Report on The Rendering of a 3D Scene depicting the City of Edinburgh using various rendering methods and techniques

Jenkinson, Matthew *
Edinburgh Napier University
Computer Graphics (SET08116)

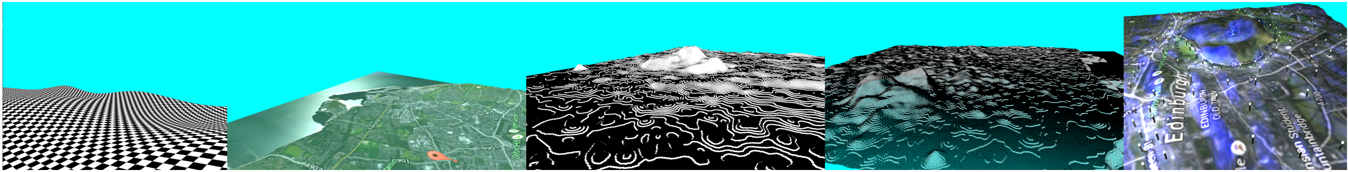


Figure 1: *Images left to right: 2 Sine waves used to test the plane builder before using maps, A demonstration of specular lighting only used on the water (global illumination), Using the vertex shader to determine the colour based on the angle of the vector, Using the wire-frame rendering mode on a fully specular black mesh (most of the work done is in the shader), A near-finished version including street-lights (not to scale).*

Abstract

The aim of this article is to describe the process of creating a 3D scene using OpenGL, within this, multiple techniques such as different types of shading will be covered. As such it may be used as a guide for beginners attempting to achieve a similar goal and provide some insight to the average computer user as to how graphics are rendered using a computer.

This is intended to be a simple overview and not a comprehensive guide.

This particular project focusses on creating a 3D rendition of Edinburgh using a height map then applying various lights and meshes to the geometry.

Keywords: ambient, attenuation, diffuse, emissive ,fragment, geometry, global illumination, gouraud, lighting, map, mesh, phong, pixel, plane, point, polygon, scene, shading, specular, spot, texture, vector, vertex

1 Introduction

Before the advent of Google Maps and Google Earth there were no easily accessible (and traversable) 3D models of The Earth's terrain. This was not so long ago, at the time however there were maps that contained height data. Unfortunately until recently computers' graphics cards have not been so powerful (relatively) so rendering such scenes in high detail was impossible. Now we can do this processing on our mobile phones!

The motivation behind this project is to show how trivial it can be to create such a scene in modern times, and to celebrate this luxury of powerful computing by adding many lights of different styles.

The effects that this program uses are as follows:

- Using a height map to generate terrain.
- Interpolation.
- Multiple textures.
- Materials.
- The GL_MIRRORED_REPEAT wrapping technique.
- Multiple meshes (generated using different methods) within the geometry.
- Different glPolygon modes.
- Multiple cameras including a free-camera.
- Vertex shading.
- Fragment (per-pixel) shading.
- Gouraud shading.
- Phong shading.
- Diffuse lighting.
- Selective specular lighting.
- Multiple light-sources.
- A positionable point light - achieved in real-time (note that this adds a little ambient at a certain height and turns off if below the plane).
- Multiple spot lights.
- Rotating geometry by one method.
- Rotating spot light by another.
- Adaptation for NVidia processors so that it does not run so slowly.

2 Difficulties, Limitations and Workarounds

The difficulty comes from several places in this project the first is long loading times during compilation, this is due to the CPU reading in the height map on a single core and storing its data in an array, then using each item in the array a position vector is generated, this is based off the location of the pixel in the image and its colour. Since the image is 1024x1024 and we want to generate a 3D position in a mesh for each pixel this results in over 1 million calculations. The outcome of this was imperfect as there was little variation in the heights it chose (note that there are 256 shades of white / grey / black but there are many more available height positions in a scene) so it had to be interpolated to make it smoother.

