装 订 线

同济大学计算机系

计算机图形学课程实验报告

小组成员 曹正青、付舒午、黄郅超
刘安民、王君豪、杨子昊

目录

1 概述

本项目名为裔拓岛(YitoIsland)，设计灵感来自游戏《艾尔登法环》(Elden Ring)，主要实现与之类似风格的“黑暗幻想风”场景渲染。

本项目实现了草地、水面、火焰、树木等实体的生成，PBR 光照（地面）和第三人称视角的实现，以及人物、地面、石台、房屋等物体的渲染。

项目分工：

2052436 曹正青：实现地形生成及沙地效果渲染以及人物模型设计

2053188 杨子昊：实现树木生成以及纹理效果渲染

2051965 刘安民：实现火焰和石头材质的效果渲染

2050653 王君豪：实现草地效果渲染

2053036 付舒午：实现光照效果渲染

2053510 黄郅超：实现水面效果渲染以及人物移动设计

下面从每个技术细节阐述整个项目

装 订 线

2 地形与人物模型部分

2.1 实现概述

地形部分使用 blender 进行建模，达到预期渲染效果后烘焙出所需要的 PBR 贴图，并在实际项目中使用。

人物部分使用 pmx 可动模型，做出相应动作后输出为 obj 模型，之后通过 obj 不同姿势循环播放的方式实现走路的效果。

2.2 地形部分

2.2.1 模型构建

采用 blender 生成一个类似小岛的模型

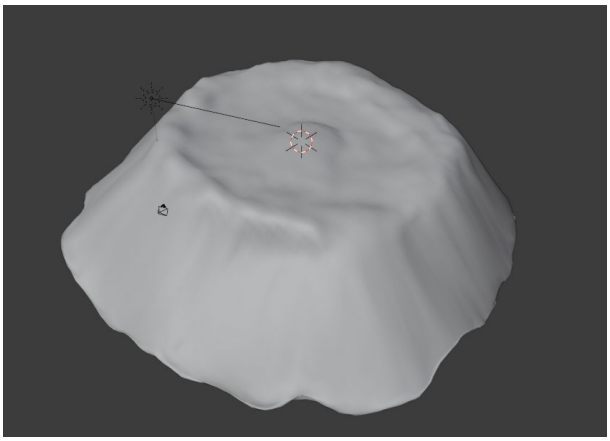


图 2.1. 未上色的模型预览

调整 shader 参数，以得到真实的颜色和光照反射效果，以实现富有真实感的沙地小岛。

主要思路是采用两种不同颜色的纯色材质，分别代表硬地和表面的浮沙，之后增大硬地材质的粗糙感，增强反射亮度，实现土的效果，浮沙材质则不需要太大的粗糙度，根据模型的地形将浮沙限定在较高处的区域，以实现比较真实的效果。

