Project Title

Rendering Complex Scenes

Team Member

朱昀玮，晋愉翔，黎可杰，陈开煦，张智淋，雒俊为

Abstract

在本项目中，我们小组重点关注计算机图形学中的渲染技术，并通过一个兼具自然环境和人工房屋的场景展现我们实现的技术。我们实现的技术主要包括大气、云、地形、草地、海洋等自然环境的渲染技术，与阴影，PBR 等提升渲染质量表现的渲染技术。

在关注渲染质量的同时，我们力图构建一个高性能，可扩展，易于使用的框架，使得场景导入更加方便，渲染流程更加清晰，便于进一步开发。为了易于使用的特性，我们引入 Component-Object 思想，将场景内的物体视作多种组件的组合，将项目组织成模块的形式开发，并且使用 json 作为物体配置文件，从配置文件中导入和创建物体。为了高性能的实时渲染，我们在项目中大量使用 Compute Shader 加速通用计算，并且使用 SSBO 和 UBO 等 buffer 减少 CPU 与 GPU 之间的数据沟通，提升了整体效率。

Motivation

在定题会议中，我们小组认为专注于渲染技术的研究更加符合我们的兴趣。为此，我们制定了一系列目标，从生活中最常见到的自然环境开始，包括天空、云、地形和海洋。同时，我们希望尝试一些与全局光照和阴影相关的技术，提升阴影质量表现，因此我们在目标中加入了 CSM、PCSS 等阴影技术与 RSM 全局光照技术。

为了项目开发的方便，我们引入了 Object-Component 的项目组织形式。

The Goal of the project

本项目的目标是实现一些自然渲染与 PTR 渲染技术，其中自然环境渲染技术目标为：基于物理的天空、程序化生成的地面、海洋、体积云。RTR 渲染技术为：PCSS+CSM 阴影，RSM 全局光照，PBR 材质，HDR 渲染，延迟渲染管线。

The Scope of the project

本项目的范围划分在实时渲染内，只关注实时渲染技术的开发。本项目将不会成为一个游戏，即没有游戏玩法上的设计，而只是关注渲染技术本身，不会使用到如物理引擎，角色动画等游戏上用到的渲染技术。但同时，我们希望场景是实时查看的，可以供使用者在场景中自由移动观察。

Involved CG techniques & Implementation

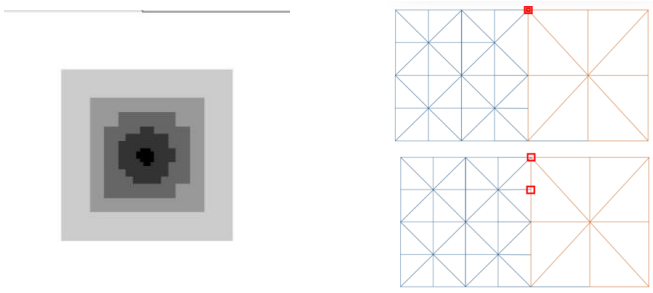
本项目中使用到的 CG 技术主要分为自然环境渲染与其他 RTR 技术。

1. Nature Rendering

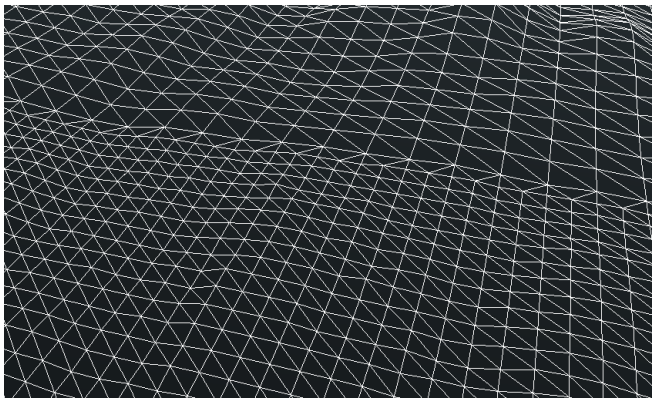
1) Terrain

程序化地形生成的方法主要使用 Farcry 5 制作团队在 2018 年 GDC 大会上展示的方法，即使用四叉树 Lod 的方式在 GPU 上动态生成地形，以便支持超大地形的快速生成。整个生成的 pipeline 全在 GPU 上完成，因此效率极高。

该算法基于高度图生成地形，有以下四个步骤：使用 compute shader 计算 Lod 等级，该步骤使用两个 node list 辅助遍历四叉树；从第一步中构建一个 lod map；实际生成 mesh；不同 lod 等级之间的 mesh 缝补。



图表 1 左图 lodMap，右图 mesh 缝补



图表 2 mesh 生成结果

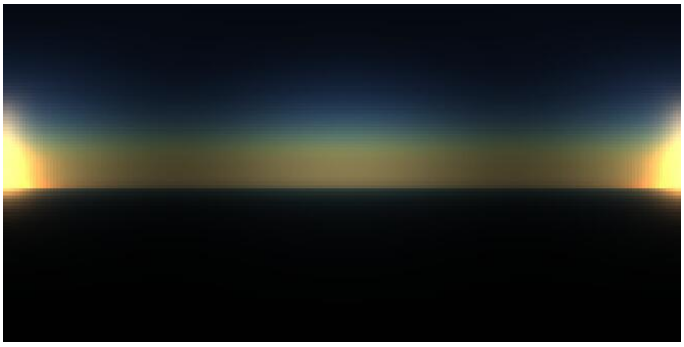
2) 草地生成

草叶为一个三角形，生成方式为在 GPU 上根据位置随机做旋转和平移，并将这个变换对应的矩阵存储在 SSBO 中。在实际渲染中，使用 InstanceID 索引 SSBO 中的矩阵，完成草地渲染。

