

硕 士 研 究 生 读 书 报 告



题目 基于GPU的光线追踪发展总结

作者姓名 张焕

作者学号 21551051

指导教师 李启雷

学科专业 软件工程

所在学院 软件学院

提交日期 二○一五年 十二月

Ray Tracing On GPU

A Dissertation Submitted to

Zhejiang University

in partial fulfillment of the requirements for

the degree of

Master of Engineering

Major Subject: Software Engineering

Advisor: QiLei Li

By

Huan Zhang

Zhejiang University, P.R. China

2016

摘要

在电影动画、虚拟现实、游戏场景等方面光线追踪算法被广泛应用，光线追踪算法的发展伴随着工程产业的进步。近些年，拥有强悍计算速率以及高内存带宽的GPU引起人们的注意，在GPU上实现光线追踪也变得火热。这篇文章首先对光线追踪进行系统性的介绍，再对GPU编程进行相关介绍，最后对现有GPU上光线追踪的技术发展进行总结。

**关键词**：光线追踪，GPU，可编程图形硬件

Abstract

Ray tracing is the important technique to make CG, VR and games’ field.It facilitates the development of industry. Recently, the powerful computation rates and high memory bandwidth of GPUs have attracted people attention.It is popular to implement ray tracing on GPU. This paper, first, introduce the ray tracing, and then we will talk about GPU programming, last, give a summary of ray tracing on GPU.

**Keywords:** ray tracing, GPU, programmable graphics hardware

# 1. 引言

如今，已经有很多计算机游戏使用运行在PC集群的实时光线追踪技术。现在的问题是是否未来一代的GPU们可以进行实时的光线追踪，而CPU仅仅使用一颗。双GPU解决方案正在进入市场，英伟达已经发布了SLI，技嘉科技也即将发布一款双GPU的产品，它将携带两个GeForce 6600 GT。现在有几种可以将光线追踪映射到GPU上的方法，其中之一就是Timothy J. Purcell[1]所提到的将整个光线跟踪器都移植到GPU上进行实现，而且他的想法已经应用在了上述的产品之中。

光线追踪要求大量的计算任务，而这些计算任务都是大量的并行任务，而近期蓬勃发展的GPU正好可以胜任这种任务，它不仅可以加速光线追踪的计算时间还减少了企业的资源消耗，节约了成本，给企业带来巨大的经济效益。CUDA, Optix等技术的成熟更是为GPU编程铺好了前进的道路，吸引了大量的开发人员为其进行开发，从而让产业落地，更好的得到使用。

# 2.光线追踪

## 2.1光线追踪介绍

光线追踪是产生实际景象的一种方法，其中每条光线的路径是沿着所见者到原点的连线，任何一种光线追踪算法的核心都离不开有效的找到由几何元素组成的场景中的光线的交点，场景的渲染由一组几何元素组成，这些几何元素通常是简单的几何形状，例如多边形、球、圆锥等。实际上任何对象都可以作为光线追踪的元素，只要能够计算出他们与光线的相交位置。其中三角形的几何元素让创建、维护、优化光线追踪器变得容易，从而使得设计和实现变得轻松。由于三角形可以近似地表示高级的几何元素，而且大多数场景对球等高级的几何元素展现要求不高，因此工业界更多的使用三角形网格。

## 2.2 光线追踪原理

如图2-1所述描述了光线投射算法[2]，从观察者的方向出发，通过屏幕的像素进入场景，计算出遮挡住光线的物体，然后在对应的屏幕像素点中进行绘制这一点的颜色。从观察者方向出发可以规避那些不能映入眼帘的光线，减少了部分的计算任务。图2-2描述的是Whitted[3]提出的算法，该算法不仅仅是当碰撞到物体后直接将颜色返回给屏幕，而是在碰撞之后分裂出反射光线、折射光线、阴影光线。在跟踪规定次数的反射光线后对相应的像素点更新颜色，折射光线的跟踪可以在绘制时显示图像的变形效果，阴影的跟踪则可以判断某一像素点是否可见，控制明暗程度。

