实时光线追踪

祝昊焜 52001910058　李宁 520021910074

2022 年 12 月 30 日

摘　要：在计算机图形学渲染领域，光线追踪算法被认为是一个有效、并被广泛使用的真实感绘制算法，光线追踪因其渲染效果的真实性，长期以来被视为下一代主流图像渲染技术。但光线追踪技术极其消耗资源，大多应用在离线渲染中，在实时渲染中多数只能在低分辨率、低帧率下模拟。本项目对实时光线追踪进行尝试，利用光线追踪算法对几何形体以及简单的网格模型进行实时渲染，实现了阴影、反射、折射等视觉效果，本项目的实现过程利用了 OpenGL 的编程接口以及图形渲染管线，以最大化利用硬件性能。本项目的优势在于，成功实现了光线追踪算法并将其应用在实时渲染领域，实现了阴影、反射、折射等多种视觉效果，使用 SMAA 算法实现抗锯齿，兼顾了性能与效果，借助 BVH 提高了实时光线追踪的性能，实现了即使是渲染多边形网格模型也能达到足够的流畅度。

Real-Time Ray Tracing

Haokun Zhu, Ning Li

Abstract: In computer graphics rendering, ray tracing algorithm is considered as an effective and widely used realistic drawing algorithm. Ray tracing has long been regarded as the next generation mainstream image rendering technology due to its realistic rendering effect. However, ray tracing techniques are extremely resource-consuming and are mostly applied in offline rendering, and almost can only be applied at low resolution and low frame rate situation in real-time rendering. This project try to implement real-time ray tracing, using ray tracing algorithms for real-time rendering of geometric shapes and simple mesh models to achieve visual effects such as shadows, reflections, refractions, etc. The implementation process of this project utilized the OpenGL programming interface and the graphics rendering pipeline to maximize the use of hardware performance. The advantage of this project is that it successfully implements ray tracing algorithm and applies it in real-time rendering, realizing various visual effects such as shadows, reflections, refractions, etc. Use SMAA algorithm to achieve anti-aliasing, taking into account the performance and efficiency. Besides, this project improves the performance of real-time ray tracing with the help of algorithms such as Bounding Volume Hierarchy(BVH), and achieves good performance even when rendering polygonal mesh models.

Key ward: Real-Time Ray Tracing, OpenGL, Graphics Pipeline, BVH, SMAA

1　简介与意义/Introduction

1.1　项目意义和依据/Significance

渲染是将三维场景的描述转换为二维图像的过程, 在动画、电影特效、游戏、几何建模等领域中得到了广泛应用。渲染主要包括光栅化和光线追踪两种方式。光栅化渲染采用局部光照原理，根据光源照射到物体上直接可见的光照效果，将场景中的几何图元映射到图像的像素点上。这种方式用硬件实现较为简单，且便于并行处理，通常用作实时渲染处理。光线追踪采用全局光照模型，通过物理原理对光线和物质之间的交互行为进行建模，不仅考虑直接光照的效果，也考虑物体间相互光照影响，比传统的光栅化渲染效果更加立体，色彩更柔和更逼真，通常用于离线渲染处理。

由于光线追踪渲染计算量巨大且非常耗时，实时光线追踪存在很多挑战。首先，光线追踪算法需要迭代测试光线与场景图元是否相交，并计算光线与图元的最近交点，这个计算量是巨大的；其次，光线的反射和折射会产

生大量的二次光线，进一步增加了计算复杂度，而且这些光线随着迭代次数的增加会变得越来越不连续，不规则的内存访问会导致带宽的瓶颈。因此，现有的实时光线追踪相比于传统光栅化手段在性能上仍然存在较大差距。

为了测试和提升实时光线追踪的效果，本项目借助 OpenGL 框架实现了一个简单的实时光线追踪场景。通过实现 BVH，提高了渲染的效率，以支持多边形网格模型的实时渲染。此外，利用 SMAA 实现了抗锯齿效果，与 MSAA 相比，其在效果接近的情况下节省了大量的计算资源。

1.2 本方法/系统框架/Article Structure

本项目是在 OpenGL 上实现的，借助了 OpenGL 提供的图形渲染管线，图1展示了 OpenGL 的图形渲染管线的框架结构 [8]，每个部分都在转换顶点数据到最终像素这一过程中处理各自特定的阶段。本项目通过完成顶点着色器和片段着色器来实现光线追踪算法，顶点着色器主要用来接收顶点位置信息，并将其传递给片段着色器，片段着色器中实现了光线追踪算法，将在第3部分进行具体介绍，BVH 的部分实现也是在片段着色器中完成的，实现了对光线追踪算法的加速。