*e-mail:40163650@live.napier.ac.uk

This brought the number of calculations in just creating the mesh to 18 million. To combat this a step value was created - a number to decide how many pixels in a direction to skip and use the previous values for. This makes for a smoother but less accurate image, it did make the loading time 16 times faster when set to a value of 4.

One small issue is the matter of speed (actual movement speed not frame-rate) when running the program on different machines, for example when testing using the NVidia architecture the movement speed of the camera is greatly reduced. The quick fix for this is to find the OpenGL provider for the computer in question and change the move amount based on that.

Figure 2:

```
bool initialise()
{
//...
        nv = "NVIDIA Corporation";
        ve = (char*)glGetString(GL_VENDOR);
//...
}
//...
bool update(float delta_time)
{
//...
        if (strcmp(nv,ve)==0)
        {
                //movAm = move amount
                movAm = 0.3f;
        }
        else
        {
                movAm = 0.03f;
        }
//...
}
```

The third difficulty faced in the project relating to performance is to do with the sheer quantity of light sources. The design used featured a 'map' that determined each light's location. There are 394 of these particular lights in total. Originally the effect these lights created used the Phong shading technique, as previously discussed there are more pixels in the geometry than vertices, and there are over 1 million vertices. To improve performance the lighting calculation for these 394 spot-lights was switched to Gouraud shading. This did increase the fps (frames-per-second) but unfortunately gave it a more jagged look in some places. In Figure 3 the blue light uses Gouraud shading whilst the white light is Phong shaded.

Figure 3:

One final limitation is that is really difficult to render a lot of 'unique' meshes into the geometry with the helper classes given, they were extracted and modified so that they themselves could create more than one 'mesh' in a mesh when given a quantity and a vector array of 3-dimensional vectors. One further improvement that could be made to this is to use a geometry shader to render them. This is planned for a later date, note that it would be beneficial to also use the geometry shader for the main mesh that is the terrain.

3 Effects & Techniques

What will be covered in this section: This section aims to cover the principals used in rendering, shading and in traversing the geometry.

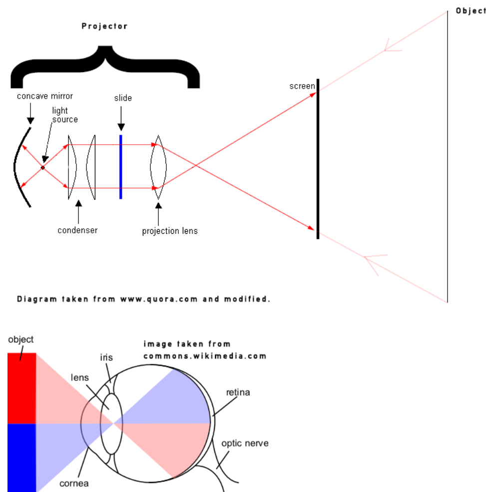
First Principles: What is light? Light is energy that can be sensed by organic receptors, it is perceived differently based on its frequency and intensity. It travels in streams of particles that are reflected, refracted, diffracted and emitted by physical objects. Light is also effected by quantum mechanics. In order to see things with a computer through a monitor as we would in the real world, i.e. in 4 dimensions, we must model it. It is impractical to attempt to model it exactly as it requires a lot of computing power and a similar effect can be achieved much more easily. So we model light as rays (after all, it has wave-particle duality) which are much easier to represent on a computer, using vectors.

Vectors: What is a vector? Put simply a vector is a collection of related numbers arranged in a rectangle. The values within the vector can be used to represent many things, a point, a line, a curve, a colour or even an equation. Vectors can be easily rendered, it was even achieved in 3D as early as the 1960s when general computing power was much lower [Lyall-Watson 1967]

Polygons: A polygon is a collection of 3 (possibly 2 with the third edge derived) or more vectors that make up a shape. The easiest shape to make is a triangle. Any one triangle can be seen as a 2D object provided that the plane is rotated to face the viewer. When a second triangle is introduced on a different plane the scene must be viewed in 3D. Many polygons can be used to create 3D meshes, this is how most objects are represented in 3 dimensions on a computer, especially in games technology and industrial design.

