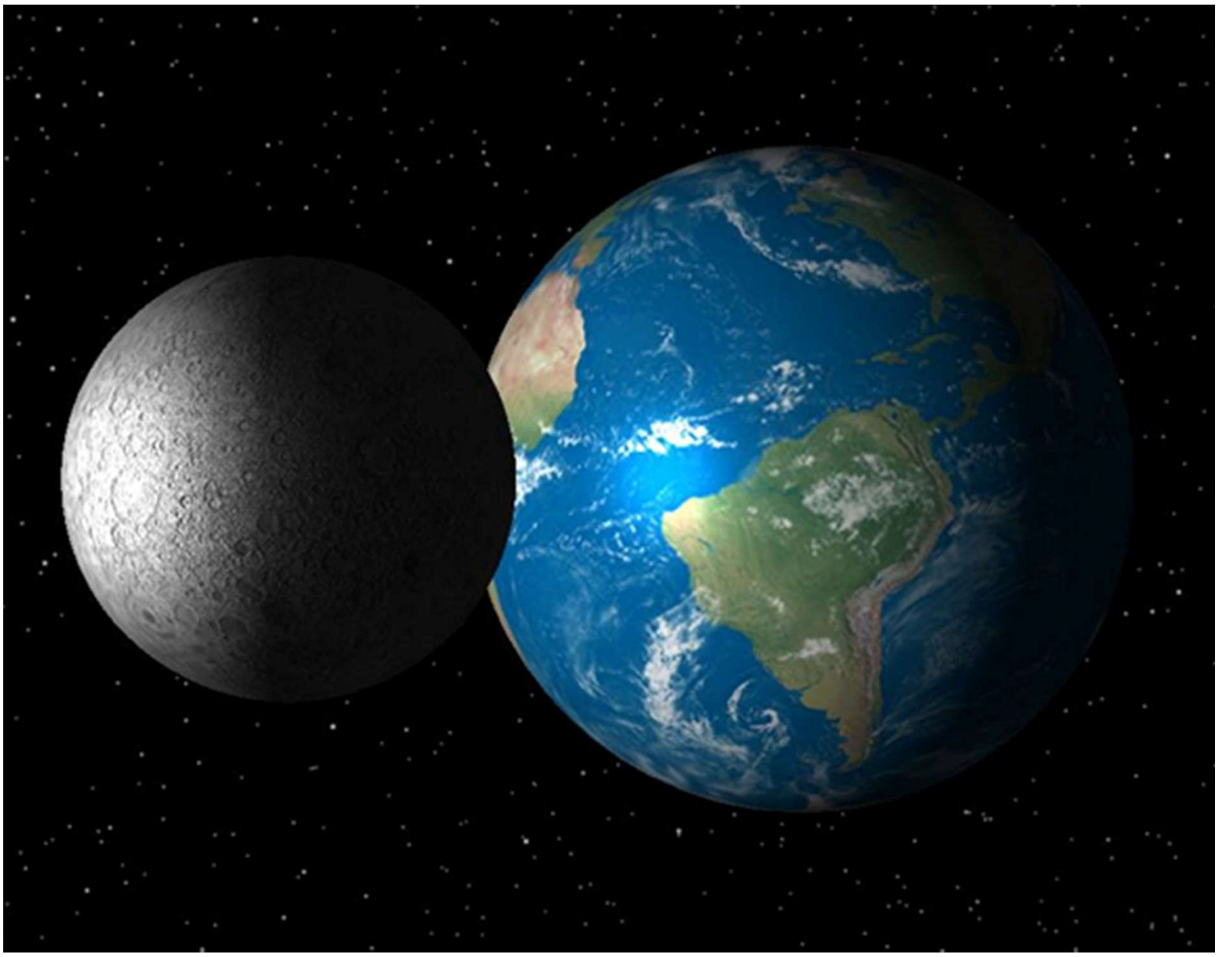
GL-Universe

CW4 – Mathematics and Graphics for Computer Games II

Elio de Berardinis, Federico Soncini

Submission Date: 31-05-2016

No. of Pages: 13



OBJECTIVE

The goal of the project was to create an environment where we could implement a basic lighting model (Single light source) and show specific renderings of textured objects in the scene. A rendering of our solar system is a good choice for this model as the Sun can be approximated as the only light source present (the light from the other distant stars is negligible).

