Real-Time 3D Rendering Engine

By Uzair Hashmi

Project Supervisor: Dr Saeed Murtaza

Dissertation submitted as partial fulfillment of the requirements of MCS/NUST for the award of BE degree in Software Engineering

Department of Computer Sciences

Military College of Signals

Rawalpindi

May 2003.

Dedication

I humbly dedicate this dissertation and the research reported herein to my teachers at MCS, and particularly to Dr Saeed Murtaza for his encouragement. I also dedicate it to my parents.

Declaration

I declare that the research and development work reported in this thesis was performed solely for the purpose of the final year degree project and was not part of any other project.

Acknowledgement

First of all my thanks go out to Dr Saeed Murtaza for originally teaching me the mathematics (linear algebra) so essential to 3D graphics, and later on for providing lots of encouragement.

This project would not have been possible without the inspiration and insight I received from the pioneers of 3D graphics and 3D gaming, particularly John Carmack of id Software.

Contents

Part 1 - Introduction *7*

1.1 Style Conventions **7**

1.2 Aims and Objectives **8**

1.2.1 Basic Aim 8

1.2.2 Feature Set 8

1.3 Development Model **9**

1.4 Work Scheduling **9**

1.5 Introduction to 3D Graphics **9**

1.5.1 What it is about 9

1.5.2 Possibilities of 3D Graphics 10

1.6 Tools Used in Development **11**

1.6.1 Platform 11

1.6.2 Compiler 12

1.6.3 Graphics Hardware API 12

1.7 Reason for Choosing OpenGl as the Graphics API **12**

1.7.1 3D Graphics Cards 12

1.7.2 Contemporary Graphics API's 12

Part 2 – OpenGl and the Graphics Pipeline *14*

2.1 Objective **14**

2.2 Rendering Pipeline **14**

2.3 OpenGL as a State Machine **15**

2.4 OpenGL Rendering Pipeline **15**

2.4.1 Display Lists 16

2.4.2 Evaluators 17

2.4.3 Per-Vertex Operations 17

2.4.4 Primitive Assembly 17

2.4.5 Pixel Operations 18

2.4.6 Texture Assembly 18

2.4.7 Rasterization....19
2.4.8 Fragment Operations....19

Part 3 – eXtreme Engine Details....20

3.1 Overall Architecture....21

3.1.1 Renderer Pipeline....21

3.2 OpenGl Window Creation and Game Code....22

3.3 OpenGl Initialization....23

3.4 Loading Map Data....25

3.4.1 Date Structures used....25
3.4.2 Format of Map File....26
3.4.3 File Loading....26

3.5 Loading Textures....26

3.5.1 Texture Structure....26
3.5.2 Texture Loading....27

3.6 Drawing the Scene....27

3.7 Lighting....29

Lighting Normals....29

Part 4 – Appendices....32

Appendix A: In-Engine Screenshots....32

Appendix B: Mathematics behind the rendering pipeline transforms.... 2

Translation....2
Rotation....2
Perspective Projection....3

Part 1 - Introduction

This thesis was written as part of my final year project at Military College of Signals. The title of the project was **eXtreme Engine:** *a Real-Time 3D Rendering Engine.* This thesis will attempt to explain all aspects related to the development of the project, and information related to the field of 3D graphics.

This part serves as an introduction to the thesis. First the style conventions are enumerated. That is followed by stating the objectives, and then the scheduling of work is described.

1.1 Style Conventions

In this project report the following style conventions are used:

- The actual text of the report (other than the headings) is written in Palatino Linotype, 12 pt.
- Code is written in Courier New, 10 pt.
- OpenGl functions start with 'gl'.
- OpenGl constants start with 'GL_'

1.2 Aims and Objectives

1.2.1 Basic Aim

As defined by the title of the project, the basic aim was to produce a 3D graphics engine capable of producing realistic-looking visual environments in real-time. However, while this is the stated aim, the real advantage of developing and implementing such an engine is that it would (and has) enabled me to understand the workings of any modern 3D graphics engine. Following the incremental model, while the engine has been developed to the required goals, it can be extended to include more features. These features could be such that the engine can be used for producing a game, an architectural planner, or any number of applications which require the simulation of a 3D world in 2D. But while the application and utilities of the project will be studied in more detail later, the goals set at the time of commencing the project were to produce just such an engine which can be extended for any 3D application.

