Project report for the CG 100433 course

Team Name

Can Can World

Team Member

2051972 缪其隽
2052218 王书石
2052312 许志康
2052328 朱嘉策
2053758 朱旭堃

Project Title

仲夏夜

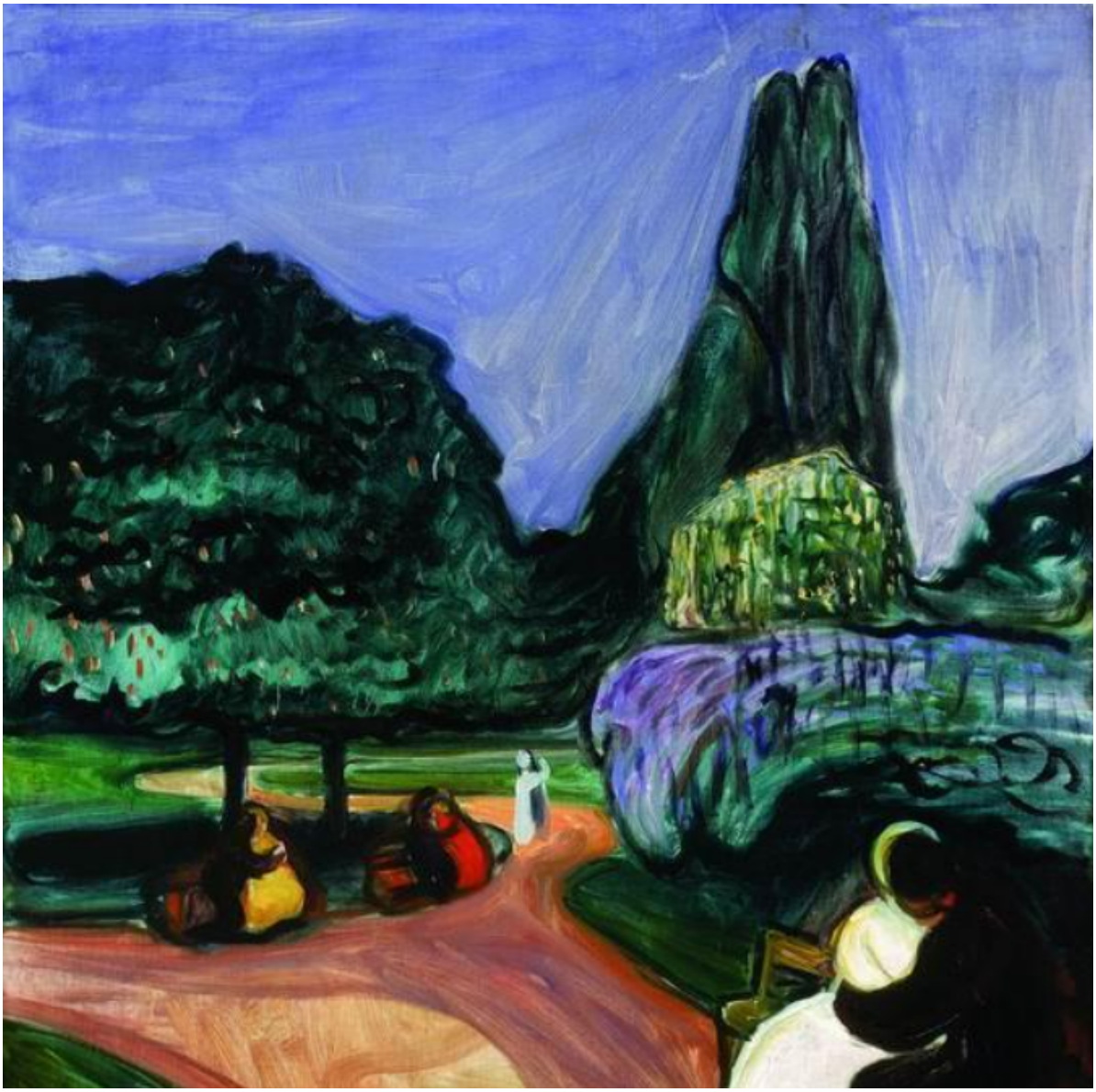
Abstract

我们的项目目标是使用以下相关技术，实现一个场景的真实感渲染：

- 基于物理的渲染
- 阴影渲染
- 基于SSR(Screen Space Reflection)的全局光照
- 体积光渲染
- 粒子系统模拟及渲染
- 海平面运动模拟及渲染

Motivation

小组成员从油画《仲夏夜》中得得到启发，试图实现夜晚户外场景，同时加入火焰、水体等元素，希望能通过该项目学习到真实感渲染以及模拟两大方面的知识。



The Goal of the Project

- 实现对场景中所有模型的PBR渲染，展现PBR技术对多种材质的表现力。
- 实现具有真实感的阴影渲染。
- 场景设置于夜晚，因此通过全局光照的实现来增加场景亮度。
- 实现体积光的渲染，来展示灯光通过窗户照入旧房屋内部的真实效果。
- 通过粒子系统的模拟，在场景中增加火焰与烟雾。
- 通过海平面运动的模拟，在场景中增加水体。
- 实现摄像机在场景中的移动和转移视角，便于测试与展示。

The Scope of the Project

- 不实现物体碰撞检测与交互
- 不实现光线追踪
- 实现对物体模型的PBR渲染
- 实现对水体、火焰、烟雾的模拟与渲染
- 天空使用立方体贴图，无法交互

Involved CG Techniques

PBR (Physically-Based Rendering)

基于物理的材质渲染和光照：通过使用金属度贴图、粗糙度贴图、漫反射遮蔽贴图、法线贴图、基本颜色贴图，使用Cook-Torrance反射率方程计算光照，综合以上贴图信息得出接近真实物理世界的效果。
反射率方程如下：
$$L_o(p, \omega_o) = \int_\Omega (k_d \frac{c}{\pi} + k_s \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)})L_i(p, \omega_i)n \cdot \omega_i\ \mathbf{d} \omega_i$$

PCSS (Percentage Closer Soft Shadow)

通过采样估算着色点与其遮挡物之间的距离，通过该距离计算PCF(Percentage Closer Filtering)滤波核的大小，最后对可见度（即阴影深度）进行滤波。滤波核大小计算如下：
$$w*{Penumbra}* = *frac{(d*{Receiver} - d*{Blocker}* *cdot w*{Light})}{d_{Blocker}}$$

SSR (Screen Space Reflection)