图 2.2. 调整参数后的模型整体预览

在两种材质的连接处，采用类似线性差值的方式，来实现比较自然的过渡效果。

装 订 线

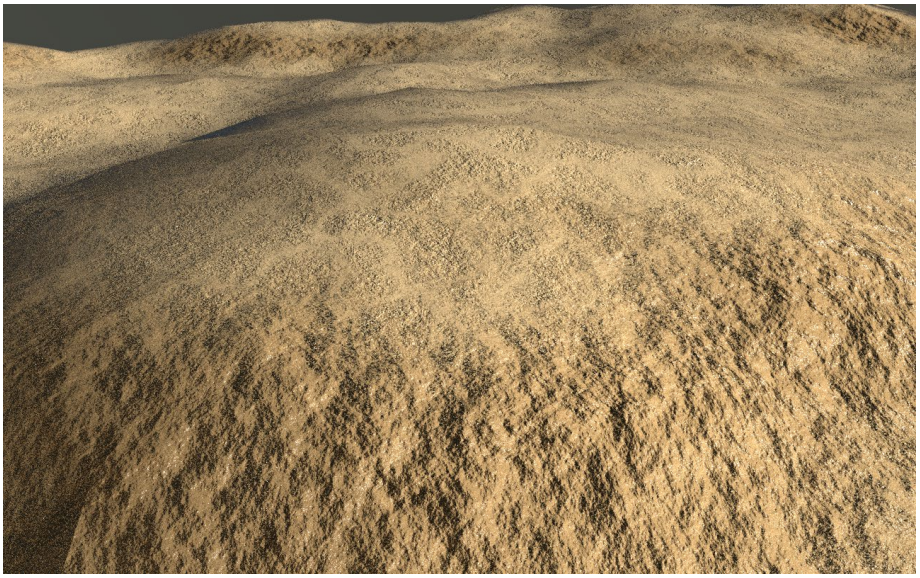


图 2.3.两种材质连接处细节展示

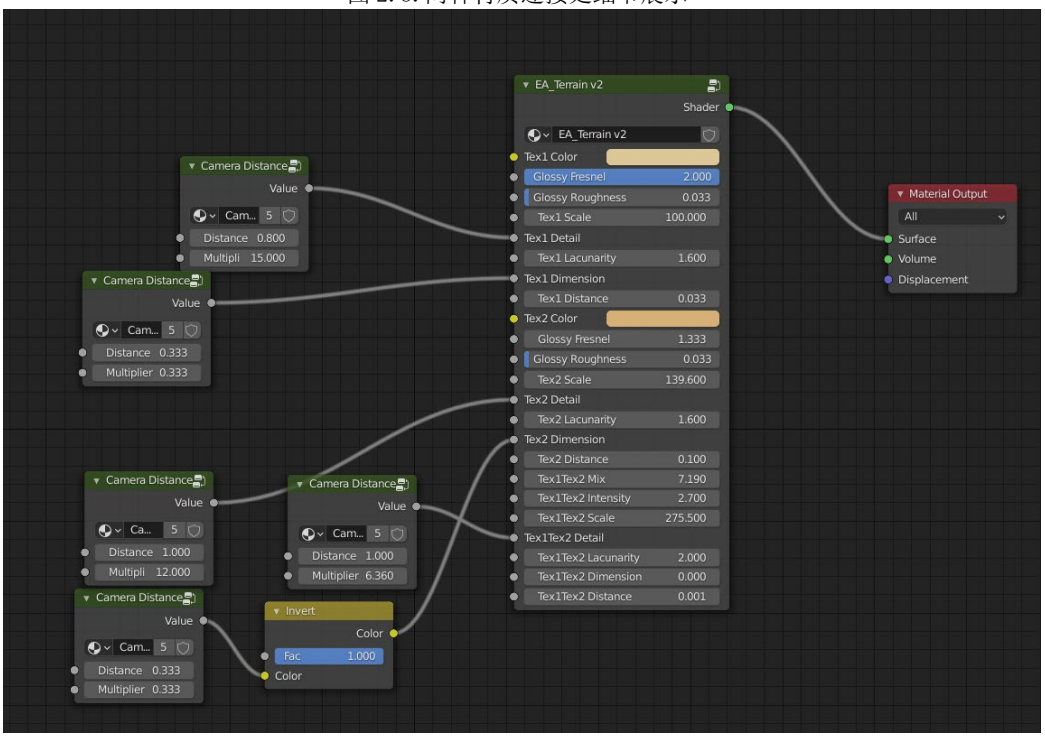


图 2.4. shader 具体参数

2.2.2 材质输出

由于是直接生成的材质，只能在 blender 中渲染，不能直接在项目中进行应用，且导出的 obj 模型也不含任何贴图材质，因此需要根据项目需要烘焙出特定的材质贴图。

在本项目中，负责光照部分的同学使用了基于 albedo, normal, metallic, roughness, ao，也即基本色、法线、金属度、粗糙度、环境光遮蔽工作流的 PBR 光照模式，因此需要分别输出相应的贴图。

使用 blender 自带的 bake 功能，自定义采样点和采样数量，分别烘焙出需要的贴图。

但是其中 metallic 贴图无法直接烘焙，且由于调整参数时并未用到 metallic 参数，也无法使用传统的仅渲染 metallic 参数的方法得到 metallic 贴图，查阅许多资料，也仅有传统的方法能够实现 metallic 贴图的烘焙，无奈只能用 Glossy 贴图，即高光贴图来代替，最后生成的效果随不能完全等同于 blender 中渲染的效果，但也差不太多

图 2.5.最后生成的五幅贴图

2.3 人物部分

2.3.1 模型构建

由于本项目的背景是游戏艾尔登法环(Elden Ring)，因此寻找了一个游戏中的开源模型。且是可动的 pmx 格式模型，因此采用改变 pmx 模型动作的方式，实现一个 4 帧循环播放的走路效果。

2.3.2 困难解决

由于原生 blender 不支持 pmx 格式模型的导入与操作，因此需要额外插件 CATS，而我所使用的 blender 版本与 CATS 插件并不兼容，因此需要回退 blender 版本。

同时输出的 obj 不能向传统.blend 文件一样直接含原生贴图，必须手动在.mtl 文件中贴上每一部分的贴图，最后达成了绝大部分贴图的正确粘贴，但是大臂部分的贴图实在无法贴上。

