



《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

作者姓名： 21307077 凌国明

# 基于 OpenGL 的三维峡谷探险迷宫设计

凌国明<sup>1</sup>

## 摘要

该报告主要介绍作者在“基于 OpenGL 的三维峡谷探险迷宫设计”中实现的部分技术，包括矮墙生成、多方向碰撞检测、光照设置、陷阱设置、动态纹理等。

## 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，包括可以翻越的障碍围墙，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十分精细，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，光照明影也会随之移动。

总体来说，游戏界面精细美观，可操作性和交互性强。

## 2. 个人工作

笔者负责的部分包括矮墙生成、多方向碰撞检测、光照设置、陷阱设置、动态纹理等。

为了使游戏过程更加曲折，增加了可跳跃通过的矮墙。矮墙需要跳跃通过，从矮墙移动到平地会导致下坠，因此笔者实现了跳跃和下坠。跳跃和下坠的过程中又涉及到碰撞检测，因此笔者实现了多方向的碰撞检测。

为了使游戏更具趣味性，在迷宫中加入了岩浆陷阱，陷阱会在“岩浆迸发”和“岩浆无溢出”两种状态之间来回转换，当玩家踩到岩浆迸发的陷阱时，玩家死亡，游戏结束。

为了使游戏的画面更加真实饱满，设置了天空盒和 phong 光照模型，通过天空盒的旋转和光源的旋转，实现了动态的光源效果。

此外还设置了藤蔓和陷阱的动态纹理，通过时间对藤蔓和陷阱的纹理进行采样，实现了动态变化的画面效果。

## 3. 技术实现

**3.1 生成矮墙和陷阱.** 为了还原峡谷凹凸不平的地面向和凶险的环境，增加游戏的可玩性，在迷宫中增加可越过的矮墙和致人死亡的陷阱。在生成迷宫后，随机选择一定数量的通道变成矮墙和陷阱，其中矮墙的高度随机生成。

在 maze.cpp 中，生成迷宫后做如下操作：首先记录所有通道的位置，方便后续对通道进行修改操作，同时将通道的高度设置为 0，方便垂直碰撞检测的实现。

```
1 std::vector<std::pair<int, int>> paths;
2 for (int i = 0; i < Height; i++) {
3     for (int j = 0; j < Width; j++) {
4         if (maze[i][j] == 0) {
5             paths.push_back(std::make_pair(i, j));
6             blockHeight[i][j] = 0.0f; // 设置该块的高度为0
7         }
8     }
9 }
```

Listing 1. 记录所有通道的位置

在所有通道中随机选择一定数量的块，把选择的通道块替换为陷阱块或矮墙块。陷阱的高度设置为 0，矮墙的高度随机生成，但为了确保迷宫有可行路径，矮墙的高度不可超过最大跳跃高度。

```

1 for (int i = 0; i < Height * Width / 4; i++) {
2     // 如果存在通道，则随机选择一个设置为矮墙
3     if (!paths.empty()) {
4         // strand(time(NULL)); // 初始化随机数生成器
5         int randomIndex = rand() % paths.size(); // 随机选
6         std::pair<int, int> trapPos = paths[randomIndex];
7         maze[trapPos.first][trapPos.second] = 3; // 设置选
8         // 中的通道为矮墙
9         blockHeight[trapPos.first][trapPos.second] = float(
10            rand() % 4 + 1) / 4 * wallSize * 2.0f; // 随
11        // 机生成矮墙的高度
12    }
13    for (int i = 0; i < Height * Width / 12; i++) {
14        // 如果存在通道，则随机选择一个设置为陷阱
15        if (!paths.empty()) {
16            int randomIndex = rand() % paths.size(); // 随机选
17            std::pair<int, int> trapPos = paths[randomIndex];
18            maze[trapPos.first][trapPos.second] = 2; // 设置选
19            // 中的通道为陷阱
20            blockHeight[trapPos.first][trapPos.second] = 0.0f;
21        }
22    }

```

Listing 2. 随机生成矮墙和陷阱

**3.2 跳跃和下坠的实现.** 垂直方向建模为匀加速直线运动，实现使用物理公式  $h = h_0 + v_0 \cdot t - \frac{1}{2} \cdot g \cdot t^2$  来实现跳跃的效果。具体来说，本游戏周期性地更新画面，每次更新都可以计算距起跳的时间  $t$ ，然后可以按上述公式算出当前高度。

```

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 *
7             * timeSinceJump * timeSinceJump;
8         player->camera->position.y = jumpHeight;
9     }