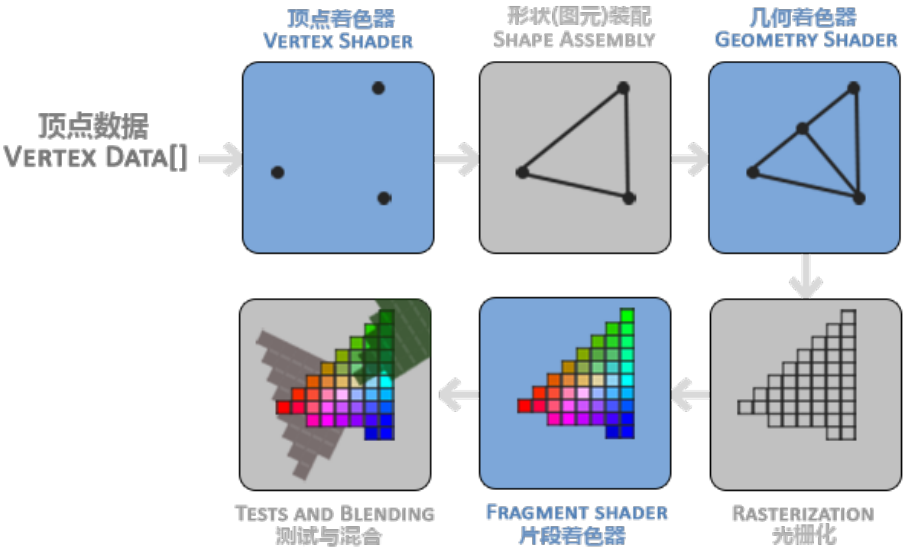


图 1: OpenGL Pipeline

图2展示了本项目实现的渲染程序的整个流程，包括一些初始化任务，模型和纹理的导入任务，以及最终的渲染。BVH 在模型导入之后进行构建，之后便传入片段着色器使用，更多内容将在第3部分进行具体介绍。

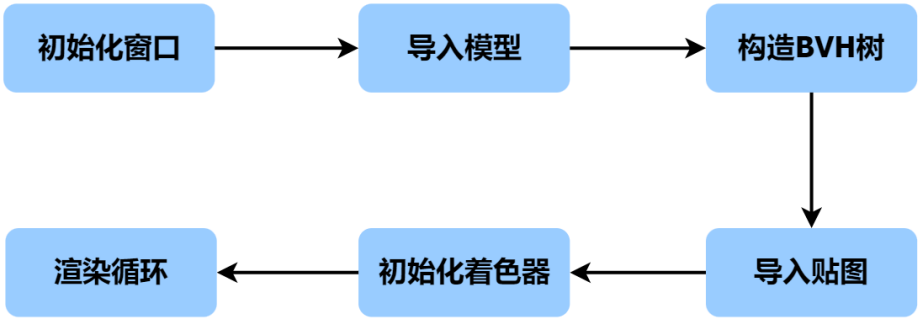


图 2: 流程图

2 相关工作/Related Works

第一个用于渲染的光线投射算法最初由 Arthur Appel 在 1968 年引入 [1]。光线投射通过从观察点对每一个像素发射一条光线并找到在世界场景中阻挡光线路径的最近物体来渲染场景，Ray Casting 只有两种射线，第一种是

眼睛发射的 eye 射线，来寻找场景中的交点，另一个是从交点发到灯光的阴影射线，看自身是否是处于阴影当中。与传统扫描线渲染算法相比，光线投射的一个显着优点是能够处理不平整的表面和固体。Turner Whitted 进一步发展了光线追踪算法 [2]，通过引入反射、折射和阴影来延长光线投射过程，他提出了递归光线追踪 (Whitted-Style Ray Tracing)：在任意一个接收到光线的点可以视作光源继续发射折射光线。这些由同一根光线（同一个像素点发射出来的光线）与物体产生的交点以及折射所产生的交点都与光源做连线并着色返回给像素点。本项目即据此实现了光线追踪算法，只不过在具体的实现中并不是使用的递归的方式，而是将其展开为了一个有限的循环，具体将在3中进行介绍。

在多边形网格模型的光线追踪算法实现中，层次包围盒 (BVH)[3] 被广泛应用于加快渲染速度。层次包围盒被构造为树形状结构。构成树叶节点的所有几何对象都包裹在包围盒中。然后将这些节点分组，并包含在较大的包围盒内。反过来，这些也以递归方式分组并包含在其他更大的包围盒中，最终构成在树的顶部具有单个包围盒的树结构。

在计算机图形学的发展过程中，为了实现抗锯齿的效果，出现了很多种方法。SSAA（Super Sample Anti-Aliasing 超级采样抗锯齿）通过增加每个像素点的采样次数来实现抗锯齿，效果很好，但性能消耗也极高。MSAA（MultiSampling Anti-Aliasing，多重采样抗锯齿）是一种特殊的 SSAA。MSAA 首先来自于 OpenGl，具体是 MSAA 只对 Z 缓存和模板缓存中的数据进行超级采样抗锯齿的处理，可以简单理解为只对多边形的边缘进行抗锯齿处理，对资源的消耗需求大大减弱。SMAA（Enhanced Subpixel Morphological Antialiasing，增强型子像素形态学抗锯齿）[5] 是后处理抗锯齿技术的一种，它的基本处理流程建立在 Jimenez 优化改造后的 MLAA（形态学抗锯齿）[4] 算法之上。MLAA 是由英特尔实验室提出的抗锯齿技术，这项技术代表着后处理式抗锯齿蓬勃发展的开端。SMAA 在此基础上进一步发展，进行了进一步的优化和扩展。

3 研究内容与方法 (或算法)/Contnts and Methods(or Algorithm)

3.1 OpenGL 渲染管线

本项目是在 OpenGL 上实现的，借助了 OpenGL 提供的图形渲染管线。OpenGL（Open Graphics Library）[9] 是一个跨语言、跨平台的应用程序编程接口（API），用于渲染 2D 和 3D 矢量图形。该 API 通常用于与图形处理单元 (GPU) 交互，以实现硬件加速渲染。OpenGL 的最新版本为 4.6，本项目使用的是 3.3 版本 [7]，所有 OpenGL 的更高的版本都是在 3.3 的基础上，引入了额外的功能，并没有改动核心架构。

