Solar system world explorer game

Tony Liao – UQ

Contents

Introduction

1. project aim

The project aims to develop an open-world game with elements of the solar system and sufficiently demonstrate knowledge of computer graphics techniques. This game will be developed using Blender and Java OpenGL utilities.

Java OpenGL implementation

Java OpenGL (JOGL) is a wrapper library that allows OpenGL to be used in Java programming languages. JOGL provides access to most features of OpenGL from its libraries that were initially written in C language. In this project, I will be using the LWJGL (Lightweight Java Game Library), which enables cross-platform access to native APIs that are useful in developing graphics, such as OpenGL and Vulkan. LWJGL is a technology that provides low-level access, and the access is high-performance and wrapped in a user-friendly layer.

1. VAO and VBO

VAO and VBO are used in OpenGL implementation to implement this project with OpenGL. VAO is a vertex array object that stores all states needed to supply vertex data. It can store the data about the 3d model. VAO has a load of slots in which you can store data, and these slots are known as attributes list:

0	Vertex positions	(VBO 1)
1	Vertex colours	(VBO 2)
2	Normal vectors	(VBO 3)
3	Texture coordinates	(VBO 4)
...	...	(VBO 5)
16	...	(VBO 6)

Figure 1: Vertex array object's attributes list with vertex buffer object

Figure 1 shows the attributes list of the VAO. It can store the vertex position, colors of all vertices or the normal vectors at each vertex, or the texture coordinate at each vertex. This depends on what data you store for the 3d model, and these data stored in the attribute lists are also known as vertex buffer objects or VBO.

VBO is just an array of numbers that represent the data. It can be positions, colors, or normal.

2 Data processing

The 3d model can be represented as a load of data that can be stored in the VAO. Every single 3d model is made from a number of triangles.

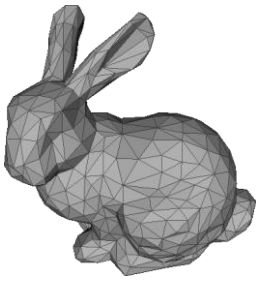

Figure 2: triangle mesh 3d rabbit model in blender

Figure 2 shows that the 3d model has many triangles in different shapes to make up a 3d rabbit model.

Each triangle has three vertices, three points in three-dimensional space, and each coordination represents X, Y, and Z. If we take each vertex of triangles on a 3d model. We will have data with a bunch of coordinates representing all the vertex's positions of the model.

This data can be put into VBO and stored in an attribute list of a VAO.

2.1 Wavefront object file format

To load the data of the 3d model and store it in VAO, I have saved the model in the blender as wavefront object file format. By doing this, the data of the 3d model can be easily saved as several arrays of vertices.

2.2 OpenGL coordinates system

To implement the model with OpenGL, we need to specify the vertices in counterclockwise directions.

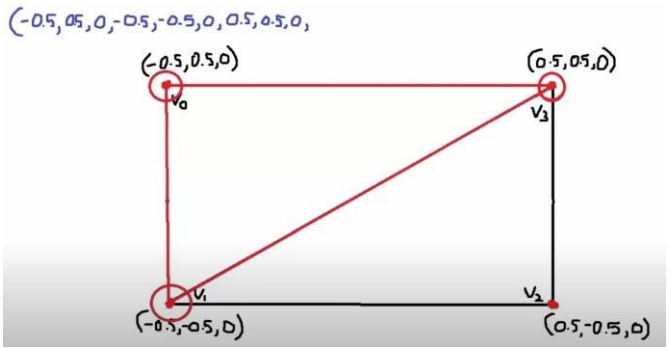

Figure 3: vertices of rectangle

From Figure 3, if we need to tell OpenGL of the vertices of a rectangle, it can be done by representing the list of vertices that is stored in counterclockwise direction.

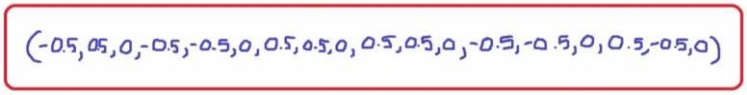

Figure 4: load of vertices of rectangle

After loading the list of vertices, we can render it in the game. This will be inefficient as the vertices might be repeated and loaded twice; thus, I will use the index buffer to tell OpenGL to connect these vertices and make triangles to be rendered.

2.3 OpenGL Index Buffer

The index buffer is a good choice to make it more efficient for OpenGL to connect the vertices. With the old ways, if we have six vertices, 18 floats, and 72 bytes will be used, where three floats are represented as x, y, and z coordinates, and one float is 4 bytes. With Index Buffer, the repeated vertices are not used twice but once, which is 12 floats and 48 bytes. Thus, this shows it uses fewer floats than the old method and is more efficient as the model takes less space in memory. The number of floats depends on the complexity of the model. If the model has lighting, an additional normal vector vertex, another three floats, is added. If there's UV texture for the model, there will also be extra texture coordinates.

3 Shader implementation

There are two types of shaders which are vertex shader and fragment shader.

The vertex shader executes one time for each vertex in the object that is being rendered, and each time it uses the vertex data stored in the thing VAO as the input to the vertex shader program, which would be the X, Y, Z coordinate of the vertex that is being processed.

I have made the vertex shader to do two things for this implementation, first is to determine the position of the vertex that is processing and should be rendered onto the screen. The second thing is to output the value of this vertex shader to the input of the fragment shader so that the fragment shader can output the pixel color we want where the vertex shader executed based on the vertices and the fragment shader executed based on the pixels.

4 Texture implementation

To implement the texture with OpenGL, I have used the texture coordinate and UV coordinates to achieve this:

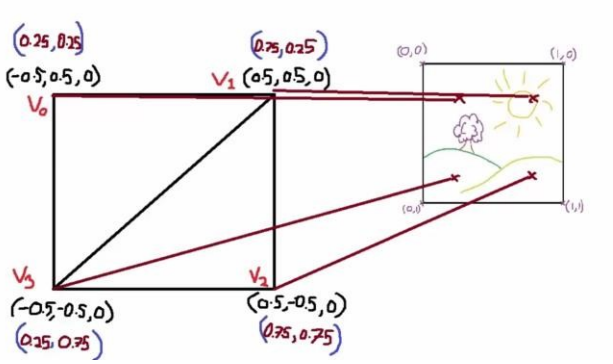

Figure 5: UV coordinates mapping

From Figure 5, each texture coordinate was given in X, Y, Z form and UV coordinates in X, Y form. The UV coordinates are used to determine what point on the texture is mapped to each vertex which v0, v1, v2, v3 were mapped to the texture showing in figure 5, but this does not map to correct spot and only render the area its mapped to. To render the whole texture, as coordinate system in OpenGL for top left is (0,0) and top right is (1,0). Bottom left is (0, 1) and bottom right is (1,1). Therefore, as shown in Figure 6 I have mapped these 4 spots to get my whole earth texture rendered.

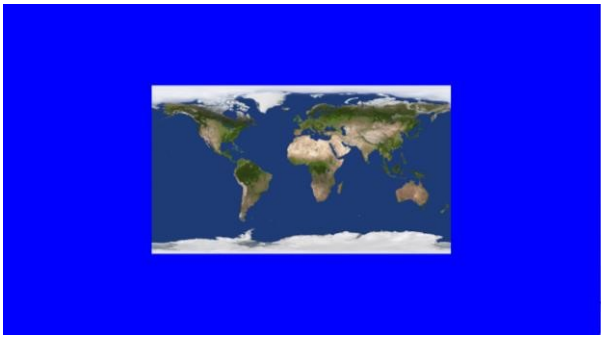

Figure 6: Earth's texture rendered with OpenGL

5 Matrix Transformation

To implement all my 3d models of spheres, each with a different position, rotation, or size but using the same model, I have used the transformation of the 4x4 matrix to achieve this.

The transformation matrix is a 4x4 matrix, and the vertex position is a 1x4 matrix with one at the very bottom of the row.

6 Projection matrix

To implement the view that makes an object smaller when the thing is at a far distance and appears bigger at a closer distance, the projection matrix is a great technique to get this done. The projection matrix changes the area of view, which gives a wider view of the object in the distance and makes them appear smaller when further away.

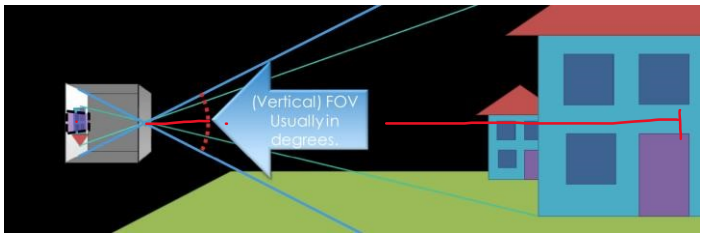

Figure 7: the field of view angle

To determine how wide the view is, the field of view can be used to define the exact shape of the frustum and the far plane distance, which is how far into the distance we can see.

7 Lighting

To implement the lighting of my 3d object, the lighting calculation is required to determine which surface faces the light and it appears brighter. To do this, we need to know which direction the model is facing, and this is where normal vectors come into place.

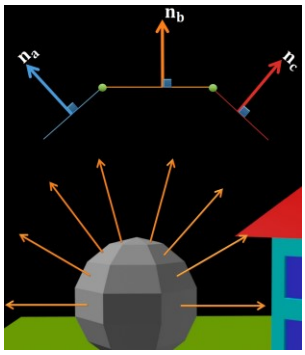

Figure 8: the normal vectors are perpendicular to the faces

To implement the lighting of my 3d object, the lighting calculation is required to determine which surface faces the light and which appears brighter. To do this, we need to know which direction the model is facing, and this is where normal vectors come into place.

The normal vectors point to the exact direction the surface faces, and every point on the surface has a normal vector perpendicular to the face, as shown in Figure 8. This is also included in the wavefront object file once I exported the blender model to the object file format, and the normal vectors are stored in VAO to get the data to use for the lighting implementation.

7.1 Light source

After retrieving the normal vertex from the object file, I now need to calculate how bright each point on the object should depend on the light source position.

To do this, after the normal vertex is saved in the object file, I find the vector from the point on the object to the light source.

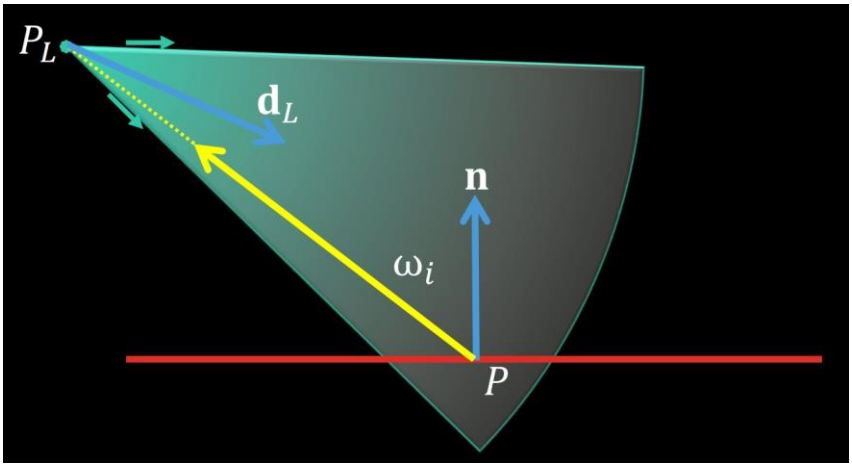

Figure 9: vector from vertex to light source and normal vector

By calculating this, one object will have two vectors, where one is the normal vector that indicates the direction of the object is facing, and one indicates the direction of the light source, as shown in Figure 9. Then to determine the object's brightness, we can compute these two vectors as if two vectors on a point are very close to each other, then this point will be brighter as two vectors pointing in identical directions. To do this calculation, I used dot product as it is well presented for two vectors of their similarity, as shown in Figure 10.