```

Listing 3. 跳跃的实现（没有垂直碰撞检测）

但是上述逻辑还没有实现垂直方向的碰撞检测，因为当时代码的逻辑是高度回到起跳点的高度时结束跳跃，而实际上因为矮墙的存在，落点高度和起跳点高度不一定相同。

于是在 `maze.h` 中增加以下属性，通过 `blockHeight` 数组记录每个块的高度。这个数组的信息在迷宫生成时进行初始化，通道和陷阱的高度设置为 0，高墙的高度设置为不可越过，矮墙的高度随机生成但小于最大跳跃高度。

```

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

```

Listing 4. 迷宫的属性

然后修改跳跃的逻辑：每次计算当前位置对应的块，当目前高度小于等于当前块的高度时，结束跳跃，设置当前高度为当前所在块的上表面的高度。这样就实现了跳跃时

```

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 *
7             * timeSinceJump * 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         // 测
10        int block_x = int(player->camera->position.z *
11            player->invWallSize + 0.5f);
12        int block_y = int(player->camera->position.x *
13            player->invWallSize + 0.5f);
14        // 检查是否结束跳跃（踩到当前块的上表面则结束跳
15        // 跃）（碰到迷宫地图上边界也结束跳跃）
16        if (jumpHeight <= player->maze->blockHeight[block_x
17            ][block_y] + 1.6f * player->maze->wallSize || |
18            jumpHeight >= jumpPeakHeight) {
19            isJumping = false;
20            jumpHeight = player->maze->blockHeight[block_x
21            ][block_y] + 1.6f * player->maze->
22            wallSize; // 重置高度，位于当前块的上表面
23            jumpInitialHeight = jumpHeight; // 下次起跳高
24            fallInitialHeight = jumpHeight; // 下次坠落高
25            // 为当前块的上表面高度
26        }
27        // 更新摄像机或玩家的垂直位置
28        // 假设摄像机或玩家的位置是 camera->position.y
29        player->camera->position.y = jumpHeight;
30    }
31 }

```

Listing 5. 跳跃的实现（有垂直碰撞检测）

当玩家从一个较高的块往一个较低的块移动时，例如从矮墙往通道上移动时，理应受到重力作用而自然下坠。于是编写以下函数，使用匀加速直线运动公式  $h = h_0 - \frac{1}{2} \cdot g \cdot t^2$  实现下坠的功能。

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。

```

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
3     // 坠
4     int block_x = int(player->camera->position.z *
5         player->invWallSize + 0.5f);
6     int block_y = int(player->camera->position.x *
7         player->invWallSize + 0.5f);
8     // 当没有在跳跃且没有在下坠且当前高度高于当前
9     // 上表面高度时，开始下坠
10    if (!isJumping && !isFalling && (player->camera->
11        position.y > player->maze->blockHeight[
12            block_x][block_y] + 1.6f * player->maze->
13            wallSize) && !isFlying) {
14        isFalling = true; // 开始下坠
15        fallStartTime = GetcurrentTimeInSeconds(); // 记录下坠开始时间
16    }
17    if (isFalling) {
18        float currentTime = GetcurrentTimeInSeconds();
19        float timeSinceFall = currentTime - fallStartTime;
20        fallHeight = fallInitialHeight - 0.5f *
21            gravity * timeSinceFall * timeSinceFall;
22        player->camera->position.y = fallHeight; // 更
23        // 新位置
24    }
25    // 停止条件涉及垂直碰撞检测

```

Listing 6. 下坠的实现（没有垂直碰撞检测）

可见上述代码没有下坠停止的逻辑，这涉及到垂直碰撞检测的实现，实现原理与跳跃相同。

```

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面高度时，开始下坠
8     if (!isJumping && !isFalling && (player->camera->
9         position.y > player->maze->blockHeight[block_x][
10            block_y] + 1.6f * player->maze->wallsizes) && !
11            isFlying) {
12         isFalling = true; // 开始下坠
13         fallStartTime = GetCurrentTimeInSeconds(); // 记录下坠开始时间
14     }
15     if (isFalling) {
16         float currentTime = GetCurrentTimeInSeconds();
17         float timeSinceFall = currentTime - fallStartTime;
18         float fallHeight = fallInitialHeight - 0.5f *
19             gravity * timeSinceFall * timeSinceFall;
20         if (fallHeight <= player->maze->blockHeight[block_x][
21            block_y] + 1.6f * player->maze->wallsizes) {
22             isFalling = false;
23             fallHeight = player->maze->blockHeight[block_x][
24                block_y] + 1.6f * player->maze->
25                    wallsizes; // 重置高度
26         }
27         jumpInitialHeight = fallHeight; // 下次起跳高度为当前块的上表面高度
28         fallInitialHeight = fallHeight; // 下次坠落高度为当前块的上表面高度
29     }
30     // 更新摄像机或玩家的垂直位置
31     // 假设摄像机或玩家的位置是 camera->position.y
32     player->camera->position.y = fallHeight;
33 }

