



# 《计算机图形学》

# 项目报告

## 期末大作业

### 基于 OpenGL 的三维峡谷探险迷宫设计



学院名称：计算机学院  
专业名称：21 级计算机科学与技术  
组长姓名：21312028 卢敦杰  
组员姓名：21307077 凌国明  
组员姓名：21307260 方宇豪

# 基于 OpenGL 的三维峡谷探险迷宫设计

卢敦杰<sup>1</sup> 凌国明<sup>2</sup> 方宇豪<sup>3</sup>

## 摘要

该技术报告从多个方面对本小组所设计的游戏“基于 OpenGL 的三维峡谷探险迷宫设计”进行了介绍：

游戏概述，技术规格，重要技术讲解，游戏实现效果展示，测试预评估，讨论反思等等…

## 1. 引言

本次期末大作业，我们小组选定的主题为“基于 OpenGL 的三维峡谷探险迷宫设计”，项目的框架基于 Qt 的 OpenGL 模块。

本次设计的 3D 迷宫游戏主要通过二维矩阵所表示的迷宫图，来构建三维立体场景。我们为系统设计了一架摄像机，来模拟用户在迷宫中的第一视角，用户可以通过鼠标、键盘控制自己在 3D 迷宫中的移动。游戏也很好地设计了碰撞检测，给玩家以一种身临其境的感觉。

游戏中设置了各种各样的元素，包括可以翻阅的障碍围墙，岩浆陷阱，以及得分机关。用户可以很好地在游戏的操作过程中与这些元素进行交互。

游戏的场景设计十分用心，峡谷与地板的贴图都是份精细，此外还有藤蔓的动态纹理。我们完成了天空盒的设计并且引入了自己的光照模型，在天空盒动态移动的时候，光照阴影也会随之移动。

总体来说，游戏界面精细美观，可操作性和交互性强。

## 2. 游戏概述

2.1 游戏概念. 玩家会置身于一个峡谷之中，峡谷为一个迷宫。玩家需要根据右下角的小地图确定终

点的位置与自己的路线。



图 1. 游戏画面 1



图 2. 游戏画面 2

2.3 用户的交互. 玩家通过“表一”中所展示的按键进行控制，此外也可以通过移动鼠标进行视角的变换。

按键事件	事件响应
W	向前方移动
S	向后方移动
A	向左方移动
D	向右方移动
I	视角向上移动
K	视角向下移动
J	视角向左移动
L	视角向右移动
Space	垂直方向跳跃
F	飞到高处

表 1. 按键事件及其响应

## 3. 技术规格

3.1 技术框架. 本游戏采用了 OpenGL 图形库和 C++ 编程语言，这是由于它们的高性能和灵活性。OpenGL 提供了一组功能强大的图形渲染指令，允许开发者直接与图形硬件通信，而无需过多关注底层硬件的具体实现。C++ 作为一门高效率的系统编程语言，其面向对象的特性使得游戏逻辑的构建和维护变得更加直观和模块化。

游戏的三维场景和视觉效果是通过 OpenGL 的各种功能实现的。我们使用了 OpenGL 的着色器语言 GLSL 来编写自定义的顶点和片段着色器，实现了复杂的光照模型和纹理映射。这使得游戏世界具有逼真的光影效果和丰富的视觉层次。

为了模拟真实世界的物理反应，我们设置了一个全局的 G 值以实现良好的下坠效果，并且完成了前后左右上等五个方向的物理碰撞检测。

**3.2. 系统架构.** 系统架构概述我们的三维第一人称峡谷迷宫游戏由多个核心组件构成，每个组件都负责游戏的一个特定功能。这些组件紧密协作，共同提供了一个流畅、互动的游戏体验。下面是各个组件的详细介绍：

迷宫生成：‘maze.cpp’ 是迷宫生成的核心模块。它负责创建迷宫的布局，确保每次游戏开始时都能生成一个唯一的迷宫。这个模块使用复杂的算法来确保迷宫的多样性和可玩性，同时还要保持合理的难度，让玩家既感到挑战又不会感到沮丧。

视角控制：‘camera.cpp’ 管理玩家的视角移动。在一个第一人称游戏中，视角的平滑和响应性是至关重要的，因此该模块包括了一套复杂的控制系统，允许玩家自由地查看环境，提供了沉浸式的游戏体验。

渲染管理：‘myglwidget.cpp’ 负责游戏的大部分渲染工作。它使用 OpenGL 来绘制游戏世界，包括迷宫的墙壁、天空、地面以及其他游戏元素。该模块确保了游戏具有高质量的视觉效果，并且通过有效的渲染管线优化了性能。

玩家控制：‘player.cpp’ 模块让玩家能够在迷宫中移动。它不仅响应玩家的输入来执行前后左右的移动，并且还包括了碰撞检测，防止玩家穿过墙壁或其他不应该通过的区域。

纹理处理：‘texture.cpp’ 用于加载和管理游戏中使用的纹理。纹理对于提高游戏的视觉质量至关重要，因此这个模块包含了从文件中读取纹理，应用纹理映射等功能，以增强游戏的真实感和沉浸感。

三维向量转换：‘vec3.cpp’ 处理三维空间中的向量运算。

## 4. 重要技术的讲解

**4.1 迷宫二维矩阵的产生.** 本游戏使用二维矩阵来表征迷宫，首先要初始化这个二维矩阵，确保迷宫是有效的。生成迷宫的关键要求是确保存在一条从起点到终点的可达路径，并保持墙体密度适中，以避免创建多个不可达的空房间，这有助于保持迷宫的可玩性和有效面积。

