

A Real-Time Path Tracing Renderer

基于Optix的实时路径追踪渲染器

现代图形学架构下的实时路径追踪算法实现



CHAPTER 1 项目简介

CHAPTER 2 算法基础

项目介绍加载中...
CHAPTER 3 项目实施

CHAPTER 4 效果展示

CHAPTER 5 总结



项目简介

What is our work?

应用介绍

渲染器是什么

渲染器(Renderer)是一个将计算机中存储的图元信息转换为2D图像的计算机软件或程序接口。它完成将3D物体绘制到屏幕上的任务。

路径追踪渲染器是什么

我们现在所用的大部分都是基于光栅化管线的渲染器，而路径追踪算法是对传统光线追踪算法的改进，借助这种算法，路径追踪渲染器能够渲染出基于物理的、真实感十足的图像。

为何需要实时渲染器

现代图形学算法使得离线渲染器可以生成十分逼真的图片，以及模拟自然界各种复杂的物理效果，但往往它们生成这样一张高精度的图像需要的时间与计算资源是巨大的。实时渲染器牺牲了一部分真实性，能够满足用户对强交互性的需求，同时还能够保证短时间内渲染出来的图片质量。

实时路径追踪的难点

将光线追踪实时化一直是图形学领域的难题，使用路径追踪算法，每生成一张图像计算机就要发射数百万根光线，每根光线在场景上要经历数次碰撞，每次碰撞还要计算碰撞点的颜色信息。如此巨大的计算量使得光线追踪虽诞生五十余年仍未得到大规模普及。

永恒的课题

在计算机图形学的渲染领域中，有一门课题就像是圣杯一般，吸引着一代又一代人。可以说，现代图形学实时渲染的研究的基本方向，就是这门课题——实时全局光照。

而渲染器中最为关键的算法，也是让画面变得真实的最关键的步骤，就是全局光照算法的设计实现。如何做到实时高效又真实可靠，这是一个非常难的问题，也是实时渲染算法设计的最核心。



D5商业级渲染器最新版本的
新一代GI（全局光照）方案

我们的工作

而我们的工作，就是通过路径追踪方式设计实现实时全局光照方案，编写一个可以实现全局光照效果的实时渲染器。

通过查阅资料，我们初步确定了通过Nvidia的RTX光线追踪加速方案与PBR+IBL算法结合，实现真正意义上的实时全局光照。

以及通过次时代的实时光线追踪算法实现全局光照。

以上工作全部通过纯代码实现。





开发工具

- CMake
- VS2019
- NVIDIA OptiX SDK



编程语言

- C++/C
- GLSL
- OpenGL
- CUDA C++

开发环境

Characteristic Overview



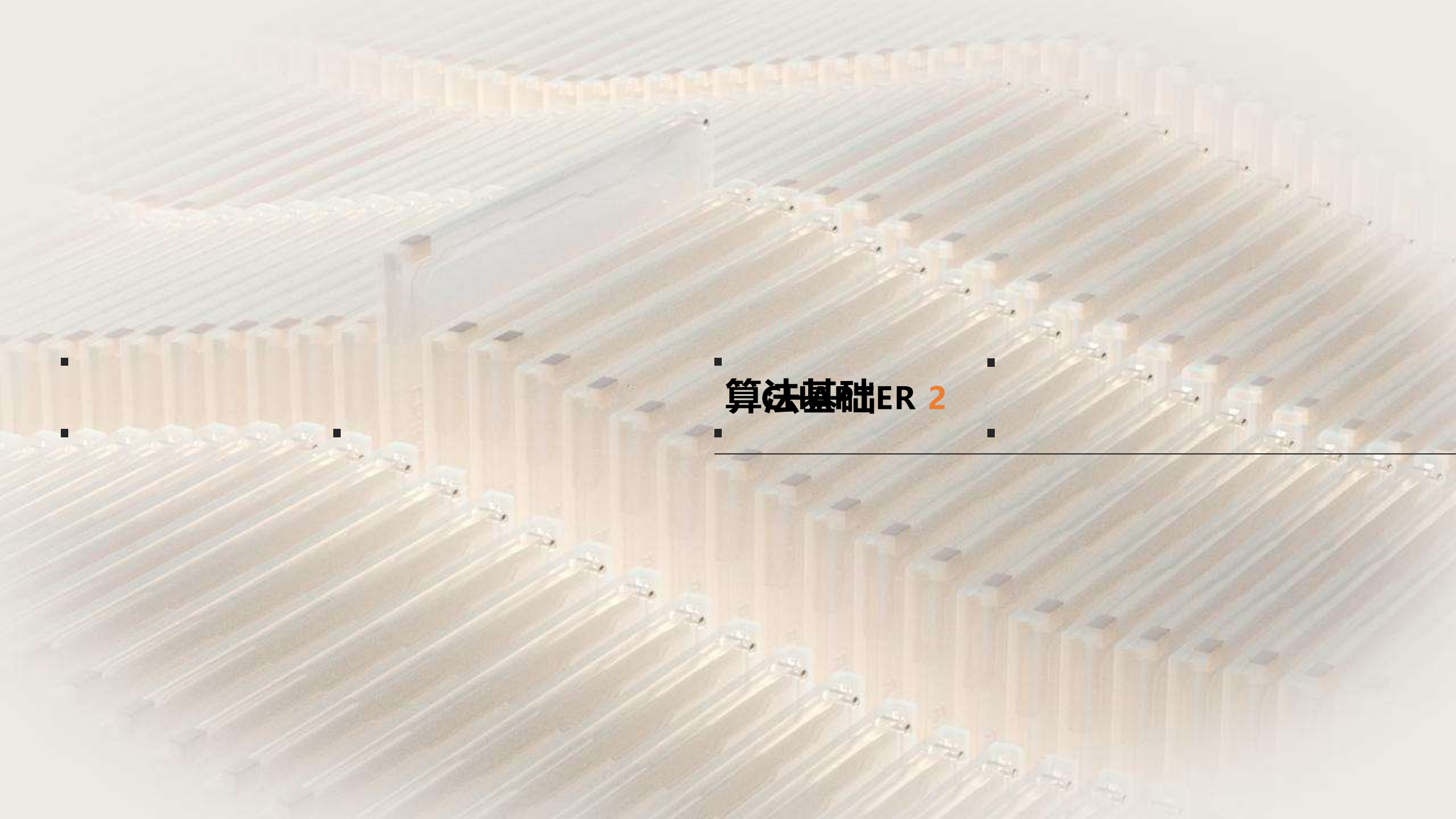
- glad
- glfw
- glm



- ImGui
- tiny_obj_loader
- stb_image

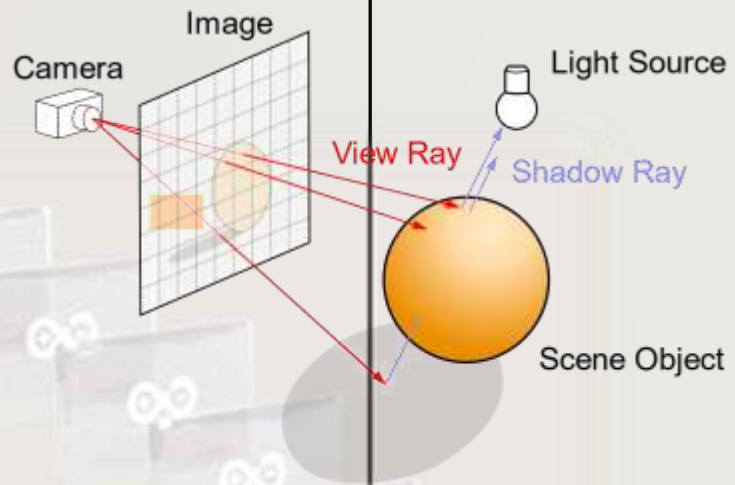
第三方库使用

Characteristic Overview



算法基础ER 2

光线传输——Light Transportation



在现实世界中，光从光源出发，经过多次与物体的碰撞后进入人眼。而在计算机渲染的过程中，我们从虚拟相机处向场景发射一束光，递归地追踪其与场景物体的碰撞过程，直至光线发现光源。根据亥姆霍兹互易原理，在不考虑环境中复杂物理因子的情况下，上述的两种过程可以看作是等价的。

光线传输算法的本质，便是构建相机与光源之间的这样一条通路，我们渲染所需要的各种信息都包括在这条路径中。

光线追踪——Ray Tracing



经典光线追踪算法产生的光线是从相机到光源的，我们采用的也是这种方式。光线从相机发射后，穿过虚拟屏幕的某一个像素，与场景中物体碰撞，然后返回该物体的颜色。若我们认为这个物体是可再次反射光线的，那么它就继续向前行进，直到找到光源为止。

