves down the tree until it cannot go down anymore, then it moves up the tree until the next "lateral" path becomes available. For our scene graph we will use depth first traversal as it makes more sense to render entire objects together;

for example, in the case of the car discussed earlier, to render the body and then the four wheels, rather than move directly to the next object in the scene.

Here is a very basic Scene Graph Class:

```
// SceneGraph.h: interface for the SceneGraph class.
//
//////////////////////////////////////////////////////////////////////

#if !defined SCENEGRAPH_H
#define SCENEGRAPH_H

class SceneObject; //avoid a circular definition

class SceneGraph
{
  public:

        SceneGraph();
        virtual ~SceneGraph();

        void updateScene(SceneObject* sceneObject, float timeElapsed);
};

#endif // SCENEGRAPH_H
```

And its associated C++ file which includes the code for traversal:

```
// SceneGraph.cpp: implementation of the SceneGraph class.
//
//////////////////////////////////////////////////////////////////////

#include "SceneGraph.h"
#include "SceneObject.h"
#include <vector>
#include <iostream>

SceneGraph::SceneGraph() {}

SceneGraph::~SceneGraph() {}

void SceneGraph::updateScene(SceneObject* sceneObject, float timeElapsed)
{
        if (!sceneObject)
        {
                std::cerr << "Attempt update of object not on the scene graph.";
                return;
        }
        else
        {
                sceneObject->update(timeElapsed);
                // and call all the children
                std::vector<SceneObject*>::iterator it =
                 sceneObject->getChildrenObjects()->begin();
                for (; it!=sceneObject->getChildrenObjects()->end(); ++it)
                {
                        updateScene(*it, timeElapsed);
                }
        }
}
```

In this code segment we have the `SceneGraph::updateScene()` method which takes a `sceneObject` and recursively calls the `update()` method on each of the object's children

using a depth first tree traversal. We use the STL vector iterator, so the vector of child objects has a starting point, provided by `begin()` and an ending point, provided by `end()`; we simply move the iterator over this range by using the `++it` call in the `for` loop.

Here is an example of the construction of an example scene graph using the classes above. In this example we pass a root node pointer to the `SceneGraph::updateScene()` method, so that all scene objects are updated.

```
/****************************************
 * EE563 Example Project 4 - Scene Graph Example
 * by: Derek Molloy
 ****************************************/

#include <windows.h> // Header File For Windows
#include <gl\gl.h>   // Header File For The OpenGL32 Library
#include <gl\glu.h>  // Header File For The GLu32 Library
#include "vec3.h"
#include "Matrix.h"
#include "vec4.h"
#include "Quat.h"
#include "SceneGraph.h"
#include "DummyObject.h"
#include <iostream>
using namespace std;

int main()
{
   SceneGraph *sg = new SceneGraph;

   std::cout << "Creating Scene Objects \n";

   DummyObject *root  = new DummyObject("Dummy Root");
   DummyObject *dummya = new DummyObject("Dummy A");
   DummyObject *dummyb = new DummyObject("Dummy B");
   root->addChild(dummya);
   root->addChild(dummyb);
   DummyObject *dummyaa = new DummyObject("Dummy AA");
   DummyObject *dummyab = new DummyObject("Dummy AB");
   DummyObject *dummyac = new DummyObject("Dummy AC");
   DummyObject *dummyaba = new DummyObject("Dummy ABA");
   dummya->addChild(dummyaa);
   dummya->addChild(dummyab);
   dummya->addChild(dummyac);
   dummyab->addChild(dummyaba);
   sg->updateScene(root, 0.0f);

   std::cout << "Scene Updated \n";

   system("pause");
   return 0;
}
```

The output of this application is:

```
Creating Scene Objects
Updating Dummy Root scene object
Updating Dummy A scene object
Updating Dummy AA scene object
Updating Dummy AB scene object
Updating Dummy ABA scene object
Updating Dummy AC scene object
```

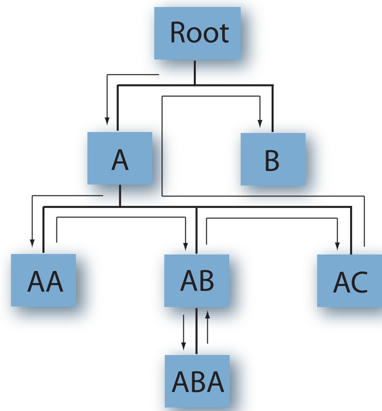



Figure 2.2: The depth first traversal that has taken place on this scene graph.

```
Updating Dummy B scene object
Scene Updated

Press any key to continue . . .
```

### 2.2.2 Putting it Together: The Scene Graph with OpenGL

Now, we will take this scene graph structure and apply it to the 3-D world. To do this we will need to add more concepts to our scene graph. The previous code example provides a template for adding any type of scene graph structure. We will begin with a Primitive type, which will allow us to describe primitives such as spheres, cubes etc. The implementation for this discussion is Example 5.

```
0   #if !defined PRIMITIVE_H
    #define PRIMITIVE_H

    #include "SceneObject.h"

5   enum PRIMITIVE_TYPE
    {
        SPHERE,
        CUBE
    };
10
    class Primitive : public SceneObject
    {
    private:
        PRIMITIVE_TYPE type;
15      float size;
        float divisions;

    public:
            Primitive(PRIMITIVE_TYPE, float, float);
20          virtual ~Primitive();

            virtual RTTI_OBJECT_TYPE getType() const { return RTTI_PRIMITIVE; }

            // Force every child to have a render and update methods
25          virtual void render(float timeElapsed);
            virtual void update(float timeElapsed);
```

```
protected:
    virtual void drawSphere();

};

#endif // PRIMITIVE_H
```

The Primitive extends the SceneObject class and so can be added directly to the scene graph and can even have child nodes below it - maybe this is not ideal? but, this is our scene graph structure, so we can do what we want.

Now, we must also describe a Transformation. Once again, this has been designed as a child of SceneObject so that it can be added directly to the scene graph, and can have further children - this is very important in the case of a transform node, as it would be meaningless as a leaf node. The Transform class really just encapsulates a single matrix; we could provide much more advanced functionality here. The render method simply applies this matrix as described below:

```
#if !defined TRANSFORM_H
#define TRANSFORM_H

#include "SceneObject.h"
#include "Matrix.h"

class Transform : public SceneObject
{
private:
    Matrix matrix;

public:
        Transform(Matrix);
        virtual ~Transform();

        virtual RTTI_OBJECT_TYPE getType() const { return RTTI_TRANSFORM; }

        // Force every child to have a render and update methods
        virtual void render(float timeElapsed);
        virtual void update(float timeElapsed);
};

#endif // TRANSFORM_H
```

And a segment of the C++ file:

```
void Transform::render(float timeElapsed)
{
    glMultMatrixf(matrix.ptr());
}
```

We can put this all together to give:

```
/******************************************
 * EE563 Example Project 5 - Scene Graph Example with Graphics
 * by: Derek Molloy
 ******************************************/

#include <windows.h> // Header File For Windows
#include <gl\gl.h>      // Header File For The OpenGL32 Library
#include <gl\glu.h>     // Header File For The GLu32 Library
#include "vec3.h"
#include "Matrix.h"
#include "vec4.h"
#include "Quat.h"
#include "SceneGraph.h"
#include "DummyObject.h"
#include "Primitive.h"
#include "Transform.h"
#include <iostream>
using namespace std;
```

```
HDC                         hDC=NULL; // Private GDI Device Context
HGLRC          hRC=NULL; // Permanent Rendering Context
HWND           hWnd=NULL; // Holds Our Window Handle
HINSTANCE    hInstance; // Holds The Instance Of The Application
GLfloat      fov = 60.0f;
int          downX = 0, downY = 0;
float        xOffset = 0.0f, yOffset = 0.0f;
SceneGraph *sg;

// Function Declarations

LRESULT CALLBACK WndProc (HWND hWnd, UINT message, WPARAM wParam,
        LPARAM lParam);
void enableOpenGL (HWND hWnd, HDC *hDC, HGLRC *hRC);
int initGL(GLvoid);
int drawGLScene(float theta);
void disableOpenGL (HWND hWnd, HDC hDC, HGLRC hRC);
void defineGLSphere(GLfloat, GLfloat);
void increaseFOV();
void decreaseFOV();
void setFOV(GLfloat);

// WinMain - the starting point of our application

int WINAPI WinMain (HINSTANCE hInst, HINSTANCE hPrevInstance,
    LPSTR lpCmdLine, int iCmdShow)
{
    WNDCLASS wc;
    HGLRC hRC;
    hInstance = hInst;
    MSG msg;
    BOOL bQuit = FALSE;
    float theta = 0.0f;

    // register window class
    wc.style = CS_OWNDC;
    wc.lpfnWndProc = WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon (NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor (NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = "EE553GLExample";
    RegisterClass (&wc);

    // create main window
    hWnd = CreateWindow ("EE553GLExample", "EE553_Example_5",
      WS_CAPTION | WS_POPUPWINDOW | WS_VISIBLE,
      0, 0, 256, 256,
      NULL, NULL, hInstance, NULL);

    // enable OpenGL for the window
    enableOpenGL (hWnd, &hDC, &hRC);
    initGL();

    // program main loop
    while (!bQuit)
    {
        // check for messages
        if (PeekMessage (&msg, NULL, 0, 0, PM_REMOVE))
        {
            // handle or dispatch messages
            if (msg.message == WM_QUIT)
            {
                bQuit = TRUE;
            }
            else
            {
                TranslateMessage (&msg);
                DispatchMessage (&msg);
            }
        }
        else
        {
            drawGLScene(theta);
            SwapBuffers (hDC);
            theta += 1.0f;
            Sleep (1);
        }
    }
    // shutdown OpenGL
    disableOpenGL (hWnd, hDC, hRC);
    // destroy the window explicitly
    DestroyWindow (hWnd);
    return msg.wParam;
}


// Window Callback Process

LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
        case WM_RBUTTONDOWN: // if the right button is down
             setFOV(60);
             xOffset=0.0f;
             yOffset=0.0f;
             return 0;
        case WM_LBUTTONDOWN:
```

```
                downX = (int)(short)LOWORD(lParam);
                downY = (int)(short)HIWORD(lParam);
          case WM_MOUSEMOVE: // if the mouse is moved
                if (wParam == MK_LBUTTON) // and the left mouse button is down
                {
                   int xPos = (int)(short)LOWORD(lParam);
                   int yPos = (int)(short)HIWORD(lParam);
                   int diffx  = xPos - downX;
                   int diffy  = yPos - downY;
                   xOffset+= ((float)diffx)/10;
                   yOffset+= ((float)diffy)/10;
                }
                return 0;
          case WM_CREATE:
               return 0;
          case WM_CLOSE:
               PostQuitMessage (0);
               return 0;
          case WM_DESTROY:
               return 0;
          case WM_KEYDOWN:
               switch (wParam)
               {
               case VK_ESCAPE:
                    PostQuitMessage(0);
                    return 0;
               case VK_LEFT:
                    xOffset-=0.1f;
                    return 0;
               case VK_RIGHT:
                    xOffset+=0.1f;
                    return 0;
               case VK_UP:
                    yOffset-=0.1f;
                    return 0;
               case VK_DOWN:
                    yOffset+=0.1f;
                    return 0;
               case VK_F1: //F1 key pressed
                    return 0;
               }
               return 0;
          case WM_CHAR:
                switch(wParam)
                {
                  case 'a':
                     decreaseFOV();
                     return 0;
                  case 's':
                     increaseFOV();
                     return 0;
                }
                return 0;
          default:
               return DefWindowProc (hWnd, message, wParam, lParam);
     }
}


//Enable OpenGL

void enableOpenGL (HWND hWnd, HDC *hDC, HGLRC *hRC)
{
     PIXELFORMATDESCRIPTOR pfd;
     int iFormat;
     // get the device context (DC)
     *hDC = GetDC (hWnd);

     // set the pixel format for the DC
     ZeroMemory (&pfd, sizeof (pfd));
     pfd.nSize = sizeof (pfd);
     pfd.nVersion = 1;
     pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
     pfd.iPixelType = PFD_TYPE_RGBA;
     pfd.cColorBits = 24;
     pfd.cDepthBits = 16;
     pfd.iLayerType = PFD_MAIN_PLANE;
     iFormat = ChoosePixelFormat (*hDC, &pfd);
     SetPixelFormat (*hDC, iFormat, &pfd);

     // create and enable the render context (RC)
     *hRC = wglCreateContext( *hDC );
     wglMakeCurrent( *hDC, *hRC );
}

// Setup our GL Scene
int initGL(GLvoid)
{
          glShadeModel(GL_SMOOTH);        // Enable Smooth Shading
          glClearColor(1.0f, 1.0f, 1.0f, 1.0f);// White Background
          glClearDepth(1.0f);                                        // Depth Buffer Setup
          glEnable(GL_DEPTH_TEST);                        // Enables Depth Testing
          glDepthFunc(GL_LEQUAL);                         // The Type Of Depth Testing To Do
          glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
                                      // Really Nice Perspective Calculations

     sg = new SceneGraph;
     DummyObject *root  = new DummyObject("Root");
     sg->bindRoot(root);

     DummyObject *dummya = new DummyObject("Dummy_A");
     DummyObject *dummyb = new DummyObject("Dummy_B");
     root->addChild(dummya);
```

```
root->addChild(dummyb);

Primitive *p = new Primitive(SPHERE, 1.0f, 36.0f);
root->addChild(p);

Matrix m1, m2;
m1.makeTranslate(2.0f, 0.0f, 0.0f);
Transform *t1 = new Transform(m1);
root->addChild(t1);
Primitive *p1 = new Primitive(SPHERE, 2.5f, 20.0f);
t1->addChild(p1);

m2.makeTranslate(-2.0f, 0.0f, 0.0f);
Transform *t2 = new Transform(m2);
root->addChild(t2);
Primitive *p2 = new Primitive(SPHERE, 2.5f, 10.0f);
t2->addChild(p2);

    return TRUE;                                    // Initialization Went OK
}

void increaseFOV() {
    if (fov<180.0f) fov+=1.0;
}

void decreaseFOV(){
    if (fov>0.0f) fov-=1.0;
}

void setFOV(GLfloat f){
    if (f>0.0f && f<180.0f) fov = f;
}

// Called to update the scene - give us the animation
int drawGLScene(float theta)                    // Draw the scene;
{                                           // theta is amount to rotate
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear Screen
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE); // display all mesh lines of the sphere

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(fov, 1.0, 0.1, 100.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    glTranslatef(xOffset, yOffset, 0);

    glColor3f(0.0f, 1.0f, 0.0f);  // green
    sg->renderScene(0.0f);

        return TRUE; // Keep Going
}

// Disable OpenGL
void disableOpenGL (HWND hWnd, HDC hDC, HGLRC hRC)
{
    wglMakeCurrent (NULL, NULL);
    wglDeleteContext (hRC);
    ReleaseDC (hWnd, hDC);
}
```