INTRODUCTION

This project combines different aspects of the graphics pipeline including geometry generation, optimization, texturing and lighting. For brevity and because some of these topics were already covered in previous coursework we will focus the background information on light rendering which is the bulk of this assignment.

Light Rendering

The basic standard lighting model is represented by the following equation [1]:

$$Color_{final} = emissive + ambient + \sum_{i=0}^{Lights} (ambient + diffuse + specular)$$

The **emissive term** represents light emitted or given off by a surface and its contribution is independent of all light sources and only relates to the material.

It is an RGB value that indicates the colour of the emitted light. This term can add colour to a vertex even in the absence of any lights illuminating the surface.

The component can be written as

***emissive* = *Material*$_e$**

The member on the right hand side refers to the material’s emissive colour component.

The **ambient term** refers to light that has bounced around so much that it seems to be coming from all directions. In fact ambient light does not appear to come from any specific direction at all. This is the reason why ambient light does not depend on the light source position. The ambient term depends on a material’s ambient reflectance, as well as the colour of the ambient light that is incident in the material. Like the emissive term, the ambient light on its own is just a constant colour, but unlike the emissive, it is affected by the global ambient light.

The ambient light is calculated by multiplying the global ambient colour by the material’s ambient property:

$$ambient = (Global_a * Material_a)$$

The equations shows that the ambient term results in the multiplication of the global ambient term by the material's ambient colour component.

The **light contribution** indicates how every active light in the scene will contribute to the final vertex colour. The total light contribution will be the sum of all the active light contributions, as denoted in the following equation.

$$contribution = \sum_{i=0}^{Lights} Light_{attenuation} * (ambient + diffuse + specular)$$

The **attenuation factor** represents the gradual decrease of the light's intensity as the light's position moves father away from the vertex position.

This factor is only taken into account when positional lights are involved: if the light is defined as a directional light, there will be no attenuation.

The final light's attenuation depends on three constants, which define the constant, linear and quadratic attenuation factors.

The formula for the attenuation factor is

$$attenuation = \frac{1}{k_c + k_l d + k_q d^2}$$

In addition to the three above mentioned constants, we have **d**, which is the scalar distance between the point being shaded and the light source.

The **light's ambient contribution** is calculated by multiplying the light's ambient colour with the material's ambient colour.

$$ambient = (Light_a \text{ x } Material_a)$$

The **diffuse term**, also known as **Lambert reflectance**, accounts for directed light reflected off a surface equally in all direction. The amount of light reflected is proportional to the angle of incidence of the light striking the surface. The element to be considered in the Lambert reflectance is the angle of the light relative to the surface normal:

$$\boldsymbol{\mathit{diffuse}} = \boldsymbol{K_d} * \boldsymbol{\mathit{lightColor}} * \max(\boldsymbol{N} * \boldsymbol{L}, 0)$$

where $\mathbf{K_d}$ is the material's diffuse colour, **lightColor** is the colour of the incoming diffuse light, $\boldsymbol{N}$ is the normalized surface normal, $\boldsymbol{L}$ is the normalized vector toward the light source.

The vector dot product of the normalized vectors N and L (from the point **p** that is being shaded) is a measure of the angle between the two vectors. The dot product value will be greater the smaller the angle between the vectors will be, and the surface will receive more incident light. Surfaces that face away from the light will produce negative dot-product values, while the $\boldsymbol{\mathit{max}}(\boldsymbol{N} * \boldsymbol{L}, 0)$ in the equation ensures that the surfaces that point away from the light will not have any diffuse light contribution.