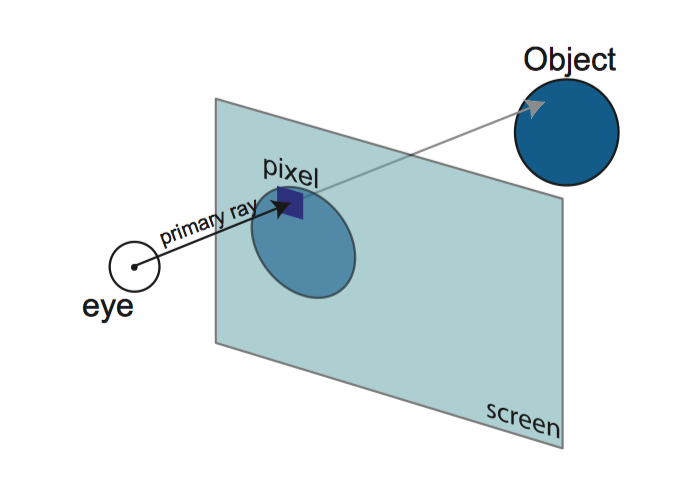


图 2-1光线投射的基本原理

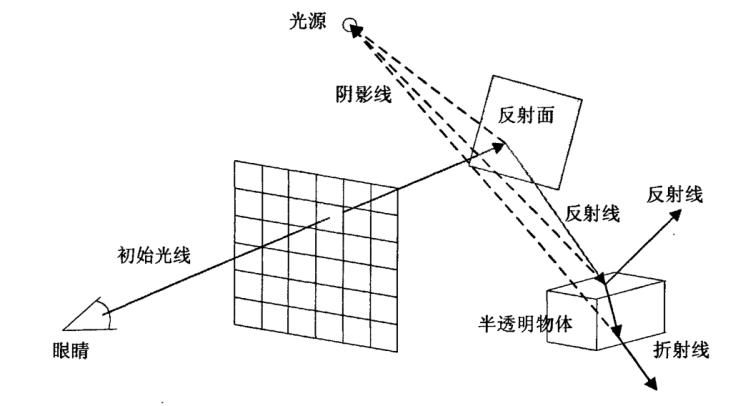


图 2-2 光线追踪的基本原理

## 2.3 光线追踪的简单实现

Turner Whitted介绍过一种通过递归的简单方法实现光线追踪。

|  |
| --- |
| RayRender{  for each pixel x, y{  //Ray originating from the camera/eye  calculatePrimaryRay(x,y,ray);  //Start Recursion for this pixel  color = RayTrace(ray);  writePixel(x,y,color);//write Pixel to Output Buffer  }  } |
| Color RayTrace(Ray& ray){  color = BackgroundColor;  RayIntersect(ray);  if(ray.length < INFINITY){  color = RayShade(ray);  if(ray.depth < maxDepth && contribution > minContrib){  if(ObjectIsReflective()) color += kr \* RayTrace(reflected\_ra  y);  if(ObjectIsTransparent()) color += kt \* RayTrace(transmitted\_r ay);  }  }  return color;  } |

|  |
| --- |
| Color RayShade(Ray& ray){  for all light sources{  shadowRay = GenerateShadowRay(ray, light[i]);  if(shadowRay.length > light\_distance)  localColor += CalcLocalColor();  }  return localColor;  } |

## 2.4 光线追踪的加速

光线追踪是十分耗时的，因为它需要判断大量的相交条件，每条光线需要和场景中的所有物体进行相交计算，所以最主要的加速光线追踪的方法是减少总共的计算次数。因此，我们通常构造一些特殊的空间结构进行遍历操作从而找到光线周围的物体，这些特殊的空间结构有：层次包围盒，BSP树，kd树，八叉树，均匀网格，自适应网格，分层网格等等。像2.3章节这样利用递归方法进行的光线追踪，我们需要限制使用递归的次数，从而转向一种迭代的方法，这样我们可以减少方法调用的时间消耗以及栈的进出次数。

# 可编程的图形硬件

我们常说的CPU 由专为顺序串行处理而优化的几个核心组成。但是GPU 则由数以千计的更小、更高效的核心组成，这些核心专为同时处理多任务而设计。

一般情况下，视觉效果都牵涉到shading，shading决定了物体的表面的明暗以及材质的表现，shading的过程一般都是固定在图形硬件中的方法，然而可编程的图形硬件可以让开发者设计自己的shading代码，我们简单介绍两种图形硬件编程的方式

