GRAPHIC PROCESSING

PROJECT DOCUMENTATION

Name: Mureșan Salomeea

Group: 30434

TABLE OF CONTENTS

SUBJECT SPECIFICATION

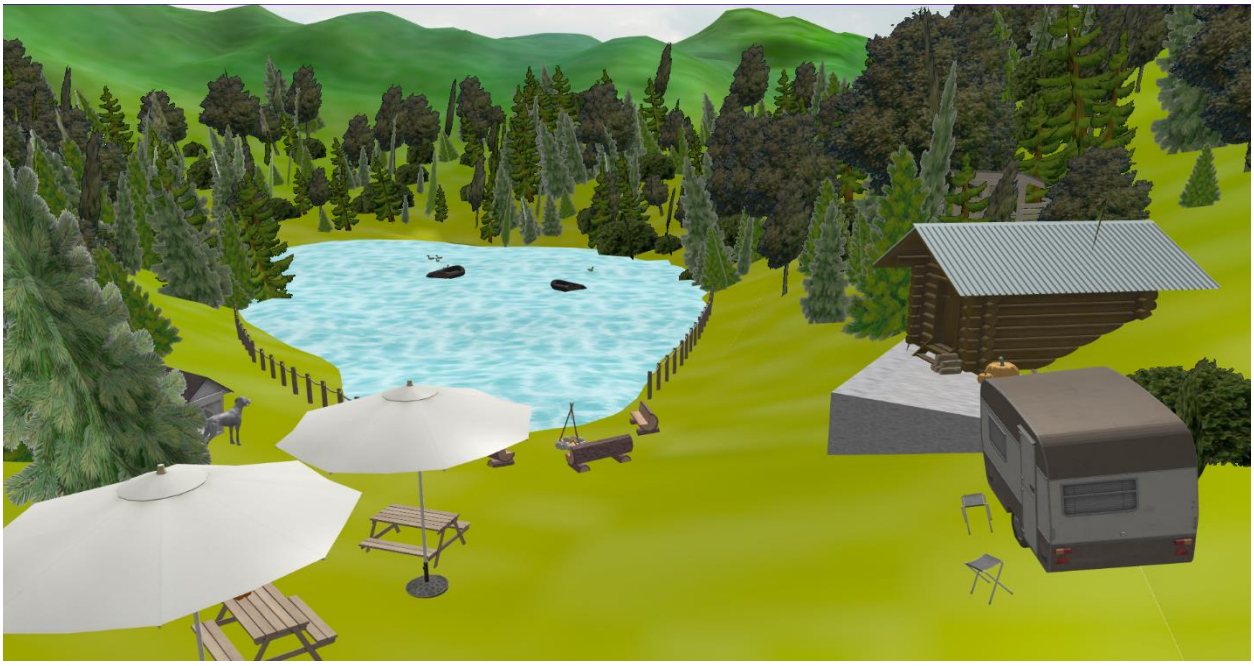
The aim of this project is to create a photorealistic presentation of 3D objects using OpenGL library. OpenGL is considered an Application Programming Interface (API) and is used to develop graphical applications by accessing features available in the graphics hardware. However, OpenGL is a specification developed and maintained by the Khronos Group. The OpenGL specification describes what is the desired result or output of each function. The actual implementation can be different between various OpenGL libraries. These libraries are implemented mainly by the graphics card manufacturers.

The theme of the project is a remote picnic and camping place in the mountains. The scene is surrounded by beautiful mountains and trees. It presents some picnic tables, a cabin and a camper, placed between 2 lakes.

