

学号： 2013220202006

姓名： 官云锐

专业方向： 软件技术

企业名称： 巨人网络

毕设课题名称：光线跟踪渲染器的设计与实现

企业指导教师： 汪军生

院内代管教师： 刘峤老师

**信息与软件工程学院**

**毕业设计（顶岗实习）**

**初期报告**

目 录

[第1章 选题背景（需求） 4](#_Toc482441912)

[1.1 综述 4](#_Toc482441913)

[1.2 渲染技术 4](#_Toc482441914)

[1.3 国内外研究情况 5](#_Toc482441915)

[1.4 小结 5](#_Toc482441916)

[第2章 相关技术与理论概述（技术） 7](#_Toc482441917)

[2.1 渲染概述 7](#_Toc482441918)

[2.2 GPGPU 7](#_Toc482441919)

[2.3 计算机图形学概述 8](#_Toc482441920)

[2.4 BVH算法 8](#_Toc482441921)

[第3章 概要设计 10](#_Toc482441922)

[3.1 场景描述 10](#_Toc482441923)

[3.2 光线跟踪原理 10](#_Toc482441924)

[3.2.1 光线原理 10](#_Toc482441925)

[3.2.2 光线发射与光线跟踪 10](#_Toc482441926)

[3.3 光照模型 11](#_Toc482441927)

[3.3.1 Phong模型 11](#_Toc482441928)

[3.3.2 阴影 12](#_Toc482441929)

[3.3.3 反射与折射 12](#_Toc482441930)

[3.3.4 摄像机 12](#_Toc482441931)

[3.3.5 相交测试 14](#_Toc482441932)

[3.3.6 平面 15](#_Toc482441933)

[3.3.7 包围盒 15](#_Toc482441934)

[3.3.8 三角形 16](#_Toc482441935)

[3.4 算法优化 16](#_Toc482441936)

[3.4.1 场景分割思想 16](#_Toc482441937)

[3.4.2 BVH算法 17](#_Toc482441938)

[第4章 详细设计 18](#_Toc482441939)

[4.1 渲染系统设计 18](#_Toc482441940)

[4.2 加速系统设计 20](#_Toc482441941)

[4.2.1 构建包围盒 20](#_Toc482441942)

[4.2.2 构造BVH 21](#_Toc482441943)

[第5章 编程实现 25](#_Toc482441944)

[5.1 主要实现技术 25](#_Toc482441945)

[5.2 系统框架 25](#_Toc482441946)

[5.3 主要功能模块的实现 25](#_Toc482441947)

[第6章 功能与性能测试 26](#_Toc482441948)

[第7章 结束语 27](#_Toc482441949)

1. 选题背景（需求）
   1. 综述

随着计算机硬件的发展，其对于图像的处理能力也越来越强大。计算机图形在我们的生活中也有了越来越多的应用。小到电影特技，游戏开发；大到灾难模拟，地理信息遥感，虚拟现实；这些产品中都有着计算机图形学的重要应用。人们对于计算机所绘制的图像的真实性，实时性上，要求也越来越高。如何快速，真实的渲染图像，是计算机图形学研究者们努力的目标。

随着人们对于虚拟现实画面的要求的不断提高，特别是游戏领域，传统的光栅化成像越来越捉襟见肘。人们即需要高质量的画质，又需要尽可能短的渲染时间，所以现在光线跟踪渲染的方式越来越受到大家重视。

* 1. 渲染技术

在现代的计算机图形学中，主要有两种渲染方式，分别为光栅化和光线跟踪。

光栅化渲染是表示在计算机上显示三维形状的流行渲染算法。栅格化目前是生成实时三维计算机图形最流行的算法。此方法采用了流水线的设计思想，将三角形图元经过三维变化，投影变换，确定那些像素在顶点表示的图元内部，生成对应的坐标值，以及图元顶点的其他信息比如法线，颜色深度等等。然后根据深度信息确定遮挡关系。这种方法的好处是并行计算能力强，独立性高便于优化，能够立即响应用户输入，并且通常有每秒 24 帧的速率。与辐射着色、光线跟踪等其它渲染技术不同，栅格化的速度非常快，但是由于它不是根据光传输的物理规律进行处理的，所以无法正确模拟许多复杂真实光照环境。

比起光栅化的渲染方式，光线跟踪算法背后有着数学和物理模型的理论支持。其基本思想是根据光线传播的原理，模拟光线在实际场景中的传播过程：从光源发出的光线经过折射反射最终会到达我们眼睛，在计算机图形学中则是到达我们的屏幕；同时光路具有可逆性，所以我们可以从屏幕，追踪射出的光线，直到达到收敛条件为止。这种按照绝对物理规律计算出来的图形具有很高的真实性能，但是在计算的复杂度上，均要求很高。

光栅化渲染的方式因为其计算的速度快，常用于计算机游戏软件；而光线跟踪的渲染方式因为其渲染图像的真实常常用于电影特效，CG动画等离线渲染的领域。不过随着时代的发展，人们对于画面的要求越来越高，为了满足人们的需求，光栅化渲染的方式为了追求更加真实的渲染效果，让设计变得越来越复杂，学习和使用成本也更加的高昂。

