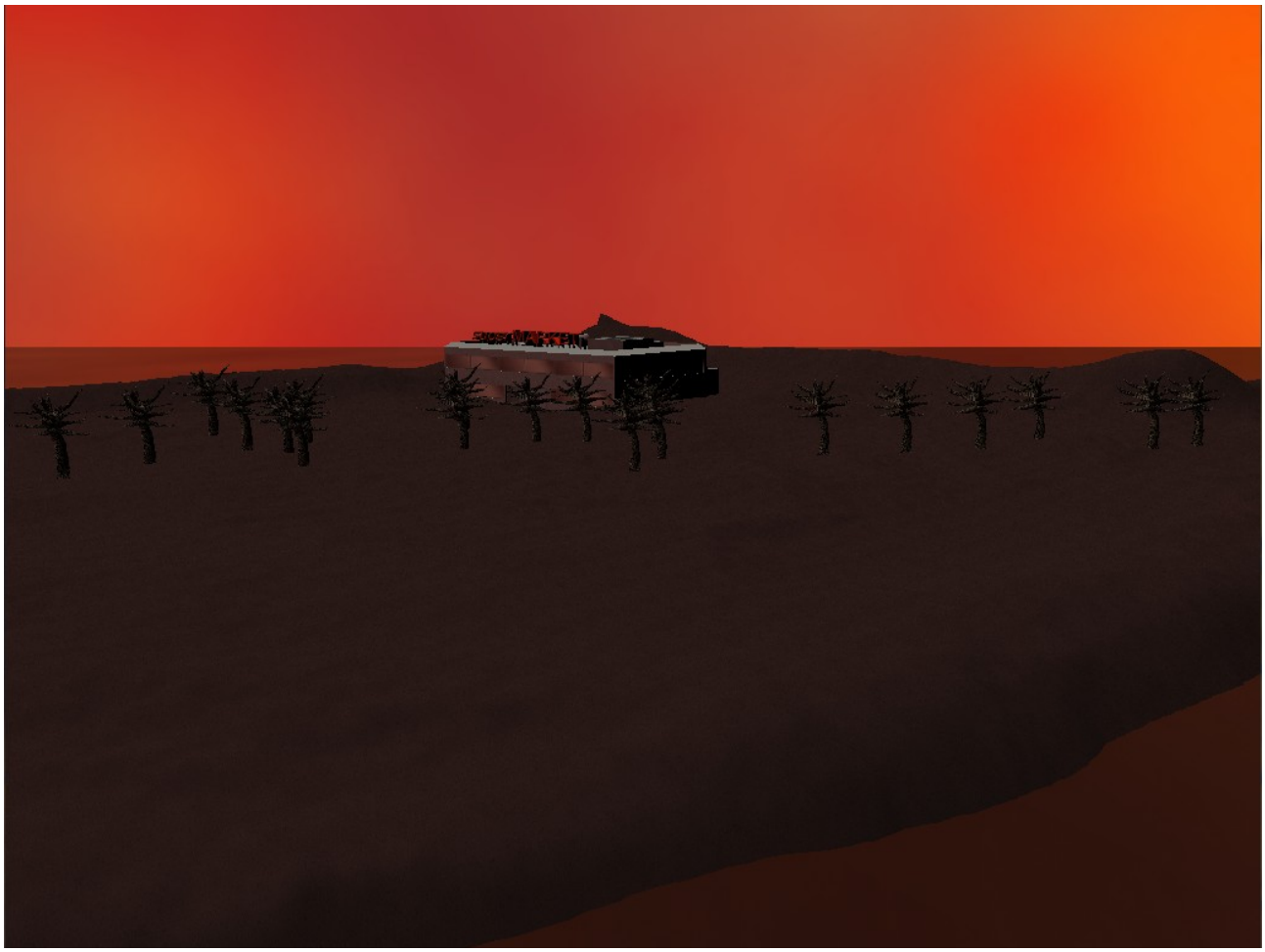
OpenGL Graphics Programming - Second Semester Coursework Report

By Vladeta Stojanovic (0602920@live.abertay.ac.uk)



Abstract Overview

This report focuses on the main features implemented for the OpenGL coursework application developed during the second semester.

The main features this report focuses on are:

- OpenGL/Win32 API interfacing
- Overview of the OpenGL fixed function rendering pipeline
- Basics of 3D projection projection
- Implementation of the Vector3 vector arithmetic and operations class
- Implementation of the Camera class used for user controlled scene navigation
- Implementation of the Terrain rendering class, including methods for constructing the 3D terrain mesh
- Overview and implementation of the MWE (Mean Weighed Equally) algorithm used for calculating surface normals used in real time hardware accelerated lighting of the 3D terrain mesh surface
- Implementation of skybox rendering

Portions of the program code were written and based on code provided by:

Matthew Bett - Win32 API base code, TGA texture loading code, MS3D model loading code
Jeromy Walsh – MWE code based on code provided for the GameDev.net article (see references)
Ben Humphrey – Vector3 class and Camera class code based on code examples provided on website (www.gametutorials.com) (see references)
Trent Pollack – Terrain class and .RAW file loading code based example code (see references)

Additional credit goes to Grant Clarke for help with implementing the MWE algorithm.

Coursework Overview

The requirement for the second semester graphics programming module coursework was to create a fully interactive 3D scene using OpenGL. A special emphasis was also placed on object orientated code design and implementation.

The coursework that was created for this module features an interactive OpenGL scene. In this scene the user can fly around a virtual landscape and examine the 3D rendered terrain. The user is able to see the fully rendered (but static) 3D models that are rendered along with the terrain. The scene also features a skybox and a cheaply imitated water body that is semi transparent and makes use of a spherically mapped texture. Finally, the scene also features standard linear fog.

The coursework program needed to make use of the Win32 API, along with OpenGL. OpenGL is used as part of the hardware rendering context (HRC) layer that is responsible for providing the hardware accelerated rendering to the hardware device context of the users application window handle. This allows OpenGL to interact with the current display device via the provided hardware driver (interfacing with the GDI module of the Win32 API, allowing the current users window to display OpenGL accelerated 3D graphics).

Below is a simple illustration of this concept:

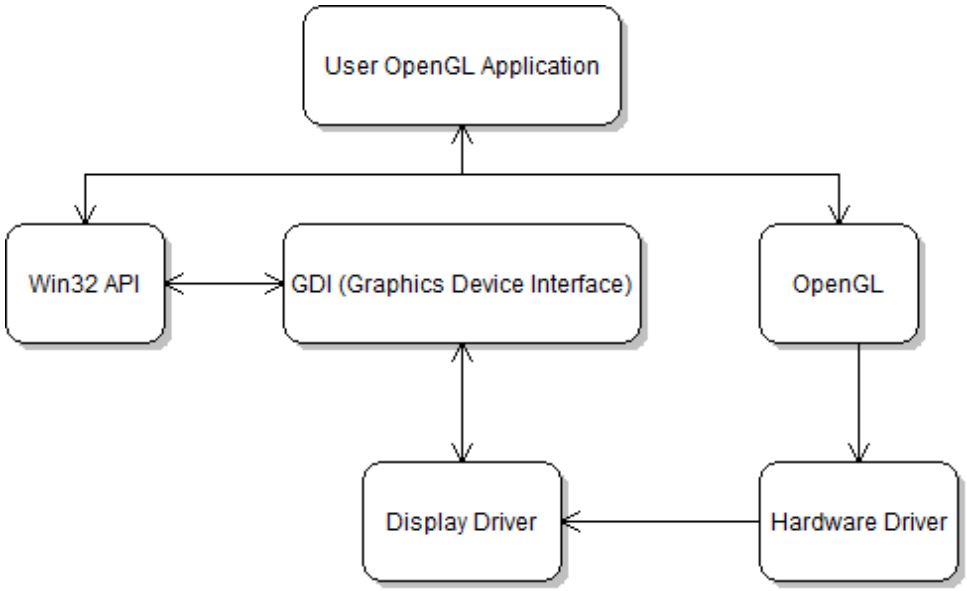


Fig 1: Win32/OpenGL Interfacing

As mentioned previously, emphasis was placed on object orientated code design and implementation. The coursework project features a modular approach to this by separating all of the main OpenGL related functionality components into classes (e.g. Terrain Class, Camera Class, Vector3 Class), along with standard procedurally programmed modules (e.g. the Milkshape 3D model loading code).

Together, all of the coursework application modules of the feature implementations work together in harmony to demonstrate a simple 3D scene in OpenGL using the Win32 API.

Implementation Overview

As mentioned above, the requirement for this coursework was to create a fully interactive 3D scene using OpenGL and Win32 API's. This section discusses the implementation of various 3D graphics features that the coursework demonstrates.

The main emphasis of the coursework was to create a scene that would very likely be found in some sort of computer game or 3D simulation. A very typical type of 3D scene that is often featured in various 3D games and simulations is that of an outdoor area, usually presented in the form of naturally occurring hills and mountains, rivers and trees (although these are very vague examples of the beauty that nature really holds).

It was decided that the main emphasis of the coursework would then be real time 3D terrain rendering. 3D terrain rendering is used widely in outdoor games such as flight simulators, real time strategy games, and more importantly: real time simulation of flood control, various geographical information system (GIS) analysis and presentations, as well as various military simulations. It can then be stated form this that the visual presentation of a 3D terrain in real time certainly holds it's weight in terms of advantages it can offer in the simulation of various scenarios.

Generally, a 3D terrain mesh is either generated procedurally or automatically (usually from offline data that is fetched from a source outside of the application, such as a user created height map image or a satellite photo). Terrain that is generated procedurally makes use of various algorithms to achieve realistic noise synthesis and slope generation that is found to be naturally occurring in real terrains. These algorithms include various fractal pattern generation algorithms and the Perlin noise algorithm.

The other way of generating terrains, as mentioned above, is from an already defined geometric value form, such a 2D image containing the height map values that are needed for the generation of the 3D terrain mesh. The coursework application makes exclusive use of this approach, mainly for simplicity, by making use of an 8-bit grayscale image to create a 3D terrain mesh based on the gradient values of each pixel in the image (this is described in more detail later on in the report).

Apart from generating 3D terrain meshes, there needs to be a viable way of displaying them. A simple wireframe representation would suffice for the simplest simulations, but for more complex scenarios, especially those where a realistic representation of the 3D terrain to the user would be beneficial, a more sophisticated display method needs to be implemented.

In these scenarios, a 3D terrain would normally be rendered either textured, smooth shaded or in some way that displays the vertex and face lighting values, along with relative colour values, of the varying colour gradients present in the mesh slopes of the 3D terrain mesh. One of the main aims of the OpenGL coursework application was to create 3D mesh terrain that would make use of hardware acceleration vertex and face normal computations to calculate the lighting gradients effecting each triangle face of the terrain mesh. This method of shading would produce desirable looking results, creating a smooth appearance in the change of lighting values between neighbouring triangle faces of the 3D terrain mesh.

A good algorithm that was chosen to achieve this smooth shading of the terrain surface (although smooth shading is a vague term used to describe many different shading algorithms that perform continuous interpolation of lighting colour value gradients between related polygon faces of a given 3D mesh), is the Mean Weighted Equally (MWE) algorithm, often referred to as Gouraud Shading.

The MWE algorithm work by calculating the normal of each face of the triangle strip that makes up the 3D terrain mesh, and the by taking each of the faces that share a common vertex point, it calculates an averaged surface normal. The averaged surface normals are then used to represent the normal vector of any given triangle of the 3D terrain mesh that shares a common vertex between it's neighbouring triangles.

Once a 3D mesh of a terrain is constructed, shaded, textured and lit, again, in most scenarios, the terrain acts only as a backdrop. Most of the time, things like water bodies, vegetation, roads, buildings, swamps, tree lines and various other geographical features need to be present along with the terrain in a given real time 3D scenario. The OpenGL framework demonstrates this by making use of the Milkshape 3D model loading code (provided by Matthew Bett), to load, position and render models of trees and buildings on the terrain. A water body is also rendered within the vicinity of the 3D terrain mesh. This water body consists of a single quad that is semi transparent and has a spherically mapped texture projected onto it's surface, giving a very basic (and perhaps crude) illusion of reflection (which is one of the most widely simulated properties of water bodies in computer graphics).

A final feature is the inclusion of liner fog, that can be turned on or off by user. The linear fog can add a sense of depth to the perspective of the scene. Fog can also be used to dynamically occlude parts of the scene, especially if the drawing distance in the rendered scene is very limited. This allows the simulated fog to act not only as an aesthetic to a 3D scene, but also as a potential optimization.

All of these features put together allow the coursework application to demonstrate greater potential for various scenario computation and analysis.