3) 基于物理的天空

基于物理的天空渲染需要考虑：Mie 散射，Rayleigh 散射，Ozone 层的吸收。本项目中使用的方法为 A scalable and Production Ready Sky and Atmosphere Rendering Technique 论文中提出的方法。该算法有两个步骤：计算 transmittance lut；计算 skyview lut；

图表 3 transmittance Lut



图表 4 skyview lut

最后，我们将基于物理的天空和一个星空的 skybox 基于亮度融合，达成了夜晚有星空的效果。

4) Ocean

基本原理:由于直接进行波形的叠加时间成本较高，于是采用算出频谱，通过 IFFT 求出波形，此次使用的为 Jerry Tessendorf 在 Simulating Ocean Water 中所提到的海洋公式。

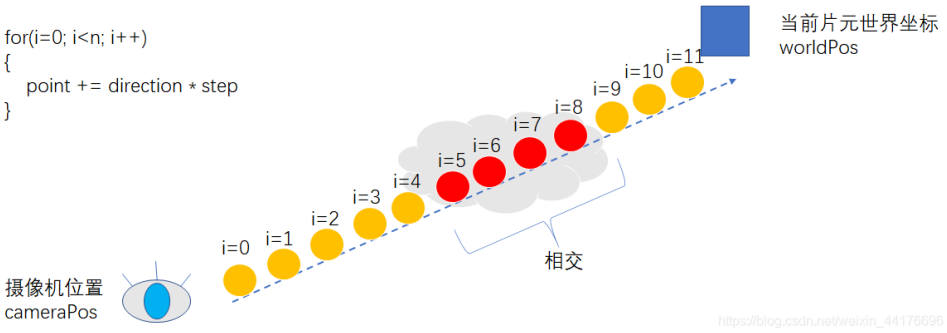
IFFT 海面：用海洋统计学公式生成 Phillips 频谱，计算高度频谱和水平偏移频谱，经快速傅里叶逆变换（IFFT）得到偏移图，计算出 Gerstner 波叠加的海面。

并行性（parallel）：采用 compute shader 来并行化海面 IFFT 以及相关频谱的计算，使得允许大规模且更多波长的细节计算。

混合（Blending）：使用混合来实现半透明的海面。

5) Volumetric Cloud

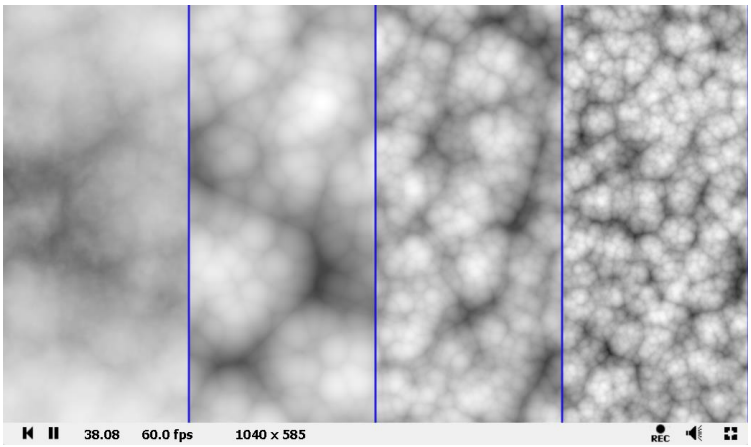
体积云（Volumetric Cloud），是用图像引擎来模拟现实世界的云的半透明、无规则的表现效果。实现体积云的最常见的技术方法是 Ray Marching，在每一步累积光照和密度。其基本原理示意图如下所示：



图表 5 Ray Marching

Ray Marching 与 Ray Tracing 类似，均从摄像机位置处射出一条 ray，不同之处在于 Ray Marching 每步进一个 step 就会停下来计算当前点，再继续前进。绘制体积云时，将从摄像机向世界空间投射光线并逐步步进，当与云层的范围（一个长方体）相交时，判定为穿过了云层，对应投影到屏幕的像素点处需要绘制云的颜色。而云的颜色根据该像素对应的 ray 积累云层密度的大小（每次步进累加的密度值）决定，穿过的云层越厚、越密，云的颜色越深，由此实现体积云的半透明基础绘制。

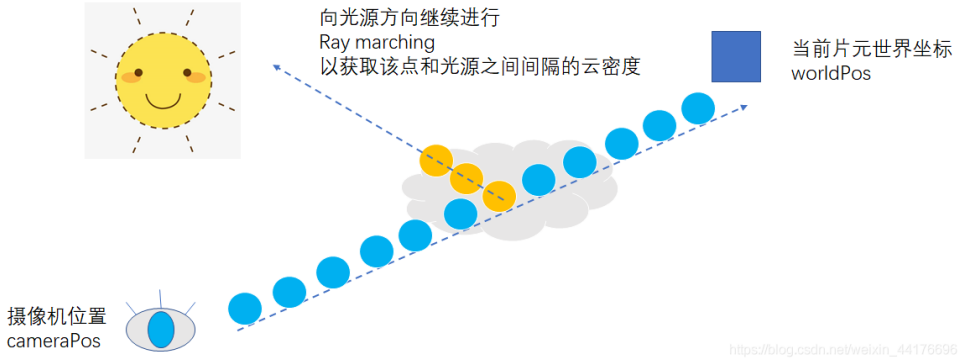
而为了表现出云形状的随机性，云层的长方体中每一点的密度需要随机化，并且相较于完全无规律的随机性，云是在一定空间范围内出现的连续的图形，因此需要使用到**连续噪声图**定义范围内每一点的密度。常见的 2D 连续噪声图有 Perlin Noise 和 Worley Noise，分别对应下图最左边一栏和右边三栏：



