

**Coursework Report**

**Graphics Programming**

**Module Name:** Graphics Programming

**Module Code:** M3I622944

**Module Leader:** Bryan Young

**Student ID:** S1717551

**Student Name:** Dawid Piotr Kubiak

*I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award*.

*Dawid Kubiak*

<https://github.com/dejwkubikson/Graphics-Programming>

Contents

[1. Statement 3](#_Toc38647843)

[2. Visualisation 3](#_Toc38647844)

[3. Vertex Shader 3](#_Toc38647845)

[3.1 Variables 4](#_Toc38647846)

[3.2 Methods 4](#_Toc38647847)

[4. Fragment Shader 5](#_Toc38647848)

[4.1 Variables 5](#_Toc38647849)

[4.2 Methods 5](#_Toc38647850)

[5. MainGame Class 6](#_Toc38647851)

[5.1 Methods 6](#_Toc38647852)

[6. External Resources 6](#_Toc38647853)

[6.1 Models 6](#_Toc38647854)

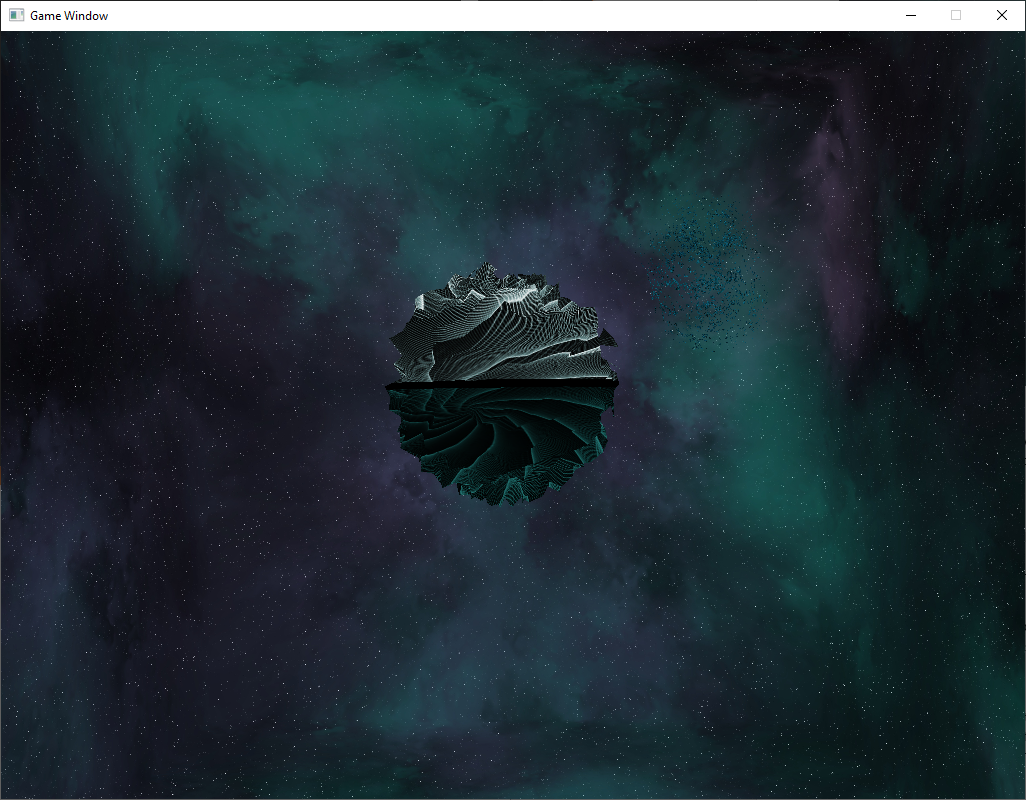
[6.2 Images 6](#_Toc38647855)

# Statement

I have unfortunately not heard back regarding confusion about the coursework. The description says that we need to have three different models in the scene and each of them must contain a different visual effect (one per model). However, one of the Shaders is the environment mapping from ‘lab 7’ which is the Skybox. I have therefore treated the Skybox as a separate model. In my scene, I have the following:

* Two different .obj models - planet and an asteroid, and one Skybox ‘model’
* One camera
* Shader class
* Three shader programs – Skybox shader, Geometry shader and additional shader
* Each shader contains different visual effect, one per model (including Skybox)
* Two of the shaders are exploding shader and environment mapping shader.

# Visualisation



# Vertex Shader

Vertex Shader processes individual vertices of the object which sets up the gl\_Position variable that is used to store the position of the current vertices.

