1
Introduction

One of the fundamental areas of Computer Graphics is the rendering of three-dimensional scenes. Rendering can be defined as the process of generating an image from a model, using computer software. The model is a description of three-dimensional objects using a strictly defined language or data structure. It might contain geometry, material, lighting and other shading information. A typical geometry representation uses polygons, usually triangles, to discretize smooth surface attributes. Nowadays, there are two major image synthesis algorithms: rasterization and ray tracing.

Rasterization is based on the painter's algorithm: each primitive is projected on a plane, determining which image pixels it affects, as shown in Figure 1.1. New pixel values are computed according to a specified shading algorithm. In order to display only the primitives closest to the viewer, the application keeps track, for each image pixel, of its corresponding depth value in a z-buffer. Incoming pixels are only stored if their z-value is smaller than the previous one.

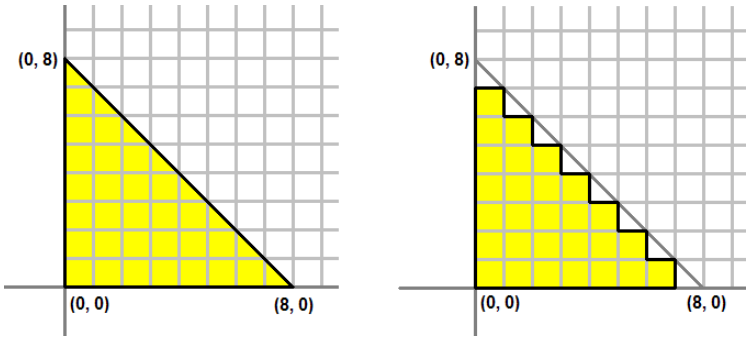


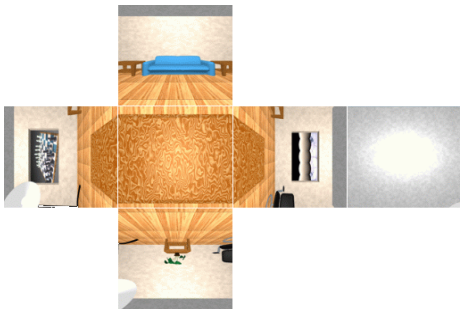
Figure 1.1: A triangle projected on the image plane (left) and its corresponding pixels after rasterization (right).

Since each polygon is processed separately, this algorithm has a typical rendering time that is linear in the number of primitives. However, specialized hardware for rasterization has been developed for several decades. A modern GPU processes hundreds of primitives in parallel, greatly reducing the overall rendering time. Moreover, many algorithms and data structures can be used to reduce total hardware workload. For instance, one can use spatial hierarchies

to cull objects that would not contribute to the final image, prior to processing them on the GPU.

On the other hand, shading computations are also done for each primitive. While this allows for a parallel hardware implementation, it also means that only local lighting information is readily available. In order to solve more complex illumination models, rasterization requires the usage of unorthodox techniques which usually are computationally expensive and can only obtain an approximation of the desired effect.

For example, reflective materials are simulated by reflection maps instead of computing accurate radiance transfer. For changing environments these maps must be re-computed each rendering frame using costly additional rendering passes (see Figure 1.2). Even worse, accurate object inter-reflections demand several map re-computations. Additionally, the resulting effects can appear distorted for close-by or curved objects.



1.2(a): Six textures forming a cube map.

1.2(b): Resulting object reflections.

Figure 1.2: The environment map is obtained by rendering the scene from the object's point of view an additional six times [Nvidia 2004].

As a general rule, any effect that requires a global understanding of the geometry and radiance distribution in a scene is computationally expensive and hard to obtain using traditional rasterization.

Ray tracing overcomes these limitations by being able to achieve sub-linear rendering time as well as global illumination effects. Essentially, ray tracing computes the incoming radiance from the three-dimensional model to the viewer. For each pixel to be stored in the resulting image, a ray is cast from the viewpoint towards the scene to determine the nearest object intersection. Shading at that intersection point is done by combining incoming radiance, obtained by casting additional rays, with local surface and material properties. Figure 1.3 illustrates this process.

This defines a recursive algorithm, where each new ray performs another shading computation and transmits this lighting information across the scene. Note that computation done by each ray is independent, and thus can be easily parallelized.