Where, the most important part is:

```
sg = new SceneGraph;
DummyObject *root = new DummyObject("Root");
sg->bindRoot(root);

DummyObject *dummya = new DummyObject("Dummy_A");
DummyObject *dummyb = new DummyObject("Dummy_B");
root->addChild(dummya);
root->addChild(dummyb);

Primitive *p = new Primitive(SPHERE, 1.0f, 36.0f);
root->addChild(p);

Matrix m1, m2;
m1.makeTranslate(2.0f, 0.0f, 0.0f);
Transform *t1 = new Transform(m1);
root->addChild(t1);
Primitive *p1 = new Primitive(SPHERE, 2.5f, 20.0f);
t1->addChild(p1);

m2.makeTranslate(-2.0f, 0.0f, 0.0f);
Transform *t2 = new Transform(m2);
root->addChild(t2);
```

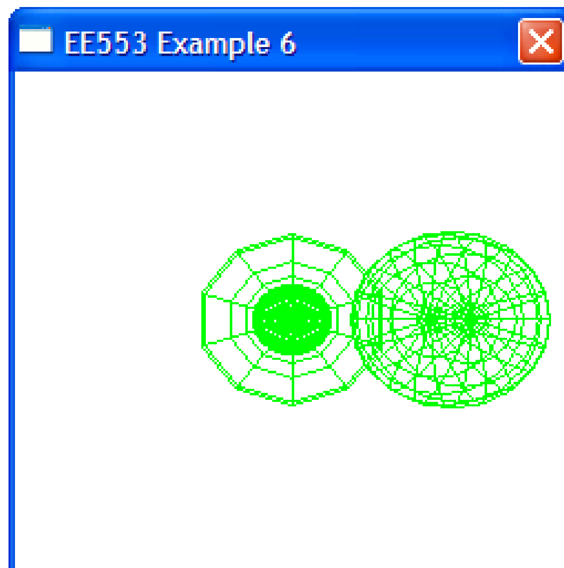



Figure 2.3: Putting it Together: The Scene Graph with OpenGL

```
    Primitive *p2 = new Primitive(SPHERE, 2.5f, 10.0f);
    t2->addChild(p2);
```

The output of this segment of code is displayed in figure 2.3 where we can see three spheres; a small one in the centre with a very high resolution (36 divisions); a bigger one to the right (20 divisions); and, the same size one again at the centre with (10 divisions). This comes back to problems that we had before, where a transform of (2, 0, 0) applied to the second sphere places it to the right, but a subsequent transform by (-2, 0, 0) returns the next object back to the origin.

This current implementation has a few problems:

- All transforms are relative to the previously drawn object on the scene graph, rather than the object above it on the scene graph - Not ideal!
- All primitives are the same colour

### 2.2.3 Making it Better: The Scene Graph with OpenGL

Now, we will improve the scene graph by adding transformations relative to the parent, rather than some previously drawn object - which we may not entirely be aware of. First we have to address the transform issues - we do this as follows:

- Push and Pop matrices, so that when we move through the scene graph from parent to child we push the current matrix; when we move from child to parent we pop the current matrix.
- We also need to alter our recursive algorithm to add a postRender() method so that we can trigger an event on a move from child to parent.

```
void Transform::render(float timeElapsed)
{
    glPushMatrix();
    glMultMatrixf(matrix.ptr());
}

void Transform::postRender(float timeElapsed)
{
    glPopMatrix();
}
```

And the recursive algorithm now becomes:

```
void SceneGraph::renderScene(SceneObject* sceneObject, float timeElapsed)
{
        if (!sceneObject)
        {
                std::cerr << "Attempt update of object not on the scene graph.";
                return;
        }
        else
        {
                sceneObject->render(timeElapsed);
                // and call all the children
                std::vector<SceneObject*>::iterator it =
                 sceneObject->getChildrenObjects()->begin();
                for (; it!=sceneObject->getChildrenObjects()->end(); ++it)
                {
                        renderScene(*it, timeElapsed);
                }

                sceneObject->postRender(timeElapsed); /**** HERE ****/
        }
}
```

To create a Material class, we can use:

```
// Material.h
/////////////////////////////////////////////////////////////////////////////

#if !defined MATERIAL_H
#define MATERIAL_H

#include "SceneObject.h"
#include "Vec4.h"

class Material : public SceneObject
{
private:
    Vec4 ambient;
    Vec4 diffuse;
    Vec4 specular;
    Vec4 emission;
    float shininess;

public:
        Material();
        virtual ~Material();

        virtual RTTI_OBJECT_TYPE getType() const { return RTTI_MATERIAL; }

        // Force every child to have a render and update methods
        virtual void render(float timeElapsed);
        virtual void update(float timeElapsed);

        virtual void setAmbient(Vec4 v) { ambient = v; }
        virtual void setDiffuse(Vec4 v) { diffuse = v; }
        virtual void setSpecular(Vec4 v) { specular = v; }
        virtual void setEmission(Vec4 v) { emission = v; }
        virtual void setAmbient(float f) { shininess = f; }
};

#endif // MATERIAL_H
```

And the C++ file:

```
#include "Material.h"
#include "gl/gl.h"

Material::Material()
{
    ambient = Vec4(0.11f, 0.06f, 0.11f, 1.0f);
    diffuse = Vec4(0.43f, 0.47f, 0.54f, 1.0f);
    specular = Vec4(0.33f, 0.33f, 0.52f, 1.0f);
    emission = Vec4(0.00f, 0.00f, 0.00f, 0.0f);
    shininess = 10;
}

Material::~Material() {}

void Material::render(float timeElapsed)
{
     glMaterialfv(GL_FRONT, GL_AMBIENT, ambient.ptr());
     glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse.ptr());
     glMaterialfv(GL_FRONT, GL_SPECULAR, specular.ptr());
     glMaterialfv(GL_FRONT, GL_EMISSION, emission.ptr());
     glMaterialfv(GL_FRONT, GL_SHININESS, &shininess);
}

void Material::update(float timeElapsed)
{
     std::cout << "Updating a Material " << std::endl;
}
#endif // MATERIAL_H
```

So, now we can create a scene graph as follows:

```
    sg = new SceneGraph;
    DummyObject *root = new DummyObject("Root");
    sg->bindRoot(root);

    Light *l = new Light();
    root->addChild(l);

    Primitive *p = new Primitive(SPHERE, 1.0f, 36.0f);
    p->getMaterial()->setDiffuse(Vec4(0.18f, 0.81f, 0.31f, 1.0f));   //green
    root->addChild(p);

    Matrix m1, m2, m3;
    m1.makeTranslate(2.0f, 0.0f, 0.0f);
    Transform *t1 = new Transform(m1);
    root->addChild(t1);
    Primitive *p1 = new Primitive(SPHERE, 2.5f, 20.0f); //blue by default
    t1->addChild(p1);

    m2.makeTranslate(-2.0f, 0.0f, 0.0f);
    Transform *t2 = new Transform(m2);
    root->addChild(t2);
    Primitive *p2 = new Primitive(SPHERE, 2.5f, 10.0f);
    p2->getMaterial()->setDiffuse(Vec4(0.89f, 0.25f, 0.14f, 1.0f));   //red
    t2->addChild(p2);
```

This is coded in Example 6, which gives the output as displayed by figure 2.4(a). It can then be altered easily, such as:

```
    sg = new SceneGraph;
```

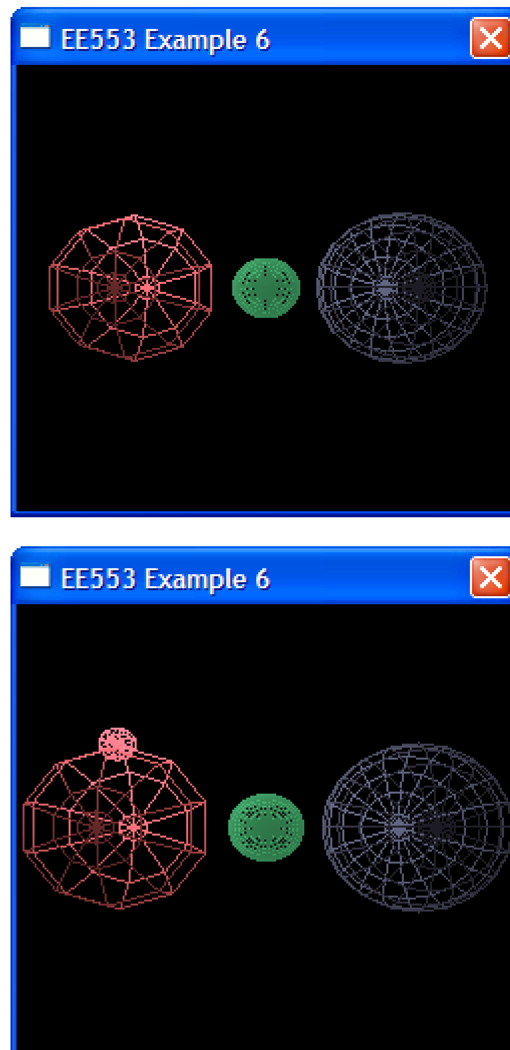



Figure 2.4: Making it Better: The Scene Graph with OpenGL

```
...

m3.makeTranslate(0.0f, 1.25f, 0.0f);
Transform *t3 = new Transform(m3);
p2->addChild(t3);
Primitive *p3 = new Primitive(SPHERE, 0.5f, 10.0f);
p3->getMaterial()->setDiffuse(Vec4(0.89f, 0.25f, 0.14f, 1.0f));    //red
t3->addChild(p3);
```

Which gives the output as displayed by figure 2.4(b).

## 2.3 Constructive Solid Geometry (CSG)

Constructive Solid Geometry (CSG) is a technique whereby a complex model can be created by using Boolean operations to combine a set of primitive shapes (spheres, cubes, cylinders etc.). It is generally used for rigid CAD type models of machinery and equipment, where it is possible that these complex models will be manufactured from such primitives. Figure 2.5 should illustrate the main concepts behind CSG - In this example we have the equation $(A - B) \cup (C \cap D)$, which is represented as a CSG tree in the same figure, with the final output displayed in figure 2.6.

CSG does suffer from problems: The tree representation is quite straightforward, but the computation time required to produce the final model is significant as 3-D Boolean

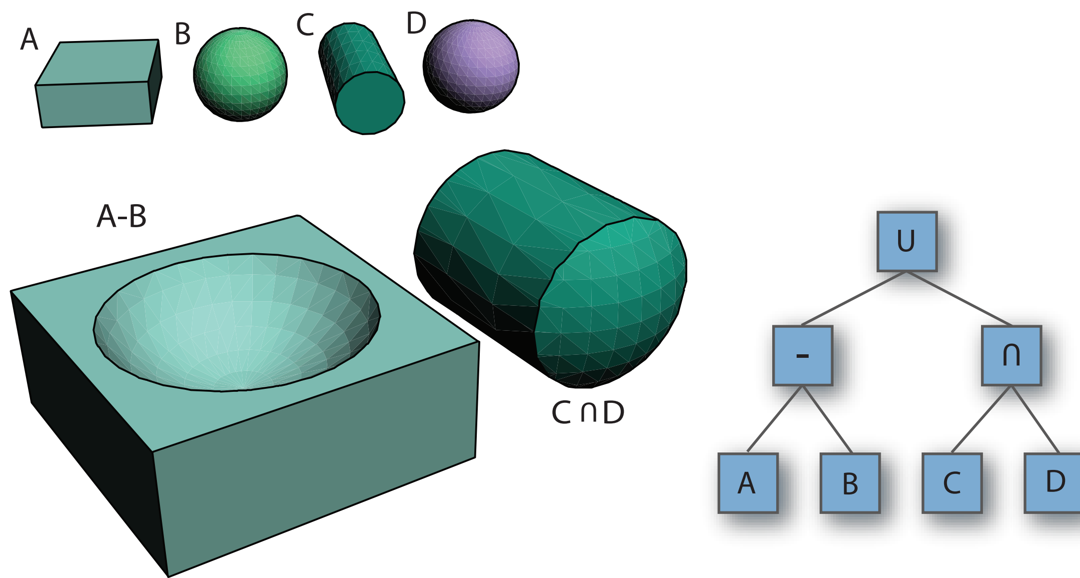



Figure 2.5: CSG Example illustrating the shapes $A$, $B$, $C$, $D$; the result of $A - B$ and of $C \cap D$; and the associated CSG Tree.

