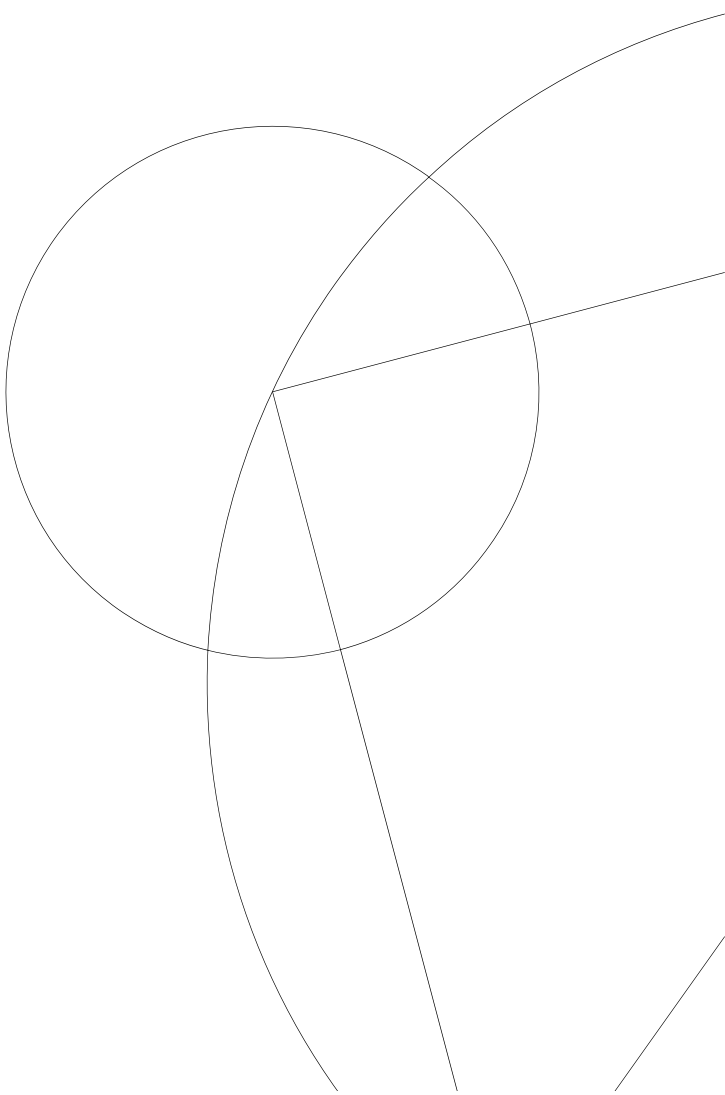
UNIVERSIDADE
BEIRA INTERIOR

Relatório do Projeto Final

Número do Projeto – 1

Projeto nº – CG-Maze

Jogo de labirinto 3D em OpenGL



Orientadores: ***Prof. Abel Gomes*** | ***Prof. João Dias***

7 de janeiro de 2026

Elementos do grupo

Nome	Nº mecanográfico
Dinis Ramos	49471
Filipe Mesquita	49701

Conteúdo

1 Introdução **4**

2 Objetivos **4**

3 Distribuição de tarefas (autoria) **4**

4 Arquitetura do jogo e fluxo de estados **4**

5 Geração automática do labirinto **5**
5.1 Estrutura de dados 5
5.2 Algoritmo 5

6 Colisões simples (grid-based) **6**

7 Movimento em primeira pessoa (FPS) e rato **7**

8 Texturas **8**

9 Áudio: som de passos **9**

10 Menu 2D: desenho e cliques **10**

11 Modos de jogo e pós-processamento "bêbado" **10**

12 Iluminação e lanterna (Spotlight) **11**
12.1 Motivação e integração no jogo 12
12.2 Parâmetros de iluminação 12
12.3 Envio de *uniforms* para o shader (lado CPU) 12
12.4 Cálculo do cone e iluminação (lado shader) 13
12.5 Controlo da lanterna pelo jogador (tecla `F`) 13
12.6 Nota técnica (limitação identificada) 14

13 Deteção de vitória **14**

14 Manual do utilizador (controlos) **14**

15 Validação e testes **14**

16 Conclusão **15**

17 Trabalho futuro **15**

1 Introdução

Este projeto consiste num jogo de labirinto em 3D desenvolvido em **C++** com **OpenGL** e **GLFW**, com navegação em primeira pessoa (FPS). O labirinto é gerado automaticamente e o jogador desloca-se com controlos típicos (WASD + rato/setas), com colisões simples baseadas numa grelha. Foram adicionadas funcionalidades de jogo e apresentação: **menu 2D**, **texturas**, **iluminação**, **lanterna (spotlight)**, **pós-processamento "bêbado"**, **som de passos**, **modos de dificuldade** e **ecrã de vitória**.