SSR技术通过屏幕空间的着色信息来计算一次反射的全局光照结果，其过程大致为如下三步：

- 采样：从着色点出发向外采样若干条光线。
- 求交：求出这些光线与场景中其他物体的交点，由于此处仅使用屏幕空间的深度信息，求出的交点为近似结果，也无法得到光线与屏幕之外的物体交点，优点是获得了更高的效率。
- 着色：以交点为光源，以交点的直接光照的着色结果作为光源的辐射率，对着色点进行着色，从而模拟了一次反射的间接光照。

体积光渲染

体积光来源于光线与空气中的介质发生作用从而进入人眼，以造成了“光路可见”的效果。具体实现思路如下：

- 采样：假设空气中的介质均匀分布，可以从摄像机位置到着色点位置连一条线，在该线段上均匀采样若干个代表介质的点。
- 计算：计算这些采样点上的介质与光线发生的作用，将这些计算结果通过一定的方式进行结合，得到最终着色值。

粒子系统模拟

粒子系统的模拟主要分为两个部分，分别是发射器（粒子产生）与仿真（模拟粒子运动过程）。

- 产生阶段主要确定粒子的初始属性，包括粒子的位置，粒子初速度，粒子颜色、粒子寿命等，需要根据不同的物理现象使用不同的统计学方法来进行模拟生成，以确保真实感。
- 模拟阶段需要每一帧更新粒子的所有属性，确保粒子属性是随时间动态改变的，从而达到一个动态图案的效果，同时每一帧更新也需要杀死到达寿命的粒子，补充新的粒子。具体的更新方法根据不同的物理现象使用不同的物理学公式即可。

海平面运动模拟

通过海洋统计学方法绘制海面，该过程可概括为两步。第一步，通过物理公式与经验公式，推导出海水的频谱；第二步，根据该频谱利用傅里叶逆变换得到高度。
频谱的计算通常需要引入菲利普频谱，同时结合海洋统计学公式，得到以下推导过程：

$$\tilde{h}(\vec{k},t) = \tilde{h}_0(\vec{k})e^{i\omega(k)t} + \tilde{h}_0^*(-\vec{k})e^{-i\omega(k)t}$$

$$\tilde{h}_0(\vec{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\vec{k})}$$

$$P_h(\vec{k}) = A\frac{e^{-1/(kL)^2}}{k^4}|\vec{k}\cdot\vec{w}|^2$$

从而得到了海面的频谱$\tilde{h}(\vec{k}, t)$
然后，利用傅里叶逆变换，将频谱从频域转换到时域，得到海面的高度：

$$h(x,t) = \sum_{\vec{k}} \tilde{h}(\vec{k},t)e^{i\vec{k}\cdot x}$$

Project Contents

- 实现了物体模型的PBR渲染，展示出不同物体材质在光照下表现出的真实感效果。
- 实现了点光源阴影，通过PCSS技术真实的展现了阴影离遮挡物远近变化导致不同的模糊度的效果。
- 实现了基于SSR的全局光照，在阴暗房屋内部可较明显的体现出亮度差异。
- 实现了体积光的渲染，在房屋内部可观察到通过窗户射入的光路。
- 实现了粒子系统的模拟和渲染，在场景中加入了运动的火焰与烟雾。
- 实现了海平面运动的模拟和渲染，在场景中加入了流动的水体。

Implementation

模型加载

使用模型导入库`Assimp`进行`.obj`格式模型的导入

地面模型生成

由于无法找到现成适合本项目的的地面模型，因此使用画高度图并转化为`.obj`文件的方式生成符合项目需要的地面模型，这样也可以使用`Assimp`库与其他物体模型进行统一的导入。

- 用绘制三角形的传统方法绘制一张指定大小指定采样数在xz平面上的网格，每个点的y值均初设为0，并计算所有索引关系；
- 制作一张地形高度图，通过bmp形式存储，以灰度值表示各点高度；
- 通过读文件方式读取bmp图种的数据，以采样点顺序遍历整个高度图，取得各个采样点的高度；
- 通过每个点所相邻的各个三角形的法向量的平均值来近似计算该点法向量；
- 将各点坐标、各三角形的索引以及各点法向量以obj文件格式输出；
- 用调用模型的统一方法调用地形obj文件，然后统一渲染。

在研究过程中，我们尝试了将高度图文件作为纹理传入shader中，并通过各采样点的纹理坐标读取高度图相应位置数值的方法，来获取每个采样点的高度，然后计算法向量，并进行渲染。该方法可行，但使用现行方法可以节省GPU的运算时间，提高效率。

PBR (Physically-Based Rendering)

在上述反射率方程中，可知关键在于计算 $D\ F\ G$ 三个函数的值，我们可以通过模型的多张参数贴图来计算：

- 反照率：反照率(Albedo)纹理为每一个金属的纹素(Texel)（纹理像素）指定表面颜色或者基础反射率。
- 法线：法线贴图使我们可以逐片段的指定独特的法线，来为表面制造出起伏不平的假象。

- 金属度：金属(Metallic)贴图逐个纹素的指定该纹素是不是金属质地的。根据PBR引擎设置的不同，美术师们既可以将金属度编写为灰度值又可以编写为1或0这样的二元值。
- 粗糙度：粗糙度(Roughness)贴图可以以纹素为单位指定某个表面有多粗糙。采样得来的粗糙度数值会影响一个表面的微平面统计学上的取向度。一个比较粗糙的表面会得到更宽阔更模糊的镜面反射（高光），而一个比较光滑的表面则会得到集中而清晰的镜面反射。
- AO：环境光遮蔽(Ambient Occlusion)贴图或者说AO贴图为表面和周围潜在的几何图形指定了一个额外的阴影因子。

法线分布函数D

目前有很多种NDF都可以从统计学上来估算微平面的总体取向度，我们用到的是Trowbridge-Reitz GGX，它需要着色点的粗糙度参数α：
$$NDF_{GGXTR}(n, h, \alpha) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2}$$
计算代码：

```
float DistributionGGX(vec3 N, vec3 H, float roughness)
{
    float a = roughness*roughness;
    float a2 = a*a;
    float NdotH = max(dot(N, H), 0.0);
    float NdotH2 = NdotH*NdotH;

    float nom   = a2;
    float denom = (NdotH2 * (a2 - 1.0) + 1.0);
    denom = PI * denom * denom;

    return nom / denom;
}
```

