基于自研 **OpenGL** 引擎的 **Portal** 类三维解谜游戏系统

计算机图形学课程大作业技术报告

徐易天，曾翔

January 6, 2026

Contents

1 引言与项目概述

本项目目标是实现一个可游玩的 Portal 风格三维解谜关卡：玩家在实验室场景中移动探索，拾取方块并放置于按钮触发机关，利用两扇相互连接的传送门穿越空间并最终到达终点。

在图形学实现上，我们未使用 Unity/Unreal 等商业引擎，而是基于 OpenGL Core Profile + C++ 自行搭建渲染与交互框架，完成了模型加载、实时渲染、第一人称漫游、碰撞检测、触发器逻辑、传送门视窗渲染与传送等核心能力。

本文按‘玩法与需求 → 系统架构 → Portal 关键渲染技术 → 物理与交互 → 工程调试 → 结果展示’组织，并在关键处给出代码与截图证据。

1.1 玩法流程与操作说明

- 移动：W/A/S/D；跳跃：Space；视角：鼠标移动；
- 拾取/放下方块：E（对准可拾取方块后按 E 进入抓取；再次按 E 放下）；
- 发射传送门：鼠标左键/右键分别放置 Portal A / Portal B；
- 抓取状态下左键：对抓取物体施加向前推力并释放（用于抛掷/推动方块）；
- 重置：T（将玩家位置重置到原点并清零速度，用于快速回到可测试状态）。
- 截图：P

1.2 自查表

完成情况如下表所示

Table 1: 大作业评分项完成情况自查表

分类	评分项要求	本项目实现方案与对应章节	状态
基础项	基本体素建模	采用网格模型（OBJ）完成基础场景构建与几何表达（Cube/PortalGun/Wall），并支持矩阵变换	✓
	网格导入导出	自研 OBJ 解析器（见 3.2），支持复杂场景加载	✓
	材质纹理显示	支持 MTL 材质解析与纹理贴图（见 3.1）	✓
	几何变换	支持平移、旋转、缩放（TRS）矩阵变换（见 2.1）	✓
	光源照明编辑	支持多光源光照模型与参数配置（代码/配置），可在运行时切换或调整	✓
	场景漫游	第一人称 FPS 漫游（WASD 移动 + 鼠标视角）（见 1.1）	✓
	动画与截图	实现运行时物理驱动的程序化动画（抓取弹簧/后坐力/机关联动）及截图保存（见 **4.6**）	✓
加分项	三维游戏	完整 **Portal** 解谜玩法（传送/重力枪/机关/胜利判定）	**EX**
	实时碰撞	实现 **SAT**（分离轴）**OBB** 精确碰撞检测（见 **5.2**）	**EX**
	对象表达	实现传送门（**Portal**）、触发器（**Trigger**）、翻转墙等高级对象	**EX**
	高级渲染	实现递归帧缓冲（**Recursive FBO**）与斜投影裁剪（见 **4.5**）	**EX**

2 系统总体架构

系统采用模块化、面向对象设计，将输入、逻辑、物理、渲染解耦，主循环按“输入 → 更新 → 物理/触发 → 渲染”的顺序驱动。

2.1 核心模块划分

1. **Application**：窗口与 OpenGL 上下文创建，主循环（deltaTime 计算、调用 update/render）。
2. **Scene / GameObject**：场景管理与对象容器，统一管理模型资源、对象实例、Portal、Trigger、光源等。
3. **Renderer**：封装 Shader、FBO、Portal 多次渲染、材质绑定与绘制流程。
4. **Model/Mesh/Texture**：OBJ/MTL 解析、顶点数据组织、纹理加载与缓存。
5. **PhysicsSystem**：刚体、重力、碰撞检测（SAT OBB）、raycast、触发器检测。
6. **Player / PortalGun / Button / Flip / Trigger**：玩家控制、抓取系统、按钮触发器与翻转机关（Flip）、终点判定等玩法对象。

2.2 运行主循环结构

主循环体现实时交互程序的经典结构：计算帧间隔 dt，处理输入、更新逻辑、渲染一帧并交换缓冲。

Listing 1: src/Application.cpp：主循环骨架（节选）

```
while (!glfwWindowShouldClose(window)) {
    float currentFrame = (float)glfwGetTime();
    deltaTime = currentFrame - lastFrame;
    lastFrame = currentFrame;

    // Clamp dt to avoid physics explosions
    if (deltaTime > 0.1f) deltaTime = 0.1f;

    // 1) input
    input.update();
    processInput(deltaTime);

    // 2) update
    Camera &activeCamera = getActiveCamera();
    scene->update(deltaTime, activeCamera);

    // 3) render
    renderer->render(*scene, activeCamera);

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

3 资源管线：Blender 建模、OBJ/MTL 解析与场景构建

3.1 建模与资源组织

所有场景与道具模型由 Blender 自行建模导出 OBJ，材质信息通过 MTL 关联，纹理以图片文件形式存储并在加载阶段建立缓存。

本项目资源目录结构如下（与工程一致）：

- `resources/obj/level/`：关卡墙体、按钮（button_flip / button_goal）、可移动方块等
- `resources/obj/portal_gun/`：Portal Gun 模型
- `resources/obj/portal_cube/`：可拾取方块模型
- `resources/texture/`：贴图
- `resources/shader/`：顶点/片元着色器

3.2 OBJ 解析：顶点/法线/UV 与面索引三角化

我们没有使用第三方 OBJ 库，而在 `Model` 中实现了基础解析：读取 v/vn/vt 与 f。对 f（多边形面）采用扇形三角化（Fan Triangulation）转为三角形列表，并建立“唯一顶点”索引缓存以复用数据。

Listing 2: src/Model.cpp：读取 f 并进行三角化（节选）

```
else if (prefix == "f") {
    // parse all vertices in this face
    std::vector<unsigned int> faceIndices;
    std::string vStr;
    while (ss >> vStr) {
        // parse "v/vt/vn"
        unsigned int vIdx=0, vtIdx=0, vnIdx=0;
        // ... split by '/'
        // push unique vertex index
        faceIndices.push_back(getOrCreateUniqueIndex(vIdx, vtIdx, vnIdx));
    }

    // fan triangulation: (0,i,i+1)
    for (size_t i = 1; i + 1 < faceIndices.size(); ++i) {
        indices.push_back(faceIndices[0]);
        indices.push_back(faceIndices[i]);
        indices.push_back(faceIndices[i+1]);
    }
}
```

3.3 模型包围盒与 OBB 构建

为了服务碰撞检测与触发器，模型加载后计算 AABB（min/max bound），并在对象层根据缩放/旋转构建 OBB（有向包围盒）。这样可以更准确地区分不同形状物体的碰撞范围。

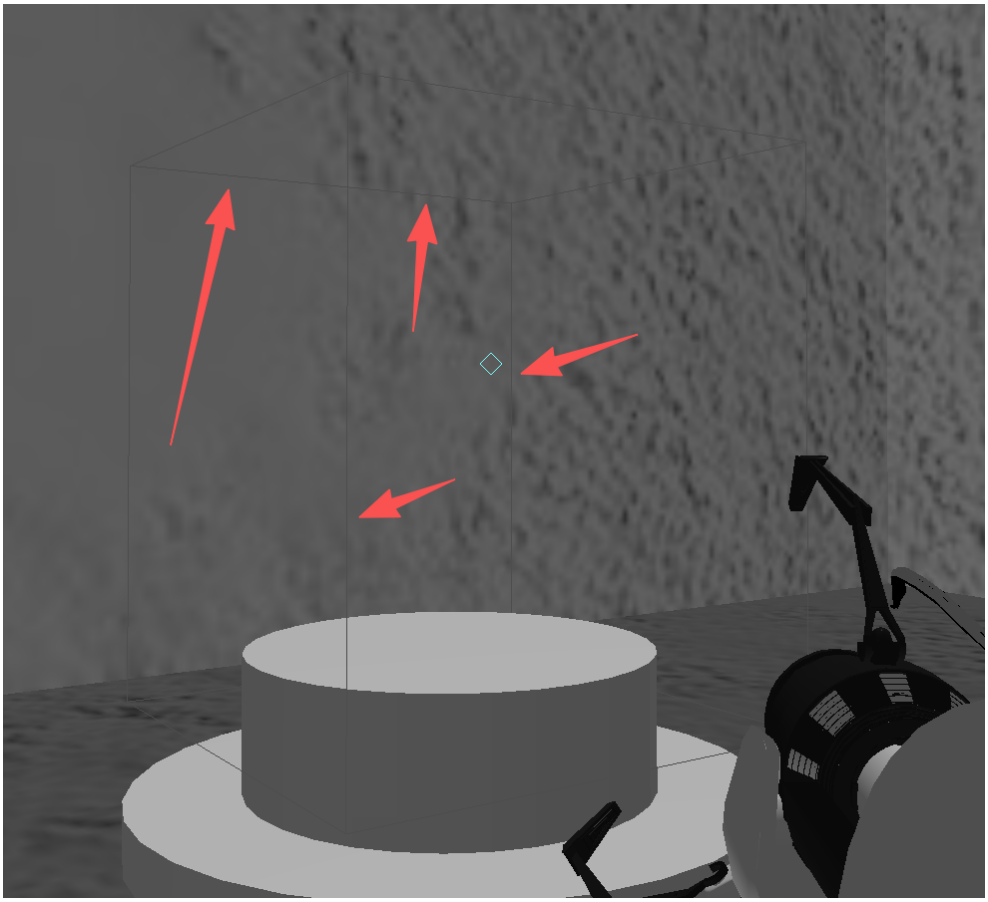


Figure 1: 图中线框为场景中的触发器（Trigger）感应范围。Trigger 系统底层复用了物理引擎的 OBB 数据结构与求交算法，此处通过可视化 Debug 视图验证了 OBB 空间计算的正确性。

3.4 物理穿梭与碰撞剔除 (Collision Culling)

除了视觉上的连通，Portal 机制还要求在物理上允许玩家“穿过”原本存在的墙壁。若不处理碰撞，玩家会被传送门背后的墙体阻挡。为此，我们结合 Trigger 系统与位掩码 (Bitmask) 技术实现了动态碰撞剔除。

3.4.1 基于掩码的碰撞过滤

我们在 `PhysicsSystem` 中引入了碰撞掩码机制。仅当两个物体的掩码进行按位与运算结果非零时（$(\text{mask}_A \& \text{mask}_B) \neq 0$），才进行碰撞解算。

Listing 3: src/PhysicsSystem.cpp：基于掩码的碰撞过滤

```
// 物理引擎中的碰撞预判
if ((a.rigidBody->collisionMask & b.rigidBody->collisionMask) == 0)
    continue; // 掩码不匹配，忽略碰撞，允许穿透
```

3.4.2 动态掩码切换逻辑

当传送门生成时，会将附着的墙壁标记为 `PORTAL_ON`。同时，我们在传送门前方设置了一个 `NearTrigger`。当玩家或物体进入该区域时，触发器回调函数会将其碰撞掩码修改为 `NEAR_PORTAL`，使其与墙壁的掩码运算结果为 0，从而实现“穿墙”。

Listing 4: src/Portal.cpp：利用 Trigger 动态修改碰撞掩码

```
// Portal 初始化时设置近距离触发器
nearTrigger->onEnter = [](GameObject *obj) {
    // 进入传送门附近，修改掩码，允许穿过墙壁
    obj->setCollisionMask(COLLISION_MASK_NEARPORTAL);
};

nearTrigger->onExit = [](GameObject *obj) {
    // 离开传送门区域，恢复默认碰撞掩码
    obj->setCollisionMask(COLLISION_MASK_DEFAULT);
};
```

通过这种机制，我们确保了玩家仅在传送门开启且靠近时才能穿过墙壁，而在其他情况下依然受物理碰撞限制。

4 渲染系统设计与实现细节

4.1 基础渲染管线

渲染模块负责：

- Shader 管理：统一编译/链接、设置 uniform（模型矩阵、视图矩阵、投影矩阵、光源参数）。
- 深度测试、面剔除、混合：保证室内场景遮挡关系正确；必要物体支持透明混合。
- 纹理绑定与材质：从 MTL/Texture 管线读取 diffuse/specular 等贴图并绑定。

渲染管线逻辑流程：

为了实现传送门的“画中画”效果，我们的渲染管线并非简单的单次绘制，而是采用了多遍渲染架构。具体流程如下：

1. 预处理阶段 **(Portal Pass)**：
    - 遍历场景中的传送门，判断是否需要递归渲染。
    - 计算 虚拟摄像机矩阵：将主摄像机相对于“入口门”的位姿变换到“出口门”空间。
    - 离屏渲染：绑定 FBO (Frame Buffer Object)，将虚拟视角的场景渲染到纹理。在此过程中启用 斜投影裁剪以防止物体遮挡。
2. 主渲染阶段 **(Main Pass)**：
    - 切换回默认帧缓冲 (Default Framebuffer)。
    - 绘制主视角的实验室场景（墙壁、方块等）。
    - 绘制传送门网格：此时将预处理阶段生成的 FBO 纹理作为材质绑定到传送门表面，从而实现空间连通的视觉效果。

4.2 帧缓冲（FBO）封装

Portal 视窗渲染依赖离屏渲染到纹理。我们封装了 `FrameBuffer`：创建 FBO、颜色纹理附件、深度模板 RBO，并提供 Bind/Unbind/Rescale 接口。

Listing 5: src/FrameBuffer.cpp：FBO 初始化（节选）

```
glGenFramebuffers(1, &fbo);
glBindFramebuffer(GL_FRAMEBUFFER, fbo);

// color texture
glGenTextures(1, &textureColorBuffer);
glBindTexture(GL_TEXTURE_2D, textureColorBuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
            GL_RGB, GL_UNSIGNED_BYTE, NULL);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                      GL_TEXTURE_2D, textureColorBuffer, 0);

// depth-stencil renderbuffer
glGenRenderbuffers(1, &rbo);
glBindRenderbuffer(GL_RENDERBUFFER, rbo);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, width, height);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
                         GL_RENDERBUFFER, rbo);
```

4.3 Portal 视窗渲染：虚拟摄像机与视图矩阵变换

Portal 的“看见另一扇门后的世界”，本质是把主相机通过入口门坐标系映射到出口门坐标系，得到一台虚拟相机，从该虚拟相机渲染场景到纹理，再把纹理贴到入口门表面。

我们在 `Portal::getTransformedView` 中实现了该变换（包含 180 度翻转以保证朝向一致）。

Listing 6: src/Portal.cpp：计算 Portal 变换后的视图矩阵（原代码节选）

```
glm::mat4 Portal::getTransformedView(glm::mat4 view) {
    glm::mat4 camTransform = glm::inverse(view);

    glm::mat4 rotation180 = glm::rotate(glm::mat4(1.0f),
        glm::radians(180.0f), glm::vec3(0, 1, 0));

    glm::mat4 destView = otherModel * rotation180
        * glm::inverse(myModel) * camTransform;

    return glm::inverse(destView);
}
```

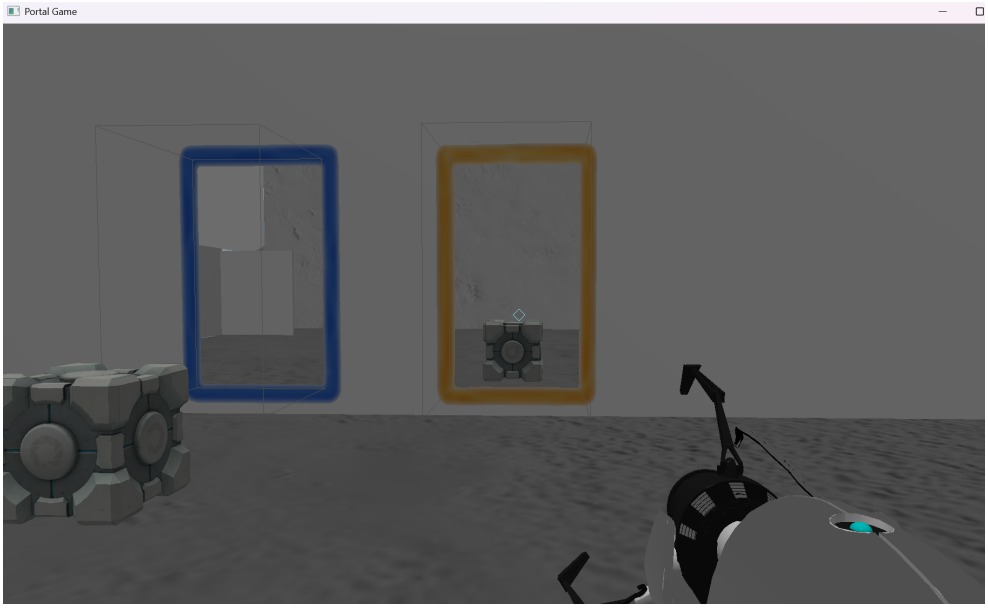


Figure 2: 透过 Portal A 观察到的画面，实际上是基于虚拟摄像机在出口门 Portal B 位置渲染的结果。矩阵变换保证了视线穿过门后的几何连续性。

4.4 斜投影裁剪（Oblique Projection Clipping）

如果仅用普通投影矩阵渲染 Portal 纹理，会出现“门后墙体遮挡/穿帮”等问题。为此我们根据门平面构造裁剪平面，并对投影矩阵近裁剪面做斜投影修改，使得虚拟相机只渲染门平面另一侧的内容。

该部分在 `Renderer` 中实现，核心是构造 q 与 c 并替换投影矩阵的第三列。

Listing 7: src/Renderer.cpp：斜投影裁剪矩阵修改（原代码节选）

