Universidade da Beira Interior
Departamento de Informática

Labirinto

Elaborado por:

António Sardinha - Nº 39477
Pedro Brito - Nº 39490
Rui Raposo - Nº 39986

Orientador:

Professor Doutor Abel João Padrão Gomes

1 de Junho de 2020

Agradecimentos

Queremos agradecer em especial ao Professor Doutor Abel João Padrão Gomes pela ajuda no desenvolvimento deste projeto e por todo o seu material de apoio que levou ao sucesso do mesmo.

Conteúdo

Capítulo 1

Introdução

1.1 Descrição da proposta

O nosso projeto consiste no desenvolvimento de um labirinto 3D em que o utilizador terá de se mover de forma a encontrar o buraco que lhe dará a vitória e consequentemente o fará progredir para o nível seguinte.

1.2 História do Jogo

Um labirinto é definido como toda a construção que possua entradas incertas para encontrar uma saída. São compostos por paredes que têm como objetivo confundir a orientação espacial questionando qual a saída certa.
O primeiro labirinto registado na história foi o egípcio. Heródoto, um escritor e vigiante grego, visitou o labirinto egípcio no século V aC. O edifício estava localizado logo acima do lago *Moeris* e em frente a *Crocodilopolis*. Heródoto afirmou que todas as obras e edifícios gregos eram inferiores ao labirinto egípcio.
O labirinto mais famoso de todos tempos é o Labirinto de Creta. Segundo a lenda, o rei Egeu foi forçado a prestar homenagem a rei Minos, cujo reino estava na ilha que agora chamamos de Creta. A homenagem incluía 7 rapazes e 7 raparigas. No subsolo, bem abaixo do palácio do rei Minos, na cidade de *Knossos*, havia um labirinto enorme. Dentro do labirinto, Minos mantinha um monstro chamado Minotauro. O Minotauro era uma criatura hedionda que era meio homem e meio touro. Os catorze jovens da Grécia seriam soltos no labirinto, onde se iriam perder e eventualmente devorados pelo monstro. Thesesus, filho do rei Aegeus, decidiu voluntariar-se como uma das vítimas de sacrifício para que ele pudesse tentar matar o Minotauro. Thesesus foi bem sucedido. Ele matou o Minotauro, depois usou uma trilha de barbante que começara a deitar na entrada do labirinto para encontrar o caminho para sair do labirinto.

1.3 Organização do Documento

Este documento encontra-se estruturado da seguinte forma:

1. O primeiro capítulo – **Introdução** – Descreve a proposta da aplicação, constituição do grupo, e descreve como está organizado o documento.

2. O segundo capítulo – **Implementação** – Descreve os métodos e escolhas de implementação.

3. O terceiro capítulo – **Conclusões e Trabalho Futuro** – É feita uma análise crítica do trabalho feito pelo grupo e uma visão futura em relação ao trabalho.

Capítulo 2

Implementação

2.1 Gestão do Projeto

- Rui Raposo
  - Planeamento e modelação do labirinto;
  - Iluminação;
  - Jogabilidade;
  - Relatório.
- António Sardinha
  - Modelação do labirinto;
  - Iluminação;
  - Texturas;
  - Jogabilidade;
  - Relatório.
- Pedro Brito
  - Planeamento do labirinto;
  - Texturas;
  - Jogabilidade;
  - Relatório.

2.2 Descrição do Código

2.2.1 Programa principal (main.cpp)

Nesta parte incluimos todas as bibliotecas que necessitámos para o desenvolvimento do projeto.

```
#define STB_IMAGE_IMPLEMENTATION
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>
#include "include/shader_m.h"
#include "camera.h"
#include "stb_image.h"
#include <iostream>
```

Definimos a posição inicial da camêra e inicializamos a primeira posição do rato no meio do ecrã.

```
// camera
Camera camera(glm::vec3(-13.5f, 0.5f, 16.0f));
float lastX = SCR_WIDTH / 2.0f;
float lastY = SCR_HEIGHT / 2.0f;
bool firstMouse = true;
```

Usamos o deltatime para garantir que independentemente do computador em que o jogo é corrido, o jogo se porte da mesma maneira.

```
// timing
float deltaTime = 0.0f;
float lastFrame = 0.0f;
```

Definimos a posição inicial do foco de luz.

```
glm::vec3 lightPos(0.0f, 15.0f, 0.0f);
```

Inicializamos o *glfw*.

```
// glfw: initialize and configure
    // ------------------------------
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
```

```
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE
        );

#ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif
```

Criamos a janela com as dimensões definidas globalmente e metemos a *window* a ouvir os comandos que o utilizador irá inserir.

```
// glfw window creation
    // --------------------
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT,
        "LearnOpenGL", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl
            ;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFrame\textit{buffers}izeCallback(window,
        framebuffer_size_callback);
    glfwSetCursorPosCallback(window, mouse_callback);
    glfwSetScrollCallback(window, scroll_callback);

    // tell GLFW to capture our mouse
    glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
```

Nesta parte definimos damos o load dos shaders.

```
// build and compile our shader zprogram
    // ------------------------------------
    Shader lightingShader("shaders/2.1.basic_lighting.vs", "
        shaders/2.1.basic_lighting.fs");
    Shader lightingShader_walls("shaders/2.1.
        basic_lighting_walls.vs" "shaders/2.1.
        basic_lighting_walls.fs");
    Shader lightingShader_intWalls("shaders/2.1.
        basic_lighting_intWalls.vs", "shaders/2.1.
        basic_lighting_intWalls.fs");
    Shader lightingShader_block("shaders/2.1.
        basic_lighting_block.vs", "shaders/2.1.
        basic_lighting_block.fs");
    Shader lampShader("shaders/2.1.lamp.vs", "shaders/2.1.lamp.
        fs");
```

Declaramos e inicializamos os *arrays* dos vértices do plano, das paredes exteriores, das paredes interiores, dos blocos e da luz.

```
// Declaramos o vetor que contem os vertices do plano
// 3 elementos -> Coordenadas do vertice
// 3 elementos -> Normais
// 2 elementos -> Coordenadas da textura
 float vertices[] = {
    ...
}


// Declaramos o vetor que contem os vertices das paredes
   exteriores
// 3 elementos -> Coordenadas do vertice
// 3 elementos -> Normais
// 2 elementos -> Coordenadas da textura
float paredes[] = {
    ...
}

// Declaramos o vetor que contem os vertices das paredes
   interiores
// 3 elementos -> Coordenadas do vertice
// 3 elementos -> Normais
// 2 elementos -> Coordenadas da textura
float verticesIntWall[] = {
    ...
}

// Declaramos o vetor que contem os vertices dos blocos
// 3 elementos -> Coordenadas do vertice
// 3 elementos -> Normais
// 2 elementos -> Coordenadas da textura
float blocos[] = {
    ...
}

//Declaramos o vetor que contem os vertices da luz
// 3 elementos -> Coordenadas do vertice
// 3 elementos -> Normais
 float esfera_light[] = {
    ...
 }
```