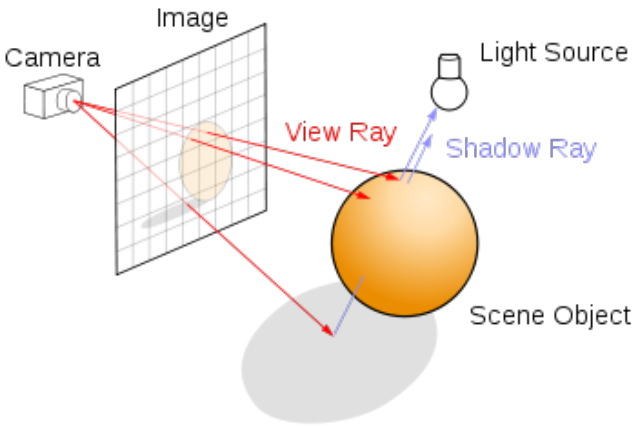


Figure 1.3: The ray tracing algorithm. In the image, shadow rays determine incoming radiance from the light source.

The amount of radiance leaving a point (L_o) is given as the sum of emitted (L_e) plus reflected radiance. The reflected light itself is the sum of the incoming light (L_i) from all directions, under a geometric optics approximation. Figure 1.4 illustrates this formulation.

In order to describe mathematically how energy is transmitted throughout different objects and materials, it is possible to define an integral equation, called the *Rendering Equation* [Kajiya 1986], as follows:

$$L_o(x, w, \lambda, t) = L_e(x, w, \lambda, t) + \int_\omega f_r(x, w', w, \lambda, t) L_i(x, w', \lambda, t)(-w' \cdot n) dw' \tag{1-1}$$

λ is a particular wavelength of light

t is time

$L_o(x, w, \lambda, t)$ is the total amount of light of wavelength λ directed outward along direction w at time t, from a particular position x

$L_e(x, w, \lambda, t)$ is emitted light

$\int_\omega \cdots dw'$ is an integral over a hemisphere

$f_r(x, w', w, \lambda, t)$ is the bidirectional reflectance distribution function, the proportion of light reflected from w' to w at position x, time t, and at wavelength λ

$L_i(x, w', \lambda, t)$ is light of wavelength λ incoming toward x from direction w' at time t

$-w' \cdot n$ is the attenuation of incoming light due to incident angle

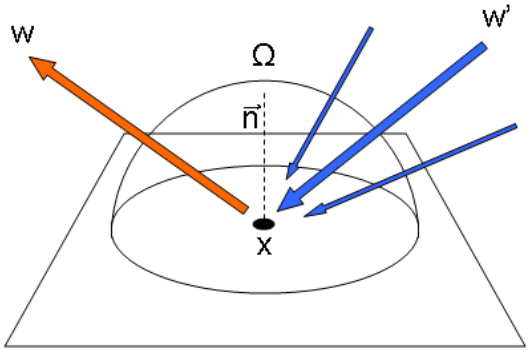


Figure 1.4: The rendering equation describes the total amount of light emitted from a point x along a particular viewing direction, given a function for incoming light and a BRDF.

Solving the rendering equation for any given scene is the primary challenge in realistic rendering. Typical illumination models used in rasterization are a simplification of this equation in order to compute a satisfactory approximation given time constraints. From another standpoint, ray tracing provides a framework for directly integrating Equation 1-1 by computing, for each intersection point, discrete radiance samples from several directions in a recursive manner. As illustrated by Figure 1.5, it is possible to compute global illumination effects such as shadows, reflections, or indirect lighting simply by casting additional rays.

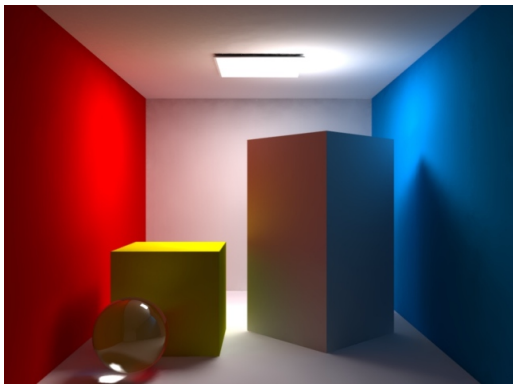
However, the computational cost of tracing millions of rays can become prohibitive. In fact, ray tracing has always been associated with off-line rendering, where an application takes several minutes or even hours to compute a single frame. Even though the resulting image can be called photorealistic, this performance still limits the general adoption of ray tracing in other Computer Graphics applications. Only recently this scenario began to change.

