# *Graphic Processing Project*

*Student: Violeta Maria Hoza*

*Group: 30434*

*Technical University of Cluj-Napoca*

## Contents

## *1. Subject specification*

The purpose of the project was to create an application which renders a realistic 3D scene using the OpenGL API, and to allow the user to interact with it, using the keyboard input or the mouse.

This project implements a 3D rural landscape visualization using OpenGL. The scene features a dynamic environment with day/night cycles, animated objects, and various visual effects. The project demonstrates graphics programming concepts including lighting, shadow mapping, fragment discarding, weather effects, and camera controls.

## *2. Scenario*

### 2.1 Scene and object description

The scene was built in Blender 3.6 by modelling the terrain and by importing and placing objects individually. Each object was textured-mapped for realism. The scene and animated objects were then loaded into OpenGL. In addition, a skybox was added for a more realistic view.

**Figure 1: Screenshot of the scene depicted in the application**

The scene features a peaceful countryside environment containing multiple components:

- a central landscape with rolling hills and a lake

- a house and a barn
- dense forest areas with individually rendered trees (with fragment discarding)
- an animated hot air balloon floating through the scene
- multiple light sources including lamps and environmental lighting
- a dynamic skybox that changes between day and night
- atmospheric effects like fog and rain
- decorative elements (fences, paths, vegetation).

The scene assets were separated into static and dynamic objects for efficiency. Static objects (such as terrain and buildings) were included in a single .obj file, while dynamic objects and objects requiring special effects (such as the trees and the air balloon) were kept in separate .obj files.

### 2.2 Functionalities

The project features several interactive functionalities that allow the user to engage with the scene:

- free-roaming camera control with keyboard movement and mouse look
- day/night cycle toggle with different lighting conditions
- dynamic shadow rendering
- fog effect with adjustable density
- rain effect
- multiple viewing modes (solid, wireframe, point)
- automated camera tour of the scene
- animated objects (hot air balloon)
- 4 point light sources of different colours that can be toggled

## *3. Implementation details*

The demo application provided in the laboratory was used as the basis for this project. Project-specific functionalities were added to the existing foundation to obtain the final result.

Some functions that play a critical role in this project are:

- renderScene(): responsible for rendering all the objects in the scene
- processMovement(): handles keyboard input for camera movement and interaction with the scene
- initModels(): loads all the 3D models into the application
- initShaders(): loads the shaders required for rendering objects and effects
- initUniforms(): initializes and sets the locations for shader variables, such as light positions and camera parameters.

## Camera navigation

The user controls directly the camera via the mouse and the keyboard. In order to be able to control the camera's movement, the methods provided in the Camera class were implemented so that the camera is able to move in all directions: UP, DOWN, LEFT, RIGHT, FORWARD, BACKWARD. The rotation of the camera was also implemented, enabling full freedom of movement within the scene.

```
// Update the camera internal parameters following a camera move event
void Camera::move(MOVE_DIRECTION direction, float speed) {
    // Calculate the front and right direction vectors
    this->cameraFrontDirection = glm::normalize(cameraTarget - cameraPosition);
    this->cameraRightDirection = glm::normalize(glm::cross(this->cameraUpDirection, this->cameraFrontDirection));

    // Update the camera position based on the direction and speed
    switch (direction) {
    case MOVE_FORWARD:
        cameraPosition += cameraFrontDirection * speed;
        break;
    case MOVE_BACKWARD:
        cameraPosition -= cameraFrontDirection * speed;
        break;
    case MOVE_RIGHT:
        cameraPosition += cameraRightDirection * speed;
        break;
    case MOVE_LEFT:
        cameraPosition -= cameraRightDirection * speed;
        break;
    case MOVE_UP:
        cameraPosition += cameraUpDirection * speed;
        break;
    case MOVE_DOWN:
        cameraPosition -= cameraUpDirection * speed;
        break;
    }

    // Update the camera target position
    cameraTarget = cameraPosition + cameraFrontDirection;
}

// Update the camera internal parameters following a camera rotate event
// yaw - camera rotation around the y axis
// pitch - camera rotation around the x axis
void Camera::rotate(float pitch, float yaw) {
    // Create a rotation matrix and apply yaw and pitch rotations
    glm::mat4 matrix = glm::mat4(1.0f);
    matrix = glm::rotate(matrix, glm::radians(yaw), glm::vec3(0.0f, 1.0f, 0.0f));
    matrix = glm::rotate(matrix, glm::radians(pitch), glm::normalize(glm::cross(cameraUpDirection, cameraTarget - cameraPosition)));
    // Update the camera target position based on the rotation
    cameraTarget = cameraPosition + glm::vec3(matrix * glm::vec4(cameraTarget - cameraPosition, 1.0f));
}
```

