Efficient Ray Tracing of Parametric Surfaces for Advanced Effects

Thesis submitted in partial fulfillment
of the requirements for the degree of

MS by Research
in
Computer Science and Engineering

by

ROHIT NIGAM
200702036
rohit_n@students.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500 032, INDIA
July 2015

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Efficient Ray Tracing of Parametric Surfaces for Advanced Effects" by Rohit Nigam, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date	Adviser: Prof. P. J. Narayanan

To my Parents, Jyoti and Gajendra Nigam and my little brother Mohit.

Abstract

Ray Tracing is one of the most important rendering techniques used in computer graphics. Ray traced images are more accurate and photo-realistic as compared to direct rendering. Ray Tracing was earlier considered impractical for rendering scenes at interactive rates because of its high computational cost. However, with the advancements in modern Graphics Processing Units (GPU) and CPUs, ray tracing at interactive rates has now become possible.

Parametric patches have been widely used in many fields to describe a model accurately. They provide a compact and effective way of representing an object and also possess the ability to remain curved on zooming. Ray Tracing of parametric surfaces was considered to be a static process because of the high complexity of intersection algorithms. With advancements in ray tracing techniques and high compute power devices, recent works on ray tracing parametric surfaces have reported near interactive results.

We present a scheme for interactive ray tracing of Bezier bicubic patches using Newton iteration in this dissertation. We use a mixed hierarchy representation as the acceleration structure. This has a bounding volume hierarchy above the patches and a fixed depth subpatch tree below it. This helps reduce the number of ray-patch intersections that needs to be evaluated and provides good initialization for the iterative step, keeping the memory requirements low. We use Newton iteration on the generated list of ray patch intersections in parallel. Our method can exploit the cores of the CPU and the GPU with OpenMP on the CPU and CUDA on the GPU by sharing work between them according to their relative speeds. A data parallel framework is used throughout starting with a list of rays, which is transformed to a list of ray-patch intersections by traversal and then to intersections and a list of secondary rays by root finding. We are able to significantly outperform multi-core CPU implementation and previous GPU implementation using the mixed hierarchy model.

Shadow and reflection rays can be handled exactly in the same manner as a result. The secondary ray list is again sent to the starting of the algorithm to perform mixed hierarchy traversal and intersection tests. We perform fixed depth multiple bounce ray tracing. We also show how our method extends easily to generate soft shadows using area light sources. These effects provide higher realism to the ray traced images.

We render a million pixel image of the Teapot model at 125 fps on a system with an Intel i7 920 and a Nvidia GTX580 for primary rays only and at about 65 fps with one pass of shadow and refection rays. We are able to ray trace bigguy in a box scene with multi-bounce at near interactive rates. We

get a speed up of about 5-30x for our hybrid Newton's method implementation over our optimized CPU implementation and about 20-50x over previous GPU implementation of Kajiya's method to ray trace Bezier surfaces. Traversing the mixed hierarchy is the most time consuming step of the algorithm. We expect to see better performance with greater cache size. The hybrid model would be optimal for systems with equal compute power of CPU and GPU. The proposed model is suitable for parallel architecture, hybrid systems and multi-GPU systems.

Global illumination effects have recently started gaining popularity with the progress in parallel architecture. We extend our algorithm to global illumination effects to demonstrate its capabilities. Global illumination effects have not been reported for parametric surfaces. We perform path tracing by tracing a large number of rays per pixel for a fixed depth. Number of samples greatly increase the quality of the image generated. We also perform more advanced effects like ambient occlusion, depth of field, motion blur and glossy surface. We are able to path trace a 512×512 image with 1000 samples per pixel in about 165 seconds. We report timings for other advanced effects. We find that ray coherence is essential for optimal performance when ray tracing Bezier surfaces on the GPU. Size of the dataset also plays a small part in the overall rendering times. The work done in this dissertation should serve as the starting point to optimally render Bezier surfaces with advanced global illumination techniques.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the world of Computer Graphics, rendering is the process of generating images, which are close to captured photographs, from a data model. Majority of the research focuses on making these images as photorealistic as possible. A lot of work goes into accelerating their generation. While animation videos focus upon the photorealistic aspect of the rendered images, computer games and many other applications rely on generating them at interactive rate.

Rasterization and ray tracing are the two basic methods used for rendering of images. Rasterization using the Painter's Algorithm or Depth Buffering determines which objects are visible to the camera and fills the image pixels accordingly. The complexity of rasterization is linear in the number of surfaces present in the data model. Rasterization can generate images at a very high rate. This makes it the most popular rendering technique for 3D computer games and applications which require image generation at interactive rates. However, rasterization does not take into account the interaction between different objects and thus fails to produce images which are photorealistic. Thus, real world effects like reflections, refractions, shadows etc., are not inherent in this approach. One can apply techniques such as reflection maps, to generate these effects, but that increases cost of rendering each frame and are only approximate in most cases.

With the advent of powerful Graphics Processing Units (GPUs) with advanced programming capability, many of the complex problems which seemed impossible to perform at interactive rates are now possible. Ray Tracing, which was previously used only to generate static images because of the high computational time, has now come at par with the required frame rates. Similarly, real time processing of objects defined using parametric surfaces has also become possible.

1.1 Ray Tracing

Ray Tracing is the most popular technique used in Computer Graphics to render visually realistic images. Ray Tracing is just the mirror opposite of how a real world camera works. Instead of measuring the amount of light entering a pixel, we shoot ray(s) through a pixel, simulate the path which it traces, taking into account its interaction with different objects in the scene and finally return with a color

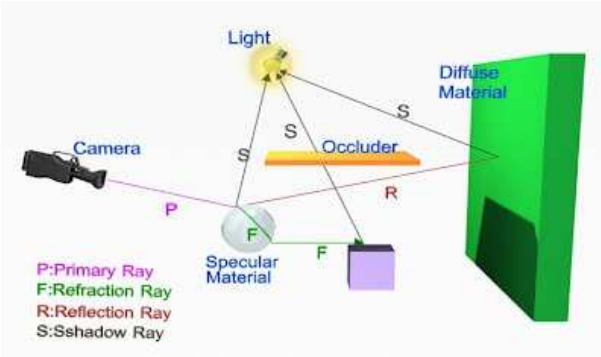


Figure 1.1: Ray Tracing (image obtained from siliconarts.co.kr)

value for the corresponding pixel. Ray traced images attain high level of visual realism, but at a high computational cost. A naive implementation of a ray tracer would shoot rays equal to the resolution of the image and each ray would interact with every object present in the scene. We can reduce the total number of rays and reduce the number of objects to interact, making the rendered time much lower, with advanced techniques.

Since Ray Tracing is basically simulation of a ray, one can calculate the exact hit points and the normals at these points where the ray intersects a surface. This feature is missing in the rasterization approach, but is essential to trace secondary rays. With the information of surface normals, we can now perform secondary ray tracing which includes reflection, refraction and shadows. Reflection occurs in case of shiny objects. Whenever a ray hits an object, a new ray can be spawned at the point of intersection having the mirror opposite direction with respect to the surface normal. It now traverses the scenes and returns with the reflected color value. Refraction occurs in case of transparent/translucent objects. Whenever a ray hits an object, the ray enters the object (exits if it started inside the object). The direction of the ray is determined by the refractive index of the material. In order to determine if the point of intersection is visible to the light source, a shadow ray is spawned in the direction of the light source. If the ray does not intersect any object upto the light source, we say that the point is illuminated and it is not shadowed. Otherwise it is under a shadow. The color value from reflection/refraction is added to the original value. The shadow bit is used to simulate the shadowed regions. Thus the final image generated after taking secondary effects into consideration is visually much more realistic.

A ray tracer shoots rays into the scene which return the color of the objects they hit. An object in a scene can be represented in many different ways. The most common of which include polygon meshes, subdivision surfaces, implicit surface and parametric surface. Each type of surface representation has its own advantages and disadvantages. Figure 1.2 shows some of these surfaces.

An implicit surface representation is of the form $F(x) = 0$, where x is the point on the surface described by function F. For example, a sphere, centered at origin, is represented by $F(x, y, z) = x^2 + y^2 + z^2 - r^2$ in this form of representation, where x, y and z are coordinates of the point and r is the radius of the sphere. Most of the times, implicit surfaces are constructed in a way to hold the

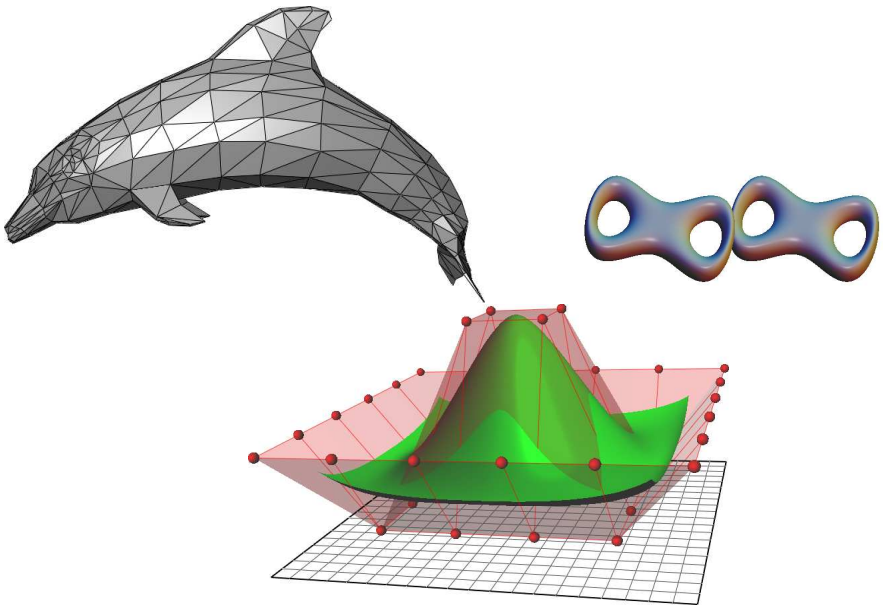


Figure 1.2: Top: Triangular Mesh Model, Implicit Surface Model. Bottom : Parametric Surface Model

following properties:

$$F(x) > 0 \qquad \text{if } x \text{ lies outside/above the surface}$$

$$F(x) = 0 \qquad \text{if } x \text{ lies on the surfaces}$$

$$F(x) < 0 \qquad \text{if } x \text{ lies inside/below the surface}$$

Implicit surfaces can be very well used for ray tracing. We need to represent a point in ray parameters and substitute this in $F(x) = 0$ to solve for intersection. This form of representation allows for a easy inside/outside/on the surface test but fails when one is required to gather consecutive points on the surface.

Polygon meshes are the most basic form of surfaces used in rendering. Polygon meshes usually comprise of triangles, sometimes quadrilaterals or other simple polygons, to represent a scene. Since triangles are easy to check for instruction and the scenes are also relatively easier to construct, triangle meshes have become the most common form of surface representation.

While polygon mesh has its advantages, it also suffers from the disadvantage of not remaining perfectly smooth when we zoom into the scene. It cannot represent objects found in nature or on the shapes of cars, etc. Since it is constructed from triangles, which are planar, this type of behaviour is expected. One can tesselate the triangles, but it would lead to preprocessing costs and also additional cost for storing in memory. To eliminate these disadvantages, one can use parametric or implicit surfaces, which have the inherent property to remain curved even at arbitrary level of zooming. Implicit surfaces have the ability to remain smooth but are limited. Parametric surfaces are more general and can define any shape required in a scene.

1.2 Parametric Surfaces

Parametric Surfaces are defined by a parametric equation. A parametric surface can be represented by $P = F(u, v)$ where u and v are surface parameters and P is a point on the surface. Parametric representation of a surface is the most general way of representing a free form surface. A unit radius circle is defined parametrically as follows:

$$(x, y) = (\cos(t), \sin(t)) \qquad \text{for } 0 \leq t < 2\pi$$

The surface normal at a point is given by the cross product of the partial derivatives at the point. If we denote the surface by $S(u, v)$, then the partial derivatives are $S_u(u, v)$ and $S_v(u, v)$ along u and v parameters respectively. To finally get the unit vector along the direction of the normal, we divide this vector by its magnitude. Thus,

$$\hat{n} = \frac{S_u \times S_v}{|S_u \times S_v|}.$$

Parametric surfaces are used widely in Computer Aided Design (CAD) and other fields. They provide a compact and effective representation of geometrical shapes for engineering, graphics, etc. The most powerful feature of parametric surfaces is their ability to stay curved and smooth even when viewed at close distances. A free-form object may take much higher number of triangles to represent it accurately as compared to parametric patch. Hence, the data representation for parametric surfaces is concise in comparison to polygon meshes. Usually, many parametric patches are joined side by side to draw a complex shape. Another important feature of parametric surfaces is their ability to represent exact boundaries for objects. While triangular meshes try to approximate the boundaries with linear edges, parametric patches represent them exactly. Parametric surfaces are also convenient when performing texture mapping as the (u, v) parameters can be again reused as (u, v) texture coordinates.

Higher order parametric surfaces are defined by a set of control points, which model the surface represented. Using these control points, one can interpolate the values of all the points on a surface. Parametric surface have the property of being contained within the convex hull of their control points. Figure 1.3 shows the representation of parametric surfaces.

There are many different types of parametric surfaces. Bezier bicubic patches are the most basic form of parametric surfaces. Each point on a Bezier curve is computed as a weighted sum of all control points. The surface in general passes only through the corner control points and is stretched towards the rest of the control points. B-Spline surfaces are a set of connected Bezier surfaces with C2 continuity. C2 continuity implies that a B-Spline surface is positionally, tangentially as well as parametrically continuous. Thus, these surfaces remain smooth at the edges of the Bezier surfaces which constitute them. These surfaces make use of the blending function, for (u, v) parameters, for blending the patches. Non-Uniform Rational Basis Splines(NURBS) surfaces represent a more generalized form of parametric surfaces. Each control point of the curve has a weight assigned to it, which makes NURBS surfaces rational. Specifying each control point in homogeneous coordinate system provides the surface its non-uniform nature.

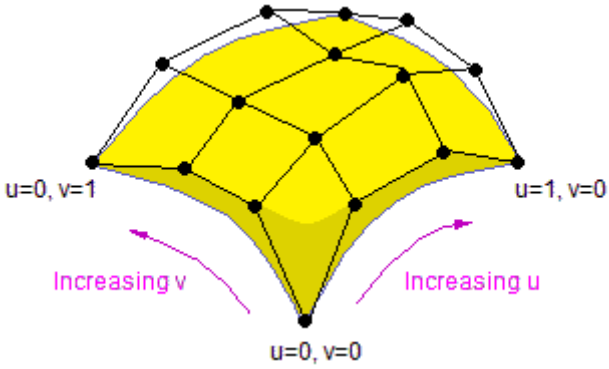


Figure 1.3: A parametric patch, modelled by its control points.

While NURBS are more general, Bezier surfaces are able to provide the level of freedom which most applications require. They are much easier to manipulate as compared to NURBS surfaces and provide a uniform degree for all the surfaces used in a data model. Additionally, one can model other generally used parametric surfaces using a relatively small number of Bezier patches. NURBS surfaces can be easily converted to Bezier patches.

Even though Bezier patches offer the advantage of being perfectly curved at arbitrary level of zooming, the mathematics involved in computing intersection of a ray with a Bezier surface is much complex and computationally expensive. Hence, for a large period of time, parametric surfaces in general have been rendered by converting them into polygon meshes, which are then sent for rendering. However, with the powerful processors available today, direct ray tracing of parametric surfaces at interactive rates has now become possible.

1.3 Rendering of Parametric Surfaces

There are basically two forms of rendering techniques which are used for displaying parametric surfaces. The first method converts a parametric patch into an equivalent triangle mesh and displays the generated the mesh. This form of rendering is the most commonly used technique in CAD systems and many other applications. Since rendering triangles is the very basic form of rendering, this method produces images very fast and is used in systems which require real-time performance.

Converting a parametric patch into triangle mesh has its disadvantages. When we convert a surface into triangulated model, holes might get created in the generated model, which produces visual artifacts in the generated image. There have been techniques developed to improve the algorithm but the problem could still persist. The other disadvantage is that the surface is now approximated by the triangles. Hence the results produced can vary from the actual results. This could lead to artifacts appearing in the image. Also, the property of parametric surface to remain smooth even at arbitrary level of zooming gets lost in this approach, the tessellation performed ahead of time. Another major disadvantage in converting to triangle mesh form is that since we do not get an actual hit point, we cannot generate the surface normal

at the hit point and perform secondary ray tracing. Thus, only primary effects are produced accurately by rendering parametric surfaces as triangular meshes. An approach with tessellation performed on the fly for each frame would produce much better results but it may consume more time in the conversation to triangular model for every frame.

The second way to render a parametric patch is by direct ray tracing. Direct ray tracing does not approximate the curve and hence surfaces appear exactly curved. With direct ray tracing, one can generate the exact hit point, calculate the surface normal at the hit points and perform secondary effects like shadows, reflection and refraction. Ray Tracing algorithm is more complex than the tessellation approach and hence takes more time in comparison. However, with the advancements in acceleration structures and increasing compute capability, interactive ray tracing is now possible.

Several techniques have been developed to ray trace parametric patches. There are iterative techniques like Newton iteration, where the initial guess of the parameter value is improved in every iteration. The method works in cases where initial guess is sufficiently closer to root. The initial guess can be generated from subdividing patches into smaller, almost flat surfaces and then generating the initial guess from bounding box intersections of these subpatches. Algebraic solutions have also been proposed, where the intersection problem is converted to root finding of a higher degree polynomial.

The very first step of a working ray tracing application is the construction of an acceleration structure. An acceleration structure helps eliminate the rays which are guaranteed not to intersect with the scene. Acceleration structures work by dividing the scene into smaller boxes, each containing very few surfaces. Thus, the ray is now required to first perform intersection test with these boxes and evaluate the actual intersection with the surface patches only in case of intersection with the box comes out to be true. Since ray-patch intersection test is expensive compared to ray-triangle intersection test, acceleration structure plays a major role in the optimized working of a ray tracing parametric surface application.

1.3.1 Acceleration Structures

An acceleration structure can be classified based on how it partitions the screen space. These structures can basically be classified into 3 broad categories.

Grids are the most basic forms of acceleration structures. A grid structure uniformly divides the scene into fixed size spatial partitions. Grids are very fast to build and hence are well suited for dynamic scenes, which require reconstruction of acceleration structure for each frame. Simple grid structure divides the scene space into fixed size cells and hence traversal might turn out slower for complex scenes.

KD-trees are more complex than grids and divide the scene such that the overall intersection tests are minimized. Thus, the reduced traversal cost of KD-trees makes overall ray tracing algorithm faster. KD-tree makes use of Surface Area Heuristics (SAH) to partition the scene such that the overall potentially intersecting object list is minimal. Intersection costs are calculated for all possible partitions and the

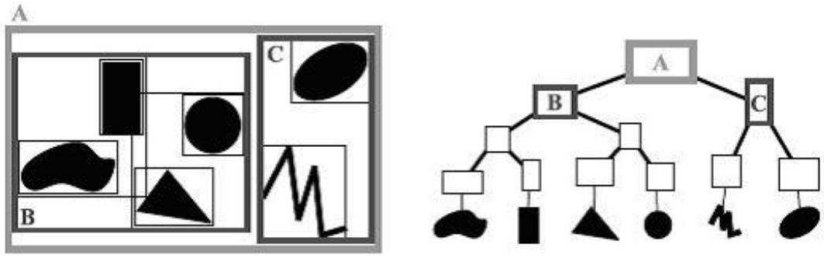


Figure 1.4: Bounding Volume Hierarchy Structure

split is made at the partition having the lowest cost. If the minimum cost turns out to be higher than the determined cost for a partition space, the partition is not subdivided further.

Unlike grid and kdtree structures which partition the scene spatially, a Bounding Volume Hierarchy (BVH) partitions the scene hierarchically. Figure 1.4 shows the BVH structure. We start with the original scene and hierarchically divide it until the final node is classified as a leaf node. The subdivision can use Surface Area Heuristics (SAH) to optimize the structure. The traversal starts at the root node and if the ray intersects the root node, only then its children are traversed. A BVH structure is faster to construct than KDtrees and faster than grid structure in terms of traversal.

For parametric surfaces, BVH has been the most popular choice. The acceleration structure is constructed from the surface patches. A leaf node usually contains one surface patch, which can be subdivided into smaller patches for generating initial values. For a ray-patch intersection test, this initial guess is then sent to the Newton Iteration step, which performs the final test and calculates the hit-point and surface normal.

