

浙江大学



计算机图形学大作业

项 目 ZeldaDemo

学生姓名 莫非, 王晓宇, 潘潇然

提交日期 2024 年 12 月 27 日

目录

1 项目概述	3
1.1 项目初衷	3
1.2 从《塞尔达传说：荒野之息》中获取的内容	3
2 地形构建	5
2.1 从高度图构建地形的流程	5
2.2 获取高度和法向量	5
3 地形纹理	7
4 水的实现	8
4.1 水的形状	8
4.2 波动与光照	8
4.3 动态反射与折射	9
5 阴影映射	10
6 人物移动与视角	10
6.1 Player 类基本介绍	10
6.2 物理引擎	11
6.3 视角移动	11
7 人物动作与交互	12
7.1 跳跃	12
7.2 滑翔	13
7.3 攀爬	13
7.4 游泳	13
7.5 动画	14
7.6 遥控炸弹	14
7.7 剑气（攻击）	16
7.8 盾牌（防御）	16
7.9 交互	17
8 场景物体导入与创建	17
8.1 Obj 模型导入	17
8.1.1 Assimp 的工作流程	17
8.1.2 stb_image 的工作流程	18
8.1.3 整合 Assimp 和 stb_image	18
8.1.4 自定义的模型着色器	18
8.2 实际模型展示	18
8.3 碰撞检测	21
9 Appendix	22

1 项目概述

1.1 项目初衷

希望能不依赖任何引擎、纯用 OpenGL 模拟《塞尔达传说：荒野之息》的游戏内容，但这太过困难，因此做了许多简化：

1. 场景只实现游戏中的新手村——初始台地，相当于从开放世界变为一个小箱庭
2. 人物的动画实现，如骨骼动画等过于复杂，因此简化为一个长方体《立方体传说：荒野窒息》
3. 人物的各种动作、能力也都进行一定的简化、抽象、魔改

在项目中，对 OpenGL 的一些操作进行封装，包括但不限于相机、三角面片与模型、帧缓冲、2D 纹理、立方体贴图纹理、纹理附件、uniform 变量块、VAO, VBO, EBO 等，使得整个项目的许多代码能够重用。但即使如此，项目也已经变得非常庞大（内容很多），纯自己写的（不包括外部库函数）有效代码行数已达到 7,651（由 VS Code Counter 插件统计）。

我们会在这个实验报告中对一些技术细节进行剖析，但为了更直观地了解，可以查看我们附件中的 demo。

[github 项目地址](#)

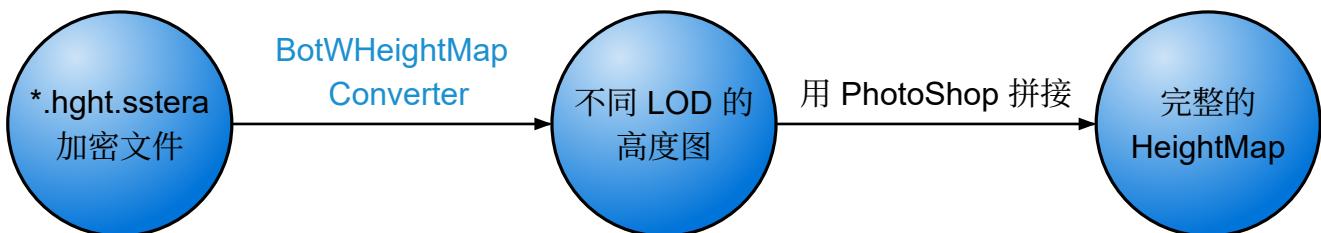
1.2 从《塞尔达传说：荒野之息》中获取的内容

这一部分跟图形学、OpenGL 关系不大，但还是想讲一讲，因为花了很多时间去做转换，最终摸索出一个 pipeline。

众所周知，由于一些原因，switch 上的游戏被破解得非常快。从一些“灰色地带”可以获取到一些解包出的破解数据，可以从中获取一些内容。这些数据是用任天堂自定义的格式加密的，但可以用一些特定工具解密，并最终转化为我们需要的格式。

但是这里叠个甲，坚决反对任何仍在生命周期内游戏机上游戏的破解，这里是真 · 仅供学习交流使用。

- 高度图的获取
 - 利用工具获取不同 LOD 级别（0 ~ 8 级）的高度图。高级 LOD 的高度图并不完整，需用低级的补全，使用 Photoshop 手动裁剪拼接。最终生成分辨率为 4096×4096 的灰度图
 - 这个分辨率对于地形建模而言开销过大，后续是降采样为 1024×1024 来使用



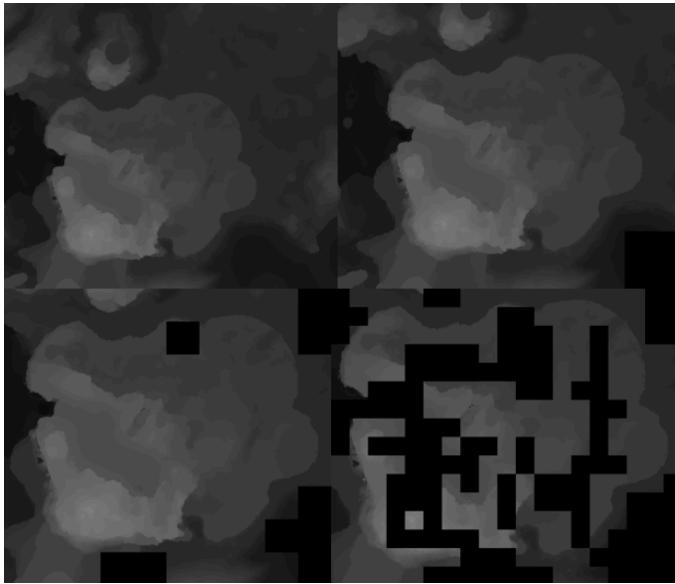


图 1-1 LOD of 5,6,7,8

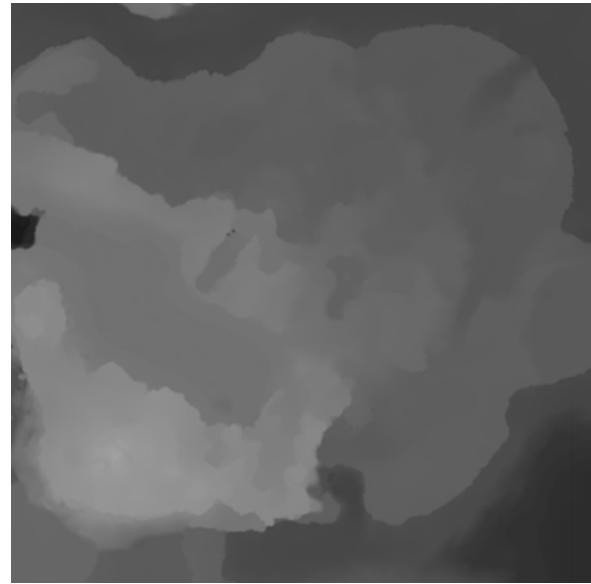


图 1-2 完整的 HeightMap

- 模型文件的获取
 - 整个转换过程参考学习了很多工具
 - 但其实很多文件在转换的过程中会损坏，且并非所有模型都能找到（地形的材质也未能找到）
 - 并且许多模型在 3ds max 里打开看起来很美好，但转换成 obj 后用自己写的着色器在游戏中加载后却是另一副样子
 - 因此最后也只采用了部分简单的模型（比如树、箱子）

