

FDU CG Final-miniRTRender(Vulkan Compute Shader)

技术路径: GPU Compute Shader (Vulkan)

曹琦 22307110076

FUDU CG Final-miniRTRender(Vulkan Compute Shader)

Part1 Time Spent

Part2 Implementation Approach

0.BaseCode作用

1. 载入背景图像
2. 物体几何变换
3. 模型三角形数组导入compute shader
4. 轴对齐包围盒AABB
5. 光线三角形求交与场景遍历
6. 顶点法向平滑与重心坐标插值
7. 材质设定

Part3 Issues and Resolutions

1. 面片渲染时缺失
2. 多模型管理--GLSL 不支持嵌套数组作为 SSBO中的成员变量
3. 针对多面高模的渲染速度较慢 : AABB 展开
4. 全反射逻辑修正
5. 宝石渲染效果不一致

Part1 Time Spent

环境搭建: 10min

阅读资料: 2h

代码实现: 15h

commit记录如下:

<fix> 修复了全反射发生时的错误逻辑	nobody 2025/6/9... 192da8f6
<style> 优化代码风格	nobody 2025/6/8... 5a20a5a8
<feat> 实现常数级的BVH，实现进一步的求交加速	nobody 2025/6/8... 0684d1c3
<style> 调整参数，使视觉效果接近reference	nobody 2025/6/8... f0ff4ea6
null	nobody 2025/6/8... b2c5d6c4
<feat> 在shader中修改光线三角形求交函数，新增重心坐标插值逻辑。实现可选的平滑着色	nobody 2025/6/7... 480f699d
<feat> 修改Triangle类结构，增加顶点法向，实现cpu端预计算顶点法向的函数	nobody 2025/6/7... ec57c053
<feat> 修改shader增加多个物体包围盒分别计算的逻辑，支持多物体场景AABB加速	nobody 2025/6/7... 4e608164
<feat> 添加model类和ModelSSBO并上传到GPU，用于在shader中管理多个模型的求交	nobody 2025/6/7... 75594840
<style> 将多个模型配置文件抽象为一个文件，将几何相关的类抽出为一个单独的头文件	nobody 2025/6/7... adeb826e
<feat> 实现cpu端预计算的坐标变换	nobody 2025/6/6... d5204cb2
<fix> 调整MIN_EPSILON的值，确保高模能够正确显示	nobody 2025/6/6... cc5cc33d
<fix> 修改shader中 scene_intersect中错误的材质使用模式	nobody 2025/5/2... 8c51f3db
<test> 尝试构建数据结构便于导入和绘制多个模型	nobody 2025/5/2... 01cb392f
<feat> 实现shadow_intersect	nobody 2025/5/2... a8c5d07b
<feat> 完成aabb求交	nobody 2025/5/2... 7c6b9c59
<feat> 完成场景求交，成功显示鸭子	nobody 2025/5/2... 159d188c
<feat> 在ubo中加入新的变量 三角形数量	nobody 2025/5/2... fade0b8b
<feat> 实现三角形求交，尚未实现顶点法向平滑	nobody 2025/5/2... 9fecb99d
绑定三角洲buffer到shader	nobody 2025/5/2... 6fcbbad1
<feat> createShaderStorageBuffers创建三角形缓冲区	nobody 2025/5/2... 7eb6ae18
<feat> 修改shader中cast_ray，不命中物体时从背景图中采样颜色数据	nobody 2025/5/2... ab7bd456
null	nobody 2025/5/2... 514e7ca7
<feat> createComputeDescriptorSets完成背景图片描述符集绑定到shader	nobody 2025/5/2... 23b1f23a
<feat> createTextureSampler() 完成背景图像采样器	nobody 2025/5/2... 46e087d2
<feat> createImageViews()函数完成背景图像的image 和imageview资源准备	nobody 2025/5/2... 78843fea
添加项目文件。	nobody 2025/5/2... d66d3d02
添加 .gitattributes 和 .gitignore。	nobody 2025/5/2... 88f5c898

Part2 Implementation Approach

本项目基于25 spring FDU CG助教提供的，由vulkan tutorial中compute shader一章的粒子系统代码改进，实现了一个简单的Whitted-style 光线追踪渲染器。

vk compute shader render实现效果:



C++ 实现参考效果:



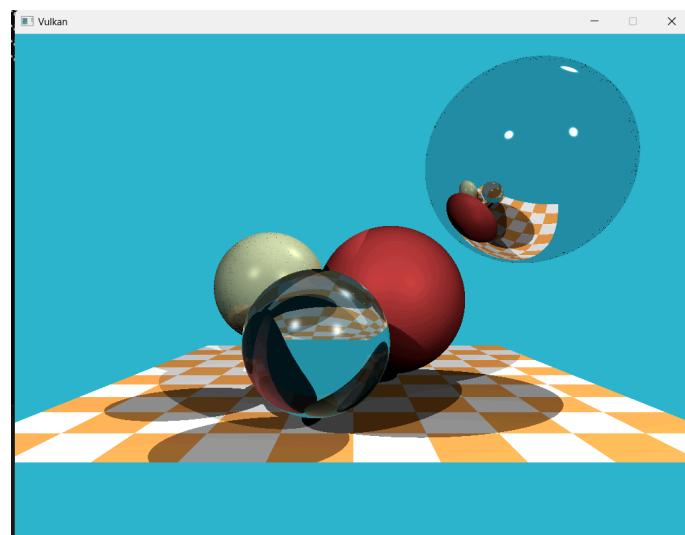
实现的整体思路和细节如下：

0. BaseCode作用

助教团队提供的BaseCode 提供了一个结构完整的vulkan程序的基本框架。包括全部基础setup如创建实例、设备、交换链、命令池和帧缓冲等关键组件，配置图形管线和计算管线，使得顶点着色器、片段着色器以及计算着色器能够加载并执行。

BaseCode 还提供了一些辅助功能。在 CPU 和 GPU 侧定义了基本的材质结构体，用于描述物体表面属性，颜色、反射率、Shiness等，并通过 Uniform Buffer将这些信息传递至 Compute Shader。此外，还提供了光线与球体求交的GLSL函数，这对我后续实现光线与三角形mesh求交提供了一个清晰的思路。由于GLSL 4.50版本还不支持递归，BaseCode 中设计了一个基于栈的非递归 cast_ray 函数原型，有效避免了在 Compute Shader 中使用递归所带来的兼容性和性能问题。

BaseCode直接运行，得到的结果如下图所示。总的来说一个完整可运行的BaseCode帮我们省去了很多搭建实验环境的前期工作，有助于我们更关注算法实现本身。



1. 载入背景图像

载入背景贴图的思路是：将背景图像上传为 GPU 可访问的纹理资源，并通过 `uniform sampler2D backgroundImage` 声明在着色器中进行采样，在 shader 的 `cast_ray()` 函数中，为未与场景物体相交的射线提供环境颜色。具体实现在 `createImageViews()`、`createTextureSampler()` 处：