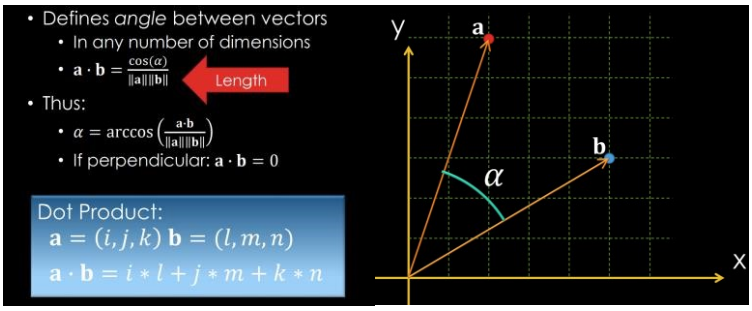

Figure 10: the dot product of two vectors

This can be done in OpenGL easily with just two lines of codes in vertex shader:

```
void main(void){
    vec4 worldPosition = transformationMatrix * vec4(position,1.0);
    gl_Position = projectionMatrix * viewMatrix * transformationMatrix * vec4(posi
    pass_textureCoordinates = textureCoordinates;

    surfaceNormal = (transformationMatrix * vec4(normal, 0.0)).xyz;
    toLightVector = lightPosition - worldPosition.xyz;

}
```

Where `surfaceNormal` variable represents the normal vector and light vector has named to `toLightVector`.

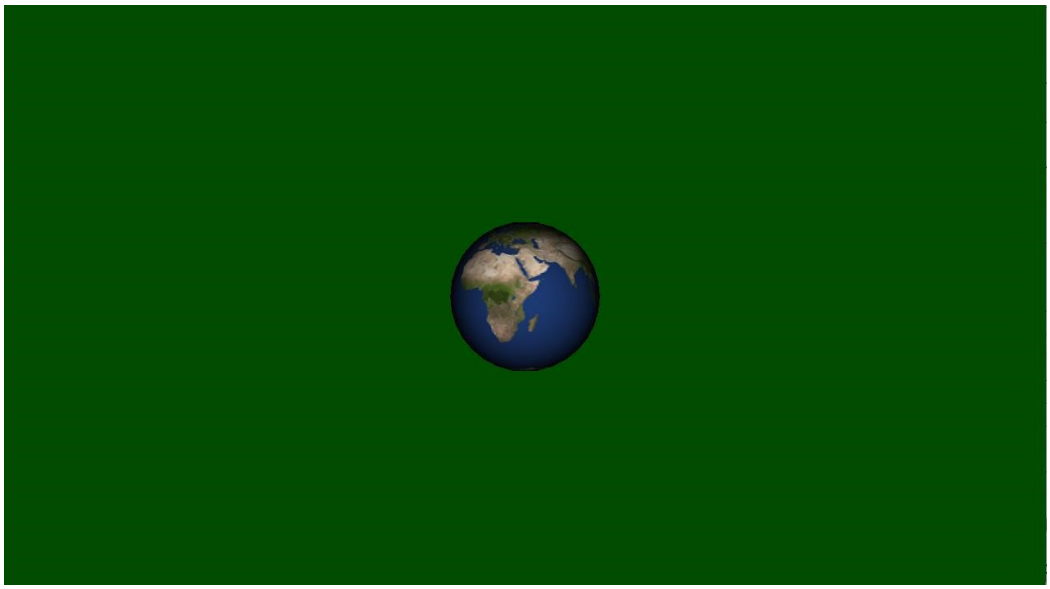

Figure 11: Earth texture applied on sphere with light source

7.2 Specular Lighting

To make my 3d model look shiny, I have utilised the specular lighting technique. Specular lighting is the reflection light on shiny objects based on surface's reflective properties.

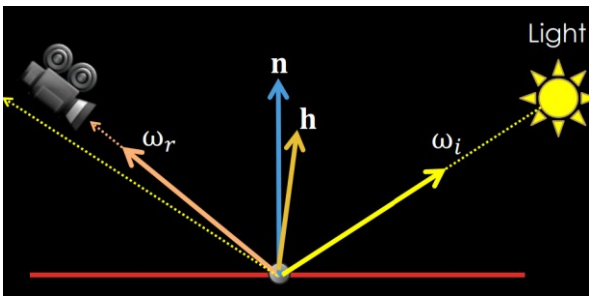

Figure 12: the reflection of light source

The specular brightness depends on the position of the camera. If the light source goes straight into the camera, it will have high specular brightness. From Figure 12, the reflected light vector is pointed a little away from the camera, resulting in lower specular brightness. To calculate the differences between these two vectors, the dot product is also used for this calculation.

The dot product calculation is supported by the OpenGL library:

```
float specularFactor = dot(reflectedLightDirection, unitVectorToCamera);
specularFactor = max(specularFactor, 0.0);
float dampedFactor = pow(specularFactor, shineDamper);
vec3 finalSpecular = dampedFactor * reflectivity * lightColour;

out_Color = vec4(diffuse, 1.0) * texture(modelTexture,pass_textureCoordinat
```

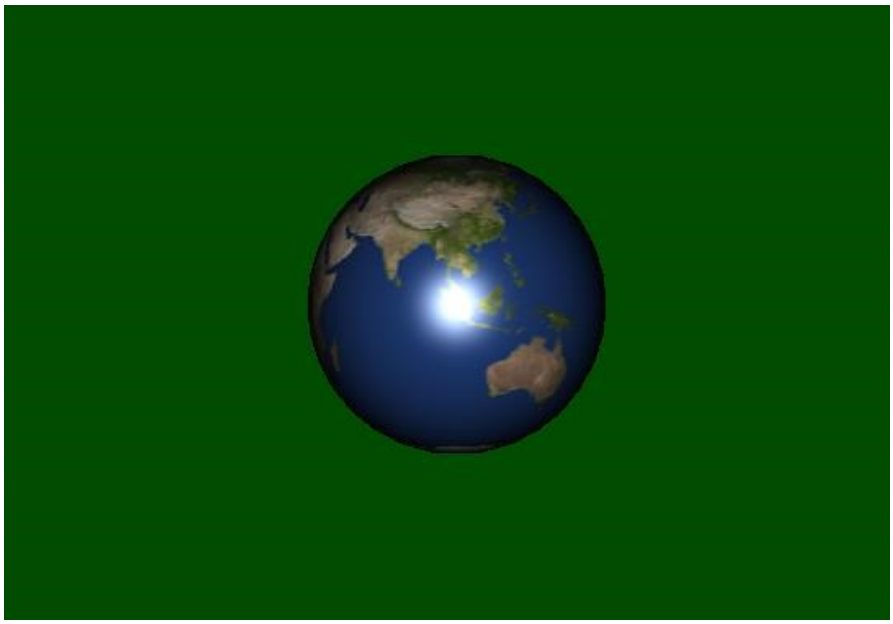

Figure 13: The specular lighting applied on earth texture

8 Optimize Rendering

To render many objects, this will be very inefficient if we render the object one by one. To overcome this issue, I have used the hash map in Java to optimize this behavior of rendering objects which can render many objects only once for all objects.

9 Terrain implementation

To implement the terrain with OpenGL in my game, I use regular vertices to generate a terrain in the code instead of creating the model in Blender.

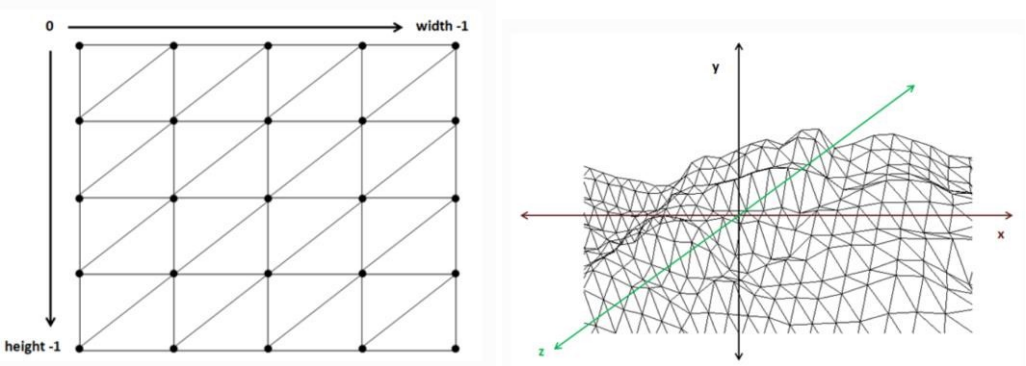

Figure 14: The height vertices of terrain

To design a flat terrain, I set the height of each vertex to 0 and then applied a simple texture to it.

9.1 Terrain texture implementation

Implementing the texture on the terrain without a certain technique will stretch it out on its surface. To solve this problem, I have used the tiling technique, as shown in Figure 15, which renders the same texture multiple times on the terrain surface so that the texture won't be stretched out.

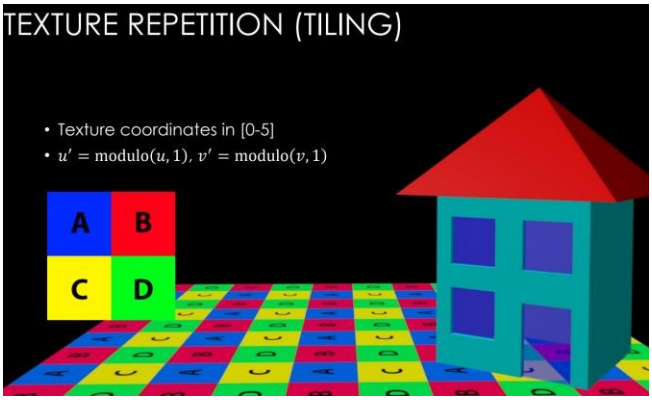

Figure 15: texture repetition

This is easy to achieve in OpenGL. I just increased the texture coordinates. Once the coordinates go over one, OpenGL will start from 0 again, as shown in Figure 16.