## Variables

### Layout Qualifiers:

vec3 VertexPosition – holds the position of the current vertices that the shader is running for.  
vec2 VertexTexCoord – holds the texture coordinates that the shader is running for.  
vec3 VertexNormal – holds the normal of the plane created by the vertices.

### Uniform Qualifiers:

mat4 transform –transform position of the object is assigned to this variable.  
float time – game’s timer is assigned to this variable.

### Out Qualifiers:

vec2 texCoords – the texture coordinates are assigned to this variable.  
vec3 normal – the normal of the plane is assigned to this variable.  
vec3 position –the position of the current vertices are assigned to this variable.  
float timer –the game’s timer is assigned to this variable.

## Methods

### random()

Returns a float value by extracting fractional content of a sine wave. The dot product, nested inside the sin() method, is used to return a single float value between 0 and 1 depending on alignment of the vertex position vector and vector (10.289, 59.354) to apply the randomness to both X and Y axis. The resultant of the sine wave is multiplied by 34384.84723 in order to randomise the fractional value even more. This is then multiplied by one tenth thousands of the game’s time in order to animate the effect at a reasonable speed. The numbers used to create the vector and multiply the sin() were picked randomly and tested to see what kind of effect they give. This creates a noise surface with pixel to pixel uncorrelated colour values between 0 and 1 (see figure 1).

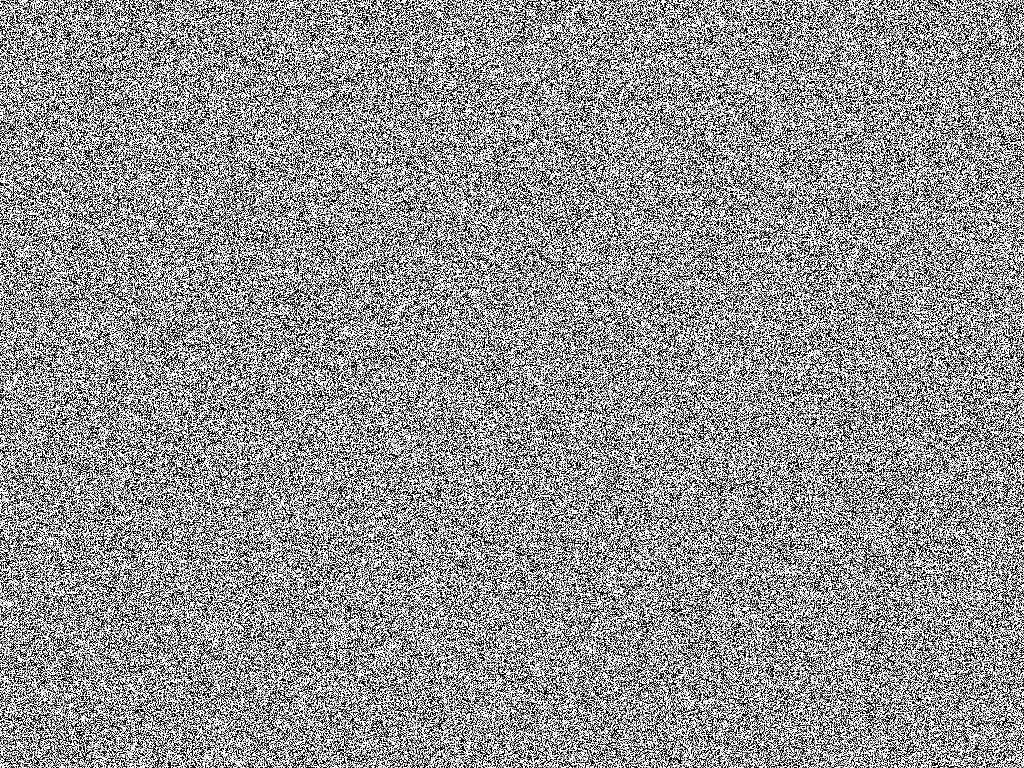


Figure 1. The random() function returns a similar surface.

### main()

The main() method assigns the Out Qualifiers at the very beginning. Then, a vec3 newPos variable is created by summing up the position vector and multiplication of the normal vector and the value returned by the random() method. This resulting vector is then converted to a vec4, multiplied by the uniform transform value and assigned to the gl\_Position variable.

# Fragment Shader

Fragment Shader define red, green, blue and alpha colours for each pixel being processed. This shader is called once per pixel and sets up the gl\_FragColor variable that is used to store the colour of the current pixel.