2 Objetivos

- Desenvolver um labirinto 3D jogável em primeira pessoa.
- Implementar geração automática do labirinto (diferentes tamanhos e dificuldades).
- Garantir que o jogador não atravessa paredes através de um sistema de colisões leve.
- Integrar UI 2D e feedback audiovisual (som e iluminação) para melhorar a experiência.

3 Distribuição de tarefas (autoria)

Tabela 1: Funcionalidades implementadas e autoria.

Funcionalidade	**Autor(es)**
Colisões simples (grid-based)	Dinis
Labirinto gerado automaticamente	Dinis
Câmara em primeira pessoa (FPS)	Filipe
Iluminação	Filipe
Texturas	Filipe
Modos de jogo (Fácil/Normal/Difícil)	Dinis (início), Filipe (final)
Spotlight (lanterna)	Filipe
Som de passos	Dinis
Menu inicial 2D	Dinis
Detetar chegada à saída (vitória)	Dinis
Modo bêbado	Dinis

4 Arquitetura do jogo e fluxo de estados

O programa usa uma máquina de estados simples (`GameState`) para separar menus do jogo 3D. Isto evita misturar lógica de UI com a renderização do labirinto e torna o ciclo principal mais limpo.

Listing 1: Estados do jogo e variáveis principais (excerto).

```
enum class GameState { MENU_MAIN, MENU_MODE, PLAYING, VICTORY };
GameState state = GameState::MENU_MAIN;

static int  gChoice = 2;          // 1 easy, 2 normal, 3 hard
static bool gDrunkMode = false;   // hard => true
```

No ciclo principal, quando o estado não é `PLAYING`, desenha-se apenas o UI 2D; caso contrário, desenha-se a cena 3D.

Listing 2: Decisão UI vs 3D no ciclo principal (excerto).

```
if (state != GameState::PLAYING)
{
    glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);

    glDisable(GL_DEPTH_TEST);
```

```
    glClear(GL_COLOR_BUFFER_BIT);

    uiShader.use();
    uiShader.setMat4("uProj", OrthoTopLeft((float)gWinW, (float)gWinH));

    Rect bg = {0, 0, (float)gWinW, (float)gWinH};
    DrawRectUI(uiShader, bg, texBg);

    if (state == GameState::MENU_MAIN) {
        DrawRectUI(uiShader, btnStart.r, btnStart.tex);
        DrawRectUI(uiShader, btnExitMain.r, btnExitMain.tex);
    } else if (state == GameState::MENU_MODE) {
        DrawRectUI(uiShader, btnEasy.r, btnEasy.tex);
        DrawRectUI(uiShader, btnNormal.r, btnNormal.tex);
        DrawRectUI(uiShader, btnHard.r, btnHard.tex);
        DrawRectUI(uiShader, btnExitMode.r, btnExitMode.tex);
    } else if (state == GameState::VICTORY) {
        DrawRectUI(uiShader, bg, texVictory);
        DrawRectUI(uiShader, btnExitVictory.r, btnExitVictory.tex);
    }

    glfwSwapBuffers(window);
    glfwPollEvents();
    continue;
}
```

5 Geração automática do labirinto

5.1 Estrutura de dados

O labirinto é uma matriz `maze[z][x]`: `1` representa parede e `0` representa caminho. Isto facilita a renderização (percorre-se a matriz e desenham-se paredes) e as colisões (testam-se células vizinhas).

5.2 Algoritmo

Foi usado *backtracking* recursivo: parte-se de uma célula inicial e abrem-se caminhos saltando 2 células de cada vez, mantendo paredes entre corredores. As direções são baralhadas para gerar labirintos diferentes.