图1已经展示了 OpenGL 渲染管线的各个阶段，下面结合本项目内容，分别对各阶段进行具体介绍：

顶点着色器 (Vertex Shader)：在 OpenGL 中，任何几何物体都是由点、线和三角形构成的，它们统称为图元。而图元又是由顶点构成的，所以顶点数据将被优先处理。顶点着色的输入是顶点数据，顶点数据是一系列顶点的集合，一个顶点是一个 3D 坐标数据的集合，通常用顶点数据表示，顶点属性一般包括 3D 位置和颜色，可能还有一些其他信息，如纹理坐标等。

顶点着色的过程就是输入一个单独的顶点，把 3D 坐标转换到另一种 3D 坐标 (物体坐标，世界坐标，屏幕坐标，视口坐标)，同时还对顶点属性进行一些基本处理。顶点着色器只负责处理顶点和与顶点有关的数据，包括顶点坐标的空间变换、顶点颜色处理和其他对于顶点的特殊处理。顶点着色器每从顶点数据流中读取一个顶点就输出一个顶点。

在本项目中，顶点着色器的实现非常简单，仅仅是将输入的顶点坐标及纹理坐标向下一阶段传递，顶点空间坐标的变换是在顶点着色器外完成的，这是因为本项目是采用第一人称视角实现的，可以利用鼠标和键盘变换观察方向，因此在顶点着色器外进行坐标变换会更加灵活。

图元装配 (Primitive Assembly)：前面介绍到，在 OpenGL 中，任何几何物体都是由点、线和三角形构成的，它们统称为图元。将顶点传入着色器后，这些顶点通过不同的连接方式会形成不同的显示效果，这时候就需要用到图元装配方式来确定最终显示出来的效果。图元装配的输入是顶点着色输出的所有顶点，它将其装配成指定图元的形状，并输出到下一个阶段。这一阶段由 OpenGL 自动完成。

几何着色器 (Geometry Shader)：几何着色器把图元形式的一系列顶点的集合作为输入，它可以通过产生新顶点构造出新的（或是其它的）图元来生成其他形状。与顶点着色器相比，几何着色器的操作对象是图元，它并且不会保证图元读入数量和读出数量相等，因为这一阶段可以抛弃或生成图元，改变图元种类。这个阶段不是必需的，本项目在实现过程中没有自定义几何着色器。

光栅化 (Rasterization Stage)：这里会把图元映射为最终屏幕上相应的像素，生成供片段着色器使用的二维片段，这些片段代表着可在屏幕上绘制的像素。将图元转换为片段是通过线性插值来完成的，光栅化根据图元上的顶点坐标，插值出片段，相当于图形区域的像素。需要注意的是，此时的像素并不是屏幕上的像素，是不带有颜色的，接下来的片段着色器完成上色的工作。

在光栅化之后会执行裁切，裁切会丢弃超出视图以外的所有像素，用来提升执行效率。光栅化阶段也是由 OpenGL 自动完成的。

片段着色器 (Fragment Shader)：片段着色器的主要目的是计算一个像素的最终颜色，这也是所有 OpenGL 高级效果产生的地方。通常，片段着色器包含 3D 场景的数据（比如光照、阴影、光的颜色等等），这些数据可以被用来计算最终像素的颜色。

一个图像就是通过每个像素的颜色组合产生的，因此片段着色器很大程度上决定了最终图像的呈现效果。通过在片段着色器上对光照进行一系列处理，可以得到阴影、反射、折射等各种不同的接近真实世界的光照特征。本项目在片段着色器中实现了 BVH 和光线追踪算法，以及各种纹理特征等，呈现出了一个精彩的视觉效果。

测试与混合 (Test and Blending)：在所有对应颜色值确定以后，最终的对象将会被传到这最后一个阶段。这个阶段检测片段的对应的深度（和模板）值，用它们来判断这个像素是其它物体的前面还是后面，决定是否应该丢弃。这个阶段也会检查 alpha 值（alpha 值定义了一个物体的透明度）并对物体进行混合。所以，即使在片段着色器中计算出来了一个像素输出的颜色，在渲染多个三角形的时候最后的像素颜色也可能完全不同。这一阶段也是由 OpenGL 自动完成的，不需要手动实现整个过程。

3.2 光线追踪算法

本项目的光线追踪算法是在 OpenGL 的片段着色器中实现的，因此是一种逐像素的光线追踪。逐像素的光线追踪指的是，从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，如图3所示 [10]。

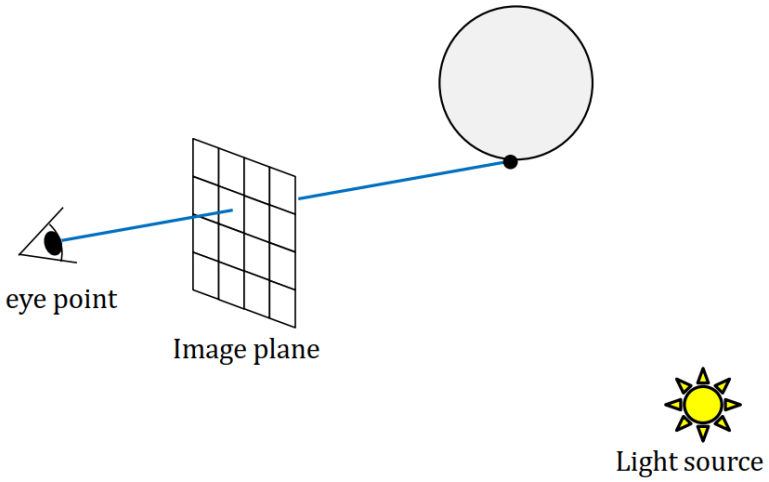


图 3: 逐像素光线追踪

下面对光线追踪算法进行一个整体的介绍：对于从视点投射到一个像素上的射线，检查场景中的每个对象，看射线是否与其中的任何一个相交。当发生射线与多个对象相交的情况时，选择交点离视点最近的物体。然后从相交点向光源投射阴影射线，如果这条特定的光线在通往光源的路上不与某个物体相交，那么这个点就被照亮了。此外，如果该交点具有反射或折射属性的话，根据反射或折射公式计算出反射或折射后的射线，对该射线进行相同的处理。根据光线传播路径，通过以不同比例混合所有交点的颜色，就得到了像素的颜色。