使用图像加载库 (`stb_image`) 读取背景贴图文件；

创建背景图像 `VkImage`，配置图像格式 (`VK_FORMAT_R8G8B8A8_UNORM`)、尺寸等信息，创建图像对象并分配设备内存；

```
937     // create background image
938     // hint: VK_FORMAT_R8G8B8A8_UNORM
939     // hint: transition image layout to VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
940     VkImageCreateInfo backgroundImageInfo = {};
941     backgroundImageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
942     backgroundImageInfo.imageType = VK_IMAGE_TYPE_2D;
943     backgroundImageInfo.format = VK_FORMAT_R8G8B8A8_UNORM; // 根据hint
944     backgroundImageInfo.extent.width = texWidth;
945     backgroundImageInfo.extent.height = texHeight;
946     backgroundImageInfo.extent.depth = 1;
947     backgroundImageInfo.mipLevels = 1;
948     backgroundImageInfo.arrayLayers = 1;
949     backgroundImageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
950     backgroundImageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
951     backgroundImageInfo.usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
952     backgroundImageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
953
954     vkCreateImage(device, &backgroundImageInfo, nullptr, &backgroundImage);
```

创建图像视图 `ImageView`：定义图像的访问方式和格式，用于绑定到描述符；

```
1039    // create background image view
1040    VkImageViewCreateInfo backgroundViewInfo = {};
1041    backgroundViewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
1042    backgroundViewInfo.image = backgroundImage;
1043    backgroundViewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
1044    backgroundViewInfo.format = VK_FORMAT_R8G8B8A8_UNORM;
1045    backgroundViewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
1046    backgroundViewInfo.subresourceRange.baseMipLevel = 0;
1047    backgroundViewInfo.subresourceRange.levelCount = 1;
1048    backgroundViewInfo.subresourceRange.baseArrayLayer = 0;
1049    backgroundViewInfo.subresourceRange.layerCount = 1;
1050
1051    vkCreateImageView(device, &backgroundViewInfo, nullptr, &backgroundImageView);
```

创建 `sampler`：设置纹理过滤与寻址模式

```
1073    // create background image sampler
1074    // hint: VK_FILTER_LINEAR
1075    // hint: VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE
1076    VkSamplerCreateInfo backgroundSamplerInfo{};
1077    backgroundSamplerInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
1078    backgroundSamplerInfo.magFilter = VK_FILTER_LINEAR; // 放大时使用线性滤波
1079    backgroundSamplerInfo.minFilter = VK_FILTER_LINEAR; // 缩小时使用线性滤波
1080    backgroundSamplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR; // 使用线性 mipmaping
1081    backgroundSamplerInfo.addressModeU = backgroundSamplerInfo.addressModeV = backgroundSamplerInfo
1082        .addressModeW = VK_ADDRESS_MODE_CLAMP_TO_EDGE; // 高于纹理范围返回黑色
1083    backgroundSamplerInfo.anisotropyEnable = VK_FALSE; // 启向异性过滤（需要设备支持）
1084    backgroundSamplerInfo.maxAnisotropy = 1.0f; // 如果开启，可设为更高值如 16.0f
1085    backgroundSamplerInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK; // 超出范围的像素返回黑色
1086    backgroundSamplerInfo.unnormalizedCoordinates = VK_FALSE; // 使用归一化坐标 [0,1]
1087    backgroundSamplerInfo.compareEnable = VK_FALSE;
1088    backgroundSamplerInfo.compareOp = VK_COMPARE_OP_ALWAYS;
1089    backgroundSamplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
1090    backgroundSamplerInfo.mipLodBias = 0.0f;
1091    backgroundSamplerInfo.minLod = 0.0f;
1092    backgroundSamplerInfo.maxLod = 0.0f;
```

更新 Descriptor Set：将图像视图和采样器绑定到描述符集中的对应绑定点；

```

1763     // update background image
1764     // hint: VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
1765     // hint: see createComputeDescriptorSetLayout function for reference
1766     VkDescriptorImageInfo backgroundImageDescriptor = {};
1767     backgroundImageDescriptor.imageView = backgroundImageView;
1768     backgroundImageDescriptor.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
1769     backgroundImageDescriptor.sampler = backgroundImageSampler;
1770
1771     descriptorWrites[4].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
1772     descriptorWrites[4].dstSet = computeDescriptorSets[i];
1773     descriptorWrites[4].dstBinding = 4; // layout(binding=4) in shader
1774     descriptorWrites[4].dstArrayElement = 0;
1775     descriptorWrites[4].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
1776     descriptorWrites[4].descriptorCount = 1;
1777     descriptorWrites[4].pImageInfo = &backgroundImageDescriptor;

```

在 Compute Shader 中采样背景贴图：通过 `texture()` 函数使用射线方向映射到球形或立方体贴图坐标，获取背景颜色。

```

3   vec3 cast_ray(vec3 orig, vec3 dir) {
4     // TODO-finished
5     // You need to modify this function to add background
6
7     vec3 color = vec3(0.0);
8     PathSegment stack[MAX_STACK_SIZE];
9     int stackSize = 0;
10    stack[stackSize++] = PathSegment(orig, dir, vec3(1.0), 0);
11
12    while (stackSize > 0) {
13      PathSegment segment = stack[--stackSize];
14      if (segment.depth ≥ MAX_DEPTH || dot(segment.throughput, segment.throughput) < 0.001) continue;
15
16      SceneHit hit;
17      scene_intersect(segment.origin, segment.direction, hit);
18      // 未命中时从背景图采样颜色
19      if (!hit.hit) {
20        vec2 uv = direction_to_uv(segment.direction);
21        vec3 bg_color = texture(backgroundImage, uv).rgb;
22        //vec3 bg_color = BACKGROUND_COLOR;
23        color += segment.throughput * bg_color;
24        continue;
25      }
26    }

```

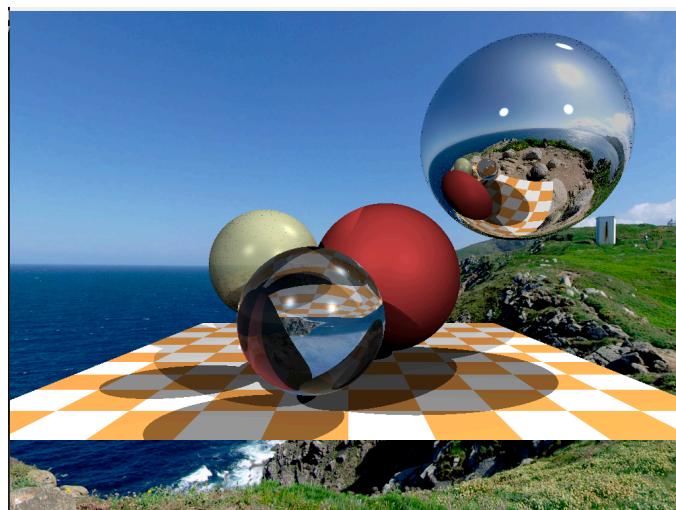
为了实现对提供的2D 全景贴图进行采样，实现一个shader函数，将光线方向向量映射为背景纹理的 UV 坐标

```

310    vec2 direction_to_uv(vec3 dir) {
311      float theta = atan(dir.z, dir.x); // [-PI, PI]
312      float phi = acos(dir.y); // [0, PI]
313      float u = (theta + PI) / (2.0 * PI); // [0, 1]
314      float v = phi / PI; // [0, 1]
315      return vec2(u, v);
316    }