1.2.2 Feature Set

The engine has the following feature set:

- Triangle representation of the world
- Texture-mapping
- Lightmapping
- User movement through keyboard input

1.3 Development Model

In software engineering terms, a combination of the incremental and prototyping models was used in order to conduct the necessary research and develop the engine. First a simple system was built as a prototype, and then built again but with increased complexity, efficiency and size. Also, in learning the technology many small programs were built, and then the concepts and algorithms were incorporated in the engine code.

1.4 Work Scheduling

- Research: 8 weeks
- Simple BGI Engine: 2 weeks
- OpenGl technology research: 2 weeks
- Implementation: 4 weeks
- Testing and debugging: 2 weeks
- Documentation: 1 week

1.5 Introduction to 3D Graphics

1.5.1 What it is about

The following section is for those not familiar with the realm of 3D graphics. Its purpose is to acquaint the reader with what the term '3D graphics' is all about and what a vast field it encompasses.

In brief, the goal of 3D graphics is to simulate on a 2D surface any 3D object. This very definition opens up a vast range of possibilities. Not only can we simulate anything in the real world, but anything that can exist in our

imagination. This immediately opens up vast possibilities for movies and computer gaming.

Consider a photo. Sure it looks good. But we can look at only from a very specific angle, range, and lighting conditions. Using a 3D engine, we can render(that is, draw) that (almost as real) image from *any place we wish,* and by changing *anything* we want in the image.

Now take a movie. A movie is simply a collection of photos, appearing very swiftly to give the impression of motion. But nothing in the movie can be changed; once it has been shot, it remains that way forever. We cannot, if we wish, look around the corner to see if the monster is standing there or not; the information needed is simply not there. With a real-time engine, we don't have a series of photos, but the world itself is defined, and we draw the picture as it appears to the camera. This means that we can produce interactive movies. This is exactly what 3D games are: interactive movies in which the player is the 'hero' of the movie.

This type of drawing of pictures which appear so fast that it looks like a movie is called *real-time rendering*. The term real-time implies that the process of rendering is so fast, animation can be simulated. This of course is the same as TV, where 30 frames are flashed on the screen every second. Similarly a real-time engine tries to render its frames so fast the user cannot tell that they are separate.

1.5.2 Possibilities of 3D Graphics

The ability to simulate just about anything graphically opens a huge range of uses for 3D engines. A few of them are:

- CAD
- Scientific visualization
- Computer gaming
- Medical imaging
- Military training
- Architectural design and evaluation
- Flight simulators for pilot training
- User interface for operating systems of the future?

1.6 Tools Used in Development

1.6.1 Platform

Windows was chosen as the OS on which the engine would run because:

- It is the most widely used OS in the world, and hence any commercial application (such as one that could be based on this engine) would find a large market by supporting Windows.
- I have experience working/programming with Windows.

1.6.2 Compiler

Microsoft Visual C++ 6.0 was used.

1.6.3 Graphics Hardware API

OpenGl was chosen as the API for interacting with the graphics hardware.

1.7 Reason for Choosing OpenGl as the Graphics API

1.7.1 3D Graphics Cards

Realistic rendering of 3D images is a computationally expensive process containing a huge number of mathematical calculations. The demands of executing the rendering quickly are increased when the engine has to be real-time. To meet this demand special graphics hardware has been developed. These 3D graphics cards perform many of the mathematical functions required.

1.7.2 Contemporary Graphics API's

However the problem with 3D cards is that dozens of them exist, with different capabilities and more importantly for the graphics programmer, interfaces. To overcome this, two standard API's have evolved:

DirectX

DirectX is a Microsoft owned API. For this reason, it only runs on Microsoft's OS, Windows. It has absolutely no support on other platforms, such as Macintosh or Linux.

OpenGl

OpenGl (Open Graphics library) was developed by Silicon Graphics Interactive (SGI) in the early 1990's. It has implementations on a whole slew of platforms, including:

- Windows
- Linux and Unix
- Macintosh
- BeOS
- Amiga

among others (less obscure).