而光线跟踪算法其优势在于，按照真实物理规律构建渲染场景，随着硬件发展的今天，将光线跟踪的思想部分甚至全部应用于实时渲染，能够极大的提升渲染画质，也是人们研究的一个重点。

在光线跟踪中，主要的性能消耗在于求解光线与几何面片的相交问题上，优化几何面片的数据组织减少每次求交过程当中的冗余计算，能够极大的提升渲染速度。这篇论文中，将使用光线跟踪的方式设计并实现一个光线跟踪渲染器，并且引入多种优化方式，与传统渲染方式进行比较。

* 1. 国内外研究情况

在国外，很多年以前就开始了对于光线跟踪算法进行详细而深入的研究，发表的图形学论文也不计其数，有许多的优化算法诞生。根据其特点，光线跟踪算法主要优化策略在场景组织上。在光线跟踪主要应用于离线渲染的时期，提出了KD-Tree这种针对静态空间的加速数据结构。随着对于实时光线跟踪算法的研究又诞生看了基于层次的结构的BVH，基于空间划分的BSP、八叉树等空间划分结构。

在数据优化方面，Hunt和Shevtsov等人提出了采用桶近似方法以达到快速计算SAH启发函数的目的。Ingo等人，通过场景面片进行分类，利用分而治之的思想实现了快速构建kd-tree的方法。SaarLand大学更是拿出了业界首个硬件加速光线追踪算法方案，这个被称作SaarCOR的光线追踪加速卡，采用FPGA(Field Programmable Gate Array现场可编程门阵列)堆砌，在66MHz的频率下就能获得和Pentium 4 2.5GHz相当的光线追踪性能。

在交互场景方面，Brigade开源引擎采用实时光线跟踪作为基础，在传统的光栅化基础上，引入光线追踪的的方式，表现力出色，其上有Arauna等游戏应用。Jonas Wagner则采用高效的Js引擎实现了实时光线跟踪的demo；Nvidia也有开发使用基于Nvidia GPU的高效光线追踪的引擎框架Optix。

* 1. 小结

光线跟踪的特点在于其思路简单的并且遵从物理规律的算设计，以及可以并行计算的过程；在光线跟踪中，每一个像素点的计算都是独立进行的，其主要的工作量在于光线和场景中多边形相交的判断。

本文主要是针对实现一个光线跟踪渲染方式的渲染器，并且尝试结合当今业界流行的两种加速方式，通过场景分割的方式对于复杂场景进行分割，减少光线和场景中物体求交的计算。并且对于单个物体，通过分治的思想，减少单个物体求交运算的时间。

1. 相关技术与理论概述（技术）
   1. 渲染概述

渲染（render）在电脑绘图中，是指：用软件从模型生成图像的过程。模型是用语言或者数据结构进行严格定义的三维物体或虚拟场景的描述，它包括几何、视点、纹理、照明和阴影等信息。

渲染器是3D引擎的核心部分，它完成将3D物体绘制到屏幕上的任务。是具备根据发送过来的三维图形顶点信息，进行顶点变换，正交投影，纹理映射等过程绘制图片信息，并且添加反射，光照等额外信息的软件或硬件工具。并且根据运行的模式，常常分为软件渲染器和硬件渲染器：

硬件渲染器是将上述流程硬件化实现加速的渲染工具，我们称之为图形卡或者显卡，硬件渲染器通常基于底层图形API(应用程序接口)构建，采用适合硬件架构的光栅化方法进行渲染。图形API负责与硬件的通信，常用的图形API包括DirectX和OpenGL。根据硬件加速结构设计一整套的渲染管线，提供可编程借口。

软件渲染器则纯粹利用CPU的能力进行计算，通常采用光线追踪的方法进行渲染。软件渲染器没有统一的应用程序标准，但是有很多通用的渲染算法，如光子映射，蒙特卡洛，辐射度等等。

* 1. GPGPU

GPGPU全称General Purpose GPU，即通用计算图形处理器。其中第一个“GP”通用目的（GeneralPurpose）而第二个“GP”则表示图形处理（GraphicProcess），这两个“GP”搭配起来即“通用图形处理”。再加上“U”(Unit)就成为了完整的通用处理器。 人们一直在寻找各种加速图像处理的方法，然而受到CPU本身在浮点计算能力上的限制，对于那些需要高密度计算的图像处理操作，过去传统的在CPU上实现的方法，并没有在处理性能与效率上有很大进步。随着可编程图形处理器单元(GPU)在性能上的飞速发展，利用GPU加速图像处理的技术逐渐成为研究热点。

CPU和GPU都是具有运算能力 的芯片，CPU负责指令运算与数值运算，而GPU以并行的数值计算为核心，通常针对特定有的算法流水线进行设计。例如现在的图形显卡，基于的光栅化渲染算法的流程，允许用户对特定的流程进行编程，这部分可以定制的部分，则体现为Directx以及OpenGL的图形API。在这样的通用化之外，各个GPU又拥有某种特点算法流程的硬件优化设计，例如AMD HD7000系列显卡加入了独特的结构用于对头发的发丝渲染。NIVIDA 则有针流体结构的加速结构。不过由于光线跟踪算法，现在尚无一套统一的流程化算法，所以尚无针对光线跟踪技术统一标准的加速显卡。在算法的研究完善后，光线跟踪技术将越来越多的使用在计算机图形学中。