```
glm::vec4 viewPlane = srcPortal.getPlaneEquation(destView);

glm::vec4 q = glm::inverse(obliqueProjection) * glm::vec4(
    (viewPlane.x > 0 ? 1 : -1),
    (viewPlane.y > 0 ? 1 : -1),
    1.0f, 1.0f);

glm::vec4 c = viewPlane * (2.0f / glm::dot(viewPlane, q));

obliqueProjection[0][2] = c.x - obliqueProjection[0][3];
obliqueProjection[1][2] = c.y - obliqueProjection[1][3];
obliqueProjection[2][2] = c.z - obliqueProjection[2][3];
obliqueProjection[3][2] = c.w - obliqueProjection[3][3];
```

4.5 递归渲染：Portal 中的 Portal

Portal 互相可见时会产生“镜廊/画中画”效果。本项目支持递归渲染，但出于性能考虑设定最大深度，例如 `MAX_PORTAL_RECURSION=3`，防止无限递归导致帧率崩溃。

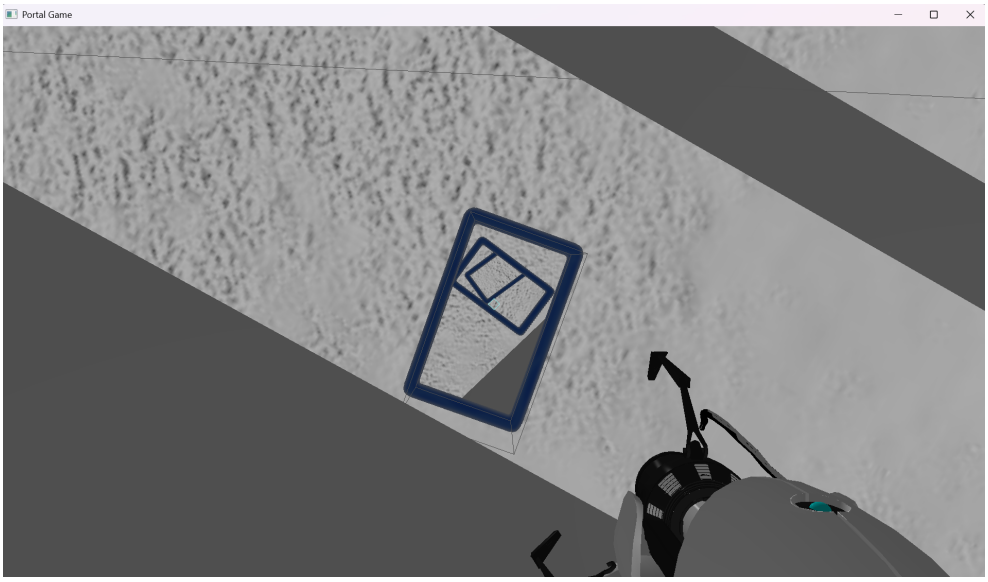


Figure 3: 展示了当两扇传送门互为可见时产生的“无限镜廊”视觉效果。渲染管线通过递归调用 `renderPortal` 函数，实现了深度为 3 的嵌套视图绘制 (MAX_PORTAL_RECURSION = 3)，有效验证了离屏渲染逻辑的完备性。

4.6 屏幕截图与图像保存

为了满足课程要求中“提供屏幕截取/保存功能”并方便记录实验结果，我们在渲染管线末端集成了截图模块。

4.6.1 像素读取与编码

截图功能的实现基于 OpenGL 标准管线与 `stb_image_write` 库：

1. 读取像素：在每帧渲染结束后，利用 `glReadPixels` 从当前帧缓冲（Back Buffer）读取屏幕宽高的 RGB 像素数据到内存 `std::vector<unsigned char>` 中。

2. 图像编码：引入 `stb_image_write.h` 单头文件库，将原始像素数据编码为 PNG 格式。由于 OpenGL 纹理坐标原点位于左下角，而图像文件存储通常从左上角开始，保存前需调用 `stbi_flip_vertically_on_write(true)` 进行垂直翻转。

4.6.2 自动化文件管理

为了防止截图文件互相覆盖，系统采用“时间戳命名法”自动生成文件名（例如 `screenshot_20240106_12000`）并检测或创建 `screenshots/` 目录，确保文件写入路径的有效性。

Listing 8: src/Application.cpp：截图功能实现核心（节选）

```
if (input.isKeyPressed(GLFW_KEY_P)) {
    auto t = std::time(nullptr);
    auto tm = *std::localtime(&t);
    std::ostringstream oss;
    oss << "screenshots/screenshot_" << std::put_time(&tm, "%Y%m%d_%H%M%S") << ".png";
    std::vector<unsigned char> pixels(width * height * 3);
    glPixelStorei(GL_PACK_ALIGNMENT, 1); // 保证像素字节对齐
```

```
    glReadPixels(0, 0, width, height, GL_RGB, GL_UNSIGNED_BYTE, pixels.data());
    stbi_flip_vertically_on_write(true);
    if (stbi_write_png(oss.str().c_str(), width, height, 3, pixels.data(), 0)) {
        std::cout << "Screenshot saved: " << oss.str() << std::endl;
    }
}
```

5 物理系统与碰撞检测（SAT + OBB）

5.1 刚体与力学更新

游戏中玩家与方块具有重力、速度与受力（抓取时的弹簧力、投掷时的推力等）。物理更新阶段对速度/位移积分，并在碰撞发生时做位置修正与速度处理。

5.2 OBB 表示与分离轴定理（SAT）

为支持旋转物体（例如被抓取的方块可能发生旋转），仅使用 AABB 难以提供稳定精确的碰撞结果。因此我们实现 OBB + SAT：

对两个 OBB，测试 15 个潜在分离轴（A 的 3 个轴、B 的 3 个轴、以及 9 个叉乘轴）。若存在任意轴使投影区间不重叠，则不碰撞；否则碰撞并取最小穿透深度作为修正方向。

Listing 9: src/PhysicsSystem.cpp：SAT 单轴测试（原代码节选）

```
static bool TestAxis(const glm::vec3 &axis,
                     const OBB &a, const OBB &b,
                     float &minPen, glm::vec3 &bestAxis) {
    float len2 = glm::dot(axis, axis);
    if (len2 < 1e-6f) return true; // ignore near-zero axis
    glm::vec3 n = axis / std::sqrt(len2);

    float aMin, aMax, bMin, bMax;
    ProjectOBB(n, a, aMin, aMax);
    ProjectOBB(n, b, bMin, bMax);

    float overlap = std::min(aMax, bMax) - std::max(aMin, bMin);
    if (overlap < 0.0f) return false;

    if (overlap < minPen) {
        minPen = overlap;
        bestAxis = n;
    }
    return true;
}
```

Listing 10: src/PhysicsSystem.cpp：SAT 15 轴检测（原代码节选）

```
bool PhysicsSystem::checkCollisionSAT(const OBB &a, const OBB &b,
                                      glm::vec3 &correction) {
    float minPen = 1e9f;
    glm::vec3 bestAxis(0.0f);
```

```
    // 1) A's local axes
    for (int i = 0; i < 3; i++)
        if (!TestAxis(a.axes[i], a, b, minPen, bestAxis)) return false;

    // 2) B's local axes
    for (int i = 0; i < 3; i++)
        if (!TestAxis(b.axes[i], a, b, minPen, bestAxis)) return false;

    // 3) 9 cross product axes
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++) {
            glm::vec3 axis = glm::cross(a.axes[i], b.axes[j]);
            if (!TestAxis(axis, a, b, minPen, bestAxis)) return false;
        }

    // correction points from A to B
    glm::vec3 d = b.center - a.center;
    if (glm::dot(d, bestAxis) < 0) bestAxis = -bestAxis;

    correction = -bestAxis * minPen;
    return true;
}
```

5.3 Raycast：用于抓取和 Portal 放置

射线检测贯穿多个玩法：

- E 拾取：从相机位置沿前向发射射线，命中可抓取物体后进入抓取状态；
- 鼠标放置 Portal：射线命中可放置墙面，计算门的位姿并生成/更新门。

6 Portal 传送：位置/速度/相机姿态的一致性

Portal “传送”不仅是把位置挪到另一端，还需要处理：

- 位置：入口局部坐标 → 180° 翻转 → 出口世界坐标；
- 速度：同样作为方向向量进行变换，保证动量方向正确；
- 相机：玩家视角 Front/Up/Right 同步变换，并处理 Roll 恢复，避免“歪脖子永久倾斜”。

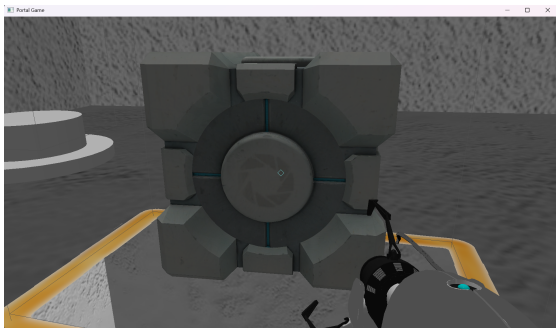
Listing 11: src/Portal.cpp：传送位置与速度变换（原代码节选）

```
glm::vec4 relPos = glm::inverse(srcRot)
    * glm::vec4(obj->position - this->position, 1.0f);

glm::vec4 newWorldPos4 = glm::vec4(linkedPortal->position, 1.0f)
    + dstRot * (rot180 * relPos);

// push forward to avoid immediate re-trigger
```

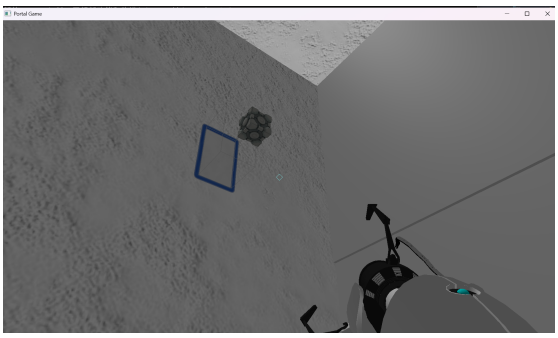
```
glm::vec3 dstForward = glm::normalize(glm::vec3(
    dstRot * glm::vec4(0,0,1,0)));
obj->position = glm::vec3(newWorldPos4) + dstForward * 0.15f;

if (obj->rigidBody) {
    glm::vec4 relVel = glm::inverse(srcRot)
        * glm::vec4(obj->rigidBody->velocity, 0.0f);
    glm::vec4 newVel4 = dstRot * (rot180 * relVel);
    obj->rigidBody->velocity = glm::vec3(newVel4);
}
```

Listing 12: src/Portal.cpp：玩家相机朝向变换与 Roll 恢复（原代码节选）

```
glm::vec4 relFront = glm::inverse(srcRot)
    * glm::vec4(player->camera.Front, 0.0f);
glm::vec4 newFront4 = dstRot * (rot180 * relFront);
glm::vec3 newFront = glm::normalize(glm::vec3(newFront4));
player->camera.Front = newFront;

// recompute yaw/pitch
float pitch = glm::degrees(glm::asin(glm::clamp(newFront.y, -1.0f, 1.0f)));
float yaw = glm::degrees(std::atan2(newFront.z, newFront.x));
player->camera.Yaw = yaw;
player->camera.Pitch = pitch;

// roll recovery timer
player->initialRoll = roll;
player->rollRecoveryDuration = abs(roll) / 180.0f * 0.6f + 0.2f;
player->rollRecoveryTimer = player->rollRecoveryDuration;
```



(a) 入口：物体具有向下的垂直速度



(b) 出口：速度向量被正确变换为水平向前

Figure 4: 通过抛射实验证明，当物体穿过传送门时，其速度向量经过矩阵变换后，大小保持不变，方向相对于出口平面正确重定向。

7 交互系统：抓取、按钮、Flip 机关与终点

7.1 重力枪抓取（弹簧阻尼跟随）

拾取不是简单把方块“绑死在相机前”，而是用弹簧阻尼力让物体平滑跟随且保留惯性，提升手感与真实感。

Listing 13: src/Player.cpp：E 拾取/放下（原代码节选）

```
if (input.isKeyPressed(GLFW_KEY_E)) {
    if (isGrabbing) {
        isGrabbing = false;
    } else {
        auto result = scene->physicsSystem->raycast(camera.Position,
                      camera.Front, 5.0f);
        if (result.hit && result.object && result.object->isTeleportable) {
            isGrabbing = true;
            grabbedObject = result.object;
        }
    }
}
```

Listing 14: src/Player.cpp：弹簧阻尼施力（原代码节选）

```
if (isGrabbing && grabbedObject) {
    glm::vec3 targetPos = position + camera.Front * 2.0f
        + glm::vec3(0.0f, height * 0.5f, 0.0f);

    if (glm::length(targetPos - grabbedObject->position) > 3.0f) {
        isGrabbing = false;
    } else {
        glm::vec3 springForce = (targetPos - grabbedObject->position) * 30.0f;
        glm::vec3 dampingForce = -grabbedObject->rigidBody->velocity * 5.0f;
        grabbedObject->rigidBody->addForce(springForce + dampingForce);
    }
}
```

7.2 按钮与 **Flip**：**Trigger** 驱动的机关联动

按钮（Button）为自身生成 Trigger 区域用于检测“有人/物压住”。Scene 在每帧更新中读取 Button 的按下/抬起沿边（getIsPressed / getIsReleased），进而驱动 Flip 机关翻转或复位；同时考虑了 Flip 旋转期间 Portal 贴附失效的处理，避免穿模与视角错误。

Listing 15: src/Button.cpp：为按钮生成 Trigger（节选）

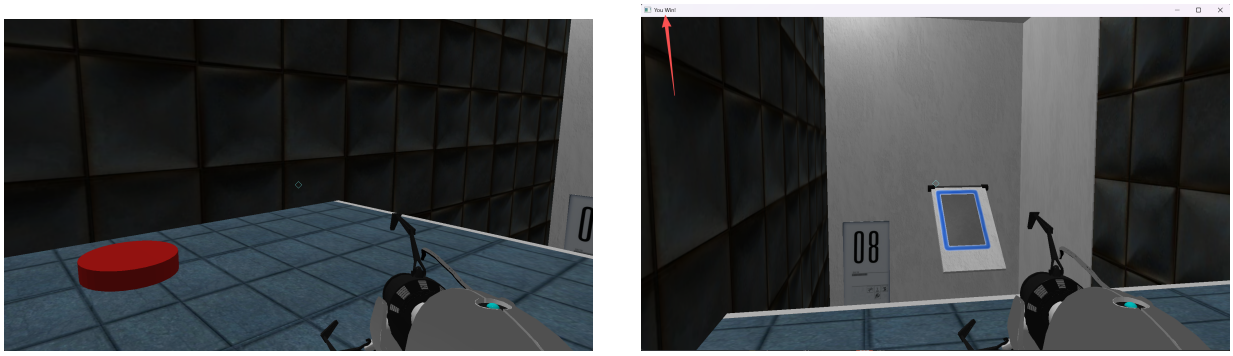
```
std::unique_ptr<Trigger> Button::createTrigger() {
    glm::vec3 min = model->minBound * scale;
    glm::vec3 max = model->maxBound * scale;

    glm::vec3 center = initialPosition + (min + max) * 0.5f;
    glm::vec3 size = max - min;

    glm::vec3 triggerMin = center - size * 0.5f + glm::vec3(-0.2f, 0.0f, -0.2f);
    glm::vec3 triggerMax = center + size * 0.5f + glm::vec3(0.2f, 0.5f, 0.2f);

    return std::make_unique<Trigger>(triggerMin, triggerMax);
}
```

Listing 16: src/Scene.h：按钮驱动 Flip 与 Portal 失效处理（节选）