```

这里参考帖子[5.球面纹理贴图 imgui 绘制3d球型纹理-CSDN博客](#)



2. 物体几何变换

在 CPU 端导入模型数据（调用 `loadObjAsTriangles()`）后，对每个三角形执行缩放，旋转，平移变换，并将变换后的顶点数据上传至 GPU 的缓冲区供着色器访问。

调用GLM提供的函数构建变换矩阵

```
void transformTriangles(std::vector<Triangle>& triangles, const glm::vec3& scale, const glm::vec3& translation, const glm::vec3& rotation) {
    /* ... */
    // 创建模型变换矩阵
    glm::mat4 model = glm::mat4(1.0f);
    model = glm::translate(model, translation);
    model = glm::rotate(model, glm::radians(rotation.z), glm::vec3(0, 0, 1));
    model = glm::rotate(model, glm::radians(rotation.y), glm::vec3(0, 1, 0));
    model = glm::rotate(model, glm::radians(rotation.x), glm::vec3(1, 0, 0));
    model = glm::scale(model, scale);

    // 应用到每个三角形
    for (int i = 0; i < (int)triangles.size(); ++i) {
        triangles[i].v0 = model * triangles[i].v0;
        triangles[i].v1 = model * triangles[i].v1;
        triangles[i].v2 = model * triangles[i].v2;
    }
}
```

物体的变换信息以全局变量形式存放在config.hpp的ModelInfo对象中

```
#pragma once
#define GLM_FORCE_RADIANS
#include <glm/glm.hpp>
#include <vector>
#include <string>
#include "geometry.hpp"
// 通用

// Duck 配置
constexpr Material duckMaterial = { {0.0, 0.5, 0.1, 0.8}, {0.6, 0.7, 0.8, 125.0}, {1.5, 0.0, 0.0, 0.0} };
constexpr glm::vec3 duckScale = { 1.0f, 1.0f, 1.0f };
constexpr glm::vec3 duckRotation = { 0.0f, 0.0f, 0.0f };
constexpr glm::vec3 duckTranslation = { 0.0f, 0.0f, 0.0f };
constexpr int duckNormalInterpolation = 0;
Model Duck(duckMaterial, duckNormalInterpolation);
ModelInfo duckInfo(Duck, "assets/duck.obj", duckScale, duckRotation, duckTranslation);
```

3. 模型三角形数组导入compute shader

首先，在 CPU 端使用模型加载库 `tiny_obj_loader` 解析模型文件，提取顶点和索引数据，并将其组织为包含每个三角形顶点位置的数组；

创建一个 GPU 可访问的缓冲区 `vkBuffer`，并为其分配设备内存，将三角形数据上传至该缓冲区；

接着在着色器中声明只读的缓冲区接口 (`layout(std140, binding = 5) readonly buffer TrianglesSBO`)，并与描述符集绑定；

最后，在 Compute Shader 中通过索引方式访问该数组，执行光线与三角形的相交计算。

`createShaderStorageBuffers()`

```

1546     // create triangle buffer,
1547     // hint: Ref to Ray buffer implementation above
1548     // Triangle buffer
1549     VkDeviceSize triangleBufferSize = sizeof(Triangle) * triangles.size();
1550     VkBuffer triangleStagingBuffer;
1551     VkDeviceMemory triangleStagingBufferMemory;
1552     createBuffer(triangleBufferSize,
1553         VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
1554         VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
1555         triangleStagingBuffer, triangleStagingBufferMemory);
1556
1557     void* data0;
1558     vkMapMemory(device, triangleStagingBufferMemory, 0, triangleBufferSize, 0, &data0);
1559     memcpy(data0, triangles.data(), (size_t)triangleBufferSize);
1560     vkUnmapMemory(device, triangleStagingBufferMemory);
1561
1562     // 创建多个 Triangle 缓冲区 (每帧一个)
1563     shaderStorageTriangleBuffers.resize(MAX_FRAMES_IN_FLIGHT);
1564     shaderStorageTriangleBuffersMemory.resize(MAX_FRAMES_IN_FLIGHT);
1565
1566     for (size_t i = 0; i < MAX_FRAMES_IN_FLIGHT; i++)
1567     {
1568         createBuffer(triangleBufferSize,
1569             VK_BUFFER_USAGE_STORAGE_BUFFER_BIT | VK_BUFFER_USAGE_TRANSFER_DST_BIT,
1570             VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
1571             shaderStorageTriangleBuffers[i], shaderStorageTriangleBuffersMemory[i]);
1572
1573         copyBuffer(triangleStagingBuffer, shaderStorageTriangleBuffers[i], triangleBufferSize);
1574     }
1575     vkDestroyBuffer(device, triangleStagingBuffer, nullptr);
1576     vkFreeMemory(device, triangleStagingBufferMemory, nullptr);
1577

```

由于GLSL不支持嵌套数组，故这里将所有模型的三角形都放在一个统一的三角形数组中传入，再通过另一个结构管理不同模型的包围盒等其他信息。详细说明见Part3-2

```

65     layout(std140, binding = 5) readonly buffer TriangleSSBO {
66         Triangle triangles[];
67     };
68
69     layout(std140, binding = 6) readonly buffer ModelSSBO {
70         Model models[];
71     };

```

4. 轴对齐包围盒AABB

在 CPU 端导入模型时为每个模型预算算其包围盒的最小和最大顶点坐标，并将这些信息封装在 `Model` 结构体中，通过 SSBO 上传至 GPU；

```

1523     // 计算包围盒
1524     for (const Triangle& tri : modelTriangles){
1525         model.bboxMin = glm::min(model.bboxMin, glm::min(tri.v0, glm::min(tri.v1, tri.v2)));
1526         model.bboxMax = glm::max(model.bboxMax, glm::max(tri.v0, glm::max(tri.v1, tri.v2)));
1527     }
1528
1529     // 将 modelTriangles 追加到 triangles 后面
1530     int startid = static_cast<int>(triangles.size());
1531     triangles.insert(triangles.end(), modelTriangles.begin(), modelTriangles.end());
1532     model.params0.x = startid; // 起始索引
1533     model.params0.y = static_cast<int>(modelTriangles.size()); // 三角形数量
1534     models.push_back(model);