* 1. 计算机图形学概述

计算机图形学是研究计算机在硬件和软件的帮助下创建计算机图形的科学学科，是计算机科学的一个分支领域，主要关注数字合成与操作视觉的图形内容。简单地说，计算机图形学的主要研究内容就是研究如何在计算机中表示图形、以及利用计算机进行图形的计算、处理和显示的相关原理与算法。图形通常由点、线、面、体等几何元素和灰度、色彩、线型、线宽等非几何属性组成。从处理技术上来看，图形主要分为两类，一类是基于线条信息表示的，如工程图、等高线地图、曲面的线框图等，另一类是明暗图，也就是通常所说的真实感图形。

计算机图形学一个主要的目的就是要利用计算机产生令人赏心悦目的真实感图形。为此，必须创建图形所描述的场景的几何表示，再用某种光照模型，计算在假想的光源、纹理、材质属性下的光照明效果。所以计算机图形学与另一门学科计算机辅助几何设计有着密切的关系。事实上，图形学也把可以表示几何场景的曲线曲面造型技术和实体造型技术作为其主要的研究内容。

本文中主要使用了计算机图形学中，使用解析几何的方式对现实空间进行建模，并且使用程序进行模拟的这部分技术。

* 1. BVH算法

光线跟踪优化策略中，一种大家公认的设计，是进行空间划分。通俗的来说，是对抽象的三维空间通过某种特定的算法，进行连续的划分。BVH即Bounding Volume Hierarchy,是一种树形的几何体组织结构。其特点为，所有的几何物体都包含于包围盒体积中，作为数据结构上树的叶子节点。然后这些几何体根据位置信息组成小组，用更大的包围盒包裹他们。

BVH的算法处理过程可以描述为,根据几何体构建包围盒；跟据集合体位置关系构建BVH树结构；根据摄像机朝向，或者某一个方向，如X轴调节叶子节点的优先级；通过包围盒筛选需要进行光线跟踪的复杂物体；将最终结果回传到程序。这个过程中，主要解决三个问题，分别为：1.物体包围盒的确定；2.将物体分配到BVH树中，并且经可能保证树的平衡度；3.确定光线和那个叶子节点相交。

BVH算法是把场景进行连续的划分的发，在场景中建立八叉树的方式。BVH树的每一个节点表示其子空间及其包围的场景。在加载场景的时候，为每个几何物体构建包围盒，根据场景中所有的几何物体计算场景最大的包围盒体积大小。根据根节点包围盒的中心节点的大小，将场景划分为八个区域，每一个区域形成了一个小的包围盒，若这个子区域中还有物体，则重复上述工作。那么对于光线求交的时候，从根节点开始，对不为空的子节点进行求交，当找到交点的时候，对这个子节点包含的包围盒进行求交，直到找到一个唯一的交点。

在实际构建BVH树的过程中，通常会额外做两个操作，第一是增加包围盒的复杂度，从三个面到七个面，第二回根据一个特定的方向对于子节点进行排序，用来减少不必要的相交计算。

1. 概要设计
   1. 场景描述

对于渲染算法而言，场景很重要。场景是一个包含将要渲染的各种物体的数据集合。场景可以很简单也可以很复杂，小到一个简单的集合体，大到一个城市，都可以是一个场景。对于计算机图形学而言，场景最细的粒度是顶点，即使用坐标（X,Y,Z）描述。顶点通过组成多边形,通常是三个点组成三角形的形式来描述复杂物体。

* 1. 光线跟踪原理
     1. 光线原理

光是能量传播的一种方式，常见的光线从光源发出，并进行传播，直至遇到能够阻挡其传播的介质。光线在空间中沿直线传播，并且不需要其他介质作为传播媒介，在传播过程中会产生如下的物理现象：

反射现象，光在遇到不同介质时候会在介质表面产生入射光线返回原有介质的情形，其运动遵循如下三条定律：

反射线、入射线与法线在同一平面上。

反射线与入射线在法线的两侧。

反射角等于入射角

折射现象，光在不同密度的介质之间穿过时，会产生光路的偏转现象，不同的介质可以体现出不同的折射角，其计算由折射率公式决定：

漫反射现象，由于物体的表面不是绝对的平整，所以上述两个过程在实际的材质上发生的时候，会在物品不同倾斜度的微表面进行，可以建模表示为照到物体表面的光线沿着四面八方产生漫反射。

* + 1. 光线发射与光线跟踪

光线发射算法最早由Arthur Appel提出，其主要思想为，从视点向屏幕的每个像素发射光线，照到光线传播路径上，照到最近的相交点，根据该点的物理性质，计算颜色的数值。如图3-1所示：