区别于传统图形学的光栅化成像方法，光线追踪方式通过模拟光线的物理行为来生成更加逼真的图像，但同时它给计算机带来的计算量也是巨大的。

左图为我们使用我们的渲染器，让光线在两面镜子之间弹射100次的渲染结果。

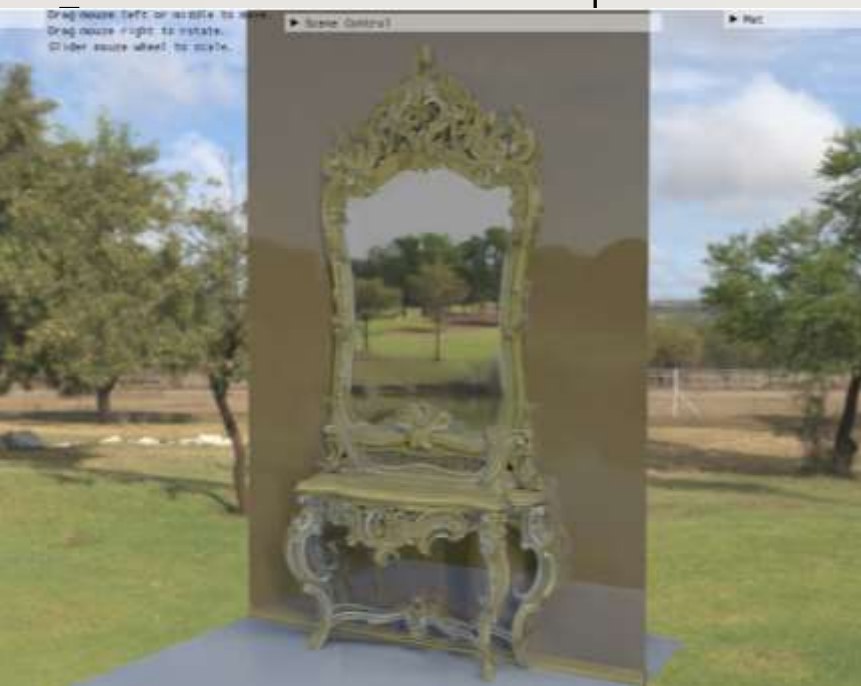
路径追踪——Path Tracing

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_{\Omega} f_r(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t) (\omega_i \cdot \mathbf{n}) d\omega_i$$

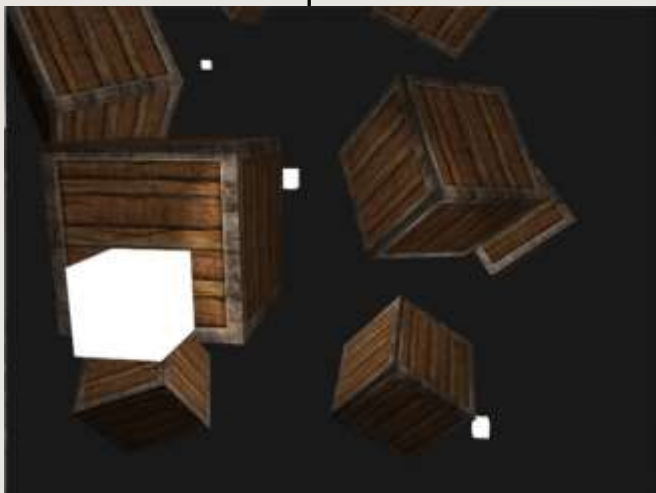
经典光线追踪算法仅仅将光线看作是一条向相机传送颜色信息的路径，这显然是不符合自然界规律的。1986年，*James Kajiya*提出了如上图所示的渲染方程，将计算机渲染技术与真实物理世界联系起来。从表达式的递归形式来看，想要在计算机中得到该积分的解析解是基本不可能的，为了得到它的数值解，*Kajiya*基于蒙特卡洛积分提出了路径追踪算法，并且这种方式被证明在统计学上是无偏的。

路径追踪渲染器的根本任务，便是解如上的渲染方程，并把光线的物理信息转换为用户所见到的各个像素。

左图为我们的渲染器的实机效果，可以看到，从相机发出的每根光线忠实地记录了它们所观测到的信息。



SELECTING FILE

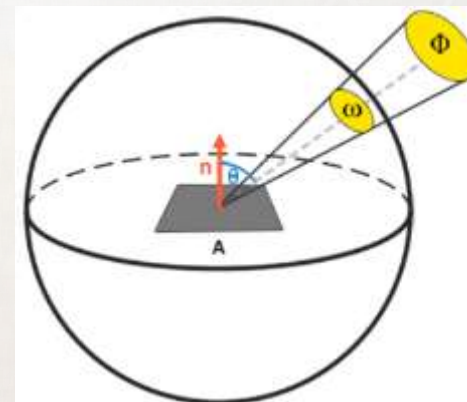
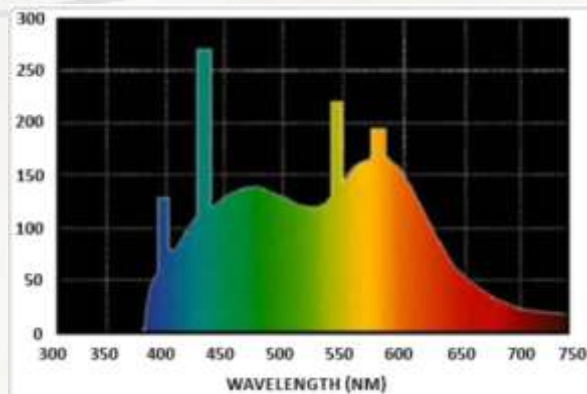


这是传统的BlinnPhong光照模型



这是辐射度光照模型

更真实的光照模型——辐射度模型



通过引入辐射度量学，我们可以得到基于物理的光照模型，**Physically Based Rendering(PBR)**.

在这个算法中，计算始终处于高动态范围，可以使用更高精度与范围的光照信息，并且通过半球采样，我们可以得到更加精准的片元能量用于光照计算

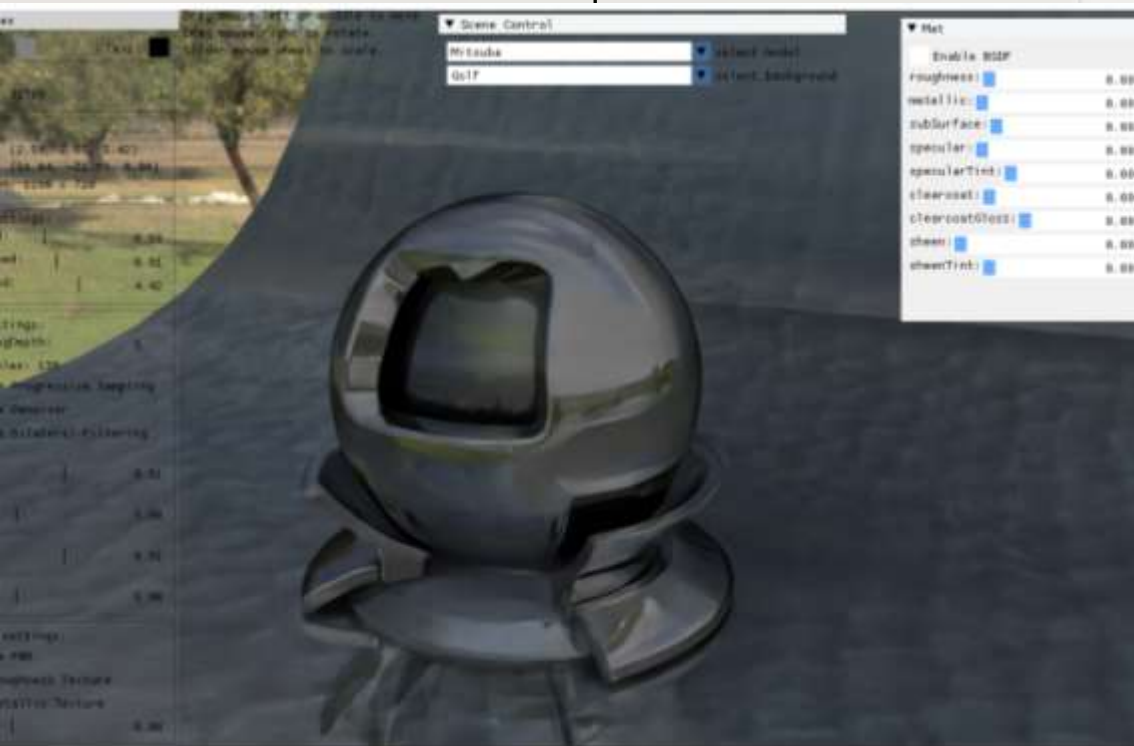
(左图为辐射度波长，右图为片元半球能量模型)

基于物理的渲染——Physically Based Rendering(PBR)

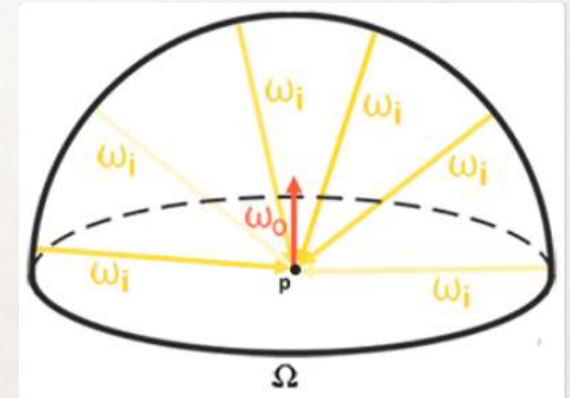
$$L_o(p, w_o) = \int_{\Omega} (k_d \frac{c}{\pi} + k_s \frac{DGF}{4(w_o \cdot n)(w_i \cdot n)}) L_i(p, w_i) (w_i \cdot n) dw_i$$