## Variables

### Layout Qualifiers:

vec4 fragColor – out variable that is used instead of the gl\_FragColor.

### In Qualifiers:

vec2 texCoords – the texture coordinates are assigned to this variable.  
vec3 normal – the normal of the plane is assigned to this variable.  
vec3 position –the position of the current vertices are assigned to this variable.  
float timer –the game’s timer is assigned to this variable.

### Uniform Qualifiers:

sampler2D image – stores the texture (GL\_TEXTURE\_2D) bound by OpenGL.  
vec4 colors[] – array that stores RGBA colours used in the graphical effect.  
int randInt – used to determine whether a ‘storm’ effect should be displayed.  
float randFloat – used to determine where to display the ‘storm’ effect.

### Other Qualifiers:

int indexColor – holds the index value of current colour from the colors[] array.  
float colorLine – holds the value where the colour line is drawn (separates the different colours)  
vec4 currCol – holds the current colour (the one that is being pushed out of the planet)  
vec4 nextCol – holds the next colour (the one that will appear on the planet)  
vec4 showCol – used for easier colour assignment as there may be four colours displayed on the planet at once.

## Methods

### main()

Firstly, the current colour index is calculated by using a modulus operator that divides the timer by size of the colors[] array. The result is converted to an integer and this value will hold the index of the currently used colour. The colour is assigned from the array to the currCol variable. The next colour is determined. The shader checks if the current colour is the last from the array which would mean that the next one is the first colour in the array or if the next colour is just the next item in the array. The colour is then assigned to nextCol variable. As the colour line (that is drawn when the colour smoothly slides) is to be drawn from the top of the model, a dot product of (0, 1, 0) vector and vector holding the texture coordinates is calculated and stored in the colorLine variable. After that, an if statement is used to draw a black colour (by assigning it to the showCol variable) which checks if the colour line minus 0.01 is lower than the current fraction of the timer, and at the same time, if the value plus 0.01 is higher than the current fraction of the timer. If none of the above is true, another if statement checks if the fraction of the timer is higher than the value of the colorLine. If so, next colour is assigned to the showCol variable, otherwise current colour. Then, two float variables are created – vertLine and horiLine. The first one holds the result of the dot product of (0, 1, 0) and normal vectors, and the second one, the dot product of (1, 0, 0) and normal vectors. Then, if the uniform randInt is lower than 25 (25% chance as the randInt is between 0 and 99) randFloat is compared against vertLine and horiLine values. The comparison is similar to the colorLine but the value is 0.012 this time. If the above is true, yellow lines (horizontal and vertical) will be drawn on the model. Finally, there are some modification to the texture itself. The texture coordinates are assigned to the cords variable – this is done to prevent modification of texCoords variable as it is read only. Sine and cosine values of the timer multiplied by 0.8 are stored in sinVal and cosVal variables. The 0.8 multiplication ‘zooms’ in the texture which prevents of showing untextured parts of the model when rotating the image. Vector (0.5, 0.5) is subtracted from the coords, in order to move the point to origin. A rotation matrix (rotate variable) is created, the cords variable is then multiplied by it, and in the end, (0.5, 0.5) vector is added back to the coords variable. The colour value of the texture is returned using texture2D() GLSL method with the image and coords as parameters. This value is multiplied by the showCol and assigned to the fragColor out variable.

# MainGame Class

## Methods

### setShaderEffect()

Without getting too much to the obvious details of how the Shader class works (initializing and binding), only the above function will be analysed in order to explain how the final effect is achieved. At the beginning, the texture is bound using the Texture class’ Bind(unsigned int unit) method. The random number from 0 to 99 is assigned to the randInt variable. The same happens for the randFloat variable, which is later divided by 100 (to get 0 to 1 value). The counter is assigned to the shader’s time variable, randInt to randInt, randFloat to randFloat. Colors array is instantiated with three different colours and is passed to the shader. All the values are passed using appropriate functions in the Shader class.

# External Resources

## Models

### Planet

<https://free3d.com/3d-model/alien-plantet-1-lybathan-13394.html>

### Asteroid 1

<https://www.cgtrader.com/free-3d-models/space/planet/asteroid-2>

## Images

### Skybox

<http://wwwtyro.github.io/space-3d/>

### Planet Texture

<https://en.clipdealer.com/vector/media/A:47629268>

### Asteroid 1 Texture

### <https://dev.medialoot.com/blog/seamless-rock-textures/>