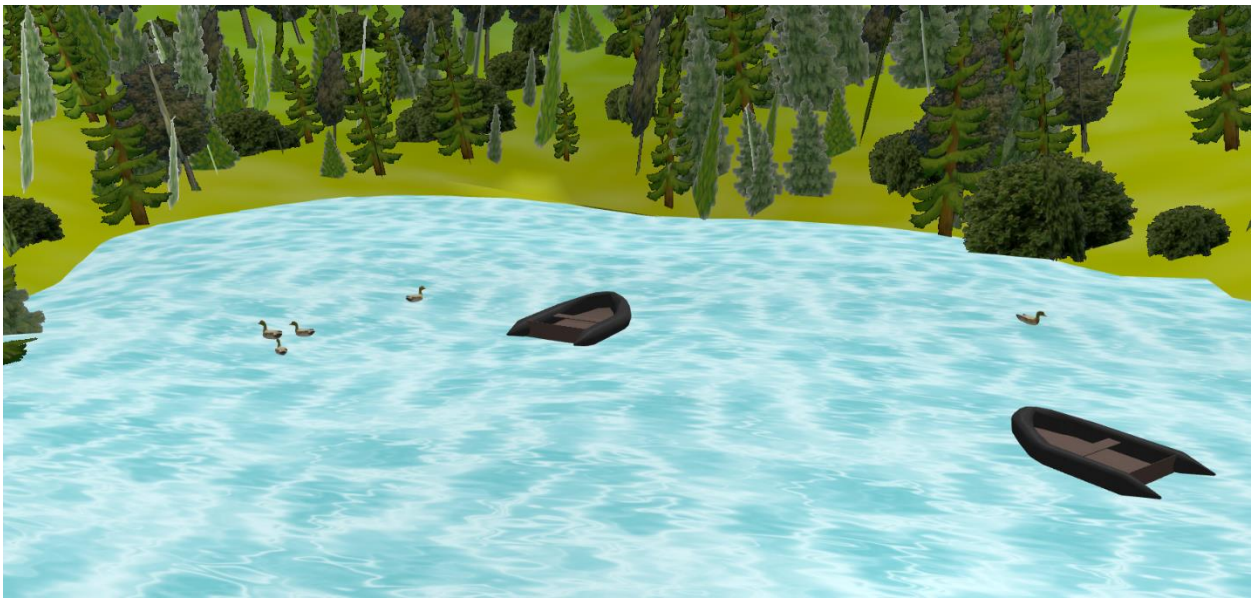
SCENARIO

SCENE AND OBJECTS DESCRIPTION

The scene presents a place in the mountains, near the high peak, where the user who is also the protagonist can take a stroll to relax and enjoy nature. The atmosphere is calming, as it should be when taking a break from hiking. The user can look at the ducks swimming on the lake, can play with the dog or can have a picnic or a lunch break on the many picnic tables available at the site. There is also a bathroom for hikers and a cabin for those in need of shelter when it gets too dark to hike anymore.



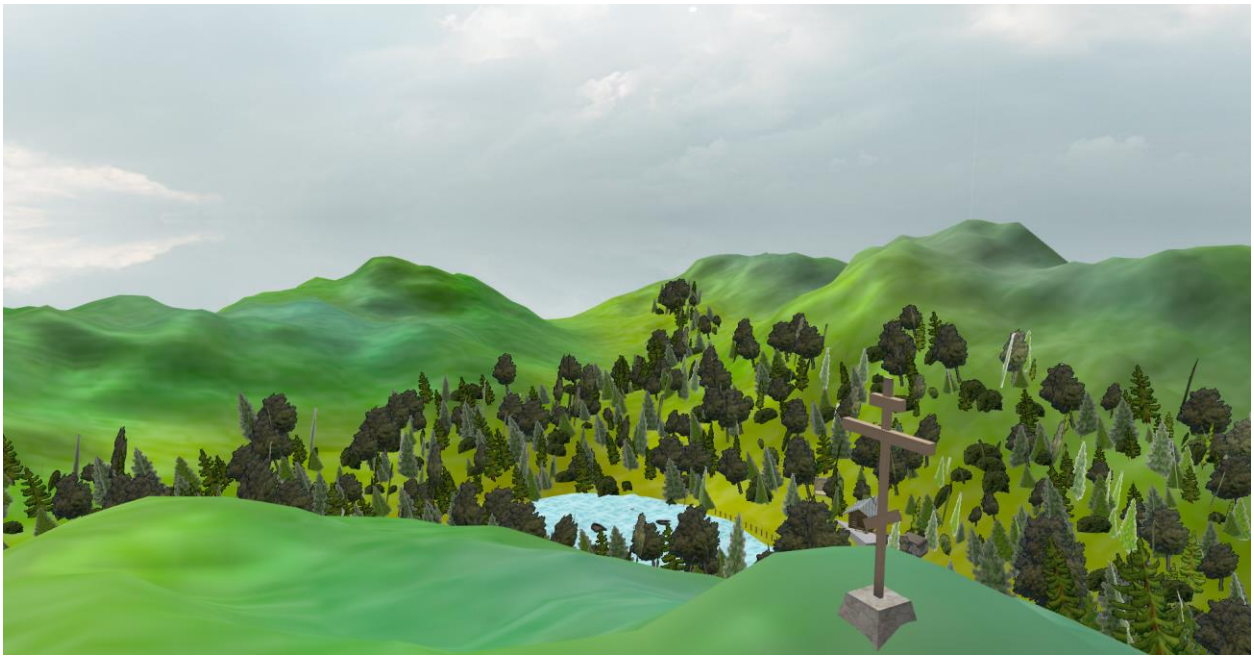
The ducks love to swim all day on the clear mountain lake, the same lake on which people can take a boat ride on the two boats available at the cabin.

