不同粒度的光线追踪

秦健行， 张亦昕， 林心怡

摘 要: 光线追踪是能够在渲染中获得高质量图像的技术，但性能上与传统光栅化模型存在差距。本小组以提升光线追踪性能为目标，尝试三种不同粒度：逐像素、逐顶点、曲面细分之后逐顶点，以分析探究着色粒度对于光线追踪性能的影响。实现过程利用了Microsoft DirectX12的编程接口和图形化流水线。项目的优势在于采用不同的粒度进行光线追踪，运用多条图形与计算流水线进行处理，最终达到预期的实验效果。

Raytracing in different granularities

Jianxing Qin, Yixin Zhang, Xinyi Lin

**Abstract**: Ray Tracing is a technology that can obtain high-quality images in rendering, but its computational performance is its drawback. The experiment group attempted to implement raytracing with three different granularities, per pixel, per vertex and per tessellated vertex, aiming to improve the performance of ray tracing. This group uses Microsoft DirectX12 API and graphics pipeline to analyse the influence of granularity on ray tracing performance. The advantages of the project are utilizing various granularity to implement ray tracing, combining multiple graphics pipeline to process the images in order to achieve the aim of project.

**Key word: Ray Tracing, Tessellation, DirectX 12 API, Graphics Pipeline**

# 简介与意义/Introduction

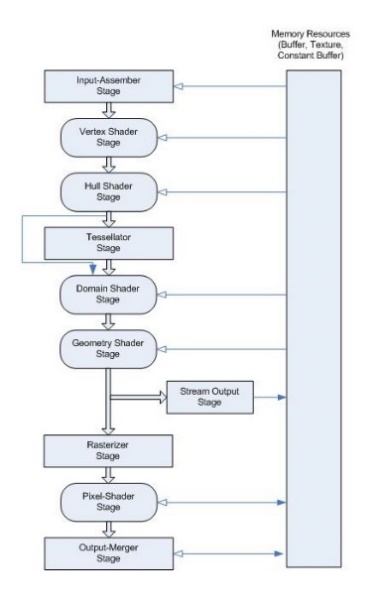
## 项目意义和依据/Significance

光线追踪技术以其出色的渲染质量在离线渲染中已经得到了很广泛的应用。近几年随着硬件性能的提升和算法的改进，实时光线追踪技术也在不断发展。以NVIDIA公司推出RTX系列GPU为标志，实时光线追踪技术在高画质3D游戏、虚拟现实等应用场景下体现出了很大的优势，前景广阔。

然而现有的实时光线追踪实现相较传统光栅化手段在性能上仍然存在较大差距。以全局光照为例，现有的近似实现已经能够产生高效高质量实时全局光照，但是其耗时依旧为传统屏幕空间反射、阴影的两三倍。

本小组将尝试进一步稀疏化光线样本的粒度，减少光线追踪数量，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。

## 本方法/系统框架/Article Structure



图表 1

图表 1展示了Direct3D 12的图形渲染管线，本小组将通过在不同的管线阶段（Vertex Shader, Pixel Shader）实现光线追踪算法，或改变某些管线阶段的顺序（Vertex Shader）来实现不同粒度的光线追踪，同时评估这些实现的渲染质量、渲染速度等性能表现。

更进一步地，本小组将探索在以上管线的不同阶段实现的可变着色率，以探索光线追踪在可变着色率技术下的性能表现。

# 相关工作/Related Works

Bongjun Jin等提出了一种适用于实时光线追踪的自适应超采样算法[5] ，以用户选择性设置的优先级为基础，通过图像与物体的属性来自适应地分布采样点，以最小化光线追踪计算量。

Qing Xu等提出了一种适用于蒙特卡洛全局光照的自适应采样技术。通过对熵在信息论领域应用的研究，尤其是非泛化的Tallis熵，来衡量像素质量和进行自适应采样。

Youngwook Kim等提出了一种高效的自适应欠采样方法[7] ，将部分光线追踪运算替换为插值运算以减小运算量，同时采用一种后校正算法来最小化欠采样带来的失真。

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

## Direct3D 12 渲染管线

Direct3D[4] 是由Microsoft公司设计开发的底层渲染/计算API，其最新版本是Direct3D 12。本小组将主要利用这套API完成开发工作。

从Direct3D 11开始，渲染管线相比以往发生了改动[1] ，如图表 1。从细节上来说，增加了曲面细分（Tessellation）功能，主要包括外壳着色（Hull Shader Stage）， 曲面细分（Tessellator Stage）和域着色（Domain Stage）阶段。

具体的渲染管线各阶段如下：

Input Assembler Stage（输入汇编器阶段）

该阶段是图形处理流水线的第一个阶段，在这个阶段，汇编器从用户输入缓冲区读取原始的数据，比如点，线和三角形等，将这些原始未处理的数据，经过简单组装之后，附加系统生成的一些数值，提供给流水线的其他阶段，以提高整个光线追踪的效率。具体实现包括原始拓扑和系统生成数值。原始拓扑就是将读入的点或者图形进行处理，形成带有基本信息的图形基元，如三角形的邻点和旋转方向等。系统生成数值，即对于三角形的每一个顶点进行标号，方便后续进行条带切割等，将每个三角形分开。