如果说路径追踪的工作是解渲染方程，那么 *PBR* 的任务就是求上图所示的反射度方程。虽然是基于物理，但它的部分参数的提供很大一部分都来自于前人所摸索出来的经验公式。基于物理的方式使渲染器生成的图片与真实图片的差距大大减小，甚至可以做到以假乱真的地步，但也正因为它是基于物理，*PBR* 模型在给渲染器增加了更大的计算负担的同时不可避免地也给渲染器的使用者暴露出更多复杂的参数。

我们提供了三种光照着色模型供用户选择。



基于图像的照明——Image Based Illumination(IBL)



全局光照是一个非常昂贵的效果，因为要考虑渲染的所有物体对于彼此的光照影响，这个计算量是非常大的。尽管路径追踪算法为全局光照提供了一个完美的解决方案，但对于实时光追渲染器来说，要使用路径追踪遍历场景中所有的光源，这种代价无疑是不可承受的。*IBL*的思路，便是通过采样一张经过预计算的环境纹理，从而间接达到全局光照的效果。

(左图为HDR贴图转换的Cubemap，右图为预计算卷积采样的示意图)

双边滤波——Bilateral Filtering

$$w(i, j, k, l) = \exp \left(-\frac{(i - k)^2 + (j - l)^2}{2\sigma_d^2} - \frac{\|I(i, j) - I(k, l)\|^2}{2\sigma_r^2} \right)$$

高斯滤波是一种抑制图像噪声、平滑图像的有效手段，但使用高斯滤波核的图片滤波往往会使图片的边缘信息丢失，造成过度模糊的效果。双边滤波在高斯滤波的基础上进行了改进，不仅考虑了空间域的颜色信息，还考虑了像素域上的信息，因此其滤波后的图像可以较完整地保留边界信息。

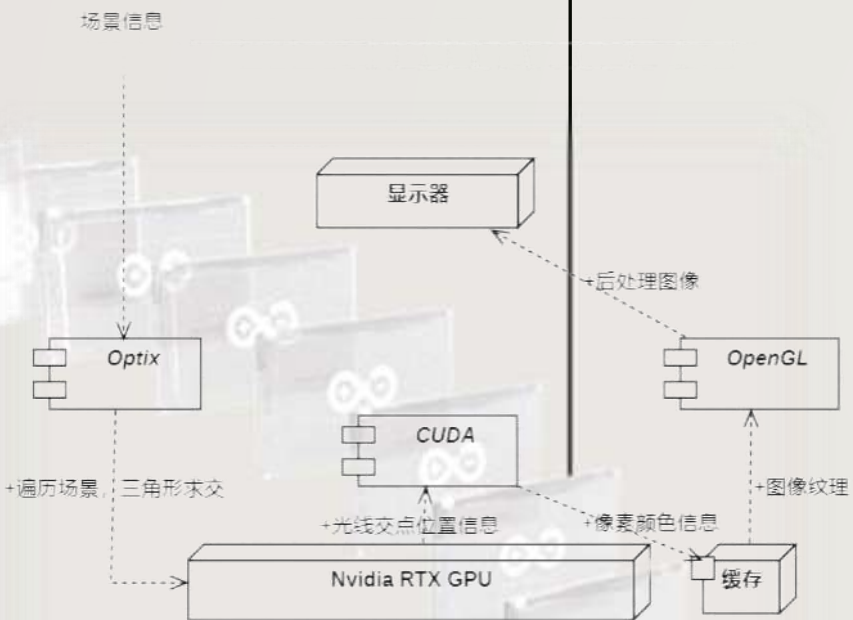




项目实现ER 3

整体架构

项目基于Nvidia RTX CPU，使用Optix与CUDA语言建立CPU与GPU端的数据通信，GPU端产出的图像最终送入OpenGL执行后处理。





PH 2018

ANNOUNCING QUANTUM

WORLD'S FIRST RAY TRACING GPU

Up to 10x
Up to 10x
Up to 5x
Up to 10x

IPS
Ops/sec
NVLlink

在SIGGRAPH2018上, *Nvidia*发布了基于Turing架构的RTX系列显卡, 该架构新增加了专用于光线追踪计算的RTCore单元, 用其专有的硬件来加速传统光线追踪算法中的光线在BVH加速结构中的遍历, 以及光线和三角形的求交测试。该系列显卡的问世, 首次使实时光线追踪成为可能。这也是我们的项目采用RTX GPU进行硬件加速的原因。

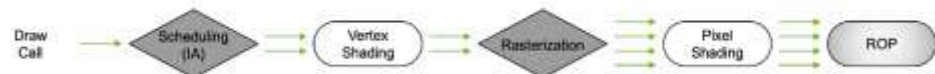
Optix 7 SDK

MODERN RAY TRACING

Ray Tracing

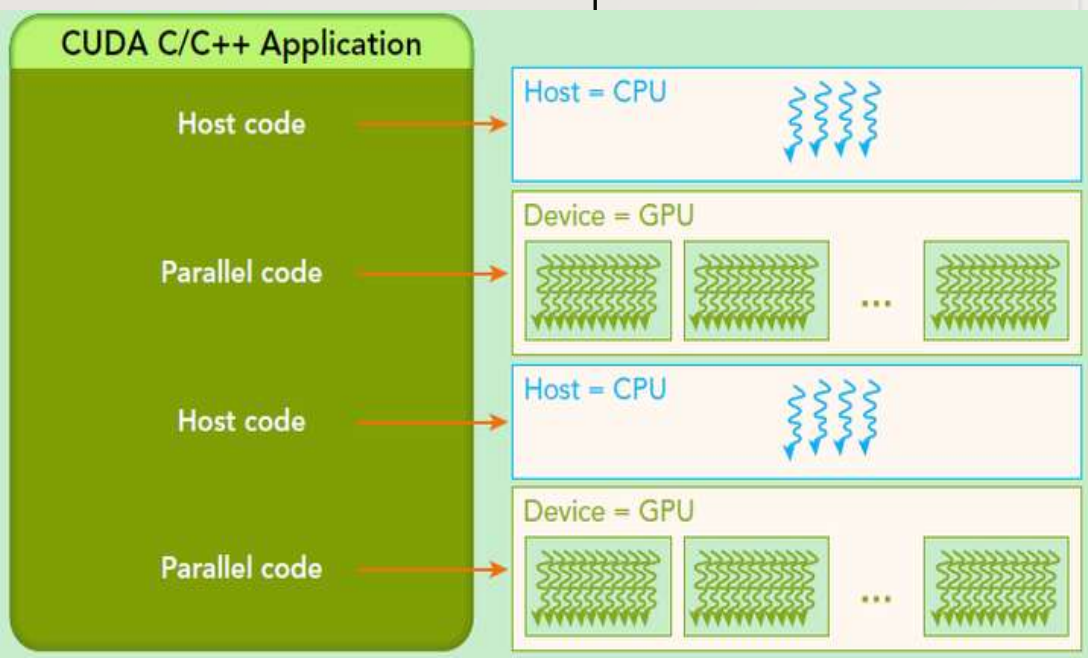


RASTER



在Nvidia的Optix7发布前，DirectX12和Vulkan就已经分别发布了它们的光线追踪API与技术标准，区别于传统的光栅化渲染管线，光线追踪渲染管线正逐步发展成为现代图形学的又一硬件渲染管线。Optix是一个基于GPU实现高性能光线追踪的应用程序框架，它为加速光线追踪算法提供了一个简单、递归且灵活的管线，这也是实现一个实时的路径追踪渲染器所需要的。

CUDA



统一计算设备架构 (*Compute Unified Device Architecture, CUDA*), 是Nvidia推出的通用设备并行计算架构。借助它所提供的GPU编程接口, 我们的应用能够更加高效地利用现代GPU的强大并行计算能力。

数据结构

```
struct Frame
{
    float4* colorBuffer;
    glm::vec2 size;
    int frameCount;
    int maxTracingDepth;
    float specular;
    float roughness;
    float metallic;
    float subSurface;
    float specularTint;
    float sheen;
    float sheenTint;
    float clearcoat;
    float clearcoatGloss;
    bool enableProgressiveRefinement;
    bool enableDenoiser;
    bool enablePBR;
    bool useRoughnessTexture;
    bool useMetallicTexture;
    bool enableBSDF;
};
```

```
struct CameraProp
{
    glm::vec3 position;
    glm::vec3 direction;
    glm::vec3 horizontal;
    glm::vec3 vertical;
};
```

```
struct LaunchParams
{
    Frame frame;
    CameraProp camera;
    OptixTraversableHandle traversable;
};
```

+

```
struct TriangleMeshSBTData {
    glm::vec3 color;
    glm::vec3 specular;
    glm::vec3 emission;
    glm::vec3* vertex;
    glm::vec3* normal;
    glm::vec2* texcoord;
    glm::ivec3* index;
    bool hasATexture;
    bool hasRTTexture;
    bool hasMTexture;
    bool hasETexture;
    cudaTextureObject_t texture;
    cudaTextureObject_t roughnessT;
    cudaTextureObject_t metallicT;
    cudaTextureObject_t emissionT;
};
```