菲涅尔项F

菲涅尔方程描述的是被反射的光线对比光线被折射的部分所占的比率，这个比率会随着我们观察的角度不同而不同。当光线碰撞到一个表面的时候，菲涅尔方程会根据观察角度告诉我们被反射的光线所占的百分比。我们使用Fresnel-Schlick近似法求得近似解：
$$F_{Schlick}(h, v, F_0)=F_0+(1-F_0)(1-(h \cdot v))^5$$
计算代码：

```
vec3 fresnelSchlick(float cosTheta, vec3 F0)
{
    return F0 + (1.0 - F0) * pow(clamp(1.0 - cosTheta, 0.0, 1.0), 5.0);
}
```

几何函数G

与NDF类似，几何函数采用一个材料的粗糙度参数作为输入参数，粗糙度较高的表面其微平面间相互遮蔽的概率就越高。我们使用的几何函数是GGX与Schlick-Beckmann近似的结合体，因此又称为Schlick-GGX：
$$G{SchlickGGX}(n, v, k)=\frac{n \cdot v}{(n \cdot v)(1-k) + k}$$
这里$k=\frac{(\alpha+1)^2}{8}$
为了有效的估算几何部分，需要将观察方向和光线方向向量都考虑进去。我们使用史密斯法(Smith's method)来把两者都纳入其中：

$$G(n,v,l,k)=G{sub}(n,v,k)G_{sub}(n,l,k)$$
计算代码：

```
float GeometrySchlickGGX(float NdotV, float roughness)
{
    float r = (roughness + 1.0);
    float k = (r*r) / 8.0;

    float nom   = NdotV;
    float denom = NdotV * (1.0 - k) + k;

    return nom / denom;
}

float GeometrySmith(vec3 N, vec3 V, vec3 L, float roughness)
{
    float NdotV = max(dot(N, V), 0.0);
    float NdotL = max(dot(N, L), 0.0);
    float ggx2 = GeometrySchlickGGX(NdotV, roughness);
    float ggx1 = GeometrySchlickGGX(NdotL, roughness);

    return ggx1 * ggx2;
}
```

PCSS (Percentage Closer Soft Shadow)

PCSS的实现分为3步：

- 通过采样计算遮挡物的平均深度
- 通过平均深度计算PCF滤波核的大小
- 对可见度进行PCF滤波

部分代码：

```
float PCSS(int index, vec3 pos){
    float Stride = 50.;
    float shadowmapSize = 1024.;
    float visibility = 0.;
    int blockerNum = 0;
    float block_depth = 0.;

    vec3 fragToLight = pos - pointLights[index].position;
    float cur_depth = length(fragToLight);
    vec3 n = normalize(fragToLight), b1, b2;
    LocalBasis(n, b1, b2);
    mat3 localToWorld = mat3(n, b1, b2);
    poissonDiskSamples(pos.xy);
    for(int i = 0; i < SHADOWS_SAMPLES; i++)
        poissonDisk_3d[i] = localToWorld * vec3(0.0, poissonDisk[i]);

    // STEP 1: avgblocker depth
    for(int i = 0; i < SHADOWS_SAMPLES; i++) {
        float shadow_depth = texture(depthMap[index], fragToLight +
poissonDisk_3d[i] * Stride / shadowmapSize).r;
        shadow_depth *= far_plane;
```

```
        if(cur_depth > shadow_depth + EPS) {
            blockerNum++;
            block_depth += shadow_depth;
        }
    }
    if(blockerNum != 0) {
        block_depth /= float(blockerNum);
    }
    // STEP 2: penumbra size
    float w_penumbra = 2.;
    if(blockerNum != 0)
        w_penumbra *= (cur_depth - block_depth) / block_depth;
    // STEP 3: filtering
    Stride = 10.;
    for(int i = 0; i < SHADOWS_SAMPLES; i++) {
        float shadow_depth = texture(depthMap[index], fragToLight +
poissonDisk_3d[i] * Stride * w_penumbra / shadowmapSize).r;
        shadow_depth *= far_plane;
        float res = float(cur_depth < shadow_depth + EPS);
        visibility += res;
    }
    return visibility / float(SHADOWS_SAMPLES);
}
```

SSR (Screen Space Reflection)

SSR的实现分为如下几步：

- 初始化：创建帧缓冲用于存储直接光照的渲染结果，在其上添加一个颜色纹理附件（用于计算着色）以及一个深度纹理附件（用于计算求交）

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

glGenTextures(1, &textureColorbuffer);
glBindTexture(GL_TEXTURE_2D, textureColorbuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGB,
GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

glGenTextures(1, &depthBuffer);
glBindTexture(GL_TEXTURE_2D, depthBuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, SCR_WIDTH, SCR_HEIGHT, 0,
GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
textureColorbuffer, 0);
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D,
depthBuffer, 0);
```

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

- 在渲染循环中，首先将直接光照渲染结果存储到帧缓冲中

```
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
//Render...
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

- 将颜色纹理附件和深度纹理附件传入SSR_shader中，进行第二趟渲染（间接光照渲染）：

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textureColorbuffer);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, depthBuffer);
//Render...
```

- 间接光照渲染细节：对每个着色点进行上述采样、求交、着色三步，使用特定函数进行采样，着色细节与上述PBR一致，关键在于求交，使用`Ray Marching`的方法，从着色点出发向采样方向按照步长走一定的距离，每走一步通过深度信息来近似检验该点是否与场景中的其他物体相交（若该点深度大于深度纹理中存储的深度，则有物体挡在当前位置的前面，近似认为与该物体相交）。同时可以通过可变步长的方法提高效率：若该步未与物体相交，则在下一步增大步长。若该步与物体相交，则以更小的步长再走一次。

```
bool RayMarch(vec3 ori, vec3 dir, out vec3 hitPos) {
    int level = 0;
    vec3 delta = normalize(dir) * STEP_LONG;
    vec3 nowPos = ori;
    float total_dist = 0.0;