Geramos os vértices e os *buffers* do plano, depois fazemos o link destes dois. Definimos quais são os atributos da posição e quais são os atributos normais.

```
// first, configure the plan's VAO (and VBO)
    unsigned int VBO, planVAO;
    glGenVertexArrays(1, &planVAO);
    glGenBuffers(1, &VBO);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices,
        GL_STATIC_DRAW);

    glBindVertexArray(planVAO);


     // PLANO
    // position attribute
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(
        float), (void*)0);
    glEnableVertexAttribArray(0);

    // normal attribute
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(
        float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(1);


    glVertexAttribPointer(14, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(
        float), (void*)(6 * sizeof(float)));
    glEnableVertexAttribArray(14);

}
```

Geramos os vértices e os *buffers* das paredes exteriores, depois fazemos o link destes dois. Definimos quais são os atributos da posição e quais são os atributos normais e ainda os atributos das coordenadas das texturas.

```
// PAREDES
    unsigned int VBO_Walls, wallsVAO;
    glGenVertexArrays(1, &wallsVAO);
    glGenBuffers(1, &VBO_Walls);

    glBindBuffer(GL_ARRAY_BUFFER, VBO_Walls);
    glBufferData(GL_ARRAY_BUFFER, sizeof(paredes), paredes,
        GL_STATIC_DRAW);

    glBindVertexArray(wallsVAO);
```

```
    // position attribute
    glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(
        float), (void*)0);
    glEnableVertexAttribArray(2);

    // normal attribute
    glVertexAttribPointer(3,  3, GL_FLOAT, GL_FALSE, 8 * sizeof(
        float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(3);


    glVertexAttribPointer(12, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(
        float), (void*)(6 * sizeof(float)));
    glEnableVertexAttribArray(12);


}
```

Geramos os vértices e os *buffers* das paredes interiores, depois fazemos o link destes dois. Definimos quais são os atributos da posição e quais são os atributos normais e ainda os atributos das coordenadas das texturas.

```
unsigned int VBO_intWalls, intWallsVAO;
    glGenVertexArrays(1, &intWallsVAO);
    glGenBuffers(1, &VBO_intWalls);

    glBindBuffer(GL_ARRAY_BUFFER, VBO_intWalls);
    glBufferData(GL_ARRAY_BUFFER, sizeof(verticesIntWall),
        verticesIntWall, GL_STATIC_DRAW);

    glBindVertexArray(intWallsVAO);

    glVertexAttribPointer(4, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(
        float), (void*)0);
    glEnableVertexAttribArray(4);

    // normal attribute
    glVertexAttribPointer(5,  3, GL_FLOAT, GL_FALSE, 8 * sizeof(
        float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(5);


    glVertexAttribPointer(11, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(
        float), (void*)(6 * sizeof(float)));
    glEnableVertexAttribArray(11);
```

```
}
```

Geramos as matrizes de vértices e os *buffers* dos blocos, depois fazemos o link destes dois. Definimos quais são os atributos da posição e quais são os atributos normais e ainda os atributos das coordenadas das texturas.

```
//Blocos
    unsigned int VBO_block, blockVAO;
    glGenVertexArrays(1, &blockVAO);
    glGenBuffers(1, &VBO_block);

    glBindBuffer(GL_ARRAY_BUFFER, VBO_block);
    glBufferData(GL_ARRAY_BUFFER, sizeof(blocos), blocos,
        GL_STATIC_DRAW);

    glBindVertexArray(blockVAO);

    glVertexAttribPointer(6, 3, GL_FLOAT, GL_FALSE, 8 * sizeof(
        float), (void*)0);
    glEnableVertexAttribArray(6);



    // normal attribute
    glVertexAttribPointer(7,  3, GL_FLOAT, GL_FALSE, 8 * sizeof(
        float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(7);



    glVertexAttribPointer(10, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(
        float), (void*)(6 * sizeof(float)));
    glEnableVertexAttribArray(10);


}
```

Geramos os vértices e os *buffers* da esfera de iluminação, depois fazemos o link destes dois. Definimos quais são os atributos da posição e quais são os atributos normais.

```
// second, configure the light's VAO (VBO stays the same; the
    vertices are the same for the light object which is also a 3
    D cube)
    unsigned int lightVAO, lightVBO;
    glGenVertexArrays(1, &lightVAO);
```

```
    glGenBuffers(1, &lightVBO);

    glBindBuffer(GL_ARRAY_BUFFER, lightVBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(esfera_light),
        esfera_light, GL_STATIC_DRAW);

    glBindVertexArray(lightVAO);

    glVertexAttribPointer(8, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(
        float), (void*)0);
    glEnableVertexAttribArray(8);

    // normal attribute
    glVertexAttribPointer(9,  3, GL_FLOAT, GL_FALSE, 6 * sizeof(
        float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(9);


}
```

Damos *load* nas texturas que vamos usar, madeira para as paredes interiores, granito para os blocos, e *obsidian* para as paredes exteriores.

```
int width, height, nrChannels;
    unsigned int madeira = loadTexture("madeira.png");


    int width2, height2, nrChannels2;
    unsigned int granito = loadTexture("granito.jpg");


      int width3, height3, nrChannels3;
    unsigned int obsidian = loadTexture("obsidian.jpg");

        int width4, height4, nrChannels4;
    unsigned int plano = loadTexture("plano.jpg");


}
```

Temos o ciclo de vida do programa, que só acaba quando a janela for fechada.

```
// render loop
    // -----------
    while (!glfwWindowShouldClose(window))
    {
    .....
    }
```

```
}
```

Garantimos que o programa corre igual para todos atualizando o delta time.

```
// per-frame time logic
        // --------------------
        float currentFrame = glfwGetTime();
        deltaTime = currentFrame - lastFrame;
        lastFrame = currentFrame;
```

Processamos o input que a janela recebe.

```
// input
        // -----
        processInput(window);
```

Renderizamos a janela.

```
// render
        // ------
        glClearColor(0.6f, 0.6f, 0.6f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Nesta parte começamos por ativar o *shader* respetivo ao plano e passamos como parâmetros a todos os argumentos respetivos á cor e brilho. Depois criamos as matrizes de projeção,*view* e modelo que vamos mandar como parâmetros para o shader. E por fim, renderizamos o plano.

