

Computer Graphics and Interactive Systems

Authors:

NIKOS LALOUTSOS

ANNA TARASIDOU

1. Project Description:

(i): Create a window with dimensions 950x950

- To implement this part of the assignment, we modified the command:

```
window = glfwCreateWindow(950, 950, u8"Άσκηση 1Γ 2024 - Κυνήγι  
θυσσαυρού", NULL, NULL);
```

so that the window dimensions are 950x950 and the assignment title appears correctly.

We add the u8 prefix before the Greek characters, so they are printed correctly as the window title.

-To ensure the window has a black background, we verify the RGB values with the following command:

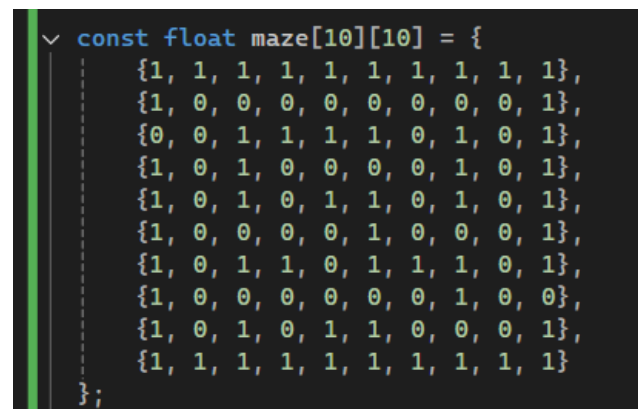
```
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
```

-Finally, to set the window to close when the SPACE key is pressed, we use :

```
while (glfwGetKey(window, GLFW_KEY_SPACE) != GLFW_PRESS &&  
glfwWindowShouldClose(window) == 0);
```

(ii): Initializing the maze

-We create a matrix with 1s and 0s, where the 1s represent the walls of the maze.



```
const float maze[10][10] = {  
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1},  
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},  
    {0, 0, 1, 1, 1, 1, 0, 1, 0, 1},  
    {1, 0, 1, 0, 0, 0, 0, 1, 0, 1},  
    {1, 0, 1, 0, 1, 1, 0, 1, 0, 1},  
    {1, 0, 0, 0, 0, 1, 0, 0, 0, 1},  
    {1, 0, 1, 1, 0, 1, 1, 1, 0, 1},  
    {1, 0, 0, 0, 0, 0, 0, 1, 0, 0},  
    {1, 0, 1, 0, 1, 1, 0, 0, 0, 1},  
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}  
};
```

-The maze is oriented vertically on the x,y plane, so the z-coordinate is constant, equal to 0, to position it at the base of the plane.

-To draw the walls of the maze from this matrix, we use a double for loop as follows:

```
for (int i = 0; i < 10; ++i) {  
    for (int j = 0; j < 10; ++j) {  
        if (maze[i][j] == 1) {  
            float x = j - 5.0f;  
            float y = 5.0f - i;  
            float z = 0.0f;
```

```

        GLfloat wall_vertices[] = {
            // Down side
            x, y, z, x + 1.0f, y, z, x, y, z + 1.0f,
            x + 1.0f, y, z, x + 1.0f, y, z + 1.0f, x, y, z + 1.0f,
            // Up side
            x, y + 1.0f, z, x + 1.0f, y + 1.0f, z, x, y + 1.0f, z + 1.0f,
            x + 1.0f, y + 1.0f, z, x + 1.0f, y + 1.0f, z + 1.0f, x, y +
            1.0f, z + 1.0f,
            // Frond side
            x, y, z + 1.0f, x + 1.0f, y, z + 1.0f, x, y + 1.0f, z + 1.0f,
            x + 1.0f, y, z + 1.0f, x + 1.0f, y + 1.0f, z + 1.0f, x, y +
            1.0f, z + 1.0f,
            // Back side
            x, y, z, x + 1.0f, y, z, x, y + 1.0f, z,
            x + 1.0f, y, z, x + 1.0f, y + 1.0f, z, x, y + 1.0f, z,
            // Left side
            x, y, z, x, y + 1.0f, z, x, y, z + 1.0f,
            x, y + 1.0f, z, x, y + 1.0f, z + 1.0f, x, y, z + 1.0f,
            // Right side
            x + 1.0f, y, z, x + 1.0f, y, z + 1.0f, x + 1.0f, y + 1.0f, z,
            x + 1.0f, y, z + 1.0f, x + 1.0f, y + 1.0f, z + 1.0f, x + 1.0f, y
            + 1.0f, z
        };
    }
}
}

```

Inside this loop, we iterate through the maze. To position the maze correctly in 3D space, we first calculate the starting coordinates for drawing by setting $x = j - 5$, $y = 5 - i$, and $z = 0$. For each cell in the maze that is equal to 1, a cube is constructed from 6 faces, meaning 12 triangles in total. That corresponds to 36 vertices. We create a unit cube (volume 1) and determine the coordinates of these 36 points based on each face of the cube.