Listing 3: Carving recursivo com direções aleatórias (excerto).

```
void carveMaze(int x, int z)
{
    static int dirs[4][2] = { {1,0}, {-1,0}, {0,1}, {0,-1} };

    for (int i = 0; i < 4; i++) {
        int r = rand() % 4;
        std::swap(dirs[i], dirs[r]);
    }

    for (int i = 0; i < 4; i++)
    {
        int nx = x + dirs[i][0] * 2;
        int nz = z + dirs[i][1] * 2;

        if (nx > 0 && nz > 0 && nx < MAZE_W - 1 && nz < MAZE_H - 1)
        {
            if (maze[nz][nx] == 1)
            {
                maze[z + dirs[i][1]][x + dirs[i][0]] = 0;
                maze[nz][nx] = 0;
                carveMaze(nx, nz);
```

```
            }
        }
    }
}
```

Listing 4: Inicialização da grelha e criação de entrada/saída (excerto).

```
void generateMaze()
{
    maze.resize(MAZE_H, std::vector<int>(MAZE_W, 1));

    maze[1][1] = 0;
    carveMaze(1, 1);

    for (int x = 0; x < MAZE_W; x++) {
        maze[0][x] = 1;
        maze[MAZE_H - 1][x] = 1;
    }
    for (int z = 0; z < MAZE_H; z++) {
        maze[z][0] = 1;
        maze[z][MAZE_W - 1] = 1;
    }

    maze[1][0] = 0; // entrada
    int exitZ = MAZE_H - 2;
    maze[exitZ][MAZE_W - 2] = 0;
    maze[exitZ][MAZE_W - 1] = 0; // sa da
}
```

6 Colisões simples (grid-based)

A colisão foi simplificada para ser eficiente: o jogador é aproximado por um círculo (raio `PLAYER_RADIUS`) e as paredes são caixas alinhadas à grelha (AABB). O teste verifica apenas as células próximas (3x3 à volta da célula do jogador), reduzindo o custo.

Listing 5: Teste de colisão círculo vs AABB em células vizinhas (excerto).

```
static inline float clampf(float v, float a, float b){
    return (v < a) ? a : (v > b) ? b : v;
}

bool checkCollision(glm::vec3 pos)
{
    const float r = PLAYER_RADIUS;
    const float r2 = r * r;

    int cx = (int)floor(pos.x / CELL_SIZE);
    int cz = (int)floor(pos.z / CELL_SIZE);

    for (int dz = -1; dz <= 1; dz++)
    for (int dx = -1; dx <= 1; dx++)
    {
        int mx = cx + dx, mz = cz + dz;
        if (mx < 0 || mx >= MAZE_W || mz < 0 || mz >= MAZE_H) continue;
        if (maze[mz][mx] != 1) continue; // s   paredes

        float minX = mx * CELL_SIZE, maxX = minX + CELL_SIZE;
        float minZ = mz * CELL_SIZE, maxZ = minZ + CELL_SIZE;

        float closestX = clampf(pos.x, minX, maxX);
        float closestZ = clampf(pos.z, minZ, maxZ);

```

```
26          float dxp = pos.x - closestX;
27          float dzp = pos.z - closestZ;
28
29          if ((dxp*dxp + dzp*dzp) < r2) return true;
30      }
31      return false;
32  }
```

7 Movimento em primeira pessoa (FPS) e rato

O movimento usa WASD e aplica *sliding*: tenta-se primeiro mover em X e depois em Z. Isto permite "raspar" paredes em vez de ficar preso em cantos.

Listing 6: Movimento com *sliding* e controlo do som de passos (excerto).

```
glm::vec3 oldPos = camera.Position;

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

glm::vec3 target = camera.Position;

// X
camera.Position = glm::vec3(target.x, oldPos.y, oldPos.z);
if (checkCollision(camera.Position)) camera.Position.x = oldPos.x;

// Z
camera.Position = glm::vec3(camera.Position.x, oldPos.y, target.z);
if (checkCollision(camera.Position)) camera.Position.z = oldPos.z;

float moved = glm::length(glm::vec2(camera.Position.x-oldPos.x, camera.
    Position.z-oldPos.z));
if (moved > 0.0001f) startFootsteps();
else stopFootsteps();
```

O rato atualiza a orientação da câmara (yaw/pitch), recorrendo á função ProcessMouseMovement() do camera.h.