OpenGl is the API of choice in all professional applications such as CAD programs, modeling programs (like the very sophisticated 3D Studio Max) and for scientific visualization.

The one great advantage for me in choosing OpenGl instead of DirectX is simply that OpenGl is much simpler to learn and use as compared to DirectX. Also, a huge amount of resources are available for learning OpenGl on the Internet, as compared to DirectX.

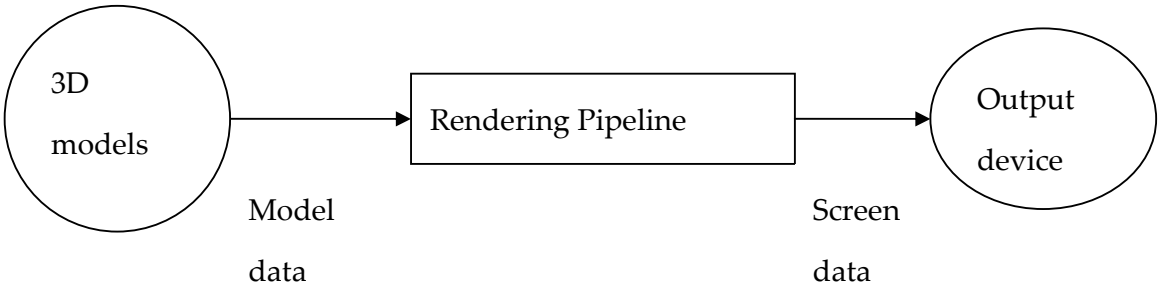
Part 2 – OpenGl and the Graphics Pipeline

2.1 Objective

This part of the report will serve as a brief tutorial to anyone not familiar with 3D technology and particularly OpenGl. The next part will be the main part of the report describing the engine developed.

2.2 Rendering Pipeline

As has been explained before, the process of creating the picture is called rendering. The whole process is very simply conceptualized as:



Thus the purpose of the rendering pipeline is simply to transform whatever data the 3D models contain into a 2D form for output on a display device. The

reason it is called a *pipeline* is that it actually consists of a number of stages, where data is sent down each stage, one after the other, somewhat like the flow of material through connected pipes forming a pipeline.

Before the advent of 3D graphics cards, the pipeline was implemented entirely in software code. But through an API like OpenGl the pipeline stages on the 3D card can be accessed.

2.3 OpenGL as a State Machine

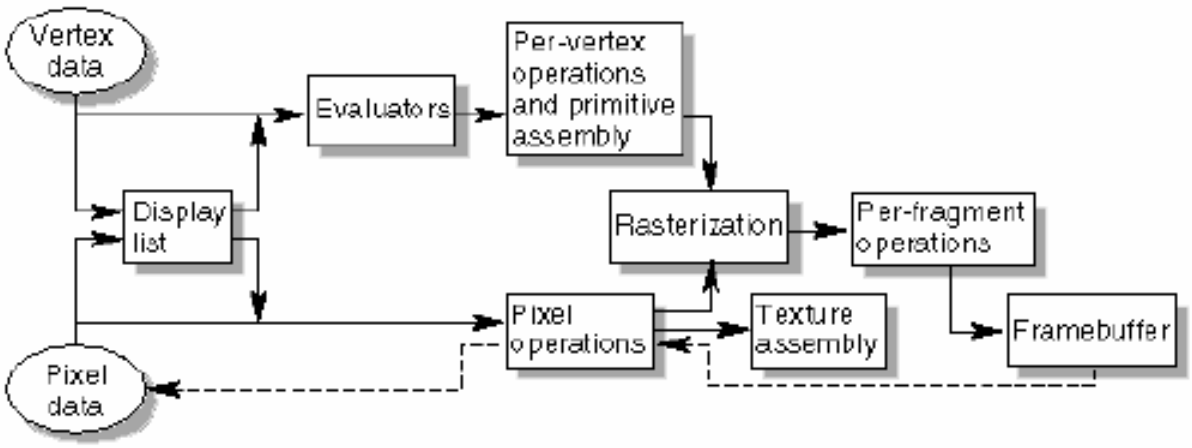
OpenGL is a state machine. It is put it into various states (or modes) that then remain in effect until they are changed. For instance, the current color is a state variable. The current color can be set to white, red, or any other color, and thereafter every object is drawn with that color until the current color is set to something else. The current color is only one of many state variables that OpenGL maintains. Others control such things as the current viewing and projection transformations, line and polygon stipple patterns, polygon drawing modes, pixel-packing conventions, positions and characteristics of lights, and material properties of the objects being drawn. Many state variables refer to modes that are enabled or disabled with the command **glEnable()** or **glDisable()**. Each state variable or mode has a default value, and at any point the system can be queried for each variable's current value. One of the six following commands is used to do this: **glGetBooleanv()**, **glGetDoublev()**, **glGetFloatv()**, **glGetIntegerv()**, **glGetPointerv()**, or **glIsEnabled()**.