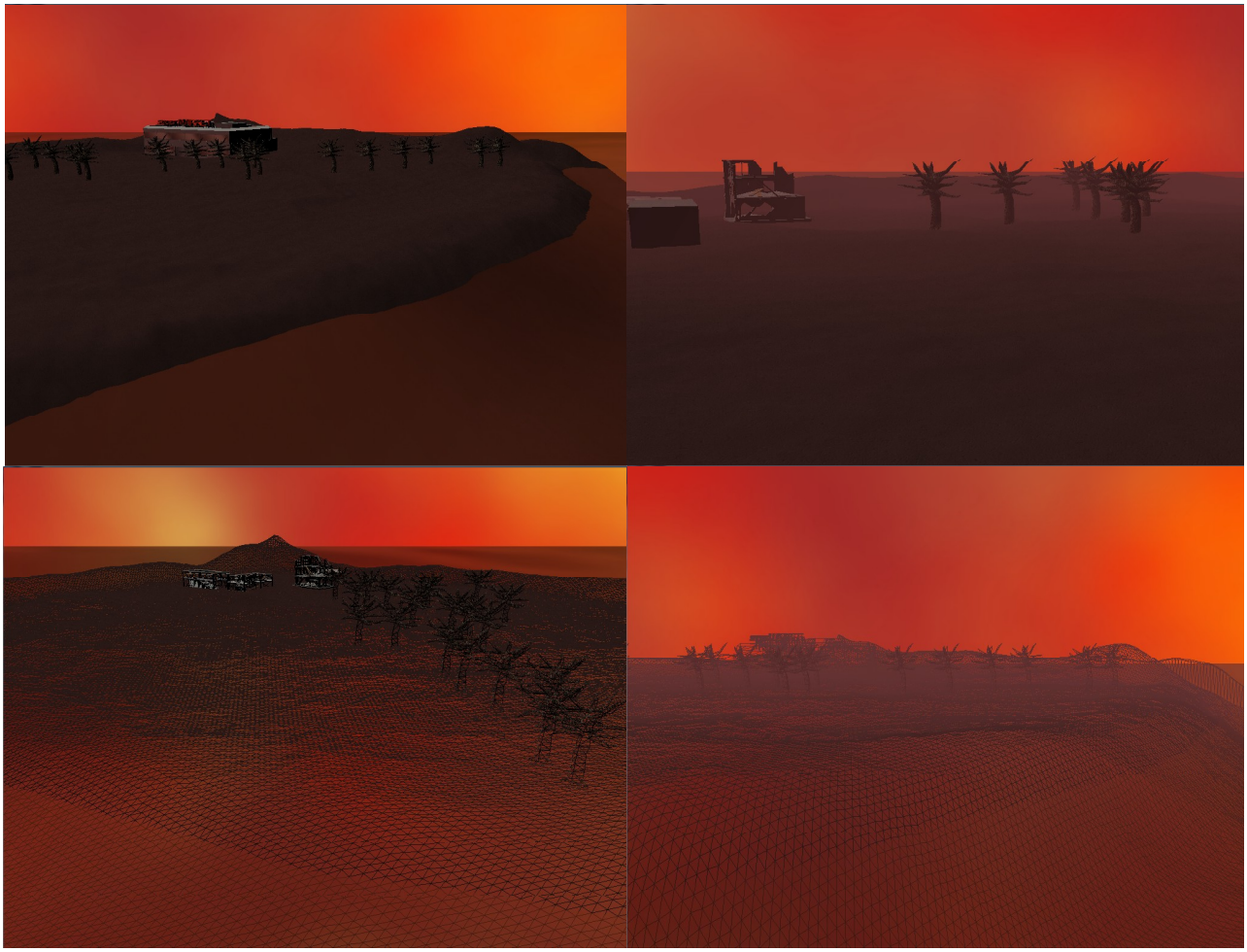


Fig 2: Screenshots of the application running demonstrating the various mentioned features

Implementation Details

This section describes the specific details of the various implementation methodologies used to construct the OpenGL coursework application.

Win32/OpenGL Interfacing

In terms of 3D graphics acceleration under Windows, OpenGL is used as a low level layer between the hardware rendering context of the application window handle and the users display driver. This allows any programmer to use OpenGL in various development environments under the Windows operating system, since the generic OpenGL implementation that is provided with Windows allows the programmers to use both software and hardware rendering.

The default OpenGL specification implementation that is provided for use with Windows is the OpenGL 1.1 specification. This specification is deprecated and no longer used by any currently developed commercial applications (unless support for OpenGL 1.1 is implemented separately as a fall back option for computers that don't support higher versions).

However, OpenGL 1.1 still has valid uses for educational use and training, as it allows novice graphics programmers to experiment with the basic fixed function pipeline features of 3D rendering (more on this in the sections below).

The coursework application makes exclusive use OpenGL 1.1 along with the Win32 API. Below is an high level overview of the implemented interface for OpenGL using Win32.

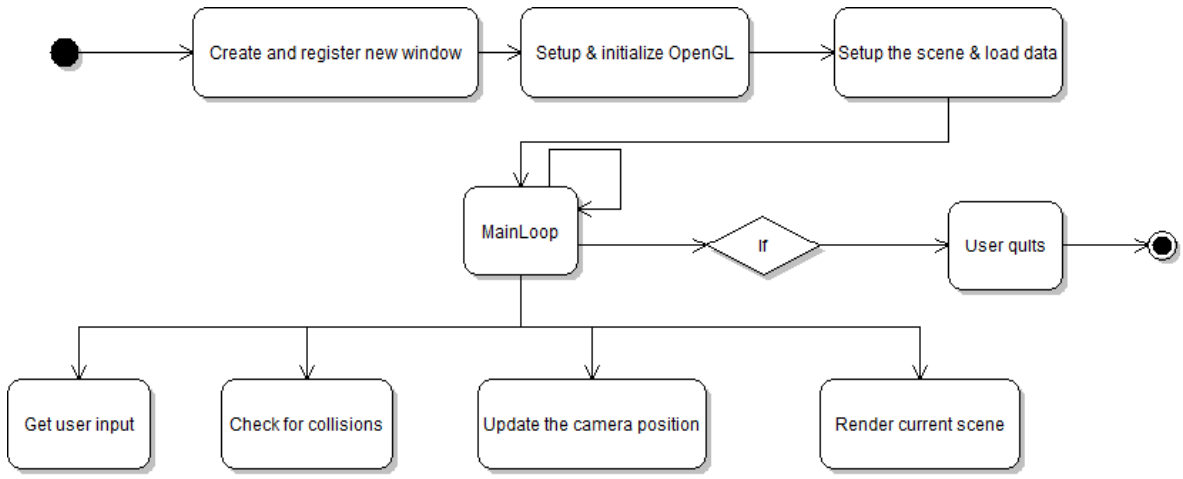


Fig 3: Flowchart diagram of the application main loop and Win32/OpenGL interfacing.
Note: The collision checking stage is omitted from the final version application.

There are a few stages of setup code that require implementation before a Windows application can actually make use OpenGL to render to the application window frame buffer. The basics of setting up an application window wont be discussed here. Instead, descriptions for each of the critical OpenGL setup stages of the windowing code implementation are discussed below in detail.

The three most important functions that are used in coursework application to set up OpenGL are:

```
bool SetPixelFormat(HDC hdc);
void ResizeGLWindow(int width, int height);
void InitializeOpenGL(int width, int height);
```

StetupPixelFormat()

Every application running under windows must request a device context identifier. This identifier tells windows what sort of display device to use when displaying the contents of the given application window.

Most native applications using Win32 will make use of the default device context associated with the current desktop device context. GDI applications that are running in windowed mode will always make use of the current desktop colour depth.

However, when using OpenGL to perform 3D rendering to an application window, the current application window must be setup to accommodate the requirements of the hardware rendering context layer associated with the application window.

These setup options include setting the colour depth, colour mode, double buffering, software or hardware rendering support and the drawing destination to where OpenGL can display it's rendered contents of the frame buffer. These are only the most basic setup options needed for a generic OpenGL fixed function pipeline application.

The code example below highlights the selected pixel format for the OpenGL coursework application.

```
PIXELFORMATDESCRIPTOR pfd = {0};
int pixelformat;

pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
pfd.nVersion = 1;
pfd.dwFlags = PFD_DRAW_TO_WINDOW | PFD_SUPPORT_OPENGL | PFD_DOUBLEBUFFER;
pfd.dwLayerMask = PFD_MAIN_PLANE;
pfd.iPixelType = PFD_TYPE_RGBA;                                  pfd.cColorBits   =
COLOUR_DEPTH;                            pfd.cDepthBits = COLOUR_DEPTH;
                pfd.cAccumBits = 0;
pfd.cStencilBits = 0;
```

The code above sets up the application window to allow OpenGL to draw to that window frame buffer. The pixel format setup code also allows the application window to make use of double buffering for displaying the contents to the frame buffer.
The colour depth and colour mode is also setup to 32 bit colour, using 8 bits for each the four RGBA colour channels. The application does not make use of the accumulation buffer or the stencil buffer.

Once the pixel format for the application window is setup, the function will return a true Boolean value up a successful setup (otherwise an error message(s) is returned).

ResizeGLWindow()

The coursework application window may need to be resized at certain times while the application is running. Resizing the application window means that the current frame buffer will need to be displayed within a new rectangular window area. Often, stretching the viewport can distort the currently rendered contents of the frame buffer, making them appear stretched out.

The main technique used to resolve this issue is to implement a simple viewing transformation routine. OpenGL does this automatically, resetting the current projection matrix by using an identity matrix function, glLoadIdentity(), to reset the current display matrix every time the window is resized. This allows the current image in the frame buffer (whose coordinates are stored in within the world coordinate system), to be properly mapped onto the new window rendering area (which uses the current device coordinate system).

```
void ResizeGLWindow(int width, int height)
{
	if (height==0)
	{
		height=1;
	}

	glViewport(0,0,width,height);

	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();

	gluPerspective(45.0f, width/height, 1.0f, 500.0f);

	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
}
```

The ResizeGLWindow() functions takes in integer value parameters of the window with and height. These parameters are used to properly set the size of the size of rendered image in the frame buffer. The glViewport() function is used to specify the transformation matrix from the normalized world coordinates to the application window coordinates. The first two values of the glViewport() function are used to specify the lower left corner of the viewport rectangle, and the width and height values are used to specify the width and height of the viewport that the normalized world coordinates get projected onto.

It is very important to note the order of matrix transformations in OpenGL. Since OpenGL uses a state driven mechanism to store the current enumerated rendering states, every function call is placed on a virtual stack (either being a projection matrix stack, a modelview matrix stack or a texture matrix stack). So when the function ResizeGLWindow() is called, the OpenGL state machine calls the matrix operations using a first in last out mechanism.

Therefore, the first function to be called by the OpenGL state machine is glLoadIdentity(), followed by glMatrixMode(GL_MODELVIEW), followed by gluPerspective(), and then only does the glLoadIdentity matrix function get executed again before the glMatrixMode() and glViewport() function are called. This makes a lot of sense, since the order of matrix operations in OpenGL make use post multiplication (each element being a matrix):

$$ABCD = ((DC)B)A$$

Perspective Projection

OpenGL allows programmers to automatically access affine matrix transformation functions. This in turn hides the underlying principals of the mathematics used to define the viewport projection matrices that are automatically calculated by OpenGL, making such computations easily applicable. However, it is important to describe the basic underlying mathematical principals of 3D viewport transformation and perspective projection.

The main aim of the OpenGL projection matrix is to transform the current view of the 3D scene into a 2D image that can be rendered onto the users application window.

A common method used in 3D computer graphics to transform a viewpoint onto a given plane using perspective projection, is to use a given point in 3D space as the viewing centre. The coordinates defined by this viewing point are then projected onto the given plane using perspective projection.

Calculating the projection is done in simplified terms by performing a ray cast vector intersection test with the viewing frustum, to determine where in the frustum the point lies (between the near and far frustum planes). The new coordinate points are then transformed back to origin, before being mapped on to the viewing plane (defined by coordinates (l t n), (r t n), (l b n) and (r b n)) .

Below is an illustration of the OpenGL viewing frustum:

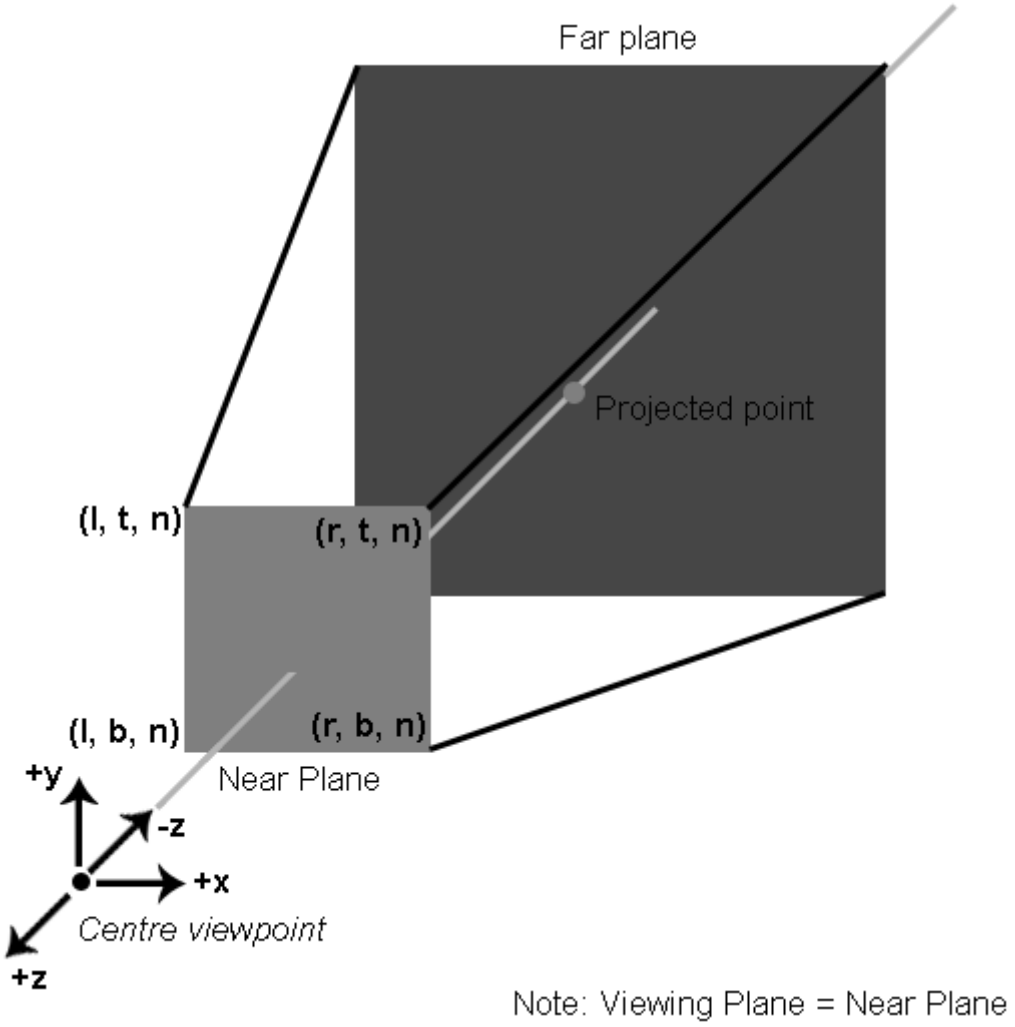


Fig 4: The OpenGL perspective rendering viewing frustum

The viewing centre is defined as a point in 3D space:

$$C(x_c, y_c, z_c)$$

And the coordinates of the point to be transformed are defined as:

$$Original\ point:\ P(x, y, z) \quad Transformed\ point\ coordinates:\ P'(x', y', z')$$

Since the coursework makes use of perspective projection to define the viewing frustum (pictured below), it is necessary to calculate the projection of a given point in 3D space onto a viewing place defined by the following parameters:

$$ax + by + cz = d$$

Where the translation of coordinates between the viewing point and the transformation point defines the plane as :

$$X = x - x_c, \quad Y = y - y_c, \quad Z = z - z_c$$

giving the viewing centre coordinates a value of (0, 0, 0)
therefore, the perspective projection plane becomes defined as:

$$aX + bY + cZ = d - (ax_c + by_c + cz_c) \Rightarrow aX + bY + cZ = d_0$$

Therefore the perspective projection matrix for the given viewpoint onto a given viewing plane can be defined as (proof not provided):

$$Perspective\ Projection = \begin{pmatrix} d-(by_c+cz_c) & ay_c & az_c & a \\ bx_c & d-(ax_c+cz_c) & bz_c & b \\ cx_c & cy_c & d-(ax_c+by_c) & c \\ -dx_c & -dy_c & -dz_c & -(ax_c+by_c+cz_c) \end{pmatrix}$$

One last final note to mention about perspective projection goes back to the ray cast from the given centre view point into the frustum, to determine if there is intersection between the point to be projected, and the near viewing plane.