Listing 7: Atualização da câmara com o rato e recentragem (excerto).

```
void mouse_callback(GLFWwindow *window, double xpos, double ypos)
{
    if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
    }

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

    lastX = xpos;
    lastY = ypos;

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

```
17
18      glfwSetCursorPos(window, centerX, centerY);
19  }
```

8 Texturas

As texturas são carregadas com `stb_image` e configuradas com `glTexParameteri` (wrapping e filtering). No UI, usa-se RGBA para suportar transparência nos botões.

Listing 8: Carregamento de textura RGBA para UI (excerto).

```
1   void prepareTextures()
2   {
3       // Wall
4       std::cout << "Loading wall texture...\n";
5
6       glGenTextures(1, &wallTexture);
7       glActiveTexture(GL_TEXTURE0);
8       glBindTexture(GL_TEXTURE_2D, wallTexture);
9
10      // Wrapping / filtering
11      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
12      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
13      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
            GL_LINEAR_MIPMAP_LINEAR);
14      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
15
16      // Flip (Blender compatibility)
17      stbi_set_flip_vertically_on_load(true);
18
19      wallData = stbi_load(wall_texture_File, &wallWidth, &wallHeight, &
            wallNrChannels, 0);
20
21      if (wallData)
22      {
23          GLenum format;
24          if (wallNrChannels == 1)
25              format = GL_RED;
26          else if (wallNrChannels == 3)
27              format = GL_RGB;
28          else if (wallNrChannels == 4)
29              format = GL_RGBA;
30
31          glTexImage2D(GL_TEXTURE_2D, 0, format, wallWidth, wallHeight, 0,
                format, GL_UNSIGNED_BYTE, wallData);
32          glGenerateMipmap(GL_TEXTURE_2D);
33      }
34      else
35      {
36          std::cout << "Failed to load wall texture\n";
37      }
38
39      stbi_image_free(wallData);
40
41      // Floor
42      std::cout << "Loading floor texture...\n";
43
44      glGenTextures(1, &floorTexture);
45      glActiveTexture(GL_TEXTURE1);
46      glBindTexture(GL_TEXTURE_2D, floorTexture);
47
48      // Wrapping / filtering
49      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

```
50      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
51      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
            GL_LINEAR_MIPMAP_LINEAR);
52      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
53
54      // Flip (Blender compatibility)
55      stbi_set_flip_vertically_on_load(true);
56
57      floorData = stbi_load(floor_texture_File, &floorWidth, &floorHeight, &
            floorNrChannels, 0);
58
59      if (floorData)
60      {
61          GLenum format;
62          if (floorNrChannels == 1)
63              format = GL_RED;
64          else if (floorNrChannels == 3)
65              format = GL_RGB;
66          else if (floorNrChannels == 4)
67              format = GL_RGBA;
68
69          glTexImage2D(GL_TEXTURE_2D, 0, format, floorWidth, floorHeight, 0,
                format, GL_UNSIGNED_BYTE, floorData);
70          glGenerateMipmap(GL_TEXTURE_2D);
71      }
72      else
73      {
74          std::cout << "Failed to load floor texture\n";
75      }
76
77      stbi_image_free(floorData);
78  }
```

9 Áudio: som de passos

O som de passos é reproduzido em loop com OpenAL. O buffer é preenchido com samples carregados via libsndfile. A source é ativada quando há movimento efetivo e parada quando o jogador pára.

Listing 9: Inicialização do OpenAL e passos em loop (excerto).

```
1   bool initAudio()
2   {
3       alDevice = alcOpenDevice(nullptr);
4       if (!alDevice) return false;
5
6       alContext = alcCreateContext(alDevice, nullptr);
7       if (!alContext || !alcMakeContextCurrent(alContext)) return false;
8
9       if (!loadWavToOpenAL("./sounds/step.wav", stepBuffer)) return false;
10
11      alGenSources(1, &stepSource);
12      alSourcei(stepSource, AL_BUFFER, stepBuffer);
13      alSourcef(stepSource, AL_GAIN, 1.0f);
14      alSourcei(stepSource, AL_LOOPING, AL_TRUE);
15
16      return true;
17  }
18
19  void startFootsteps(){ if(!isStepPlaying){ alSourcePlay(stepSource);
        isStepPlaying=true; } }
20  void stopFootsteps(){  if(isStepPlaying){  alSourceStop(stepSource);
        isStepPlaying=false; } }
```

10 Menu 2D: desenho e cliques

O menu usa retângulos (`Rect`) e botões (`Button`) com teste de interseção baseado na posição do rato. A renderização é feita com uma projeção ortográfica com origem no topo-esquerdo.

