Fast Raytracing of Point Based Models using GPUs

M.Tech Dissertation

Submitted by

Sriram Kashyap M S
Roll No: 08305028

Under the guidance of

Prof. Sharat Chandran

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai

2010

Abstract

Advances in 3D scanning technologies have enabled the creation of 3D models with several million primitives. In such cases, point-based representations of objects have been used as modeling alternatives to the almost ubiquitous triangles. However, since most optimizations and research have been focused on polygons, our ability to render points has not matched their polygonal counterparts when we consider rendering time, or sophisticated lighting effects.

We further the state of the art in point rendering, by demonstrating effects such as reflections, refractions, shadows on large and complex point models at interactive frame rates using Graphic Processing Units (GPUs). We also demonstrate fast computation and rendering of caustics on point models. Our system relies on efficient techniques of storing and traversing point models on the GPU.

Acknowledgement

I thank my guide Prof. Sharat Chandran for his invaluable support and guidance. I thank Rhushabh Goradia for his contributions to the project, and for his motivational influence. I also thank Prof. Parag Chaudhuri for his valuable inputs during our discussions. I thank the Stanford 3D scanning repository and Cyberware for providing high quality models to work with. Finally, I thank the VIGIL community for their continued support.

Table of Contents

List of Figures

Chapter 1

Introduction

3D objects in a scene can be represented in several ways. The most popular technique is to break the scene into many polygons and rasterize each polygon independently. With the increase in the complexity of geometry, points as primitives are increasingly being used as an alternative to polygons [PZvBG00, RL00]. These points can be thought of as samples from the surface that we want to model. As soon as triangles get smaller than individual pixels, the rationale behind using traditional rasterization can be questioned. Perhaps more important is the considerable freedom modelers enjoy with points. Point models enable geometry manipulation without having to worry about preserving topology or connectivity [ZPvBG01]. Simultaneously, modern 3D digital photography and 3D scanning systems [LPC+00] acquire both geometry and appearance of complex, real-world objects in terms of a large number of points. Points are a natural representation for such data.

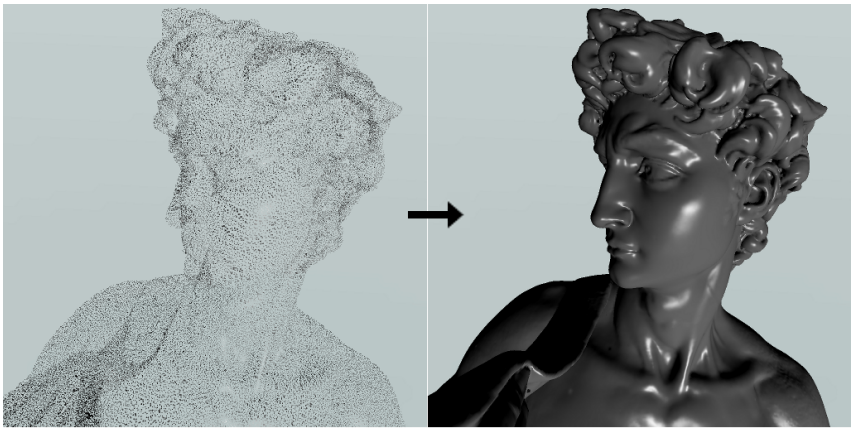

Figure 1.1: Rendering of point models.

While modeling and editing point based geometry is an interesting topic in itself [ZPK+02], we are concerned with rendering of points. Most of the research in this area, has so far has been devoted to rasterization of point models by various splatting techniques [LW85, KL04, PZvBG00, RL00, ZPvBG01]. Although this technique is straightforward and blends well with existing hardware architectures, the complexity of rendering is now linearly dependent on the complexity of the scene (which can be very large in the case of point models). Further, high-quality photorealistic effects such as shadows, reflections and refractions are increasingly harder to achieve.

1.1 Raytracing

The aforementioned drawbacks of rasterization can be resolved using a different approach to render point models, an approach known as raytracing. Raytracing is a technique where the scene is rendered by shooting rays out of the viewers eye, and identifying which parts of the scene each ray hits. These rays, known as primary rays, hit objects in the scene, which contribute to the color of the pixel through which the ray was shot. These rays can also reflect or refract from the surface of objects, thereby producing secondary rays which can hit other objects.Further, at each hit point, a test can be performed to check whether each light source in the scene is visible from this point, thereby producing accurate shadows. This technique is capable of producing a very high degree of photorealism, in a very succinct manner. It can simulate a wide variety of optical effects, such as reflection and refraction, caustics and scattering.

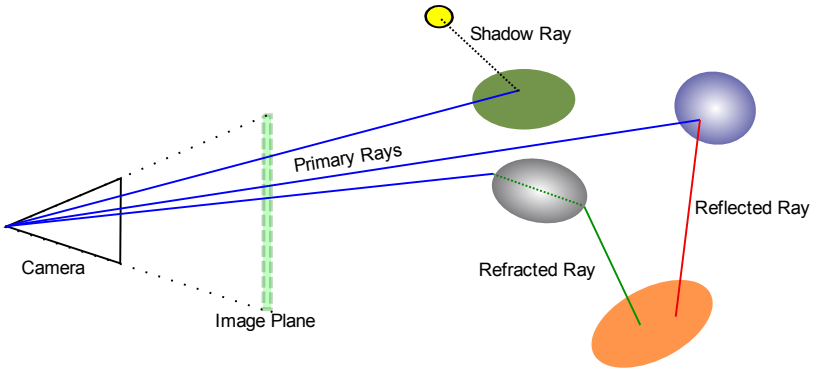

Figure 1.2: Illustration of Raytracing.

Raytracing is the natural choice when it comes to rendering scenes containing large amounts of geometry. This is because there are algorithms to traverse rays through a given scene in time that is proportional to the logarithm of the scene complexity. The actual time taken to raytrace an image is governed primarily by the resolution at which we are rendering the image (since there are more rays to trace). More importantly, raytracing is a succinct technique which models various interesting light transport phenomena like reflections and refractions in a simple manner. It is much harder to render such effects in a rasterization framework, because rasterization does not simulate light rays in the scene. Instead, the effects are rendered in a contrived and non-intuitive manner that usually results in coarse approximations and places additional burden on artists.

Raytracing has traditionally been a slow and memory intensive process, used in applications where quality is more important than speed, such as rendering photorealistic stills and films. In realtime applications like games and walkthroughs, Graphics Processing Units (GPUs) are used to perform hardware accelerated triangle rasterization, which is typically an order of magnitude faster than raytracing. However, with the advent of high speed, fully programmable GPUs, this speed gap is closing fast. General purpose programming languages for GPUs, such as CUDA and OpenCL have inspired several projects geared towards fast raytracing on commodity graphics hardware.

1.1.1 Raytracing Point Models

The key challenge faced while rendering (both rasterization and raytracing) point models is that they do not contain connectivity information. This means we have to deal with an ill defined surface representation. To handle this issue, three main approaches for raytracing point models have emerged.

The first one proposed by [SJ00] and [WS03] used ray-beams and ray-cones respectively to perform intersections with singular points. [WS03] introduced a similar concept of tracing ray-cones into a multi-resolution hierarchy. These approaches can only be used for offline rendering, as tracing ray-cylinders or cones requires one to traverse large portions of the acceleration data-structure, thereby increasing the computation time.

The second approach is to raytrace implicitly constructed point-set surfaces. [AA03] proposed this method where the intersection of the rays was performed with the locally reconstructed surfaces. It resulted in an computationally expensive algorithm. This approach was improved by [WS05], with an interactive ray-tracing algorithm for point models.

The third approach, presented by [LMR07, KGCC10], is to raytrace splat models i.e. grow the point primitive such as to cover a small area around it (disks or ellipse). This approach is conceptually quite simple compared to the others.

1.2 Problem Statement

This project aims to develop a fast GPU raytracer that can render the surface represented by a given list of points and associated normals. The raytracer should support reflection, refraction, shadows, caustics and other interesting light transport phenomena.

1.3 Contributions

1. We design a GPU friendly, memory efficient, variable height octree. This enables us to perform fast ray traversal in large scenes. We also design an efficient mechanism for several threads on the GPU to trace rays in parallel.

2. Since no explicit surface representation is available for point models, a surface representation must be defined to support ray-object intersections. We formulate techniques to raytrace a splat based point representation. To overcome certain issues associated with splat based raytracing, we also develop a GPU friendly surface representation based on implicit surfaces.

3. Using the above traversal technique and surface representations, we develop a GPU raytracer for point models that supports reflections, refractions and shadows at interactive frame rates.

4. We demonstrate fast generation of caustics as an application of our raytracer.

5. We present a simple technique to texture point models.

We use the NVIDIA Compute Unified Device Architecture (CUDA) to program the GPU. Details on the architecture are presented in §7.1. Note that all experiments are performed on a machine with 2.6 GHz Core 2 Quad processor, 8 GB DDR3 RAM, and an NVIDIA GTX 275 GPU with 896 MB RAM. All images are rendered at 1024×1024 unless otherwise stated.

1.4 Overview

The remainder of this document is organized as follows:

- Parallel raytracing on the GPU: Here we describe the data structures and algorithms used to accelerate raytracing on the GPU.
- Splat based raytracing of point models: We present methods to efficiently store splat data in a ray-acceleration hierarchy, and discuss the speed/quality tradeoffs that are possible while raytracing splats.
- Implicit surface representation of point models: We design a GPU optimized data structure for rendering surfaces represented by point models, which gives us better speed and quality than the splat based approach, at the expense of more precomputation.
- Local and global shading: We describe our lighting and shading model, texturing, and the generation and rendering of caustics.

Chapter 2

Raytracing on GPUs

Raytracing fundamentally aims to find the first object that a ray hits while traversing a scene. Performing this search in a brute force manner involves checking each ray for intersection with each primitive in the model. While this is acceptable for simple models, for models with millions of primitives (as is the case with point models), this method is extremely slow. To speed up this process, the primitives of the 3D model are stored in a special data structure generally referred to as an acceleration structure. We describe how such a structure can be implemented on the GPU.

While raytracing on both CPUs and GPUs can benefit from an acceleration structure, a GPU raytracer has to tackle a host of other issues:

1. Lack of recursion: Raytracing, which is inherently recursive, should be iteratively formulated.

2. Ray Coherence: Rays that take similar paths should run in the same multiprocessor, so as to reduce warp divergence.

3. Memory latency: Raytracing is a memory bound problem, and GPU main memory suffers from very high latency.

In this chapter, we describe how these challenges have been overcome. We describe the octree structure used to traverse the scene, the data structures used to encode scene information on the GPU, the fast ray traversal algorithm, and a technique to improve ray coherence.

GPU programs are written in the form of "kernels", which are pieces of code that run in parallel on different data units. As a result, there are several possible GPU raytracing architectures. We implement two of these approaches, a monolithic kernel approach and a modular approach, and discuss their advantages and shortcomings.

2.1 Related Work

With the introduction of fully programmable GPUs, there has been considerable interest in GPU based raytracers in recent years. [CHH02] develop a raytracer which uses CPU to perform ray traversal and pixel shaders on the GPU to perform ray-triangle intersections. [CHCH06] present a raytracing technique called Geometry Images where the scene geometry is encoded

in images and processed on graphics hardware. They demonstrate effects such as reflection and depth of field. [HSHH07] implement an interactive GPU raytracer based on stackless kd tree traversal. All of the above techniques are developed with polygonal models in mind.They are typically limited to scenes with about 100K to 500K polygons. More recently, [CNLE09] have implemented a realtime GPU raycaster for out-of-core rendering of massive volume datasets. They stream large datasets from secondary memory onto the GPU and demonstrate up to 60 frames per second (fps) for raycasting, and around 30 fps when shadow rays are generated. They are easily able to achieve soft shadows and depth of field due to their multi-resolution representation of volume data. Octrees are used to store voxels in a multi-resolution hierarchy. Top-down octree traversal is performed to trace rays through the scene.

More recently, [LK10] introduced the Sparse Voxel Octree, a data structure specialized to render surfaces. They improve the performance of GPU raytracing by introducing beam optimizations, where they calculate the approximate starting point of primary rays in batches, instead of computing it each time for every ray. [GL10] propose a new raytracing pipeline for GPUs, based on ray sorting and breadth-first frustum traversal. They report up to $2\times$ speedup over conventional raytracing, for primary rays. When ray divergence is high, the cost of frustum traversal and sorting becomes very high and their method becomes slower than conventional raytracing.