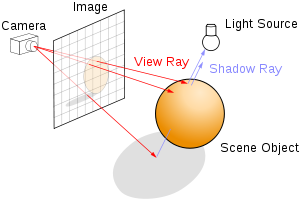


图 3‑1 光线跟踪原理

Turner Whitted优化了光线发射算法提出了光线跟踪算法，原有的光线发射算法在光线与第一最近的物体相交之后就结束了计算。Whitted 提出的算法，在光线光线与物体碰撞之后，会产生三种新的光线：折射光线，反射光线，阴影光线。其中反射光线和折射光线颜色之小于等于1（考虑到材质吸收）。弱平面为光滑平面，反射光线沿着镜面反射方向进行传播，最先相交的物体便是漫反射中看到的物体。折射光线在经过向量的变换之后，进行如下的计算，同样最先击中的物体就是半透明物体透过的物体。阴影光线用于检测当前点对于光源来说是否可见如果有不透明物体对光线遮挡，则后面的计算都不用进行跟踪计算。

* 1. 光照模型
     1. Phong模型

如上文所说，光做用于物体的效果并不是单一的照亮，否则我们的物体会变得是只有颜色没有层次感的单一几何体。模拟光线影响一个物体的最直接的光照模型是Phong光照模型，其通过特殊的参数设定，将环境光，反射光，散射光对于物体的影响单独计算，并且进行加和得到最终的颜色数值。

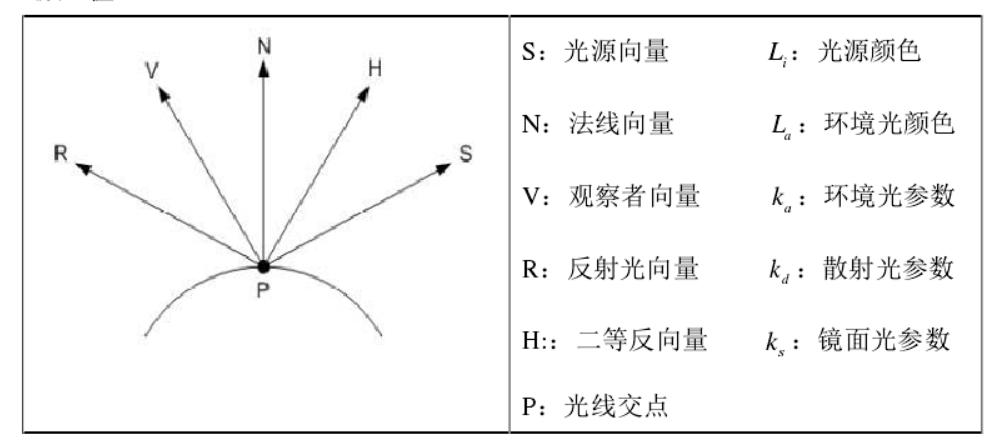


图 3-2 Phong光照模型

* + 1. 阴影

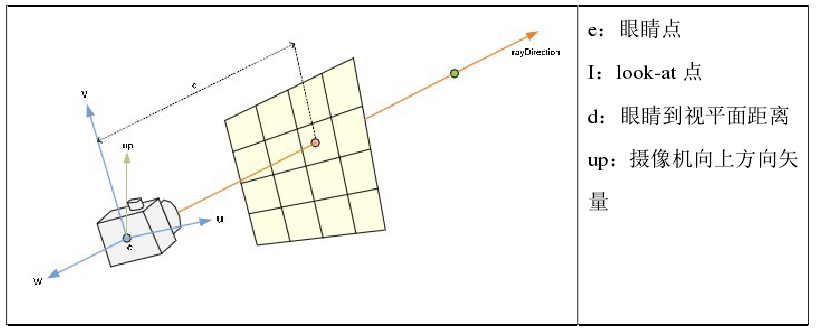
在照亮物体之后，下一个需要设计的则是阴影问题。区别于光栅化算法要实现阴影的复杂设计，光线跟踪算法中的实现更加简单。过程为，检测物体的一点与光源之间是否存在遮挡，如果存在则这一点不被光源照亮，产生阴影效果。

* + 1. 反射与折射

如前文提到的，该进的光线跟踪算法将光线照射到物体之后的效果分为反射光与折射光。反射光线会和折射光线经过方向变换和亮度的微调会再一次进行相交测试，通过测试物体的颜色会通过光线，反应在物体的表面。

* + 1. 摄像机

摄像机的是计算机图形学中，对于人眼的抽象建模。其通常由视锥体作为其模型如图所示：



我们通过摄像机所处的位置Position，朝向Look-At,距离Distance 等变量构建摄像机的投射平面。投射平面是一个看作由小网格构成的平面，每一个小网格和屏幕的像素点一一对应，如一个12\*8像素的平面则有96个小网格。