1.4 Newton Iteration

Newton iteration, better known as Newton-Raphson method or simply Newton's method, is used to iteratively generate better approximations to the roots of a real function. In this method, we start with a initial guess to the root value and in each iteration move closer towards the root until we achieve the desired accuracy. Mathematically, for a one variable function $f(x)$, the Newton iteration step is:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \tag{1.1}$$

where x_0 is the initial guess value for x and x_1 is the new value. Essentially, x_1 is the intersection point of the line tangent to f at x_0 with the x-axis. Figure 1.5 shows some initial steps of the method. Newton's method is able to achieve quadratic convergence.

Newton iteration method only works if we can calculate the derivative of the function. One can approximate the derivative if exact derivative is not available. The other major drawback of Newton iteration is that the method is not guaranteed to converge. If the initial estimate of the root is sufficiently far away from the actual value, the iterations may not converge and takes the value further away from

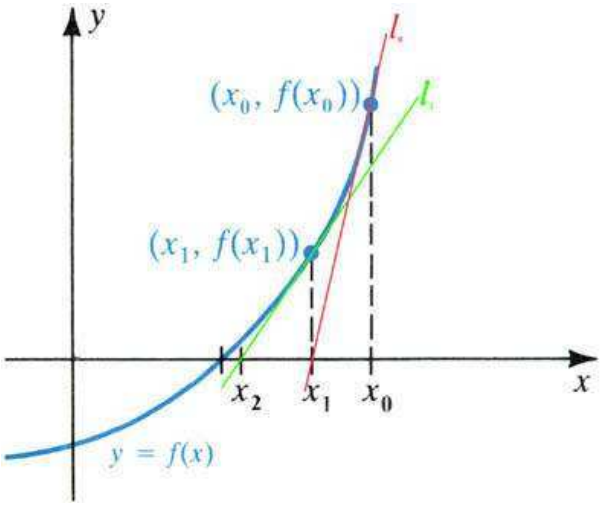


Figure 1.5: Newton Iteration steps (image from rohan.sdsu.edu)

the root. Thus, generating good initial values is of utmost importance. The method may also fail to converge in case the tangent turns out to be parallel or nearly parallel to the x-axis.

Newton's method offer several advantages. The primary advantage is that the method is simple to implement and does not have any complex steps for the iteration stage. This leads to faster computations and hence much faster result generation compared to complex methods with higher degree of convergence. Newton iteration converges quadratically when initial guess is closer to the actual root value.

The Newton iteration can be generalized to multi-variate case. In multi-variate case, we have n algebraic equations in n variables:

$$\begin{aligned} f_1(x_1, \ldots, x_n) &= f_1(X) = 0 \\ &\ldots\ldots \\ f_n(x_1, \ldots, x_n) &= f_n(X) = 0 \end{aligned}$$

where $X = [x_1, \ldots, x_n]^T$. When we represent the system of equations as $f(X) = 0$, the Newton Iteration formula for multi-variate case becomes:

$$X = X - J_f^{-1}(X) \cdot f(X)$$

where $J_f(X)$ is the Jacobian matrix of $f(X)$:

$$J_f(X) = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \cdots & \cdots & \cdots \\ \frac{\partial f_n}{\partial x_1} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}$$

In case of Ray Tracing Bezier Surfaces, we need to solve for both u and v parameters simultaneously. Hence we use bivariate Newton iteration in this case. The iteration step is defined as:

$$\begin{bmatrix} u_{n+1} \\ v_{n+1} \end{bmatrix} = \begin{bmatrix} u_n \\ v_n \end{bmatrix} - J^{-1} \cdot R,$$

where R is the intersection equation vector in (u,v) and J in the Jacobian matrix of R.

1.5 Global Illumination

Global Illumination refers to a set of algorithms which provide much higher degree of photo-realism to a computer generated image. Global Illumination not only takes into account the light coming directly from the light source (direct illumination), but also considers reflections coming off from all the surfaces in the scene (indirect illumination). As opposed to specular reflection, where an incident ray is only reflected at one angle, global illumination uses diffuse inter-reflection, wherein the incident ray is reflected at many angles to provide much more realism to the image.

The photo-realism of these images comes at a very high computational cost. Each pixel gets sampled more than once and each subsequent intersection spawns more than one ray, thereby considerably increasing the overall complexity. Global Illumination can also be achieved by sampling the pixel for a large number of rays and finally calculating the color value of the pixel, taking into account the color value for each of the samples. Some of the common algorithms used to achieve global illumination effect include radiosity, ray tracing, photon mapping, beam tracing, path tracing, Metropolis Light Transform (MLT), ambient occlusions, etc. These algorithms can also be used together to further improve upon the photorealism of the scenes.

Distributed ray tracing is another global illumination method. Ray tracing traverses rays sampled at regular intervals. It ignores all the information which lies in between these regions. Cook [9] used Monte Carlo based Stochastic Sampling to sample the image at appropriate non-uniformly spaced points. This leads to a noise which looks much more photo-realistic than anti-aliased images. This technique can be used to simulate advanced effects such as motion blur, depth of field, gloss, translucency and penumbrae.

1.6 GPU and Hybrid Computing

Graphics Processing Units (GPUs) were originally designed to accelerate the rendering of image frames onto the display. GPUs perform operations such as transformations, lighting calculations, etc, much faster than the CPUs, providing faster graphics processing speed. During the initial stages, the primary use of GPU was to accelerate 3D computer gaming. This was followed by adding programmable shader models, which could now process each pixel and each vertex, apply texture before rendering on the screen, etc.

GPUs have now evolved into more general parallel computing devices. GPUs are now used for many compute intensive problems, referred as General-purpose computing on GPU (GPGPU). These include image processing, matrix manipulation, speech processing, computer vision, machine learning, sorting, etc.

Modern GPUs have become the power-house for very heavy computational requirements. Nvidia GTX 580 has 512 cores and can deliver a performance of about 1$\dot{5}$ TFLOPS. The newly launched Nvidia GTX 680 consists of 1536 CUDA cores and has a theoretical processing power of up to 3.1 TFLOPS. However, to harness the immense power which the GPUs provide, one needs a careful implementation of the parallel algorithms which cater to its architecture.

Along with the GPUs, CPUs have also progressed fast both in the number of CPU cores as well as the computing power of each core. A high end CPU consists of 8 to 16 cores while 2 to 4 cores have become the most common configuration used on most of the systems. While the GPUs are more suitable for parallelizable tasks, CPUs work faster for serialized operations.

Most of the GPGPU efforts perform the compute intensive tasks on the GPU while the CPU sits idle. The computational power offered by a high end CPU is comparable to a high end GPU today. For a common system, where the CPU is almost as powerful as the GPU, using both the GPU and CPU provides considerable speedup.

A hybrid programming model can be basically classified into two categories. The first category is where the operations are divided between the GPU and the CPU and both perform these instructions in parallel and independently of each other. For example, the parallel, compute intensive operations are performed by the GPU, while serialized operations are done on the CPU. The second category is where the data is divided between the CPU and the GPU according to their compute capability. In this case, the GPU and the CPU perform almost similar operations but on different data. The final results are merged either on the CPU or the GPU. A mixture of the two methods can also be used for further optimization.

1.7 Contributions of the thesis

In this dissertation, we present ray tracing of Bezier surfaces using a mixed hierarchy model. The mixed hierarchy model consists of two hierarchical trees. We use Newton iteration to solve the intersection test and develop a hybrid system for the complete algorithm. Using true normals generated after primary pass, we perform secondary shading with shadows, reflection and refraction. We are able to render images with advanced effects like soft shadows, multiple bounce tracing, path tracing, depth of field, etc. By tracing huge amount of rays to generate the advanced effects, we demonstrate the capability of our system. The implementation has been optimized for both the architectures.

The main contributions of the thesis are as follows:

1. In this thesis, we present a new acceleration structure to ray trace parametric surfaces, called the mixed hierarchy structure, which performs optimally on the GPU as well as on the CPU.

The proposed acceleration structure has low memory requirements and is much faster in terms of traversal. In our approach, we build a second level hierarchy tree, below the main BVH tree. This tree not only reduces the overall potential ray patch intersection to be tested but also leads to better initial values and much less traversal times in comparison to previous work. The proposed structure maps well to the parallel GPU architecture. We are able to significantly outperform previous implementations and our own implementation of several other methods to ray trace Bezier surfaces.

2. We implement the Newton's method effectively, both on the GPU as well on the CPU, taking into consideration the strengths of both and finally combine it to produce the hybrid implementation. Even though the functions performed by the GPU and the CPU are the same, the implementations differ so as to optimize the operation according to their strengths. We achieve significant speedup over the GPU implementation. We are able to ray trace Bezier surfaces at more than interactive rates using the proposed mixed hierarchy structure and the hybrid implementation. Even for very large models, we are able to render the image at about 13 frames per second.

3. We also perform secondary shading effects including shadows, reflection, refraction, multiple-bounce ray tracing, soft shadows, etc. We test out our algorithm for massive ray list by performing advanced global illumination effects for parametric surfaces. Global Illumination effects have not been reported for parametric surfaces and we are the first to show the results. These images show higher degree of realism and are much faster to compute using the mixed hierarchy approach we present. We demonstrate the capability of the algorithm by performing these advanced effects. Secondary effects and advanced global illumination effects have not been investigated for Bezier surfaces.

4. The proposed model is suitable for parallel architecture and is expected to be even faster for multi-GPU setting as well as for future hardware. We start with a ray list and divide it among the available devices based on their compute capabilities. Secondary ray list is processed in the same way, structure of the algorithm remaining the same. We find that time per ray depends on the model as well coherence among rays. The difference in rendering times for global illumination images clearly highlights that ray coherence is an important factor when ray tracing Bezier surfaces.

Chapter 2

Background and Related Work

Computer Graphics is basically the process of converting a 3D scene into a 2D image. Graphics came into existence when the computer world moved on from punching cards towards CRTs as output device and lighting pen as input device. In 1963, Sutherland [47] presented his paper sketchpad : a man machine graphical communication system, which is marked by many as the origin of computer graphics. Sketchpad allowed one to draw simple shapes on a vector graphics display monitor with a light pen input device. Many of the modern computer graphics applications can be related back to Sutherland's sketchpad.

This was followed by Jack Bresenham's work on rasterization. He showed how to draw lines and circles on a raster device. In the 1960s, Steve Coon developed the Coons patches [2], which are said to be the beginning of the parametric surfaces and developed Computer Aided geometric design concepts. It was during this time that the earlier work done by Pierre Bezier on parametric curves and surfaces came into light.

Shading was discovered by Gouraud and Phong at the University of Utah. Gouraud shading [15] is a method to produce continuous shading of surfaces comprising of polygon meshes. This is done by calculating the lighting at the corners of a triangle and this information is then linearly interpolated for all pixels of the triangle to produce a smooth shading effect for the entire surface. Phong Shading [38] produces highlights which are much less dependent on the underlying polygons. However, it requires more calculation, which includes interpolation of the surface normal and the evaluation of the intensity function for each pixel. Phong also developed a reflection model, which takes local illumination into account. According to this model, the light reflected by a surface is a combination of diffuse reflection as well as specular reflection, which are produced by shiny objects. The model also takes into consideration the ambient lighting.

All these discoveries advanced computer graphics and lead to the development of rendering algorithms such as ray tracing to bring photo-realism to the realm of Computer Generated Imagery (CGI). CGI was first used in the science fiction movie Westworld in 1973. This was followed by the use of 3D imagery in Futureworld and the highly acclaimed movie of its time, Star Wars. Modern movie

makers and game makers use advanced rendering techniques and physics engines to produce highly photorealistic animations.

2.1 Ray Tracing

Appel [3] presented the first ray casting algorithm used for rendering. The idea was to shoot rays from the eye or camera center into the scene and find the closest object which the ray would intersect. A ray was shot through each pixel and the color of the object intersected by the ray became the color of the pixel. Shading algorithms, which were used in graphics, were used to determine the color returned by the ray. The advantage of ray casting is that intersection with non-planar objects, which earlier was not possible with scanline rendering, now became possible.

The first work on ray tracing was done by Turner Whitted in 1980 [58]. While earlier ray casting approaches stopped at the first intersection, Whitted proposed a new recursive technique to take secondary effects also into consideration. Thus, secondary effects like reflection, refraction and shadows were also now included and this provided much more realism to the generated images. This has formed the basis for all the work following it on ray tracing.

The next step for ray tracing was acceleration of the image generation process. This was accomplished by the use of acceleration structure, which reduced the total number of intersection tests to be performed. Acceleration structures along with advancement in hardware made it possible to perform ray tracing at interactive rates. While the earlier form of ray tracers usually dealt with static scenes, this gradually changed to ray tracing scenes of dynamic nature. This lead to major works done in optimizing the construction of acceleration structures.

Grid data structure is simple and fast to construct. It partitions the space into fixed size partitions to reduce the intersection tests. Purcell et al. [40] use CPU for constructing the structure once and then store it on GPU for further traversal. They were the first to implement a full ray tracing application for static scenes entirely on the GPU. Ize et al. [20] presented techniques on improving the overall quality of the structure. Patidar et al. [36] divide the scene into discrete slabs according to the camera view direction to utilize the spatial coherence of ray-triangle intersection. Guntury et al. [17] developed perspective grids, which were constructed both for primary as well as shadow rays to utilize upon the coherence.

KD-Tree structure are more complex to build and hence take more construction time compared to grid structure. However, the traversal time is less for KD-trees. They were earlier considered to be suitable for static scenes alone because of the large construction time [46]. Zhou et al. [60] performed construction as well as traversal of the KD-trees on the GPU. The work was further extended by Hou et al. [19]. They used better memory allocation techniques to render very large models as well.

Bounding Volume hierarchy takes less time for construction than KD-tree. The traversal time for BVH is also less than grid structures and many times at par with KD-trees. Wald et al. [54] use several of the heuristics used for KD-tree construction and use them in faster construction of BVH. They report

that using their technique, BVH can be constructed up to 10 times faster than KD-Tree construction. Lauterbach et al. [28] improve upon this. They used Morton curve to optimize the construction of BVH on GPU and also to optimize the traversal time. They call this algorithm Linear Bounding Volume Hierarchy (LBVH) algorithm. Pantaleoni and Luebke [35] improve upon the LBVH model in computational and memory efficiency by using a hierarchical approach. They call their algorithm Hierarchical LBVH (HLBVH).

Interactive ray tracing was investigated by Muss et al. [32]. Parker et al. [45] demonstrated it using specially designed hardware to perform the task for almost all kinds of geometry.

2.2 Rendering Parametric Surfaces

The ray-patch intersection algorithms can be classified into four basic categories. These include subdivision, numerical methods, algebraic methods and Bezier clipping.

Recursive subdivision of the patch in the parameter space followed by tessellation has been used for fast rendering of bicubic patches. These algorithms make use of the convex hull property of Bezier surface. If a ray does not intersect the convex hull of a parametric surface, it does not intersect the surface. Whitted [58] used bounding spheres to bound a surface. If a ray intersects a bounding sphere, the surface is subdivided using the method described by Catmull and Clark [7] and bounding spheres are produced for each surface. Subdivision is performed until ray intersects no bounding sphere or the bounding sphere is smaller than a predefined value. While Whitted performs in 3 dimension, Woodward [59] operates in two dimension. They perform intersection in rays orthogonal viewing coordinate system. Recent work in this area was done by Benthin et al. [5]. They achieved about 5 fps using such a method on a high-end processor. Real-time rendering using view-dependent, GPU-assisted tessellation was achieved recently. Patney and Owens used a Reyes-style adaptive surface subdivision exploiting the data-parallelism of the GPU [37]. Eisenacher et al. use a view dependent adaptive subdivision and proposed error estimates for quality rendering [12]. These methods are the fastest to render Bezier surfaces but cannot produce exact surface normals or accurate secondary rays.

Several numerical techniques have been developed to compute the exact point of intersection to render Bezier bicubic patches. Toth used the multivariate Newton iteration using interval arithmetic to solve for ray-patch intersection [50]. The algorithm is able to render parametric surfaces where the bound on the surface and its first derivative is computable. Sweeney and Bartels [48] continuously refine the control point matrix using Oslo's algorithm until the generated geometry closely resembles the surface. The generated mesh is then intersected with the ray to generate initial values, which are later used in the Newton iteration stage. Joy and Bhetanabhotla [22] make use of quasi-newton iteration to solve for ray-patch intersection. They utilize ray coherence and use the solution of neighboring rays to compute the initial value. Martin et al. [31] subdivide each surface based on the degree of curvature and construct a BVH from the axis aligned bounding boxes of these subpatches. Ray Tracing of Parametric Surfaces was demonstrated by Geimer and Abert [13, 1]. They used Newton iteration with

patch subdivision for good initial guesses. They were able to render teapot model at about 6 fps on the PowerMac G5 2GHz processor with 512^2 image resolution for Bezier patches. GPU based Tessellation of parametric surfaces was proposed by Guthe et al. [18]. Ray Casting of trimmed NURBS Surfaces using a combination of Newton iteration and Bezier clipping on the GPU was done by Pabst et al. [34]. Our method constructs a BVH for original surfaces, and constructs a subtree generated by subdividing every surface hierarchically along u and v parameter directions. This leads to lesser ray-bounding box tests, provides tighter bounds and better initial values.

Algebraic solutions to the ray intersection problem were proposed by Kajiya [23] and Manocha and Krishnan [30]. Kajiya solved the ray-patch intersection problem without using any subdivision by representing a ray using two orthogonal planes. He used Laguerre's method to solve the resulting 18-degree polynomial equation. Manocha and Krishnan used an eigenvalue-based method to solve for the intersections. They find the polynomial roots by using eigenvalues of its characteristic matrix.

Nishita et al. presented Bezier Clipping [33] that exploits the convex hull property of the Bezier surfaces. By iteratively cutting down on parts of the surface which are known to not have an intersection with the ray, they were able to isolate it to a small region where intersection occurs. This was later improved by Campagna et al. 1997 [6] and Wang et al. [56]. A hardware based design of a pipelined architecture for this was proposed by Lewis et al. [29]. Efremov et al. [11] improve upon the original algorithm to make it robust and numerically more stable for NURBS surfaces.

2.2.1 Ray Tracing Bezier Patches on GPU

Ray Tracing of Bezier Surfaces on the GPU was done recently by Lahabar [27]. She used Kajiya's method to find ray-patch intersections on the GPU. A stack-less BVH is constructed from the axis aligned bounding boxes of the patches. The entire BVH is stored in depth first format along with skip pointers [49]. Traversal of the BVH is done on the GPU in parallel. Each thread gets mapped to one ray. Each thread evaluates the intersection of a ray with the bounding boxes of the BVH, starting from the root node. The traversal step produces a list of potential ray-patch intersections.

A ray is represented as the intersection of two planes. This leads to formation of two bicubic equations in u and v parameters. A resultant polynomial of 18 degree is formed from the two bicubic equations formulated earlier. Each potential ray-patch intersection gets mapped to one GPU thread. Every thread first computes the two bicubic equations in parallel.

Next the resultant polynomial is calculated for each potential ray patch intersection. This step is optimized for the GPU by choosing the grid size and number of threads per block accordingly. The resulting 18 degree polynomial is stored in the global memory in column format, with the coefficient of the highest order term stored at the beginning and the rest following.

The resultant polynomial is solved using Laguerre method. Each thread gets mapped to one ray-patch intersection. The roots are initialized with $(\frac{1}{\sqrt{2}} + i\frac{1}{\sqrt{2}})$. Every thread computes the 18 roots of the polynomial. Roots are found one at a time using Laguerre's method. Real roots result in the polynomial

being deflated by a monomial whereas complex roots result in deflation by a real quadratic, as roots always appear in conjugate pairs. Consequently, the polynomial stays real at all times.

All the roots which lie between $[0, 1]$ are stored in the memory. These roots are then substituted in the bicubic equations formed earlier and the second parameter is calculated by finding the GCD of the two cubic polynomials. After both u, v parameters have been calculated, lying between $[0, 1]$, hit point calculation is performed. The closest hit point and the surface normal at the hit point is stored in memory. This is later used in ray tracing secondary rays.