最终效果图如下：

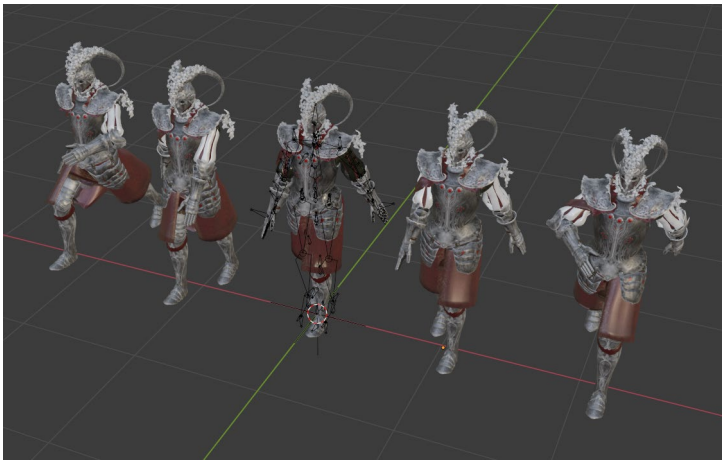


图 x.6.最终得到的 4 帧走路动画

3 草地部分

实现细节：本部分希望实现在地形上绘制草地，希望草地经可能真实。实现主要分为两部分，草簇的绘制与草簇位置的选取

草簇的绘制：主要技术:Gemetry Shader, Billboards, LOD

草簇的绘制主要在几何着色器中完成：

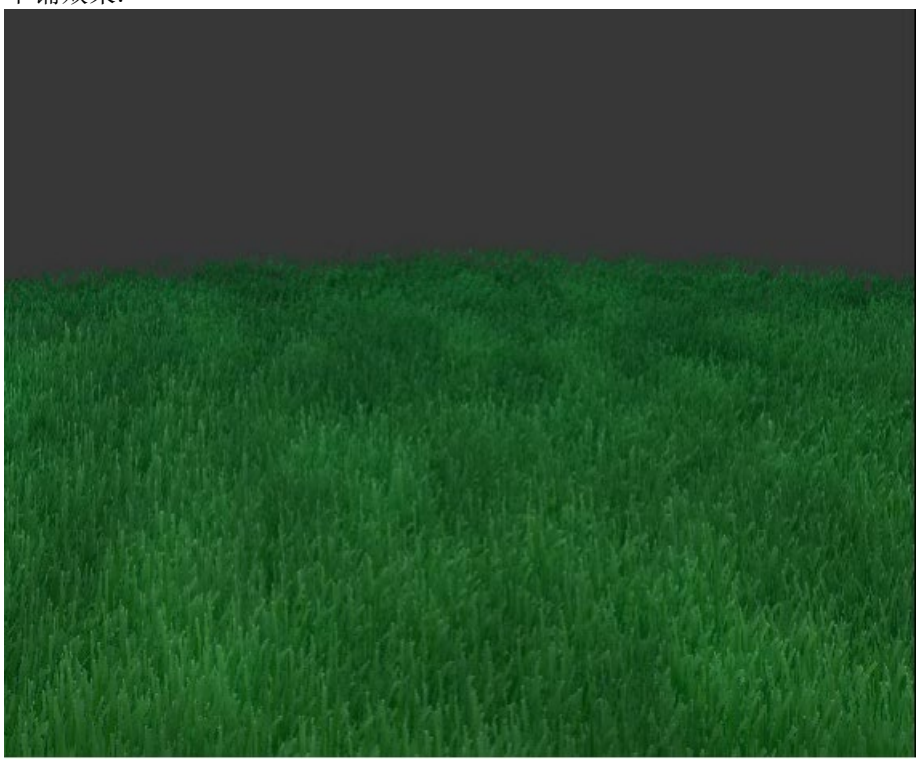
Billboards 图元生成，并且模拟风吹效果：

```
/* MAIN FUNCTIONS */
void createQuad(vec3 base_position, mat4 crossmodel){
    vec4 vertexPosition[4];
    vertexPosition[0] = vec4(-0.25, 0.0, 0.0, 0.0);     // down left
    vertexPosition[1] = vec4( 0.25, 0.0, 0.0, 0.0);     // down right
    vertexPosition[2] = vec4(-0.25, 0.5, 0.0, 0.0);     // up left
    vertexPosition[3] = vec4( 0.25, 0.5, 0.0, 0.0);     // up right

    vec2 textCoords[4];
    textCoords[0] = vec2(0.0, 0.0);                     // down left
    textCoords[1] = vec2(1.0, 0.0);                     // down right
    textCoords[2] = vec2(0.0, 1.0);                     // up left
    textCoords[3] = vec2(1.0, 1.0);                     // up right

    // wind
    vec2 windDirection = vec2(1.0, 1.0); float windStrength = 0.15f;
    vec2 uv = base_position.xz/10.0 + windDirection * windStrength * u_time ;
    uv.x = mod(uv.x,1.0);
    uv.y = mod(uv.y,1.0);
    vec4 wind = texture(u_wind, uv);
    mat4 modelWind = (rotationX(wind.x*PI*0.75f - PI*0.25f) * rotationZ(wind.y*PI*0.75f - PI*0.25f));
    mat4 modelWindApply = mat4(1);

    // random rotation on Y
    mat4 modelRandY = rotationY(random(base_position.zx)*PI);

    // Loop of billboard creation
    for(int i = 0; i < 4; i++) {
        if (i == 2 ) modelWindApply = modelWind;
        gl_Position = u_projection * u_view *
            (u_model * gl_in[0].gl_Position + modelWindApply*modelRandY*crossmodel*(vertexPosition[i]*grass_size));

        gs_out.textCoord = textCoords[i];
        gs_out.colorVariation = fbm(gl_in[0].gl_Position.xz);
        EmitVertex();
    }
    EndPrimitive();
}
```

应用纹理并且使用里摄像机距离控制 LOD

```
void createGrass(int numberQuads){
    mat4 model0, model45, modelm45;
    model0 = mat4(1.0f);
    model45 = rotationY(radians(45));
    modelm45 = rotationY(-radians(45));

    switch(numberQuads) {
        case 1: {
            createQuad(gl_in[0].gl_Position.xyz, model0);
            break;
        }
        case 2: {
            createQuad(gl_in[0].gl_Position.xyz, model45);
            createQuad(gl_in[0].gl_Position.xyz, modelm45);
            break;
        }
        case 3: {
            createQuad(gl_in[0].gl_Position.xyz, model0);
            createQuad(gl_in[0].gl_Position.xyz, model45);
            createQuad(gl_in[0].gl_Position.xyz, modelm45);
            break;
        }
    }
```