2.2 Octree – Acceleration Structure on GPU

As described earlier, when tracing huge number of rays for image synthesis, it is not efficient to test for intersection of each ray with all objects in the given scene. It is therefore necessary to construct a data structure that minimizes the number of intersection tests performed. We partition the space containing the primitives into a hierarchical octree data structure, each node being a rectangular axis-aligned volume.

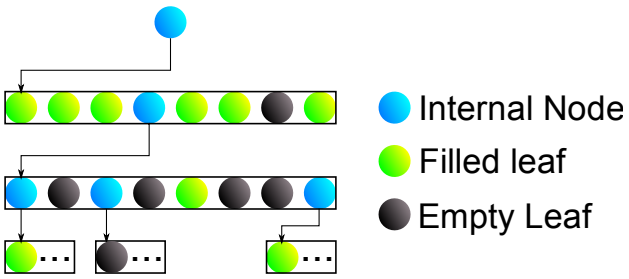

Figure 2.1: Representation of an octree. Each internal node contains eight children (representing the eight octants)

The root node of this octree represents the entire model space. The model space is recursively divided into eight octants, each represented as an internal node, an empty leaf or a filled leaf. If the current node is divided further, its an internal node. If it does not have any splats in it, it is an empty leaf, else its a filled leaf. Every internal node in the octree has 8 children.

The octree structure is stored on the GPU as a 1D-texture. We use textures because of the texture cache, which enables faster data access than if we used the GPU global memory directly. Currently, 1D textures in CUDA allow for 2^{27} elements to be accessed, allowing us to create

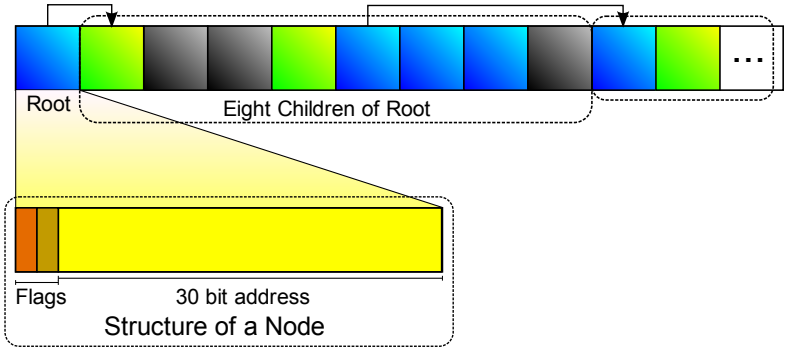

Figure 2.2: Octree Node-Pool in texture memory of the GPU.

large octrees. Each location in the texture is called a texel. A texel is generally used to store color values (RGBA). CUDA allows each texel to contain 1, 2, or 4 components, of 8 or 32 bits each. Thus a texel can store up to $128 - bits$ of data. We make use of these texels to store the octree nodes instead of color values. All the nodes that the octree contains are stored in a specific texel in this texture. Each node can be indexed with an integer address. This array of nodes stored in the texture is called the *Node-Pool*.

It is desirable to minimize the storage per node so that we can store larger octrees on limited memory, and also to improve the texture cache hit-miss ratio. We store each octree node as a single 32 bit value. The first 2 bits are used as flags to represent internal nodes (00), filled leaves (10) and empty leaves (01). The remaining 30 bits are used to store an address. In case of internal nodes, this address is a pointer to the first child node. Filled leaves use these 30 bits to store a pointer to the data that they contain. In the case of empty leaves, these 30 bits are left undefined.

Further, all the 8 children of any internal node are stored in a fixed order. We make use of a local Space Filling Curve (SFC) ordering amongst the children. We store no other extra information in this octree, to keep it as memory efficient as possible. Note that we do not require a parent pointer, as we never have to traverse upwards in the octree. Also, each type of node actually has different information stored in it. We handle this by storing additional information in separate textures/arrays.

2.2.1 Ray Traversal

The main parallelism in ray-tracing stems from the fact that each ray is independent from the other ray. To exploit this, we spawn as many threads on the GPU, as there pixels to render. So, we can think of each ray in the raytracer, as a single GPU thread.

For rays starting from outside the octree, the intersection of the ray with the bounding box of the root node is computed. We increment this intersection point by a small ε in the direction of the ray, so that it is inside the octree. The leaf node to which this point belongs is determined. This node now becomes the current node.

If the current node is a filled leaf, we check if the ray intersects any of the contained objects/primitives. If the ray does not intersect any of the objects stored in that node or if the node is empty, we find the point where this ray exits the current node, and increment it by a small

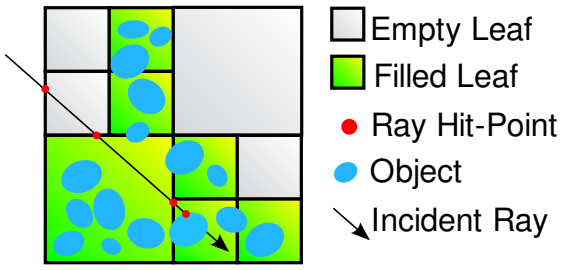

Figure 2.3: Tracing a ray through the acceleration structure.

number, ε, in the ray direction. If this point is outside the root node of the octree, it means that the ray left the scene. Otherwise, we find the leaf node corresponding to this point and iterate this entire process till the ray hits an object or exits the scene.

2.2.2 Top-down traversal

As mentioned above, each traversal starts from the root node. During the traversal, the current node is interpreted as a volume with range $[0,1]^3$. The ray's origin $p \in [0,1]^3$ (3 is the dimension) is normalized such that it is located within this node. Note that p is a vector with (x,y,z) components, each normalized to the range $[0,1]$. Now, it is necessary to find out which child-node n is contains ray-origin p. Thus if R is the current node, we let $p \in [0,1]^3$ be the point's local co-ordinates in the node's volume bounding box, and find child node n containing p. If c is the pointer to the grouped children of the node in the node pool, the offset to the child node n containing p is simply $SFC(2*p)$, where $SFC()$ of a vector is defined as:

$$SFC(V) = floor(V.x)*4 + floor(V.y)*2 + floor(V.z); \tag{2.1}$$

Now it is necessary to update p to the range of the newly found child node and continue the descent further. The appropriate formula to update p is:

$$p = p*N - int(p*N) \tag{2.2}$$

The new p is the remainder of the $p*N$ integer conversion. Now the traversal loops until a leaf (or an empty node) is found. After finding the node containing the point, the algorithm continues as mentioned in § 2.2.1.

2.2.3 Neighbor Precomputation

Note that we traverse the octree structure from the root each time we want to query a point in the octree. When a ray exits from one leaf and enters another, we are effectively performing a neighbor finding operation in the octree. Instead of doing this each time a ray hits a node, we could instead pre-process the tree so that the neighbor information is readily available for the ray, and we no longer have to begin our search from the root node. We can do this with the addresses of the six neighboring nodes that are at the same level or higher than the current node. Using this structure, we were able to rapidly traverse through the octree without need for a full top-down traversal at each step. Although this method can be faster than our current

method, we chose the existing traversal model for its simplicity of representation. Currently, we store 4 bytes of data to represent each octree node. The information about node boundaries is not stored at each node. Instead, it is stored at the root level, and dynamically reconstructed for each node in the tree, as we traverse down the tree.We would need 16 additional bytes to store center and side length of each node. Further, to store the neighbor information, we would require an additional 24 bytes of data, bringing the total to 44 extra bytes of storage, which is 11x the current requirement.

We evaluated neighbor precomputation on a CPU raytracer and found it achieves around 30 percent speedup depending on the scene complexity. We are able to achieve a similar speedup on the GPU by storing the octree data as a texture. The texture cache is optimized for locality of reference. The octree texture stores level 1 nodes, then the level 2 nodes, and so on. Since level 2 nodes are stored close to the level 1 nodes in the tree, when we read level 1, level 2 is loaded into texture cache, and so on. Thus, the first few levels of octree lookups are essentially free, thereby giving us an advantage similar to neighbor precomputation. Note that this does not work for lower levels because the data is too large to fit into the texture cache.

2.3 Ray Coherence

One of the standard techniques used to speed up ray tracing is that of coherent rays. Multiple rays that follow very similar paths through the acceleration structure are said to be coherent with respect each other. CPU raytracers generally exploit coherence through the use of SIMD units, where ray-bundles or ray-packets are traced at a time by packing multiple rays inside SIMD registers and performing the same operations on all the rays in a packet, but essentially reducing processing time by a factor proportional to the SIMD width (usually 4 on current CPUs).

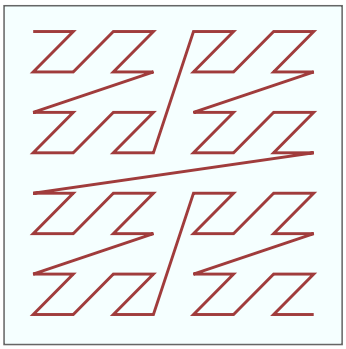

Figure 2.4: The Z-Order Space Filling Curve. source: Wikipedia

A similar trick can be employed on GPUs. NVIDIA GPUs process threads in batches of 32, called “warps”. As explained in §7.1, all threads in a warp are bound by the instruction scheduler to take the same path. This means that it makes sense for us to assign coherent rays to threads in a warp. Threads are assigned to warps based on their thread-ids. CUDA threads are provided with a unique thread-id which they can use to identify which part of the data they are working on. The first 32 thread-ids are assigned to the first warp, the next 32 to the second and so on. We can exploit coherence of rays by mapping this linear thread-id to rays, in a

spatially coherent manner. The obvious mapping function that comes to mind is the Z-Curve, a type of space filling curve (SFC). A given linear ordering of thread-ids can be converted to the corresponding Z-order by using this simple mapping function:

x = Odd-Bits(Thread-ID)
y = Even-Bits(Thread-ID)

The (x,y) pairs can be obtained in constant time from a given thread-id. These pairs denote the location on the screen through which the ray originates. A clear visual picture of the z-ordering can be seen in Fig.2.3. A 30% performance boost can be obtained by assigning the rays to threads using the Z-order, as opposed to assigning rays using linear sweeps across the screen.

2.4 Raytracing Architecture

As described earlier, GPU programs are written in the form of "kernels", which are pieces of code that run in parallel on different data units. We can think of each ray as being processed by a separate thread. The simplest approach to building such a raytracer is to write a monolithic kernel that reads rays from a ray-buffer and processes them to produce color values on the screen. This is similar to the approach taken while building CPU raytracers.

This architecture is not recommended while programming on GPU. The reason is that a large kernel increases the register and memory footprint of each thread. This means that fewer threads can be scheduled on each GPU multiprocessor. Since the GPU hides memory latency by scheduling as many threads as possible, having fewer threads can increase the impact of memory latency on the running time. NVIDIA calls this phenomenon "occupancy". Occupancy is the ratio of number of threads that have been scheduled on each multiprocessor, to the theoretical limit of number of threads that *can* be scheduled on each multiprocessor. If the number of registers required is less than or equal to 16, then the kernel has an occupancy of 1. For larger kernels, CUDA imposes a limit of 60 registers and spills the rest of the register requirement onto GPU global memory. This causes occupancy levels of 0.25 or lesser.

To circumvent this problem, the recommended model is to spread the computations over multiple simple kernels (Fig. 2.4). The advantage is not only that we increase occupancy, but the code also becomes easy to manage and modular. The disadvantage is that after a kernel terminates, the GPU "forgets" its internal state. So the results of computations that have to be carried over from one kernel to another, have to be written to global memory. What we observe in practice is that the occupancy advantage of smaller kernels is more or less nullified by the added overhead of transferring data between kernels. In some cases, this overhead causes an overall slowdown. This means that rather than indiscriminately splitting kernels into smaller pieces, we should achieve a compromise between the modularity offered by smaller kernels, and the data transfer overhead that comes with them. The relative time taken by each of these stages in the pipeline has been reported in Fig. 2.4.