Finally, the last contribution to lighting is the **specular term**, which represents light scattered from a surface predominantly around a mirror direction. The specular term is prominent on very smooth and shiny surfaces, such as polished metals. Unlike the emissive, ambient and diffuse terms, the specular depends on the location of the viewer.

The specular term is dependent on the angle between the normalized vector from the point we want to shade to the eye position (**V**) and the normalized direction vector pointing from the point we want to shade to the light (**L**). This is known as the **Phong lighting model.**

The mathematical equation is:

$$\mathbf{L} = \mathit{normalize}(\mathbf{L}_p - \mathbf{p})$$

$$\mathbf{V} = \mathit{normalize}(\mathbf{eye}_p - \mathbf{p})$$

$$\mathbf{R} = 2(\mathbf{L} \cdot \mathbf{N})\mathbf{N} - \mathbf{L}$$

$$\mathbf{specular} = \mathbf{k}_s * \mathbf{L}_s * (\mathbf{R} \cdot \mathbf{V})^{\alpha}$$

where **p** is the position of the point we want to shade, **N** is the surface normal at the location we want to shade, $\mathbf{L_p}$ is the position of the light surface, $\mathbf{eye_p}$ is the position of the eye, **V** is the normalized direction vector from the point we want to shade to the eye, **L** is the normalized direction vector pointing from the point we want to shade to the light source, **R** is the reflection vector from the light source about the surface normal, $\mathbf{L_s}$ is the specular reflection constant for the material that is used to shade the object, α is he “shininess” constant for the material. The higher the shininess of the material, the smaller the highlight on the material.

The specular term can be calculated also in another way that differs slightly from the Phong model but builds upon it. The **Blinn-Phong model (Figure1)** calculates the half-way vector between the view vector and the light vector, instead of calculating the angle between the view vector and the reflection vector.

The variable **H**, equalling the half-angle vector between the view vector and the light vector, can now be added to the previous formula.

$$\mathbf{L} = normalize(\mathbf{L}_p - \mathbf{p})$$

$$\mathbf{V} = normalize(\mathbf{eye}_p - \mathbf{p})$$

$$\mathbf{H} = normalize(\mathbf{L} + \mathbf{V})$$

$$\mathbf{specular} = \mathbf{k}_s * \mathbf{L}_s * (\mathbf{N} \cdot \mathbf{H})^\alpha$$