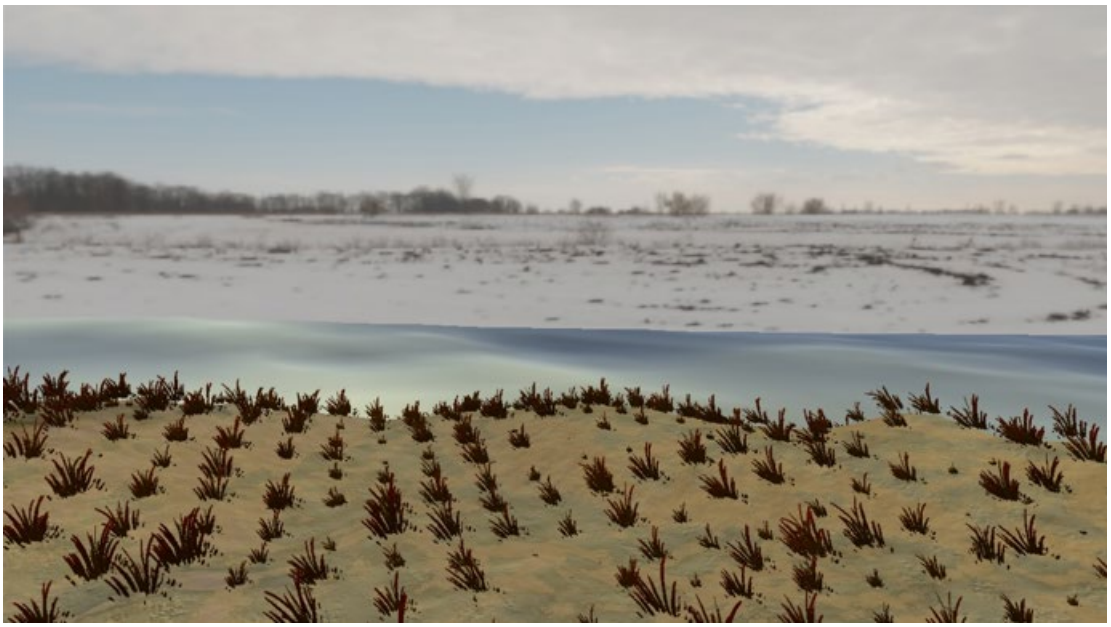
平铺效果:



位置选取：通过遍历地形数据，通过地形的顶点数据通过一定的噪声加筛选，实现草长在地上的效果:

```
// grass
std::vector<glm::vec3> grass_positions;
for (auto &mesh : landModel.meshes) {
    for (auto &vertex : mesh.vertices) {
        glm::vec3 pos = vertex.Position;
        if (-10.f < pos.y && pos.y < 10.f && -10.f < pos.x && pos.x < 10.f)
            grass_positions.push_back(pos);
    }
}
```

最终效果：



装

订

线

4 光照部分

具体实现：通过 PBR 技术尽可能真实地模拟现实中的光照，是场景效果更为真实。本次场景中共设置了四个点光源，其中一个在火焰中，我们对强度和颜色进行了修改，契合场景，然后是对于整个外围场景（天空盒）进行计算得到辐射度图作为环境光来影响场景，使场景中的模型更为真实。最终主要是对地面和房子模型添加了 PBR 光照效果。

光照部分的实现主要是在几何着色器中，

```
6
7  //材料的属性，在外面设置
8  //uniform vec3 albedo;     //反照率,只包含表面颜色
9  //uniform float metallic; //金属度，
10 //uniform float roughness;//粗糙度，
11 //uniform float ao;        //环境光遮蔽
12 // 材料参数，贴图
13 uniform sampler2D albedoMap;
14 uniform sampler2D normalMap;
15 uniform sampler2D metallicMap;
16 uniform sampler2D roughnessMap;
17 uniform sampler2D aoMap;
18
```

最终主要是设计模型的各类参数贴图得到相应的材料属性，然后根据已有的反射方程计算所有场上光源的各部分的影响结合到一起输出最终效果。

然后我们就使用帧缓冲加载生成相应的光照贴图，并输出到模型上即可。

```
//放入pbr光照贴图
//************************************************************
//1、辐射度贴图
Shader shaderIrradiance("shader/cubeMap.vs","shader/irradianceMap.fs");
unsigned int irradianceMap = loadIrradianceMap(shaderIrradiance, envCubemap);

//2、预滤器
Shader shaderPrefilter("shader/prefilter.vs", "shader/prefilter.fs");
unsigned int prefilterMap = loadPrefilterMap(shaderPrefilter, envCubemap);

//3、brdf
Shader shaderBrdf("shader/brdf.vs", "shader/brdf.fs");
unsigned int brdfLUTTexture = loadBrdfMap(shaderBrdf);
//************************************************************
```

```
//渲染模型
//1、land  TODO::
Shader shaderModels("shader/pbr_model.vs", "shader/pbr_model.fs");
shaderModels.use();
shaderModels.setInt("irradianceMap", 5);
shaderModels.setInt("prefilterMap", 6);
shaderModels.setInt("brdfLUT", 7);
shaderModels.setInt("albedoMap", 0);
shaderModels.setInt("normalMap", 1);
shaderModels.setInt("metallicMap", 2);
shaderModels.setInt("roughnessMap", 3);
shaderModels.setInt("aoMap", 4);


Model landModel("model/land/land.obj");
Model modelStone1("res/StonePlatform_Obj/StonePlatform_A.obj");
Model modelStone2("res/StonePlatform_Obj/StonePlatform_B.obj");
Model modelStone3("res/StonePlatform_Obj/StonePlatform_C.obj");
Model modelHouse("res/house/cliffHouse/house.obj");
//读纹理数据
unsigned int albedoLand = loadTexture("model/land/Diffuse.png");
unsigned int normalLand = loadTexture("model/land/Normal.png");
unsigned int metallicLand = loadTexture("model/land/Glossy.png");
```