Lahabar was able to render the teapot model for 512×512 resolution at 14 fps for primary rays alone. Kajiya's algorithm is very complex and computationally expensive. The algorithm requires double precision arithmetic to perform the intersection test, which leads to slower rendering times. The BVH structure being constructed from the Bezier patch data is not sufficient and leads to higher number of ray patch intersection tests. Our mixed hierarchy model leads to tighter bounds, reducing the overall number of patches to be tested. We use Newton iteration as the intersection test. The method is simple and makes use of single precision arithmetic, which leads to faster rendering times as compared to Kajiya's method.

2.3 Global Illumination

In the mid 1980s, computer graphics got divided into two branches, both of which thrived to achieve photo-realism in computer generated imagery. The first approach was ray tracing, proposed by Turner Whitted [58]. The second was the radiosity method, proposed by Goral et al. [14] which was inferred from the heat transfer researches.

Cook et al. [10] first explored the area of distributed ray tracing. They explained how distributing the rays can generate photorealistic effects like glossy reflections, translucency, penumbras, depth of field and motion blur. Ray Tracing traverses rays sampled at regular intervals. Thus, it ignores all the information which lies in between these regions. Cook [9] used Monte Carlo based Stochastic Sampling to sample the image at appropriate non-uniformly spaced points. This leads to a noise which looks much more photo-realistic than anti-aliased images. We use this idea to incorporate in our framework for producing Global Illumination while ray tracing parametric surfaces.

Kajiya [24] formed the rendering equation, which is the basis for all global illumination methods. Kajiya also introduced Monte Carlo algorithm to solve for path tracing. The idea is to sample all the light energy reaching a pixel from all path sources leading to all light sources. While Monte Carlo based algorithms are capable of handling most of the general lighting effects, they are often slow to converge.

Radiosity was introduced by Goral et al. [14]. The algorithm works by splitting up the geometry into a series of subpatches and then using linear equations to calculate how illumination travels from a light source and across these patches. Arvo [4] worked upon diffuse reflection of indirect light to achieve global illumination. The algorithm traces rays from point light sources to objects. It works in two passes. The first pass generates rays from light source to object, assigning each ray with some

energy. This step produces illumination information, which is called Illumination map. The second pass performs normal ray tracing. Bilinear interpolation is performed on the illumination map values of nearby points to generate the ambient component in the illumination model.

The two methods were then combined, each method being used in the areas where they work best, to produce even higher realism [55, 44, 42]. The model was further improved by [8, 61]. Since pure radiosity methods suffer from high level of discretization, direct and indirect lighting components are separated to achieve better results [41, 57, 42]. Shirley et al. [42] perform separate calculations for hard lighting and soft lighting. Hard light contains the details, such as hard edge of a shadow or caustic, while soft lighting refers to the indirect diffuse lighting. They compute hard lighting using illumination ray tracing along with normal view ray tracing, while soft lighting is computed using radiosity.

Bidirectional Path Tracing was proposed by Lafortune and Willems [25]. It was further worked upon by them [26] and by Veach and Guibas [51, 52]. Bidirectional path tracing generates two subpaths, one starting at the camera center and other starting at a light source. It then considers all paths formed by the joining every prefix of one subpath with every suffix of the other. Much of the work henceforth has been done on minimizing the variance by applying different importance sampling techniques.

Photon mapping was invented by Jensen [21]. It works in two passes. The first pass emits a user specified amount of photons to be bounced in the scene and then each bounce is stored in a data structure called photon map. The second pass samples each pixel in the scene and calculates the contribution of each photon hit to the pixel's color and illumination.

Veach and Guibas [53] presented Metropolis Light Transport (MLT) to solve the light transport problem. The algorithm works by first constructing paths from the camera center to the light source using bidirectional path tracing and then modifies them to generate new paths. The modification is done according to the impact they make towards the ideal image. The algorithm uses little storage and is much more efficient than the previous approaches.

We demonstrate the capability of our algorithm by performing these advanced effects. We are able render scenes with global illumination effects like path tracing, depth of field, ambient occlusion, motion blur and glossy surface. We take about 165 seconds to path trace a bigguy in a box scene with 1000 samples per pixel at 512×512 resolution.

Chapter 3

Ray Tracing using a Mixed Hierarchy Model

Ray tracing is an embarrassingly parallel operation. Each ray operates independently of other rays. This property makes ray tracing algorithms suitable for the parallel architecture. With higher resolutions and large data sets, naive ray tracing becomes computationally expensive. We need to reduce the overall intersection tests in order to speed up the process. The need to reduce the intersections is greater with Bezier patches as the ray-patch intersection test is more complex and computationally expensive as compared to ray-triangle intersection test. We also need better initialization values when working with parametric patches. We propose a new acceleration structure, a mixed hierarchy structure, which consists of two level of hierarchies. The proposed structure leads to tighter bounds, eliminating more non-intersecting patches and also provides better initialization values, which speeds up the Newton iteration process. We make use of the mixed hierarchy model to ray trace Bezier surfaces using Kajiya's method and Newton iteration method.

3.1 Ray Patch Intersections

A Bezier bicubic patch can be described in the matrix form as

$$Q(u,v) = [U][M][P][M]^T[V]^T, \tag{3.1}$$

where $[U] = [u^3u^2u1]$, $[V] = [v^3v^2v1]$ with u, v being the parameters in the range $[0, 1]$. The Control Point Matrix P and the Bezier basis matrix are given by

$$P = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix}$$

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

[M] is the Bezier Basis Matrix and [P] contains the 16 control points that define a patch. Any point on the patch can be evaluated from the above equation by substituting their u, v values. The normal at that point can be obtained as the cross product of the parametric derivatives $\partial Q/\partial u \times \partial Q/\partial v$. The partial derivative with respect to u is given by

$$\partial Q/\partial u = \begin{bmatrix} 3u^2 & 2u & 1 & 0 \end{bmatrix} MPM^t \begin{bmatrix} v^3 \\ v^2 \\ v \\ 1 \end{bmatrix}$$

$\partial Q/\partial v$ can be computed similarly.

A ray is represented as the intersection of two planes $(\hat{n_1}, d_1)$ and $(\hat{n_2}, d_2)$, with normals $\hat{n_1}$ and $\hat{n_2}$ and distances from origin d_1, d_2. The ray patch intersection equation then becomes

$$R(u, v) = \begin{bmatrix} \hat{n_1} \cdot Q(u, v) + d_1 \\ \hat{n_2} \cdot Q(u, v) + d_2 \end{bmatrix}. \tag{3.2}$$

A variety of numerical techniques can be applied to solve the equation. The algorithm to solve the above equation needs to suit the architecture of the GPU, which has a large number of simple cores with a high SIMD width. We use the Newton iteration method, which is highly parallel and simple.

Newton method offers several advantages. The first advantage of Newton method is that it converges quadratically, if the initial guess value is close to the actual value. We can ensure this by subdividing the original patches optimally, to get better initialization values. Secondly, surface derivative exist and can be computed at a cost at par to that of surface evaluation. We can calculate surface evaluation and partial derivatives together, which leads to lower global memory accesses and hence lower computational times on the GPU.

We start with the initial guesses (u_0, v_0) for the parameters. Each step of the newton iteration can then be written as :

$$\begin{bmatrix} u_{n+1} \\ v_{n+1} \end{bmatrix} = \begin{bmatrix} u_n \\ v_n \end{bmatrix} - J^{-1} R(u_n, v_n), \tag{3.3}$$

where J is the Jacobian matrix of R, given by

$$J = \begin{bmatrix} \hat{n_1} \cdot Q_u(u, v) & \hat{n_1} \cdot Q_v(u, v) \\ \hat{n_2} \cdot Q_u(u, v) & \hat{n_2} \cdot Q_v(u, v), \end{bmatrix} \tag{3.4}$$

where Q_u and Q_v denote the partial derivative with respect to the respective parameters. The inverse of the Jacobian Matrix is calculated by

$$J^{-1} = \frac{adj(J)}{det(j)}$$

The adjoint of J can be found by calculating the cofactor matrix and then transposing it.

$$C = \begin{pmatrix} J_{11} & -J_{12} \\ -J_{21} & J_{22} \end{pmatrix}$$

Thus, adjoint adj(J) becomes

$$adj(J) = \begin{pmatrix} J_{22} & -J_{12} \\ -J_{21} & J_{11} \end{pmatrix}$$

We terminate the iterations when the value of R falls below a threshold ϵ or if it starts to diverge. Experimentally, we also found that a maximum iteration count of 16 combined with our initialization gives high quality and fast results. The method can be extended to NURBS surfaces and other of parametric surfaces easily, as we follow very generic methods.

3.2 Computational Model and Platform

The CPU and the GPU architectures have improved significantly in the past years. In this paper, we explore the best way to to exploit this computational power for ray tracing parametric surfaces. The GPU is our primary computation platform due to its higher total computation power. Adapting ray tracing to the restricted architecture of the GPU involves optimizing the data layout and using efficient operations. The relatively large SIMD width of 32 of today's Nvidia GPUs places high premium on thread divergence in operation and memory access. The CPU has also become more capable. We must get best performance for the algorithm by exploiting the CPU also, which otherwise would stay mostly idle. Since both the CPU and GPU architecture function differently, we try to optimize both the parts to gain maximum benefits from each of them.

Algorithm 1 describes our overall approach. We generate the ray lists for each frame at the start. These are processed in a data-parallel manner using a thread per ray in the traversal phase. This generates potential ray-patch intersections list by traversing the BVH in parallel for each ray (Step 5). This list is further refined by traversing through the subpatch tree, generating initial parameter values as well. The refined potential (ray, patch) intersections list is processed in parallel using Newton's method using the initial values found in the previous step. Lists for shadow rays, secondary rays, and intersection points are generated by this process. These lists can further be used for traversal for further bounces. The shading information is accumulated and applied in the reverse direction. We have implemented each step on the CPU and the GPU. These are combined by dividing the work between them to get the best overall performance.

The data structure is maintained in a GPU friendly fashion to improve the performance. The main acceleration structure used in the algorithm is mixed hierarchy structure, comprising of two levels of

Algorithm 1 Overview of hybrid ray tracing

Preprocessing on the CPU:

1: Create a BVH of the patches using surface area heuristic. Store patches in depth first order with skip pointers.
2: Create subpatch tree of AABB for all patches.

Repeat every frame for each ray in parallel (GPU or CPU)

3: Mark back-facing patches in parallel
4: Form the Ray List, initially one for each window pixel
5: Compute two planes in the world space for each ray in the list
6: Traverse the BVH for each ray in parallel
7: Traverse the subpatch tree to reduce the number of potential intersections and to get the initial guess
8: Generate the (ray,patch) intersection list
9: Solve for (u, v) using the Newton Iteration in the range $[0, 1]$ for each (ray,patch) pair in the list in parallel

For each ray in parallel (GPU or CPU) :

10: Find the intersection point. Find normal **n**.
11: Generate secondary ray lists for shadow, reflection and refraction. Repeat steps 5 to 10 for the new ray lists generated.
12: Compute the final color by combining information returned by all secondary rays with shading information at the point.

Bounding Volume Hierarchy. BVH is most commonly represented as binary tree model on the CPU. The BVH tree is stored in a depth first on the GPU layout by making use of the skip offsets [49], which stores the index of the next node to traverse in case of a miss. Thus every ray in the ray list traverses through the tree to generate a list of potential Ray-patch intersections. In order to achieve coherency, we traverse a bundle of rays together leading to further speed up. The list of generated (ray, patch) intersections is then processed strictly in a data parallel manner, first to further reduce the list using tighter bounds and generate initial values and then to retrieve the final hit point for each ray. We use a work-division approach to combine CPU and GPU for processing, with a fraction of rays handled by the CPU based on its relative compute-power. We use OpenMP to spawn threads for rays. We infer that the hybrid model benefits most when the CPU performance is competitive to the GPU overall.

3.3 Hybrid Ray Tracing Algorithm

Figure 3.1 describes the overall hybrid algorithm. We begin with a list of rays which potentially intersects the objects. The list is divided between the CPU and the GPU based of the relative compute capabilities. Our algorithm has two stages: traversal of the mixed hierarchy structure and Newton iteration. Each stage is posed as a manipulation of the input data structures to generate the output structures. The GPU performs these stages using three kernels in parallel: *traverse, recheck* and *Newton*. The CPU performs this in a single step for each ray using a core, with early ray termination. Ray coherence af-

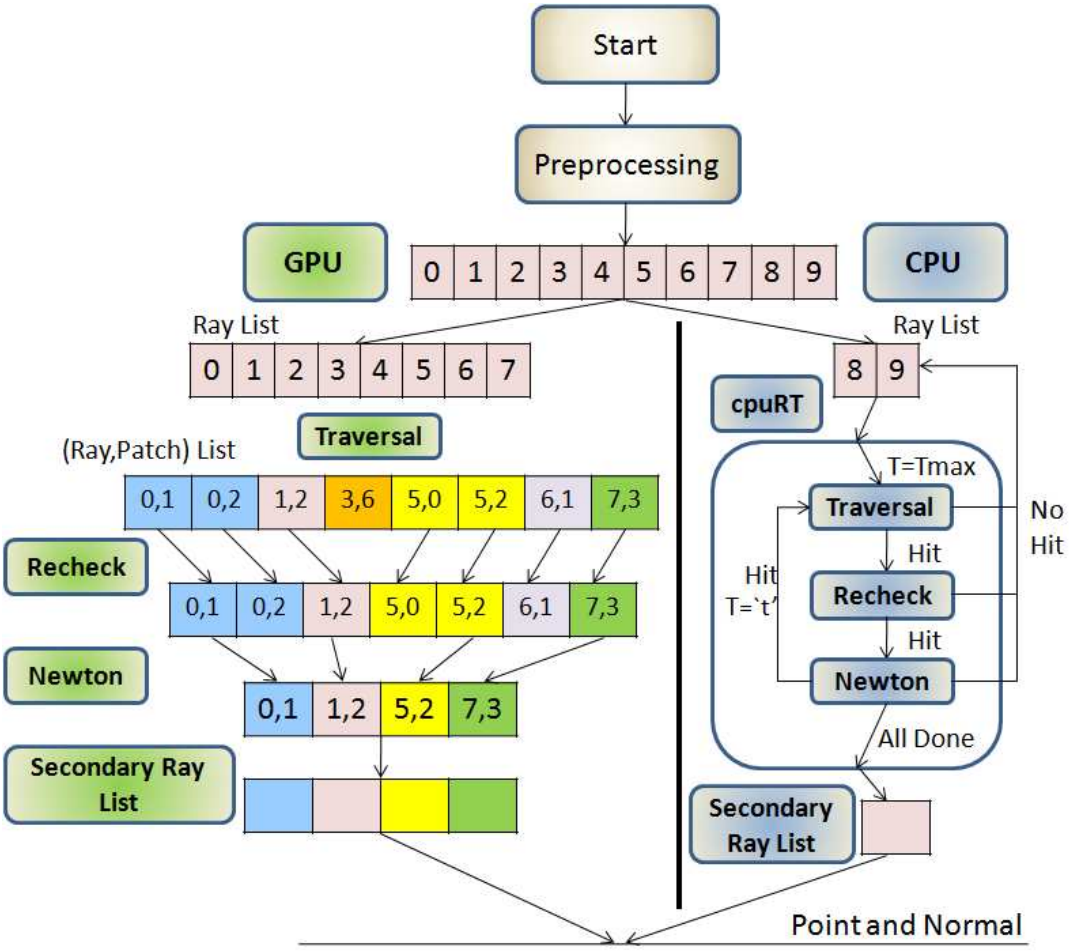


Figure 3.1: Overview of our hybrid ray tracing system

fects only the mixed hierarchy traversal step on the GPU as a bundle of rays may follow similar paths down the BVH. Newton iteration treats each potential intersection independently. Coherence at the data structure level is obtained using memory layout that is efficient for GPU threads. We obtain similar tracing performance for primary and secondary rays. Newton iteration is performed using double precision numbers to avoid artifacts, which doubles the memory requirements and computation time on the GPU. We describe the individual components in detail in the following subsections.

3.4 Mixed Hierarchy Representation

We use a mixed hierarchy as the acceleration structure. It has two parts: a conventional bounding volume hierarchy above individual patches of the model and a sub-patch tree for each patch below it, as shown in Figure 3.2. We create a BVH of axis-aligned bounding boxes of the patches in the object space first using the approach by Gunther et al. that approximates the surface area heuristic (SAH) using

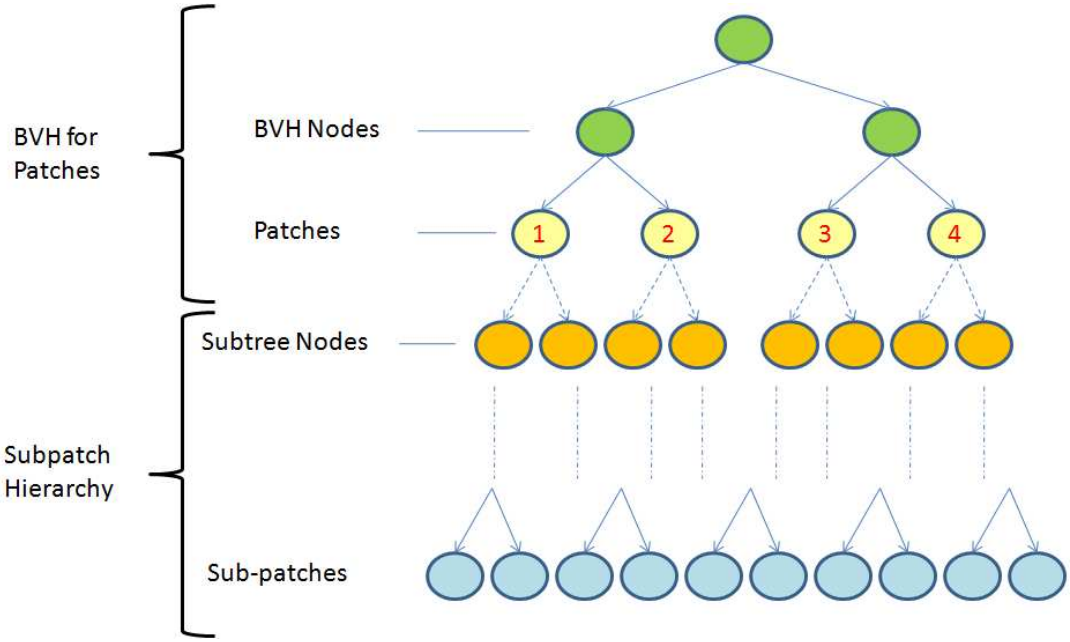


Figure 3.2: Mixed hierarchy structure combines a BVH with a subpatch tree

streamed binning of centroids [16]. The BVH is stored in memory in the depth first layout with skip offsets [49]. A BVH node is represented using 56 bytes: 6 double values for the axis aligned bounding box, 1 integer for the patch id (with -1 for non-leaf nodes), and 1 integer to store the skip pointer. This representation of the BVH fits the SIMD model of the GPU. Each ray can be traversed independently down the BVH to identify the patches it potentially intersects.

Typical objects built using Bezier patches use large patches whose bounding boxes are not very tight. The above process will generate a large number of potential ray-patch intersections for them. We can reduce this number by subdividing the patches to smaller subpatches, possibly based on a flatness criterion [31]. The BVH gets deeper but the bounding boxes are tighter. This will, however, result in excessive memory usage as each subpatch needs to have independent control points. This can be a limiting factor on the GPU due to their limited memory space.

We create a tree of subpatches for each patch to reduce memory requirement while maintaining tight bounding boxes. A binary tree is created for each patch by alternately subdividing it along the u and v axes using De Castelajau's algorithm. Axis-aligned bounding boxes are formed for each using the convex hull property of the Bezier surfaces. The control points are discarded after the tree is constructed as they are needed only to form the bounding box. Each ray is checked for intersection with these bounding boxes. When a potential intersection is identified, we add the entry to the potential (rayID,patchID) intersection list. Thus, the actual intersection algorithm works on the original patches. This keeps the memory requirements low while increasing efficiency. A subdivision depth of 6 was empirically found sufficient for our scenes. A lower depth lead to bad initial values while a higher depth leads to considerable amount of time spent on finding these values. A fixed depth makes it possible to use the GPU's

shared memory to store the skip pointers and the subpatch numbers for the subtree traversal. Intersection with the bounding box of the leaf subpatches are used to find an initial guess for the iterative step.