```
auto button_flip = objects.find("button_flip");
if (button_flip != objects.end()) {
    Button *btn = dynamic_cast<Button *>(button_flip->second.get());
    if (btn && btn->getIsPressed()) {
        auto flipWall = objects.find("flip_wall");
        if (flipWall != objects.end()) {
            Flip *flip = dynamic_cast<Flip *>(flipWall->second.get());
            if (flip) flip->flip();
        }
    } else if (btn && btn->getIsReleased()) {
        auto flipWall = objects.find("flip_wall");
        if (flipWall != objects.end()) {
            Flip *flip = dynamic_cast<Flip *>(flipWall->second.get());
            if (flip) flip->reset();
        }
    }
}
```

7.3 终点判定与通关重置

终点同样通过按钮（Button + Trigger）实现：Application 在主循环中检测终点按钮（button_goal）的按下沿，一旦被触发则调用 `glfwSetWindowTitle(window, "You Win!")` 给出胜利反馈。



(a) 终点平台　　(b) 窗口标题栏显示通关状态（You Win!）

Figure 5: 当玩家进入终点区域触发 Trigger 后，后端逻辑不仅在控制台输出日志，还通过 `glfwSetWindowTitle` 实时更新窗口标题，向玩家提供直观的胜利反馈。

8 关键工程问题与调试经验

8.1 OBJ 资源丢失与路径问题

开发中常见问题：运行时报 `Failed to open OBJ file: ...`，根因通常为工作目录与资源相对路径不一致。解决方式：

- 统一可执行程序运行目录（例如把工作目录设为项目根）；
- 或改为从配置文件读取资源根路径；
- 或使用 CMake 在构建时复制 resources 到输出目录。

8.2 Portal 递归与性能权衡

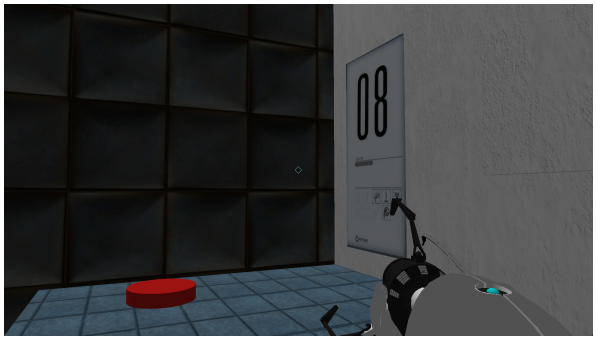
递归层数越深，FBO 渲染次数越多，GPU 开销成倍增加。我们通过限制最大递归深度并在门不可见/距离过远时跳过 Portal 渲染来平衡画面与帧率。

8.3 传送后立即反复触发（“抖门/死循环”）

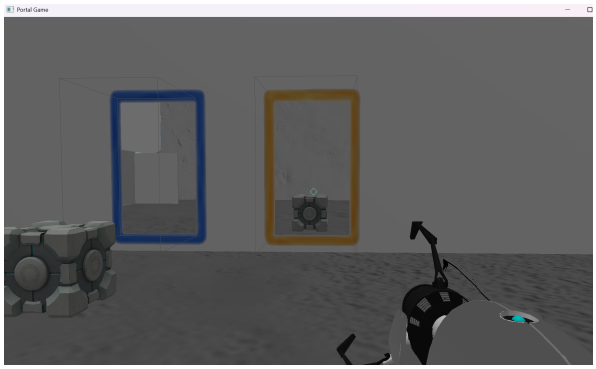
典型现象：物体穿门后立刻又被入口触发器检测到，来回传送造成死循环。我们采用“前向推开”策略：传送到出口后沿门法线方向推离一小段距离（见前文 `teleportForwardPush`），并结合 nearTrigger/碰撞 mask 做短暂保护窗口，避免反复触发。

9 实验结果与展示

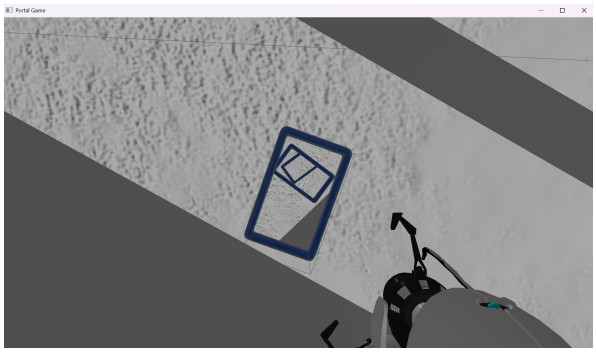
为了直观展示本项目的核心功能与渲染效果，我们在实际运行中截取了以下关键场景，涵盖了从基础场景渲染、Portal 核心机制到物理交互与游戏通关的完整流程。



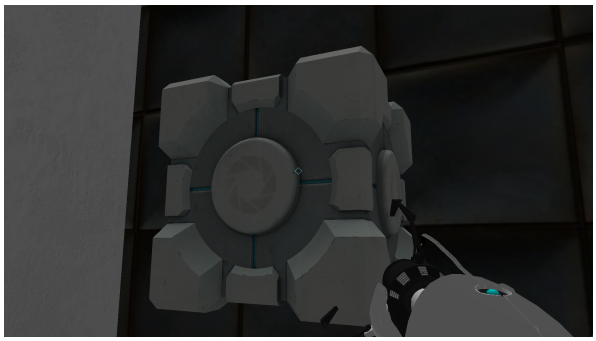
(a) 实验室场景全景渲染（Blender 建模导入）



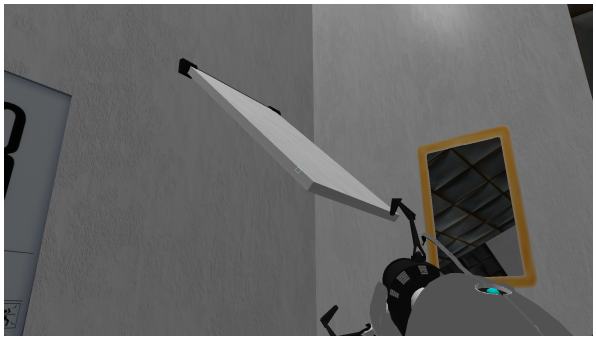
(b) Portal 视窗效果：A 门表面实时显示 B 门视角



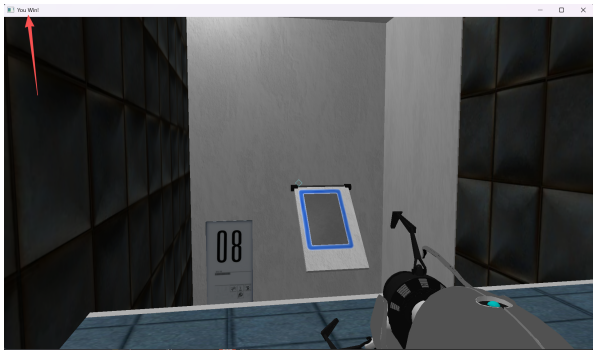
(c) 递归渲染：当双门互视时产生的“画中画”效果



(d) 物理交互：使用重力枪吸附并携带方块移动



(e) 逻辑解谜：方块压下按钮，触发 Flip 翻转打开通道



(f) 通关判定：到达终点区域后触发胜利逻辑

Figure 6: Portal 游戏系统核心功能综合演示

10 总结与展望

通过本次大作业，我们完整实现了一个可玩的 Portal 类三维解谜关卡，覆盖从资源管线、实时渲染到物理交互与高级 Portal 技术的完整链路，达成课程要求并具备扩展性。未来改进方向：

- 更完善的光照与阴影（Shadow Mapping / SSAO），提升真实感；
- 使用 BVH/空间划分优化碰撞与 raycast 性能；
- 更丰富的机关组合与关卡编辑能力；
- Portal 放置限制与更完善的剪裁/模板缓冲方案对比实现。

A 附录：源代码清单

说明：本附录通过 \lstinputlisting 直接嵌入工程源代码，此处只展示关键代码，具体代码于 src 文件夹中。

A.1 Application / 主循环

Listing 17: src/Application.cpp

```cpp
#include "stb_image_write.h"
#include "Button.h"
#include "Application.h"
#include "Flip.h"
#include "PortalGun.h"
#include "InputManager.h"
#include "Trigger.h"

#include <iostream>
#include <cmath>
#include <ctime>
#include <sstream>
#include <iomanip>
#include <filesystem>

#include <glm/gtc/matrix_transform.hpp>

float lastX = 1920.0f / 2.0f;
float lastY = 1080.0f / 2.0f;
bool firstMouse = true;
float deltaTime = 0.0f;
float lastFrame = 0.0f;

Application::Application(int width, int height, const std::string &title)
    : width(width), height(height), title(title), window(nullptr), fallbackCamera(glm::vec3(0.0f,
        0.0f, 3.0f)) {
}

Application::~Application() {
    shutdown();
}

bool Application::initialize() {
    // 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
```

```
42
43      // glfw window creation
44      window = glfwCreateWindow(width, height, title.c_str(), NULL, NULL);
45      if (window == NULL) {
46          std::cout << "Failed to create GLFW window" << std::endl;
47          glfwTerminate();
48          return false;
49      }
50      glfwMakeContextCurrent(window);
51      glfwGetFramebufferSize(window, &width, &height);
52      glfwSetWindowUserPointer(window, this);
53      glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
54      glfwSetCursorPosCallback(window, mouse_callback);
55      glfwSetScrollCallback(window, scroll_callback);
56      glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);
57
58      // glad: load all OpenGL function pointers
59      if (!gladLoadGL((GLADloadfunc)glfwGetProcAddress)) {
60          std::cout << "Failed to initialize GLAD" << std::endl;
61          return false;
62      }
63
64      stbi_set_flip_vertically_on_load(true);
65      Texture::InitDefaultTextures();
66
67      glEnable(GL_DEPTH_TEST);
68
69      // --- Initialize Core Systems ---
70      scene = std::make_unique<Scene>();
71      renderer = std::make_unique<Renderer>(width, height);
72      renderer->initialize();
73
74      // --- Initialize input manager ---
75      input.initialize(window);
76
77      // --- Initialize Player ---
78      scene->player = std::make_unique<Player>(glm::vec3(0.0f, 0.0f, 0.0f));
79      scene->lightPos = glm::vec3(0.0f, 15.0f, 0.0f);
80
81      // --- Load Resources ---
82      scene->addModelResource("portal_gun", std::make_unique<Model>("resources/obj/portal_gun/
            portal_gun.obj"));
83      scene->addModelResource("cube", std::make_unique<Model>("resources/obj/wall/cube.obj"));
84      scene->addModelResource("portal_cube", std::make_unique<Model>("resources/obj/portal_cube/
            portal_cube.obj"));
85
86      // --- Portal A ---
87      scene->portalA = std::make_unique<Portal>(width, height);
88      scene->portalA->position = glm::vec3(100.0f, 1.0f, 101.0f);
89      scene->portalA->scale = glm::vec3(0.8f, 1.4f, 0.005f);
90      scene->portalA->name = "PortalA";
```

```
    scene->portalA->type = PORTAL_A;
    scene->portalA->isActive = false;
    scene->portalA->init(scene.get());

    // --- Portal B ---
    scene->portalB = std::make_unique<Portal>(width, height);
    scene->portalB->position = glm::vec3(100.0f, 0.0f, 100.0f);
    scene->portalB->rotation = glm::vec3(0.0f, 180.0f, 0.0f);
    scene->portalB->scale = glm::vec3(0.8f, 1.4f, 0.005f);
    scene->portalB->name = "PortalB";
    scene->portalB->type = PORTAL_B;
    scene->portalB->isActive = false;
    scene->portalB->init(scene.get());

    // --- Link Portals ---
    scene->portalA->setLinkedPortal(scene->portalB.get());
    scene->portalB->setLinkedPortal(scene->portalA.get());

    // --- Portal Gun ---
    scene->portalGun = std::make_unique<PortalGun>(scene->modelResources["portal_gun"].get());
    scene->portalGun->position = glm::vec3(0.5f, -0.5f, -1.0f);
    scene->portalGun->scale = glm::vec3(0.05f);

    createScene();

    // --- Initialize Skybox ---
    std::vector<std::string> faces = {
        "resources/skybox/right.jpg",
        "resources/skybox/left.jpg",
        "resources/skybox/top.jpg",
        "resources/skybox/bottom.jpg",
        "resources/skybox/front.jpg",
        "resources/skybox/back.jpg"
    };
    scene->skybox = std::make_unique<Skybox>(faces);

    // --- Initialize Physics World State ---
    if (scene->physicsSystem) {
        scene->physicsSystem->update(0.0f);
    }

    return true;
}

void Application::createScene(int level) {
    (void *)level;
    scene->addModelResource("banner", std::make_unique<Model>("resources/obj/level/banner.obj"));
    scene->addModelResource("button_flip", std::make_unique<Model>("resources/obj/level/
        button_flip.obj"));
    scene->addModelResource("button_goal", std::make_unique<Model>("resources/obj/level/
        button_goal.obj"));
```

```
140     scene->addModelResource("movable", std::make_unique<Model>("resources/obj/level/movable.obj"))
            ;
141     scene->addModelResource("wall1", std::make_unique<Model>("resources/obj/level/wall1.obj"));
142     scene->addModelResource("wall2", std::make_unique<Model>("resources/obj/level/wall2.obj"));
143     scene->addModelResource("wall3", std::make_unique<Model>("resources/obj/level/wall3.obj"));
144     scene->addModelResource("wall4_p", std::make_unique<Model>("resources/obj/level/wall4_p.obj"))
            ;
145     scene->addModelResource("wall5", std::make_unique<Model>("resources/obj/level/wall5.obj"));
146     scene->addModelResource("wall6_p", std::make_unique<Model>("resources/obj/level/wall6_p.obj"))
            ;
147     scene->addModelResource("wall7", std::make_unique<Model>("resources/obj/level/wall7.obj"));
148     scene->addModelResource("wall8", std::make_unique<Model>("resources/obj/level/wall8.obj"));
149     scene->addModelResource("wall9", std::make_unique<Model>("resources/obj/level/wall9.obj"));
150     scene->addModelResource("wall10", std::make_unique<Model>("resources/obj/level/wall10.obj"));
151     scene->addModelResource("wall11_p", std::make_unique<Model>("resources/obj/level/wall11_p.obj"
            ));
152     scene->addModelResource("wall12_p", std::make_unique<Model>("resources/obj/level/wall12_p.obj"
            ));
153
154     auto addStaticObj = [&](const std::string &name, const std::string &modelName, bool canPortal)
            {
155         auto obj = std::make_unique<GameObject>(scene->modelResources[modelName].get());
156         obj->isTeleportable = false;
157         obj->canOpenPortal = canPortal;
158         // Register physics on the object before moving it into the scene map
159         scene->addPhysics(obj.get(), true);
160         scene->addObject(name, std::move(obj));
161         };
162
163     addStaticObj("banner", "banner", false);
164     addStaticObj("wall1", "wall1", false);
165     addStaticObj("wall2", "wall2", false);
166     addStaticObj("wall3", "wall3", false);
167     addStaticObj("wall4", "wall4_p", true);
168     addStaticObj("wall5", "wall5", false);
169     addStaticObj("wall6", "wall6_p", true);
170     addStaticObj("wall7", "wall7", false);
171     addStaticObj("wall8", "wall8", false);
172     addStaticObj("wall9", "wall9", false);
173     addStaticObj("wall10", "wall10", false);
174     addStaticObj("wall11", "wall11_p", true);
175     addStaticObj("wall12", "wall12_p", true);
176
177     auto buttonFlip = std::make_unique<Button>(scene->modelResources["button_flip"].get(), glm::
            vec3(0.0f), glm::vec3(0.0f), glm::vec3(1.0f));
178     buttonFlip->isTeleportable = false;
179     buttonFlip->canOpenPortal = false;
180     scene->addTrigger("button_flip_trigger", buttonFlip->createTrigger());
181     scene->addPhysics(buttonFlip.get(), true);
182     scene->addObject("button_flip", std::move(buttonFlip));
183
```

```
184     auto flipWall = std::make_unique<Flip>(scene->modelResources["movable"].get(), glm::vec3(0.0f)
            , glm::vec3(0.0f), glm::vec3(1.0f));
185     flipWall->isTeleportable = false;
186     flipWall->canOpenPortal = true;
187     flipWall->setPivot(glm::vec3(0.0f, 0.0f, 0.0f));
188     flipWall->setPosition(glm::vec3(-1.5f, 5.5f, -5.0f));
189     scene->addPhysics(flipWall.get(), true);
190     scene->addObject("flip_wall", std::move(flipWall));
191 
192     auto cube = std::make_unique<GameObject>(scene->modelResources["portal_cube"].get());
193     cube->isTeleportable = true;
194     cube->canOpenPortal = false;
195     cube->scale = glm::vec3(0.05f);
196     cube->position = glm::vec3(7.0f, -4.0f, -2.0f);
197     scene->addPhysics(cube.get(), false);
198     scene->addObject("portal_cube", std::move(cube));
199 
200     auto button_goal = std::make_unique<Button>(scene->modelResources["button_goal"].get(), glm::
            vec3(0.0f), glm::vec3(0.0f), glm::vec3(1.0f));
201     button_goal->isTeleportable = false;
202     button_goal->canOpenPortal = false;
203     scene->addTrigger("button_goal_trigger", button_goal->createTrigger());
204     scene->addPhysics(button_goal.get(), true);
205     scene->addObject("button_goal", std::move(button_goal));
206 }
207 
208 void Application::run() {
209     // Reset time to avoid large dt on first frame
210     lastFrame = static_cast<float>(glfwGetTime());
211 
212     while (!glfwWindowShouldClose(window)) {
213         // per-frame time logic
214         float currentFrame = static_cast<float>(glfwGetTime());
215         deltaTime = currentFrame - lastFrame;
216         lastFrame = currentFrame;
217 
218         // Clamp deltaTime to avoid physics explosions (e.g. during debugging or lag spikes)
219         if (deltaTime > 0.1f) deltaTime = 0.1f;
220 
221         // input
222         // poll and update input state first
223         input.update();
224         processInput(deltaTime);
225 
226         // --- Logic Update ---
227         Camera &activeCamera = getActiveCamera();
228         scene->update(deltaTime, activeCamera);
229         auto button_goal = scene->objects.find("button_goal");
230         if (button_goal != scene->objects.end()) {
231             Button *btn = dynamic_cast<Button *>(button_goal->second.get());
232             if (btn && btn->getIsPressed()) {
```

