OpenGL游戏项目源代码详细分析文档

目录

一、项目概述

本项目是一个基于OpenGL的3D游戏，实现了一个"躲猫"（CatDodge）游戏。玩家控制一只猫躲避从天而降的物体，需要尽可能长时间生存。项目采用现代OpenGL（3.3+）和C++实现。

技术栈

- **图形库**: OpenGL 3.3 Core Profile
- **窗口管理**: GLFW
- **数学库**: GLM (OpenGL Mathematics)
- **模型加载**: Assimp
- **音频系统**: OpenAL
- **字体渲染**: stb_truetype
- **图像加载**: stb_image

二、文件功能详细分析

2.1 main.cpp - 程序入口和主循环

核心职责：

- 初始化GLFW窗口和OpenGL上下文
- 设置事件回调函数（键盘、鼠标、滚轮）
- 管理游戏状态机（菜单/游戏中/游戏结束）
- 实现相机系统（第一人称/第三人称切换）
- 主游戏循环（更新和渲染）

关键功能模块：

1. 窗口初始化

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

- 设置OpenGL 3.3 Core Profile
- 创建1280x920窗口
- 启用深度测试和sRGB

2. 状态管理

```
enum class State { MENU, PLAYING, GAMEOVER };
```

- **MENU**: 主菜单界面
- **PLAYING**: 游戏进行中
- **GAMEOVER**: 游戏结束界面

3. 相机系统

- **第三人称视角**: 相机平滑跟随玩家
    - 使用线性插值（lerp）实现平滑跟随

    - 相机位置: `player.pos + offset(0, 4, 14)`
    - 相机朝向玩家上方0.6单位处

- **第一人称视角**: 相机位于玩家位置

    - 位置: `player.pos + (0, 0.6, -0.6)`
    - 鼠标控制yaw/pitch角度
    - 限制pitch范围(-89°, 89°)

4. 事件处理

- **键盘回调**: 记录按键状态到`keys[1024]`数组
- **鼠标回调**:
    - 鼠标移动：计算yaw/pitch，更新`cameraFront`
    - 鼠标滚轮：第三人称时调整FOV（视野角度）
- **视角切换**: V键在第一人称和第三人称间切换

5. 主循环

```
1. 处理事件（glfwPollEvents）
2. 计算时间差（dt）
3. 更新游戏逻辑（如果状态为PLAYING）
4. 设置相机和投影矩阵
5. 渲染游戏场景
6. 渲染UI覆盖层
7. 交换缓冲区
```

2.2 Game.cpp/h - 游戏核心逻辑

核心职责：

- 游戏对象管理（玩家、掉落物体、地板）
- **碰撞检测系统**（核心功能）
- 物体生成和生命周期管理
- 阴影贴图渲染
- 游戏状态重置

关键数据结构：

Falling结构体

```
struct Falling {
    glm::vec3 pos;          // 世界空间位置
    glm::vec3 vel;          // 速度向量
    glm::vec3 color;        // 颜色（用于渲染）
    bool alive;             // 是否存活（参与碰撞检测）
    float rot;              // 当前旋转角度（弧度）
    glm::vec3 rotAxis;      // 旋转轴
    float rotSpeed;         // 旋转速度（弧度/秒）
    glm::vec3 modelScale;   // 模型缩放
    glm::mat4 modelMatrix;  // 世界变换矩阵
    glm::vec3 halfExtents;  // 包围盒半长
    int modelIndex;         // 使用的模型索引（0-2）
};
```

OBB结构体（有向包围盒）

```
struct OBB {
    glm::vec3 center;   // 世界空间中心点
    glm::vec3 axis[3];  // 三个正交轴（单位向量）
    float half[3];      // 沿每个轴的半长度
};
```

主要函数：

LoadResources()

- 加载3种掉落物体模型（bucket, jar, teapot）
- 加载地板模型
- 设置模型缩放
- 计算地板顶部Y坐标（用于碰撞检测）

Reset()