2.4 OpenGL Rendering Pipeline

Most implementations of OpenGL have a similar order of operations, a series of processing stages called the OpenGL rendering pipeline. This ordering, as

shown in the figure below, is not a strict rule of how OpenGL is implemented but provides a reliable guide for predicting what OpenGL will do.

Geometric data(vertices, lines, and polygons) follow the path through the row of boxes that includes evaluators and per-vertex operations, while pixel data (pixels, images, and bitmaps) are treated differently for part of the process. Both types of data undergo the same final steps (rasterization and per-fragment operations) before the final pixel data is written into the framebuffer.



Order of Operation

2.4.1 Display Lists

All data, whether it describes geometry or pixels, can be saved in a *display list* for current or later use. (The alternative to retaining data in a display list is processing the data immediately - also known as *immediate mode*.) When a display list is executed, the retained data is sent from the display list just as if it were sent by the application in immediate mode

2.4.2 Evaluators

All geometric primitives are eventually described by vertices. Parametric curves and surfaces may be initially described by control points and polynomial functions called basis functions. Evaluators provide a method to derive the vertices used to represent the surface from the control points. The method is a polynomial mapping, which can produce surface normal, texture coordinates, colors, and spatial coordinate values from the control points.

2.4.3 Per-Vertex Operations

For vertex data, next is the "per-vertex operations" stage, which converts the vertices into primitives. Some vertex data (for example, spatial coordinates) are transformed by 4 x 4 floating-point matrices. Spatial coordinates are projected from a position in the 3D world to a position on your screen. If advanced features are enabled, this stage is even busier. If texturing is used, texture coordinates may be generated and transformed here. If lighting is enabled, the lighting calculations are performed using the transformed vertex, surface normal, light source position, material properties, and other lighting information to produce a color value.

2.4.4 Primitive Assembly

Clipping, a major part of primitive assembly, is the elimination of portions of geometry which fall outside a half-space, defined by a plane. Point clipping simply passes or rejects vertices; line or polygon clipping can add additional vertices depending upon how the line or polygon is clipped. In some cases, this is followed by perspective division, which makes distant geometric objects appear smaller than closer objects. Then viewport and depth (z coordinate) operations are applied. If culling is enabled and the primitive is a

polygon, it then may be rejected by a culling test. Depending upon the polygon mode, a polygon may be drawn as points or lines. The results of this stage are complete geometric primitives, which are the transformed and clipped vertices with related color, depth, and sometimes texture-coordinate values and guidelines for the rasterization step.

2.4.5 Pixel Operations

While geometric data takes one path through the OpenGL rendering pipeline, pixel data takes a different route. Pixels from an array in system memory are first unpacked from one of a variety of formats into the proper number of components. Next the data is scaled, biased, and processed by a pixel map. The results are clamped and then either written into texture memory or sent to the rasterization step. If pixel data is read from the frame buffer, pixel-transfer operations (scale, bias, mapping, and clamping) are performed. Then these results are packed into an appropriate format and returned to an array in system memory. There are special pixel copy operations to copy data in the framebuffer to other parts of the framebuffer or to the texture memory.

A single pass is made through the pixel transfer operations before the data is written to the texture memory or back to the framebuffer.

2.4.6 Texture Assembly

An OpenGL application may wish to apply texture images onto geometric objects to make them look more realistic Some OpenGL implementations may have special resources to accelerate texture performance. There may be specialized, high-performance texture memory. If this memory is available, the texture objects may be prioritized to control the use of this limited and valuable resource.