To allow the maze to be colored, we use the following approach:
We define two vectors—one to store the colors of the cube edges and another for the colors of the maze walls.

```

// Vertices and colors
std::vector<GLfloat> allWallVertices;
std::vector<GLfloat> allWallColors;

```

και μέσα στη for loop που σχεδιάζουμε τον λαβύρινθο προσθέτουμε:

```

// Add vertices
allWallVertices.insert(allWallVertices.end(), std::begin(wall_vertices),
std::end(wall_vertices));

for (int v = 0; v < 36; ++v) {
    allWallColors.push_back(0.0f);
    allWallColors.push_back(0.0f);
    allWallColors.push_back(1.0f);
}

```

And upon exiting the loop, we create the corresponding buffers as follows:

```

GLuint vertexbuffer;
glGenBuffers(1, &vertexbuffer);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glBufferData(GL_ARRAY_BUFFER, allWallVertices.size() *
sizeof(GLfloat), allWallVertices.data(), GL_STATIC_DRAW);

GLuint colorbuffer;
glGenBuffers(1, &colorbuffer);
glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
glBufferData(GL_ARRAY_BUFFER, allWallColors.size() * sizeof(GLfloat),
allWallColors.data(), GL_STATIC_DRAW);

```

To load the vectors and colors.

For drawing the player, we followed these steps:

We find the appropriate coordinates so that the player is positioned exactly at the center of the cubes at the start of the maze, which are:

```

// Start coordinates
float characterX = -4.50f;
float characterY = 3.5f;
float characterZ = 0.5f;

```

In this way, the small square will be at the starting point of the maze, precisely at the center of the corresponding empty cell.

For the player's color, we store the information in the array static const GLfloat cubeColorData[] as {1, 1, 0}, which corresponds to the yellow color.

```
static const GLfloat cubeColorData[] 1,1,0
```

-In the main function, before proceeding to draw the character, we use the program:

```
glUseProgram(programID1);
```

which, as explained later, contains the color information for the maze and the character from the two shaders: P1CVertex and P1CFragment.

Then, we draw the square in exactly the same way as the maze, making sure its sides have a length of 0.5 instead of 1.

```

GLfloat cubeVertices[] = {
// Back (Z = +0.25)
characterX - 0.25f, characterY - 0.25f, characterZ + 0.25f,
characterX + 0.25f, characterY - 0.25f, characterZ + 0.25f,
characterX - 0.25f, characterY + 0.25f, characterZ + 0.25f,

characterX + 0.25f, characterY - 0.25f, characterZ + 0.25f,
characterX + 0.25f, characterY + 0.25f, characterZ + 0.25f,
characterX - 0.25f, characterY + 0.25f, characterZ + 0.25f,

// Front (Z = -0.25)
characterX - 0.25f, characterY - 0.25f, characterZ - 0.25f,
characterX + 0.25f, characterY - 0.25f, characterZ - 0.25f,
characterX - 0.25f, characterY + 0.25f, characterZ - 0.25f,

```

```
characterX + 0.25f, characterY - 0.25f, characterZ - 0.25f,  
characterX + 0.25f, characterY + 0.25f, characterZ - 0.25f,  
characterX - 0.25f, characterY + 0.25f, characterZ - 0.25f,
```

```
// Left (X = -0.25)
```

```
characterX - 0.25f, characterY - 0.25f, characterZ + 0.25f,  
characterX - 0.25f, characterY + 0.25f, characterZ + 0.25f,  
characterX - 0.25f, characterY - 0.25f, characterZ - 0.25f,
```

```
characterX - 0.25f, characterY + 0.25f, characterZ + 0.25f,  
characterX - 0.25f, characterY + 0.25f, characterZ - 0.25f,  
characterX - 0.25f, characterY - 0.25f, characterZ - 0.25f,
```

```
// Right (X = +0.25)
```

```
characterX + 0.25f, characterY - 0.25f, characterZ + 0.25f,  
characterX + 0.25f, characterY + 0.25f, characterZ + 0.25f,  
characterX + 0.25f, characterY - 0.25f, characterZ - 0.25f,
```

```
characterX + 0.25f, characterY + 0.25f, characterZ + 0.25f,  
characterX + 0.25f, characterY + 0.25f, characterZ - 0.25f,  
characterX + 0.25f, characterY - 0.25f, characterZ - 0.25f,
```

```
// Up (Y = +0.25)
```

```
characterX - 0.25f, characterY + 0.25f, characterZ + 0.25f,  
characterX + 0.25f, characterY + 0.25f, characterZ + 0.25f,  
characterX - 0.25f, characterY + 0.25f, characterZ - 0.25f,
```

```
characterX + 0.25f, characterY + 0.25f, characterZ + 0.25f,  
characterX + 0.25f, characterY + 0.25f, characterZ - 0.25f,  
characterX - 0.25f, characterY + 0.25f, characterZ - 0.25f,
```

```
// Down (Y = -0.25)
```