- 清空掉落物体列表
- 重置玩家位置到原点
- 重置生成计时器
- 设置玩家存活状态

SpawnObject()

- 随机生成掉落物体
- 随机位置：X/Z范围(-4, 4)，Y范围(4, 7)
- 随机速度：水平方向(-1, 1)，垂直方向(0.5, -1)
- 随机旋转：角度、轴、速度
- 随机选择模型（0-2）

Update() - 游戏逻辑更新

1. 更新玩家位置（限制在地板范围内）
2. 生成新物体（定时器触发）
3. 更新玩家模型矩阵
4. **碰撞检测**（详见碰撞检测章节）
5. 物理更新（重力、位置、旋转）
6. 清理死亡物体

Render() - 渲染

1. **Shadow Pass**: 从光源视角渲染深度图
2. **Main Pass**: 正常渲染所有物体
    - 设置光照参数（方向光）
    - 绑定阴影贴图
    - 渲染地板、玩家、掉落物体

2.3 Player.cpp/h - 玩家控制

核心职责：

- 根据键盘输入移动玩家
- 基于相机方向计算移动方向

移动逻辑：

```
void Player::Update(float dt, const bool keys[1024],
                    const glm::vec3 &cameraFront,
                    const glm::vec3 &cameraUp)
{
    // 提取水平前方向量（去除Y分量）
    glm::vec3 forward = glm::normalize(glm::vec3(cameraFront.x, 0.0f,
cameraFront.z));
```

```
    // 计算右方向量
    glm::vec3 right = glm::normalize(glm::cross(forward, cameraUp));

    // 根据按键移动
    if (keys[GLFW_KEY_W]) pos += speed * cameraFront;  // 向前
    if (keys[GLFW_KEY_S]) pos -= speed * cameraFront;  // 向后
    if (keys[GLFW_KEY_A]) pos -= right * speed;        // 向左
    if (keys[GLFW_KEY_D]) pos += right * speed;        // 向右

    // 固定Y高度
    pos.y = groundY;
}
```

关键点：

- 移动速度：`moveSpeed = 5.0f`
- 固定高度：`groundY = 0.5f`
- 移动方向基于相机朝向，实现自由移动

2.4 StaticModel.cpp/h - 3D模型加载和渲染

核心职责：

- 使用Assimp库加载多种3D模型格式
- 解析模型网格、纹理、材质
- 计算模型包围盒
- 渲染模型（支持纹理、透明、阴影）

模型加载流程：

1. Assimp导入

```
Assimp::Importer importer;
const aiScene* scene = importer.ReadFile(path,
    aiProcess_Triangulate |           // 三角化
    aiProcess_GenSmoothNormals |      // 生成法线
    aiProcess_FlipUVs |               // 翻转UV
    aiProcess_CalcTangentSpace |      // 计算切空间
    aiProcess_JoinIdenticalVertices | // 合并相同顶点
    aiProcess_OptimizeMeshes |        // 优化网格
    aiProcess_PreTransformVertices    // 预变换顶点
);
```

2. 网格处理

- 提取顶点数据（位置、法线、UV）
- 提取索引数据（面片）
- 创建OpenGL缓冲区（VAO/VBO/EBO）

3. 材质处理

- 加载漫反射纹理
- 检测Alpha通道（透明支持）
- 特殊处理毛发材质（alpha混合 + 深度写入禁用）
- 智能纹理路径查找（多个备选路径）

4. 包围盒计算

```
void ComputeBBoxRecursive(aiNode* node, const aiScene* scene,
                          const glm::mat4& parentTransform)
```

- 递归遍历场景图
- 变换每个顶点到世界空间
- 计算最小包围盒（bboxMin/bboxMax）

渲染特性：

- **Alpha测试**: 用于透明物体（如玩家模型）
- **Alpha混合**: 用于毛发材质
- **阴影支持**: DrawDepth()用于阴影贴图渲染