```
233                 //change window title to "You Win!"
234                 glfwSetWindowTitle(window, "You Win!");
235             }
236         }
237 
238         // render
239         renderer->render(*scene, activeCamera);
240 
241         // glfw: swap buffers and poll IO events
242         glfwSwapBuffers(window);
243         glfwPollEvents();
244     }
245 }
246 
247 void Application::shutdown() {
248     glfwTerminate();
249 }
250 
251 Camera &Application::getActiveCamera() {
252     if (scene->player) {
253         return scene->player->camera;
254     }
255     return fallbackCamera;
256 }
257 
258 void Application::processInput(float deltaTime) {
259     if (input.isKeyDown(GLFW_KEY_ESCAPE))
260         glfwSetWindowShouldClose(window, true);
261 
262     if (scene->player) {
263         scene->player->processInput(input, scene.get(), deltaTime);
264     }
265 
266     if (input.isKeyPressed(GLFW_KEY_T)) {
267         if (scene->player) {
268             scene->player->position = glm::vec3(0.0f, 0.0f, 0.0f);
269             scene->player->rigidBody->velocity = glm::vec3(0.0f);
270         }
271     }
272 
273     if (input.isKeyPressed(GLFW_KEY_P)) {
274         auto t = std::time(nullptr);
275         auto tm = *std::localtime(&t);
276 
277         std::ostringstream oss;
278         oss << "screenshots/screenshot_" << std::put_time(&tm, "%Y%m%d_%H%M%S") << ".png";
279         std::string filename = oss.str();
280 
281         int width = this->width;
282         int height = this->height;
283         std::vector<unsigned char> pixels(width * height * 3);
```

```
284
285         glPixelStorei(GL_PACK_ALIGNMENT, 1);
286         glReadPixels(0, 0, width, height, GL_RGB, GL_UNSIGNED_BYTE, pixels.data());
287         namespace fs = std::filesystem;
288         if (!fs::exists("screenshots")) {
289             fs::create_directory("screenshots");
290         }
291
292         stbi_flip_vertically_on_write(true);
293
294         if (stbi_write_png(filename.c_str(), width, height, 3, pixels.data(), 0)) {
295             std::cout << "Screenshot saved: " << filename << std::endl;
296         }
297         else {
298             std::cerr << "Failed to save screenshot. Make sure 'screenshots' folder exists." << std
                ::endl;
299         }
300 }
301 }
302
303 void Application::framebuffer_size_callback(GLFWwindow *window, int width, int height) {
304     glViewport(0, 0, width, height);
305     Application *app = static_cast<Application *>(glfwGetWindowUserPointer(window));
306     if (app) {
307         app->width = width;
308         app->height = height;
309         if (app->renderer) {
310             app->renderer->resize(width, height);
311         }
312     }
313 }
314
315 void Application::mouse_callback(GLFWwindow *window, double xposIn, double yposIn) {
316     Application *app = static_cast<Application *>(glfwGetWindowUserPointer(window));
317     if (!app) return;
318
319     float xpos = static_cast<float>(xposIn);
320     float ypos = static_cast<float>(yposIn);
321
322     if (firstMouse) {
323         lastX = xpos;
324         lastY = ypos;
325         firstMouse = false;
326     }
327
328     float xoffset = xpos - lastX;
329     float yoffset = lastY - ypos;
330
331     lastX = xpos;
332     lastY = ypos;
333
```

```
334     app->getActiveCamera().ProcessMouseMovement(xoffset, yoffset);
335 }
336 
337 void Application::scroll_callback(GLFWwindow *window, double xoffset, double yoffset) {
338     Application *app = static_cast<Application *>(glfwGetWindowUserPointer(window));
339     if (!app) return;
340     app->getActiveCamera().ProcessMouseScroll(static_cast<float>(yoffset));
341 }
```

Listing 18: src/Application.h

```
1  #pragma once
2  
3  #include "Shader.h"
4  #include "Camera.h"
5  #include "Scene.h"
6  #include "Renderer.h"
7  #include <vector>
8  #include "InputManager.h"
9  #include "Player.h"
10 
11 #include <string>
12 #include <memory>
13 
14 #include <glad/gl.h>
15 #include <GLFW/glfw3.h>
16 
17 class Application {
18 public:
19     Application(int width, int height, const std::string &title);
20     ~Application();
21 
22     bool initialize();
23     void createScene(int level = 1);
24     void run();
25     void shutdown();
26 
27 private:
28     void processInput(float deltaTime);
29 
30     static void framebuffer_size_callback(GLFWwindow *window, int width, int height);
31     static void mouse_callback(GLFWwindow *window, double xpos, double ypos);
32     static void scroll_callback(GLFWwindow *window, double xoffset, double yoffset);
33     static void mouse_button_callback(GLFWwindow *window, int button, int action, int mods);
34 
35     int width, height;
36     std::string title;
37     GLFWwindow *window;
38 
39     std::unique_ptr<Renderer> renderer;
40     std::unique_ptr<Scene> scene;
41 
```

```
42      Camera fallbackCamera;
43      Camera &getActiveCamera();
44
45      // Input
46      InputManager input;
47  };
```

A.2 渲染与帧缓冲

Listing 19: src/Renderer.cpp

```
1   #include "Renderer.h"
2
3   #include <glad/gl.h>
4   #include <glm/gtc/matrix_transform.hpp>
5
6   Renderer::Renderer(int width, int height) : width(width), height(height) {}
7
8   void Renderer::initialize() {
9       glEnable(GL_DEPTH_TEST);
10
11      // build and compile shaders
12      auto shader = std::make_unique<Shader>("shaders/default.vert", "shaders/default.frag");
13      shader->setBool("useAlphaTest", false);
14      auto portalShader = std::make_unique<Shader>("shaders/screen.vert", "shaders/screen.frag");
15      shaderCache["default"] = std::move(shader);
16      shaderCache["portal"] = std::move(portalShader);
17      // HUD manager
18      hud = std::make_unique<HUD>();
19      hud->initialize();
20  }
21
22  void Renderer::resize(int width, int height) {
23      this->width = width;
24      this->height = height;
25      glViewport(0, 0, width, height);
26  }
27
28  void Renderer::drawScene(Scene &scene, Shader &shader, const glm::mat4 &view, const glm::mat4 &
        projection, const glm::vec3 &viewPos) {
29      shader.use();
30      shader.setMat4("projection", projection);
31      shader.setMat4("view", view);
32      shader.setVec3("viewPos", viewPos);
33
34      // Directional light
35      shader.setVec3("dirLight.direction", -0.2f, -1.0f, -0.3f);
36      shader.setVec3("dirLight.ambient", 0.3f, 0.3f, 0.3f);
37      shader.setVec3("dirLight.diffuse", 0.4f, 0.4f, 0.4f);
38      shader.setVec3("dirLight.specular", 0.5f, 0.5f, 0.5f);
39
```

```
    // Point light
    shader.setVec3("pointLights[0].position", scene.lightPos);
    shader.setVec3("pointLights[0].ambient", 0.2f, 0.2f, 0.2f);
    shader.setVec3("pointLights[0].diffuse", 0.8f, 0.8f, 0.8f);
    shader.setVec3("pointLights[0].specular", 1.0f, 1.0f, 1.0f);
    shader.setFloat("pointLights[0].constant", 1.0f);
    shader.setFloat("pointLights[0].linear", 0.09f);
    shader.setFloat("pointLights[0].quadratic", 0.032f);

    shader.setFloat("material.shininess", 32.0f);

    for (auto &pair : scene.objects) {
        pair.second->draw(shader);
    }

    if (scene.skybox) {
        scene.skybox->draw(view, projection);
    }
}

void Renderer::renderPortal(Scene &scene, Portal *portal, glm::mat4 view, const glm::mat4 &
    projection, int recursionDepth) {
    if (recursionDepth <= 0) return;//last frame TODO:render a foo texture
    glm::mat4 transformedCam = portal->getTransformedView(view);// get transformed camera view
    renderPortal(scene,portal, transformedCam, projection, recursionDepth - 1);
    portal->beginRender();//render deeper level scene(for current portal)
    glm::vec4 worldPlane = portal->getPlaneEquation();
    glm::vec4 viewPlane = worldPlane * glm::inverse(transformedCam);
    glm::mat4 obliqueProjection = projection;
    glm::vec4 q = glm::inverse(obliqueProjection) * glm::vec4(
        (viewPlane.x > 0.0f ? 1.0f : -1.0f),
        (viewPlane.y > 0.0f ? 1.0f : -1.0f),
        1.0f,
        1.0f
    );
    glm::vec4 c = viewPlane * (2.0f / glm::dot(viewPlane, q));
    obliqueProjection[0][2] = c.x - obliqueProjection[0][3];
    obliqueProjection[1][2] = c.y - obliqueProjection[1][3];
    obliqueProjection[2][2] = c.z - obliqueProjection[2][3];
    obliqueProjection[3][2] = c.w - obliqueProjection[3][3];

    glm::vec3 virtualCamPos = glm::vec3(glm::inverse(transformedCam)[3]);
    drawScene(scene, *shaderCache["default"], transformedCam, obliqueProjection, virtualCamPos);//
        render current level scene
    if (recursionDepth > 1) {
        auto portalShader = shaderCache["portal"].get();
        portalShader->use();
        portalShader->setMat4("projection", obliqueProjection);
        portalShader->setMat4("view", transformedCam);
        auto shader = shaderCache["default"].get();
        shader->use();
```

```
89          shader->setMat4("projection", obliqueProjection);
90          shader->setMat4("view", transformedCam);
91          portal->drawPrev(*portalShader, *shader);// render the previous frame texture on the
                portal surface
92          portalShader->unuse();
93      }
94      portal->endRender(width, height);
95  }
96
97  void Renderer::render(Scene &scene, Camera &camera) {
98      glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)width / (float)
            height, 0.1f, 100.0f);
99      glm::mat4 view = camera.GetViewMatrix();
100
101     // 1. Render Portal Views
102     if (scene.portalA) renderPortal(scene, scene.portalA.get(), view, projection,
            MAX_PORTAL_RECURSION);
103     if (scene.portalB) renderPortal(scene, scene.portalB.get(), view, projection,
            MAX_PORTAL_RECURSION);
104
105     // 2. Render Main Pass
106     glClearColor(0.05f, 0.05f, 0.05f, 1.0f);
107     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
108
109     drawScene(scene, *shaderCache["default"], view, projection, camera.Position);
110
111     // 3. Draw Portals
112     auto portalShader = shaderCache["portal"].get();
113     portalShader->use();
114     portalShader->setMat4("projection", projection);
115     portalShader->setMat4("view", view);
116     auto shader = shaderCache["default"].get();
117     shader->use();
118     shader->setMat4("projection", projection);
119     shader->setMat4("view", view);
120     if (scene.portalA) scene.portalA->draw(*portalShader, *shader);
121     if (scene.portalB) scene.portalB->draw(*portalShader, *shader);
122
123
124     // for (auto &pair : scene.triggers) {
125     // pair.second->drawOBBDebug(*shader);
126     // }
127
128     // 4. Draw Portal Gun
129     if (scene.portalGun) scene.portalGun->draw(*shader);
130
131     // Draw HUD
132     if (hud) hud->render();
133 }
```

Listing 20: src/Renderer.h

```cpp
#pragma once

#include "Shader.h"
#include "Scene.h"
#include "Camera.h"
#include "HUD.h"

#include <memory>
#include <unordered_map>

#include <glad/gl.h>
#include <glm/glm.hpp>

constexpr int MAX_PORTAL_RECURSION = 3;

class Renderer {
public:
    Renderer(int width, int height);
    ~Renderer() = default;

    void initialize();
    void render(Scene &scene, Camera &camera);
    void resize(int width, int height);

private:
    void drawScene(Scene &scene, Shader &shader, const glm::mat4 &view, const glm::mat4 &
        projection, const glm::vec3 &viewPos);
    void renderPortal(Scene &scene, Portal *portal, glm::mat4 view, const glm::mat4 &projection,
        int recursionDepth);
    int width, height;
    std::unordered_map<std::string, std::unique_ptr<Shader>> shaderCache;
    std::unique_ptr<HUD> hud;
};
```

A.3 Portal 系统

Listing 21: src/Portal.cpp

```cpp
#include "Portal.h"
#include "Scene.h"
#include "Player.h"

Portal::Portal(int width, int height, glm::vec3 pos, glm::vec3 rot, glm::vec3 scale)
    : GameObject(nullptr, pos, rot, scale), linkedPortal(nullptr) {
    frameBuffer[0] = std::make_unique<FrameBuffer>(width, height);
    frameBuffer[1] = std::make_unique<FrameBuffer>(width, height);

    // Setup quad VAO/VBO
    float vertices[] = {
        // positions // normals // texture coords
```

```
        -1.0f, -1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, // bottom left
         1.0f, -1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 0.0f, // bottom right
         1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // top right

        -1.0f, -1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, // bottom left
         1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 1.0f, 1.0f, // top right
        -1.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 1.0f // top left
    };
    glGenVertexArrays(1, &contentVAO);
    glGenBuffers(1, &contentVBO);

    glBindVertexArray(contentVAO);

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

    // 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);
    // texture coord attribute
    glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float), (void *)(6 * sizeof(float))
        );
    glEnableVertexAttribArray(2);

    glBindVertexArray(0);

    // Load frame textures
    frameTextureA = std::make_unique<Texture>("resources/texture/portal_blue.png", ".", "
        texture_diffuse");
    frameTextureB = std::make_unique<Texture>("resources/texture/portal_yellow.png", ".", "
        texture_diffuse");
}

Portal::~Portal() {
    glDeleteVertexArrays(1, &contentVAO);
    glDeleteBuffers(1, &contentVBO);
}

void Portal::init(Scene *scene) {
    createFrames(scene->modelResources["cube"].get(), 0.15f, 0.2f);
    registerFramesPhysics(scene, COLLISION_MASK_PORTALFRAME);
    nearTrigger = new Trigger(glm::vec3(100.0f), glm::vec3(101.0f));
    nearTrigger->onEnter = [](GameObject *obj) {
        obj->setCollisionMask(COLLISION_MASK_NEARPORTAL);
        };
    nearTrigger->onInside = [](GameObject *obj) {
        obj->setCollisionMask(COLLISION_MASK_NEARPORTAL);
```

```
60          };
61      nearTrigger->onExit = [](GameObject *obj) {
62          obj->setCollisionMask(COLLISION_MASK_DEFAULT);
63          };
64      nearTrigger->isActive = false;
65      scene->addTrigger(name + "NearTrigger", std::unique_ptr<Trigger>(nearTrigger));
66      teleportTrigger = new Trigger(glm::vec3(100.0f), glm::vec3(101.0f));
67      teleportTrigger->onEnter = [this](GameObject *obj) {
68          if (!linkedPortal || !obj || !obj->isTeleportable) return;
69
70          // Build rotation matrices for source (this) and destination (linkedPortal)
71          // Use Ry * Rx * Rz order to match checkRaycast logic and avoid gimbal lock
72          glm::mat4 srcRot = glm::mat4(1.0f);
73          srcRot = glm::rotate(srcRot, glm::radians(rotation.y), glm::vec3(0.0f, 1.0f, 0.0f));
74          srcRot = glm::rotate(srcRot, glm::radians(rotation.x), glm::vec3(1.0f, 0.0f, 0.0f));
75          srcRot = glm::rotate(srcRot, glm::radians(rotation.z), glm::vec3(0.0f, 0.0f, 1.0f));
76
77          glm::mat4 dstRot = glm::mat4(1.0f);
78          dstRot = glm::rotate(dstRot, glm::radians(linkedPortal->rotation.y), glm::vec3(0.0f, 1.0f,
                0.0f));
79          dstRot = glm::rotate(dstRot, glm::radians(linkedPortal->rotation.x), glm::vec3(1.0f, 0.0f,
                0.0f));
80          dstRot = glm::rotate(dstRot, glm::radians(linkedPortal->rotation.z), glm::vec3(0.0f, 0.0f,
                1.0f));
81
82          // Add a 180 degree rotation around the portal's local Y to map facing correctly.
83          glm::mat4 rot180 = glm::rotate(glm::mat4(1.0f), glm::radians(180.0f), glm::vec3(0.0f, 1.0f
                , 0.0f));
84
85          // Transform position: compute local position in source's local axes, apply 180deg, then
                transform to dest world
86          glm::vec4 relPos = glm::inverse(srcRot) * glm::vec4(obj->position - this->position, 1.0f);
87          glm::vec4 newWorldPos4 = glm::vec4(linkedPortal->position, 1.0f) + dstRot * (rot180 *
                relPos);
88          // push slightly forward along the destination portal normal to avoid immediate re-trigger
89          glm::vec3 dstForward = glm::normalize(glm::vec3(dstRot * glm::vec4(0.0f, 0.0f, 1.0f, 0.0f)
                ));
90          const float teleportForwardPush = 0.15f;
91          obj->position = glm::vec3(newWorldPos4) + dstForward * teleportForwardPush;
92
93          // Transform velocity if present (treat as direction vector, w=0) with 180deg flip
94          if (obj->rigidBody) {
95              glm::vec4 relVel = glm::inverse(srcRot) * glm::vec4(obj->rigidBody->velocity, 0.0f);
96              glm::vec4 newVel4 = dstRot * (rot180 * relVel);
97              obj->rigidBody->velocity = glm::vec3(newVel4);
98          }
99
100         // If this is the Player, also rotate the camera's orientation (Front/Yaw/Pitch)
101         Player *player = dynamic_cast<Player *>(obj);
102         if (player) {
103             // Transform camera Front vector through src->rot180->dst
```