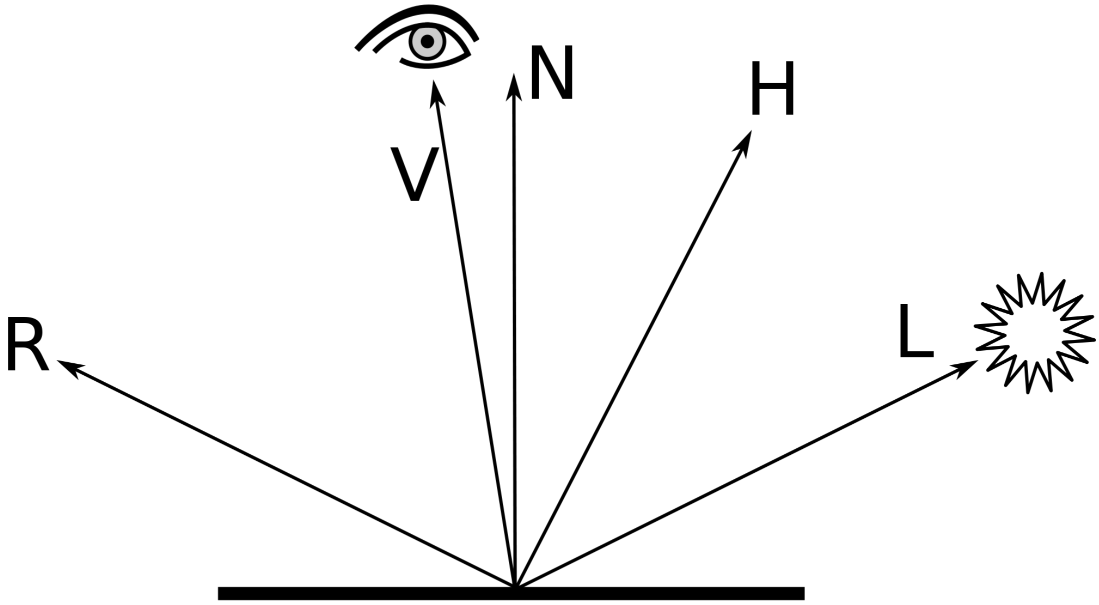


Figure 1 – *The Blinn-Fong Vectors*

IMPLEMENTATION

The Project contains two interchangeable scenes (by pressing the SpaceBar):

1) Small Geocentric scene, where the Earth rotates along its main axis at the centre of the universe with the moon translating around it. Both the Earth and the Moon are illuminated by the light coming from the sun which is also rotating around the Earth. This rendering is not scientifically correct (Galileo taught us!) but allows us to concentrate the rendering on our planet with greater details. **Figure 2**

2) The Solar System with most of its planets (Venus, Earth, Mars, Jupiter, Neptune). Here the Sun is placed in the middle of the scene and textured while the planets are rotating around it with different speeds, directions and rotations along their main axes. **Figure 3**