```

利用了分离轴定理 (Separating Axis Theorem)，通过计算光线原点 `orig` 沿方向 `dir` 与包围盒边界 `bboxMin` 和 `bboxMax` 的交点，分别求出进入和离开包围盒的距离 `tmin` 和 `tmax`；然后根据这些值计算出光线穿过 AABB 的最近和最远距离 `tNear` 和 `tFar`，若 `tNear <= tFar` 且 `tFar > MIN_EPSILON`，说明光线确实穿过了该包围盒，返回 `true`，否则返回 `false`。

```

85     bool ray_aabb_intersect(vec3 orig, vec3 dir, vec3 invDir, vec3 bboxMin, vec3 bboxMax) {
86         // implement the bbox intersect for optimization
87         vec3 t0 = (bboxMin - orig) * invDir;
88         vec3 t1 = (bboxMax - orig) * invDir;
89         vec3 tmin = min(t0, t1);
90         vec3 tmax = max(t0, t1);
91         float tNear = max(max(tmin.x, tmin.y), tmin.z);
92         float tFar = min(min(tmax.x, tmax.y), tmax.z);
93
94         return tNear <= tFar && tFar > MIN_EPSILON;
95     }

```

在 Compute Shader 的光线和场景求交函数中，先判断是否与 AABB 相交再进一步计算三角形相交，以加速光线与模型的相交判断，从而提升整体光线追踪性能。

```

232     for(int i = 0; i < models.length(); i++){
233         if(ray_aabb_intersect(orig, dir, invDir, models[i].bboxMin.xyz, models[i].bboxMax.xyz)){
234             int startindex = int(models[i].params0.x);
235             int count = int(models[i].params0.y);
236             int endindex = startindex + count;
237             int normalinterpolation = int(models[i].params0.z);
238             for (int j = startindex; j < endindex; j++){
239                 Triangle tri = triangles[j];
240                 float t;
241                 vec3 hitPoint, hitNormal;
242                 if (ray_triangle_intersect(orig, dir, tri, normalinterpolation, t, hitNormal)) {
243                     if (t > MIN_EPSILON && t < nearest.distance) {
244                         hitPoint = orig + dir * t;
245                         nearest.hit = true;
246                         nearest.distance = t;
247                         nearest.point = hitPoint;
248                         nearest.normal = hitNormal;
249                         // 设置材质
250                         nearest.material = tri.material;
251                     }
252                 }
253             }
254         }
255     }

```

5. 光线三角形求交与场景遍历

在 shader 中通过 Möller-Trumbore 算法实现光线与三角形求交

参考帖子[\[数学\] Möller-Trumbore 算法 - 知乎](#)

$$\begin{bmatrix} t \\ b1 \\ b2 \end{bmatrix} = \frac{1}{S_1 \cdot E_1} \begin{bmatrix} S_2 \cdot E_2 \\ S_1 \cdot S \\ S_2 \cdot d \end{bmatrix}$$

Where:

$$\begin{aligned} S &= o - P_0 \\ E_1 &= P_1 - P_0 \\ E_2 &= P_2 - P_0 \\ S_1 &= d \times E_2 \\ S_2 &= S \times E_1 \end{aligned}$$

```

bool ray_triangle_intersect(vec3 orig, vec3 dir, Triangle tri, int
normalinterpolation, out float t, out vec3 normal) {
    vec3 v0 = tri.v0.xyz;
    vec3 v1 = tri.v1.xyz;
    vec3 v2 = tri.v2.xyz;
    vec3 edge1 = v1 - v0;
    vec3 edge2 = v2 - v0;
    // 计算光线方向和第一个边的叉积 (P)
    vec3 h = cross(dir, edge2);
    float a = dot(edge1, h);
    // 如果 a 接近 0, 光线和平面平行
}

```

```

if (a > -MIN_EPSILON && a < MIN_EPSILON)
    return false;

float f = 1.0 / a;
// 计算从原点到顶点的向量
vec3 s = orig - v0;
// 计算 u 参数
float u = f * dot(s, h);
if (u < 0.0 || u > 1.0)
    return false;

// 计算第二个叉积 (q)
vec3 q = cross(s, edge1);
// 计算 v 参数
float v = f * dot(dir, q);
if (v < 0.0 || u + v > 1.0)
    return false;
// 计算交点距离 t
t = f * dot(edge2, q);

if (t <= MIN_EPSILON)
    return false;

// 设置法线
if (normalinterpolation == 0) {
    // 使用平面法线 (Flat Shading)
    normal = normalize(cross(edge1, edge2));
} else {
    // 使用重心坐标顶点法线插值 (Smooth Shading)
    float w = 1.0 - u - v;
    normal = normalize(w * tri.v0_norm.xyz + u * tri.v1_norm.xyz + v *
tri.v2_norm.xyz);
}

return true;
}

```

场景遍历在 shader 中通过 `scene_intersect()` 函数中串行执行，首先初始化一个最近交点结构体，随后依次检测光线是否与地板相交，若满足条件则更新最近交点信息；对一组球体进行循环检测，保留距离最近的有效交点；遍历加载的模型对象，利用 AABB 包围盒进行初步剔除，仅对可能相交的模型对应的三角形数组进行精确求交计算，以提升性能。每个模型的三角形索引范围由参数指定，函数从中取出对应三角形并逐个进行光线追踪测试，一旦发现更近的有效交点，则更新当前最近交点的数据。

6. 顶点法向平滑与重心坐标插值

首先对仅包含三个顶点位置的 `Triangle` 结构体修改为包含每个顶点的位置、法向量

```

struct Triangle
{
    glm::vec4 v0;
    glm::vec4 v1;
    glm::vec4 v2;
    Material material;
    glm::vec4 v0_norm;
    glm::vec4 v1_norm;
    glm::vec4 v2_norm;
};