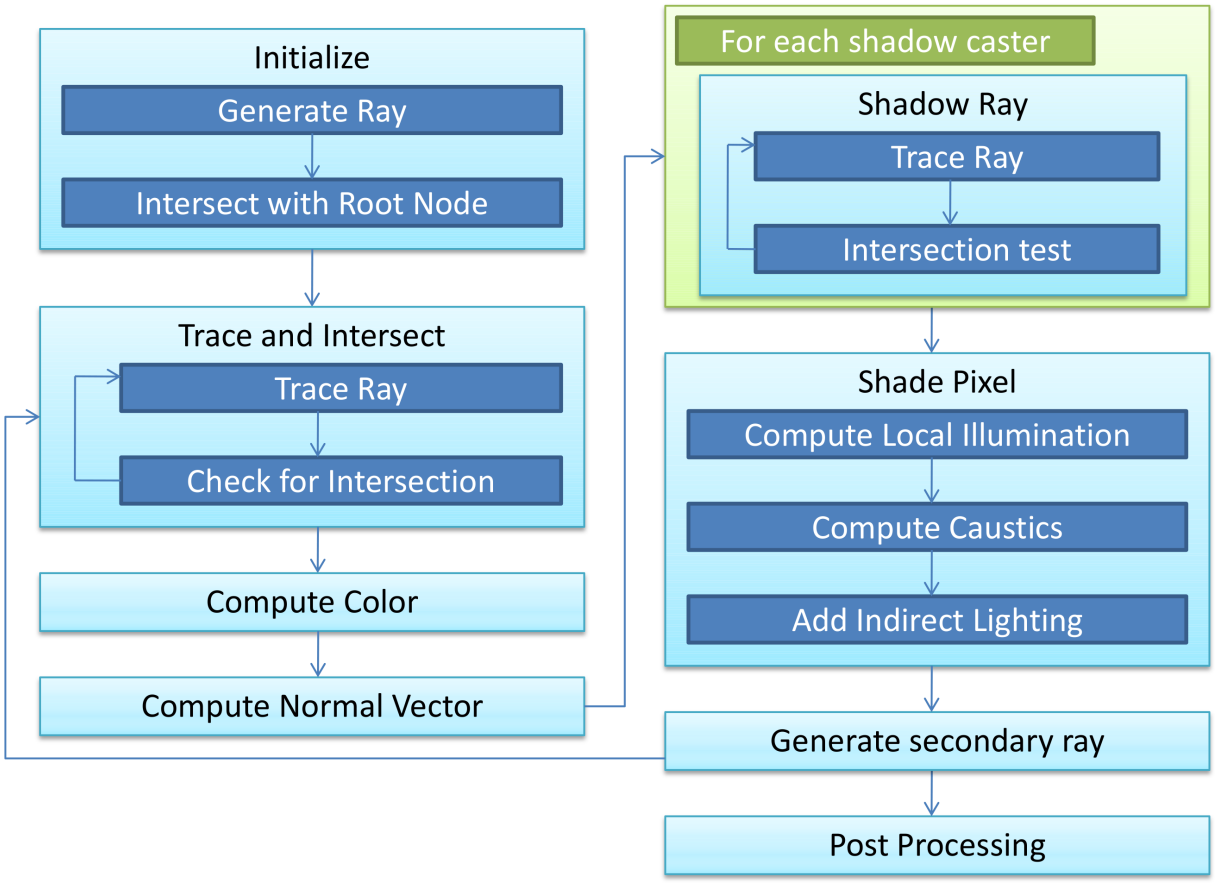

Figure 2.5: Modular Architecture. Each light blue box is a GPU kernel. A single trace and intersect kernel was found to be faster than a separate trace kernel followed by an intersect kernel, both looping on the CPU. The color and normal computations were placed in their own kernels due to the large number of registers required to perform normal interpolation and color interpolation. The CPU loops (green box) over each shadow casting light, and calls a shadow ray kernel similar to the trace and intersect kernel. The only difference is that the intersection test is simpler since we don't need exact hit parameters. After shading, the secondary ray kernel computes reflected or refracted rays and the whole process starts over again. If no secondary ray is generated, the post process kernel is invoked. This kernel was added to tone map the floating point image buffer generated by the raytracer to a 24 bit per pixel buffer that can be displayed by conventional monitors. Other post processing effects can be added here as well.

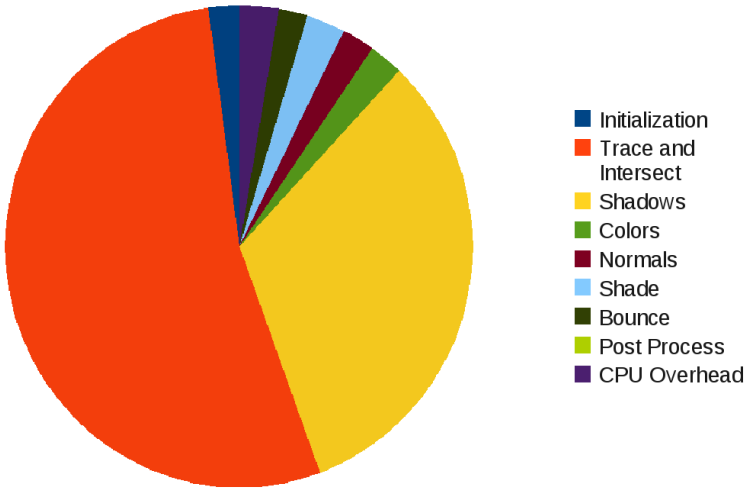

Figure 2.6: Relative time taken by each stage of the raytracing pipeline. A major chunk of the time is taken in tracing rays through the scene. The chart shows timings for a scene with a single shadow casting light source. Since shadow computation also involves tracing, it takes up a significant chunk of the time. The sections of the chart are labeled counter clockwise, starting with Initialization. It can be seen that the post process kernel takes the least time. This kernel currently only clamps floating point color values to 3 byte color values. The CPU overhead is mainly due to the OpenGL window management and copying the CUDA output buffer to a Pixel Buffer Object for display. Note that this chart has been generated using the surface representation method described in Chap. 4, since this method takes up the least ray-surface intersection time.

Chapter 3

Splat Based Raytracing

3.1 Splats

As mentioned in § 1.1.1, the main issue with raytracing point models is that both points and rays are entities with zero volume, and to calculate the intersection of rays with a point sampled surface, one, or both have to be "fattened" in some way.

A popular solution is to assign a finite extent to each point [PZvBG00, ZPvBG01, LMR07]. Apart from the position and normal, we associate a radius with each point sample. This forms an oriented disk that we refer to as a *splat*. The radius is chosen such that each point on the original surface is covered by the footprint of at least one splat.

In a splat based approach, it is important to obtain good splat sampling and correct splat radii so that the resultant model is hole free [WK04]. In this work, we utilize a simple technique based on finding the first 'k' near neighbors of each point (typically around 9 neighbors). The distance to the farthest near neighbor is chosen to be the radius. The rationale is that each point sample is typically surrounded by 8 to 9 other samples on the surface.

Given a model consisting of splats as defined above, we sub-divide the model space using an adaptive octree (leaves appear at various depths). The root of the tree represents the complete model space and the sub-divisions are represented by the descendants in the tree; each node represents a volume in model space.

3.2 Point Data Structure on the GPU

As mentioned in § 2.2, the input point data is stored in a 1D texture referred to as the data pool. Every point is the data pool has the following attributes:

- Co-ordinates of the point (x, y, z – 3 floats)
- Normal at the point defining the local surface (n_1, n_2, n_3 – 3 floats)
- Radius of the splat around the point (r – 1 float)
- Material Identifier, which stores the material properties like color, reflectance, transmittance etc of the point (mID – 1 float)

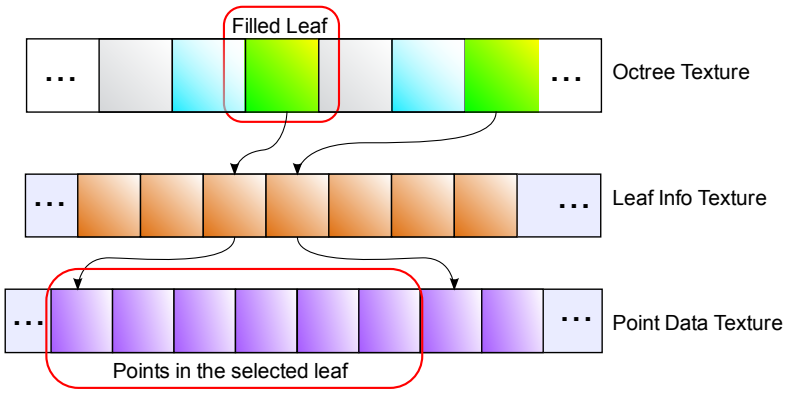

Figure 3.1: A Filled Leaf in the Node-Pool, and its associated data.

Each float occupies 4 bytes (32-bits) in memory. Thus, with every point we need to store a total of 8 floats or 32 bytes. Note that each texel can hold a maximum of 128-bits of information. So, we store each point as 2 consecutive texels to store one single point.

A filled leaf can contain several points. This is due to the way we construct the octree. We divide the octree in a fashion such that each leaf contains a maximum of X points (usually, X is 10 to 50). Thus, to quickly access all the X points belonging to the specific leaf, we store them contiguously in memory.

Each filled leaf should know where its point data is stored. Point data is stored in a separate point data texture § 3.2. The starting and ending point indices for each filled leaf are stored in a third 1D-texture, which can also store any additional information that a filled leaf node would require. This texture is referred to as the Leaf-Info texture. Each texel in the Leaf-Info texture is a 32 bit value representing the beginning index of a leaf node's data block. The immediate next texel gives the data block end for that specific leaf. The leaf in the node pool stores the location of this texel in the Leaf-Info texture.

To access the point data after reaching a filled leaf, we follow the leaf pointer to this texture where we obtain the begin and end of this leaf's data block. Note that we need not explicitly store the end location for a leaf, as the value in the next texel is the begin for some other leaf and end for the current leaf. Thus the size of the texture is equal to number of filled leaves in the tree.

3.3 Ray-Splat intersection

Once we hit a leaf node, we need to check which of the splats in this node intersects the incoming ray. At a high level, ray-splat intersections are handled as ray-disk intersections. The key difference is that in splat based models, rays can hit multiple splats on the surface, and it now becomes necessary to interpolate the parameters at these points. We choose to record the position of each hit, and the surface normal at that point. We then perform a weighted average of these parameters to obtain the final hit position and normal vector. The weights for averaging are inversely proportional to the distance of the intersection point on each disk, from the centers of those respective disks. As shown in Fig. 3.3, the final hit point is calculated as follows. Let

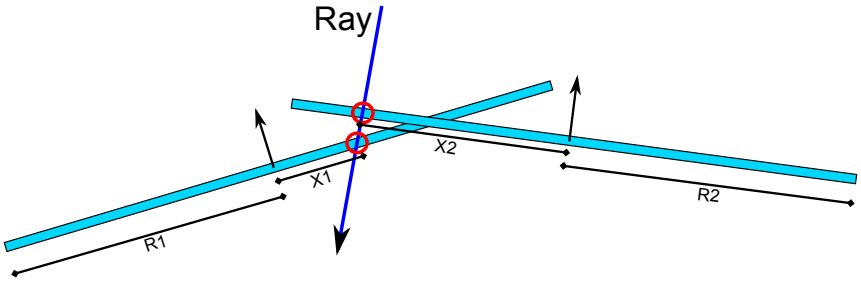

Figure 3.2: Ray-splat intersection. A single ray can intersect multiple splats due to overlapping splats on the surface.

the hit-point on the splats be V_1 and V_2 respectively. Then, the final hit location is given by:

$$V = \frac{V_1(1-\frac{X_1}{R_1}) + V_2(1-\frac{X_2}{R_2})}{(1-\frac{X_1}{R_1}) + (1-\frac{X_2}{R_2})} \tag{3.1}$$

Other attributes like the surface normal, color and material properties can be similarly blended at the point of intersection.

After the interpolated hit information has been obtained, the the material-id of the nearest splat stored in the current cell is retrieved. This material id is used to shade the pixel according to the local illumination at this point, and also to send out secondary rays (reflection, refraction and shadows). After each reflection or refraction, the current ray intensity is reduced by the coefficient specified in the material file. Finally, the color value at any given pixel is multiplied by the current ray intensity (starts from 1.0), and added to an accumulator. Ray traversal stops when the ray hits a diffuse surface or when it has bounced more than b times, b being some pre-defined maximum bounce limit set by the user.

3.4 Splat Replication and Culling

The octree construction algorithm described in § 2.2 adds the centers of each splat to the octree. This can cause problems when the splat center is inside a particular node, but the splat itself spans multiple neighboring nodes (as seen in Fig. 3.3). In such situations, a ray can pass through a node, without hitting a splat that it should have hit, because the node does not know that it contains that particular splat. This can cause rendering artifacts in the form of holes in the model. To fix this, [LMR07] have proposed a method where the octree is constructed using splat centers, and a second pass is performed where the splats are added to any leaf nodes that they intersect. This second pass does not change the structure of the octree. This method causes individual splats to be replicated several times. Typically, we could expect about $5\times$ to $10\times$ increase in the number of splats. When the initial number of splats is close to a few million, this increase can be drastic in terms of memory usage. On the CPU, an obvious workaround is to replicate pointers to the data, rather than the actual data itself. This approach is not suitable on GPU implementations because the actual data is now scattered all over the memory. A bandwidth starved system such as the GPU cannot efficiently access random locations in mem-

ory, and we find that storing the point data of each leaf node in contiguous locations offers a significant performance boost (around 30 percent).