This is determined calculating the distance between the centre viewpoint and the viewing plane, and the point to be projected and the viewing plane. E.g.:

$$D_c = ax_c + by_c + cz_c - d \quad \textit{for the centre viewing point}$$

$$D_p = ax_p + by_p + cz_p - d \quad \textit{for the point being projected}$$

Where it needs to be determined if Dc and Dp are of different signs. If Dc and Dp are of different signs, then calculation of perspective projection is possible.

Perspective Transformation from a point in the 3D frustum to the Normalized Device Context (NDC)

Once a perspective projection for a given point in 3D space is calculated and transformed back to the origin (the viewing plane), it needs to be transformed onto the Normalized Device Context (NDC).

What this means is that if the given point is transformed within the viewing frustum, it will be projected onto the viewing plane. Once it is projected onto the viewing frame, it needs to projected within the NDC frustum. What makes the NDC frustum special is that it is of unit length for all negative and positive x y z coordinates (essentially a unit length cube).

The point is then orthographically projected parallel to the z-axis, giving all the new point coordinates the same value of unit length. The z coordinate value can be discarded, and what the calculation produces is a 2D image coordinate for the orthographically projected point. These new 2D x and y point coordinates can then be used to map the rendered point from the frame buffer onto the users hardware device context window handle.

The NDC frustum is defined by transforming the perspective viewing frustum between the near and far clipping planes, into cube. The perspective frustum is made up of two z depth planes, far and near. Transforming the viewing frustum into the NDC frustum is accomplished by using the following transformation matrix:

$$N_{frustum} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{(f+n)}{(f-n)} & -1 \\ 0 & 0 & \dfrac{(2fn)}{(f-n)} & 0 \end{pmatrix}$$

where near and far planes are set be at: $z' = \pm 1$
therefore (proof tranquated), the near and far z planes are represented as:

$$\alpha = \frac{(f+n)}{(f-n)} \text{ and } \beta = \frac{(2fn)}{(f-n)}$$

The frustum between z = -n and z = -f is transformed into the NDC frustum.

The final matrix transformation that might be applied to the point projected onto the viewport, concerns the origin and the aspect ratio of the viewport. If the origin and the aspect ration of the viewport is different to that of the NDC frustum plane front plane (given coordinate values between 0 to 1), a further transformation matrix must be applied to the projected point in order to map it properly onto the users viewport.

The matrix responsible for this task is defined as:

$$T_{\text{scale,translate}} = \begin{pmatrix} \frac{1}{2}w & 0 & 0 & 0 \\ 0 & \frac{1}{2}h & 0 & 0 \\ 0 & 0 & \frac{1}{2}ZR & 0 \\ X_0 & Y_0 & \frac{1}{2}ZR & 1 \end{pmatrix}$$

H and W represent the users window width and height, ZR being the z-buffer range and X_0 and Y_0 being the new coordinates of the frame origin.

One last note to mention is that the NDC frustum matrix transformation is defined differently in OpenGL. This is because OpenGL uses column major matrices (making the OpenGL implementation of the NDC frustum matrix transposed), defined as:

$$N^{T}_{frustum} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{(f-n)} & \frac{-(2fn)}{(f-n)} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

InitializeOpenGL()

```
void InitializeOpenGL(int width, int height)
{
    ghdc = GetDC(ghwnd);

    if (!SetPixelFormat(ghdc))
        PostQuitMessage (0);


    ghrc = wglCreateContext(ghdc);
    wglMakeCurrent(ghdc, ghrc);

    ResizeGLWindow(width, height);

 g_Camera.PositionCamera(0, 40.0f, 400.0f,   0, 0, -1.0f,   0, 1, 0);
}
```

The final function that is featured in the OpenGL setup code is the InitializeOpenGL() function. The function does a 5 things:

1: The function assigns the current hardware rendering context handle to the application window handle, thus giving OpenGL access to the applications window:

```
ghdc = GetDC(ghwnd);
```

This function gives the OpenGL hardware device context access to the window handle.

2: The function calls the SetPixelFormat function, which sets up the display attributes associated with the graphics hardware device context:

```
if (!SetPixelFormat(ghdc))
    PostQuitMessage (0);
```

If the pixel format is not support, the application will post a quit message callback to the main window procedure.

```
    ghrc = wglCreateContext(ghdc);
    wglMakeCurrent(ghdc, ghrc);
```

3: The next two lines of code specify the current rendering window for the application. This is referred to as the rendering context for the window.

```
ResizeGLWindow(width, height);
```

4. Then the window resizing function is called. The application window will be given a default size of 800x600 pixels.

```
 g_Camera.PositionCamera(0, 40.0f, 400.0f,   0, 0, -1.0f,   0, 1, 0);
```

5. Finally, the users cameras is given a default starting position. More details about the camera implementation will be covered in depth in later section of this report.

Fixed Function Pipeline Overview

The OpenGL rendering pipeline is used to describe a number of stages that are involved in the process of rendering an image to the users application window. This involves the initial stages from the actual OpenGL API calls, to the rasterization of vertex and fragment data, to copying the image to the frame buffer, which is then used to display the final rendered image (or frame in most cases).

A very high level view of the standard fixed function OpenGL rendering pipeline can be described in the five basic stages (well four actually, since OpenGL API function calls are not really part of any of the processing stages), illustrated below:

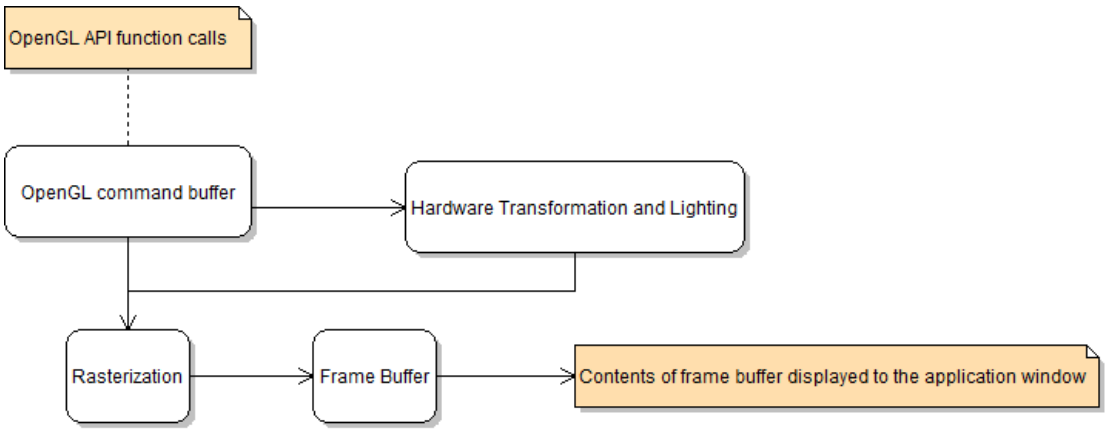


Fig 5: A high level version of the standard fixed function OpenGL pipeline.

The first stage consists of fetching all the function calls from the OpenGL API and storing the function calls in the OpenGL command buffer. The function calls either deal with vertex or fragment data.

Most of the time, vertex data needs to have transformation and lighting calculations applied every frame, but in some cases, where transformations and lighting calculations are not needed, this stage can be omitted.

Next the vertex and fragment data gets assembled at the rasterization stage. This stage of processing involves constructing the actual frame that will be rendered, which is constructed from the vertex, fragment and texture data that is fed to the rasterizer.

The rendered frame is then placed in the frame buffer, from where it is displayed to the users application window, via the hardware rendering device context that is setup for that particular window. All of this describes the most basic stages of using the standard fixed function OpenGL rendering pipeline.
However, there are many more stages involved in the process. These stages will now be briefly discussed.

The OpenGL fixed function pipeline became virtually deprecated when OpenGL 2.0 was released in 2004, with standardized added support for writing assembly level shader code, as well as the initial introduction of GLSL (OpenGL Shader Language). This transformed the current graphics pipeline on graphics cards that supported OpenGL 2.0 from fixed function to a fully programmable graphics pipeline. It is unfortunately beyond the scope of this report to focus on the programmable graphics pipeline and shaders. The next OpenGL coursework application will inevitably make use of GLSL shaders, so the theory and implementation details of using shaders will addressed and explained at that time.

Going back to the fixed function pipeline, below is a detailed diagram of all the stages of processing that are found in the fixed pipeline:

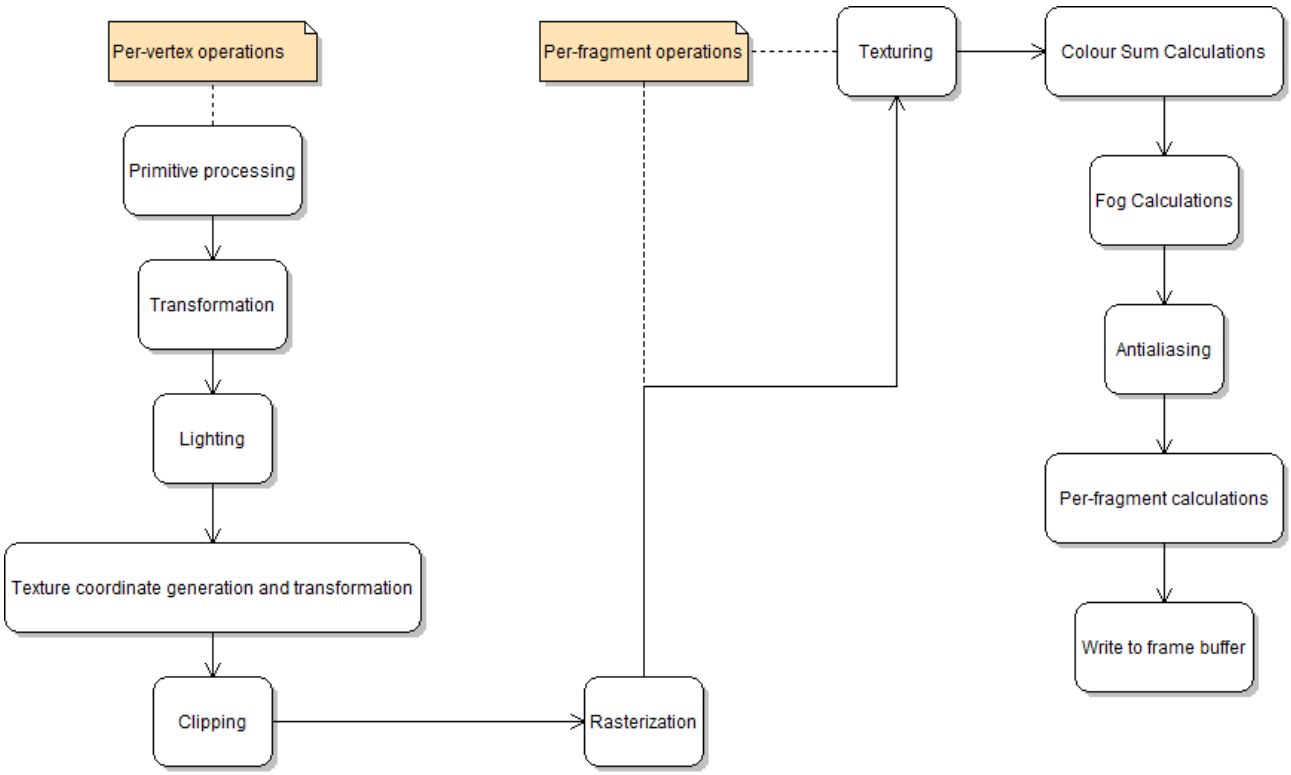


Fig 6: A more detailed overview of the OpenGL fixed function rendering pipeline.

An good and important side note to mention is that at the present time, almost all personal computers have GPU's with a programmable pipeline, therefore the fixed function pipeline is actually emulated by all of today's modern graphics cards that implement the OpenGL 2.0 specification (and above, going up to 4.0 at the time of writing this report).

The initial stages of the OpenGL fixed function pipeline deal with vertex data. Once the operations that need to be performed on the vertex data are computed, the current geometric primitive (e.g. a triangle strip or a quad), is rasterized and converted to fragment data (aka pixel data). The fragment data then undergoes further processing through the requested fragment processing stages, before being written to the frame buffer and displayed to the users window via the hardware rendering device context handle.

The processing stages in the fixed function pipeline can be separated into per-vertex processing stages and per-fragment processing stages.

Now to briefly explain each stage:

Note: For simplicity, it will be assumed that a single primitive object is being rendered.

Per-vertex processing stage (Primitive Processing)

This is initial stage of the per-vertex processing. It is at this stage that the attributes of the vertex based data in the current scene is passed in. The vertex data is usually made up of indexed vertex array data, which includes information such as the position, normal vector, colour and texture coordinates of the primitive object.