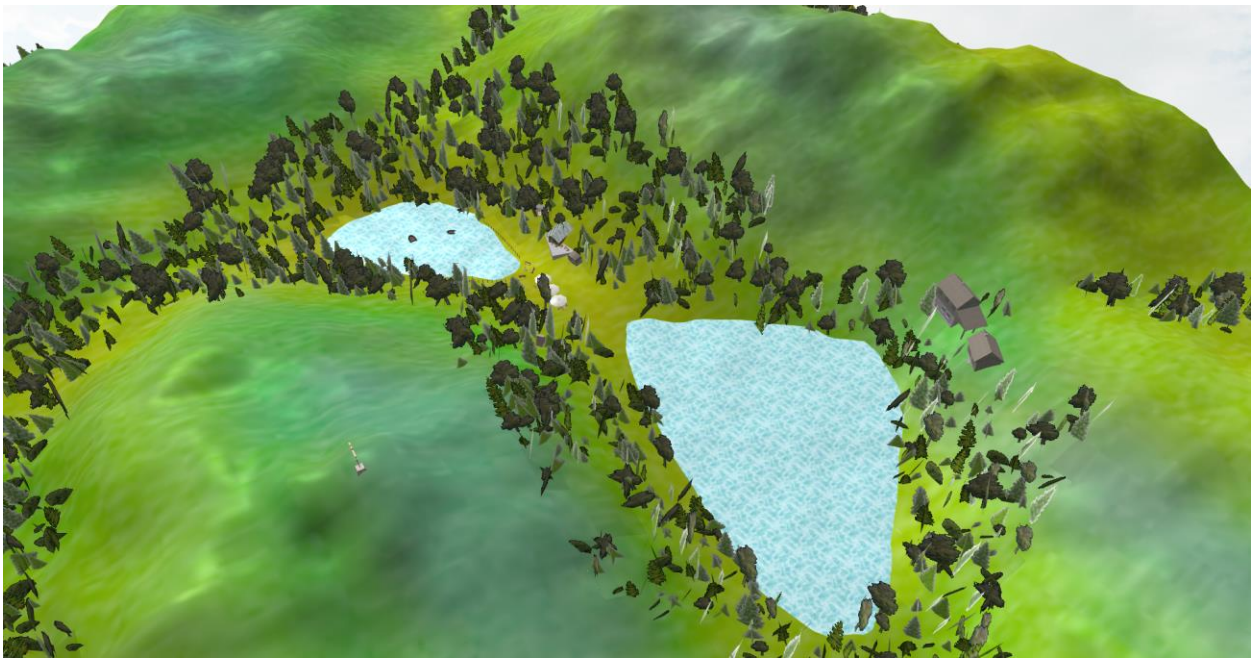


When it gets cold and dark, you can start a campfire right next to the lake and sit on the wooden benches sculpted by the owner of the cabin, with the company of the friendly dog Fluffy, which is always happy to see new mountain lovers at the site (that's why his tail is always wagging).

Oh! It looks like you are not alone here, someone already set up their camper before you got here. I'm sure everyone here is friendly anyway, so you don't have to worry about bad neighbors.

You can also enjoy a hike on the highest peak and take in the beauty of nature and its breathtaking views.





