不同粒度的光线追踪

小组成员：叶增渝, 李超

摘 要:

实时光线追踪技术的问世使得渲染结果更加真实立体，然而相比于传统的光栅化渲染方法，光线追踪技术的耗时更长，因此，在光线追踪渲染质量和渲染耗时之间寻找一个平衡便成了一项重要的课题。本小组的思路是稀疏化光线样本的粒度，即在不同粒度下实现光线追踪功能，通过对渲染质量和渲染性能的比较来找到适合应用的粒度。本小组首先实现了曲面细分阶段，以能够灵活地改变粒度大小，然后在曲面细分后逐顶点、逐像素、逐顶点等三种粒度下实现了光线追踪，最后比较了不同粒度下光线追踪的质量和性能并给出分析。本小组的优势和创新点第一在于利用DirectX12内联光线追踪的新特性来实现不同粒度下的光线追踪，第二在于对如何在光线追踪渲染质量和渲染耗时之间寻求平衡这一工程问题给出了建议和方案。

**Ray tracing with different granularity**

Zengyu Ye,Chao Li

**Abstract**:

The advent of real-time raytracing technology makes rendering results more realistic and three-dimensional, but raytracing technology takes longer time than traditional raster rendering methods, so finding a balance between raytracing rendering quality and rendering time is an important issue.The idea of our team is to sparse the granularity of ray samples, that is, to realize ray tracing function at different granularity, and to find suitable granularity for application by comparing rendering quality and rendering performance.The team first realized the surface subdivision stage to flexibly change the particle size, and then realized ray tracing in three particle sizes of surface subdivision: vertex by vertex, pixel by pixel, vertex by vertex. Finally, the quality and performance of ray tracing under different particle sizes were compared and analyzed.The advantages and innovations of our team are as follows: first, we use DirectX12's new feature of inline ray tracing to achieve ray tracing at different granularity; second, we give suggestions and solutions on how to strike a balance between raytracing rendering quality and rendering time.

**Key word：Inline、光线追踪、粒度、DirectX 12**

# **简介与意义/Introduction**

## 项目意义和依据/Significance

2018年,实时光线追踪技术元年。在这一年，微软宣布了DirectX Ray Tracing (DXR) 的问世，Nvidia宣布了支持实时光线追踪技术的RTX系列显卡，同时诸如UE、Unity等引擎相继宣布兼容实时光线技术。此后，相关实时光线追踪技术得到了长足研究与发展。实现高效高质量的实时全局光线追踪能够使最终得到的渲染结果更加真实、立体，一直是科学界、工业界大力追求的一个研究方向。而由于光线追踪技术的耗时较长，为保持实时性，实现渲染耗时与渲染质量之间的平衡成为了一项重要的课题。而这项课题需要制定合理的实时渲染预算与方案、善用光照传输缓存技术，甚至探索全新的混合渲染管线等等，具有巨大的研究意义。

DX9时代低频球谐光照就可以逐顶点着色并保持一定质量，目前依旧应用于Unity等引擎的默认光照模式。1spp路径追踪配合降噪可以近似实现高效高质量实时全局光照，但其耗时依旧为传统屏幕空间反射、阴影的两三倍。在本课程的学习中本小组了解到，从到Gauroud Shading对每一个面片进行相同的颜色插值，到对一个面片进行线性插值赋予颜色，再到Phong Shading的像素层级进行的颜色赋值。我们不难知道，一个粗粒度的光线追踪所需要计算量相对较少，那么耗时相对更加细粒度的光线追踪更少，而从质量方面看，假设面片的数量较少，我们会比较容易看到一个个色块，获得的结果也会质量较低，但如果面片足够小，那么即使是粗粒度的光线追踪也能达到良好的效果，但具体如何寻找两者的平衡点，还是需要进行实际评估。因此，值得尝试更进一步稀疏化光线样本的粒度，减少光线追踪数量，在性能与质量之间找到一个更加良好的平衡点，由于实时光线追踪技术在图形学中尚处于初步探索发展阶段，这样的探索也并非易事，具有很强的探索意义，能为未来的游戏等渲染场景提供能力更加平衡的实时光线追踪方案。

因此，本小组尝试探索一种稀疏采样方式，提高有效采样率尽可能保证质量并带来因光线追踪数减小的性能提升。首先本小组通过前期调研了解并决定使用的图形管线与光线追踪形式，并研读了微软官方文档、微软DirectX12光线追踪和《DIRECTX 12 3D游戏开发实战》[6]（以下简称为“DircetX 12龙书”）参考书中含有曲面细分阶段的渲染流水线的代码实例，了解了对应管线从初始化到销毁的全部流程与hlsl文件的编写规范，最终确定需要修改的shader stage，在不同要求下在不同位置加入光线追踪程序，最终实现不同粒度的光线追踪。

