Graphic Processing project

Student: Bradea Maria Mihaela
group 30434, january 2024

Contents

Contents **1**

1 Subject Specification **2**

2 Scenario **2**
2.1 Scene and objects description 2
2.2 Functionalities 3

3 Implementation details **4**
3.1 Functions and special algorithms 4
3.1.1 Possible solutions 4
3.1.2 The motivation of the chosen approach 6
3.2 Graphics model 6
3.3 Data structures 6
3.4 Class hierarchy 6

4 Graphical user interface presentation / user manual **6**

5 Conclusions and further developments **6**

6 References **6**

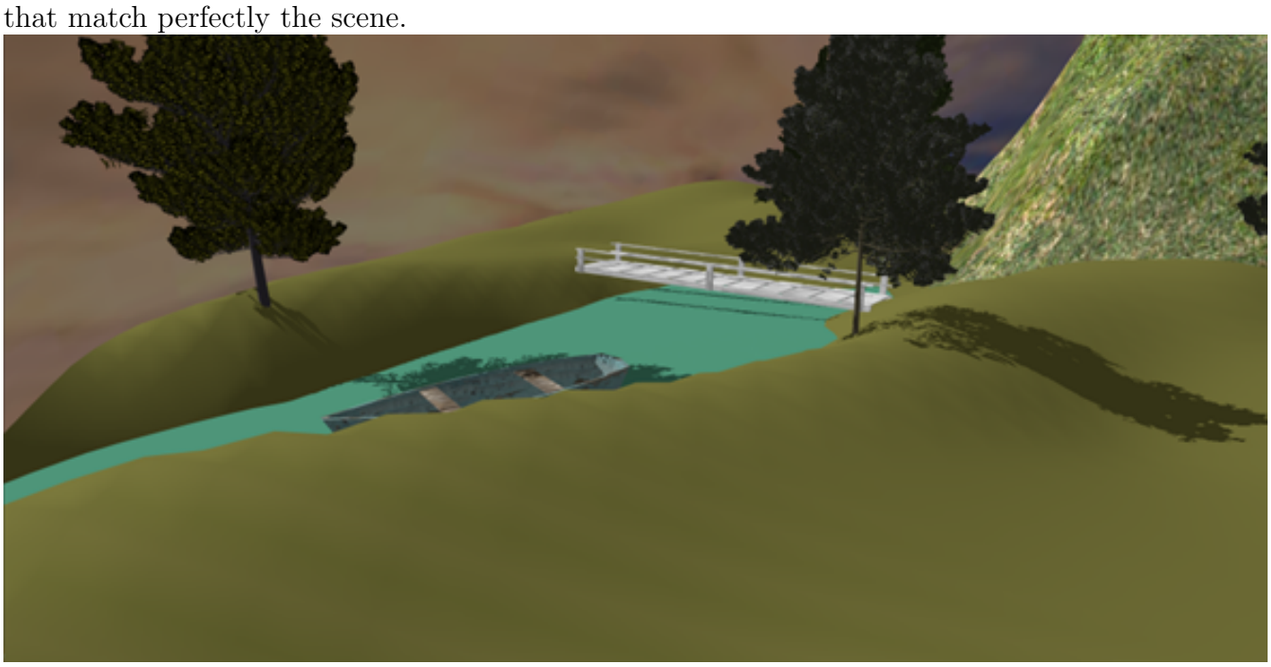
1 Subject Specification

The project's purpose is creating an implementation in OpenGl using the GLFW and GLEW libraries for rendering a 3D scene. The program creates a window, initializes shaders and handles user input for camera movement. The scene includes a ground plane and an object representing a hot air balloon, that are rendered using shaders for basic and depth mapping effects. The depth map is utilized for shadow rendering. The camera can be controlled with keyboard inputs: A, S, D, W, Q, E, and mouse movements handle camera rotation. Also, the scene also includes two animations: first one, in which the camera will move constantly as velocity and in forward direction, move that can be slowed or stopped and also reversed depending on how much the user presses the button S, and the second one in which, the hot air baloon will move up and down, without stopping, as it would do in an infinite loop. The code structure includes separate classes for the window, shaders, camera, and 3D models. Additionally, error checking functions are incorporated for OpenGL calls. The application utilizes time-based animation to simulate camera movement and provides a dynamic rendering environment.