FUNCTIONALITIES

By using the keyboard and mouse, the user can view and walk through the entire scene. They can also toggle between different display modes, such as solid view and smooth/polygonal faces. The camera position can be changed, as well as the camera target. Additionally, the fog density can be changed.

IMPLEMENTATION DETAILS

FUNCTIONS AND SPECIAL ALGORITHMS

One of the most important function that is used in this project is renderScene(). This function is responsible for setting up and drawing the objects of the scene. First, the function clears the color and depth buffers of the framebuffer. This function is called in a loop, until the glfw window is closed.

The function then calls other renderObject functions, such as renderDuck(), renderWater() and so on. The model matrix is used to animate and place the present objects at the wanted location in the scene.

The framebuffer is then unbound, and the code sets the current shader program to “myBasicShader”. The “lightSpaceTrMatrix” uniform is set again, this time to the result of the computeLightSpaceTrMatrix() function. The “view” matrix uniform is also set to the result of the getViewMatrix() method of the “myCamera” object.

Another important technique used is shadow mapping (not working). The first thing was to set up the light’s perspective and render the scene from the light’s point of view of the shadow map texture. This was done by creating a special framebuffer with the shadow map texture attached as a depth attachment. Then, the scene is rendered with the light’s perspective matrix and the depth values are stored in the shadow map texture. In the final rendering pass, the shadow map texture is bound and the fragment’s position in light space is transformed and used to look up the closest depth value in the shadow map. This value is then compared to the fragment’s actual depth to determine whether the fragment is in shadow or not. A bias value is added to account for possible inaccuracies in the shadow map.

For achieving the fog effect, I used the exponential computation: where the density of the fog increases exponentially with the distance from the camera. This was implemented in the fragment shader.

I also used fragment discarding for the trees, because normal 3d modeled trees had too many vertexes for my computer to handle. In the fragment shader, I checked every color from texture if the alpha value is lower than 0.1f, and if so, that specific fragment is discarded.

THE MOTIVATION OF THE CHOSEN APPROACH

All of those approaches are presented carefully in the Graphics Processing laboratory, so it was easier for me to implement them in such a way. I know there are also a lot of other approaches when speaking of shadow computation, fog generation etc., but it was a good practice to redo parts of the laboratory from scratch.

GRAPHICS MODEL

Objects and textures have been downloaded from the internet and then manipulated using the Blender tool, to create the final scene. All 3D objects were exported as .obj from Blender and then loaded in OpenGL.

The trees present in the scene were created by joining two perpendicular planes and adding a .png texture with transparent background.

DATA STRUCTURES

Data structures used in this project are the ones provided during the graphics processing laboratory.

CLASS HIERARCHY

- Camera.cpp
    - C++ implementation of a camera class for use in a graphics program. It contains a constructor, methods for moving and rotating the camera, and methods for printing the current camera position and target.
    - The constructor takes three arguments - the initial camera position, target position, and up direction. It calculates the camera front, right, and up directions using the initial positions.
    - The getViewMatrix() method returns the view matrix of the camera using the glm::lookAt() function, which takes the camera position, target position, and up direction as arguments.
    - The move() method updates the camera position based on the input direction and speed, which can be one of six possible values - MOVE_FORWARD, MOVE_BACKWARD, MOVE_LEFT, MOVE_RIGHT, MOVE_UP, or MOVE_DOWN. The camera's target position is also updated accordingly.
    - The rotate() method updates the camera position and target position based on input pitch and yaw values, which represent the camera's rotation around the x and y axes. It then recalculates the camera's front, right, and up directions.
    - The printPosition() and printTarget() methods print the camera's current position and target position, respectively.
    - Finally, the moveTo() method allows the user to manually set the camera's position and target position to a new position and target position.
- Mesh.cpp
    - Mesh class in C++ using OpenGL. It defines a constructor that takes in three vectors of Vertex, GLuint, and Texture types, respectively. It also defines a drawing function Draw() that applies textures to the mesh and draws it using OpenGL. The setupMesh() function initializes the buffer objects and arrays for the mesh.
    - The Mesh class also defines a getter function getBuffers() that returns a Buffers struct that contains the VAO (Vertex Array Object), VBO (Vertex Buffer Object), and EBO (Element Buffer Object) of the mesh.
    - This implementation provides a basic framework for rendering a mesh using OpenGL, with the ability to apply textures to the mesh.
- Model3D.cpp
    - Contains code for loading and drawing a 3D model using the OpenGL graphics library. The code is written in C++ and consists of several functions.
    - The LoadModel function is overloaded and takes a file name and a base path. This function reads an OBJ file and calls the ReadOBJ function, which does the parsing of the file and fills the data structure.
    - The ReadOBJ function uses the tinyobj library to parse the OBJ file and extract the vertex, normal, and texture coordinate data, as well as the material information. It then creates a Vertex struct for each vertex and stores it in a vector, along with the corresponding indices for each triangle. The material information is also stored in a Material struct.