```
104             glm::vec4 relFront = glm::inverse(srcRot) * glm::vec4(player->camera.Front, 0.0f);
105             glm::vec4 newFront4 = dstRot * (rot180 * relFront);
106             glm::vec3 newFront = glm::normalize(glm::vec3(newFront4));
107             player->camera.Front = newFront;
108 
109             // Transform Up vector
110             glm::vec4 relUp = glm::inverse(srcRot) * glm::vec4(player->camera.Up, 0.0f);
111             glm::vec4 newUp4 = dstRot * (rot180 * relUp);
112             glm::vec3 newUp = glm::normalize(glm::vec3(newUp4));
113             player->camera.Up = newUp;
114 
115             // Transform Right vector
116             glm::vec4 relRight = glm::inverse(srcRot) * glm::vec4(player->camera.Right, 0.0f);
117             glm::vec4 newRight4 = dstRot * (rot180 * relRight);
118             glm::vec3 newRight = glm::normalize(glm::vec3(newRight4));
119             player->camera.Right = newRight;
120 
121             // Recompute yaw/pitch from new front vector
122             float pitch = glm::degrees(glm::asin(glm::clamp(newFront.y, -1.0f, 1.0f)));
123             float yaw = glm::degrees(std::atan2(newFront.z, newFront.x));
124             player->camera.Yaw = yaw;
125             player->camera.Pitch = pitch;
126 
127             // Calculate Roll from the new Up and Right
128             glm::vec3 right0 = glm::normalize(glm::cross(newFront, player->camera.WorldUp));
129             float cosRoll = glm::dot(newRight, right0);
130             glm::vec3 crossVec = glm::cross(right0, newRight);
131             float sinRoll = glm::dot(crossVec, newFront);
132             float roll = glm::degrees(atan2(sinRoll, cosRoll));
133             player->camera.Roll = roll;
134 
135             // Start roll recovery
136             player->initialRoll = roll;
137             player->rollRecoveryDuration = abs(roll) / 180.0f * 0.6f + 0.2f;
138             player->rollRecoveryTimer = player->rollRecoveryDuration;
139 
140             // Also move camera position to follow player properly
141             player->camera.Position = player->position + glm::vec3(0.0f, player->height * 0.4f, 0.0
                    f);
142         }
143         };
144     teleportTrigger->isActive = false;
145     scene->addTrigger(name + "TeleportTrigger", std::unique_ptr<Trigger>(teleportTrigger));
146 }
147 
148 void Portal::createFrames(Model *cubeModel, float thickness, float depth) {
149     for (int i = 0; i < 4; ++i) {
150         frames[i] = std::make_unique<GameObject>(cubeModel);
151     }
152     // Apply initial transform
153     updateFramesTransform();
```

```
    // set initial scales: top/bottom span portal width, left/right span portal height
    frames[0]->scale = glm::vec3(scale.x, thickness, depth);
    frames[1]->scale = glm::vec3(scale.x, thickness, depth);
    frames[2]->scale = glm::vec3(thickness, scale.y, depth);
    frames[3]->scale = glm::vec3(thickness, scale.y, depth);
}

void Portal::registerFramesPhysics(Scene *scene, uint32_t collisionMask) {
    if (!scene) return;
    for (int i = 0; i < 4; ++i) {
        if (frames[i]) {
            scene->addPhysics(frames[i].get(), true, collisionMask);
        }
    }
}

void Portal::updateFramesTransform() {
    // Compute axes from portal rotation (Ry * Rx * Rz)
    glm::mat4 rotationMat = glm::mat4(1.0f);
    rotationMat = glm::rotate(rotationMat, glm::radians(rotation.y), glm::vec3(0.0f, 1.0f, 0.0f));
    rotationMat = glm::rotate(rotationMat, glm::radians(rotation.x), glm::vec3(1.0f, 0.0f, 0.0f));
    rotationMat = glm::rotate(rotationMat, glm::radians(rotation.z), glm::vec3(0.0f, 0.0f, 1.0f));

    glm::vec3 axes[3];
    axes[0] = glm::vec3(rotationMat[0]); // right
    axes[1] = glm::vec3(rotationMat[1]); // up
    axes[2] = glm::vec3(rotationMat[2]); // forward

    float halfW = scale.x;
    float halfH = scale.y;
    // Determine current thickness/depth from frame scales if present
    float thickness = 0.05f;
    float depth = 0.1f;

    // Top
    if (frames[0]) {
        frames[0]->rotation = rotation;
        frames[0]->position = position + axes[1] * (halfH + 0.18f - thickness * 0.5f) - axes[2] *
            (depth * 2.0f - 0.05f);
    }
    // Bottom
    if (frames[1]) {
        frames[1]->rotation = rotation;
        frames[1]->position = position + axes[1] * (-halfH - 0.18f + thickness * 0.5f) - axes[2] *
             (depth * 2.0f - 0.05f);
    }
    // Left
    if (frames[2]) {
        frames[2]->rotation = rotation;
        frames[2]->position = position + axes[0] * (-halfW - 0.18f + thickness * 0.5f) - axes[2] *
             (depth * 2.0f - 0.05f);
```

```
202     }
203     // Right
204     if (frames[3]) {
205         frames[3]->rotation = rotation;
206         frames[3]->position = position + axes[0] * (halfW + 0.18f - thickness * 0.5f) - axes[2] *
                (depth * 2.0f - 0.05f);
207     }
208 }
209 
210 void Portal::checkRaycast(RaycastHit result, glm::vec3 playerRight) {
211     if (result.hit) {
212         if (!result.object->canOpenPortal) {
213             return;
214         }
215         if (onObject) {
216             onObject->setCollisionMask(COLLISION_MASK_DEFAULT);
217         }
218         result.object->setCollisionMask(COLLISION_MASK_PORTALON);
219         onObject = result.object;
220 
221         //set position and rotation
222         position = result.point + result.normal * 0.05f;
223         float pitch = -glm::degrees(std::asin(glm::clamp(result.normal.y, -1.0f, 1.0f)));
224         float yaw = 0.0f;
225         if (glm::abs(result.normal.x) > 1e-5 || glm::abs(result.normal.z) > 1e-5) {
226             yaw = glm::degrees(std::atan2(result.normal.x, result.normal.z));
227         }
228         auto portalUp = glm::cross(playerRight, result.normal);
229         float roll = 0.0f;
230         if (abs(portalUp.y) <= 1e-5) {//only roll when on floor/ceil
231             if (result.normal.y > 0) {
232                 roll = glm::degrees(std::atan2(portalUp.x, portalUp.z));
233             } else { //ceil
234                 roll = -glm::degrees(std::atan2(portalUp.x, portalUp.z));
235             }
236         }
237         rotation = glm::vec3(pitch, yaw, roll);
238 
239         // move trigger
240         glm::mat4 rotationMat = glm::mat4(1.0f);
241         rotationMat = glm::rotate(rotationMat, glm::radians(rotation.y), glm::vec3(0.0f, 1.0f, 0.0
                f));
242         rotationMat = glm::rotate(rotationMat, glm::radians(rotation.x), glm::vec3(1.0f, 0.0f, 0.0
                f));
243         rotationMat = glm::rotate(rotationMat, glm::radians(rotation.z), glm::vec3(0.0f, 0.0f, 1.0
                f));
244 
245         glm::vec3 axes[3];
246         axes[0] = glm::vec3(rotationMat[0]); // right
247         axes[1] = glm::vec3(rotationMat[1]); // up
248         axes[2] = glm::vec3(rotationMat[2]); // forward (portal normal)
```

```
249
250         // Portal width/height from portal->scale (assume x=width, y=height)
251         float halfWidth = scale.x;
252         float halfHeight = scale.y;
253         nearTrigger->setFromCenterAxesExtents(position + result.normal * 0.9f, axes, glm::vec3(
                halfWidth, halfHeight, 1.0f));
254         teleportTrigger->setFromCenterAxesExtents(position - result.normal * 0.4f, axes, glm::vec3
                (halfWidth, halfHeight, 0.46f));
255         // Update frames to follow portal
256         updateFramesTransform();
257         isActive = true;
258         if (linkedPortal->isActive) {
259             nearTrigger->isActive = true;
260             teleportTrigger->isActive = true;
261             linkedPortal->nearTrigger->isActive = true;
262             linkedPortal->teleportTrigger->isActive = true;
263         }
264     }
265 }
266
267 void Portal::beginRender() {
268     currentBuffer = (currentBuffer + 1) % 2;
269     frameBuffer[currentBuffer]->Bind();
270     glClearColor(0.1f, 0.1f, 0.1f, 1.0f);
271     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
272 }
273
274 void Portal::endRender(int scrWidth, int scrHeight) {
275     frameBuffer[currentBuffer]->Unbind();
276     glViewport(0, 0, scrWidth, scrHeight);
277 }
278
279 glm::mat4 Portal::getTransformedView(glm::mat4 view) {
280     if (!linkedPortal) return view;
281
282     glm::mat4 myModel = glm::mat4(1.0f);
283     myModel = glm::translate(myModel, position);
284     myModel = glm::rotate(myModel, glm::radians(rotation.y), glm::vec3(0.0f, 1.0f, 0.0f));
285     myModel = glm::rotate(myModel, glm::radians(rotation.x), glm::vec3(1.0f, 0.0f, 0.0f));
286     myModel = glm::rotate(myModel, glm::radians(rotation.z), glm::vec3(0.0f, 0.0f, 1.0f));
287
288     glm::mat4 otherModel = glm::mat4(1.0f);
289     otherModel = glm::translate(otherModel, linkedPortal->position);
290     otherModel = glm::rotate(otherModel, glm::radians(linkedPortal->rotation.y), glm::vec3(0.0f,
            1.0f, 0.0f));
291     otherModel = glm::rotate(otherModel, glm::radians(linkedPortal->rotation.x), glm::vec3(1.0f,
            0.0f, 0.0f));
292     otherModel = glm::rotate(otherModel, glm::radians(linkedPortal->rotation.z), glm::vec3(0.0f,
            0.0f, 1.0f));
293
294     // Calculate Relative Transform (Camera -> Me)
```

```
295     glm::mat4 camTransform = glm::inverse(view);
296 
297     // Transform Camera to My Local Space
298     glm::mat4 camInLocal = glm::inverse(myModel) * camTransform;
299 
300     // Rotate 180 degrees around Y (to face out)
301     glm::mat4 rotation180 = glm::rotate(glm::mat4(1.0f), glm::radians(180.0f), glm::vec3(0.0f, 1.0
            f, 0.0f));
302     glm::mat4 destView = otherModel * rotation180 * glm::inverse(myModel) * camTransform;
303 
304     return glm::inverse(destView);
305 }
306 
307 glm::vec4 Portal::getPlaneEquation() {
308     if (!linkedPortal) return glm::vec4(0.0f);
309 
310     glm::mat4 modelMatrix = glm::mat4(1.0f);
311     modelMatrix = glm::translate(modelMatrix, linkedPortal->position);
312     modelMatrix = glm::rotate(modelMatrix, glm::radians(linkedPortal->rotation.y), glm::vec3(0.0f,
            1.0f, 0.0f));
313     modelMatrix = glm::rotate(modelMatrix, glm::radians(linkedPortal->rotation.x), glm::vec3(1.0f,
            0.0f, 0.0f));
314     modelMatrix = glm::rotate(modelMatrix, glm::radians(linkedPortal->rotation.z), glm::vec3(0.0f,
            0.0f, 1.0f));
315 
316     glm::vec3 normal = glm::vec3(modelMatrix * glm::vec4(0.0f, 0.0f, 1.0f, 0.0f));
317     normal = glm::normalize(normal);
318 
319     // Plane equation: Ax + By + Cz + D = 0
320     // D = -dot(N, P)
321     float d = -glm::dot(normal, linkedPortal->position);
322 
323     return glm::vec4(normal, d);
324 }
325 
326 void Portal::draw(Shader &portalShader, Shader &shader) {
327     drawBuffer(currentBuffer, portalShader);
328     DrawFrame(shader);
329 }
330 
331 void Portal::drawPrev(Shader &portalShader, Shader &shader) {
332     drawBuffer((currentBuffer + 1) % 2, portalShader);
333     DrawFrame(shader);
334 }
335 
336 void Portal::DrawFrame(Shader &shader) {
337     shader.use();
338     shader.setBool("useAlphaTest", true);
339     Texture *frameTex = (type == PORTAL_A) ? frameTextureA.get() : frameTextureB.get();
340     if (frameTex) {
341         glActiveTexture(GL_TEXTURE0);
```

```
342         glBindTexture(GL_TEXTURE_2D, frameTex->ID);
343         shader.setInt("material.texture_diffuse1", 0);
344 
345         // Set material properties for frame
346         shader.setVec3("material.ambientColor", glm::vec3(1.0f));
347         shader.setVec3("material.diffuseColor", glm::vec3(1.0f));
348         shader.setVec3("material.specularColor", glm::vec3(1.0f));
349         shader.setFloat("material.shininess", 32.0f);
350 
351         // Calculate frame position: slightly forward along normal to avoid z-fighting
352         glm::mat4 rotationMat = glm::mat4(1.0f);
353         rotationMat = glm::rotate(rotationMat, glm::radians(rotation.y), glm::vec3(0.0f, 1.0f, 0.0
                f));
354         rotationMat = glm::rotate(rotationMat, glm::radians(rotation.x), glm::vec3(1.0f, 0.0f, 0.0
                f));
355         rotationMat = glm::rotate(rotationMat, glm::radians(rotation.z), glm::vec3(0.0f, 0.0f, 1.0
                f));
356         glm::vec3 normal = glm::normalize(glm::vec3(rotationMat * glm::vec4(0.0f, 0.0f, 1.0f, 0.0f
                )));
357         glm::vec3 framePos = position + normal * 0.01f;
358 
359         glm::mat4 frameModel = glm::mat4(1.0f);
360         frameModel = glm::translate(frameModel, framePos);
361         frameModel = glm::rotate(frameModel, glm::radians(rotation.y), glm::vec3(0.0f, 1.0f, 0.0f)
                );
362         frameModel = glm::rotate(frameModel, glm::radians(rotation.x), glm::vec3(1.0f, 0.0f, 0.0f)
                );
363         frameModel = glm::rotate(frameModel, glm::radians(rotation.z), glm::vec3(0.0f, 0.0f, 1.0f)
                );
364         frameModel = glm::scale(frameModel, scale + glm::vec3(0.2f));
365         shader.setMat4("model", frameModel);
366 
367         glBindVertexArray(contentVAO);
368         glDrawArrays(GL_TRIANGLES, 0, 6);
369         glBindVertexArray(0);
370     }
371     shader.setBool("useAlphaTest", false);
372     glActiveTexture(GL_TEXTURE0);
373 }
374 
375 void Portal::drawBuffer(int bufferIndex, Shader &portalShader) {
376     if (!isActive || !linkedPortal->isActive) return;
377     glActiveTexture(GL_TEXTURE10);
378     glBindTexture(GL_TEXTURE_2D, frameBuffer[bufferIndex]->GetTextureID());
379     portalShader.use();
380     portalShader.setInt("reflectionTexture", 10);
381 
382     // Calculate model matrix
383     glm::mat4 model = glm::mat4(1.0f);
384     model = glm::translate(model, position);
385     model = glm::rotate(model, glm::radians(rotation.y), glm::vec3(0.0f, 1.0f, 0.0f));
```

```
    model = glm::rotate(model, glm::radians(rotation.x), glm::vec3(1.0f, 0.0f, 0.0f));
    model = glm::rotate(model, glm::radians(rotation.z), glm::vec3(0.0f, 0.0f, 1.0f));
    model = glm::scale(model, scale);

    portalShader.setMat4("model", model);

    glBindVertexArray(contentVAO);
    glDrawArrays(GL_TRIANGLES, 0, 6);
    glBindVertexArray(0);
}
```

A.4 物理与触发器

Listing 22: src/PhysicsSystem.cpp

```
#include "PhysicsSystem.h"
#include "GameObject.h"