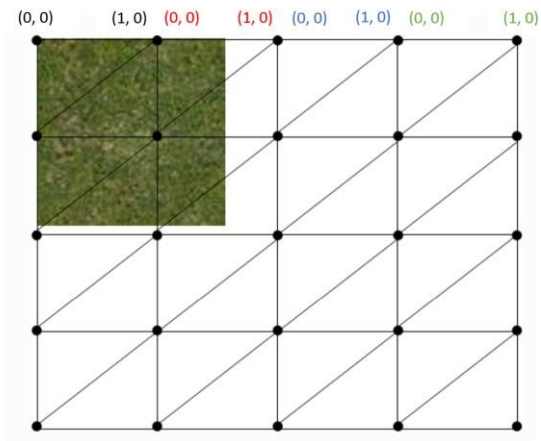

Figure 16: texture coordinates for terrain

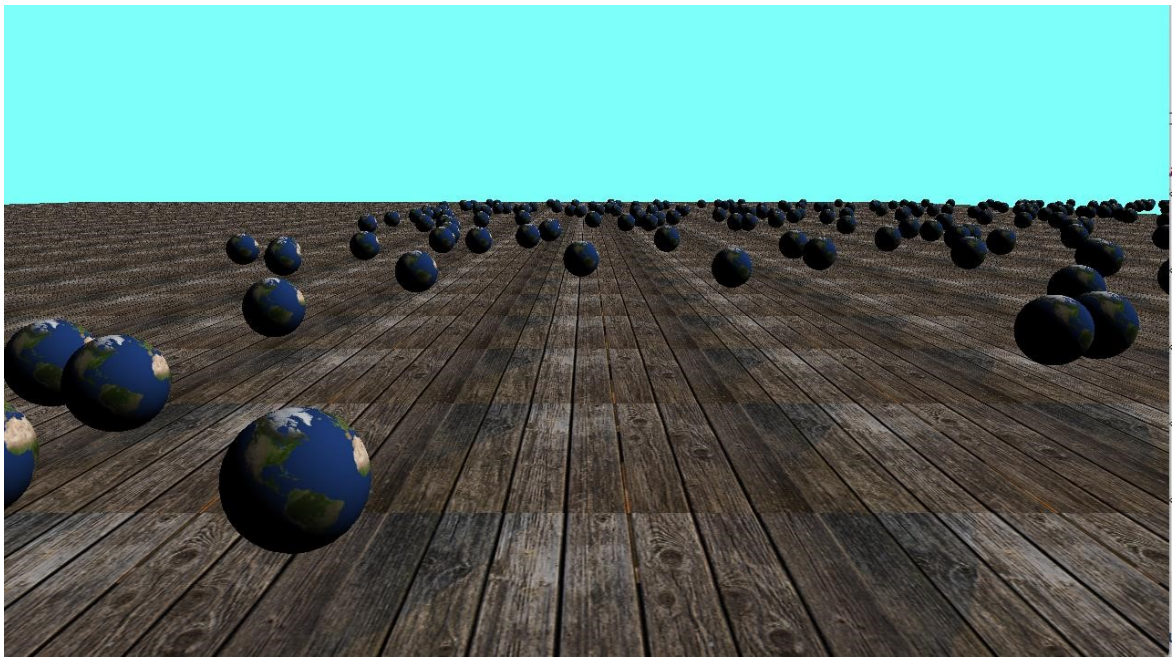

Figure 17: multiple earth 3d models on wooden texture flat terrain

Using this technique, I have successfully implemented multiple earth spheres on the flat terrain with wooden texture in the game without being stretched out.

10 Transparency implementation

Transparency is not easy to implement in OpenGL. OpenGL will see an object that has some transparency and calculates the color of the rendered object as a mixture of colors and renders the transparent object. But this led to a problem, in which other rendered objects behind the

transparent object will not appear through the transparent object when the transparent object has been rendered, and it will never change or update the color of the transparent object.

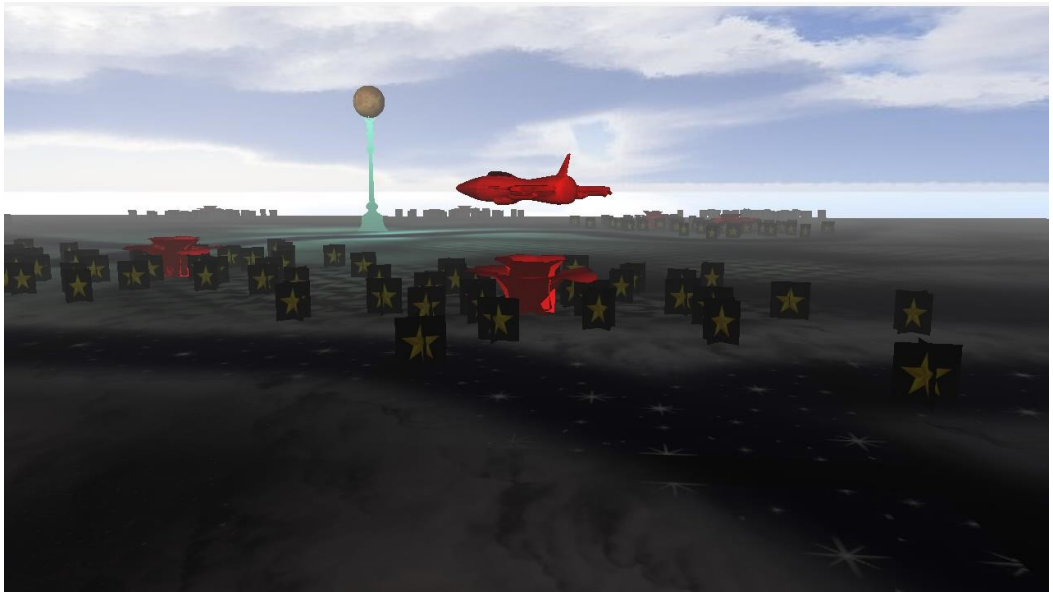

Figure 18: star object's dark border covered the red flower

The illusion of transparency is broken, as shown in Figure 18. To solve this problem, I tell OpenGL not to render the transparent bit of the object. This avoids all the issues mentioned before.

11 Fog implementation

To implement the fog in the game, as shown in Figure 19, I simulated fog by fading the objects into the color of the sky depending on how far the object is. The object from a far distance will be rendered entirely in the sky color, which will appear hidden by fog.

Figure 19: Fog effect in game

12 Render multiple textures

To make my game's terrain look better, I decided to add paths on it. To do this, I have to tell OpenGL to render multiple textures using the blend map.

Figure 20. Blend map for the game

The blend map represents multiple textures on terrain, and the colors represent multiple textures where green, red, and blue represent three different textures. I have created a blend map paint and referenced the blend map in shaders to let OpenGL know where we should render each texture.

Usually, in OpenGL, one texture fits in one unit. But luckily, OpenGL also allows you to fit different textures in separated units than can be accessed in the shader.

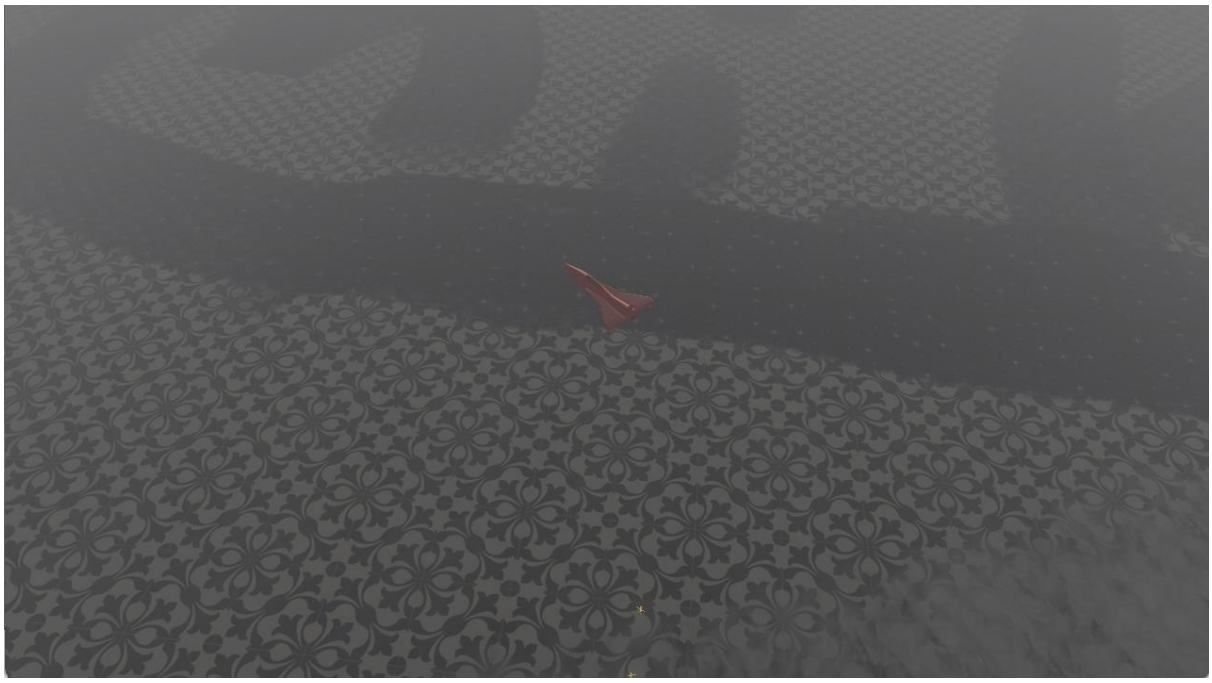

Figure 22: multiple textures applied on terrain in the game

As shown in Figure 22, I have successfully used my own designed blend map and rendered different textures on the terrain in the game.

13 Player in the game

To implement a player in this game, I calculated the frame rate per second of the render and removed the camera movement, which the player movement will replace.

The player's moving speed depends on the frame rate per second. The faster the frame rate is, the faster the player moves.

Then I can determine the player's moving direction and rotation of the player:

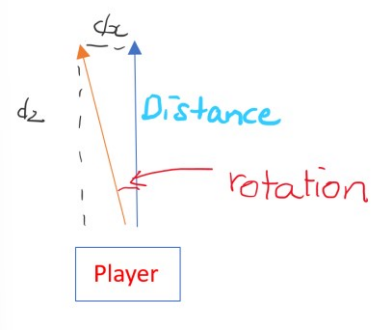

Figure 23: the player next position calculation.

To determine the next position of the player after rotating to the left, we have to know how far dx and dz are. Then we can perform dx and dz transformation to the current x and z position, which will give the new position of the player.

These dx and dz values can be retrieved by using the simple trigonometry formula:

$$\sin(\theta) = \frac{x}{d} \rightarrow \quad x = d \sin(\theta)$$

$$\cos(\theta) = \frac{z}{d} \quad \rightarrow \quad z = d \cos(\theta)$$

Those values need to be calculated to compute the player's next position after the movement using a math function in Java, as shown in Figure 24.

```
float dx = (float) (distance * Math.sin(Math.toRadians(super.getRotY())));
float dz = (float) (distance * Math.cos(Math.toRadians(super.getRotY())));
super.increasePosition(dx, 0, dz);
```

Figure 24: code implementation for player next position

Figure 25: spaceship stand on star, space, and cloud textures with floating stars.

From Figure 25, I used the spaceship as the controllable object for the player and currently, it can move forward and backward and do rotation.

14 Camera

Trigonometry math is required to implement the camera in this game with a third-person view.

Figure 26: camera designed for different views.

To implement the camera's functionality, I have planned from 2 perspectives, distance, and angle, for different views, as shown in Figure 26.