Transformation

At this stage the vertex position of the primitive object is transformed for the world coordinate space to the application viewport space. This is where the use of the viewing transformation comes in.
It is also at this stage that the normal vectors for the primitive are transformed (assuming the primitive has a normal vector assigned to it). The normal vector is multiplied by the inverse transpose of the model view matrix (world coordinate matrix). The normal vector is usually normalized, since a unit length normal vector is required to perform lighting calculations. The normal vector may also be rescaled at this stage.

Lighting

The lighting computation stage of the pipeline takes in the primitive vertices colour, normal vector and position. It computes and outputs two colour values, one primary and one secondary, and checks whether the back faces of the primitive need to be lit as well.

In the fixed function pipeline, the a single hard coded lighting equation for the surface shading of the primitive is used by default (usually defined GL_SMOOTH). More details about smooth shading will be given later.

This stage also takes into account the number of lights present in the scene (eight lights being the maximum number of lights allowed), along with any additional material information about the primitive (e.g. diffuse, ambient and specular colour values), when computing the lighting equation for the surface shading of the primitive.

Texture coordinate generation and transformation

Assuming that the primitive has texture coordinates assigned to it, this where they get processed. Texture coordinates can either by specified manually or automatically generated by OpenGL. The computation at this stage involves transforming every texture coordinate by the texture matrix, in order to project the texture coordinates of the primitive onto the texture map in video memory. Common texture mapping projections include planar (almost always user specified, usually by a UV map), spherical and cylindrical mapping (more on this later).

Clipping

This is where clipping occurs. Any primitive that is projected onto the viewport space is projected within the viewing frustum set up by the application. Clipping determines which regions of the primitive will be displayed and which will not. Newly generated vertices of the area of the primitive where clipping occurs are assigned new vertex attributes (which are updated every frame).

Per-fragment processing stage (Rasterization)

This stage of processing takes in the interpolated fragment values of the primitive , which include texture coordinates and colour coordinates (provided by the lighting and material value computed surface shading interpolation, from the previous lighting stage). The result of the computed data that is processed and passed out of this stage is a single colour value corresponding to the computed fragment.

Texturing

This stage takes in the interpolated fragment texture coordinate value as well as the passed in colour value from the previous stage. The output is a new colour based on the result of the texture lookup that is associated with the active texture unit (that being either a 2D texture or something like a cube map texture).

The process computes and outputs a new fragment colour value that is based on the result of the texture lookup (which either blends or replaces the current fragment colour value). The new fragment colour value is then passed on.

Colour sum calculations

Assuming this stage is enabled, the passed in fragment colour value, along with a passed in secondary colour value (from the lighting calculation), get their RGB colour channels added together and clamped to a floating point value from 0 to 1. The new colour value of the current fragment is passed onto the next stage.

Fog calculations

Assuming fog is enabled, the passed on colour fragment value is interpolated with the fog colour using one of the three hard coded fog factor equations (linear, exponential or second-order exponential). The fog equations use the fog coordinate value that is either interpolated along with the subsequent fragment values at the rasterization stage, or is passed in along with the vertex attributes at the primitive processing stage. The coursework application makes use of linear fog (more on this later on).

Antialiasing

Again, assuming antialiasing is enabled, this stage computes the smooth edges of the primitive by multiplying the alpha value of currently processed fragment with what is known as a coverage value. The coursework application does not make use of antialiasing.

Per-fragment calculations

At this stage all of the final fragment computations get carried out. These include any enabled per-fragment tests in the order of scissor testing, alpha testing, stencil testing, depth testing, blending and dithering. It is beyond the scope of this report to describe in detail what each of these tests do, but it is important to note that this stage determined whether or not the current fragment should be written to the frame buffer.

These test are mostly used when dealing with things like transparent sprites rendered over quads (e.g. billboarding), or when rendering particle effects.

Write to frame buffer

Once the fragment has successfully passed thru all the per-fragment processing stages, it gets written to the frame buffer. The frame buffer is used to hold frame data that is displayed on the users application window.

It should be noted that the coursework application makes exclusive use of double buffering.
This will be discussed in more detail in later sections of this report.

Using object orientated programming techniques

As mentioned previously, one of the main requirements for this semesters OpenGL coursework was that the application that needed to be developed need to make use of object orientated design and code implementations.

It was decided early on that designing and implementing three different classes would be beneficial to the project, as it would allow for a modular approach to adding and testing new features, as well as cleanly structuring the layout of the code.

Below are details about the design and implementation of the three main classes used in the coursework application: the Vector3, Camera and Terrain classes.

Vector3 Class

Vector3 Class
GLfloat x, y, z: Directional 3D vector component real numbers
Vector3 Constructor() Vector3 + Operator: Overloaded 3D vector component addition operator Vector3 - Operator: Overloaded 3D vector component subtraction operator Vector3 * Operator: Overloaded 3D vector component multiplication operator Vector3 / Operator: Overloaded 3D vector component division operator Vector3 Cross(): Returns the cross product between two 3D vectors Vector3 Normalize(): Normalizes a given 3D vector GLfloat Magnitude(): Returns the real number value of the magnitude of the given 3D vector

Fig 7: The Vector3 class

Below is a basic description of each of the member functions of the Vector3 class, as well the associated data members of the class.

Vector3::Vector3 Constructor()

The Vector3 class constructor can take in three different parameter values. All three are GLfloat type floating point numbers, representing the x, y and z directional components of the instantiated Vector3 class object.

Overloaded operators:

```
Vector3 operator+(Vector3 vVector)
{
  return Vector3(vVector.x + x, vVector.y + y, vVector.z + z);
}

Vector3 operator-(Vector3 vVector)
{
      return Vector3(x - vVector.x, y - vVector.y, z -
vVector.z);
}

Vector3 operator*(GLfloat num)
{
      return Vector3(x * num, y * num, z * num);
}

Vector3 operator/(GLfloat num)
{
      return Vector3(x / num, y / num, z / num);
}
```

The featured operators have been overloaded to accommodate the needs of computing vector arithmetic.

Vector3 Vector3::Cross()

A common use for vector cross products in 3D computer graphics is to find a new vector that is orthogonal (.e.g. perpendicular) to the two vectors from which it gets derived from.

If there are two vectors , v1 and v2, the cross product between those two vectors (which will be called u), is calculated by using the formula:

$$\left\|(\vec{v}_1 \times \vec{v}_2)\right\| = \left\|\vec{v}_1\right\| \times \left\|\vec{v}_2\right\| \times \sin\theta$$

Where sin θ is the angle between the vectors v1 and v2, and where the vector lengths of v1 and v2 are used to calculate the component values of the cross product vector u.

It should be noted that if the vectors v1 and v2 are parallel, then sin θ = 0, which produces a zero vector. Also, if sin θ = 1, then the cross product will be a unit vector (assuming in that case that vectors v1 and v2 are of unit length). Since the cross product is used to compute the perpendicular between the two vectors v1 and v2, sin θ will always be equal to 1 (90°).

The calculating the cross product is defined as:

$$\vec{v}_1 \times \vec{v}_2 = \left(v_{1y}v_{2z} - v_{2y}v_{1z}, v_{1z}v_{2x} - v_{2z}v_{1x}, v_{1x}v_{2y} - v_{2x}v_{1y}\right)$$

The most common use of the vector cross product is using the cross product to generate a perpendicular vector product between tow given vectors. A good example would be a unit vector generated between three points (vertices) that make up a face of a triangle. This unit vector is referred to as the normal vector.

Depending on the construction of the triangle face (using either clockwise or anti-clockwise vertex welding), the generated normal vector can either be pointing outwards (v2 x v1) or inwards (v1 x v2). Normal vectors of a triangle face that point inwards are usually used to represent the backface of a triangle. In most cases, due to optimization, backfaces of triangles are not rendered (this is referred to as backface culling).

Below is a simple diagram used to illustrate the concept of normal vectors of a triangle face:

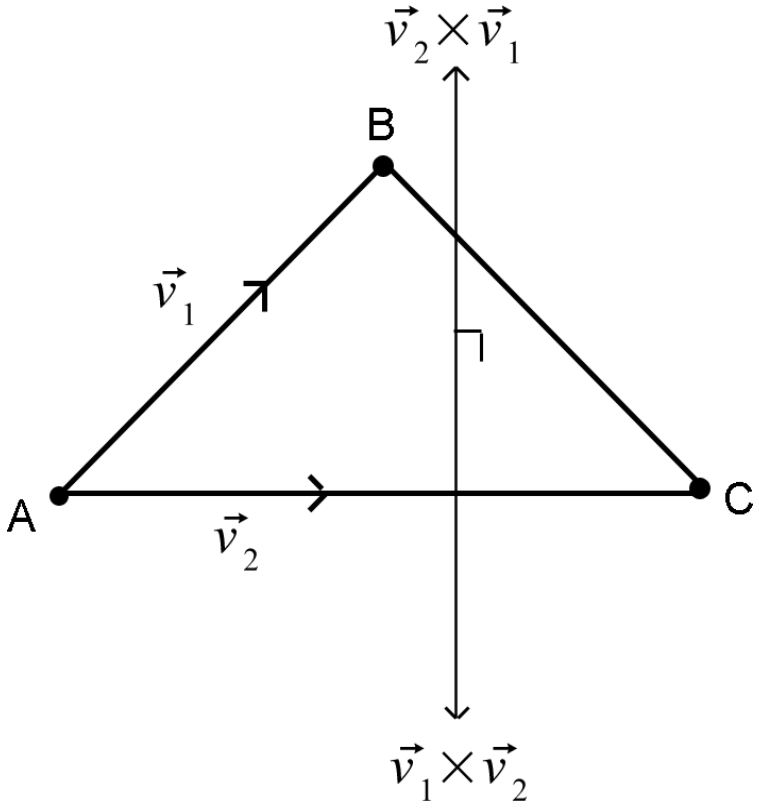


Fig 8: A normal vector of a triangle face.

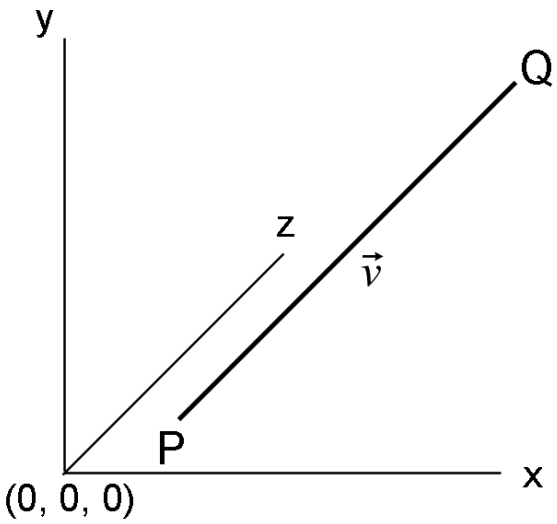
Finally, the vector cross product calculation can be implemented in the Vector3 class.

```
Vector3 Vector3::Cross(Vector3 vVector1, Vector3 vVector2)
{
      Vector3 vNormal;

      vNormal.x = ((vVector1.y * vVector2.z) - (vVector1.z *     vVector2.y));

      vNormal.y = ((vVector1.z * vVector2.x) - (vVector1.x *     vVector2.z));

      vNormal.z = ((vVector1.x * vVector2.y) - (vVector1.y *     vVector2.x));

      return vNormal;
}
```

The Cross() function of the Vector3 class computes the cross product between two vectors, and returns the computed normal vector as a new Vector3 object instance.

GLfloat Vector3::Magnitude()

In 3D space the magnitude (aka length) of a given vector is based on 3D Euclidean distance equation, which essentially measures the distance between two different points in 3D space.



$$\|\vec{v}\| = \sqrt{((p_1 - q_1)^2 + (p_2 - q_2)^2 + (p_3 - q_3)^2)}$$

Fig 9: Vector length formula and diagram

In the case of the Vector3 object, that would be starting and the ending points.

```
GLfloat Vector3::Magnitude(Vector3 vNormal)
{
      return (GLfloat)sqrt( (vNormal.x * vNormal.x) +
                      (vNormal.y * vNormal.y) +
                      (vNormal.z * vNormal.z) );
}
```

The Vector3::Magnitude() function takes in a Vector3 object as a normalized vector, and from it computes the magnitude (length) of the normal vector. The value is returned as GLfloat type floating point number.

Vectro3 Vector3::Normalize()

A normalized vector is often used to determine the direction of an entity that the vector is representing. In order to use a given vector as a direction vector, the vector used must be of unit length (1). Most of the time, calculating only the length of a given vector will not produce a unit vector, therefore the given vector must be normalized in order to be used a direction vector.

The following formula is used to normalize a given vector:

$$\hat{v} = \vec{v} / \left\| \vec{v} \right\|$$

Applying the equation above to a given vector will result in the vector magnitude being given a value of 1. E.g.

$$\left\| \vec{v} \right\| \times 1 / \left\| \vec{v} \right\|$$

Below is the code implementation of the vector normalization routine in the Vector3 class.

```
Vector3 Vector3::Normalize(Vector3 vVector)
{
	GLfloat magnitude = Magnitude(vVector);
	vVector = vVector / magnitude;

	return vVector;
}
```

The Normalize function returns the normalized vector from the input vector. The normalized vector is computed by dividing the input vector by it's magnitude.

Camera Class

Camera Class
Vector3 PositionVector Vector3 ViewVector Vector3 UpVector Vector3 StrafeVector
Camera(): Class constructor PositionCamera(): Set gluLookAt() RotateView(): Rotate the camera arond a certain axis at a certain angle SetViewByMouse(): Rotate the camera based mouse input StrafeCamera(): Strafe the camera in left or directions based on keyboard input MoveCamera(): Move the camera backwards of forwards based on keyboard input GetInput(): Get user keyboard input Update(): Update the current camera position based on user input and collision detection results Look(): Set gluLookAt() using updated 3D camera position values

Fig 10: The camera class (note that camera collision was not implemented in the coursework)

The first main application of the Vector3 class is found in the camera class. The camera class is used to control the projected perspective rendering viewport. The camera class does not modify the viewport, but it does give the user the ability to fly around the 3D scene and examine it. The main functionality of the camera class is controlled via keyboard and mouse input.