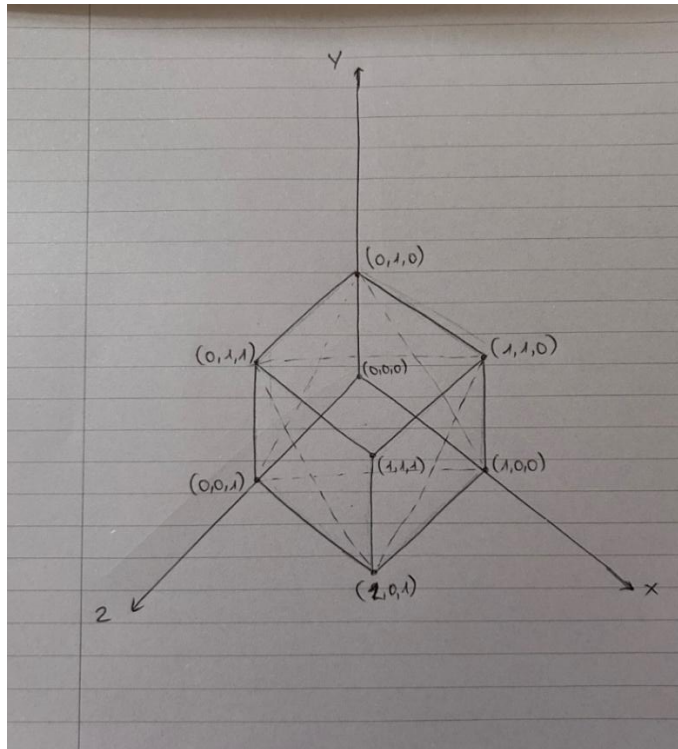
```
characterX - 0.25f, characterY - 0.25f, characterZ + 0.25f,  
characterX + 0.25f, characterY - 0.25f, characterZ + 0.25f,  
characterX - 0.25f, characterY - 0.25f, characterZ - 0.25f,
```

```
characterX + 0.25f, characterY - 0.25f, characterZ + 0.25f,  
characterX + 0.25f, characterY - 0.25f, characterZ - 0.25f,  
characterX - 0.25f, characterY - 0.25f, characterZ - 0.25f
```

Finally, we load the new buffers for the color and our character:

```
GLuint cubeVertexBuffer;  
glGenBuffers(1, &cubeVertexBuffer);  
glBindBuffer(GL_ARRAY_BUFFER, cubeVertexBuffer);  
glBufferData(GL_ARRAY_BUFFER, sizeof(cubeVertices), cubeVertices,  
GL_STATIC_DRAW);
```

```
GLuint cubeColorBuffer;  
glGenBuffers(1, &cubeColorBuffer);  
glBindBuffer(GL_ARRAY_BUFFER, cubeColorBuffer);  
glBufferData(GL_ARRAY_BUFFER, sizeof(cubeColorData), cubeColorData,  
GL_STATIC_DRAW);
```



6 vertices, 2 triangles for each face of the cube, for each of the 6 faces of the cube.

(iii): Creating the treasure

- Initially, to draw the treasure as a cube, we started as follows:
- We initialized its buffer.
- We defined a variable that determines the size of the treasure.
- We set the variables charX, charY, and charZ that specify its coordinates, which we will explain later.

```
GLuint treasureVertexBuffer;
GLfloat trsize = 0.4f; // Define trsize as a GLfloat to represent the size
of the treasure
```

```
float charX = xCoord[start] - 5 + 0.5f;
float charY = 5 - yCoord[start] + 0.5f;
float charZ = 0.5f;
```

-Next, we had to create the texture for the treasure and encountered the following problem:

It was not possible to implement it with the two existing shaders because the colors of the objects forming the maze and the player's cube were affected. The problem was solved when we created two separate shaders whose exclusive function is to apply the texture onto the treasure cube.

-We created a new `programID1` and a new `MatrixID1` to load the two new shaders we developed, which will later be used to apply a texture onto the treasure.

```

GLuint programID = LoadShaders("P1BVertexShader.vertexshader",
"P1BFragmentShader.fragmentshader");
GLuint MatrixID = glGetUniformLocation(programID, "MVP");

GLuint programID1 = LoadShaders("P1CVertexShader.vertexshader",
"P1CFragmentShader.fragmentshader");
GLuint MatrixID1 = glGetUniformLocation(programID1, "MVP");

```

The P1B shaders contain the necessary information for loading the texture:

P1B.VertexShader:

```

#version 330 core

layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;

out vec2 UV;

uniform mat4 MVP;

void main(){

    gl_Position = MVP * vec4(vertexPosition_modelspace,1);
    UV = vertexUV;
}

```

P1B.FragmentShader:

```

#version 330 core
in vec2 UV;

out vec3 color;

uniform sampler2D myTextureSampler;

void main()
{

    color = texture (myTextureSampler, UV).rgb;
}

```

-The P1C shaders contain the necessary information for the colors used in the rest of the program and are the same as those from part 1B.

After resolving the issue with implementing the texture, we proceeded to create the cube representing the treasure.

-We created two arrays, ``xCoord`` and ``yCoord``, whose elements, in pairs, represent the coordinates of the empty spaces in the maze.

-The arrays do not include the start and end points of the maze.

For example, `xCoord[0]` and `yCoord[0]` correspond to the coordinate pair `[x, y] = [1, 1]`, which is an empty spot in the maze.

```
const int xCoord[] = {1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 5, 5, 5, 6, 6, 6, 6, 6, 6, 6, 7, 7, 7, 8, 8, 8, 8, 8, 8, 8, 8};  
const int yCoord[] = {1, 2, 3, 4, 5, 6, 7, 8, 1, 5, 7, 1, 3, 4, 5, 7, 8, 1, 3, 5, 6, 7, 1, 3, 7, 1, 2, 3, 4, 5, 7, 8, 1, 5, 8, 1, 2, 3, 4, 5, 6, 7, 8};
```

-We used the random library to utilize it in a function that will generate random numbers for us.

```
#include <random>  
  
int getRandomNumber() {  
    static std::random_device rd;  
    static std::mt19937 gen(rd());  
    static std::uniform_int_distribution<> distr(0, 42);  
  
    return distr(gen);  
}
```

-This function generates random numbers from 0 up to 42 (i.e., a range of 43 numbers), which will be used as the coordinate pair where the treasure will move.