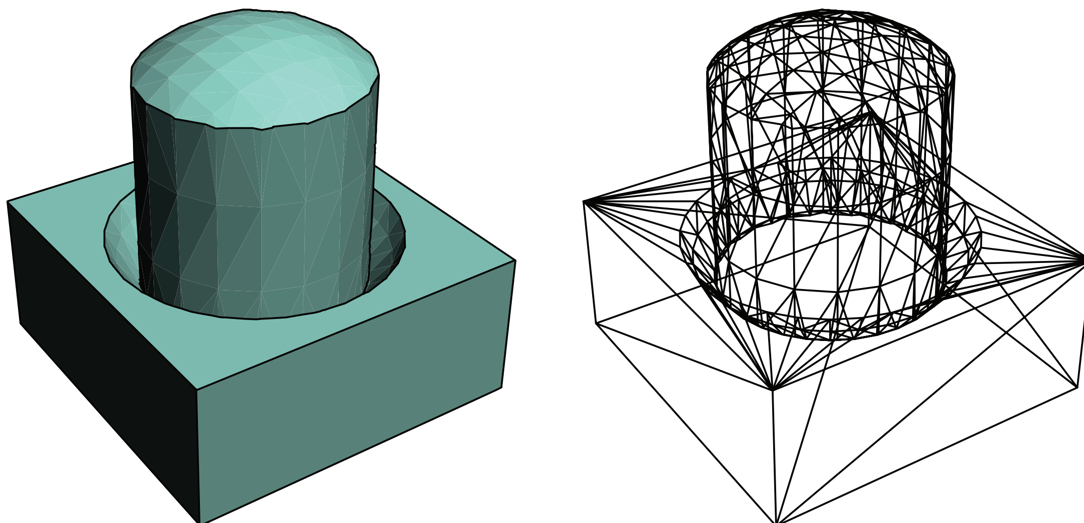


Figure 2.6: CSG Example illustrating the result of $(A - B) \cup (C \cap D)$, in both solid and wireframe form (with hidden lines visible.)

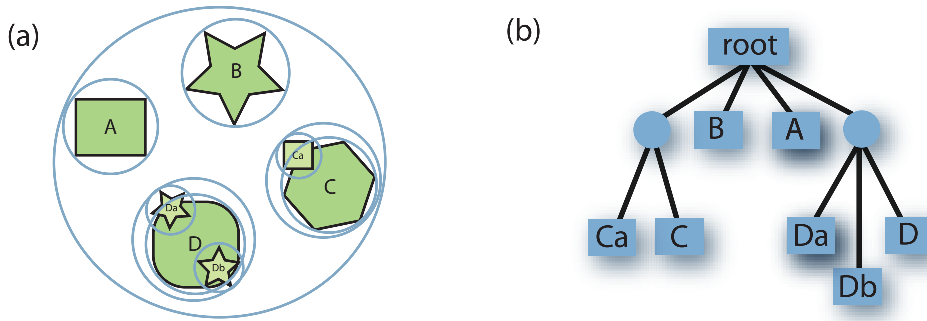



Figure 2.7: An illustration of the bounding volume hierarchy constructed using circles (spheres for 3-D).

operations are computationally expensive, especially as the shape becomes more complex. The number of operations available is also quite restrictive: there is the *union* of two shapes $A \cup B$ giving the set of vertices that are in either $A$ and $B$; *intersection* $A \cap B$ giving the set of all vertices that are in both shapes; *difference* $A - B$ giving the set of vertices that are in $A$, but not in $B$. CSG is a good methodology for CAD modelling as there is a strong relationship between primitives and industrial processes; for example, subtracting a cylinder from an object is equivalent to drilling a hole in that object.

## 2.4 Scene Graphs and Bounding Volume Hierarchies

Bounding volumes such as bounding spheres and boxes can be used as stand-in primitives for objects with complex geometries. If we have such an object that is contained within a bounding box, and we wish to determine if the object is within a specific region of space - Rather than test if any part of the object is within the region of space, we can test if the bounding box is within that region of space. If the bounding box is not in that region of space we can ignore the entire object, but if it is, then we can examine the object in more detail. The bounding volume of each object is calculated the first time the objects are added to the scene graph; This means that it is very efficient to use the bounding volume rather than examine each object independently, especially with operations such as culling and collision detection. A bounding box is also quite easy to calculate as we simply iterate through the vertices of the shape to find the minimum and maximum $x$ and $y$ values.

A Bounding Volume Hierarchy (BVH) is a tree of bounding volumes where the root node includes every object in the scene and at the leaf nodes each bounding volume is just large enough to contain each scene object. The tree takes on the same hierarchical shape as the scene graph. Once again, we can quickly determine if an object is in a particular region of space using its bounding volume, but the hierarchy also allows us to determine if the segmented objects' bounding volumes contained in the child nodes are also within this region of space. The tree like structure allows all these tests to be performed very quickly. It is common practice for us to use the same object-oriented tree structure for the scene graph and also for the bounding volume hierarchy. Figure 2.7(b) illustrates the creation of a bounding volume hierarchy from the scene in figure 2.7(a).

As can be seen in figure 2.7(b), the BVH structure is a tree, where leaf nodes contain the geometry and each group node can have $n$ children. Each parent node has a bounding volume that encloses the geometry in its entire subtree; therefore, the root node's bounding volume encloses the entire scene. The BVH is excellent for performing queries such as "does a raytracing ray intersect with objects in the scene". The first test will take place at the root node, if this node's bounding volume is missed by the ray then the ray has missed the entire subtree; otherwise testing will continue recursively.

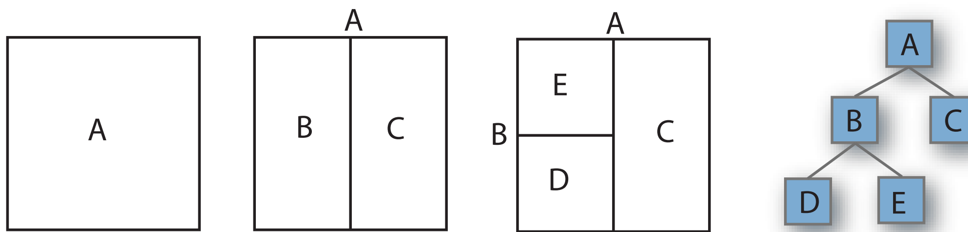



Figure 2.8: A 2-D axis-aligned BSP Example, illustrating the segmentation of 2-D space $A$ by a line that splits the space into two; the BSP tree is given on the right hand side

If the scene is dynamically changing then we can update the tree at runtime. If a particular BV node has moved, we can very quickly check if it is still contained within its parent BV. If it is then the BVH is valid, but if not, the node can be removed and the parent's BVH must be recomputed. This can have implications further up the tree also.

## 2.5 Space Subdivision Structures

We have examined Scene Graph implementations that use trees to describe the high-level relationships between the various scene objects, and we have discussed CSG which uses trees to create the scene objects themselves. Now we will examine another use of the tree structure for describing the 3-D world itself in terms of its geometry, which will allow us to improve the real-time performance of our applications. The techniques to be discussed here are used in a wide range of real-time applications, such as culling, collision detection and ray tracing. Once again the relationships that we are going to represent with a tree structure are hierarchical, where the parent object in the hierarchy will enclose the child object below it.

### 2.5.1 Binary Space Partitioning Trees

CSG Trees were used to describe the relationships between component parts of an object and previously in our scene graph we used trees to represent the hierarchical relationship between objects in a scene. Trees can also be used to capture the spatial relationship between objects in a scene. This is particularly important when performing visibility tests on the scene; if objects cannot be seen by the camera then from an efficiency perspective there is no point in rendering them to the scene.

Binary Space Partition Trees (or BSP trees for short) where introduced by Fuchs, Kedem, and Naylor around 1980 (?). The first ever computer game to use BSP trees was Doom[1], created by John Carmack and John Romero; since then all first person shooting games have used this technique.

Binary Space Partitioning (BSP) is a technique for recursively subdividing a 3-D space into two non-overlapping regions using a plane, referred to as *a hyperplane*[2]. Any point in 3-D space lies within only one of these regions. BSP is a hierarchal approach where the space that is divided can be further subdivided using the same space partitioning approach until some condition is met, resulting in a space-partitioning tree, which is particularly useful when building techniques for dealing with hidden surface removal. The basic properties of BSPs are that objects on one side of a hyperplane cannot intercept an object on the other side; and given a particular view point objects on the same side of the hyperplane are closer than objects on the other side.

The algorithm to build a BSP tree is fairly straightforward:

[1] `http://www.idsoftware.com`

[2] A hyperplane in $n$-dimensional space is an $n-1$-dimensional object that can be used to divide the $n$-dimensional space into two $n$-dimensional spaces - e.g. in 2-D a line is used

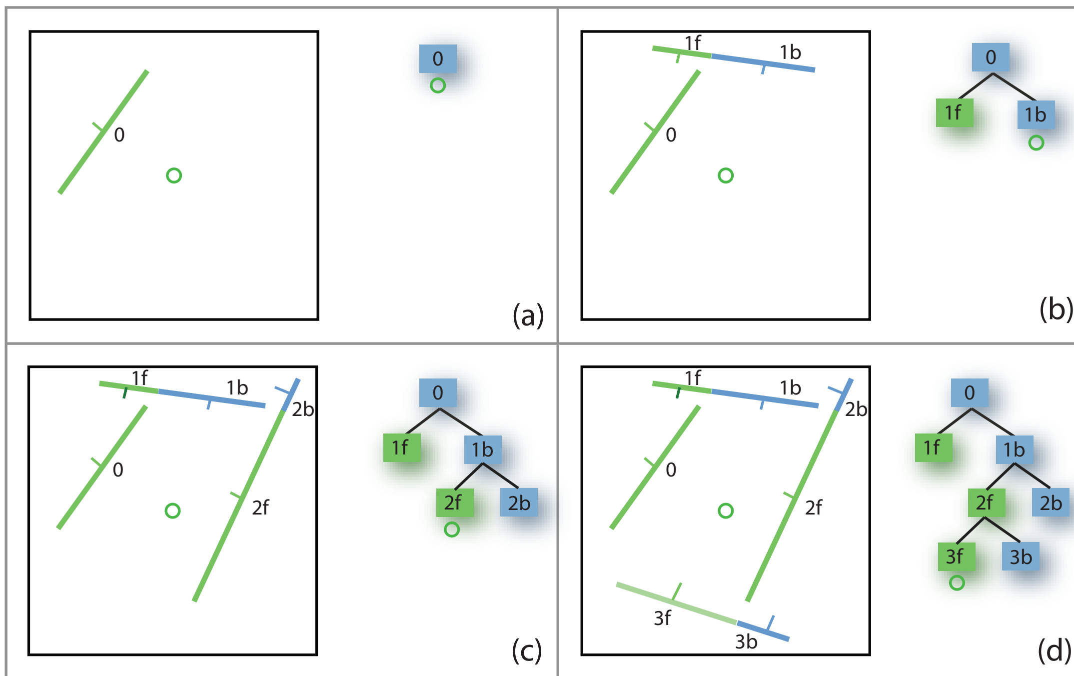



Figure 2.9: This 2-D example illustrates the sample creation of a BSP Tree using lines to create the tree. Each line represents a partition plane (a) creates the partition plane and thus the root node, (b) through (d) illustrates the addition of further planes, with *f* representing the front side and *b* representing the back side.

- Select a partition plane - The choice of planes is application dependent, but often axis aligned. In an ideal situation this will result in a balanced tree, but a poor choice will result in a large number of splits and an increase in the number of polygons. There is usually a trade-off between a well-balanced tree and a large number of splits.

- Segment the current set of polygons using the chosen plane - If a polygon lies entirely to one side or other of the plane then it is not modified and is added to the partition set for the side that it is on. If the polygon spans the partition plane then it is split into two pieces, which are added to the set on the correct side of the plane.

- Repeat again using the new sets of polygons - The termination condition is a application specific, often based on a maximum number of nodes in a leaf node, or maximum tree depth.

Figure 2.9 illustrates the core idea behind the creation of a non-axis aligned BSP tree in the 2-D case. It takes a set of lines (polygons in the 3-D case) that is a part of a scene and divides them into smaller sets, where each subset is a convex set of lines (i.e. each line is in-front-of every other line in the same set. In this example we choose an arbitrary plane in space and divide the lines, by putting the lines in the positive side of the plane in the left subtree and the lines on the negatives side in the right sub tree. Where the partition planes intersect we divide the line (again, polygon in 3-D) according to the plane that the line defines. We can therefore split a line into two parts *f*(in front) and *b*(behind). As can be seen from this straightforward example, it will be difficult to balance the tree. If there are too many polygons created in the tree then there will be a computational cost in maintaining and rendering the tree, but if the tree is unbalanced it will be expensive to traverse.