图表 6 Noise https://www.shadertoy.com/view/3dVXDc

将噪声图以纹理的方式传递给 shader 后，通过使用二维坐标对其进行采样，即可获得随机且连续的密度值，由此实现 2D 平面的云朵绘制。在 xz 平面的噪声采样基础上，在 y 方向上进行进一步处理，包括但不局限于改用 3D 噪声进行采样、调整竖直方向密度差使得中部较密高部和低部较疏等方式，即可绘制出一个较为逼真的立体体积云。

最后一步，考虑光照的问题，与 Ray Tracing 方法中的 shadow ray 类似，当前位置处于云层范围内时，由该点向光源方向再 march 一次，计算到光源隔了多厚的云，得出该点的亮暗。而在实际实现中，常将云层分为亮处和暗处进行粗略估计。



图表 7 Shading

2. RTR techniques

1) Shadows

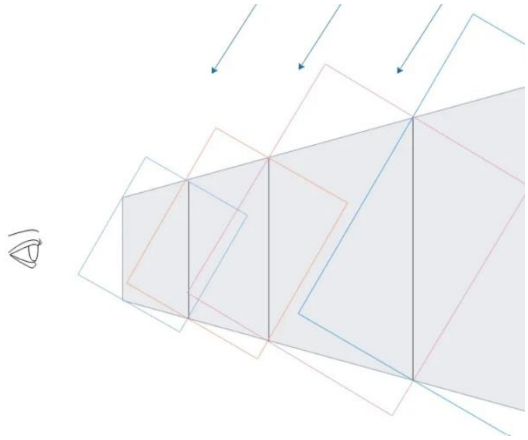
在阴影部分，项目主要使用 shadow maps 方法对于点光源和平行光源实现了阴影效果。shadows maps 主要的思想是将投光物作为一个新的“光相机”，使用光相机观察场景从而得到深度贴图，对于每个 fragment,比较通过计算得出的在光空间下的深度和 shadow map 中的深度，若计算得出的深度大于 shadow map 中存储的深度，则认为该 fragment 位于此光源的阴影中。

对于点光源，我们采用万象阴影贴图的方法，对于每个点光源，从点光源出发向立方体的 6 个面看去从而生成 6 张深度贴图，我们将其以 cube map 的形式传入 fragment shader 中。

对于平行光源，我们采用级联阴影贴图的方法，将视锥体按照和近平面的距离划分为 5 个体积各异的平截头体，并分别在光空间下计算其的紧凑包围盒，再通过包围盒的尺寸信息和平行光的方向计算出正交投影矩阵。

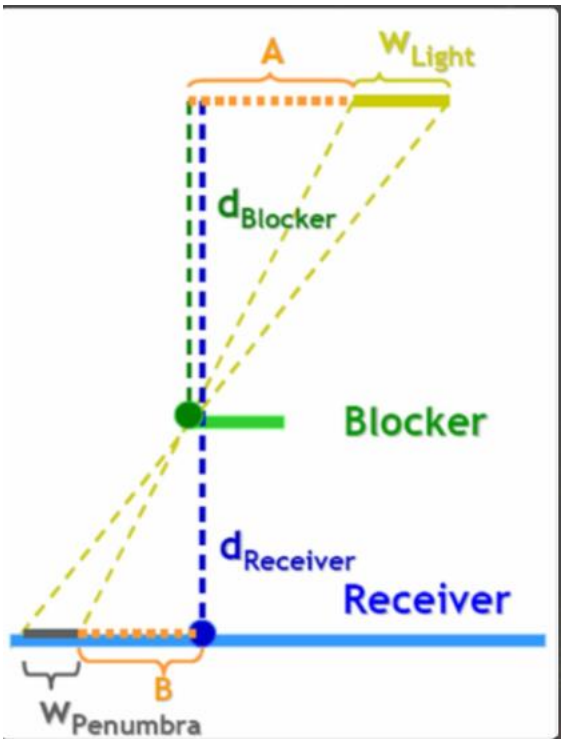
由于 shadow map 方法存在边缘锯齿的问题，我们对于点光源和平行光源分别

使用 PCF 和 PCSS 的方法来实现引用边缘虚化和软阴影的效果。



图表 8 CSM

在不使用 PCF/PCSS 的情况下，要判断一个 fragment 是否处在阴影中，我们只会将其在光空间的深度和对应的 shadow map 中的唯一一个 texel 的深度比较。由于 shadow map 的分辨率是有限的，因此会出现边缘锯齿形的阴影。在 PCF 中，对于一个 fragment，我们对一个固定区域内多个 texel 进行采样，将采样的结果平均得到阴影系数。