    while(true) {
        if(outScreen(nowPos))
            return false;
        vec3 nextPos = nowPos + delta;
        float curDepth = GetDepth(nextPos);
        float depth = GetGBufferDepth(GetScreenCoordinate(nextPos));
        if(depth - curDepth >= 1e-5) {
            total_dist += sqrt(dot(delta, delta));
            if(total_dist - MAX_DIST >= 1e-5)
                return false;
            nowPos = nextPos;
            level++;
            delta *= 2.0;
        }
        else {
            level--;
            delta /= 2.0;
            if(level < 0)
                break;
        }
    }
    hitPos = nowPos;
    return true;
}
```

在完成SSR的实现后，发现只需将其第一步的光线采样改为按照镜面反射射出一条光线，将其应用在水体的渲染上，便可以实现水面上的倒影效果（详见`water_fs.glsl`）。

体积光渲染

上文提到体积光的渲染分为采样与计算两步，因此主要介绍计算方法。
每个采样点上都需要计算5个参数，来自光源的光照亮度L，采样点到眼睛的透光率T，光线在视线方向上的散射值P，阴影值V，介质密度D。

- 光照亮度L：通过计算光照强度除以光已经传播的距离的平方实现，即$L=\frac{Intense}{Distance^2}$
- 透光率T：光在介质内内传播，会被吸收一部分，剩下的部分才能透过介质达到观察者眼中。我们使用Beer–Lambert法则对其进行估算。这个法则描述的是入射光强度和透光强度的比值：$Out = In \cdot e^{-c*d}$。其中c是物质密度，d是距离。
- 散射值P：光的散射是向四面八方的，在一个以尘埃为球心的球体中几乎所有方向都有可能反射到光线，而且每个方向散射出去的光线亮度应该是不一样的，而这些散射出去的光线亮度总合应该和射到尘埃上的那束光线亮度一样，即满足能量守恒。不同散射方向得到的光线亮度可以通过HG公式计算：
  $$p(\theta)=\frac{1}{4\pi}\frac{1-g^2}{(1+g^2-2gcos\theta)^{3/2}}$$
  其中g值为代表介质的散射性质的参数。
- 阴影值V：即光源照到介质的位置的光路中间有没有其他遮挡物，与阴影实现算法类似，不再赘述。由于这不是计算可见的阴影，因此不需要采用高耗时的与采样有关的算法（PCF/PCSS），计算硬阴影即可。
- 介质密度D：给定一个参数即可。在本项目中，由于场景实际情况，只在旧房屋内实现体积光（旧房屋内有尘埃，也就是介质），因此在具体的代码实现上，会判断采样点是否位于房屋内，若在房屋外，则置介质密度为0。

在对每个着色点进行完上述参数的计算后，则可以相乘得到该采样点的贡献，将各采样点的贡献叠加即可得到最终着色结果：

```
vec4 volumeScattering(vec3 rO, vec3 finalPos)
{
    const float lightIntense = 30.;

    float transmittance = 1.0;
    vec3 scatteredLight = vec3(0.0, 0.0, 0.0);
    float step_long = distance(finalPos, rO) / SCATTER_SAMPLES;

    vec3 rD = normalize(finalPos - rO) * step_long;
    vec3 nowPos = rO + rand_2to1(finalPos.xy) * rD;
    while(true) {
        if(dot(nowPos - rO, nowPos - rO) - dot(finalPos - rO, finalPos - rO) >
EPS)
            break;
        float vLight = lightIntense / dot(nowPos - lightPos, nowPos - lightPos);
        float D = evaluateDensity(nowPos);
        scatteredLight += D * vLight * phaseFunction(rO, lightPos, nowPos) *
transmittance * step_long * visibility(nowPos);
        transmittance *= exp(-D * step_long);
        nowPos += rD;
    }
    return vec4(scatteredLight, transmittance);
}
```

粒子系统模拟

火焰

首先确定火焰粒子的具体属性，在本次实现内，火焰粒子的属性包括位置、速度、颜色、寿命、年龄、大小。

产生阶段：

- **粒子位置初始化**
  实际的火焰燃烧通常为焰心部分火焰旺盛，向外逐渐减弱，这个特性使用粒子分布来描述就是中心区域粒子密度大，向外逐渐稀疏，在数学上呈正态分步，所以假设火焰发射平面为$Y=0$，粒子初始位置的赋值操作如下：
  $$X = \frac{1}{\sqrt{2 πr}} \times e^{-\frac{(X_i - X_0)^2}{2r^2}}$$$$Z = \frac{1}{\sqrt{2 πr}} \times e^{-\frac{(Z_i - Z_0)^2}{2r^2}}$$上式对于计算以及编程实现较为复杂，所以项目中利用叠加过程初始化模拟粒子位置，可以实现类似正态分布的效果：
  $$X = \sum*{i=0}^{n}Rand() * Adj_value$$$$Z = \sum*{i=0}^{n}Rand() * Adj_value$$$Rand()$为$[-1, 1]$的随机数，$Adj_value$为调整因子，更改这个参数，可以调整火焰外观。
- **粒子速度初始化**
  保证火焰的局部随机性，又要维持火焰的整体外观，初速度计算如下：
  $$Init_V = (MAX_V - MIN_V) * rand() + MIN_V$$可以自行调整最大初速度和最小初速度来调整火焰高度以及变化速度。
- **粒子寿命、年龄初始化**
  火焰中心粒子寿命长，边缘粒子寿命短，所以在初始化时粒子寿命与到火焰中心的距离有关，具体计算方式如下：
  $$Life = (MAX_LIFE - MIN_LIFE) * rand() + MIN_LIFE$$如果粒子到中心位置小于火焰半径，则$Life *= 1.3$即可，而所有粒子初始化时刚刚出生，所以$Age = 0$
  调整粒子寿命，可以改变火焰高度和外观
- **粒子大小、颜色初始化**
  $$Size = INIT_SIZE$$$$Color = INIT_COLOR$$给出一个初始化大小和颜色即可，模拟过程中更改粒子大小和颜色透明度。

模拟阶段：

- **位置、速度更新**
  火焰粒子运动阶段，粒子做沿Y轴向上的匀加速运动,所以粒子的位置和速度更新是较为简单的，$Pos += v \times dt$, $v += acc \times dt$。
- **颜色、大小更新**
  火焰在燃烧过程中明暗、大小变化也是近似符合动态分布的，一开始火焰粒子较小，然后逐渐增大，当到达一定时间后，它又逐渐变小至消失，粒子大小更新方法如下：
  $$factor = \frac{1.0}{(Age - Life / 2)^2 + 1.0}$$$$Size = factor \times INIT_SIZE$$而对于火焰的明暗变化，则是修改火焰粒子颜色中的透明度，来形成火焰中心亮，一定距离外逐渐变暗至消失，$$Color.a = factor$$
- **年龄、粒子更新**
  对于粒子的年龄，每一帧做$Age += dt$操作即可，重点是当$Age > Life$时，该粒子就已经死亡了，需要产生新的粒子，这个操作采用如下代码完成：