Camera control is based on the camera position, direction and the positive y vector (aka “up vector“). The current camera view is then computed by calling the gluLookAt() function, which takes in the three mentioned sets of data and converts them into a view model matrix that is added to the MODELVIEW matrix stack.

The three sets of data mentioned (again: camera position, direction and an up vector), are used to represent the 9 different x y z values for each of the three different sets of data. Now what the camera control functionality of the camera class actually does is that it modifies all those values per frame, based on the current user input associated with camera control. But what exactly does the user control?

A simple representation of a camera would be a projected viewport plane of the viewing frustum of the camera object:

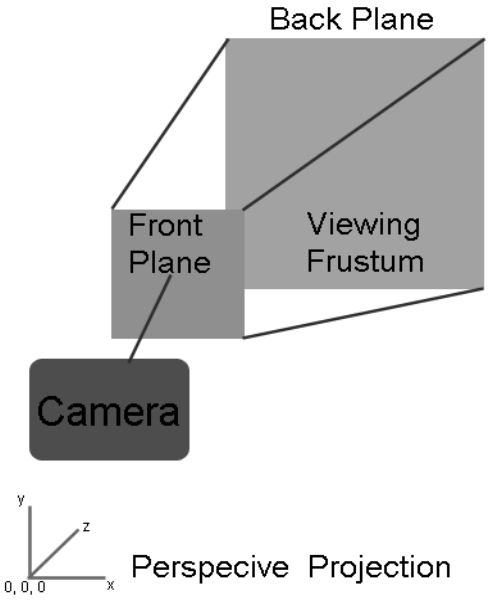


Fig 11: Camera setup

Whatever is rendered within the viewing frustum (is not clipped), is projected onto the viewport plane (via MODELVIEW projection matrix stack concatenations), that gets sent to the frame buffer. So a simple description of the camera is that it is an object, a class containing a set of modifiers, which modify the projection values of the perspective rendered viewing frustum. These modifications are represented as rotations between the positive and negative x y z axis, as well as the current camera position.

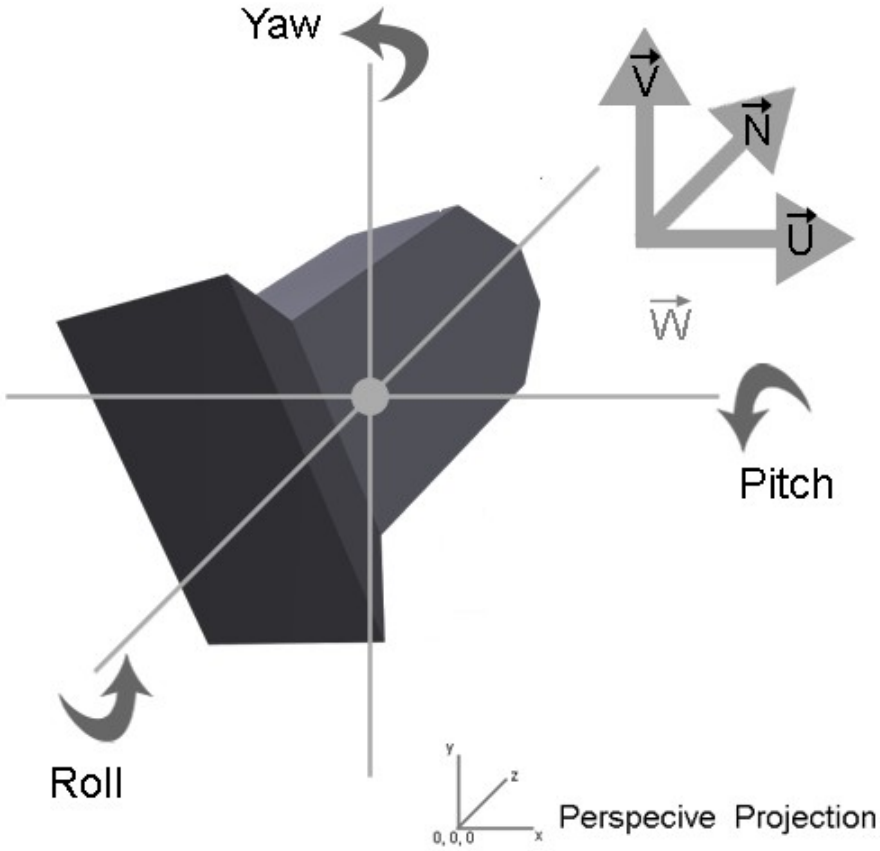


Fig 12: An illustrated definition of the camera object in 3D world coordinate space

The above diagram illustrates the camera entity in 3D world space. The position of the camera is represented by the three vectors: U, V, N, with the forth vector W being used to represent the camera position in world coordinate space. The rotation of the camera is performed along the x y z axis (positive and negative), where such rotations are referred to as “yaw”, “pitch” and “roll” (or more traditionally as “azimuth“, “elevation“ and “roll“).

While gluPerspective (which is used to setup the volume of the viewing frustum along with the aspect ratio of the viewport projection matrix) is called only once when the application is setup, gluLookAt() is used to modify all of the perspective orientation parameters every frame. However, in order to allow the user control of these parameters, new values must be updated every frame. The camera coordinate values are updates using user input (as mentioned above).

Now it's time to examine the camera class functionality in more detail.

The three main functions that provide the user with control over the camera object are:

RotateView()
StrafeCamera()
MoveCamera()

With keyboard and mouse input function being implemented using the functions:

SetViewByMouse()
CheckForMovement()

And the gluLookAt setup and update functions being implemented as:

Update()
Look()

Below are detailed descriptions of each the member functions.

MoveCamera()

```
void Camera::MoveCamera(GLfloat speed)
{
      Vector3 vVector = m_vView - m_vPosition;

      vVector = vVector.Normalize(vVector);

      m_vPosition.x += vVector.x * speed;
      m_vPosition.z += vVector.z * speed;
      m_vView.x += vVector.x * speed;
      m_vView.z += vVector.z * speed;
}
```

The MoveCamera() function allows the user to move the camera along the N camera vector (defined as an equated increment in the users camera position and view vectors). The general way of computing movement for the camera is defined as:

*Camera Vector Position += Camera Forward Vector * Camera Speed Constant*

This allows the user to move the camera backwards and forwards. As the camera vector position is incremented (either by a positive of negative camera speed constant value, in which case it's decremented automatically), the camera's view vector is also updated using the same mechanism, thus the user is able to view examine the scene by moving forward and backward.

StrafeCamera()

```
void Camera::StrafeCamera(GLfloat speed)
{
      m_vPosition.x += m_vStrafe.x * speed;
      m_vPosition.z += m_vStrafe.z * speed;

      m_vView.x += m_vStrafe.x * speed;
      m_vView.z += m_vStrafe.z * speed;
}
```

The strafe camera function allows the user to move along the U camera vector, once again defined in the same manner as the implementation of the position and view vectors found in the MoveCamera() function. Except this time, a new strafing vector is used instead of the default movement vector.

The equated incrimination along the U camera axis allows the user to move the camera left and right, thus giving the user greater control over the examination of the currently rendered scene.

RotateView()

```
void Camera::RotateView
(GLfloat angle, GLfloat x, GLfloat y, GLfloat z)
{
      Vector3 vNewView;

      Vector3 vView = m_vView - m_vPosition;

      GLfloat cosTheta = (GLfloat)cos(angle);
      GLfloat sinTheta = (GLfloat)sin(angle);

vNewView.x  = (cosTheta + (1 - cosTheta) * x * x) * vView.x;
vNewView.x += ((1 - cosTheta) * x * y - z * sinTheta)* vView.y;
vNewView.x += ((1 - cosTheta) * x * z + y * sinTheta)* vView.z;

vNewView.y  = ((1 - cosTheta) * x * y + z * sinTheta)* vView.x;
vNewView.y += (cosTheta + (1 - cosTheta) * y * y)    * vView.y;
vNewView.y += ((1 - cosTheta) * y * z - x * sinTheta)* vView.z;

vNewView.z  = ((1 - cosTheta) * x * z - y * sinTheta)* vView.x;
vNewView.z += ((1 - cosTheta) * y * z + x * sinTheta)* vView.y;
vNewView.z += (cosTheta + (1 - cosTheta) * z * z)    * vView.z;

      m_vView = m_vPosition + vNewView;
}
```

Rotations in 3D space are usually defined using Euler angles. It is necessary to compute rotations in the x y z axis in order for the user to be able to orientate the camera in the 3D scene. Essentially, the RotateView() function is a high level implementation of 3D matrix transformations applied to the corresponding view and position vectors.

Rotation in the x y z axis can be defined by 3 separate 4x4 matrices, making use of homogenous coordinates:

$$R_{\theta_x,\vec{i}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta_x) & \sin(\theta_x) & 0 \\ 0 & -\sin(\theta_x) & \cos(\theta_x) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Fig 13: The 4x4 homogenous matrix for rotation about the x axis

$$R_{\theta_y,\vec{j}} = \begin{pmatrix} \cos(\theta_y) & 0 & -\sin(\theta_y) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta_y) & 0 & \cos(\theta_y) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Fig 14: The 4x4 homogenous matrix for rotation about the y axis

$$R_{\theta_z,\vec{k}} = \begin{pmatrix} \cos(\theta_z) & \sin(\theta_z) & 0 & 0 \\ -\sin(\theta_z) & \cos(\theta_z) & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Fig 15: The 4x4 homogenous matrix for rotation about the z axis

A very important note to mention (again) at this point is that OpenGL uses column major matrices, therefore when implementing any sort of 4x4 homogenous matrix transformation into OpenGL, it has to be transposed first. As it can be seen from the three different rotation matrices above, rotations along the x y z axis in 3D space are rather straightforward (in same comparison to performing rotations in 2D space using 3x3 homogenous matrices).

However, in order to perform rotations by a given angle along any axis, make use of a normalized (unit) position vector (obtaining the position vector will be described below), that is multiplied with the 4x4 homogenous coordinate rotation matrix for unit vector rotation about any given axis passing through the origin.

The position vector, can for now, be defined as :

$$\hat{v} = \alpha\,\vec{i} + \beta\,\vec{j} + \gamma\,\vec{k} \;\; passing\,through\,origin(0,0,0)$$

Where alpha, beta and gamma represent the x y z values of the given unit vector. Therefore, the rotation about the axis given by the unit vector passing though the origin can be defined as:

$$\begin{pmatrix} x' & y' & z' & 1 \end{pmatrix} = \begin{pmatrix} x & y & z & 1 \end{pmatrix} R_{\theta,\hat{v}}$$

The rotation matrix for unit vector rotation about any given axis passing through the origin, can be defined as (note that the proof is not provided):

$$R_{\theta_{x,\vec{i}}} = \begin{pmatrix} \alpha^2(1-\cos(\theta))+\cos(\theta) & \alpha\beta(1-\cos(\theta))+\gamma\sin(\theta) & \alpha\gamma(1-\cos(\theta))-\beta\sin(\theta) & 0 \\ \alpha\beta(1-\cos(\theta))-\gamma\sin(\theta) & \beta^2(1-\cos(\theta))+\cos(\theta) & \beta\gamma(1-\cos(\theta))+\alpha\sin(\theta) & 0 \\ \alpha\gamma(1-\cos(\theta))+\beta\sin(\theta) & \beta\gamma(1-\cos(\theta))-\alpha\sin(\theta) & \gamma^2(1-\cos(\theta))+\cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The implementation of the 4x4 rotation matrix can be seen in the code excerpt provided.

SetViewByMouse()

The most important part of the SetViewByMouse() function is defined below:

```
if(currentRotX > 1.0f)
{
      currentRotX = 1.0f;
}
else if(currentRotX < -1.0f)
{
      currentRotX = -1.0f;
}
else
{
      Vector3 vAxis = vAxis.Cross(m_vView - m_vPosition,   m_vUpVector);
      vAxis = vAxis.Normalize(vAxis);

      RotateView(angleZ, vAxis.x, vAxis.y, vAxis.z);
      RotateView(angleY, 0, 1, 0);
}
```

As mentioned above, concerning the 4x4 rotation matrix, the axis vector around which the rotation occurs must be of unit length. What the SetViewByMouse() function is that it computes the axis vector by finding the cross product between the difference of the current view and position vectors, and the current up vector. The value returned by the cross product is then assigned to the axis vector, before it gets normalized.

At this point, rotation along the Z (yaw) and Y (pitch) axis can occur (rotation around the X axis is not computed). Rotation in the x axis is clamped to a unit length of 1.0f or -1.0f (in radians).

CheckForMovement()

This function simply checks if the user has pressed a certain key on the keyboard that corresponds to a particular event in the application. Such events include camera movement and strafing, as well as things like toggling wireframe and fog rendering.

Truncated example below:

```
if(GetKeyState(VK_UP) & 0x80 || GetKeyState('W') & 0x80)
{
      MoveCamera(C_SPEED);
}
if(GetKeyState(VK_LEFT) & 0x80 || GetKeyState('A') & 0x80)
{
      StrafeCamera(-C_SPEED);
}
if(GetKeyState('Q') & 0x80)
{
      m_vPosition.y += 0.1;
}
if(GetKeyState('V') & 0x80)
{
      glPolygonMode(GL_FRONT, GL_LINE);
}
if(GetKeyState('F') & 0x80)
{
      glEnable(GL_FOG);
}
```

Look()

```
void Camera::Look()
{
      gluLookAt(m_vPosition.x, m_vPosition.y, m_vPosition.z,
              m_vView.x,       m_vView.y,     m_vView.z,
              m_vUpVector.x, m_vUpVector.y, m_vUpVector.z);
}
```

The camera class in it's most basic essence is just an advanced functionality layer added on top of the gluLookAt() function. All of the vector calculation values associated with the camera class, which are generated every frame, are plugged into the gluLookAt() function, which in turn updates the viewing frustum every time the scene is drawn (per frame).