## 本方法/系统框架/Article Structure

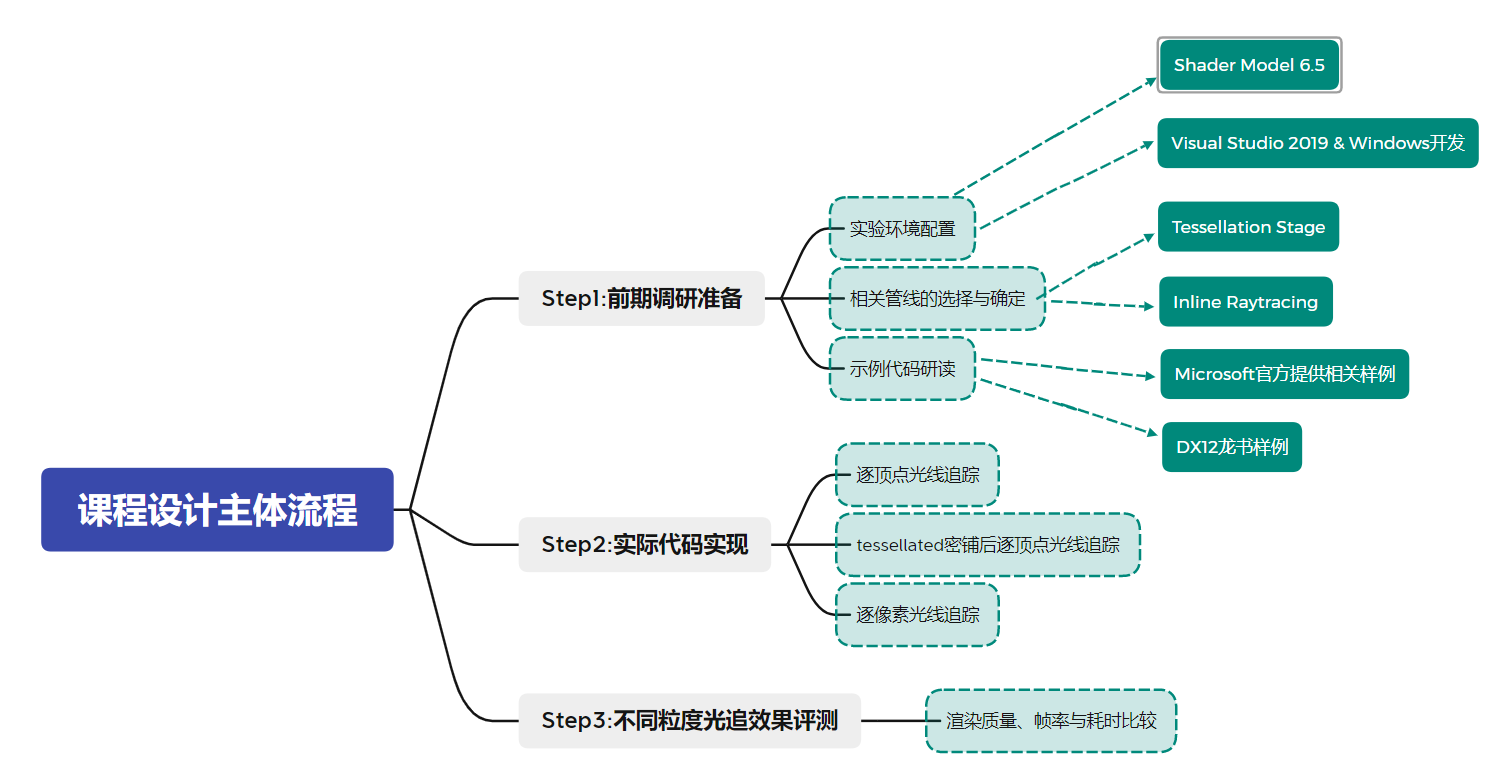


图1 课程设计主体流程图

# **相关工作/Related Works**

1.现代常用引擎默认使用的球谐光照的原理了解

2.1spp路径配合降噪实现的高效高质量全局实时光照实现分析

3.业界成熟的实时光线追踪方案了解,诸如经典的KDTre等[1][2][3][4][5]。

# **研究内容与方法(或算法)/Contents and Methods(or Algorithm)**

Tips:本项目实现的所有项目工程文件均以龙书中的结构文件为项目基础减少Initialization复杂性，以Microsoft官方样例的一些帮助文件简化流程进行构建。

工作环境：

（1）DirectX 12库(使用dxdiag诊断，一般Windows电脑已含有其SDK，无需下载)

（2）GPU： NVIDIA GeForce RTX 2060

（3）代码环境：Viusal Studio 2019（通用Windows平台开发）

（4）Shader Model版本：Shader Model 6.5

前期调研:

传统地想要实现光线追踪的功能，往往需要新建光线追踪加速结构并构建一个相应的光追管线，这对于本小组改变光线追踪的粒度进行测试带来了一定的程序迁移困难，幸运的是实时光线追踪技术提出了一种不需要额外独立管线的光线追踪实现形式—inline raytracing，通过简单地定义加速结构与RayQuery，开发者可以在任意shader stage通过添加inline raytracing相关

函数实现不同阶段、粒度的光线追踪，大大减少了重复代码量。

根据本课程了解到的图形学课程知识可知粗细粒度的划分起源于shading时采用的方式与选择的对象（如Phong Shading与Gauroud Shading之间的差别，图2所示），那么我们可以根据shader stage的不同将本项目中的粒度粗略划分为逐顶点、 tessellated密铺后逐顶点、逐像素三种光线追踪粒度。