```

在加载模型数据后，根据模型是否开启 `Normalinterpolation` 选项决定是否预计算顶点法向；

预算实现思路：为每个唯一的顶点位置（以顶点 `glm::vec4` 作为键，使用自定义的异或哈希函数）维护一个累加的法向量；遍历所有三角形，对每个三角形的三个顶点累加其对应的面法线（通过叉乘计算）；最后对每个顶点的累加法向进行归一化，并将结果写入对应三角形结构体的顶点法线字段中，供后续在着色器中插值使用。在这里由于 `glm vec4` 使用的是 `float` 存储数据，所以使用异或导致不同顶点哈希到同一个值的碰撞概率很小。

```

198     void computeVertexNormals(std::vector<Triangle>& triangles) {
199     /* ... */
200     struct VertexHash {
201         size_t operator()(const glm::vec4& v) const {
202             return std::hash<float>()(v.x) ^ std::hash<float>()(v.y) ^ std::hash<float>()(v.z);
203         }
204     };
205     // 这里使用异或hash有可能会出现少量碰撞导致的法向错误
206
207     std::unordered_map<glm::vec4, glm::vec3, VertexHash> vertexNormals;
208
209     for (auto& tri : triangles) {
210         // 计算当前三角形的面法线
211         glm::vec3 edge1 = glm::vec3(tri.v1 - tri.v0);
212         glm::vec3 edge2 = glm::vec3(tri.v2 - tri.v0);
213         glm::vec3 faceNormal = glm::normalize(glm::cross(edge1, edge2));
214
215         // 累加到三个顶点的法线中
216         vertexNormals[tri.v0] += faceNormal;
217         vertexNormals[tri.v1] += faceNormal;
218         vertexNormals[tri.v2] += faceNormal;
219     }
220
221     // 归一化并写入每个三角形的顶点法线
222     for (auto& tri : triangles) {
223         tri.v0_norm = glm::vec4(glm::normalize(vertexNormals[tri.v0]), 0.0f);
224         tri.v1_norm = glm::vec4(glm::normalize(vertexNormals[tri.v1]), 0.0f);
225         tri.v2_norm = glm::vec4(glm::normalize(vertexNormals[tri.v2]), 0.0f);
226     }
227 }
228 }
```

这些数据随着三角形SSBO上传到GPU端由shader调用；

在计算光线与三角形交点时，使用经典的 Möller-Trumbore 算法恰好会计算出该交点对应的重心坐标参数 `u` 和 `v`（其中 `w = 1 - u - v`）作为中间变量，利用这一点可以轻松完成对交点法向的重心坐标插值。

```

// 设置法线
if (normalInterpolation == 0) {
    // 使用平面法线 (Flat Shading)
    normal = normalize(cross(edge1, edge2));
} else {
    // 使用重心坐标顶点法线插值 (Smooth Shading)
    float w = 1.0 - u - v;
    normal = normalize(w * tri.v0_norm.xyz + u * tri.v1_norm.xyz + v * tri.v2_norm.xyz);
}
```

对玻璃鸭子开启平滑的效果：



7. 材质设定

项目沿用了tinyraytracer中的材质参数设定

```
struct Material {  
    vec4 albedo;  
    vec4 diffuse_specular;  
    vec4 refractive;  
};
```

分量	含义	说明
Albedo.x	漫反射 (Diffuse) 强度系数	控制漫反射成分的强弱，0 表示无漫反射
Albedo.y	高光反射 (Specular) 强度系数	控制镜面高光部分的亮度
Albedo.z	反射 (Reflection) 贡献系数	控制反射光线对最终颜色的影响程度
Albedo.w	折射 (Refraction) 贡献系数	控制折射光线对最终颜色的影响程度
diffuse_specular.r	红色通道	决定物体在漫反射光照下的红色分量
diffuse_specular.g	绿色通道	决定物体在漫反射光照下的绿色分量
diffuse_specular.b	蓝色通道	决定物体在漫反射光照下的蓝色分量
diffuse_specular.a	Shiness	控制镜面反射的锐利程度，即高光区域的集中程度
refractive	折射率	控制光线穿过表面时的方向变化

材质信息在导入三角形时一并写入到了triangle对象中，通过SSBO上传到GPU由shader调用；

shader在cast_ray函数实现了基于材质属性的光线追踪路径积分

```

326     vec3 n = hit.normal;
327     vec3 v = -segment.direction;
328     vec3 diffuse = vec3(0.0);
329     vec3 specular = vec3(0.0);
330     vec3 lights[3] = {ubo.light0.xyz, ubo.light1.xyz, ubo.light2.xyz};
331     float lightsintensity[3] = {ubo.light0.w, ubo.light1.w, ubo.light2.w};
332     for (int i = 0; i < 3; i++) {
333         vec3 light_dir = normalize(lights[i] - p);
334         float light_dist = length(lights[i] - p);
335         vec3 shadow_origin = dot(light_dir, n) < 0.0 ? p - n * MIN_EPSILON : p + n * MIN_EPSILON;
336         float attenuation = 1;
337         if (shadow_intersect(shadow_origin, light_dir, light_dist)) continue;
338
339         float diff = attenuation * max(0.0, dot(n, light_dir));
340         diffuse += diff * hit.material.diffuse_specular.rgb;// 乘上衰减
341
342         vec3 reflect_dir = reflect(-light_dir, n);
343         float spec = attenuation * pow(max(0.0, dot(reflect_dir, v)), hit.material.diffuse_specular.w);
344         specular += spec * hit.material.diffuse_specular.rgb;// 乘上衰减
345     }
346
347     color += segment.throughput * (diffuse * hit.material.albedo.x + specular * hit.material.albedo.y);
348
349     if (hit.material.albedo.z > 0.0 && stackSize < MAX_STACK_SIZE) {
350         vec3 next_dir = normalize(reflect(segment.direction, n));
351         vec3 offset = dot(next_dir, n) < 0.0 ? -n * MIN_EPSILON : n * MIN_EPSILON;
352         vec3 next_origin = p + offset;
353         vec3 next_throughput = segment.throughput * hit.material.albedo.z;
354         stack[stackSize++] = PathSegment(next_origin, next_dir, next_throughput, segment.depth + 1);
355     }
356
357     if (hit.material.albedo.w > 0.0 && stackSize < MAX_STACK_SIZE) {
358         vec3 refract_dir = custom_refract(segment.direction, n, hit.material.refractive.x, 1.0);
359         if (length(refract_dir) > 0.0001) {
360             refract_dir = normalize(refract_dir);
361             vec3 offset = dot(refract_dir, n) < 0.0 ? -n * MIN_EPSILON : n * MIN_EPSILON;
362             vec3 next_origin = p + offset;
363             vec3 next_throughput = segment.throughput * hit.material.albedo.w;
364             stack[stackSize++] = PathSegment(next_origin, refract_dir, next_throughput, segment.depth + 1);
365         }
366     }
}