## 3.1流水线

第一种方法是在图形硬件的shader流水线中使用可编程的shading语言。流水线的结构如图3-1，在shader流水线中含有两种可编程的部分，分别为vertex program和fragment program。

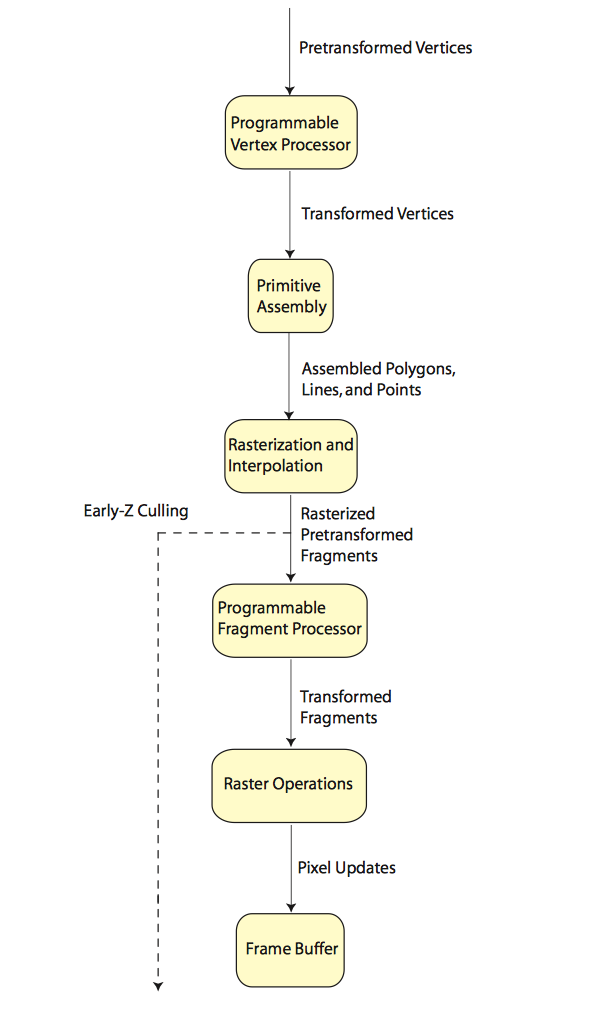


图 3-1 简化的流水线

其中vertex program部分操作顶点的各种属性(位置、颜色、纹理坐标等)，顶点处理器以一个顶点作为输入，并输出一个顶点。在处理完各个定点之后，这些顶点信息会作为图元装配和光栅化的输入，这些定点信息被转换为片段，例如点、线、三角形等，又或者是椎体、剪裁的平面等。下面的一步是fragment program，这一步主要是处理一些传统的图像操作，例如插值计算，纹理应用，烟雾效果，颜色求和等，最终目的是确定某一像素应该显示的颜色值。尽管可编程的shading语言最初是为了渲染而设计的，但是它同样能够高效的实现并行计算。这是因为每个元素(定点或者像素)的计算是由图形硬件上的vertext program、fragment program并行计算的，例如英伟达的GeForce 8800 GTX可以并行的跑128个线程来处理顶点或像素。

## 3.2 流计算

第二种方法是使用基于流计算模型的流计算语言，在流计算模型中，一个处理器对一个元素集合做相同的计算并将计算后的元素集合作为输出，计算的代码叫做“kernel”，这些元素集合叫做“stream”。这个模型使得在整个元素集合中做并行处理变得简单，因为kernel对所有元素做同样的操作。另外，kernel中的计算是独立于输出结果的，也就是说我们可以将数据放入计算之中得到结果，并且不用担心数据之间的依赖链。实际上3.1所讲的shader流水线可以映射到流计算模型中——将vertex、fragment想象为stream，如图3.2。