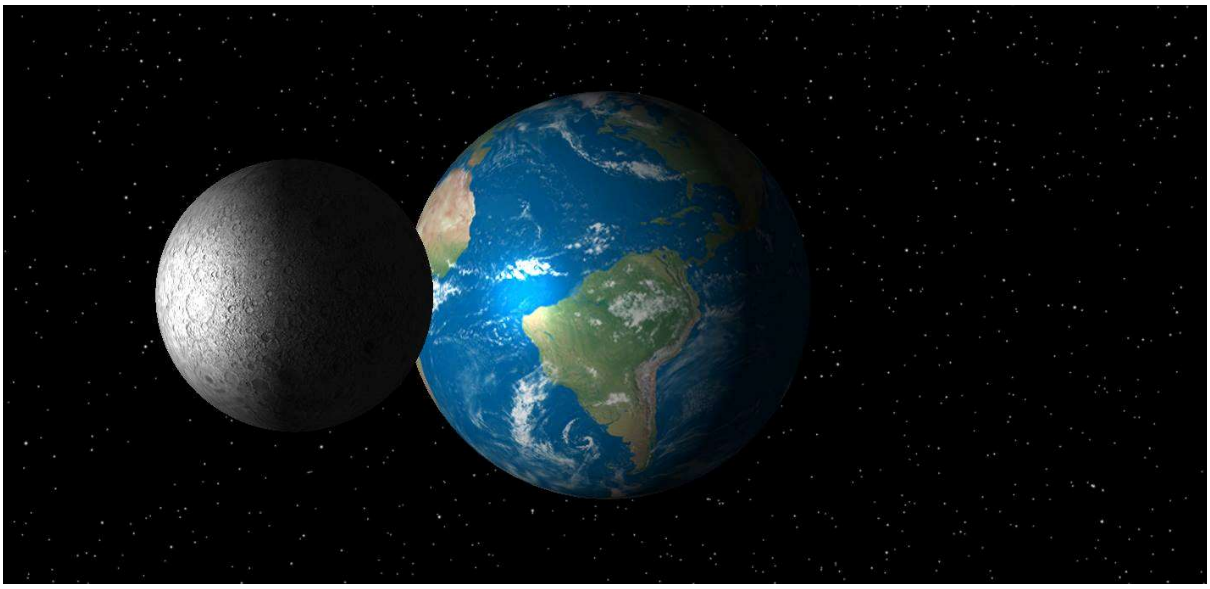


Figure 2 – *The Geo-Centric Scene*

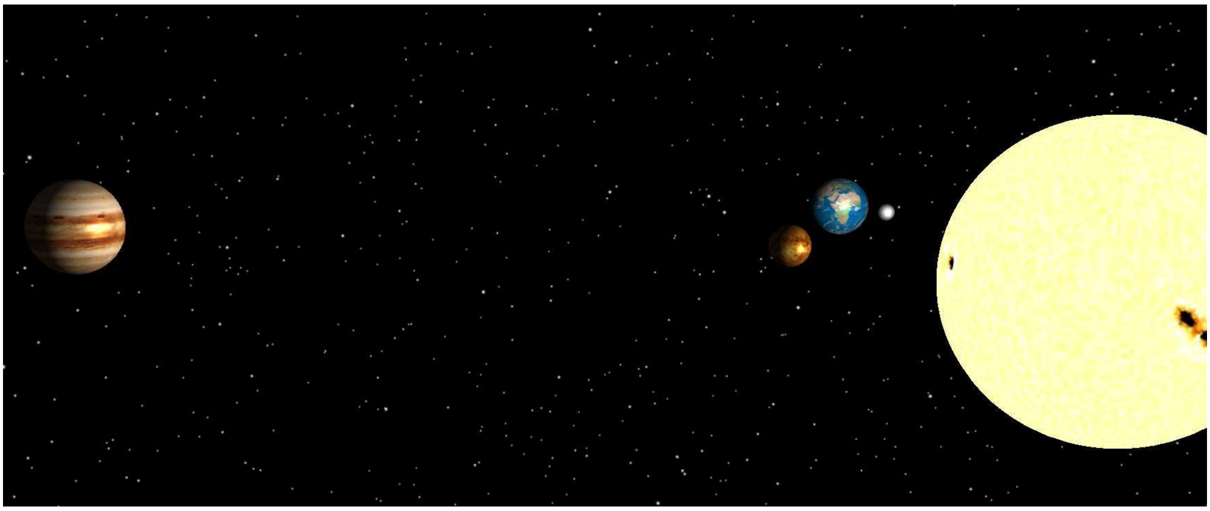


Figure 3 – *The Helio-Centric Scene*

We created our project in VisualStudio 2015 using C++ as main coding language, OpenGL 3.3 as our Graphics API and GLSL for our shaders.

As it's usually done when dealing with OpenGL to facilitate its use we downloaded, compiled and linked to our main project the following libraries:

- Free OpenGL Utility Toolkit (FreeGLUT 2.8.1): FreeGLUT is an open-source alternative to the OpenGL Utility Toolkit (GLUT) library. The purpose of the GLUT library is to provide window, keyboard, and mouse functionality in a platform-independent way [2].

- OpenGL Extension Wrangler (GLEW 1.13.0): The OpenGL Extension Wrangler library provides accessibility to the OpenGL core and extensions that may not be available in the OpenGL header files and libraries provided by your compiler [3].

- OpenGL Math Library (GLM 0.9.5): The OpenGL math library is written to be consistent with the math primitives provided by the GLSL shading language. This makes writing math functions in C++ very similar to the math functions found in GLSL shader code [4].

- SOIL: The Simple OpenGL Image Library is the easiest way to load textures into OpenGL. With a single method, textures can be loaded and an OpenGL texture object can be obtained [5].

After these libraries were added to our project in VisualStudio, Compiled, linked and tested for errors we started by creating the shaders needed to render the objects in our scene before moving on to the main program. For brevity, as the code is heavily commented in the solution we will only refer to the main functions, parameters or files and add relevant explanations here when needed. The set-up of a simple OpenGL context with the dependencies described above and creation of a simple geometry were obtained following some established practices [1][6][7].

Shaders

First we created a pair of very simple shaders (Vertex + Fragment) to render our light source (the Sun). The Vertex Shader of this pair (*simpleShader.vert*) accepts a position as input and a single uniform variable in the form of the Model, View, Projection Matrix (*MVP*). The output of this shader is simply the clip-space transformed input position to be passed to the fragment shader.