Cameras: At this point we have a representation of an object stored in a computer, but how do we display that on a screen? In nature humans use 2 eyes which act as cameras and a brain which acts as a processor to interpret our 3D surroundings. Rays of light from a source are reflected off objects and into our eyes, from this we can tell the colour of the object and its position. Computers work in a similar way to represent this phenomenon although they are more like projectors, in this example imagine a permeable screen, an object behind the screen and a source of light in front of it (our projector / computer). Rays are projected from the projector in every direction, if they hit the object they are traced back to the screen and a pixel at that point is coloured appropriately, this is known as ray-tracing. With enough rays we can build up a view-projection matrix (a type of vector which is a collection of vectors) and render it to the screen.

Figures 4 & 5:



Lighting, Colouration & Texturing: The most simple form of lighting and colouring is ambient. With this style of lighting we add the colour of the object (as given by the texture or material) with the colour of the light. Colours are represented as vectors of 4 components, during the addition the first values are added together, so are the second, third and fourth - each with the corresponding element. The values stored in a colour are the red, green, blue and opacity components. To obtain the colour of a pixel on a textured object, the texture has to be mapped to the surface. Why would we use ambient lighting? Scenes that are set in places such as underwater will have a blue / green component to them throughout. A room lit by flame will have an orange glow to it.

Global illumination is a technique used to represent sunlight. This is feasible because rays of light from The Sun can be considered parallel because their source is so far away that the angle between them is negligible. The equation for this requires the normals of the surface (a unit vector perpendicular to the plane the surface is on, it is calculated using the cross product of 2 of the polygon's edge vectors) and the direction of the light (this is also normalized, divided by its length, length is found using Pythagoras). We find the dot product of these 2 vectors and multiply the result by the light colour and the surface colour. There are 2 methods to finding the surface colour, diffuse and specular.

$$final_colour = light_colour * surface_colour * (normal \cdot light_direction)$$

When transforming a mesh, one must also remember to transform its normals so that it can be shaded properly.

Diffuse lighting uses this basic model. Specular lighting also takes into account the position of the viewer's eye. The specular effect gives a glossy appearance to objects whereas diffuse gives them a matte look. We use the eye direction, E and the light direction, L. Then we calculate the half vector between them, H. The equation is similar to the previous one (note that N represents the normal and m represents the shininess of the object).

$$E = \frac{eye_position - object_position}{||eye_position - object_position||}$$

$$H = \frac{L + E}{||L + E||}$$

$$final_colour = light_colour * surface_colour * (N \cdot H)^m$$

A point light has a position and emits light in all directions from it. A spot light also has a position, this however only emits light in one direction in the shape of a cone. Both of these have a limit to how far they are emitted, they do not just cut out after a certain distance, they have an attenuation factor to determine how much of the scene they light. This is given by the following equations.

$$constant = a_given_value$$

$$linear = \frac{2}{range}$$

$$quadratic = \frac{1}{range^2}$$

$$attenuation = \frac{1}{constant + linear * distance + quadratic * distance^2}$$

This is just multiplied to the colour. [Chalmers 2014-15]

There are 2 main approaches that lighting can be applied to the scene with use of the shaders, they are Gouraud shading and Phong shading. Gouraud takes advantage of the vertex shader and uses it to calculate the light at each vertex in a scene, the colour of polygon is then calculated from its vertices' colours. It is worth noting that the lighting calculations are fairly heavy so Gouraud shading is seen as a short-cut. Phong shading calculates the lighting for each individual pixel in the scene, this means that it is more accurate but it is quite difficult for older / lower quality GPUs to use frequently, which means that it has not often been used until fairly recently. In terms of programming it uses the same calculations but they are done in the fragment shader and not the vertex shader.