    - The Draw function draws the 3D model using the Shader class object passed to it as an argument. It does this by iterating over each mesh in the model and calling the Draw function of the mesh object.
- Main.cpp
- Shader.cpp
    - This is used to load, compile, and link shader programs in OpenGL. The implementation file contains the definitions for the member functions of the class.
    - The first function readShaderFile is a private function that takes a file name as input, reads the contents of the file, and returns the contents as a string.
    - The next two functions shaderCompileLog and shaderLinkLog are private functions that take a shader ID or a shader program ID as input, respectively. They check the compilation or linking status of the shader or shader program and print an error message if the compilation or linking fails.
    - The loadShader function is a public function that takes two file names as input, one for the vertex shader and one for the fragment shader. It reads the contents of the files using readShaderFile, compiles the shaders, links them into a shader program, and stores the ID of the shader program in the class member variable shaderProgram.
    - The useShaderProgram function is a public function that simply sets the current shader program to the one stored in the shaderProgram member variable using glUseProgram.
- SkyBox.cpp
    - This class loads a skybox texture and renders it as the background of the scene.:
    - The Load() function receives a vector of six file paths to the texture faces, and it loads them into a cubemap texture using the LoadSkyBoxTextures() function.
    - The InitSkyBox() function sets up the vertex array object and the vertex buffer object used to render the skybox. It defines the vertices of a cube centered at the origin, which will be rendered using the cubemap texture in the Draw() function.
    - The Draw() function renders the skybox. It sets the view and projection matrices and enables the GL_LEQUAL depth function to render the skybox behind everything else. It then binds the vertex array object and the cubemap texture and renders the skybox using glDrawArrays(). Finally, it disables the GL_LEQUAL depth function and unbinds the vertex array object.

GRAPHICAL USER INTERFACE PRESENTATION / USER MANUAL

The function processMovement() is responsible for handling the user's input and updating the camera's position and view accordingly.

When a key is pressed, the function checks which key is pressed and performs the corresponding action.

If the user presses the **W, A, S, or D** keys, the camera moves forward, left, backward, or right, respectively. If the user presses the **Q or E** keys, the camera rotates to the left or right, respectively. If the user presses the Up or Down keys, the camera moves up or down, respectively. If the user presses the **1, 2, or 3** keys, the function changes the polygon mode to wireframe, point, or fill mode, respectively. If the user presses the **left or right arrow** keys, the function increases or decreases the density of fog, respectively. If the user presses the **Space** key, the function prints the camera's current

position and target. Finally, if the user presses the **Z** key, the function sets the animation flag to true, indicating that the camera animation should start.

The function updates the view matrix and normal matrix every time the camera's position or orientation changes. The view matrix is used to transform the world coordinates to the camera's view space, and the normal matrix is used to transform the normals of the vertices to the camera's view space for lighting computations.

CONCLUSIONS AND FURTHER DEVELOPMENTS

A further development for this project can be successfully implementing the shadows and light sources, also rendering rain, snow and adding more objects in the scene.

In conclusion, this was the hardest project that I have ever worked on. I really like the subject, but this project required a lot of hard work and attention to detail. My favorite part was setting the scene in blender and modeling the mountains and objects. With a lot of sleepless nights and many mental breakdowns I managed to do a mediocre project in OpenGL.

REFERENCES

https://moodle.cs.utcluj.ro/course/view.php?id=526

https://www.youtube.com/playlist?list=PLrgcDEgRZ_kndoWmRkAK4Y7ToJdOf-OSM

https://www.blender.org/

https://learnopengl.com/

https://rigmodels.com/

https://polyhaven.com/