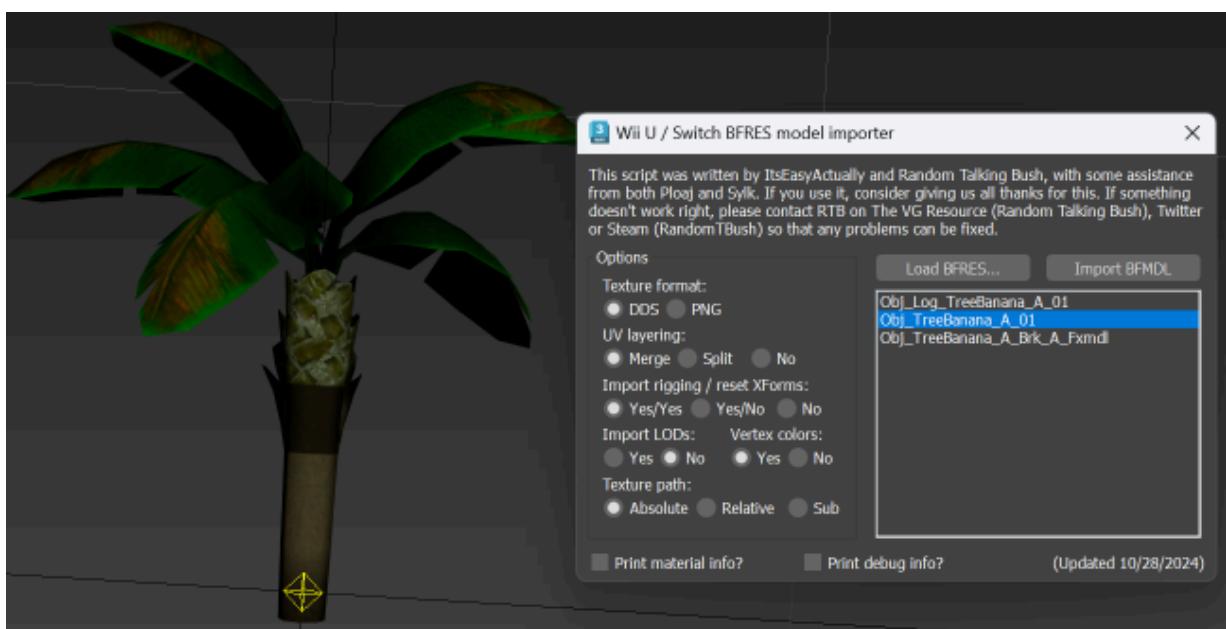
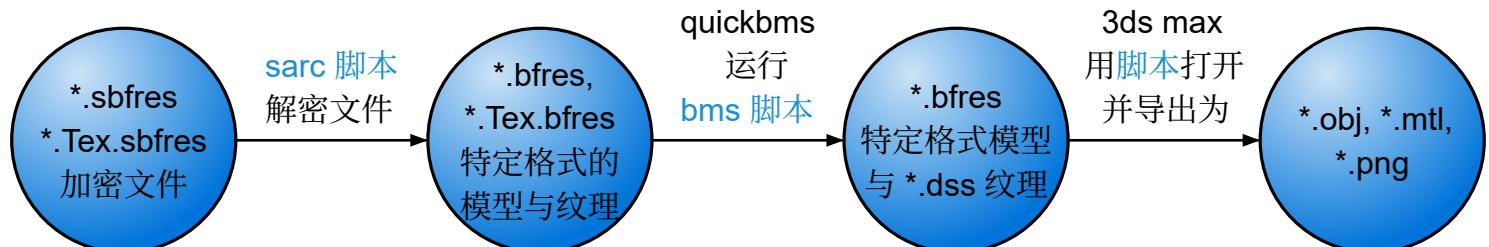


图 1-3 用 3ds max 运行脚本导入模型

2 地形构建

我们使用 tile-based terrain 构建我们的山脉（地形）。整体为一个 grid，包含许多 tile，每个 tile 中包含两个三角形，三角形的每个顶点具有： `position`, `texCoords`, `normal`, `indice`, `tangents`, `bitangents`，我们需要从高度图中一一计算出这些值

2.1 从高度图构建地形的流程

```
1 void generateMesh(unsigned char* heightMap, const int sampleNum,
2                     const int smooth_times);
```

G Cpp

1. 根据分辨率采样，得到每点高度
 - 根据世界坐标中点的 (x, z) 值，算出它在高度图中对应哪个像素。以该像素的灰度值作为其高度，从而得到完整的坐标 (x, y, z) 。其中坐标尺度的转换不赘述
 - 这样采样出来的点会因为分辨率受限导致极度高低不平、坑坑洼洼，因此对每个像素，同时采样其周围一圈像素，以一定权重进行平均
 - 通过权重在平滑一般地形的同时避免悬崖也被平滑
 - 另外，实践发现，使用较低采样数量进行多次平滑，比单次采样较多数的效果好得多
2. 两两成边，叉乘得到面的法向；平均得到点的法向
 - 每个 tile 由四个顶点构成，分为上三角和下三角
 - 点与点之间两两相连构成边，两个边叉乘得到面的法向
 - 但在我们的实现里，面的法向并不重要，于是可以把每个顶点所参与构成的那些三角形的法向做平均，得到该点的法向量
 - 即这里分成两阶段处理，先计算面的法向，再得到点的法向，另外对点的法向量也进行上述平滑操作
3. 计算索引数组和纹理坐标数组
 - 索引数组没什么好说的，使每个 tile 只需要 4 个顶点而不是 6 个，减少 $\frac{1}{3}$ 的开销
 - 纹理坐标根据在 grid 中的位置赋予，乘上一个尺度因子
4. 计算每点的切线和副切线
 - 切线和副切线是来自采样法向贴图时计算 TBN 矩阵的要求
 - 在有每个顶点位置后可以通过 [learnOpenGL](#) 网站的算法自然计算出每点的切线和副切线

2.2 获取高度和法向量

- `getHeight` 和 `getNormal` 函数

```
1 float getHeight(const float& worldX, const float& worldZ) const;
2 glm::vec3 getNormal(const float& worldX, const float& worldZ) const;
```

G Cpp

- 另外，我们通过重心坐标插值实现了得到世界内任意点 (x, z) 在 terrain 上的高度和法向量
- 这个高度和法向在实现后续人物的站位和朝向时很有用
- 具体做法，就是实现一个 `barycentricCoord` 函数，首先计算出 (x, z) 落在哪个三角形内，然后根据点在三角形内的位置，平滑地进行插值计算
- 重心坐标利用面积进行计算，进一步转化为叉乘

```
1 glm::vec3 barycentricCoord(const glm::vec2 p1, const glm::vec2 p2,
```

G Cpp

```

2           const glm::vec2 p3, const glm::vec2 pos) const
3   {
4       glm::vec3 u = glm::cross( // i, j, k
5           glm::vec3(p3.x - p1.x, p2.x - p1.x, p1.x -
6           pos.x),
7           glm::vec3(p3.y - p1.y, p2.y - p1.y, p1.y -
8           pos.y));
9       if (std::abs(u.z) < 1)
10           Warn("Barycentric coordinate is degenerate, u.z = " +
11           std::to_string(u.z));
12       return glm::vec3(1.f - (u.x + u.y) / u.z, u.y / u.z, u.x / u.z);
13   }