迷宫可以视为一个图，因此可以使用 Kruskal 随机生成算法来生成迷宫。Kruskal 算法通过连通图中的点来构造最小生成树，从而确保任意两点之间存在唯一路径。该算法的运行过程类似于不断拆除墙体，将相邻的空房间相连。为了生成不同的迷宫，应将所有墙体视为权重相同的边，并在拆除墙体前打乱它们的顺序。与 BFS 和 Prim 算法不同，Kruskal 算法在完成前可能存在多个不连通的分支。

```

1 class MazeGeneratorhhh {
2 public:
3     MazeGeneratorhhh(int rows, int cols) : rows(rows), cols
4         (cols), uf(rows * cols) {
5         srand(static_cast<unsigned>(time(nullptr)));
6         initializeEdges();
7         generateMaze();
8     }
9     std::vector<std::vector<int>> getMazeMatrix() {
10        std::vector<std::vector<int>> mazeMatrix(rows * 2 +
11            1, std::vector<int>(cols * 2 + 1, 1));
12        for (auto& edge : connectedEdges) {
13            int x1 = edge.first / cols;
14            int y1 = edge.first % cols;
15            int x2 = edge.second / cols;
16            int y2 = edge.second % cols;
17            mazeMatrix[2 * x1 + 1][2 * y1 + 1] = 0;
18            mazeMatrix[2 * x2 + 1][2 * y2 + 1] = 0;
19            mazeMatrix[x1 + x2 + 1][y1 + y2 + 1] = 0;
20        }
21    return mazeMatrix;
22 }
23 private:
24     int rows, cols;
25     UnionFindxxx uf;
26     std::vector<std::pair<int, int>> edges;
27     std::vector<std::pair<int, int>> connectedEdges;
28     void initializeEdges() {
29         for (int i = 0; i < rows; ++i) {
30             for (int j = 0; j < cols; ++j) {
31                 if (i > 0) edges.emplace_back(i * cols + j,
32                     (i - 1) * cols + j);
33                 if (j > 0) edges.emplace_back(i * cols + j,
34                     i * cols + j - 1);
35             }
36         }
37         void generateMaze() {
38             std::random_shuffle(edges.begin(), edges.end());
39             for (const auto& edge : edges) {
40                 int cell1 = edge.first;
41                 int cell2 = edge.second;
42                 if (uf.find(cell1) != uf.find(cell2)) {
43                     uf.unite(cell1, cell2);
44                     connectedEdges.push_back(edge);
45                 }
46             }
47         }
48     };
49 }
```

Listing 1. 迷宫生成-kruskal

以上是 Kruskal 算法的实现。先初始化迷宫，将任意两个空房间之间的墙体作为边加入 edges。其中 edges 代表可以拆除墙的位置，connectedEdges 代表已拆除墙的位置。然后打乱墙体顺序即可。

以下是迷宫生成，并查集的实现。

```

1 class UnionFindxxx {
2 public:
3     UnionFindxxx(int size) : parent(size), rank(size, 1) {
4         std::iota(parent.begin(), parent.end(), 0); // 初
5             始化每个元素的父节点为自身
6     }
7     int find(int x) {
8         if (x != parent[x]) {
9             parent[x] = find(parent[x]); // 路径压缩
10        }
11        return parent[x];
12    }
13    void unite(int x, int y) {
14        int rootX = find(x);
15        int rootY = find(y);
16        if (rootX != rootY) {
17            if (rank[rootX] > rank[rootY]) {
18                parent[rootY] = rootX;
19            } else if (rank[rootX] < rank[rootY]) {
20                parent[rootX] = rootY;
21            } else {
22                parent[rootY] = rootX;
23                rank[rootX]++;
24            }
25        }
26    }
27    private:
28        std::vector<int> parent;
29        std::vector<int> rank;
30    };
31 }
```

Listing 2. 迷宫生成-并查集

**4.2 同时处理多键输入.** 如果仅仅使用 opengl 中内置的 qkeyevent，那么一次只能输入一个按键，操控起来十分麻烦，因此需要进行多键输入的支持：

```

1 void MyGLWidget::keyPressEvent(QKeyEvent* e)
2 {
3     keyStates[e->key()] = true;
4     updatespacestate();
5 }
6 void MyGLWidget::keyReleaseEvent(QKeyEvent* event)
7 {
8     keyStates[event->key()] = false;
9     updatespacestate();
10}
11 void MyGLWidget::updatespacestate(void)
12 {
13     //Key Control
14     if (keyStates[Qt::Key_Dots])
15     if (keyStates[Qt::Key_Period])\dots
16 }
```

Listing 3. 实现多键输入

keyPressEvent 和 keyReleaseEvent 函数处理键盘按键的按下和释放事件。当按键按下时，对应键的状态设置为 true，表示激活；当按键释放时，状态设置为 false，表示非激活。keyStates 是一个映射（可能是一个字典或哈希表），它将每个按键映射到其当前状态（激活或非激活）。updatespacestate 函数是游戏逻辑的核心，它根据当前激活的按键更新玩家的状态。这个函数会检查 keyStates 映射，以确定哪些键当前被激活，并执行相应的行动。

**4.3 跳跃效果与下坠效果的实现.** 在我们的整体程序中，经过 5ms 会渲染一次 paintgl，所以我们只

需要根据时间计算好垂直轴 y 的位置即可实现跳跃与下坠，代码如下：

```

1 void MyGLWidget::updateJumpState() {
2     if (isJumping && !player->isFixed()) {
3         float currentTime = GetCurrentTimeInSeconds();
4         float timeSinceJump = currentTime - jumpStartTime;
5         float jumpHeight = jumpInitialHeight + jumpVelocity *
6             timeSinceJump - 0.5f * gravity * timeSinceJump *
7             timeSinceJump;
8         // 计算当前位置对应的块，从而实现 z 轴方向的碰撞检
9         int block_x = int(player->camera->position.z *
10             player->invWallSize + 0.5f);
11         int block_y = int(player->camera->position.x *
12             player->invWallSize + 0.5f);
13         // 检查是否结束跳跃（踩到当前块的上表面则结束跳
14         // 跃）（碰到迷宫地图上边界也结束跳跃）
15         if (jumpHeight <= player->maze->blockHeight[block_x]
16             [block_y] + 1.6f * player->maze->wallSize || jumpHeight
17             >= jumpPeakHeight) {
18             isJumping = false;
19             jumpHeight = player->maze->blockHeight[block_x]
20                 [block_y] + 1.6f * player->maze->
21                 wallSize; // 重置高度，位于当前块的上表面
22             jumpInitialHeight = jumpHeight; // 下次起跳高
23             fallInitialHeight = jumpHeight; // 下次坠落高
24             fallHeight = jumpHeight; // 当前块的上表面高度
25         }
26         // 更新摄像机或玩家的垂直位置
27         // 假设摄像机或玩家的位置是 camera->position.y
28         player->camera->position.y = jumpHeight;
29     }
30 }
```