The mixed hierarchy simultaneously reduces the number of ray-patch intersections to be checked and keeps the memory requirements reasonable. The average number of patches per ray reduces to 1.2-1.5 when using the subpatch tree compared to about 3.4 when only the patch BVH is used. For the subpatch hierarchy, we tried using a varied depth based on the flatness of the generated subpatches [13]. This requires additional memory to keep track of sub-tree starting index, total number of nodes, and floating point memory to retrieve and store initial values, compared to just 1 integer required to store subpatch number in our method. We cannot also use the shared memory under this arrangement. Thus, flatness-based subdivision performs about 30% slower than the fixed-depth subdivision that we use.

3.5 Generating Rays

Ray Tracing of a frame starts with ray generation. We calculate a projection of the top level BVH box and calculate a bound on the pixels to be ray traced. We generate a ray list for these pixels. Newton method requires each ray to be represented as the intersection of two planes. We use the twin-plane representation for each ray [23]. This is computed in parallel on the GPU alone for each ray. We use M planes for each of the M rows and N planes for each of the N columns. The ray for pixel (i, j) uses the corresponding planes. These planes are evaluated in the world space and are constructed on the GPU using a small kernel of M + N threads.

We adapt the method used by Eisenacher [12] to reduce the number of patches to be checked for primary intersections by eliminating those that face backwards. This is performed by computing the dot product of the direction vector from the origin to the corner control points for each patch with the respective normals at these points. If all four dot products are positive, the patch can be discarded as it is backfacing. We perform this for each subpatch during the BVH traversal step, flagging each subpatch as cullable or not. If all the subpatches of a patch are backfacing, then the patch is marked as culled. The flags of the subpatches are propagated up using an AND operation at each parent node. This results in a culling bit for all the nodes of the BVH. This step saves about 33% of the total time for Bigguy/Killeroo scenes. This is the only view dependent step in our ray tracing process. The removal of back facing patches reduces the overall patches to be considered in traversal stage considerably, bringing the overall potential ray-patch intersections down.

3.6 GPU Ray Tracing

Ray Tracing algorithm on the GPU has two phases, the traversal phase and the Newton iteration phase. The GPU traversal comprises of two stages. Stage 1 identifies the ray intersections with the bounding boxes of all nodes of the BVH. This results in a list of potential patches to be marked for intersection with the ray. In Stage 2, the subpatch tree is traversed to remove more patches and to get

the initial value. The BVH is traversed using a packet of 4×4 rays to increase coherency. The complete traversal step generates the potential (ray,patch) intersection list, along with the initial values which are used in the Newton iteration step.

3.6.1 BVH Traversal

A *traverse* kernel performs the Stage 1 calculations with each thread processing a ray. Intersections are recorded by storing the patch ID in a queue for each ray. Node position either gets incremented on intersection or gets updated to the skip pointer value in case of a miss. The number of intersections are saved for each ray. No stack is needed as the BVH is stored in depth first order [39]. Since each ray can have a variable number of intersections, a scan operation identifies the starting index for each ray. These intersections are compacted to a linear array of potential intersections.

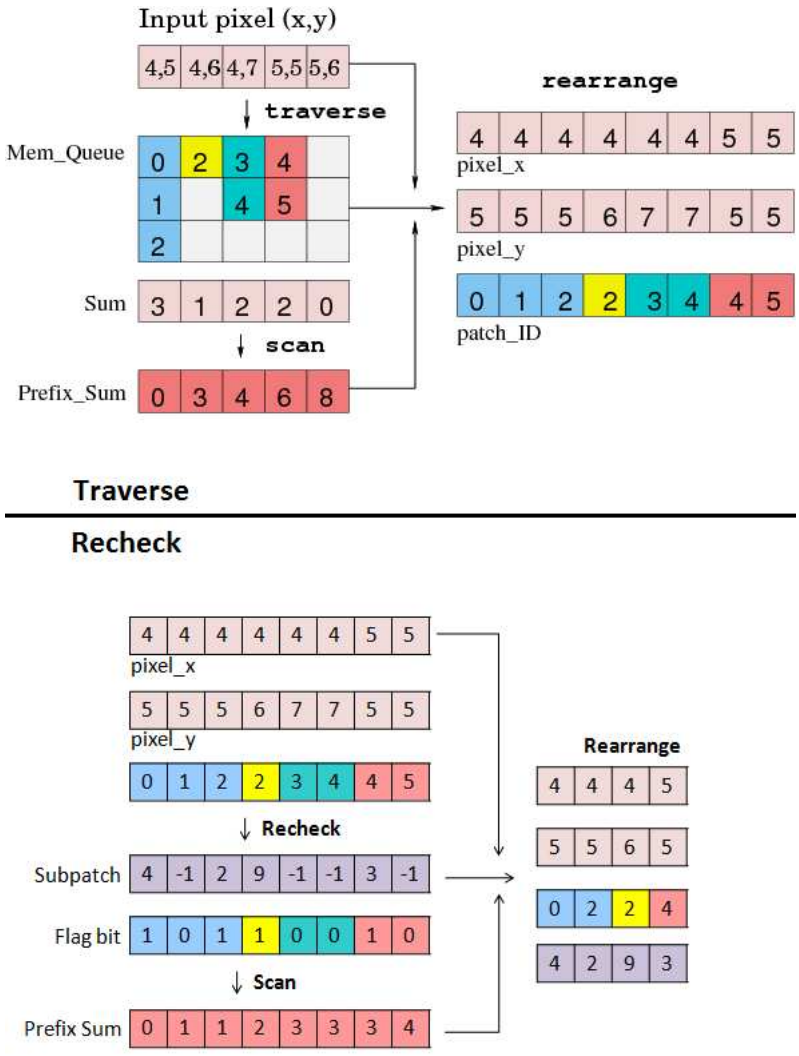


Figure 3.3: Mixed Hierarchy traversal on the GPU

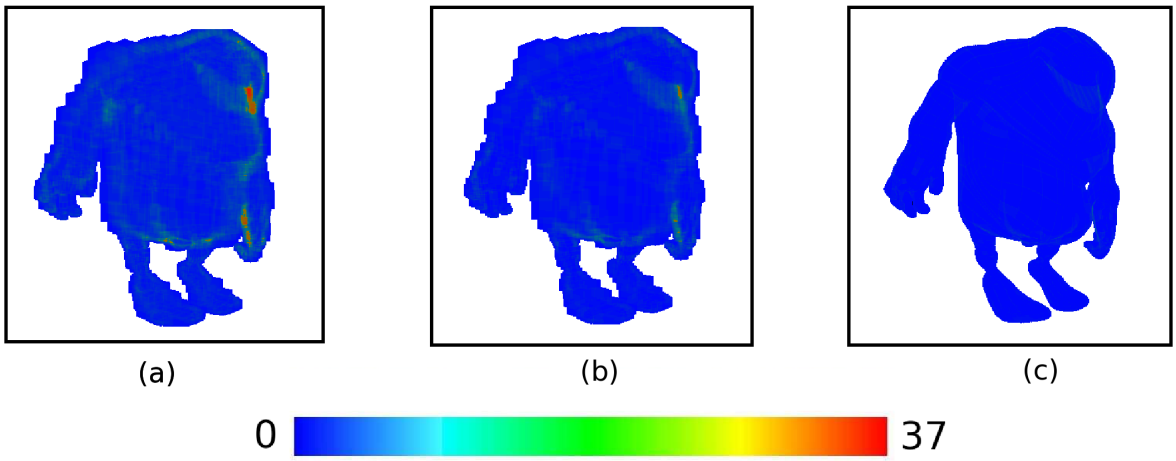


Figure 3.4: Heat map showing number of potential patches for each ray after traversing BVH (a), after applying BFC (b) and after applying BFC + recheck kernel (c).

The traversal step can be performed in two ways. In the first, we assume a maximum depth complexity for the rays and assign a fixed-size blocks to each ray to store the intersections. While this method works a bit faster, it is not scalable to situations when the depth complexity is not known beforehand. We implemented a method to dynamically allocate global memory to handle this problem. Each ray is allocated a small block initially. When the current allocation is used up, the ray is given another such block from a pool of global memory. Such allocation requires atomic operations on an index to the global memory pool to maintain consistency across multiple rays that may ask for a new block simultaneously. The number of ray-patch intersections and the list of blocks for each ray are kept track of. These are used in a *compaction* step to create a single ray-patch intersection list. Atomic operations are fast on latest GPUs and do not result in significant performance degradation, as the probability of atomic clashes is low in practice. In practice, the performance penalty of the dynamic allocation of space is less than one percent.

3.6.2 Subpatch Tree traversal

A *recheck* kernel performs Stage 2 by processing each ray-patch intersection in parallel. This leads to more parallelism and better performance. The traversal starts at the root of the subpatch tree. Upon intersection, the node value gets incremented to reach the next node in the traversal step or is updated to the value of its skip pointer. The minimum t value for distance along the ray also gets updated for a hit. Each node intersection test also returns a *min-t* value for that node. The t value, updated earlier, is used at later stages as a bound to prevent traversing nodes with greater *min-t* values, thus also avoiding its child nodes. The early termination done this way leads to an overall speedup of about 10% for this stage. The output of this stage is a list of potential ray-patch intersections and the initial values of u and

v, which is the centroid of the bounding box. The list is compacted using a scan operation to yield the (ray,patch) intersections list actually evaluated.

Recheck kernel leads to significant reduction in the overall potential ray patch intersection. This can be seen from Figure 3.4. We see a decrease in patches marked for intersection per ray as we apply Back Face Culling (BFC) and *recheck* kernel.

Another advantage of using a subpatch tree for each patch is the ability to use the t value along the ray for early termination. The t values of an intersection with a bounding box at the leaf of the subpatch tree can be used as a bound to not process subtree nodes farther than it. This early termination is only for the subtree stage and a false positive here leads to a slightly worse initial guess. This saves about 15-20 intersection tests per subpatch tree. The fixed depth of tree enables it to be stored in the shared memory for the rays to process it using multiple threads. This leads to faster performance on the GPU. Figure 3.3 shows the complete traversal step performed in our algorithm. The *Prefix_Sum* provides the starting index in the new compressed array for each rays.

3.6.3 Kajiya's Algorithm

We use the mixed hierarchy model and Kajiya's method of ray tracing. Kajiya's method is complex in finding ray patch intersection and hence there is a serious need to eliminate most patches before applying the algorithm. Mixed hierarchy proves helpful in this case as it eliminates many of false positive patches generated after only BVH traversal. We improve upon the work done in our group by Lahabar. We improve upon the root finding part of the algorithm.

A *Solve* kernel is used to perform polynomial evaluation to find the root and then deflate the root from the polynomial. This was the most complex step of the algorithm. With this implementation, compute cores of the GPU were underutilized due to high register count. We split the solve kernel into two kernels, *evaluate* and *deflate*. Evaluate kernel performs the root finding step while deflate kernel uses the root computed earlier to deflate the polynomial. This leads to lower register count for both the kernels, thereby improving the occupancy and in turn improve the timing. We iterate over the evaluate and deflate kernels for finding all the roots of the polynomial.

The deflate kernel deflates the polynomial by a monomial in case of real root and by binomial in case of complex root. This leads to divergence between threads, which slows the overall working of the threads on the GPU. We transform the step so as to make it less divergent. This leads to low divergence between threads in same warps leading to more throughput.

The earlier implementation used to initialize roots with $(\frac{1}{\sqrt{2}} + i\frac{1}{\sqrt{2}})$. While this a fairly good initialization value, but considering we are only concerned with real roots lying between $[0, 1]$, there was need for better initialization. We distribute the initial values at equally spaced intervals between 0 and 1 on the real number line. This leads to faster convergence for real roots. This initialization also leads to finding real roots between $[0, 1]$ faster and usually in the earlier iterations for root finding. We experimentally determined that we were able to gather all the real roots by 12 iterations. Since one complex

root finding leads to deflation by binomial, we find 16-18 roots with 12 iterations and all of the real roots lying between 0 and 1.

Using the above steps, we are able to get 5-8 times speed up over previous implementation. However, Kajiya's numerical approach is much complex as compared to other methods and thus performs slower. Also, Kajiya's method requires double precision arithmetic, which is still expensive on the modern day GPUs as compared to single precision. There had been work done on ray tracing parametric patches using Newton iteration by Geimer and Abert [13] and Geimer et al. [1]. The method is much simpler and can be performed using single precision, we would lead to better rendering times on the GPU. The drawback with Newton iteration is that it could lead to inconvergence due to bad initialization values. With the mixed herirarchy approach, we are able to provide better initialization values. We apply Newton iteration with our mixed hierarchy approach to ray trace Bezier surfaces.

3.6.4 Newton Iteration

Newton Iteration is performed in parallel for all potential ray-patch intersections with input as the ray and patch IDs, the initial (u, v) values and the plane equations. Each thread is assigned one (ray,patchid) pair to solve using newton iteration. This leads to significant improvements over the traditional one ray per thread approach. Graphics Hardware benefit much more from having large-scale data to work on parallely than to have lesser data with some form of early termination. The key step in Newton iteration is the evaluation of $Q(u, v)$, $Q_u(u, v)$ and $Q_v(u, v)$. They are evaluated together since global memory accesses are expensive on GPU. Our data layout and operation sequence ensures completely coherent memory access and high performance. We use the fused multiply add (FMA) operations to evaluate of Q, Q_u and Q_v to improve the GPU performance. Since the values generated from the intersections are used to perform secondary effects, with multiple bounces, we use double precision for better accuracy.

3.6.5 CPU Ray Tracing

The CPU algorithm can use early termination at the BVH traversal step. The allocated rays are split across $2c$ threads using OpenMP, where c is the number of CPU cores to yield best experimental performance. The rays assigned to a thread are then traversed sequentially. All stages are combined into a single step on the CPU. For each ray, first BVH traversal is performed. If it gets a hit in the traversal step, the recheck step is performed. If the intersection survives the recheck step, Newton iteration is performed in the same thread to generate the intersections and secondary rays. We also store the distance t along the ray from origin to the hit point. This value is used when the BVH traversal resumes so that any bounding box with a t higher than the *min-t* encountered so far can be terminated. Early termination at this step saves many unnecessary computations, typically leading to about 15% increase in performance in the CPU algorithm. We observe maximum benefit from early ray termination in the *shadow* pass (covered in next chapter), as rays can be terminated at the first hit, regardless of the distance.

3.7 Calculating and Storing Hit Points

Hit-points and partial derivatives are calculated at the end of the Newton iteration. The surface normal is generated by cross-multiplying the two derivatives. Along with hit-point and surface normal, we also calculate the distance value along the ray. We divide the entire potential ray patch intersection lists into segments, consisting of one ray. Thus all potentially intersecting patches for one ray belong to same segment. In an unsigned integer, we use the first 24 bits to store the distance value and the next 8 bits to store the index of a potential ray-patch intersection in its segment. This value is generated for each potential intersection, with maximum value set for non-intersection case and is stored in a new distance array.

Using this information, we perform a segmented inclusive scan on the distance array, with 'min' as the operator. This step outputs the minimum distance value for each segment. Thus we now know the closest hit-point for each ray. The segment index value is extracted out of the minimum distance value and the corresponding hit-point, surface normal and patch intersected information is then compacted for each intersecting ray.

The produced output is then used to calculate shadow ray directions and reflection/refraction ray direction. After performing secondary effects, shading is performed for each ray intersecting a surface.

3.8 Results and analysis

We report ray tracing performance for primary rays for rendering 1024×1024 images of the following models: Teapot (32 patches), Bigguy (3570 patches), and Killeroo (11532 patches). The BVH of the mixed hierarchy is built on the CPU using the streamed binning algorithm [16]. The leaf patches are recursively subdivided 6 times. The experiments used an Intel 2.67GHz Intel Core i7 920 processor with 4 cores. The GPU used is Nvidia GTX580 with 512 cores, on CUDA4.0. Single precision is used in the traversal step while double precision is used during the Newton Iteration step. We used a threshold of 10^{-15} for convergence of root finding and a maximum iteration count of 16, taken from previous publications. This accuracy level is required so as to correctly perform multiple levels of secondary rays. All the figures are rendered at 1024×1024 resolution unless stated otherwise.

Table 3.1 shows the time taken to render different scenes for pure CPU, pure GPU and Hybrid system. We average our measurements for 10-15 render times for each scene. The total time includes the time to construct the planes. The scenes had about 20% shaded pixels. 9 Bigguy has 28% screen coverage. The total traversal times comprising of all stages are reported in the table for CPU, GPU and Hybrid algorithms. We also mention the individual time for traversal and newton stage on the GPU. The time taken for the BVH traversal is about 50% of the total traversal step. Table 3.1 also shows the break-down of time taken during *Traversal* stage and *Newton Iteration* stage.

Model	# Patch	# of ray-patch intersections		Computation time on GPU in milliseconds		Total frame time in milliseconds		
		Total	Per ray	Traversal	Newton	CPU	GPU	Hyb
Teapot	32	126589	1.6	5.53	3.18	74	8.71	8.01
Bigguy	3570	142779	1.49	11.8	2.79	110	14.59	13.18
Killeroo	11532	147116	1.55	19.08	3.3	193	22.38	20.43
2 Killeroos	23064	317494	1.65	35.15	7.14	356	42.29	38.58
9 Bigguys	32130	570136	2.09	64.69	12.36	2092	77.05	75.9

Table 3.1: Rendering times on an Intel i7 920 + Nvidia GTX580 for different models and passes at 1024×1024 screen resolution: primary (P), shadow (S) and reflection (R). Third column gives the total and the fourth column the average number of ray-patch intersections computed. CPU is our implementation for multicore system, GPU refers to GTX 580 timings and Hyb refers to the hybrid timings.

Model	Pure GPU GTX 580 (ms)	Pure GPU Tesla K20c (ms)	Multi-GPU GTX 580 + C2050 (ms)
Teapot	8.71	5.42	5.63
Bigguy	14.59	10.61	9.84
Killeroo	22.38	17.85	15.69
2 Killeroos	42.29	34.19	28.75
9 Bigguys	77.05	58.92	51.12

Table 3.2: Rendering time comparison of Multi-GPU (GTX 580 + C2050) against Pure GPU (GTX 580) for different models

Number of patches influence the mixed hierarchy traversal phase alone. Reducing the number using the additional subpatch tree has a big impact on the overall performance, as a result. The number of ray-patch intersections is about 1.5. We achieve 125 fps for the Teapot model for primary rays.

We are able to achieve about 415 fps on the GPU, about 53 fps for the multi-core mode on an intel i7 920 2.67 GHz and about 15.4 fps on an Intel Core 2 duo 2.20 Ghz system for teapot scene with 512×512 resolution. In comparison, [13] got 6.1 fps on a dual processor PowerMac G5 2GHz processor and hence our method is doing better, even after accounting for the technology difference.

From the table, we can see that our GPU-only tracer is about 7-9 times faster and hybrid is about 9-10 times faster than the optimized multicore version for majority of the cases. An average overall speedup of 5-15% is achieved by the hybrid version over the GPU-only version. The performance of the GPU kernels is limited by the number of registers available as the computation resources of the GPU are kept busy.

In some cases, where either the amount of rays to trace are huge or when there are too many patches to be check for intersection, the GPU memory becomes a bottleneck. In order to overcome this limitation, we modify our algorithm to render the complete screen in parts. Thus, the ray list generated at the beginning is divided equally and each part can either be solved parallely on multiple GPUs or

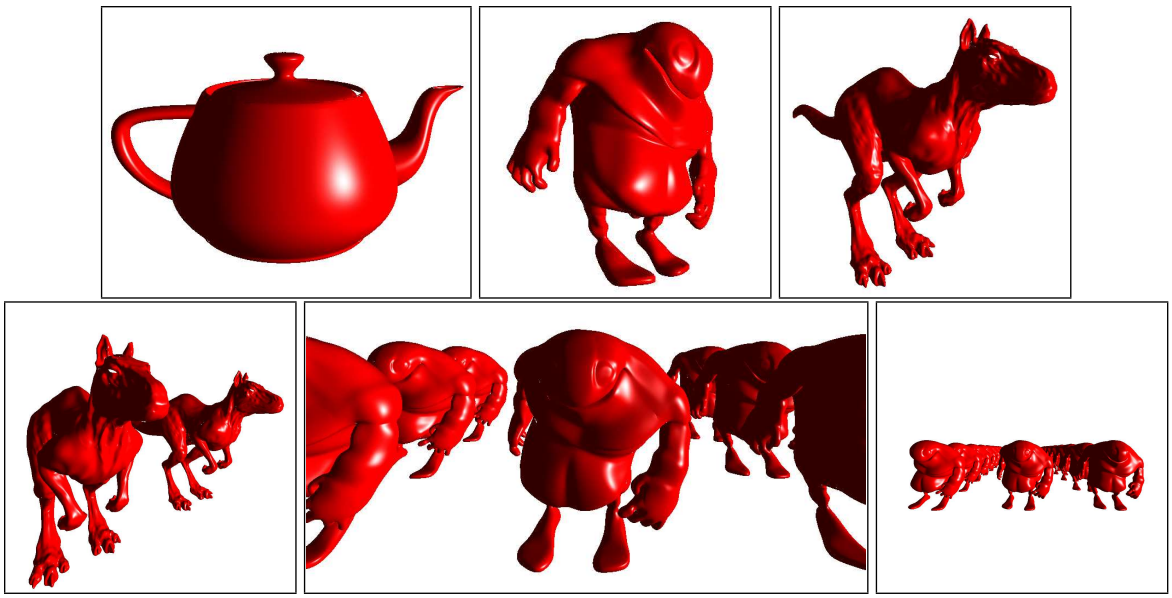


Figure 3.5: Top row, left to right: Views of Teapot, Bigguy and Killeroo. Bottom row, left to right: Views of 2 Killeroos, 9 Bigguys and 24 Bigguys for primary pass only.

