

Solar System Model

Dimitrios Tsolis

40204497@live.napier.ac.uk

Edinburgh Napier University - Module Title (SET08116)

Abstract

The aim of this project was to render a realistic scene using the skills developed in the Computer Graphics Module at Edinburgh Napier University using OpenGL and C++. The scene produced in this project is a solar system, inspired by my adoration for the stars and the Star Wars movies [1]. Advanced 3D effects and techniques are used to accomplish the generated result, such as lighting, shadowing and texturing. These techniques are widely used in a variety of games.



Figure 1: **Millenium Falcon** - Chase Camera Inspiration

Keywords – Shadowing, Lighting, Multi-Texturing, Skybox, Multiple Cameras, Material Shading

1 Introduction

Referencing The key effects being used are lighting, texturing, shadowing, skybox, material shading and transformation hierarchy. Various rendering techniques are required to produce a realistic depiction of the scene. The different lights being used in the scene are a spot light and a point light in the middle of the sun. Transformation techniques are also being applied to the different meshes of the scene, and each of the objects render have their own textures. Furthermore, in order to create the stars image around the solar system a skybox effect is being used. In order to make the earth more realistic, normal mapping is being implemented on the project.

Diffuse light is being applied to the scene, adding some depth perception to the objects, as well as specular light

which adds shininess of materials to objects and changes depending on the camera position. The transformation and translating of the meshes in the scene is the scaling and the positioning of the objects. Texturing combined with the lighting is applied to give the objects a more realistic aesthetic. A more advanced technique being used is normal mapping, which uses the texture and a normal mapped version of the texture, which helps determines the direction the pixels are facing, giving an extra feeling of depth to the object.

2 Related Work

The implementations required for this project have been based on the Computer Graphics workbook, while some of the skills attained had to be further developed. For example, I learned how to apply textures from the workbook, but I also learned how to apply multiple textures and normal mapping by creating texture arrays and binding them to the corresponding meshes in the render boolean by looking up information on-line and experimenting with OpenGL.

3 Implementation

A variety of visual elements were involved in the creation of the scene. These elements are:

- Texturing
- Normal Mapping
- Lighting
- Shadow Mapping
- Transform Hierarchy
- Skybox

Apart from these visual elements, multiple cameras and the rotation of the planets around the sun were also implemented.

3.1 Texturing

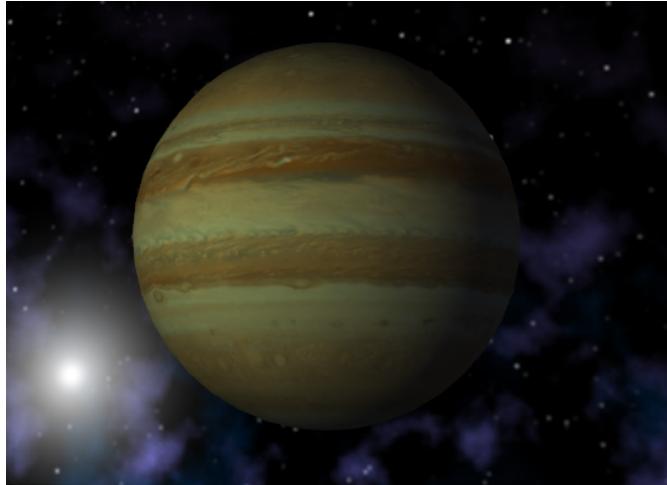


Figure 2: **Jupiter Texture** - Texture Example

In this project I have created a mesh array and a texture array. In order to apply a texture to a specific mesh, the same name is given to the mesh and the texture in their corresponding arrays. So during the render method, I have created a for loop to render each mesh, and each time a mesh is rendered by going through the for loop, the texture is bound to the mesh by checking for a texture with the exact same name. During the for loop, the key element of the mesh array is the same in the texture array, and therefore, the texture is bound. An example of a texture can be seen in figure 2.