Listing 4. 跳跃的实现

当玩家处于跳跃状态且不在固定位置时，该函数被调用。计算自跳跃开始以来经过的时间。使用基本的物理方程计算跳跃高度，其中考虑了初始跳跃高度、跳跃速度、时间和重力加速度。检查玩家是否到达了预设的跳跃顶点或者触碰到迷宫的某个块的上表面，这将结束跳跃过程。如果结束跳跃，将跳跃高度重置为当前块的上表面高度，并更新相关状态，准备下一次跳跃或下坠。最后更新玩家的垂直位置，即摄像机的 Y 坐标。

```

1 void MyGLWidget::updateFallState() {
2     // 计算当前位置对应的块，从而实现 z 轴方向的自然下坠
3     int block_x = int(player->camera->position.z * player->
4         invWallSize + 0.5f);
5     int block_y = int(player->camera->position.x * player->
6         invWallSize + 0.5f);
7     // 当没有在跳跃且没有在下坠且当前高度高于当前上表
8     // 面高度时，开始下坠
9     if (!isJumping && !isFalling && (player->camera->
10         position.y > player->maze->blockHeight[block_x][
11             block_y] + 1.6f * player->maze->wallSize) && !
12             isFlying) {
13             isFalling = true; // 开始下坠
14             fallStartTime = GetCurrentTimeInSeconds(); // 记录
15             // 下坠开始时间
16         }
17         if (isFalling) {
18             float currentTime = GetCurrentTimeInSeconds();
19             float timeSinceFall = currentTime - fallStartTime;
20             float fallHeight = fallInitialHeight - 0.5f *
21                 gravity * timeSinceFall * timeSinceFall;
22             // 检查是否结束下坠（踩到当前块的上表面）
23             if (fallHeight <= player->maze->blockHeight[block_x]
24                 [block_y] + 1.6f * player->maze->wallSize) {
25                 isFalling = false;
26                 fallHeight = player->maze->blockHeight[block_x]
27                     [block_y] + 1.6f * player->maze->
28                     wallSize; // 重置高度
29                 jumpInitialHeight = fallHeight; // 下次起跳高
30                 fallInitialHeight = fallHeight; // 下次坠落高
31                 fallHeight = jumpHeight; // 当前块的上表面高度
32             }
33             // 更新摄像机或玩家的垂直位置
34             // 假设摄像机或玩家的位置是 camera->position.y
35             player->camera->position.y = fallHeight;
36         }
37 }
```

Listing 5. 下坠的实现

首先，检查玩家是否处于非跳跃、非下坠状态，且高于当前块的上表面高度，且不处于飞行状态。如果条件满足，则开始下坠。如果正在下坠，计算下坠开始后的时间。使用物理方程计算下坠高度，只考虑了初始高度和由重力加速度导致的下坠。检查玩家是否触碰到迷宫的某个块的上表面，这将结束下坠过程。如果结束下坠，将下坠高度重置为当前块的上表面高度，并更新相关状态。更新玩家的垂直位置，即摄像机的 Y 坐标。

**4.4 多方向碰撞的检测.** 我们所实现在水平方向物理碰撞的效果十分简单，只需要判断位置是否合法，如果位置非法则保持不变。代码如下：

```

1 void Player::FBwardMove(Action action)
2 {
3     //记录当前位置
4     posX = camera->position.x;
5     posZ = camera->position.z;
6     if (action == forward)
7     {
8         //向前移动
9         camera->FBwardMove(forward);
10    } else if (action == backward)
11    {
12        //向后移动
13        camera->FBwardMove(backward);
14    }
15    //如果位置非法，则恢复为移动前位置
16    if (!maze->isValid(int(camera->position.z*invWallSize
17        +0.5f), int(camera->position.x*invWallSize+0.5f),
18        camera->position.y))
19    {
20        camera->position.x = posX;
21        camera->position.z = posZ;
22    }
23 }
24 void Player::HorizontalMove(Action action)
25 {
26     //记录当前位置
27     posX = camera->position.x;
28     posZ = camera->position.z;
29     if (action == toleft)
30     {
31         //向左移动
32         camera->HorizontalMove(topleft);
33     } else if (action == toright)
34     {
35         //向右移动
36         camera->HorizontalMove(toright);
37     }
38     //如果位置非法，则恢复为移动前位置
39     if (!maze->isValid(int(camera->position.z*invWallSize
40        +0.5f), int(camera->position.x*invWallSize+0.5f),
41        camera->position.y))
42     {
43         camera->position.x = posX;
44         camera->position.z = posZ;
45     }
46 }
```

Listing 6. 水平方向物理碰撞的检测

为实现垂直碰撞检测，需记录每个块的高度。

```

1 class Maze{
2     int maze[100][100];      //迷宫
3     float blockHeight[100][100]; //迷宫中每个元素块的高度
4 }
```

Listing 7. 迷宫的属性

垂直方向上的碰撞检测则相对较复杂。本游戏为迷宫中的每个块设定一个高度，表示这个块的上表面的高度。每次垂直移动时（包括跳跃和下坠）检测当前高度是否高于当前块的上表面的高度，如果高于，则垂直移动合法，允许移动，否则禁止此次垂直移动。

```

1 // 检查是否结束跳跃（踩到当前块的上表面，则结束跳跃）
2 if (jumpHeight <= player->maze->blockHeight[block_x][
3     block_y] + 1.6f * player->maze->wallsiz || 
4     jumpHeight >= jumpPeakHeight) {
5     isJumping = false;
6     jumpHeight = player->maze->blockHeight[block_x][block_y]
7         ] + 1.6f * player->maze->wallsiz; // 重置高度,
8         位于当前块的上表面
9         jumpInitialHeight = jumpHeight; // 下次起跳高度为当前
10        块的上表面高度
11        fallInitialHeight = jumpHeight; // 下次坠落高度为当前
12        块的上表面高度
13    }
14 }
```