-We initialize a random number in a variable that will serve as the treasure's initial coordinates.

```
int start = getRandomNumber();
```

-Then, we need to assign the treasure its initial coordinates:

```
float charX = xCoord[start] - 5 + 0.5f;  
float charY = 5 - yCoord[start] + 0.5f;  
float charZ = 0.5f;
```

We do this by setting the three variables charX, charY, and charZ to take coordinates from the arrays xCoord and yCoord using the start index, which we ensured is random and not at the maze's starting position.

To map the coordinates into the maze space, we perform the calculations xCoord - 5 and 5 - yCoord.

To position the treasure exactly at the center of the empty cell, we add 0.5 to both coordinates.

-Before defining the array with the triangle vertices, we insert the command:

```
glUseProgram(programID);
```

So that from this point onward, our program will use the P1B shaders, which are responsible for the treasure's texture.

-Just like in the player's drawing and using the data above, we created an array that defines the vertices of each triangle—for each of the two triangles composing every face of the treasure cube.


```

GLfloat treasureVertices[] = {
    // Back (Z = +trsize)
    charX - trsize, charY - trsize, charZ + trsize,
    charX + trsize, charY - trsize, charZ + trsize,
    charX - trsize, charY + trsize, charZ + trsize,

    charX + trsize, charY - trsize, charZ + trsize,
    charX + trsize, charY + trsize, charZ + trsize,
    charX - trsize, charY + trsize, charZ + trsize,

    // Front (Z = -trsize)
    charX - trsize, charY - trsize, charZ - trsize,
    charX + trsize, charY - trsize, charZ - trsize,
    charX - trsize, charY + trsize, charZ - trsize,

    charX + trsize, charY - trsize, charZ - trsize,
    charX + trsize, charY + trsize, charZ - trsize,
    charX - trsize, charY + trsize, charZ - trsize,

    // Left (X = -trsize)
    charX - trsize, charY - trsize, charZ + trsize,
    charX - trsize, charY + trsize, charZ + trsize,
    charX - trsize, charY - trsize, charZ - trsize,

    charX - trsize, charY + trsize, charZ + trsize,
    charX - trsize, charY + trsize, charZ - trsize,
    charX - trsize, charY - trsize, charZ - trsize,

    // Right (X = +trsize)
    charX + trsize, charY - trsize, charZ + trsize,
    charX + trsize, charY + trsize, charZ + trsize,
    charX + trsize, charY - trsize, charZ - trsize,

    charX + trsize, charY + trsize, charZ + trsize,
    charX + trsize, charY + trsize, charZ - trsize,
    charX + trsize, charY - trsize, charZ - trsize,

    // Up (Y = +trsize)
    charX - trsize, charY + trsize, charZ + trsize,
    charX + trsize, charY + trsize, charZ + trsize,
    charX - trsize, charY + trsize, charZ - trsize,

    charX + trsize, charY + trsize, charZ + trsize,
    charX + trsize, charY + trsize, charZ - trsize,
    charX - trsize, charY + trsize, charZ - trsize,

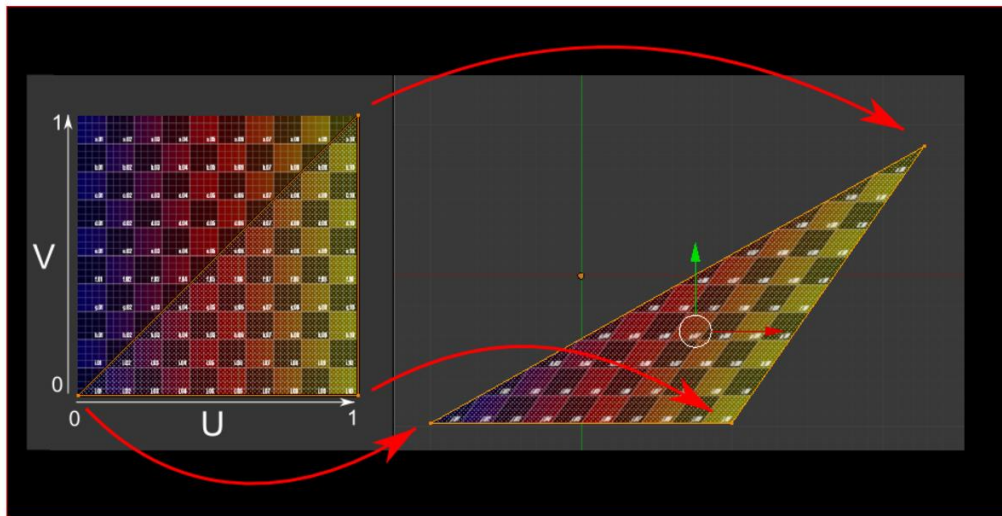
    // Down (Y = -trsize)
    charX - trsize, charY - trsize, charZ + trsize,
    charX + trsize, charY - trsize, charZ + trsize,
    charX - trsize, charY - trsize, charZ - trsize,

    charX + trsize, charY - trsize, charZ + trsize,
    charX + trsize, charY - trsize, charZ - trsize,
    charX - trsize, charY - trsize, charZ - trsize
};

```

-We also defined the **UVData** array that holds the information for the UV coordinates calculated for the vertices of each triangle from the above vertex array.

-We utilized the material provided in the lab videos to determine these coordinates.