#include <algorithm>
#include <iostream>

#include <glm/gtx/norm.hpp>


PhysicsSystem::PhysicsSystem() : gravity(glm::vec3(0.0f, -9.81f, 0.0f)) {}

PhysicsSystem::~PhysicsSystem() {}

void PhysicsSystem::setGravity(const glm::vec3 &g) {
    gravity = g;
}

void PhysicsSystem::addObject(GameObject *obj, RigidBody *rb, AABB *col) {
    PhysicsObject physObj;
    physObj.gameObject = obj;
    physObj.rigidBody = rb;
    physObj.collider = col;
    physicsObjects.push_back(physObj);
}

void PhysicsSystem::removeObject(GameObject *obj) {
    physicsObjects.erase(std::remove_if(physicsObjects.begin(), physicsObjects.end(),
        [obj](const PhysicsObject &pObj) { return pObj.gameObject == obj; }), physicsObjects.end()
            );
}

void PhysicsSystem::update(float dt) {
    // Ground detection
    for (auto &obj : physicsObjects) {
        if (obj.rigidBody && !obj.rigidBody->isStatic && obj.collider) {
            // Raycast downward to detect ground
```

```
36          glm::vec3 rayOrigin = obj.gameObject->position;
37          glm::vec3 rayDirection = glm::vec3(0.0f, -1.0f, 0.0f); // Down
38          float rayLength = (obj.collider->max.y - obj.collider->min.y) * obj.gameObject->scale.y
                + 0.1f; // Slightly more than collider height
39          RaycastHit hit = raycast(rayOrigin, rayDirection, rayLength);
40          obj.rigidBody->isOnGround = hit.hit;
41      } else {
42          obj.rigidBody->isOnGround = false;
43      }
44  }
45
46  // 1. Integration
47  for (auto &obj : physicsObjects) {
48      if (obj.rigidBody && !obj.rigidBody->isStatic) {
49          integrate(obj, dt);
50      }
51
52      // Update World OBB
53      if (obj.collider && obj.gameObject) {
54          // Calculate rotation matrix from Euler angles
55          glm::mat4 rotationMat = glm::mat4(1.0f);
56          rotationMat = glm::rotate(rotationMat, glm::radians(obj.gameObject->rotation.x), glm::
                vec3(1.0f, 0.0f, 0.0f));
57          rotationMat = glm::rotate(rotationMat, glm::radians(obj.gameObject->rotation.y), glm::
                vec3(0.0f, 1.0f, 0.0f));
58          rotationMat = glm::rotate(rotationMat, glm::radians(obj.gameObject->rotation.z), glm::
                vec3(0.0f, 0.0f, 1.0f));
59
60          // Extract axes
61          glm::vec3 axes[3];
62          axes[0] = glm::vec3(rotationMat[0]); // Right
63          axes[1] = glm::vec3(rotationMat[1]); // Up
64          axes[2] = glm::vec3(rotationMat[2]); // Forward
65
66          // Calculate center and half extents
67          glm::vec3 localCenter = (obj.collider->min + obj.collider->max) * 0.5f;
68          glm::vec3 localExtent = (obj.collider->max - obj.collider->min) * 0.5f;
69
70          // Apply scale
71          glm::vec3 scaledExtent = localExtent * obj.gameObject->scale;
72
73          // Transform center to world space
74          // Note: We need to rotate the local center offset first, then add to position
75          glm::vec3 rotatedCenterOffset = glm::vec3(rotationMat * glm::vec4(localCenter * obj.
                gameObject->scale, 1.0f));
76          glm::vec3 worldCenter = obj.gameObject->position + rotatedCenterOffset;
77
78          obj.worldOBB = OBB(worldCenter, axes, scaledExtent);
79      }
80  }
81
```

```
    // 2. Collision Detection & Resolution
    checkCollisions();
}

void PhysicsSystem::integrate(PhysicsObject &obj, float dt) {
    RigidBody *rb = obj.rigidBody;
    GameObject *go = obj.gameObject;

    // Apply Gravity
    if (rb->useGravity) {
        rb->addForce(gravity * rb->mass);
    }

    // F = ma -> a = F/m
    glm::vec3 acc = rb->acceleration + (rb->force / rb->mass);

    // v = v0 + at
    rb->velocity += acc * dt;

    // Apply Damping/Friction
    float dampingFactor = rb->isOnGround ? rb->friction : 0.05f; // Ground friction vs air
        resistance
    rb->velocity *= (1.0f - dt * dampingFactor);

    rb->velocity.y = std::max(rb->velocity.y, -40.0f);

    // x = x0 + vt
    go->position += rb->velocity * dt;

    // Clear forces for next frame
    rb->clearForces();
}

void PhysicsSystem::checkCollisions() {
    for (size_t i = 0; i < physicsObjects.size(); ++i) {
        for (size_t j = i + 1; j < physicsObjects.size(); ++j) {
            PhysicsObject &a = physicsObjects[i];
            PhysicsObject &b = physicsObjects[j];

            if (!a.rigidBody->isCollisionEnabled || !b.rigidBody->isCollisionEnabled) continue;
            if (a.rigidBody->isStatic && b.rigidBody->isStatic) continue; // Skip static-static

            // Check collision masks
            if ((a.rigidBody->collisionMask & b.rigidBody->collisionMask) == 0) continue;

            glm::vec3 normal;
            float penetration;
            if (checkCollisionSAT(a.worldOBB, b.worldOBB, normal, penetration)) {
                resolveCollision(a, b, normal, penetration);
            }
        }
```

```
    }
}

// Helper for SAT test on a single axis
bool TestAxis(const glm::vec3 &axis, const OBB &a, const OBB &b, float &minPenetration, glm::vec3 &bestAxis) {
    if (glm::length2(axis) < 1e-6f) return true; // Skip near-zero axis
    glm::vec3 nAxis = glm::normalize(axis);

    // Project A
    float rA = glm::abs(glm::dot(a.axes[0], nAxis) * a.halfExtents.x) +
        glm::abs(glm::dot(a.axes[1], nAxis) * a.halfExtents.y) +
        glm::abs(glm::dot(a.axes[2], nAxis) * a.halfExtents.z);

    // Project B
    float rB = glm::abs(glm::dot(b.axes[0], nAxis) * b.halfExtents.x) +
        glm::abs(glm::dot(b.axes[1], nAxis) * b.halfExtents.y) +
        glm::abs(glm::dot(b.axes[2], nAxis) * b.halfExtents.z);

    // Distance between centers projected
    float dist = glm::abs(glm::dot(b.center - a.center, nAxis));

    float penetration = rA + rB - dist;

    // If penetration is negative, there is a gap -> no collision
    if (penetration < 0) return false;

    if (penetration < minPenetration) {
        minPenetration = penetration;
        bestAxis = nAxis;
    }
    return true;
}

bool PhysicsSystem::checkCollisionSAT(const OBB &a, const OBB &b, glm::vec3 &outNormal, float &outPenetration) {
    float minPenetration = std::numeric_limits<float>::max();
    glm::vec3 bestAxis;

    // Test 3 axes of A
    for (int i = 0; i < 3; i++) {
        if (!TestAxis(a.axes[i], a, b, minPenetration, bestAxis)) return false;
    }

    // Test 3 axes of B
    for (int i = 0; i < 3; i++) {
        if (!TestAxis(b.axes[i], a, b, minPenetration, bestAxis)) return false;
    }

    // Test 9 cross products
    for (int i = 0; i < 3; i++) {
```

```
        for (int j = 0; j < 3; j++) {
            glm::vec3 axis = glm::cross(a.axes[i], b.axes[j]);
            if (!TestAxis(axis, a, b, minPenetration, bestAxis)) return false;
        }
    }

    outPenetration = minPenetration;

    // Ensure normal points from B to A
    if (glm::dot(bestAxis, a.center - b.center) < 0) {
        outNormal = -bestAxis;
    } else {
        outNormal = bestAxis;
    }

    return true;
}

void PhysicsSystem::resolveCollision(PhysicsObject &a, PhysicsObject &b, const glm::vec3 &normal,
     float penetration) {
    // Helper for velocity reflection
    auto reflectVelocity = [](RigidBody *rb, float &velocityComponent, float restitution, float
         sign) {
        // Only reflect if moving towards the collision
        if (velocityComponent * sign < 0) {
            velocityComponent = -velocityComponent * restitution;

            // Stop micro-bouncing (resting contact threshold)
            if (std::abs(velocityComponent) < 0.5f) {
                velocityComponent = 0.0f;
            }
        }
        };

    if (!a.rigidBody->isStatic && !b.rigidBody->isStatic) {
        // Both dynamic: split penetration
        a.gameObject->position += normal * penetration * 0.5f;
        b.gameObject->position -= normal * penetration * 0.5f;

        // Simple velocity reflection along normal
        glm::vec3 vRel = a.rigidBody->velocity - b.rigidBody->velocity;
        float vRelNormal = glm::dot(vRel, normal);

        // Do not resolve if velocities are separating
        if (vRelNormal > 0) return;

        float e = std::min(a.rigidBody->restitution, b.rigidBody->restitution);
        float j = -(1 + e) * vRelNormal;
        j /= (1 / a.rigidBody->mass + 1 / b.rigidBody->mass);

        glm::vec3 impulse = j * normal;
```

```
        a.rigidBody->velocity += impulse / a.rigidBody->mass;
        b.rigidBody->velocity -= impulse / b.rigidBody->mass;

    } else if (!a.rigidBody->isStatic) {
        // A is dynamic
        a.gameObject->position += normal * penetration;

        glm::vec3 vRel = a.rigidBody->velocity; // B is static, vB = 0
        float vRelNormal = glm::dot(vRel, normal);

        if (vRelNormal > 0) return;

        float e = std::max(a.rigidBody->restitution, b.rigidBody->restitution);

        // Apply impulse
        glm::vec3 vn = vRelNormal * normal;
        glm::vec3 vt = vRel - vn; // Tangential velocity (friction could be applied here)

        // Debug print
        // std::cout << "Collision! Normal: " << normal.x << ", " << normal.y << ", " << normal.z
        // << " vRelNormal: " << vRelNormal
        // << " e: " << e << std::endl;

        a.rigidBody->velocity = vt - vn * e;

        // Threshold for resting
        // if (glm::length(a.rigidBody->velocity) < 0.1f) a.rigidBody->velocity = glm::vec3(0.0f);
        if (glm::length(a.rigidBody->velocity) < 0.5f) {
            // Only zero out if normal velocity is small (resting contact)
            if (std::abs(vRelNormal) < 1.0f) {
                a.rigidBody->velocity = glm::vec3(0.0f);
            }
        }

    } else if (!b.rigidBody->isStatic) {
        // B is dynamic
        b.gameObject->position -= normal * penetration;

        glm::vec3 vRel = -b.rigidBody->velocity; // A is static, vA = 0
        float vRelNormal = glm::dot(vRel, normal);

        if (vRelNormal > 0) return;

        float e = std::max(a.rigidBody->restitution, b.rigidBody->restitution);

        glm::vec3 vn = vRelNormal * normal;
        glm::vec3 vt = vRel - vn;

        // Note: vRel was A-B, so -vB.
        // New vB should be reflected.
        // Let's simplify: just reflect B's velocity along normal
```

```
        glm::vec3 vB = b.rigidBody->velocity;
        float vBn = glm::dot(vB, normal);
        glm::vec3 vBnVec = vBn * normal;
        glm::vec3 vBtVec = vB - vBnVec;

        b.rigidBody->velocity = vBtVec - vBnVec * e;

        // if (glm::length(b.rigidBody->velocity) < 0.1f) b.rigidBody->velocity = glm::vec3(0.0f);
        if (glm::length(b.rigidBody->velocity) < 0.5f) {
            if (std::abs(vRelNormal) < 1.0f) {
                b.rigidBody->velocity = glm::vec3(0.0f);
            }
        }
    }
}

RaycastHit PhysicsSystem::raycast(const glm::vec3 &origin, const glm::vec3 &direction, float
     maxDistance) {
    RaycastHit hit;
    hit.distance = maxDistance;

    glm::vec3 invDir = 1.0f / direction;

    for (auto &obj : physicsObjects) {
        // Ray vs OBB
        // Transform ray to OBB local space
        OBB &obb = obj.worldOBB;

        glm::vec3 p = obb.center - origin;

        glm::vec3 f(
            glm::dot(obb.axes[0], direction),
            glm::dot(obb.axes[1], direction),
            glm::dot(obb.axes[2], direction)
        );

        glm::vec3 e(
            glm::dot(obb.axes[0], p),
            glm::dot(obb.axes[1], p),
            glm::dot(obb.axes[2], p)
        );

        float t[6] = { 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f };
        // Test against 3 pairs of planes
        float tmin = 0.0f;
        float tmax = maxDistance;

        for (int i = 0; i < 3; i++) {
            float r = obb.halfExtents[i];
            if (std::abs(f[i]) > 1e-6f) {
                float t1 = (e[i] + r) / f[i];
```

```
331                 float t2 = (e[i] - r) / f[i];
332                 if (t1 > t2) std::swap(t1, t2);
333                 if (t1 > tmin) tmin = t1;
334                 if (t2 < tmax) tmax = t2;
335                 if (tmin > tmax) goto next_obj;
336                 if (tmax < 0) goto next_obj;
337             } else if (-e[i] - r > 0 || -e[i] + r < 0) {
338                 goto next_obj;
339             }
340         }
341 
342         if (tmin > 0 && tmin < hit.distance) {
343             hit.hit = true;
344             hit.distance = tmin;
345             hit.point = origin + direction * tmin;
346             hit.object = obj.gameObject;
347 
348             // Calculate normal
349             glm::vec3 localPoint = hit.point - obb.center;
350             glm::vec3 normal(0.0f);
351             float minDepth = std::numeric_limits<float>::max();
352 
353             for (int i = 0; i < 3; ++i) {
354                 float dist = glm::dot(localPoint, obb.axes[i]);
355                 float depth = obb.halfExtents[i] - std::abs(dist);
356                 if (depth < minDepth) {
357                     minDepth = depth;
358                     normal = obb.axes[i] * (dist > 0 ? 1.0f : -1.0f);
359                 }
360             }
361             hit.normal = normal;
362         }
363 
364     next_obj:;
365     }
366 
367     return hit;
368 }
369 
370 bool PhysicsSystem::checkPlayerCollision(const AABB &playerAABB, glm::vec3 &outCorrection,
        uint32_t playerMask) {
371     outCorrection = glm::vec3(0.0f);
372     bool collided = false;
373 
374     // Convert Player AABB to OBB for SAT check
375     glm::vec3 center = (playerAABB.min + playerAABB.max) * 0.5f;
376     glm::vec3 extents = (playerAABB.max - playerAABB.min) * 0.5f;
377     glm::vec3 axes[3] = { glm::vec3(1,0,0), glm::vec3(0,1,0), glm::vec3(0,0,1) };
378     OBB playerOBB(center, axes, extents);
379 
380     for (auto &obj : physicsObjects) {
```

```
        // Check collision mask
        if ((obj.rigidBody->collisionMask & playerMask) == 0) continue;

        // if (!obj.rigidBody->isStatic) continue; // Allow collision with dynamic objects

        glm::vec3 normal;
        float penetration;
        if (checkCollisionSAT(playerOBB, obj.worldOBB, normal, penetration)) {
            // Accumulate correction? Or just take the largest?
            // For simple character controller, resolving one by one is okay-ish,
            // but taking the max penetration is safer to avoid jitter.
            // Here we just apply the first one found (simple) or sum them (can be unstable).
            // Let's try resolving immediately.