Listing 8. 跳跃时的垂直碰撞检测

跳跃时，如果当前高度小于等于当前块的高度，这意味着玩家踩到当前块的上表面，则结束跳跃，

```

1 // 检查是否结束下坠（踩到当前块的上表面）
2 if (fallHeight <= player->maze->blockHeight[block_x][
3     block_y] + 1.6f * player->maze->wallsiz) {
4     isFallling = false;
5     fallHeight = player->maze->blockHeight[block_x][block_y]
6         ] + 1.6f * player->maze->wallsiz; // 重置高度
7     jumpInitialHeight = fallHeight; // 下次起跳高度为当前
8     块的上表面高度
9     fallInitialHeight = fallHeight; // 下次坠落高度为当前
10    块的上表面高度
11 }
12 }
```

Listing 9. 下坠时的垂直碰撞检测

下坠时，如果当前高度小于等于当前块的高度，这意味着玩家踩到当前块的上表面，则结束跳跃，

由于游戏支持同时处理多键输入，所以玩家的移动可以是水平移动和垂直移动的叠加。由于每次水平移动都会进行水平方向的碰撞检测，每次垂直移动都会进行垂直方向的碰撞检测，所以叠加起来可以得到多方向的碰撞检测，允许玩家以任意方式移动，而不会进入墙体内部。

**4.5 光照模型的实现.** 1. 环境光 (Ambient): 模拟间接光照，是各向同性的，公式为  $I_a = k_a L_a$ 。

2. 漫反射 (Diffuse): 模拟从光源直接照射到物体上并均匀散射的光，亮度与入射角的余弦值成正比，公式为  $I_d = k_d L_d \cos \theta$  或点积形式  $I_d = k_d L_d (n \cdot l)$ 。

3. 镜面反射 (Specular): 模拟光线在平滑表面的镜面反射，亮度与观察者视角有关，公式为  $I_s = k_s L_s \cdot \max((r \cdot v)^\alpha, 0)$ 。

总的光照效果是这三个分量的和，还可能包含距离衰减。代码如下所示：

```

1 GLfloat materialAmbient[] = { 1.0f, 1.0f, 1.0f, 1.0f }; // 环境光属性 (红, 绿, 蓝, Alpha)
2 GLfloat materialDiffuse[] = { 0.5f, 0.5f, 0.5f, 1.0f }; // 漫反射属性 (红, 绿, 蓝, Alpha)
3 GLfloat materialSpecular[] = { 0.8f, 0.8f, 0.8f, 1.0f }; // 镜面反射属性 (红, 绿, 蓝, Alpha)
4 GLfloat shininess[] = { 64.0f }; // 光泽度 (数值越高, 反射越强烈)
5 glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, materialAmbient);
6 glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, materialDiffuse);
7 glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, materialSpecular);
8 glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, shininess);
9 // 设置光源的属性
10 GLfloat lightSpecular[] = { 0.7f, 0.7f, 0.8f, 1.0f }; // 镜面反射属性 (红, 绿, 蓝, Alpha)
11 GLfloat lightDiffuse[] = { 0.6f, 0.6f, 0.9f, 1.0f }; // 漫反射属性 (红, 绿, 蓝, Alpha)
12 GLfloat lightAmbient[] = { 0.2f, 0.2f, 0.4f, 1.0f }; // 环境光属性 (红, 绿, 蓝, Alpha)
13 GLfloat lightConstAttenuation[] = { 1.0f }; // 光源恒定衰减系数
14 GLfloat lightLinearAttenuation[] = { 0.0f }; // 光源线性衰减系数
15 GLfloat lightQuadAttenuation[] = { 0.00f }; // 光源二次方衰减系数
16 //GLfloat lightPosition[] = { 1000.0f, 1300.0f, 1000.0f, 1.0f }; // 光源位置 (X, Y, Z, W) W代表点光源
17 GLfloat LightModelAmbient[] = { 0.6f, 0.6f, 0.6f, 1.0 }; // 全局环境光属性 (红, 绿, 蓝, Alpha)
18 glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
19 glLightfv(GL_LIGHT0, GL_AMBIENT, lightAmbient);
20 glLightfv(GL_LIGHT0, GL_DIFFUSE, lightDiffuse);
21 glLightfv(GL_LIGHT0, GL_SPECULAR, lightSpecular);
22 glLightfv(GL_LIGHT0, GL_CONSTANT_ATTENUATION,
            lightConstAttenuation);
23 glLightfv(GL_LIGHT0, GL_LINEAR_ATTENUATION,
            lightLinearAttenuation);
24 glLightfv(GL_LIGHT0, GL_QUADRATIC_ATTENUATION,
            lightQuadAttenuation);
25 glLightModelfv(GL_LIGHT_MODEL_AMBIENT, LightModelAmbient);
26 glEnable(GL_LIGHTING);
27 glEnable(GL_LIGHT0);
28 //Material
29 glEnable(GL_DEPTH_TEST);

```

Listing 10. 光照模型的参数设置

为了实现动态的效果，需要让光源动起来，即围绕着模型中心进行旋转，代码如下：

```

1 void MyGLWidget::updateLightPosition() {
2     // 示例：让光源绕着某个轴旋转
3     float radius = 1000.0f; // 光源旋转半径
4     float speed = 2.0f; // 旋转速度
5     static float angle = 0.0f; // 初始角度
6
7     // 计算新的光源位置
8     lightPosition[0] = radius * cos(angle);
9     lightPosition[2] = radius * sin(angle);
10    angle += speed * frameTime; // 根据帧时间更新角度
11
12    // 更新光源位置
13    glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
14 }