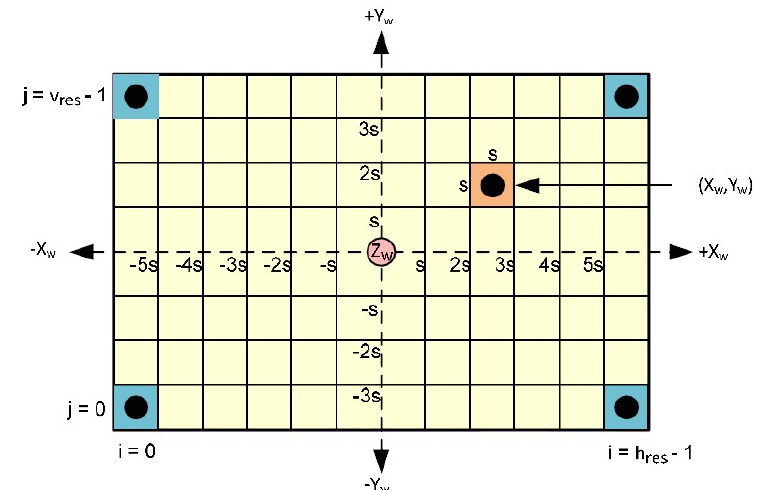


图 3-3 投影平面

在计算机图形学中，世界坐标，投影屏幕坐标和自身坐标系都有严格的区别，投影屏幕是一个二维的坐标系统，如图3-3所展示的，图中的每一个像素点都可以用屏幕坐标（i,j）表示，同时我们也可以用这个来表示像素点在世界坐标的的位置。发射的光线从（0，0）开始，遍历每一个像素，将摄像机的Up与Look-At叉乘可以得到摄像机投影屏幕在世界坐标中的定义，使用下述的公式将世界坐标和屏幕坐标的两个轴进行映射：

可以用简单的算法描述为，传入一个屏幕坐标，使用算法进行映射到[-1,1]区间，然后进行世界坐标的转换：

/\*\*注意这里x乘的顺序，这是会有方向影响的对于向量\*/

RightAxi = Front.cross(Up);

UpAxi = RightAxi.cross(Front);

/\*\*这里是计算在一个视角内，能够映射的范围我们期望的是在[-1,1]之间\*/

FovScale = tan(fov\*0.5\*RayTracing::PI / 180.0) \* 2;

/\*……\*/

RayTracing::Vector3 dxr = RightAxi\*((sx-0.5)\*FovScale);

RayTracing::Vector3 dxu = UpAxi\*((sy - 0.5)\*FovScale);

return RayTracing::Ray(Position,(Front+dxr+dxu).normalize());

* + 1. 相交测试

计算机图形学是一种使用数学算法将二维或三维图形转化为计算机显示器的栅格形式的科学。在三维图像的计算中我们需要处理光线和球体，平面，盒体以及三角形的相交策略。对于这样简单的物体在数学上通常有两种表现形式，一种是使用隐式方程和参数方程进行描述。

对于光线的定义，如3-4所示，在数学上使用一个原点O表示光源的起点，D表示光线的传递方向，T表示光线上任意一点的沿着D方向到O的距离。

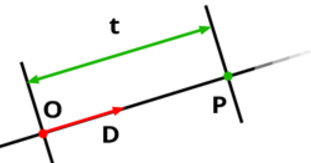


图 3-4光线模型

对于一个球来说，参数方程式形如对于球上任意一点，满足如下的关系：

另外一种表示方式是使用隐式方程进行表示，正是我们常见的解析几何的表示方式，对于我们是实例物体来说则是：

在光线跟踪的渲染方式中，参数表面比起隐式方程，使用的更少，但它可用于计算位于隐式对象表面上的点的纹理。在这种简单的几何物体中，计算光与隐式表面相交计算量更加的简单。同时求解，射线与隐式表面的交点通常比其他几何类型更快。

如果光线与球体相交，则在光线上的点同时也在球体上，也就是说，这一点同时满足光线与球体的方程。

对其进行进一步的展开：



在上图所展示的方程中，除了t之外所有变量均是已知的点，使用求根公式可以对其进行求解，其中a=D^2, b = 2OD,c=O^2-D^2 ,二次方程的根一般为两个，小的那个表示点更加靠近观察点，而大的那个表示点更加的远离观察点，通俗的来说就是面向观察点的为球的正面，反之为背面。

* + 1. 平面

平面是所有的相交的中最容易的一个，在几何上，一个平面在空间中的位置可以用改平面上一点P0和垂直于平面的向量n表示。空间中任意一点P1若在这个平面上，则P1与P0的向量与n的点积为零，同时若P1是从观察点发出的光线上的一点可以得到如下的计算公式：

* + 1. 包围盒

空间中一个包围盒物体可以看做是定义了一组与坐标系的每个轴平行的线的界限的集合，我们也可以使用线性方程来表示。 例如，边界体积的最小范围的x分量的线性方程可以写为X=Bounds.X，其表示包围盒体积平行于x轴的一条边界，其通常有两个数值，一个表示最大值一个表示最小值。当我们需要判断是与平面的交点在包围盒范围之内。

* + 1. 三角形

三角形是计算机图形学中，复杂几何物体的最基本构成单位，拥有复杂外形的几何体都是由许多的三角形物体组成的。空间中不共线的三个点定义了一个平面。每个三角形平面都有一个法线。计算光线与三角形是否有交点，可以大致设计为两个步骤：步骤一，计算光线与三角形所在的平面是否相交；步骤二，判断交点是否在三角形内部。在查找光线与三角形交点的时候，将上文的平面方程变形为：

对于xyz三个点，若都在光线射线上，则应该满足X=Ox+tDx，带入等式化简得到在这个平面上交点的位置。接下来要做的事情是判断这个点在三角的内部还是外部。判断一个点在三角形内部还是外部的方式，在图形学上称为“Inside-Outside”测试。如图3-5所示，对于向量v0v1，判断点v2在其左边还是右边的方式为求得v0v2与v0v1的向量积与法向量再求点积如果这个数值是大于零，则在左边。