```

Listing 7. 下坠的实现（有垂直碰撞检测）

下坠结束条件与跳跃结束条件是相近的：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，

**3.3 多方向碰撞的检测.** 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。因此，我们不仅需要进行垂直方向的碰撞检测，而且还要进行水平方向的碰撞检测，允许玩家以任意方式移动，而不会进入墙体内部。

垂直方向的碰撞检测已经在 3.2 节说明。水平方向物理碰撞十分简单，只需要在水平移动时，判断目标位置是否合法，合法则更改位置，非法则不允许位置更改。

```

1 /**
2  * 函数：isValid
3  * 函数描述：判断位置是否是可行路径（包括通道，矮墙上方，陷阱上方）
4  * 参数描述：x, y 目标位置
5 */
6 bool Maze::isValid(int x, int y, int z)
7 {
8     return isArea(x, y) && (maze[x][y] != 1 && z >= 1.6f *
9         wallsizes + blockHeight[x][y]);
}

```

Listing 8. 判断水平移动的目标位置是否合法

以上是检测水平移动目标位置是否合法的代码，通过 maze 的标号和 blockHeight 的高度信息来判断位置是否是可行路径（包括通道，矮墙上方，陷阱上方）

```

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    }
11    else if (action == backward)
12    {
13        //向后移动
14        camera->FBwardMove(backward);
15    }
16    //如果位置非法，则恢复为移动前位置
17    if (!maze->isValid(int(camera->position.z*invWallSize +
18        +0.5f), int(camera->position.x*invWallSize+0.5f),
19        camera->position.y))
20    {
21        camera->position.x = posX;
22        camera->position.z = posZ;
23    }
}

```

Listing 9. 前后移动（带水平碰撞检测）

```

1 void Player::HorizontalMove(Action action)
2 {
3     //记录当前位置
4     posX = camera->position.x;
5     posZ = camera->position.z;
6     if (action == toleft)
7     {
8         //向左移动
9         camera->HorizontalMove(toleft);
10    }
11    else if (action == toright)
12    {
13        //向右移动
14        camera->HorizontalMove(toright);
15    }
16    //如果位置非法，则恢复为移动前位置
17    if (!maze->isValid(int(camera->position.z*invWallSize +
18        +0.5f), int(camera->position.x*invWallSize+0.5f),
19        camera->position.y))
20    {
21        camera->position.x = posX;
22        camera->position.z = posZ;
23    }
}

```

Listing 10. 左右移动（带水平碰撞检测）

以上水平运动的碰撞检测，和跳跃或下坠时的垂直碰撞检测叠加，即可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。

**3.4 光照模型的实现.** 环境光是一种模拟间接光照的方法，它不考虑物体与光源之间的相对位置或角度。这种光照是各向同性的，即在所有方向上都是一致的。环境光通常用于模拟由于场景中其他对象的反射而产生的间接光。

漫反射模拟光线直接照射到物体表面并均匀散射的效果。这种反射不会产生明显的光亮斑点，而是提供物体表面的基本颜色感觉。其亮度与光线入射角的余弦值成正比，这意味着与光线垂直的表面会接收到最多的光。

镜面反射模拟光线在平滑表面上的镜面反射效果。这种反射产生亮点或高光，并且其亮度取决于观察者的视角。

Phong 光照模型通过结合这三种光照效果，来创建逼真的视觉效果。它通过调整环境光、漫反射和镜面反射的参数，可以模拟不同材质和光照条件下的物体外观。

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=1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 11. phong 光照模型的设置

以上代码设置了环境光 (Ambient)、漫反射 (Diffuse)、镜面反射 (Specular) 的各种参数，包括三种光的属性 (红, 绿, 蓝, Alpha)，光源的位置，光源衰减系数等，并开启光照模型，使得全局物体由这个模型渲染。