2.5 Audio.cpp/h - 音频系统

核心职责：

- 使用OpenAL管理3D音频
- 加载和播放WAV文件
- 支持循环播放

实现细节：

初始化

```
device = alcOpenDevice(nullptr);           // 打开默认音频设备
context = alcCreateContext(device, nullptr); // 创建上下文
alcMakeContextCurrent(context);            // 设置为当前上下文
```

WAV加载

- 使用dr_wav.h库解码WAV文件
- 支持单声道/立体声
- 转换为16位PCM格式

播放

- 创建OpenAL源（source）
- 绑定缓冲区（buffer）
- 设置循环属性
- 开始播放

2.6 Shader.h - 着色器管理

核心职责：

- 加载、编译、链接顶点和片段着色器
- 提供便捷的uniform设置函数

关键功能：

着色器编译

```
1. 读取着色器文件
2. 创建着色器对象（glCreateShader）
3. 编译着色器（glCompileShader）
4. 检查编译错误
5. 创建程序对象（glCreateProgram）
6. 附加并链接着色器
7. 检查链接错误
```

Uniform设置

- `setMat4()`: 矩阵4x4
- `setVec3()`: 3D向量
- `setFloat()`: 浮点数
- `setInt()`: 整数
- 自动获取uniform位置

2.7 TextRenderer.cpp/h - 文本渲染

核心职责：

- 使用stb_truetype加载TTF字体
- 将字体烘焙到位图纹理（Atlas）
- 渲染文本字符串到屏幕

实现原理：

字体加载

1. 读取TTF文件到内存
2. 使用`stbtt_BakeFontBitmap()`烘焙字符到位图
3. 创建OpenGL纹理（GL_RED格式，单通道）
4. 存储每个字符的UV坐标和位置信息

文本渲染

1. 遍历字符串每个字符
2. 使用`stbtt_GetBakedQuad`获取字符四边形
3. 将像素坐标转换为NDC坐标
4. 生成顶点数据（位置+UV）
5. 动态上传到VBO并绘制

特性：

- 支持ASCII字符（32-127）
- 使用NDC坐标系统
- Alpha混合支持

2.8 UI.cpp/h - 用户界面

核心职责：

- 管理菜单按钮（主菜单、游戏结束菜单）
- 处理鼠标交互（点击、悬停）
- 渲染按钮和文字

按钮系统：

按钮结构

```
struct UIButton {
    float cx, cy;       // 中心点（NDC坐标）
    float w, h;         // 宽度和高度
    std::string label; // 按钮文字
    bool hovered;       // 是否悬停
};
```

交互检测

```
bool PointInRectNDC(float px, float py, const UIButton& b)
```

- 检测鼠标位置是否在按钮矩形内
- 使用NDC坐标系统

动作系统

- `outAction = 0`: 无动作
- `outAction = 1`: 开始游戏
- `outAction = 2`: 退出
- `outAction = 3`: 重试

三、碰撞检测系统深度解析

3.1 碰撞检测架构

游戏采用**两阶段碰撞检测**策略，平衡了准确性和性能：

```
第一阶段：Broad Phase（粗检测）
    ↓ 使用包围球快速剔除
第二阶段：Narrow Phase（精确检测）
    ↓ 使用OBB+SAT精确检测
```

3.2 OBB（有向包围盒）详解

为什么使用OBB而不是AABB?