```
 // be sure to activate shader when setting uniforms/drawing
    objects
        lightingShader.use();

        lightingShader.setVec3("light.pos", lightPos);
        lightingShader.setVec3("viewPos", camera.Position);
        lightingShader_walls.setInt("texture0",0);


        glm::vec3 lightColorPlan;
        lightColorPlan.r = 2.0f;
        lightColorPlan.g = 0.7f;
        lightColorPlan.b = 1.3f;

        glm::vec3 diffColorPlan = lightColorPlan * glm::vec3(0.5
            f);
        glm::vec3 ambientColorPlan = diffColorPlan * glm::vec3
            (0.2f);
```

```
        lightingShader.setVec3("light.ambient", ambientColorPlan
            .r, ambientColorPlan.g, ambientColorPlan.b);
        lightingShader.setVec3("light.diffuse", diffColorPlan.r,
             diffColorPlan.g, diffColorPlan.b);
        lightingShader.setVec3("light.specular", 1.0f, 1.0f, 1.0
            f);

        lightingShader.setVec3("plan.ambient", 0.0f,1.0f,0.0f);
        lightingShader.setVec3("plan.diffuse", 0.0f,1.0f,0.0f);
        lightingShader.setVec3("plan.specular", 0.5f, 0.5f, 0.5f
            );
        lightingShader.setFloat("plan.shine", 32.0f);


        // view/projection transformations
        glm::mat4 projection = glm::perspective(glm::radians(
            camera.Zoom), (float)SCR_WIDTH / (float)SCR_HEIGHT,
            0.1f, 100.0f);
        glm::mat4 view = camera.GetViewMatrix();
        lightingShader.setMat4("projection", projection);
        lightingShader.setMat4("view", view);

        // world transformation
        glm::mat4 model = glm::mat4(1.0f);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, plano);
        lightingShader.setMat4("model", model);

        // render the cube
        glBindVertexArray(cubeVAO);
        glDrawArrays(GL_TRIANGLES, 0, 98359);
```

Nesta parte começamos por ativar o *shader* respetivo ás paredes externas e passamos como parâmetros a todos os argumentos respetivos á cor e brilho. Depois criamos as matrizes de projeção,*view* e modelo que vamos mandar como parâmetros para o *shader*. E por fim, renderizamos as paredes externas.

```
    // be sure to activate shader when setting uniforms/drawing
        objects
        lightingShader_walls.use();

        lightingShader_walls.setVec3("light.pos", lightPos);
        lightingShader_walls.setVec3("viewPos", camera.Position)
            ;
        lightingShader_walls.setInt("texture0",0);

        glm::vec3 lightColorExtWall;
```

```
lightColorExtWall.r = 1.0f;
lightColorExtWall.g = 1.0f;
lightColorExtWall.b = 1.0f;

glm::vec3 diffColorExtWall = lightColorExtWall * glm::
    vec3(0.5f);
glm::vec3 ambientColorExtWall = diffColorExtWall * glm::
    vec3(0.2f);

lightingShader_walls.setVec3("light.ambient",
    ambientColorExtWall.r, ambientColorExtWall.g,
    ambientColorExtWall.b);
lightingShader_walls.setVec3("light.diffuse",
    diffColorExtWall.r, diffColorExtWall.g,
    diffColorExtWall.b);
lightingShader_walls.setVec3("light.specular", 1.0f, 1.0
    f, 1.0f);

lightingShader_walls.setVec3("extWall.ambient", 1.0f,1.0
    f,1.0f);
lightingShader_walls.setVec3("extWall.diffuse", 1.0f,1.0
    f,1.0f);
lightingShader_walls.setVec3("extWall.specular", 0.5f,
    0.5f, 0.5f);
lightingShader_walls.setFloat("extWall.shine", 32.0f);

// view/projection transformations
projection = glm::perspective(glm::radians(camera.Zoom),
     (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
view = camera.GetViewMatrix();
lightingShader_walls.setMat4("projection", projection);
lightingShader_walls.setMat4("view", view);

// world transformation
model = glm::mat4(1.0f);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, obsidian);
lightingShader_walls.setMat4("model", model);

// render the plan
glBindVertexArray(wallsVAO);
glDrawArrays(GL_TRIANGLES, 0, 144);
```

Nesta parte começamos por ativar o *shader* respetivo ás paredes internas e passamos como parâmetros a todos os argumentos respetivos á cor e brilho. Depois criamos as matrizes de projeção,*view* e modelo que vamos mandar como parâmetros para o *shader*. E por fim, renderizamos as paredes internas.

```
        lightingShader_intWalls.use();


        lightingShader_intWalls.setVec3("light.pos", lightPos);
        lightingShader_intWalls.setVec3("viewPos", camera.
            Position);
        lightingShader_intWalls.setInt("texture0",0);

        glm::vec3 lightColorIntWall;
        lightColorIntWall.r = 1.5f;
        lightColorIntWall.g = 1.0f;
        lightColorIntWall.b = 1.5f;

        glm::vec3 diffColorIntWall = lightColorIntWall * glm::
            vec3(0.5f);
        glm::vec3 ambientColorIntWall = diffColorIntWall * glm::
            vec3(0.2f);

        lightingShader_intWalls.setVec3("light.ambient",
            ambientColorIntWall.r, ambientColorIntWall.g,
            ambientColorIntWall.b);
        lightingShader_intWalls.setVec3("light.diffuse",
            diffColorIntWall.r, diffColorIntWall.g,
            diffColorIntWall.b);
        lightingShader_intWalls.setVec3("light.specular", 1.0f,
            1.0f, 1.0f);

        lightingShader_intWalls.setVec3("intWall.ambient", 0.217
            f, 0.017f, 0.0f);
        lightingShader_intWalls.setVec3("intWall.diffuse",
            0.217f, 0.017f, 0.0f);
        lightingShader_intWalls.setVec3("intWall.specular", 0.5f
            , 0.5f, 0.5f);
        lightingShader_intWalls.setFloat("intWall.shine", 32.0f)
            ;



        // view/projection transformations
        projection = glm::perspective(glm::radians(camera.Zoom),
             (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
        view = camera.GetViewMatrix();
        lightingShader_intWalls.setMat4("projection", projection
            );
        lightingShader_intWalls.setMat4("view", view);

        // world transformation
```

```
        model = glm::mat4(1.0f);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, madeira);
        lightingShader_intWalls.setMat4("model", model);

        // render the plan
        glBindVertexArray(intWallsVAO);
        glDrawArrays(GL_TRIANGLES, 0, 900);
```

Nesta parte começamos por ativar o *shader* respetivo aos blocos e passamos como parâmetros a todos os argumentos respetivos á cor e brilho. Depois criamos as matrizes de projeção,*view* e modelo que vamos mandar como parâmetros para o *shader*. E por fim, renderizamos os blocos.