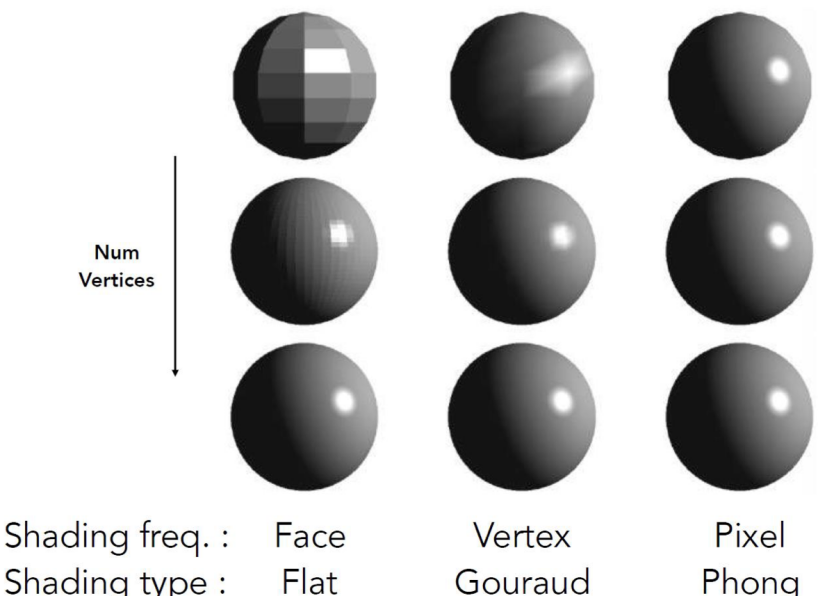


图2 不同shading结果示例

本小组选择采用传统流水线以及inline raytracing进行工程实现上述三种不同粒度的光线追踪并进行比较。

前期知识准备：

主要包含以下四个方面。

（1）DirectX 12入门：

参考DircetX 12龙书及Microsoft公布的DirectX相关示例源码中的HelloWorld与RayTracing相关示例代码进行学习，对内部采用的各类函数通过Microsoft相关开发文档进行了解学习，掌握一个程序的基本组件及其依赖关系。

（2）inline raytracing入门:

参考DXR 1.1的相关文献与档案进行了解学习。

（3）tessellation入门：

参考DirectX12龙书的tessellation示例代码进行学习，掌握实现tessellation所需的各种组件。

解决方案：

本小组在前期知识准备阶段学习了DirectX的各类API和渲染管线的初始化、建立过程。DirectX12使用了COM接口技术取代了C++中原本的指针类，使其独立于编程语言，向后兼容。Direct12X的一个simple app的建立包含了4个主要阶段，分别为Init、Update、Render、Destroy四个主要阶段，其中Init阶段为最主要的阶段，负责创建各类结构、资源。High Level Shader Language (hlsl) 高级着色器语言文件在DirectX12主要负责对各个着色器的描述实现。DX12中渲染的模型实体由vertex buffer与index buffer两个数据结构存储，它们除了自己顶点的位置信息，还可以定义、存储额外的信息。同时为了让DirectX12了解如何处理这些数据，我们需要定义输入布局信息 (Input Layout descriptor)。除了DirectX12中的通用API以外，描述符堆 (Descriptor Heap) 和根签名 (Root Signature) 是应当关注的两个结构。其中描述符堆是我们存放所有同种类描述符的一个堆结构。描述符类型有RTV (render target view) 用以存放渲染的对象，DSV (depth stencil view) 用以存放深度/模板缓冲，UAV (unordered access view)，SRV (shader resource view) 用以创建shader资源和CBV (const buffer view) 用以存放常量缓冲区的资源。根签名中包含根参数 (Root Parameter) ，描述其内部每一个输入槽对应的ID3D12DescriptorHeap类型并进行相应初始化，每一个输入槽都对应着本项目hlsl文件中的register声明，并且需要在draw的时候根据其DescriptorHeap类型设置不同的视图 (View)。本小组在项目中根据需求来定义一些描述符堆和根签名。这些知识是实现本项目的基础。

通过研读龙书中tessellation stage的示例代码，发现在Initialize()函数所调用的BuildShadersAndPSOs()中，读入了四个着色器文件，分别对应Vertex Shader，Hull Shader，Domain Shader和Pixel Shader，然后根据自身的model与四个文件的内容进行渲染，我们想要在其中实现inline raytracing，那么就应当在四个hlsl文件中对shader进行修改，也就是在代表shader的hlsl文件中找到对应部分的位置，添加inline raytracing内容并判断光线可能遇到的各种情况，实现光线追踪，而不同粒度的实现应当是在不同shader的合适阶段添加raytracing从而产生不同粒度的效果。

本小组首先在传统管线上实现了tessellation stage。Tessellation技术是将几何体细分为更小的三角形，并以某种方式把这些新生成的顶点偏移到合适的位置，从而以增加三角形数量的方式丰富网格的细节的一种技术。在渲染流水线中tessellation stage是一个可选阶段，实现tessellation stage的理由通常是基于GPU实现动态细节级别（根据网格与摄像机的距离或者依据其他因素），降低物理模拟与动画特效的计算量以及节约内存。在本项目中，通过改变曲面细分因子的大小，可以改变顶点的数量，从而实现不同粒度的光线追踪，所以tessellation在本项目中是一个非常重要的手段。

本小组实现的tessellation stage在原有流水线代码的基础上实现了外壳着色器阶段、镶嵌器阶段和域着色器阶段等三个阶段，如图3所示。

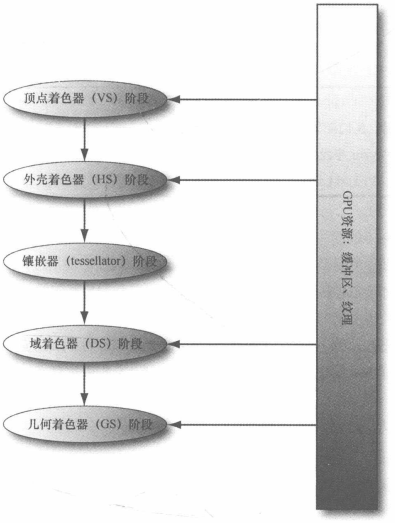


图3 包含tessellation stage的渲染流水线

实现tessellation stage的第一步是向IA提交具有若干控制点的面片 (patch)，而不是向IA提交三角形。即需要将PrimitiveType从triangle改为patch。在本项目中，本小组选择的是具有三个控制点的面片，因此可以把三角形的三个顶点视为其三个控制点。