3.2 Normal Mapping

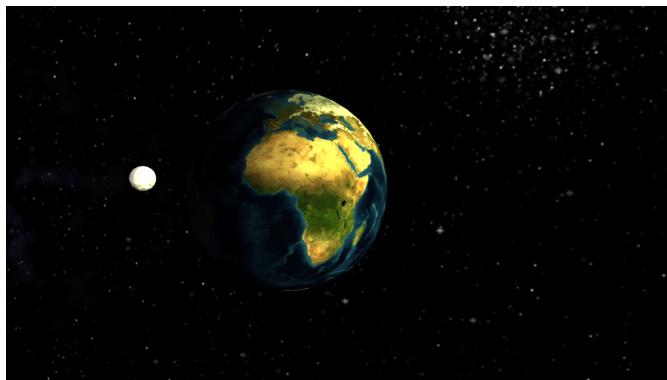


Figure 3: **Normal Mapped Earth**

Normal mapping is a method used for making fake bumps and dents in the object, basically to make it more realistic. This is achieved by calculating the normals on a per pixel basis instead of per vertex basis. A normal map is basically a texture where the x, y and z axis of the normals are represented by red, green and blue values respectively. In order for normal mapping to be used, we need to work within the tangent coordinate space, which is based on the normal at a particular point of the object. So when normal mapping is used, the

normal, the binormal and the tangent become the axes in our coordinate space. To do this, we simply create a transformation matrix:

$$TBN = \begin{bmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{bmatrix}$$

where T is the tangent, B is the binormal and N is the normal. These are transformed by the normal matrix in order to get the actual surface values. In our normal mapping shader, we use the TBN matrix to transform the sampled normal. In this project, normal mapping is only applied to the earth (See figure 3).

3.3 Lighting



Figure 4: **Moon** - Emissive light example

General Lighting Lighting involves three pieces of physical transformation, which are light emitters, materials and sensors.

Light sources emit light. A simplistic model of light is considered to have three components:

- **Ambient Light:** light that shines everywhere at an equal measure
- **Diffuse Light:** light that has a direction and reacts with an object's surface in a different way depending on the direction of the light
- **Specular Light:** specular highlights

In this scene, we are not using ambient light, but diffuse and specular, which are calculated in the fragment shaders.

Materials describe the surfaces of objects. In our scene, the material description in the shader has three values: emissive colour, diffuse reflection colour, specular reflection colour and shininess. The diffuse reflection colour illustrates how the diffuse and the ambient light interacts with the object, the specular reflection colour describes how the specular light interacts with the object, the shininess describes how shiny the objects looks and the emissive colour is the light that emits from the object. Most of these objects in the scene have those values being set throughout the main cpp file. An example of

a material with a relatively high emissive light compared to the other objects is the moon(See figure 4), since most of the planets in this project have their emissive colour set to 0.

Finally, in order for an object to be visible, the light that is interacting with the object has to partially be absorbed by a light sensor. The more light it absorbs, the less visible it is.

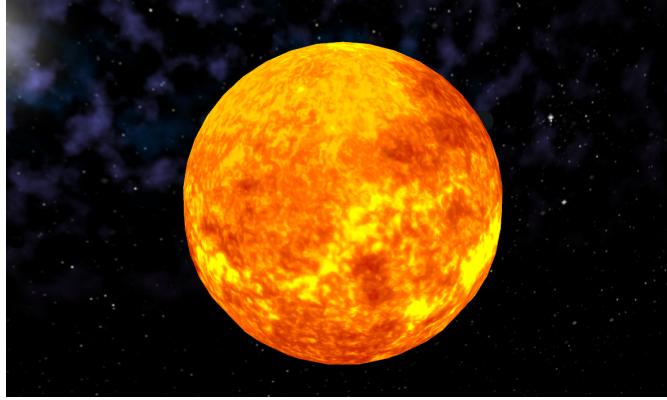


Figure 5: **Point Light** - Main Light Source for the project

Point Light Point lights are light sources which have a position and an area which they light. In the scene being created, we have one main light source, which is a point light, and it is located in the middle of the sun. In order to let the light go out from inside the sphere, the normals of the sun mesh had to be reversed. I did that by creating a different sun effect which has identical shaders with the normal effect, apart from the vertex shader. In that vertex shader, when the transformed normal is calculated, it is timed by -1, so the normals are reversed and the light can shine throughout the scene.