The UV coordinates have been arranged in exactly the same order as each face of the cube is drawn, ensuring correct mapping.

```
float UVData[] = {  
// Back (Z = +0.4)  
    0.0f, 0.0f,  
    1.0f, 0.0f,  
    0.0f, 1.0f,  
    1.0f, 0.0f,  
    1.0f, 1.0f,  
    0.0f, 1.0f,
```

```
// Front (Z = -0.4)  
    0.0f, 0.0f,  
    1.0f, 0.0f,  
    0.0f, 1.0f,  
    1.0f, 0.0f,  
    1.0f, 1.0f,  
    0.0f, 1.0f,
```

```
// Left (X = -0.4)  
    0.0f, 0.0f,  
    1.0f, 0.0f,  
    0.0f, 1.0f,  
    1.0f, 0.0f,  
    1.0f, 1.0f,  
    0.0f, 1.0f,
```

```
// Right (X = +0.4)  
    0.0f, 0.0f,  
    1.0f, 0.0f,  
    0.0f, 1.0f,  
    1.0f, 0.0f,  
    1.0f, 1.0f,  
    0.0f, 1.0f,
```

```
// Up (Y = +0.4)  
    0.0f, 0.0f,  
    1.0f, 0.0f,  
    0.0f, 1.0f,
```

```

        1.0f, 0.0f,
        1.0f, 1.0f,
        0.0f, 1.0f,

// Down (Y = -0.4)
        0.0f, 0.0f,
        1.0f, 0.0f,
        0.0f, 1.0f,
        1.0f, 0.0f,
        1.0f, 1.0f,
        0.0f, 1.0f
};

```

-Then, we create buffers for the vertex coordinates and the UV coordinates. Next, we set up the vertex attributes to use this data.

-Following that, we create and configure a texture, which is linked to the shader program.

-The MVP matrix is sent to the shader, and finally, the triangles forming the object are drawn.

-After drawing, the vertex attributes are disabled for cleanup.

```

//treasure
GLuint treasureVertexBuffer;
glGenBuffers(1, &treasureVertexBuffer);
glBindBuffer(GL_ARRAY_BUFFER, treasureVertexBuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(treasureVertices), treasureVertices,
GL_STATIC_DRAW);

GLuint treasureUVBuffer;
glGenBuffers(1, &treasureUVBuffer);
glBindBuffer(GL_ARRAY_BUFFER, treasureUVBuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(UVData), UVData, GL_STATIC_DRAW);

// Bind Vertex Buffer
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, treasureVertexBuffer);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

// Bind UV Buffer
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, treasureUVBuffer);
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 0, (void*)0);

GLuint textureID;
glGenTextures(1, &textureID);

glBindTexture(GL_TEXTURE_2D, textureID);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, data);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

// Bind Texture
glActiveTexture(GL_TEXTURE0);

```

```
glUniform1i(glGetUniformLocation(programID, "myTextureSampler"), 0);

// Send MVP matrix
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, &MVP[0][0]);

// Draw Call
glDrawArrays(GL_TRIANGLES, 0, 36);

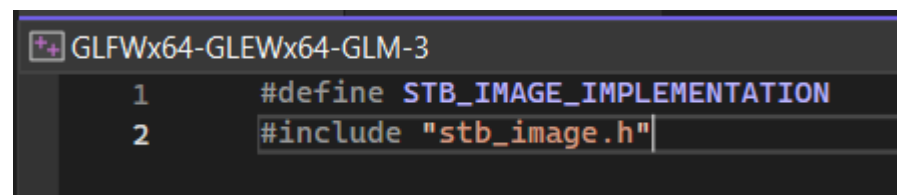
// Disable attributes after drawing
glDisableVertexAttribArray(0);
glDisableVertexAttribArray(1);
```

-To load the desired image onto the cube, we use this command:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, data);
```

-Where the data argument is obtained as follows:

Initially, we needed to download a library for image loading, which we added to the project folder and created its own .cpp file, containing:



```
GLFWx64-GLEWx64-GLM-3
1  #define STB_IMAGE_IMPLEMENTATION
2  #include "stb_image.h"
```

-We added the image coins.jpg, which was provided to us, into the project folder.

-We defined three variables for the image's width, height, and number of color channels:

```
int width, height, nrChannels;
```

Then, we loaded the image with the command:

```
unsigned char* data0 = stbi_load("coins.jpg", &width, &height,
&nrChannels, 0);
```

Due to a bonus question we implemented later, the variable is named data0 instead of data.

The explanation for loading multiple images follows later in the bonus questions section of the report.

To make the treasure appear every 5 seconds, as we have defined, we initially needed to add the following libraries to the program:

```
#include <time.h>
#include <chrono>
```

In **main**, we define an auto start variable that records the current time moment:

```
auto start = std::chrono::steady_clock::now(); // Initial starting time
```

We also initialize a Boolean variable touched as false:

```
boolean touched = false;
```

This variable becomes true when the player touches the treasure.

Then, inside the do loop, we initialize a variable now which records the current time each iteration and calculate the difference between start and now to check if 5 seconds have passed.

Using an if statement, we check if the player has not found the treasure and if 5 seconds have passed, then we randomly change the treasure's position and texture. After this, we reset start = now so that this process repeats every 5 seconds.

Once inside this if block, we initialize a random variable using getRandomNumber(). Then we initialize the treasure's coordinates and translate them to the game space. We also store the player's current coordinates as local variables at that time. Additionally, we assign a random texture to the treasure, which we will explain later.

Next, we check if the treasure's coordinates coincide with the player's coordinates. If the player's and treasure's positions overlap, we simply increment the random number by 1, unless it's the last element of the array—in that case, we subtract 10 instead.

Here is the relevant code snippet:

```
std::chrono::duration<int> elapsed_seconds =  
std::chrono::duration_cast<std::chrono::seconds>(now - start);  
  
if (elapsed_seconds.count() >= 5 && touched == false) {  
    data = pics[getRandomNum3()];  
  
    int rand = getRandomNumber();  
    int k1 = characterX - 0.5;  
    int w1 = characterY - 0.5;  
    int tx = xCoord[rand] - 5;  
    int ty = 5 - yCoord[rand];  
    if ((tx == k1) && (ty == w1)) {  
        if (rand == 42) {  
            charX = xCoord[rand - 10] - 5 + 0.5f;  
            charY = 5 - yCoord[rand - 10] + 0.5f;  
            charZ = 0.5;  
            start = now;  
        }  
        if (rand != 42) {  
            charX = xCoord[rand + 1] - 5 + 0.5f;  
            charY = 5 - yCoord[rand + 1] + 0.5f;  
            charZ = 0.5;  
            start = now;  
        }  
    }  
    else {  
        charX = xCoord[rand] - 5 + 0.5f;  
        charY = 5 - yCoord[rand] + 0.5f;
```

```

        charZ = 0.5;
        start = now;
    }
}

```

(iv):

For checking if the player has found the treasure, we modified the cube movement code from the previous exercise.

At each check for the four possible movements, we define the following variables:

```

int k = characterX - 0.5;
int w = characterY - 0.5;
int tx = charX - 0.5;
int ty = charY - 0.5;

```

These represent the coordinates of the player's cube and the treasure, respectively.

From now on, at each check of the next position, if the next position matches the current treasure position, we set `touched = true`; so that the treasure's position and texture stop changing. This prevents any issues during the shrinking and disappearing process.

If the coordinates match during any check of the next movement (up, down, left, right), then the treasure starts shrinking. Once it shrinks to half its size, it disappears. When the player touches the treasure, a sound effect and a visual effect play.

The updated movement code is as follows:

`//MOVE LEFT`

```

if (glfwGetKey(window, GLFW_KEY_J) == GLFW_PRESS) {
    if (!key_J_pressed) {
        int k = characterX - 0.5;
        int w = characterY - 0.5;
        int tx = charX - 0.5;
        int ty = charY - 0.5;
        if (w == ty && k == tx + 1) {
            touched = true;
            if (!isRunning) {

                isRunning = true;
                std::thread m(playMP3Thread);
                m.detach(); // Independent thread

                std::thread t(minimizeTreasureThread);
                t.detach(); // Independent thread

                std::thread z(FlashThread);
                z.detach(); // Independent thread

            }
        }
        if (characterX == -4.50f && characterY == 3.50f && characterZ ==
0.5f) { // Special teleport condition
            characterX += 9.0f;
            characterY -= 5.0f;
        }
    }
}

```

```

        else if (maze[int(5 - w)][int((k - 1) + 5)] == 0 && (w != ty or k
!= tx + 1)) { // Check if space to the left is not a wall
            characterX -= 1.0f;

        }

        key_J_pressed = true;
    }
}
else {
    key_J_pressed = false;
}

// MOVE RIGHT
if (glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS) {
    if (!key_L_pressed) {
        int k = characterX - 0.5;
        int w = characterY - 0.5;
        int tx = charX - 0.5;
        int ty = charY - 0.5;
        if (w == ty && k == tx - 1) {
            touched = true;
            if (!isRunning) {
                isRunning = true;
                std::thread m(playMP3Thread);
                m.detach(); // Independent thread

                std::thread t(minimizeTreasureThread);
                t.detach(); // Independent thread

                std::thread z(FlashThread);
                z.detach(); // Independent thread
            }
        }
        if (characterX == 4.50f && characterY == -1.50f && characterZ ==
0.5f) { // Special teleport condition
            characterX -= 9.0f;
            characterY += 5.0f;
        }
        else if (maze[int(5 - w)][int((k + 1) + 5)] == 0 && (w != ty or k
!= tx - 1)) { // Check if space to the right is not a wall
            characterX += 1.0f;
        }
        key_L_pressed = true;
    }
}
else {
    key_L_pressed = false;
}

// MOVE UP
if (glfwGetKey(window, GLFW_KEY_I) == GLFW_PRESS) {
    if (!key_I_pressed) {
        int k = characterX - 0.5;
        int w = characterY - 0.5;
        int tx = charX - 0.5;
        int ty = charY - 0.5;
        if (w + 1 == ty && k == tx) {
            touched = true;
            if (!isRunning) {
                isRunning = true;
                std::thread m(playMP3Thread);
                m.detach(); // Independent thread

```

```

        std::thread t(minimizeTreasureThread);
        t.detach(); // Independent thread

        std::thread z(FlashThread);
        z.detach(); // Independent thread
    }
    if (maze[int(5 - (w + 1))][int(k + 5)] == 0 && (w + 1 != ty or k
!= tx)) { // Check if space above is not a wall
        characterY += 1.0f;
    }
    key_I_pressed = true;
}
else {
    key_I_pressed = false;
}