以上描述的是一个递归过程，由于本项目的光线追踪算法是在片段着色器中实现的，片段着色器使用的是 GLSL 语言，不支持递归调用，因此本项目将递归过程转化成了一个循环过程。对于一条射线，若其遇到光源或者没有遇到任何物体，则提前结束循环；若遇到的材质没有折射或反射属性，只需计算因漫反射和环境光产生的颜色即可，然后也提前结束循环；当遇到的材质有反射或者折射属性中的一种时，利用公式计算出反射或折射的方向，计算交点的颜色并继续循环；比较麻烦的是若遇到的材质既有反射又有折射，在具体的实现中进行了部分

妥协，只计算一次反射，光线的方向依据折射确定，继续下面的循环过程。该算法的伪代码如1所示。

Algorithm 1 Ray Tracing

```
function RayTracing(ray, max_depth)
    for depth from 1 to max_depth do
        object = CalculateIntersect(ray)
        if object = background then
            color += environment*mask(background)
            break
        end if
        if object = light then
            color += light*mask(light)
            break
        end if
        if object = ObjectWithoutReflectAndRefract then
            color += diffuse*mask(object)
            break
        end if
        if object = ObjectWithReflectOrRefract then
            color += diffuse*mask(object)
            ray=ReflectOrRefract(ray)
        end if
        if object = ObjectWithReflectAndRefract then
            color += diffuse*mask(object)+getReflectedColor(ray)
            ray=Refract(ray)
        end if
    end for
    return color
end function
```

为了实现反射和折射效果，需要计算发射和折射的方向。反射的方向计算可以简单的使用反射定律进行计算，课上已经进行了详细介绍，这里不再赘述。折射使用的是斯涅尔定律，如公式1所示，其中 η、η' 分别为入射方向和折射方向的折射率，θ、θ' 分别为入射角和折射角。

$$\eta \cdot \sin\theta = \eta' \cdot \sin\theta' \tag{1}$$

对于入射光线 R，可以分别计算出反射光线 R' 垂直于折射面的分量 $\mathbf{R}'_{\perp}$ 和平行于折射面的分量 $\mathbf{R}'_{\parallel}$，如公式2、3所示。

$$\mathbf{R}'_{\perp} = \frac{\eta}{\eta'}(\mathbf{R} + \cos\theta\mathbf{n}) \tag{2}$$

$$\mathbf{R}'_{\parallel} = -\sqrt{1 - |\mathbf{R}'_{\perp}|^2}\mathbf{n} \tag{3}$$

对于垂直分量，可以消去 $\cos\theta$，如公式4所示。

$$\mathbf{R}'_{\perp} = \frac{\eta}{\eta'}(\mathbf{R} + (-\mathbf{R}\cdot\mathbf{n})\mathbf{n}) \tag{4}$$

对于反射光线的光照强度计算，为了追求更真实的效果，本项目实现了近似的菲涅尔反射。菲涅耳反射加入了反射/折射与视点角度之间的关系，引入菲涅尔反射率控制发射光线的强度。菲涅尔反射源自于菲涅尔效应，举例来说，如果你站在湖边，低头看脚下的水，你会发现水是透明的，反射不是特别强烈；如果你看远处的湖面，你会发现水并不是透明的，但反射非常强烈，这就是菲涅尔效应。简单的讲，就是视线垂直于表面时，反射较弱，而当视线非垂直表面时，夹角越小，反射越明显。精确的菲涅尔反射率计算是非常复杂的，本项目使用了 Schlick 提出的菲涅耳反射率的近似，如公式5所示，其中 n 是法线方向，l 是入射方向，F 是菲涅尔反射率，F_0 是 0 度时的菲尼尔反射率，可以通过公式6进行简单计算。

$$F \approx F_0 + (1 - F_0)\left(1 - (n\cdot l)^+\right)^5 \tag{5}$$

$$F_0 = \left(\frac{n-1}{n+1}\right)^2 \tag{6}$$

在光线追踪算法中，需要计算射线与各种几何图形的求交，这部分内容课上已经介绍的非常详尽了，这里就不再赘述。

3.3 层次包围盒 (BVH)

光线与物体相交，是光线追踪的主要时间瓶颈，时间与物体数量成线性关系。但它是对同一模型的重复搜索，所以可以用二分搜索的方式把它变成对数搜索。对于同一个模型，发射到上面的射线有多条，因此可以对模型进行类似的排序，然后每条射线的交点都可以在对数时间内完成搜索。

BVH 的核心思想就是用体积略大而几何特征简单的包围盒来近似描述复杂的几何对象，并且这种包围盒是嵌套的，只需要对包围盒进行进一步的相交测试，就可以越来越逼近实际对象。树形结构很好的满足了 BVH 所要实现的特征，在一个 BVH 树中，非叶节点是一个个包围盒，包含了许多图元，其子节点是包含一部分图元的包围盒；叶子节点则是最小的包围盒，有若干个图元，若射线与该包围盒相交，则需要遍历所有图元求交。对于 BVH 树一个直观的描述如图4所示 [10]。