输入：从用户缓冲区读取的点，线和三角形。

输出：组装好并附加好数据的基元，提供给其他流水线阶段

Vertex Shader Stage（顶点着色器阶段）

对于单个顶点，在该阶段中会将从输入汇编器中获得的顶点进行变形，纹理，变换，灯光处理等。这一步中会对图形所有的顶点都进行处理。

输入：具有 VertexID 和 InstanceID，即系统生成数值的顶点。

输出：经处理过的单个顶点。

Hull Shader Stage（外壳着色器阶段）

这是一个可以通过编程修改的阶段。主要进行两个步骤，控制点阶段和patch-constant阶段。

控制点阶段对于每一个控制点运行一次，读取该控制点，并生成一个输出控制点。

patch-constant阶段，对于生成的patch，生成他们的边缘曲面细分因子以及其他的一些附加数据。这一阶段对于输入输出的控制点，只有读访问权限，可以对于多个patch并行计算。

输入：控制点1至32个，它们形成一个低阶曲面。

输出：控制点1至32个，它们形成一个几何块（patch），为后续的处理使用。

Tessellator Stage（曲面细分阶段）

创建域的采样模式，表示patch并生成一组连接这些样本的较小对象，例如三角形，点或者线。从低细节细分曲面开始，将它们转换为 GPU 上的高细节基元。细分平铺（或分解）高阶表面为适合渲染的结构。还会将高阶曲面（使模型保持简单和高效）转换为许多三角形，以便在 Direct3D 图形流水线内进行详细渲染。

分为两个阶段，第一阶段使用32位浮点算法处理细分因子，第二阶段根据所选的分区类型生成点或者拓扑列表。

输入：使用从外壳着色器阶段传入的细分因子和分割patch的算法，对于每个patch运行一次。

输出：对于patch，曲面细分会生成好表面纹理的（u，v，w）相对坐标和表面的拓扑，输入至下一阶段。

Domain Shader Stage（域着色阶段）

计算输出patch中细分点的顶点位置，在每一个曲面细分输出点运行一次，具有输出patch及其数据的读访问权限。

输入：来自外壳着色器的输出控制点。同时每一个来自曲面细分阶段的输出坐标都会调用一次域着色器。

输出：输出patch中细分点的顶点位置。

域着色器完成后，曲面细分完成，管线数据继续到下一个管线阶段。

在本小组的实现中，由于此处引入曲面细分的目的是为了增加光线追踪时采样点的数目，而不在意细分后的曲面是否光滑，故此处不采用较为复杂PN三角形细分方法，而选择使用较为基础的基于三角形重心坐标的线性插值。在编写外壳着色器(Hull Shader)时，顶点信息从顶点着色器（Vertex Shader）输出后，hull shader以每个三角形图元的三个顶点为input patch，由于线性插值只需要图元的三个顶点作为控制点，因此hull shader main仅仅将输入顶点的坐标与法向量传递给domain shader，而不生成其他控制点。

外壳着色器在配置曲面细分（Tessellator）时，此处本小组为三角形图元边界和内部选择同样的细分因子n，以便后续步骤导出细分后的模型信息，细分因子n也是用于控制本项目中光线追踪粒度的主要因素。同样，为与之前步骤保持一致，输出图元结构的顺序也选择顺时针环绕。

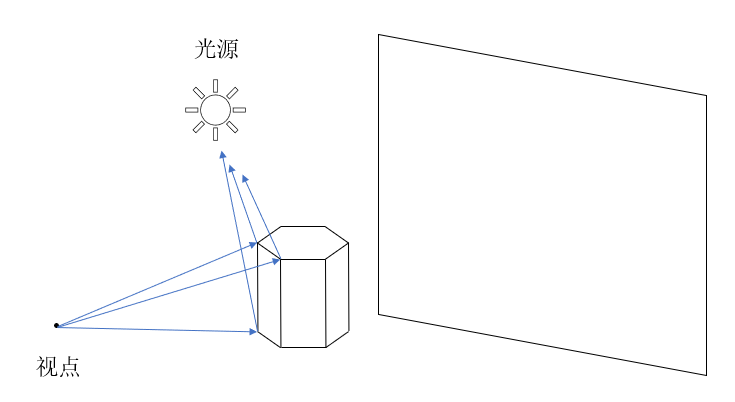
曲面细分之后的域着色器（Domain Shader）读入Hull shader产生的图元的控制点信息和Tessellator产生的uv坐标，并对控制点的位置坐标和法向量进行线性插值。注意该阶段输出的顶点位置坐标为齐次裁剪空间中的4维向量。

## 光线追踪算法

本项目最基本的两种算法是逐顶点的光线追踪算法与逐像素的光线追踪算法，在此基础之上，本小组通过曲面细分（Tessellation）等技术调整采样方式，实现不同粒度的光线追踪，并对比分析其性能的差异。

逐顶点的光线追踪算法

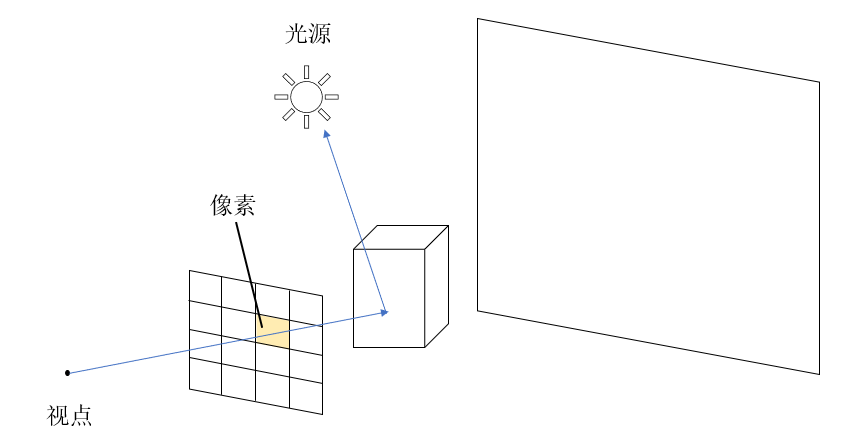
逐顶点的光线追踪算法主要在vertex shader中完成。如图表 2，视点向场景中的每一个顶点分别投射光线，并通过追踪该光线，为场景中的每一个顶点计算着色。在此之后在pixel shader中，将场景投影到屏幕空间上并进行光栅化处理，通过对顶点的线性插值来计算其他像素的颜色。