Here is the C++ pseudo code on how you might code a BSP Tree[3]:

```
struct BSP_tree
{
    plane      partition;
    list       polygons;
    BSP_tree *front, *back;
};
```

This struct stores the partitioning plane for the node, the list of polygons coincident with the partitioning plane and the pointers to its children. For this example, there will always be at least one polygon in the coincident list and the child pointers will be initialised to NULL.

```
void   Build_BSP_Tree (BSP_tree *tree, list polygons)
{
    polygon *root = polygons.Get_From_List ();
    tree->partition = root->Get_Plane();
    tree->polygons.Add_To_List(root);
    list  front_list , back_list ;
    polygon *poly;

    while ((poly = polygons.Get_From_List ()) != 0)
    {
        int    result  = tree->partition.Classify_Polygon (poly);
        switch (result)
        {
            case COINCIDENT:
                tree->polygons.Add_To_List (poly);
                break;
            case IN_BACK_OF:
                back_list .Add_To_List(poly);
                break;
            case IN_FRONT_OF:
                front_list .Add_To_List(poly);
                break;
            case SPANNING:
                polygon   *front_piece , *back_piece;
                Split_Polygon (poly, tree->partition, front_piece , back_piece);
                back_list .Add_To_List (back_piece);
                front_list .Add_To_List (front_piece );
                break;
        }
    }
    if ( !  front_list .Is_Empty_List())
    {
        tree->front = new BSP_tree;
        Build_BSP_Tree (tree->front, front_list );
    }
    if ( !  back_list .Is_Empty_List())
    {
        tree->back = new BSP_tree;
        Build_BSP_Tree (tree->back, back_list);
    }
}
```

This routine recursively constructs a BSP tree using the above algorithm. It takes the first polygon from the input list and uses it to partition the remainder of the set. The routine then calls itself recursively with each of the two partitions. This implementation

[3]Pseudo code modified from: http://www.opengl.org/resources/code/samples/bspfaq/

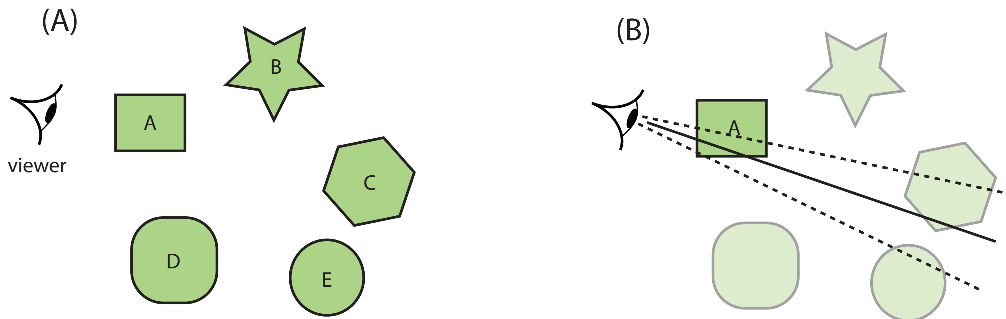



Figure 2.10: (a) Illustrates a viewer looking at 2-D objects in a scene; (b) Illustrates the actual objects that are seen by the viewer in this scene.

Figure 2.11: A simple test scene and the z-buffer representation

assumes that all of the input polygons are convex. Probably the most common application of BSP trees is hidden surface removal in 3-D. BSP trees provide an elegant, efficient method for sorting polygons via a depth first search . This fact can be exploited in a back to front "Painter's Algorithm" approach to the visible surface problem, painting close objects first. The BSP Tree is particularly computationally efficient as the calculation can be carried out in the pre-processing of the map and not at run-time.

## 2.6 Hidden Surface Removal

Visibility Testing is an important problem in 3-D Computer Graphics as it impacts on the efficiency of our algorithms: Very simply put, if an object is not visible then we should not display it. So, figure 2.10(a) assumes we have a set of objects in our scene graph which are in front of a viewer - should we display all these figures using OpenGL? Well no, if we look closely at the viewer and examine its field of view, we can see that only object $A$ is visible to the viewer. We wish to generate the image that the viewer would see and so judging from 2.10(b) we would only have to render object $a$.

The simple solution would be to use a z-buffer (depth buffer), a buffer that stores the z-depth of each pixel currently at each point on the 2-D screen. If another object in the scene is to be rendered, the algorithm compares the depth value of the object's rendering to the current screen z-buffer value at that pixel, and will choose the closer pixel. It update the z-buffer with the closest pixel (as demonstrated in 2.11).

The z-buffer approach was traditionally a very computationally expensive solution to the depth test problem. It requires additional memory to store the z-buffer 2-D array and extra computational cost in performing the depth test at every pixel. A software algorithm was developed to reduce this cost; the Painter's Algorithm.

Figure 2.12: The simple Painters's Algorithm shown on the LHS with the numbers representing the order in which the different regions are painted; 1 is painted first and 4 is painted last. The RHS illustrates a situation where the Painter's Algorithm fails as there are no planes separating the objects in the scene.

### 2.6.1 The Painter's Algorithm and BSP

The Painter's Algorithm is a fairly simple algorithm for determining the visibility of polygons in 3-D computer graphics. The algorithm refers to a painter that paints the distant parts of the scene first and then paints this over with parts that are nearer (See figure 2.12). The Painter's Algorithm sorts all the polygons in the scene by their depth and then paints them in the same order as the painter, from furthest to closest. This very simple algorithm solves the visibility problem, but it has incurred the computational cost of painting distant regions that will be painted over by closer regions. One condition for a successful painter's algorithm is that there be a single plane which separates any two objects. This means that it might be necessary to split polygons in certain configurations. For example, the RHS of figure 2.12 illustrates a case that cannot be drawn correctly when using the painter's algorithm.

One reason that BSP trees are appropriate for the Painter's Algorithm is that splitting difficult polygons can be an automated part of tree construction. To draw the contents of the tree you can perform a back to front tree traversal, which begins at the root node and classifies the eye point with respect to its partition plane. We draw the subtree at the far child from the eye, then the polygons in this node, followed by the near subtree. This is repeated recursively for each subtree.

It is just as easy to traverse the BSP tree in front-to-back order as it is for the back-to-front order just discussed. We can use this to our advantage in a scan line renderer by using a write mask which will prevent pixels from being written more than once. This will produce significant speedups if a complex lighting model is evaluated for each pixel, because the painter's algorithm will blindly evaluate the same pixel many times. The trick to making a scan line approach successful is to have an efficient method for masking pixels. One way to do this is to maintain a list of pixel spans which have not yet been written to for each scan line. For each polygon scan converted, only pixels in the available spans are written, and the spans are updated accordingly. [add reference]

When building a BSP tree specifically for hidden surface removal, the partition planes are usually chosen from the input polygon set. However, any arbitrary plane can be used provided there are no intersecting or concave polygons, as in figure 2.12. Here is a simple C++ pseudo code example of a back to front tree traversal:

```
0 void Draw_BSP_Tree(BSP_tree *tree, point eye)
  {
     real result = tree->partition.Classify_Point(eye);
     if (result > 0)
     {
```

```
5       Draw_BSP_Tree(tree->back, eye);
        tree->polygons.Draw_Polygon_List();
        Draw_BSP_Tree(tree->front, eye);
    }
    else if (result < 0)
10  {
        Draw_BSP_Tree(tree->front, eye);
        tree->polygons.Draw_Polygon_List();
        Draw_BSP_Tree(tree->back, eye);
    }
15  else // result is 0
    {
        // the eye point is on the partition plane...
        Draw_BSP_Tree(tree->front, eye);
        Draw_BSP_Tree(tree->back, eye);
20  }
}
```

If the eye point is classified as being on the partition plane, the drawing order is unclear. This is not a problem if the `Draw_Polygon_List` routine is smart enough not to draw polygons that are not within the viewing frustum. The coincident polygon list does not need to be drawn in this case, because those polygons will not be visible to the user. It is possible to substantially improve the quality of this example by including the viewing direction vector in the computation. You can determine that entire subtrees are behind the viewer by comparing the view vector to the partition plane normal vector. This test can also make a better decision about tree drawing when the eye point lies on the partition plane. It is worth noting that this improvement resembles the method for tracing a ray through a BSP tree, which is discussed in another section of the notes. Front to back tree traversal is accomplished in exactly the same manner, except that the recursive calls to `Draw_BSP_Tree` occur in reverse order.

As we have seen, the idea behind having a pre-calculated BSP tree is that it allows us to get the correct depth order of polygons in the scene for any point in space. Figure 2.13 illustrates the core operations on a BSP. In figure 2.13(a) a scene with 5 objects has been partitioned using a simple polygon-aligned BSP Tree algorithm. In this scene, object $E$ has been split by the partition lines into two separate objects, the larger part on the forward side of $L_3$ and the larger part on the behind side of $L_3$. 2.13(b) shows the calculated BSP tree, where circular nodes represent the partition lines and square nodes represent the leaf geometry nodes. In this example $E*$ represents a partial object $E$ and $f$ is the forward and $b$ is the behind side of each partition line. Now, in 2.13(c) an observer enters the scene and we wish to calculate the nearest object. We can do this by traversing the tree (rather than calculating the Euclidean distance to every object in the scene). Starting at the root node, is the observer in front of, or behind $L_1$? In this case the observer is behind $L_1$, so in 2.13(d) we travel down the $b$ side of the tree to $L_3$, repeating this for $L_3$ we find that the observer is again behind $L_3$ and finally we find that the observer is in front of $L_5$. At this point we find that the closest object to the observer is $D$. In 2.13(e) we see that we can then use this point to find the visibility order of all the objects in the tree. Starting at $D$ we traverse to the previous circular node, and then down the tree, finding $E*$ to be the next closest object. The path is traversed as illustrated in (e) until the visibility order is determined (in this case as, $D$, $E*$, $E*$, $C$, $B$, $A$, where $A$ is the most distant).

OpenGL achieves hidden surface removal through the use of a depth buffer and depth testing. Depth buffering must be enabled with the command `glEnable(GL_DEPTH_TEST)`. Before drawing a scene the depth buffer should be cleared with the command `glClear(GL_DEPTH_BUFFER_BIT)`. The command `glCullFace(GLenum mode)` indicates which polygons should be discarded (culled) before they are converted to screen coordinates. The parameter mode is either `GL_FRONT, GL_BACK`, or `GL_FRONT_AND_BACK` to indicate front-facing,

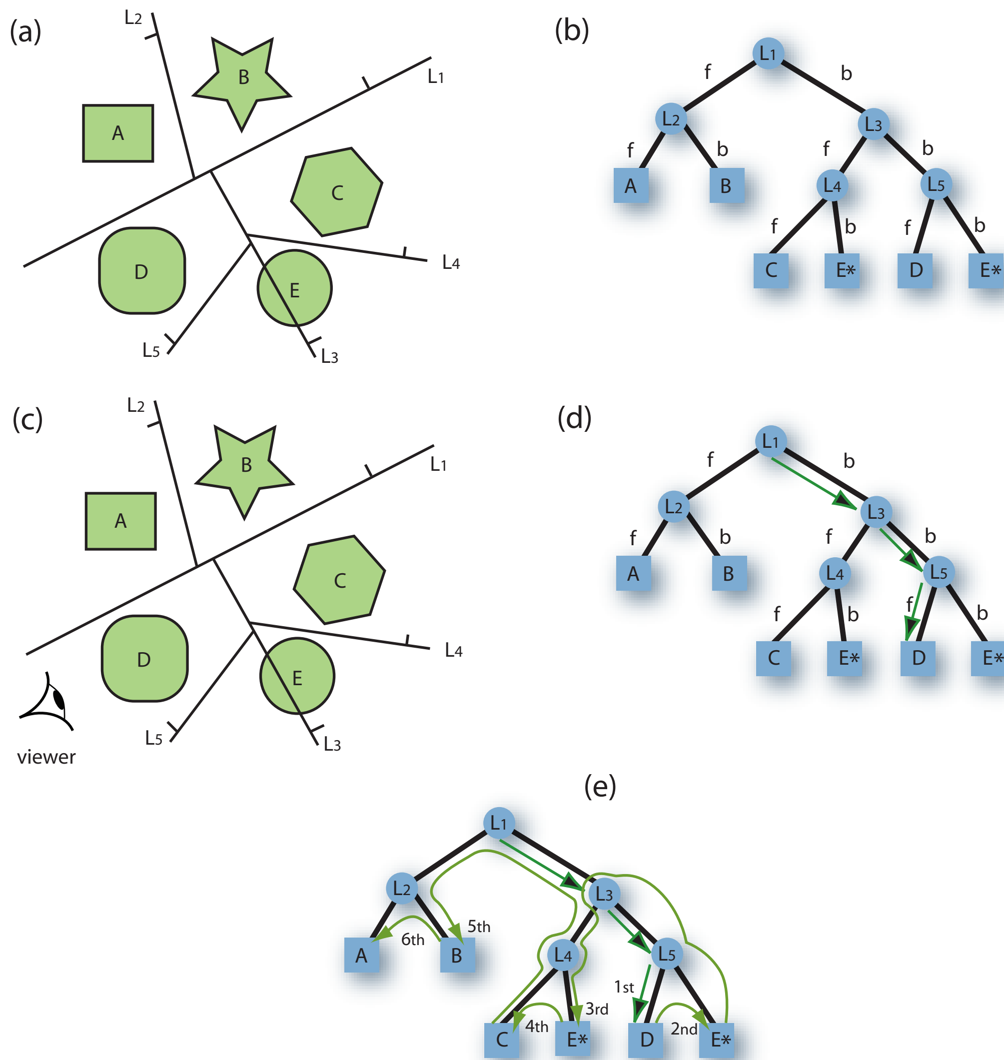



Figure 2.13: An illustration of the operations on a BSP Tree; (a) illustrates a simple scene which has been partitioned, (b) illustrates the creation of a BSP tree from (a), (c) shows a viewer entering the scene, (d) illustrates the algorithm for finding the closest object, and (d) illustrates a search to find the visibility order of all the objects.

back-facing, or all polygons. To take effect, the state variable for culling, `GL_CULL_FACE` must be enabled.