```
for (GLuint i = 0; i < unsigned int(dt * 1000); i++)
{
    int unusedParticle = firstUnusedParticle();
    respawnParticle(particles[unusedParticle]);
}
```

在每一次更新中利用`firstUnusedParticle`函数寻找死掉的粒子，再调用`respawnParticle`函数将其“复活”（即重新初始化粒子属性），防止粒子数量不断增加。

渲染阶段：

- **顶点着色器部分**
  接收粒子属性即可，包括粒子的位置、大小以及颜色，将其传入下一级着色器
- **几何着色器部分（billboard技术）**
  在一般情况下，我们在书写OpenGL代码是不需要使用几何着色器，但是在粒子的渲染过程中，我们实际上是根据粒子位置渲染一个四边形来模拟一个火焰粒子，以减少渲染时的开销。但是渲染四边形就会有一个问题，根据观察者所在的位置不同，所看到的四边形状态也是不同的，比如看到四边形的正面或者看到四边形的边，会很大程度影响火焰的真实性。
  所以为了解决这个问题，使用billboard技术，使得观察者无论在什么方向观察四边形，永远使用四边形的正面朝向观察者，这个操作需要在几何着色器中完成。

```
vec3 Pos = gl_in[0].gl_Position.xyz;

vec3 toCamera = normalize(gCameraPos - Pos);

vec3 up = vec3(0.0, 1.0, 0.0);

vec3 right = cross(toCamera, up) * gs_in[0].ourScale;
```

具体操作如上述代码，根据顶点着色器传入的坐标以及通过`uniform`形式传入的摄像机位置，计算出始终朝向观察者的边的向量

```
Pos -= right;

gl_Position = projection * view * model * vec4(Pos, 1.0);

TexCoord = vec2(0.0, 0.0);

EmitVertex();
```

再使用上述代码，将四边形的四个顶点依次传入片段着色器即可。

- **片段着色器部分**
  负责四边形颜色、纹理的输出，代码如下：

```
void main()

{
    vec4 texColor;
    if(fchoice > 0.5)
        texColor = texture(flame, TexCoord) * fColor;
    else
        texColor = texture(Round, TexCoord) * fColor;

    if(texColor.r + texColor.g + texColor.b < 0.2)
        discard;
    FragColor = texColor;
}
```

- **混合（Blending)**
  在之前的颜色变化中，只修改了颜色变量的透明度，修改后若要展示出效果则需要用到OpenGL中的混合技术。
  设置透明度，则可以使粒子的颜色由其本身颜色和背后物体颜色按比例混合而成，混合函数的设置：

  ```
  glBlendFunc(GL_SRC_ALPHA, GL_ONE);
  ```

  从而真正火焰颜色从焰心至四周的变化。

烟

首先确定烟粒子的具体属性，在本次实现内，烟粒子的属性包括位置、速度、颜色、寿命、年龄、大小。
产生阶段：

- **粒子位置初始化**
  在本项目中，烟的位置置于烟囱上，所以可以看作是从一个圆形范围内产生，又由于烟的分布不像火那样很明显地服从正态分布，所以烟粒子的初始位置直接随机到圆内任意位置即可：
  $$X = R \times rand()\times cos(Angle) $$$$Z = R \times rand()\times sin(Angle) $$其中$rand, Angle$为$[0,1]$和$[0, 2π]$的随机值
- **粒子速度初始化**
  由于常见的烟一般是具有发散效果的，即向上运动的过程中会渐渐发散，所以在此处初速度的定义上也给每个粒子一个向外发散的速度，而对于粒子的竖直上升，在位置变化时引入，不通过速度的初始化完成。
  $$azimuthAngle = rand()\ / \ RAND_MAX$$$$azimuthAngle \ *= maxEmissionAngle$$$$polarAngle \ *= 2π$$$$V_x = sin(azimuthAngle) * cos(polarAngle)$$$$V_z = sin(azimuthAngle) * sin(polarAngle)$$$$V_y = cos(azimuthAngle)$$其中$maxEmissionAngle$限制了粒子的发散程度，其值越大，发散效果越趋近于一个球体。
- **粒子寿命、年龄、颜色初始化**
  粒子寿命赋予其随机值即可，年龄和颜色以及粒子大小赋初值。

模拟阶段：

- **粒子位置更新**
  $$Pos += v \times dt + 0.5 \times gravity \times Age^2$$这样在位置更新处引入竖直方向上的变化效果，便可以做到烟的上升和发散效果的结合。
- **粒子大小更新**
  $$ageFactor = 1.0 - (Age/Life)$$$$ageFactor = clamp(ageFactor, 0.0, 1.0)$$$$Size = ageFactor \times INIT_SIZE$$可以做到随着烟的上升逐渐稀疏的效果。
- 粒子年龄、寿命更新
  此处的操作与火焰粒子相同，不再赘述

渲染阶段：
由于烟不如火焰聚集，常常表现为发散、稀疏的一个形式，所以不需要使用四边形来渲染一个烟的粒子，直接以DrawPoints的形式即可，那么它的渲染就比较简单。
顶点着色器直接设置点的位置和大小

```
float scale = ageFactor * 32.0f;
gl_Position = projection * view * model * vec4(Position, 1.0);
gl_PointSize = scale;
```

片段着色器上输出纹理、颜色即可