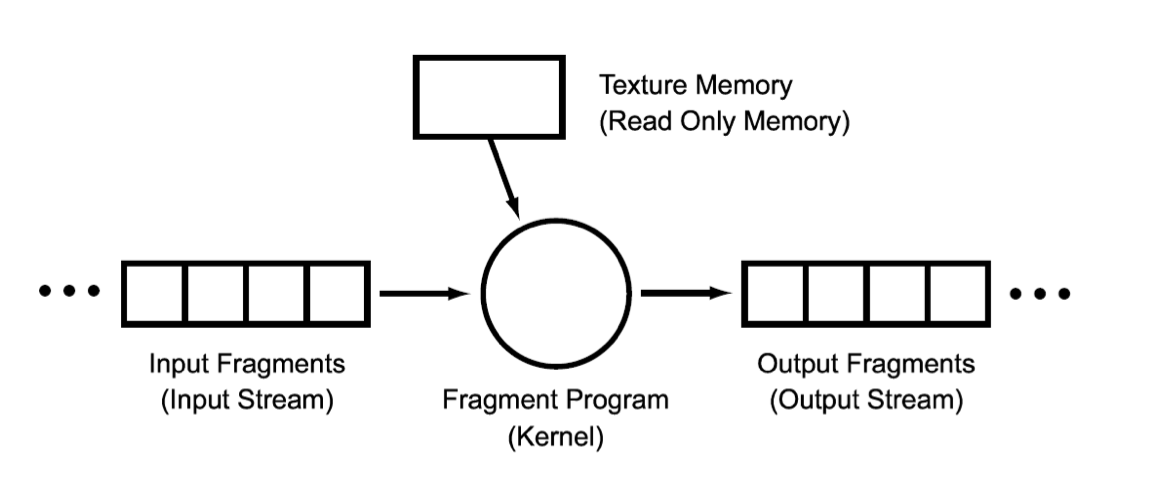


图 3-2 流水线映射到流计算模型

流计算语言通常会根据特殊的硬件进行特殊的设计，因为这样可以更灵活的控制可编程的方法。例如，英伟达的CUDA就仅仅支持GeForce 8系列及其后续产品。而且流编程语言的语法和其他运行在CPU上的高级语言的语法极为相似。在CUDA中，我们写一个运行在图形硬件上的程序就像写一个C++的多线程程序。我们现在有多种流处理语言选择，例如OpenCL和CTM。

# GPU上的光线追踪

因为有效的使用加速数据结构是提升光线追踪性能的关键，近期的研究大多关注图形硬件上实现的光线追踪，我们根据不同的数据结构对其进行总结。

## 4.1均匀栅格

在图形硬件上完成光线追踪的第一个光线追踪器就是利用的均匀栅格[4]，均匀栅格将整个场景的包围盒细分为均匀大小的体素——包围盒中的矩形子空间。每个体素包含一组三角形覆盖在它上面。当光线追踪初始化阶段时，光线和场景包围盒的交点将被计算，基于这个交点就可以找到光线通过哪个体素进入的均匀栅格。当我们找到这个初始体素后，光线沿着光线的方向进入下一个体素，只要体素中包含三角形，那么光线与三角形的交点就是实际的交点。因为光线三角形的交点仅仅由被光线穿过的体素确定，所以均匀栅格减少了光线三角形交点的计算数量。

均匀栅格上的光线追踪能够映射到流处理器模型，我们仅需要知道现在的体素索引，光线的初始位置，光线的方向就可以计算出下一个体素在哪里。所以体素的索引，光线的原点以及光线的方向作为输入流，kernel完成单步的光线追踪并把下一个体素的索引作为结果输出。

均匀栅格的结构对于三角形均匀分布的情况效率很高，但是对不均匀分布的三角形情况就不好了，例如在一个大体育场中的小茶壶问题，如果我们在体育场中创建均匀栅格，一个单一的体素可能包含茶壶的所有三角形。这样的话，我们对于光线三角形的交点计算量一点都没有减少。

## 4.2 kD树

避免4.1所述的问题的一种加速数据结构是kD树，kD树根据平面的层次将场景切分成多个小区域。每个kD树的非叶节点存储了一个平面，这个平面将空间分为两面。这个细分的操作一直执行到符合某一条件(每个叶节点的三角形数目达到标准)。最后每个叶节点存储了一组三角形，这些三角形覆盖在他们之上，图4-1展示了2D中的kD树[5]，S0，S1，S2是分割空间的平面，n0，n1，n2，n3是叶节点，右面的树状结构显示了kD树的层次结构。基于kD树的光线追踪可以利用本地栈来有效的实现，但是对于可编程的图形硬件来说利用本地栈是不可能的。为了避免使用栈，Foley等人提出了两个不用栈的kD树的追踪方法[5]，分别为kD-restart和kD-backtrack。图4-2描述了两种算法的大体含义。