Similarly to the previous one the fragment Shader (*simpleShader.frag*) only has one uniform in the form of a color variable (*color*) passed by the main program. This shader simply outputs this color to the location obtained from the vertex program.

Next, we created a pair of shaders needed to texture and light up the objects in the scenes according to the basic standard lighting model described before. The vertex Shader of this pair (*TexturedLit.vert*) takes in the vertex positions, the surface normals and the texture coordinates and outputs their transformation in clip-space (through the ModelViewProjection matrix) and in world space (Through the Model matrix).

The fragment shader (*TexturedLit.frag*) takes in the transformed geometric variables from the fragment shader as well as a series of uniforms passed by the main program. These uniforms are related to the position of the camera (*EyePosW*), the position of the light source in world space (*LightPosW*), the light's diffuse and specular contribution (*LightColor*), the materials properties (Emissive, Diffusive, Specular, Shinines) the ambient and the 2D sampler for the textures. In the "main()" body of the fragment shader the emissive, diffusive (Lambert) and specular contributions (Phong or Blinn-Phong models) are calculated and assigned to the output variable (*out_color*).

The main program

The main program creates the OpenGL context, sets up all the variables to create the geometries and links the shaders to render the scenes correctly.

First of all we grouped all the dependencies (FreeGLUT, GLEW, GLM, SOIL) in one single header file named "*AllHeadersPCH*". This is included to our main program along with the "*Camera.h*" header to handle the camera movement in OpenGL.

We then defined all the global variables referring to Shaders attributes, Macros for the buffers, Window handles, mouse and keyboard hotkeys, Clock events, reference to the 2 shader programs, all the shaders uniforms and the texture elements.

Next, all the callback functions are declared (*DisplayGL(), IdleGL, KeyboardGL(), MouseGL()* etc.). These functions will handle all the window events, will update the scene and translate the keyboard and mouse inputs. Their bodies are defined at the end of the main program and are mostly housekeeping and/or optimization/input. *DisplayGL()* and *IdleGL,* update and draw the scene at every frame. We will refer to them at the end of the section when discussing the rendering.

Subsequently we initialize the OpenGL context through the function *InitGL(int argc, char* argv[])* and the GLEW extensions with the function *InitGLEW().*

We then define the functions to load and compile the shaders (*LoadShader(GLenum shaderType, const std::string& shaderFile))* and to create the shader program (Vertex + Fragment) from the compiled objects (*CreateShaderProgram(std::vector<GLuint> shaders))* as well as the function to load the textures through SOIL (*LoadTexture(const std::string& file*)).

As our scene will be populated by celestial objects we predictably chose to approximate them with spheres. To create this simple geometry we use the function

"*SolidSphere(float radius, int slices, int stacks)"* where we divide the sphere in slices along its main axis and stacks around its circumference. The number of stacks and slices will determine the precision of the geometry. For the level of details needed in our scene 32 slices and 32 stacks are sufficient.

After the stacks and slices are defined, the index buffer is created and linked to the vertex attribute to create the Vertex Buffer Objects (VBOs) that will define the Vertex Buffer Array (VAO) for the Sphere to be processed by the GPU.

Finally we create the entry point of our program with the main (*main(int argc, char* argv[])).* Here we call all the previously mentioned initialization functions, shaders loader and compilers, load all the textures and obtain the references to the uniforms in the shaders for the light rendering. At the end of the main we call the freeGLUT functions that will handle the callbacks: *glutMainLoop().*

The most important callbacks, as mentioned before, are DisplayGL and IdleGL.

IdleGL will run whenever no other events need to be processed on the window's event queue. Since we don't really know how much time will pass between calls to the Idle function (this may be different on different computers) we should try to determine the elapsed time each frame and use that to move things smoothly in the scene. For this implementation, we are updating the position of the camera when the user presses the W, A, S, D, Q, and E keys on the keyboard. If so, we will pan the camera at 1 unit / second and if the Shift key is held down, we will move the camera at 5 units / second. We are also updating the position of the different celestial objects (Sun, Moon, Earth, Other Planets). The call to the function *glutPostRedisplay()* will make sure the scene is redrawn by calling DisplayGL. (Mouse Interaction with the scene are handled by the MouseGL callback).

