Army Outpost

HALMAI ERIK 30432

Contents

1. Subject specification

OpenGL is an Application Programming Interface (API) that is used to develop graphical applications by accessing features available in the graphics hardware. The OpenGL specification describes what is the desired result or output of each function. It operates as a large state machine, consisting of a collection of variables that define how it should operate. By writing small programs, called shaders, we can use the GPUs at their full capacity. These shaders are written in OpenGL Shading Language (GLSL) and are attached to the OpenGL application and ran on the GPU.

My project uses OpenGL with GLM, GLEW and GLFW libraries to represent a realistic scene. It also uses multiple fragment and vertex shaders to appropriately illuminate and draw the scene's objects.

2. Scenario

The scenario is an army outpost in the forest, up on the mountains, far from any civilization and hidden from enemies. It is isolated place which can be fully explored by the user.

2.1. Scene and object description

The scene is simple and static, but it offers a look into the isolated lifestyle of the three soldiers, that live day by day in a dangerous environment, risking their lives to server their country. One of the soldiers is present in the scene, greeting the player with his trusty friend, Rex, the dog. He is outside, resting his arm on his tank, as he is right now on duty, to survey the area. The player can be interpreted as one of the soldiers living in the outpost. The limited resources are shown, most of the objects represent the war equipment of the soldiers. They only have the things needed to survive and fight back the enemies.

The scene shows the tanks, one for each soldier, that are textured with different camos. The soldiers live inside of a barrack, textured with a tile like photo, where they can get some sleep after a hard day of fighting. The soldier on duty also has his gun close, supported on the barrack's wall, in case anything goes wrong. He is also holding a lamp to have more visibility in the foggy area. A barricade wall is also present, that can be used as cover by the soldiers. Everything is surrounded by a dense forest, keeping the out of the sight of enemies. The objects have realistic textures, being illuminated by the light sources and mapping shadows onto the ground and surrounding objects.

2.2. Functionalities

- Look around using the camera: the player can use his/het mouse to freely look around the scene.
- Movement of the camera: the player can move through the scene using the keyboard (W + A + S + D)
- Viewing in GL_POINT, GL_LINE and GL_FILL polygonal modes: switch between the modes using the buttons Z, X and C
- Collision detection: the player can't be moved inside of the first tank (the one next to which the soldier stands) and the barrack.
- Movement of the dog: Rex is animated to move back and forth next to the barrack

3. Implementation details

3.1. Functions and special algorithms

There are multiple algorithms for animations, camera rotation and movement, lighting computation, shadow computation, fog computation and collision detection.

3.1.1. Animations

Rex, the dog, is the object animated in the scene. The animation algorithm is simple. The model matrix is modified and sent to the main shader. The model is modified using transition operations until the dog reaches a set z coordinate. After that it is rotated 180 degrees and it goes back to its original position. The process is repeated endlessly. To know when to go forward or backwards, bool values are used that are set when the dog reaches the set z coordinate.

3.1.2. Camera rotation and movement

The camera rotation and movement takes place in the Camera class which was given and updated to implement these.

For the movement, there are 4 possible directions, forward, backwards, left or right, each controlled with the W, A, S or D keys. The movement direction is sent to the Camera class, along with the camera speed, which is a set value and the camera position is updated using the

cameraFrontDirection or the cameraRightDirection variables. The camera target is also updated, which will be cameraPosition + cameraFrontDirection.

For the rotation, the pitch and yaw values are calculated using the current and the last positions of the mouse cursor. A sensitivity constant is also set and used to calculate these. After this, the pitch and yaw values are sent to the Camera class, where the cameraFrontDirection is calculated using these values. Also, the cameraRightDirection will be the normalized cross product of cameraFrontDirection and a general up direction (0.0f, 1.0f, 0.0f) and the cameraUpDirection will be the normalized cross product of the cameraRightDirection and the cameraFrontDirection. The cameraTarget is also updated because the cameraFrontDirection changed.

3.1.3. Lighting computation

For lighting, I used the Blinn-Phong model over the Gouraud model, as it improves performance by avoiding the expensive computation of the reflection vector and it also improves the specular reflections in certain conditions. This algorithm uses the ambient, diffuse and specular light components which are all calculated inside of the main (basic) fragment shader. These calculate the light provided by the directional light (the sun).

The scene also has a point light, which have a set position and an attenuation over some distance, so only the objects near the light will be affected by it. The formula used to calculate this attenuation is the following:

$$Att = \frac{1.0}{K_c + K_l * d + K_q * d^2}$$

In the formula, K_c is a constant, usually 1.0, K_l is the linear term, K_q is the quadratic term and d is the distance between the point light and the fragment's position.