To mitigate these effects, we propose a technique by which we can reduce the total number of splats to around $2\times$ the original, in most cases.

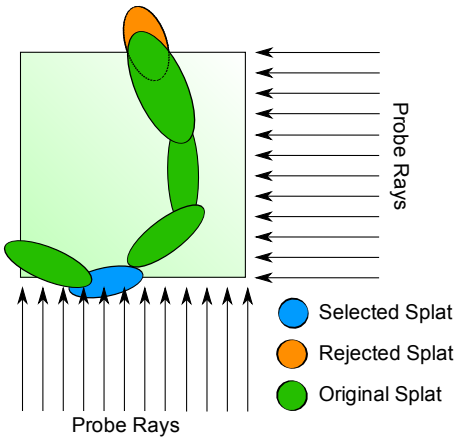

Figure 3.3: Culling replicated splats. The splats that were originally in the node are shown in green. The blue splat was selected, while the orange splat was culled (no probe rays hit it)

The key idea here, is that by adding extra splats, we are solving the problem of holes in the model. After the splat replication process, if we can run some kind of post process to find out which of these splats are really contributing towards patching up holes, then we can remove other redundant splats. We do this by first adding all splats to all nodes that they intersect, and then pruning away redundant splats. We take each leaf node that contains splats, and raytrace it from various viewpoints using a set of probe rays. We first check for the ray-splat intersection with the original splats of the leaf node.

If a probe ray goes through the leaf node, but does not hit any of its original splats, we check if the ray hits any of the newly replicated splats. If the ray now hits a replicated splat, we increment a counter associated with the splat, which tells us how many rays have hit this replicated splat. In the end, we retain only those replicated splats that have been hit at least once by a probe ray (Fig. 3.4).

Note that aggressive splat culling can lead to artefacts in the form of incorrect surface normals, and in extreme cases, holes in the surface (Fig. 3.4). While holes can be fixed by using more conservative culling (by increasing the number of probe rays), the surface normal errors cannot be avoided. The reason for this is that culling tries to ensure that each part of the surface is represented by some splat. It does not account for the fact that the normal at that point on the surface can be properly reconstructed only if all the splats that contribute to the surface are blended together. Thus splat culling is suitable for raycasting, but produces noticeable artefacts if used with reflections and refractions.

Figure 3.4: Impact of splat culling on image quality. The images on the left are rendered with splat culling (2 million splats total, at 6.5 fps), the images in the middle are without splat culling (11 million splats total, at 2 fps), and on the right are 2× difference images. The original model is 1 million points. While splat culling gives very good memory savings and increases frame rates (Fig. 3.5), the reduced quality of silhouettes and normals is clearly visible in a close-up shot. This makes splat culling a good candidate for models that are viewed from afar.

3.5 Deep Blending

Working with normals is particularly tricky when specular objects are present. Consider the situation shown in Fig. 3.5. Splats S_1 and S_3 are associated with Leaf A. However, since the octree is a regular space partitioning technique, it fails to record the presence of S_2 being related to Leaf A. These intersections are quite important for normal blending, as can be seen in Fig. 3.5.

To circumvent this problem, [LMR07] generates a complete normal field over the surface of each and every splat. Storing this normal field with every splat implies a large memory footprint, thus making it infeasible for use on the GPU.

Our solution to this problem is a two pass approach illustrated in Fig. 3.5. In the first pass, we iterate over all the splats in the current node (Leaf A in the current example) and determine a *reference splat*. This is the splat whose intersection point with a candidate ray R is closest to its center (in the example, the ray is R_1 and the reference splat is S_1). We now consider all intersecting splats along the ray, within a small interval (based on input point data) around the reference splat intersection point. This generally involves accessing splats from *only* the next node along the ray direction. This process is a bit computationally intensive (Fig. 3.5), but it

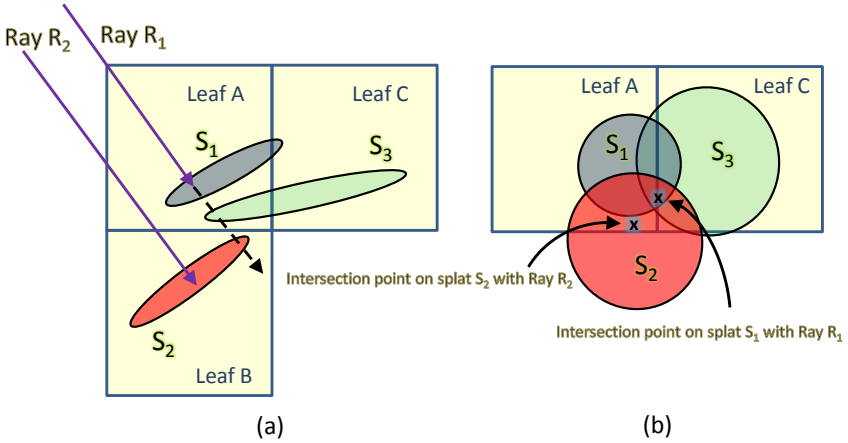

Figure 3.5: A 2D side view of selected splats is shown in (a). Ray R_1 hits splat S_1 and S_3. S_3 has been added to Leaf A even though the splat center lies outside leaf A. However, the ground truth (top view (b)) indicates that even S_2 represents the same surface, and has simply been placed in Leaf B due to the spatial quantization of the octree. We note that ray R_1 intersects all three splats and hence all three should be considered for blending, otherwise the difference in normals with a neighboring ray R_2 is too large.

Figure 3.6: (a) Incorrect normal blending, (b) Correct normal blending, (c) Difference image.

significantly increases the quality of results (Fig. 3.5).

3.6 Seamless Raytracing

The algorithm as described above works perfectly in a ray-casting environment, where only primary rays are used. In case of secondary reflections/refractions and shadow rays, the algorithm begins to break down because of the problems associated with overlapping splats.

Consider the situation in Fig.3.6. In this case, a ray that gets reflected can hit a splat that is really part of the same surface, and keep getting reflected multiple times along the same surface. This can cause the raytracer to slow down drastically, and also produce shading artifacts on reflective surfaces. The same problem manifests itself in the case of refraction, as shown in Fig.3.6. Here, a ray that gets refracted can hit another splat (or potentially several other splats)

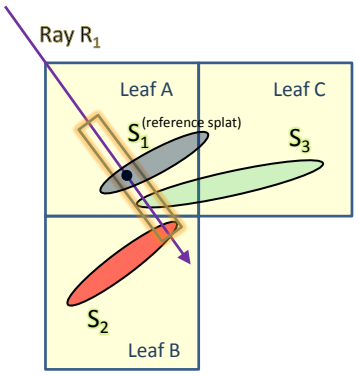

Figure 3.7: Splat S_1 is considered as a reference splat. We then record all intersections of the ray R_1 with splats in a small interval around the reference splat intersection point (highlighted by a rectangle). Splat S_2 is thus considered for blending.

on the same surface, thereby producing incorrect results. All prior techniques to handle this inherent problem of splat based raytracing have in some way or the other involved the use of a "minimum distance between intersections". This means that if a ray encounters an intersection point that is within some distance δ of its source, this intersection is ignored. This trick breaks down in cases where there are sharp corners in the scene, or when the point models are dense and complex, where reflections occur within very short distances of each other, as illustrated in Fig.3.6. The consequence of this is that we can see "seams" in the raytraced output at regions where we would expect to see reflections (like at the corners of a room).

To prevent this problem, we introduce the concept of "Seamless Raytracing" of splat models. This is essentially a technique that allows us to prevent incorrect collisions with splats from the same surface. The idea here, is to associate some "intelligence" with each ray that is being traced in the scene. Let us assume that a ray can tell what kind of surface it must intersect next, a front face, or a back face. Front face intersections occur most of the time on the outer surfaces of objects. Thus, a ray, when it begins its life, expects to hit a front face. This is of course, assuming that the ray is not starting inside an object. Back facing intersections occur during refraction, when the ray enters an object and hits its surface from *inside*. We can represent these situations by associating a flag with each ray in the scene. If the flag is set, the ray expects to hit a front face, and will ignore all back facing hits. Similarly, if the flag is not set, the ray expects to hit back facing surfaces. We can find out what kind of intersection occurred, by looking at the dot product of the ray direction with the surface normal at that point. If this value is positive, we have a back facing hit, and if it is negative, we have a front face hit.

All rays start with this bit set. A reflection does not affect this bit. A refraction on the other hand, flips the bit. It is easy to see how all the problem cases listed above are easily handled by this system. Multiple splats belonging to the same surface have similarly oriented surface normals.

The effect of this on raytracing is illustrated in Fig.3.6. The corners of the reflective room are clearly incorrect when using the δ threshold method. Using seamless raytracing, we are able to correctly reproduce the reflections in corners.

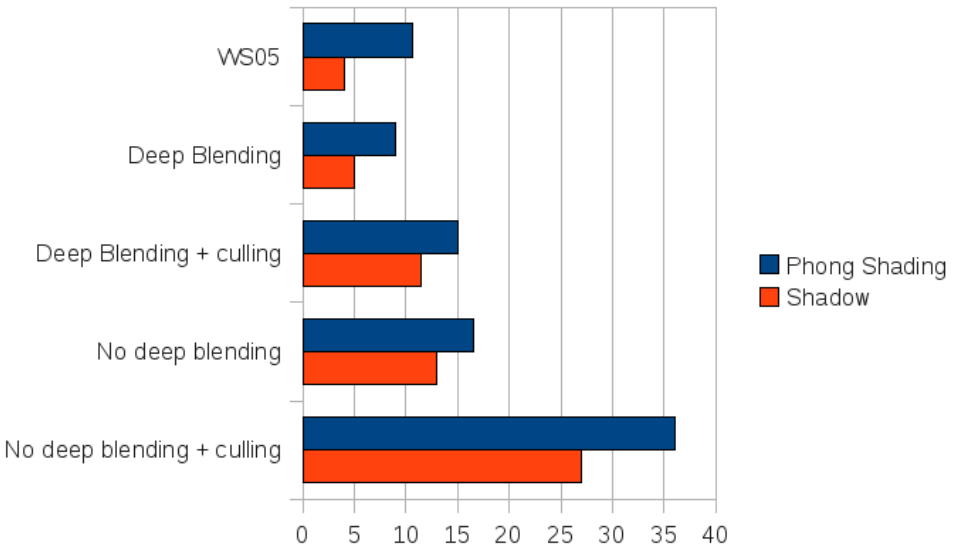

Figure 3.8: Timing comparison of various methods for 1 million point model of David, rendered at 512×512. X axis denotes frames per second. Note that while the frame rate increases drastically with splat culling and without deep blending, the image quality suffers.

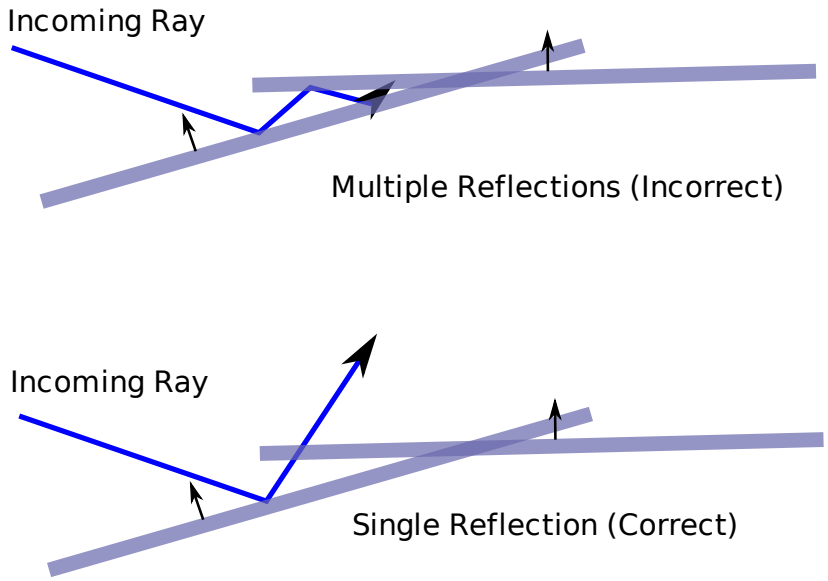

Figure 3.9: Incorrect reflections due to overlapping splats.