接下来实现三个着色器。外壳着色器(hull shader)由常量外壳着色器 (Constant HS) 和控制点外壳着色器 (control point HS) 组成。首先实现常量外壳着色器。每处理一次面片，常量外壳着色器就会被调用一次。它以面片的所有控制点和面片的标识ID为输入，输出为面片的曲面细分因子 (tessellation factor)。曲面细分因子说明了tessellation stage处理后该面片被分成的份数。如图4中左图所示，EdgeTess=6时，三角形的边被分为了6份。再实现控制点外壳着色器。每输出一个控制点，控制点外壳着色器都会被调用一次。控制点外壳着色器可以改变曲面的表示方式，比如把一个三角形转换为3次贝塞尔三角形面片，并输出转换后的控制点。在本项目中，无须利用控制点外壳着色器对曲面进行修改，因此本小组仅仅将控制点外壳着色器作为一个传递着色器使用，不对控制点产生任何的修改。

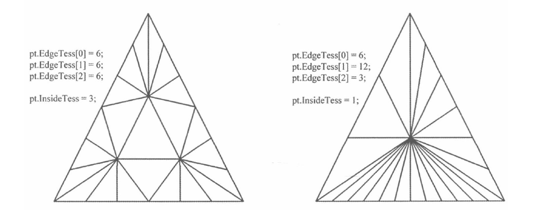


图4 基于不同曲面细分因子对三角形细分的示意图

镶嵌器全权交由硬件接管和处理，无法对其进行编程。该阶段以常量外壳着色器输出的曲面细分因子为输入，对面片进行曲面细分操作处理。若曲面细分因子均为0，则在后续处理中被抛弃，相当于没有做镶嵌操作。处理后面片会分成许多三角形，具有更多的顶点。镶嵌器的输出是这些新建的三角形和顶点。

域着色器以常量外壳着色器输出的曲面细分因子、控制点外壳着色器输出的面片的所有控制点以及镶嵌器输出的顶点位置参数坐标作为输入，根据参数坐标来计算顶点的世界坐标和其他附加信息，输出为顶点。

最后，在BuildShadersAndPSOs()函数中，即在初始化PSO时读入以上创建好的shader，就可以实现tessellation。Tessellation stage中数据的流向如图5所示。