```
        lightingShader_block.use();

        lightingShader_block.setVec3("light.pos", lightPos);
        lightingShader_block.setVec3("viewPos", camera.Position)
            ;
        lightingShader_block.setInt("texture0",0);


        glm::vec3 lightColorBlock;
        lightColorBlock.r = 2.0f;
        lightColorBlock.g = 1.0f;
        lightColorBlock.b = 1.0f;

        glm::vec3 diffColorBlock = lightColorBlock * glm::vec3
            (0.5f);
        glm::vec3 ambientColorBlock = diffColorBlock * glm::vec3
            (0.2f);

        lightingShader_block.setVec3("light.ambient",
            ambientColorBlock.r, ambientColorBlock.g,
            ambientColorBlock.b);
        lightingShader_block.setVec3("light.diffuse",
            diffColorBlock.r, diffColorBlock.g, diffColorBlock.b)
            ;
        lightingShader_block.setVec3("light.specular", 1.0f, 1.0
            f, 1.0f);

        lightingShader_block.setVec3("block.ambient", 0.227f,
            0.245f, 0.249f);
        lightingShader_block.setVec3("block.diffuse", 0.227f,
            0.245f, 0.249f);
        lightingShader_block.setVec3("block.specular", 0.5f, 0.5
            f, 0.5f);
        lightingShader_block.setFloat("block.shine", 32.0f);
```

```
        // view/projection transformations
        projection = glm::perspective(glm::radians(camera.Zoom),
            (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
        view = camera.GetViewMatrix();
        lightingShader_block.setMat4("projection", projection);
        lightingShader_block.setMat4("view", view);

        // world transformation
        model = glm::mat4(1.0f);
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, granito);
        lightingShader_block.setMat4("model", model);

        // render the plan
        glBindVertexArray(blockVAO);
        glDrawArrays(GL_TRIANGLES, 0, 396);
```

Nesta parte começamos por ativar o *shader* respetivo à lâmpada e criamos as matrizes de projeção,*view* e modelo que vamos mandar como parâmetros para o *shader*. E por fim, renderizamos a lâmpada.

```
         // also draw the lamp object
        lampShader.use();
        lampShader.setMat4("projection", projection);
        lampShader.setMat4("view", view);
        model = glm::mat4(1.0f);
        model = glm::translate(model, lightPos);
        //model = glm::scale(model, glm::vec3(0.2f)); // a
            smaller plan
        lampShader.setMat4("model", model);

        glBindVertexArray(lightVAO);
        glDrawArrays(GL_TRIANGLES, 0, 2880);
```

Aqui damos *swap* nos buffers e lemos os eventos *InputOutput*.

```
// glfw: swap buffers and poll IO events (keys pressed/released,
    mouse moved etc.)
        glfwSwapBuffers(window);
        glfwPollEvents();
```

Damos *de-allocate* nos recursos se já serviram o seu propósito.

```
// optional: de-allocate all resources once they've outlived
   their purpose:
    glDeleteVertexArrays(1, &planVAO);
```

```
    glDeleteVertexArrays(1, &wallsVAO);
    glDeleteVertexArrays(1, &intWallsVAO);
    glDeleteVertexArrays(1, &lightVAO);
    glDeleteBuffers(1, &VBO);
    glDeleteBuffers(1, &VBO_Walls);
    glDeleteBuffers(1, &VBO_intWalls);
```

Limpamos todos os outros recursos.

```
// glfw: terminate, clearing all previously allocated GLFW
    resources.
    glfwTerminate();
    return 0;
```

Quando as dimensões da janela mudam garantimos que o *viewport* coincide com as novas dimensões da janela.

```
// process all input: query GLFW whether relevant keys are
    pressed/released this frame and react accordingly
//
    -----------------------------------------------------------------------

void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);

    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
        camera.ProcessKeyboard(FORWARD, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
        camera.ProcessKeyboard(BACKWARD, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
        camera.ProcessKeyboard(LEFT, deltaTime);
    if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
        camera.ProcessKeyboard(RIGHT, deltaTime);
}
```

Processamos os *inputs* do utilizador e atualizamos a posição da camêra conforme estes *inputs*.

```
// glfw: whenever the window size changed (by OS or user resize)
    this callback function executes
//
    -----------------------------------------------------------------------

void framebuffer_size_callback(GLFWwindow* window, int width,
    int height)
```

```
{
    // make sure the viewport matches the new window dimensions;
        note that width and
    // height will be significantly larger than specified on
       retina displays.
    glViewport(0, 0, width, height);
}
```

Sempre que o utilizador move o rato, esta função é chamada e atualiza a rotação da câmera no jogo.

```
// glfw: whenever the mouse moves, this callback is called
// -------------------------------------------------------
void mouse_callback(GLFWwindow* window, double xpos, double ypos
    )
{
    if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos; // reversed since y-
       coordinates go from bottom to top

    lastX = xpos;
    lastY = ypos;

    //printf("(%f, %f)\n", lastX, lastY);

    camera.ProcessMouseMovement(xoffset, yoffset);
}
```

Permite ao utilizador fazer *zoom*.

```
// glfw: whenever the mouse scroll wheel scrolls, this callback
    is called
// -------------------------------------------------------------

void scroll_callback(GLFWwindow* window, double xoffset, double
    yoffset)
{
    camera.ProcessMouseScroll(yoffset);
}
```

Nesta função, damos *load* na textura com ajuda da biblioteca "stb_image.h". Caso a textura tenha sido carregada com sucesso então vamos começar por ver em que formato está a imagem da textura, e depois consoante este formato vamos usar as funções do GL para carregar a textura para o *opengl*.