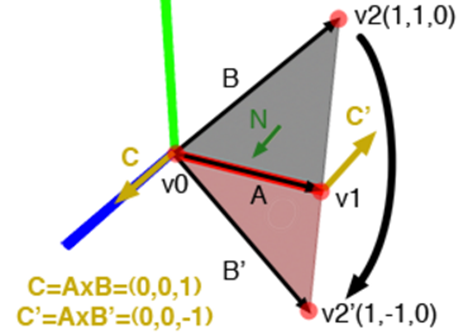


图 3-5 判断左右的示意图

所以三角形内部的点，是按照顺时针方向，处于三角形边的右边的点。

* 1. 算法优化
     1. 场景分割思想

光线跟踪算法的一个大的消耗在于对于每一个像素点，我们都要去采用逆向光线跟踪的方式对场景中的物体进行相交性测试。如果是由多个三角形面片组成的复杂物体，则需要更多次数的相交计算。所以减少相交计算次数的是一种常见的优化思想。现在常用的一种加速方式，是使用树形结构去分割场景。这种结构拥有管理上的先天优势，它能使本来看起来杂乱无章的东西组成有条理的结构，并且益于访问，只需要一个简单的递归语句就能访问其任何的子节点。同时对于复杂物体，通过生成更加简单的物体轮廓减少相交次数的判断，也是优化中的一个重要的思路。通过上述两种方式，减少遍历次数，也就减少了相交计算的运算次数。

* + 1. BVH算法

分割场景的一种常见方式，是想办法找到当前摄像机范围内的物体，只对这些物体进行相交测试，这个就需要用到空间划分的算法，例如BSP以及Kd-tree。Kd-tree是BSP算法的一种特殊结构，这两种算法本质都是通过递归把场景中的多边形，划分为一个一个的小集合生成树形结构，这种算法更加适用于静态场景并且，算法空间冗余度大。另外一种减少相交测试的方式则是这篇论文要用的BVH算法。其基本思想为，将场景中的所有物体都使用一个简单的形状，如长方体或者球体进行包裹。在进行光线跟踪测试的时候，会首先测试光线与这些包围盒是否相交，如果和其中的包围盒相交，则对包围盒内部的子物体也进行相交性测试，直到找到一个最近的包围盒。最后在对这个包围盒内部的物体进行测试渲染。当然，包围盒能被包含在一个更大的包围盒里，这样的方式允许我们对他们进行层次组织。这个算法的有点在于其对于动态场景来说有很高的灵活性，并且算法占用内存空间较少。

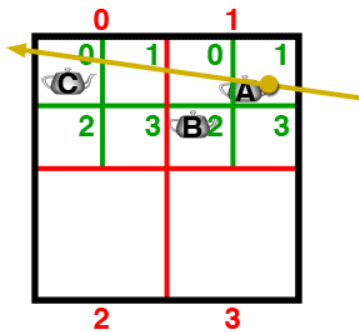


图 3-6 一种树形划分方式

1. 详细设计
   1. 渲染系统设计

光线跟踪系统根据上文的描述，可以将其分为渲染部分，加速部分，相交测试部分以及文件管理部分。本小节主要针对于渲染部分以及文件管理部分进行叙述。

在渲染部分，上文所提到的摄像机，光线以及场景对象是这个部分的基础部件。光线是渲染系统的根本，拥有原点和方向两个向量属性，以及能够根据距离得到交点的功能，所以我们对其的设计如下：

class Ray

{

public:

Ray(const Vector3&,const Vector3&);

Ray(const Ray&);

~Ray();

public:

Vector3 GetOrigin() const { return Origin; }

Vector3 GetDirection()const { return Direction; }

Vector3 GetPoint(double) const;

private:

Vector3 Origin;

Vector3 Direction;

};

对于用于显示最终画面的摄像机，根据其数学模型，需要设计摄像机的位置，朝向，正方向，以及视口宽度（FOV）的大小，并且摄像机应该负责整个系统中光线生产部分的代码操作，其设计代码如下：

class Camera

{

public:

Camera(const RayTracing::Vector3&,const RayTracing::Vector3&, const RayTracing::Vector3&,double);

~Camera();

/\*\*

\* 根据在屏幕上的一个像素点，计算他的坐标反推出其在三维空间的射线的人方程。

\*/

RayTracing::Ray GenerateRay(double sx,double sy);

private:

/\*\*这里的三个分别为摄像机位置，朝向，以及世界的正方向，我们需要根据这三个进行计算，计算出来投影平面\*/

RayTracing::Vector3 Position;

RayTracing::Vector3 Front;

RayTracing::Vector3 Up;

double FovAng;

double FovScale;

/\*\*构建投影平面\*/

RayTracing::Vector3 RightAxi;

RayTracing::Vector3 UpAxi;

};

文件管理系统，在整个渲染器的设计中主要功能是对于3D模型文件进行读写操作。我们需要有一种特定的格式来存储场景文件。现在常见的3D模型文件都是由3DMax以及Maya这样的建模软件所生产的。因为软件的不同，所以存贮方式在业界也有多条规范，例如著名的FBX格式就是其中的一种。在选择场景格式的时候，我有如下的几条考虑：