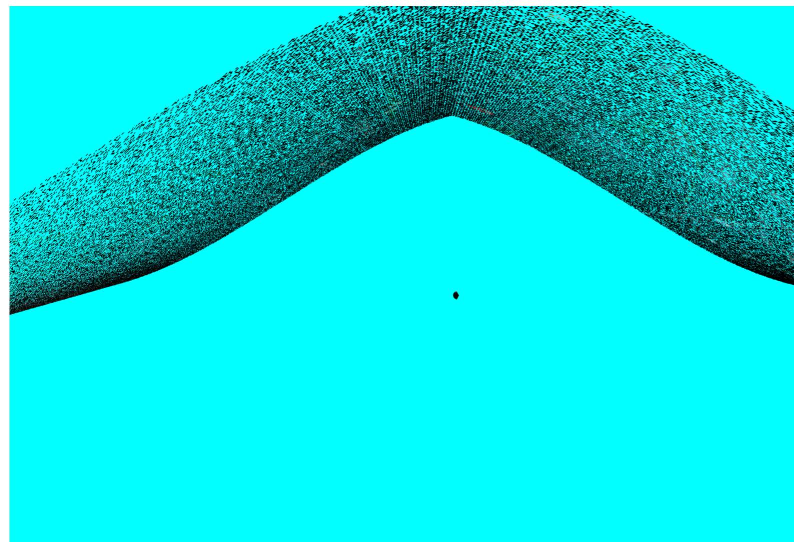
4 How these techniques have been used in the project

This project uses multiple meshes, the main one is a plane whose heights have been transformed using data from a height map.

Download Link to the Data used.

To achieve this the helper method for creating planes was modified so that it could take an image as a parameter. Then an interpolation method was created so that it didn't appear so staggered.

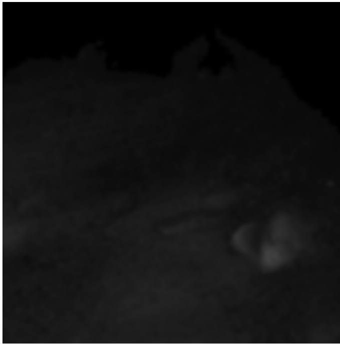
Figure 6:



This is the side view of the mesh generated by the height map in Figure 7 before being interpolated

Figure 7:

Figure 8:



The height map used in the project. - Dark, low, light, high.

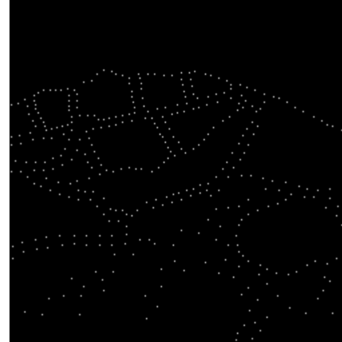
Originally the project used global illumination to light the scene but this was changed to a large point light so that it could be transformed to simulate the movement of The Sun. This particular point light had an effect applied to it in the fragment shader to create an orange hue (ambient) when it was at a certain height to mimic The Sun setting.

The terrain has selective specular shading so that just the water is shaded, again this was a test of height. In Figure 1 one can see the results of testing the normal angles and applying shading to them, there were many tests to see what could be done with shading based on 3D position, colour and normal angle however these did not make it into the final scene as there was no practical reason to use them. Here is the code used to determine which sections have specular lighting applied to them with the spot lights.

```
vec4 primary = mat.emissive + diffuse;
vec4 secondary = specular;

secondary.w = 1.0;
primary.w = 1.0;

vec4 spot_colour;

if (position.y < 0.1)
{
spot_colour = primary * tex_colour + secondary;
}
else
{
spot_colour = primary * tex_colour;
}
```

The street lights were added to the terrain using another map (this was at 1/4 resolution to reduce loading times so not as many pixels had to be read, their x and z positions were then multiplied by 4 to compensate.) They had to be placed at the same time as the height map was generated so that they could get appropriate y co-ordinates from the mesh. Each light is represented by a white pixel.