```

所有模型的配置参数均在config.hpp中定义便于调整：

```

36 // Dragon 配置
37 constexpr Material dragonMaterial = { {0.9f, 0.1f, 0.0f, 0.0f}, {0.013072f, 0.078431f, 0.143791f, 10.0f}, {1.0f, 0.0f, 0.0f
38 constexpr glm::vec3 dragonScale = { 0.7f, 0.7f, 0.7f };
39 constexpr glm::vec3 dragonRotation = { 0.0f, 110.0f, 0.0f };
40 //<Translation x = "-4.15" y = "-3.0" z = "-7.0" / >
41 constexpr glm::vec3 dragonTranslation = { -2.5f, -2.175f, -4.45f };
42 constexpr int dragonNormalInterpolation = 1;
43 Model Dragon(dragonMaterial, dragonNormalInterpolation);
44 ModelInfo dragonInfo(Dragon, "assets/dragon-mesh.obj", dragonScale, dragonRotation, dragonTranslation);
45
46 // Venus 配置
47 constexpr Material venusMaterial = { {0.0, 10.0, 0.8, 0.0}, {1.0, 1.0, 1.0, 1425.0}, {1.0, 0.0, 0.0, 0.0} };
48 constexpr glm::vec3 venusScale = { 5.5f, 5.5f, 5.5f };
49 constexpr glm::vec3 venusRotation = { 0.0f, 0.0f, 0.0f };
50 constexpr glm::vec3 venusTranslation = { -1.0f, 2.0f, -19.0f };
51 constexpr int venusNormalInterpolation = 1;
52 Model Venus(venusMaterial, venusNormalInterpolation);
53 ModelInfo venusInfo(Venus, "assets/venus-mesh.obj", venusScale, venusRotation, venusTranslation);
54
55 // FudanLogo 配置
56 constexpr Material fudanLogoMaterial = { {0.7f, 0.7f, 0.0f, 0.0f}, {0.515625f, 0.3984275f, 0.32421875f, 12.0f }, {1.0f, 0.
57 constexpr glm::vec3 fudanLogoScale = { 0.4f, 0.4f, 0.4f };
58 constexpr glm::vec3 fudanLogoRotation = { 90.0f, 0.0f, 0.0f };
59 constexpr glm::vec3 fudanLogoTranslation = { 0.85f, 1.95f, -4.15f };
60 constexpr int fudanLogoNormalInterpolation = 0;
61 Model FudanLogo(fudanLogoMaterial, fudanLogoNormalInterpolation);
62 ModelInfo fudanLogoInfo(FudanLogo, "assets/fudanlogo-mesh.obj", fudanLogoScale, fudanLogoRotation, fudanLogoTranslation);
63

```

Part3 Issues and Resolutions

1. 面片渲染时缺失

问题：在导入三角形较小的高模如venus后，出现了一些面片丢失的问题如下图所示。



解决：经过排查确定，模型的三角形数组在上传到GPU后并没有出现数据丢失，故推测是由浮点运算精度导致光线三角形求交时误将相交的面计算为不相交。最终确定问题出现在shader中一个参数的配置上：

```
const float MIN_EPSILON = 0.001;
```

`MIN_EPSILON` 是一个数值稳定性参数，其核心作用是避免光线与表面的自相交 (self-intersection) 问题，在光线与物体相交后，为了避免后续生成的反射/折射/阴影光线从表面出发时因浮点精度问题误判为与同一物体相交，会将光线起点沿法线方向偏移一个 `MIN_EPSILON` 的微小距离；

```
349     if (hit.material.albedo.z > 0.0 && stackSize < MAX_STACK_SIZE) {  
350         vec3 next_dir = normalize(reflect(segment.direction, n));  
351         vec3 offset = dot(next_dir, n) < 0.0 ? -n * MIN_EPSILON : n * MIN_EPSILON;  
352         vec3 next_origin = p + offset;  
353         vec3 next_throughput = segment.throughput * hit.material.albedo.z;  
354         stack[stackSize++] = PathSegment(next_origin, next_dir, next_throughput, segment.depth + 1);  
355     }
```

当 `MIN_EPSILON` 过大时，光线起点会被推离表面一个较大的距离。而在venus这种高模模型（有复杂的mesh、锐利边缘）中，偏移可能导致光线起点“穿透”到模型的另一侧，跳过原本应渲染的几何部分；也有可能偏移方向可能与相邻面的法线方向冲突，导致错误遮挡。也就引起了问题中了一些面片缺失的效果。

于是将默认值的 `1e-4` 调整到更小的 `1e-7`，却发现出现了新的问题--远距离物体出现黑点，如下图所示：



分析原因是当 `MIN_EPSILON` 过小时（低于了float的精度），就会导致偏移量不足，光线起点可能仍位于表面“内部”，造成了自相交，这些黑点就是错误绘制的阴影。

我的解决方案：

先将 `MIN_EPSILON` 调整到比较合理的值 `1e-5`，

使用动态 `EPSILON`，根据着色点距离计算，

近距离：使用小偏移，保持几何精度。

远距离：适当增大偏移，避免浮点精度问题。

```
float getDynamicEpsilon(float distance) {
    // 基础偏移 + 距离比例因子 * 着色点距离摄像机距离
    return MIN_EPSILON + 0.0001 * distance;
}
```

最终效果见文档中展示的效果图

2. 多模型管理--GLSL 不支持嵌套数组作为 SSBO中的成员变量

问题：为了实现管理多个模型的不同信息：如该模型的三角形数组，其AABB参数，是否开启法向平滑等；我需要将不同模型的信息打包到结构体 `Model` 中，再将信息通过SSBO上传至GPU，但是GLSL不支持这种嵌套数组结构，因此无法直接在着色器中声明类似 C++ 中 `Triangle models[][]` 的结构

```
// GLSL 不支持 嵌套数组
struct Model {
    Triangle triangles[];
    glm::vec4 bboxMin;
    glm::vec4 bboxMax;
    bool normalInterpolation;
    Material material;
};

Model models[];
```

解决：扁平化存储 + 索引管理

将所有模型的三角形合并为一个全局三角形数组

```
layout(std140, binding = 5) readonly buffer TrianglesSSBO {
    Triangle triangles[];
};
```

所有模型共享这一个大数组，每个模型只需要记录自己在该数组中的起始索引和三角形数量

定义模型结构体，包含起始索引与三角形数量

```
struct Model
{
    // 为了满足std140 布局的对齐规则将前三个参数整合到params0中
    ivec4 params0; // x=startIndex, y=count, z=normalInterpolation (0/1),
    w=padding
    //int startIndex:起始索引
    //int count:三角形数量
    //bool normalInterpolation:是否启用表面平滑
    vec4 bboxMin;
    vec4 bboxMax;
    Material material;
};
```