1. 文件结构要清晰，这样才能为接下来的文件数据的读取提供足够的方便。
2. 第二要有主流的建模软件支持，这样才能方便的获得测试模型。

在综合进行考虑之后，我选择RIB格式作为模型文件格式。RenderMan规范，简称RISpec，是一个开放的应用程序接口，由皮克斯开发，用于描述三维模型并把它转换成逼真的数字图像。RenderMan规范作为建模程序和渲染程序之间的通信协议（或称为接口），用于生成逼真的数字图像。规范类似于PostScript，不过是用来描述三维场景而非二维页面布局。RenderMan渲染接口，主要包括两大部分，第一部分是RenderMan Interface，定义了着色计算所必须的一些基本数据类型，和RenderMan接口字节流协议（RenderMan Interface Bytestream Protocol），简写为RIB，rib也是RenderMan场景描述文件的扩展名；规范的第二部分，是RenderMan着色语言，RenderMan Shading Language，简称RSL，使用RenderMan着色语言编写的源程序，其源代码文件扩展名通常是sl。其字节流格式如下所示：

PointsPolygons [4 4 4 4 4 4] [0 1 3 2 2 3 5 4 4 5 7 6 6 7 1 0 1 7 5 3 6 0 2 4] "P" [-0.5 -0.5 0.5 0.5 -0.5 0.5 -0.5 0.5 0.5 0.5 0.5 0.5 -0.5 0.5 -0.5 0.5 0.5 -0.5 -0.5 -0.5 -0.5 0.5 -0.5 -0.5] "facevarying normal N" [0 0 1 0 0 1 0 0 1 0 0 1 0 1 0 0 1 0 0 1 0 0 1 0 0 0 -1 0 0 -1 0 0 -1 0 0 -1 0 -1 0 0 -1 0 0 -1 0 0 -1 0 1 0 0 1 0 0 1 0 0 1 0 0 -1 0 0 -1 0 0 -1 0 0 -1 0 0] "facevarying float s" [0.375 0.625 0.625 0.375 0.375 0.625 0.625 0.375 0.375 0.625 0.625 0.375 0.375 0.625 0.625 0.375 0.625 0.875 0.875 0.625 0.125 0.375 0.375 0.125] "facevarying float t" [0 0 0.25 0.25 0.25 0.25 0.5 0.5 0.5 0.5 0.75 0.75 0.75 0.75 1 1 0 0 0.25 0.25 0 0 0.25 0.25] "constant string primtype" ["mesh"]

有了上述的基础工作，根据光线跟踪算法的基本流程，首先摄像机生成光线其次，其次对场景中的物体进行求交运算，如果找到交点，则对于交点的材质——即光照模型的最终体现进行计算，将得到的结果写入摄像机屏幕中。

* 1. 加速系统设计
     1. 构建包围盒

在有了上述的代码之后，接下来要做的事情就是为我们的场景构造包围盒。在这个体统中我们采用长方体进行包围盒的构建。如同上文所叙述的，球体和长方体都是计算机图形学中的基础物体，使用二者并无明显优劣之分。球体的优势在于起计算量简单，并且需要存储空间更少——只需要存球心和半径。而长方体对比起球体来说生成更加的简单，但是存储更加的复杂。

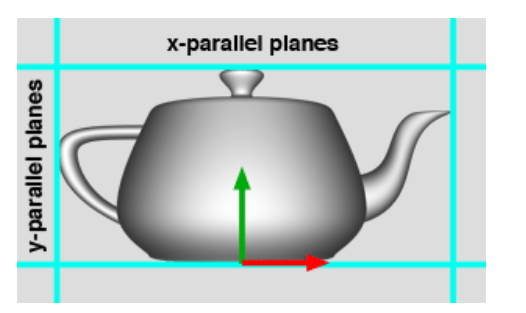


图 4-1 包围盒构建的一种方式

如上图展示的，我们从二维情况下进行思考，构建包围盒的一种方式就是用平行于坐标的的平面对物体进行逼近，图4-1中体现为找到物体在XY平面上分别的最大和最小值，这样就可以简单的构造包围盒体积。所以包围盒的存储格式设计代码如下：

class Extends

{

public:

Extends();

~Extends();

void ExtendBy(const Extends&);

RayTracing::Vector3 GetCenter() const;

bool Intersect(const RayTracing::Ray&,double&,double&);

public:

std::vector<double\*> mBoundary;

StaticMesh\* mMesh;

};

在单个的物体进行加载的时候，读取每一个顶点时候，就可以计算出当前物体的包围盒体积。

* + 1. 构造BVH

介绍完如何构建单个的包围盒之后，接下来的工作是针对整个场景构建整个场景的包围盒层次结构。BVH树在这个系统中的使用是采用一种八叉树的形式存在。树中任一节点的子节点恰好只会有八个，或零个，也就是子节点不会有0与8以外的数目。那么，这要用来做什么？如果场景是一个立方体，我们最少可以切成八个相同等分的小立方体。将整个游戏空间比喻为一个房间，房间里某个角落藏着一枚金币，需要很快的把金币找出来，就可以把房间当成一个立方体，先切成八个小立方体，然后排除掉没有放任何东西的小立方体，再把有可能藏金币的小立方体继续切八等份….如此下去，平均在Log8(房间内的所有物品数)的时间内就可找到金币。因此，八叉树就是用在3D空间中的场景管理，可以很快地知道物体在3D场景中的位置，或侦测与其它物体是否有碰撞以及是否在可视范围内。