Model	K GTX 480 (ms)	IK GTX 480 (ms)	N GTX 480 (ms)
Teapot	71	12.59	1.62
Bigguy	147	18.57	4.43
Killeroo	164	21.26	6.76

Table 3.3: Comparison of different methods. K refers to Kajiya's Method implementation on GPU by Lahabar [27] , IK refers to improved Kajiya's method by us and N refers to Newton Iteration. All the results are taken on GTX 480 for same screen coverage and screen resolution 512×512.

sequentially on a single GPU system. Figure 3.5 (f) shows the image generated of 24 bigguys, rendered at 12 fps. The complete list was divided into 4 parts to be solved iteratively on the GPU.

Figure 3.4 shows the heatmap of the number of patches checked for intersection for each ray, after applying different stages of the algorithm. The first image displays heatmap after applying just the BVH traversal stage, without using back face culling (BFC). Next image displays results after applying back face culling. Finally, the last image shows the results after applying the *recheck* kernel, after BVH traversal with back face culling. We see a significant drop in color value after applying BFC. We see a very close approximation of the exact boundary for the bigguy model in last heatmap image. Thus recheck leads to a reduction in numbers of rays to be checked for intersection and also reduction in the potential patches for each ray.

Figure 3.6 displays the timings of BVH traverse and recheck kernel steps of traversal. The recheck kernel takes more time than traverse when the number of patches are low. As the patches increase,

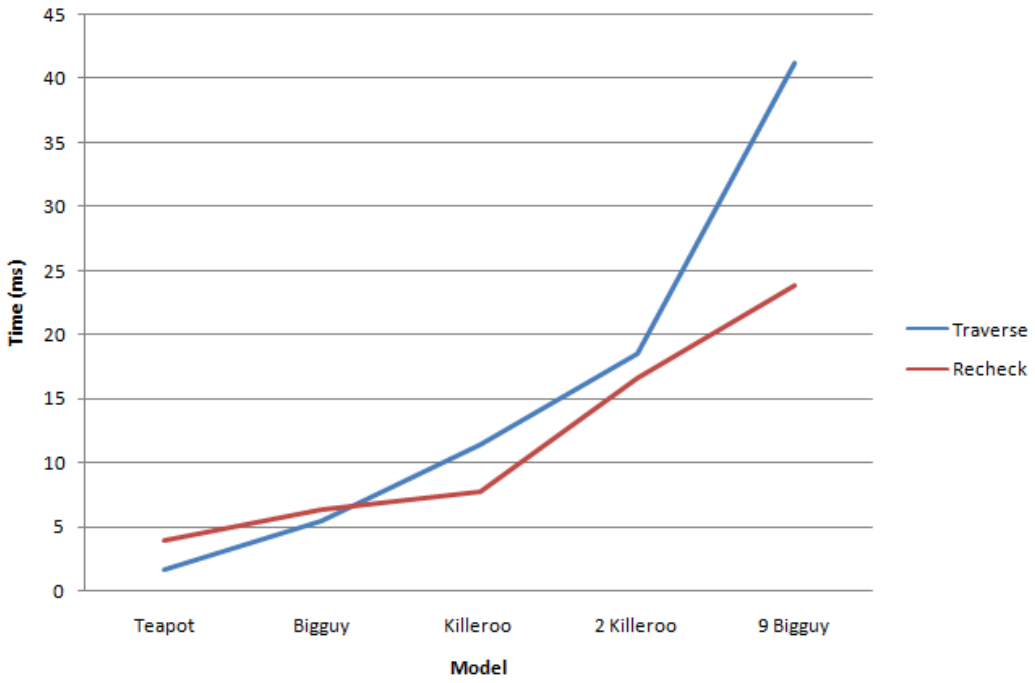


Figure 3.6: Traverse and Recheck Kernel Time for different models.

traverse kernel becomes the most time consuming step of the algorithm. We see a linear increase in time as the number of patches increases in both the kernels. However, the rate of increase is more in case of BVH traverse kernel as compared to recheck kernel. Lesser rate of increase in time for recheck kernel indicates that it benefits more from coherence as compared to traverse kernel.

Figure 3.7 shows the histogram of number of iterations required for each successful intersection to be found in the newton iteration stage. We see that with a better depth at sub-patch level, majority of the intersections require only 3-4 level of iterations. Almost all intersections are found upto 8 iterations.

3.9 Conclusions

We presented a method to ray trace bicubic parametric patches at interactive frame rates using hybrid computing in this paper. We implemented the Newton iteration method effectively on the CPU and the GPU. A mixed hierarchy keeps the number of ray-patch intersections down. Our method will extend to NURBS surfaces and other parametric forms. The performance could be higher on a GPU if more registers are available on it. Faster performance can be achieved using multiple GPUs simultaneously.

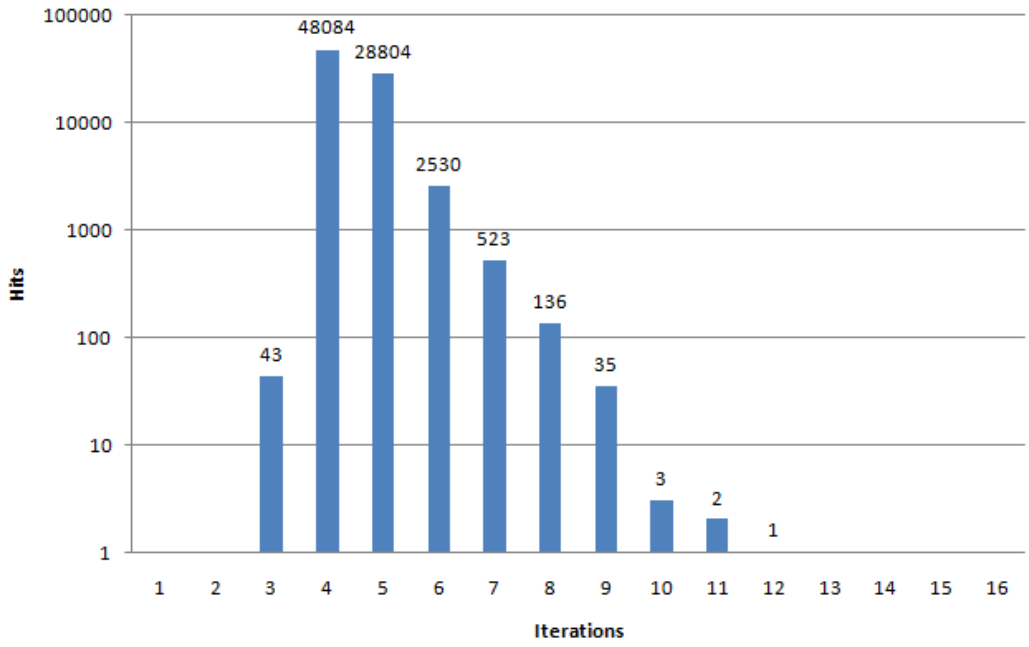


Figure 3.7: Newton Iteration Steps required for root finding in the Bigguy scene.

Chapter 4

Tracing Secondary Ray

The images produced from primary rays alone consider only the orientation of the surface and the position of the light source. They do not take into consideration other surfaces present in the scene and their impact at the point of interest. Thus, in order to achieve even greater realism, secondary rays are generated at the first point of intersection.

Shadow is one of the missing features in the primary pass. If the light rays coming towards the hit-point get obstructed by another surface, then the point of interest is considered as shadowed. The color value at a shadowed point is calculated by taking only the ambient light into consideration.

A surface can be specular in nature. A specular surface also takes into consideration the light coming from other surfaces in the scene. When a light ray hits an object in the scene, a part of it gets reflected. For a mirror-like surface, the angle of incidence is equal to the angle of reflection. The amount of light energy transferred keeps getting reduced as the number of bounces increases.

A ray after hitting a transparent/translucent surface gets transmitted into the surface. The angle of refracted ray is determined using Snell's law, which takes into consideration the refractive index of the two mediums as well as the angle of incidence. To accurately shade materials such as glass, we require both the reflected ray and refracted ray.

Shadow, reflection, or refraction rays can be generated for each intersection point of the primary ray, based on the material properties. We record the secondary rays in separate lists indexed by the primary ray. The planes to represent each are also computed and recorded similarly. These produce new ray-lists that can be processed in the same manner starting with Step 5 of Algorithm 1. Our scheme is equally efficient on secondary rays as a result.

Shadow rays are traced after primary rays, followed by reflection rays. The shadow, reflection, and refraction rays are independent of each other and can be traced together. This can be done on multiple GPUs or on a multi-GPU system. It can also be done on a single GPU by tracing all rays together. The massively multithreaded model of the top-end GPUs can handle a large number of threads. Joint tracing of shadow and reflection rays gives the same performance as separate processing. This process can be repeated for further bounces.

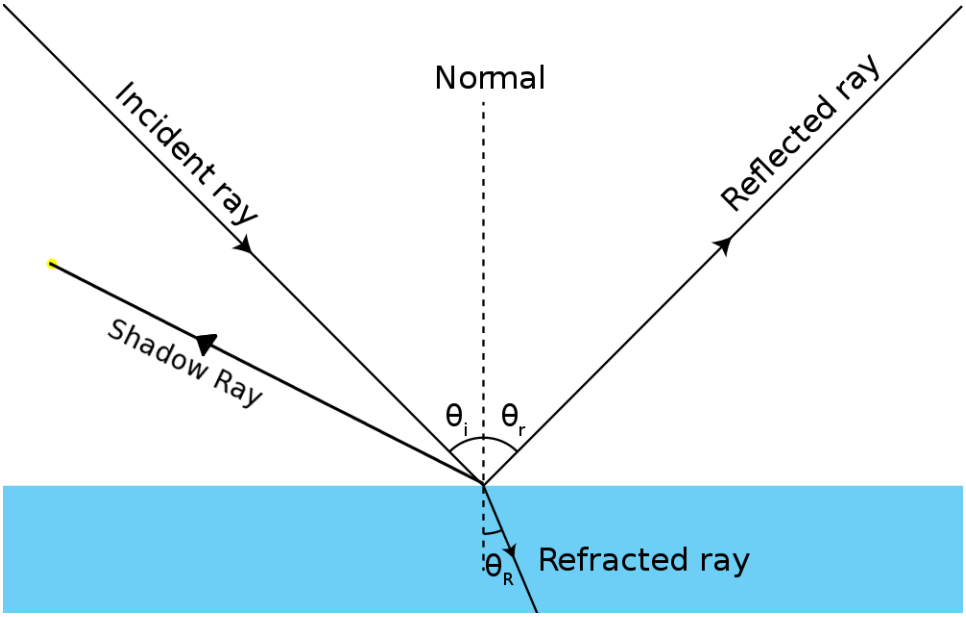


Figure 4.1: Figure showing shadow, reflection and refraction rays

4.1 Shadows

After the first pass, all the hit-points are compacted into one single list. From each of the hit-point, a ray is shot towards the point light source(s). Since we know both the hit-point and the light source point, calculating direction is fairly simple. After the direction calculation, we move to BVH traversal stage.

We use the same BVH constructed during the primary pass to traverse for secondary rays as well. However, back-patch culling is not used in case of shadow rays as a secondary ray can intersect with back-facing patches also. Scalability is provided by the use of extra memory blocks and atomic operations as before. Since we earlier ray traced rays within a block to provide coherence during the primary traversal stage, these rays tend to intersect at very close points. Since all the shadow rays are directed towards the point light source, the shadow rays tend to follow a similar path when traversing the BVH tree. This leads to coherence in case of shadow rays, although it is less as compared to primary stage.

After the traversal stage, recheck kernel and Newton iteration is applied exactly as done during the primary stage. In case of shadow rays, we are only required to know if the ray hits any surface lying between the hit-point and the light source. We need not perform a min-reduction to find the closest hit point value. Thus, we check for each potential ray-patch intersection in parallel. If the 't' value lies between the light source and the primary ray hit-point, it writes shade value as 1 in the shade array at the corresponding index value.

This also benefits CPU ray tracing. Once we find out if a shadow ray hits a surface, we need not check for any other potential surface intersections for that ray. Thus, we can discard doing the com-

plete algorithm for which intersection has been detected as early as the BVH traversal step itself. This provides significant improvements in the timings for the CPU code for shadow rays.

One of the problems with secondary rays is surface acne. Surface acne occurs due to floating point precision errors, when a ray intersects the same surface, some ϵ distance away, $|\epsilon| \rightarrow 0$. In case of shadow rays, incorrect self-intersection causes black dots to be present on the surface. To counter the problem, we perturb the ray origin for secondary rays by a small ϵ value, in the shadow/reflection direction. We also store the last surface intersected by the ray. This used at the very last part of the algorithm, where we check the actual distance value for each ray. If the distance comes out to be smaller than a fixed value and the ray intersects the same surface, we then discard the ray-patch intersection.

4.2 Reflection

Reflection occurs in case of specular surfaces. In case of ray tracing, we assume surfaces to be mirror-like. This leads us to make an assumption about the direction of the reflected ray. For a perfectly mirror-like surface, the angle of incidence is always equal to the angle of reflection. Thus, the direction of the reflected ray is given by

$$\vec{r} = \vec{i} - 2(\vec{i} \cdot \vec{n})\vec{n} \tag{4.1}$$

4.2.1 Plane Formation

For the secondary rays, we form a new vector with value 1 for the dimension whose value is the least in the reflection, refraction or shadow ray direction while keeping rest of the values 0. We take the cross product of this vector with the ray direction vector to get the plane normal vector. Another cross product with the normal at the point of intersection gives the normal vector for other plane. Finally, substituting the point of intersection in the plane equation, gives the plane constant, thus forming the complete plain equation.

After the traversal and Newton Iteration step, we again find the closest hit point for each ray. The point of intersection of the reflected ray and the normal at the point of intersection are stored. These are then used to calculate the color returned by the reflected ray. Every surface has a reflection coefficient, which is multiplied with the reflection and then added to the final color of the pixel. Reflection also suffers from surface acne and hence very small perturbation along the normal is required to correct it.

4.3 Refraction

Refraction occurs when the light wave changes its direction due to change in the medium. In ray tracing, for surfaces which are transparent/translucent, we trace a refracted ray which gets transmitted

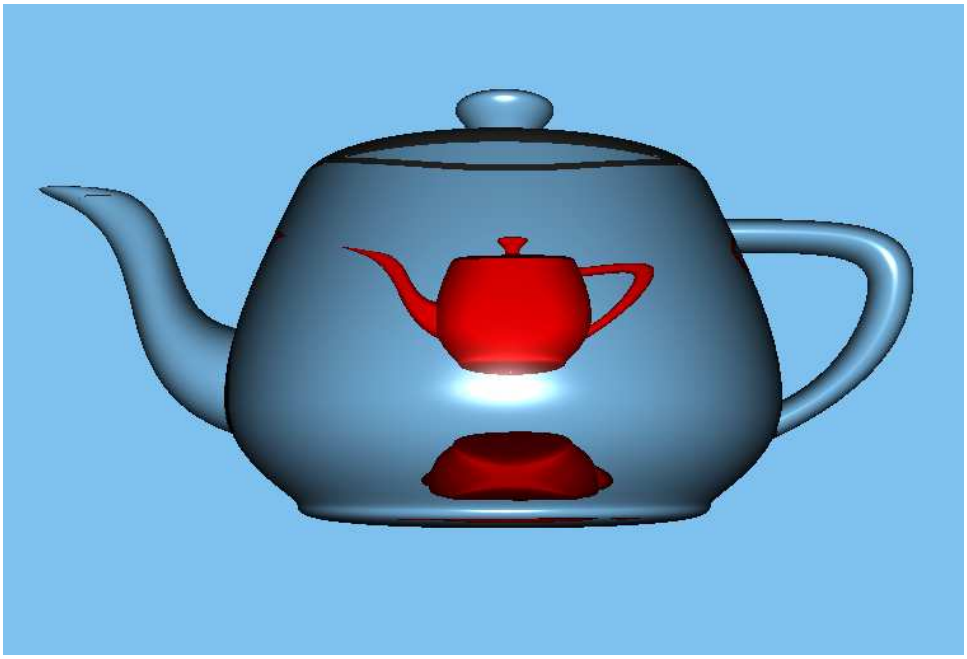


Figure 4.2: 2 Teapots with 1 level of refraction rendered at 1024×1024 resolution.

into the medium. When the refractive index of the medium changes, rays tend to bend either towards the normal (in the case of going from low index to high index medium) or away from the normal. The direction of the refracted ray is given by Snell's Law:

$$\eta_1 \sin\theta_i = \eta_2 \sin\theta_R \tag{4.2}$$

where η_1 and η_2 are refractive indices of two mediums, θ_i is the angle of incidence and θ_R is the angle of refraction (Refer Figure 4.1). The exact equation to calculate the direction of transmitted ray is given by:

$$\vec{t} = \frac{\eta_1}{\eta_2}\vec{i} - \left(\frac{\eta_1}{\eta_2}\cos\theta_i + \sqrt{1 - \sin^2\theta_R}\right)\vec{n} \tag{4.3}$$

where

$$\cos\theta_i = \vec{i} \cdot \vec{n}$$
$$\sin^2\theta_R = \left(\frac{\eta_1}{\eta_2}\right)^2 (1 - \cos^2\theta_i)$$

Here, $\vec{t}$ is the transmitted ray, $\vec{i}$ is the incident ray and $\vec{n}$ is the surface normal, in the direction opposite to the direction of the incident ray. All other steps are performed exactly as reflection.

With refraction, one needs to perform multi-bounce in order to simulate it correctly. Thus, when a ray hits a transparent surface, first secondary intersection occurs with the back-patches of the object. After this intersection, the ray again comes back to the first medium and intersects another surface in

Model (# patch)	Pass	# of ray-patch intersections		Computation time on GPU in milliseconds		Total frame time in milliseconds		
		Total	Per ray	Traversal	Newton	CPU	GPU	Hyb
Teapot	P	126589	1.6	5.53	3.18	74	8.71	8.01
(32)	S	51670	0.74	2.59	1.27	29	3.86	3.6
Fig 4.3(a)	R	71528	1.02	3.09	1.39	34	4.48	4.0
Bigguy	P	142779	1.49	11.8	2.79	110	14.59	13.18
(3570)	S	110732	1.19	11.24	3.02	64	14.26	12.62
Fig 4.3(b)	R	59505	0.64	8.47	2.13	58	10.6	9.12
Killeroo	P	147116	1.55	19.08	3.3	193	22.38	20.43
(11532)	S	136960	1.46	16.62	4.83	90	21.45	19.12
Fig 4.3(c)	R	48637	0.52	11.92	2.55	68	14.47	12.64
2 Killeroos	P	317494	1.65	35.15	7.14	356	42.29	38.58
(23064)	S	231183	1.22	21.52	6.23	196	27.75	24.88
Fig 4.3(d)	R	138562	0.73	28.54	7.45	180	35.99	31.09
9 Bigguys	P	570136	2.09	64.69	12.36	2092	77.05	75.9
(32130)	S	435832	1.63	38.33	10.04	954	48.37	46.2
Fig 4.3(e)	R	297427	1.11	57.2	14.17	621	71.37	69.71

Table 4.1: Rendering times on an Intel i7 920 + Nvidia GTX580 for different models and passes at 1024×1024 screen resolution: primary (P), shadow (S) and reflection (R). Third column gives the total and the fourth column the average number of ray-patch intersections computed. CPU is our implementation for multicore system, GPU refers to GTX 580 timings and Hyb refers to the hybrid timings.

the scene. Thus, we need to perform $2x$ bounces in order to simulate x refraction bounces. Figure 4.2 shows a ray traced image containing transparent teapot made from glass. We trace the image for two bounces (1 refraction bounce). We are able to render the scene at 1024×1024 resolution in about 33 ms.