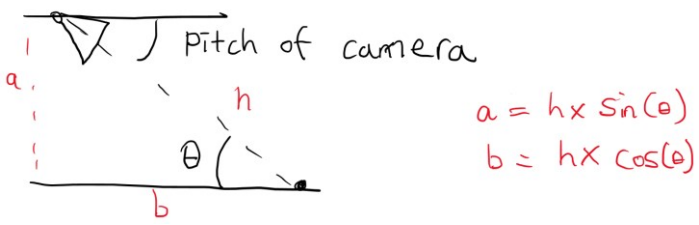

Figure 27: The calculation for the camera.

As shown in Figure 27, to find the distance between the camera and the player, I must compute a and b, the formulas used for the player's movement, which is the hypotenuse multiple by sin theta or by cos theta.

After designing this camera, I then faced a problem: the camera won't follow the player if I move the camera around. Then I used and computed the value of the yaw of the camera to overcome this issue:

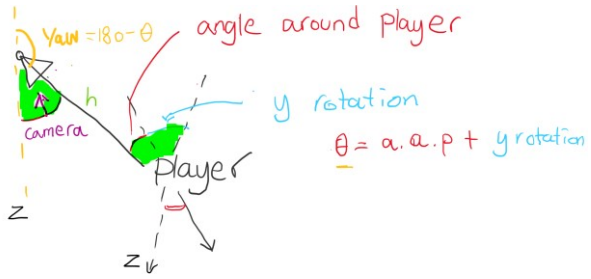

Figure 28: the camera yaw calculation.

Where the camera's yaw is just 180 degrees, subtract the angle around the player and y rotation of the player. This will ensure the camera always follows the player's movement while moving around the camera.

15 Mipmapping

To improve the performance of the rendering, I have used a technique called mipmapping which only rendered low-resolution texture for the object far away from the camera and is not noticeable. This is easy to get it done as OpenGL has provided the library function to achieve this purpose which is GL30.glGenerateMipmap().

16 Light attenuation

The lighting caused by a light gets dimmer if object is further away from the light source. This is called light attenuation as in figure 29.

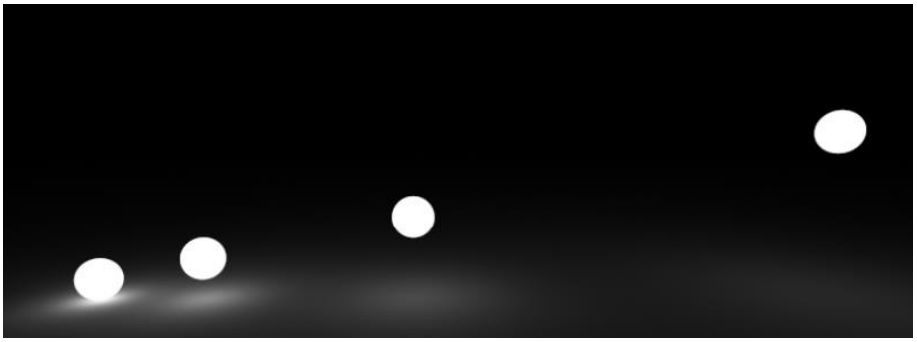

Figure 29: The light attenuation

I decided to implement it for my eight-planet lamps, which can show cool effects and draw the user's attention quickly. This is not easy to implement. To do this, I have calculated the attenuation factor and the approximated original brightness. The light brightness will vary with distance, and the longer distance, the dimmer the light is. Of this fact, I have created an attenuation factor that is inspired by the power series expansion.

$$\sum_{n=0}^{\infty} nx^n = x + 2x^2 + 3x^3 + 4x^4 + \dots$$

Figure 30: power series expansion

From figure 30, I have used the constant coefficient and x as the distance value. Then use the current brightness divided the computed attenuation factor.

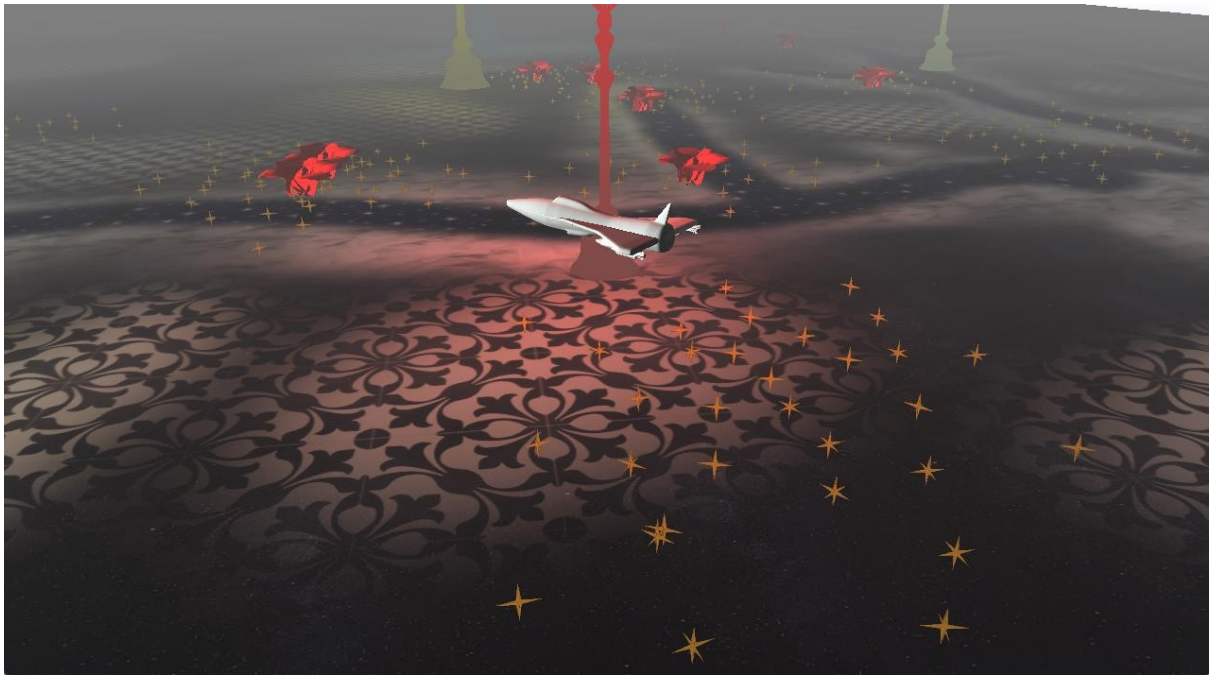

Figure 31: The light attenuation in the game

As the result shown in figure 31, I have successfully implemented the light attenuation for the game by this method.

17 Skybox

Several techniques have been used to implement the sky box in the game. A skybox is a method that makes a game level appear larger by creating the background. Which is simply a box with a nice texture of sky all around it, and the camera will always be at the center of the box and never reach the edge of the skybox.

17.1 Cube map textures

A cube map texture is the 2d textures on each side of a 3d cube, and the whole is just counted as one texture. A cube map is perfectly suited for creating a skybox instead of using six individual textures to texture the skybox. I created six separate images but stored them as one texture to do this. This means that before rendering, I only must bind that one cube map and implement the skybox more efficiently.

17.2 Sampling texture from cube map

Usually, when we sample a 2d texture, we can use a two-dimension vector to get the location of it.

But in a cube map, it is a 3d direction vector for each location of the texture sampling.

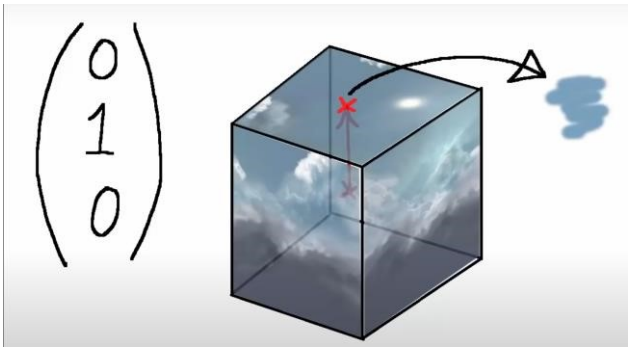

Figure 32: texture sampling from cube map

As shown in Figure 32, sampling the texture from a cube map is a three-dimensional vector, whereas (0, 1, 0) points up in the direction. This shows the flexibility and ability of texture sampling from cube maps which we can sample cube maps using the relevant direction vector. This is why the cube map is perfect for the skybox, as we can use the vertex position of the cube as the direction vector for sampling the cube map.

17.3 Sky textures splitting

To render the textures into six faces of the cube map, I have used the image splitting tool to achieve this.

Figure 33: the skybox texture used in the game

From Figure 33, I have obtained the skybox texture from outside resources and then split it into six textures for six faces, which are front, back, top, bottom, left, and right as shown in Figure 34.

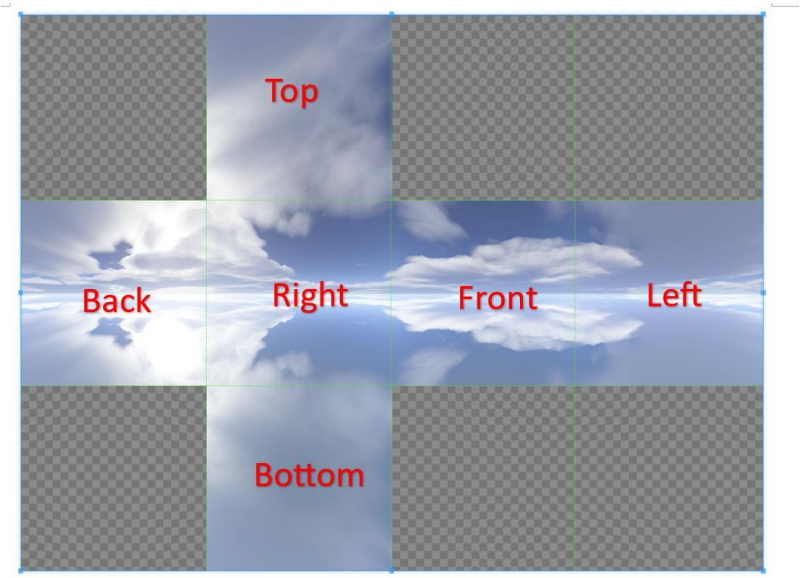

Figure 34: the skybox texture positions after split

After retrieving the textures for each face of cube map, then in OpenGL I have used the code from outside recourse as well to implement the skybox:

```
int texID = GL11.glGenTextures();
GL13.glActiveTexture(GL13.GL_TEXTURE0);
GL11.glBindTexture(GL13.GL_TEXTURE_CUBE_MAP, texID);
```

Figure 35: textures for cube map implementation in OpenGL

This code snippet in figure 35 first the texture is active by OpenGL so that the textures can be used. Then we bind the texture and activate texture unit 0 so we can store data in it and assigned the first parameter to `GL13.GL_TEXTURE_CUBE_MAP` to indicate to OpenGL that this texture is going to be a cube map.