数据由CPU端读取，由Optix
送入GPU

CUDA

- __anyhit_radiance()
- __anyhit_shadow()
- __closesthit_radiance()
- __closesthit_shadow()
- __miss_radiance()
- __miss_shadow()
- __raygen_renderFrame()

```
glm::vec3 result = glm::vec3(0);
for (int i = 0; i < optixLaunchParams.frame.maxTracingDepth; i++) {
    uint32_t u0, u1;
    packPointer(&prd, u0, u1);
    optixTrace(optixLaunchParams.traversable, convert_vec3(rayOri), convert_vec3(rayDir), 0.01f, 1e16f, 0.0f, OptixVisibilityMask::
        OPTIX_RAY_FLAG_DISABLE_ANYHIT, RADIANCE_RAY_TYPE, RAY_TYPE_COUNT, RADIANCE_RAY_TYPE, u0, u1);
    result += prd.radiance * prd.lastEnergy;
    if (prd.energy.x == 0 || prd.energy.y == 0 || prd.energy.z == 0)
        break;
    rayOri = prd.origin;
    rayDir = prd.direction;
}
```

GPU着色器程序一览，由这些函数执行光线追踪相关内容

```
prd.origin = newOri + 1e-3f * N;
prd.lastEnergy = prd.energy;
// using Russian Roulette to determine which ray type the generated ray is.
float roulette = getRandomFloat(prd.seed); // just a random float number
float diffChance = glm::saturate(kd.x + kd.y + kd.z) / 3.0f + 0.0000001f;
float coatChance = glm::saturate(optixLaunchParams.frame.clearcoat * optixLaunchParams.frame.clearcoat * 0.25f);
float specChance = glm::saturate((1.0f - diffChance - coatChance) + 0.0000001f);
if (roulette < specChance) { // shading model is specular
    glm::vec3 F = calcF(F0, VdotH);

    float G = calcG(NdotL, NdotV, alpha2);

    prd.direction = sdir;
    prd.energy *= sResult((1.0f / specChance) * Fs * Gs / NdotL / NdotV * LdotV * PI);
}
else if (roulette < specChance + diffChance) // shading model is diffuse
{
    glm::vec3 ddir = sampleHemisphere(N, prd.seed);
    prd.direction = ddir;

    float roulette2 = getRandomFloat(prd.seed) * 0.99998f + 0.00001f;
    if (roulette2 < optixLaunchParams.frame.subSurface)
        prd.energy *= sResult((1.0f / diffChance / roulette2) * Cotlin * sss / kd.x + Fsheen * PI);
    else
        prd.energy *= sResult((1.0f / diffChance / (1.0f - roulette2)) * Cotlin * Fd / kd.x + Fsheen * PI);
}
else if (roulette < specChance + diffChance + coatChance) // shading model is clearcoat
{
    prd.direction = sdir;
    prd.energy *= sResult((1.0f / coatChance) * clearcoat / Ds / NdotL / NdotV * LdotV * PI);
}
else // dropout
{
    prd.energy = glm::vec3(0);
    prd.radiance = emission;
}
```

SELECTIN

PBR着色部分，路径追踪算法的实现

光线追踪算法的核心，递归式地产生光线

```
static __forceinline__ __device__ unsigned int wang_hash(unsigned int seed) {
    seed = (seed ^ 61) ^ (seed >> 16);
    seed *= 9;
    seed = seed ^ (seed >> 4);
    seed *= 0x27d4eb2d;
    seed = seed ^ (seed >> 15);
    return seed;
}

// 产生0-1的随机浮点数
static __forceinline__ __device__ float getRandomFloat(unsigned int seed) {
    return (wang_hash(seed) & 0xFFFFFFFF) / 16777216.0f;
}

static __forceinline__ __device__ float RadicalInverse(unsigned int bits) {
    bits = (bits << 16u) | (bits >> 16u);
    bits = ((bits & 0x55555555) << 1u) | ((bits & 0xAAAAAAAA) >> 1u);
    bits = ((bits & 0x33333333) << 2u) | ((bits & 0xCCCCCCCC) >> 2u);
    bits = ((bits & 0x0F0F0F0F) << 4u) | ((bits & 0xF0F0F0F0) >> 4u);
    bits = ((bits & 0x00FF00FF) << 8u) | ((bits & 0xFF00FF00) >> 8u);
    return float(bits) * 2.3283064365386963e-10;
}
```

采用Thomas Wang的Wang_hash算法生成随机浮点数

CUDA

```
vec2 delta = vec2(0.001f, 0.001f) * vec2(0, y_radius);
vec2 delta2 = vec2(0.001f, 0.001f) * vec2(x_radius, 0);
vec4 col = texture(screenTexture, TexCoords);
vec4 col0a = texture(screenTexture, TexCoords - delta);
vec4 col0b = texture(screenTexture, TexCoords + delta);
vec4 col1a = texture(screenTexture, TexCoords - 2.0f * delta);
vec4 col1b = texture(screenTexture, TexCoords + 2.0f * delta);
vec4 col2a = texture(screenTexture, TexCoords - 3.0f * delta);
vec4 col2b = texture(screenTexture, TexCoords + 3.0f * delta);

vec4 col2 = texture(screenTexture, TexCoords);
vec4 col20a = texture(screenTexture, TexCoords - delta2);
vec4 col20b = texture(screenTexture, TexCoords + delta2);
vec4 col21a = texture(screenTexture, TexCoords - 2.0f * delta2);
vec4 col21b = texture(screenTexture, TexCoords + 2.0f * delta2);
vec4 col22a = texture(screenTexture, TexCoords - 3.0f * delta2);
vec4 col22b = texture(screenTexture, TexCoords + 3.0f * delta2);

float w = 0.37004405286;
float w0a = compareColor(col, col0a, x_factor) * 0.31718061674;
float w0b = compareColor(col, col0b, x_factor) * 0.31718061674;
float w1a = compareColor(col, col1a, x_factor) * 0.19823788546;
float w1b = compareColor(col, col1b, x_factor) * 0.19823788546;
float w2a = compareColor(col, col2a, x_factor) * 0.11453744493;
float w2b = compareColor(col, col2b, x_factor) * 0.11453744493;

float w2 = 0.37004405286;
float w20a = compareColor(col2, col20a, y_factor) * 0.31718061674;
float w20b = compareColor(col2, col20b, y_factor) * 0.31718061674;
float w21a = compareColor(col2, col21a, y_factor) * 0.19823788546;
float w21b = compareColor(col2, col21b, y_factor) * 0.19823788546;
float w22a = compareColor(col2, col22a, y_factor) * 0.11453744493;
float w22b = compareColor(col2, col22b, y_factor) * 0.11453744493;

vec3 res;
res = w * col.xyz;
res += w0a * col0a.xyz;
res += w0b * col0b.xyz;
res += w1a * col1a.xyz;
res += w1b * col1b.xyz;
res += w2a * col2a.xyz;
res += w2b * col2b.xyz;

res += w2 * col2.xyz;
res += w20a * col20a.xyz;
res += w20b * col20b.xyz;
res += w21a * col21a.xyz;
res += w21b * col21b.xyz;
res += w22a * col22a.xyz;
res += w22b * col22b.xyz;

res /= (w + w0a + w0b + w1a + w1b + w2a + w2b + w2 + w20a + w20b + w21a + w21b + w22a + w22b);
```

```
// 余弦与均匀混合采样
static __forceinline__ __device__ glm::vec3 sampleHemiSphereAlpha(glm::vec3 normal, float alpha, unsigned int seed) {
    float cosTheta = powf(glm::max(0.001f, getRandomFloat(seed)), 1.0f / (1.0f + alpha));
    float sinTheta = sqrtf(1.0f - cosTheta * cosTheta);
    float phi = 2.0f * PI * getRandomFloat(seed);
    glm::vec3 tangentSpaceDir = glm::vec3(cos(phi) * sinTheta, sin(phi) * sinTheta, cosTheta);

    return tangentSpaceDir * getTangentSpace(normal);
}

// GGX重要性采样
static __forceinline__ __device__ glm::vec3 sampleHemiSphereGGX(glm::vec3 normal, float alpha, float alpha2, unsigned int seed) {
    float rx = getRandomFloat(seed);
    float ry = getRandomFloat(seed);
    float cosTheta = sqrtf((1.0f - ry) / (1.0f + ry * (alpha2 - 1.0f)));
    float sinTheta = sqrtf(1.0f - cosTheta * cosTheta);
    float phi = 2.0f * PI * rx;
    glm::vec3 tangentSpaceDir = glm::vec3(cos(phi) * sinTheta, sin(phi) * sinTheta, cosTheta);

    return tangentSpaceDir * getTangentSpace(normal);
}
```