4.4 Results

We report ray tracing performance for primary rays for rendering 1024×1024 images of the following models: Teapot (32 patches), Bigguy (3570 patches), Killeroo (11532 patches), 2 Killeroos (23064) and 9 bigguy (32130 patches). The experiments used an Intel 2.67GHz Intel Core i7 920 processor with 4 cores. The GPU used is Nvidia GTX580 with 512 cores, on CUDA4.0. All the figures are rendered at 1024×1024 resolution unless stated otherwise.

Table 4.1 shows the timings for different secondary passes for different models. We are able to render teapot model with shadows and reflection at about 64 fps at a resolution of 1024×1024 using our hybrid algorithm. We are able to render the Bigguy model with secondary rays at about 28.6 fps. We see that our GPU implementation is about 4-8 times faster and hybrid about 5-9 times faster for than the optimized multicore CPU algorithm for all cases except the 9 Bigguys scene. As the number of patches increases in the 9 Bigguys scene, CPU becomes very slow and GPU and Hybrid times improve

Model	Pass	Potential Intersections	Time(ms)		
			Traversal	Newton	Total
Teapot	P	126589	3.58	1.82	5.4
	S	51670	0.75	0.18	0.93
	R	71528	1.02	0.3	1.32
Bigguy	P	142779	8.91	1.71	10.62
	S	110732	9.25	2.05	11.3
	R	59505	7.52	1.83	9.35
Killeroo	P	147116	15.53	2.28	17.81
	S	136960	13.94	3.48	17.42
	R	48637	10.63	1.99	12.62
2 Killeroos	P	317494	29.34	4.87	34.21
	S	231183	17.86	4.43	22.29
	R	138562	24.7	5.63	30.33
9 Bigguys	P	570136	50.9	8.18	59.08
	S	435832	30.14	6.94	37.08
	R	297427	45.59	11.05	56.64

Table 4.2: Rendering times on Nvidia K20c for different models and passes at 1024×1024 screen resolution: primary (P), shadow (S) and reflection (R). Third column gives the total number of ray-patch intersections computed.

considerably. We see that our hybrid algorithm is 8-15% faster as compared to GPU except for 9 bigguys scene. Since CPU contribution becomes less for higher patch scene, the overall advantage in terms of percentage gets reduced.

Table 4.2 shows the timings for different secondary passes for different models on NVIDIA K20c. K20c is the latest NVIDIA GPU with 2496 cores as compared to 512 on NVIDIA GTX 580. We see in general a speedup of about 15-20% on K20c as compared to GTX 580 and about 10% improvement as compared to hybrid algorithm. We see heavy speedup in teapot scene. This can be attributed largely to the high L2 cache size in K20c, which leads to faster memory accesses for smaller models like teapot.

We see shadow rays performing relatively better as compared to reflection. Since we performed primary intersections for a block of 4×4 rays, we get hit points of neighborhood area together. Shadow rays from these hit points travel in the same direction, towards the light source. This leads to some coherence in the shadow rays. With the cache memory present in the newer GPUs, the overall timing for shadow rays come out to be lesser than reflection rays, even though number of rays traversed is the same.

4.5 Multi Bounce

In real world, a ray bounces off many objects before reaching the eye. Thus, one level of reflection is just an approximation of the actual image. In order to provide more realism, we implement multiple

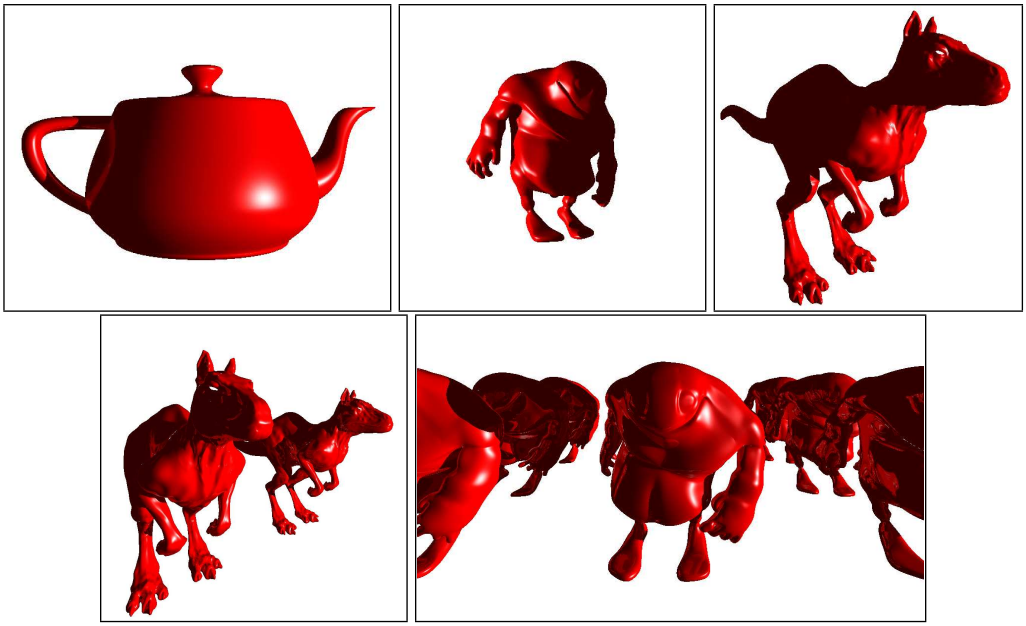


Figure 4.3: Top row, left to right: Views of Teapot, Bigguy and Killeroo. Bottom row, left to right: Views of 2 Killeroos, 9 Bigguys with shadow and reflection.

bounce reflections. True recursion is difficult to support on the GPU. Fixed-bounce ray tracing has been implemented, with data for previous bounce present. Shadows are calculated for both primary rays and reflected rays. Figure 4.4 shows a comparison of results for 1 bounce and 3 bounce image. The figure shows 3 levels of reflection in the marked region. We are able to render an image with 1024×1024 resolution for 1 level of reflection in about 24.9 ms and for 3 levels of reflection in 27.3 ms on NVIDIA GTX 580.

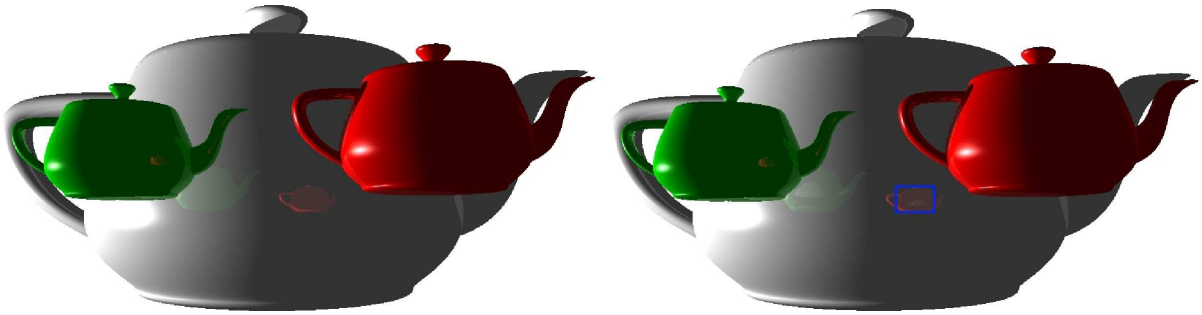
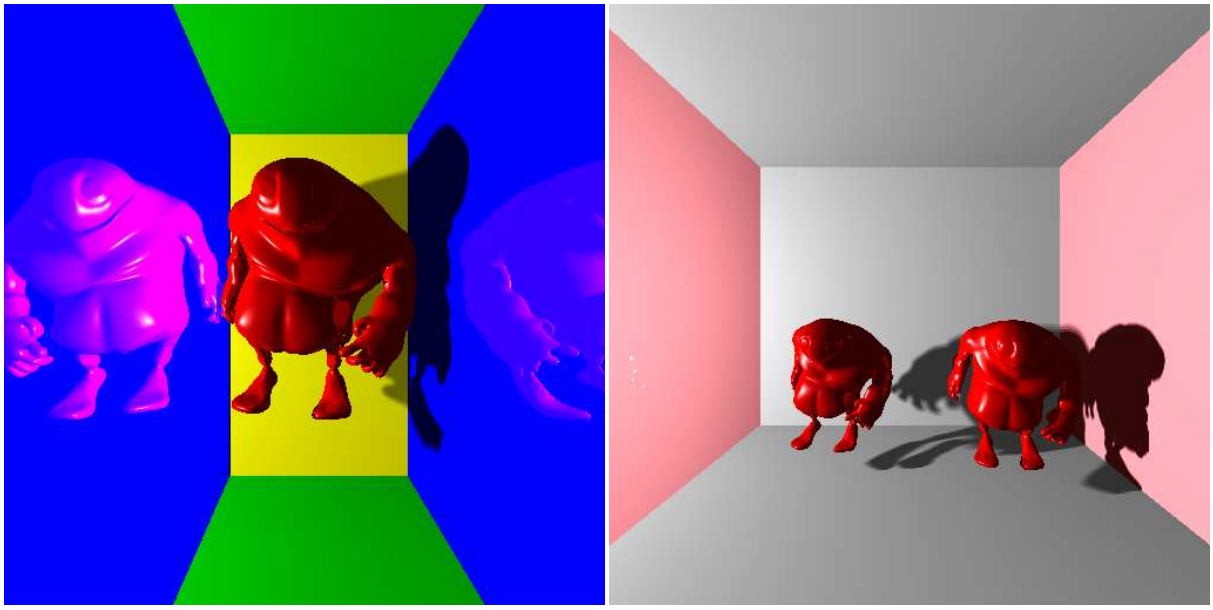


Figure 4.4: 3 Teapots with shadows and (a) 1 bounce (b) 3 bounces. The marked region shows 3 levels of reflection.



(a) (b)

Figure 4.5: Soft Shadows for Bigguy(s) in a box scene rendered at 512×512 resolution.

4.6 Soft Shadows

In ray tracing applications, we conveniently assume light source to be point source. In reality, light source is never a point source, but an area light source. Point light source model is accurate when the light sources are very far away, but inaccurate when representing light sources which are either large or very close to the scene. When we use point source as light, the shadows produced are hard shadows. Hard shadows are the ones which produced sharp edges. These are well defined and each pixel on the image is either shadowed or not shadowed. Thus, one bit is enough to represent if a pixel is shadowed or not.

Real world shadows are much softer. Soft shadows are visually much more realistic but are more expensive to calculate. Soft shadows gradually fade away towards the boundaries. This simulates the actual real world shadows, where hard boundaries are not present.

Soft shadows can be simulated in two ways. The first method is consider the area light source as an array of point light sources, which are equally spaced throughout the area light source. The method is called uniform sampling method. Thus, one now ray traces shadow rays to each of these point sources, instead of just one light source. The other way is to sample the light area, using Monte Carlo methods, to generate a list of random points on the light source and calculate soft shadows from the information. Both the methods tend to generate almost similar results visually. We present the former method in this chapter.

In our implementation of soft shadows, we use 8×8 point light sources to simulate an area light source. After the primary pass, each point of intersection spawns 64 shadow rays. After ray generation

step, we partition the rays according to the maximum GPU limit. Rays are then processed in batches. This step leads to maximum utilization of the GPU, instead of doing it in batches for each point source.

Each shadow ray is traversed parallelly and independently on the GPU. We add the shadow bit values for each of the rays and finally average it out to get the shadow value. During shading phase, instead of multiplying the color value with shadow bit, we multiply it with the shadow value (lying between 0 and 1). This leads to softer shadows, with hard shadows occurring only for deeply obscured points.

We use a box scene to demonstrate the shadows. Figure 4.5(a) shows the bigguy model in a box scene with soft shadows rendered at 512×512 resolution in 167.4 ms. Each point on the bigguy and the background box shoots 16 rays towards the light source. We can observe softer shadow boundaries on the walls as well as patches. Figure 4.5(b) shows the 2 bigguys in a box scene with 16 shadow rays for hit-point. The scene takes about 158.8 ms to render on NVIDIA GTX 580.

Chapter 5

Global Illumination

Ray tracing provides a simple measure to perform shading and also to simulate effects like reflection, refraction and shadows, which was not possible with basic rendering methods. However, ray traced images have too many sharp features, which is not found in nature. Plain ray tracing images look far from real world images. Real world images do not have sharp boundaries, perfect reflections and refractions, objects can be in focus or out of focus and objects can also be moving at the time of capturing image.

This happens because rays traced are through the pixels, thereby leaving out spaces between the pixels unaccounted for. In order to provide more realism to graphics images, Cook [10] developed distributed ray tracing.

5.1 Distributed Ray Tracing and Stochastic Sampling

Ray tracing is a discrete algorithm as it only considers pixels which are sampled at regular intervals. Thus, we see a loss of information, which is found in between the pixels. This leads to aliasing artifacts. Supersampling tries to reduce aliasing by using more than one regularly sampled point per pixel. However, aliasing still exists and supersampling only leads to reduction in the aliasing.

Cook [9] presented a new approach called Stochastic Sampling to eliminate aliasing. In stochastic sampling, image is sampled at nonuniform intervals rather than being sampled at uniform intervals. By applying Monte Carlo integration, one can calculate the color of the pixels from the color of all the nonuniform samples taken for that pixel. Stochastic sampling replaces aliasing with noise of correct average intensity. Thus visually, image appears much better than the aliased images.

Stochastic sampling also leads to providing additional capabilities to the traditional ray tracing algorithms. By sampling the pixels in time, lens area, reflection angle, refraction angle etc., we can now generate effects such as motion blur, depth of field, gloss, translucency etc. which were previously performed by course approximations. This is called distributed ray tracing. In distributed ray tracing, we do not require any more rays than a super sampled image. However, by distributing these rays

non-uniformly throughout the pixel, we can replace the aliasing with noise and also produce advance effects.

5.2 Path Tracing

Our data parallel approach for ray tracing parametrics extends easily to global illumination effects like path tracing. Path tracing generates a large number of rays for each pixel and integrates their color values to generate the final image. At the basic level, path tracing involves tracing a very large number of rays. Path Tracing of parametric patches has not been shown in the past. We extend our method to path tracing to establish its scalability and effectiveness.

We implement unbiased Monte Carlo path tracer to render the images. Our implementation is mostly derived from Realistic Ray Tracing by Peter Shirley [43]. In our path tracing implementation, each pixel is sampled using a user defined number of samples s. We perform stratified sampling by dividing each pixel into 4 quadrants and spawning s rays in random directions going through their respective quadrant. This leads to a uniform sampling for better convergence. In our approach, this results in increasing the number of primary rays. The super-sampling of sub-pixels also leads to anti-aliasing. This removes all the rough, sharp edges except around the light source. The ray list now has s times the number of rays. The data parallel approach easily extends to this and we continue from step 5 of Algorithm 1 using a large ray list. A ray is made to intersect with the scene, consisting of an open box with 5 walls and a parametric object at the center of the box (Figure 5.1). For demonstration purpose, the walls are made purely diffusive while the patches are purely reflective. Thus, on intersection with the wall, a new ray is spawned in a random direction to continue with the path tracing. On intersection with a patch, the reflected ray is generated from the incident ray. A more detailed model, including materials BRDF, can also be applied in a similar way. The ray generation stops when either one of the three conditions are met: (a) Ray hits a light source (b) Ray does not hit anything (c) Maximum depth is reached. We used a maximum depth of 10 for the test scenes.

For case (a), we store the color calculated throughout the path at the original ray index. For case (b), no color gets stored since the ray does not hit any light sources. For case (c), we perform direct illumination. For each ray unfinished after maximum depth, a new ray is spawned towards a random point on the area light source. This is done for both diffusive as well as reflective surface. If the point does not appear to be shadowed, its color is stored into the color array.

After finding out the color values for all the rays, the mean is calculated by applying a simple addition reduction operation for all the rays. Since GPU has a limited amount of memory, we perform the above procedure iteratively for a fixed block of pixels depending upon the value of samples per pixel. We apply a gamma correction of 2.2 on the color values generated. Apart from this, no other form of post-processing is done on the final image. Figure 5.1 shows some of the rendered images with different number of samples per pixel. We are able to render bigguy in a box image at 512×512 resolution for

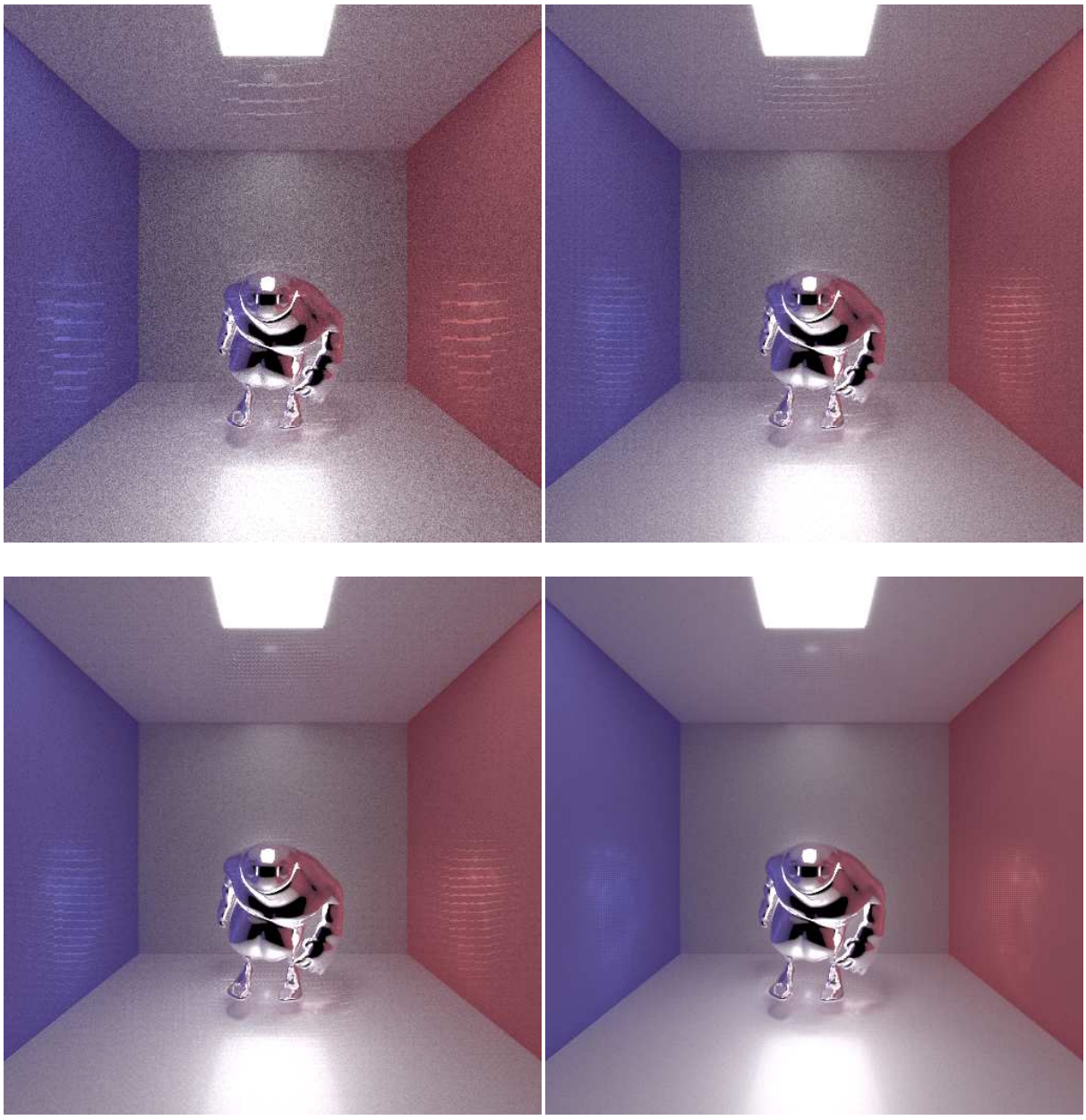


Figure 5.1: Path Tracing Bigguy in a box scene with (a) 400 spp, (b) 1000 spp, (c) 3000 spp and (d) 10000 spp, being rendered at 512×512 resolution.