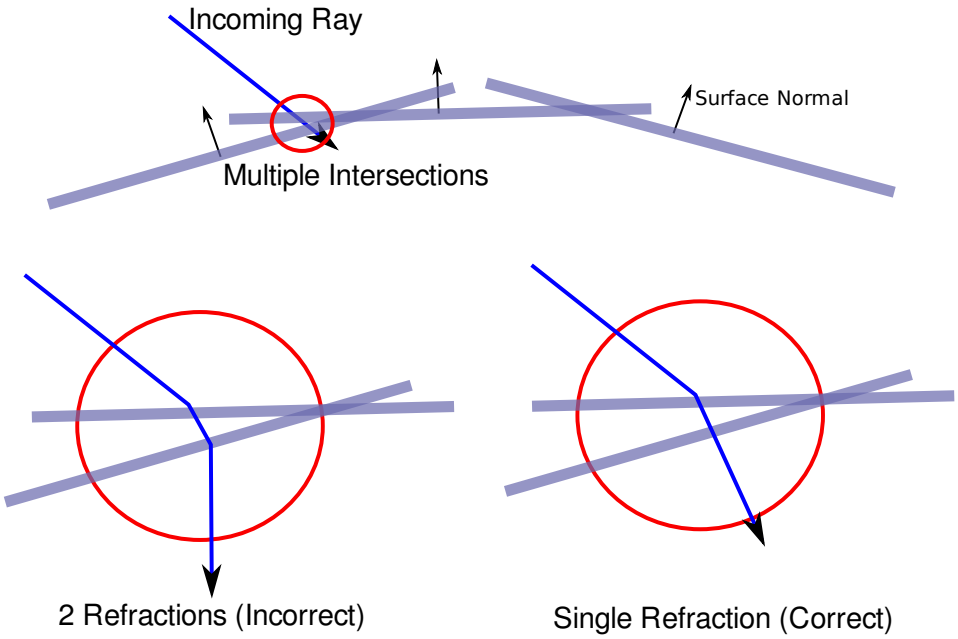

Figure 3.10: Incorrect refractions due to overlapping splats.

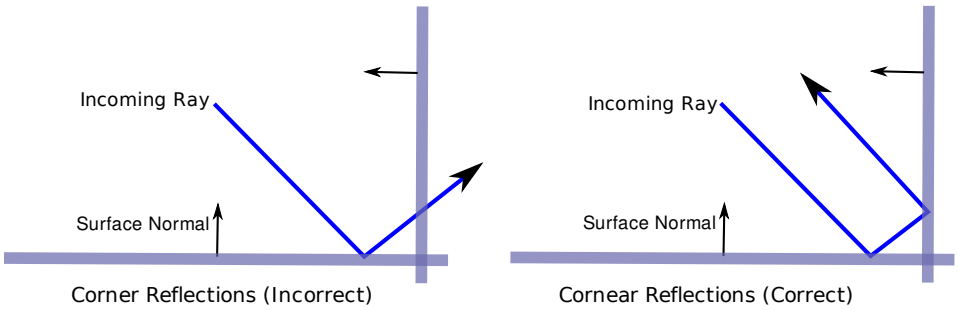

Figure 3.11: Incorrect reflection due to the “minimum distance between intersections” trick.

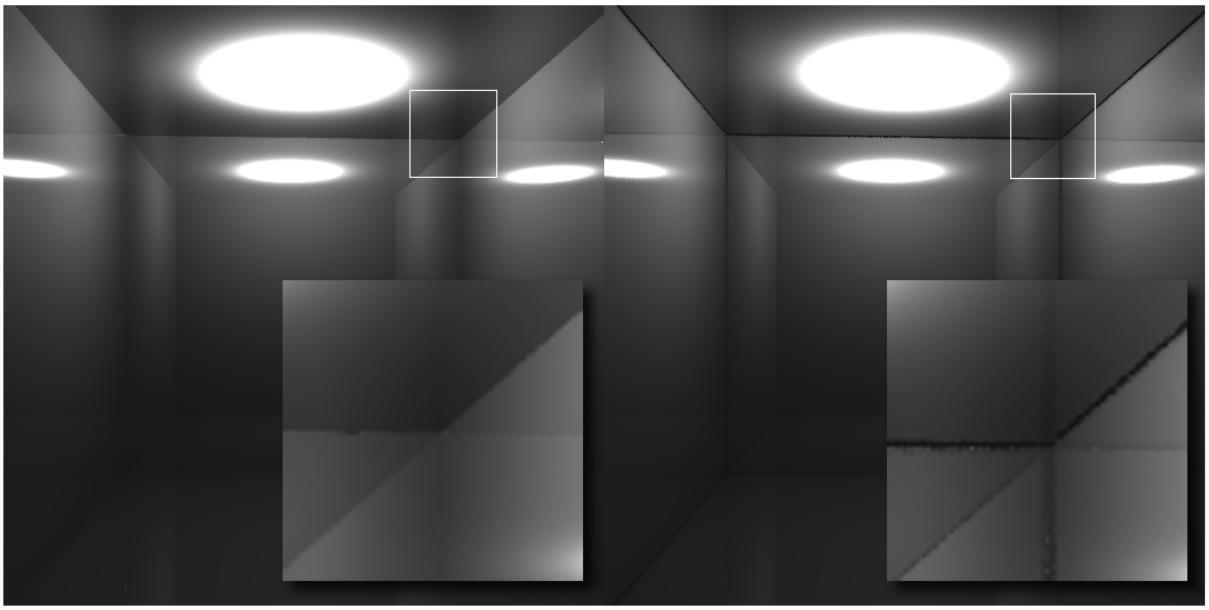

Figure 3.12: Seamless raytracing (left) vs. regular raytracing (right). Notice how regular raytracing is unable to capture the effect of reflection at corners. Since the reflected ray strikes a surface that is too close to its origin, it assumes that this surface should be ignored, and moves into the background, thus generating a black patch. Seamless raytracing is able to identify this case because the reflected ray strikes a front facing surface. This intersection is registered as a valid hit, and we can observe correct reflections at the corners.

Chapter 4

Implicit Surface Octrees

While splat based raytracing is intuitive and simple, it has its share of shortcomings. The most common issue with splat based methods in any rendering system is that splats do not properly represent edge boundaries. At sharp corners and regions of high curvature, the splats protrude out of the surface and cause object silhouettes to be incorrectly rendered. At edges, the splat blending step fails because rays intersect only one splat (Fig. 4). In a rasterization pipeline, this problem is usually handled using point-and-edge models [ZK07] or clip lines [ZRB+04]. These methods require some way to mark the edges in the model and are not general enough to work on arbitrary models. [Bal03] develop techniques to find edges at run-time, but we do not follow this direction, since what they are introducing is really a new rendering paradigm to reduce shading costs.

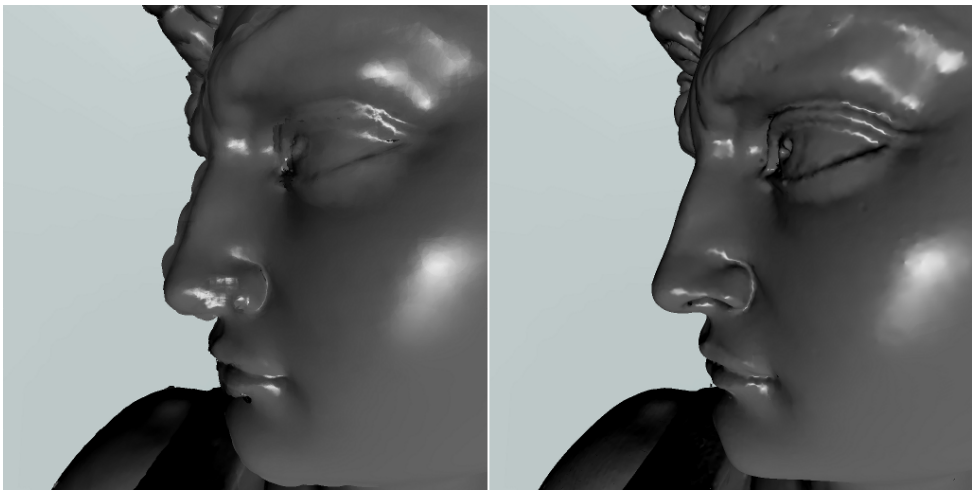

Figure 4.1: Result of splat based raytracing compared to a reference image produced by our implementation of [WS05]. The splat render contains border artefacts where splats are clearly visible.

Therefore, to produce high quality renderings of point data, we need to revisit the surface approximation method used in the raytracer. A more accurate reconstruction can be obtained by defining a moving least squares (MLS) approximation to the surface as in [WS05]. The method involves defining a signed distance field in space, by using the points, their normals and their

radii. The surface is then the zero-isosurface of this distance field.

4.1 Related work

Implicit surface representations of point models have been raytraced in [AA03] and more recently in [WS05], as described in §1.1.1. We implemented [WS05] on the GPU and found that it runs exceedingly slowly. The main reason for this is that each leaf can contain an arbitrary number of splats, that we have to load at runtime and compute the implicit surface definition. Since we cannot dynamically allocate memory on the GPU, we need to load these splats from memory each time we want to evaluate the function. The recommended number of evaluations (as used in [WS05]) is 4, which means we need to iterate through the point data 4 times per leaf node. The authors solve this problem by restricting the number of splats stored in each leaf to a small number (3 to 4). This requires a large number of subdivisions, which in turn increases the number of splats due to replication across nodes. On the GPU, we are limited to around 700 MB of memory and cannot afford to arbitrarily replicate data. Further, as explained in §3.4, we cannot mitigate replication cost by replicating pointers instead of actual data. Due to these reasons, our implementation of this technique runs about 3 to 4 times slower than the deep splat blending based raytracer.

An alternative approach, [KWPH06] uses adaptive-resolution octrees to interactively render isosurfaces. The distance field values are available as input volume data. The method works by sampling the volume at octree leaf centers, and using neighbor finding operations in a min-max tree to find the eight nearest samples and perform trilinear interpolation. It uses a CPU implementation, citing the GPU's incapabilities to handle memory-heavy models.

The use of octrees for defining and rendering surfaces can also be see in [LK10]. The technique uses polygonal models as input to generate a Sparse Voxel Octree. The representation power of a voxel is enhanced by storing two planes in each voxel, such that the surface inside the voxel is bounded by these planes. This allows rendering of sharp corners and thin surfaces, using voxels of relatively large size (12 to 13 levels of an octree)

4.2 Defining the implicit surface

Given a collection of points, the implicit surface approximation can be computed as follows:

Each point $P = (p_i, n_i, r)_{i=1}^{N}$ is defined by its position p_i, normal n_i and its local radius of influence r. We assume the surface to be smooth manifold and orthogonal to the normal within this radius r, although multiplied by a decreasing weight function (gaussian in our case)

$$w_i(P) = \frac{1}{\sqrt{2\pi r^2}} e^{\frac{-\|x-p_i\|^2}{2r^2}} \tag{4.1}$$

We can then define a weighted average of position and normal of neighboring points.

$$\bar{p}(x) = \frac{\sum w_i(x) p_i}{\sum w_i(x)}, \quad \bar{n}(x) = \frac{\sum w_i(x) n_i}{\sum w_i(x)} \tag{4.2}$$

This gives us a function that implicitly defines a surface at query point Q:

$$f(Q) = (Q - \bar{p}(x))\bar{n}(x) \tag{4.3}$$

The root of this function gives us a locally smooth surface at Q.

4.3 Implicit Surface Octree

An Implicit Surface Octree (ISO), is an octree where each leaf node contains a surface patch (or, the leaf could be empty). This surface patch is defined as follows. We consider each of the eight corners of every octree leaf as a query point Q. At each query point, we evaluate the isovalue from the neighboring points using the method described in §4.2. The radius of the search is set equal to the maximum radius of influence of any point in that leaf. A positive isovalue means that Q is outside the surface while a negative value means it is on the inside. A zero value indicates the point is present on the surface. We similarly calculate the average weighted normals at the corners.

Every octree leaf now has a set of 8 function values evaluated at its corners. We can define a smooth surface within each leaf, by trilinearly interpolating the isovalues and normal values stored at the leaf's corners. This is similar to the approach followed in [KWPH06]. Given these values, we no longer require the original points and thus, just the ISO suffices.

The construction of the ISO data structure is a pre-computation stage. The ISO is then transferred to the GPU during the start of ray tracing. We organize the input points in an octree, as described in §3.1 and §2.2. The ISO construction takes 60 seconds for the 1 Million point model of David, and 15 seconds for the 0.5 Million point model of the Dragon.