## Lighting model

In this project there are five sources of light. One of them is a global, directional light source which comes from somewhere up in the sky, and the other are light sources are point lights of different colours coming from a fire near the forest, a lamp inside the barn and the streetlamps in front of the house.

The global illumination in this project was implemented in the shaders, using the fixed-pipeline lighting (the Gouraud model), which combines various independent lighting components to achieve the overall illumination effect of a spot on an object.

The components used are:

- Ambient light – a diffuse, non-directional light that approximates scattered light throughout the scene.

- Diffuse light – light that scatters evenly in all directions from the light source. This component models how light intensity varies based on the angle between the light source and the surface normal.
- Specular highlighting – the light reflected directly from a surface which mimics the reflective properties of the materials. This component depends on the surface normal, the light's direction and the viewing direction.

```
// Function to compute directional light contribution
void computeDirLight()
{
    //vec3 cameraPosEye = vec3(0.0f);
    //compute eye space coordinates
    //vec4 fPosEye = view * model * vec4(fPosition, 1.0f);
    vec3 normalEye = normalize(normalMatrix * fNormal);

    //normalize light direction
    //vec3 lightDirN = vec3(normalize(view * vec4(lightDir, 0.0f)));
    vec3 lightDirN = normalize(lightDir);

    //compute view direction (in eye coordinates, the viewer is situated at the origin
    vec3 viewDir = normalize(- fPosEye.xyz);

    //compute half vector
    vec3 halfVector = normalize(lightDirN + viewDir);

    //compute ambient light
    ambient = ambientStrength * lightColor;
    ambient *= texture(diffuseTexture, fTexCoords);

    //compute diffuse light
    diffuse = max(dot(normalEye, lightDirN), 0.0f) * lightColor;
    diffuse *= texture(diffuseTexture, fTexCoords);

    //compute specular light
    float specCoeff = pow(max(dot(normalEye, halfVector), 0.0f), 32);
    specular = specularStrength * specCoeff * lightColor;
    specular *= texture(specularTexture, fTexCoords);
}
```

For the point light sources, I employed the Blinn-Phong lighting model. his method calculates the distance from each light position to the processed fragment, applying attenuation to balance the light intensity based on distance. The components of ambient, diffuse, and specular lighting are combined for realistic illumination.

```
// Function to compute point light contribution
void computePointLight(vec3 pointLightPos, vec3 pointLightColor)
{
    float constant = 1.0f;
    float linear = 0.35f;
    float quadratic = 0.44f;

    vec3 normalEye = normalize(normalMatrix * fNormal);


    vec3 lightDir2 = normalize(pointLightPos - fPosition);

    vec3 halfVector = normalize(lightDir2 + lightDir2);
    float specCoeff = pow(max(dot(normalEye, halfVector), 0.0f), 32);

    float distance2 = length(pointLightPos - fPosition.xyz);
    float attenuation2 = 1.0 / (constant + linear * distance2 + quadratic * (distance2 * distance2));

    ambient += ambientStrength * pointLightColor * attenuation2;

    diffuse += max(dot(normalEye, lightDir2), 0.0f) * pointLightColor * attenuation2;
    specular += specularStrength * pow(max(dot(normalEye, reflect(-lightDir2, normalEye)), 0.0f), 32) * pointLightColor * attenuation2;
}
```

## Shadow generation

Shadows in the scene are generated using shadow mapping, which involves rendering the scene from the light's perspective. This technique helps determine which parts of the scene are in shadow by comparing the fragment's depth with the light's depth map. To avoid visual artifacts, such as shadow acne, additional techniques were implemented for depth comparison and over-sampling rectification.