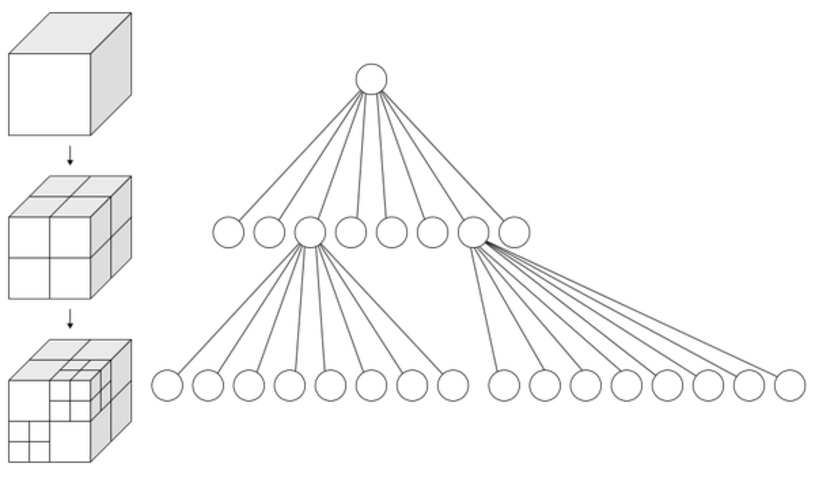


图 4-2 八叉树示意图

根据上面提到的概念，我们可以使用算法递归的构建BVH树的。场景从根节点开始，初始化根节点将其子节点全部初始化为空。每当读取一个几何形体，就加入到这棵树中。在加入到树中的时候，我们需要创建子空间，这个步骤如下：1先计算包围盒的中心点坐标，这个点不是要真是存在，只是一个虚拟的参考点。2得到中心点坐标之后，我们选用X轴（其他的坐标轴也可以）进行分割，可以将场景分为左右两部分，这个时候左边的部分序号是小于4的，右边的部分序号大于4。3判断物体的X坐标，查询物体应该在场景中的那个部分。4在以Y轴进行分割，把场景分割为上下两个部分，并且根据物体的Y轴的坐标判断物体所处于的场景中位置，如果在上半部分则加2。5最后在根据Z轴对场景进行前后部分的分割，如果在上部分，则物体应该在得子节点序号+1。

每当一个物体加入到树中，我们根据坐标先找到根节点下，他所处于的子空间，如果这个子空间为空，即还没有物体处于其中，则创建子空间然后将物体加入其中。如果子空间不为空，从子空间开始递归的进行上述的递归步骤。核心算法如下：

void insert(OctreeNode\*& node, const Extents\* extents, const BBox<>& bbox, uint32\_t depth)

{

if (node->isLeaf) {

if (node->nodeExtentsList.size() == 0 || depth == 16) {

node->nodeExtentsList.push\_back(extents);}

else {

node->isLeaf = false;

while (node->nodeExtentsList.size()) {

insert(node, node->nodeExtentsList.back(), bbox, depth);

node->nodeExtentsList.pop\_back();

}

insert(node, extents, bbox, depth);

}

}

else {

// be inserted into

Vec3f extentsCentroid = extents->centroid();

Vec3f nodeCentroid = (bbox[0] + bbox[1]) \* 0.5;

BBox<> childBBox;

uint8\_t childIndex = 0;

if (extentsCentroid.x > nodeCentroid.x) {

childIndex = 4;

childBBox[0].x = nodeCentroid.x;

childBBox[1].x = bbox[1].x;

}

else {

childBBox[0].x = bbox[0].x;

childBBox[1].x = nodeCentroid.x;

}

if (extentsCentroid.y > nodeCentroid.y) {

childIndex += 2;

childBBox[0].y = nodeCentroid.y;

childBBox[1].y = bbox[1].y;

}

else {

childBBox[0].y = bbox[0].y;

childBBox[1].y = nodeCentroid.y;

}

if (extentsCentroid.z > nodeCentroid.z) {

childIndex += 1;

childBBox[0].z = nodeCentroid.z;

childBBox[1].z = bbox[1].z;

}

else {

childBBox[0].z = bbox[0].z;

childBBox[1].z = nodeCentroid.z;

}

// Create the child node if it doesn't exsit yet and then insert the extents in it

if (node->child[childIndex] == nullptr)

node->child[childIndex] = new OctreeNode;

insert(node->child[childIndex], extents, childBBox, depth + 1);

}

}

以上代码便是构造BVH树的详细过程，BVH的非叶子节点被抽象的包围盒包裹，当光线进入场景之后，会从根节点开始对每个叶子节点进行相交性测试。对于击中的节点，进行距离的比较，选取距离近的节点进行递归测试，直到找到叶子节点中需要进行最终检测和渲染的物体。

1. 编程实现
   1. 主要实现技术
   2. 系统框架
   3. 主要功能模块的实现
2. 功能与性能测试
3. 结束语