图表 9 PCSS

PCSS 可以看做是自适应尺寸滤波核的 PCF，而采样区域大小取决于投光物、遮挡物和被遮挡物三者的位置关系，如下图所示：

其中 $w_{penumbra}$ 可以看做是我们需要的滤波核尺寸，$d_{receiver}$ 是投光物和被遮挡物的距离，$$d_{blocker}$$ 是投光物到遮挡物的距离，w_{light}是光源的宽度，由相似三角形可以得到

$$w_{penumbra} = (d_{receiver} - d_{blocker})w_{light}/d_{blocker}$$

由此我们便可以得到 PCSS 中自适应滤波核的大小。

2) RSM

使用 Reflective Shadow Maps（RSM）进行实现。这是一种实时渲染中用于模拟场景中光和阴影相互作用的技术。它是常用于创建实时图形中阴影的阴影映射算法的变体。它扩展了“从光源空间观察”这一思路。在记录深度信息的同时也在帧缓冲中记录世界坐标、法线、光照颜色信息，在最终渲染过程中利用这些信息，将光源附近的物体的反射光线作为虚拟光源，从而渲染间接光照的效果。

总体算法流程如下：

从光源的视角进行一次渲染，保存上述四种信息至纹理中。

在渲染点附近进行随机采样，使之成为“虚拟点光源”，这里假设屏幕空间里面离这个着色点越近的虚拟点光源贡献越大：

$$s + rmax\xi 1sin(2\pi\xi 2), t + rmax\xi 1cos(2\pi\xi 2))$$

其中 ξ1，ξ2 为随机数

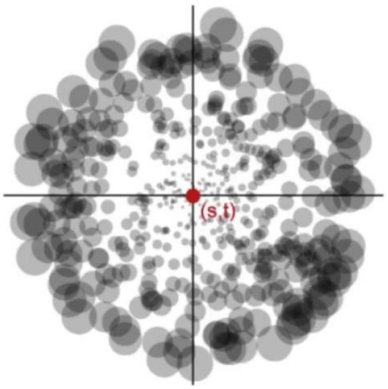


Figure 4: Sampling pattern example. The sample density decreases and the sample weights (visualized by the disk radius) increases with the distance to the center.

图表 10 RSM

根据渲染方程和衰减公式，计算辐射强度

$$Ep(x,n) = \Phi p \frac{max\{0, < n_p \mid x - x_p >\} max\{0, < n \mid x_p - x >\}}{|| x - x_p ||^4}$$

用这所有的虚拟点光源来计算间接光照

次级光照计算

```
vec3 projCoords=FragPosLightSpace.xyz/FragPosLightSpace.w;
projCoords=projCoords*0.5+0.5;

vec3 indirect=vec3(0.0,0.0,0.0);
for (int i=0; i<sample_num; i=i+1){
    vec3 r=texelFetch(randomMap, ivec2(i, 0), 0).xyz;
    vec2 sample_coord=projCoords.xy+r.xy*sample_radius;
    float weight=r.z;
```

```
    vec3 target_normal=texture(normalMap,
sample_coord).xyz;
    vec3 target_worldPos=texture(worldPosMap,
sample_coord).xyz;
    vec3 target_flux=texture(fluxMap, sample_coord).rgb;

    vec3 dis=FragPos-target_worldPos;
    vec3 indirect_result=target_flux*max(0,
dot(target_normal, dis))*max(0, dot(Normal, -dis));
    indirect_result
*=  weight/max(1.0,pow(length(dis),4.0));
    indirect+=indirect_result;
}
indirect=clamp(indirect/sample_num, 0.0, 1.0);

return indirect * RSM_INTENSITY;
```

最后将直接光照和间接光照结果合并。这种实现方式较简单且效果较真实，缺点是有针对多光源效率差、没有考虑间接光的遮挡问题、受限于采样率，效率会较低。代码实现过程中适当减少了采样数来提高效率。

实现方法

与实体的三角片元绘制不同，体积云并非通过 CPU 向 GPU 传递顶点绘制，而是在 shader 中通过算法直接生成。而为了在 Vertex shader 中获得世界坐标，仅需将 view 和 projection 传入即可，同时需要传入的是点光源坐标和时间（用于云层的移动）。

```
// pass projection matrix to shader (note that in this case it could change every frame)
glm::mat4 projection = glm::perspective(glm::radians(camera.Zoom), (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
shader.setMat4("projection", projection);

// camera/view transformation
glm::mat4 view = camera.GetViewMatrix();
shader.setMat4("view", view);

shader.setVec3("cameraPos", camera.Position);
shader.setVec3("lightPos", glm::vec3(0.0f, 400.0f, 0.0f));
shader.setFloat("offset", (float)glfwGetTime());
```

在 Vertex shader 中，将传入的坐标乘以 view 矩阵的逆矩阵得到对应的世界坐标：

```
#version 330 core
layout (location = 0) in vec3 aPos;
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
out vec4 worldPos;
out vec4 bgColor;
void main()
{
    vec4 const_bk = vec4(0.53f, 0.8f, 1.0f, 1.0f);
    worldPos = inverse(view) * vec4(aPos, 1.0f);
    gl_Position = projection * vec4(aPos, 1.0f);
    bgColor = const_bk;
}
```

对于体积云绘制主要在 Fragment shader 中实现。getCloud 函数主要实现 Ray Marching 的过程并返回当前点的颜色（处于云层内，通过上述技术原理获取云的颜色，否则返回背景