In the fragment shader, I transformed coordinates from object space to light space, checked if the point lies outside the camera frustum, and calculated both the depth of the light's nearest surface and the current fragment's depth. Additionally, I addressed the issue of shadow acne to improve the visual accuracy of the shadows, and I rectified the over sampling.

```
// Function to compute shadow contribution
float computeShadow()
{
    // perform perspective divide
    vec3 normalizedCoords = fragPosLightSpace.xyz / fragPosLightSpace.w;
    // Transform to [0,1] range
    normalizedCoords = normalizedCoords * 0.5 + 0.5;
    if (normalizedCoords.z > 1.0f)
        return 0.0f;
    // Get closest depth value from light's perspective
    float closestDepth = texture(shadowMap, normalizedCoords.xy).r;
    // Get depth of current fragment from light's perspective
    float currentDepth = normalizedCoords.z;
    // Check whether current frag pos is in shadow
    float bias = 0.005f;
    float shadow = currentDepth - bias > closestDepth ? 1.0f : 0.0f;
    return shadow;
}
```

An alternative approach, such as point shadows or projected shadows, could have been used, but shadow mapping provides a more efficient and flexible solution for this type of scene.

## Fog generation

The fog generation was done by blending the colour of the fog with the fragment colour based on the distance of the fragment in the fragment shader. To determine the fog coefficient, I implemented the square exponential formula:

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

In the shaders, I calculated the object's position and its relative distance from the viewing camera. By applying these values in the formula, I obtained the fog factor which was ten used to determine the fragment colour through interpolation.

```
// Function to compute fog contribution
float computeFog() {
    vec4 fPosEye = view * model * vec4(fPosition, 1.0f);
    float fragDist  = length(fPosEye.xyz);
    float factor = exp(-pow(fragDist * fogDensity, 2));
    return clamp(factor, 0.0f, 1.0f);
}
```

This effect was designed to simulate the appearance of fog, with its intensity controlled by user input via keyboard keys. The fog effect can be toggled on/off and its density adjusted in real-time.

**Rain effect**

The rain effect in the scene is implemented using a collection of raindrop objects, each represented by a Rain structure which holds the position and velocity of each raindrop. The logic involves initializing the raindrops (initializing a number of raindrops with random positions and a constant velocity), updating their positions (by applying a downward velocity), and rendering them in the scene.

**Scene tour**

The scene tour functionality provides an automated navigation through predefined waypoints, creating a dynamic walkthrough of the 3D scene. Each waypoint is represented by a Waypoint structure, which holds both the camera's position and the target point it should focus on. The sequence of waypoints is defined in a std::vector called tourPath, providing a guided path with specific positions and orientations that highlight various parts of the scene. This allows the camera to smoothly traverse the scene, showcasing various perspectives and highlighting key features of the environment.

The main loop of the application checks if the inTour flag is enabled, indicating that the tour mode is active. During the tour, the camera is reset to avoid any manual positioning conflicts, and an interpolation function computes the camera's current position between two waypoints using a linear blend controlled by the parameter t. The direction of the camera is updated by normalizing the vector between the next target and the interpolated position. The interpolation parameter t increments gradually, controlling the transition speed for smooth movement. When the transition between two waypoints is complete (t >= 1.0f), the camera moves to the next segment of the tour until all waypoints are visited, at which point the tour concludes by setting inTour to false. This approach provides a dynamic and visually engaging camera path while ensuring efficient control over movement and scene framing.

```cpp
// Define tour waypoints for camera movement
struct Waypoint {
    glm::vec3 position;
    glm::vec3 target;
};

// Tour path definition with waypoints
std::vector<Waypoint> tourPath = {
    {{27.0f, 8.0f, -3.5f}, {0.0f, 2.0f, 0.0f}},
    {{20.0f, 7.0f, 5.0f}, {-2.0f, 2.0f, 0.0f}},
    {{15.0f, 6.0f, 10.0f}, {-3.0f, 2.0f, -2.0f}},
    {{5.0f, 5.5f, 8.0f}, {-8.0f, 2.0f, -2.0f}},
    {{-5.0f, 5.5f, 5.0f}, {-10.0f, 2.0f, -3.0f}},
    {{-10.0f, 6.0f, 0.0f}, {-5.0f, 2.0f, 0.0f}},
    {{-5.0f, 6.5f, -10.0f}, {0.0f, 2.0f, 0.0f}},
    {{5.0f, 7.0f, -8.0f}, {0.0f, 1.0f, 2.0f}},
    {{15.0f, 7.5f, -5.0f}, {0.0f, 2.0f, 0.0f}},
    {{27.0f, 8.0f, -3.5f}, {0.0f, 2.0f, 0.0f}}
};

glm::vec3 interpolate(glm::vec3 start, glm::vec3 end, float t) {
    return start * (1.0f - t) + end * t;
}


// Check if the tour flag is set to true
if (inTour) {
    resetCamera();

    static int currentWaypoint = 0;
    static float t = 0.0f;

    if (currentWaypoint < tourPath.size() - 1) {
        glm::vec3 newPosition = interpolate(tourPath[currentWaypoint].position, tourPath[currentWaypoint + 1].position, t);
        glm::vec3 newTarget = tourPath[currentWaypoint + 1].target;  // Focus on next point
        myCamera.setCameraPosition(newPosition);
        myCamera.setCameraDirection(glm::normalize(newTarget - newPosition));

        t += 0.01f;  // Adjust speed for smooth transition
        if (t >= 1.0f) {
            t = 0.0f;
            currentWaypoint++;
        }
    }
    else {
        //currentWaypoint = 0;  // Restart the tour after completing
        inTour = false;
    }
}
```