Listing 10: Estruturas do UI e projeção ortográfica topo-esquerdo (excerto).

```
struct Rect { float x, y, w, h; };

struct Button {
    Rect r;
    GLuint tex = 0;
    bool contains(float mx, float my) const {
        return mx >= r.x && mx <= (r.x + r.w) && my >= r.y && my <= (r.y + r.
            h);
    }
};

glm::mat4 OrthoTopLeft(float w, float h)
{
    return glm::ortho(0.0f, w, h, 0.0f, -1.0f, 1.0f);
}
```

Listing 11: Gestão de cliques e transição de estados (excerto).

```
void mouse_button_callback(GLFWwindow* window, int button, int action, int
    mods)
{
    if (button != GLFW_MOUSE_BUTTON_LEFT || action != GLFW_PRESS) return;

    if (state == GameState::MENU_MAIN) {
        if (btnStart.contains(gMouseX, gMouseY)) {
            state = GameState::MENU_MODE;
            BuildMenuLayout();
        } else if (btnExitMain.contains(gMouseX, gMouseY)) {
            glfwSetWindowShouldClose(window, true);
        }
    }
    else if (state == GameState::MENU_MODE) {
        if (btnEasy.contains(gMouseX, gMouseY)) StartGame(1, window);
        else if (btnNormal.contains(gMouseX, gMouseY)) StartGame(2, window);
        else if (btnHard.contains(gMouseX, gMouseY)) StartGame(3, window);
        else if (btnExitMode.contains(gMouseX, gMouseY)) {
            state = GameState::MENU_MAIN;
            BuildMenuLayout();
        }
    }
    else if (state == GameState::VICTORY) {
        if (btnExitVictory.contains(gMouseX, gMouseY)) {
            glfwSetWindowShouldClose(window, true);
        }
    }
}
```

11 Modos de jogo e pós-processamento “bêbado”

Existem três modos:

- **Fácil**: labirinto pequeno (menos complexidade).
- **Normal**: labirinto maior com lanterna (spotlight).
- **Difícil**: labirinto maior com filtro de pós-processamento (efeito “bêbado”) e lanterna.

O efeito de pós-processamento usa um FBO: renderiza a cena para uma textura e depois desenha um quad de ecrã inteiro com um shader que distorce UVs ao longo do tempo.

Listing 12: Criação de FBO para pós-processamento (excerto).

```
void createSceneFBO(int w, int h)
{
    if (sceneFBO == 0) glGenFramebuffers(1, &sceneFBO);
    glBindFramebuffer(GL_FRAMEBUFFER, sceneFBO);

    if (sceneColorTex == 0) glGenTextures(1, &sceneColorTex);
    glBindTexture(GL_TEXTURE_2D, sceneColorTex);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, w, h, 0, GL_RGB, GL_UNSIGNED_BYTE,
        nullptr);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
        GL_TEXTURE_2D, sceneColorTex, 0);

    if (sceneRBO == 0) glGenRenderbuffers(1, &sceneRBO);
    glBindRenderbuffer(GL_RENDERBUFFER, sceneRBO);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, w, h);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
        GL_RENDERBUFFER, sceneRBO);

    glBindFramebuffer(GL_FRAMEBUFFER, 0);
}
```

Listing 13: Shader de pós-processamento "bêbado" (exemplo simplificado).

```
#version 330 core
out vec4 FragColor;
in vec2 TexCoord;

uniform sampler2D sceneTex;
uniform float time;
uniform float intensity;

void main()
{
    vec2 uv = TexCoord;
    uv.x += sin(uv.y * 10.0 + time) * 0.01 * intensity;
    uv.y += cos(uv.x * 10.0 + time) * 0.01 * intensity;

    FragColor = texture(sceneTex, uv);
}
```

12 Iluminação e lanterna (Spotlight)

A iluminação da cena é calculada em *shaders* (GLSL), permitindo controlar a aparência das paredes e do chão de forma eficiente e consistente. O modelo implementado segue uma abordagem clássica (Phong/Blinn-Phong simplificado), combinando:

- **componente ambiente** (luz mínima global para evitar preto absoluto);
- **componente difusa** (depende do ângulo entre a normal e a direção da luz);
- **atenuação** com a distância (*falloff*), para simular uma fonte realista;
- **spotlight** (lanterna) com cone de iluminação e transição suave entre interior/exterior.