            // Important: SAT normal points from B to A.
            // Here A is player, B is obj. So normal points towards player.
            // We want to push player OUT of obj.

            outCorrection += normal * penetration;
            collided = true;

            // Simple interaction: Push dynamic objects
            if (!obj.rigidBody->isStatic) {
                // Apply a small impulse to the object away from the player
                // Normal points from Obj to Player, so -normal is force direction
                glm::vec3 pushForce = -normal * 10.0f;
                pushForce.y = 0.0f; // Keep it horizontal for now to avoid stomping
                obj.rigidBody->addForce(pushForce);

                // Also wake it up if sleeping (not implemented yet, but good practice)
            }
        }
    }
    return collided;
}
```

Listing 23: src/Trigger.cpp

```
#include "Trigger.h"
#include "GameObject.h"

#include <cmath>

#include <glad/gl.h>
#include <glm/gtc/type_ptr.hpp>

Trigger::Trigger(const OBB &obb) : bounds(obb) {}

void Trigger::check(GameObject *obj) {
    if (!isActive || !obj) return;

    bool inside = isPointInside(obj->position);
```

```
    bool wasInside = objectsInside.count(obj) > 0;

    if (inside && !wasInside) {
        objectsInside.insert(obj);
        if (onEnter) onEnter(obj);
    } else if (!inside && wasInside) {
        objectsInside.erase(obj);
        if (onExit) onExit(obj);
    }

    if (inside && onInside) {
        onInside(obj);
    }
}

void Trigger::drawOBBDebug(Shader &shader) {
    if (!isActive) return;

    // Prepare 8 corner points of the OBB in world space
    glm::vec3 c = bounds.center;
    glm::vec3 ax = bounds.axes[0] * bounds.halfExtents[0];
    glm::vec3 ay = bounds.axes[1] * bounds.halfExtents[1];
    glm::vec3 az = bounds.axes[2] * bounds.halfExtents[2];

    glm::vec3 corners[8];
    // order: 0(-x,-y,-z),1(+x,-y,-z),2(-x,-y,+z),3(+x,-y,+z),
    // 4(-x,+y,-z),5(+x,+y,-z),6(-x,+y,+z),7(+x,+y,+z)
    corners[0] = c - ax - ay - az;
    corners[1] = c + ax - ay - az;
    corners[2] = c - ax - ay + az;
    corners[3] = c + ax - ay + az;
    corners[4] = c - ax + ay - az;
    corners[5] = c + ax + ay - az;
    corners[6] = c - ax + ay + az;
    corners[7] = c + ax + ay + az;

    // 12 edges, each edge has 2 vertices -> 24 vertices
    float verts[24 * 3];
    int vi = 0;
    auto pushEdge = [&](int a, int b) {
        verts[vi++] = corners[a].x; verts[vi++] = corners[a].y; verts[vi++] = corners[a].z;
        verts[vi++] = corners[b].x; verts[vi++] = corners[b].y; verts[vi++] = corners[b].z;
        };

    // bottom face (y -)
    pushEdge(0, 1);
    pushEdge(1, 3);
    pushEdge(3, 2);
    pushEdge(2, 0);
    // top face (y +)
    pushEdge(4, 5);
```

```
    pushEdge(5, 7);
    pushEdge(7, 6);
    pushEdge(6, 4);
    // vertical edges
    pushEdge(0, 4);
    pushEdge(1, 5);
    pushEdge(2, 6);
    pushEdge(3, 7);

    // Use a simple dynamic VBO/VAO (static to reuse across calls)
    static unsigned int dbgVAO = 0, dbgVBO = 0;
    if (dbgVAO == 0) {
        glGenVertexArrays(1, &dbgVAO);
        glGenBuffers(1, &dbgVBO);
    }

    shader.use();
    // set model to identity (we already provide world-space positions)
    shader.setMat4("model", glm::mat4(1.0f));

    glBindVertexArray(dbgVAO);
    glBindBuffer(GL_ARRAY_BUFFER, dbgVBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(verts), verts, GL_DYNAMIC_DRAW);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void *)0);

    // Draw lines
    glDrawArrays(GL_LINES, 0, 24);

    glBindVertexArray(0);
}

bool Trigger::isPointInside(const glm::vec3 &point) const {
    // Transform point to OBB local space
    glm::vec3 d = point - bounds.center;

    // Project d onto each axis and check against halfExtents
    for (int i = 0; i < 3; ++i) {
        float dist = glm::dot(d, bounds.axes[i]);
        if (std::abs(dist) > bounds.halfExtents[i]) {
            return false;
        }
    }
    return true;
}
```

Listing 24: src/Trigger.h

```
#pragma once
#include "PhysicsSystem.h"
#include "Shader.h"

```

```
#include <unordered_set>
#include <functional>

#include <glm/glm.hpp>

class GameObject;

class Trigger {
public:
    Trigger(const OBB &obb);
    Trigger(const glm::vec3 &min, const glm::vec3 &max) {
        glm::vec3 center = (min + max) * 0.5f;
        glm::vec3 halfExtents = (max - min) * 0.5f;
        glm::vec3 axes[3] = {
            glm::vec3(1.0f, 0.0f, 0.0f),
            glm::vec3(0.0f, 1.0f, 0.0f),
            glm::vec3(0.0f, 0.0f, 1.0f)
        };
        bounds = OBB(center, axes, halfExtents);
    }

    OBB bounds;
    bool isActive = true;

    // Callbacks
    // Pass the object pointer that triggered it.
    std::function<void(GameObject *obj)> onEnter;
    std::function<void(GameObject *obj)> onExit;
    std::function<void(GameObject *obj)> onInside;

    // Check a specific object
    void check(GameObject *obj);
    void drawOBBDebug(Shader &shader);

    // Reset the OBB bounds
    void setBounds(const OBB &obb) { bounds = obb; }

    // Helper to set OBB from center, axes and half extents
    void setFromCenterAxesExtents(const glm::vec3 &center, const glm::vec3 axes[3], const glm::
        vec3 &halfExtents) {
        bounds = OBB(center, axes, halfExtents);
    }

private:
    std::unordered_set<GameObject *> objectsInside;

    bool isPointInside(const glm::vec3 &point) const;
};
```

A.5 玩家与交互

Listing 25: src/Player.cpp

```cpp
#include "Player.h"
#include "Scene.h"

#include <iostream>
#include <algorithm>

Player::Player(glm::vec3 startPos)
    : GameObject(nullptr, startPos),
    camera(startPos),
    isGrounded(false) {

    isTeleportable = true;
    // Initialize RigidBody
    rigidBody = std::make_unique<RigidBody>();
    rigidBody->velocity = glm::vec3(0.0f);
    rigidBody->useGravity = false; // We handle gravity manually in update()
    rigidBody->isStatic = false;
    rigidBody->collisionMask = COLLISION_MASK_DEFAULT;

    // Initialize collider (centered on position)
    glm::vec3 min = position + glm::vec3(-radius, -height * 0.5f, -radius);
    glm::vec3 max = position + glm::vec3(radius, height * 0.5f, radius);
    collider = std::make_unique<AABB>(min, max);

    // Set correct camera height
    camera.Position = position + glm::vec3(0.0f, height * 0.4f, 0.0f);
}

void Player::processInput(const InputManager &input, Scene *scene, float dt) {
    glm::vec3 targetVel(0.0f);

    glm::vec3 forward = camera.Front;
    forward.y = 0.0f;
    forward = glm::normalize(forward);

    glm::vec3 right = camera.Right;
    right.y = 0.0f;
    right = glm::normalize(right);

    if (input.isKeyDown(GLFW_KEY_W)) targetVel += forward;
    if (input.isKeyDown(GLFW_KEY_S)) targetVel -= forward;
    if (input.isKeyDown(GLFW_KEY_A)) targetVel -= right;
    if (input.isKeyDown(GLFW_KEY_D)) targetVel += right;

    if (glm::length(targetVel) > 0.0f) {
        targetVel = glm::normalize(targetVel) * moveSpeed;
    } else if (!isGrounded) {
        targetVel.x = rigidBody->velocity.x;
        targetVel.z = rigidBody->velocity.z;
    } else {
```

```
        targetVel = glm::vec3(0.0f);
    }

    // Smoothly interpolate horizontal velocity (simple acceleration)
    float accel = 10.0f;
    if (!isGrounded) accel = 2.0f; // Less control in air

    rigidBody->velocity.x = glm::mix(rigidBody->velocity.x, targetVel.x, accel * dt);
    rigidBody->velocity.z = glm::mix(rigidBody->velocity.z, targetVel.z, accel * dt);

    // Jump
    if (input.isKeyPressed(GLFW_KEY_SPACE) && isGrounded) {
        rigidBody->velocity.y = jumpForce;
        isGrounded = false;
    }

    //grabbing
    if (input.isKeyPressed(GLFW_KEY_E)) {
        if (isGrabbing) {
            isGrabbing = false;
        } else {
            auto result = scene->physicsSystem->raycast(camera.Position, camera.Front, 5.0f);
            if (result.hit && result.object && result.object->isTeleportable) {
                isGrabbing = true;
                grabbedObject = result.object;
            }
        }
    }

    //respawn portal
    if (input.isMousePressed(GLFW_MOUSE_BUTTON_LEFT)) {
        if (isGrabbing) {
            glm::vec3 pushForce = camera.Front * 10.0f;
            grabbedObject->rigidBody->addForce(pushForce);
            isGrabbing = false;
        } else {
            scene->portalGun->fire();
            auto result = scene->physicsSystem->raycast(camera.Position, camera.Front, 100.0f);
            scene->portalA->checkRaycast(result,camera.Right);
        }
    }
    if (input.isMousePressed(GLFW_MOUSE_BUTTON_RIGHT) && !isGrabbing) {
        scene->portalGun->fire();
        auto result = scene->physicsSystem->raycast(camera.Position, camera.Front, 100.0f);
        scene->portalB->checkRaycast(result,camera.Right);
    }

    // Roll recovery after teleport
    if (rollRecoveryTimer > 0) {
        rollRecoveryTimer -= dt;
        if (rollRecoveryTimer < 0) rollRecoveryTimer = 0;
```

```
        float t = rollRecoveryTimer / rollRecoveryDuration;
        camera.Roll = initialRoll * t;
        camera.updateCameraVectors();
    }
}

void Player::update(float dt, PhysicsSystem *physicsSystem) {
    // Apply Gravity
    rigidBody->velocity.y -= gravity * dt;

    // Terminal velocity
    rigidBody->velocity.y = std::max(rigidBody->velocity.y, -20.0f);

    // Proposed movement
    glm::vec3 displacement = rigidBody->velocity * dt;

    position.y += displacement.y;
    // Update collider (centered on position)
    collider->min = position + glm::vec3(-radius, -height * 0.5f, -radius);
    collider->max = position + glm::vec3(radius, height * 0.5f, radius);

    if (!rigidBody->isCollisionEnabled) {
        // Skip collision check
    } else {
        glm::vec3 correction;
        if (physicsSystem->checkPlayerCollision(*collider, correction, rigidBody->collisionMask))
            {
            position += correction;
            if (rigidBody->velocity.y < 0 && correction.y > 0) {
                isGrounded = true;
                rigidBody->velocity.y = 0;
            } else if (rigidBody->velocity.y > 0 && correction.y < 0) {
                rigidBody->velocity.y = 0;
            }
        } else {
            isGrounded = false;
        }
    }


    // Move X/Z
    glm::vec3 horizontalDisp = glm::vec3(displacement.x, 0.0f, displacement.z);
    position += horizontalDisp;

    collider->min = position + glm::vec3(-radius, -height * 0.5f, -radius);
    collider->max = position + glm::vec3(radius, height * 0.5f, radius);

    if (rigidBody->isCollisionEnabled) {
        glm::vec3 correction;
        if (physicsSystem->checkPlayerCollision(*collider, correction, rigidBody->collisionMask))
            {
```

```
            position += correction;
        }
    }

    if (isGrabbing && grabbedObject) {
        glm::vec3 targetPos = position + camera.Front * 2.0f + glm::vec3(0.0f, height * 0.5f, 0.0f
            );
        if (glm::length(targetPos - grabbedObject->position) > 3.0f) {
            // Too far, release
            isGrabbing = false;
        } else {
            glm::vec3 springForce = (targetPos - grabbedObject->position) * 30.0f;
            glm::vec3 dumplingForce = -grabbedObject->rigidBody->velocity * 5.0f;
            grabbedObject->rigidBody->addForce(springForce + dumplingForce);
        }
    }

    // Sync Camera
    camera.Position = position + glm::vec3(0.0f, height * 0.4f, 0.0f); // Eyes at center + 0.4*
        height
}
```

Listing 26: src/Scene.h

```
#pragma once

#include "GameObject.h"
#include "Portal.h"
#include "Model.h"
#include "Skybox.h"
#include "PortalGun.h"
#include "PhysicsSystem.h"
#include "Player.h"
#include "Trigger.h"
#include "Button.h"
#include "Flip.h"

#include <vector>
#include <memory>
#include <unordered_map>

#include <glm/glm.hpp>


struct Scene {
    // Resource Management
    std::unordered_map<std::string, std::unique_ptr<Model>> modelResources;

    // Scene Graph
    std::unordered_map<std::string, std::unique_ptr<GameObject>> objects;
    std::unordered_map<std::string, std::unique_ptr<Trigger>> triggers;

```

```
    // Special Objects
    std::unique_ptr<Portal> portalA;
    std::unique_ptr<Portal> portalB;
    std::unique_ptr<Skybox> skybox;
    std::unique_ptr<PortalGun> portalGun;
    std::unique_ptr<Player> player;

    // Physics
    std::unique_ptr<PhysicsSystem> physicsSystem;

    Scene() {
        physicsSystem = std::make_unique<PhysicsSystem>();
    }

    // Lighting
    glm::vec3 lightPos;

    void addModelResource(const std::string &name, std::unique_ptr<Model> model) {
        if (modelResources.count(name)) {
            printf("Model resource %s already exists!\n", name.c_str());
            return;
        }
        modelResources[name] = std::move(model);
    }

    void addObject(std::string name, std::unique_ptr<GameObject> obj) {
        if (objects.count(name)) {
            printf("GameObject %s already exists!\n", name.c_str());
            return;
        }
        obj->name = name;
        objects[name] = std::move(obj);
    }

    void addPhysics(GameObject *obj, bool isStatic, uint32_t collisionMask =
         COLLISION_MASK_DEFAULT, float mass = 1.0f, float restitution = 0.2f, float friction = 0.5
         f) {
        obj->rigidBody = std::make_unique<RigidBody>();
        obj->rigidBody->isStatic = isStatic;
        obj->rigidBody->mass = mass;
        obj->rigidBody->restitution = restitution;
        obj->rigidBody->friction = friction;
        obj->rigidBody->collisionMask = collisionMask;
        if (!obj->collider) {
            obj->collider = std::make_unique<AABB>(obj->model->minBound, obj->model->maxBound);
        }
        physicsSystem->addObject(obj, obj->rigidBody.get(), obj->collider.get());
    }

    void addTrigger(std::string name, std::unique_ptr<Trigger> trigger) {
        if (triggers.count(name)) {
```

```
78          printf("Trigger %s already exists!\n", name.c_str());
79          return;
80      }
81      triggers[name] = std::move(trigger);
82  }
83
84  void update(float dt, const Camera &camera) {
85      // Update Physics
86      if (physicsSystem) {
87          physicsSystem->update(dt);
88      }
89
90      if (player) {
91          player->update(dt, physicsSystem.get());
92      }
93
94      for (auto &pair : objects) {
95          pair.second->update(dt, camera);
96      }
97
98      if (portalGun) {
99          portalGun->update(dt, camera);
100     }
101
102     for (auto &pair : triggers) {
103         if (player) {
104             pair.second->check(player.get());
105         }
106         for (auto &objPair : objects) {
107             pair.second->check(objPair.second.get());
108         }
109     }
110
111     auto button_flip = objects.find("button_flip");
112     if (button_flip != objects.end()) {
113         Button *btn = dynamic_cast<Button *>(button_flip->second.get());
114         if (btn && btn->getIsPressed()) {
115             auto flipWall = objects.find("flip_wall");
116             if (flipWall != objects.end()) {
117                 Flip *flip = dynamic_cast<Flip *>(flipWall->second.get());
118                 if (flip) {
119                     flip->flip();
120                 }
121             }
122         } else if (btn && btn->getIsReleased()) {
123             auto flipWall = objects.find("flip_wall");
124             if (flipWall != objects.end()) {
125                 Flip *flip = dynamic_cast<Flip *>(flipWall->second.get());
126                 if (flip) {
127                     flip->reset();
128                 }
```

```
            }
        }
    }

    auto flip_wall = objects.find("flip_wall");
    if (flip_wall != objects.end()) {
        Flip *flip = dynamic_cast<Flip *>(flip_wall->second.get());
        if (flip && flip->getIsRotating()) {
            if (portalA->getOnObject() == flip) {
                portalA->setOnObject(nullptr);
                portalA->isActive = false;
                portalA->position = glm::vec3(100.0f, 0.0f, 100.0f);
                portalA->getNearTrigger()->isActive = false;
                portalA->getTeleportTrigger()->isActive = false;
                portalB->getNearTrigger()->isActive = false;
                portalB->getTeleportTrigger()->isActive = false;
            }
            if (portalB->getOnObject() == flip) {
                portalB->setOnObject(nullptr);
                portalB->isActive = false;
                portalB->position = glm::vec3(100.0f, 0.0f, 100.0f);
                portalA->getNearTrigger()->isActive = false;
                portalA->getTeleportTrigger()->isActive = false;
                portalB->getNearTrigger()->isActive = false;
                portalB->getTeleportTrigger()->isActive = false;
            }
        }
    }
};
```

Listing 27: src/Button.cpp

```
#include "Button.h"

#include <iostream>

Button::Button(Model *model, glm::vec3 pos, glm::vec3 rot, glm::vec3 scale)
    : GameObject(model, pos, rot, scale), initialPosition(pos) {
    // Button presses down 0.05 units
    pressedPosition = pos - glm::vec3(0.0f, 0.05f, 0.0f);
}

std::unique_ptr<Trigger> Button::createTrigger() {
    glm::vec3 min = model->minBound * scale;
    glm::vec3 max = model->maxBound * scale;

    glm::vec3 center = initialPosition + (min + max) * 0.5f;
    glm::vec3 size = max - min;

    glm::vec3 triggerHalfExtents = size * 0.5f;
    triggerHalfExtents.y = 1.0f; // Height of trigger area
```