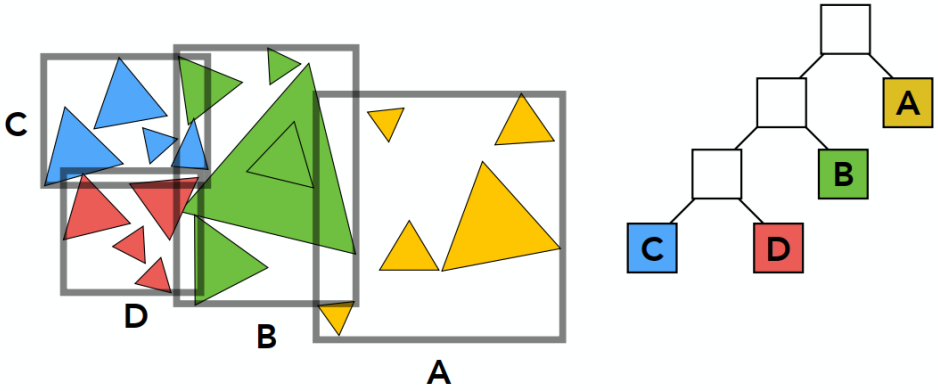


图 4: BVH 树

对于包围盒的选择可以有很多种，为了实现最快速的射线与包围盒求交，本项目选用的是最简单的轴对齐包围盒 (AABB)。轴对齐包围盒是一个简单的 3D 的六面体，每一边都平行于一个坐标平面，矩形边界框不一定是正方体，它的长、宽、高可以彼此不同。计算射线与 AABB 盒求交，一般使用的是“平板法”，即将每个面看作一个平板，对于一对相对的面，计算射线通过两个面的先后时间，若通过所有近面的最大时间小于通过所有远面的最小时间，则说明射线与包围盒相交。具体的计算过程课上已经讲的非常详尽了，这里不再赘述。

对于具体的实现，本项目在导入模型的时候构造 BVH 树，在片段着色器中遍历 BVH 树求交。为了方便将数据在片段着色器内外进行传输，本项目将树形结构以数组的形式表示，构造的是一个完全二叉树，BVH 树的叶子节点包含的图元数量可以自行指定，第4部分将对不同图元数量的性能差异进行探究。此外，由于 BVH 树占用的内存较大，难以直接将其传送到片段着色器中，本项目巧妙的通过纹理数据传输的方式把 BVH 树传送给片段着色器，完美解决了这个问题。这里给出片段着色器中 BVH 树的遍历算法，如2所示。

3.4 抗锯齿：SMAA

SMAA 是一种用于改善图形渲染质量的抗锯齿算法。它利用视网膜对图像的特殊敏感性，在不增加渲染时间的前提下，提供了比传统抗锯齿算法更高的视觉效果。SMAA 算法的基本原理是，通过在图像的每个像素周围使用特殊的形态学滤波器，来检测像素周围的颜色过渡。这些过渡被称为边缘，并且通常具有较大的对比度。检测到边缘后，SMAAR 算法会使用边缘平滑技术来消除边缘的锯齿效应，从而提高图像的视觉效果。下面对 SMAA 原理进行详细介绍：

Algorithm 2 Intersection Determination with BVH

```
function HITBVH(ray, root) distance = INF
    stack.pushback(root)
    while stack is not empty do
    node=stack.pop()
    if node is a leaf node and Intersect(ray, node) then
        distance = min(distance, CalculateDistance(ray,node))
    end if
    if node is a middle node then
        if Interrect(ray, node.leftchild) then
            stack.pushback(node.leftchild)
        end if
        if Interrect(ray, node.rightchild) then
            stack.pushback(node.rightchild)
        end if
    end if
    endwhile
    return distance
end function
```

SMAA 的核心原理来源于 MLAA。MLAA 的基本思路是：检测每帧图像上的边缘（通常可对亮度、颜色、深度或者法线进行边缘检测），然后对这些边缘进行模式识别，归类出 Z、U、L 三种形状，根据形状对边缘进行重新矢量化，并对边缘上的像素根据覆盖面积计算混合权重，将其与周围的颜色进行混合，从而达到平滑锯齿的目的。

在 MLAA 中，需要解决一下几个问题：

1. 每个点需要计算四个方向的锯齿边界信息；
2. 沿着锯齿边界两侧，搜索边界结束位置时，非常慢；
3. 重矢量化需要消耗大量带宽，混合系数计算很复杂；

其对应的解决方案分别为：

1. 像素边缘/锯齿边界是共享的，每个像素其实只需要计算左侧和上侧的边界；
2. 借助双线性采样加速搜索；
3. 使用预计算好的贴图，避免实时计算；

SMAA 是在 MLAA 之上改进而来的，这里主要关注 SMAA 的改进点：

SMAA 在常规边缘检测的基础上加入了对局部对比度的考量，如图5所示 [5]，灰点为当前像素，c 为像素颜色，以检测左边缘为例，C_l 必须同时满足：(1) 大于一定阈值；(2) 大于 C_l 、C_r 、C_t 、C_b 、C_{2l} 强度最大值的一半这两个条件时，才会被判定为边缘。这种考虑了局部对比度的检测可以有效避免误检。