```

Listing 11. 光源的旋转移动

#### 4.6 动态纹理贴图的实现.

动态纹理的原理是通过实时生成或实时修改纹理来模拟变化的环境效果，动态纹理则可以根据场景中的事件、时间的推移或用户的交互动态变化。动态纹理技术增强了游戏或模拟环境的真实感和互动性，使得用户体验更加丰富和吸引人。

为了简化动态纹理的实现过程，同时仍然提升画面质量和增加游戏的可玩性及趣味性，本游戏采

用了一种较为简便的方法：使用几张不同但相近的纹理图像。这些纹理图像在视觉上有细微的差别，模拟动态变化的效果，比如模拟岩浆的迸发和藤蔓的随风摇曳。在游戏运行时，这些纹理会按一定的顺序和时间间隔被循环应用到相应的物体上，从而创造出动态变化的效果。这种方法相比于复杂的物理模型或算法生成的动态纹理，大大降低了实现的复杂性和计算成本，同时依然能够有效地增强游戏环境的真实感和互动性。

```

1 void MyGLWidget::paintMaze()
2 {
3     //开启纹理
4     glEnable(GL_TEXTURE_2D);
5     float baseZ, baseX;
6     float wallSize = player->maze->wallSize; //墙体大小
7     float dwallSize = wallSize * 10.0f; //墙体高度
8     updateCounter = (updateCounter + 1) % 1500; //画面更新次数
9     if(updateCounter % 150 == 0) //画面更新次数为150的倍数时，为陷阱切换纹理
10        currentFrameForTrap = (currentFrameForTrap + 1) % frameSizeForTrap;
11    if(updateCounter % 50 == 0) { //画面更新次数为50的倍数时，为藤蔓切换纹理
12        currentFrameForWall = (currentFrameForWall + 1) % frameSizeForWall;
13    }
14    //绘制墙
15    if (player->maze->maze[i][j] == 1 && player->maze->isDynamicWall[i][j] == false){
16        \dots
17    }
18    //绘制动态墙
19    else if (player->maze->maze[i][j] == 1 && player->maze->isDynamicWall[i][j]){
20        //选取纹理
21        glBindTexture(GL_TEXTURE_2D, player->maze->textures
22            .textures2D[2 + frameSizeForTrap +
23                currentFrameForWall]);
24        \dots
25    }
26    //绘制陷阱
27    else if (player->maze->maze[i][j] == 2){
28        //选取纹理
29        glBindTexture(GL_TEXTURE_2D, player->maze->textures
30            .textures2D[2 + currentFrameForTrap]);
31    }
32 }

```

Listing 12. 动态纹理实现

以上是动态纹理实现的代码，这里的动态纹理是通过预定的一组图片实现的，这些图片代表了纹理在不同时间点的不同状态。具体来说，paintMaze 函数在每次调用时更新一个计数器（updateCounter），并根据这个计数器的值来决定是否切换纹理

#### 4.7 岩浆陷阱的实现.

为了增强迷宫游戏的互动性和挑战性，实现了岩浆陷阱的生成和绘制。在岩浆迸发时，踩上陷阱会导致玩家死亡，游戏结束；而岩浆没有溢出时踩上陷阱不会导致玩家死亡。

岩浆陷阱的动态效果的实现方法已在 4.6 说明，以下是判断玩家是否踩中陷阱的代码，需判断当前块的标号和玩家高度

```

1 bool Player::checkTrap()
2 {
3     // 计算当前位置对应的块，从而查看是否为陷阱
4     int block_x = int(camera->position.z * invWallSize +
5         0.5f);
6     int block_y = int(camera->position.x * invWallSize +
7         0.5f);
8     // 如果当前位置对应的块是陷阱，而且当前摄像机高度小于
9     // 1.601，则判断踩中陷阱
10    return maze->isTrap(block_x, block_y) && camera->
11        position.y < 1.601f * maze->wallsiz;
12 }

```

Listing 13. 检测是否踩中陷阱

每次画面更新时，要检查玩家是否踩中迸发的岩浆，如果踩中，则游戏结束，固定玩家的位置和视角；如果没有踩中陷阱，或者踩中陷阱时岩浆没有迸发，则无事发生。

```

1 void MyGLWidget::paintGL()
2 {
3     // 省略
4     if (player->checkTrap() && (currentFrameForTrap == 1 ||
5         player->isFixed()))
6     {
7         glDisable(GL_DEPTH_TEST);
8         QPainter painter;
9         painter.begin(this);
10        QPen pen;
11        pen.setColor(Qt::red);
12        QFont font;
13        font.setPointSize(60);
14        painter.setFont(font);
15        painter.setPen(pen);
16        painter.drawText(width() / 2 - 160, height() / 2, "You Died");
17        painter.end();
18        glEnable(GL_DEPTH_TEST);
19        setMouseTracking(false);
20        player->setFixed();
21    }

```

Listing 14. 检测玩家是否死亡

玩家死亡不能移动的效果可以通过固定摄像机来实现，将前后移动速度、左右移动速度、垂直移动速度和视角移动速度都为 0

```

1 void Camera::setFixed()
2 {
3     fforward_speed = vertical_speed = horizontal_speed = 0.0f;
4     pitch_speed = yaw_speed = 0.0f;

```

Listing 15. 设置玩家死亡

玩家死亡之后更新的每一帧都会触发 `player->setFixed()` 和 `drawText` 的死亡提示，从而实现游戏的结束。

#### 4.8 小地图的实现.

为了优化游戏体验，防止因迷宫太大导致玩家迟迟无法通过，游戏中设置了一个小地图（二维迷宫示意图）。通过小地图，玩家更容易理解迷宫的整体结构，从而做出有效决策。

小地图被设定为 150 像素大小。小地图在屏幕上的位置根据窗口大小计算得出，位于窗口右上角，X 坐标留有 20 像素的边距。通过 `glViewport` 设置小地图的绘制区域。使用正交投影 (`gluOrtho2D`)