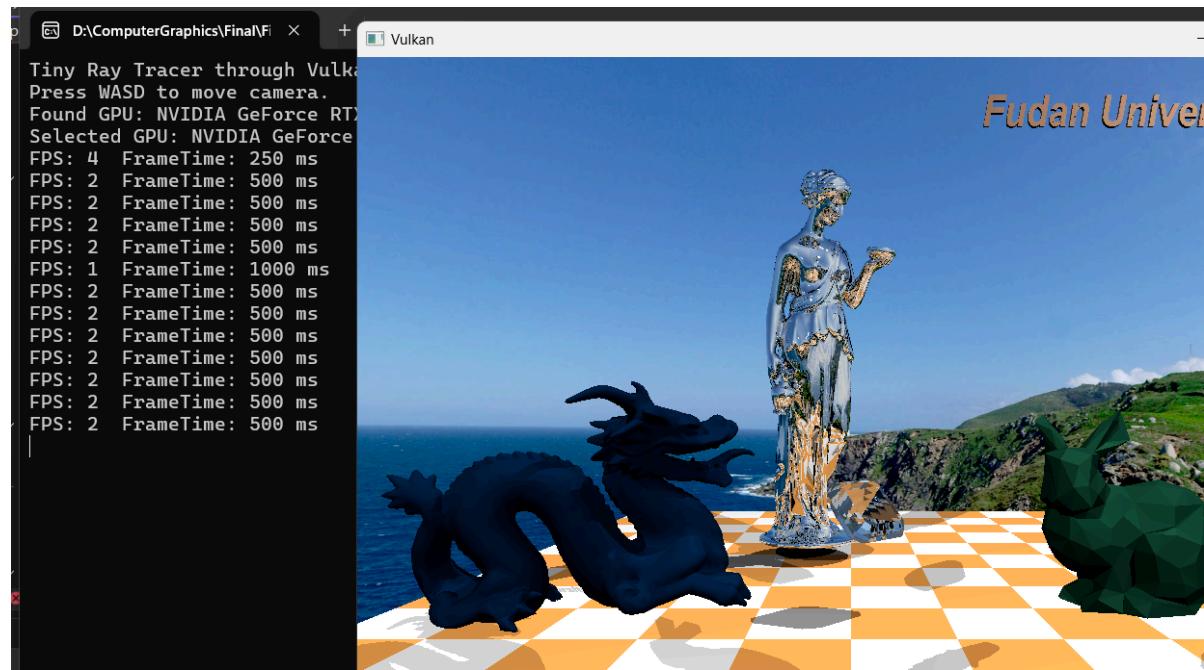
通过这种管理方式，在场景遍历时就可以以如下形式遍历每个模型的三角形了。

```
for(int i = 0; i < models.length(); i++){
    if(ray_aabb_intersect(orig, dir, invDir, models[i].bboxMin.xyz,
models[i].bboxMax.xyz)){
        int startindex = int(models[i].params0.x);
        int count = int(models[i].params0.y);
        int endindex = startindex + count;
        int normalinterpolation = int(models[i].params0.z);
        for (int j = startindex; j < endindex; j++){
            .....
        }
    }
}
```

3. 针对多面高模的渲染速度较慢：AABB 展开

问题：导入维纳斯这种有上万个三角面的模型后，遍历效率非常低下，仅依赖单个模型级别的 AABB 加速结构已无法显著提升效率，因为每次光线进入该包围盒后仍需遍历其全部三角形，无法有效剔除大量不可能相交的几何体。

优化前帧率约为每秒2帧：



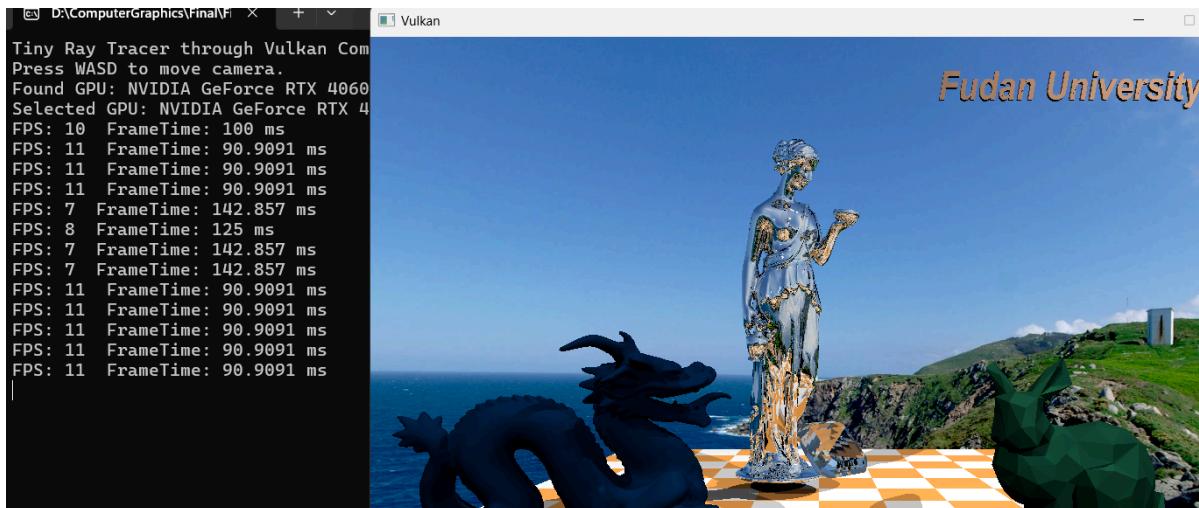
解决：本项目采用了一种简化的，基于批次（batch）的细分包围盒策略：将每个模型内部的三角形每 `batch_size` 个划分为一个批次，并为每个批次单独构建一个局部的 AABB 包围盒。这样，光线在进入模型整体包围盒后，会进一步判断是否与某个三角形批次的包围盒相交，只有通过该粗略剔除阶段的批次才会被进一步展开并执行精确的三角形级求交运算。

```

1502 // 常量级别AABB展开
1503 std::vector<std::vector<Triangle>> batches;
1504 const size_t batchSize = 64;
1505 for (size_t i = 0; i < modelTriangles.size(); i += batchSize) {
1506     std::vector<Triangle>::iterator next = modelTriangles.begin() + std::min(i + batchSize, modelTriangles.size());
1507     batches.emplace_back(modelTriangles.begin() + i, next);
1508 }
1509 for (size_t batchIdx = 0; batchIdx < batches.size(); ++batchIdx) {
1510     Model tempmodel(model);
1511     // 计算当前 batch 的包围盒
1512     for (const Triangle& tri : batches[batchIdx]) {
1513         tempmodel.bboxMin = glm::min(tempmodel.bboxMin, glm::min(tri.v0, glm::min(tri.v1, tri.v2)));
1514         tempmodel.bboxMax = glm::max(tempmodel.bboxMax, glm::max(tri.v0, glm::max(tri.v1, tri.v2)));
1515     }
1516     int startid = static_cast<int>(triangles.size());
1517     triangles.insert(triangles.end(), batches[batchIdx].begin(), batches[batchIdx].end());
1518     tempmodel.params0.x = startid; // 起始索引
1519     tempmodel.params0.y = static_cast<int>(batches[batchIdx].size()); // 三角形数量
1520     models.push_back(tempmodel);
1521 }