```

我们最终的 terrain 使用了多达六百万个顶点，从而实现了一个高度、起伏变化如此之大的地形。并且跟《荒野之息》是非常相似的（毕竟得到了它的高度图）。

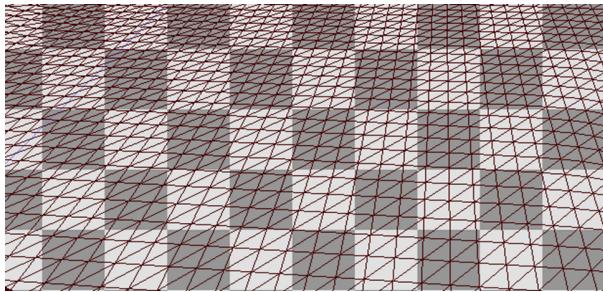


图 2-1 近处细节

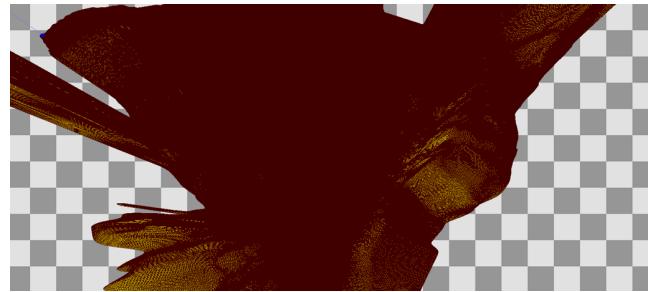


图 2-2 远景查看（混成一团）

在地形构建初期，还没有实现纹理（直接使用法向量作为颜色）也还未实现人物并把相机绑定上去，那时拍摄的俯瞰地形图：

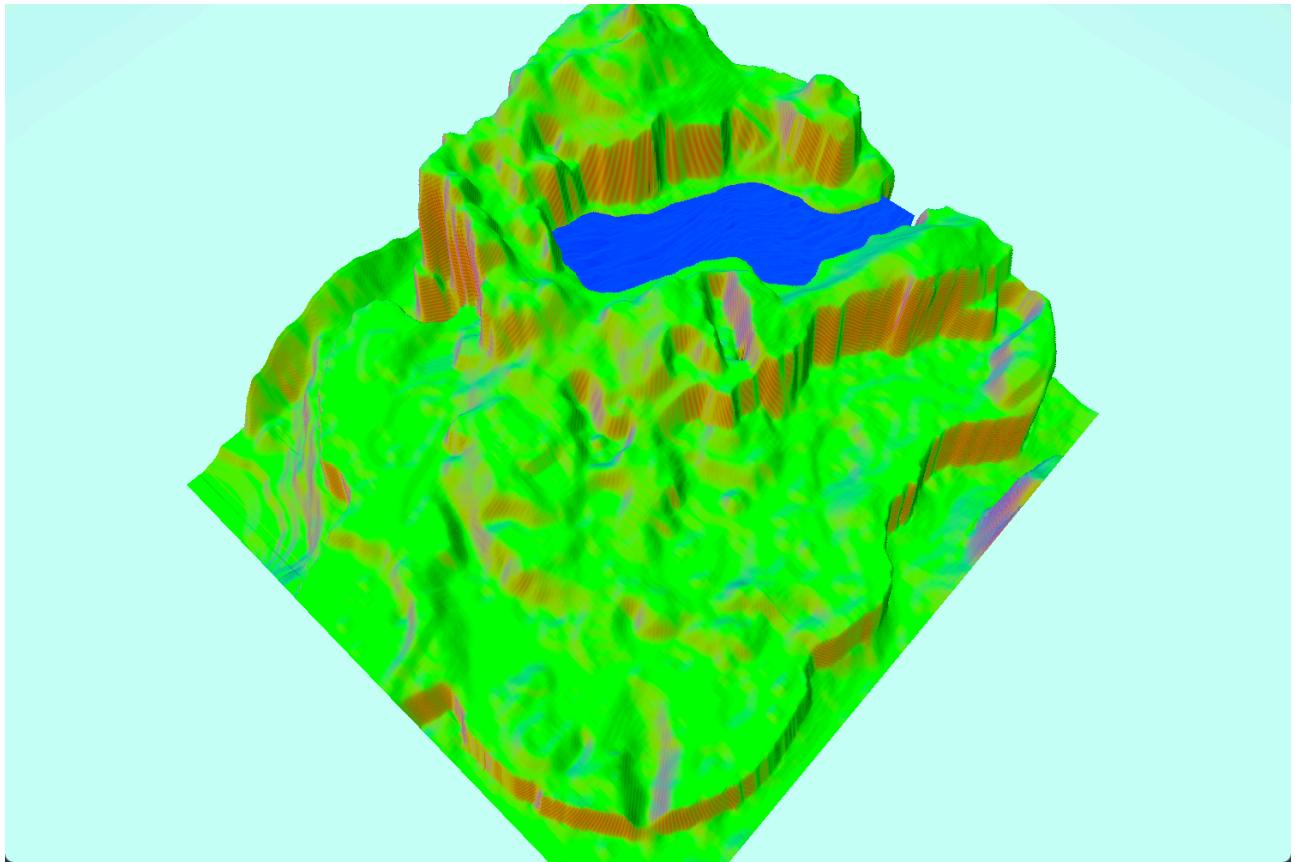


图 2-3 （初期）地形俯瞰

3 地形纹理

为一个高度变化、起伏巨大的山脉赋予纹理十分具有挑战，显然需要用多个纹理（草地、岩石、雪地……）才能较好地表达这一场景。进一步，如何平滑地过渡也是一个挑战，其核心在于如何控制材质之间的混合权重。

✗ 第一个想法

利用一张额外的 jpg 纹理图片（或 png 格式，可以有四个通道），并不作为采样颜色的来源，而是利用每个 `texel` 值的三个通道，归一化后作为三张纹理的分量。

但难点在于，没有数据、时间和耐心自己画这么一个庞大的系统（即使实现一个材质编辑器，想要画得合理，依旧十分困难）。

✓ 另一个想法

因此最后采用程序化生成的方式，自动插值融合各个纹理。我搜集了四张纹理，分别是：草地、雪草相间、雪地、岩石。

具体的逻辑是，对每一点我们有高度值和法向量：当高度在 210 以下，使用草地纹理；在 210 ~ 240 之间在草地和雪草相间纹理中插值；240 ~ 270 之间在雪草相间和雪地纹理之间插值。这样得到基于高度的纹理值后，根据法向量跟竖直方向的夹角，再进行跟岩石纹理的插值；如果夹角过大就直接使用岩石纹理。



图 3-1 依据高度混合纹理

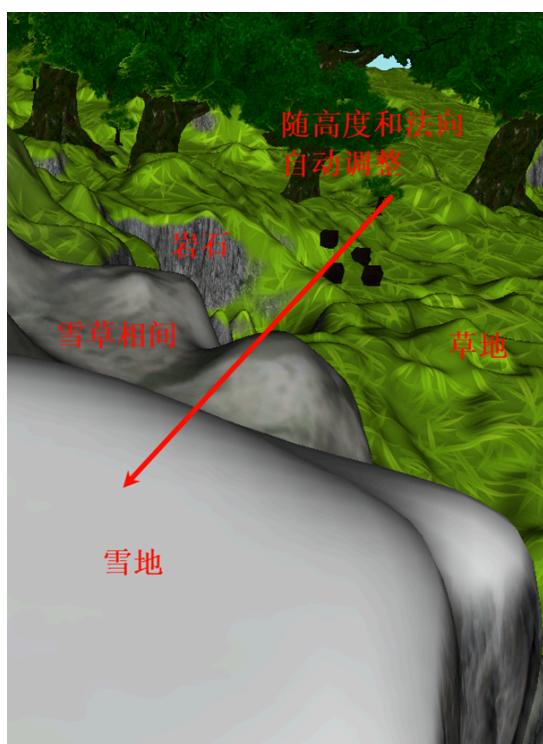


图 3-2 纹理变化

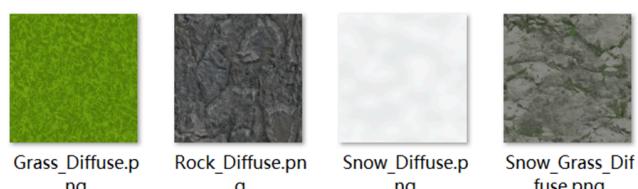


图 3-3 采用的 diffuse 纹理

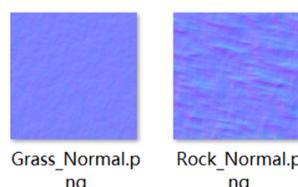


图 3-4 采用的 normal 纹理

- 此外我们还利用法向纹理实现了法向贴图，融合思路类似，就不赘述
 - 法向贴图需要计算每点的 TBN 矩阵，由之前 terrain 的切线和副切线可以计算得到
 - 利用法向贴图可以计算光照时得到更逼真的效果
- 地形使用了 Blinn-Phong 光照模型与阴影映射（见 [小节 5](#)）