实现效果

放入场景，加载场景模型最终效果

装 订 线

装 订 线

5 光照部分

5.1 概述

火焰渲染主要使用粒子系统，通过若干二维“光斑”的纹理贴图组合叠加，并对叠加得到整体的各个属性进行简单的物理模拟，实现三维火焰的基本视觉效果。

flame.png

5.2 技术路线

5.2.1. 粒子位置分布

粒子在三维空间的分布遵循二维正态分布：

$$Pos(x) = \frac{1}{\sqrt{2\pi r}} e^{-\frac{(X_i - X_0)^2}{2r^2}}$$
$$Pos(y) = 0$$
$$Pos(x) = \frac{1}{\sqrt{2\pi r}} e^{-\frac{(Z_i - Z_0)^2}{2r^2}}$$

为了减轻运算压力，我们采用多次随机采样的方法近似正态分布：

$$Pos(x) = \frac{1}{n}\sum_{i=0}^{n} Rand1() * Adj\_value$$
$$Pos(y) = 0$$
$$Pos(z) = \frac{1}{n}\sum_{i=0}^{n} Rand2() * Adj\_value$$

代码如下：

```
glm::vec3 record(0.0f);
// 使用多次采样的方法模拟高斯分布
for (int y = 0; y < SAMPLING_COUNT; y++) { //生成高斯分布的粒子，中心多，外边少
    record.x += (2.0f * float(rand()) / float(RAND_MAX) - 1.0f);
    record.z += (2.0f * float(rand()) / float(RAND_MAX) - 1.0f);
}
record.x = record.x * Radius / SAMPLING_COUNT;
record.z = record.z * Radius / SAMPLING_COUNT;
record.y = Position.y;
```

5.2.2. 粒子初始速度/大小/寿命

粒子速度、大小和寿命的初始值应当有一定的随机性：

$$Value = MinValue + (MaxValue - MinValue) * Rand()$$

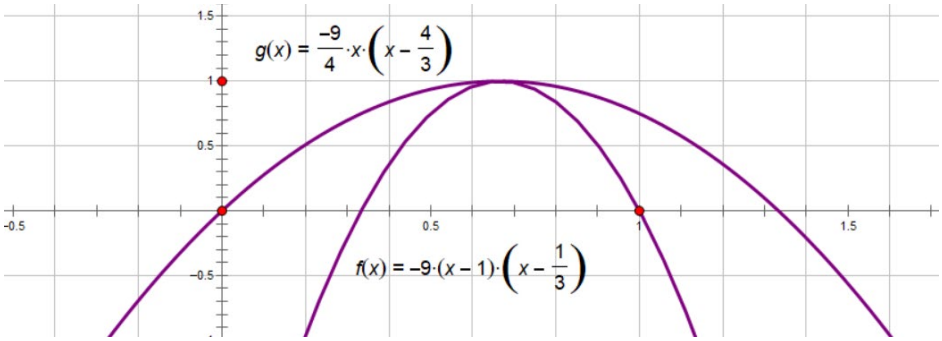
由此得到在预设的最小值和最大值之间随机的粒子属性。

5.2.3. 粒子透明度/大小随时间变化模型

粒子透明度和大小属性应当满足从最小值增加到最大值再降低到最小值的变化状态。在此使用两条二次函数进行模拟，函数最高点均为 1.

$$f_1(t) = -\frac{9}{t_0^2} * (t - t_0) * \left(t - \frac{t_0}{3}\right)$$
$$f_2(t) = -\frac{9}{4t_0^2} * t * (t - \frac{4t_0}{3})$$

设$t_0 = 1$，函数图像如下：



两个函数最高点相同，取$g(x)$左半和$f(x)$右半连接。代码见下：

```
// 使用两条对称轴在 2 * life / 3 位置的二次函数拟合大小和亮度的变化曲线
GLfloat factor = 0.0f;
if (p.CurLife >= 2 * p.Life / 3)
    factor = -9 / (p.Life * p.Life) * (p.CurLife - p.Life) * (p.CurLife - p.Life / 3);
else
    factor = -9 / (4 * p.Life * p.Life) * p.CurLife * (p.CurLife - 4 * p.Life / 3);
// 将factor值域映射到[0,1]
```