图表 2

逐像素的光线追踪算法

逐像素的光线追踪算法在pixel shader中完成。如图表 3，视点分别向屏幕空间中的每一个像素投射光线，并通过追踪所投射的光线，直接计算出屏幕空间中每一个像素的着色。



图表 3

在场景中模型的镶嵌度较高时，逐顶点光线追踪能够取得相对较好的效果，而大多数情况下模型的精度不足以使逐顶点的光线追踪达到令人满意的效果，这一点在模型三角形面积较大时尤为明显。相较于逐顶点光线追踪，逐像素的光线追踪算法进行了更为密集的采样，能达到更好的图像质量。然而，更高的采样密度意味着更高的开销，很多不必要的采样不仅无法改善图像质量，更严重影响了逐像素光线追踪的性能。

由于两种光线追踪算法存在上述不足，本项目通过Hull Shader控制曲面细分来调节顶点密度、改善模型精度从而提升图像质量；同样，可变着色率（VRS）

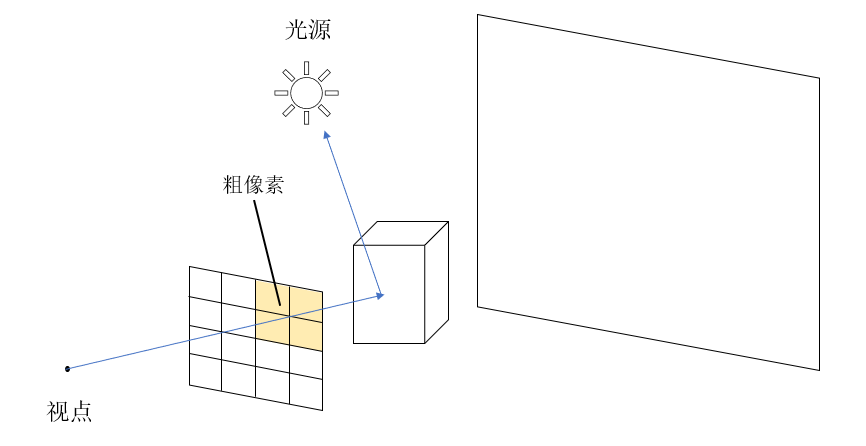
VRS（Variable Rate Shading）是一种独立于渲染目标分辨率和光栅化率来指定像素着色器调用率的技术，允许改变局部区域内像素着色的次数[2] 。起因是当前 GPU 的原始性能并不总是能够跟上当前显示器中像素数量所需的执行吞吐量，而使用可变速率着色可以保持场景的视觉保真度的同时降低某些位置的着色率来提供高质量的视觉效果。

当今使用的像素绑定时提高帧速率的软件技术，如动态分辨率渲染（DRR）和棋盘格渲染 （CBR），使用的是相关软件技术，在性能上存在一定的缺陷。而硬件方面可以实现更好的性能和图像质量。

适用VRS技术的情况: 远处的物体， 运动中的物体，对象发生模糊或者视觉失真时，用户视觉周边的物体，动态分辨率渲染和省电等。

具体技术[1] 是通过像素的合并，在使用MSAA（多重采样抗锯齿）超采样技术的图像中，对于目标像素或者目标图像块，将一组像素作为一个单元进行着色，然后将结果广播给该组中的所有样本。

如图表 4，引入粗像素（Coarse Pixel）的概念，在确保图像质量的前提下稀疏化光线样本的密度，以求改善光线追踪的性能。



图表 4

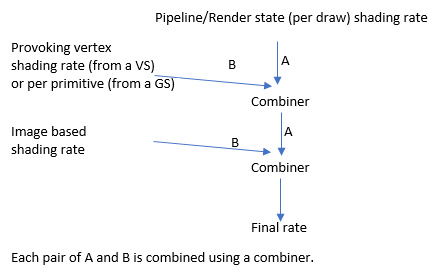
具体支持程度见下表：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 粗像素尺寸 | 1xMSAA | 2xMSAA | 4xMSAA | 8xMSAA | 16xMSAA |
| 1x2 | 支持 | 支持 | 支持 | 不支持 | 不支持 |
| 2x1 | 支持 | 支持 | 支持 | 不支持 | 不支持 |
| 2x2 | 支持 | 支持 | 支持 | 不支持 | 不支持 |
| 2x4 | 有条件的支持 | 有条件的支持 | 不支持 | 不支持 | 不支持 |
| 4x2 | 有条件的支持 | 不支持 | 不支持 | 不支持 | 不支持 |
| 4x4 | 有条件的支持 | 不支持 | 不支持 | 不支持 | 不支持 |

实现此技术时，会在原图像基础上建立一个新的层，这个层有许多相关性质，使得像素合并，着色等操作更易于在此层中实现。

附加的Tier 2，每次描绘可以选择不同的着色率，从三种来源获得的的着色率可以被组合器组合起来。屏幕空间图像平铺大小可为 16x16 或更小。同时可以保证应用程序请求的着色率准确传递。在Tier 2 中，应用可以创建一个“LOD mask”图像，指示不同质量的区域，采用不同尺寸的粗像素。可以在流水线的多个阶段，VS，GS和MS阶段中设置SV\_ShadingRate，确定粗像素着色率，默认值为1x1，即不进行粗粒度着色。

着色率的来源如图表 5:



图表 5

组合器（Combiner）可以根据需求进行选择，如选择最大值或最小值等。

像素着色器的输入基于其源顶点进行插值。由于可变着色率会影响像素着色器每次调用写入的目标区域，因此它与属性插值交互。三种类型的插值是中心，质心和样本。粗像素的中心插值位置是整个粗像素区域的几何中心，质心插值位置将考虑粗像素内细像素的所有样本。

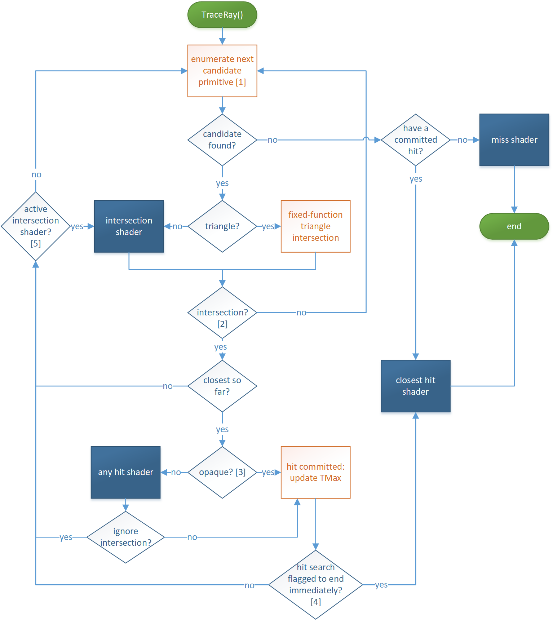
对于粗像素，VRS在Tier 2中使用一个tile对其进行分析和处理，描述的渲染目标区域的宽度和高度，会影响应用程序想要分配的屏幕空间图像的大小。tile一般是16x16。offset参数为每个精细像素选择一个16x16的网格，有多个精细像素。可表达的范围和随后使用的位数取决于粗略的像素大小。