4 水的实现

4.1 水的形状

每一片水（河流或湖泊）都有一个自己的高度，以及一系列具有 (x, z) 坐标与纹理坐标的顶点（只实现了平面水，没有做瀑布之类）。

- 从地图中用少量点勾勒出大致形状，使用 `GL_TRIANGLE_FAN` 绘制
 - 首先从游戏中有颜色的地图中用 `photoshop` 的魔棒工具取出水的位置
 - 然后编写 `python` 脚本，实现在图上点击并生成点的 (x, z) 位置和纹理坐标 (u, v) ，详见 `utils/river_generator/process_point.py`

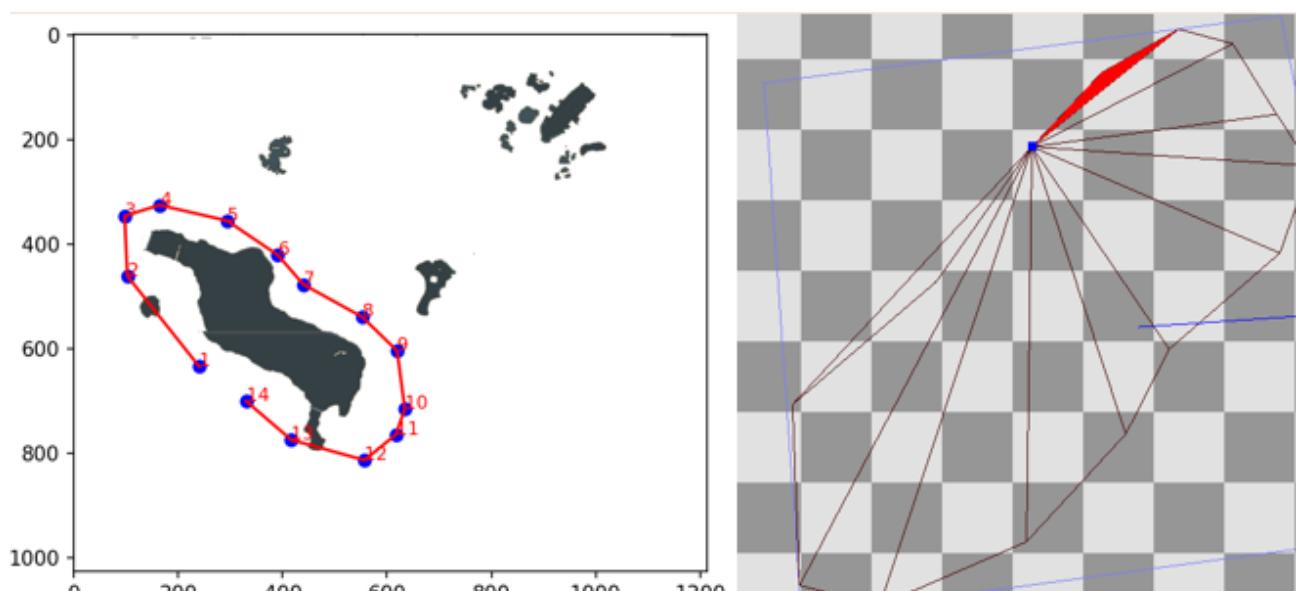


图 4-1 水的轮廓

4.2 波动与光照

- 波动效果利用扰动纹理与法向纹理实现

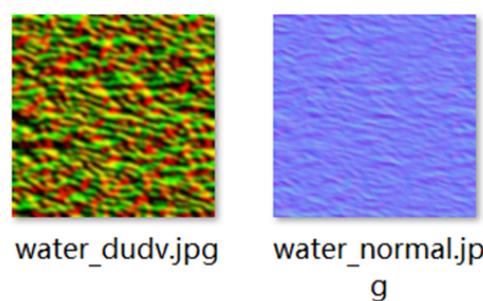


图 4-2 扰动纹理与法向纹理

- 利用每个点的纹理坐标 (u, v) ，从 UV 偏移纹理中采样得到偏移值 (du, dv) ，加上随时间的扰动得到新的纹理坐标 (u', v')
- 新的纹理坐标在法向纹理中采样，从而每点在每一时刻得到的法向量都不同

- 进一步，在 Blinn-Phong 光照模型中算出的 diffuse 和 specular 就不同，形成波纹效果

4.3 动态反射与折射

动态反射与折射的思路跟阴影映射的方法是类似的，都用到了帧缓冲的概念。通过渲染到对应帧缓冲，从而可以将其视为一个纹理附件，在真正渲染到默认帧缓冲时能够进行一些后处理。

- 反射

- 渲染水上 terrain、场景其它物件、天空盒，将它们绘制到反射帧缓冲中
- 开启裁剪平面，只渲染水上物体，降低开销
- 反射使用镜面相机（当前相机位置跟水面镜面对称位置），对 Up 向量也要做更改

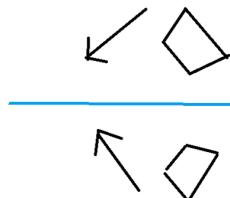


图 4-3 对称相机

- 折射：

- 只渲染水下 terrain 到折射帧缓冲中，没有其它物体需要绘制（没有做水中物体）
- 开启裁剪平面，只渲染水下物体，降低开销
- 从而，在正常渲染时，从折射、反射的帧缓冲所绑定的纹理附件上可以采样出对应点的颜色，反射、折射、蓝色三者进行混合

综合考虑以上形状、波动、光照、反射折射（以及后面的阴影）后，实现下面这样逼真的水面效果：

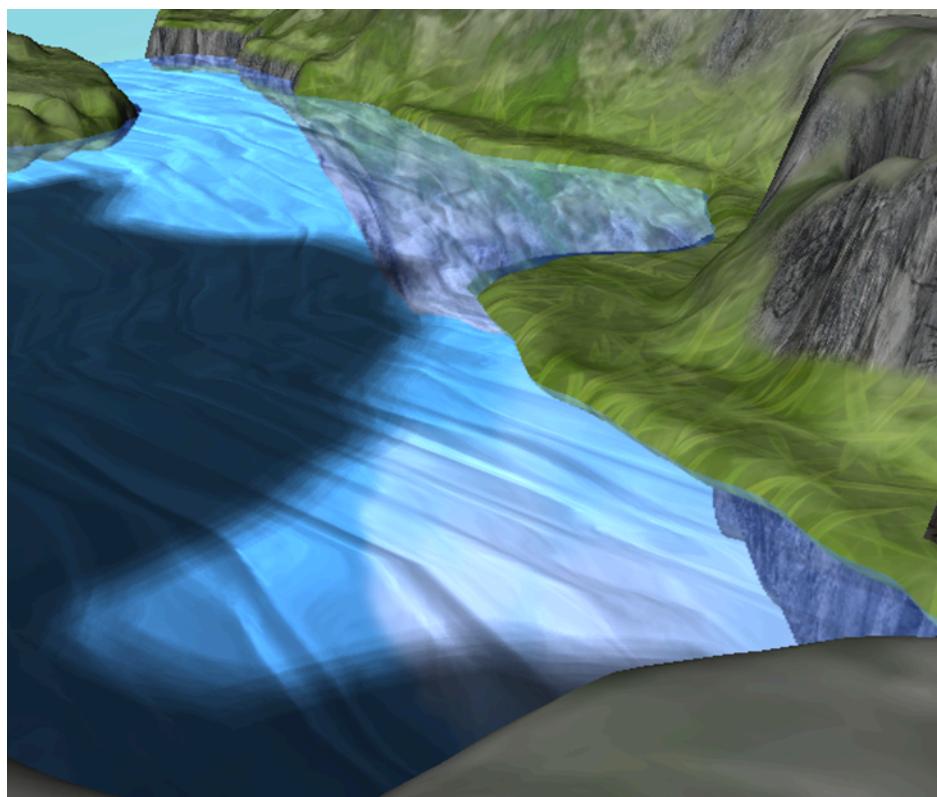


图 4-4 水面效果

5 阴影映射

动态阴影映射，跟水的反射折射非常类似，也是渲染到一个帧缓冲中，只不过不需要颜色附件，只需要一个深度附件用来存储绘制时 z-buffer 的深度值。

- 具体流程
 - 以光源位置为相机，渲染 terrain 和各种物件，结果存储到阴影帧缓冲
 - 正常渲染到默认帧缓冲时，可以从中获取物体到光源的遮挡距离，判断是否在阴影中
 - 对 Blinn-Phong 光照模型的 diffuse, specular 乘以阴影（ambient 不乘以阴影因子避免全黑）
- 高级阴影
 - 使用阴影偏移(shadow bias)解决阴影失真
 - 软阴：
 - 使用 PCF 方法采样每个点在深度图中附近的一些点，判断是否在阴影中并平均阴影因子
 - 用 poissonDisk 生成 32 个采样点，提高采样效率