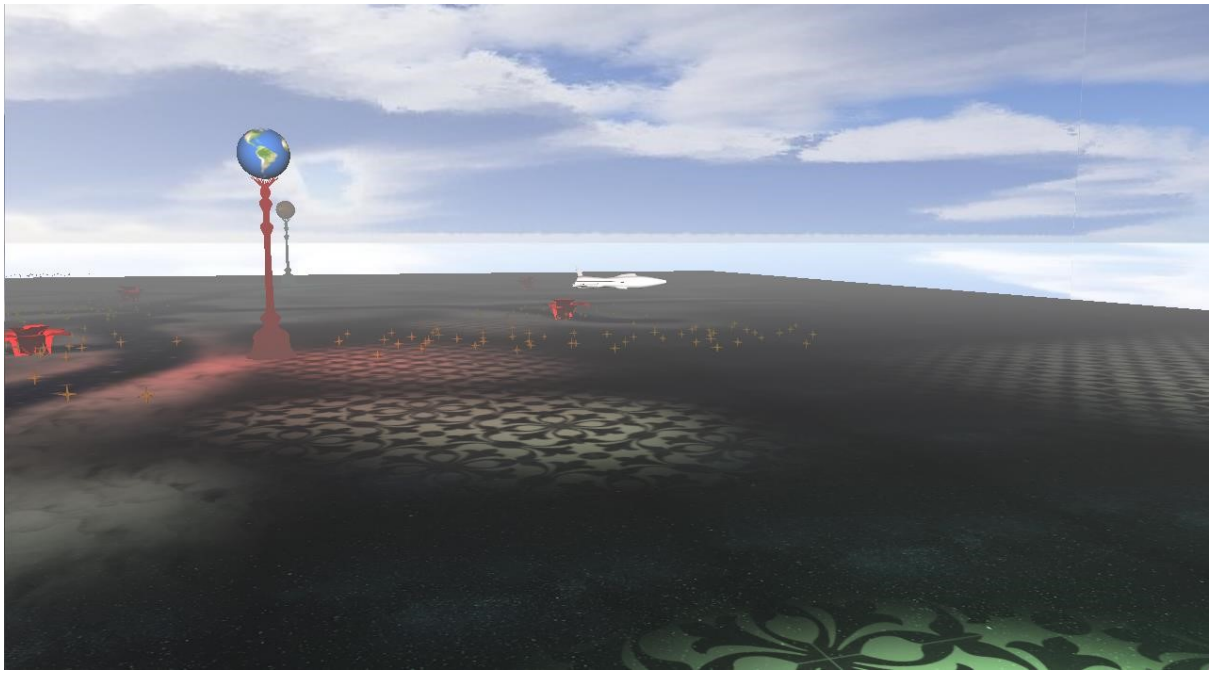

Figure 36: skybox in the game

After implementing the skybox as shown in figure 36, the whole game looks much better and provides a better visual for the sky in the game world.

18 Particle Effects

My initial idea is to render particle effects on the plane and toggle by when the user switches to the flying mode when the space button is pressed. This will make the game more fun and interesting to explore the solar system game world.

To implement the particle effects on the plane, it will be just multiple small quads that are being rendered in the game. The first thing I need to ensure is that particles are always facing the camera, so the particle doesn't perform its rotation. This can be done by applying matrix transformation, allowing the camera to look around the world and see objects from different angles. The way to do this is to multiply the view matrix with the model matrix and give the result of the model view matrix, which is the matrix containing no rotation at all, and this comes with the matrix math:

$$\mathrm{M} = \mathrm{T}(P) = \begin{bmatrix} 1 & 0 & 0 & P_x \\ 0 & 1 & 0 & P_y \\ 0 & 0 & 1 & P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 37: the 4x4 matrix identity on first three rows

As shown in Figure 37, this is the matrix with identity on the first three rows, which means there's no rotation at all, and that's exactly what the model view matrix should look like. To achieve this, I apply the model matrix to be the transpose of the view matrix, which will always give the result of this 4x4 matrix with identity on the first three rows by the implementation in Figure 38.

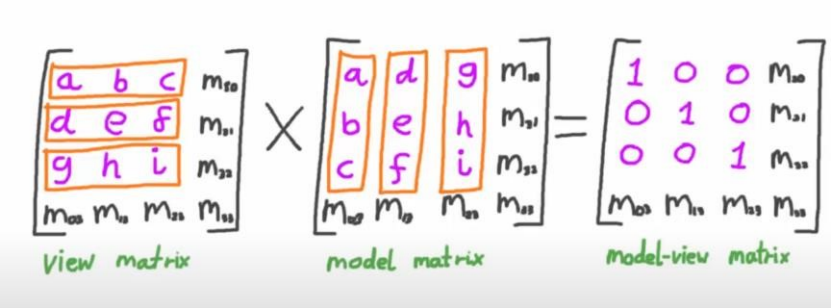

Figure 38: the view matrix multiplied with model matrix (transpose of view matrix)

This can be done in OpenGL with following codes:

```
Matrix4f.rotate((float) Math.toRadians(rotation), new Vector3f(0, 0, 1), modelMatrix, modelMatrix);
Matrix4f.scale(new Vector3f(scale, scale, scale), modelMatrix, modelMatrix);
Matrix4f modelViewMatrix = Matrix4f.mul(viewMatrix, modelMatrix, null);
shader.loadModelViewMatrix(modelViewMatrix);
```

After this implementation, the particles can render without rotation and work perfectly as expected.

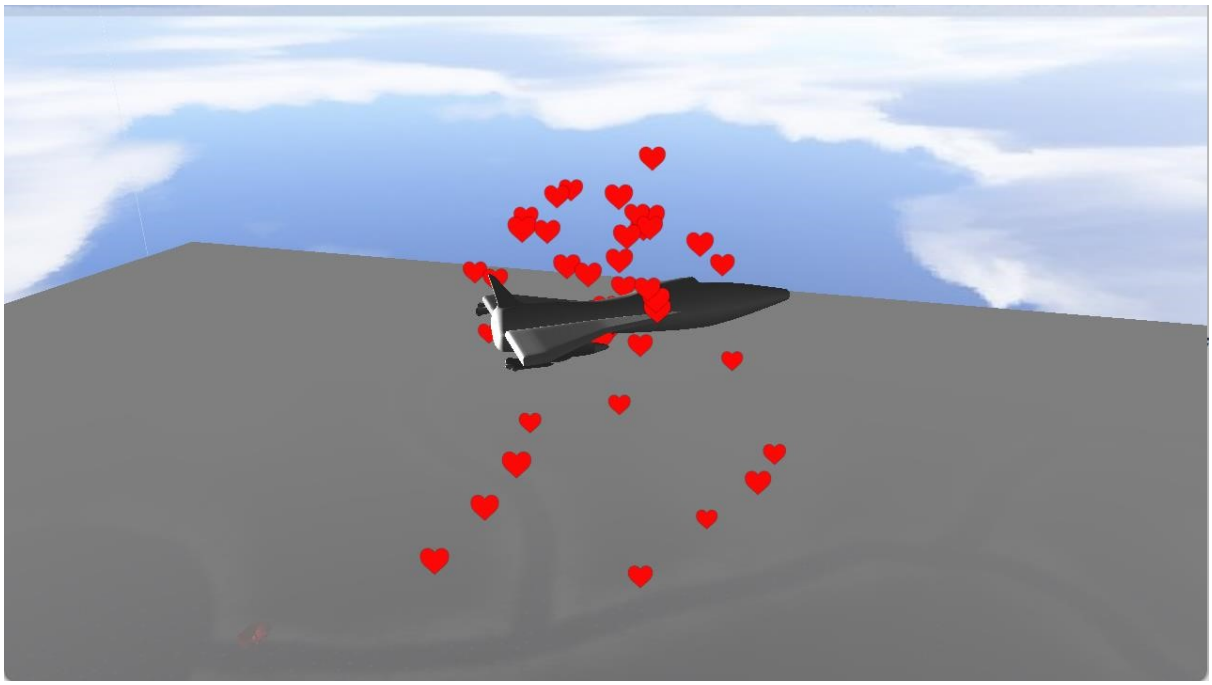

Figure 39: the particle effect (with heart texture)

From Figure 39, I have chosen the heart texture for the spaceship's particle effect to make it looks more romantic.

19 Anisotropic Filtering

The anisotropic filtering is just an improved version of mipmapping that I have done in the previous chapter, where the object will be rendered lower resolution as the object move far away from the camera. But this will lead to a problem if the texture notice that the surface only takes up fewer pixels but at different angles. This surface is much wider than that number of pixels, so the texture will be stretched out on the surface and causes the blurring. To fix this, anisotropic filtering is one of the solutions. Anisotropic filtering in this game achieved this by not only using linearly downscaled versions of texture like in mipmapping and used the lowresolution version of the texture at different aspect ratios.

20 Antialiasing

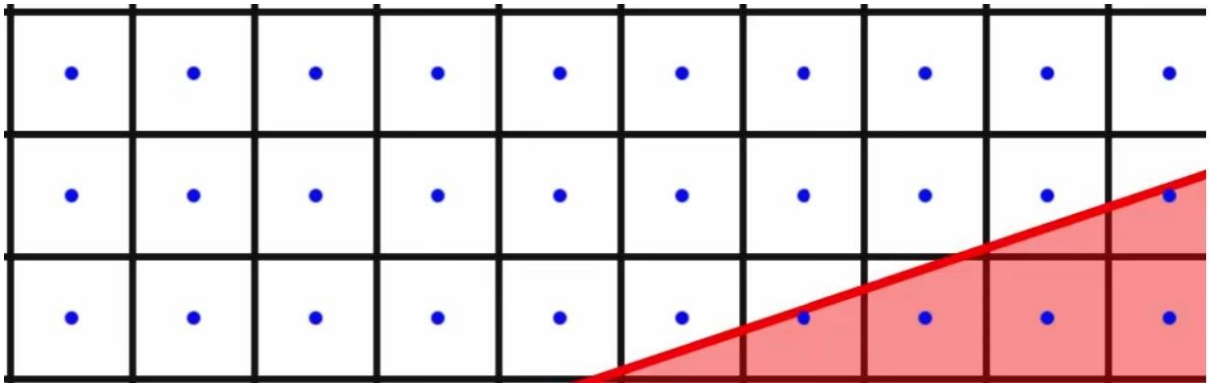

Figure 40: The edge of some polygon to be rendered with blue dot in middle.

From Figure 40, OpenGL tests which pixels should display the polygons by taking a sample represented as the blue dot in the middle of each pixel. If the sample is inside, then the pixel inside is considered to be inside the polygon.