```
// utility function for loading a 2D texture from file
// ---------------------------------------------------
unsigned int loadTexture(char const * path)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);

    int width, height, nrComponents;
    unsigned char *data = stbi_load(path, &width, &height, &
        nrComponents, 0);
    if (data)
    {
        GLenum format;
        if (nrComponents == 1)
            format = GL_RED;
        else if (nrComponents == 3)
            format = GL_RGB;
        else if (nrComponents == 4)
            format = GL_RGBA;

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

        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
            GL_CLAMP_TO_EDGE );
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
            GL_CLAMP_TO_EDGE );
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
            GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
            GL_NEAREST);

        stbi_image_free(data);
    }
    else
    {
        std::cout << "Texture failed to load at path: " << path
            << std::endl;
        stbi_image_free(data);
```

```
    }

    return textureID;
}
```

2.2.2 *Fragment Shader* do Plano

No ficheiro 2.1.basic_lighting.fs criámos as estruturas necessárias dos materiais e da luz que queremos pintar, neste caso o plano. Também inicializámos todas as variáveis precisas para pintar o material e atribuir-lhe a cor desejada.

```
out vec4 FragColor;
struct Plan
{
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shine;
};

struct Luz
{
    vec3 pos;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

in vec3 Normal;
in vec3 FragPos;


uniform vec3 viewPos;
uniform Plan plan;
uniform Luz light;
```

Depois, na função main realizamos todas as operações que irão atribuir ao plano as propriedades da cor introduzidas no ficheiro principal (main.cpp).

```
void main()
{
    // ambient Plan
    vec3 ambientPlan = light.ambient * plan.ambient;

    // diffuse Plan
```

```
    vec3 normPlan = normalize(Normal);
    vec3 lightDirPlan = normalize(light.pos - FragPos);
    float diffPlan = max(dot(normPlan, lightDirPlan), 0.0);
    vec3 diffusePlan = light.diffuse * (diffPlan * plan.diffuse)
        ;

    // specular Plan
    vec3 viewDirPlan = normalize(viewPos - FragPos);
    vec3 reflectDirPlan = reflect(-lightDirPlan, normPlan);
    float specPlan = pow(max(dot(viewDirPlan, reflectDirPlan),
        0.0), plan.shine);
    vec3 specularPlan = light.specular * (specPlan * plan.
        specular);

    vec4 tex = texture(texture1,TexCoords);
    vec4 result = vec4(ambientPlan + diffusePlan + specularPlan
        ,1.0);

    FragColor =  mix(tex, result, 0);
}
```

2.2.3 *Vertex Shader* do Plano

NO ficheiro 2.1.basic_lighting.vs vamos passar os *buffers* dos vértices, das normais e da textura Isto vai desenhar o plano no ecrã.

```
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 14) in vec2 aTexCoords;



out vec2 TexCoords;

out vec3 FragPos;
out vec3 Normal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{

  TexCoords = aTexCoords  * (10.0f,10.0f);
    FragPos = vec3(model * vec4(aPos, 1.0));
```

```
    Normal = mat3(transpose(inverse(model))) * aNormal;

    gl_Position = projection * view * vec4(FragPos, 1.0);
}
```

2.2.4 *Fragment Shader* das Paredes Exteriores

No ficheiro 2.1.basic_lighting_walls.fs criámos as estruturas necessárias dos materiais e da luz que queremos pintar, neste caso as paredes exteriores. Também inicializámos todas as variáveis precisas para pintar o material, atribuir-lhe cor e aplicar-lhe a textura desejada.

```
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;


struct ExtWall
{
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shine;
};


struct Luz
{
    vec3 pos;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

in vec3 Normal;
in vec3 FragPos;


uniform vec3 viewPos;
uniform ExtWall extWall;
uniform Luz light;
```

De seguida, na função main realizamos todas as operações que irão atribuir às paredes exteriores as propriedades da cor introduzidas no ficheiro principal (main.cpp) e também a textura desejada.

```
void main()
{
    // ambient ExtWall
    vec3 ambientExtWall = light.ambient * extWall.ambient;

    // diffuse ExtWall
    vec3 normExtWall = normalize(Normal);
    vec3 lightDirExtWall = normalize(light.pos - FragPos);
    float diffExtWall = max(dot(normExtWall, lightDirExtWall),
        0.0);
    vec3 diffuseExtWall = light.diffuse * (diffExtWall * extWall
        .diffuse);

    // specular ExtWall
    vec3 viewDirExtWall = normalize(viewPos - FragPos);
    vec3 reflectDirExtWall = reflect(-lightDirExtWall,
        normExtWall);
    float specExtWall = pow(max(dot(viewDirExtWall,
        reflectDirExtWall), 0.0), extWall.shine);
    vec3 specularExtWall = light.specular * (specExtWall *
        extWall.specular);

    // texture
    vec4 tex = texture(texture1,TexCoords);
    vec4 result = vec4(ambientExtWall + diffuseExtWall +
        specularExtWall,1.0);

    FragColor =  mix(tex, result, 0);
}
```

2.2.5 *Vertex Shader* das paredes exteriores

No ficheiro 2.1.basic_lighting_walls.vs vamos passar os *buffers* dos vértices, das normais e das texturas. Isto vai desenhar as paredes exteriores no ecrã.

```
layout (location = 2) in vec3 aPos;
layout (location = 3) in vec3 aNormal;
layout (location = 12) in vec2 aTexCoords;



out vec2 TexCoords;
```

```
out vec3 FragPos;
out vec3 Normal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{

  TexCoords = aTexCoords * (10.0f,2.0f);;
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = mat3(transpose(inverse(model))) * aNormal;

    gl_Position = projection * view * vec4(FragPos, 1.0);
}
```

2.2.6 *Fragment Shader* das Paredes Interiores

No ficheiro 2.1.basic_lighting)_intWalls.fs criámos as estruturas necessárias dos materiais e da luz que queremos pintar, neste caso as paredes interiores. Também inicializámos todas as variáveis precisas para pintar o material, atribuir-lhe cor e aplicar-lhe a textura desejada.

```
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;


struct IntWall
{
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shine;
};


struct Luz
{
    vec3 pos;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
```

```
};

in vec3 Normal;
in vec3 FragPos;


uniform vec3 viewPos;
uniform IntWall intWall;
uniform Luz light;
```

De seguida, na função main realizamos todas as operações que irão atribuir às paredes interiores as propriedades da cor introduzidas no ficheiro principal (main.cpp) e também a textura desejada.