DisplayGL is where we draw our scene and will be invoked whenever the render window needs to be redrawn (which is guaranteed to occur whenever the glutPostRedisplay() method is called). According to the ModeType variable the GPU will display the Geo-centric or Helio-Centric scene.

To render the Geo-Centric, we'll draw 3 spheres. The first sphere will represent the sun. This will be an unlit white sphere that will rotate about 90,000 Km around the centre of the scene. The position of the only light in the scene will be the same as the object that represents the sun. The Earth is placed at the centre of the scene and rotates around its poles, but its positions stays fixed at the centre (the Earth will not be translated).The final object will be the moon. The moon appears to rotate around the earth but at a distance of 60, 000 Km away from the earth.

For the Helio-Centric scene the sun is placed in the centre and the planets will rotate around it. The sun light is still represented by a white unlit sphere at the centre but this time the second shader (TexturedLitShader) is overlapped to apply the relevant texture and lighting effects to the sun object.

In both scenes the rendering is achieved through the following steps (except for the discussed differences):

1- Create a sphere VAO that used to render the sun, earth, moon and/or Planets.
2- Setup the uniform properties in the shaders (Simple or TextureLit): MVP matrices, positions, scaling, rotations, light properties, material properties.
3- Render the sun light with the Simple Shader.
4- Render Earth, Moon, Planets or Sun object with TextureLit shader.
5- Render the distant stars with a big sphere containing the whole scene disabling culling or enabling frontFace to draw on the inside of it. (TextureLit shader).
6- The VAO, shader program, and texture are unbound to leave the OpenGL state machine the way we found it.

TEAMWORK

We generally worked well together, had meetings almost every day and confronted each other on the research we did on the relevant topics on our own. Working together on previous projects (Car game) certainly helped enhancing our teamwork as we already knew each other personally and professionally. We were both eager to learn more about OpenGl and GLSL so we worked on similar things at the same time, in the pursuit of addressing some new and challenging topics. Delving into low level implementations and shading language gave us the chance to think about game programming in deeper ways. For the future we would like to explore more low level topics in GPU programming and create shaders to explore even more complex lighting simulations.

CONCLUSIONS AND FUTURE WORK

OpenGL and C++ are certainly not the easiest tools to render graphics especially today when powerful tools such as Unity, Unreal and CryEngine are available and accessible to everyone. Despite the relative complexity and inaccessibility though, OpenGL grants unparalleled control over the GPU and its resource optimization. After the steep learning curve and thanks to robust online resources we were able to master its methods and to produce a rendering that is beautiful and believable although not scientifically accurate.

If we are to continue working on this project it would make sense to add the following improvements:

1- Enhanced Lighting model with precise shadows (Eclipses) and individual light properties values for each planet (through accurate research).
2- More scientifically valid Solar system in terms of number of planets, distances and scale.
3- Rendering of the earth atmosphere.
4- More realistic rendering of the burning Sun.

FURTHER INFORMATION

A demo Video of The Project can be found on Youtube: https://youtu.be/zbCkPMF4MAc

The project source code can be found on GitHub: https://github.com/eliodeberardinis/Math_CW4/

REFERENCES

[1] Jeremiah Van Oosten – Introduction to OpenGL and GLSL: http://www.3dgep.com/introduction-to-opengl-and-glsl/

[2] FreeGlut: http://freeglut.sourceforge.net/

[3] GLEW: http://glew.sourceforge.net/

[4] GLM: http://glm.g-truc.net/0.9.5/index.html

[5] SOIL: http://www.lonesock.net/soil.html

[6] Nvidia: http://http.developer.nvidia.com

[7] 3D Game Engine Programming Tutorial

APPENDIX A – ADDITIONAL PICTURES