```
texCoords = vec2(gl_PointCoord.x, gl_PointCoord.y);
texColor = texture2D(texture, texCoords);
gl_FragColor = vec4(texColor.r, texColor.r, texColor.r, 0.08) * v_color;
```

海平面运动模拟

N-S 方程，**纳维 - 斯托克斯方程**是流体力学中描述**粘性牛顿流体**的方程，表述了**粘性不可压缩流体**的**动量守恒**。该方程目前大约只有一百多个**特解**被解出来，是**最复杂的微分方程**之一：

$$\rho\frac{Du}{Dt} = F - \nabla p + \mu\nabla u^2$$

流体连续性方程，描述了流体的质量守恒：

$$\frac{\partial\rho}{\partial t} + \nabla\cdot(\rho u) = 0$$

线性化后，即假设水平面为 $x⊥$ ，*即 $x⊥=(x,z)$* ，且 $x=(x_⊥,y)$ ，其中 y **垂直于海平面向下**，**质量守恒方程**可转化为：

$$(\nabla_\perp^2 + \frac{\partial^2}{\partial y^2})\phi(x_\perp, t) = 0$$

假设 u 很小，且海平面压力为0，**动量守恒**可转化为：

$$\frac{\partial\phi(x_\perp, t)}{\partial t} = -gh(x_\perp, t)$$

现在只考虑**垂直于水平面的流体速度**，即在 y **方向的速度**，再假设流体在 x **平面的速度为0**，最后根据速度势公式，并联立上述两方程，可得最后方程为：

$$\frac{\partial^4 h(x_\perp, t)}{\partial t^4} = g^2\nabla_\perp^2 h(x_\perp, t)$$

根据**平面波理论**，我们可以假设

$$h(x_\perp, t) = h_0 e^{ik\cdot x_\perp - i\omega t}$$

代入可得

$$h(x, t) = h_0 e^{-i(\sqrt{gk}t - k\cdot x)}$$

考虑到$h(x,t)$的周期性，使用$(N×N,M×M)$网格度量x平面，那么可以使用**傅里叶变换**来表示高度场

$$h(x, t) = \sum_{\mathbf{k}} \tilde{h}(\mathbf{k}, t)e^{i\mathbf{k}\cdot x}$$

这明显是$h(x,t)$的**离散傅里叶逆变换 (IDFT)**，其中 $\tilde{h}(\mathbf{k}, t)=\tilde{h*{0}}(\mathbf{k}) e^{i \omega(\mathbf{k}) t}+{\tilde{h}{0}}}^{*}(-k) e^{i \omega(\mathbf{k}) t}$ 为高度频谱，$\omega(\mathbf{k})=\sqrt{g\mathbf{k}}$为色散频率，$g$为重力加速度；

引入**方向波谱 (菲利普频谱)**

$$P_h(\mathbf{k}) = S(\omega, \theta) = S(\omega)D(\omega, \theta)$$

其中，$ S(\omega) $ 为非定向波谱， $ D(\omega, \theta) $ 为方向扩展函数， ω 为角频率，$\theta $ 为波矢量与风向的夹角

对于非定向波谱，可以使用海洋统计学的经验公式 (现有理论无法求解):

$$S(\omega) = \frac{A}{k^4} e^{-\frac{1}{k^2 l^2}}$$

同样，扩展函数取

$$D(\omega, \theta) = |\mathbf{k} \cdot \mathbf{w}|^2$$

那么定向波谱可表示为

$$P_h(\vec{k}) = A \frac{e^{-1/(kL)^2}}{k^4} |\vec{k} \cdot \vec{w}|^2$$

```
float cOcean::phillips(int n_prime, int m_prime) {
    glm::vec2 k(M_PI * (2 * n_prime - N) / length, M_PI * (2 * m_prime - N) /
length);
    float k_length = glm::length(k);
    if (k_length < 0.000001) return 0.0;

    float k_length2 = k_length * k_length;
    float k_length4 = k_length2 * k_length2;

    float k_dot_w = glm::dot(glm::normalize(k), glm::normalize(w));
    float k_dot_w2 = k_dot_w * k_dot_w;

    float w_length = glm::length(w);
    float L = w_length * w_length / g;
    float L2 = L * L;

    float damping = 0.001;
    float l2 = L2 * damping * damping;

    return A * exp(-1.0f / (k_length2 * L2)) / k_length4 * k_dot_w2 * exp(-
k_length2 * l2);
}
```

其中$\mathbf{w}$为风向，$l=\frac{v^2}{g}$，v为风速，A是菲利普频谱参数(影响波高)

引入两个正态分布随机数

$$\xi_x, \xi_i \sim N(0,1) = \frac{1}{\sqrt{2\pi}} e^{-\frac{(x,i)^2}{2}}$$

```
float uniformRandomVariable() {
    return (float)rand() / RAND_MAX;
}

std::complex<float> gaussianRandomVariable() {
    float x1, x2, w;
    do {
        x1 = 2.f * uniformRandomVariable() - 1.f;
        x2 = 2.f * uniformRandomVariable() - 1.f;
        w = x1 * x1 + x2 * x2;
    } while (w >= 1.f);
```

```
    w = sqrt((-2.f * log(w)) / w);
    return std::complex<float>(x1 * w, x2 * w);
}
```

根据概率统计学可得

$$\tilde{h}_0(\vec{k}) = \frac{1}{\sqrt{2}}(\xi_r + i\xi_i)\sqrt{P_h(\vec{k})}$$

最终得到高度函数的近似解

$$\tilde{h}(\vec{k}, t) = \tilde{h}_0(\vec{k})e^{i\omega(k)t} + \tilde{h}_0^*(-\vec{k})e^{-i\omega(k)t}$$

```
std::complex<float> cOcean::hTilde_0(int n_prime, int m_prime) {
    std::complex<float> r = gaussianRandomVariable();
    return r * sqrt(phillips(n_prime, m_prime) / 2.0f);
}

std::complex<float> cOcean::hTilde(float t, int n_prime, int m_prime) {
    int index = m_prime * Nplus1 + n_prime;

    std::complex<float> htilde0(vertices[index].a, vertices[index].b);
    std::complex<float> htilde0mkconj(vertices[index]._a, vertices[index]._b);

    float omegat = dispersion(n_prime, m_prime) * t;

    float cos_ = cos(omegat);
    float sin_ = sin(omegat);

    std::complex<float> c0(cos_, sin_);
    std::complex<float> c1(cos_, -sin_);

    std::complex<float> res = htilde0 * c0 + htilde0mkconj * c1;

    return res;
}
```

由于我们使用IDFT暴力求解的话效率过低，只能支持极低的采样数，所以我们可通过**快速傅里叶逆变换IFFT**以进行**海洋运动的数字模拟**，得到每个采样点的坐标和纹理坐标；每个点的法向量可通过该点相邻三角形的法向量平均值来近似计算；

```
void cOcean::evaluateWavesFFT(float t) {
    float kx, kz, len, lambda = -1.0f;
    int index, index1;
    for (int m_prime = 0; m_prime < N; m_prime++) {
        kz = M_PI * (2.0f * m_prime - N) / length;
        for (int n_prime = 0; n_prime < N; n_prime++) {
            kx = M_PI * (2 * n_prime - N) / length;
            len = sqrt(kx * kx + kz * kz);
            index = m_prime * N + n_prime;