蒙特卡洛采样算法，我们将均匀采样与余弦采样混合，使其更适应某些反射面的性质。

GLSL+OpenGL

在OpenGL端实现的图像双边滤波算法。使用预计算的权重矩阵，对其所覆盖像素进行卷积操作。用户可以自由选择滤波核的大小与滤波程度，以达到较好的滤波效果。

CUDA

```
if (optixLaunchParams.frame.enableProgressiveRefinement) {  
    rgba += float(optixLaunchParams.frame.frameCount) * convert_float4(optixLaunchParams.frame.colorBuffer[fbIndex]);  
    rgba /= (float)optixLaunchParams.frame.frameCount + 1.0f;  
}
```

时间性采样技术，将连续多帧图像的信息按权重累加起来，可以显著降低图像的噪声，同时也提高了渲染器对噪声的容忍度。

C++

pinHole针孔相机模型，对相机视图矩阵的精确计算为用户提供了人性化的、平滑的相机交互模式。

```
if (!ImGui::IsAnyItemActive())  
{  
    // 鼠标中键拖动平移  
    if (ImGui::IsMouseDown(ImGuiMouseButton_Middle))  
    {  
        dx += io.MouseDelta.x * moveSpeed;  
        dy += io.MouseDelta.y * moveSpeed;  
        renderer.launchParams.frame.frameCount = 0;  
        cameraFrame.modified = true;  
    }  
    // 鼠标右键拖动旋转  
    else if (ImGui::IsMouseDown(ImGuiMouseButton_Right))  
    {  
        phi += io.MouseDelta.y * rotationSpeed;  
        theta += io.MouseDelta.x * rotationSpeed;  
        phi = cameraFrame.scalarModAngle(phi);  
        theta = cameraFrame.scalarModAngle(theta);  
        renderer.launchParams.frame.frameCount = 0;  
        cameraFrame.modified = true;  
    }  
    // 鼠标滚轮缩放  
    else if (io.MouseWheel != 0.0f)  
    {  
        dz += scaleSpeed * io.MouseWheel;  
        renderer.launchParams.frame.frameCount = 0;  
        cameraFrame.modified = true;  
    }  
    cameraFrame.camPitch = phi + PI;  
    cameraFrame.camYaw = theta;  
    cameraFrame.moveLeftRight = dx;  
    cameraFrame.moveBackForward = dz;  
    cameraFrame.moveUp = dy;  
    cameraFrame.camRotationMatrix = glm::eulerAngleXYZ(cameraFrame.camPitch, cameraFrame.camYaw, cameraFrame.camRoll);  
    cameraFrame.camTarget = cameraFrame.DefaultForward * cameraFrame.camRotationMatrix;  
    cameraFrame.camTarget = normalise(cameraFrame.camTarget);  
    cameraFrame.camRight = cameraFrame.DefaultRight * cameraFrame.camRotationMatrix;  
    cameraFrame.camForward = cameraFrame.DefaultForward * cameraFrame.camRotationMatrix;  
    glm::vec3 camUp = glm::cross(glm::vec3(cameraFrame.camForward.x, cameraFrame.camForward.y, cameraFrame.camForward.z),  
                                glm::vec3(cameraFrame.camRight.x, cameraFrame.camRight.y, cameraFrame.camRight.z));  
    cameraFrame.camUp = glm::vec4(camUp.x, camUp.y, camUp.z, 0);  
    cameraFrame.camPosition += cameraFrame.moveLeftRight * cameraFrame.camRight;  
    cameraFrame.camPosition += cameraFrame.moveBackForward * cameraFrame.camForward;  
    cameraFrame.camPosition += cameraFrame.moveUp * cameraFrame.camUp;  
    cameraFrame.moveLeftRight = 0.0f;  
    cameraFrame.moveBackForward = 0.0f;  
    cameraFrame.moveUp = 0.0f;  
    cameraFrame.camTarget = cameraFrame.camPosition + cameraFrame.camTarget;  
    cameraFrame.setOrientation(cameraFrame.camPosition, cameraFrame.camTarget, cameraFrame.camUp);  
}
```

```

// SAH 优化构建 BVH
BVHNode* buildBVHwithSAH(std::vector<Triangle>& triangles, int l, int r, int n) {
    if (l > r) return 0;

    BVHNode* node = new BVHNode();
    node->box.min = XMFLOAT3(1145141919, 1145141919, 1145141919);
    node->box.max = XMFLOAT3(-1145141919, -1145141919, -1145141919);

    // 计算 box.min, box.max
    for (int i = l; i <= r; i++) {
        // 最小点 box.min
        float minx = min(triangles[i].p1.x, min(triangles[i].p2.x, triangles[i].p3.x));
        float miny = min(triangles[i].p1.y, min(triangles[i].p2.y, triangles[i].p3.y));
        float minz = min(triangles[i].p1.z, min(triangles[i].p2.z, triangles[i].p3.z));
        node->box.min.x = min(node->box.min.x, minx);
        node->box.min.y = min(node->box.min.y, miny);
        node->box.min.z = min(node->box.min.z, minz);

        // 最大点 box.max
        float maxx = max(triangles[i].p1.x, max(triangles[i].p2.x, triangles[i].p3.x));
        float maxy = max(triangles[i].p1.y, max(triangles[i].p2.y, triangles[i].p3.y));
        float maxz = max(triangles[i].p1.z, max(triangles[i].p2.z, triangles[i].p3.z));
        node->box.max.x = max(node->box.max.x, maxx);
        node->box.max.y = max(node->box.max.y, maxy);
        node->box.max.z = max(node->box.max.z, maxz);
    }

    // 不多于 n 个三角形 返回叶子节点
    if ((r - l + 1) <= n) {
        node->n = r - l + 1;
        node->index = l;
        int maxI = -2147483647, minI = 2147483647;
        if (l != r) {
            for (int i = l; i <= r; i++) {
                if (triangles[i].index < minI)
                    minI = triangles[i].index;
                if (triangles[i].index > maxI)
                    maxI = triangles[i].index;
            }
            node->box.index = XMINT4(minI, maxI, triangles[l].index, triangles[r].index);
        }
        else
            node->box.index = XMINT4(minI, -1, triangles[l].index, triangles[r].index);
        return node;
    }

    // 否则递归建树
    float Cost = 2147483647;
    int Axis = 0;
    int Split = (l + r) / 2;
    for (int axis = 0; axis < 3; axis++) {
        // 分别按 x, y, z 轴排序
        if (axis == 0) std::sort(&triangles[0] + l, &triangles[0] + r + 1, [](const Triangle& t1, const Triangle& t2) {
            return t1.p1.x + t1.p2.x + t1.p3.x < t2.p1.x + t2.p2.x + t2.p3.x;
        });
        if (axis == 1) std::sort(&triangles[0] + l, &triangles[0] + r + 1, [](const Triangle& t1, const Triangle& t2) {
            return t1.p1.y + t1.p2.y + t1.p3.y < t2.p1.y + t2.p2.y + t2.p3.y;
        });
        if (axis == 2) std::sort(&triangles[0] + l, &triangles[0] + r + 1, [](const Triangle& t1, const Triangle& t2) {
            return t1.p1.z + t1.p2.z + t1.p3.z < t2.p1.z + t2.p2.z + t2.p3.z;
        });
    }

```

Cpp

```

// 按最佳轴分割
if (Axis == 0) std::sort(&triangles[0] + l, &triangles[0] + r + 1, [](const Triangle& t1, const Triangle& t2) {
    return t1.p1.x + t1.p2.x + t1.p3.x < t2.p1.x + t2.p2.x + t2.p3.x;
});
if (Axis == 1) std::sort(&triangles[0] + l, &triangles[0] + r + 1, [](const Triangle& t1, const Triangle& t2) {
    return t1.p1.y + t1.p2.y + t1.p3.y < t2.p1.y + t2.p2.y + t2.p3.y;
});
if (Axis == 2) std::sort(&triangles[0] + l, &triangles[0] + r + 1, [](const Triangle& t1, const Triangle& t2) {
    return t1.p1.z + t1.p2.z + t1.p3.z < t2.p1.z + t2.p2.z + t2.p3.z;
});

// 递归
node->left = buildBVHwithSAH(triangles, l, Split, n);
node->right = buildBVHwithSAH(triangles, Split + 1, r, n);

return node;

```

我们利用树形结构，采用递归的方式建立BVH加速结构体，并使用SAH启发式算法提升BVH树的精度。

对路径追踪算法的创新

经典路径追踪算法是对光线相交点所在的半球面进行蒙特卡洛积分，然后将采样得到的结果按渲染方程逐项累加。这种方式不仅耗时，大量采样的结果可能因为光线没有找到光源而被抛弃。