Update()

```
void Camera::Update()
{
      Vector3 vCross = vCross.Cross(m_vView - m_vPosition, m_vUpVector);

      m_vStrafe = m_vStrafe.Normalize(vCross);

      SetViewByMouse();
      CheckForMovement();
}
```

Finally, the camera update function is called every time before the scene gets drawn. The update function normalizes the strafing vector and then calls SetViewByMouse() and CheckForMovement().

Before the strafing vector is normalized, as in with the vector rotation, a unit vector is computed (note that the cross product is calculated, before being normalized, very time the camera is updated. This is the same routine used to update camera rotations every frame).

Terrain Class

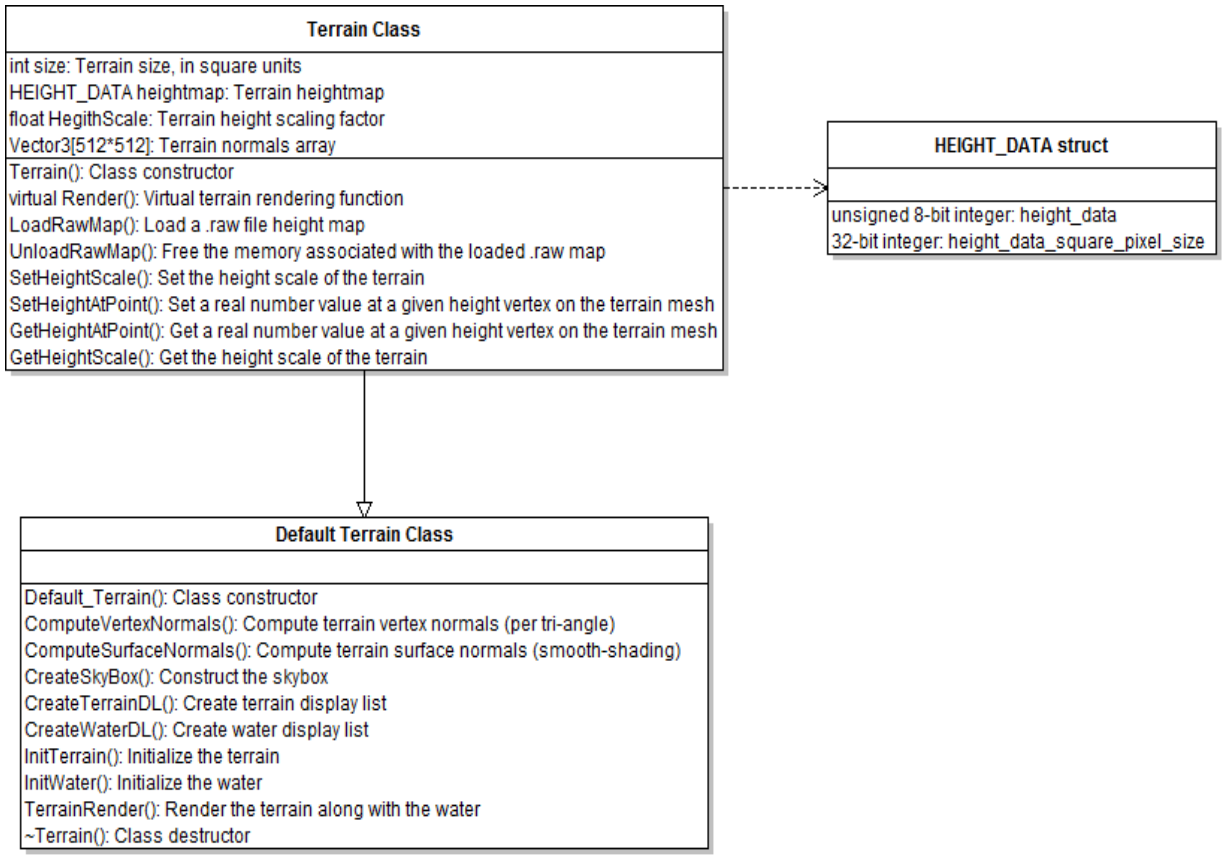


Fig 16: The Terrain class

The terrain class is made up of three main components. The first component is the HEIGHT_DATA struct used to store the data that is loaded in from a an 8-bit 256 grayscale colour RAW image file. The RAW image file is used to construct the terrain data.

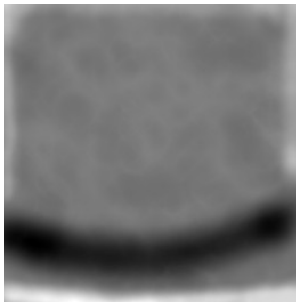


Fig 17: The 256 grayscale image used as the heightmap

The terrain HEIGHT_DATA struct is defined below.

```
struct HEIGHT_DATA
{
      unsigned char* m_Data;
      int m_Size_;
};
```

The terrain size is set to a size of 256x256 pixels. In the implementation of the terrain rendering code in the OpenGL coursework application, the size of the terrain data is very important.

The reason for this is that each vertex of the triangle strip that makes up the terrain mesh is based on the gradient value calculated from the heightmap. The heightmap is stored as a one dimensional array. The one dimensional array is used to iterate thru and to index the vertex values of the z coordinate of the terrain mesh.

The value (aka height) of each of the vertices of the terrain mesh is calculated as follows:

```
int index = (z * m_Size) + x;
```

With this in mind, the terrain mesh is then constructed using the following mechanism:

```
glNewList(TERRAIN_LIST, GL_COMPILE);
{
	glPushMatrix();

	for(int z = 0; z <= m_Size; ++z)
	{
		glBegin(GL_TRIANGLES);
		{
			for(int x = 0; x <= m_Size; ++x)
			{
				int index = (z * m_Size) + x;

				//for each set of neighbouring triangles
				glVertex3f(x, m_heightData.m_Data[index] *
		m_heightScale, z);
			}
		}
		glEnd();
	}
	glPopMatrix();
}
glEndList();
```

For efficiency purposes, the terrain rendering construction code makes use of OpenGL display lists. This prevents the program from having to compute the mesh vertex, normal (assigning the already precomputed terrain surface normal values) and texture coordinates on a per-frame basis. Rather, the display list is called once every frame, and the precomputed terrain mesh construction is poped off the display list stack.

The two main algorithm implementations found in the terrain class are the ComputeNormals() and ComputeVertexNormals() functions. These function implement the MWE algorithm in order to calculate the surface and vertex normals for each of the triangle faces, in order to make use hardware lighting on the terrain mesh. Described in the section below are detailed algorithm and implementation details of the MWE algorithm.

Hardware accelerated 3D terrain surface lighting

Traditionally, using graphics hardware and/or the CPU to compute lighting on complex meshes was a very costly operation in terms of performance. It wasn't until the introduction of graphics cards that featured transformation and lighting capabilities that it became a viable solution to use hardware accelerated lighting for shading the surfaces of game geometry.

Very detailed and large game geometry (detailed in terms of the number of triangles it features), was usually used sparsely, and when rendered, it was rendered with minimal overhead cost in terms of performance. This meant that features like multitexturing and dynamic LOD (level of detail generation), was employed, in order to save computing resources reserved for computing other game logic/rendering calculations.

However, today that is no longer the case, and making use of hardware accelerated lighting, even on the deprecated OpenGL FFP, has become a trivial matter (although implementing hardware lighting computations for surface normals using the standard OpenGL FFP is rather tedious).

Implementing the MWE normal vector calculation algorithm

The computation of the MWE algorithm is broken up into tow different stages.

Stage 1: Computation of surface normals.

$$\vec{N}_{surface} = \vec{A} \, X \, \vec{B}$$

The computation of the surface normal plane is calculated by finding the cross product between the two vectors A and B that lie in the plane. The result is a vector cross product that is then normalized.

Stage 2: Computation of the vertex normals

$$\vec{N}_{NWE} = \sum_{i=0}^{i=5} \left(\frac{\vec{N}_{si}}{6} \right)$$

Computing the surface normals then involves finding the sum of all the surface normals the given vertex is part of, and dividing it by 6. The range of the surface normals is from 1 to 5. The reason behind this is that for most constructed terrain meshes made up of triangles based on heightmap data, there will always be 6 vertices connected to one central vertex. This is due to the fact that the triangles are all facing the same direction, between the two sets that constructed in the terrain mesh construction loop.

Below is an illustration to clarify this concept:

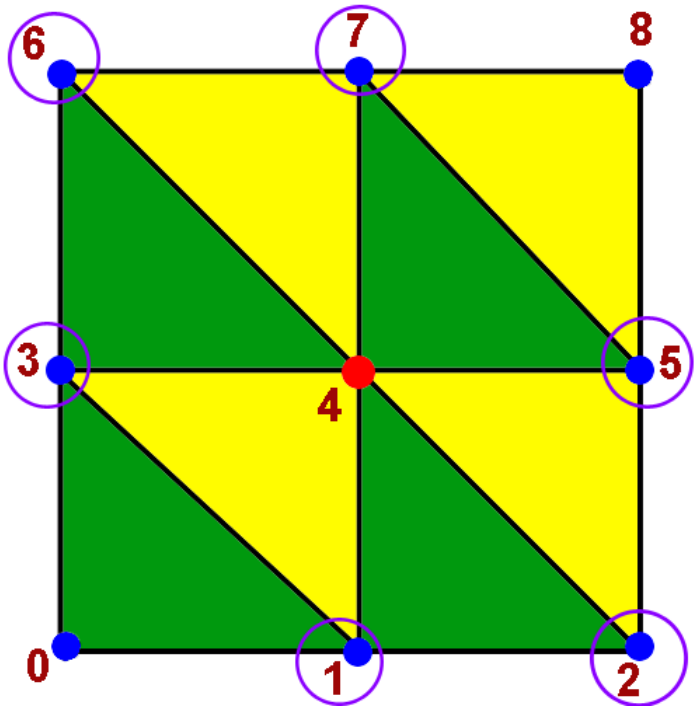


Fig 18: A set of four surface quad segments of the terrain mesh

Now it's time to see how these two algorithms were implemented. The first algorithm to be implemented was the ComputeNormals() algorithm, which was used to compute the surface normals of the terrain.

The most important computation by performed this function is the computation of the cross product between the two vectors A and B which lie in the plane of the given surface. The given surface is made up 2 faces, which each share two common vertex points. An example of a quad would be the combination of face C with vertices 1, 4 and 2, and face D, with vertices 2, 4, and 5. The cross product between the vectors of these two faces in computed and then normalized. This in turn computes the unit normal for every given surface quad in the terrain mesh.

An example of the computation would be as follows:

It is assumed that a quad made up of two triangle is used.

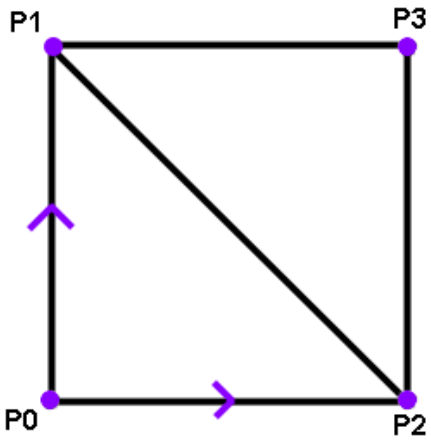


Fig 19: A quad

Then, the iteration for each quad of the terrain mesh is computed as follows:

Points of lower left triangle are defined as:

$$P_0=[i, h_{0,} j] \quad P_1=[i, h_{1,} j+1] \quad P_2=[i+1, h_{2,} j]$$

where $\vec{A}=P_1-P_0$ and $\vec{B}=P_2-P_0$

and $\hat{N}_0=\frac{(\vec{A}\,X\,\vec{B})}{\|(\vec{A}\,X\,\vec{B})\|}$

=> calculating vector A for all lower left triangles:

$$\sum_{j=0}^{j=vertDeep}\left[\sum_{i=0}^{i=vertWide}\vec{A}_{ij}\right]$$

$$\sum_{j=0}^{j=vertDeep}\left[\sum_{i=0}^{i=vertWide}\vec{B}_{ij}\right]$$

$$\sum_{j=0}^{j=vertDeep}\left[\sum_{i=0}^{i=vertWide}\hat{N}_{0\ ij}\right]$$

For very top right angle in surface quads, the process is repeated, except the new points are now iterated through as:

$$P_0 = [i+1, h_{0,} j]$$
$$P_1 = [i, h_{1,} j+1]$$
$$P_2 = [i+1, h_{2,} j+1]$$

where $\vec{A}=P_1-P_2$ and $\vec{B}=P_2-P_0$

and $\hat{N}_1=\frac{(\vec{A}\,X\,\vec{B})}{\|(\vec{A}\,X\,\vec{B})\|}$

The code implementation for the above algorithm is as follows:

```
void DEFAULT_TERRAIN::ComputeNormals()
{
	int normalIndex = 0;

	for (int z = -1; z <= m_Size - 1; ++z)
	{
		for (int x = -1; x <= m_Size - 1; ++x)
		{

		// Set/Compute the normal for triangle 1
		float height0 = GetHeightScale( x,    z );
		float height1 = GetHeightScale( x,  z+1 );
		float height2 = GetHeightScale( x+1,  z );

	Vector3 normal1( height0 - height2, 1.0f, height0 -height1 );

		// Set/Compute the normal for triangle 2
		height0 = height2;
		height2 = GetHeightScale( x+1, z+1 );

	Vector3 normal2( height1 - height2, 1.0f, height0 - height2 );


		normal1 = normal1.Normalize(normal1);
		normal2 = normal2.Normalize(normal2);


		// Set the surface normals in the array
		m_TerrainNormals[normalIndex++] = normal1;
		m_TerrainNormals[normalIndex++] = normal2;

		}
	}
}
```

There are slight adjustments in the implementation of the algorithm. The first adjustment is that the iteration takes place from I and J values (set and z and x) from -1 to the size of the terrain mesh, instead of 0 to the size of the terrain mesh. The reason for this is that if the area of the height field computed is 1 unit larger then that of the original height field (this the one extra iteration), all of the quads, including the ones on the edge of the terrain mesh, will have their surface normals computed. This extra iteration is also necessary in order to compute the vertex normals for the quads.