            h_tilde[index] = hTilde(t, n_prime, m_prime);
            h_tilde_slopex[index] = h_tilde[index] * std::complex<float>(0, kx);
```

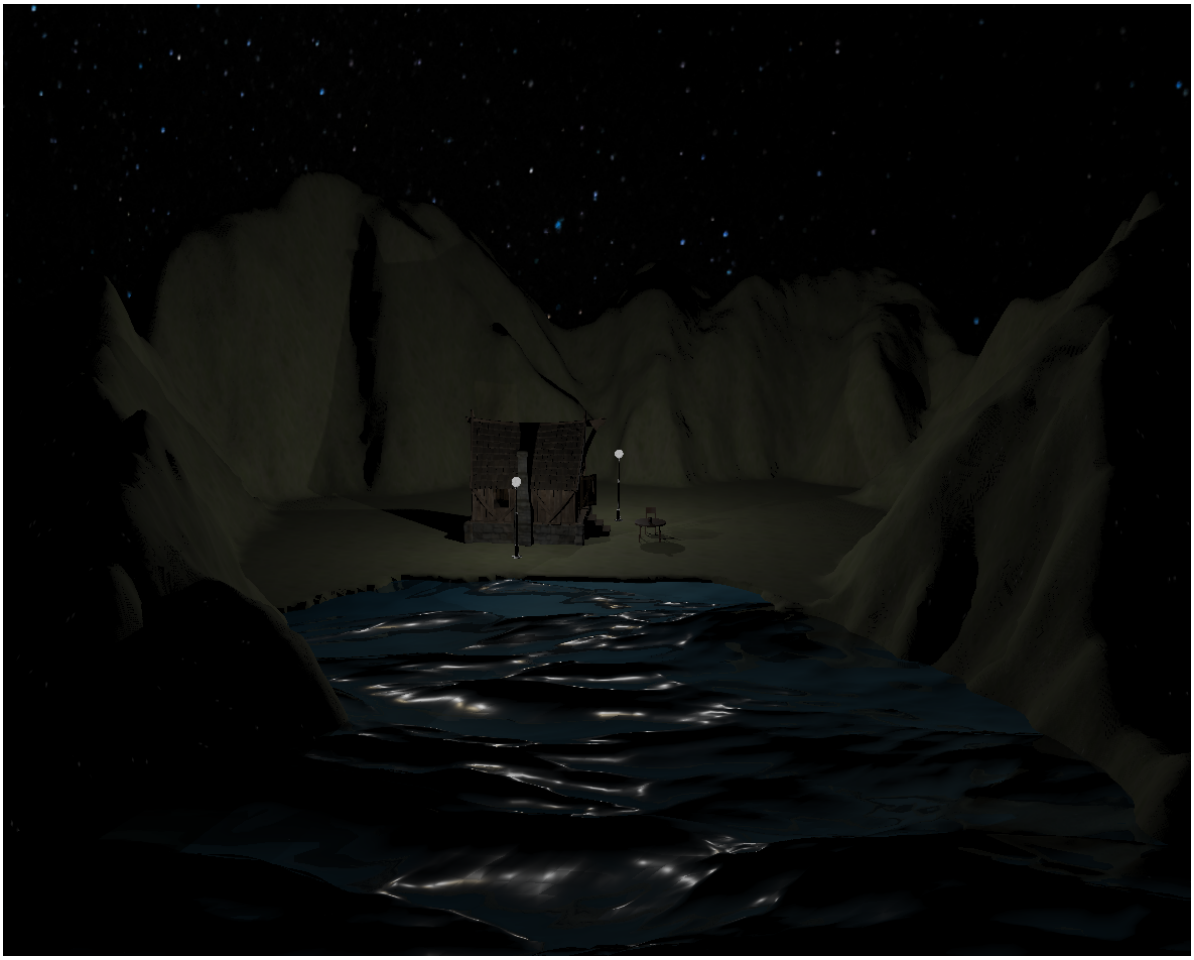
```
            h_tilde_slopez[index] = h_tilde[index] * std::complex<float>(0, kz);
            if (len < 0.000001f) {
                h_tilde_dx[index] = std::complex<float>(0.0f, 0.0f);
                h_tilde_dz[index] = std::complex<float>(0.0f, 0.0f);
            }
            else {
                h_tilde_dx[index] = h_tilde[index] * std::complex<float>(0, -kx
/ len);
                h_tilde_dz[index] = h_tilde[index] * std::complex<float>(0, -kz
/ len);
            }
        }
    }
    for (int m_prime = 0; m_prime < N; m_prime++) {
        fft->fft(h_tilde, 1, m_prime * N);
        fft->fft(h_tilde_slopex, 1, m_prime * N);
        fft->fft(h_tilde_slopez, 1, m_prime * N);
        fft->fft(h_tilde_dx, 1, m_prime * N);
        fft->fft(h_tilde_dz, 1, m_prime * N);
    }
    for (int n_prime = 0; n_prime < N; n_prime++) {
        fft->fft(h_tilde, N, n_prime);
        fft->fft(h_tilde_slopex, N, n_prime);
        fft->fft(h_tilde_slopez, N, n_prime);
        fft->fft(h_tilde_dx, N, n_prime);
        fft->fft(h_tilde_dz, N, n_prime);
    }
    int sign;
    float signs[] = { 1.0f, -1.0f };
    vector3 n;
    for (int m_prime = 0; m_prime < N; m_prime++) {
        for (int n_prime = 0; n_prime < N; n_prime++) {
            index = m_prime * N + n_prime;     // index into h_tilde..
            index1 = m_prime * Nplus1 + n_prime;    // index into vertices
            vertices[index1].nx = 0;
            vertices[index1].ny = 0;
            vertices[index1].nz = 0;
            sign = signs[(n_prime + m_prime) & 1];
            h_tilde[index] = h_tilde[index] * (float)sign;
            // height
            vertices[index1].y = stride * h_tilde[index].real();
            // displacement
            h_tilde_dx[index] = h_tilde_dx[index] * (float)sign;
            h_tilde_dz[index] = h_tilde_dz[index] * (float)sign;
            vertices[index1].x = stride * (vertices[index1].ox +
h_tilde_dx[index].real() * lambda);
            vertices[index1].z = stride * (vertices[index1].oz +
h_tilde_dz[index].real() * lambda);
            // normal
            h_tilde_slopex[index] = h_tilde_slopex[index] * (float)sign;
            h_tilde_slopez[index] = h_tilde_slopez[index] * (float)sign;
            n = glm::normalize(vector3(0.0f - h_tilde_slopex[index].real(),
1.0f, 0.0f - normalize(h_tilde_slopez[index].real())));