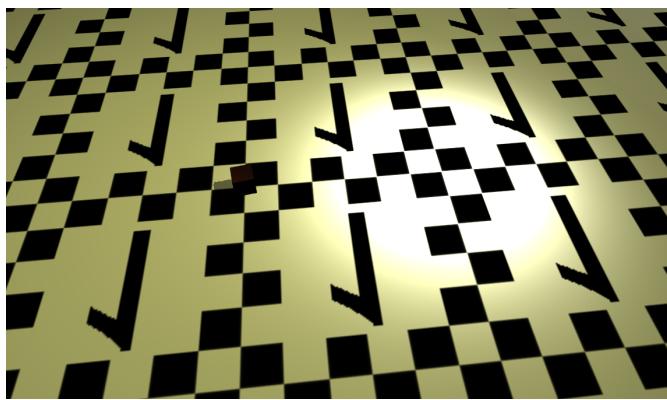


Figure 6: **Spot Light Example** - This is the spotlight that is bound to the shadow map

Spot Lights Aside from the point light, we have several other spotlights. Spotlights are very similar to point lights, but in this case the lights are facing towards a specific direction. In this coursework, 9 different spotlights are being created. Eight of those spotlights are rotating along along with the planets, with a position over the planets

and a direction towards them. The ninth spotlight is under the sun, facing towards the plane and is bound to the shadow map(See figure 6). Nine different spotlights are being created by using a vector container, which holds the spotlights and their key values. In order for the spotlights to shine, each spotlight should be initialized before rendering. The spotlights in the project can be turned on and off at any time.

3.4 Shadow Mapping

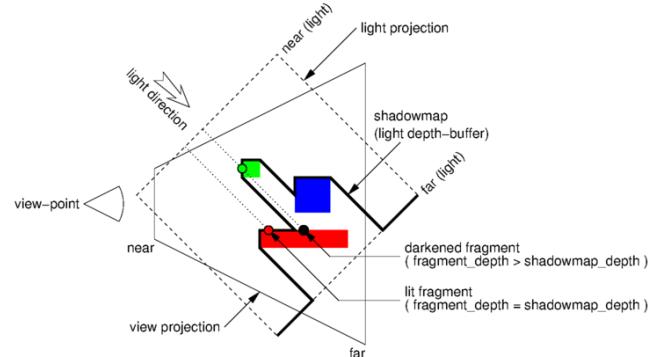


Figure 7: **Shadow Mapping**

In order to implement shadows in this project, we used the shadow mapping technique (See figure 7). Shadow Mapping utilises the depth buffer in order to determine whether an object is in shadow or not. The concept of shadow mapping is that the scene is rendered from the point of view of the light source, then the depth information is gathered, and thereupon that information is used to determine if the object is in shadow. In this scene, I am creating a spot light below the sun facing towards the plane, with a 90 degree point of view, and I am binding the shadow map to that specific light. Therefore, the shadow of the cube is created(See figure 8).