AABB（轴对齐包围盒）：

- 只能表示轴对齐的盒子
- 对于旋转的物体，包围盒会变大很多
- 导致大量误判

OBB（有向包围盒）：

- 可以表示任意旋转的盒子
- 更贴合旋转后的模型形状
- 精确度高，误判少

OBB数据结构

```
struct OBB {
    glm::vec3 center;    // 世界空间中心点
    glm::vec3 axis[3];   // 三个正交轴（单位向量）
    float half[3];       // 沿每个轴的半长度
};
```

几何意义：

- OBB可以看作是由中心点和三个正交轴定义的长方体
- 每个轴有自己的半长，决定了盒子在该方向上的大小

3.3 BuildOBBFromModel() - 从模型矩阵构建OBB

函数目的： 将模型的局部包围盒（bboxMin/bboxMax）通过模型矩阵变换到世界空间，得到OBB。

实现步骤：

步骤1：计算局部中心点和半长

```
glm::vec3 localCenter = (bboxMin + bboxMax) * 0.5f;
glm::vec3 localHalf = (bboxMax - bboxMin) * 0.5f;
```

步骤2：变换中心点到世界空间

```
glm::vec3 worldCenter = glm::vec3(modelMatrix * glm::vec4(localCenter,
1.0f));
```

- 使用齐次坐标（w=1）进行变换
- 只取xyz分量得到3D坐标

步骤3：提取线性变换部分

```
glm::mat3 M3 = glm::mat3(modelMatrix);
```

- 提取3x3左上角矩阵（旋转+缩放）
- 忽略平移部分（已用于中心点变换）

步骤4：变换三个轴方向

```
// X轴
glm::vec3 ux = glm::vec3(M3 * glm::vec3(1.0f, 0.0f, 0.0f));
float lenx = glm::length(ux);
obb.axis[0] = (lenx > 1e-6f) ? ux / lenx : glm::vec3(1, 0, 0);
obb.half[0] = lenx * localHalf.x;
```

```
// Y轴（同理）
// Z轴（同理）
```

数学原理：

- 局部坐标系的单位向量(1,0,0), (0,1,0), (0,0,1)经过M3变换后
- 得到的向量的**方向**就是世界空间的轴方向
- 向量的**长度**就是缩放因子
- 归一化得到单位轴向量
- 长度乘以局部半长得到世界空间半长

退化处理：

- 如果某个轴长度为0（或极小），使用默认轴
- 避免除零错误

3.4 SAT（分离轴定理）算法详解

理论基础

分离轴定理（Separating Axis Theorem）：

> 对于两个凸体，如果它们不相交，则存在至少一条分离轴（separating axis），使得两个物体在该轴上的投影不重叠。

逆定理：

> 如果两个凸体相交，则不存在分离轴（所有轴上的投影都重叠）。

算法流程

对于两个OBB A和B，需要测试15条轴：

1. **OBB A的3个轴方向** (A.axis[0], A.axis[1], A.axis[2])
2. **OBB B的3个轴方向** (B.axis[0], B.axis[1], B.axis[2])
3. **两个OBB轴的叉积** (3×3 = 9条)

为什么需要叉积轴?

考虑两个OBB斜着摆放的情况：

```
    B
   /|
  / |
 /  |
A   |
 \  |
  \ |
   \|
```

仅使用各自的轴可能无法找到分离轴，叉积轴可以检测这种情况。

投影半径计算

核心函数：

```
auto projectIntervalRadius = [](const OBB &O, const glm::vec3 &axis) ->
float {
    float r = 0.0f;
    r += O.half[0] * fabs(glm::dot(O.axis[0], axis));
    r += O.half[1] * fabs(glm::dot(O.axis[1], axis));
    r += O.half[2] * fabs(glm::dot(O.axis[2], axis));
    return r;
};
```

数学推导：

OBB可以表示为：

```
O = { center + a*axis[0] + b*axis[1] + c*axis[2] }
    其中 a ∈ [-half[0], half[0]]
         b ∈ [-half[1], half[1]]
         c ∈ [-half[2], half[2]]
```

在测试轴`axis`上的投影为：

```
投影范围 = [center·axis - r, center·axis + r]
其中 r = half[0]*|axis[0]·axis| + half[1]*|axis[1]·axis| +
half[2]*|axis[2]·axis|
```