来设置 2D 视图，这适用于平面的小地图显示。循环遍历迷宫的每个单元格，如果是墙壁，则用白色绘制一个正方形 (`glRectf`) 来表示。根据玩家在迷宫中的位置，计算并绘制玩家的标记，使用黑色来表示。代码如下所示：

```

1 void MyGLWidget::drawMiniMap()
2 {
3     int miniMapSize = 150; // 小地图的大小
4     int mapX = width() - miniMapSize - 20; // 小地图的X坐标
5     int mapY = 20; // 小地图的Y坐标
6     glPushAttrib(GL_ALL_ATTRIB_BITS); // 保存当前OpenGL状态
7     glViewport(mapX, mapY, miniMapSize, miniMapSize); // 设置视口大小
8     glMatrixMode(GL_PROJECTION);
9     glLoadIdentity();
10    gluOrtho2D(0, player->maze->Width, 0, player->maze->
11        Height); // 设置正交投影
12    glMatrixMode(GL_MODELVIEW);
13    glLoadIdentity();
14    // 绘制迷宫
15    for (int i = 0; i < player->maze->Height; i++)
16    {
17        for (int j = 0; j < player->maze->Width; j++)
18        {
19            float aj = player->maze->Width - j - 1;
20            if (player->maze->isWall(i, j))
21            {
22                glColor3f(1.0f, 0.0f, 0.0f); // 白色表示墙壁
23                glRectf(aj, i, aj + 1, i + 1); // 绘制一个单元格表示墙壁
24            }
25        }
26    }
27    // 标出玩家位置
28    glColor3f(0.0f, 0.0f, 0.0f); // 黑色表示玩家位置
29    float playerX = player->camera->position.x / 130.0f -
30        0.6f; // 计算玩家在小地图上的X坐标
31    float playerZ = player->camera->position.z / 130.0f +
32        0.5f; // 计算玩家在小地图上的Y坐标
33    // std::cout << playerX << "," << playerZ << "\n" << endl;
34    // 绘制玩家位置标记
35    float aX = player->maze->Width - playerX - 1;
36    glRectf(aX - 0.1f, playerZ - 0.1f, aX + 0.1f, playerZ +
37        0.1f);
38    glPopAttrib(); // 恢复之前保存的OpenGL状态
39    glViewport(0, 0, width(), height());
40    // 恢复到主视图
41 }

```

Listing 16. 小地图的绘制

#### 4.9 天空盒的实现.

天空盒是一种在计算机图形学中常用的技术，用于创建广阔且真实感的天空背景，从而增强游戏的真实感和游戏体验。它的原理是将玩家置于一个巨大的立方体（或盒子）内部，每个面上贴有天空、云彩、太阳、星星等图案。这些图案通常是全景照片或艺术作品，能够在玩家视角转动时提供连续且一致的背景，创造出开阔无边的天空环境。尽管玩家在游戏中移动，天空盒本身并不会移动或改变，这样保持了场景的一致性，同时大大减少了渲染负担。

在初始化时，加载天空盒六个面的纹理，方便后续绘制，这部分代码省略。以下是绘制天空盒的代码，每次刷新画面时，绘制一个巨大立方体，为立方体的每个面贴上天空盒的纹理

```

1 void MyGLWidget::paintSkyBox()
2 {
3     //开启立方体贴图
4     glEnable(GL_TEXTURE_CUBE_MAP);
5     float baseZ, baseX;
6     float wallsize = player->maze->wallsize;
7     float fwallsize = 20.0f * wallsize * float(player->maze
8         ->Width) * 1.5f;
9     baseX = 1.0f * float(player->maze->Width) * wallsize;
10    baseZ = 1.0f * float(player->maze->Height) * wallsize;
11    //绑定纹理
12    glBindTexture(GL_TEXTURE_CUBE_MAP, skyBox.texturesCube
13        [0]);
14    glBegin(GL_QUADS);
15    //right
16    glTexCoord3f(1.0f, 1.0f, -1.0f);
17    glVertex3f(baseX+fwallsize, -fwallsize, baseZ+fwallsize
18        );
19    glTexCoord3f(1.0f, 1.0f, 1.0f);
20    glVertex3f(baseX+fwallsize, -fwallsize, baseZ-fwallsize
21        );
22    glTexCoord3f(1.0f, -1.0f, 1.0f);
23    glVertex3f(baseX+fwallsize, fwallsize, baseZ-fwallsize)
24    ; //left //top //down //front //back 省略
25    glEnd();
26    glDisable(GL_TEXTURE_CUBE_MAP);
}

```

Listing 17. 绘制天空盒

**4.10 机关的实现.** 为了增强迷宫游戏的互动性和挑战性, 本游戏引入了一个特殊机关, 玩家必须先激活这个机关才能从迷宫的出口离开。这个机关的实现相对简单: 在由 Kruskal 算法生成的迷宫中, 选择任意一个空房间放置机关, 并将其标记为特殊值。玩家通过检查所在位置的值来确定是否达到机关位置, 并在到达机关处时激活它。此外, 我们设置了一种机制, 在玩家到达终点时, 会检查机关是否已激活, 这是通过一个布尔类型的成员变量来实现的。如果机关未激活, 则会显示提示信息“机关尚未激活”, 如果激活, 则显示“完成”以提示游戏结束。

在 maze.cpp 的 generate\_mechanism 中

```

1 for (int i = 0; i < Height * Width / 12; i++) {
2     // 如果存在通道, 则随机选择一个设置为机关
3     if (!paths.empty()) {
4         int randomIndex = rand() % paths.size(); // 随机选
5         // 择一个索引
6         std::pair<int, int> trapPos = paths[randomIndex];
7         maze[trapPos.first][trapPos.second] = 4; // 设置选
8         // 中的通道为机关
9         blockHeight[trapPos.first][trapPos.second] = 0.0f;
10        // 机关高度为0
11        break;
12    }
}