In OpenGL every polygon has a front and a back side. Each side can be drawn in the same mode or in different modes to allow cut-away views of solid objects. The command `glPolygonMode(GLenum face, GLenum mode)` controls the drawing mode for a polygon's front and back faces. The parameter face can be `GL_FRONT, GL_BACK`, or `GL_FRONT_AND_BACK`. The mode can be `GL_POINT, GL_LINE`, or `GL_FILL` to indicate whether the polygon should be drawn as points, outlined, or filled. By default, both the front and back faces are drawn filled:

```
glEnable(GL_DEPTH_TEST); // enable depth testing; required for z-buffer

glClear(GL_DEPTH_BUFFER_BIT); // clears the depth buffer
RenderScene();                // render the scene
```

Also, to draw the scene with back-face culling enabled (as it is more efficient), use:

```
glEnable(GL_DEPTH_TEST);      // enable depth testing; required for z-buffer
glEnable(GL_CULL_FACE);       // enable polygon face culling
glCullFace(GL_BACK);          // tell opengl to cull the back face

glClear(GL_DEPTH_BUFFER_BIT); // clears the depth buffer
RenderScene();                // render the scene
```

# Chapter 3

# Real-Time 3D Computer Graphics Techniques

## 3.1 Introduction

This chapter will introduce various 3D Computer Graphics Techniques that we can use to accelerate the systems that we build for real-time visualisation.

## 3.2 Texture Mapping in OpenGL

In 3D texture mapping a 2D image is attached to the surface of a polygon. This process can be invaluable in creating realistic scenes within gaming and virtual reality environments. Texture mapping has evolved over the years to include features such as bump mapping, where the surface normals can be irregular to give the impression of texture changes on an object.

OpenGL uses raw RGB image data for texture mapping. The thing you have to do before OpenGL can use the raw texture data is upload it to the video memory. Once a texture is uploaded to the video memory it can be used throughout the time in which your application is running.

Before a texture can be uploaded to the video memory there is some setup that must take place so OpenGL knows what to do with the image data that is passed to it. First we need a texture name. This is essentially a number that OpenGL uses to index all the different textures. To get a texture name, all we need do is call the function `glGenTextures()`.

```
0  GLuint texture;
   glGenTextures( 1, &texture );   // only one texture id
```

where `glGenTextures()` has the form `void glGenTextures( GLsizei n, GLuint *textures )` where `n` is the number of texture ids to provide and the second parameter specifies the array in which to store the ids (in my example here I only need one variable as I require only one texture).

Once the texture name is assigned we can then switch between our different textures using the `glBindTexture()` call - you can think of this in the same way that we switch colours for painting.

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

states that we wish to select our 2D texture that has the name `texture`. There are 1D and 2D textures but we are concerned with 2D textures here. Remember that this value is just an integer and we could probably have made the same call by something like `glBindTexture (GL_TEXTURE_2D, 0)`.

If we then wish to allow our texture to be mixed with colour for shading we can set the GL_TEXTURE_ENV_MODE value, where GL_MODULATE takes the colour and alpha value from the texture and multiplies it by the color data from our colour and lighting values:

```
glTexEnvf( GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE );
```

If the value parameter is GL_TEXTURE_ENV_MODE then the different texture functions that are defined are GL_MODULATE, GL_DECAL, GL_BLEND, GL_REPLACE, GL_ADD or GL_COMBINE. The default value is GL_MODULATE. Each of these values describes how the texture and colour/light values should be combined: *Replace* replaces the original surface colour with the texture colour, removing any lighting computed for the surface (unless the texture has provided it); *Decal* is like replace as the image is blended with the surface colour value but the $\alpha$ value is not modified (in Ireland we called them transfers - see through stickers); and, *modulate* is where we multiply the texture by the surface colour giving a shaded textured surface.

The **glTexParameteri** then sets the various parameters for this OpenGL texture. These parameters allow us to describe how the texture should appear on the objects that we are texture mapping. For example we could do something like:

```
    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

**glTexParameteri()** is where we can decide on effects such as texture filtering and mipmapping. We also can setup whether the texture wraps over at the edges or is clamped at the ends.

In this example above the first two lines of the example states that we are going to repeat the texture over our mapped object. In each case we are working with GL_TEXTURE_2D and the first example GL_TEXTURE_WRAP_S is the value that we are about to set - in this case we wish the texture to repeat over the mapped object so we use GL_REPEAT but an alternative would have been GL_CLAMP. This is only being performed for the $s$ coordinates, which describes the '$x$' coordinate of the texture. GL_CLAMP will clamp the $s$ coordinates to the range $[0, 1]$ and will prevent artifacts. When we use GL_REPEAT the integral value is ignored causing the texture to repeat over and over again. GL_TEXTURE_WRAP_S is by default set to GL_REPEAT. We repeat this again for the $t$ coordinates on the next line.

The next lines set the GL_TEXTURE_MAG_FILTER and GL_TEXTURE_MIN_FILTER values. When OpenGL maps a texture to a surface it is unlikely that one pixel on a texture (often called a texel) is mapped to one pixel in the screen space. It is possible that this texel will be mapped to several screen pixels, or that several texels will be mapped to one screen pixel. This value allows you to control how OpenGL magnifies or minifies a texel. We can set the texture magnification function to either GL_NEAREST or GL_LINEAR; and, we can set the texture minification filter to one of GL_NEAREST, GL_LINEAR, GL_NEAREST_MIPMAP_NEAREST, GL_NEAREST_MIPMAP_LINEAR, GL_LINEAR_MIPMAP_NEAREST and GL_LINEAR_MIPMAP_LINEAR. Each one of this values has a different trade off between performance and viewing quality.

We need to enable and select textures for our scene. We can turn on/off texture mapping by calling glEnable() or glDisable() and passing the GL_TEXTURE_2D value to each.

```
        glEnable(GL_TEXTURE_2D);
```

Finally, we need to specify the texture coordinates that describe the coordinates on the texture itself and how it should be mapped to the object in our scene. In OpenGL we use the $s$ and $t$ coordinates for this description and where a texture image is described in the range $[0, 1]$ but going outside of this range is possible where the texture is repeated or

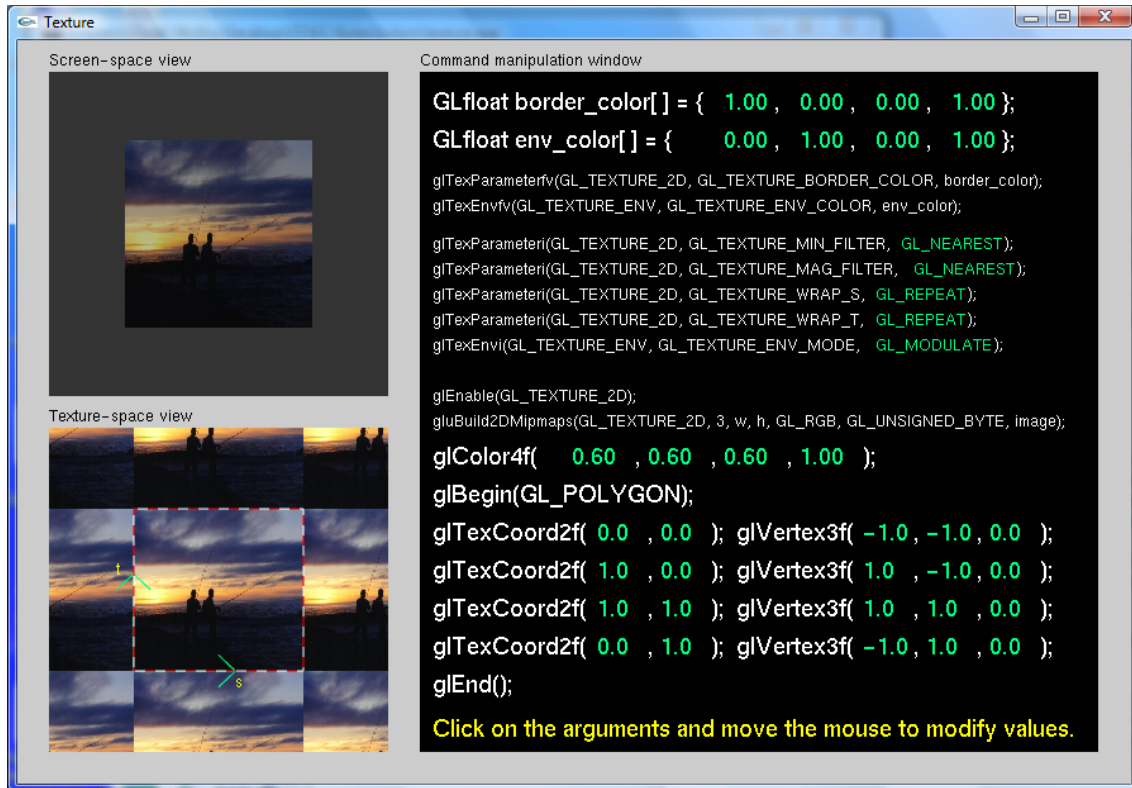



Figure 3.1: The Nate Robins' Tutorial on Texturing in OpenGL (`texture.exe`). You can right click the various parameters to change their values.

clamped. For a repeated texture the values $[1, 2]$ will describe the exact same image. So for example:

```
glBegin( GL_QUADS );
        glTexCoord2d(0.0f,0.0f);  glVertex2d(0.0f,0.0 f);
        glTexCoord2d(1.0f,0.0f);  glVertex2d(1.0f,0.0 f);
        glTexCoord2d(1.0f,1.0f);  glVertex2d(1.0f,1.0 f);
        glTexCoord2d(0.0f,1.0f);  glVertex2d(0.0f,1.0 f);
glEnd();
```

Will map the entire texture image to a single polygon.

There is a good tutorial on texture mapping in the Nate Robins' tutors set. Figure 3.1 illustrates this tutorial in action.

In the case of the polygon above we specified the texture coordinates for each vertex. This is not too difficult as there are only four texture coordinates for the polygon and they align with those of a standard image. It can become more difficult when the texture coordinates are on more complex objects. On a sphere with 36 subdivisions in longitude and latitude for example we could divide the image into these sections and calculate how these relate to the 2D texture image, for example breaking the image into a 36 x 36 grid. A more straightforward way to map the image is to use the `glTexGen*()` methods that automatically calculate the texture coordinates for the object.

Figure 3.2, illustrates a textured cube and a textured sphere in this example. The cube has been textured by providing the texture coordinates at each vertex and the sphere has used automatic calculation of texture coordinates. The code for texturing the sphere is as follows:

```
void Primitive::drawSphere()
{
        GLfloat step = 360.0/divisions;
        GLfloat radius = size/2;
```

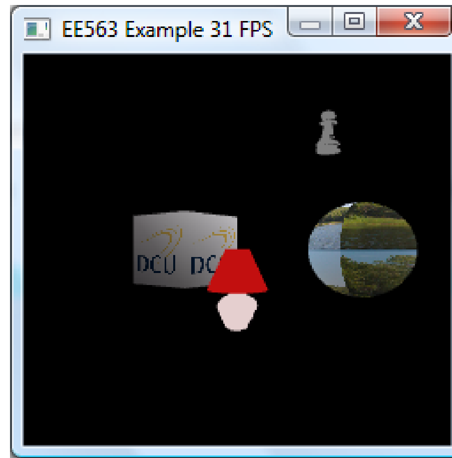



Figure 3.2: Texture Example 7 illustrating a textured cube and a textured sphere.