```
void main()
{
    // ambient IntWall
    vec3 ambientIntWall = light.ambient * intWall.ambient;

    // diffuse IntWall
    vec3 normIntWall = normalize(Normal);
    vec3 lightDirIntWall = normalize(light.pos - FragPos);
    float diffIntWall = max(dot(normIntWall, lightDirIntWall),
        0.0);
    vec3 diffuseIntWall = light.diffuse * (diffIntWall * intWall
        .diffuse);

    // specular IntWall
    vec3 viewDirIntWall = normalize(viewPos - FragPos);
    vec3 reflectDirIntWall = reflect(-lightDirIntWall,
        normIntWall);
    float specIntWall = pow(max(dot(viewDirIntWall,
        reflectDirIntWall), 0.0), intWall.shine);
    vec3 specularIntWall = light.specular * (specIntWall *
        intWall.specular);


    vec4 tex = texture(texture1,TexCoords);
    vec4 result = vec4(ambientIntWall + diffuseIntWall +
        specularIntWall,1.0);

    FragColor =  mix(tex, result, 0.65);
}
```

2.2.7 *Vertex Shader* das paredes Interiores

No ficheiro 2.1.basic_lighting_intWalls.vs vamos passar os *buffers* dos vértices, das normais e das texturas. Isto vai desenhar as paredes interiores no ecrã.

```
#version 330 core
layout (location = 4) in vec3 aPos;
layout (location = 5) in vec3 aNormal;
layout (location = 11) in vec2 aTexCoords;



out vec2 TexCoords;

out vec3 FragPos;
out vec3 Normal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{

  TexCoords = aTexCoords;
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = mat3(transpose(inverse(model))) * aNormal;

    gl_Position = projection * view * vec4(FragPos, 1.0);
}
```

2.2.8 *Fragment Shader* dos Blocos

No ficheiro 2.1.basic_lighting_block.fs criámos as estruturas necessárias dos materiais e da luz que queremos pintar, neste caso os blocos. Também inicializámos todas as variáveis precisas para pintar o material, atribuir-lhe cor e aplicar-lhe a textura desejada.

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D texture1;
```

```
struct Block
{
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    float shine;
};


struct Luz
{
    vec3 pos;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
};

in vec3 Normal;
in vec3 FragPos;


uniform vec3 viewPos;
uniform Block block;
uniform Luz light;
```

De seguida, na função main realizamos todas as operações que irão atribuir aos blocos as propriedades da cor introduzidas no ficheiro principal (main.cpp) e também a textura desejada.

```
void main()
{
    // ambient Block
    vec3 ambientBlock = light.ambient * block.ambient;

    // diffuse Block
    vec3 normBlock = normalize(Normal);
    vec3 lightDirBlock = normalize(light.pos - FragPos);
    float diffBlock = max(dot(normBlock, lightDirBlock), 0.0);
    vec3 diffuseBlock = light.diffuse * (diffBlock * block.
        diffuse);

    // specular Block
    vec3 viewDirBlock = normalize(viewPos - FragPos);
    vec3 reflectDirBlock = reflect(-lightDirBlock, normBlock);
    float specBlock = pow(max(dot(viewDirBlock, reflectDirBlock)
        , 0.0), block.shine);
    vec3 specularBlock = light.specular * (specBlock * block.
```

```
        specular);


    vec4 tex = texture(texture1,TexCoords);
    vec4 result = vec4(ambientBlock + diffuseBlock +
        specularBlock,1.0);

    FragColor =  mix(tex, result, 0.65);
}

}
```

2.2.9 *Vertex Shader* dos blocos

No ficheiro 2.1.basic_lighting_block.vs vamos passar os *buffers* dos vértices, das normais e das texturas. Isto vai desenhar os blocos no ecrã.

```
#version 330 core
layout (location = 6) in vec3 aPos;
layout (location = 7) in vec3 aNormal;
layout (location = 10) in vec2 aTexCoords;

out vec2 TexCoords;

out vec3 FragPos;
out vec3 Normal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{

  TexCoords = aTexCoords;
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = mat3(transpose(inverse(model))) * aNormal;

    gl_Position = projection * view * vec4(FragPos, 1.0);
}
```

2.2.10 *Fragment Shader* da Luz

Neste shader da lâmpada (2.1.lamp.fs) simplesmente pintamos a lâmpada com a cor especificada, neste caso, branco.

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0); // set alle 4 vector values to 1.0
}
```

2.2.11 *Vertex Shader* da Luz

No ficheiro 2.1.lamp.vs vamos passar os *buffers* dos vértices e das normais. Isto vai desenhar a lâmpada no ecrã.

```
#version 330 core
layout (location = 8) in vec3 aPos;
layout (location = 9) in vec3 aNormal;

out vec3 FragPos;
out vec3 Normal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{

  FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = mat3(transpose(inverse(model))) * aNormal;

    gl_Position = projection * view * vec4(FragPos, 1.0);