```

Listing 18. 机关的实现

在 player.cpp 中增加以下代码

```

1 bool Player::checkMoney()
2 {
3     // 计算当前位置对应的块, 从而查看是否为机关
4     int block_x = int(camera->position.z * invWallSize +
5         0.5f);
6     int block_y = int(camera->position.x * invWallSize +
7         0.5f);
8     // 如果当前位置对应的块是机关, 则激活机关
9     return maze->isMoney(block_x, block_y);
}

```

Listing 19. 机关的实现

在 myglwidget.cpp 中增加以下代码

```

1 if (player->checkWin())
2 {
3     if (player->money == true) {
4         glDisable(GL_DEPTH_TEST);
5         QPainter painter;
6         painter.begin(this);
7         QPen pen;
8         pen.setColor(Qt::red);
9         QFont font;
10        font.setPointSize(60);
11        painter.setFont(font);
12        painter.setPen(pen);
13        painter.drawText(width() / 2 - 120, height() / 2,
14            "Finish");
15        painter.end();
16        glEnable(GL_DEPTH_TEST);
17        setMouseTracking(false);
18        player->setFixed();
19    }
20    else {
21        glDisable(GL_DEPTH_TEST);
22        QPainter painter;
23        painter.begin(this);
24        QPen pen;
25        pen.setColor(Qt::blue);
26        QFont font;
27        font.setPointSize(60);
28        painter.setFont(font);
29        painter.setPen(pen);
30        painter.drawText(width() / 2 - 120, height() / 2,
31            "The mechanism has not been activated yet");
32        painter.end();
33    }
34}
35 if (player->checkMoney()) {
36     glDisable(GL_DEPTH_TEST);
37     player->money = true;
38     QPainter painter;
39     painter.begin(this);
40     QPen pen;
41     pen.setColor(Qt::blue); // 你可以根据需要选择颜色
42     QFont font;
43     font.setPointSize(30);
44     painter.setFont(font);
45     painter.setPen(pen);
46     painter.drawText(width() / 2 - 200, height() / 2,
47         "Mechanism activation");
48     painter.end();
49     glEnable(GL_DEPTH_TEST);
50 }

```

Listing 20. 机关的实现

**4.11 周期性刷新画面.** 为了实现玩家位置和视角的移动, 以及动态纹理的展示, 本游戏需要周期性地刷新纹理和摄像机的参数, 具体是这样做的:

```

1 MyGLWidget::MyGLWidget(QWidget *parent)
2 :QOpenGLWidget(parent)
3 {
4     timer = new QTimer(this); // 实例化一个定时器
5     timer->start(10); // 时间间隔设置为10ms, 可以根据需要调整
6     connect(timer, SIGNAL(timeout()), this, SLOT(update()));
7     setWindowTitle("TuWei Maze");
8     this->resize(QSize(1200, 800));
9     setMouseTracking(true);
10    player = new Player;
11    firstMouse = true;
12 }

```

Listing 21. 周期性刷新画面

将 timer 的 timeout 绑定到画面的更新操作上, 每次 timeout 都要调用 paintGL 更新画面, 每次更新要检测水平移动和垂直移动, 从而更新摄像机的参数; 同时要检查是否更新动态纹理; 检查游戏状态 (死亡或胜利)。

**4.12 加载 2D 纹理.** LoadTexture2D 函数用于加载 2D 纹理到 OpenGL 中。它首先打开一个指定的 BMP 文件并检查其有效性。通过读取文件头信息，函数获取图像的尺寸，并根据这些维度计算所需的内存空间，为图像数据动态分配内存。接着，它从文件中读取图像数据，同时转换颜色格式，因为 OpenGL 一般使用 RGB 格式，而 BMP 存储的是 BGR 格式。完成这些后，函数在 OpenGL 中生成一个新的纹理对象，并设置相应的纹理参数，如纹理过滤和包裹模式，以确保纹理在各种尺寸的表面上正确显示。最后，它使用 gluBuild2DMipmaps 创建纹理的 Mipmap，这是一种优化技术，可以在纹理缩放时保持图像质量。整个过程涉及图像的读取、处理，并在 OpenGL 中以合适的格式加载，以便在 3D 图形渲染中使用。

```

1 void Textures::LoadTexture2D(const char* filename)
2 {
3     GLint width, height, i; //位图大小
4     GLubyte* image; //像素储存地址,中间
5     FILE* pf; //位图打开地址
6     BITMAPFILEHEADER fileHeader; //位图文件头(包含文件类型,文件头大小)
7     BITMAPINFOHEADER infoHeader; //位图信息头(包含位图宽高,色素总量大小)
8     fopen_s(&pf, filename, "rb"); //只读方式打开图片文件
9     if (pf == nullptr) {
10         std::cout << "Failed to read files!" << std::endl;
11         return;
12     }
13     fread(&fileHeader, sizeof(BITMAPFILEHEADER), 1, pf);
14     if (fileHeader.bfType != 0x4D42) {
15         std::cout << "It is not the bmp file!" << std::endl;
16         fclose(pf);
17         return;
18     }
19     fread(&infoHeader, sizeof(BITMAPINFOHEADER), 1, pf);
20     width = infoHeader.biWidth;
21     height = infoHeader.biHeight;
22     if (infoHeader.biSizeImage==0) //计算图片像素总量
23         infoHeader.biSizeImage = width * height * 4;
24     image = (GLubyte*)malloc(sizeof(GLubyte) * infoHeader.
25     biSizeImage); //申请空间
26     if (image == nullptr) {
27         std::cout << "The space is not enough!" << std::endl;
28         fclose(pf);
29         free(image);
30         return;
31     }
32     fseek(pf, fileHeader.bfOffBits, SEEK_SET); //将文件读写头移到文件头处
33     fread(image, infoHeader.biSizeImage, 1, pf);
34     for (i = 0; i < infoHeader.biSizeImage; i += 4) {
35         //openGL识别的是BGR,所以要置换过来
36         std::swap(image[i], image[i+2]);
37     }
38     fclose(pf);
39     glGenTextures(1, &textures2D[count2D]); //纹理设置,线性滤波
40     glBindTexture(GL_TEXTURE_2D, textures2D[count2D]);
41     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
42     GL_LINEAR); //纹理设置,线性滤波
43     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
44     GL_LINEAR);
45     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
46     GL_REPEAT); //纹理超出\不足则重复绘制
47     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
48     GL_REPEAT);
49     gluBuild2DMipmaps(GL_TEXTURE_2D, 4, width, height,
50     GL_RGBA, GL_UNSIGNED_BYTE, image); //分配纹理编号
51     count2D++;
52 }
```