```
    GLfloat x,y,z, c = 3.14159f/180.0f;

    float splaneCoefficients [] = {0.25, 0, 0, 0};
    float tplaneCoefficients [] = {0, 0.5, 0, 0};

    // can change to GL_EYE_LINEAR and GL_SPHERE_MAP
    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
    glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
    glTexGenfv(GL_S, GL_OBJECT_PLANE, splaneCoefficients);
    glTexGenfv(GL_T, GL_OBJECT_PLANE, tplaneCoefficients);
    glEnable(GL_TEXTURE_GEN_S);
    glEnable(GL_TEXTURE_GEN_T);

    for (GLfloat phi=-80.0f; phi<80.0; phi+=step)
    {
        ...
        for (GLfloat theta=-180.0f; theta<=180.0f; theta+=step)
        {
            ...
            glVertex3f(x,y,z);
            ...
            glVertex3f(x,y,z);
        }
        glEnd();
    }

    // Close one end

    GLfloat closeRing_rad = c * 80.0;
    glBegin(GL_TRIANGLE_FAN);
      ...
    glEnd();

    // Close the other

    glBegin(GL_TRIANGLE_FAN);
      ...
    glEnd();

    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);
}
```

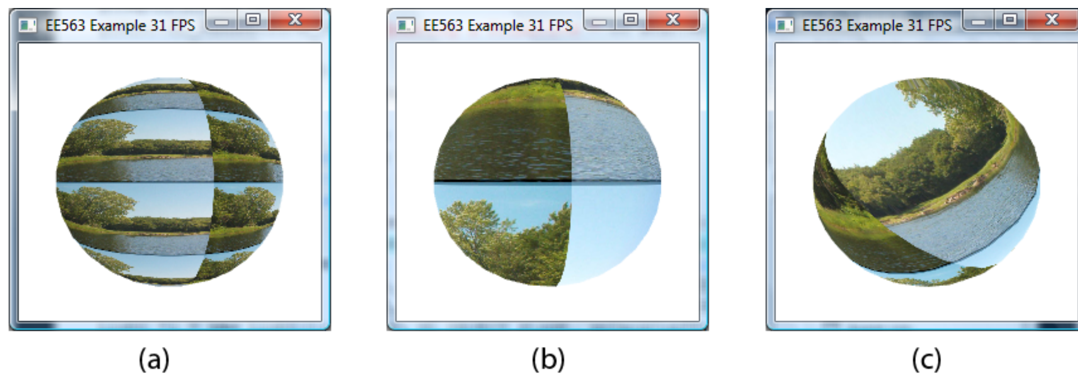



Figure 3.3: Examples of the modifications of the s plane and t plane coefficients: In (a) we have $s = [0.25, 0, 0, 0]$ and $t = [0, 0.25, 0, 0]$, (b) we have have $s = [0.125, 0, 0, 0]$ and $t = [0, 0.125, 0, 0]$ and (c) we have $s = [0.125, 0.125, 0, 0]$ and $t = [0, 0.125, 0.125, 0]$.

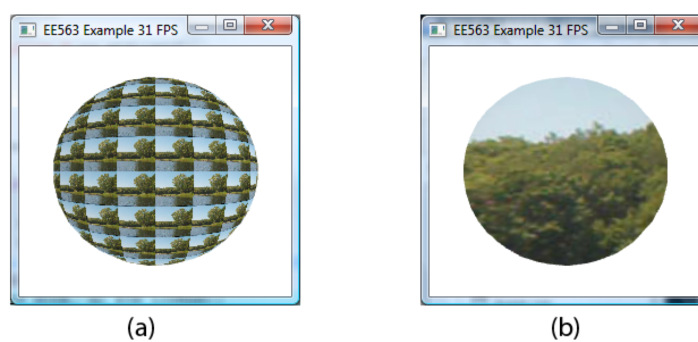



Figure 3.4: Examples of the modifications to the texture generation mode. Figure 3.3(b) illustrates the situation when the `GL_TEXTURE_GEN_MODE` is `GL_OBJECT_LINEAR`: In (a) we change this to `GL_EYE_LINEAR` and in (b) to `GL_SPHERE_MAP`.

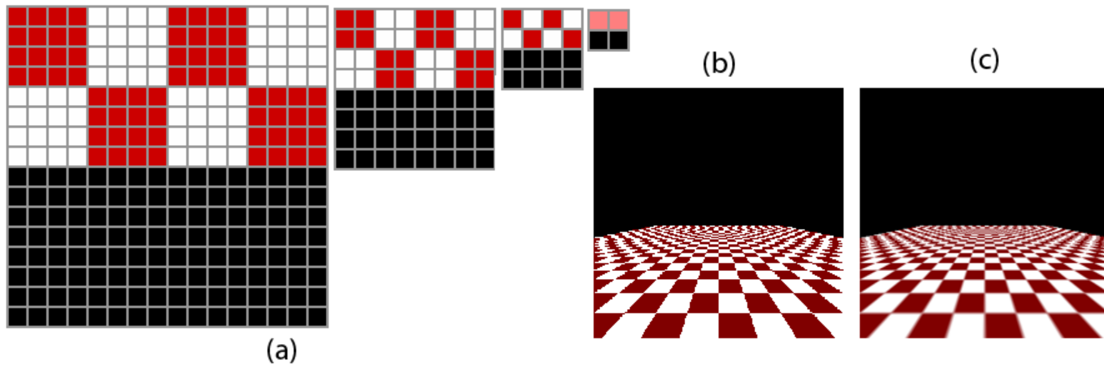



Figure 3.5: Mipmapping examples: (a) illustrates several mipmaps of the same texture, starting with a 16x16 size, reducing to 8x8, 4x4 and a 2x2 version. (b) is not using mipmaps and (c) is using mipmaps.

### 3.2.1 Mipmapping

When a scene is rendered in OpenGL textured objects are drawn at varying distances from the viewpoint, some close and some distant. As a textured object moves away from the viewer the texture must be scaled down so it can be applied to the smaller object. The problem that arises from scaling the texture down is that as objects move further away from the viewpoint the texture map may flicker due to this filtering. The solution to this problem is mipmapping. Mipmapping is the process by which a set of scaled texture maps of decreasing resolution are generated from a single high resolution image and used to improve accuracy during texture mapping. Each individual mipmap is a distinct texture that looks like the original but is a down scaled version of the previous mipmap level. Mipmapping allows OpenGL to apply the appropriate level of detail for a texture when rendering rather than involving shrinking the original image.

To generate and use these mipmaps we first generate each mipmap by taking half the size of the previous mipmap and then scaling it. If we have a base image that is 64x64 in size, the lower levels of detail will be 32x32, 16x16, 8x8, 4x4, 2x2 and 1x1. There are two ways to calculate the mipmaps and use them in OpenGL: by hand, or through the `gluBuild2DMipmaps()` function. If mipmaps are generated using your own algorithm, the `glTexImage2D()` function would be used with an increasing level parameter for each successive level of detail, with level 0 being the base texture level. More often than not, the `gluBuild2DMipmaps()` utility function will be used to generate a set of mipmaps for a given texture. `gluBuild2DMipmaps()` will take care of scaling and uploading the mipmaps to memory. This utility function is the same as `glTexImage2D()`, except for the fact that it builds mipmaps.

Once the different levels of detail have been generated they can be used to improve the quality of a textured object as OpenGL will automatically select the best level of detail to use for the texture without the involvement of the programmer. The mipmap that is selected can be defined with the `glTexParameter()` function. The trade off with mipmaps is that they will require additional texture memory and some extra computation must be made to generate them.

## 3.3 Level of Detail (LOD)

The Level of Detail concept is that we can use more simple version of an object to represent that object when it is distant from the camera/viewer. Why would we do this? Well, one reason would be to preserve our frame rate... more triangles generally equates to a lower frame rate. For example, if we were representing a person in a scene who was made up of 50,000 triangles, it would be possible to represent that person with a lot less triangles

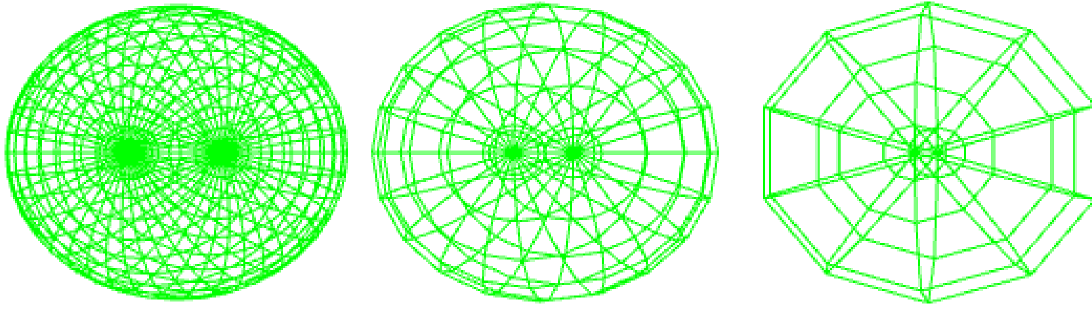


Figure 3.6: LOD Example of a sphere with 40, 20, and 10 subdivisions

if they were only occupying 20-30 pixels of screen space - perhaps we could represent the distant person by 1,000 triangles or less. A dodecahedron that we used before looks like a sphere from a large enough distance and so can be used to replace it so long as it is viewed from this or a greater distance. However, if it must ever be viewed more closely, it will look like a dodecahedron. If we decide to use the dodecahedron to stand in for a sphere then as the dodecahedron travels closer to the camera we must swap the dodecahedron with a higher detailed sphere.

It is also possible to use other techniques such as fog to limit the depth of field and to allow us to ignore objects when we know that they will not be visible due to the foggy conditions. It is also more representative of the real-world, where a slight fog/haze limits our viewing depth. In general LOD algorithms consist of three separate parts, which are generation, selection and switching.

*Generation*: One mechanism to generate different LOD models of the same object would be to do it manually by hand. However, the quality of the different models could lead to noise when the different models are switched. Automated polygonal simplification tools can be used for levels of detail generation. They removes primitives from an original mesh in order to produce simpler models which retain the key visual characteristics of the original object. The result is typically a series of simplifications (as shown in 3.6), which can be used in various conditions, such as in order to maintain a constant frame rate. A good balance can be determined between the richness of the models and the time it takes to display them.

An alternative to LOD generation in certain cases is to use procedural modelling, which allows real-time detail to be added to a model as it is approached. One example of this is the use of fractals to model terrain, trees, plants within a scene. Such models are generated on-the-fly according to our procedural algorithm.

*Selection* is when we select which LOD is appropriate based on some criteria, such as distance from the object or the area that the object will appear on the screen. A metric, called the benefit function, is evaluated for the current viewpoint and the currently location of the object and the value of this function allows us to choose an appropriate LOD. There are different benefit functions, such as range-based that uses a user-defined distance value to the object to decide if the object is visible and to what level of detail. An alternative approach is to use a project area-based LOD selection that often uses a bounding box to determine the screen space coverage.

*Switching* is when we change from one LOD to another and is prone to sudden changes (called popping). The most straightforward switching mechanism is Discrete Geometry LOD where the different representations of the same object contain different numbers of primitives. The different LODs can be stored as indexed triangle strips and when required the higher/lower level LOD is simply swapped into the next rendered frame. An alternative approach is to use a Blend LOD which performs a blend operation over a number of frames

to smooth the transition. It is however much more computationally expensive.

One LOD implementation could be as follows:

```
class Range
{
        float range;
        SceneObject* child;
                  public:
        Range(SceneObject* s, float r) { child = s; range = r; }
        float getRange() { return range; }
        SceneObject* getChild() { return child; }
};

class LOD : public SceneObject
{
    private:
      std::vector<Range> *rangeList;
      SceneObject *selectedChild;

    public:
      LOD();
      virtual ~LOD();
      virtual RTTI_OBJECT_TYPE getType() const { return RTTI_LOD; }
      void addChild(SceneObject* child, float range);

      // Force every child to have a render and update methods
      virtual void render(float timeElapsed);
      virtual void postRender(float timeElapsed);
      virtual void update(float timeElapsed);
      virtual void switchChild(float distance);
};
```

Where we have a Range class that contains a single scene object with a single range value, i.e. the model that should be used at this range. The LOD class then contains a vector of Range objects, again each object with its associated range value. The LOD class is a child of the SceneObject class so that it can be added as usual to the scene graph. There is a slight problem though - if we just add the different child objects as per usual then how will the recursive tree parsing algorithm know which child to render and which to ignore. It won't, so for this (my) implementation we will have the LOD children separated from the recursive tree parsing algorithm. This will allow the LOD object to still have its own non-LOD children. Importantly, this is only one possible implementation and it may not be suitable for all scene graph designs.

The implementation of the addChild() and switchChild(float) methods are as follows:

```
void LOD::addChild(SceneObject* child, float range)
{
    Range r(child, range);
    rangeList->push_back(r);
    selectedChild = child;
}

void LOD::switchChild(float distance)
{
    float curDist = 999999.0f;
    for(int i=0; i<rangeList->size(); i++)
    {
        // find object with smallest r that is greater than distance
        Range range = rangeList->at(i);
        float r = range.getRange();
        SceneObject *o = range.getChild();
```