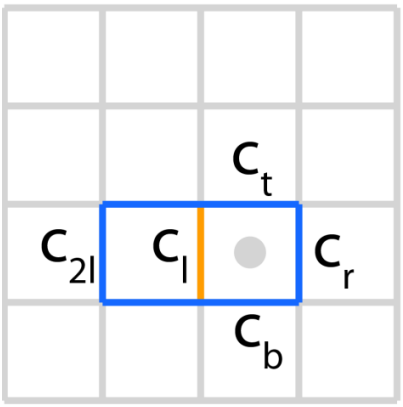


图 5: SMAA 边缘检测

在 MLAA 中，主要识别的是水平边缘和垂直边缘，SMAA 加入了对角线边缘的识别，如图6所示 [5]。在计算覆盖率时，使用的是和 MLAA 相同的处理方法，直接采用预计算的贴图，将预计算好的覆盖率/混合系数保存到贴图中，以加速计算。

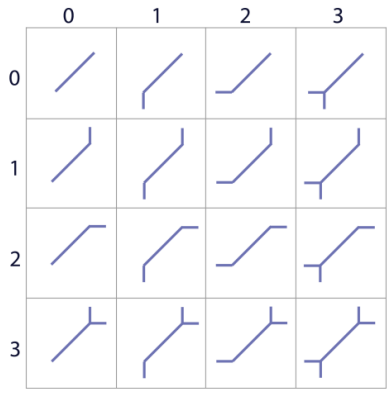


图 6: SMAA 对角线边缘识别

此外，SMAA 还引入了更精确的边缘搜索方法，既提高了搜索边界的速度，还提高了搜索边界的质量。最终实现的效果是，SMAA 对锯齿的处理非常精细，得到的效果也非常好，可以说是基于后处理方法处理抗锯齿的极限。如果要得到更好的抗锯齿效果，还可以和其他的抗锯齿方式进行结合。

对于 SMAA 的具体实现，本项目直接使用了 SMAA 的官方实现 [6]，将其作为一个可选项嵌入到本项目的图形渲染中，其效果将在第4部分进行展示。

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

4.1　最终效果

图7展示了最终的实时光线追踪的渲染效果，实现了阴影、反射、折射等视觉效果，还加入了典型的交大场景以及交大校徽，均在图中标注了出来。图中有多个几何体，如正方体、球体、圆柱、圆锥、圆环等，还导入了一个网格模型，画面内容十分丰富。

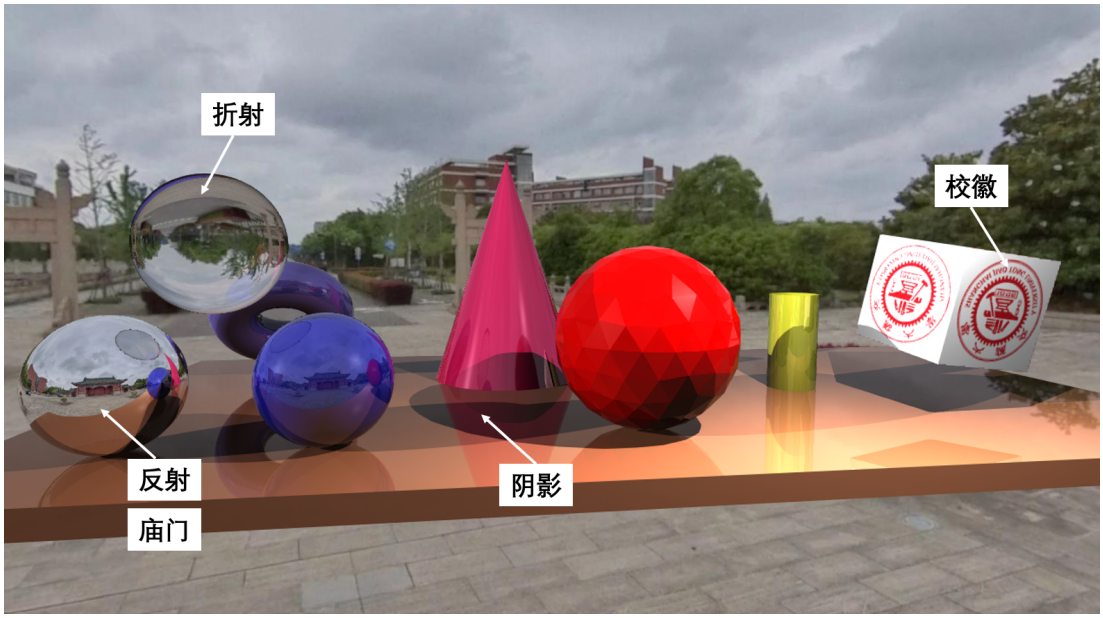


图 7: 渲染结果

4.2 网格模型导入效果

与几何模型相比，网格模型由数百个三角形网格组成（就本项目而言），在计算光线与物体求交的时候，计算量增大了数倍，因此帧率会有明显的下降，导入网格模型前后的帧率对比如图8所示。

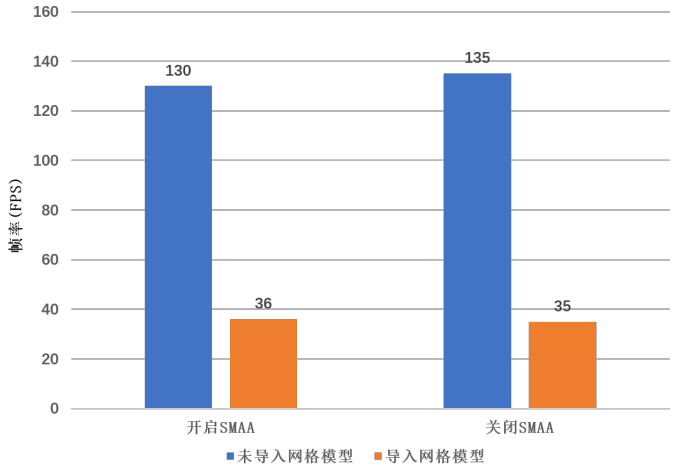


图 8: 导入网格模型前后及开启/关闭 SMAA 前后帧率对比

4.3 递归深度

由3.2可知，光线追踪算法最主要的是递归的过程（本项目将其展开为了循环），递归的深度对渲染速度有很大的影响，图9展示了不同递归深度的帧率情况。可见，随着递归深度的增加，帧率在不断下降。然而，当递归深度太小，会出现如图11所示的情况，整体亮度较低，且出现很多暗影，因此，在实时渲染过程中要做好渲染效果和效率之间的平衡。