几何意义：

- `|axis[i]·axis|`是OBB第i个轴在测试轴上的投影长度

- 乘以半长得到该方向上的贡献
- 三个方向的贡献之和就是OBB在测试轴上的投影半径

分离检测

```
auto testAxis = [&](const glm::vec3 &axis) -> bool {
    // 归一化测试轴
    glm::vec3 axisN = normalize(axis);

    // 计算两个中心点在测试轴上的距离
    float dist = fabs(glm::dot(T, axisN));  // T = B.center - A.center

    // 计算两个OBB的投影半径
    float ra = projectIntervalRadius(A, axisN);
    float rb = projectIntervalRadius(B, axisN);

    // 判断是否分离
    return dist <= (ra + rb) + 1e-6f;  // 重叠返回true
};
```

判断逻辑：

```
如果 dist > (ra + rb):
    → 两个投影区间分离
    → 存在分离轴
    → 两个OBB不相交
    → 返回false（找到分离轴）

如果 dist <= (ra + rb):
    → 两个投影区间重叠
    → 该轴不是分离轴
    → 继续测试下一条轴
```

所有轴都测试完：

- 如果任何一条轴返回false → 不相交
- 如果所有轴都返回true → 相交

性能优化

1. **提前退出**：找到第一条分离轴即可返回false
2. **包围球预检测**：先用简单的包围球测试剔除远离的物体
3. **轴归一化缓存**：OBB的轴已经归一化，可以直接使用

3.5 碰撞检测完整流程

在`Game::Update()`中实现：

步骤1：构建玩家OBB（每帧一次，在循环外）

```
OBB playerOBB = BuildOBBFromModel(playerModel.bboxMin,
                                  playerModel.bboxMax,
                                  player.modelMatrix);
float playerSphereR = computeBoundingSphereRadius(...);
```

步骤2：对每个掉落物体循环

a) 物理更新

```
// 重力
o.vel += glm::vec3(0.0f, -9.8f * dt * 0.2f, 0.0f);
// 位置更新
o.pos += o.vel * dt;
// 旋转更新
o.rot += o.rotSpeed * dt;
```

b) 更新模型矩阵

```
glm::mat4 mm(1.0f);
mm = glm::translate(mm, o.pos);           // 平移
mm = glm::rotate(mm, o.rot, o.rotAxis);  // 旋转
mm = glm::scale(mm, o.modelScale);        // 缩放
o.modelMatrix = mm;
```

c) 构建物体OBB

```
OBB objOBB = BuildOBBFromModel(fallingModels[o.modelIndex].bboxMin,
                               fallingModels[o.modelIndex].bboxMax,
                               o.modelMatrix);
float objSphereR = glm::length(glm::vec3(objOBB.half[0], ...));
```

d) Broad Phase - 包围球测试

```
float centersDist = glm::length(objOBB.center - playerOBB.center);
if (centersDist <= (playerSphereR + objSphereR)) {
    // 进入精确检测
}
```

为什么需要Broad Phase?

- SAT算法需要测试15条轴，计算量较大
- 包围球测试只需计算一次距离，非常快速
- 可以快速剔除大部分不相关的物体
- 只有可能碰撞的物体才进行精确检测

包围球半径计算：

```
// 使用OBB的半长向量长度作为包围球半径
float radius = glm::length(glm::vec3(half[0], half[1], half[2]));
```

这是OBB的外接球半径，保证包围球完全包含OBB。

e) Narrow Phase - SAT精确检测

```
if (OBBIntersectSAT(playerOBB, objOBB)) {
    playerDead = true;  // 碰撞发生
    std::cout << "[Collide] player hit by falling object\n";
    o.alive = false;
    break;  // 找到碰撞即可退出
}
```

f) 地面碰撞检测

```
float objBottomY = objOBB.center.y - objOBB.half[1];
const float EPS = 1e-4f;
if (objBottomY <= floorTop + EPS) {
    // 物体接触地面
    // 对齐到地面
    o.pos.y = floorTop + objOBB.half[1];

    // 更新模型矩阵
    glm::mat4 mm(1.0f);
    mm = glm::translate(mm, o.pos);
    mm = glm::rotate(mm, o.rot, o.rotAxis);
    mm = glm::scale(mm, o.modelScale);
    o.modelMatrix = mm;
```

```
        // 停止运动
        o.vel = glm::vec3(0.0f);
        o.alive = false;  // 不再参与碰撞检测
    }
```