            // for tiling
```

```
            if (n_prime == 0 && m_prime == 0) {
                vertices[index1 + N + Nplus1 * N].y = stride *
h_tilde[index].real();
                vertices[index1 + N + Nplus1 * N].x = stride * (vertices[index1
+ N + Nplus1 * N].ox + h_tilde_dx[index].real() * lambda);
                vertices[index1 + N + Nplus1 * N].z = stride * (vertices[index1
+ N + Nplus1 * N].oz + h_tilde_dz[index].real() * lambda);
            }
            if (n_prime == 0) {
                vertices[index1 + N].y = stride * h_tilde[index].real();
                vertices[index1 + N].x = stride * (vertices[index1 + N].ox +
h_tilde_dx[index].real() * lambda);
                vertices[index1 + N].z = stride * (vertices[index1 + N].oz +
h_tilde_dz[index].real() * lambda);
            }
            if (m_prime == 0) {
                vertices[index1 + Nplus1 * N].y = stride *
h_tilde[index].real();
                vertices[index1 + Nplus1 * N].x = stride * (vertices[index1 +
Nplus1 * N].ox + h_tilde_dx[index].real() * lambda);
                vertices[index1 + Nplus1 * N].z = stride * (vertices[index1 +
Nplus1 * N].oz + h_tilde_dz[index].real() * lambda);
            }
        }
    }
    for (int i = 0; i < N * N * 2; i++)
    {
        unsigned int pIndex1 = indices[i * 3];
        unsigned int pIndex2 = indices[i * 3 + 1];
        unsigned int pIndex3 = indices[i * 3 + 2];
        float x1 = vertices[pIndex1].x;
        float y1 = vertices[pIndex1].y;
        float z1 = vertices[pIndex1].z;
        float x2 = vertices[pIndex2].x;
        float y2 = vertices[pIndex2].y;
        float z2 = vertices[pIndex2].z;
        float x3 = vertices[pIndex3].x;
        float y3 = vertices[pIndex3].y;
        float z3 = vertices[pIndex3].z;
        //求边
        float vx1 = x2 - x1;
        float vy1 = y2 - y1;
        float vz1 = z2 - z1;
        float vx2 = x3 - x1;
        float vy2 = y3 - y1;
        float vz2 = z3 - z1;
        //叉乘求三角形法线
        float xN = vy1 * vz2 - vz1 * vy2;
        float yN = vz1 * vx2 - vx1 * vz2;
        float zN = vx1 * vy2 - vy1 * vx2;
        float Length = sqrtf(xN * xN + yN * yN + zN * zN);
        xN /= Length;
        yN /= Length;
        zN /= Length;
        //顶点法线更新
```

```
            vertices[pIndex1].nx += xN;
            vertices[pIndex1].ny += yN;
            vertices[pIndex1].nz += zN;
            vertices[pIndex2].nx += xN;
            vertices[pIndex2].ny += yN;
            vertices[pIndex2].nz += zN;
            vertices[pIndex3].nx += xN;
            vertices[pIndex3].ny += yN;
            vertices[pIndex3].nz += zN;
        }
    }
```

Results

场景总览



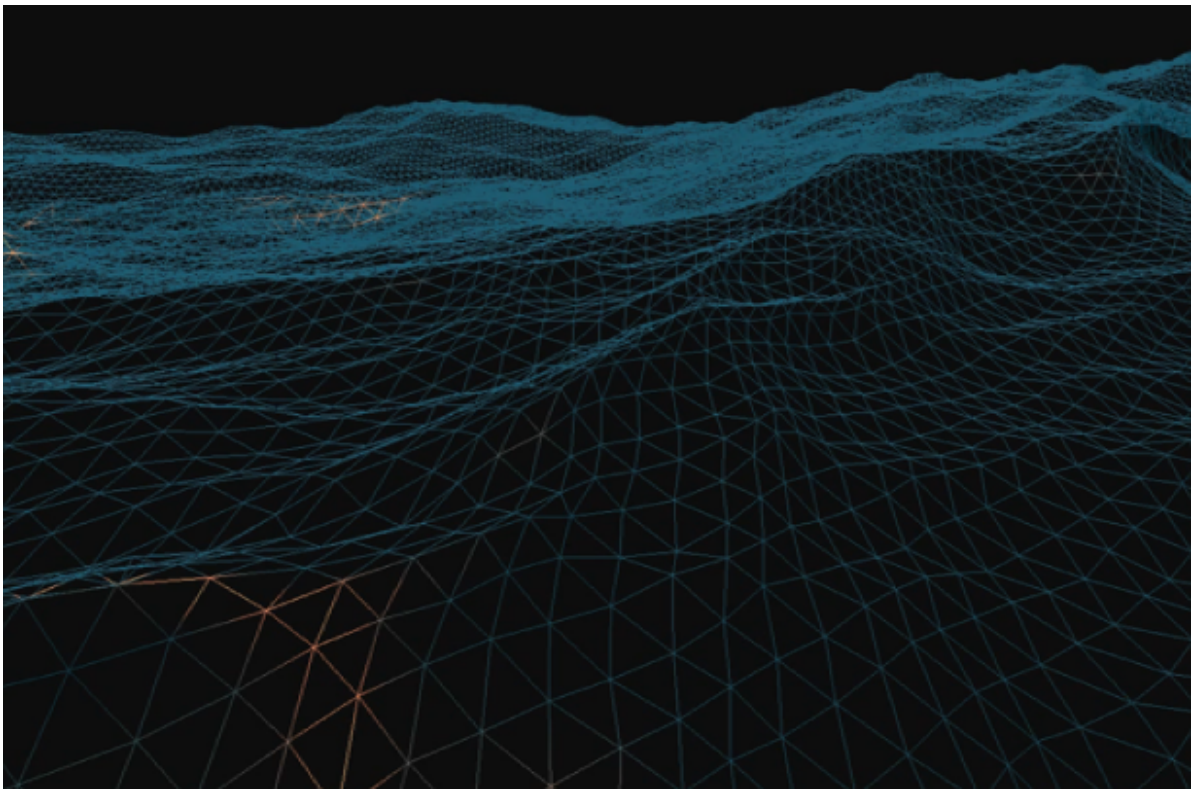
PBR在不同材质物体上的渲染效果

PCSS阴影效果

体积光效果

烟、火焰效果

水体效果



基于SSR在水面上的倒影效果

Roles in Group

缪其隽 (100%)：PBR渲染、阴影渲染、基于SSR的全局光照、体积光渲染
王书石 (100%)：粒子系统模拟、火焰与烟的渲染
许志康 (100%)：粒子系统模拟、火焰与烟的渲染
朱嘉策 (100%)：地面模型的生成、海平面运动模拟与渲染
朱旭堃 (100%)：地面模型的生成、海平面运动模拟与渲染

References

[1] https://zhuanlan.zhihu.com/p/21425792
[2] https://blog.csdn.net/qq_31615919/article/details/78968434
[3] https://zhuanlan.zhihu.com/p/150570892
[4] Roberto Lopez Mendez. Particle System: Realtime smoke rendering with OpenGL® ES 2.0.
[5] https://www.keithlantz.net/2011/10/ocean-simulation-part-one-using-the-discrete-fourier-transform/
[6] https://blog.csdn.net/qq_39300235/article/details/103582460
[7] https://zhuanlan.zhihu.com/p/64414956