300 samples per pixel at about 60 secs, for 1000 samples at about 165 secs, for 3000 samples at about 8.5 mins and for 10000 samples at about 28.5 minutes on a NVIDIA GTX 580 card.

5.3 Advanced Effects

In this section, we discuss some of the advanced global illumination effects, which we have performed for Bezier surfaces. We use our ray tracing algorithm and expand it to produce these advanced effects.

5.3.1 Ambient Occlusion

Ambient occlusion has become widely popular because of its heavy usage in film industry. Ambient Occlusion is a very crude way to approximate the effect of environment lighting. It is a form of shading method which is global in nature and takes into consideration light coming from different parts of the scene rather than just the light sources. One can consider it to be a smart ambient term which varies over the surface of the model based on how much visible the surface is to the environment. Ambient Occlusion highlights the surface details of a model and also provides with soft shadows, thereby looking much more realistic than normal local shading models.

Ambient occlusion deals with diffuse surfaces and diffuse lighting. On a cloudy day, the light is all diffused and cannot be said to be coming from one source. This is where diffuse lighting comes into picture and ambient occlusion is required in these types of situations.

A surface is illuminated not only by light sources but also from inter-surface reflections. Ambient Occlusion takes into account the geometry of the scene which is occluding the environment from the point being shaded. Thus, it provides much more realism to the images as compared to local shading methods.

The basic idea behind ambient occlusion is to know how much part of the environment is getting occluded from each point on the model and how much of the external environment it can actually see. This information provides us with a diffuse lighting term, which can either be used as it is for shading or can used in conjunction with other shading methods.

The Ambient Occlusion equation at a point is given by

$$O_p = \frac{1}{\pi} \int_{\Omega} V_p(\vec{\omega})(\vec{n} \cdot \vec{\omega}) d\omega$$

where $V_p(\vec{\omega})$ is the visibility function at p in direction of $\vec{\omega}$, $\vec{n}$ is the surface normal and $d\omega$ is the infinitesimal solid angle through the hemisphere Ω. The integral is solved via Monte Carlo Integration.

5.3.2 GPU Algorithm

The first pass consists of primary rays being shot into the scene, to generate the primary intersection points. We store the hit-points and the surface normals at these points. We then proceed to shoot a fixed amount of rays, in random directions passing through the hemisphere centered around the surface normal and away from the surface.

Since the number of rays shot from each point varies typically between 128 and 1024, it leads to generation of a massively large ray list. This ray list is then divided into chunks of maximum GPU capacity. Each chunk is then processed parallely on the GPU.

5.3.2.1 Generating Rays

We generate the '*n*' occlusion rays in parallel on the GPU. The normal information is stored in the shared memory and each CUDA block processes one or two points. For each point, two random numbers are generated, lying between 0 and 1.

We first calculate to vectors orthogonal to the surface normal and orthogonal to each other. The plane formed by these two vectors is termed the horizontal plane. Stratified sampling is performed by dividing the horizontal plane (and thereby the hemisphere) into 4 quarters and shooting equal rays through each of these quarters. The angle is given by

$$\theta_i = \pi/2 * (u + j), \qquad u \in [0, 1] \text{ and } j \text{ is the quadrant number}$$

The second random number is used as it is as the distance along the normal. Thus, using these two parameters, one can generate an occlusion ray, going through the '*j*th' quarter.

5.3.2.2 Occlusion Parameters

Several parameters determine the final shade value for a point. Firstly, in our experiments, we found that using stratified sampling to get ray directions instead of random ray directions provides a much better estimate of the occlusion. Maximum distance up to which a ray is considered as occluded also plays a major role. We can also generate shade values for each ray as opposed to bit values. These values can then depend upon the distance of point of intersection and the shade value for each ray can be attenuated accordingly. Lastly, we used a bias for the distance along the normal direction, as the light coming close from the horizon has very less impact as compared to the light coming from the front.

5.3.2.3 Shading

After the Newton iteration stage, each ray is checked for intersection and also the distance between the point of origin and the point of intersection for the ray. A shade value is generated based on this. This shade value is generated for each ray and is then averaged out. Finally, the averaged shade value is used

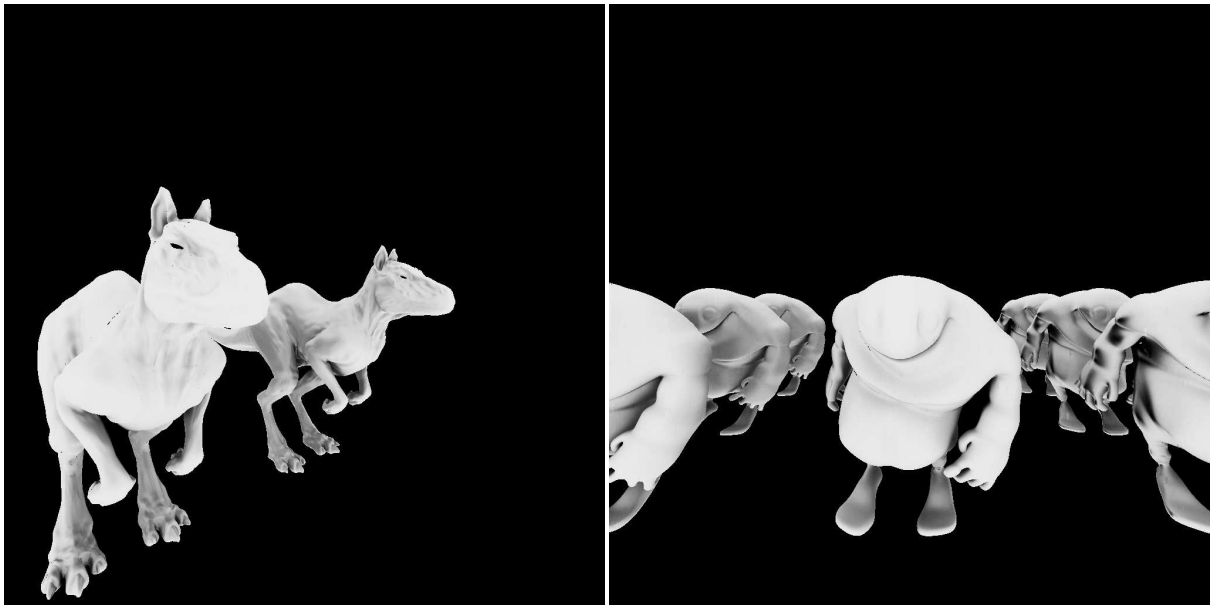


Figure 5.2: Scenes generated using Ambient Occlusion. We use 1024 rays per hit point to shoot out rays in random direction around the hemisphere for a screen resolution of 1024×1024.

as the color value to generate a gray-scaled image. Figure 5.2 displays some of the images generated using ambient occlusion for different test scenes. The images are rendered at 1024×1024 resolution with 1024 rays for each hit-point. We render the 2 killeroo scene in 33 seconds while 9 Bigguys scene takes about 65 seconds to render.

Table 5.1 shows the timings for different models and samples per pixel. The models are rendered on NVIDIA GTX 580 with resolution 1024×1024. From the table, we can infer that the time per ray remains constant when the number of samples is increased. This can be attributed for mainly two reasons. With more number of samples, the ray coherence increases which leads to better performance and almost same time per ray. Higher ray coherence will lead to better performance on the newer GPU which are now equipped with L1 and L2 cache. The parallel architecture of GPU benefits greatly from heavy parallel operations. This leads to slight improvement in ray time as we increase the number of rays until the calculations performed are sufficiently large.

5.3.3 Depth of Field

Depth of Field is the effect in which some objects appear to be in focus while all other objects in front or towards the back of the focus range appear out of focus. Photographers and cinematographers make use of depth of field to direct attention to certain parts in an image. One of the many reasons why ray traced images are not so realistic is because every object in the image is in focus. This happens because ray tracing considers a pin-hole camera, which makes everything in the scene to be in focus. Many a times, real world images contain objects which are out-of-focus.

Model	Samples per pixel	Number of rays	Time (ms)	Time per ray (μs)
Bigguy	64	4496635	547.7	0.122
	128	8924091	1031.5	0.115
	256	17779003	2006.3	0.113
	512	35488827	3988.1	0.112
	1024	70908475	7954.6	0.112
Killeroo	64	6088875	984.2	0.162
	128	12084075	1835.2	0.152
	256	24074475	3536.4	0.147
	512	48055275	7026	0.146
	1024	96016875	14019.4	0.146
2 Killeroos	64	12303395	2299.8	0.187
	128	24417507	4304.4	0.176
	256	48645731	8311.8	0.171
	512	97102179	16565.1	0.170
	1024	194015075	33069.7	0.170
9 Bigguys	64	17386330	4537.7	0.261
	128	34505178	8483.1	0.246
	256	68742874	16449.2	0.239
	512	137218266	32793.8	0.239
	1024	274169050	65472.2	0.239

Table 5.1: Rendering time and time per ray for different models and samples per pixel at 1024×1024 resolution for ambient occlusion on NVIDIA GTX 580.

Depth of field effect can be simulated either by doing post-processing on a normally generated image or do it directly by using distributed ray tracing. We demonstrate depth of field effect using the latter method because the former method cannot be always correct. Using distributed ray tracing, we successfully try to approximate the physics behind depth of field and the results produced look much more photorealistic.

In optics, when an object is not in focus, the rays project onto a region on the film instead of converging at the focus. This region is called Circle of Confusion. Thus, Circle of Confusion is a region on the film/ image plane caused by a cone of light rays which are not converging at the focus when imaging a point source. The diameter of the Circle of Confusion (CoC) increases with the size of the lens and the distance from the plane in focus.

In normal ray tracing, we shoot rays from camera center (center of the lens) 'o' to a point 'p' on the focal plane. To simulate depth of field, instead of shooting rays from center, we start by randomly selecting a point on the Circle of Confusion (centered at 'o') and continuing the procedure from there on. The random point on the circle is calculated by generating two random points. First random point is transformed to provide us with the angle and the second point is used to provide the distance from the center (parametric equation for circle). Without going into the mathematical details behind depth of

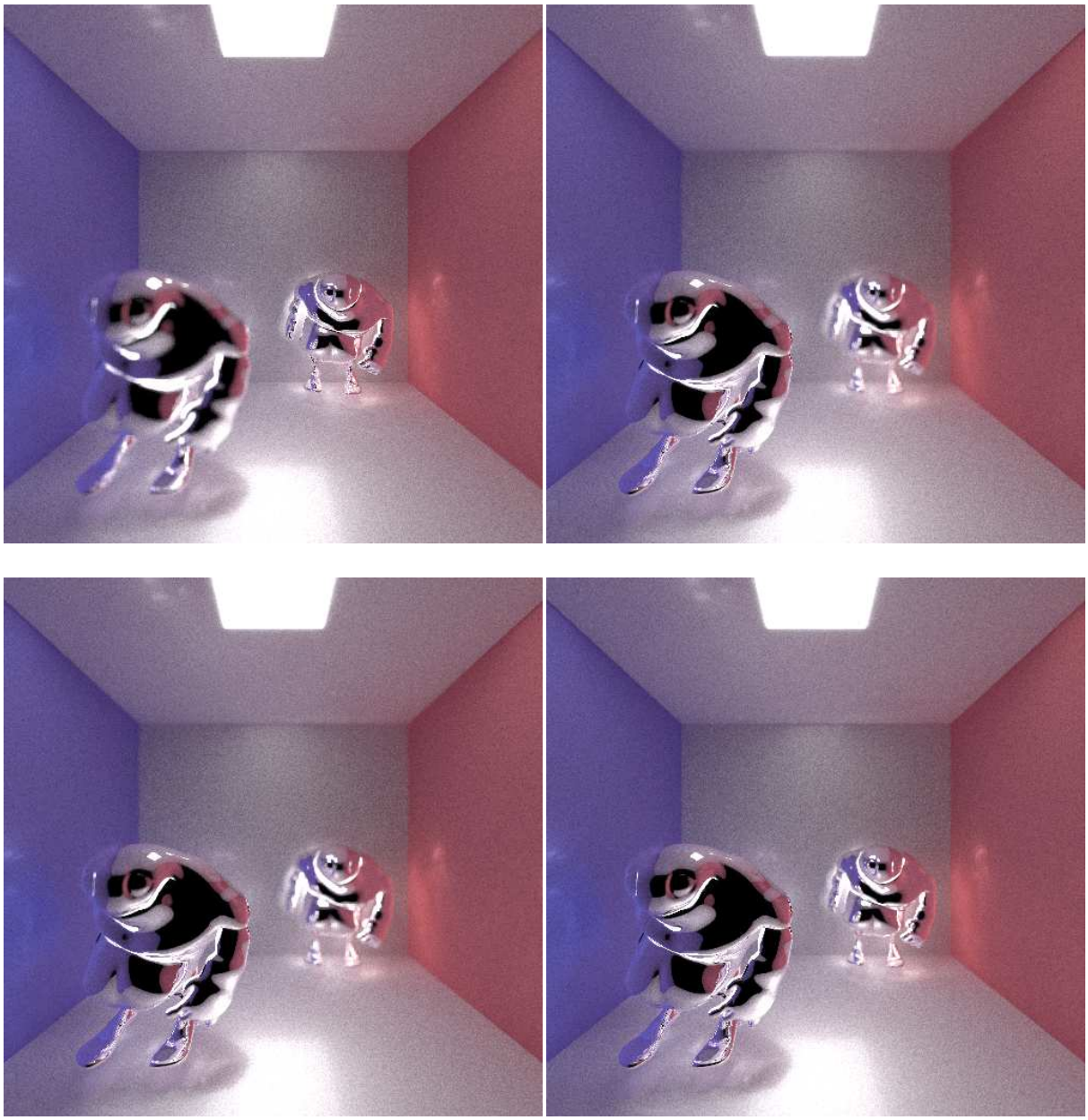


Figure 5.3: Depth of Field for a scene with 1000 spp and rendered at 512×512 resolution. The figures show (a) Traced with focus at far end, aperture = 0.8 (b) focus at near end, aperture = 0.8 (c) near focus, aperture =0.8 and (d) near focus, aperture = 0.2.

Model	Focus	CoC	Time (ms)	Time per ray (μs)
Bigguy	Far	0.8	193	0.736
	Near	0.8	193	0.736
	Near	0.2	192	0.733
2 Bigguys	Far	0.8	273.5	1.043
	Near	0.8	272.5	1.04
	Near	0.2	271	1.034

Table 5.2: Rendering time and time per ray for different models and camera properties at 512×512 resolution for Depth of Field.

field, the distributed ray tracing equation gets transformed in the following way:

$$C = \frac{1}{n}\sum_{i=1}^{n} raytrace(o, p_i - o) \qquad \text{Pinhole Camera}$$

$$C = \frac{1}{n}\sum_{i=1}^{n} raytrace(q_i, p_i - q_i) \qquad \text{Depth of Field}$$

where n is the total number of samples taken, o is the camera origin, p_i is point on the image plane and q_i is point on the Circle of Confusion, centered around 'o'. Figure 5.3 shows images ray traced with depth of field. Shifting the focus plane changes the region of the image in focus. Increasing the aperture size leads to decrease in the area of focus and enhances the depth of field effect. We render the images in about 272 seconds for 512×512 resolution with 1000 samples per pixel.

Table 5.2 shows results for different models and camera properties. We see that the time per ray remains constant when we change the focus of the camera. However, it decreases slightly when the radius of Circle of Confusion is decreased. When we decrease the radius, coherence increases among neighboring rays, which leads to a slight gain in performance.

5.3.4 Motion Blur

In ray tracing, we consider the scene to be either static or dynamic. Animation for dynamic scenes is produced by generating a sequence of still images and then displaying these images in correct order. The process, however, fails to captures the motion of an object during the time camera shutter is open. If an object is in motion during the time camera shutter is open, it causes blurriness in the image. This blur caused due to motion is termed as motion blur. In real world cameras, the color of each pixel for a frame is calculated by averaging out the color values during the time shutter remains open.

To simulate motion blur, we need to sample the rays temporally as well. Thus the rays are now distributed both spatially and temporally. Each ray is assigned a time value, which it then uses to calculate the new position of the object and perform ray tracing accordingly.

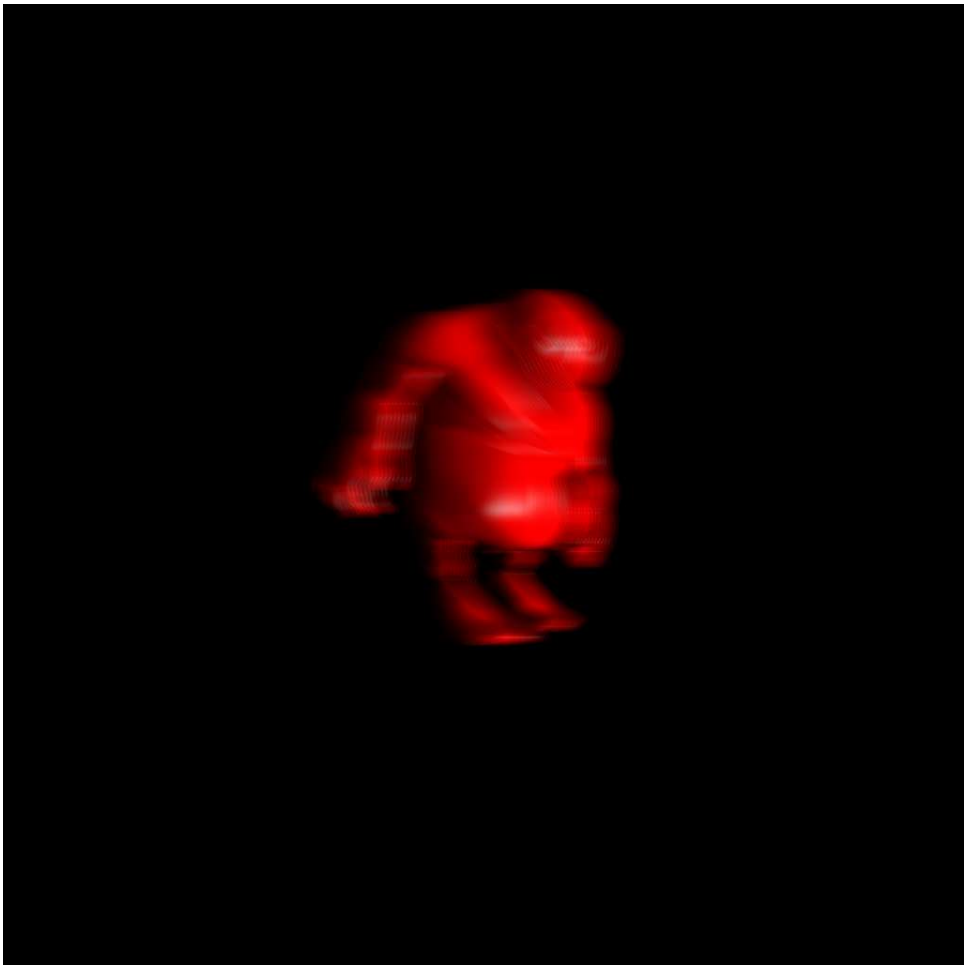


Figure 5.4: Motion Blurred image captured by shooting rays across time. Image rendered at 512×512 resolution.

We demonstrate motion blur by moving the camera by the same distance as the object would have moved, but in opposite direction. By moving the camera, we can make use of the same acceleration structure for all rays, which otherwise would have required to be rebuilt for all the time steps. Figure 5.4 shows the rendered motion blurred image. We average out the color values coming from rays in different time steps. One can use different filtering scheme to produce motion blur like box filter or weighted filtering. We render the scene with 10 time steps, one ray per time step, in 115 ms.

5.3.5 Gloss

In our implementation of path tracing, we demonstrated a scene where the parametric object in the center was considered to be purely specular. This meant that the rays which intersected the object where traced only in the reflected direction. Real world objects are not purely specular. Hence the reflections are a bit blurred rather than being exact.