2.4.7 Rasterization

Rasterization is the conversion of both geometric and pixel data into *fragments*. Each fragment square corresponds to a pixel in the framebuffer. Line and polygon stipples, line width, point size, shading model, and coverage calculations to support antialiasing are taken into consideration as vertices are connected into lines or the interior pixels are calculated for a filled polygon. Color and depth values are assigned for each fragment square.

2.4.8 Fragment Operations

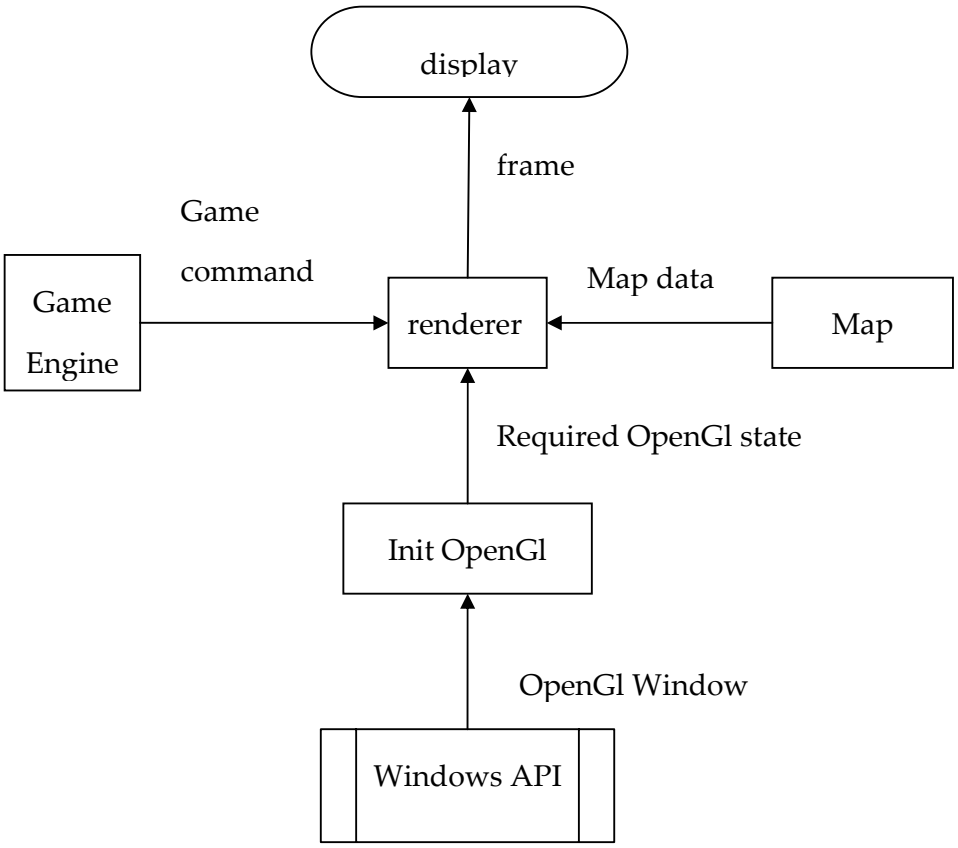
Before values are actually stored into the framebuffer, a series of operations are performed that may alter or even throw out fragments. All these operations can be enabled or disabled. The first operation which may be encountered is texturing, where a texel (texture element) is generated from texture memory for each fragment and applied to the fragment. Then fog calculations may be applied, followed by the scissor test, the alpha test, the stencil test, and the depth-buffer test (the depth buffer is for hidden-surface removal). Failing an enabled test may end the continued processing of a fragment's square. Then, blending, dithering, logical operation, and masking by a bitmask may be performed. Finally, the thoroughly processed fragment is drawn into the appropriate buffer, where it has finally advanced to be a pixel.

Part 3 – eXtreme Engine Details

This part is where I will explain all the workings of the engine that has been developed, along with the associated code.