```
20
21      float topSurfaceY = initialPosition.y + max.y;
22      glm::vec3 triggerCenter = center;
23      triggerCenter.y = topSurfaceY + triggerHalfExtents.y * 0.5f;
24
25      glm::vec3 axes[3] = {
26          glm::vec3(1.0f, 0.0f, 0.0f),
27          glm::vec3(0.0f, 1.0f, 0.0f),
28          glm::vec3(0.0f, 0.0f, 1.0f)
29      };
30
31      auto trigger = std::make_unique<Trigger>(
32          triggerCenter - triggerHalfExtents,
33          triggerCenter + triggerHalfExtents
34      );
35
36      trigger->onEnter = [this](GameObject *obj) {
37          this->objectsOnButton++;
38          };
39
40      trigger->onExit = [this](GameObject *obj) {
41          this->objectsOnButton--;
42          if (this->objectsOnButton < 0) this->objectsOnButton = 0;
43      };
44
45      return trigger;
46  }
47
48  void Button::update(float dt, const Camera &camera) {
49      lastPressedState = isPressed;
50      isPressed = (objectsOnButton > 0);
51      glm::vec3 target = isPressed ? pressedPosition : initialPosition;
52
53      glm::vec3 diff = target - position;
54      float dist = glm::length(diff);
55
56      if (dist > 0.0001f) {
57          float moveStep = pressSpeed * dt;
58          if (moveStep >= dist) {
59              position = target;
60          } else {
61              position += glm::normalize(diff) * moveStep;
62          }
63      }
64  }
65
66  bool Button::getIsPressed() {
67      return (lastPressedState != isPressed && isPressed);
68  }
69
70  bool Button::getIsReleased() {
```

```
    return (lastPressedState != isPressed && !isPressed);
}
```

Listing 28: src/Flip.cpp

```
#include "Flip.h"
#include <glm/gtx/quaternion.hpp>
#include <glm/gtx/matrix_decompose.hpp>

Flip::Flip(Model *model, glm::vec3 pos, glm::vec3 rot, glm::vec3 scale)
    : GameObject(model, pos, rot, scale) {
    initialPosition = pos;
    initialRotation = rot;

    glm::quat qx = glm::angleAxis(glm::radians(initialRotation.x), glm::vec3(1.0f, 0.0f, 0.0f));
    glm::quat qy = glm::angleAxis(glm::radians(initialRotation.y), glm::vec3(0.0f, 1.0f, 0.0f));
    glm::quat qz = glm::angleAxis(glm::radians(initialRotation.z), glm::vec3(0.0f, 0.0f, 1.0f));
    glm::quat qInit = qx * qy * qz;

    rotationAxis = qInit * glm::vec3(0.0f, 0.0f, 1.0f);
    maxAngle = 45.0f;

    glm::vec3 center = model->getCenter();
    pivotLocal = center;
}

void Flip::setFlipAngle(float angle) {
    maxAngle = angle;
}

void Flip::setSpeed(float s) {
    speed = s;
}

void Flip::flip() {
    targetAngle = maxAngle;
}

void Flip::reset() {
    targetAngle = 0.0f;
}

void Flip::update(float dt, const Camera &camera) {
    if (std::abs(currentAngle - targetAngle) < 0.01f) {
        currentAngle = targetAngle;
        canOpenPortal = true;
        isRotating = false;
    } else {
        canOpenPortal = false;
        isRotating = true;
        float step = speed * dt;
        if (currentAngle < targetAngle) {
```

```
48            currentAngle += step;
49            if (currentAngle > targetAngle) currentAngle = targetAngle;
50        } else {
51            currentAngle -= step;
52            if (currentAngle < targetAngle) currentAngle = targetAngle;
53        }
54    }
55
56    glm::mat4 initModel = glm::mat4(1.0f);
57    initModel = glm::translate(initModel, initialPosition);
58    initModel = glm::rotate(initModel, glm::radians(initialRotation.x), glm::vec3(1.0f, 0.0f, 0.0f
          ));
59    initModel = glm::rotate(initModel, glm::radians(initialRotation.y), glm::vec3(0.0f, 1.0f, 0.0f
          ));
60    initModel = glm::rotate(initModel, glm::radians(initialRotation.z), glm::vec3(0.0f, 0.0f, 1.0f
          ));
61    glm::vec3 pivotWorld = initialPosition +
62        glm::quat(glm::radians(initialRotation)) * (pivotLocal * scale);
63
64    // Rotation Quaternion
65    glm::quat qRot = glm::angleAxis(glm::radians(currentAngle), rotationAxis);
66
67    // New Rotation
68    // R_new = qRot * R_init
69    glm::quat qInit = glm::quat(glm::radians(initialRotation));
70    glm::quat qFinal = qRot * qInit;
71
72    glm::vec3 centerNew = pivotWorld + qRot * (initialPosition - pivotWorld);
73    this->position = centerNew;
74    this->rotation = initialRotation + rotationAxis * currentAngle;
75}
```

A.6 模型与网格

Listing 29: src/Model.cpp

```
1  #include "Model.h"
2
3  #include <iostream>
4  #include <fstream>
5  #include <sstream>
6  #include <map>
7
8
9  Model::Model(std::string const &path) {
10     minBound = glm::vec3(std::numeric_limits<float>::max());
11     maxBound = glm::vec3(std::numeric_limits<float>::lowest());
12     loadModel(path);
13 }
14
15 glm::vec3 Model::getCenter() const {
```

```
    return (minBound + maxBound) * 0.5f;
}

float Model::getNormalizationScale() const {
    float maxDim = std::max(std::max(maxBound.x - minBound.x, maxBound.y - minBound.y), maxBound.z
        - minBound.z);
    if (maxDim == 0.0f) return 1.0f;
    return 2.0f / maxDim;
}

void Model::Draw(Shader &shader) {
    for (unsigned int i = 0; i < meshes.size(); i++)
        meshes[i].Draw(shader);
}

void Model::loadMTL(std::string const &path) {
    std::ifstream file(path);
    if (!file.is_open()) {
        std::cout << "Failed to open MTL file: " << path << std::endl;
        return;
    }

    std::string line;
    std::string currentMtlName;

    while (std::getline(file, line)) {
        std::stringstream ss(line);
        std::string prefix;
        ss >> prefix;

        if (prefix == "newmtl") {
            ss >> currentMtlName;
            materials[currentMtlName].name = currentMtlName;
        } else if (prefix == "Ka") {
            glm::vec3 color;
            ss >> color.x >> color.y >> color.z;
            materials[currentMtlName].ambientColor = color;
        } else if (prefix == "Kd") {
            glm::vec3 color;
            ss >> color.x >> color.y >> color.z;
            materials[currentMtlName].diffuseColor = color;
        } else if (prefix == "Ks") {
            glm::vec3 color;
            ss >> color.x >> color.y >> color.z;
            materials[currentMtlName].specularColor = color;
        } else if (prefix == "Ns") {
            float shininess;
            ss >> shininess;
            materials[currentMtlName].shininess = shininess;
        } else if (prefix == "map_Kd") {
            std::string token;
```

```
            while (ss >> token) {
                if (token == "-s") {
                    std::string uStr, vStr, wStr;
                    ss >> uStr >> vStr >> wStr;
                    try {
                        float u = std::stof(uStr);
                        float v = std::stof(vStr);
                        materials[currentMtlName].textureScale = glm::vec2(u, v);
                    }
                    catch (...) {}
                } else if (token == "-o") {
                    std::string temp;
                    ss >> temp >> temp >> temp; // Skip u v w
                } else if (token == "-mm") {
                    std::string temp;
                    ss >> temp >> temp; // Skip base gain
                } else if (token == "-bm") {
                    std::string temp;
                    ss >> temp; // Skip mult
                } else {
                    materials[currentMtlName].diffuseMap = token;
                    break;
                }
            }
        }
    }
}

void Model::loadModel(std::string const &path) {
    std::ifstream file(path);
    if (!file.is_open()) {
        std::cout << "Failed to open OBJ file: " << path << std::endl;
        return;
    }

    directory = path.substr(0, path.find_last_of('/'));

    std::vector<glm::vec3> temp_positions;
    std::vector<glm::vec2> temp_texCoords;
    std::vector<glm::vec3> temp_normals;

    // Per-mesh data
    std::vector<Vertex> vertices;
    std::vector<unsigned int> indices;
    std::vector<Texture> textures;

    // Map unique vertex string "v/vt/vn" to index
    std::map<std::string, unsigned int> uniqueVertices;

    std::string line;
    std::string currentMaterial = "";
```

```
117
118     // Helper to flush current mesh
119     auto flushMesh = [&]() {
120         if (!vertices.empty()) {
121             glm::vec3 ambient = glm::vec3(1.0f);
122             glm::vec3 diffuse = glm::vec3(1.0f);
123             glm::vec3 specular = glm::vec3(0.5f);
124             float shininess = 32.0f;
125
126             if (materials.find(currentMaterial) != materials.end()) {
127                 ambient = materials[currentMaterial].ambientColor;
128                 diffuse = materials[currentMaterial].diffuseColor;
129                 specular = materials[currentMaterial].specularColor;
130                 shininess = materials[currentMaterial].shininess;
131
132                 glm::vec2 scale = materials[currentMaterial].textureScale;
133                 if (scale.x != 1.0f || scale.y != 1.0f) {
134                     for (auto &v : vertices) {
135                         v.TexCoords.x *= scale.x;
136                         v.TexCoords.y *= scale.y;
137                     }
138                 }
139             }
140             meshes.push_back(Mesh(vertices, indices, textures, ambient, diffuse, specular,
                 shininess));
141             vertices.clear();
142             indices.clear();
143             textures.clear();
144             uniqueVertices.clear();
145         }
146         };
147
148     while (std::getline(file, line)) {
149         std::stringstream ss(line);
150         std::string prefix;
151         ss >> prefix;
152
153         if (prefix == "mtllib") {
154             std::string mtlFile;
155             std::getline(ss, mtlFile);
156             // Trim leading whitespace
157             size_t first = mtlFile.find_first_not_of(' ');
158             if (std::string::npos != first) {
159                 mtlFile = mtlFile.substr(first);
160             }
161             loadMTL(directory + "/" + mtlFile);
162         } else if (prefix == "usemtl") {
163             flushMesh(); // Start new mesh on material change
164             ss >> currentMaterial;
165
166             // Load textures for this material
```

```
            if (materials.find(currentMaterial) != materials.end()) {
                // Diffuse map
                std::string diffPath = materials[currentMaterial].diffuseMap;
                if (!diffPath.empty()) {
                    bool skip = false;
                    for (unsigned int j = 0; j < textures_loaded.size(); j++) {
                        if (std::strcmp(textures_loaded[j].path.data(), diffPath.c_str()) == 0) {
                            textures.push_back(textures_loaded[j]);
                            skip = true;
                            break;
                        }
                    }
                    if (!skip) {
                        Texture texture(diffPath.c_str(), directory);
                        texture.type = "texture_diffuse";
                        textures.push_back(texture);
                        textures_loaded.push_back(texture);
                    }
                }
            }
        } else if (prefix == "v") {
            glm::vec3 pos;
            ss >> pos.x >> pos.y >> pos.z;
            temp_positions.push_back(pos);

            // Update bounds
            if (pos.x < minBound.x) minBound.x = pos.x;
            if (pos.y < minBound.y) minBound.y = pos.y;
            if (pos.z < minBound.z) minBound.z = pos.z;
            if (pos.x > maxBound.x) maxBound.x = pos.x;
            if (pos.y > maxBound.y) maxBound.y = pos.y;
            if (pos.z > maxBound.z) maxBound.z = pos.z;
        } else if (prefix == "vt") {
            glm::vec2 tex;
            ss >> tex.x >> tex.y;
            temp_texCoords.push_back(tex);
        } else if (prefix == "vn") {
            glm::vec3 norm;
            ss >> norm.x >> norm.y >> norm.z;
            temp_normals.push_back(norm);
        } else if (prefix == "f") {
            std::string vertexStr;
            std::vector<std::string> faceVertices;
            while (ss >> vertexStr) {
                faceVertices.push_back(vertexStr);
            }

            // Triangulate (fan)
            for (size_t i = 1; i < faceVertices.size() - 1; ++i) {
                std::string v[3] = { faceVertices[0], faceVertices[i], faceVertices[i + 1] };

```

```
            for (int j = 0; j < 3; ++j) {
                if (uniqueVertices.count(v[j]) == 0) {
                    uniqueVertices[v[j]] = static_cast<unsigned int>(vertices.size());

                    Vertex vertex;
                    std::stringstream vss(v[j]);
                    std::string segment;
                    std::vector<std::string> indicesStr;

                    while (std::getline(vss, segment, '/')) {
                        indicesStr.push_back(segment);
                    }

                    // Position
                    int posIdx = std::stoi(indicesStr[0]) - 1;
                    vertex.Position = temp_positions[posIdx];

                    // TexCoord
                    if (indicesStr.size() > 1 && !indicesStr[1].empty()) {
                        int texIdx = std::stoi(indicesStr[1]) - 1;
                        vertex.TexCoords = temp_texCoords[texIdx];
                    } else {
                        vertex.TexCoords = glm::vec2(0.0f, 0.0f);
                    }

                    // Normal
                    if (indicesStr.size() > 2 && !indicesStr[2].empty()) {
                        int normIdx = std::stoi(indicesStr[2]) - 1;
                        vertex.Normal = temp_normals[normIdx];
                    } else {
                        vertex.Normal = glm::vec3(0.0f, 0.0f, 0.0f);
                    }

                    vertices.push_back(vertex);
                }
                indices.push_back(uniqueVertices[v[j]]);
            }
        }
    }
    flushMesh();
}
```

A.7 相机与对象基类

Listing 30: src/Camera.cpp

```
#include "Camera.h"

Camera::Camera(glm::vec3 position, glm::vec3 up, float yaw, float pitch) : Front(glm::vec3(0.0f,
    0.0f, -1.0f)), MovementSpeed(SPEED), MouseSensitivity(SENSITIVITY), Zoom(ZOOM) {
```

```
    Position = position;
    WorldUp = up;
    Yaw = yaw;
    Pitch = pitch;
    Roll = 0.0f;
    updateCameraVectors();
}

Camera::Camera(float posX, float posY, float posZ, float upX, float upY, float upZ, float yaw,
     float pitch) : Front(glm::vec3(0.0f, 0.0f, -1.0f)), MovementSpeed(SPEED), MouseSensitivity(
     SENSITIVITY), Zoom(ZOOM) {
    Position = glm::vec3(posX, posY, posZ);
    WorldUp = glm::vec3(upX, upY, upZ);
    Yaw = yaw;
    Pitch = pitch;
    Roll = 0.0f;
    updateCameraVectors();
}

glm::mat4 Camera::GetViewMatrix() const {
    return glm::lookAt(Position, Position + Front, Up);
}

void Camera::ProcessKeyboard(Camera_Movement direction, float deltaTime) {
    float velocity = MovementSpeed * deltaTime;
    if (direction == FORWARD)
        Position += Front * velocity;
    if (direction == BACKWARD)
        Position -= Front * velocity;
    if (direction == LEFT)
        Position -= Right * velocity;
    if (direction == RIGHT)
        Position += Right * velocity;
    if (direction == UP)
        Position += WorldUp * velocity;
    if (direction == DOWN)
        Position -= WorldUp * velocity;
}

void Camera::ProcessMouseMovement(float xoffset, float yoffset, GLboolean constrainPitch) {
    xoffset *= MouseSensitivity;
    yoffset *= MouseSensitivity;

    Yaw += xoffset;
    Pitch += yoffset;

    if (constrainPitch) {
        if (Pitch > 89.0f)
            Pitch = 89.0f;
        if (Pitch < -89.0f)
            Pitch = -89.0f;
```

```
    }

    updateCameraVectors();
}

void Camera::ProcessMouseScroll(float yoffset) {
    Zoom -= (float)yoffset;
    if (Zoom < 1.0f)
        Zoom = 1.0f;
    if (Zoom > 45.0f)
        Zoom = 45.0f;
}

void Camera::updateCameraVectors() {
    // calculate the new Front vector
    glm::vec3 front;
    front.x = cos(glm::radians(Yaw)) * cos(glm::radians(Pitch));
    front.y = sin(glm::radians(Pitch));
    front.z = sin(glm::radians(Yaw)) * cos(glm::radians(Pitch));
    Front = glm::normalize(front);
    // calculate the new Right and Up vectors
    glm::vec3 right = glm::normalize(glm::cross(Front, WorldUp));
    glm::vec3 up = glm::normalize(glm::cross(right, Front));
    // apply roll rotation around Front axis
    glm::mat4 rollRot = glm::rotate(glm::mat4(1.0f), glm::radians(Roll), Front);
    Right = glm::normalize(glm::vec3(rollRot * glm::vec4(right, 0.0f)));
    Up = glm::normalize(glm::vec3(rollRot * glm::vec4(up, 0.0f)));
}
```

Listing 31: src/GameObject.cpp

```
#include "GameObject.h"
#include "PhysicsSystem.h"

GameObject::GameObject(Model *model, glm::vec3 pos, glm::vec3 rot, glm::vec3 scale)
    : model(model), position(pos), rotation(rot), scale(scale) {
}

GameObject::~GameObject() = default;

void GameObject::setScaleToSizeX(float sizeX) {
    if (!model) return;
    float originalWidth = model->maxBound.x - model->minBound.x;
    if (originalWidth < 1e-6f) return;
    float factor = sizeX / originalWidth;
    scale = glm::vec3(factor);
}

void GameObject::setCollisionMask(uint32_t mask) {
    if (rigidBody) {
        rigidBody->collisionMask = mask;
    }
```

```
22 }
23
24 void GameObject::setCollisionEnabled(bool enabled) {
25     if (rigidBody) {
26         rigidBody->isCollisionEnabled = enabled;
27     }
28 }
```