Over the past few years, ray tracing algorithms have been extensively improved. Great advances in intersection procedures and spatial acceleration structures, together with increased hardware processing power, have made real-time ray tracing a reality. Spatial structures can be used to achieve sub-linear rendering time. They are used to quickly compute the nearest subset of primitives along a given ray. This greatly reduces the total number of intersection tests, effectively trading intersection cost with structure traversal cost.

Another improvement has been the development of techniques that perform several ray computations in parallel. One idea uses packets of nearby rays together with SIMD hardware instructions to trace them at the same time. Other coarse-grained implementations use clusters to distribute ray

1.5(a): Indirect illumination.

1.5(b): Soft shadows and caustics from area light.



1.5(c): Diffuse inter-reflections.

1.5(d): Realistic material properties.

Figure 1.5: Real-time global illumination effects obtained with ray tracing [Wald et al. 2003].

intersection and shading computations across several computers.

Initially, real-time ray-tracing research focused on obtaining maximum performance for static scenes. In these, great efforts are spent building the best possible acceleration structure. That is, one that returns the nearest subset of scene objects in the least amount of time. These implementations can afford to spend several seconds, or even minutes, pre-constructing a data structure that is best suited for the ray traversal algorithm.

On the other hand, several applications require the display of object animations and realtime user interaction. In order to ray trace these dynamic scenes, it becomes clear that the acceleration structure must be updated — if not rebuilt — every time movement occurs.

Only recent results have shown good performance in ray tracing these types of scenes. Some techniques involve updating a hierarchy, maintaining its original topology, while others aim to rebuild the entire structure from scratch. Current processing power limits these implementations to scenes with a few hundred thousand triangles.

State-of-the-art research have also shown that ray tracing on the GPU can achieve similar if not better performance than the best known algorithms

on the CPU. Most of these techniques have been restricted to static scenes. To our knowledge, only a single GPU implementation has been able to support moving and deformable objects. Therefore, several techniques remain unexplored.

The goal of this work is to develop a ray tracing solution that is capable of harnessing the parallel processing power of a GPU to render dynamic scenes. Similar to other proposals, we have focused on the strategy of fully rebuilding the acceleration structure in order to support scenes where any of the following is true:

- Objects can have rigid body motion
- Objects can have deformations
- Movement can be unstructured

Given the restricted programming architecture of modern GPUs, we have chosen to use the *Uniform Grid* as an acceleration structure. Its main advantages are the simplicity of ray traversal through a regular spatial subdivision, as well as a straightforward construction algorithm. The main contributions of this research are:

1. A novel algorithm for building Uniform Grids in parallel shared-memory architectures
2. An optimized implementation of this construction algorithm using the GPU
3. A ray-tracing procedure also implemented on the GPU, using the grid structure to accelerate ray traversal
4. An advanced shading implementation including textures, shadow and reflection rays

Results demonstrate that our grid reconstruction algorithm is not only scalable with scene size, but also achieves faster rebuild times than state of the art CPU procedures. In addition, our ray-tracing procedure is able to obtain competitive rendering rates for static and even fully animated scenes.

This document is organized in 7 chapters. The next chapter reviews related research in GPU ray tracing, the different acceleration structures commonly used and several ray-tracing results on dynamic scenes. Chapter 3 presents our proposed architecture, introducing the basic concepts for ray tracing using Uniform Grids and how to best utilize the graphics hardware. Chapter 4 describes our proposed method for storing and rebuilding the grid

data inside the GPU. In Chapter 5, we describe the ray traversal algorithm used to trace rays through the Uniform Grid on the graphics hardware. Results and performance numbers are evaluated in Chapter 6, where several test scenes identify the benefits and limitations of our approach. Finally, Chapter 7 concludes this research and introduces several future work that can further improve ray tracing dynamic scenes on the GPU.