The second adjustment comes from calculating the top right triangle of the quads. Note that only the height points 0 and 2 are set. The reason for this is that because the triangles that make up the quad do not change their direction, the iterative value for P1 of each computed quad does not need to be adjusted.

Once the surface normals are computed for each of the quads on the terrain mesh, the result of the normals calculated used for the surface lighting looks like this:

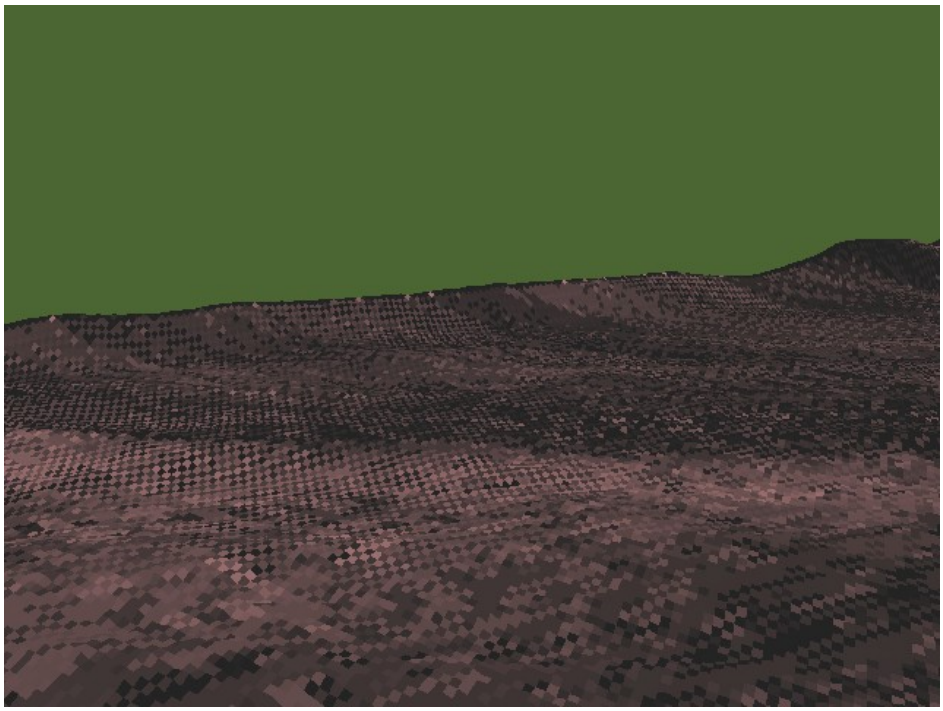


Fig 20: Using GL_FLAT to display the surface normals (no surface vertex normals calculated)

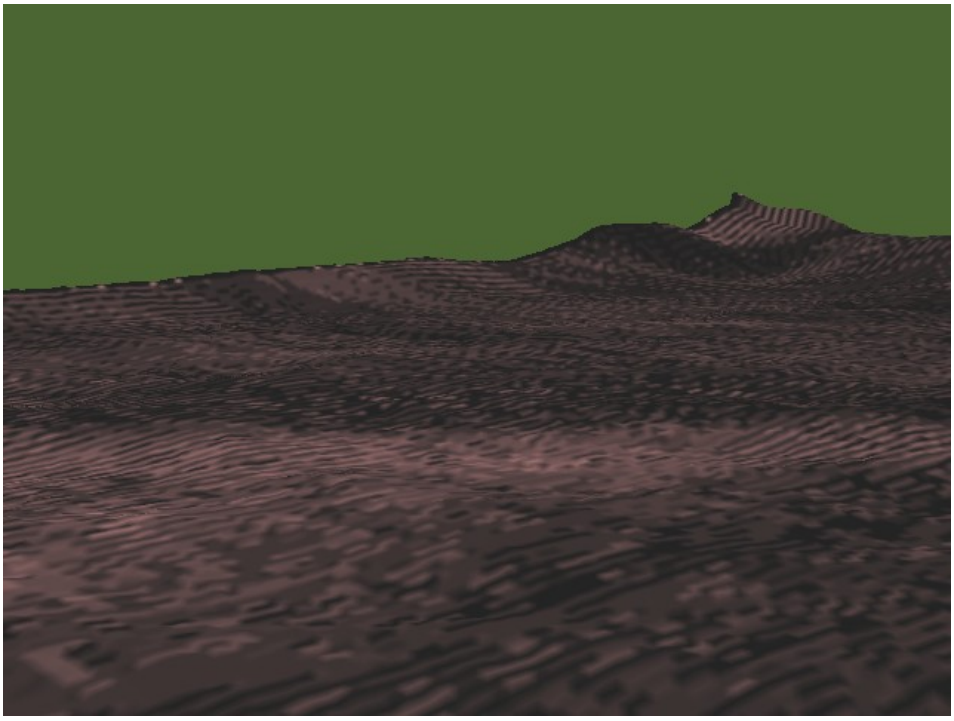


Fig 21: Using GL_SMOOTH to display the surface normals (no surface vertex normals calculated)

As it can be seen from the above pictures, the terrain shading produced does not interpolate the vertex normals for the common vertex point of each of the quads in the terrain mesh. Even when using GL_SMOOTH, the result does not look very good. In order to fix this, the vertex normal for each of the quad surfaces must be computed, which will allow for the surface shading of the terrain mesh to be interpolated correctly.

At this stage, computing the vertex normals involves implementing the algorithm for adding the 6 adjacent surface normals of the common vertex, and dividing them by 6. The important aspect of this algorithm is determining which given set of 6 surface faces is adjacent to the current triangle vertex, which is shared as a common vertex between those 6 neighbouring faces. The algorithm is called The Sliding Six Algorithm, and is defined as follows (based on the provided diagram):

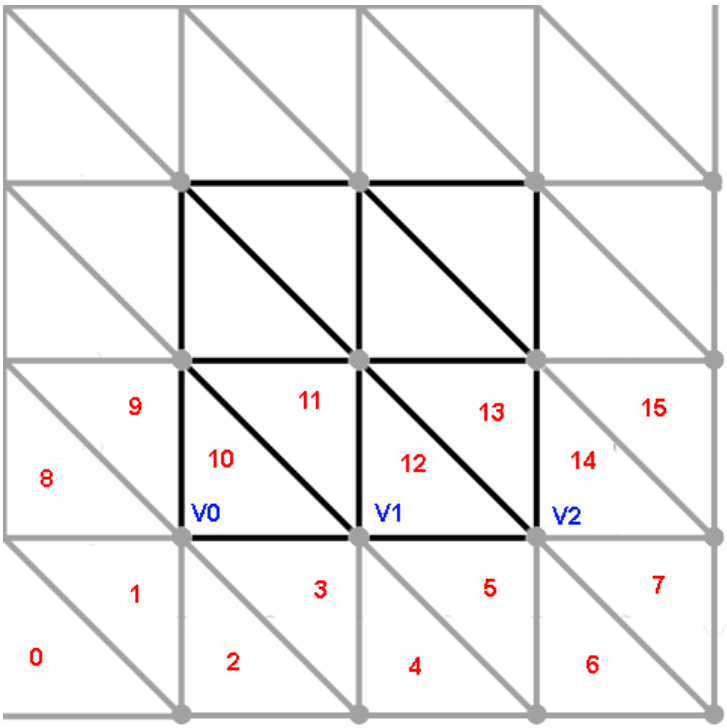


Fig 22: Example of a set of quads with an extra row and column of quads calculated around them

The solved algorithm can be used to iterate through the given set of common vertices shared between any 6 faces:

$$\sum_{vertices=0}^{vertices=2} f(vertices)$$

where the vertices are the main common points between the relative faces, labeled as $V_{0,}V_1$ and V_2 and where

$$f(vertices)=\frac{\sum_{n=0}^{n=5}(face_n+face_{n...}\,n+1=5)}{6}$$

From the above example, it can then be concluded that each row of triangles is sequential from one common vertex to the next (as V0 shares a common vertex point with faces 1 to 3, and V1 shares a common vertex point with faces 3 to 5). This algorithm can then implemented to calculate the unified vertex normals for each of the quad faces of the triangle mesh.

```
void DEFAULT_TERRAIN::ComputeVertexNormals()
{
      Vector3 vec[6];
      Vector3 vertexNormal;
      int triIndex     = 1;
      int indexPlusOne = 2;
      int indexPlusTwo = 3;
      int vertIndex    = 0;

      // Computes the offset based on the number of triangles across
      int rowOffset = ( ( m_Size + 1 ) * 2 ) - 1;
      for( int z = 0; z < m_Size; z++ )
      {
            for( INT x = 0; x < m_Size; x++ )
            {
                  indexPlusOne = triIndex + 1;
                  indexPlusTwo = triIndex + 2;

                  // Get the three triangles below the vertex
                  vec[0] = m_TerrainNormals[ triIndex ];
                  vec[1] = m_TerrainNormals[ indexPlusOne ];
                  vec[2] = m_TerrainNormals[ indexPlusTwo ];

                  // Get the three triangles above the vertex
                   vec[3] = m_TerrainNormals[ rowOffset + triIndex ];
               vec[4] = m_TerrainNormals[ rowOffset + indexPlusOne ];
               vec[5] = m_TerrainNormals[ rowOffset + indexPlusTwo ];

            // Sum the vectors and then divide by 6 to average them
            vertexNormal = (vec[0] + vec[1] + vec[2] + vec[3] +
      vec[4] + vec[5]) / 6;

            vertexNormal = vertexNormal.Normalize(vertexNormal);

                  // Assign the normals to the stored vertex normals
                  m_TerrainNormals[vertIndex] = vertexNormal;

                  // Increment the triangle and vertex indices
                  triIndex += 2;
                  vertIndex++;
            }
            triIndex += 2;
      }
}
```

Note that the row offset for each of the calculated vertex normals is defined as:

```
int rowOffset = ( ( m_Size + 1 ) * 2 ) - 1;
```

The reason for this is that the defined triangleIndex counter is used to iterate through the given terrain quad faces, moving right and indexing three triangle faces of each row of triangles, which define the bottom three faces of the common vertex point that is shared between a given set of 6 faces. However, the top three faces of this given set need to be accounted for as well, therefore, the above rowOffset counter is used to index the upper three quad surface faces of the set of 6 faces that share a common vertex.

The end result is a dramatic improvement in the shading of the terrain mesh surface:

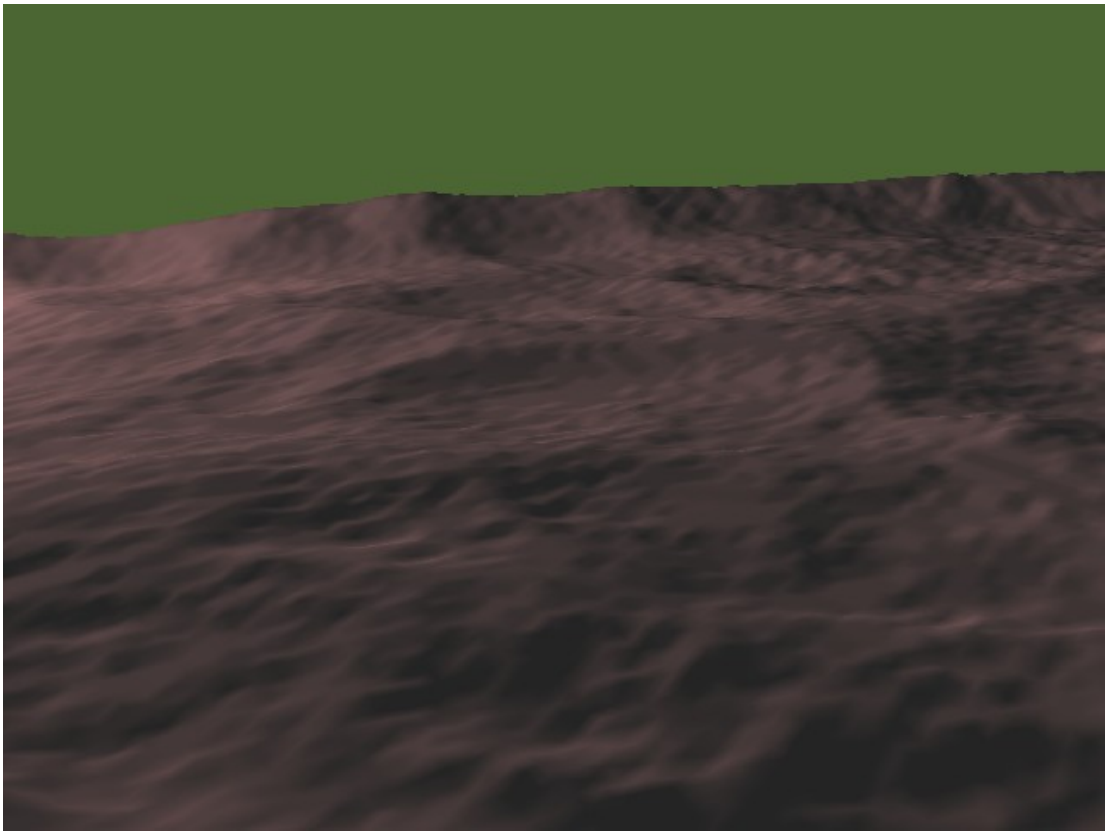


Fig 23: The final result of the MWE algorithm implementation. Here, both the surface normals and the surface vertex normals are calculated.