## 本课题组使用的实验方法

本实验组计划使用三种不同的粒度，以图像的帧率为指标来验证性能和光线追踪粒度的关系。使用如下三种不同的粒度：

对于每个像素进行光线追踪；对于每个顶点进行光线追踪；在原模型基础上进行曲面细分，增加顶点个数，再进行光线追踪计算颜色。

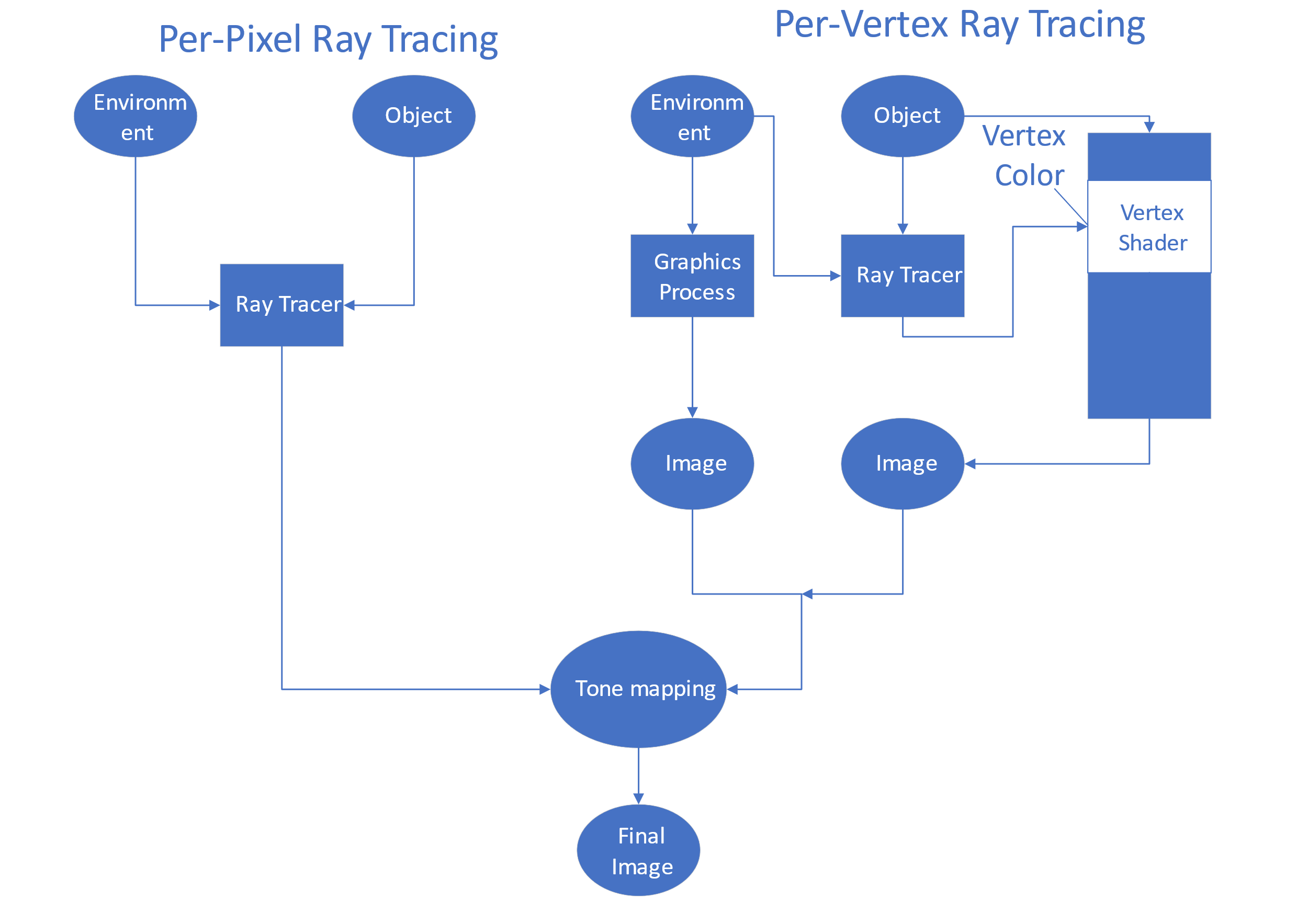
### 逐像素光线追踪和基本的光线追踪实现



图表 6

基本的光线追踪理论已经在上文介绍，具体到DirectX12，我们主要使用其TraceRay() 函数，该函数如图表6所示，首先判断光线是否与物体相交，未能相交则交由miss shader处理这一情况。相交时，首先判定相交基元是否为三角形，若非三角形，则需通过intersection shader 求交，三角形则可自动求出与物体的交点。与物体的最近对于与物体的其他交点，可以根据需求交由any hit shader进行处理。主要由 closest hit shader 通过与物体最近的交点来计算物体表面的颜色，在本实验组的工作中，也主要实现了表层颜色的处理。

在本实验组的实现中，对于生成的光线设定了一个深度，光线的反射和计算次数不能超过该深度，超过则视为光线未相交，按照对应情况处理；如果光线未能与物体相交，就将像素颜色着色为环境颜色。如果与物体上像素相交，该点颜色有两部分组成，反射颜色和Phong光照颜色。反射颜色由一个反照率系数和反射光线方向，交由TraceRay() 函数进行光线追踪得到。Phong光照颜色由几部分共同构成，在未处于阴影中时主要是漫反射颜色和高光颜色，如果处于阴影中，该光线就被视为阴影光线(Shadow Ray)，阴影处的颜色没有高光。将这两部分加在一起，就得到光线首次与物体相交处的颜色，储存在RayPayload数据结构中。



图表 7

### 逐顶点光线追踪

逐顶点的光线追踪在光线与着色方面，与逐像素着色相似，如图表 7，其区别在于，需要多条图形流水线进行处理.

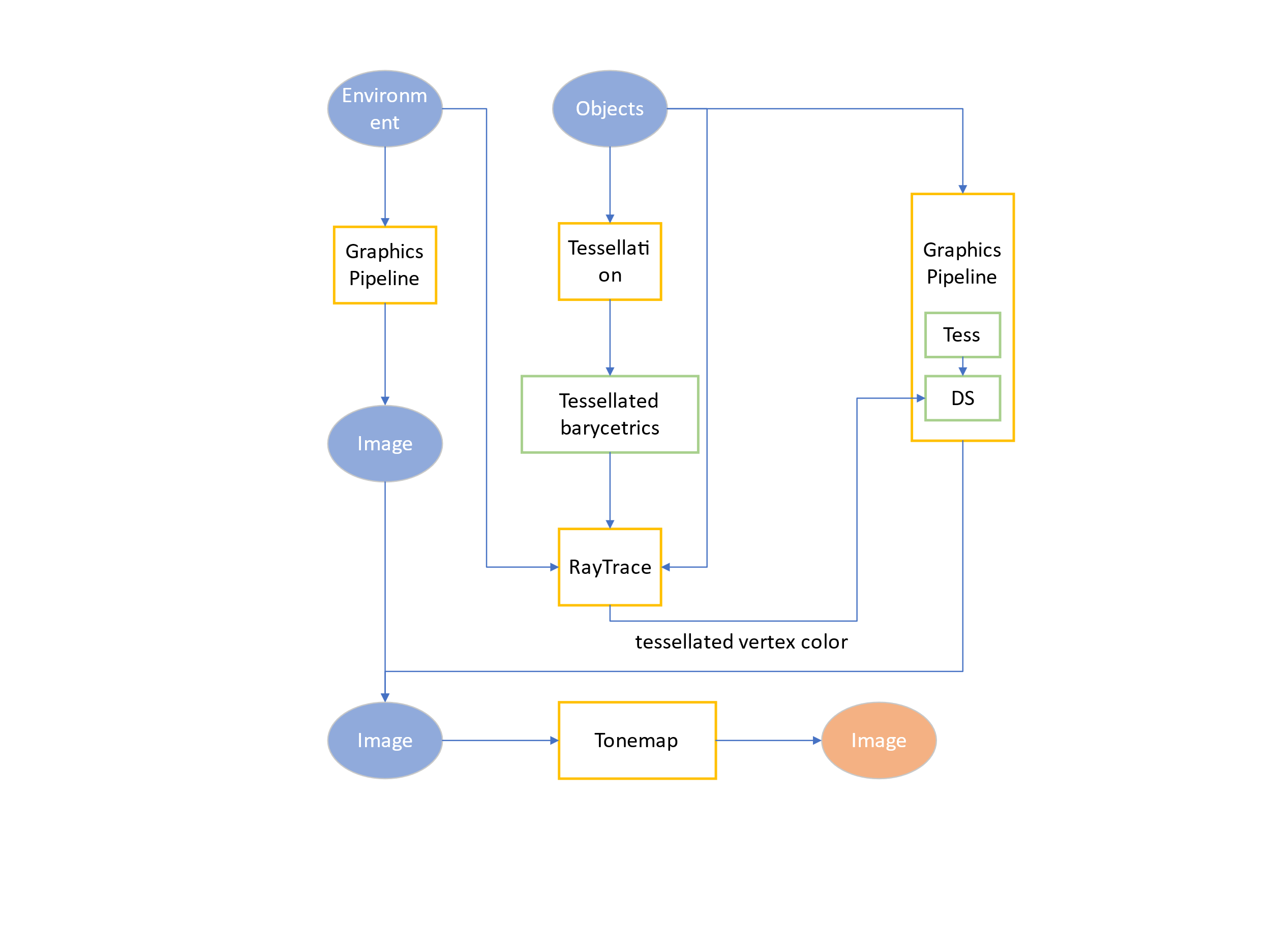
首先基本的着色后的模型，需要再次放入流水线中执行一次光栅化，对其每个像素进行着色，才能正常显示。还需要一条单独的流水线，利用Z-Buffer算法，计算像素点的深度。还设置了其他流水线，用于计算背景颜色以及Tone Mapping。Tone Mapping用于解决场景中对比强烈导致无法正常显示的问题，使之在显示器上有一个合理的显示。