图 5-1 Poisson Disk



图 5-2 soft shadow

6 人物移动与视角

6.1 Player 类基本介绍

- Player 类包括了 Player 的长度、位置、方向、速度、颜色、状态、盾牌和剑气的参数等基本信息。
- 其中 Player 的状态包括：
 - IDLE_LAND (在陆地静止)
 - WALKING_LAND
 - RUNNING_LAND
 - IDLE_WATER (在水面静止)
 - SWIMMING_WATER
 - FAST_SWIMMING_WATER
 - CLIMBING
 - JUMPING
- 为了在渲染中减少开销，在初始化时利用 BoxGeometry 类获取 vertex 和 index，并设置绑定 VAO, VBO, EBO。

- 同时，我们在着色器调用前计算好 projection 矩阵和 view 矩阵的乘积，以及 normal 矩阵，以减少 GPU 的计算。

6.2 物理引擎

- `Update()` 函数计算长方体在坐标 (x, z) 下的 y 坐标和法向量（朝向）。
 - 首先根据长方体中心位置从地形采样高度和法向量，对长方体进行旋转得到初步位置。
 - 之后基于此从地形中采样长方体四个角的高度和法向量，取平均更新物体法向量。
 - 最后保证长方体底部不会陷进地形，且能随地形角度倾斜。
- `ProcessMoveInput()` 函数根据键盘输入更新长方体位置
 - 根据状态和方向更新当前帧长方体位移量
 - 根据原先位置坡度移动对应位移量
 - 调用 `Update()` 调整长方体的位置以符合物理逻辑

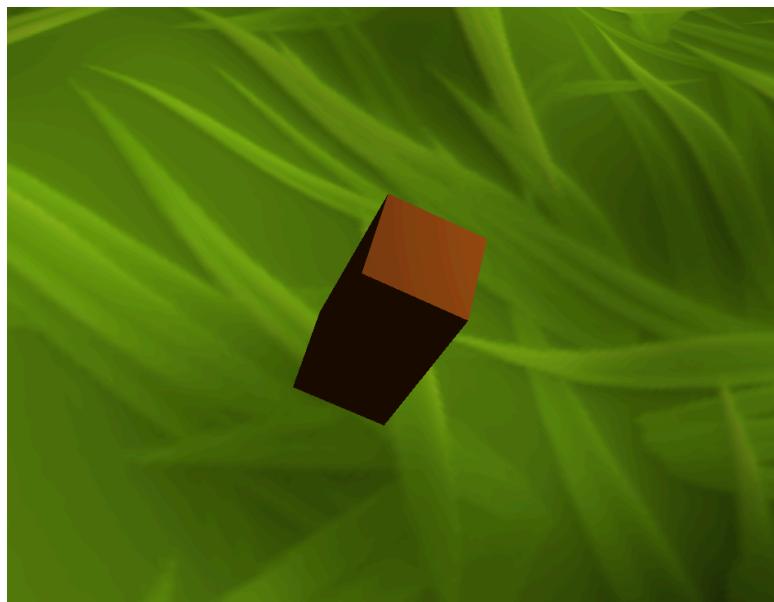


图 6-1 站立实例

6.3 视角移动

- 使用第三人称越肩视角。固定物体在画面正中心，相机在半球体上移动，挖去球顶部分和一部分球底以避免突变
 - 垂直角度限制在 20° 到 80°
- 物体和摄像机构成的线段划分成 50 份，检测相机是否被物体遮挡。如被遮挡则沿线段移动、缩近
 - 相机位置低于对应点地形高度，则说明被遮挡

```

1 for(i = 0; i < maxSteps; ++i) {
2     currentDistance += step;
3     glm::vec3 samplePos = playerPos + direction * currentDistance;
4     float terrainHeight = terrain->getHeight(samplePos.x, samplePos.z);
5     if(samplePos.y < terrainHeight + 1.0f) { // Prevent camera from
6         finalPos = playerPos + direction * (currentDistance - step);
7         break;
8     }