我们采用蒙特卡洛多重重要性采样的方式，将余弦采样和GGX重要性采样结合起来，加速相机与光源间的光路传输过程，从而提升算法的收敛速度。

另外，受路径追踪算法使用俄罗斯轮盘赌的方式终止光线弹射的启发，我们用这种方式来随机地选择物体表面的物理特性。这样每次弹射的计算量便大大减少。

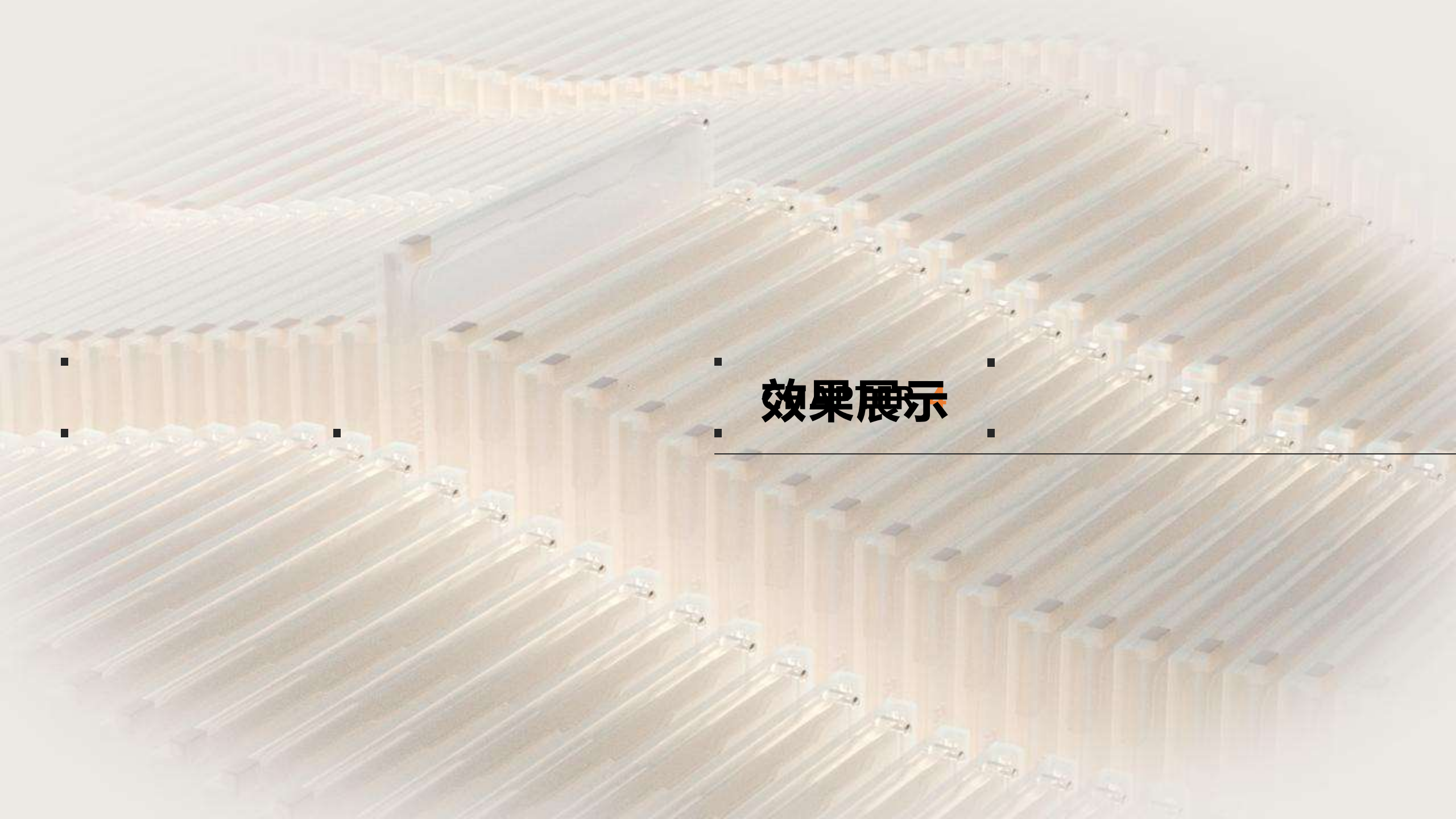
对于单根光线来说，这种方式可能违背了能量守恒，但可以证明，对于整体图像而言，它生成的图像相对于真实图片依然是无偏的。

```
prd.origin = newOri + 1e-3f * N;
prd.lastEnergy = prd.energy;
// using Russian Roulette to determine which ray type the generated ray is.
float roulette = getRandomFloat(prd.seed); // just a random float number
float diffChance = glm::saturate(kd.x + kd.y + kd.z) / 3.0f + 0.0000001f;
float coatChance = glm::saturate(optixLaunchParams.frame.clearcoat * optixLaunchParams.frame.clearcoat * 0.25f);
float specChance = glm::saturate((1.0f - diffChance - coatChance) + 0.0000001f);
if (roulette < specChance) { // shading model is specular
    glm::vec3 F = calcF(F0, VdotH);

    float G = calcG(NdotL, NdotV, alpha2);

    prd.direction = sdir;
    prd.energy *= sResult((1.0f / specChance) * Fs * Gs / NdotL / NdotV * LdotV) * PI;
}
else if (roulette < specChance + diffChance) // shading model is diffuse
{
    glm::vec3 ddir = sampleHemiSphere(N, prd.seed);
    prd.direction = ddir;

    float roulette2 = getRandomFloat(prd.seed) * 0.99998f + 0.00001f;
    if (roulette2 < optixLaunchParams.frame.subSurface)
        prd.energy *= sResult((1.0f / diffChance / roulette2) * Cdlin * sss / kd.x + Fsheen * PI);
    else
        prd.energy *= sResult((1.0f / diffChance / (1.0f - roulette2)) * Cdlin * Pd / kd.x + Fsheen * PI);
}
else if (roulette < specChance + diffChance + coatChance) // shading model is clearcoat
{
    prd.direction = sdir;
    prd.energy *= sResult((1.0f / coatChance) * clearcoat / Ds / NdotL / NdotV * LdotV) * PI;
}
else // dropout
{
    prd.energy = glm::vec3(0);
}
prd.radiance = emission;
```