## Data structures

The main data structures leveraged in this application are the ones defined in the OpenGL libraries, namely the vectors, matrices, and the data structures for the uniform variable's locations (GLuint, GLint, GLfloat, GLboolean). Additionally, data structures such as Model3D and Shader were employed, serving as essential components for organizing and manipulating 3D entities within the project. These structures facilitated efficient handling of objects, textures, and shaders in the scene. Furthermore, standard C++ data types like bool and float were utilized to manage various aspects of the program's logic.

## Class hierarchy

The project is structured around the classes provided in the laboratory works, containing classes that are used for modeling the objects, the scene, for camera manipulation and the shaders. All these classes are imported in the main class and combined together to create the scene representation.

## *4. Graphical user interface presentation / user manual*

Visualization modes:

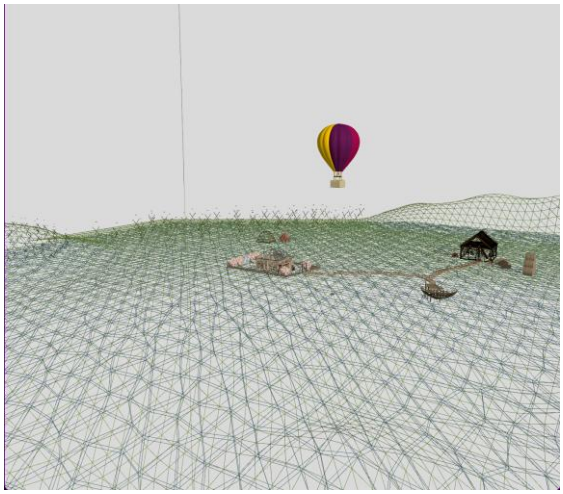

**Figure 2: Wireframe visualization**

**Figure 3: Point visualization**

**Figure 4: With fog**

**Figure 5: Just with point lights**

The user can interact with the scene using the following keys on the keyboard:

- W: move camera forward
- A: move camera left
- S: move camera backward
- D: move camera right
- E:  move camera up
- R: move the camera down
- Up/Down: rotate camera up/down
- Left/Right: rotate camera left/right
- Mouse: Look around
- TAB: Toggle mouse capture
- N: Toggle day/night cycle

- F: Toggle fog effect
- L: Toggle point lights
- 1: switch to wireframe view mode
- 2: switch to point view mode
- 3: switch to solid view mode
- 0: start/stop automated tour
- M: Toggle shadow map visualization
- P: toggle rain
- 5/6: rotate directional light
- 7: increase fog density
- 8: decrease fog density
- C: print camera position

## *5. Conclusions and further developments*

The project involved learning to operate in Blender and OpenGL, thus developing skills in this regard. The result is a relatively complex scene, with a few animations and functionalities implemented with graphical processing algorithms learned from the GP laboratory over the semester or from individual research.

Some further developments would be point shadows, effects such as wind, snow, lightning, thunder, moving boats, dynamic vegetation movement, additional elements in the scene and new animations.

## *6. References*


- Texture for the skybox: https://www.humus.name/
- Free 3D models can be downloaded from: https://free3d.com/, https://www.cgtrader.com/free-3d-models, https://www.turbosquid.com/
- https://learnopengl.com/
- GP laboratory works