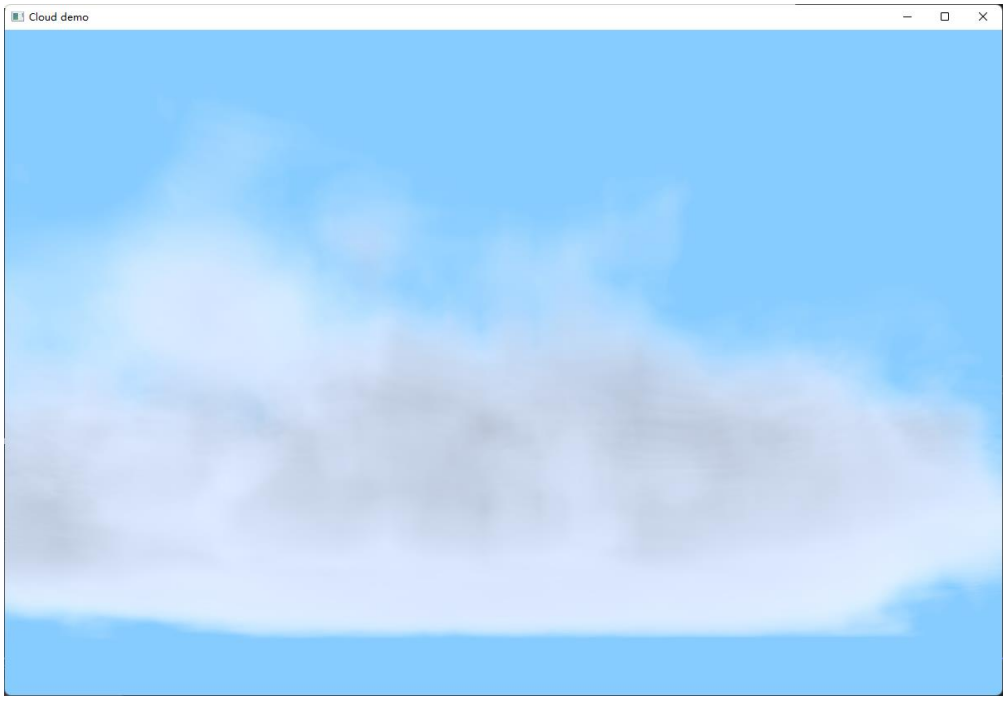
颜色)。

```
// 获取体积云颜色
vec4 getCloud(vec3 cur_pos) {
    vec3 direction = normalize(cur_pos - cameraPos);   // marching direction
    if (direction.y < 0.0) {
        return bgColor;
    }
    // vec3 step = vec3(1.0 / direction.y);   // step length
    vec3 step = direction * step_para;
    vec4 colorSum = vec4(0);                  // accumulated color
    vec3 point = cameraPos;                   // start position

    // if camera is under the cloud, then move the start position to cloud_bottom
    if(point.y < cloud_bottom) {
        point += direction * (abs(cloud_bottom - cameraPos.y) / abs(direction.y));
    }
    // if camera is above the cloud, then move the start position to cloud_top
    if(cloud_top < point.y) {
        point += direction * (abs(cameraPos.y - cloud_top) / abs(direction.y));
    }
    // ray marching
    for(int i = 0; i < march_times; i++) {
        point += step * (step_para + i);
        if(cloud_bottom>point.y || point.y>cloud_top || -width>point.x || point.x>width || -width>point.z || point.z>width) {
            break;
        }
        // 采样
        float density = getCloudNoise(point);                       // current cloud density
        vec3 ldirectoin = normalize(lightPos - point);              // light direction
        float lightDensity = getCloudNoise(point + ldirectoin);     // sample towards light
        float delta = clamp(density - lightDensity, 0.0, 1.0);      // densitiy difference
        // 控制透明度
        density *= transparency;
        // 颜色计算
        vec3 base = mix(baseBright, baseDark, density) * density;   // base color
        vec3 light = mix(lightDark, lightBright, delta);            // light factors
        // 混合
        vec4 color = vec4(base*light, density);                     // final color of the pixel
        colorSum = color * (1.0 - colorSum.a) + colorSum;           // mix colors
    }
    return colorSum;
}
```

```
void main()
{
    FragColor = bgColor;
    vec4 cloud = getCloud(worldPos.xyz);                                // color of the cloud
    FragColor = vec4((FragColor.rgb*(1.0 - cloud.a) + cloud.rgb),1.0f);    // mixed with background color
}
```

最终效果如下所示



3. Others

1) 模型加载

前期尝试

本项目在立项时，希望能够呈现出一间现代化的房屋。考虑到房屋中模型众多，且本项目并没有计划实现在项目中实时编辑物体，因此我们需要一种方式便捷地定义场景中的物体，使其模型、材质、位置等信息能够一一对应。

我们首先从网上下载了房屋模型资源，并试图从 FBX 格式的文件中获取其材质信息。然而由于我们得到的 FBX 文件是二进制格式的，无法分析其文件结构，且这种方式也无法获取模型在场景中的位置信息，因此我们放弃了自行实现对 FBX 文件的解析。

考虑到我们的模型资源来自 Unity 资源商店，其位置信息的存储方式是 Unity 预制体，具体格式为 YAML 文件的一种变体， 因此我们决定自行实现这种文件格式的解析，从而直接获取模型所需的所有信息。然而，由于现有的用于 C++解析 YAML 文件的库对这种变体格式处理能力不足，且 Unity 文件中存储的位置信息并非全局坐标系下的坐标，无法直接在本项目中使用，因此我们决定另寻出路。