```

Cpp

9 }

- 当摄像机和物体距离过近时，降低物体透明度
 - 注意启用透明度后要在初始化窗口时加入对应设置

```
1 glEnable(GL_BLEND);
2 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

CPP

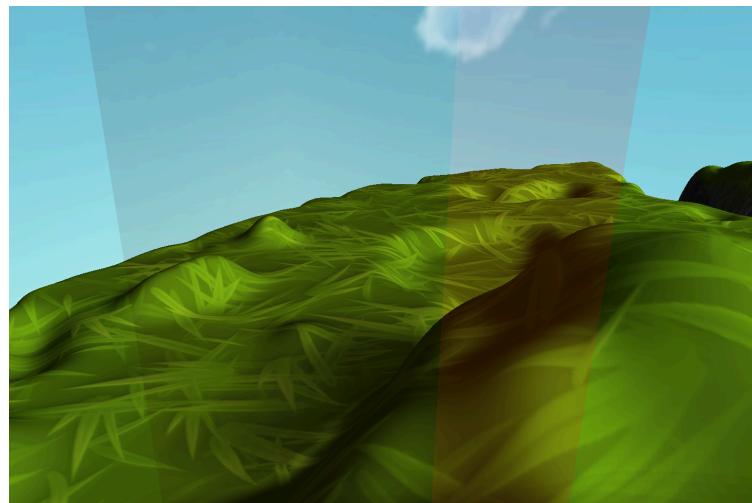


图 6-2 透明效果

7 人物动作与交互

我们做了跳跃、滑翔、攀爬、游泳等基础动作，并为它们赋予简单的切换动画。但在这样复杂的地形中，难免会有经常性的闪烁与突兀现象。此外我们还实现了一些进阶动作比如炸弹、攻击、防御，能够跟场景进行交互。

7.1 跳跃

- 当玩家按下空格键会进行跳跃，长方体状态切换为 JUMPING
- 支持向任意方向进行跳跃和原地跳跃
 - 通过长方体的 direction 向量和上向量进行运算可以得到指向四个方向的向量
 - 无方向输入时，水平跳跃分量设为 0
- 每帧按物理逻辑更新长方体坐标
- 若 y 坐标低于地形高度，跳跃状态结束，调用 Update() 函数更新姿态

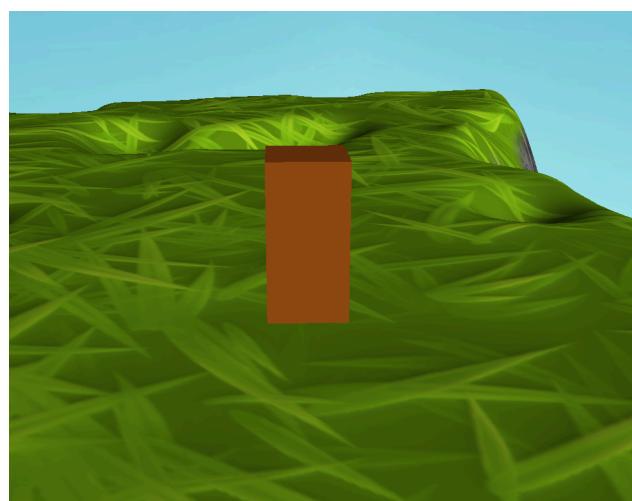


图 7-1 跳跃

7.2 滑翔

- 实现逻辑和跳跃是一模一样的，我们只需要在滑翔状态减慢竖直下落加速度即可
- 其次我们在滑翔的时候方向是随视角移动的，我们只需要改变飞翔起来的 `Direction` 变量即可

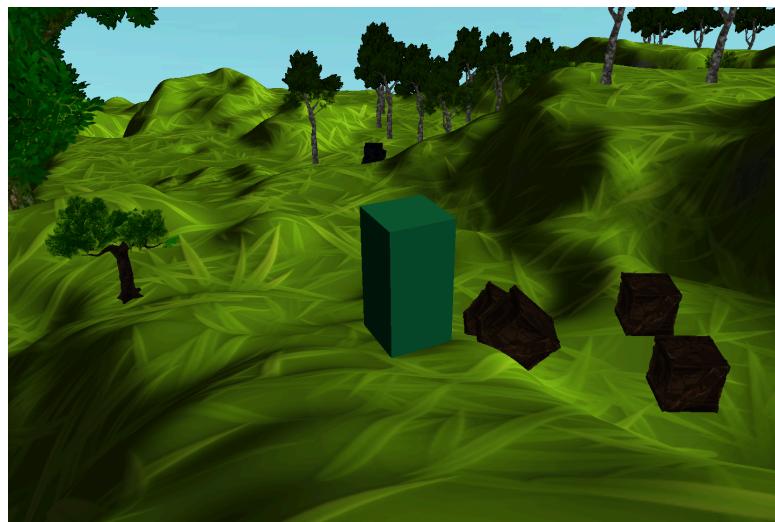


图 7-2 滑翔

7.3 攀爬

- 采样(x,z)处地形法向量，与物体右向量叉乘得到物体上向量
- 根据上向量计算物体攀爬角度，直接将模型进行旋转操作，即可在视觉上发生了攀爬的动作

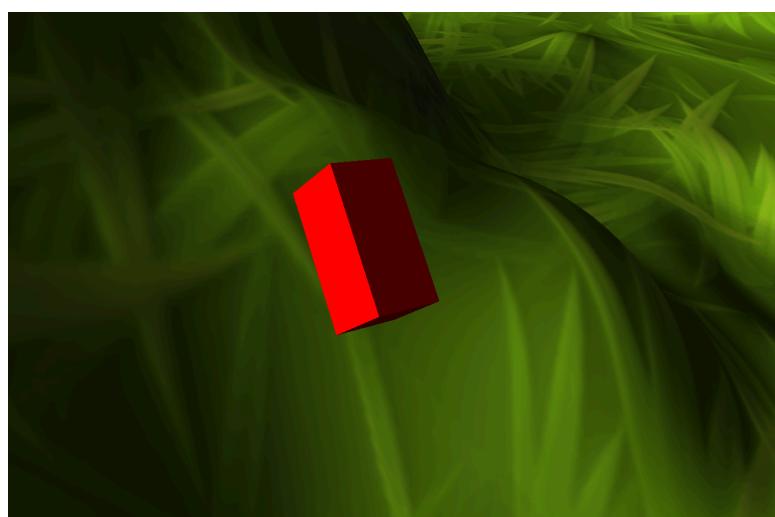


图 7-3 攀爬

7.4 游泳

- 当长方体中心高度低于水面高度时，长方体状态切换为 `IDLE_WATER` , `SWIMMING_WATER` , `FAST_SWIMMING_WATER` 中的一种
- 进入水中后，长方体会呈现倒下的姿态，上向量始终为 $(0, 1, 0)$, 同时颜色会变化成橙色

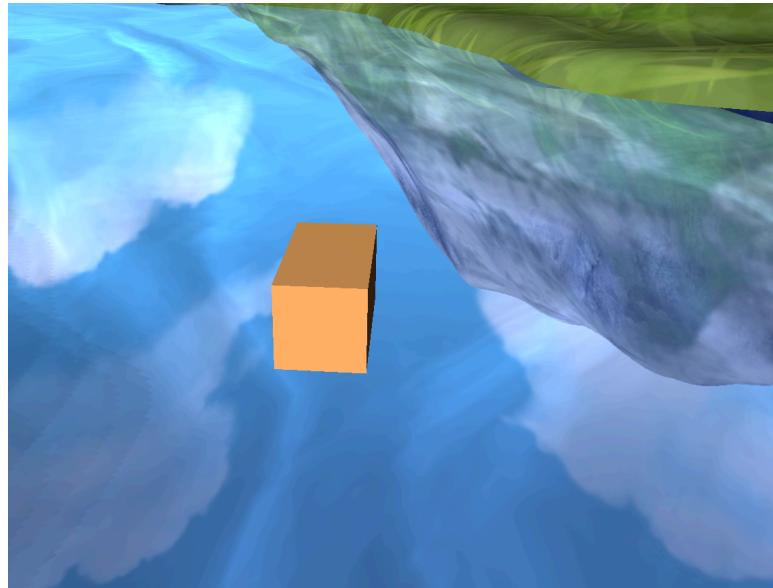


图 7-4 游泳

7.5 动画

- 我们这里的主函数中产生了 `DeltaTime` 变量，我们通过这个变量的值累加，可以得到近似于硬件语言中的时钟信号，我们产生了 1s 的时钟周期
- 利用时钟周期，我们在每次需要完成动画的时候，将结束帧与开始帧之间的变换矩阵乘以一个缩放因子，直接作用到我们的初态中，缩放因子随着时钟接近于 1，最后到 1 时完成了动画的显示

7.6 遥控炸弹

在立项之初，我们构思过《荒野之息》中十分具有特色的希卡之石上的四项能力（遥控炸弹、磁铁、时间静止器、冰柱发生器）该如何实现，大致上都有了一个实现思路的雏形，其中遥控炸弹和冰柱是相对最容易实现的。然而受限于时间，我们最终只实现了遥控炸弹。



图 7-5 《荒野之息》中的四项能力

在《荒野之息》中，按下手柄的 LB 肩键可以掏出遥控炸弹，再按下 LB 键丢出，此时根据炸弹是圆形或是方形决定其运动性质，再按下则会引爆，破坏物体或造成伤害。

这里我们将其简化为类似于“黏性炸弹”，碰到地形后直接固定，不考虑后续运动。另外也没有做生命值机制与敌人，因此只会破坏爆炸范围内的物体。

- 使用 `Bomb` 类进行管理，炸弹使用 `obj` 文件导入
- `Bomb` 有四个状态，分别表示为 0, 1, 2, 3。
 - 3 表示初始状态
 - 1 表示举起炸弹，表现为炸弹会出现在长方体正上方，并且会随长方体移动而跟随

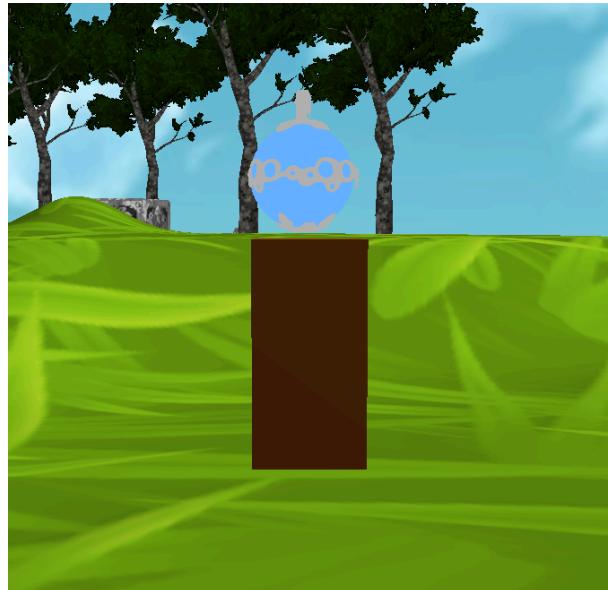


图 7-6 举起炸弹

- 2 表示丢出炸弹，炸弹在空中会按照斜抛运动轨迹更新位置，落地后会停在原地