3.1 Overall Architecture

The architecture of the engine can be seen in the figure below:



3.1.1 Renderer Pipeline

As mentioned before, it is the job of the renderer to draw each frame. The steps performed in the renderer are:

1. Transforms (rotation, translation, scaling)
2. Transform from World Space to View Space
3. View Projection
4. Trivial Accept/Reject Culling
5. Perspective Divide - Transform to Clip Space
6. Clipping
7. Transform to Screen Space

Each of these operations is performed by setting up the appropriate OpenGl states and calling the associated functions on the graphics hardware.

The following sections will describe each major module and its functions.

3.2 OpenGl Window Creation and Game Code

In many ways the setting up a window for the display in Windows was one of the more difficult tasks of the project, as it involves many steps related to Windows programming. Since the purpose of this report is to explain the principles underlying 3D graphics and those which underlie the engine I have developed I will not go into the details.

WinMain() is of course the first function that runs in the program. It calls win_CreateGLWindow(), where the actual window is created. Also, there is

win_KillGLWindow() function called when the program exits to properly kill the window.

WinMain() runs an infinite loop until the user presses escape or an error condition occurs. During each loop it calls r_main(), which is responsible for the actual drawing and is explained later.

WinMain() is also responsible for flipping the buffers. Two buffers are used, one in which the current scene is being drawn and another in which the previous one is displayed to the user. When the drawing is complete WinMain() interchanges the two buffers. This ensures that the user does not have to wait while the frame being rendered but instead watches a continuous collection of frames.

As the key input handling interface is inside WinMain(), therefore the game code responsible for player movement around the world has also been included in it. Moving forward or backward, for instance, changes the current player x and z positions. Also, to simulate a more realistic motion, 'view bobbing' is incorporated in the code for moving forwards or back. The sine curve is used to make it look like the player's head is bobbing up and down.

3.3 OpenGl Initialization

In order to properly run the engine, it is not just enough to create a window. As stated before OpenGl is a state machine which keeps track of a large number of state variables. The requisite state variables have to be set to proper values. This is done in the function r_Init(), called after the OpenGl window has been created.

r_init(), before doing any of its processing, loads the world and its specified textures, These functions will be explained later. Since each line is important at this stage, I will explain them all:

Since the engine uses texture mapping, it must be enabled:

```
glEnable(GL_TEXTURE_2D);
```

The next line sets the blending function:

```
glBlendFunc(GL_SRC_ALPHA,GL_ONE);
```

The background color should be black, so:

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
```

The Depth Buffer holds the z-values of each pixel in a frame. This is necessary for hidden surface removel. It is enabled by:

```
glClearDepth(1.0);
glDepthFunc(GL_LESS);
glEnable(GL_DEPTH_TEST);
```

For proper lighting smooth shading needs to be enabled:

```
glShadeModel(GL_SMOOTH);
```

glHint() tells OpenGL that we want the best perspective correction to be done. This causes a small performance penalty, but makes the perspective view look a bit better:

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
```

The next few lines actually deal with lighting setup and are explained later.

```
glLightfv( GL_LIGHT0, GL_AMBIENT,  ambience );
glLightfv( GL_LIGHT1, GL_DIFFUSE,  diffuse );
```

```
      glLightfv( GL_LIGHT0, GL_POSITION, LightPosition1 );
      glLightfv( GL_LIGHT1, GL_POSITION, LightPosition2 );

      glEnable(  GL_LIGHT0   );
      glEnable(  GL_LIGHT1   );

      glEnable(  GL_LIGHTING );
```

3.4 Loading Map Data

3.4.1 Date Structures used

First I'll explain the data structures used to describe objects to be rendered. At the lowest level there is a vertex of a triangle:

```
typedef struct vertex_struct
{
      float x, y, z;
      float u, v;
} tri_vertex;
```

The u and v coordinates are used in texture mapping to properly map a texture image to its associated triangle.

Triangles are used to represent the world. A triangle consists of an array of 3 vertexes and a an identifier *tri_texture* for the texture which is associated with this particular triangle.

```
typedef struct triangle_struct
{
      tri_vertex vertex[3];
      int tri_texture;
} triangle;
```

Finally a *model* is defined as consisting of a pointer to triangles and the number of triangles in the model. This is useful if models such as cars, humans etc are to be introduced.

```
typedef struct model_struct
{
      int numtriangles;
      triangle * tri;
```

```
} model;
```

3.4.2 Format of Map File

Based on the above definitions, the file was decided to have the format:

```
NUMTEXTURES 3
NUMTRIS 36

X1 Y1 Z1 U1 V1 tex1
X2 Y2 Z2 U2 V2 tex1
X3 Y3 Z3 U3 V3 tex1

X1 Y1 Z1 U1 V1 tex2
X2 Y2 Z2 U2 V2 tex2
X3 Y3 Z3 U3 V3 tex2
```

Each horizontal line thus contains the definition of one vertex.