3.1.4. Shadow computation

The shadow computation is done in the basic fragment shader, inside of a function returning a float number that is than combined with the ambient, diffuse and specular values to form the color.

Each object of the scene is first rendered from the light's point of view, using the depthMapShader, getting the depth values. These values are stored in the shadow map which is passed to the basic fragment shader as a uniform. After this the scene is rendered using the basic shader and the function discussed above.

3.1.5. Fog computation

There are three methods of computing the fog: linear fog, exponential fog and square exponential fog. Out of these three, the square exponential fog is used in the project, as it gives the best results. The formula being:

$$fogFactor = e^{-(fragmentDistance * fogDensity)^2}$$

The fog computation also happens in the basic fragment shader and it works by applying a fog color to the objects. The fog color value depends on the distance between the object and the camera.

3.1.6. Collision detection

The collision is implemented for one of the tanks and the barrack of the soldiers. It is done inside of the function that processes movement.

Each time a move button is pressed (W, A, S or D), the coordinates of the camera's current position are checked and if the camera is "inside" one of the two objects, the camera is moved with the reverse direction of the initial move. The reverse direction move is only performed if the reverse direction of the initial move isn't equal to the current direction of the move. I did this to make it less likely to get stuck when the camera is right next to one of the collision objects.

3.2. Graphics model

OpenGL uses shader to use the GPUs at their maximum capacity and render realistic scenes. For this project, four vertex and four fragment shaders were used. All these shaders fulfill different tasks, which are:

- Basic fragment & vertex shader: They draw the objects and apply the shadow and fog attributes to calculate the object's color.
- Depth map fragment & vertex shader: Renders the objects from the light's point of view for shadow calculation.
- Light cube fragment & vertex shader: Draws a light cube around the light source to make it visible.
- Skybox fragment & vertex shader: Draws the skybox.

Each object has its .obj and .mtl file along with an image having its texture. In the .mtl file the texture is mapped onto the object. To texture most of the objects, I used blender along with texture images from the internet. The objects were downloaded from Free3d.

3.3. Data structures

Most of the used data structures are glm structures. These hold the parameters of the objects and the light. GLuInt structures are also used to get the shader uniforms' locations. Besides this, the Model3D, Mesh, Camera, Skybox, Shader and Window classes are used. These will be discussed in more detail below.

3.4. Class hierarchy

The following are all the classes used in the project. Some of them were given and some were changed during the implementation.

- Camera class: move and rotate the camera according to the actions of the user, who uses the keyboard and the mouse.

- Model3D & Mesh classes: offers the functions to easily draw the object on the screen. The Model3D reads the object along with its .obj and .mtl files and the Mesh class draws it.
- Skybox class: loads the faces of the skybox and draws it using a given shader.
- Window class: sets up the OpenGL window and holds all its internal parameters.
- Main class: the class that renders the scene and combines the functionalities of all the classes.

Most of the classes are part of the gps namespace.

4. Graphical user interface presentation / user manual

The user interface is very simple. Only the rendered scene is on the screen and the mouse cursor is disabled, so that it won't get out of the window during mouse movement.

As for the user manual, the possible actions are the following:

- Player movement: using the W + A + S + D keys.
- Camera rotation: using mouse movement.
- Viewing in GL_POINT, GL_LINE and GL_FILL polygonal modes: use the P key for GL_FILL, the Q key for GL_LINE and the I key for GL_POINT.

The application can be launched from Visual Studio, by selecting the Solution Configuration to be Release and setting the Solution Platform to be x64.

5. Conclusions and further developments

It was a challenging and interesting project to make. I found it fascinating to work with OpenGL, as I heard about it many times when playing PC games, but I never really understood what it does or how it works. I created many simple games while I was in school, but most of the functionalities offered by OpenGL were already implemented in the game's engine, so I never got to work on the lower level stuff. This project gave me the opportunity to do so.

As for future development, I would make this into a more interactive sandbox game, with quests. The player would be one of the soldiers fighting from this outpost and their mission would be to, first of all, survive, by hunting or by infiltrating the nearby enemy bases and stealing their supplies. I would also add an inventory system that would be displayed on the screen when a button is pressed and I would add collision detection on the remaining objects and animate them (make the tanks usable, etc).

6. References

- Free3d.com
- Learnopengl.com
- Blinn-Phong lighting
- Gouraud lighting