关键点：

- 使用OBB底部Y坐标检测地面接触
- 使用小阈值EPS避免浮点误差
- 对齐物体到底面，避免穿透
- 落地后标记为死亡，不再参与碰撞检测

步骤3：清理死亡物体

```
falling.erase(std::remove_if(falling.begin(), falling.end(),
                             [](const Falling &f) { return !f.alive; }),
              falling.end());
```

3.6 碰撞检测性能分析

时间复杂度：

- Broad Phase: O(n) - n个物体，每个物体一次距离计算
- Narrow Phase: O(k×15) - k个候选物体，每个15次轴测试
- 总体: O(n + k×15)，其中k << n

空间复杂度：

- OBB存储: O(1)每个物体
- 总体: O(n)

实际性能：

- 大部分物体在Broad Phase被剔除
- 通常只有1-3个物体进入Narrow Phase
- 帧率保持在60fps以上

3.7 与其他碰撞检测方法对比

方法	精度	性能	实现难度	适用场景
AABB	低（旋转后误差大）	高	低	静态场景
OBB+SAT	**高**	**中**	**中**	**本游戏**
包围球	最低	最高	最低	Broad Phase
GJK算法	高	中	高	复杂凸体
像素级检测	最高	最低	低	特殊需求

四、核心算法原理

4.1 线性插值（Lerp）- 相机平滑跟随

实现：

```
glm::vec3 desiredPos = game.player.pos + offset;
float followSpeed = 6.0f;
float t = glm::clamp(followSpeed * dt, 0.0f, 1.0f);
smoothCamPos = glm::mix(smoothCamPos, desiredPos, t);
```

原理：

- 使用GLM的`mix`函数实现线性插值
- `t`是插值系数，基于时间步长和速度
- 限制t在[0,1]范围，避免过度插值
- 实现平滑的相机跟随效果

4.2 物理模拟 - 重力和运动

重力模拟：

```
o.vel += glm::vec3(0.0f, -9.8f * dt * 0.2f, 0.0f);
```

- 标准重力加速度：-9.8 m/s²
- 乘以0.2作为时间缩放因子（游戏速度）

- 每帧累加到速度

位置积分：

```
o.pos += o.vel * dt;
```

- 欧拉积分：`x = x + v*dt`
- 简单但有效

旋转动画：

```
o.rot += o.rotSpeed * dt;
```

- 每帧增加旋转角度
- 实现物体下落时的旋转效果

4.3 阴影贴图算法

Shadow Mapping原理：

1. **Shadow Pass（深度渲染）**
    - 从光源视角渲染场景
    - 只记录深度值到深度纹理
    - 使用正交投影（方向光）
2. **Main Pass（正常渲染）**
    - 从相机视角渲染场景
    - 将片段位置变换到光源空间
    - 比较片段深度和阴影贴图深度
    - 如果片段深度大于阴影贴图深度 → 在阴影中

实现细节：

```
// 计算光源VP矩阵
glm::mat4 lightVP = lightProj * lightView;

// Shadow Pass
```

```
glBindFramebuffer(GL_FRAMEBUFFER, depthFBO);
glViewport(0, 0, SHADOW_SIZE, SHADOW_SIZE);
glClear(GL_DEPTH_BUFFER_BIT);
// 渲染所有物体到深度纹理

// Main Pass
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glBindTexture(GL_TEXTURE_2D, depthMap);
// 正常渲染，片段着色器中采样阴影贴图
```