3.4.3 File Loading

The file is loaded into model1 by the function InitWorld(). It first stores the number of triangles and number of textures. Then it creates model1, allocating memory dynamically based on the number of triangles. Reading one line (thus one vertex) at a time, it assigns values for x, y, z, u, v, and tri_texture to the triangles in model1.

3.5 Loading Textures

3.5.1 Texture Structure

```
typedef struct tex_Struct
{
      GLubyte     *image_data;
      GLuint      bits_per_pixel;
      GLuint      width;
      GLuint      height;
      GLuint      texture_id;
} Texture_Image;
```

This structure is very straightforward and self-explanatory. The image_data pointer points to actual place in memory where the image of the texture is

placed. texture_id is necessary because in OpenGl textures are bound by GLuint.

3.5.2 Texture Loading

InitTextures() allocates an array of Texture_Images, using the number of texes found in InitWorld() and saved in numtexes. Then it loops through each textures[] element, and calls BindTGA_toTexture() with the texture index and the name of the TGA file holding the texture.

BindTGA_toTexture() fills in the fields of textures[*i*], by opening the TGA file. I have decided to use TGA files because they are simple to work with (no compression) and because unlike bitmaps they support the alpha channel.

3.6 Drawing the Scene

This is most important part of the engine. It is here that the scene is drawn. OpenGl uses all the state variables defined previously in r_init in drawing the scene. r_init() is called from WinMain() repeatedly until the end of the program.

First the screen and depth buffers are cleared as this is a fresh frame:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

OpenGl maintains 4 x 4 modelview and projection matrices for drawing the scene. This function call resets the modelview matrix:

```
glLoadIdentity();
```

These floats are used to hold the value of each coordinate of each triangle:

```
GLfloat     vert1_x, vert1_y, vert1_z,
            vert2_x, vert2_y, vert2_z,
            vert3_x, vert3_y, vert3_z;
```

```
GLfloat     u_current, v_current;
```

These values hold the values by which the modelview matrix should transform the vertices:

```
GLfloat translate_x = -x_pos;
GLfloat translate_z = -z_pos;
GLfloat translate_y = -movement_bobbing-0.25f;
GLfloat rotate_y = 360.0f - y_rotation;

int numtriangles;
```

Thses commands work on the modelview matrix to specify the view:

Rotate up snd down to look up and down:

```
glRotatef(x_rotation,1.0f, 0, 0);
```

Rotate based on direction player is facing:

```
glRotatef(rotate_y, 0, 1.0f, 0);
```

Translate the scence based on played position:

```
glTranslatef(translate_x, translate_y, translate_z);
```

numtriangles is assigned the number of triangles in the model.

After the each triangle is processed:

It is bound to its associated texture in textures[] with:

```
            int tex_id;
            tex_id = model1.tri[tri_loop].tri_texture;
            glBindTexture(GL_TEXTURE_2D,
textures[tex_id].texture_id);
```

Then the vertices of the triangle and texture coordinates are defined between a glBegin() and glEnd pair, like:

```
glTexCoord2f(u_current, v_current);
glVertex3f(vert1_x, vert1_y, vert1_z);
```

However during this stage the engine also does lighting calculations, as explained in the next section.