// MOVE DOWN
if (glfwGetKey(window, GLFW_KEY_K) == GLFW_PRESS) {
    int k = characterX - 0.5;
    int w = characterY - 0.5;
    int tx = charX - 0.5;
    int ty = charY - 0.5;
    if (w - 1 == ty && k == tx) {
        touched = true;
        if (!isRunning) {
            isRunning = true;
            std::thread m(playMP3Thread);
            m.detach(); // Independent thread

            std::thread t(minimizeTreasureThread);
            t.detach(); // Independent thread

            std::thread z(FlashThread);
            z.detach(); // Independent thread
        }
        if (!key_K_pressed) {
            if (maze[int(5 - (w - 1))][int(k + 5)] == 0 && (w - 1 != ty or k
!= tx)) { // Check if space below is not a wall
                characterY -= 1.0f;
            }
            key_K_pressed = true;
        }
    }
    else {
        key_K_pressed = false;
    }
}

```

In the if statement that checks whether the player has "touched" the treasure—in order to trigger sound playback, shrinking, and disappearance of the cube, as well as visual and sound effects simultaneously and independently of the main do loop—we made use of threads.

We added the appropriate library:

```
#include <thread>
```


We used three threads, but here we analyze only the one responsible for shrinking the treasure.

First, we had to create a function that would handle the shrinking and eventual disappearance of the treasure.

The function `minimizeTreasure()` uses the variable:

```
GLfloat trsize = 0.4f; // Define trsize as a GLfloat to represent the size of the treasure
```

which we had defined earlier, so that it can modify the size of the cube.

The function reduces `trsize` progressively, with a small delay between each step, until the cube shrinks to half its size or less, at which point it disappears. Here's an outline of what the function looks like:

```
void minimizeTreasure() {
    for (int i = 0; i < 4; i++) {
        if (trsize > 0.2) {
            trsize = trsize * 0.99f;
        }
        else if (trsize <= 0.2) {
            charX = 0 - 5 + 0.5;
            charY = 5 - 0 + 0.5;
            trsize = 0.0f;
            break;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
}
```

The check performed in the function is that, as long as the cube has not yet shrunk to half its size, it multiplies its size by 0.99. This value was chosen to make the shrinking process smooth and noticeable to the user.

The following command is used to prevent execution errors and allow for a small delay between each size update:

```
std::this_thread::sleep_for(std::chrono::milliseconds(1));
```

This introduces a slight pause, ensuring smoother visual shrinking and avoiding potential performance issues.

In the else part of the statement—when the treasure size has reached or dropped below the threshold—we modify the treasure's coordinates and size to zero, effectively making it disappear.

- After implementing `minimizeTreasure()`, we create the `void minimizeTreasureThread()`, which calls `minimizeTreasure()`, () as long as a boolean variable we have defined (initially set to false) is true:

```
std::atomic<bool> isRunning(false);
```

-This variable is also used in all the threads we have created.

-Then we make the program wait a little before repeating the while loop, to avoid any errors.

```
void minimizeTreasureThread() {
    while (isRunning) {
        minimizeTreasure();
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
}
```

So, in the movement check, when it is time to execute `minimizeTreasure()`, we set:

```
isRunning = true;
```

create a new thread, and make it independent from the rest of the program:

```
std::thread t(minimizeTreasureThread);
t.detach(); // Independent thread
```

(v):

For the implementation of the camera function, we kept the code we already had and added to it.

The part that pre-existed is the following:

```
void camera_function(GLFWwindow* window) {
    // Define static camera position and target
    static glm::vec3 cameraPosition = glm::vec3(0.0f, 0.0f, 20.0f);
    // Camera position
    glm::vec3 cameraTarget = glm::vec3(0.0f, 0.0f, 0.25f);
    // Target position at the maze center
    glm::vec3 upVector = glm::vec3(0.0f, 1.0f, 0.0f);
    // Up vector for orientation

    // Define rotation angles
    static float angleX = 0.0f;
    static float angleY = 0.0f;
    static float zoom = 20.0f; // Start with some initial zoom level

    // Define panning values
    static glm::vec3 panOffset = glm::vec3(0.0f, 0.0f, 0.0f);

    // Check key presses for rotation and zoom
    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) {
        angleX += 0.1f; // Rotate around the X-axis (positive)
    }
    if (glfwGetKey(window, GLFW_KEY_X) == GLFW_PRESS) {
        angleX -= 0.1f; // Rotate around the X-axis (negative)
    }
    if (glfwGetKey(window, GLFW_KEY_Q) == GLFW_PRESS) {
        angleY += 0.1f; // Rotate around the Y-axis (positive)
    }
    if (glfwGetKey(window, GLFW_KEY_Z) == GLFW_PRESS) {
        angleY -= 0.1f; // Rotate around the Y-axis (negative)
    }
}
```

```

if (glfwGetKey(window, GLFW_KEY_KP_ADD) == GLFW_PRESS) {
    zoom -= 0.1f; // Zoom in
}
if (glfwGetKey(window, GLFW_KEY_KP_SUBTRACT) == GLFW_PRESS) {
    zoom += 0.1f; // Zoom out
}

// Limit the zoom level
zoom = glm::clamp(zoom, 5.0f, 50.0f); // Restrict zoom level to avoid
clipping or too far away

```

Following that, we added the following:

We defined the variables

```

// Define panning values
static glm::vec3 panOffset = glm::vec3(0.0f, 0.0f, 0.0f);

float panSpeed = 0.01f;

```

The variable panSpeed sets the panning speed to 0.01.

The variable panOffset stores the offset for the panning movement.