将模型进行光线追踪，可以确定好每个顶点的颜色，之后利用光栅化流水线，就可以将处理好的模型像素化，使之能够正常显示。

### 基于曲面细分的逐顶点光线追踪

基于曲面细分的逐顶点光线追踪流程见图表 8。

因为底板(Ground Mesh)在每个面只有两个三角形进行光线追踪时稳定程度不高，图像效果差，本实验组重新实现了底板的细分方式，可以按照需求进行更细粒度的曲面划分，将每个面划分为更多的三角形，生成更多的顶点。将底板按照正方体进行顶点生成划分，之后指定顶点组合成三角形面片，在相对的面上指定不同的顶点环绕方向，就可以使每个平面上的面片正常显示。这样在每个平面上可以更灵活的感知光线，使光线追踪之后的底板随着物体反射光线变得更加协调。

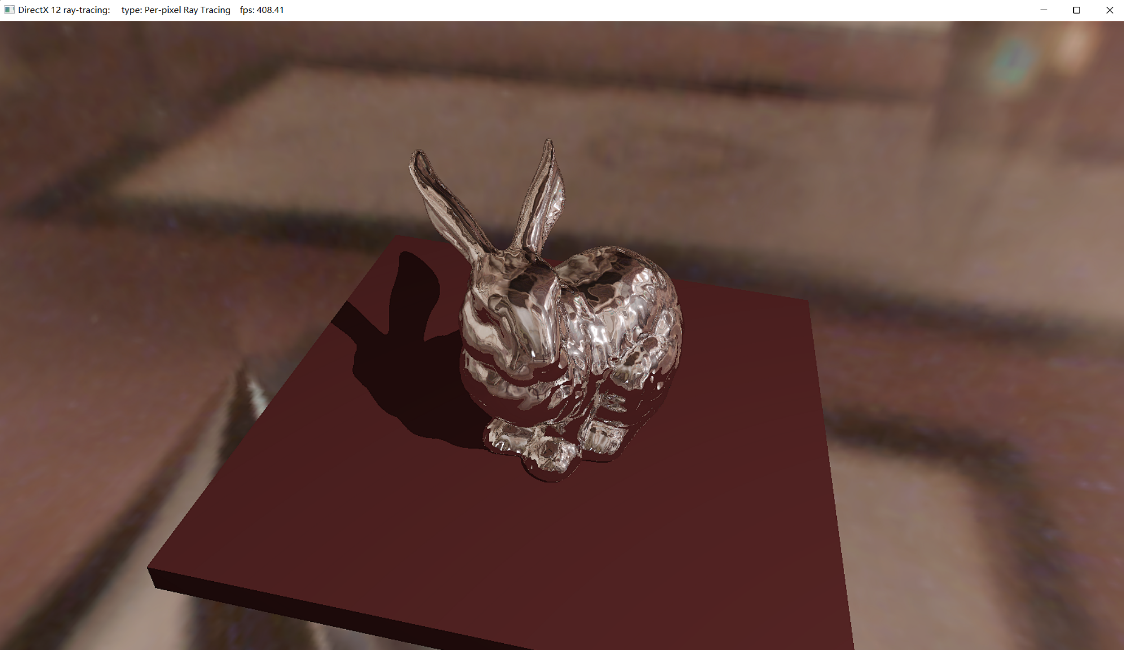


图表 8

在实现对定点进行曲面细分之后的逐顶点光线追踪中，本小组遇到了一定的困难。小组成员的显卡配置不支持Inline Ray Tracing，所以只能使用如下方法。将物体进行曲面细分之后，进行光线追踪确定颜色，得到控制点和颜色的对应关系，但是这样无法把之前已经计算好的模型完整导出，从而无法将颜色映射到模型表面。因此，本小组在实现逐顶点曲面细分光线追踪时，使用了如下方法：在曲面细分时，首先根据曲面细分程度得到多个顶点在一个面片（patch）内的重心坐标，我们将该坐标通过哈希函数（hash）映射到一维数组中。每一个面片映射成最长为MaxVertexPerPatch的一个一维数组。这样所有的面片就可以通过光线追踪计算得到颜色，得到一个由颜色三维向量构成的数组。由于在这条图形流水线中处理的曲面细分模型无法导出，本小组采用了另一条图形流水线，重新计算一个完全相同的曲面细分模型，将颜色与模型上的控制点一一对应，这样就得到了一个着色好的模型，之后与逐顶点光线追踪相同，再进行光栅化等步骤即可得到最后的图像。