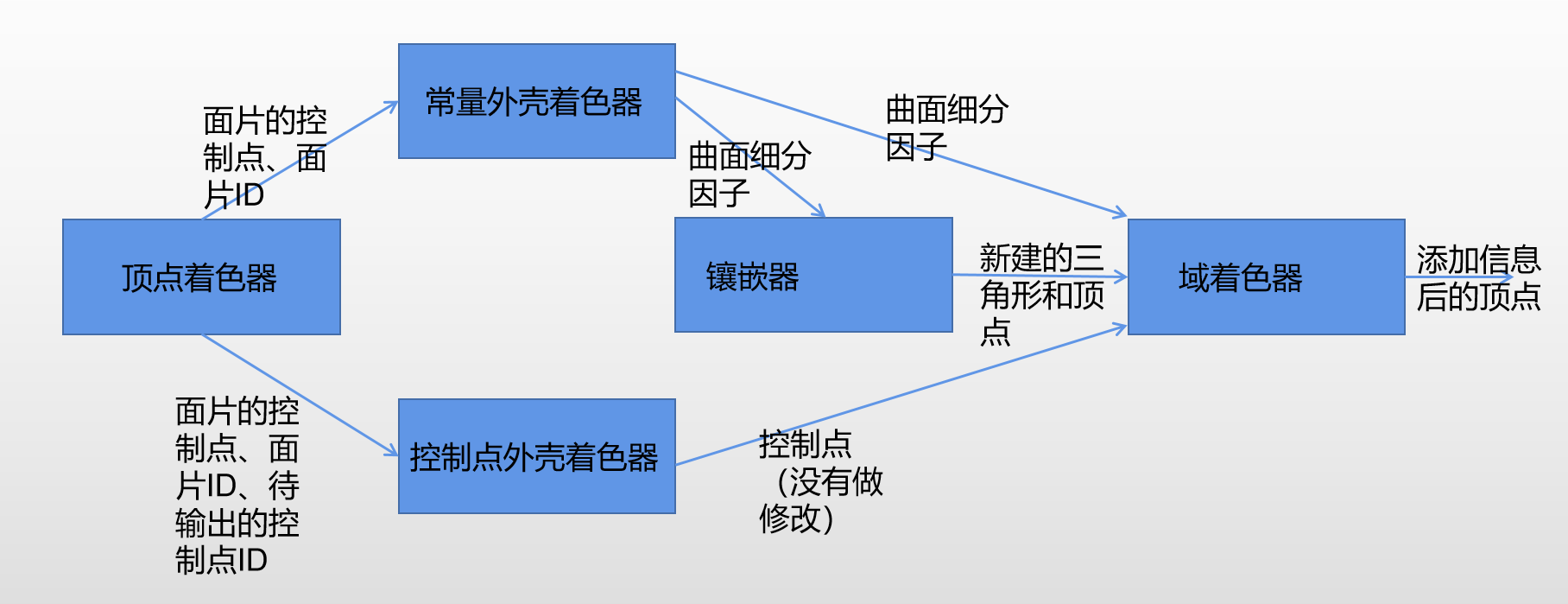


图5 tessellation stage中数据的流向

实现tessellation stage的目的是只需要改变曲面细分因子即可灵活地在任意粒度上实现光线追踪。因此，本小组的下一项工作是利用DirectX12实现光线追踪 (DXR)。运行DXR的GPU配置要求最低是RTX20系显卡，Shader Model6.5版本，在代码中枚举出系统中的显卡后，使用CheckFeatureSupport方法可以检测显卡是否支持DXR。如果不支持，可以继续调用D3D12EnableExperimentalFeatures方法来检测显卡是否支持fallback layer raytracing。Fallback layer raytracing是微软提供的一个在系统硬件不支持DXR的情况下用软件方法来模拟DXR的库，如果显卡支持fallback layer raytracing，那么可以调用该库中的API来实现光线追踪，但是其性能要比DXR差，渲染速度更慢，且更加繁琐，对于带镶嵌器的光线追踪需要对模型进行中途保存，导致难度增加。使用上述方法检测显卡支持程度后，本小组选择直接使用DXR来实现光线追踪。

传统光线追踪需要像光栅化流水线中的各个shader一样，设计出一个单独的raytracing shader，并且设计出raytracing流水线。然而，DXR中引入了光线追踪的另一种组织形式——inline raytracing（内联光线追踪）。Inline raytracing不要求使用者另外创建一个raytracing shader和raytracing流水线，使用者只需要创建一个加速结构，并且可以在任何的shader stage里直接写入光线追踪代码，实现光线追踪的功能。对于小规模的光线追踪，inline raytracing代码简单

和速度快的特性大大方便了使用者按照需求来灵活地实现光线追踪。本小组通过在不同的shader stage中写入光线追踪代码，实现了不同粒度的光线追踪（tessellation后逐顶点、逐像素、逐顶点）。

传统的raytracing shader中需要有加速结构、光线发射函数 (raygeneration)、最近碰撞函数 (closesthit) 和未碰撞函数 (miss) 等重要部件，在inline raytracing中同样需要定义这些部件。

首先是加速结构[3][4]。在光线追踪中，如果要计算光线和每一个三角形是否有交点，那么复杂度将会非常大。因此，可以为物体创建边与坐标轴平行的AABBs包围盒，先计算光线与物体的AABBs是否有交点，再计算光线具体与该AABBs中的哪些三角形有交点。但是我们在本例渲染中没有涉及非常复杂的多物体场景，所以我们使用最基本的Triangle Mesh。用于上述加速过程的结构体就是加速结构。本小组在Initialize()中添加了BuildAccelerationStructure()函数，其中建立了两个加速结构——底层加速结构和顶层加速结构（这里主要是由于我们只对但物体进行渲染，所以仅需要一个底层结构）。底层加速结构中存储的是一个局部区域的物体vertex和index信息。顶层加速结构中拥有一系列instance对象，每个instance对象都指向了一个底层加速结构以及存储了一些额外信息（例如变换矩阵，以对物体及其AABBs进行坐标转换等）。利用加速结构计算是否有交点的过程由硬件执行。在shader文件中，只需要声明一个加速结构的全局变量并与寄存器绑定就可以使用该加速结构。

然后本小组在shader文件中实现了inline raytracing的代码。首先要定义一个RayQuery对象，之后的所有操作都需要用到这个对象。然后初始化一条从观察点出发到达目标像素的光线[5]，这部分代码相当于传统raytracing shader中的光线发射函数。接下来调用RayQuery::TraceRayInline方法来建立一个追踪过程，注意，此时并未开始光线追踪，紧接着调用RayQuery::Proceed()方法后才开始光线追踪。然后对追踪结果分情况处理。如果没有与任何三角形发生碰撞，那么将目标像素的颜色设置为仅含漫反射光照形成的光照，这一部分代码相当于传统raytracing shader中的未碰撞函数。如果与三角形发生了碰撞，则计算散射光照和直接光照，用二者的加和作为目标像素的颜色，同时获取漫反射材料的α值作为目标像素的α值。这部分代码相当于传统raytracing shader中的最近碰撞函数。

至此，本小组完全实现了inline raytracing。接下来要做的是在不同粒度下实现上述的光线追踪算法。要实现tessellated密铺后逐顶点光线追踪，根据tessellation的原理，域着色器的输出是tessellation stage最终产生的顶点，所以需要在域着色器中写入上述inline raytracing的代码，并且这样的光线追踪粒度会随着曲面细分因子的系数改变而发生改变。要实现逐像素光线追踪，那么需要在piexel shader中写入上述inline raytracing的代码。要实现逐顶点光线追踪，那么需要在vertex shader中写入上述inline raytracing的代码。

# **实验结果与分析/Experiment Results and Analysis**

下面给出本项目的效果图。

首先展示tessellation效果图（图6）。从左上至右下依次为tess=1,tess=3,tess=5和根据远近获取tess（tess基值为10）。

本小组在展示tessellation效果时，将光栅的fiillmode改为了wireframe（即在一个三角形中只有顶点及顶点之间的连线上色，其他区域为背景色），以使得镶嵌器输出结果更为突出。可以看到随着曲面细分因子的不断增大，三角形面片被分为越来越小的三角形。随着三角形越来越密集，图9中右下角的图片直观上看来与普通模式下甚至没有分别，人眼看不出任何背景色。可以确认镶嵌器起到了强大的曲面细分作用。