实现时曾尝试使用

$$Value = \frac{1.0}{\left(Age - \frac{Life}{2}\right)^2 + 1.0} * MaxValue$$

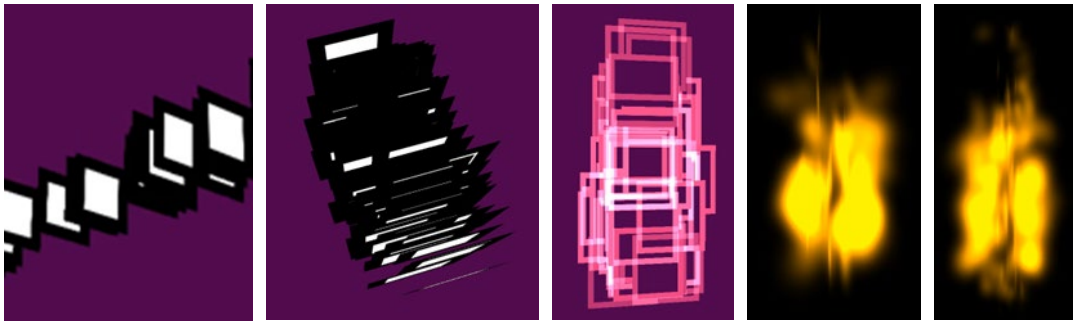
作为变化函数，但效果不佳，即使作函数映射到[0,1]结果仍然不理想，故更换为二次函数。

5.2.4. 粒子系统使用

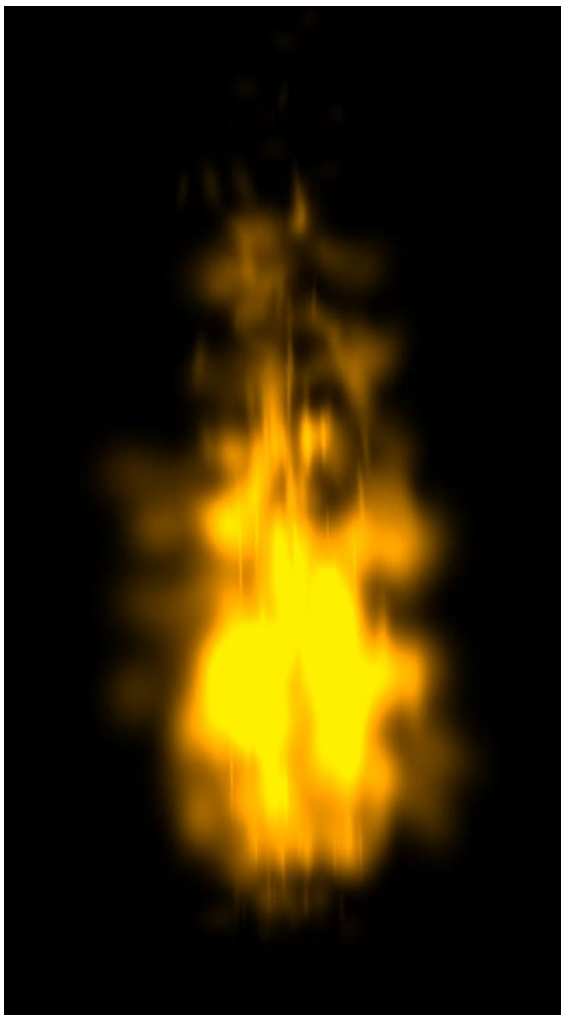
所谓粒子系统，通俗来说，即多次重复渲染相同物体，只是渲染的物体较小，因此呈现效果类似粒子分散。粒子系统的使用不再赘述，但需要注意：如果使用透明度贴图，绘制时需关闭深度检测，修改混合模式，启用 GL_BLEND，绘制完成后恢复原样。由于绘制时深度检测暂时关闭，还需注意火焰和其他元素的绘制顺序。此外，需注意纹理绑定与设为活跃，防止纹理贴图错乱。

5.3. 结果展示

下面展示了开发过程中火焰的效果变化：



最终结果：



在项目中的效果（未加光照）：

6 树木部分

树木部分实现方法

实现细节

本项目的目标是实现类似艾尔登法环的某个小岛风格的场景。由于艾尔登法环的小岛的树木树干非常多且较为庞大，经过尝试后发现导入模型并不能较好模拟最终的效果，所以采用了 OpenGL 中渲染流水线的方法在场景中生成树，并且采用了分形的思想和 L-system 的算法，这样可以控制生成任意形状的树，不再受给定模型的限制。

L-system 是允许通过使用迭代来定义复杂形状来实现分形的算法。通过使用一种数学语言，其中将初始字符串与重复评估的规则进行匹配，并将结果用于生成几何图形。每次评估的结果成为下一次几何迭代的基础，给人以生长的感觉。L-system 主要关注分形属性的自相似性和发展算法，核心概念是重写，这是通过重写的几何体递归地替换初始状态来工作的。通过重写 Turtle 命令控制树木生长形状。Turtle 命令是一个字符串，将字符串操作系统与一个图形例程结合起来，该例程将字符串解释为用于绘制具有位置 (XYZ) 和航向（角度）的“Turtle”的命令。通过遵循该命令，Turtle 在移动时描绘出一个形状。