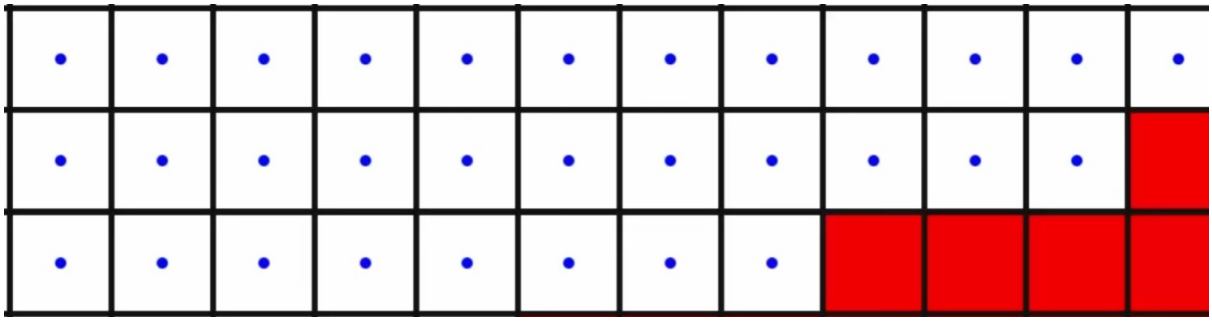

Figure 41: The pixels inside the samples

From Figure 41, after OpenGL tests which pixels should display the polygons, the pixels are either inside or outside the polygon, creating a very hard edge and making the individual pixels obvious.

To prevent the edges of objects from appearing quite jagged and pixelated, the screen is just made up of a finite number of pixels. There's a technique called anti-aliasing that is perfectly suited for this issue. In this game, I will use Multisampling Anti-Aliasing (MSAA) to make the edges of objects look smoother and more natural.

20.1 Multisampling Anti-Aliasing

Multisampling anti-aliasing addresses this problem by taking multiple samples per pixel instead of just one, as shown in Figure 42.

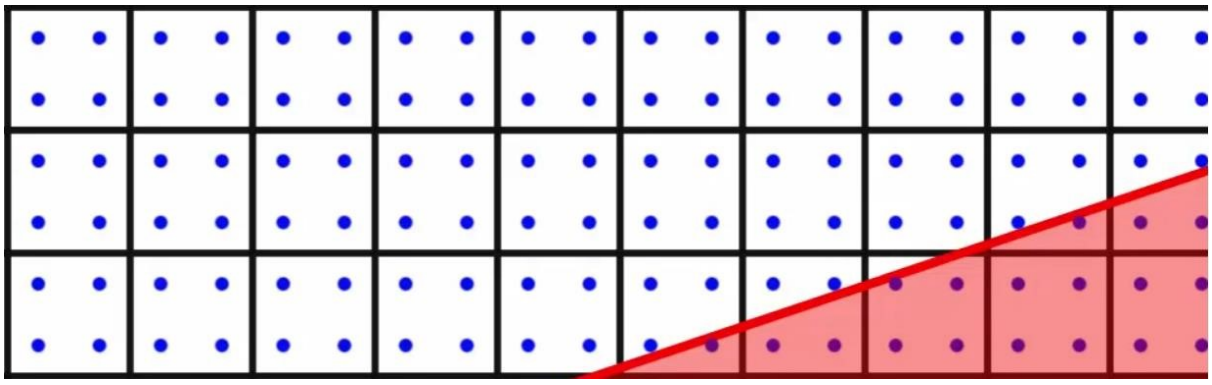

Figure 42: Multiple samples per pixel

This time four samples are going to be taken for each pixel and determine whether each of sub sample is inside the polygon or not.

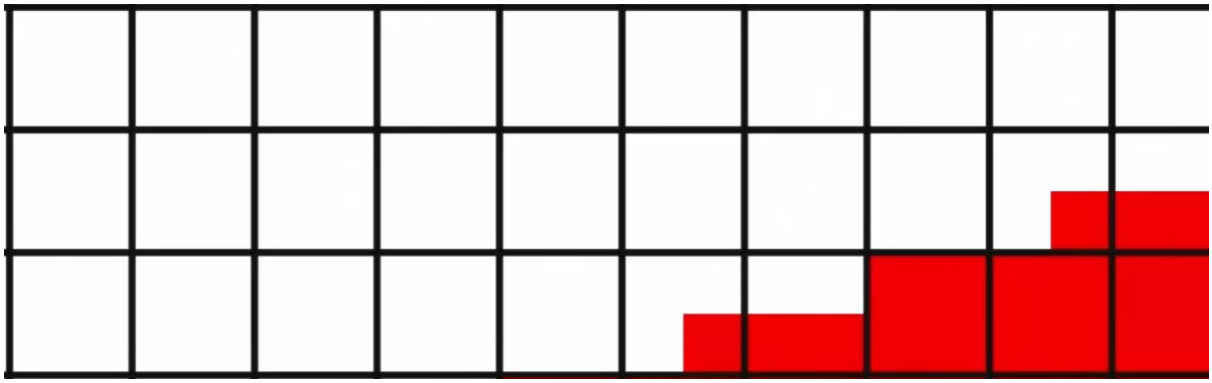

Figure 43: The pixels inside multiple samples

However, this can't just display on the screen, as some pixels only display one colour. I have down sampled this to the screen resolution before displaying it in the game to resolve it. Just imagine that the colour of each pixel is determined by averaging all the sub-samples in that pixel, as shown in Figure 44.

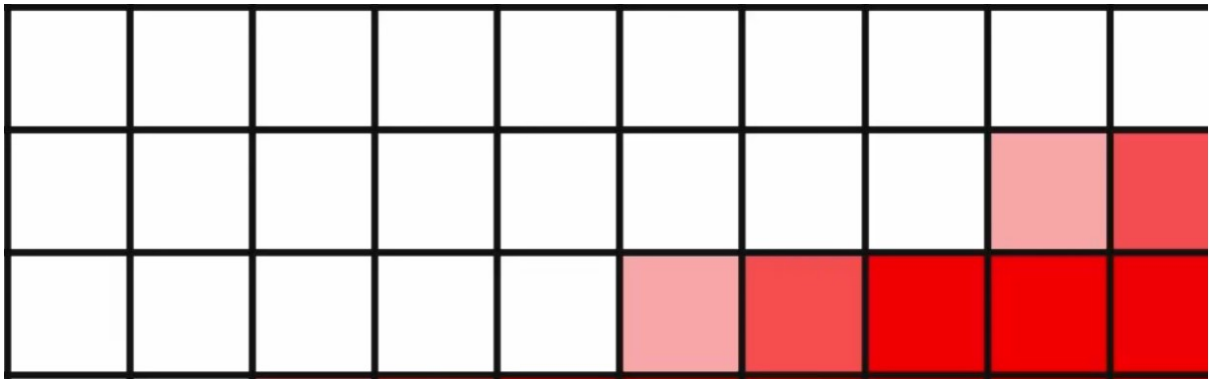

Figure 44: Averaging all of the sub samples

This can be done by the following line of code:

```
Display.create(new PixelFormat().withSamples(16), attribs);
```

I have chosen 16 samples that will be used per pixel as it will give the better quality and without further cost of performance.

21 Post processing effect

Post processing works by rendering the scene to a texture rather than straight to the screen, basically it has an off-screen image of the 3d scene then process that image the same way as

photoshop. To implement this, I first used the frame buffer objects which is combination of the several types of screen buffers:

- A colour buffer for writing colour values
- A depth buffer to write and test depth information.
- A stencil buffer to discard certain fragments based on some conditions.

The rendering of scene to a different frame buffer allows me to do cool post processing effects in the game.

21.1 Gaussian blur

My idea was to implement and gives the viewer the illusion of the bright light sources and brightly lit regions in the game. This glow effect is achieved with a post-processing effect called bloom, which is required to blur the image first and extract the bright regions of the scene. To do this, I have used a technique called Gaussian blur. Gaussian blur is based on the Gaussian curve, can be mathematically represented in different forms, and generally has a bell-shaped curve.

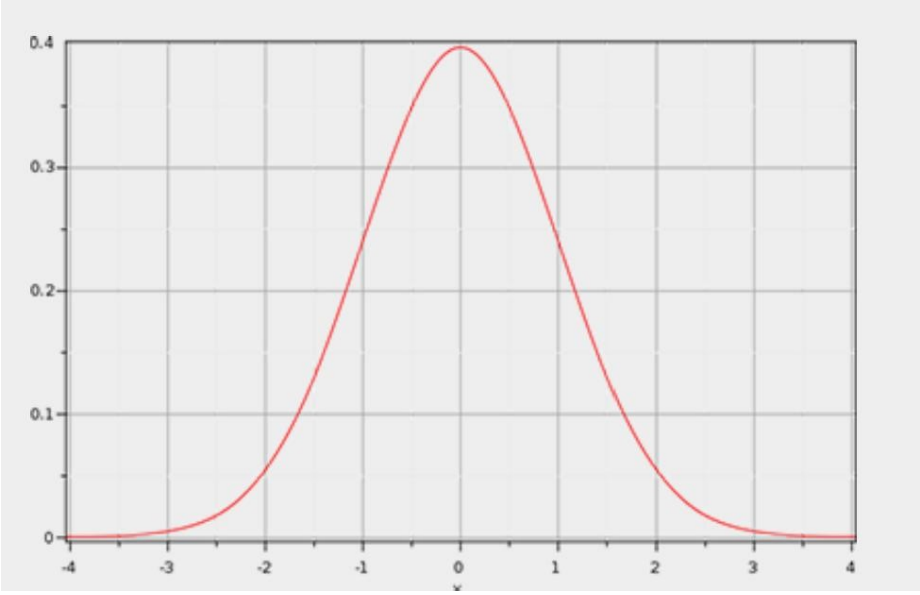

Figure 45: Bell curve shape for Gaussian blur

As shown in Figure 45, if the Gaussian curve has a larger area close to the center of the bell shape curve, using its values as weights to blur a scene will result more naturally as samples that were close by have higher precedence. Each pixel has its weight to determine how much they affect the output color, and those pixels further away from the center will have less effect

on the output and have lower weights. Therefore, I can sample a larger area of pixels that will result in a blurrier and give a better and more realistic blur called Gaussian blur.

I have taken a blur kernel to sample texture for each fragment to implement the Gaussian blur in the game.

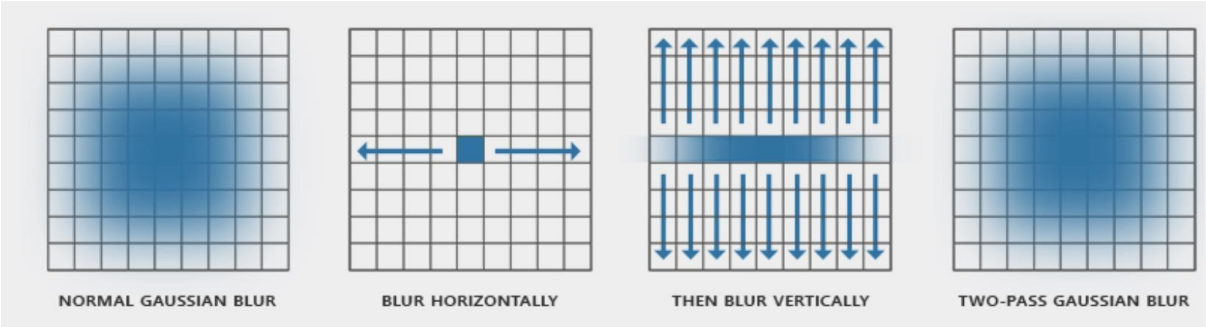

Figure 46: Different stages of Gaussian blur kernel

As in Figure 46, a blur, all the pixels in the kernel would have their weight to determine and find the output colour where the pixel close to the centre of the pixel will have the biggest effect on the output colour. However, if we used normal Gaussian blur, this would not be very good for performance. Instead, I do a horizontal blur with horizontal weights, then the vertical blur. Splitting the blur into stages reduces a huge number of samples per pixel.