实现方案

我们首先自定义了一种 Json 文件的格式，将本项目中的物体与 Json 文件建立起关联，此后只需编辑 Json 文件内容，即可调整场景中的物体。单个物体的 Json 文件主要包含其位置、旋转、缩放信息，以及模型、贴图文件路径（光源物体则包含其光源类型），整个场景的 Json 文件则包含各个物体的文件路径，以及是否启用地形、草地、海洋等全局信息。

```json
{
    "name": "Bed",
    "components": {
        "Transform": {
            "position": [
                -65.25587,
                26.92455,
                81.46814
            ],
            "rotation": [
                9.334668E-06,
                0,
                0
            ],
            "scale": [
                0.01,
                0.01,
                0.01
            ]
        },
        "MeshFilter": {
            "pack": ".\\asset\\model\\meshes\\Bed.FBX"
        },
        "MeshRenderer": {
            "material": {
                "textures": {
                    "material.albedo": ".\\asset\\model\\textures\\Bed_AlbedoTransparency.dds",
                    "material.metallic": ".\\asset\\model\\textures\\Bed_MetallicSmoothness.dds",
                    "material.normal": ".\\asset\\model\\textures\\Bed_Normal.dds",
                    "material.ao": ".\\asset\\model\\textures\\AoDefault.png"
                }
            },
            "shader": "pbr",
            "drawmode": "triangles"
        }
    },
    "isDeferred": true
}
```

图表 11 单个物体 Json 文件内容实例

由于场景中物体众多，手动编辑 Json 文件并不现实，因此我们决定在 Unity 编

辑器中通过 C#脚本，直接将 Unity 场景中的模型转换为本项目中所需的 Json 文件。在转换过程中，可以使用 Unity 提供的 API 输出物体在全局坐标下的位置，解决了坐标系不匹配的问题。另外，由于本项目中尚未实现对透明物体的渲染支持，因此在转换过程中剔除了透明物体（主要是玻璃、植物），防止渲染这些物体时产生意外的效果。

由于我们获取的模型中，一个网格文件包含多个使用不同贴图的部分，而本项目当前仅支持单个模型使用单套贴图，因此我们需要对 FBX 网格进行拆分。我们在 Unity 中查找到了 FBX 网格导出插件，将所有组合物体拆分为了多个网格并导出为 FBX，为其分别生成 Json 文件以便导入。

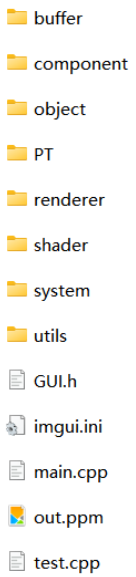
为了提升模型加载速度，我们采用了多线程的方式加载模型。在项目前期，我们尝试在渲染过程中，边渲染边加载场景，防止程序启动时卡顿时间过长。但是 OpenGL 无法在多个线程中同时操作状态机，这意味着无法在多个线程中创建 VAO,VBO 等 buffer object。最后，我们仍然采用了先加载再渲染的方式，将待加载的物体按照 json 中规定的顺序分为 16 组，每组用一个线程加载

2） 材质压缩

在加载部分模型到本项目中后，所使用的内存已经超出了上限，无法加载所有模型。占用内存的主要是模型的材质贴图，由于先前使用的贴图格式是未压缩的格式，因此内存占用过多。我们使用 NVIDIA Texture Tools Exporter 将 png 格式的贴图转换为 OpenGL 可进行采样的压缩格式 dds，从而大幅降低了内存占用率。

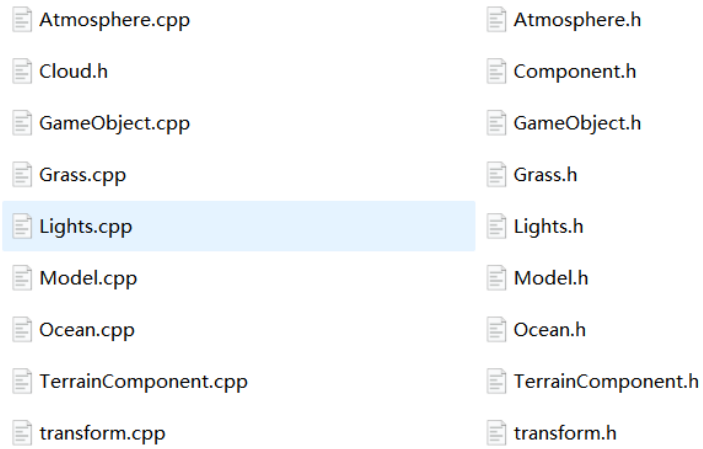
Project contents

项目结构如图 12 所示。



图表 12 项目结构

buffer 文件夹中，封装了 SSBO，UBO，FrameBuffer 等 Buffer 结构，方便开发使用。Component 文件夹中存储了各种组件，包括 transform：存储物体的位置信息；Model 为模型导入文件；Lights 为灯光类型；Cloud、Atmosphere 为 Sky 特殊的组件，实现云和大气效果；TerrainComponent、Grass 和 Ocean 为 Terrain 类特殊的组件，实现地形 Mesh、草地和海洋。



图表 13 Component

Object 文件夹中放置 Terrain、Sky、Camera 等特殊类。
Renderer 文件夹中存放和渲染相关的所有组件和系统，包括 renderManager, MeshFilter（几何信息组件），MeshRenderer（绘制组件），RenderPass（包括 deferred pass 等）RenderScene（场景类），Texture（存储单张图片），Material（存储一张材质）。
Shader 文件夹中存放所有相关的 shader.