3.7 Lighting

Lighting is a crucial part of any 3D engine. Without decent lighting no scene would look real, even it was very complex. To implement lighting 2 lights, an ambient light (coming from all directions) and a diffuse light (localized) are defined in r_init:

```
glLightfv( GL_LIGHT0, GL_AMBIENT,  ambience );
glLightfv( GL_LIGHT1, GL_DIFFUSE,  diffuse );
```

Their positions are specified here:

```
glLightfv( GL_LIGHT0, GL_POSITION, LightPosition1 );
glLightfv( GL_LIGHT1, GL_POSITION, LightPosition2 );
```

Each light is enabled, along with lighting:

```
glEnable(  GL_LIGHT0   );
glEnable(  GL_LIGHT1   );
glEnable(  GL_LIGHTING );
```

Lighting Normals

However for proper lighting OpenGl needs to know the normal to each triangle. Thus functions are defined in math_lib.c for calculating the normal:

3D vector:

```
typedef struct vec_3f_struct
{
      float x, y, z;
} vec_3f;
```

Vector operations:

Vector subtraction:

```
vec_3f Sub_vec_3f(vec_3f vec1, vec_3f vec2)
{
      vec1.x -= vec2.x;
      vec1.y -= vec2.y;
      vec1.z -= vec2.z;

      return vec1;
}
```

Vector addition:

```
vec_3f DotProduct_vec3f(vec_3f vec1, vec_3f vec2)
{
      vec_3f tempVec;
      tempVec.x = vec1.y * vec2.z  -  vec1.z * vec2.y;
      tempVec.y = vec1.z * vec2.x  -  vec1.x * vec2.z;
      tempVec.z = vec1.x * vec2.y  -  vec1.y * vec2.x;

      return tempVec;
}
```

OpenGl specifies that all normals be unit length. Thus for normalization (setting to unity) of a vector its magnitude is required:

```
float Magnitude_vec3f(vec_3f vec)
{
      return (float)sqrt( vec.x*vec.x + vec.y*vec.y +
vec.z*vec.z );
}

vec_3f Normalize_vec3f(vec_3f vec)
{
      vec_3f tempVec;
      float mag_vec = Magnitude_vec3f(vec);

      tempVec.x = vec.x / mag_vec;
      tempVec.y = vec.y / mag_vec;
      tempVec.z = vec.z / mag_vec;

      return tempVec;
}
```

```
      glNormal3f( normal_vector.x, normal_vector.y, normal_vector.z
);
```

Then the normal for the triangle is calculated in r_main as:

```
      vec_3f      normal_vector,
                  vec1, vec2, vec3;

                  vec1.x = vert1_x;
                  vec1.y = vert1_y;
                  vec1.z = vert1_z;

                  vec2.x = vert2_x;
                  vec2.y = vert2_y;
                  vec2.z = vert2_z;

                  vec3.x = vert3_x;
                  vec3.y = vert3_y;
                  vec3.z = vert3_z;

normal_vector = Normalize_vec3f(DotProduct_vec3f( Sub_vec_3f(vec1,
vec2), Sub_vec_3f(vec2, vec3) ));
```

And in glBegin() the normal is defined before the vertices as:

```
glNormal3f( normal_vector.x, normal_vector.y, normal_vector.z );
```

This ensures that all lights behave properly as they would in the real world.

Part 4 – Appendices

Appendix A: In-Engine Screenshots

Yellow light turned on:

Blue light also on:

Appendix B: Mathematics behind the rendering pipeline transforms

The functions called in r_main() actually create certain matrices for OpenGl to multliply with the current matrix. Thus these matrices can be thought of as transformation matrices. What these matrices really are is defined below.

Translation

The call **glTranslate*(**x, y, z**)** generates **T**, where

$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 1 & 0 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

The call **glRotate*(**a, x, y, z**)** generates R as follows:

Let v = (x, y, z)T, and u = v/||v|| = (x', y', z')T.

Also let

$$S = \begin{bmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{bmatrix} \quad \text{and } m = uu^T + (\cos a)(1 - uu^T) + (\sin a)S$$

Then

$$R = \begin{bmatrix} m & m & m & 0 \\ m & m & m & 0 \\ m & m & m & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ where *m* represents elements from M, which is a 3x3 matrix.

For instance, if the rotation is about the x-axis:

glRotate*(a, 1, 0, 0): $$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos a & -\sin a & 0 \\ 0 & \sin a & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective Projection

The call **glFrustum(**l, r, b, t, n, f**)** generates **R**, where

$$R = \begin{bmatrix} \frac{2n}{r-1} & 0 & \frac{r+1}{r-1} & 0 \\ 0 & \frac{2n}{1-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$