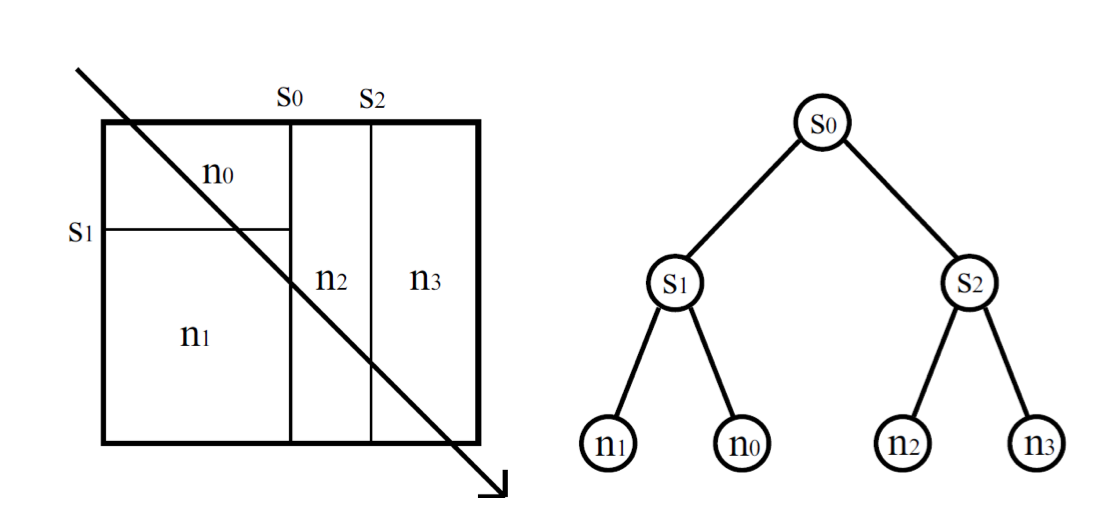


图 4-1 2D kD树

**kD-restart**

当光线离开叶节点后该方法简单地从根部重新开始光线追踪，光线不会重复访问同样的叶节点，因为光线的区间被缩短了。如果n代表节点的数量，那么最坏情况的时间复杂度是O(nlog(n))，相对于栈的kD-树的时间复杂度O(n)，额外多出的O(log(n))来自于我们想要获取光线访问过的节点数量。

**kD-backtrack**

该方法利用指向父节点的指针来进行回溯操作，从而避免了额外的时间开销，Foley指出，放入栈中的节点其实就是上一节点的子节点，因此借用回溯操作可以在避免使用栈的同时获得栈的效果。该方法的最坏情况是O(n)，但是回溯父节点需要额外的存储开销，因为每个节点都要存储一个指向父节点的指针。

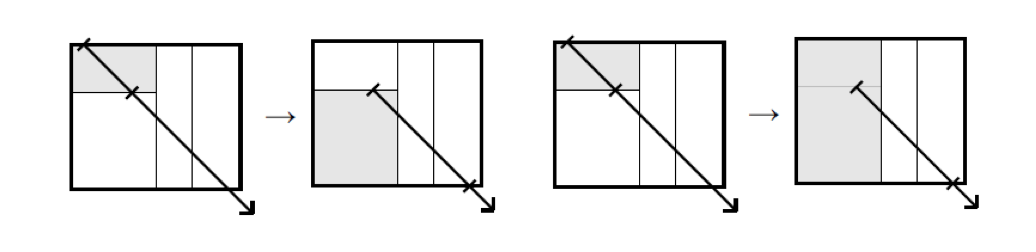


图 4-2 kD-restart和kD-backtrack

## 4.3 包围盒层次(BVH)

均匀网格和kD树都是将一个空间切分成更小的空间，其实我们可以讲那些三角形分类到更小的子集中，而不是操作空间。BVH使用这种方法构造存储包围容量的树，包围容量是一个充满三角形集合封闭图形，虽然图形的形状可以自己定义，但是轴对齐的包围盒(AABB)，因为AABB在计算交点时效率较高，如图4-3所示是一个用AABB实现的BVH，每个节点都是一个包含子节点的AABB结构，根部的AABB就是整个场景。叶节点包含一组三角形。