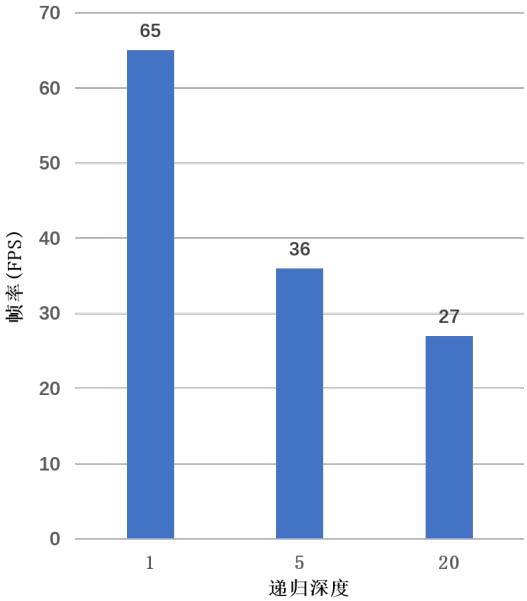


图 9: 不同递归深度帧率对比

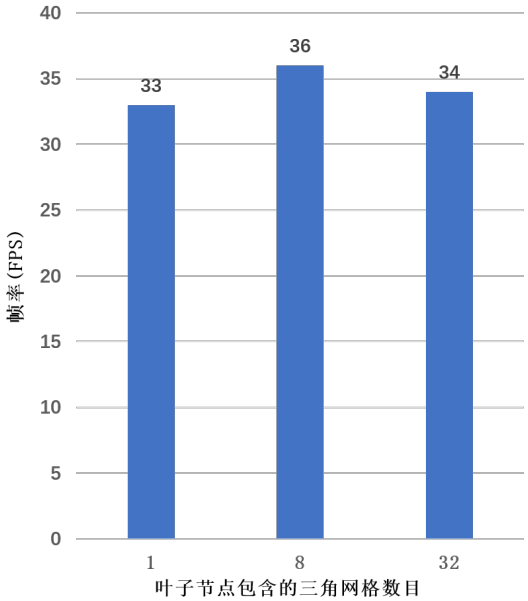


图 10: 叶子节点不同三角网格数目帧率对比

4.4 BVH 树叶子节点

在 BVH 树中，仅叶子节点中包含三角形网格，因此，每个叶子节点包含的三角形网格数量对渲染速度会产生一定影响。如图]10所示，叶子节点包含的网格数量过少时，BVH 就失去了其存在的意义；但包含的网格过多时，也会导致射线与大量三角形计算求交，使得渲染速度下降。因此，在构建 BVH 树时，对于叶子节点要选择适当的三角形网格数。

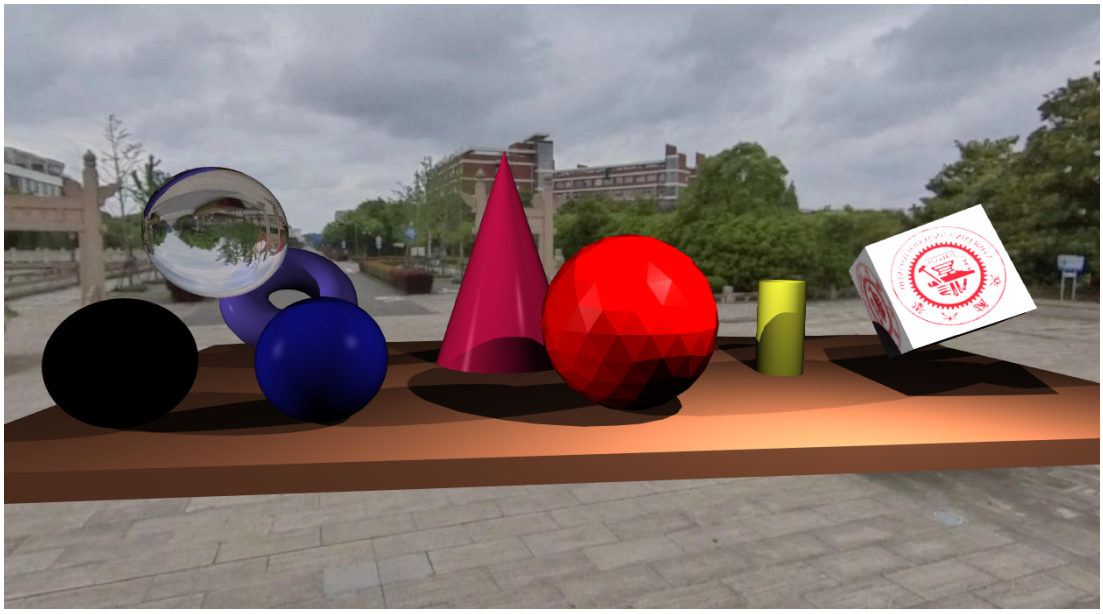
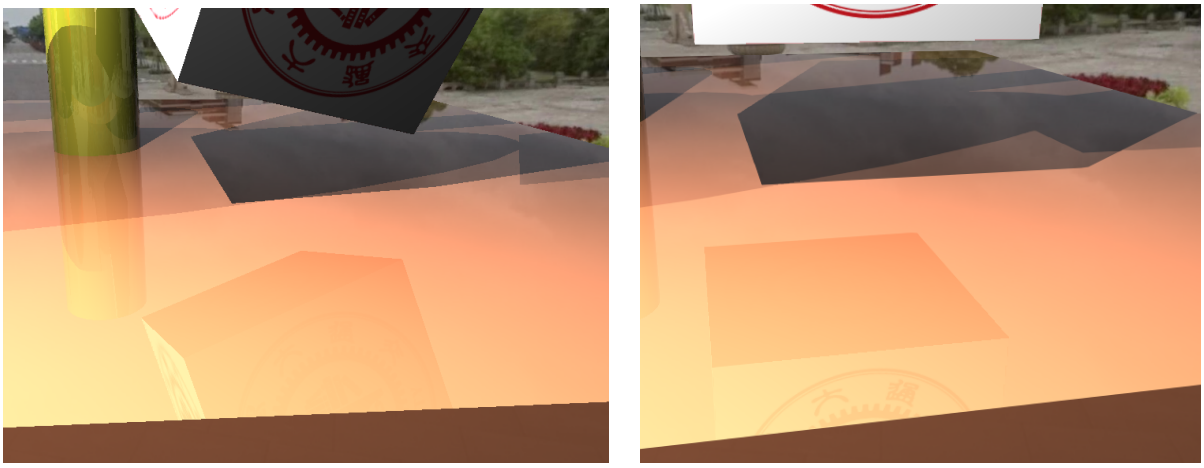