2 Scenario

2.1 Scene and objects description

The main scene in the project is ground1.obj, that ilustrates which illustrates a wonderful landscape around a mountain with a beautiful view and a multicolored sky. This object is actually a collection of object that were put together in Blender and then exported as a single one. It also contains different species of animals that live in such an environment: deers and a lynx. As decorations, there are multiple types of trees of different heights and shapes, having as texture 3 different materials. Another objects are constitued by an old cabin, a wonderful blue lake with a boat and an old bridge. This scene contains another object, hotbaloon.obj that represent, as the name indicates, an hot air baloon that will move contstantly in the scene. It has bright, pretty colors that match perfectly the scene.



2.2 Functionalities

1. **Camera control:** Allows users to navigate the scene using the keyboard and mouse. W, A, S, D keys control forward, left, backward, and right movements, respectively. Q and E keys enable rotation around the vertical axis and mouse movements contribute to camera

orientation.

2. **Time-based animations:** Provides dynamic animation effects for both the camera and the hot air balloon object. Simulates an up-and-down motion for the hot air balloon, creating a visually appealing and dynamic scene. Enables a time-based camera movement to simulate forward motion, allowing users to experience continuous exploration within the environment.

3. **Window resize:** Includes error-checking functions to identify and report OpenGL errors. Enhances user experience by providing feedback on potential issues.

4. **Realistic lighting and shadows:** Incorporates realistic lighting effects with a specified light direction and color. Implements shadow mapping to create realistic shadows that dynamically respond to changes in the light source's position.

3 Implementation details

3.1 Functions and special algorithms

3.1.1 Possible solutions

several functions that serve specialized purposes, contributing to the overall functionality and visual appeal of the OpenGL-based 3D scene. Here are descriptions of some of these special functions:

- **computeLightSpaceTrMatrix:** This function calculates the light space transformation matrix used in shadow mapping. As algorithm, it combines a view matrix, representing the perspective from the light source, with an orthographic projection matrix. The resulting matrix is crucial for transforming vertices into the light space coordinates when rendering the shadow map.

- **initFBO:** This function initializes the Framebuffer Object (FBO) for off-screen rendering during the creation of the depth map for shadow mapping. As algorithm, it creates a depth texture attachment for the FBO, specifying parameters such as filtering and wrapping. The FBO is then configured to render only the depth information, optimizing the shadow mapping process.