```

// Check key presses for panning
float panSpeed = 0.01f; // Adjust speed of panning
if (glfwGetKey(window, GLFW_KEY_H) == GLFW_PRESS) {
    panOffset.x -= panSpeed; // Move left along the x-axis
}
if (glfwGetKey(window, GLFW_KEY_G) == GLFW_PRESS) {
    panOffset.x += panSpeed; // Move right along the x-axis
}
if (glfwGetKey(window, GLFW_KEY_B) == GLFW_PRESS) {
    panOffset.y += panSpeed; // Move up along the y-axis
}
if (glfwGetKey(window, GLFW_KEY_T) == GLFW_PRESS) {
    panOffset.y -= panSpeed; // Move down along the y-axis
}

// Calculate rotation matrices around X and Y
glm::mat4 rotationX = glm::rotate(glm::mat4(1.0f), angleX,
glm::vec3(1.0f, 0.0f, 0.0f));
glm::mat4 rotationY = glm::rotate(glm::mat4(1.0f), angleY,
glm::vec3(0.0f, 1.0f, 0.0f));

// Apply the rotations and set the camera position
glm::vec3 rotatedPosition = glm::vec3(rotationY * rotationX *
glm::vec4(0.0f, 0.0f, zoom, 1.0f));
cameraPosition = rotatedPosition + panOffset;

// Update the view matrix
ViewMatrix = glm::lookAt(
    cameraPosition, // Camera position
    cameraTarget + panOffset, // Target position (adjusted for
panning)
    upVector // Up vector
);
}

```

Depending on the pressing of specific keys, the offset changes along the x or y axis, moving the view left/right or up/down respectively.

****BONUS**

(a) For applying the 3 random textures, we first created a function that selects a random number from 0 to 2.

```
int getRandomNum3() {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    static std::uniform_int_distribution<> distr(0, 2);

    return distr(gen);
}
```

Inside the main function, we load the 3 images — for convenience, we chose all images to have the same dimensions.

```
int width, height, nrChannels;
unsigned char* data0 = stbi_load("coins.jpg", &width, &height,
&nrChannels, 0);
unsigned char* data1 = stbi_load("tree.jpg", &width, &height,
&nrChannels, 0);
unsigned char* data2 = stbi_load("tom.jpg", &width, &height,
&nrChannels, 0);
```

Then, we create an array named pics, whose elements are the 3 images:

```
unsigned char* pics[] = { data0, data1, data2 };
```

To have an initial random texture, we define a variable data which is a random element from the pics array:

```
unsigned char* data = pics[getRandomNum3()];
```

Finally, we update the data variable by using the same line of code in the part that changes the texture position every 5 seconds — now it also randomly changes the texture every 5 seconds.

(b) For the visual effect, we chose to make the player's cube flash repeatedly when it finds the treasure, as follows:

The main idea is to modify the existing array CubeColorData[], periodically changing its color — one moment yellow, the next red. This creates the flashing visual effect.

So, we define the following functions:

```
void ChangeCubeColorToRed() {
    for (int i = 0; i < sizeof(cubeColorData) / sizeof(cubeColorData[0]);
i += 3) {
        cubeColorData[i] = 1.0f;
        cubeColorData[i + 1] = 0.0f;
        cubeColorData[i + 2] = 0.0f;
    }
}
```

```
void ChangeCubeColorToYellow() {
    for (int i = 0; i < sizeof(cubeColorData) / sizeof(cubeColorData[0]);
i += 3) {
        cubeColorData[i] = 1.0f;
```

```

        cubeColorData[i + 1] = 1.0f;
        cubeColorData[i + 2] = 0.0f;
    }
}

```

Essentially, one function encodes the color to red, while the other to yellow.

Then, we create the `void FlashRedToYellow()` function which periodically calls these functions every 100ms.

```

void FlashRedToYellow() {
    ChangeCubeColorToRed();

    // Delay
    std::this_thread::sleep_for(std::chrono::milliseconds(100));

    ChangeCubeColorToYellow();
}

```

Φυσικά, για να τρέχει ανεξάρτητα από το υπόλοιπο πρόγραμμα, φτιάχνουμε το αντίστοιχο νήμα, όπως το εξηγήσαμε και προηγουμένως.

```

void FlashThread() {
    while (isRunning) {
        FlashRedToYellow();
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}

```

We call this thread inside the movement control.

For the sound effect, we downloaded an mp3 file which we then converted to a .wav file.

We included the corresponding libraries:

```

#include <windows.h>
#include <mmsystem.h>
#include <mciapi.h>

#pragma comment(lib, "Winmm.lib")

```

After that, we also added Winmm.lib to the project properties under Additional Dependencies.

Next, we implement the playMP3 function.

— Note: Initially, we tried to make it work with an mp3 file (hence the name), but later we never changed the name after switching to a .wav file.

Since we want the sound to play only once and not repeatedly, inside the function we declare a static local variable: `static bool hasPlayed = false;`

and we call the sound file only if `hasPlayed = false;` once it plays, we set `hasPlayed = true;` so the sound does not play again.

```

void playMP3() {
    static bool hasPlayed = false;

    if (!hasPlayed) {
        PlaySound(TEXT("sus.wav"), NULL, SND_SYNC | SND_FILENAME);
        hasPlayed = true;
    }
}

```

Similarly (as with `minimizeTreasureThread()`) we implement the thread for `playMP3()`:

```

void playMP3Thread() {
    while (isRunning) {
        playMP3();
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
    }
}

```

We call this thread inside the movement control along with the other related functionality.

2. Implementation details:

We implemented the project on Windows using Microsoft Visual Studio x64.

The P1B shaders correspond to the treasure and its texture.

The P1C shaders concern the maze and the character.

Just before the end of the do loop, we added the

```
commandstd::this_thread::sleep_for(std::chrono::milliseconds(16));
```

to ensure the loop runs synchronized and to prevent the program from being overloaded.