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

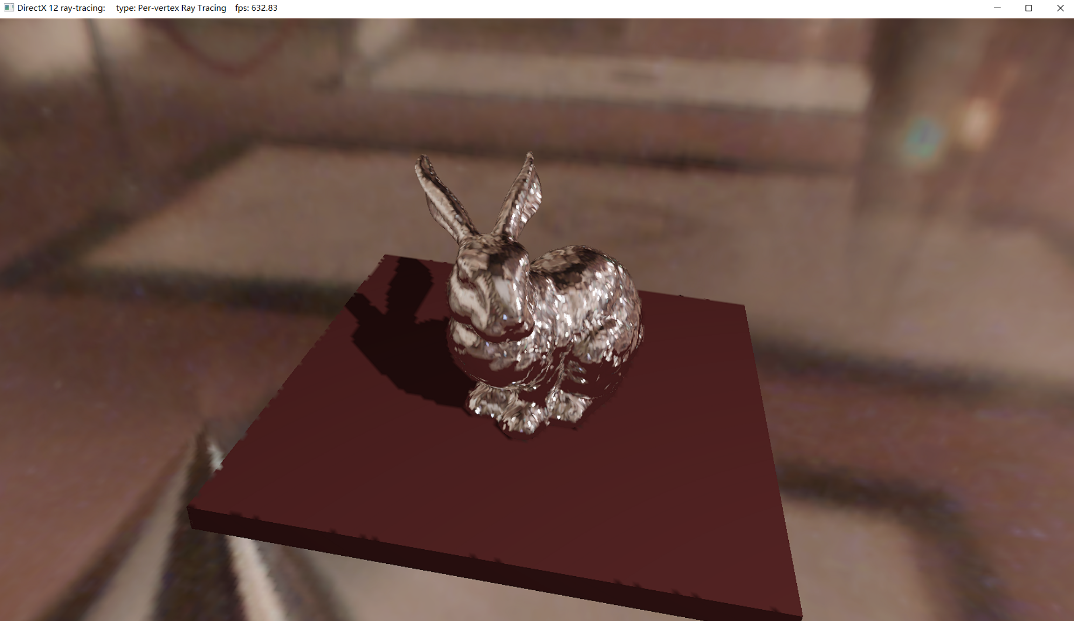
逐像素光线追踪：



图表 9

如图表 9所示，在逐像素的光线追踪条件下，图像的质量较高，表面的像素点可以体现比较强的真实感，帧率在400帧每秒左右。

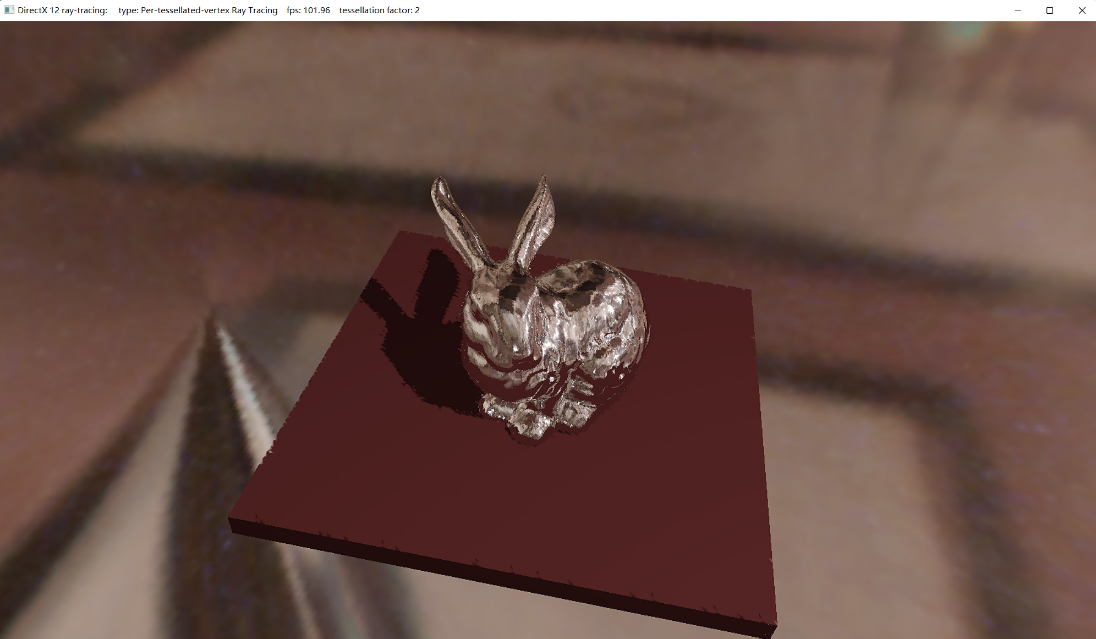
逐顶点光线追踪：



图表 10

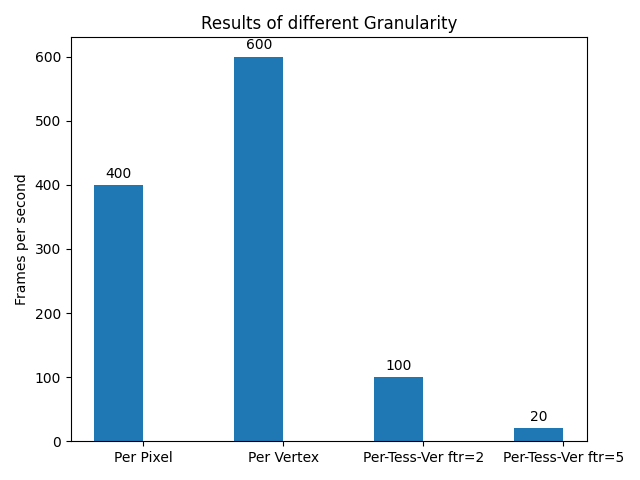
如图表 10所示，与图表 9相比，可以看到图像的质量出现了一定的下降，模型表面的色块变得更大了一些，同时在底板上，也采用了细分方法，可以看到地板上的投影出现了一定的锯齿现象，这是采用每条边细分为64段之后，以三角形顶点为单位进行了光线追踪，而产生的正常现象。降低光线追踪粒度之后，可以看到帧率上升到了600至700左右。反映出了加粗粒度带来的性能提升。