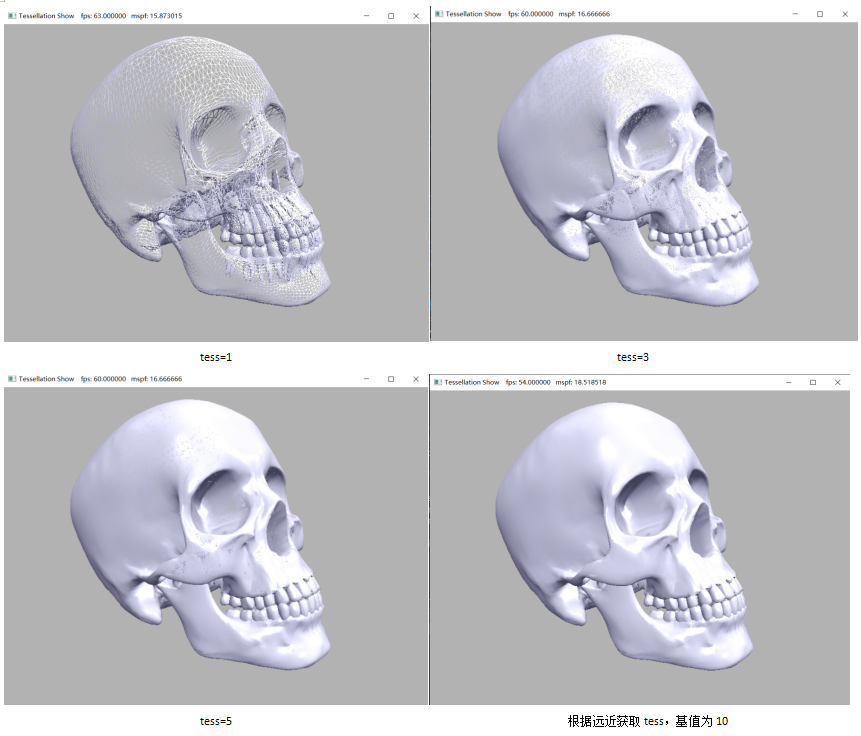


图6 tessellation效果图

然后展示不同粒度下光线追踪的效果图（图7和图8）。图7中从左上至右下依次为逐像素、逐顶点、tessellated密铺后逐顶点（根据远近获取tess，基值为1）和tessellated密铺后逐顶点（根据远近获取tess，基值为5）。图8为tessellated密铺后逐顶点（根据远近获取tess，基值为10）。可以看到当我们在per vertex粒度上进行光追时展现的真实性不如per pixel粒度， per vertex在很多亮斑部分呈现较为明显的线性插值效果，光斑形状十分规整，没有体现凹凸感，而per pixel则更好地体现了skull模型的凹凸形态，光斑更加地不规则。在tessellation

stage进行光线追踪时产生的效果与曲面细分因子的大小有很大关系，越小越接近per vertex粒度，越大越接近per pixel粒度。

在渲染质量上，结果非常符合本小组的预期，即对越小的单位进行光线追踪，得到的渲染质量也就越好。

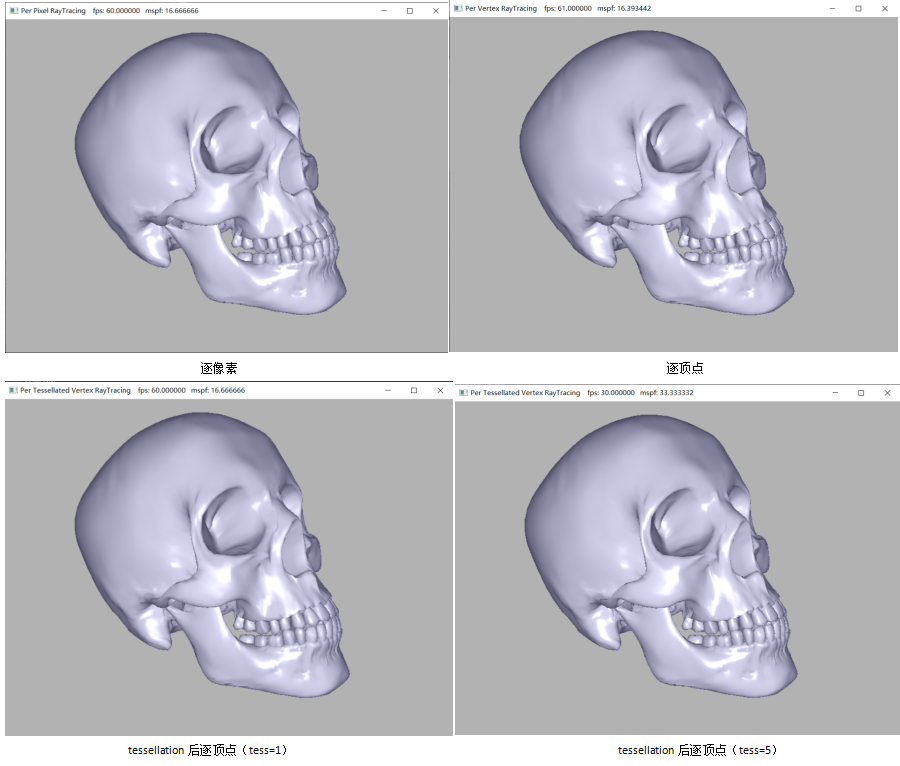
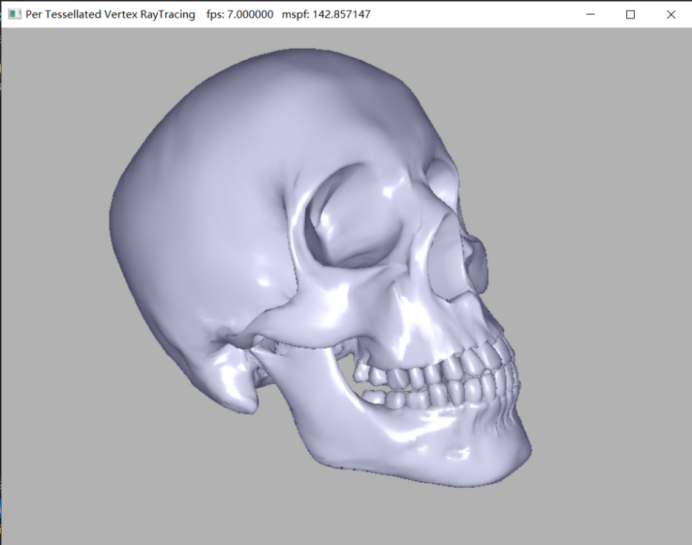