21.2 Bloom effect

The bloom effect can make things look like they are glowing. To implement this, I have created two frame buffer objects (FBO) where one contains bright parts of the scene that will glow, and one contains the scene as the normally rendered scene.

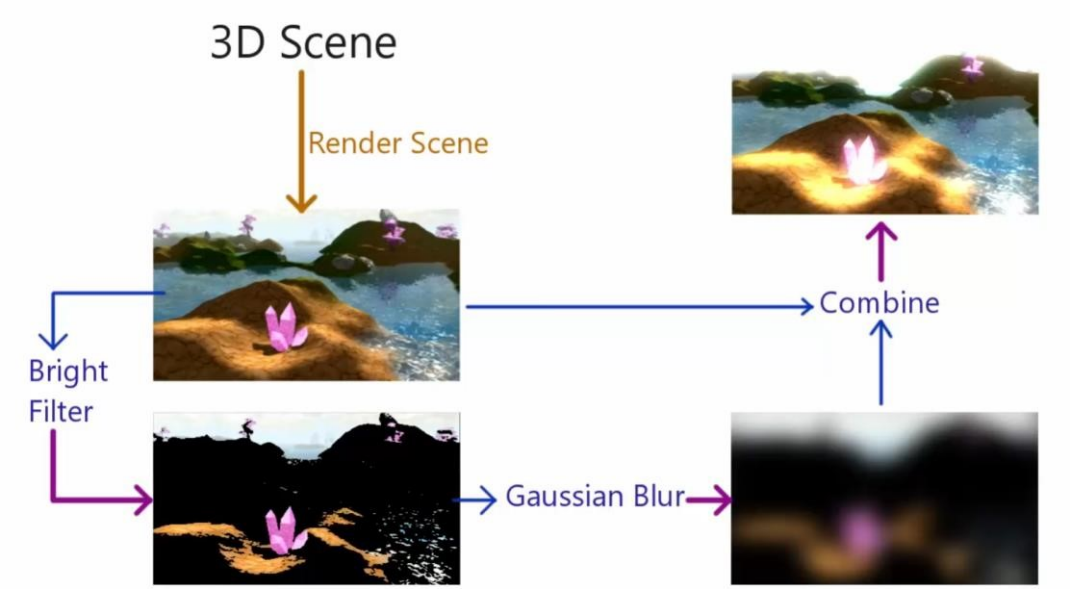

Figure 47: flow chart of bloom effect implementation

As in Figure 47, after creating two FBOs, I just blurred the scene using Gaussian blur and then combined all previous stages to get a result of the bloom effect on an object.

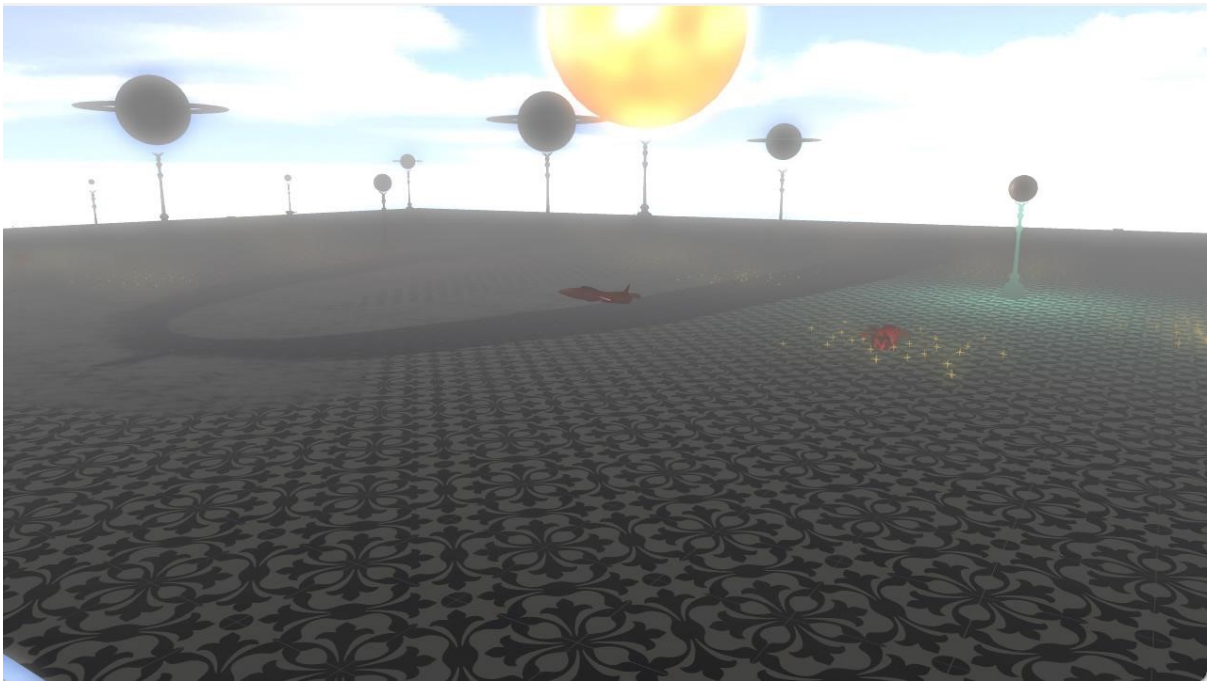

Figure 48: The bloom effect in the game

I then implemented the bloom effect by this method successfully, as shown in Figure 48. I have achieved the bloom effect on the sun, star, and skybox. This gives more realistic and greater visuals for the game.

Blender implementation

To implement the 3d model in this project, I have used the blender to achieve this. Blender is a free and open-source 3D computer graphics software tool that is handy for beginners.

It provides lots of functionality and features for the user to create a 3d model, either low poly or high poly. This makes it the best choice for this project, as it supports the wavefront object file format and easily works with OpenGL together.

1. F16 Flighting Falcon Spaceship

The first idea was to develop a simple human model as the player, but this will take lots of time to create a blender model for a high poly human 3d model. To mitigate this, I have created an F16 Flighting Falcon as the spaceship in the game world, which saved lots of time for completing this project, as shown in Figure 49.

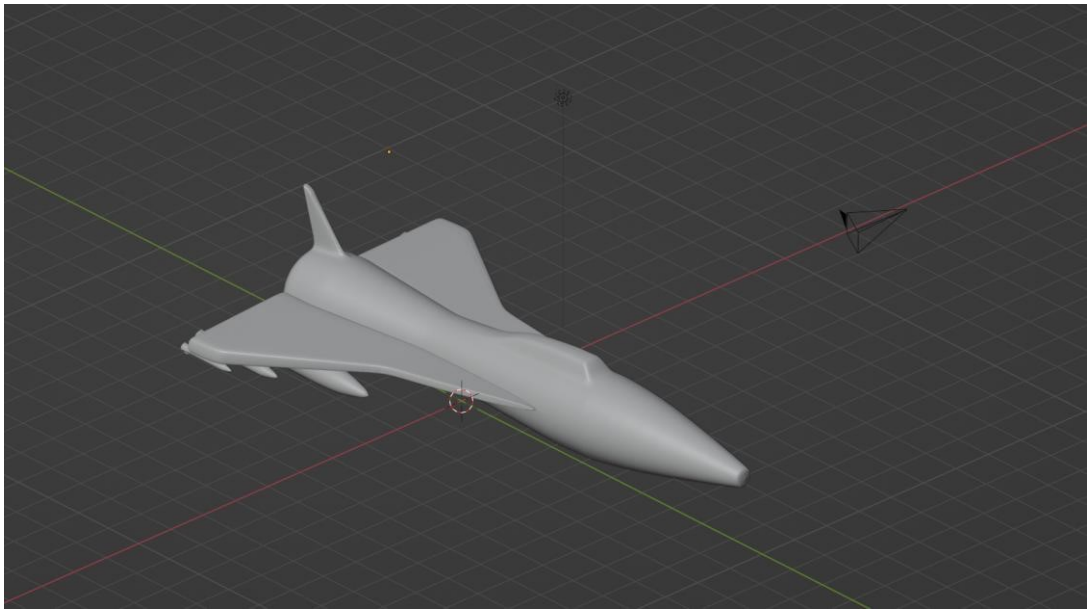

Figure 49: F16 fighting falcon in blender

To make it look romantic, I have picked a red texture with a few scratches on this fighting falcon, as shown in Figure 50.

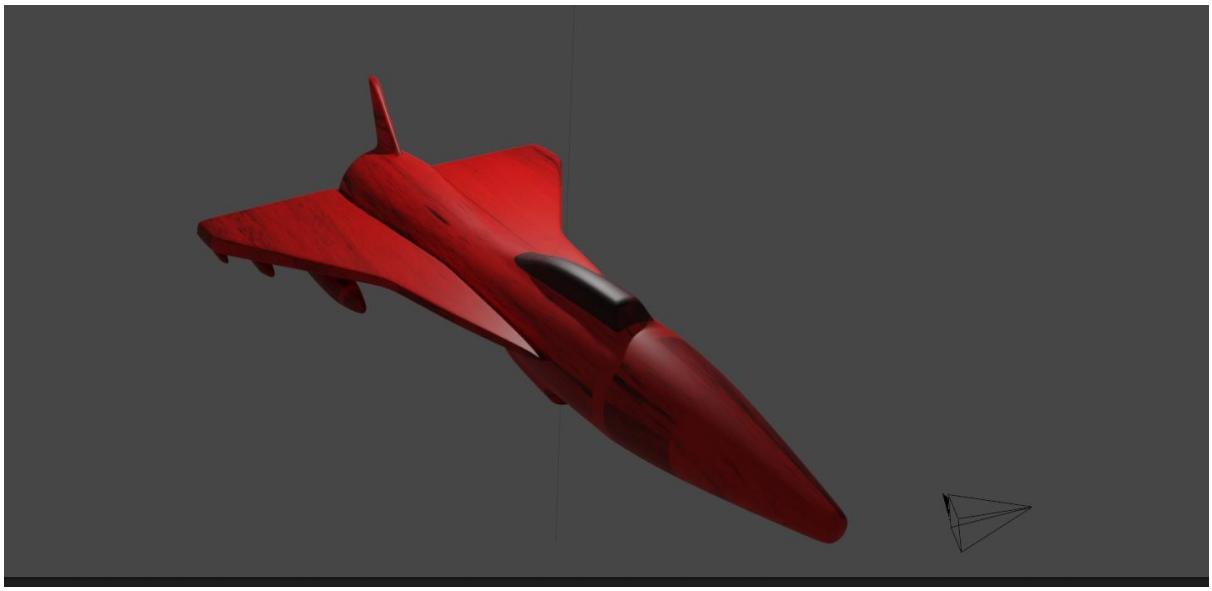

Figure 50: Red f16 fight falcon

2. Planets in solar system

The planet model in this game is simple, where the Earth, Sun, Mars, and Venus do not have rings and simply a default UV sphere model in the blender, as shown in Figure 51. Jupiter, Uranus, and Neptune need an extra model to present their rings.