Results

项目效果在视频文件中展示，此处不再赘述。

Problems & Solutions

1. 材质文件过大，超出 32GB
   使用 DDS 格式，改格式为 OPENGL 插件支持的图片格式。将 PNG、JPG 预处理为 DDS，再导入，可以有效降低内存占用。
2. 为了使得能进行大规模并行的渲染，采用了 compute shader，但一开始对 compute shader 的使用以及多线程的调用方式不熟悉；以及之前不清楚 texture2D 在 shader 里的传入方式，后期通过学习学会。

```
#version 430 core       You, 2 weeks ago • Ocean …

layout(local_size_x=8,local_size_y=8,local_size_z=1) in;//一个线程组的线程信息

layout(rgba32f,binding=5) uniform image2D InputRT;
layout(rgba32f,binding=6) uniform image2D OutputRT;
```

3. lerp 函数之前实现的时候比例一直是反的，修改后发现对一个 float 进行了 noramlize 的操作（误以为是归一化），结果直接变成 1.0，屏幕空间一堆白色大块像素点

```
vec3 lerp(vec3 a,vec3 b,float t)
{
    //a + (b - a) * t
    return a+(b-a)*t;
}
```

4. IFFT 的优化理论部分学习了挺久，了解了在 IDFT 基础上降低复杂度的原理
5. 噪声生成

对于需要采样的噪声图的获取，最开始的想法是仅使用 2D Perlin Noise 或者 2D Worley Noise，在 y 方向上仅设置高处和低处密度较小，中部密度较大，由此产生的问题是在 y 方向上并没有体现出随机性，视觉上像数个比较规则的球。因此考虑将 2D 噪声改为 3D 噪声。

由于 3D 噪声图无法直接通过采样图片的形式实现，因此考虑在 Fragment shader 中直接实现一个根据 xyz 坐标生成的 3D Perlin Noise，但产生的下一个问题是开销过大，当摄像机对准云层时，出现非常严重的卡顿现象。

查阅相关资料后，找到一种折中的方式：通过连续采样两次 2D 噪声图的方式生成 3D 噪声。具体实现如下图所示：

```
float noise(vec3 x)
{
    vec3 p = floor(x);
    vec3 f = fract(x);
    f = smoothstep(0.0, 1.0, f);
    vec2 uv = (p.xy+vec2(37.0, 17.0)*p.z) + f.xy;
    float v1 = texture2D(noisetex, (uv)/256.0, -100.0 ).x;
    float v2 = texture2D(noisetex, (uv + vec2(37.0, 17.0)) / 256.0, -100.0).x;
    return mix(v1, v2, f.z);
}
```