图 11: 递归深度为 1 时的渲染结果

4.5 SMAA 抗锯齿效果

SMAA 是一种对性能损失较小的抗锯齿算法，如图8所示，开启 SMAA 并未对帧率造成特别大的影响，但其效果却十分显著，如图12所示，开启 SMAA 后，锯齿现象得到了明显改善。



(关闭 SMAA)　　(开启 SMAA)

图 12: SMAA 效果对比

5 特色与创新/ Distinctive or Innovation Points

- 本项目将交大特色场景充分融合了进去，使用世界贴图，还原了交大庙门附近的场景，并用该场景展现了反射、折射等效果。此外，还通过纹理贴图将校徽元素融入到了本项目中。
- 本项目在实时渲染中实现了光线追踪算法，在实现阴影、反射、折射等光学效果的情况下依然实现了较高的帧率。在光线追踪算法中还融入了菲涅尔反射，以实现更真实的视觉效果。
- 本项目实现了 BVH 树结构，将其融入到了光线追踪算法中，加快了渲染速度，在几何模型之外也支持了网格模型的渲染。
- 本项目引入了 SMAA 算法实现抗锯齿，在兼顾性能的情况下取得了优秀的抗锯齿效果。

6 补充说明

- 光线追踪（Ray Tracing）：光线追踪是一种用于在图形渲染中模拟光照的技术。它通过模拟光线在真实三维环境中的传播，来生成二维图像中的光照。
- 实时光线追踪（Real-Time Ray Tracing）：将光线追踪应用在实时渲染场景中，即称为实时光线追踪。
- 开放图形库（OpenGL）：开放图形库是用于渲染 2D、3D 矢量图形的跨语言、跨平台的应用程序编程接口。
- 层次包围盒（Bounding Volume Hierarchy）：层次包围盒是一组几何对象上的树结构。。构成树叶节点的所有几何对象都包裹在包围盒中。然后将这些节点分组，并包含在较大的包围盒内。反过来，这些也以递归方式分组并包含在其他更大的包围盒中，最终构成在树的顶部具有单个包围盒的树结构.
- 抗锯齿（Anti-Aliasing）：抗锯齿是一种消除显示器输出的画面中图物边缘出现凹凸锯齿的技术。
- 增强型子像素形态学抗锯齿（Enhanced Subpixel Morphological Antialiasing）：SMAA 是一种用于改善图形渲染质量的抗锯齿算法。它利用视网膜对图像的特殊敏感性，在不增加渲染时间的前提下，提供了比传统抗锯齿算法更好的视觉效果。

References

[1] Arthur Appel. “Some techniques for shading machine renderings of solids”. In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. 1968, pp. 37–45.

[2] Turner Whitted. “An improved illumination model for shaded display”. In: *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*. 1979, p. 14.

[3] Christer Ericson. *Real-time collision detection*. Crc Press, 2004.

[4] Alexandre De Pereyra. “MLAA: Efficiently Moving Antialiasing from the GPU to the CPU”. In: *Intel Labs* (2011).

[5] J. Jimenez et al. “SMAA: Enhanced Subpixel Morphological Antialiasing”. In: *Computer Graphics Forum* 31.2pt1 (2012), pp. 355–364.

[6] Jorge Jimenez. *SMAA*. [Online]. https://github.com/iryoku/smaa.

[7] OpenGL. *The OpenGL Graphics System:A Specification(Version 3.3 (Core Profile)- March 11, 2010)*. [Online]. https://registry.khronos.org/OpenGL/specs/gl/glspec33.core.pdf.

[8] Joey de Vries. *Learn OpenGL*. [Online]. https://learnopengl.com/.

[9] Wikipedia. *OpenGL*. [Online]. https://en.wikipedia.org/wiki/OpenGL.

[10] Lingqi Yan. *GAMES101*. [Online]. https://sites.cs.ucsb.edu/~lingqi/teaching/games101.html.