曲面细分之后，逐顶点光线追踪效果见图表 11：



图表 11

可以看到，图像的质量相对于逐顶点的光线追踪有所提升，同时本小组设置了曲面细分因子，提高其数值可以将每个三角形面片分的更细，细分因子为2时，帧率在100帧每秒左右。当细分因子为5时，图像以与逐像素光线追踪没有很明显的区别，但是帧率只有20左右。



图表 12

我们可以看到，理论上来说，曲面细分之后的粒度要比逐像素光线追踪更粗一些，而比逐顶点光线追踪更细一些，因此帧率上应该介于两者之间。但是事实上图表 12统计的帧率却出现了比较大的下滑。这是因为本小组在实现曲面细分时，计算了两次曲面细分，不同三角形面片细分之后还存在着边缘控制点重复计算的问题。同时将光线追踪计算得到的颜色存储在缓存区内，用哈希函数不断频繁进行访问映射，这些导致整体计算成本快速增加，所以帧率比较低。

# 特色与创新/ Distinctive or Innovation Points

传统的光线追踪只在逐像素层面进行，本小组工作的创新之处在于尝试了不同层面的光线追踪来比较其性能差距，这都需要完全理解与掌握DirectX12 流水线，灵活运用其各种API。同时在助教老师的协助下。我们利用助教老师提供的项目基础架构和图形数据[9] ，在此基础上实现了逐顶点的光线追踪。同时结合上课盛斌教授提及的曲面细分以提高图形质量的方法，应用到光线追踪中，实现了曲面细分之后的逐顶点光线追踪。同时利用哈希函数和两次细分分别计算颜色和模型的方式解决了无法将模型传出流水线的问题。最后成功对比了不同粒度条件下性能的差距，同时分析了曲面细分导致性能下降的原因，这些都是本小组结合算法实现的创新之处。

# 补充说明：

## 术语：

光线追踪(Ray Tracing)：在二维屏幕上呈现三维图像的方法，用虚拟的光线反射折射等对于物体进行着色

API(Application Programming Interface)：应用程序编程接口，在代码中调用硬件设备进行处理的函数或者变量。

内联光线追踪[10] (Inline Ray Tracing)：另一种形式的光线追踪，可以不使用任何单独的动态着色器或着色器表。可用于任何着色器阶段，将追踪得到的结果导出，放在缓冲区内随时可用。

哈希函数(Hash)：又叫散列函数，把输入按照一定的散列算法变化成与输入相关的固定长度的输出，便于查找和定位。

# 致谢：

感谢盛斌教授开设这样一门优秀的课程，让我们能够接触到计算机图形学、光线追踪等与前沿技术接轨的概念，深入浅出地讲授了图形学中的许多概念与算法，让我们有了更好的理解。感谢助教徐添辰老师给我们提供了基础的项目架构，以及在项目实现过程中一直耐心给我们答疑解惑。感谢上海交通大学计算机科学与工程系给作为学生的我们安排了如此高质量的课程，让我们从中获益匪浅。

References:

1. Direct3D Graphics Pipeline <https://docs.microsoft.com/en-us/windows/uwp/graphics-concepts/graphics-pipeline>
2. Introduction to Variable Rate Shading (VRS) <https://www.intel.com/content/www/us/en/developer/articles/guide/getting-started-with-variable-rate-shading-on-intel-processor-graphics.html>
3. Direct3D 12 Variable Rate Shading <https://microsoft.github.io/DirectX-Specs/d3d/VariableRateShading.html>
4. Direct3D documentation <https://docs.microsoft.com/en-us/windows/win32/direct3d>
5. Bongjun Jin, Insung Ihm, Byungjoon Chang, Chanmin Park, Wonjong Lee, and Seokyoon Jung. 2009. Selective and adaptive supersampling for real-time ray tracing. In *Proceedings of the Conference on High Performance Graphics 2009* (*HPG '09*). Association for Computing Machinery, New York, NY, USA, 117–125. DOI:https://doi.org/10.1145/1572769.1572788
6. Q. Xu, M. Sbert, M. Feixas and J. Sun, "A New Adaptive Sampling Technique for Monte Carlo Global Illumination," 2007 10th IEEE International Conference on Computer-Aided Design and Computer Graphics, 2007, pp. 191-196, doi: 10.1109/CADCG.2007.4407879.
7. Kim, Y., Seo, W., Kim, Y. *et al.* Adaptive undersampling for efficient mobile ray tracing. *Vis Comput* **32,** 801–811 (2016). <https://doi.org/10.1007/s00371-016-1251-y>
8. Wikipedia: Tone Mapping <https://en.wikipedia.org/wiki/Tone_mapping>
9. Github Stars XUSG： <https://github.com/StarsX/XUSG>
10. Inline Raytracing: https://devblogs.microsoft.com/directx/dxr-1-1/

时间安排与分工统计表

|  |  |  |  |
| --- | --- | --- | --- |
| **组员信息**（含组长） | | | |
| 学生姓名 | 秦健行 | 学 号 | 519030910270 |
| 项目分工 |  | | |
| 学生姓名 | 张亦昕 | 学 号 | 519030910273 |
| 项目分工 |  | | |
| 学生姓名 | 林心怡 | 学 号 | 519030910262 |
| 项目分工 |  | | |
| **时间安排/**  **Schedule** | （如选题、方案制定、试验研究、数据处理、研制开发、撰写总结报告等）（Such as topic selection， program formulation， experimental research， data processing， research and development， writing summary reports， etc。） | | |