由于 2D 噪声是通过采样 texture 的方式实现，且 noise 函数计算并不复杂，一次渲染的时间平均能控制在 10ms 上下，且云的形状效果比较理想。

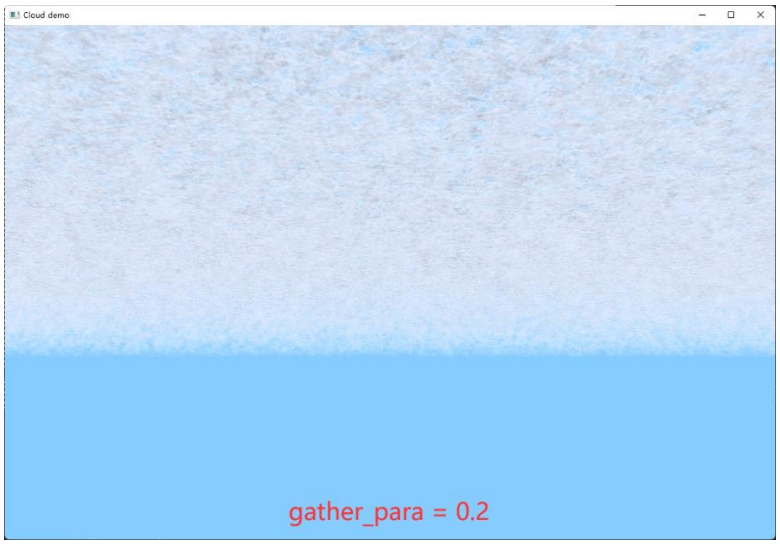
为了丰富云形状的细节，采用如下的方式对噪声多次采样：

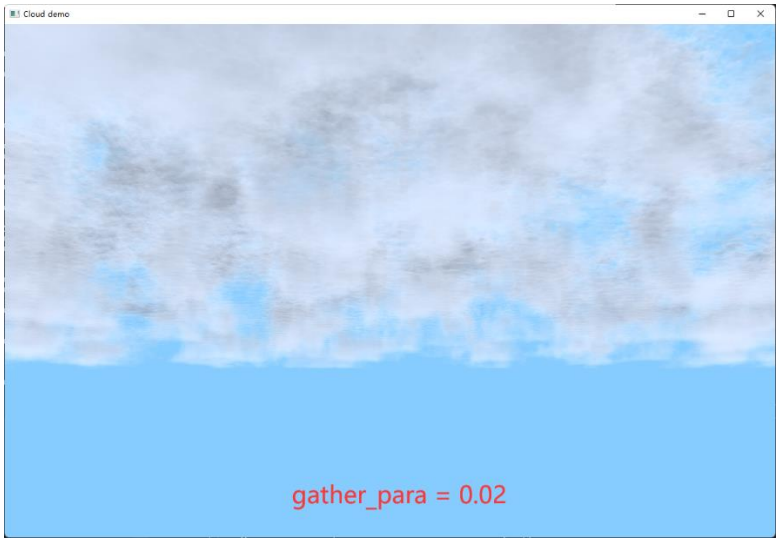
```
float getCloudNoise(vec3 pos) {
    vec3 coord = pos;
    float mid = (cloud_bottom + cloud_top) / 2.0;
    float h = cloud_top - cloud_bottom;
    float weight = 1.0 - 2.0 * abs(mid - pos.y) / h;
    weight = pow(weight, 0.5);
    float v = 1.0;
    if(coord.y < cloud_bottom) {
        v = 1.0 - smoothstep(0.0, 1.0, min(cloud_bottom - coord.y, 1.0));
    }
    else if(coord.y > cloud_top) {
        v = 1.0 - smoothstep(0.0, 1.0, min(coord.y - cloud_top, 1.0));
    }
    coord.z -= float(offset) * 100;
    coord *= gather_para;
    float n = noise(coord) * 0.5;   coord *= 3.0;
    n += noise(coord) * 0.25;       coord *= 3.01;
    n += noise(coord) * 0.125;      coord *= 3.02;
    n += noise(coord) * 0.0625;
    float noise = smoothstep(0.0, 1.0, pow(max(n - 0.5, 0.0) * (1.0 / (1.0 - 0.5)), 0.4));
    noise *= weight;
    noise *= v;
    return noise;
}
```

可以看到 getCloudNoise 函数中调用了四次 noise 函数，即对噪声进行了四次采样，并使用了 smoothstep 进行了平滑处理。

4. 云的大小和分散程度

在绘制时，发现云呈现出小而细碎的效果，而实际的云的形状应为大团并比较分散，即单位空间内云朵应体积较大且数量较少。应对此问题的解决方法是在上述 getCloudNoise 函数中，将 coord 乘以一个参数 gather_para 进行调整，此参数越小，云越大、越聚集。不同 gather_para 下云的表现效果如下三张图所示：

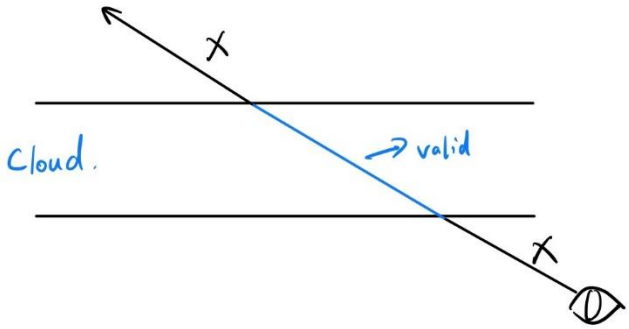




6. 由于 Ray Marching 同 Ray Tracing 一样，需要向世界空间中各个方向投射 ray，因此开销较大，需要进行优化，去除无效的渲染。

去除无效 Ray Marching

由于云层的范围是人为规定的，因此对于步进而言，仅在云层范围内的是有效的，射出云层外的和射入云层前的 marching 均是无效的。由此可以通过两种方式实现优化：



首先是射入云层前的部分，处理方式是如果判断起点在云层的下方，则沿着 direction 将步进起点移动到云层的低端（cloud_bottom）：

```
// if camera is under the cloud, then move the start position to cloud_bottom
if(point.y < cloud_bottom) {
    point += direction * (abs(cloud_bottom - cameraPos.y) / abs(direction.y));
}
```

然后是射出云层的部分，处理方式是如果判断当前步进已经离开了云层范围，则直接 break 结束步进循环：

```
for(int i = 0; i < march_times; i++) {
    point += step * (step_para + i);
    if(cloud_bottom>point.y || point.y>cloud_top || -width>point.x || point.x>width || -width>point.z || point.z>width) {
        break;
    }
```

可变步长采样

Ray Marching 是按照一定的步长进行步进的，步长越小，绘制出来的云的品质也就越高；同时步进次数和步长的乘积决定了 Ray Marching 的范围。事实上，远处的云对于品质的要求并不高，并且希望确保云层范围内所有的云均在步进范围内，因此可以采用可变步长进行采样，具体为近处采用小步长步进，远处使用大步长进行步进，如下图所示，随着步进次数增加，i 增加，会使步长逐渐增大。

```
for(int i = 0; i < march_times; i++) {
    point += step * (step_para + i);
    if(cloud_bottom>point.y || point.y>cloud_top || -width>point.x || point.x>width || -width>point.z || point.z>width) {
        break;
    }
```

Roles in group

- 张智淋: Volumetric Cloud
- 雒俊为: Reflective Shadow map
- 黎可杰: Ocean
- 陈开煦: Shadow
- 晋愉翔: Model Loading
- 朱昀玮: Procedural Terrain, Physically-based sky, Grass, part of Model loading, Framework design.

Reference

[1] Dachsbacher C , Stamminger M . Reflective shadow maps[J] interactive 3d graphics and games. 2005.
[2] illaire, Sébastien. (2020). A Scalable and Production Ready Sky and Atmosphere Rendering Technique. Computer Graphics Forum. 39. 13-22. 10.1111/cgf.14050.
[3] Jeremy Moore(2018). Terrain Rendering in Far Cry 5. Game developers conference.
[4] Joey. Learn Opengl