- **initUniforms:** This function initializes shader uniforms for the main scene and depth mapping shaders. As algorithm, it sets up various uniform variables in the shaders, including model, view, and projection matrices, as well as light direction and color. This function ensures that the shaders have the necessary information to correctly render the scene and shadows.

```
glm::mat4 computeLightSpaceTrMatrix() {
    glm::mat4 lightView = glm::lookAt(lightDir, glm::vec3(0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
    const GLfloat near_plane = -600, far_plane = 20.0f;
    glm::mat4 lightProjection = glm::ortho(-50.0f, 50.0f, -50.0f, 50.0f, near_plane, far_plane);
    glm::mat4 lightSpaceTrMatrix = lightProjection * lightView;
    return lightSpaceTrMatrix;
}
```

```
void initUniforms() {
    glGenFramebuffers(1, &shadowMapFBO);
    glGenTextures(1, &depthMapTexture);
    glBindTexture(GL_TEXTURE_2D, depthMapTexture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
        SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    float borderColor[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
    glBindFramebuffer(GL_FRAMEBUFFER, shadowMapFBO);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMapTexture,
        0);
    glDrawBuffer(GL_NONE);
    glReadBuffer(GL_NONE);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    for (int i = 0; i <= 0; i++) {
        shader[i].useShaderProgram();

        // create model matrix for teapot
        model = glm::rotate(glm::mat4(1.0f), glm::radians(angle), glm::vec3(0.0f, 1.0f, 0.0f));
        modelLoc[i] = glGetUniformLocation(shader[i].shaderProgram, "model");
        // get view matrix for current camera
        view = myCamera.getViewMatrix();
        viewLoc[i] = glGetUniformLocation(shader[i].shaderProgram, "view");
        // send view matrix to shader
        glUniformMatrix4fv(viewLoc[i], 1, GL_FALSE, glm::value_ptr(view));
        // compute normal matrix for teapot
        normalMatrix = glm::mat3(glm::inverseTranspose(view * model));
        normalMatrixLoc[i] = glGetUniformLocation(shader[i].shaderProgram, "normalMatrix");
        // create projection matrix
        projection = glm::perspective(glm::radians(45.0f),
                              (float)myWindow.getWindowDimensions().width / (float)myWindow.getWindowDimensions().height,
                              0.1f, 100.0f);

        projectionLoc[i] = glGetUniformLocation(shader[i].shaderProgram, "projection");

        // send projection matrix to shader
```

```
// create projection matrix
projection = glm::perspective(glm::radians(45.0f),
                      (float)myWindow.getWindowDimensions().width / (float)myWindow.getWindowDimensions().height,
                      0.1f, 100.0f);

projectionLoc[i] = glGetUniformLocation(shader[i].shaderProgram, "projection");

// send projection matrix to shader
glUniformMatrix4fv(projectionLoc[i], 1, GL_FALSE, glm::value_ptr(projection));

//set the light direction (direction towards the light)
lightDir = glm::vec3(0.0f, 1.0f, 1.0f);
lightDirLoc[i] = glGetUniformLocation(shader[i].shaderProgram, "lightDir");

// send light dir to shader
glUniform3fv(lightDirLoc[i], 1, glm::value_ptr(lightDir));

//set light color
lightColor = glm::vec3(1.0f, 1.0f, 1.0f); //white light
lightColorLoc[i] = glGetUniformLocation(shader[i].shaderProgram, "lightColor");
// send light color to shader
glUniform3fv(lightColorLoc[i], 1, glm::value_ptr(lightColor));
```

```
void initFBO() {
    glGenFramebuffers(1, &shadowMapFBO);
    glGenTextures(1, &depthMapTexture);
    glBindTexture(GL_TEXTURE_2D, depthMapTexture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
        SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
    float borderColor[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColor);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
    glBindFramebuffer(GL_FRAMEBUFFER, shadowMapFBO);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMapTexture,
        0);
    glDrawBuffer(GL_NONE);
    glReadBuffer(GL_NONE);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

3.1.2 The motivation of the chosen approach

The chosen approach in the code prioritizes modularity and object-oriented design, evident in classes like Shader, Model3D, and Camera. Abstraction of shaders and 3D models enhances code organization and flexibility. Time-based animation and sophisticated use of FBOs for shadow mapping contribute to a visually engaging and efficient 3D scene. Specialized functions, such as glCheckError, aid debugging. Overall, the approach emphasizes clarity, maintainability, and user interaction for an effective OpenGL implementation.

3.2 Graphics model

A common graphics model involves the use of three-dimensional (3D) models, which are composed of vertices, edges, and faces. These models are often defined using mathematical representations like polygons or parametric surfaces: vertices, edges, faces, textures, shaders, transformations, lighting, renderings.

3.3 Data structures

The most important data structures include: matrices, vectors, uniform buffers, vertex/ index buffer object, framebuffer object and shader storage buffer object.

3.4 Class hierarchy

The following classes are included in the project:

- **Camera:** Manages camera functionality.
- **Main:** serves as the entry point for the program. It encompasses the application loop, initializes the OpenGL window, sets up rendering states, loads models and shaders, and orchestrates the rendering process.
- **Mesh:** Handles 3D meshes.
- **Shader:** Handles shaders.
- **Model3D:** Handles 3D model loading and rendering.

4 Graphical user interface presentation / user manual

The graphical user interface is simple, as it can be seen above in the photos, and the usage is even more simple. The user can control the camera by the keyboard, namely A, S, D, Q, W, and E and by the movement of the mouse.

5 Conclusions and further developments

With this project I discovered how many things that seemed very complicated, have a logical basis and can be implemented quite easily using mathematical transformations and functions. I also discovered how beautiful it is to work when you have a visual basis. I hope that in the future I can create a wider universe and offer a greater functionality of the objects.

6 References

- https://moodle.cs.utcluj.ro/my/
- https://www.youtube.com/playlist?list=PLrgcDEgRZ$_k ndoWmRkAK4Y7ToJdOf - OSM$
- https://free3d.com/3d-models/
- https://www.blender.org/