4.3.1 Pseudo Point Replication

We construct an octree only over the input point locations $p_i(i = 0...N)$ only without considering their respective radii of influence r_i. Leaves which contain points are termed as *filled* and *active* (Leaves 1 and 3 in Fig. 4.3.1). However, there are leaves which are within the points influence but do not contain the point itself (Leaf 2 in Fig. 4.3.1). We term such leaves as *passive*. By our method, isovalues would not be calculated for leaf 2 as it does not contain any point and hence termed as an "empty leaf". To prevent such holes, we need to find passive leaves and calculate the isovalues at their corners. The finding of such passive leaves is a simple test of box-disk intersection (disk representing the point's radius of influence) while recursing through the octree. Note that we never replicate the points themselves in passive nodes. Once we find a passive node, we simply change its status from "empty" to a "filled" leaf and calculate the respective isovalues.

4.4 GPU Octree Structure

The octree structure on GPU is similar to that described in §3.2. The main difference is that we do not store the Leaf-Info and Point Data textures. Instead, each leaf node in the octree refers to a corresponding set of values in a data pool. Each entry in the data pool stores the isovalues, the normals, and the color values at the corners of the corresponding leaf node. This organization is described in Fig.4.4.

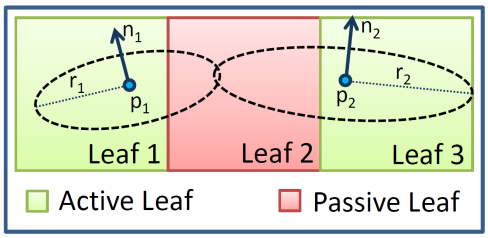

Figure 4.2: Point p_1 (with normal n_1 and radius of influence r_1) is located in leaf 1 and point p_2 (with normal n_2 and radius of influence r_2) in leaf 3. Leaves containing the points (1 and 3) are termed as "filled" or *active leaves*. Leaf 2 does not contain any points (is "empty") but is under the radius of influence of both points p_1 and p_2. Such leaves are termed *passive leaves*. Passive leaves are a part of the surface but due to point sampling, are considered empty. In such a situation, if a ray passes through leaf 2, it would not perform any surface intersections, resulting in a hole in the surface. We detect such passive leaves and calculate isovalues at its corners so that the surface is properly defined within it, while maintaining continuity across neighboring leaves.

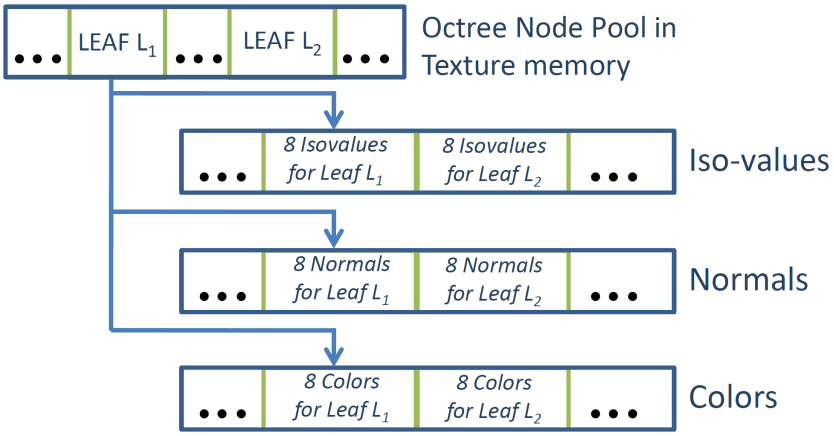

Figure 4.3: Figure shows the links between the node and the data pools. The leaves from the node pool point to a structure of arrays containing the 8 iso-surface, normal and color values evaluated previously at its corners.

To reduce the memory footprint, we can quantize the isovalues, normals and color values, so that they occupy one byte per component. Thus each node will have eight isovalues (8 bytes), eight normals (24 bytes) and eight color values (24 bytes), for a total of 56 bytes. Quantization artefacts are not noticeable in the case of isovalues and normals, because of the trilinear interpolation that smooths out values in space. The only issue that arises is that color values can no longer contain high dynamic range (HDR) data.

Note that each corner value is shared between eight octree nodes. We could store only unique values separately, and then have eight pointers in each node, to index into this array of values. We instead choose to replicate the data in each node. The main reason for this is that memory lookups are slow. We would have to perform a second level of indirection if we stored

pointers to data instead of actual data. Secondly, data quantization reduces the impact of this data replication. Each corner stores 7 bytes of information, as opposed to a pointer that would require 4 bytes per corner, plus the actual 7 bytes of storage. In the best case, the pointer method would consume $8 \times 4 + 7 = 39$ bytes, while the replication method consumes 56 bytes, which is a 43 percent increase. While this sounds like a good improvement, in practice, since we store leaf nodes only at the surface, the average number of leaf nodes sharing a corner around 4. This reduces the memory advantage of the pointer method to around 21 percent, which is offset by the 25 percent speedup that data replication provides.

4.5 Ray Surface Intersections

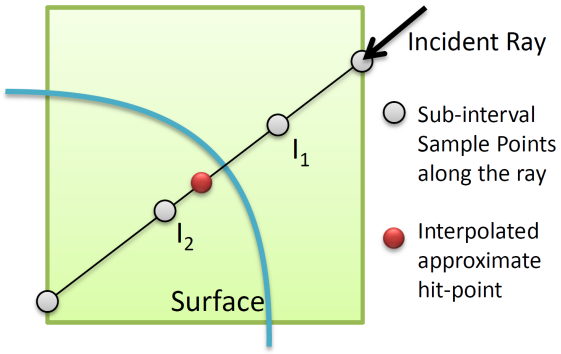

Figure 4.4: Figure shows a ray hitting a surface within the leaf. The surface hit-point is found by marching along the ray within the leaf, and evaluating the surface value at the sub-interval sample points (shown in gray). On a sign change in the evaluated isovalue, we do a weighted interpolation between the positive and negative sample points (I_1 and I_2 in this figure) to approximately find the hit point (shown in red).

On finding a filled leaf along the ray direction, we proceed to find whether the ray intersects the surface defined in this leaf. The ray is sampled at regular intervals within the leaf (Fig. 4.5). At each sample, we use trilinear interpolation to compute the isovalue from the values stored at corners. If we detect a sign change between consecutive samples (say I_1 and I_2), we know that the surface exists between $I_1 - I_2$. We now do a weighted interpolation of positions of I_1 and I_2 using their respective isovalues as weights (the sample having value closer to 0 has more weight than the other since its closer to the surface defined by isovalue 0). The simple interpolation routine makes ray-surface intersection extremely light on memory and computations.

To perform smooth shading and generation of secondary rays (shadows, reflection and refractions), we need correct normals at the intersection points. As we did with isovalues, we interpolate the normals from the corners of the leaf, to obtain the normal at the intersection point. This maintains smoothness within the leaf (due to interpolation) and across the leaves (due to the pre-computed normals being shared across adjacent nodes). If storing normals is not feasible, an approximate normal can be calculated for the same computational cost by calculating the gradient of the isovalues at the intersection point.

4.5.1 Continuity

When adjacent leaf nodes share a common surface but are at different levels, a discontinuity is produced in the surface. This is because the larger nodes sample the surface at larger intervals while smaller nodes have a more accurate sampling of the surface.

To ensure continuity across adjacent nodes, we can restrict all leaf nodes to be formed at the same level (similar to voxels). This approach tends to be wasteful in regions of low curvature, like floors and walls. An alternative approach is to run a post-process on the ISO to ensure that adjacent leaf nodes that share a surface, do not differ by more than one level. In case they do, we subdivide the nodes such that this condition is satisfied. This subdivision can be performed in an iterative manner by checking for inconsistent leaf nodes and subdividing the bigger leaf. If the maximum level difference between leaf nodes in the ISO is L, then we need L iterations to converge to an ISO that satisfies the constraint. What we observe in practice is that in ISOs with seven or more levels, this constraint ensures that the seam between leaf nodes at different levels is very thin. We can now patch this seam by allowing rays to hit surfaces that are slightly outside a leaf node. This slightly extrapolates surfaces outside leaf nodes, and covers up any seams that may exist (Fig. 4.5.1).

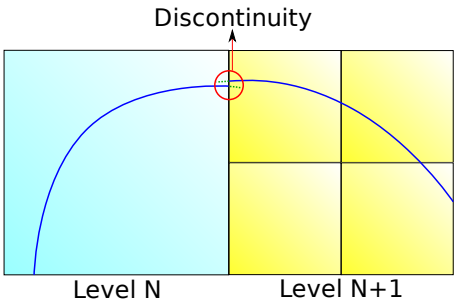

Figure 4.5: Discontinuity due to difference in adjacent leaf levels. The dotted green lines show the extrapolated surface which patches up the hole caused by the discontinuity. Similar discontinuity also exists in normal and color interpolation, but cannot be perceived in practice. This is the reason that the patched up surface appears to be continuous and smooth.

4.5.2 Seamless Raytracing

When secondary rays and shadow rays are involved, the ray-surface intersection test must also be carefully considered for reasons other than normal blending. Note that we have two different interpolations while determining the ray-surface hit point. The first interpolation deals with trilinearly interpolating isovalues from the corners of the leaf. The second interpolation (linear) takes place between the interval values (I_1 and I_2 in Fig. 4.5) where the sign change occurs and the surface is detected. This interval approximation gives us an approximate hit point, which can be below or above the actual iso-surface (Fig. 4.5.2). If the surface is specular, the reflected or the refracted ray generated from that hit point might hit the same surface again. This can cause the raytracer to slow down drastically, and also produce shading artifacts on reflective or refractive surfaces.

One way to solve this problem is to find the root of the iso-surface function using iterative methods ([WS05, KWPH06]) rather than linearly interpolation. But this method tends to make the ray-intersection routine heavy. Instead, this problem can be solved using the seamless raytracing technique described in §3.6.

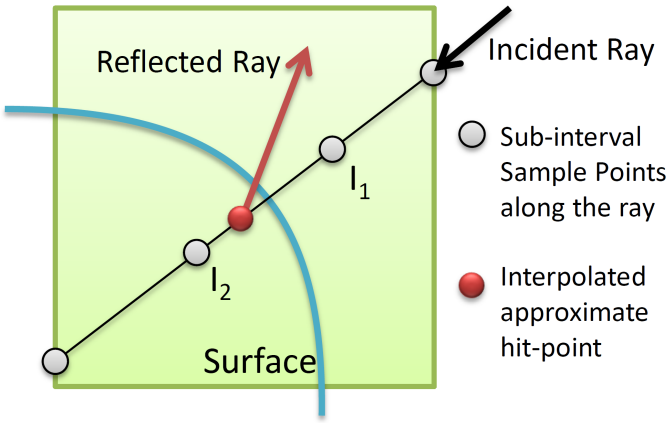

Figure 4.6: Approximate intersection point and the need for seamless raytracing

4.6 Results

Implicit Surface Octrees provide an interesting alternative to render point models. The method outperforms all previous methods that raytrace point models on the GPU, at a given rendering quality. To illustrate this, we reproduce the splat-based raytracer timings graph (given in Fig. 3.5), and add the ISO results to it. Fig 4.6 shows that the ISO renderer running at full quality, is as fast as the splat based raytracer at a low quality setting (with all speed optimizations turned on). We see the same trend in the results for the Dragon model (Fig. 4.6).

One observation we can make here is that the size of the ISO can be essentially independent of the actual number of points in the model, thereby allowing us to render the same model at various levels of detail (Fig. 4.6). This can be very useful while dealing with limited memory GPUs, since we can render very large point data sets by building an ISO at lower levels of detail. This enables us to handle large datasets (like the 12 million point Sponza Atrium), which the splat based raytracer cannot load into memory. Our implementation allows us to set the minimum and maximum levels at which leaf nodes are created, thus allowing us to control the level of detail.