```

选择每64个三角形面作为一个包围盒batch优化效果,平均帧率提升近5倍



4. 全反射逻辑修正

问题：原有的实现方案中发生全反射时，逻辑是向 $(1, 0, 0)$ 方向发射一条折射光线。导致宝石上发生全反射的面的视觉效果不符合物理。

```

    vec3 custom_refract(vec3 I, vec3 N, float eta_out, float eta_in) {
        bool isEntering = dot(I, N) < 0.0;
        vec3 faceNormal = isEntering ? N : -N;
        float cosi = clamp(dot(-I, faceNormal), 0.0, 1.0);
        float eta = isEntering ? eta_in / eta_out : eta_out / eta_in;
        float sint2 = eta * eta * (1.0 - cosi * cosi);
        if (sint2 > 1.0) {
            return vec3(1.0, 0.0, 0.0); // 非物理的简化处理
        }
        float k = sqrt(1.0 - sint2);
        vec3 refractDir = eta * I + (eta * cosi - k) * faceNormal;
        return normalize(refractDir);
    }

```

解决：在 `custom_refract()` 函数处，发生全反射时返回0向量，由 `cast_ray()` 函数做特殊处理处理，发生全反射时，仅保留一条全反射的光线进行采样。

```

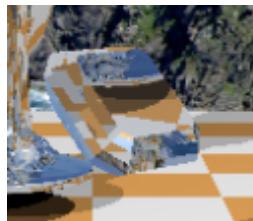
355     if (hit.material.albedo.w > 0.0 && stackSize < MAX_STACK_SIZE) {
356         vec3 refract_dir = custom_refract(segment.direction, n, hit.material.refractive.x, 1.0);
357         if (length(refract_dir) > 0.0001) {
358             refract_dir = normalize(refract_dir);
359             vec3 offset = dot(refract_dir, n) < 0.0 ? -n * MIN_EPSILON : n * MIN_EPSILON;
360             vec3 next_origin = p + offset;
361             vec3 next_throughput = segment.throughput * hit.material.albedo.w;
362             stack[stackSize++] = PathSegment(next_origin, refract_dir, next_throughput, segment.depth + 1);
363         }
364     } else {
365         //处理全反射
366         vec3 reflect_dir = normalize(reflect(segment.direction, n));
367         vec3 offset = dot(reflect_dir, n) < 0.0 ? -n * MIN_EPSILON : n * MIN_EPSILON;
368         vec3 next_origin = p + offset;
369         vec3 next_throughput = segment.throughput * 1;
370         stack[stackSize++] = PathSegment(next_origin, reflect_dir, next_throughput, segment.depth + 1);
371         skip_reflect = true;
372     }
373 }
374
375 if (hit.material.albedo.z > 0.0 && stackSize < MAX_STACK_SIZE && skip_reflect == false) {
376     vec3 next_dir = normalize(reflect(segment.direction, n));
377     vec3 offset = dot(next_dir, n) < 0.0 ? -n * MIN_EPSILON : n * MIN_EPSILON;
378     vec3 next_origin = p + offset;
379     vec3 next_throughput = segment.throughput * hit.material.albedo.z;
380     stack[stackSize++] = PathSegment(next_origin, next_dir, next_throughput, segment.depth + 1);
381 }
382
383
384 return clamp(color, vec3(0.0), vec3(1.0));
385 }
386

```

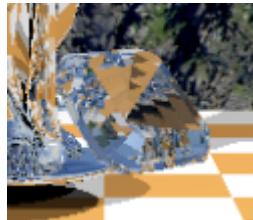
5. 宝石渲染效果不一致

问题：在对比最中效果时，发现宝石的上表面渲染效果与参考图不同，

(参考图)



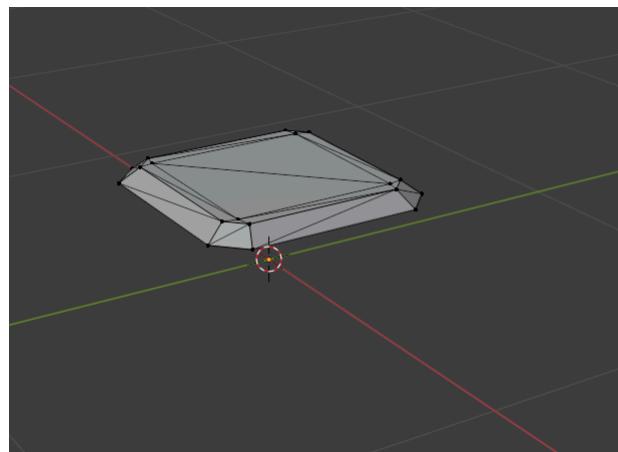
(我的实现)



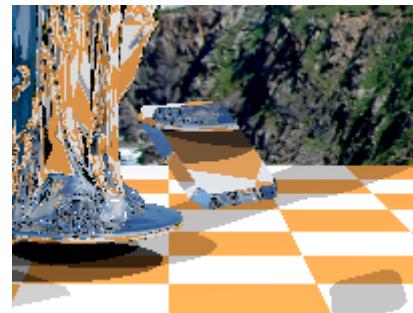
检查全反射逻辑、光线offsetting设置，非递归实现多次折射逻辑均没有问题。

从物理直觉上来分析，宝石的上表面的成像不应该像参考图那样平整。应该需要反映出宝石背部的结构。

于是将宝石的模型下侧顶点去除后做了实验：



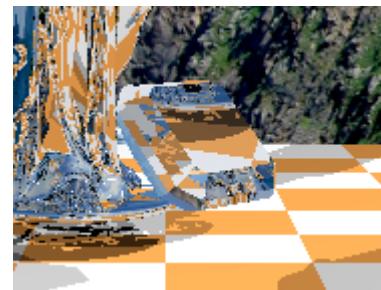
得出了和参考图一致的结果



故我认为参考图可能没有正确处理光线在宝石内部多次折射反射形成的效果，仅计算了在宝石上表面的一次折射。

造成这种问题的可能原因是参考图使用了backface culling来做加速，剔除了宝石背面的结构，导致raytracing时没有计算到。

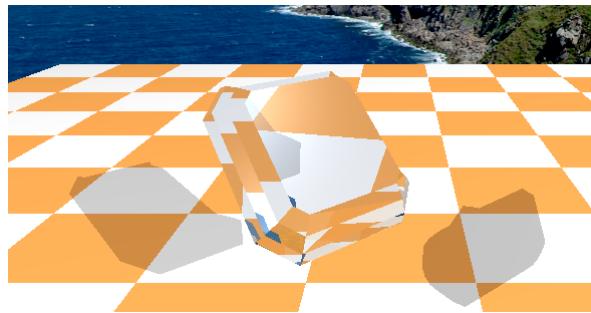
于是我在我的项目中尝试实现了一个粗糙的backface culling，剔除掉所有背朝摄像机的面也得到了类似参考图的效果：



但是我认为在ray tracing中做背面剔除是不合理的，即透明的宝石需要呈现出其背部的结构对光线的影响效果，如下图所示



而不应该是这样：



故最后我保留了我的实现效果，不做修改。



| end