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 12. 光源的旋转移动

3.4 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->Width) * 1.5f;
8     baseX = 1.0f * float(player->maze->Width) * wallSize;
9     baseZ = 1.0f * float(player->maze->Height) * wallSize;
10    //绑定纹理
11    glBindTexture(GL_TEXTURE_CUBE_MAP, skyBox.texturesCube[0]);
12    glBegin(GL_QUADS);
13    //right
14    glTexCoord3f(1.0f, 1.0f, -1.0f);
15    glVertex3f(baseX+fwallSize, -fwallSize, baseZ+fwallSize);
16    glTexCoord3f(1.0f, 1.0f, 1.0f);
17    glVertex3f(baseX+fwallSize, -fwallSize, baseZ-fwallSize);
18    glTexCoord3f(1.0f, -1.0f, 1.0f);
19    glVertex3f(baseX+fwallSize, fwallSize, baseZ-fwallSize);
20    glTexCoord3f(1.0f, -1.0f, -1.0f);
21    glVertex3f(baseX+fwallSize, fwallSize, baseZ+fwallSize);
22    //left //top //down //front //back 省略
23    glEnd();
24    glDisable(GL_TEXTURE_CUBE_MAP);
25 }

```

Listing 13. 绘制天空盒

以上代码使用预先加载的纹理文件，为天空盒的六个面分别贴上纹理。

**3.5 动态纹理.** 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    }
21    //选取纹理
22    glBindTexture(GL_TEXTURE_2D, player->maze->textures.textures2D[2 + frameSizeForTrap + currentFrameForWall]);
23    \dots
24 }
25 //绘制陷阱
26 else if ((player->maze->maze[i][j] == 2) {
27    //选取纹理
28    glBindTexture(GL_TEXTURE_2D, player->maze->textures.textures2D[2 + currentFrameForTrap]);
29    \dots
30 }
31 }
```

Listing 14. 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

**3.6 岩浆陷阱的实现.** 陷阱生成的必要性和生成方法已经在 3.1 节说明。在这里特别强调，岩浆陷阱有两种状态：溢出和没有溢出，陷阱在这两种状态间来回转换，通过动态纹理实现转换效果。

岩浆陷阱的致死逻辑是：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。

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->wallSize;
12 }
```

Listing 15. 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    }
22 }
```

Listing 16. 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     fforward_speed = vertical_speed = horizontal_speed = 0.0f;
3     pitch_speed = yaw_speed = 0.0f;
4 }
```

Listing 17. 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. 效果展示

4.1 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 1. 光源点 1 下的矮墙



图 2. 光源点 2 下的矮墙

4.2 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 3. 藤蔓摆动 1

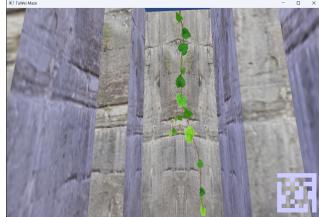


图 4. 藤蔓摆动 2

4.3 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。

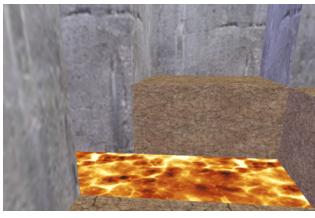


图 5. 岩浆溢出



图 6. 岩浆没有溢出

## 5. 总结与思考

首先谈谈遇到的困难。我负责的部分，最难的是多方向的碰撞检测，特别是垂直方向的碰撞检测。一开始编写代码的时候，脑子里没有清晰的代码架构，想到什么写什么，想到从通道向矮墙跳跃，就按照自己的思路编写。但是写完后测试时才发现，跳跃其实分为多种情况，分别是：平地向矮墙跳跃、矮墙向矮墙跳跃、矮墙向平地跳跃，而我只实现了平地向矮墙跳跃。

经过一番思考，才在 `maze.h` 中增加迷宫每块的高度记录，在垂直运动时，通过判断摄像机视角和当前块的上表面高度的关系，来实现垂直的碰撞检测。有了这样的构思后，写起代码就非常轻松，测试也很快通过了。

跳跃测试通过后，又发现了一个不符合实际规律的情况：从矮墙上往平地移动，理应受到重力作用而自然下坠，但是这种情况，玩家是不处于跳跃状态的，因此增加了自然下坠的代码逻辑，模拟玩家受到重力作用自然下坠。

通过实现跳跃和下坠的逻辑，我明白了一个道理：在编程中，深思熟虑地思考所有可能的情况和规划代码架构至关重要。一开始，由于缺乏清晰的规划，我在实现跳跃时遇到了障碍。然而，当我开始仔细考虑如何有效地处理不同的游戏情景时，问题的解决变得更加明确和可行。这不仅提高了编程效率，也确保了代码的可扩展性和可维护性。

总的来说，经过这次大作业，我学习到的不仅仅是计算机图形学的知识，还总结出了一点宝贵的经验教训。这次大作业使我受益匪浅。