```
        if ((r <= curDist) && ( r >= distance))
        {
            curDist = r;
            selectedChild = o;
20      }
    }
}
```

The addChild() method over-rides the parent addChild() method, but has different arguments. This is due to the different organisational structure of the scene graph, allowing the child object being added to the RangeList vector. The default selected child is the last child that was added to the rangeList. The switchChild(float) method switches to the child that is the most appropriate for the current distance between the viewer (camera) and the LOD object. This method implementation iterates through the rangeList vector and finds the object with the smallest range that is closer than the distance between the viewer and the LOD object.

## 3.4 Billboarding

Billboarding is a technique by which we can render an image onto a polygon (the billboard) that is always facing the viewer. As the view changes the polygon orientation changes, but the position remains the same (unless it changes for some other reason). Billboarding can be used to represent effects such as smoke, fire, explosions, clouds etc. The data structure of a billboard is a polygon (usually a quadrilateral), a surface normal and an up direction. These two vectors (surface normal and up direction) are sufficient to describe the rotation required to place the polygon at its required orientation. The position of the billboard is then described by a single point - possibly the centre of the polygon.

It would be usual for the surface normal $\vec{n}$ and the up vector $\vec{u}$. One of these vectors must be maintained in the given direction. To create an orthonormal basis for the rotation matrix we have to establish a set of three mutually perpendicular vectors to orient the billboard. We have the normal direction $\vec{n}$ and the up vector $\vec{u}$ and can use these to calculate a 'right' vector by getting the cross-product of $\vec{r} = \vec{n} \times \vec{u}$. But, because the $\vec{n}$ and the up vector $\vec{u}$ are not necessarily orthogonal we can then use the new $\vec{r}$ vector with the $\vec{n}$ vector to calculate the third orthogonal basis vector, using: $\vec{u'} = \vec{r} \times \vec{n}$.

The new up vector can then be used to form a rotation matrix with the form: $M = (\vec{r}, \vec{u'}, \vec{n})$. A translation matrix can then be used to place this polygon in the required location. We still need to determine the method of calculating the up vector and surface normal for the billboard. There are several different types of billboard techniques:

- $S$creen-aligned billboard - The polygon that is used for the billboard is always parallel to the screen, the billboard's normal vector is the negative of the view plane normal, and it has a constant up vector which is the same as the camera's up vector.

- $W$orld-oriented billboard - Due to the nature of perspective projection objects that are away from the view direction axis appear warped. This appears quite regular; however, it does not work well for screen-aligned billboarding as it does not distort correctly. World-oriented (viewpoint) billboarding makes the desired normal equal the vector from the centre of the billboard to the viewer's position and will distort the billboard the same way as real geometry does.

- $A$xial billboard - Commonly used to represent trees axial billboarding has a fixed world up vector in line with the trunk of the tree, but the texture normals do not face directly at the viewer, rather rotate around the trunk to align itself so as to face the viewer as much as possible.

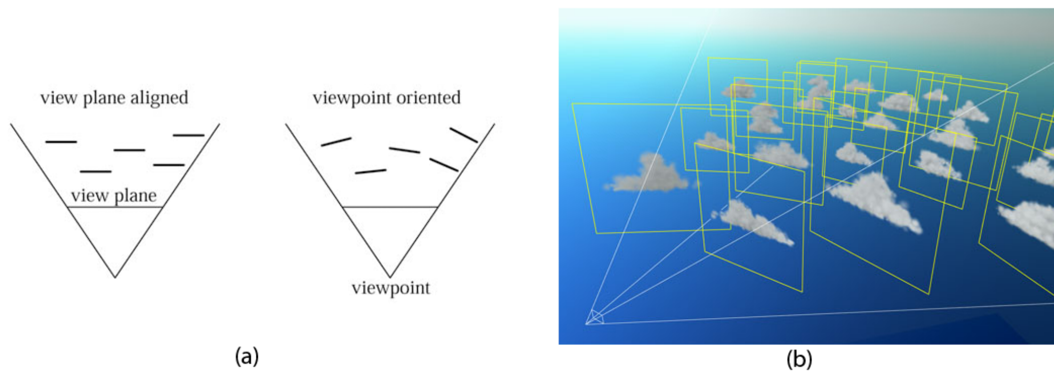



Figure 3.7: Billboarding example: (a) illustrates screen aligned and viewpoint oriented billboarding and (b) illustrates the use of billboards

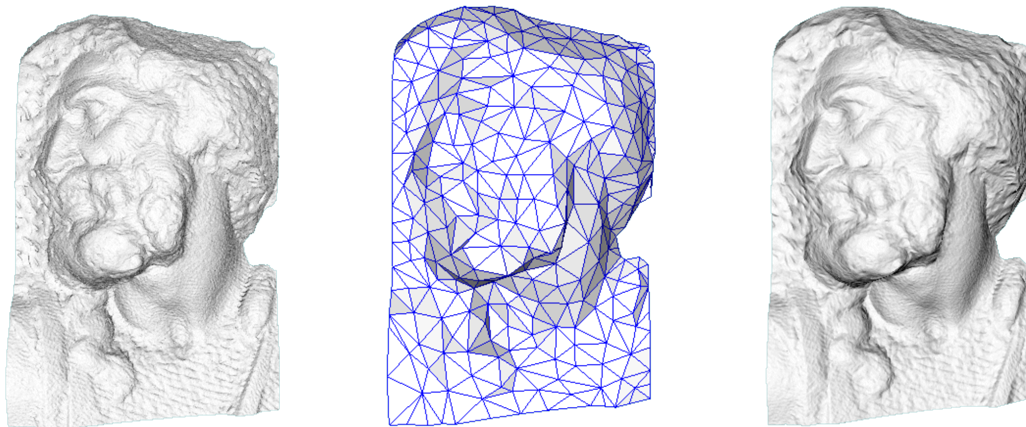


Figure 3.8: Bump Mapping Example (a) Original Model (b) Simplified Mesh (c) Result when using Simplified Mesh and Bump Mapping where the shape detail itself is enhanced. Each texel of the bumpmap contains some information about the physical shape of the object at the corresponding point.

- *I*mpostors - A billboard that is rendered to stand in for a rendered complex object, which is then mapped onto the billboard. They can be used to display rendered buildings in the distance, or to fill in for a few frames when the viewpoint has not changed much.

## 3.5 Mappings

### 3.5.1 Bump Mapping

Bump Mapping was introduced by Jim Blinn in 1978 and was originally applied to grayscale images. With 'bump mapping' a perturbation to the surface normal of the object being rendered is looked up in a height map and applied before the illumination calculation is performed, i.e. bump mapping means that you take the surface normal from a surface texture. The result is a richer, more detailed surface representation that more closely resembles the details present in the natural world. Bump mapping is also referred to as per pixel lighting, a technique that allows the programmer to specify a surface normal for each pixel rather than just the geometric primitive (which will usually have many pixels associated with it). Bump mapping provides us with a way of creating the illusion of complex geometry without having to create and process complex geometry. Interestingly, if we examine the specifications for the computer graphic card GPUs you will see that the

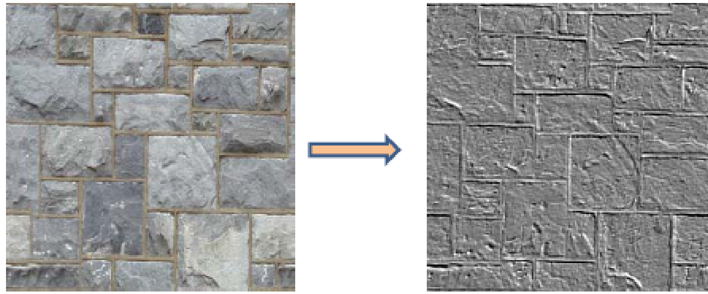


Figure 3.9: Emboss Bump Mapping

number of texels (texture elements) per second possible is roughly an order of magnitude larger than the number of vertices per second possible. Figure 3.8 demonstrates the use of bump mapping on a simplified mesh to illustrate that a similar level of detail can still be represented on a greatly simplified geometric object. The visual illusion of the presence in the surface of small bumps, holes, irregularities, carvings, engraving, scales, and so on; if managed efficiently, this can be achieved at a very small fraction of the rendering time that would be necessary for an object with similar characteristics modelled as geometry. Bump mapping does have drawbacks - in particular, bump mapping will lead to poor rendering of the silhouette as the object will still have a smooth geometry (rather than the perceived bumpy surface). There are two main techniques for Bump mapping:

- Emboss bump mapping
- Normal Mapping

Emboss bump mapping uses texture maps to generate bump mapping effects without requiring a custom renderer. This multi-pass algorithm is an extension and refinement of texture embossing. The algorithm itself consists of: Render the image as a texture; Shift the texture coordinates at the vertices; and then Re-render the image as a texture, subtracting from the first image. In effect it duplicates the first texture, shifts it over the desired amount of bump, darken the texture underneath, cut out the appropriate shape from the texture on top, and blend the two textures into one. This is called a two-pass emboss bump mapping because it requires two textures. It is fairly simple to implement, but is applied to the diffuse lighting only and so displays poorly from certain angles. It is a good choice for planar surfaces (like the bricks in figure 3.9) and for 3D text displays.

Both bump maps and normal maps work by modifying the normal angle. Where Bump maps are textures that store an intensity, the relative height of pixels from the viewpoint of the camera (See figure 3.9) - Normal maps are images that store a direction, the direction of normals directly in the RGB values of an image (See figure 3.10)

Normal maps are also generated from heightmaps. It is the most common bump mapping used today. Per pixel normals are represented in tangent space using a second RGB texture (like on the RHS of figure 3.10).

However, we are combining (one representing normals, the other a conventional texture) in a more sophisticated way. The tangent space refers to a co-ordinant system that is relative to the current plane. That is, in tangent space, the normal vector (0,0,1) is coincident with the normal of the geometry on which we are placing the the bump map.

You can think of bump maps as perturbing the direction of the geometric normal at each pixel. The per pixel normals are encoded in an RGB texture map as described by table 3.1.

To do bump mapping use an OpenGL 1.4 feature called texture combiners which provide a texture environment for more sophisticated mixing of textures. Texture combiners allow multiple texture fragments to be combined. By default, you can simply choose one of the texture environment modes (GL_DECAL, GL_REPLACE, GL_MODULATE, or

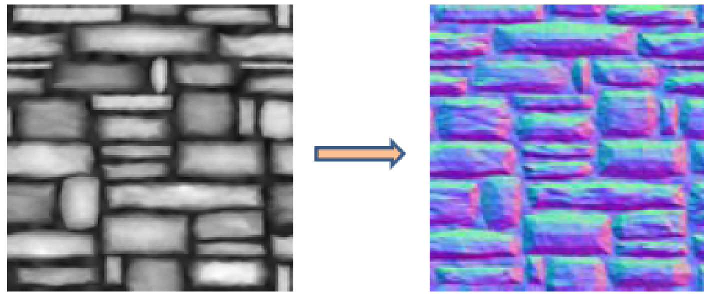


Figure 3.10: Normal Bump Mapping

| RGB Encoding | Tangent Space | Comment |
|---|---|---|
| 127 127 127 | 0 0 0 | The origin |
| 127 127 255 | 0 0 1 | Straight up in the direction of geometric normal |
| 127 127 0 | 0 0 -1 | Straight down, opposite of the geometric normal |
| 0 127 127 | -1 0 0 | -X direction in the plane |
| 255 127 127 | 1 0 0 | +X direction in the plane |
| 127 0 127 | 0 -1 0 | -Y direction in the plane |
| 127 255 127 | 0 1 0 | +Y direction in the plane |

Table 3.1: Encoding Texture Maps

GL_ADD) for each texture unit, and the results of each texture application are then added to the next texture unit. Texture combiners add a new texture environment, GL_COMBINE, that allows you to explicitly set the way texture fragments from each texture unit are combined. To use texture combiners, you call the glTexEnv function in the following manner:

```
0 glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);
```

Texture combiners are controlled entirely through the glTexEnv function. You need to select which texture combiner function you want to use, which can be either GL_COMBINE_RGB or GL_COMBINE_ALPHA. The third argument is the texture environment function that you want to employ (for either RGB or alpha values). For example, to select the GL_REPLACE combiner for RGB values, you would call the following function:

```
0 glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_REPLACE);
```

This combiner does little more than duplicate the normal GL_REPLACE texture environment. The following steps are required to perform bump mapping:

- Set up texture environment for the bump map (texture combiners).
- Set up texture environment for a second texture (if desired).
- Disable lighting.
- Calculate the light direction at each vertex and transform into RGB tangent space at each vertex of the geometry.
- Colour the vertex with this colour.

Set up texture environment for bump map (texture combiners):

```
0 glActiveTextureARB(GL_TEXTURE0_ARB);
  glBindTexture(GL_TEXTURE_2D,mybumptex);
  glEnable(GL_TEXTURE_2D);
  glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,GL_COMBINE_EXT);
  glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_DOT3_RGB_EXT);
```

Calculate the Light Direction: After the geometry to be bump mapped has been transformed into place do:

```
0 glGetFloatv(GL_MODELVIEW_MATRIX, mv_mat)
  glGetLightfv(GL_LIGHT0, GL_POSITION, lp_vec)
  //gives us the position of the light in eye coordinates)
```

### 3.5.2 Displacement Mapping

Displacement mapping is a computer graphics technique in contrast to bump mapping that uses a texture map to cause an effect where the actual geometric position of points over the textured surface are displaced along the local surface normal, according to the value the texture function evaluates to at each point on the surface. It gives surfaces a greater sense of depth and detail, permitting self-occlusion, self-shadowing and silhouettes. It is computationally expensive due to the large amount of additional geometry.

## 3.6 Shadows

*Shading* is a process used in drawing for depicting levels of darkness on paper by applying media more densely or with a darker shade for darker areas, and less densely or with a lighter shade for lighter areas. It is the process of altering a colour based on its angle to lights and its distance from lights to create a photorealistic effect. In contrast, *shadowing* is the process of creating a shadow by testing whether a pixel is visible from the light source, by comparing it to a z-buffer or depth image of the light source's view, stored in the form of a texture.

Umbra is the darkest part of a shadow where the source of light is completely concealed by the occluding object. Penumbra is the region where only a portion of the occluding body is obscuring the light source (the outer part of the shadow).

### 3.6.1 Algorithms for Computing Shadows

#### z-buffer algorithm

This method follows directly from the idea that shadow points are hidden from light. In other words, shadows are hidden surfaces from the point of view of a light. If we pretend that the light point is the centre of projection (i.e. an eye point), we can render the scene from the light's point of view, using a Z-buffer to compute surfaces visible to the light. The Z-buffer resulting from this will record all of the points that are closest to the light. Any point that has a further value at a given pixel is invisible to the light and hence is in shadow. The algorithm involves [1]:

Pre-computing phase: (for each light source)

- Make light point the centre of projection.
- Calculate the transformation matrices.
- Transform object using light point matrices.
- Render object using z-buffer - lighting is skipped
- Save computed zbuffer depth info

Object rendering phase:

- Make eye point be centre of projection.

[1]This discussion on shadows is modified from: http://web.cs.wpi.edu/~matt/courses/cs563/talks/shadow/shadow.html

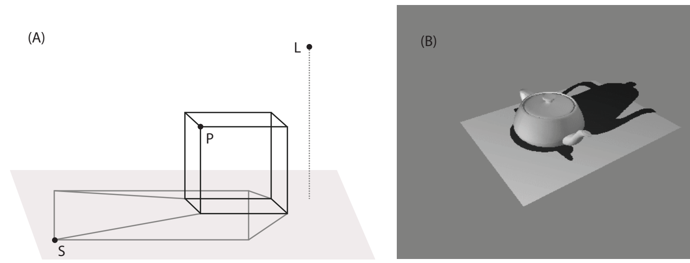



Figure 3.11: Shadows

- Recalculate transformation matrices.
- Transform object using eye point matrices.
- Render object using zbuffer.
- For every pixel visible from eye:
  - Transform world point corresponding to pixel to shadow coordinates.
  - For every light source:
    - Sample saved zbuffer for that light.
    - if shadow coordinates $\leq$ zbuffer value - pixel is in shadow

**Transforming Polygons to Ground Plane**

The equations for transforming a polygon onto the $z = 0$ plane, opposite the direction that the light is shining from. Jim Blinn describes two primary cases for calculating shadows where the light is at infinity or there is a local light source. Figure 3.11 (a) illustrates the general form of transformation of polygons to the ground plane and (b) shows an example output.

This method uses the geometric relationship of light sources and polygons, i.e. similar triangles, to calculate each polygon's projection onto the $z = 0$ plane. The shadow polygon generated in this way should be generated for every light source. So for $N$ lights there will be $N$ projections of each polygon.

This computes the projected shadow points of the polygon, which we can fill, producing a shadow polygon. Ray casting was introduced as a method for visibility calculation and shadow determination. A ray is cast from the eye to each pixel, ray surface intersections are performed and the surface with the minimum hit distance is declared the visible surface. This technique also models reflection and refraction and generates shadows. The principle behind ray tracing for hard shadow determination is very simple a shadow ray is shot from the intersection point to the light source. If the ray intersects any object between its origin and the light source then it is in shadow, otherwise it is not. Ray tracing requires no additional storage and pre-processing for shadow determination; it is evaluated as needed. However, shadow determination complexity is very expensive and uses no coherence information with a cost of $O(En)$ per ray shot, since ray surface intersections are required for all surfaces. The main advantage of ray tracing is that shadow determination is handled in the almost identical manner for cast, reflected and refracted rays. The ray tracing process is very floating point intensive. Thus, the visibility and shadow tests might give incorrect results due to numerical errors. This is usually seen as little holes in the images.

Pros and Cons of the two Algorithms:

The Ground Transformation Algorithm

- Requires no extra memory, and easily handles any number of light sources.

- However, it only shadows onto ground plane, so it cannot handle objects which shadow other complex objects.
- Every polygon is rendered $N$ times, where $N$ is number of light sources.

The Z-Buffer Algorithm

- Can shadow any scene which can be rendered using Z-buffer.
- However, it requires a separate memory buffer for each light source.
- Again, every polygon is rendered $N$ times, where $N$ is number of light sources, but $N - 1$ views do not need lighting calculations.

The Z-buffer algorithm is more versatile, with its ability to add shadows to scenes of arbitrary complexity. Also, the precomputed shadow buffers can be used to render views from any eye point as long as the relative positions of the lights and objects are constant between these views. However, if memory resources are limited, the ground transformation algorithm produces pleasing results if only ground shadowing is required.

# Chapter 4

# Appendices

## 4.1 Installing Dev C++

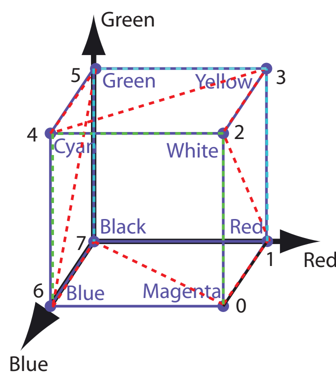
## 4.2 Exercise Solutions

### 4.2.1 Solution: A Rotating Colour Solid

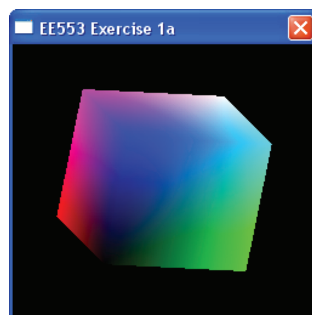
The exercise requested that you write the code to generate a rotating colour solid which demonstrates the range of RGB colours that has a variable diameter. Please use the structure as illustrated in figure 4.1, where (a) illustrates the colour cube, and (b),(c) show screen grabs of my solution. Figure 4.1(a) illustrates the solution used, where the red dashed lines go from 0 to 7 and back to 0 and 1 to draw a quad strip and the two faces are drawn by connecting nodes 0,2,4,6 and 1,3,5,7 as individual quads.

The long verbatim solution is to use something like:

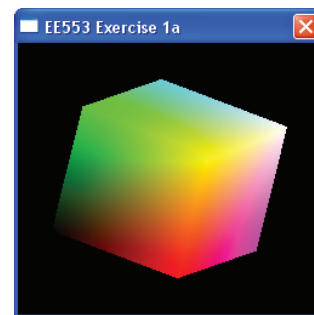
```
void drawGLColorCube1(GLfloat dia)
{
    // The long verbatim Solution

      GLfloat colours [8][3]  = {
            {1.0, 0.0, 1.0}, {1.0, 0.0, 0.0},  //magenta ,red
            {1.0, 1.0, 1.0}, {1.0, 1.0, 0.0},  //white, yellow
            {0.0, 1.0, 1.0}, {0.0, 1.0, 0.0},  //cyan ,green
            {0.0, 0.0, 1.0}, {0.0, 0.0, 0.0}   //blue, black
            };
      GLfloat vertices [8][3]  = {
            {-dia, dia,  dia}, {-dia, -dia, dia},
            { dia, dia,  dia}, { dia, -dia, dia},
            { dia, dia, -dia}, { dia, -dia, -dia},
            {-dia, dia, -dia}, {-dia, -dia, -dia}
            };

      //edges first
      glBegin(GL_QUAD_STRIP);
                  glColor3fv(colours [0]);
```




(a)




(b)




(c)

Figure 4.1: The Colour Cube Exercise (a) Colour Cube ;(b),(c) Screen Grabs of my solution.

```
20                      glVertex3fv( vertices [0]);

                        glColor3fv(colours [1]);
                        glVertex3fv( vertices [1]);

25                      glColor3fv(colours [2]);
                        glVertex3fv( vertices [2]);

                        glColor3fv(colours [3]);
                        glVertex3fv( vertices [3]);
30
                        glColor3fv(colours [4]);
                        glVertex3fv( vertices [4]);

                        glColor3fv(colours [5]);
35                      glVertex3fv( vertices [5]);

                        glColor3fv(colours [6]);
                        glVertex3fv( vertices [6]);

40                      glColor3fv(colours [7]);
                        glVertex3fv( vertices [7]);

                        glColor3fv(colours [0]);
                        glVertex3fv( vertices [0]);
45
                        glColor3fv(colours [1]);
                        glVertex3fv( vertices [1]);
                glEnd();

50      // then top
                glBegin(GL_QUADS);
                        glColor3fv(colours [0]);
                        glVertex3fv( vertices [0]);

55                      glColor3fv(colours [2]);
                        glVertex3fv( vertices [2]);

                        glColor3fv(colours [4]);
                        glVertex3fv( vertices [4]);
60
                        glColor3fv(colours [6]);
                        glVertex3fv( vertices [6]);
                glEnd();

65      //then bottom
                glBegin(GL_QUADS);
                        glColor3fv(colours [1]);
                        glVertex3fv( vertices [1]);

70                      glColor3fv(colours [3]);
                        glVertex3fv( vertices [3]);

                        glColor3fv(colours [5]);
                        glVertex3fv( vertices [5]);
75
                        glColor3fv(colours [7]);
                        glVertex3fv( vertices [7]);
                glEnd();
   }
```

which can be called by `drawGLColorCube(0.5f);`, but we could also introduce arrays of arrays as look up tables `strip`, which draws the four sides of the cube as a quad strip and `topandbottom` which draws two separate quads.

```
0  void drawGLColorCube2(GLfloat dia)
   {
      // The long verbatim Solution
       GLfloat colours [8][3]  = {
             {1.0, 0.0, 1.0}, {1.0, 0.0, 0.0},  //magenta ,red
5            {1.0, 1.0, 1.0}, {1.0, 1.0, 0.0},  //white, yellow
             {0.0, 1.0, 1.0}, {0.0, 1.0, 0.0},  //cyan ,green
             {0.0, 0.0, 1.0}, {0.0, 0.0, 0.0}   //blue, black
             };
       GLfloat vertices [8][3]  = {
10           {-dia, dia,  dia}, {-dia, -dia, dia},
             { dia, dia,  dia}, { dia, -dia, dia},
             { dia, dia, -dia}, { dia, -dia, -dia},
             {-dia, dia, -dia}, {-dia, -dia, -dia}
             };
15     int strip [10]  = {0, 1, 2, 3, 4, 5, 6, 7, 0, 1};
       int topandbottom[8] = {0, 2, 4, 6, 1, 3, 5, 7 };

       //edges first
       glBegin(GL_QUAD_STRIP);
20        for(int i=0; i<10; i++)
          {
            glColor3fv(colours[strip [i ]]);
            glVertex3fv( vertices [strip [i ]]);
          }
25     glEnd();
       // then top and bottom
             glBegin(GL_QUADS);
         for(int i=0; i<8; i++)
               {
30                      glColor3fv(colours[topandbottom[i]]);
                        glVertex3fv( vertices [topandbottom[i]]);
         }
       glEnd();
   }
```

There is a more advanced solution to this problem that uses Vertex Arrays, which allow us to encapsulate the information in our cube structure and draw it using only a small number of function calls. When using this approach it first has to be enabled using the `glEnableClientState()` function to enable the arrays that we wish to use. These arrays reside on the client side (rather than in the OpenGL state machine). We then must identify where the arrays are using the `glVertexPointer()` and `glColorPointer()` functions. The parameters in these functions specify (in order) the number of dimensions, the type of the data, that the elements are contiguous in memory and finally the pointer to the array of data. Finally we need to draw the elements themselves: Here we use `GL_POLYGON` to draw the six faces of the cube, where the vertex indices are given in the `cubeIndex` array. This array has 24 elements, broken into 6 groups of 4 vertices. The `glDrawElements()` function can be called to draw each face, where the parameters specify, the way to draw `GL_POLYGON`, the number of elements that we wish to draw (4 in the case of `GL_POLYGON`), the type of data contained in the index array and finally a pointer that points to the next index to use (in this case we wish to use 4 vertices then skip to the next 4 etc.).

```
void drawGLColorCube3(GLfloat dia)
{
                    // enable the arrays on the client side
    glEnableClientState(GL_COLOR_ARRAY);
    glEnableClientState(GL_VERTEX_ARRAY);

    GLfloat colours[8][3] = {
        {1.0, 0.0, 1.0}, {1.0, 0.0, 0.0},  //magenta ,red
        {1.0, 1.0, 1.0}, {1.0, 1.0, 0.0},  //white, yellow
        {0.0, 1.0, 1.0}, {0.0, 1.0, 0.0},  //cyan ,green
        {0.0, 0.0, 1.0}, {0.0, 0.0, 0.0}   //blue, black
        };
    GLfloat vertices[8][3] = {
        {-dia, dia,  dia}, {-dia, -dia,  dia},
        { dia, dia,  dia}, { dia, -dia,  dia},
        { dia, dia, -dia}, { dia, -dia, -dia},
        {-dia, dia, -dia}, {-dia, -dia, -dia}
        };

    // identify where the arrays are
    glVertexPointer(3, GL_FLOAT, 0, vertices);
    glColorPointer(3, GL_FLOAT, 0, colours);

    GLubyte cubeIndex[24] = { 0, 1, 3, 2,
                              2, 3, 5, 4,
                              4, 5, 7, 6,
                              6, 7, 1, 0,
                              0, 2, 4, 6,  //top
                              1, 3, 5, 7}; //bottom
    for(int i=0; i<6; i++)
    {
      // Draw the cube elements
      glDrawElements(GL_POLYGON, 4, GL_UNSIGNED_BYTE, &cubeIndex[4*i]);
    }
}
```

We can even make this section of code shorter; the creation of the arrays and the intialisation of the state do not need to be called each time the cube is rotated and drawn, hence it would be possible to place these lines of code in the intialisation phase of the program and only call the `glDrawElements()` function 6 times in the `drawGLColorCube3()` function. And to be perfectly correct as each face of the cube is a quadrilateral it is possible to use `GL_QUADS` rather than `GL_POLYGON` to draw the cube using this line of code:

```
            glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, cubeIndex);
```

because `GL_QUADS` starts a new quadrilateral after each four vertices.