```
1 void Bomb::moveParabola(Terrain* terrain, float t) { Cpp
2     if (land) return;
3     velocity.y -= gravity * t;
4     position += velocity * t;
5     float terrainHeight = terrain->getHeight(position.x, position.z);
6     if (position.y <= terrainHeight + 1.0f) {
7         land = true;
8     }
9 }
```

- 0 表示引爆炸弹，引爆后会产生爆炸效果，持续 1 秒。当有可交互物体在爆炸范围内时，会产生对应的交互效果



图 7-7 引爆炸弹

- 初始状态设置为 3，每按一次 Q 键状态会加 1 并模 3

7.7 剑气（攻击）

我们没有做复杂的人物动作，只简化为一个抽象的无底座四棱锥(宽度缩小实现剑气效果)，但又想赋予其攻击与防御（虽然好像没有能防御的东西 x）的能力，因此这里我们做了简化与抽象的攻击防御动作。

- 这里的实现逻辑主要是处理按键操作，即左键按下的逻辑处理
 1. 我们这里先将人物状态中攻击信号置为 True
 2. 将剑气动画的关键帧导入并计算缩放因子
 3. 结合动画帧的实现，完成剑气的挥出
 4. 利用人物状态的攻击信号与其他物体产生碰撞逻辑的检测，实现攻击逻辑

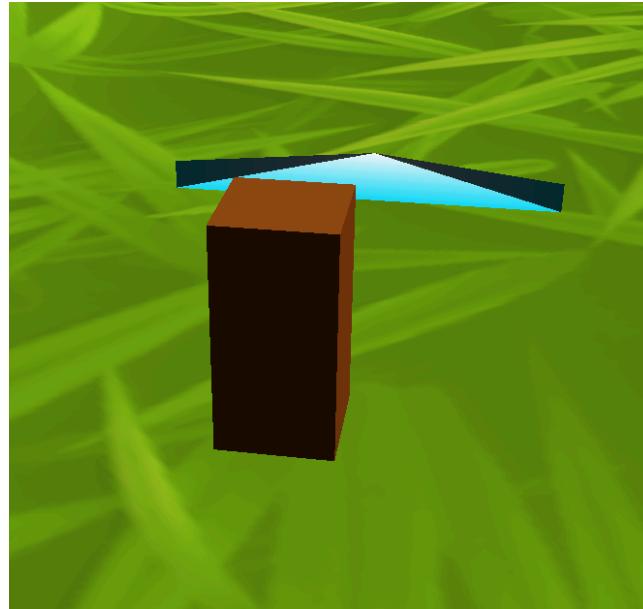


图 7-8 剑气效果

7.8 盾牌（防御）

- 实现和剑气相一致，重点在动画帧的完成，
- 剑气与盾牌的着色器特殊设置了渐变，使得显示更加逼真



图 7-9 盾牌

7.9 交互

《荒野之息》的一大特色就在于丰富而符合直觉的交互，但这里我们显然没法做这么复杂，做了大量的简化。

- 使用炸弹和剑气可以对可交互物体造成伤害，当可交互物体在炸弹的爆炸范围或剑气的攻击范围之中，视为造成伤害。
 - 可破坏的树木（白桦树和苹果树）受到伤害后变为树桩（原本是先变为倒下的树干，再被破坏成木柴捆）



图 7-10 树桩

- 木箱受到伤害后消失（暂未实现粒子效果）
- 铁箱受到伤害后表面出现损毁痕迹

8 场景物体导入与创建

8.1 Obj 模型导入

Ref: [LearnOpenGL-模型](#)

使用 `assimp + stb_image`，我们这里以 `.obj` 和 `.mtl` 文件导入树木和箱子，材质为 `.png` 格式。

在使用 OpenGL 导入和渲染 3D 模型时，`Assimp` 用于加载和处理模型文件（如 `.obj` 和 `.mtl`），而 `stb_image` 用于加载纹理图像。

8.1.1 Assimp 的工作流程

1. 加载模型文件：

- 使用 `aiImportFile` 函数加载 `.obj` 文件。`.obj` 文件通常包含模型的几何数据（顶点、法线、纹理坐标等）。
- 如果模型使用了材质，`Assimp` 会同时加载关联的 `.mtl` 文件。`.mtl` 文件定义了材质属性（如漫反射颜色、镜面反射颜色、纹理路径等）。

2. 解析模型数据：

- `Assimp` 将模型数据解析为一个场景对象（`aiScene`），其中包含多个网格（`aiMesh`）和材质（`aiMaterial`）。
- 每个 `aiMesh` 包含顶点数据（位置、法线、纹理坐标等）和索引数据（用于绘制三角形）。
- 每个 `aiMaterial` 包含材质属性（如颜色、纹理路径等）。

3. 提取数据:

- 遍历 `aiScene`，提取每个 `aiMesh` 的顶点和索引数据，并将其存储到 OpenGL 的缓冲区 (VBO 和 EBO) 中。
- 遍历 `aiMaterial`，提取材质属性 (如纹理路径)，并使用 `stb_image` 加载纹理。

8.1.2 stb_image 的工作流程

1. 加载纹理文件:

- 使用 `stbi_load` 函数加载纹理文件。函数返回图像的像素数据、宽度、高度和通道数。
- 如果加载失败，可以检查文件路径或格式是否正确。

2. 生成 OpenGL 纹理:

- 使用 OpenGL 的 `glGenTextures` 生成纹理对象。
- 使用 `glTexImage2D` 将图像数据上传到 GPU。
- 设置纹理参数 (如 `GL_TEXTURE_WRAP_S`, `GL_TEXTURE_MIN_FILTER` 等)。
- 这些生成纹理的重复性工作都被整合到纹理类中。

8.1.3 整合 Assimp 和 stb_image

1. 加载模型:

- 使用 Assimp 加载 `.obj` 和 `.mtl` 文件，解析模型和材质数据。
- 提取顶点、法线、纹理坐标和索引数据，并将其存储到 OpenGL 的缓冲区中。

2. 加载纹理:

- 从 `.mtl` 文件中提取材质信息。
- 使用 `stb_image` 加载纹理图像，并生成 OpenGL 纹理对象。

3. 渲染模型:

- 绑定纹理和材质属性。
- 使用 OpenGL 的 `glDrawElements` 或 `glDrawArrays` 绘制模型。

8.1.4 自定义的模型着色器

自定义了一个模型着色器，把先前描述的阴影映射、光照模型等纳入考虑，并且做了法向贴图、透明贴图等。