效果展示

精确性



Helmet在不同环境下的渲染效果





使用Phong光照模型

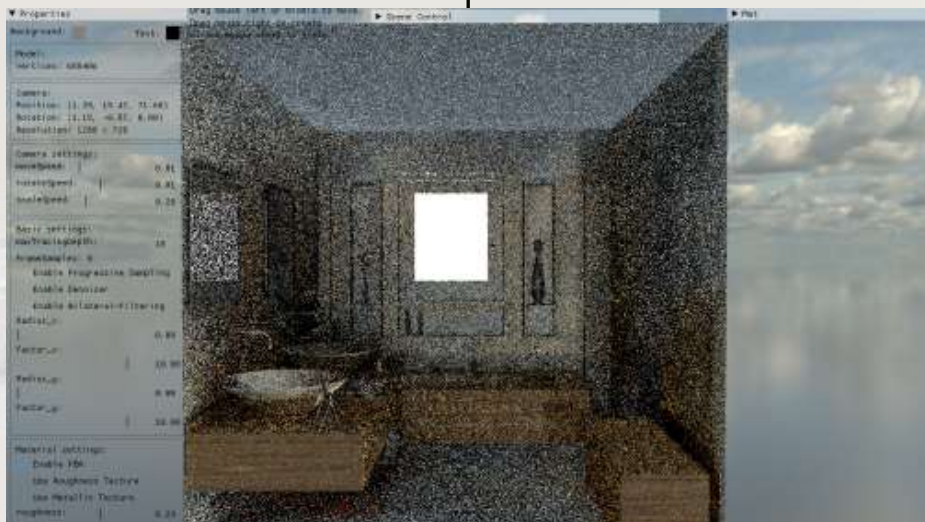


使用BRDF光照模型

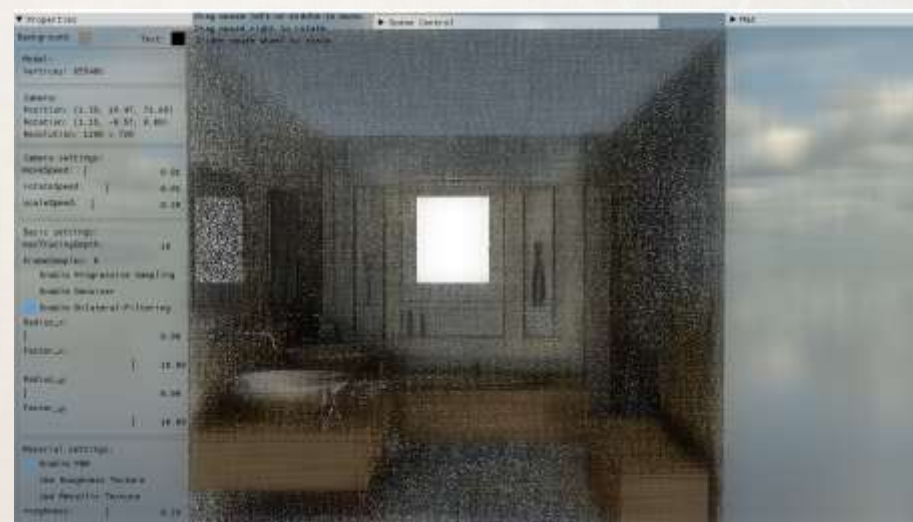


使用Disney BSSRDF光照模型





不使用时间性采样的
原始图像



不使用时间性采样、使用
双边滤波的图像



TAA叠加20帧的图像



TAA叠加20帧的双边滤波图像



TAA叠加80帧的图像



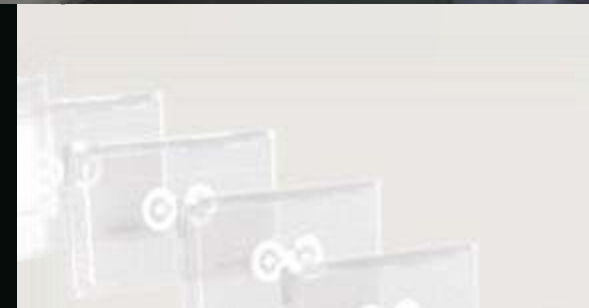
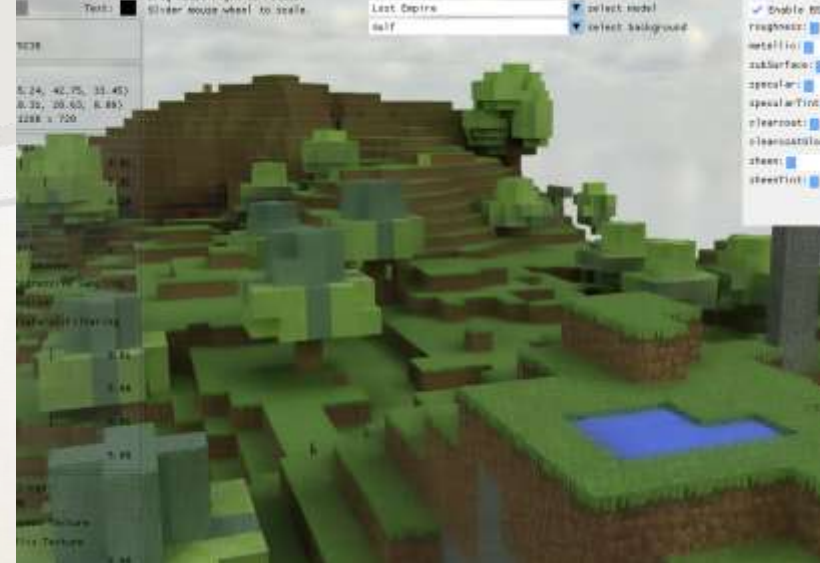
TAA叠加80帧的双边滤波图像