12.1 Motivação e integração no jogo

A lanterna aumenta a legibilidade do labirinto em modos mais difíceis e melhora a atmosfera do jogo, limitando a visibilidade a uma zona em frente ao jogador. No nosso projeto:

- no **modo Fácil** a lanterna é desativada (`flashlightMode = false`);
- no **modo Normal/Difícil** a lanterna está disponível;
- o jogador pode **ligar/desligar** durante o jogo (tecla `F`).

12.2 Parâmetros de iluminação

Os parâmetros principais usados no motor foram:

Tabela 2: Parâmetros principais da lanterna e iluminação.

Parâmetro	Significado
`flashlightMode`	Indica se a lanterna está disponível (modo de jogo)
`flashlightOn`	Lanterna ligada ou desligada pelo jogador
`cutOff`	Ângulo interno do cone do spotlight (em radianos, luz mais intensa)
`outerCutOff`	Ângulo externo do cone do spotlight (queda suave da luz)
`constant / linear / quadratic`	Fatores de atenuação da luz com a distância
`lightColor`	Cor da luz da lanterna
`lightPos`	Posição da lanterna no mundo
`lightDir`	Direção do spotlight da lanterna
`viewPos`	Posição da câmara/jogador

12.3 Envio de *uniforms* para o shader (lado CPU)

A lanterna é uma luz "presa" ao jogador: a posição da luz é a posição da câmara e a direção do cone é o vetor `Front` (para onde o jogador olha). O excerto seguinte mostra a configuração típica dos *uniforms* antes de desenhar a cena:

Listing 14: Configuração da lanterna (spotlight) no shader de iluminação (excerto).

```
lightingShader.use();

// Cor da luz
lightingShader.setVec3("lightColor", 1.0f, 1.0f, 1.0f);

// Posi  o e dire  o da lanterna (presa    c mara)
lightingShader.setVec3("lightPos", camera.Position);
lightingShader.setVec3("lightDir", camera.Front);
lightingShader.setVec3("viewPos", camera.Position);

// Disponibilidade e estado da lanterna
lightingShader.setBool("flashlightMode", flashlightMode);
lightingShader.setBool("flashlightOn", flashlightOn);

// Atenua  o da luz com a dist ncia
lightingShader.setFloat("constant",  1.0f);
lightingShader.setFloat("linear",    0.09f);
lightingShader.setFloat("quadratic", 0.032f);

// Cutoffs do spotlight em cosseno (definem o cone de ilumina  o)
lightingShader.setFloat("cutOff",      cos(glm::radians(innerCutOff)));
lightingShader.setFloat("outerCutOff", cos(glm::radians(outerCutOff)));
```

12.4 Cálculo do cone e iluminação (lado shader)

O spotlight da lanterna é calculado comparando o ângulo entre:

- a direção da luz para o fragmento ($\vec{L} = \text{lightPos} - \text{FragPos}$);
- a direção principal do foco ($\vec{D} = -\text{lightDir}$).

Em vez de calcular o ângulo com arccos, usa-se o produto escalar e compara-se com $\cos(\theta)$, o que é mais barato:

$$\text{theta} = \langle \hat{L}, \hat{D} \rangle, \quad \text{intensity} = \text{clamp}\left(\frac{\text{theta} - \text{outerCutOff}}{\text{cutOff} - \text{outerCutOff}}, 0, 1 \right).$$

Assim, existe transição suave entre o cone interior e exterior. A intensidade final do spotlight também é multiplicada pela atenuação com a distância:

$$\text{attenuation} = \frac{1}{\text{constant} + \text{linear} \cdot d + \text{quadratic} \cdot d^2}, \quad d = \|\text{lightPos} - \text{FragPos}\|.$$

O cálculo final do spotlight combina diffuse, specular e a intensidade do cone com a atenuação:

Listing 15: Trecho do spotlight com transição suave e atenuação (GLSL).

```
vec3 lightDirection = normalize(lightPos - FragPos);
float theta = dot(lightDirection, normalize(-lightDir));

float intensity = clamp((theta - outerCutOff) / (cutOff - outerCutOff), 0.0,
    1.0);

float distance = length(lightPos - FragPos);
float attenuation = 1.0 / (constant + linear * distance + quadratic *
    distance * distance);

float diff = max(dot(norm, lightDirection), 0.0);
vec3 diffuse = diff * lightColor;

vec3 viewDir = normalize(viewPos - FragPos);
vec3 reflectDir = reflect(-lightDirection, norm);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 16.0);
vec3 specular = 0.08 * spec * lightColor;

vec3 lighting = (diffuse + specular) * intensity * attenuation;

FragColor = vec4(ambient + lighting * texColor, 1.0);
```

12.5 Controlo da lanterna pelo jogador (tecla `F`)

O jogador pode ligar ou desligar a lanterna pressionando a tecla `F`. Para evitar que um único pressionar alterne várias vezes no mesmo quadro, é aplicado **debounce**, ou seja, deteta-se a transição "não premida → premida".

Listing 16: Alternar a lanterna com debounce (excerto).

```
static bool fPressedLastFrame = false;

// Detetar pressionamento da tecla F
bool fPressed = (glfwGetKey(window, GLFW_KEY_F) == GLFW_PRESS);

// Alternar estado da lanterna apenas quando a tecla passa de n o premida
    para premida
if (fPressed && !fPressedLastFrame)
    flashlightOn = !flashlightOn;

// Guardar estado da tecla para o pr ximo quadro
```

```
fPressedLastFrame = fPressed;

// No shader, se a lanterna estiver desligada, apenas se aplica luz ambiente
if (!flashlightOn)
{
    FragColor = vec4(ambient, 1.0);
    return;
}
```

12.6 Nota técnica (limitação identificada)

Durante testes foi identificado um comportamento anómalo na iluminação da lanterna (mencionado na secção de trabalho futuro).

13 Deteção de vitória

A vitória é detetada ao comparar a célula do jogador com a zona de saída do labirinto. Ao atingir a saída, o estado muda para `VICTORY` e o programa passa a desenhar o ecrã final.

Listing 17: Exemplo de deteção de vitória (excerto).

```
int px = (int)floor(camera.Position.x / CELL_SIZE);
int pz = (int)floor(camera.Position.z / CELL_SIZE);

if (pz == MAZE_H - 2 && px == MAZE_W - 1)
{
    stopFootsteps();
    state = GameState::VICTORY;
    BuildMenuLayout();
    continue;
}
```

14 Manual do utilizador (controlos)

- **W/A/S/D**: movimentação (frente/esquerda/trás/direita)
- **Rato**: controlar a orientação da câmera (de forma limitada, caso o progrma esteja a correr dentro do WSL)
- **F**: ligar/desligar lanterna
- **Setas**: controlar a orientação da câmera
- **M**: aumentar sensibilidade da câmera com as setas
- **N**: diminuir a sensibilidade da câmera com as setas
- **ESC**: sair da aplicação

15 Validação e testes

Foram realizados testes funcionais para confirmar o comportamento do jogo:

- **Geração do labirinto**: confirmação visual de paredes/corredores e existência de saída.
    1. Foi encontrado um *bug* que faz com que a saída do labirinto fique fechada por blocos de parede que não deviam existir.
- **Colisões**: tentativa de atravessar paredes em diferentes ângulos; verificação do *sliding*.
- **Lanterna/iluminação**: verificação do cone de luz e alternância com `F` (debounce).

1. Foi encontrado um *bug* que faz com que as paredes cujas normais estão orientadas para Z e -Z não sejam iluminadas corretamente.

- **UI 2D**: testes de clique em botões e transições entre estados.
- **Som**: validação de início/paragem do loop de passos (apenas quando há movimento real).

16 Conclusão

O projeto cumpre os objetivos propostos: permite jogar um labirinto 3D em primeira pessoa com geração automática e colisões simples. A presença de UI 2D, áudio, texturas e efeitos de shader melhora a apresentação e a jogabilidade, e os modos de dificuldade tornam o jogo mais variado.

17 Trabalho futuro

- Melhorar a zona de vitória com um *trigger* dedicado e feedback visual no fim.
- Aplicar outras componentes às texturas (por exemplo, NormalMap).
- Adicionar temporizador e pontuação por dificuldade.
- Introduzir mais variedade visual (decoração, mais texturas, etc.).
- Resolver *bug* na iluminação das paredes.
- Resolver *bug* na geração do labirinto.