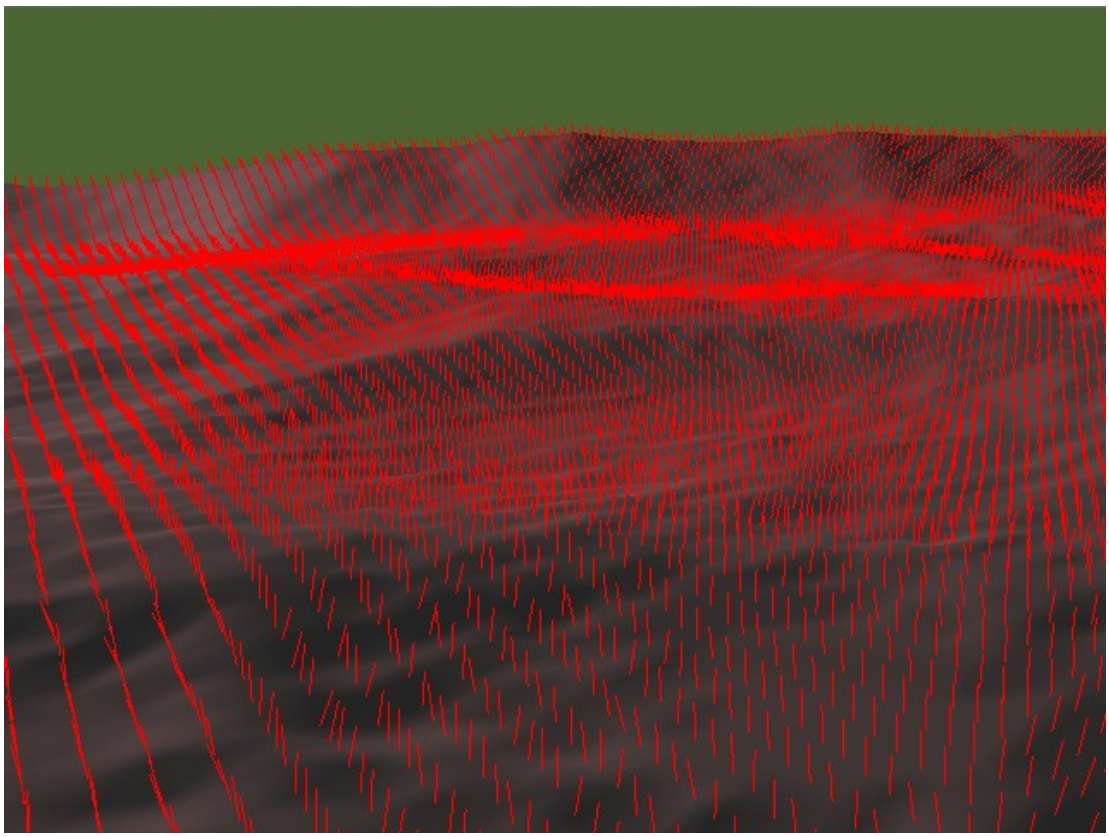


Fig 24: Displaying the vertex normals of each of the common vertices shared by each set of six faces.

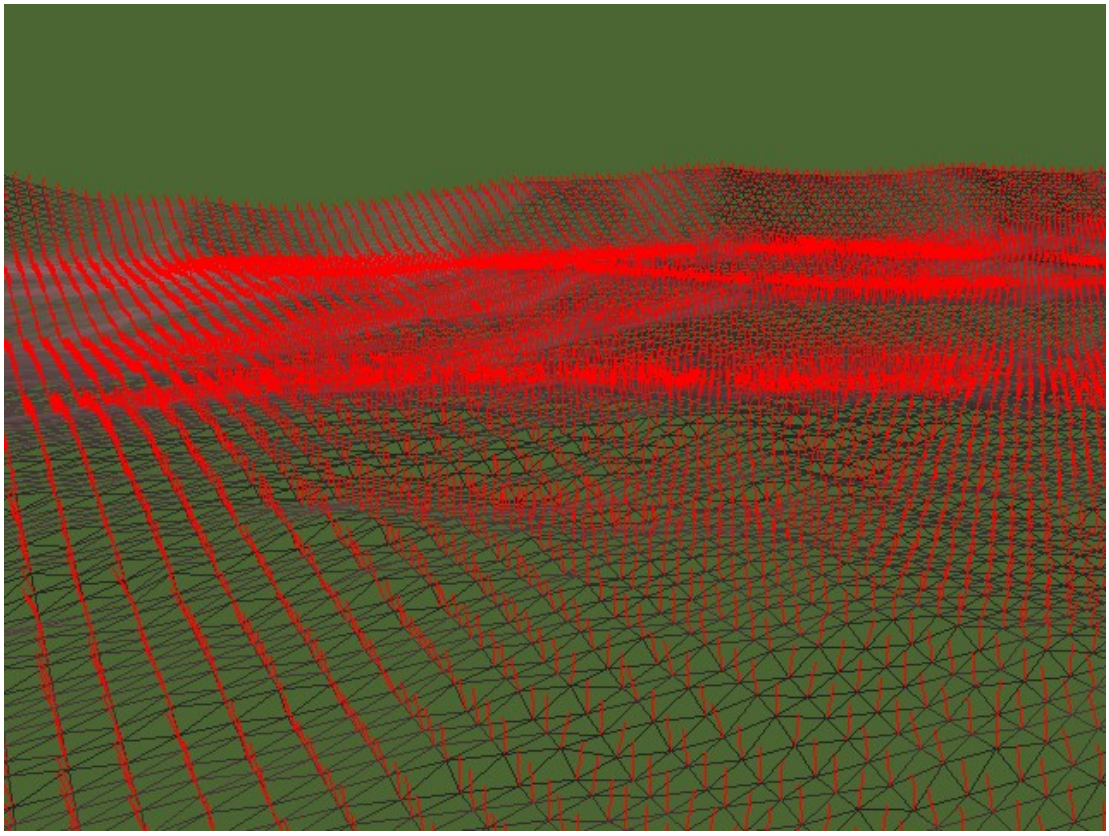


Fig 25: A more clearer example of the above, using wireframe display.

Rendering the final scene

This section will briefly outline each of the other components of the application which contribute to the final interactive 3D scene.

Setting up and rendering the scene

The final rendered OpenGL scene is setup with the following parameters:

```
//Scene rendering setup
glShadeModel(GL_SMOOTH);
glClearColor(0.3f, 0.4f, 0.2f, 1.0f);
glClearDepth(1.0f);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LEQUAL);
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
glEnable(GL_LIGHTING);
glEnable(GL_TEXTURE_2D);
glEnable(GL_CULL_FACE);
//lighting 1 setup
glLightfv(GL_LIGHT1, GL_AMBIENT,  Light_Ambient);
glLightfv(GL_LIGHT1, GL_DIFFUSE,  Light_Diffuse);
glEnable(GL_LIGHT1);

//Setup the fog
glFogi(GL_FOG_MODE, GL_EXP2);
glFogfv(GL_FOG_COLOR, fog_color);
glFogf(GL_FOG_DENSITY, fog_density);
glHint(GL_FOG_HINT, GL_LINEAR);

//Setup the texturing parameters
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
```

The basic setup of these scene above includes most of the active parameters used by OpenGL. Some of the parameters are enabled and disabled depending in what other functions and context they are called.

The setup of the scene does the following:

- Enables smooth shading of all the rendered 3D geometry in the scene (per-vertex lighting)
- Enables frustum culling for backfaces of rendered triangles
- Enabled lighting
- Enables texturing
- Enables simple linear fog
- Enables depth testing (render order sorting)
- Enables linear filtering of applied 2D texture maps (making use of mipmapping).

The fog and lighting in the scene is setup with the following parameter values:

```
//Lighting global parameter variables
GLfloat Light_Ambient[] = {0.4f, 0.4f, 0.4f, 1.0f};
GLfloat Light_Diffuse[] = {0.6f, 0.4f, 0.4f, 1.0f};
GLfloat Light_Position[] = {0.0f, 1000.0f, -200.0f, 1.0f};

//Fog global parameter variables
GLfloat fog_density = 0.008f;
GLfloat fog_color[4] = {0.4f, 0.2f, 0.2f, 0.3f};
```

This gives the lighting colour a rusted red/murky orange shading colour, which is used to apply shading to the surfaces of the geometry in the rendered 3D scene.

The fog parameters create a dark red linear fog colour, that is of average thickness.

Using Skyboxes

One final part of the coursework that was implemented as a feature was skybox rendering. Making use of skyboxes allows rendering scenes to present the user with perceived depth and perspective, making the scene look bigger (by making it look like it is a smaller part of something).

A skybox is a six sided cube that is textured from the inside. It is not lit, and outside faces of it are not rendered. Since the skybox is projected from the origin of where it placed, it can be of unit size. The way the skybox is projected in the final rendered scene is the most important aspect of the skybox implementation.

The skybox is drawn before the camera transformations are calculated, and before the rest of the scene is drawn. What this in turn produces is an effect in which the skybox will always be drawn in the background. Since the GL_DEPTH_TEST is disabled for skybox rendering, the z-depth of the skybox is always calculated at a constant value, therefore it is either always projected on top of everything in the scene (at the near plane of the viewing frustum, with the z depth value set to 0), or behind everything else, with its depth still set to 0.

Finally, use the GL_CLAMP extension parameter is used to clamp the texture edges of the skybox to each of the cube faces. This prevents from seams showing up at the edges of the skybox texture.

One final note to mention, is that the camera Loo() function must be called every time the skybox is drawn. This is to make sure the camera is always looking at the skybox. Then, the skybox is translated around the current camera position of the user. This allows the user to view the skybox from all sides and angles.

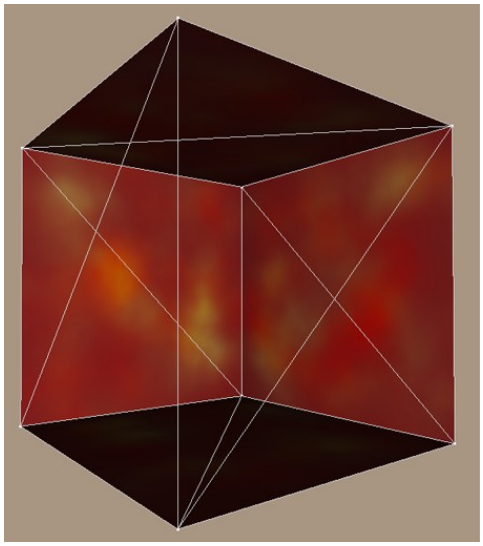


Fig 26: An example of a skybox.

Below is a truncated example of the implementation of the skybox:

```
void DEFAULT_TERRAIN::CreateSkyBox()
{
	glDisable(GL_LIGHTING);
	glDisable(GL_DEPTH_TEST);
	glDisable(GL_CULL_FACE);

	glPushMatrix();

	glLoadIdentity();

	//The tricky bit...
	//First, make sure the camera is always looking at the skybox
	g_Camera.Look();



	//Then  translate the skybox around the current camera position
	//Disabling the depth testing will then make the skybox seem
	//infinitely large (see below In DrawScene())
	glTranslatef(g_Camera.GetPosX(),g_Camera.GetPosY(),  g_Camera.GetPosZ());

	//Set the skybox colour to white
	glColor3f(255.0f, 255.0f, 255.0f);

	//for each of the skybox faces
	//bind the relative texture
	//construct a quad (of unit length)


	glPopMatrix();

	//glDisable(GL_TEXTURE_2D);//temp,    remove    once    terrain    gets
	//textured
	glEnable(GL_LIGHTING);
	glEnable(GL_DEPTH_TEST);
	glEnable(GL_CULL_FACE);
}
```

Coursework Review and Critical Analysis

This section outlines the final review and critical analysis of the produced coursework.

Shortcomings

The application does not feature multitexturing, as it was originally planned. The reason for this this is that working with the extensions needed by OpenGL to perform multitexturing was rather complicated to implement. The complicated part involved setting up GLEW to handle the needed extensions. This caused numerous compile errors, and when the program did finally compile, no textures were rendered. It was then decided to rather focus on clearly understanding and implementing the basic functional features of the coursework application, rather then spending time on something that is not really necessary for the final application.

The second shortcoming was the lack of collision detection. Originally, spherical bounding volumes were to be implemented for collision checking between the users camera and scene geometry. However, due to time constraints associated with implementing the MWE algorithm, the implementation of collision detection was neglected.

Future work

The future work on the project will include the following feature additions/improvements:

- Implementation of a game factory class for a cleaner project setup and OOP functionality.
- Implementation of spherical collision between the users camera and scene geometry.
- Fully re-written rendering code that will completely replace the deprecated FFP rendering code with GLSL programmable shading functionality.
- Shader based multitexturing.
- Shader based per-pixel lighting (implementation of a true Phong lighting model).
- Larger terrains (1024x1024 heightmap)
- Implementation of the ROAM rendering algorithm for dynamic terrain LOD adjustment.

Conclusion

The design and development of the second semester OpenGL coursework application was a very steep learning curve that proved to be a fruitful learning experience. Much has been learned about the use of OpenGL and object orientated programming, as well dealing with larger projects with multiple files.

The work done for and lessons learnt from this coursework act as stepping stones towards a path in becoming a professional in real time 3D graphics programming.

Appendix A - References

OpenGL Architecture Review Board. 1997. *The Official Guide to Learning OpenGL 1.1, Second Edition.* Addison-Wesley Publishing Company.

Wright, R. S, Jr. Lipchak, B. Haemel, N. 2007. *OpenGL SuperBible, Fourth Edition*. Addison-Wesley Publishing Company.

Angel, E. 2000. *Interactive Computer Graphics: A Top-Down Approach With OpenGL*. Addison-Wesley Publishing Company.

Van Verth, M. J. Bishop, M. L. 2008. *Essential Mathematics for Games & Interactive Applications: A Programmer's Guide, Second Edition.* Morgan Kaufmann.

DeLoura, M. 2000. *Game Programming Gems.* Charles River Media.

Polack, T. 2002. *Focus on 3D Terrain Programming.* Premier Press.

LaMonthe, A. 2002. *Tricks of the Windows game programming gurus, Second Edition*. Sams Publishing.

Walsh, J. 2005. Normal Computations for Heightfield Lighting. Available from: http://www.gamedev.net/reference/articles/article2264.asp [Accessed 9 May 2010].

Lucas, N. T. 2009. CS0880a Applied Mathematics 2: Mathematical Methods - Part 1. Available from: http://portal.abertay.ac.uk/learning/module/co/cs/cs0880a/Methods%202/PDF%20Notes%20Part%201/ [Accessed 9 May 2010].

Ben Humphrey. 2002. *Camera Part 5.* Available from: http://www.gametutorials.com/gtstore/pc-11-1-camera-part5.aspx [Accessed 9 May 2010].