Figure 9:



In the project all spot lights use Gouraud shading, that is because there are 396 of them (when including the light-house and the light above it.) If this were Phong shaded there would be many more calculations to do than the 400 million it does in the vertex shader (over 1 million vertices, about 400 lights). The point light however does use Phong. The result of this is a drop in frame-rate but with all calculations in the fragment shader it was impossible to render the scene in real-time and traverse it.

With regards to texturing, the scene uses multiple textures, one for the terrain, 2 for the light-house, one of which is used again for the street-lights. The head of each street-light uses the same texture but this cannot be seen as its emissive property of its material is set to full white. The terrain texture is transformed to synchronise with its mesh, then its edges are mirrored so that it looks more realistic where it is repeating.

Some objects in the scene are transformed, for example the light-house 'bulb' is rotated along with the light it emits, the bulb is rotated using the helper method. The spot light is rotated using a quaternion. The spot light used as the sun can be controlled using the scroll-wheel on the user's mouse. It is rotated about a point.

4.1 List of Libraries and Headers included in the Project:

- graphics_framework.h
- glm\glm.hpp
- iostream
- atlimage.h
- string.h
- ctype.h
- The libENUgraphics Library
- assimp.lib
- FreeImage.lib
- glew3.lib, glew3dll.lib, glew32.lib & glew32s.lib
- libnoise.lib
- Assimp32.dll

- D3DCompiler_42.dll
- D3DX9_42.dll
- FreeImage.dll
- glew3.dll & glew32.dll
- libnoise.dll

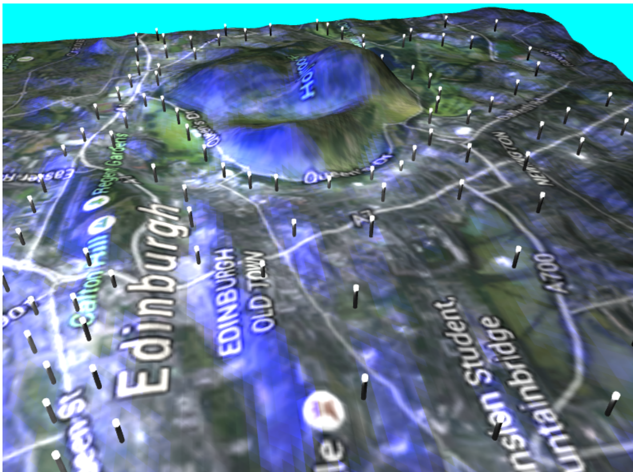
Note that not all of these were actually utilised.

5 Conclusion

In this project many aspects of rendering were covered. The main feature of this project is the terrain generated with the height map and the lighting effects applied to it. In the future this project is to be improved, it is desirable to optimise the terrain and mesh generation by using the geometry shader and it is also desirable to apply new post-processing techniques to the scene in real-time. Some aspects that are missing from this project that were intended were blended textures, bump mapping (of textures) and transparency. This could be included in a future update.

In the future projects will not contain as many objects (or at least fewer meshes with as many objects making them up) and light sources to reduce calculations, for the effect that is achieved the number of FPS lost is not worth it. The map reading part of the program works well and shall be used again, it has many uses, not just for creating terrain but for adding objects. Using an actual bump map like ones used in targa files should also increase performance, though this is not a practical solution if one would want to add collision detection to the mesh. Again, the geometry shader is the way forward here, especially for creating many copies of the same object.

Figure 10:



References

CHALMERS, D. K., 2014-15. Set08116 computer graphics — workbook. [Online; link]. 3

LYALL-WATSON, M., 1967. Tomorrow's world — elliot light-pen. [Online; accessed 13-March-2016; link]. 2