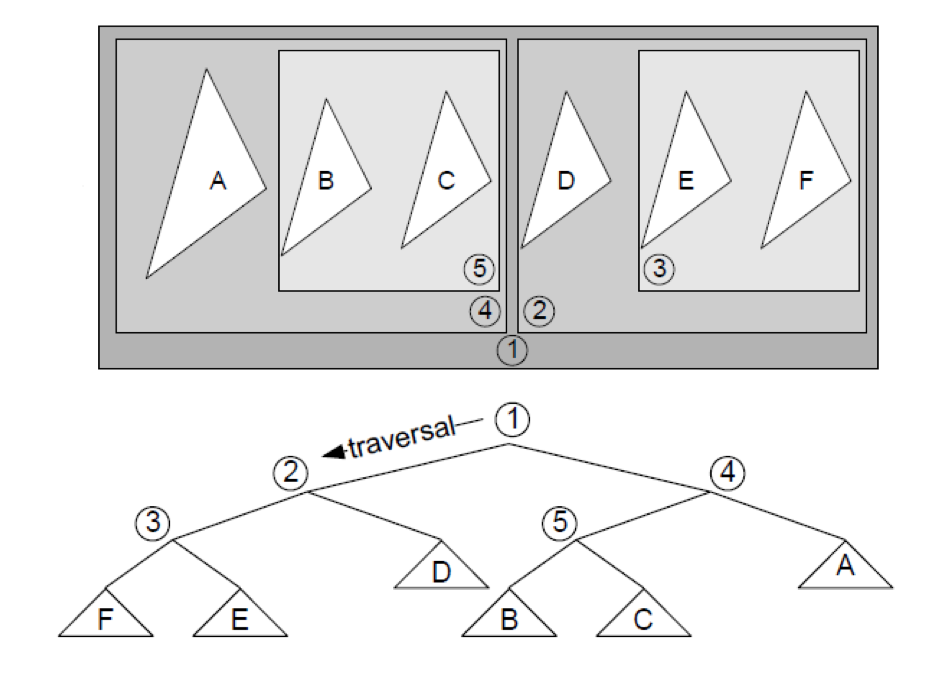


图 4-3 AABB

BVH在CPU上的实现同样需要栈，因此在图形硬件上仍然需要重新设计。ThreadingBVH[6]用来解决这个问题，Threading将节点的父节点和兄弟节点链接在一起，其实兄弟节点就是下一次追踪的节点。这些链接的选择是在光线追踪时确定的，依赖于光线是否与节点发生碰撞，因此也叫作碰撞丢失链接。

# 小结

光线追踪是计算机图形学领域最热门的研究方向之一，通过光线追踪我们可以表现物体的阴影、反射和透明度，工业上也大量使用光线追踪来制作高质量的图像，近年来GPU的快速发展解放了CPU的大量耗时操作，将光线追踪移交给GPU来做也逐渐变成主流，CUDA、OptiX等开发语言的提出也引来了大量的开发人员来为GPU计算做出贡献，由于CPU与GPU硬件结构的不同，人们通过算法来改进GPU上的耗时操作，充分发挥GPU并行计算的优势，我们相信，GPU能力的挖掘还远远没有到达尽头，关于动态场景的操作以及无栈算法的研究已逐渐成为主流，这也意味着光线追踪的速度和效果将要更上一层楼。

[1] Timothy John Purcell, Ray Tracing on a Stream Processor, 2004.

[2] Appel A. Some techniques for machine rendering of solids. In:AFIPS Conf erence Proceedins. San Francisco, California, USA, 1968, 32, 37-45

[3] Whitted J T. An improved illumination model for shaded display of curve d surfaces. In: Proceedings of International Conference on Computer Gra phics and Interactive Techniques. Atlanta, GA, USA, 1980, 26-27

[4] PURCELL , T.J., BUCK , I., MARK , W.R., AND HANRAHAN, P.2002. Ray traci ng on programmable graphics hardware. ACM Trans. Graph. 21, 3, 703–712.

[5] FOLEY, T., AND SUGERMAN, J. 2005.Kd-tree acceleration structures for a gpu raytracer.In HWWS’05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, ACM, New York, NY, USA, 15–22.

[6] THRANE , N., AND SIMONSEN, L.O.2005.A comparison of acceleration struct ures for GPU assisted ray tracing. Master’s thesis, University of Aarh us.