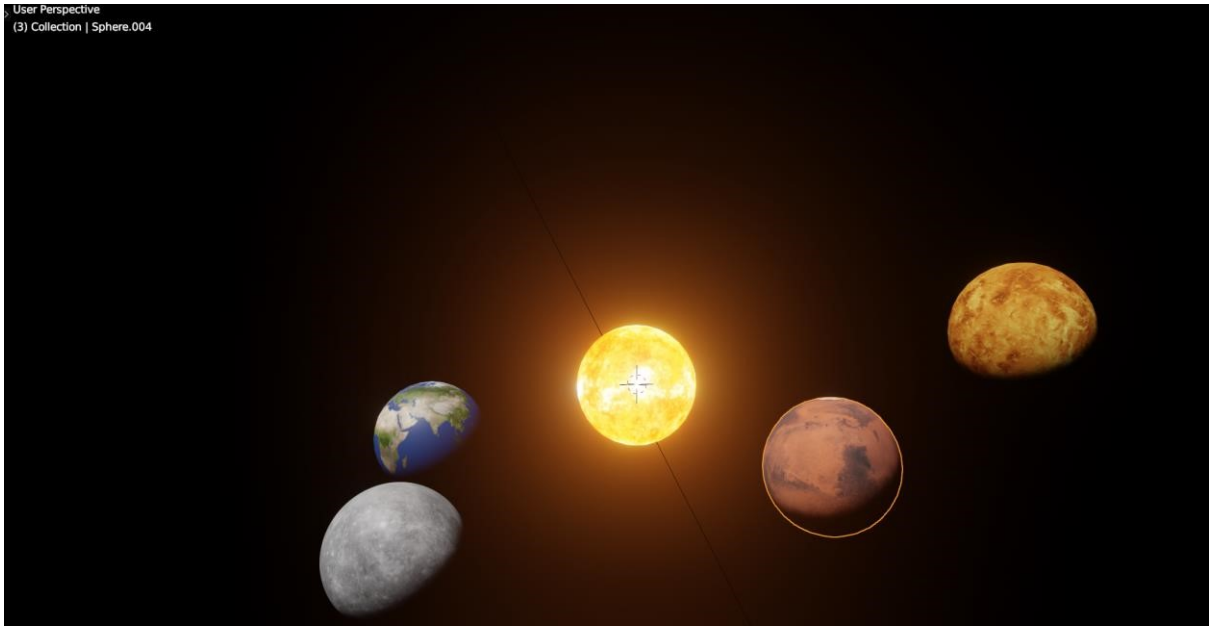

Figure 51: Planets without ring in blender

3. Magical lamp

The magical lamp uses magic to hold the sphere model of each planet and is transparent in the game with implemented attenuation lighting. As it will be transparent and with attenuation lighting, therefore the texture for the magical lamp will not be applied, as shown in Figure 52.

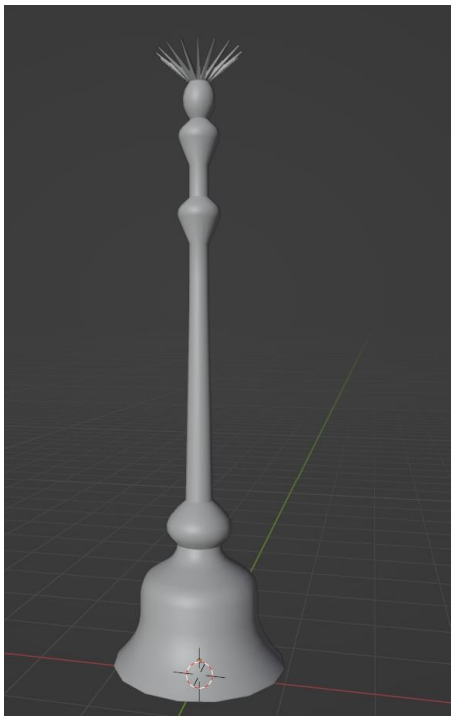

Figure 52: Magical lamp without texture

Result

The development of this game shows how the planets in solar systems differ from each other in terms of velocity or size, and it is also fun to play with developed scenes. The development of games using OpenGL demonstrates the use of 3D objects, shaders, texture, skybox, lighting, and how matrix transformations work.

All the codes used for this development are available on my GitHub repository. The video demonstration is available at the hyperlinks below:

Completed game demo.mp4

Particle effect demo.mp4

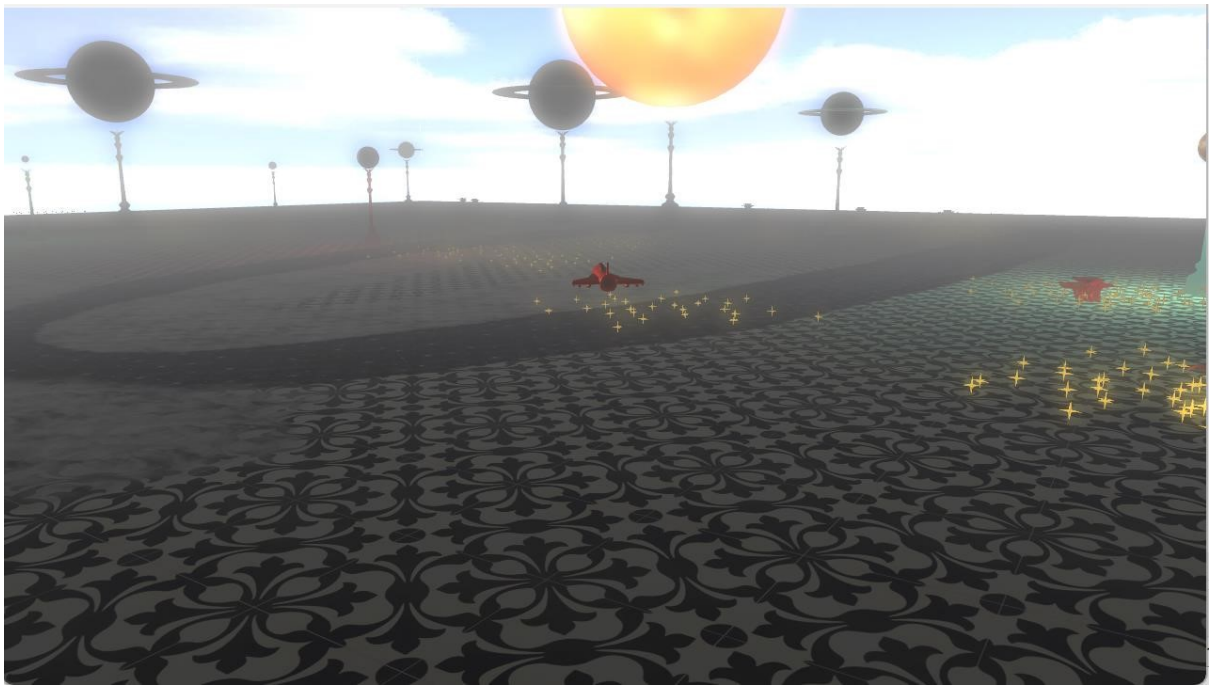

Figure 44: Different size of each planet in game world

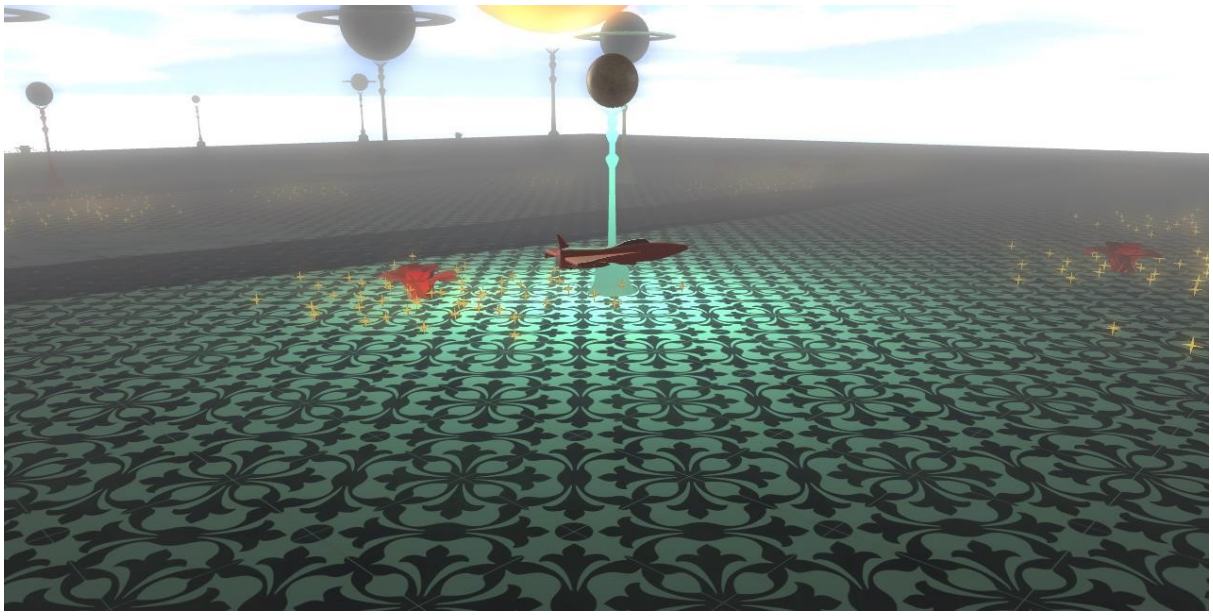

Figure 45: planet will have different attenuation lighting

Self-evaluation

I believe I had achieved all the marking criteria and deserved a 7 based on the marking criteria following:

- This project is well-written and concise, covering all my project progression and the techniques I used.
- I have used text and image examples, which support each other well.
- I have demonstrated the techniques I used with modeling, texture, and rendering results.
- I used all appropriate techniques to achieve this project which I mostly learned from and related to this course.
- I have used a large variety of techniques: lighting, camera, shader program, skybox, and attenuation lighting.
- The techniques chosen were carefully considered and compared with alternatives demonstrating insight into the design decision process. For example, I have implemented the particle effect with heart texture that can show better visuals by adding the element of romance and more interaction for the game.
- I have used independent research extending beyond the specific course-taught techniques such as blend map, mipmap, and OpenGL index buffering.
- The final product looks visually appealing and complete, so I have made all textures for the model, created all the necessary models, and implemented everything for this game world.

References

[1] ahbejarano (2023). *3D Game Development with LWJGL 3*. [online] GitHub. Available at: https://github.com/lwjglgamedev/lwjglbook-bookcontents [Accessed 29 May 2023].

[2] Learnopengl.com. (2019). *Learn OpenGL, extensive tutorial resource for learning Modern OpenGL*. [online] Available at: https://learnopengl.com/ [Accessed 29 May 2023].

[3] www.youtube.com. (n.d.). *ThinMatrix - YouTube*. [online] Available at: https://www.youtube.com/@ThinMatrix [Accessed 29 May 2023].

[4] www.youtube.com. (n.d.). *GetIntoGameDev - YouTube*. [online] Available at: https://www.youtube.com/@GetIntoGameDev [Accessed 29 May 2023].