At the beginning of this chapter, we compared the splat based raytracing approach to our reference implementation of [WS05]. This implementation uses the entire point data set, and has no approximations. So we consider it as a reference renderer and compare its results to the ones produced by ISO. (Note that while this reference renderer can produce high quality images, it is very sensitive to splat footprint (radius) and produces images with several holes in most cases. Still, it works reasonably well for the 1 million point model of David.) As can be seen in Fig. 4.6, our method produces results comparable to the reference render. It slightly smoothens out the surface (this is natural, since our method is an approximation), but runs

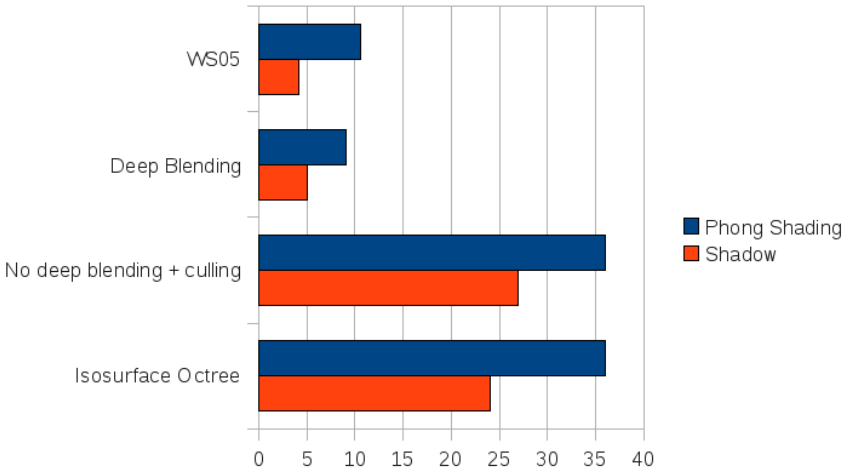

Figure 4.7: Timing comparison for 512×512 render of 1 Million point model of David.

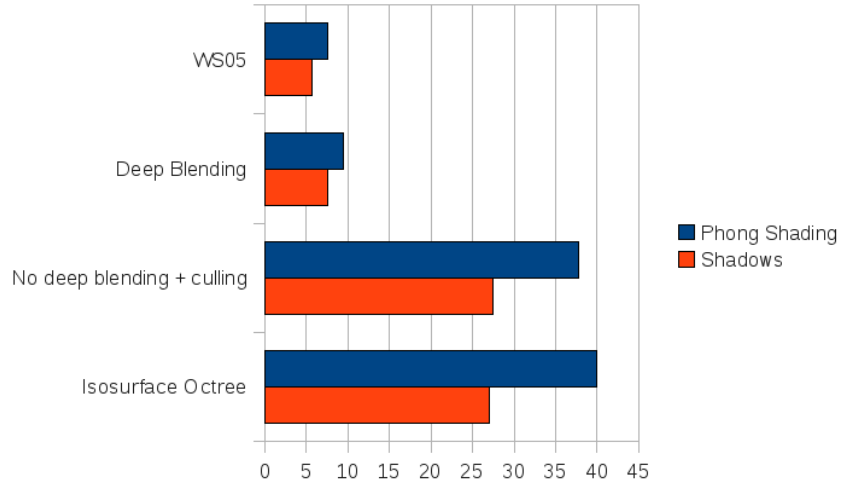

Figure 4.8: Timing comparison for 512×512 render of 0.5 Million point model of Dragon.

around $17\times$ faster on the GPU. We achieve close to 10 FPS at 1024×1024, while the reference renderer runs at 0.5 fps. Further, we consume lesser memory since the ISO has only 1,087,966 leaf nodes, while the reference renderer needs to replicate the splats $11\times$ (from 1,001,943 to 11,052,273). For models larger than this, we run out of GPU memory on our GTX 275 (896 MB), while ISO only occupies around 84 MB on the GPU. While we can attempt to reduce the memory footprint of the reference renderer by using quantization similar to that used in ISOs (§4.4), the frame rates will not show any significant improvement.

ISOs can be seen as a more expressive form of voxels since they define a smooth surface inside each voxel. This can be seen in Fig. 4.6, where we render a 4 level deep ISO which is equivalent to a $16 \times 16 \times 16$ voxel grid. With such a small ISO, we are able to render a full torus with smooth normals, as can be seen from the refractions of the background.

ISOs also provide a form of smoothness control. The implicit surface definition (§4.2)

Figure 4.9: Dragon model at various levels of detail.

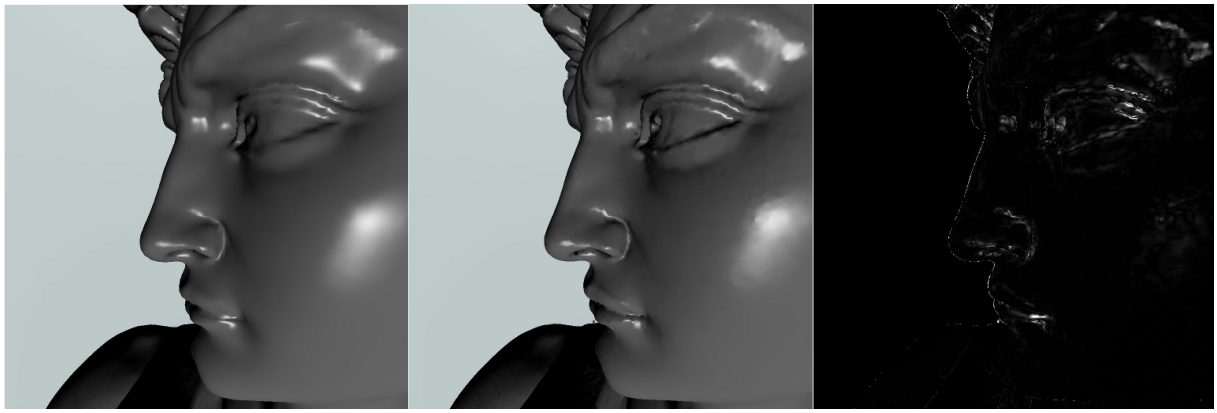

Figure 4.10: Left: ISO rendering. Middle: Reference render. Right: 2× difference image

contains a radius term in the weight function. The radius acts as the standard deviation in the gaussian function. If we multiply the radius by a constant factor, we can control the variance of the weight function and change how sharp or smooth the model appears. Reducing the variance will make the model sharper, while increasing it makes it smoother, as can be seen in Fig. 4.6.

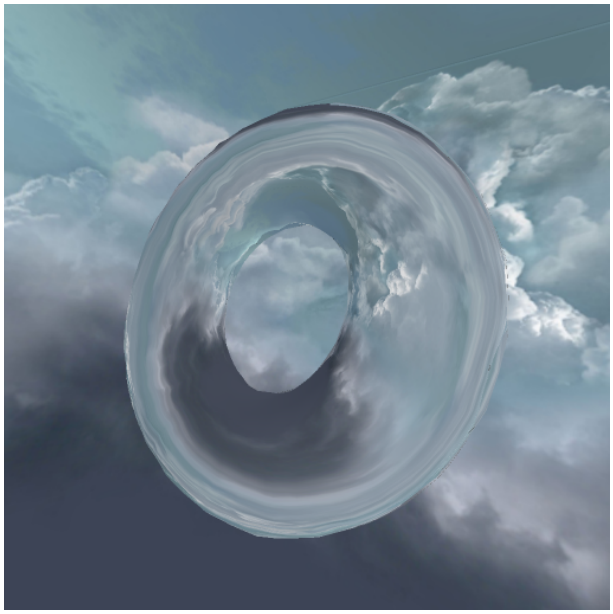

Figure 4.11: Expressive power of Implicit Surface Octrees. A refractive torus is rendered against an environment map. The ISO is only four levels deep. Since each leaf node has a smooth surface definition, the torus can be rendered with so few leaf nodes.

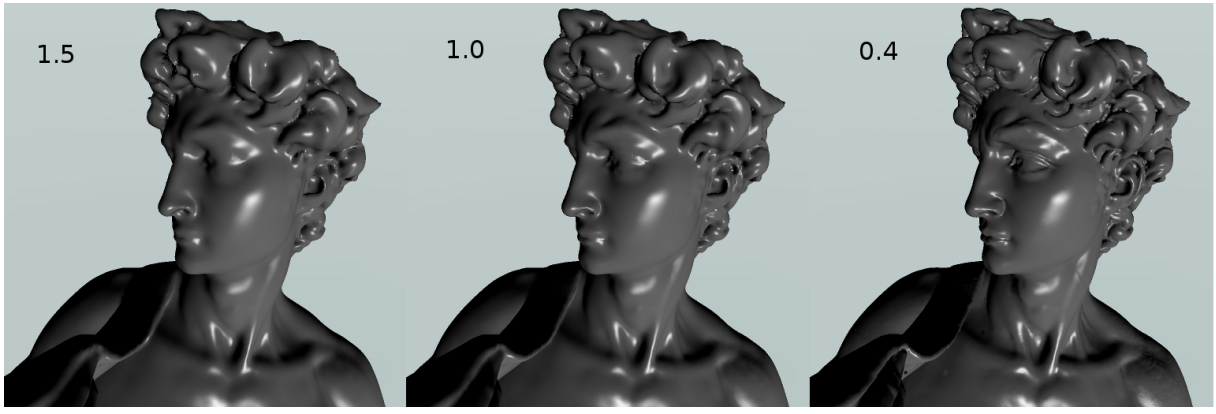

Figure 4.12: Varying degrees of smoothing applied to the David dataset.

Chapter 5

Lighting and Shading

Up until now, we have described a system that can traverse rays through a scene, find hit points and generate secondary rays. To generate the actual image, we need to compute the color value at each hitpoint. This process is called shading. Each ray produces a set of hit parameters, namely the hitpoint, the surface normal at the point and the surface material properties at that point. The shading kernel takes these values along with the light sources defined in the scene, and calculates the color at the point. This accounts for the local illumination at the point. Effects like diffuse inter-reflections and caustics, which are global illumination effects, are pre-computed offline. This global contribution is stored in an illumination map in the case of caustics, and baked into the color values of the octree leaf nodes in the case of diffuse inter-reflections (radiosity). The shading kernel simply queries these values at the given hitpoint and adds them to the local contribution, to produce a full global illumination solution for the scene.

5.1 Material Properties

The raytracer uses a simple material model that includes the following properties:

1. Ambient Color
2. Diffuse Color
3. Specular Color
4. Specular Exponent
5. Refractive Index
6. Reflectivity
7. Transmissivity
8. Diffuse reflectance

The specular color and exponent are used in Phong shading. Reflectivity, transmissivity and diffuse reflectance specify what fraction of light gets perfectly reflected, transmitted and contributes to local shading. They add up to unity unless the material absorbs light. These

properties are stored as an array in the constant buffer of the GPU. Each splat or leaf node (in the case of implicit surface octrees) is assigned a material-id. This material-id is used by the shading kernel to shade the hitpoint. The reflectivity, transmissivity and refractive index are used to generate secondary rays. The material properties can be changed at runtime, and the local illumination will remain correct, since there is no pre-computation that involves material properties. The global illumination components on the other hand, use the material properties in the precomputation step, and they will no longer be correct.

5.2 Lights

The raytracer supports simple point lights. Each light source has a position, color, and a flag to enable shadow casting. As with the materials, lights are stored as an array in the constant buffer. Lights are stored independent of the scene, ie: there is no spatial hierarchy defined on the light sources. The raytracer supports dynamic change of position, color and shadow-casting properties of any light source, as long as only a local illumination solution is desired. For global illumination with caustics and diffuse maps, a precomputation pass is required and moving the light sources will change the solution.

Before the shading kernel, a shadow kernel runs once on each light source. A shadow flag is set for each light source, for each hitpoint. The shading kernel iterates through each light source for which the shadow flag is off. It computes the color value due to this particular light source and adds it to the overall color of the fragment. The light source is used to compute local lambertian and phong shading components.

Note that since materials and lights are stored in the constant buffer, there is a limit to the maximum number of materials and lights. we typically set this limit to 16. Further, the shadow flag is stored in a 8 bit buffer for each hitpoint. Thus, there can only be eight shadow casting light sources in the scene, and these have to be the first eight lights defined for the scene. While this limit could easily be extended by using a larger shadow buffer, a large number of shadow casting light sources will drastically reduce the frame rate.

5.3 Light Maps

A light map is a collection of points, normals and color values at those points. Such an map can be the output of a precomputation phase where a path tracer or radiosity solver is used to compute the indirect illumination and caustics. The points in the light map are stored in an octree hierarchy to facilitate near neighbor queries. As a data structure, the light map answers a light gather query, ie: queries of the form "what is the illumination on a query point q, with normal vector n, and light-gather footprint g". Only those points in the light map, which are within the light-gather footprint, contribute towards the illumination at the query point.