}
```

2.3 Apresentação do Labirinto

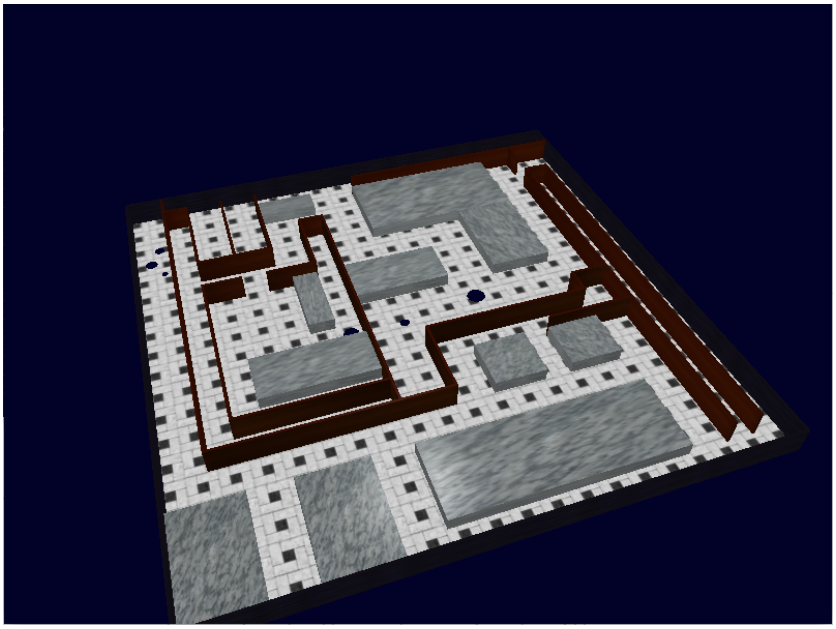

Figura 2.1: Vista geral do labirinto

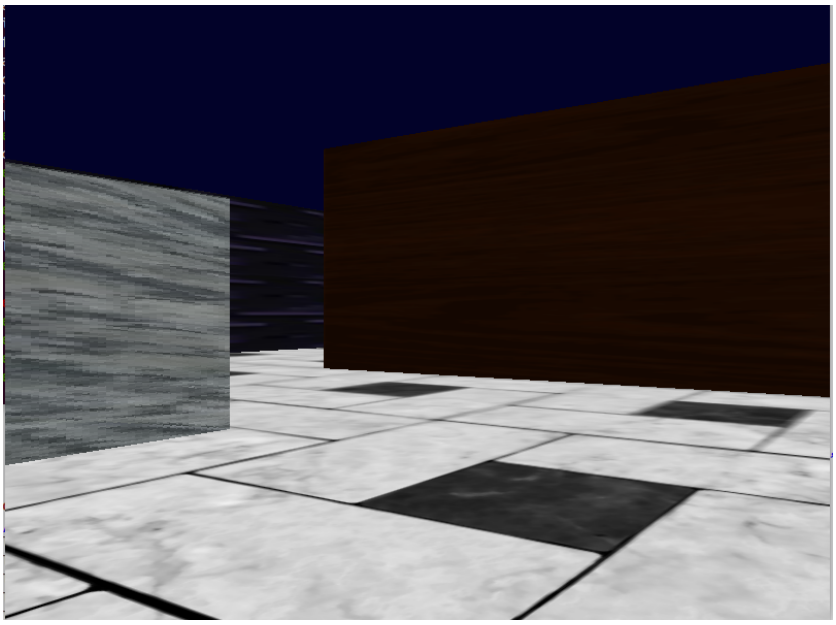

Figura 2.2: Visão do jogador em primeira pessoa

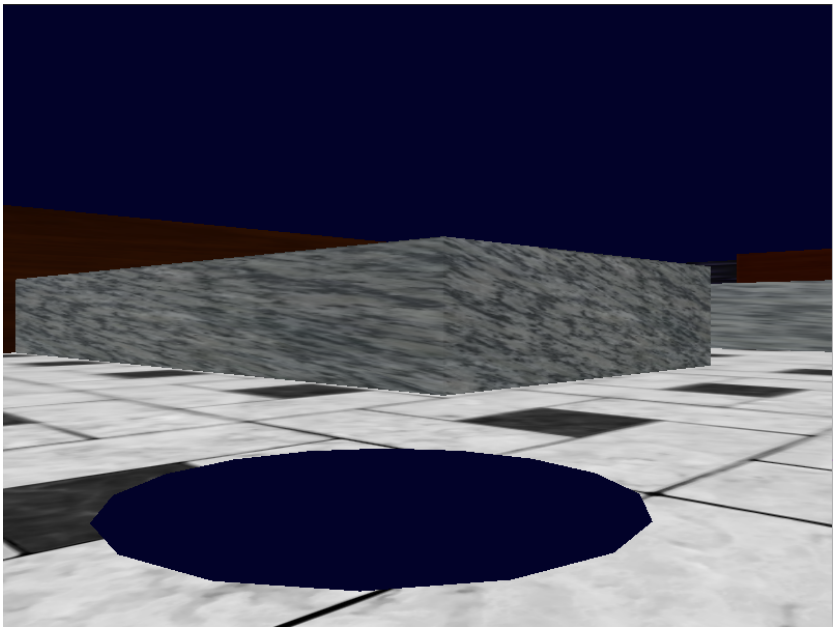

Figura 2.3: Alguns dos obstáculos do labirinto

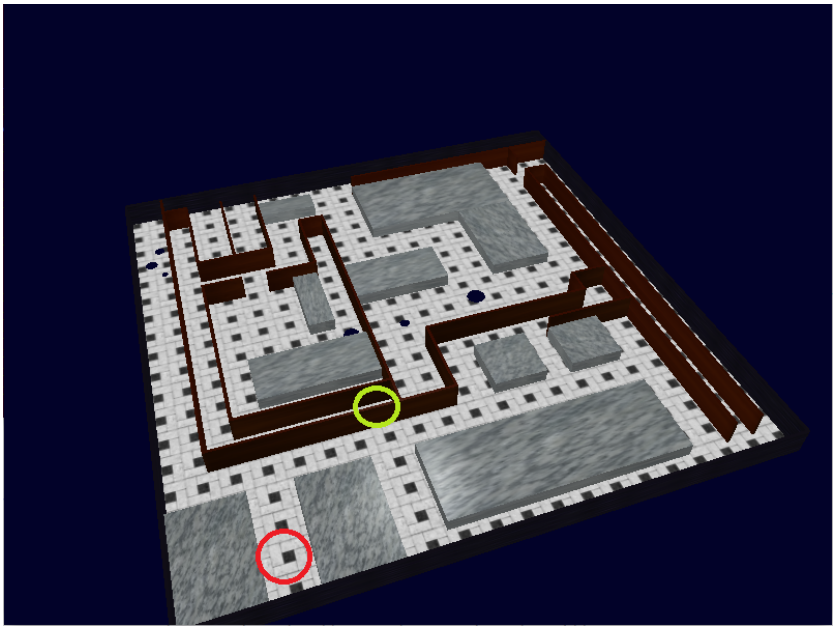

Figura 2.4: Nível 1 – Vermelho corresponde ao *spawn* e verde ao buraco vencedor

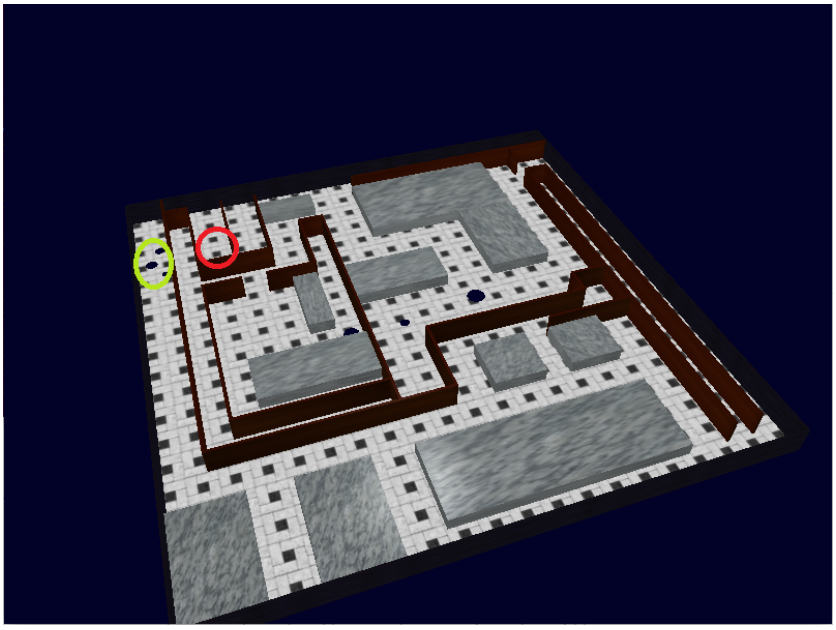

Figura 2.5: Nível 2 – Vermelho corresponde ao *spawn* e verde ao buraco vencedor

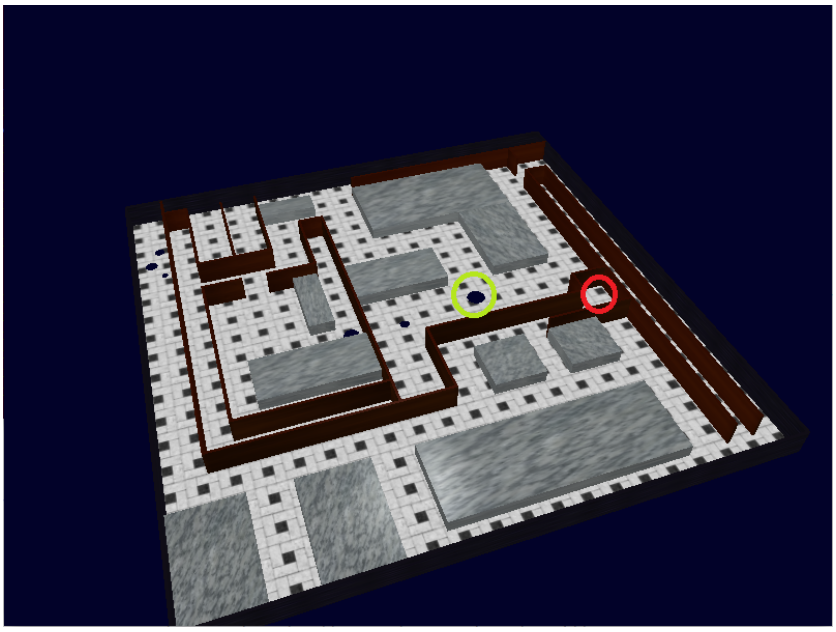

Figura 2.6: Nível 3 – Vermelho corresponde ao *spawn* e verde ao buraco vencedor

2.4 Escolhas de Implementação

Para modelar o labirinto, optámos por usar o *software Blender* e importámos o modelo para o *OpenGL* com um *ObjLoader* para termos acesso a toda a informação do modelo.

Em relação á parte da iluminação, criámos uma esfera para representar o foco de luz. Para conseguir dar diferentes cores aos objetos e texturas, tivemos de implementar novos *shaders*, tais como o *shaders* para as paredes internas, o *shaders* para as paredes externas e o *shaders* para os blocos de granito que se encontram espalhados pelo mapa. Estes *shaders* que desenvolvemos foram baseados nos *shaders* desenvolvidos nas aulas práticas.

Para a jogabilidade do utilizador, optámos por usar um movimento em primeira pessoa e a possibilidade de também controlar a direcção do seus movimentos com a câmera. Não é permitido ao jogador mover a câmera na vertical pois assim não consegue fazer batota ao ver por cima das paredes.

Como dificuldade, implementámos vários níveis em que o utilizador terá de percorrer o labirinto á procura do buraco vencedor. Neste tópico da dificuldade, também decidimos que o buraco vencedor não se deveria distinguir dos outros buracos pois dá uma outra vertente ao nosso labirinto que achámos interessante, ou seja, o utilizador não só terá de aprender o caminho do labirinto até ao buraco mas também lembrar-se de quais os buracos que já experimentou para conseguir encontrar o buraco vencedor por exclusão de partes.