例如：

A= “ [&FFFA] //// [&FFFA] //// [&FFFA] 会执行以下操作：

“ ：缩放当前分支长度

[&FFFA]：向下倾斜(&)并绘制一个分支，然后插入一个递归副本(A)。（重复三次，中间有旋转）

////：逆时针旋转四次。

生成树的结果：

本项目中的 Turtle 命令的定义，参考了该网站中的定义方式：
https://www.sidefx.com/docs/houdini/nodes/sop/lsystem.html

在本项目中，生成的树木形状和大小定义在 json 文件中，并由 nlohmann json 方法在 C++中解析 json 语句。最终生成了两棵树，分别读入了两个 json 文件，tree3.json 和 tree4.json(项目中还尝试了 tree1.json 和 tree2.json，但效果不好所以没有采用)：

tree3.json：

```
{
    "branchRadius": 0.6,
    "branchStep": 0.75,
    "branchAngle": 0.9,
    "branchThicknessRatio": 0.6,
    "branchStepRatio": 0.75,
    "minRadius": 0.00,
    "maxLeafRadius": 0.12,
    "leafLength": 0.4,
    "leafWidth": 0.4,
    "leafDensity": 15,
    "leafTextureFile": "./texture/leaf4.png",
    "woodTextureFile": "./texture/wood4.bmp",
    "Turtle": {
        "x": 0,
        "y": -6,
        "z": 0
    },
    "LSystem": {
        "Start": "F{5.5}",
        "Rules" : [{
                    "Input" : "F{x}",
                    "Output" : "[&F{x}][+F{x}]",
                    "Condition" : "x < 5"
                    },
                    {
                    "Input" : "F{x}",
                    "Output" : "F{x}[/+F{1.2 *  x}][&F{x * 1.2}]['F{x * 1.2}][\\&F{1.2 * x}][-F{x}]",
                    "Condition" : "x > 4 && x < 7"
                    },
                    {
                    "Input" : "F{x}",
                    "Output" : "F{x}[/+F{1.1 *  x}][\\&F{1.2 * x}][-F{x}]",
                    "Condition" : "x > 7"
                    }]
    },
    "iterations" : 5
}
```

tree4.json：

通过读入 json 数据并解析，可以得到树干顶点和叶子顶点，分别对其添加纹理即可。

```
{
    "branchRadius" : 0.21,
    "branchStep" : 0.55,
    "branchAngle" : 0.6,
    "branchThicknessRatio" : 0.6,
    "branchStepRatio" : 0.65,
    "minRadius": 0.00,
    "leafTextureFile": "./texture/leaf3.png",
    "woodTextureFile": "./texture/wood1.bmp",
    "Turtle": {
        "x": 0,
        "y": -6,
        "z": 0
    },
    "LSystem": {
        "Start": "F{5.5}",
        "Rules" : [{
                    "Input" : "F{x}",
                    "Output" : "[&F{x}][+F{x}]",
                    "Condition" : "x < 5"
                    },
                    {
                    "Input" : "+",
                    "Output" : "+F{3}+"
                    },
                    {
                    "Input" : "F{x}",
                    "Output" : "F{x}[/+F{1.1 *  x}][\\&F{1.2 * x}][-F{x}]",
                    "Condition" : "x > 5"
                    }]
    },
    "iterations" : 5
}
```

在生成纹理的过程中，存在了树干贴图出现摩尔纹而导致贴图效果不佳的问题:

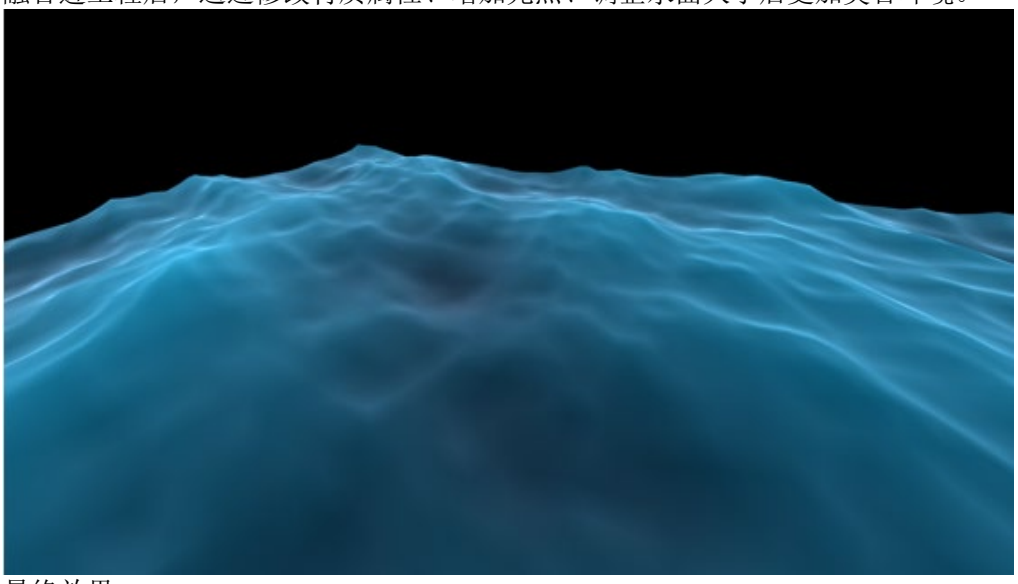
使用 GL_NEAREST_MIPMAP_NEAREST 参数实现 mipmapping，从而减弱摩尔纹效果：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_NEAREST_MIPMAP_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_NEAREST_MIPMAP_NEAREST);
```

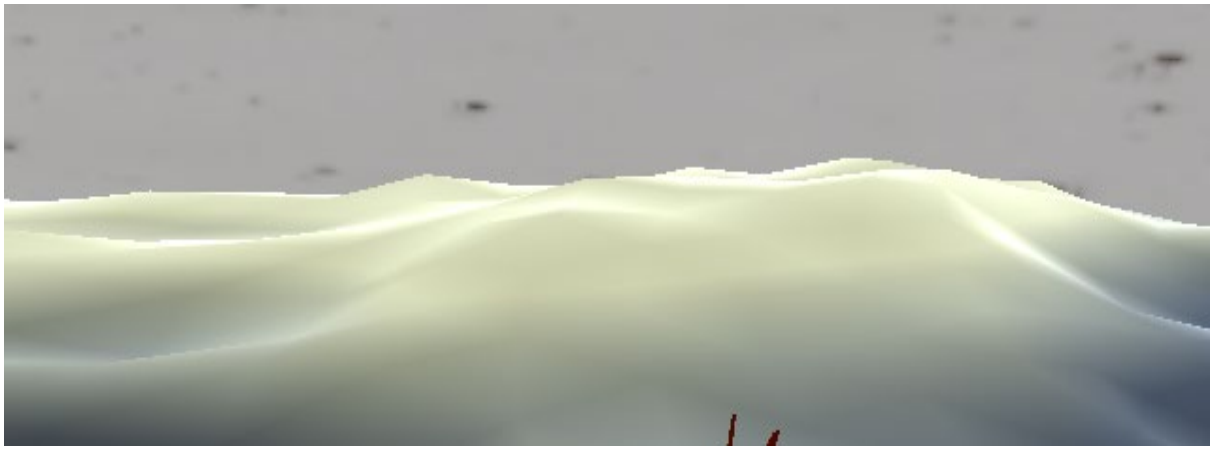
效果如下：

7 水流部分

水面变换形成波浪效果使用了 fft 算法。
快速傅里叶变换（fft），是一种可在 O（nlogn）时间内完成离散傅里叶变换的高效、快速计算方法集的统称。
FFT 的基本思想是把原始的 N 点序列，依次分解成一系列的短序列。充分利用 DFT 计算式中指数因子 所具有的对称性质和周期性质，进而求出这些短序列相应的 DFT 并进行适当组合，达到删除重复计算，减少乘法运算和简化结构的目的。
基于 FFT 的水体渲染方法，也被称为基于频谱的方法，其核心思想是基于 FFT 构造出水体的表面高度。通过变换水体表面高度，形成视觉上的波动效果，模拟出水面波浪。
融合进工程后，通过修改材质属性、增加光照、调整水面大小后更加契合环境。



最终效果：



8 人物移动部分

8.1 第三人称摄像头

在摄像机类中添加人物的坐标和摄像机与人物距离参数，通过俯仰角 Pitch、偏航角 Yaw 和人物坐标以及距离参数进行三角函数运算，计算出摄像机坐标。

```
Position.x = playerPos.x + cos(glm::radians(Yaw)) * cos(glm::radians(Pitch)) * distanceFromPlayer;
Position.y = playerPos.y + sin(glm::radians(Pitch)) * distanceFromPlayer + 50.0f;
Position.z = playerPos.z + sin(glm::radians(Yaw)) * cos(glm::radians(Pitch)) * distanceFromPlayer;
```

修改 lookat 函数，target 为人物模型头部坐标，使摄像机对准人物。

修改参数 Front 为由摄像机指向人物模型的单位向量。

```
// calculate the new Front vector
glm::vec3 front;
front = playerPos - Position;
Front = glm::normalize(front);
// also re-calculate the Right and Up vector
Right = glm::normalize(glm::cross(Front, WorldUp));
Up = glm::normalize(glm::cross(Right, Front));
```

由此得到第三人称摄像机。

人物俯角效果：

装 订 线

8.2. 人物转向移动

通过起始 Front 向量与当前 Front 向量进行余弦定理运算，得出两个向量的夹角的 cos 值，根据 acos()函数得出两个向量的夹角，即为人物模型应该旋转的角度。

```
// 人物转向
if (Front.x > 0) {
    playerRotate = acos(Front.z / sqrt(Front.x * Front.x + Front.z * Front.z)) * 180.0f / 3.14f;
}
else {
    playerRotate = acos(-Front.z / sqrt(Front.x * Front.x + Front.z * Front.z)) * 180.0f / 3.14f;
    playerRotate += 180.0f;
}
```

人物转身效果：

人物移动式进行根据时间进行模型切换，改变任务姿势，形成移动效果

装 订 线

```
// 人物移动动作
static int count = 0;
if (flag == 0) {
    count = 0;
    flag = 3;
}
else if (flag == 1) {
    count++;
    if (count >= 20) {
        flag = 3;
    }
}
else if (flag == 2) {
    count++;
    if (count >= 20) {
        flag = 4;
    }
}
else if (flag == 3) {
    count++;
    if (count >= 30) {
        flag = 4;
        count = 0;
    }
    else if (count >= 10 && count < 20) {
        flag = 1;
    }
}
```

人物移动效果：