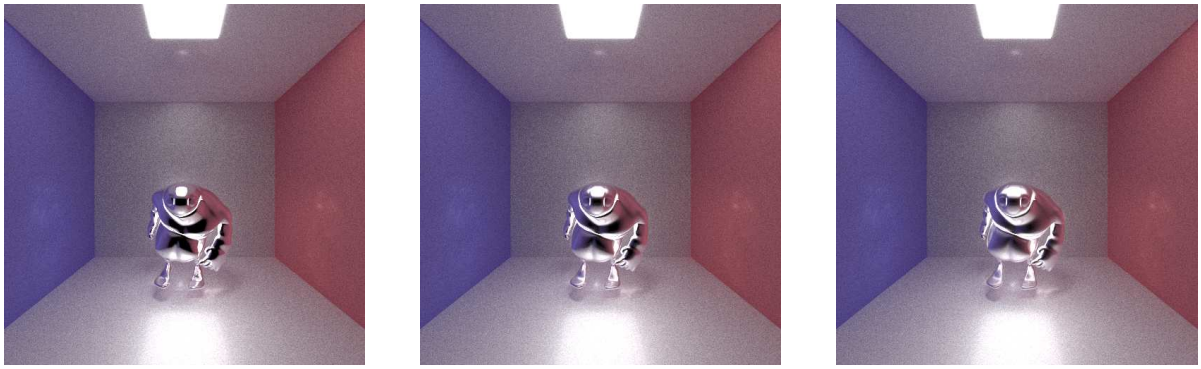


Figure 5.5: Glossy reflections for a bigguy model, using 1000 samples per pixel and 4 reflection rays for each intersection, rendered at 512×512 resolution with (a) s value as 0.1 (b) s value as 0.4 and (c) s value as 0.6.

Glossy surfaces are simulated by distributing the reflected rays around the reflection direction. Thus, instead of using one ray shooting out towards the reflection direction, we consider a packet of rays shooting out in random direction through a solid angle centered at the reflection direction. The color returned is then averaged out to produce the glossiness. We can even model different types of surfaces by making use of its Bidirectional Reflectance Distribution Function (BRDF). The reflected direction is calculated as follows

$$R_i = R + s(pu + qv)$$

where R is the reflection direction, s is the glossiness index (blur size or the solid angle), p and q are random numbers in $[-1, 1]$ and u, v are vectors orthogonal to R. The ray direction R_i is normalized and ray traced further. In our experiment, we produce ray packets only for the first pass and from then on, each hit with the parametric surface produces one ray, which can either be in the reflection direction or in a random direction around the reflection direction. Figure 5.5 displays rendered images of bigguy with glossy surface for different values of s. As we increase the glossiness index, reflections start to get more blurry. Increasing the blur size also leads to a small increase in the overall rendering time. We are

Number of rays	Rays per hit	Gloss	Time (ms)	Time per ray (μs)
270768585	4	Low	244.7	0.904
270768585	4	Med	248.9	0.919
270768585	4	High	252.1	0.931
990607950	16	Med	792.6	0.8

Table 5.3: Rendering time and time per ray for bigguy in a box scene with different number of rays per hit-point and glossiness level rendered at 512×512 resolution.

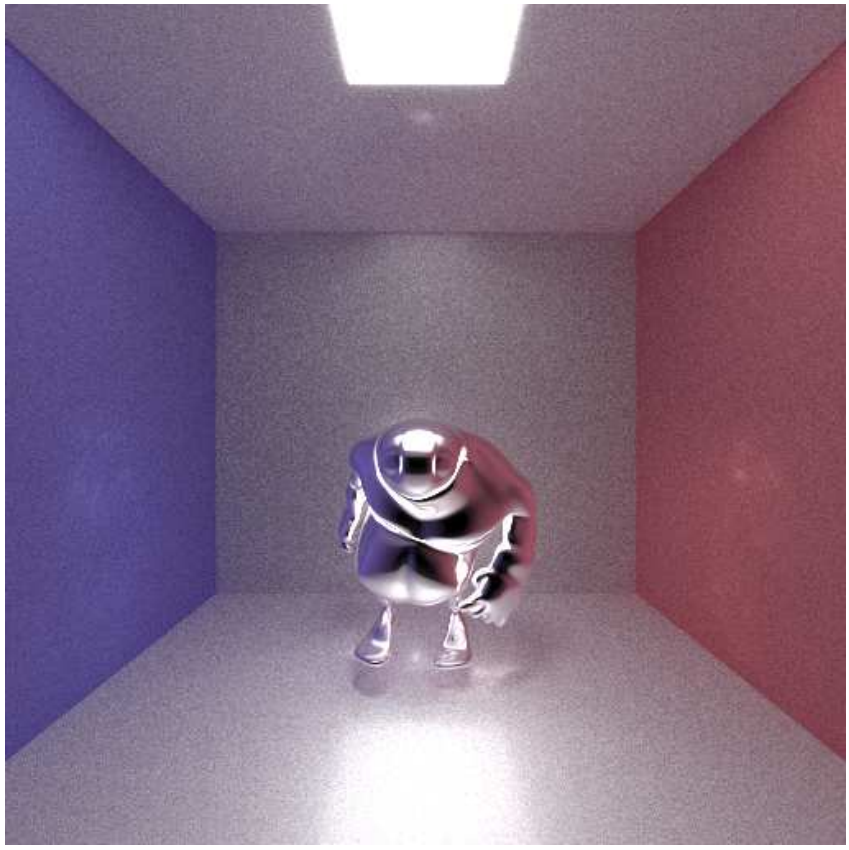


Figure 5.6: Glossy reflections for a bigguy model, using 1000 samples per pixel and 16 reflection rays for each intersection, rendered at 512×512 resolution with 0.4 as s value.

able to render the image with small blur at about 244.7 seconds, medium blur at about 248.8 seconds and a larger blur at about 252 seconds. Figure 5.6 shows a glossy surface rendered with 16 reflection rays and medium blur in about 13.2 minutes.

Table 5.3 shows rendering times for Bigguy in a box scene for different number of rays per hit and glossiness levels. We see improvement in rendering times when rendering with 16 rays per hit-point. This can be attributed to higher level of coherence among the rays and also to increase in total number of rays. We see higher ray time in case of gloss (Table 5.3) as compared to depth of field (Table 5.2). When we shoot rays in random direction along the reflection direction in case of gloss effect, rays lose their coherence and hence we can see an increase in time per ray. From these results, we can conclude that coherence is essential when tracing rays, more important in case of global illumination effects. Sorting rays at each step, to increase the coherence, would lead to even greater speedups.

Chapter 6

Conclusions and Future Work

We presented a method to ray trace Bezier surfaces at interactive frame rates using a mixed hierarchy structure, leveraging the compute power of both the CPU and the GPU. Ray tracing is an embarrassingly parallel process, making it ideal for the parallel architecture. The proposed mixed hierarchy model leads to efficient performance on the GPU. Shared memory accesses are much faster as compared to global memory accesses. We made use of this property to build a mixed hierarchy structure with the lower hierarchy being constructed for fixed depth. With fixed depth subtree, we used shared memory to store information about the lower subtree like skip pointer, subpatch id, etc. This lead to faster access times and provided speedup while traversing the structure on the GPU. A lower subtree also lead to tighter bounds, which reduced the potential ray patch intersection list.

After finding out the potential ray patch intersections, we investigated two methods to solve for ray patch intersection test. We improved Kajiya's method to ray trace parametric surfaces. The method is complex and requires use of double precision arithmetic, which is slow in comparison to single precision arithmetic on the GPU. We also implemented the Newton's method to solve for the intersection problem. Newton's method is simpler and makes use of single precision arithmetic. A higher depth at the lower subtree of the mixed hierarchy model lead to better initial values, which is responsible for better and faster convergence of the Newton iteration.

To make use of all the computing power available to us, we developed a hybrid ray tracing system. We split the initial rays between the CPU and the GPU according to the compute capabilities of the two. The ray list is then processed parallely on the CPU and the GPU. We have optimized the implementations so as to suit the needs of the two architectures. We have also implemented a multi-GPU method which performs ray tracing simultaneously on all the GPUs attached with the system. The work can be further extended to solving the problem on multiple systems with multiple GPUs and multiple CPU cores, to lower the final rendering time for very large models.

The major part of the rendering time is spent on mixed hierarchy traversal. The performance could be higher on a GPU if more registers are available on it. We traverse a block of neighboring rays together to provide higher ray coherence. With modern GPUs supporting L1 and L2 caches, coherence is essential for optimal performance. We significantly outperform the previous implementations and

achieve a speedup of 5-30 times over previous CPU or GPU implementations. With better initialization values, Newton's method required only 6-8 iterations to converge to the actual roots.

We used the hit-points and true normals generated in the primary step to solve for secondary rays. We are able to render images with shadows and reflection at interactive to near interactive rates. The structure of the algorithm makes it capable of handling multiple bounces and area light sources.

We investigated global illumination effects for Bezier surfaces. To the best of our knowledge, ours is the first implementation to render Bezier patches with advanced effects. We rendered images with advanced effects like path tracing, ambient occlusion, depth of field, motion blur and glossy surface, to produce better photo-realistic results. We deduced that ray coherence is necessary to optimally path trace massive number of rays. With lesser blur radius and higher number of rays, coherence is higher and hence time per ray improves.

The strength of our method is the exploitation of all computing power available on a system. All heavy operations are shared by both the CPU and the GPU, resulting in high speed. The data parallel formulation also enables its extension to area light sources, path tracing, etc. The traversal step is the critical step of our approach. The limited memory available on the GPU restricts our approach in the number of rays handled and the number of bounces traced.

The mixed hierarchy structure can be investigated for higher order parametric surfaces like NURBS. With few different fixed depth subtrees and capability of modern GPUs to invoke multiple kernels simultaneously, the algorithm could benefit ray tracing higher order parametric surfaces. The top level tree for the mixed hierarchy model is a BVH structure. BVH works well for both static and dynamic scenes. However, with current works on acceleration structure, it would be interesting to investigate other structures like KD-Trees, etc. to act as the top level structure. Rearrangement of rays to provide higher coherence would greatly enhance the performance of the algorithm. One can combine the shadow and reflection/refraction ray list, rearrange it and then apply the algorithm to render images at much higher rate. Coherence is even more essential in case of global illumination effects. Ray ordering should greatly reduce the rendering times of generated images. One can investigate faster and complex global illumination methods like photon mapping, bidirectional path tracing, etc. for Bezier surfaces. Parametric patches provide compact form of representation than polygonal mesh, leading to smaller acceleration structure and lower traversal times. One can expect them to level up with the polygonal mesh equivalent for ray tracing complex objects in near future.

Related Publications

- Hybrid Ray Tracing and Path Tracing of Bezier Surfaces Using A Mixed Hierarchy
  Rohit Nigam and P. J. Narayanan
  Proceedings of the Eighth Indian Conference on Computer Vision, Graphics and Image Processing, 2012.

Bibliography

[1] O. Abert, M. Geimer, and S. Muller. Direct and fast ray tracing of nurbs surfaces. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 161 –168, sept. 2006.

[2] D. V. Ahuja and S. A. Coons. Geometry for Construction and Display. *Ibm Systems Journal*, 7:188–205, 1968.

[3] A. Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[4] J. Arvo. Backward ray tracing. In *In ACM SIGGRAPH 86 Course Notes - Developments in Ray Tracing*, pages 259–263, 1986.

[5] C. Benthin, I. Wald, and P. Slusallek. Interactive ray tracing of free-form surfaces. In *International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction, South Africa*, pages 99–106, 2004.

[6] S. Campagna and P. Slusallek. Improving bézier clipping and chebyshev boxing for ray tracing parametric surfaces. In *Proceedings of 3D Image Analysis and Synthesis 96*, pages 95–102, 1996.

[7] E. Catmull and J. Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. *Computer-aided design*, 10(6):350–355, 1978.

[8] S. E. Chen, H. E. Rushmeier, G. Miller, and D. Turner. A progressive multi-pass method for global illumination. In *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '91, pages 165–174, New York, NY, USA, 1991. ACM.

[9] R. L. Cook. Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5(1):51–72, Jan. 1986.

[10] R. L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, Jan. 1984.

[11] A. Efremov, V. Havran, and H.-P. Seidel. Robust and numerically stable bezier clipping method for ray tracing nurbs surfaces. In *IN PROCEEDINGS OF 21ST SPRING CONFERENCE ON COMPUTER GRAPHICS*, pages 127–135. Press, 2005.

[12] C. Eisenacher, Q. Meyer, and C. T. Loop. Real-time view-dependent rendering of parametric surfaces. In *Proc. of Symposium on Interactive 3D Graphics, Boston, USA*, pages 137–143, 2009.

[13] M. Geimer and O. Abert. Interactive ray tracing of trimmed bicubic bézier surfaces without triangulation. In *Proc. of Winter School of Computer Graphics*, pages 71–78, 2005.

[14] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph.*, 18(3):213–222, Jan. 1984.

[15] H. Gouraud. Continuous shading of curved surfaces. *Computers, IEEE Transactions on*, C-20(6):623 – 629, june 1971.

[16] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek. Realtime ray tracing on gpu with bvh-based packet traversal. In *Proc. of IEEE/Eurographics Symposium on Interactive Ray Tracing, Germany*, pages 113–118, 2007.

[17] S. Guntury and P. J. Narayanan. Ray tracing dynamic scenes with shadows on the gpu. In *Eurographics Workshop on Parallel Graphics and Visualization*, 2010.

[18] M. Guthe, A. Balázs, and R. Klein. Gpu-based trimming and tessellation of nurbs and t-spline surfaces. *ACM Trans. Graph.*, 24(3):1016–1023, July 2005.

[19] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, and D. Manocha. Memory-scalable gpu spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):466–474, Apr. 2011.

[20] T. Ize, P. Shirley, and S. Parker. Grid creation strategies for efficient ray tracing. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 27–32, Washington, DC, USA, 2007. IEEE Computer Society.

[21] H. W. Jensen. Global illumination using photon maps. In *Proceedings of the Eurographics workshop on Rendering techniques '96*, pages 21–30, 1996.

[22] K. I. Joy and M. N. Bhetanabhotla. Ray tracing parametric surface patches utilizing numerical techniques and ray coherence. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 279–285, New York, NY, USA, 1986. ACM.

[23] J. T. Kajiya. Ray tracing parametric patches. *SIGGRAPH Comput. Graph.*, 16(3):245–254, July 1982.

[24] J. T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, Aug. 1986.

[25] E. P. Lafortune and Y. D. Willems. Bi-directional path tracing. In *PROCEEDINGS OF THIRD INTERNATIONAL CONFERENCE ON COMPUTATIONAL GRAPHICS AND VISUALIZATION TECHNIQUES (COMPUGRAPHICS 93*, pages 145–153, 1993.

[26] E. P. Lafortune and Y. D. Willems. A theoretical framework for physically based rendering. *Computer Graphics Forum*, 13(2):97–107, 1994.

[27] S. Lahabar. Exploiting the graphics hardware to solve two compute intensive problems: Singular value decomposition and ray tracing parametric patches. 2010.

[28] C. Lauterbach, M. Garl, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on gpus. In *In Proc. Eurographics 09*, 2009.

[29] R. R. Lewis, R. Wang, and D. Hung. Design of a pipelined architecture for ray/bezier patch intersection computation. In *Canadian Journal of Electrical and Computer Engineering*, volume 28, 2002.

[30] D. Manocha and S. Krishnan. Algebraic pruning: A fast technique for curve and surface intersection. Technical report, TR93-062, University of North Carolina at Chapel Hill, 1994.

[31] W. Martin, E. Cohen, R. Fish, and P. Shirley. Practical ray tracing of trimmed nurbs surfaces. *Journal of Graphics Tools*, 5:27–52, 2000.

[32] M. J. Muuss. To wards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium*, June 1995.

[33] T. Nishita, T. W. Sederberg, and M. Kakimoto. Ray tracing trimmed rational surface patches. In *Proc. of SIGGRAPH*, pages 337–345, 1990.

[34] H.-F. Pabst, J. Springer, A. Schollmeyer, R. Lenhardt, C. Lessig, and B. Froehlich. Ray casting of trimmed nurbs surfaces on the gpu. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 151 –160, sept. 2006.

[35] J. Pantaleoni and D. Luebke. Hlbvh: hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 87–95, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[36] S. Patidar and P. J. Narayanan. Ray casting deformable models on the gpu. In *Proceedings of the 2008 Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, ICVGIP '08, pages 481–488, Washington, DC, USA, 2008. IEEE Computer Society.

[37] A. Patney and J. D. Owens. Real-time reyes-style adaptive surface subdivision. *ACM Trans. Graph.*, 27(5):143:1–143:8, Dec. 2008.

[38] B. T. Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[39] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, Sept. 2007.

[40] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 703–712, New York, NY, USA, 2002. ACM.

[41] H. E. Rushmeier. *Realistic image synthesis for scenes with radiatively participating media*. PhD thesis, Ithaca, NY, USA, 1988. AAI8821226.

[42] P. Shirley. A ray tracing method for illumination calculation in diffuse-specular scenes. In *Proceedings on Graphics interface '90*, pages 205–212, Toronto, Ont., Canada, Canada, 1990. Canadian Information Processing Society.

[43] P. Shirley and R. K. Morley. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2 edition, 2003.

[44] F. Sillion and C. Puech. A general two-pass method integrating specular and diffuse reflection. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '89, pages 335–344, New York, NY, USA, 1989. ACM.

[45] W. M. Steven Parker, P. pike J. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. In *Symposium on interactive 3D graphics*, pages 119–126, 1999.

[46] G. Stoll. Part ii: Achieving real time-optimization techniques. *SIGGRAPH 2005 Course on Interactive Ray Tracing*, 2005.

[47] I. E. Sutherland. Sketchpad: A Man-Machine Graphical Communication System. In E. C. Johnson, editor, *Proceedings of the 1963 Spring Joint Computer Conference*, volume 23 of *AFIPS Conference Proceedings*, pages 329–346, Baltimore, MD, 1963. American Federation of Information Processing Societies, Spartan Books Inc.

[48] M. Sweeney and R. Bartels. Ray tracing free-form b-spline surfaces. *Computer Graphics and Applications, IEEE*, 6(2):41 –49, feb. 1986.

[49] N. Thrane, L. O. Simonsen, and A. Peter. A comparison of acceleration structures for gpu assisted ray tracing. *Master's thesis, University of Aarhus*, 2005.

[50] D. L. Toth. On ray tracing parametric surfaces. In *Proc.of Computer Graphics*, volume 19, pages 171–179, 1985.

[51] E. Veach and L. Guibas. Bidirectional estimators for light transport. In G. Sakas, S. Mller, and P. Shirley, editors, *Photorealistic Rendering Techniques*, Focus on Computer Graphics, pages 145–167. Springer Berlin Heidelberg, 1995.

[52] E. Veach and L. J. Guibas. Optimally combining sampling techniques for monte carlo rendering. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 419–428, New York, NY, USA, 1995. ACM.

[53] E. Veach and L. J. Guibas. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 65–76, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[54] I. Wald. On fast construction of sah-based bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 33–40, Washington, DC, USA, 2007. IEEE Computer Society.

[55] J. R. Wallace, M. F. Cohen, and D. P. Greenberg. A two-pass solution to the rendering equation: A synthesis of ray tracing and radiosity methods. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, pages 311–320, New York, NY, USA, 1987. ACM.

[56] S.-W. Wang, Z.-C. Shih, and R.-C. Chang. An improved rendering technique for ray tracing bézier and b-spline surfaces. In *Journal of Visualization and Computer Animation 11*, pages 209–219, 2000.

[57] G. J. Ward, F. M. Rubinstein, and R. D. Clear. A ray tracing solution for diffuse interreflection. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '88, pages 85–92, New York, NY, USA, 1988. ACM.

[58] T. Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, June 1980.

[59] C. Woodward. Theory and practice of geometric modeling. chapter Ray tracing parametric surfaces by subdivision in viewing plane, pages 273–287. Springer-Verlag New York, Inc., New York, NY, USA, 1989.

[60] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 papers*, SIGGRAPH Asia '08, pages 126:1–126:11, New York, NY, USA, 2008. ACM.

[61] K. Zimmerman and P. Shirley. A two-pass solution to the rendering equation with a source visibility preprocess. In P. Hanrahan and W. Purgathofer, editors, *Rendering Techniques 95*, Eurographics, pages 284–295. Springer Vienna, 1995.