图7 不同粒度下光线追踪的效果图（一）



tessellation后逐顶点（曲面细分因子=10）

图8 不同粒度下光线追踪的效果图（二）

接下来给出不同粒度下的光线追踪性能的分析。本小组用FPS的值来衡量性能。本小组分别在不同粒度下取多次FPS值并求平均值，最后将不同粒度和对应的FPS制成表1。首先比较逐像素和逐顶点光线追踪的性能，逐像素光线追踪的FPS平均值为58，其上限不超过60，而逐顶点光线追踪的平均值为63，最低值为60，其原因非常显然，因为逐像素比逐顶点需要计算的光线更多，产生的开销更大。在这里，因为本小组渲染的场景比较简单，所以差别不是特别显著。如果在一个复杂场景中，逐顶点的性能会非常明显地高于逐像素的性能。其次，在tessellated密铺后逐顶点的各粒度中横向比较，可以明显看出随曲面细分因子基值的增加，tessellated密铺后逐顶点光线追踪的fps越来越小，性能降低。这是因为随曲面细分因子的基值增加，曲面细分后得到的顶点越来越多，那么需要在更多顶点上调用光线追踪的计算过程，总开销越来越大，性能降低。最后，逐顶点和tessellated密铺后逐顶点（根据远近获取tess，基值为1）的fps相仿，因此性能相仿。其原因在于当曲面细分因子基值为1时，tessellation stage没有对三角形分出更多顶点，实质上相当于逐顶点光线追踪，因此二者性能相仿。

表1 不同粒度及其对应的FPS

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 粒度 | tessellated密铺后逐顶点（根据远近获取tess，基值为1） | tessellated密铺后逐顶点（根据远近获取tess，基值为5） | tessellated密铺后逐顶点（根据远近获取tess，基值为10） | 逐像素 | 逐顶点 |
| FPS平均值 | 62.0 | 30.0 | 7.0 | 58.0 | 63.0 |

另外，在项目过程中本小组发现即使采用相同的tessellation策略，在正式运行的时候，tessellated密铺后逐顶点光线追踪其帧数相较另外两者低很多，产生的GPU负担尤其显著。本小组认为有以下三点原因：第一，一个几何体在世界空间中必然会有部分面片没有出现或被遮挡在相机视角以外，对这些不可见面元进行细分实际上是一种资源浪费，因为这并不会对最终显示的图像产生任何影响。第二，每一个由镶嵌器输出的三角其内部的顶点都会调用Domain Shader，而这会导致在两面片连接处的顶点被多次重复处理，产生重复计算的资源浪费。第三，在Pixel Shader与Vertex Shader中，两者被调用的次数虽然可能也非常高，但是不会出现重复处理，且DirectX12采取了一些诸如顶点剔除的优化，所以不会出现明显的掉帧现象，在GPU的可承受范围之内。

综合上述定性与定量分析，本小组认为如果要在tessellated密铺后逐顶点光线追踪，一方面，应当选取一个适中的tess值，以达到渲染质量和性能的平衡。另一方面，应当采取一定措施对tessellation stage进行优化。优化的思路是寻找一些不必要的顶点，对这些顶点进行的光线追踪对最后的渲染效果没有影响，通过剔除这些顶点来减少不必要的光线追踪开销。本小组在项目过程中已经进行了优化，因为对离摄像机较远的面片进行细分是一种计算上的浪费，人眼很难察觉到远处细微的变化，所以本小组根据面片与摄像机的距离来设置曲面细分因子的值，使远处细分出更少的顶点甚至不进行细分，但是仍有进一步优化的空间，这非常基础同时效果也不是非常明显。在tessellation后产生的不可见顶点上所做的光线追踪对最后效果没有任何影响，对重复顶点反复进行光线追踪也是一笔不必要的开销，所以应当剔除tessellation后不可见的顶点以及面片边缘处在多个面片连接处的重复顶点，这一过程应当在tessellation stage中尽早进行，例如在向IA提交面片之后就将不可见的面片进行剔除，这样既减少了不必要的tessellation开销，又减少了不必要的光线追踪开销。但本项目的重点不在于此，因此只是提出一些优化设想。