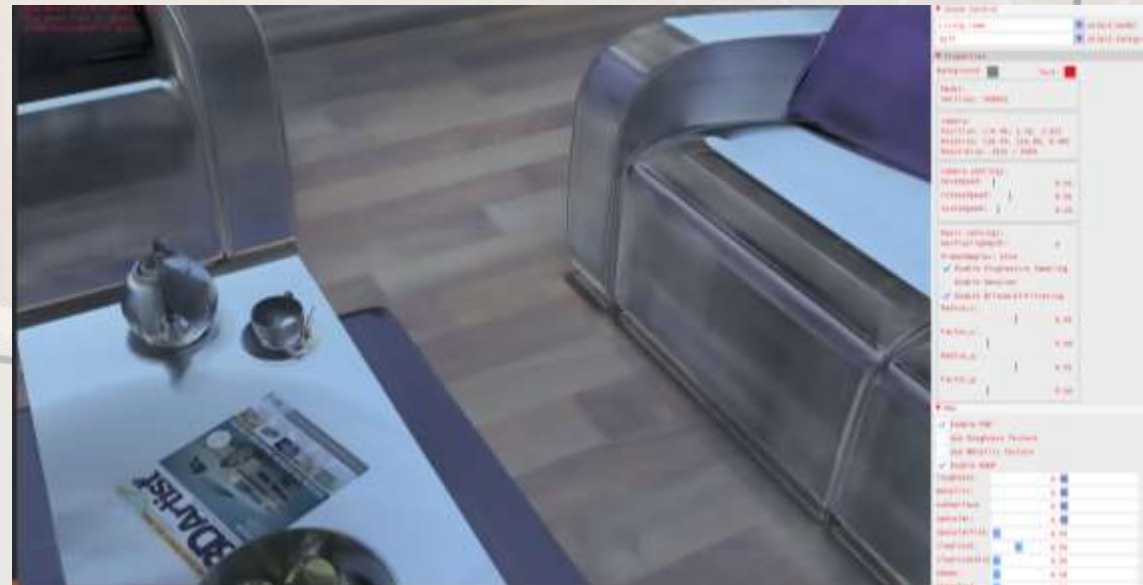
第0帧原始图像



使用Optix AI降噪后的原始图像

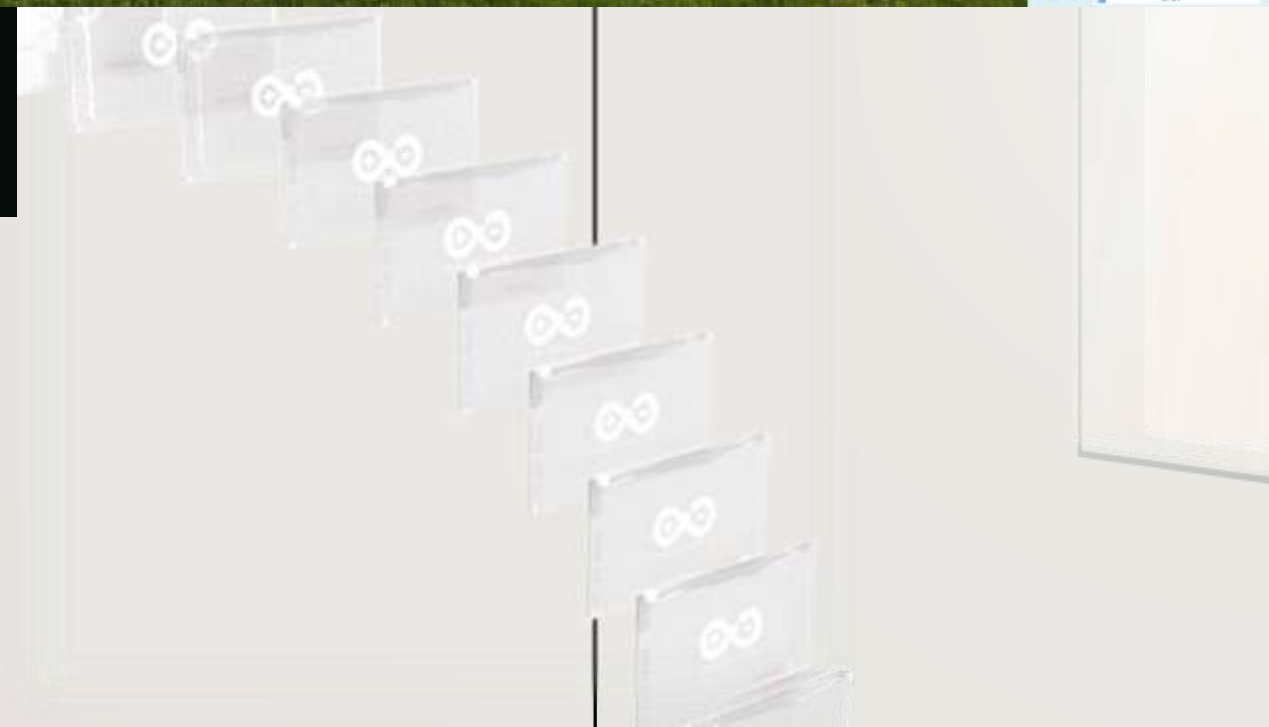


不同视角下的渲染效果





自定义材质系统



实时性

注：所有图片均采集自RTX3060Laptop显卡

SELECTING FILE



实时性

注：所有图片均采集
自RTX3060Laptop显卡

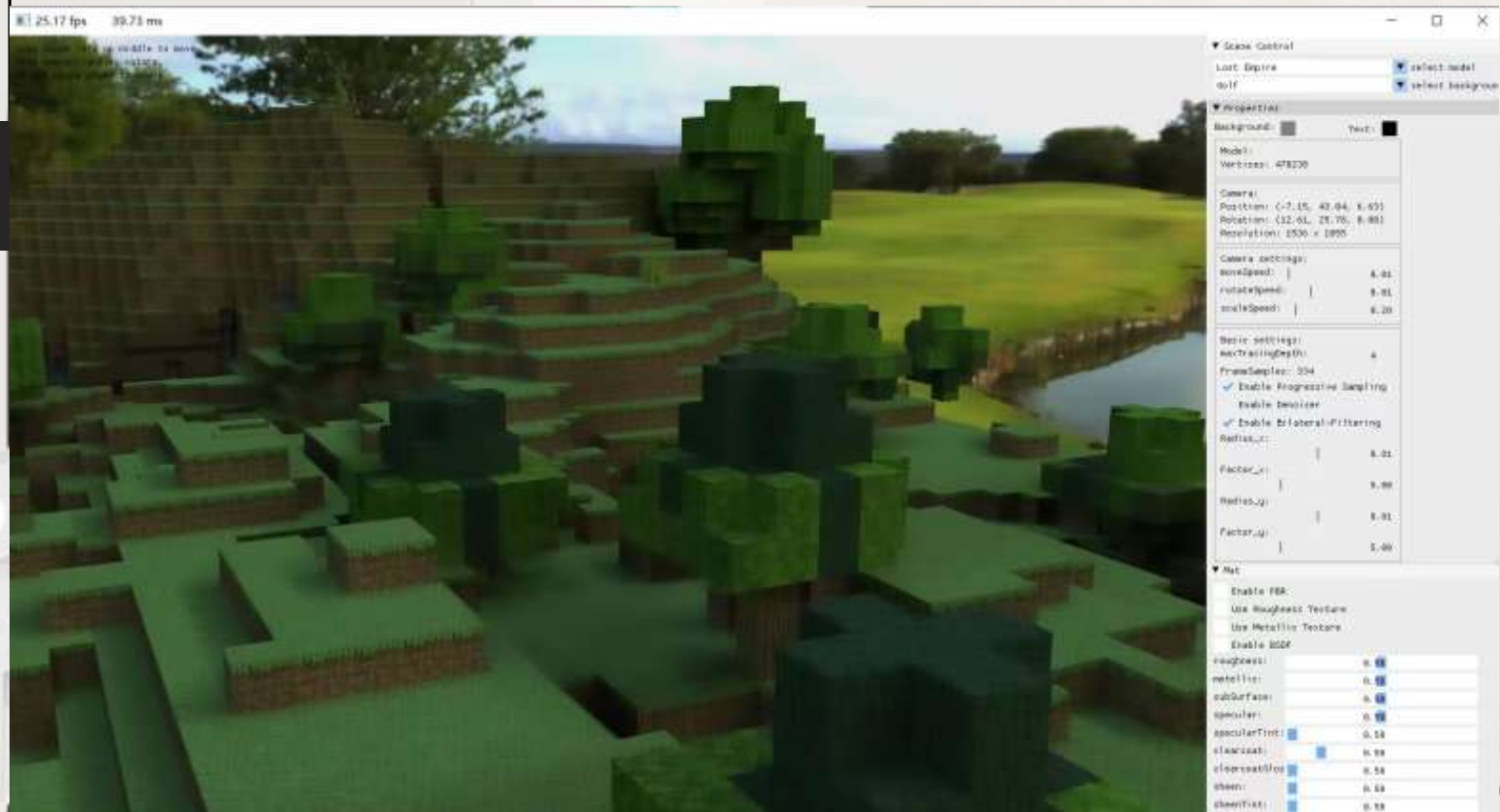


SELECTING FILE

实时性

注：所有图片均采集
自RTX3060Laptop显卡

SELECTING FILE



稳定性

san-miguel.obj X

F: > San_Miguel > san-miguel.obj

```
24815715 f 5933216/2221667/6721515 5933228/2221668/6721527 5933215/2221669/6721514
24815716 f 5933216/2221667/6721515 5933227/2221668/6721526 5933228/2221669/6721527
24815717 f 5933215/2221667/6721514 5933226/2221668/6721525 5933214/2221669/6721513
24815718 f 5933215/2221667/6721514 5933228/2221668/6721527 5933226/2221669/6721525
24815719 f 5933221/2221667/6721520 5933232/2221668/6721531 5933220/2221669/6721519
24815720 f 5933221/2221667/6721520 5933233/2221668/6721532 5933232/2221669/6721531
24815721 f 5933220/2221667/6721519 5933227/2221668/6721526 5933216/2221669/6721515
24815722 f 5933220/2221667/6721519 5933232/2221668/6721531 5933227/2221669/6721526
24815723 f 5933218/2221667/6721517 5933230/2221668/6721529 5933231/2221669/6721530
24815724 f 5933218/2221667/6721517 5933231/2221668/6721530 5933219/2221669/6721518
24815725 f 5933217/2221667/6721516 5933230/2221668/6721529 5933218/2221669/6721517
24815726 f 5933219/2221667/6721518 5933231/2221668/6721530 5933233/2221669/6721532
24815727 f 5933219/2221667/6721518 5933233/2221668/6721532 5933221/2221669/6721520
24815728
```

选择 D:\Graphic\Ray_Tracing\OptiX\pathtracer\build\PathTracer.exe

```
From ../model/San_Miguel/ loading models...
Finished to load OBJ files.
Done loading obj file - found 2203 shapes with 287 materials.
```

Properties

Background:  Text: 

Model:
Vertices: 9821669

Camera:
Position: (8.05, 11.64, 5.66)

SELECTING FILE

21.29 fps 46.97 ms

稳定性



Background:

Text:

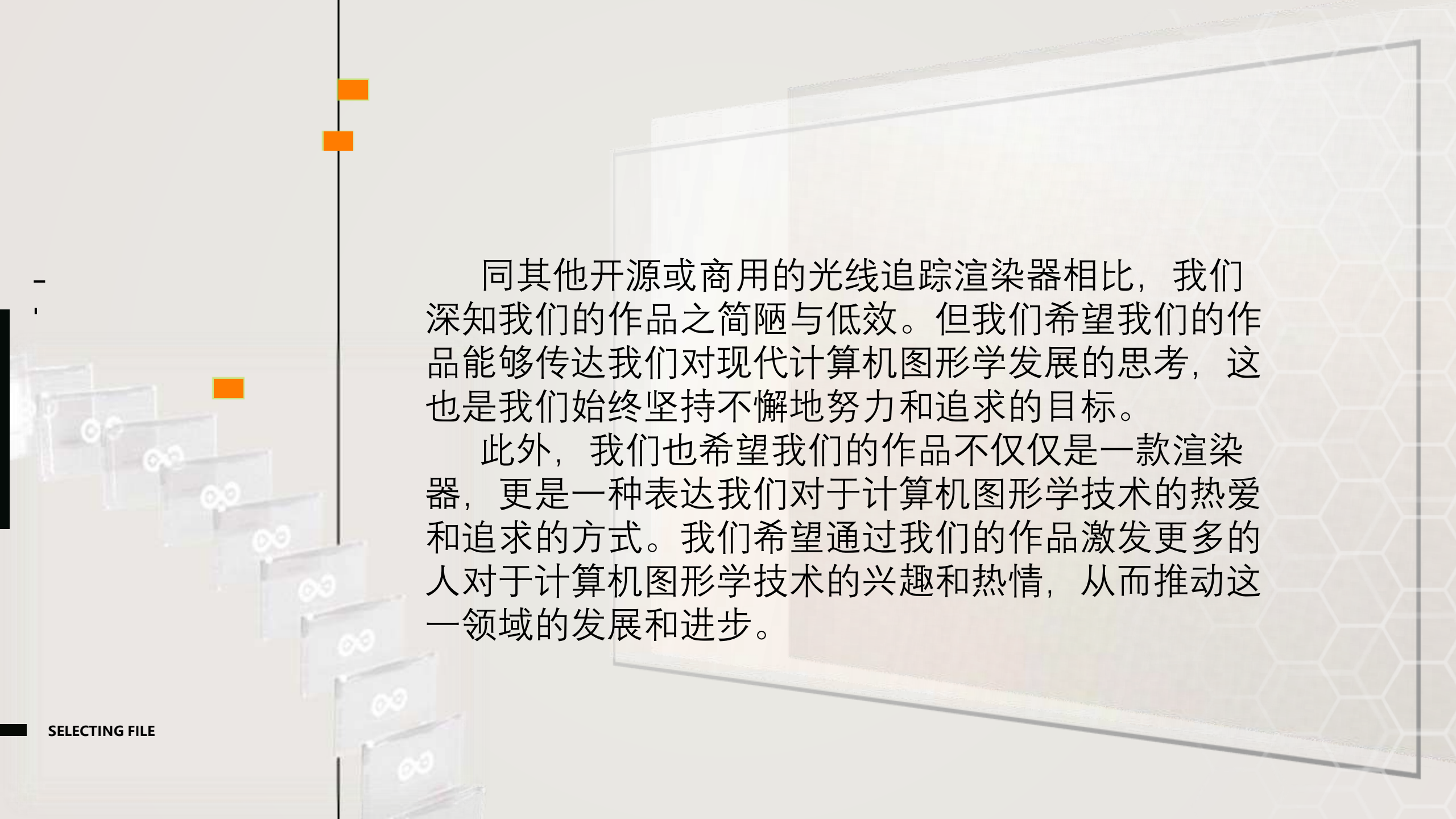
Model:

Vertices: 9021669

Camera:



总结TER 5



同其他开源或商用的光线追踪渲染器相比，我们深知我们的作品之简陋与低效。但我们希望我们的作品能够传达我们对现代计算机图形学发展的思考，这也是我们始终坚持不懈地努力和追求的目标。

此外，我们也希望我们的作品不仅仅是一款渲染器，更是一种表达我们对于计算机图形学技术的热爱和追求的方式。我们希望通过我们的作品激发更多的人对于计算机图形学技术的兴趣和热情，从而推动这一领域的发展和进步。