优化技巧：

- 使用多边形偏移（Polygon Offset）减少阴影痤疮
- 使用PCF（Percentage Closer Filtering）实现软阴影
- 使用阴影贴图级联（CSM）处理大场景

五、代码架构设计

5.1 整体架构

```
main.cpp（入口层）
    ├── 窗口和上下文管理
    ├── 事件处理
    └── 主循环
        │
        ├── Game（游戏逻辑层）
        │    ├── Player（玩家控制）
        │    ├── StaticModel（模型加载）
        │    ├── 碰撞检测系统
        │    └── 物理模拟
        │
        ├── UI（界面层）
        │    └── TextRenderer（文字渲染）
        │
        ├── Audio（音频层）
        └── Shader（着色器管理层）
```

5.2 设计模式

1. 单一职责原则

- 每个类负责一个特定功能

- `Player`: 只负责玩家控制
- `StaticModel`: 只负责模型加载和渲染
- `Audio`: 只负责音频管理

2. RAII（资源获取即初始化）

- OpenGL资源自动管理
- 析构函数中释放资源
- 避免内存泄漏

3. 状态机模式

```
enum class State { MENU, PLAYING, GAMEOVER };
```

- 清晰的游戏状态管理
- 状态转换逻辑集中

4. 策略模式（碰撞检测）

- Broad Phase和Narrow Phase可替换
- 易于扩展新的碰撞检测方法

5.3 数据流向

更新流程：

```
输入事件 → 状态更新 → 物理模拟 → 碰撞检测 → 渲染
```

渲染流程：

```
Shadow Pass → Main Pass → UI Overlay → 交换缓冲区
```

5.4 内存管理

资源管理：

- 模型：使用智能指针或RAII管理
- 纹理：GLuint自动管理
- 音频：OpenAL自动管理
- 着色器：编译时创建，程序结束时销毁

性能优化：

- 对象池（掉落物体列表复用）
- 延迟删除（标记死亡，下一帧清理）
- 预分配容器大小

六、关键代码位置索引

碰撞检测相关

- **OBB结构定义**: `Game.cpp:18-23`
- **BuildOBBFromModel()**: `Game.cpp:26-64`
- **OBBIntersectSAT()**: `Game.cpp:66-118`
- **碰撞检测调用**: `Game::Update()` 中 `Game.cpp:342-394`
- **地面碰撞检测**: `Game.cpp:396-417`

物理模拟

- **重力更新**: `Game.cpp:359`
- **位置积分**: `Game.cpp:360`
- **旋转更新**: `Game.cpp:362`

渲染相关

- **Shadow Pass**: `Game.cpp:428-502`
- **Main Pass**: `Game.cpp:504-588`
- **模型渲染**: `StaticModel.cpp:345-413`

玩家控制

- **移动逻辑**: `Player.cpp:9-30`
- **位置限制**: `Game.cpp:307-311`

七、总结

本项目实现了一个完整的3D游戏系统，核心亮点包括：

1. **高效的碰撞检测系统**
    - 两阶段检测策略（Broad Phase + Narrow Phase）
    - OBB + SAT算法实现精确碰撞
    - 性能优化良好
2. **完善的物理模拟**
    - 重力系统
    - 运动积分
    - 旋转动画
3. **现代OpenGL渲染**
    - 阴影贴图
    - Phong光照模型
    - 透明和Alpha测试
4. **良好的代码架构**
    - 模块化设计
    - 清晰的职责划分
    - 易于扩展和维护

技术亮点：

- 使用工业级算法（SAT）确保碰撞检测的准确性
- 两阶段检测平衡了性能和精度
- 完整的3D渲染管线
- 现代C++特性应用

这份代码展示了从底层数学到高级渲染的完整游戏开发流程，是一个优秀的OpenGL学习项目。

文档生成时间: 2024年 **项目版本**: OpenGL 3.3 Core Profile **作者**: [项目开发者]