# **特色与创新/ Distinctive or Innovation Points**

相比于传统光栅化方式，光线追踪渲染得到的场景更加接近现实。然而光线追踪的性能这一技术问题依旧是制约其广泛应用的瓶颈。本小组为了在保证光线追踪质量的前提下提高光线追踪性能，基于稀疏化光线样本粒度这一思路完成了本项目。利用tessellation来实现对粒度的灵活控制，并实现了逐像素、逐顶点、tessellated密铺后逐顶点等三个粒度下的光线追踪，进行定性和定量的分析后给出了一种稀疏化光线样本粒度以提高光线追踪性能的可行方案，能够在保证光线追踪质量的前提下确保实时性。本小组还对tessellation后逐顶点光线追踪进行了一些优化，减少不必要的顶点数量以减少光线追踪耗时，进一步确保实时性。另外，本小组创新性地使用了DirectX12的inline raytracing新特性，简化了在不同阶段进行光线追踪的问题。

# **补充说明：**

6.1、术语：

|  |
| --- |
| GPU：图形处理单元，进行绘图运算工作的处理器。  渲染流水线：渲染流水线是指以一台具有确定位置和朝向的虚拟摄像机进行观察，根据某个3D场景的几何描述生成该3D场景的2D图像的一整套处理步骤。  Tessellation stage:曲面细分阶段，指渲染流水线中参与对几何图形进行镶嵌处理的阶段。镶嵌处理指将几何体细分为更小的三角形，并以某种方式把这些新生成的顶点偏移到合适的位置，从而以增加三角形数量的方式丰富网格的细节。  Patch：面片，具有若干个控制点。控制点是用来面片构造过程的点，控制点越多，对面片形状绘制的控制越容易。  Tessellation factor (tess)：曲面细分因子，指示了在曲面细分阶段中将面片镶嵌处理后的份数。  Inline RayTracing:一种光线追踪技术，使得不再需要新建一个独立的光线追踪管线，易于在各个shader stage进行不同粒度的光线追踪  世界坐标系：又称全局场景坐标系，是一种假象的绝对坐标系，不依赖于任何物体，在指定后不变且唯一。  局部坐标系：通常是一种以目标物体的中心为原点（也有例外，视情况而定），并且坐标轴与该物体对齐的简易便用坐标系。  世界变换：将局部坐标系内的坐标变换到世界坐标系中的坐标的过程叫作世界变换，所使用的变换矩阵叫作世界矩阵。  FPS：frames per second，每秒渲染的帧数，FPS越高，显示效果越流畅。在本项目中，FPS用来衡量光线追踪的性能，FPS越高说明光线追踪耗时越短，性能越高。 |

References:

1. Andreas Polychronakis, George Alex Koulieris, and Katerina Mania. 2021. Emulating Foveated Path Tracing. In Motion, Interaction and Games (MIG '21). Association for Computing Machinery, New York, NY, USA, Article 10, 1–9.
2. Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. 2007. Interactive k-d tree GPU raytracing. In Proceedings of the 2007 symposium on Interactive 3D graphics and games (I3D '07). Association for Computing Machinery, New York, NY, USA, 167–174.
3. Tie Jun, Liu Guopeng, Zheng Lu. Ray tracing algorithm of grid structure based on inner bounding box [J]. Computer and Digital Engineering, 2017,45(05):817-819+906.
4. Ulises Olivares et al. Efficient construction of bounding volume hierarchies into a complete octree for ray tracing[J]. Computer Animation and Virtual Worlds, 2016, 27(3-4) : 358-368.
5. Shipeng Feng and Dongxu Li. A research on novel direct vector method for ray tracing[J]. Optik - International Journal for Light and Electron Optics, 2013, 124(23) : 6377-6381.
6. 《DirectX 12 3D游戏开发实战》（美）弗兰克. D. 卢娜 (Frank D. Luna) 著；王陈译 北京：人民邮电出版社， 2019. 1

Docs Webistes:

1. DirectX 12示例代码:https://github.com/microsoft/DirectX-Graphics-Samples/tree/master/Samples/Desktop
2. DirectX 12开发文档:https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics
3. VRS文档: https://docs.microsoft.com/en-us/windows/win32/direct3d12/vrs
4. Inline Raytracing文档: https://github.com/microsoft/DirectX-Specs/blob/master/d3d/Raytracing.md#inline-raytracing

时间安排与分工统计表

|  |  |  |  |
| --- | --- | --- | --- |
| **组员信息**（含组长） | | | |
| 学生姓名 | 叶增渝 | 学 号 | 519030910168 |
| 项目分工 | DX12项目实现、PPT制作与演讲、使用文档与效果演示制作 | | |
| 学生姓名 | 李超 | 学 号 | 519030910198 |
| 项目分工 | 尝试实现fallback layer raytracing、项目总结报告撰写 | | |
| 学生姓名 |  | 学 号 |  |
| 项目分工 |  | | |
| **时间安排/**  **Schedule** | 2021.10.29选题  2021.10.30-2021.11.14前期调研  2021.11.15-2021.11.28前期知识储备与学习  2021.11.29-2021.12.19研制开发，代码调试  2021.12.04-2021.12.05撰写中期报告  2021.12.20-2021.12.25总结实验结果，撰写总结报告，准备presentation | | |