```

1  in vec2 texCoords;
2  in vec3 worldFragPos;
3  in vec4 shadowFragPos;
4  in mat3 TBN;
5  // ...
6  uniform sampler2D texture_diffuse1;
7  uniform sampler2D texture_normal1;
8  uniform sampler2D texture_trans1;
9  uniform sampler2D shadowMap;
10 // ...

```

glsl

8.2 实际模型展示

1. 树木分类

- 橡树：不可摧毁



图 8-1 橡树林

- 白桦树与苹果树：可摧毁



图 8-2 白桦林

2. 箱子分类

- 木箱：攻击一次后消失

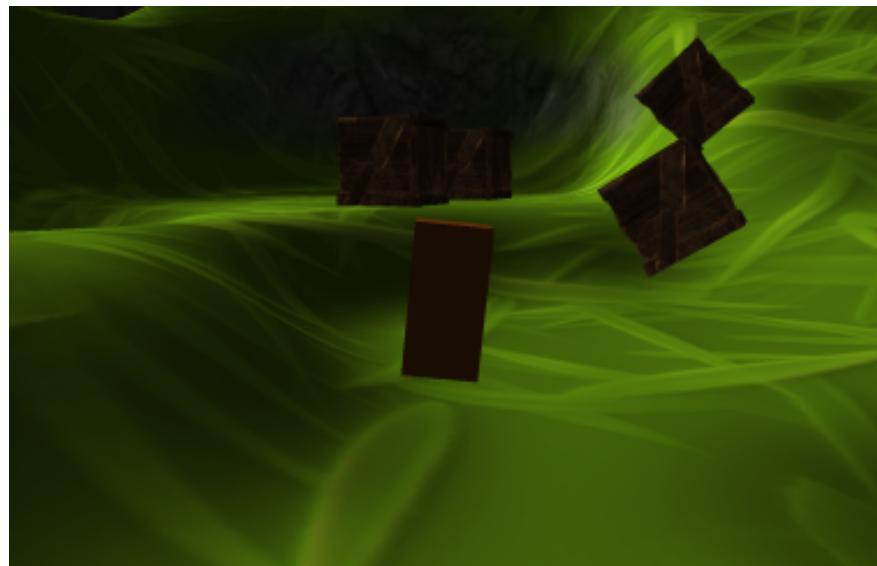


图 8-3 木箱

- 铁箱：攻击一次后破损，不可彻底摧毁

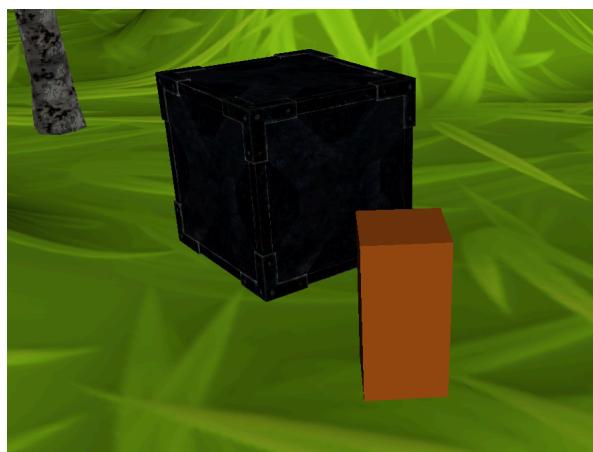


图 8-4 未被破坏铁箱



图 8-5 破损铁箱

- 特制铁箱：不可被摧毁



图 8-6 特制铁箱

8.3 碰撞检测

这里我们说明的主要是物体之间的碰撞检测,主要包括有 AABB 和 OBB 式碰撞检测

攻击交互的判定其实是类似的——我们将武器简化为相应的模型,实际上还是进行了模型之间的碰撞检测,

AABB (Axis-Aligned Bounding Box)

- 定义:

AABB 是与坐标轴对齐的立方体 (3D),AABB 可以用 $(\min_x, \min_y, \min_z, \max_x, \max_y, \max_z)$ 表示。

- 特性:

边与坐标轴平行,因此计算简单。无法旋转,只能通过平移和缩放来调整。

- 碰撞检测:

检测两个 AABB 是否相交,只需比较它们在每个轴上的投影是否重叠。我们实现了人物与木箱、铁箱、树木的碰撞检测,确保了人物不会产生地形穿模的问题(点名某育 x,保证了大世界的可游玩性;此外我们的攻击破坏物体也做了相同逻辑的判断,可以流畅地显示破坏地形的动画,苦于时间限制,我们本希望能做出更丰富的地形甚至 NPC 供娱乐,这里便只实现一些基础的物品。

Algorithm 1: 3D AABB Collision Detection

input: Two 3D AABBs a and b , defined by $(\min_x, \min_y, \min_z, \max_x, \max_y, \max_z)$

output: Boolean indicating whether a and b intersect

```

1 if  $a.\max_x < b.\min_x$  or  $a.\min_x > b.\max_x$  then
2   return False                                ▷ No overlap on the x-axis
3 end
4 if  $a.\max_y < b.\min_y$  or  $a.\min_y > b.\max_y$  then
5   return False                                ▷ No overlap on the y-axis
6 end
7 if  $a.\max_z < b.\min_z$  or  $a.\min_z > b.\max_z$  then
8   return False                                ▷ No overlap on the z-axis
9 end
10 return True                                 ▷ AABBs intersect in all axes

```

我们工程中实现了 AABB 式检测碰撞,由于我们的物体与方块均为立方体,AABB 算法能在计算简便的同时保证碰撞检测是基本有效的,源代码可读性比伪代码要低,此处就不展开源码分析了。PS:需要注意的是,我们的炸弹检测做了球体与立方体的碰撞检测,这里的实现不难因此我们略过了对它的介绍。

9 Appendix

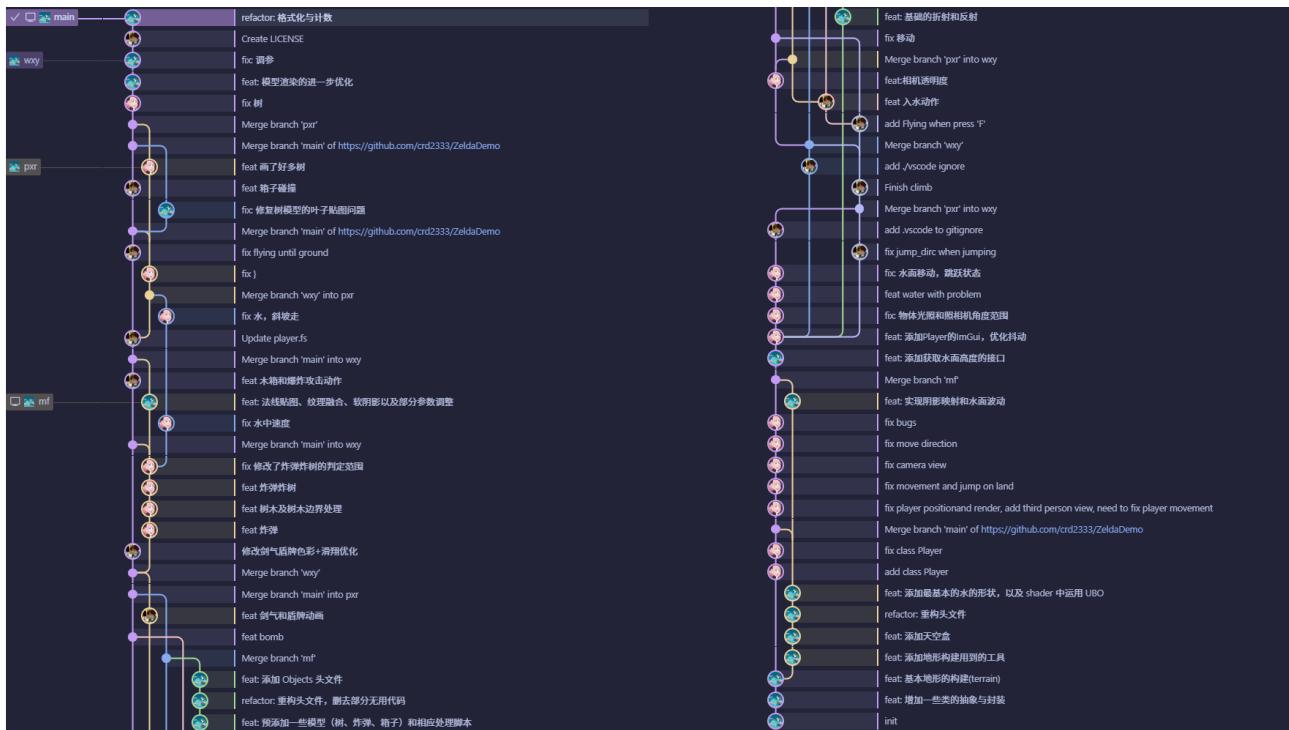


图 9-1 git 提交记录

language	files	code	comment	blank	total
C++	40	4,854	724	822	6,400
GLSL	20	558	42	117	717
Python	6	407	33	92	532
Markdown	1	2	0	0	2

表 9-1 VSCode Counter: Languages

path	files	code	comment	blank	total
total	67	5,821	799	1,031	7,651
include (without third-party)	20	1,281	188	296	1,765
resources	21	753	80	153	986
src	16	2,711	377	331	3,419
utils	10	1,076	154	251	1,481

表 9-2 VSCode Counter: Directories

完结撒花!!!