Listing 22. 加载 2D 纹理

**4.13 加载立方体纹理.** LoadTextureCube 函数的核心原理是在 OpenGL 中创建并加载一个立方体纹理，用于构建天空盒。函数首先生成一个新的 OpenGL 立方体纹理对象并将其绑定。接着，对于天空盒的每一个面（共六个），函数循环遍历，打开对应的 BMP 文件，读取并验证其格式。然后，从文件中提取图像尺寸信息，为每个图像分配适量内存。图像数据被读取到内存中，并进行颜色格式的转换（从 BGR 到 RGB），以符合 OpenGL 的标准。对每个面的图像处理完成后，通过 gluBuild2DMipmaps 将其加载为立方体纹理的一部分。最后，函数为整个立方体纹理设置纹理过滤和包裹模式的参数，确保其在 3D 环境中正确渲染。

```

1 void Textures::LoadTextureCube(const char (*filenames)[100])
2 {
3     GLint width, height;
4     GLubyte* image;
5     FILE* pf;
6     BITMAPFILEHEADER fileHeader;
7     BITMAPINFOHEADER infoHeader;
8     glGenTextures(1, &texturesCube[countCube]);
9     glBindTexture(GL_TEXTURE_CUBE_MAP, texturesCube[
10     countCube]);
11     countCube++;
12     for (int j = 0; j < 6; j++)
13     {
14         fopen_s(&pf, filenames[j], "rb");
15         if (pf == nullptr) {
16             std::cout << "Failed to read files!" << std::endl;
17             return;
18         }
19         fread(&fileHeader, sizeof(BITMAPFILEHEADER), 1, pf);
20         if (fileHeader.bfType != 0x4D42) {
21             std::cout << "It is not the bmp file!" << std::endl;
22             fclose(pf);
23             return;
24         }
25         fread(&infoHeader, sizeof(BITMAPINFOHEADER), 1, pf);
26         width = infoHeader.biWidth;
27         height = infoHeader.biHeight;
28         if (infoHeader.biSizeImage) //计算图片像素总量
29             infoHeader.biSizeImage = width * height * 4;
30         image = (GLubyte*)malloc(sizeof(GLubyte) * infoHeader.
31         biSizeImage); //申请空间
32         if (image == nullptr) {
33             std::cout << "The space is not enough!" << std::endl;
34             fclose(pf);
35             free(image);
36             return;
37         }
38         fseek(pf, fileHeader.bfOffBits, SEEK_SET); //将文件读写头移到文件头处
39         fread(image, infoHeader.biSizeImage, 1, pf);
40         for (int i = 0; i < infoHeader.biSizeImage; i += 3)
41         {
42             //openGL识别的是BGR,所以要置换过来
43             std::swap(image[i], image[i+2]);
44         }
45         fclose(pf);
46         gluBuild2DMipmaps(GL_TEXTURE_CUBE_MAP_POSITIVE_X+j,
47         3, width, height, GL_RGB, GL_UNSIGNED_BYTE,
48         image);
49     }
50 }
```

Listing 23. 加载立方体纹理

## 5. 游戏实现的效果展示

5.1 进入游戏后的正常界面. 点击进入游戏以后画面如下所示：可以看到高耸的峡谷和脚下的道路，抬头可以看到天空。



图 3. 游戏画面 1



图 4. 游戏画面 2

5.2 矮墙和光照效果. 以下两张图展示了矮墙的效果，同时展示了光源位置不同时，画面渲染的表现



图 5. 光源点 1 下的矮墙



图 6. 光源点 2 下的矮墙

5.3 岩浆陷阱. 陷阱有两种状态，一种是岩浆已溢出，一种是岩浆未溢出，未溢出的时候可以安全通过。陷阱在溢出和未溢出两种状态之间来回转换。



图 7. 岩浆溢出



图 8. 岩浆没有溢出

5.4 动态纹理. 迷宫中的墙体有一定概率长出藤蔓，采用动态纹理实现藤蔓的随风摇曳的效果。以下是藤蔓摆动的示意图



图 9. 藤蔓摆动 1



图 10. 藤蔓摆动 2

5.5 特殊机关. 本游戏引入了一个特殊机关，玩家必须先激活这个机关才能从迷宫的出口离开。



图 11. 机关示意 1



图 12. 机关示意 2

5.6 飞起来以后看天空盒与峡谷. 点击 F 以后可以飞起来，这时候可以看到峡谷的全貌与天空盒的全貌



图 13. 天空



图 14. 峡谷

5.7 游戏结束. 游戏结束的方式有两种，一种是到达终点会显示游戏解释，一种是调入岩浆会显示死亡



图 15. 胜利



图 16. 死亡

## 6. 团队分工

组员信息	姓名	分工
21312028	卢敦杰	总体设计、文件组织
		用户第一视角设置与变换
		用户键盘，鼠标输入响应
		跳跃的实现
		光源与天空盒的旋转
		水平方向的碰撞检测
		bmp 图片的读取与贴图
		资料收集，游戏测试
		论文的撰写，ppt 的完善
		小地图的实现
21307077	凌国明	矮墙生成
		跳跃逻辑的完善
		下坠的实现
		垂直方向碰撞检测
		岩浆陷阱设置
		动态纹理
		天空盒的绘制
		资料收集，游戏测试
		论文的完善
		光照设置
21307620	方宇豪	3D 地图的生成
		动态纹理的完善
		随机迷宫生成算法
		终点雪花特效
		终点机关的实现
		资料收集，游戏测试
		ppt 的汇报
		ppt 的撰写

表 2. 小组分工表