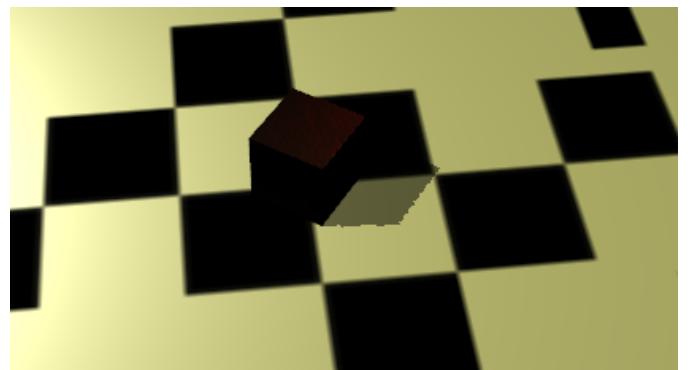


Figure 8: **Shadow of the rotating cube**

3.5 Transform Hierarchy

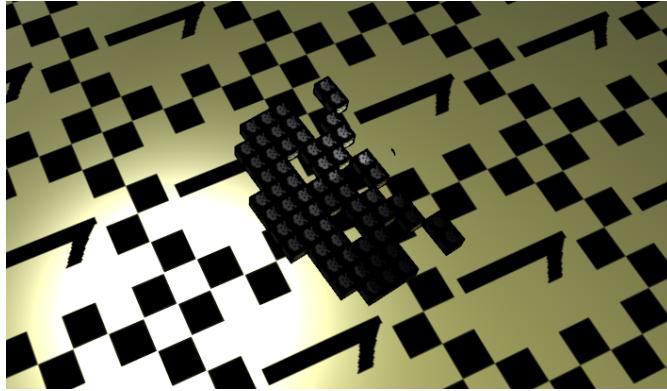


Figure 9: **Space Invader** - Transform Hierarchy Implementation

In the scene, transformation hierarchy is applied in one of the objects(See figure 9). During the transformation hierarchy render, the projection matrix(P) and the view matrix(V) are calculated only whenever the camera moves. The model Matrix only changes when a model is moved, rotated, or transformed. When we calculate the model matrix in this case, we set it to be the space invader transform matrix, and then we set it to be the sun transformation matrix(which is rotating around its own y axis) times itself. This makes the space invader object inherit the sun's rotation. The object is located in the middle of the sun.

3.6 Skybox

A skybox is implemented in the scene. That gives off the illusion of the stars located around the solar system. In order to do that, a simple cube is rendered, with the inside faces rendered and a star cube map texture applied to it. Furthermore, the depth buffer is temporarily disabled during the skybox render, ensuring that the skybox appears off in the distance. This is achieved by making the position of the skybox equal to the position of the active camera, which means that the camera can never leave the skybox.

3.7 Multiple Cameras



Figure 10: **Millenium Falcon Chase Camera**

There are two cameras in this project: a chase camera and a free camera. The free camera allows free movement around the scene, while the chase camera chases around a specific object as it moves, which in our case is the millenium falcon(See figure 10). This is done by having a global boolean which is set to either true or false every time we want to change the active camera. When we are rendering the objects, we calculate the view matrix using the camera that is currently active.

3.8 Rotating Planets

The planets rotate around the sun by setting the position of each planet during the update method using cosine for the x axis and sine for the z axis and adding the position of the sun.

```
1 meshes["mercury"].get_transform().position = (vec3(cos(<-
velocity*3.0f)*35.0f, 0.0f, sin(velocity*3.0f)*35.0f) + <-
meshes["sun"].get_transform().position);
```

2

Velocity is a float. Each time the project goes through the update method, delta time is subtracted from velocity times the equation for an elliptical orbit. The equation applied in this case is not exactly correct, but it results in an approximation of an elliptical orbit for the planets. The x and z of the position is then multiplied by a number, which is the radius we wish to give to each planet.

4 Further Work

One of the features I would like to apply in the future is the shadows of all the planets in the plane using a spotlight above the sun. Due to errors, this was not implemented in the scene. Moreover, I would like to apply the correct equation to the velocity of the planets, but I could not completely understand the equation because it required further research, and due to time constraints it was not applied.

5 Conclusion

In conclusion, rendering a 3-Dimensional scene using OpenGL and C++ requires a lot of reading around, experimenting, and practising with different visual elements to see what's best. The combination of the variety of techniques being used in this project create an interesting, visually pleasing and gripping result.

References

- [1] George Lucas, "Star wars: A New Hope," May 1997.