2.5 Manual de Instalação

1. Fazer o *download* da pasta do projeto.
2. Inicializar a linha de comandas (cmd) e entrar na diretoria do projeto.
3. Compilar o projeto `g++ maze.cpp -lGL -lGLEW -lglfw`
4. Executar a aplicação `./a.out`

2.6 Manual de Utilização

1. Para se poder mover no labirinto basta usar as letras W, A, S, D.
2. Pode usar o rato para controlar a câmera.
3. O objectivo é encontrar o buraco vencedor, mas tenha cuidado, o buraco vencedor não é diferente dos outros, por isso, caso perca o jogo num certo buraco, convém lembrar-se desse buraco para não voltar a cair nele.

4. Assim que encontrar o buraco vencedor, será automaticamente passado para o nível seguinte e o buraco vencedor muda de sítio.

Capítulo 3

Conclusões e Trabalho Futuro

3.1 Reflexão Crítica

Numa visão geral estamos satisfeitos com o resultado do projeto, conseguimos implementar a maior parte das funcionalidades que desejávamos. Houve uma boa coordenação e comunicação entre todos os membros do grupo e todos fizeram as suas tarefas em devido prazo.

3.2 O que faltou fazer?

Os tópicos do nosso trabalho que faltaram fazer foram:

- Menus;
- Texto Gráfico;
- Pontuação.

3.3 Trabalho Futuro

Algumas ideias de trabalho futuro:

- Adicionar gravidade ao labirinto para um novo modo de jogo em que o jogador rodava o tabuleiro com o movimento do rato ou com as teclas movendo a bola;
- Implementar outro modo de visão em que o utilizador vê o tabuleiro de cima;

- Aumentar a dificuldade do labirinto substancialmente adicionando mais blocos, paredes interiores e buracos perdedores;
- Adicionar *Skybox*.

Capítulo 4

Bibliografia

- https://learnopengl.com/;
- http://www.di.ubi.pt/~agomes/cg/;
- https://docs.blender.org/manual/en/dev/;
- http://www.unmuseum.org/maze.htm.