If the light map is used to store a photon map, then the gather phase adds up the intensities of the samples within the gather footprint, since photon gather amounts to density estimation. Let the points in the illumination map be denoted by p_i, and their corresponding color values be c_i. Then, for a query point q, with gather radius g, the color value output will be:

$$y_i = (\frac{\|q - p_i\|}{g})^2 \tag{5.1}$$

$$color = \sum \frac{2}{\pi} \times (1 - y_i) \times c_i \tag{5.2}$$

In case the light map is used to store a diffuse map (output of radiosity step), then the output is a weighted average of the samples within the gather footprint.

$$y_i = \frac{1}{(1 - \frac{\|q - p_i\|}{g})^2} \tag{5.3}$$

$$color = \frac{\sum y_i \times c_i}{\sum y_i} \tag{5.4}$$

Note that the above summations are only performed on those sample points that are withing the gather radius. A further restriction can be to use only those points that have a normal vector similar to that of the query point, ie: the dot product of these normals is high. This can prevent spurious color bleeding when two surfaces are close to each other. For example, at wall corners, we would like to differentiate between samples meant for one wall and the other. Since the normal vectors of these samples have a low dot product, one wall will not pick up color samples from the adjacent wall.

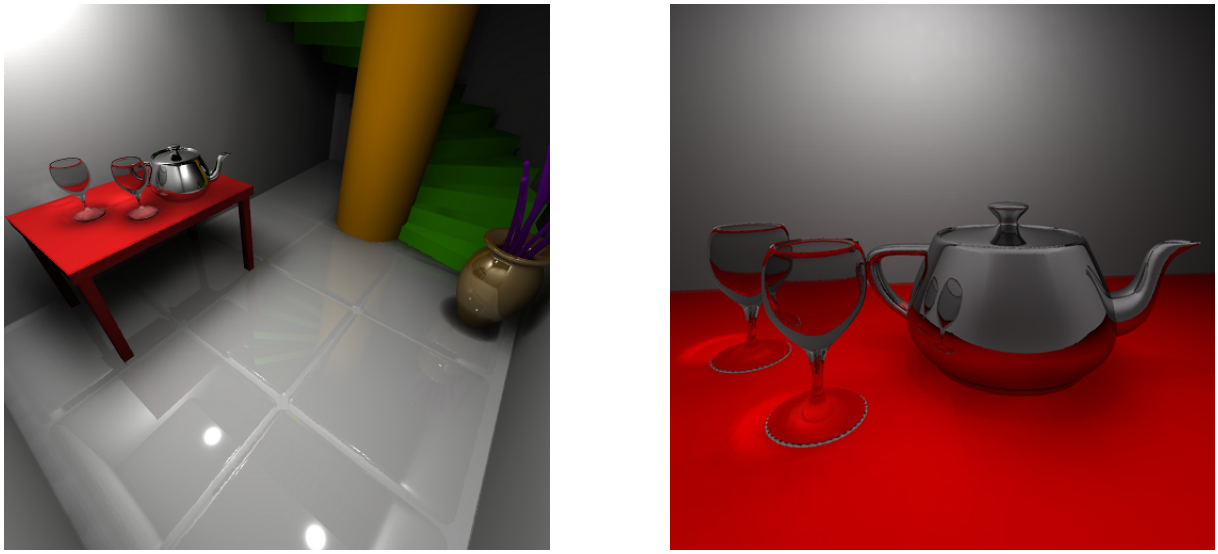

Figure 5.1: Scenes rendered with precomputed Light Maps.

5.4 Caustics

Caustics are bright patches of light produced when light is concentrated by reflection (like a parabolic reflector), or refraction (like a sphere or magnifying glass). In the regular expression syntax [Jen96], rays of type `LD?S*E` are handled in classic ray tracing, where `L` represents the light source, `S` and `D` represent specular surface and *D* diffuse surfaces, and `E` represents the eye or the camera. The general path for caustics is `LS+D`, i.e., caustic effects generated on a diffuse surface are caused after at least one specular hit from a ray originating from the light source. Path tracing is a popular way to compute caustic illumination maps. By placing the camera at

the light source position and using our existing raytracing framework, we simulate path tracing. Our algorithm consists of two steps.

5.4.1 Photon Shooting

In path tracing, photons, with some specific power, are emitted from the light source and traced through the scene. These photons, emitted from a light source, should have a distribution corresponding to the distribution of emissive power of the light source. If the power of the light is P_l and the number of emitted photons is n_e, we define the power of each emitted photon as $P_{photon} = P_l/n_e$. For the light source, we define the maximum number of photons (n_e) it can contribute.

Photon shooting is done in parallel, on the GPU (pre-computed before the ray tracing starts). A thread may conceptually be thought of as being attached to a photon being shot from the light source. We cover the light source with a cube and treat each of its faces as a shooting plane. We pixelize each of the cube's faces with some resolution, dependent on the required quality of caustic map (we use 512×512). We then shoot multiple rays per pixel with P_{photon} power per photon and their directions randomly jittered within the respective pixel. This process reuses the GPU raytracer we have developed. The only major difference is that instead of rendering an image, we output the photon intensity and hitpoint, whenever it hits a diffuse surface.

In our current implementation we allow only `LS+D` paths as we believe these to be significant contributors to caustics than the more general `LD+S+D` paths. Thus, most photons hitting the diffuse surface or exiting the scene (not intersecting the octree) are quickly culled. The ones that strike the specular surface are registered and traced further until they hit a diffuse surface or exceed some pre-defined bounce limit (we set it to 10). The power of every photon is reduced on every hit depending on the reflectivity/transmissivity of the specular surface. Once the photon hits a diffuse surface, we add its intensity and deposit-point co-ordinate into a caustic light map. We do this for all light sources in the scene. Note that this is a pre-computation phase and does not happen while ray tracing.

5.4.2 Photon Gathering

Typically, the photon gathering pass extracts information about incoming flux at any point in the scene. This is performed by gathering the caustic values from the caustic light map while ray tracing, as described in §5.3. However, the cost for performing near neighbors query for density estimation, is quite expensive when compared to the fast ray tracer we use for rendering. Further, the recursion used while performing the caustics gather operation is expensive to implement on the GPU.

The caustic illumination map is quite sparse and hence an octree based subdivision can store this data efficiently. Further, since the caustics are concentrated, it is preferable to have octrees for caustics with greater depths than for storing points or diffuse colors. Similar to ISO, we introduce a *caustic octree* which is sparse and has greater depths. A local density estimation for every corner of every leaf node in the caustic octree is performed as shown in Fig. 5.4.2. We then store this caustic octree as a texture on the GPU. This octree is similar to the one described in §2.2. Each leaf from caustic node pool has access to 8 intensity values, each corresponding to one corner of the node.

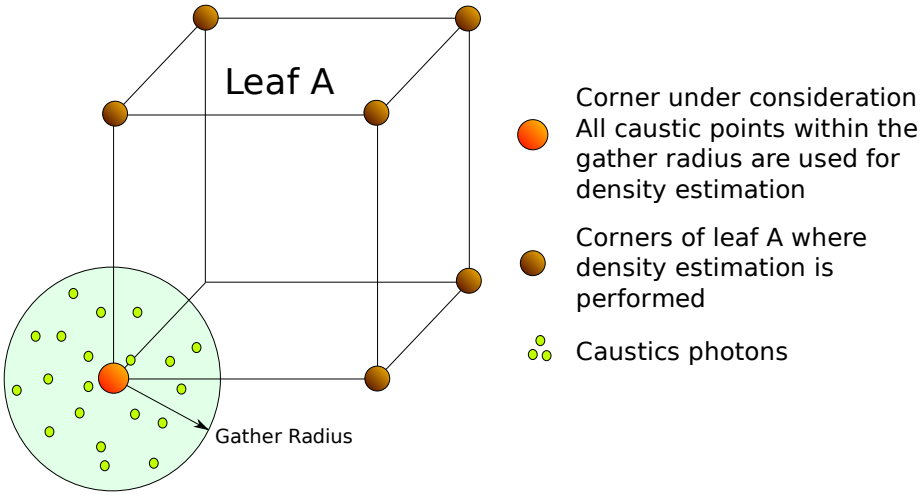

Figure 5.2: A classic density estimation technique is employed as a preprocessing step for leaf corners.

To render caustics, the shading kernel simply queries the caustic octree with the point to be shaded. This query uses the top-down traversal described in §2.2.2. On finding the leaf in the caustic octree, we apply tri-linear interpolation on the caustic brightness values stored at its corners to compute the caustic intensity at the query point. The application of tri-linear interpolation from corners of an octree node is not only much faster than the near neighbors search but also provides good quality results. Fig. 5.4.2 shows scenes rendered using the caustics photon map.

5.5 Textures

Since only a few materials can be defined for a given scene, the eight color values stored at each leaf corner are used to vary the colors on a node by node basis (in the splat based raytracer, each splat is associated with a single color value, which has a similar function). These color values can be used to represent indirect illumination as seen in §5.3, or they can be used as a texture map for the scene.

To generate such a texture map, we propose a simple technique. Consider a point sample p_i with normal n_i. We wish to calculate the texture color at this point and we want to do this without explicitly creating texture coordinates, or performing any sort of uv mapping. Let us say we have a single image available to us. We create three orthogonal, axis aligned planes, and tile this texture on each plane. Let the normals of these planes be n_x, n_y and n_z. Further, let the 2D projection of point p_i on each of these planes be p_x, p_y and p_z. We sample the texture using these three projected points, to obtain three color values c_x, c_y and c_z. The final color assigned to the point sample, is simply a weighted average of these three color values, where the weights are the dot products of each plane's normal vector with the point's normal vector.

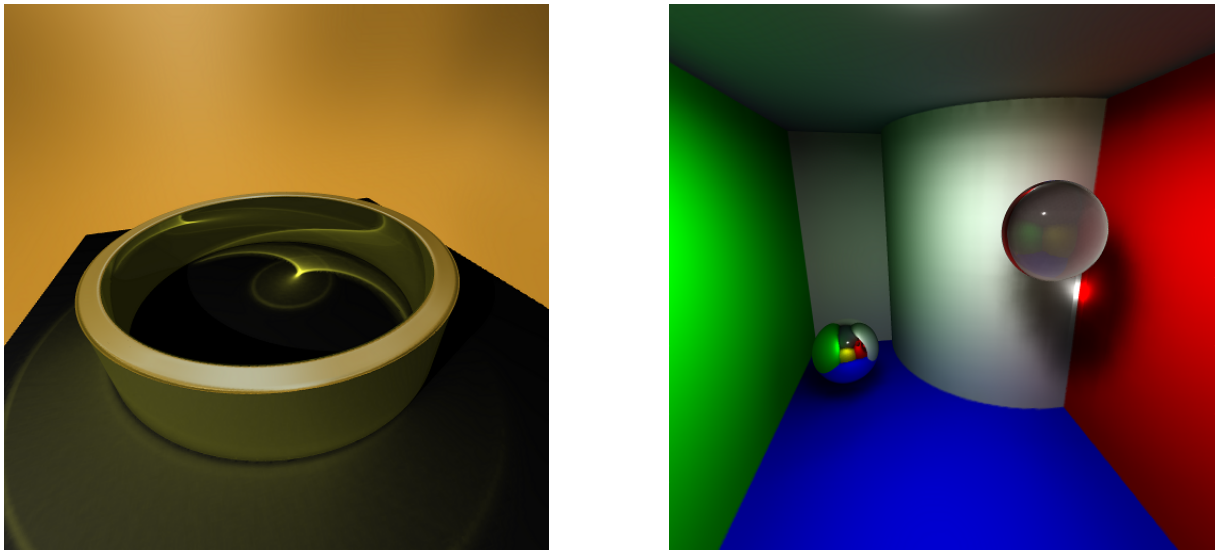

Figure 5.3: (a) Reflective ring, forming a characteristic cardioid caustic pattern (0.7 Million photons, 6 seconds to trace, 10 seconds to gather), (b) Refractive sphere focusing light on the wall, producing a round caustic pattern (1.25 Million photons, 13 seconds to trace, 15 seconds to gather). This scene is also rendered with a diffuse light map, demonstrating diffuse inter-reflections, color bleeding and soft shadows.

$$color = \frac{c_x \times (n_i.n_x) + c_y \times (n_i.n_y) + c_z \times (n_i.n_z)}{(n_i.n_x) + (n_i.n_y) + (n_i.n_z)} \tag{5.5}$$

This turns out to be a simple but effective way to map a texture onto any arbitrary surface. For added complexity, a different texture can be used for each texture plane. Further, more than 3 texture planes, or non-axis-aligned planes can also be used if required.

To actually map the texture onto the point model and render it, we treat these color values obtained from the texture generation phase, as a light map, and make a small change in the shading kernel, that allows the light source to modulate the color values obtained from the light map. Originally, the light map color was simply added to the fragment in the shading step. Now it gets multiplied by the dot product of the light vector and surface normal, and the light map emulates a traditional texture map (Fig. 5.5, Fig. 5.5).

Figure 5.4: Bunny and dragon models rendered with texturing

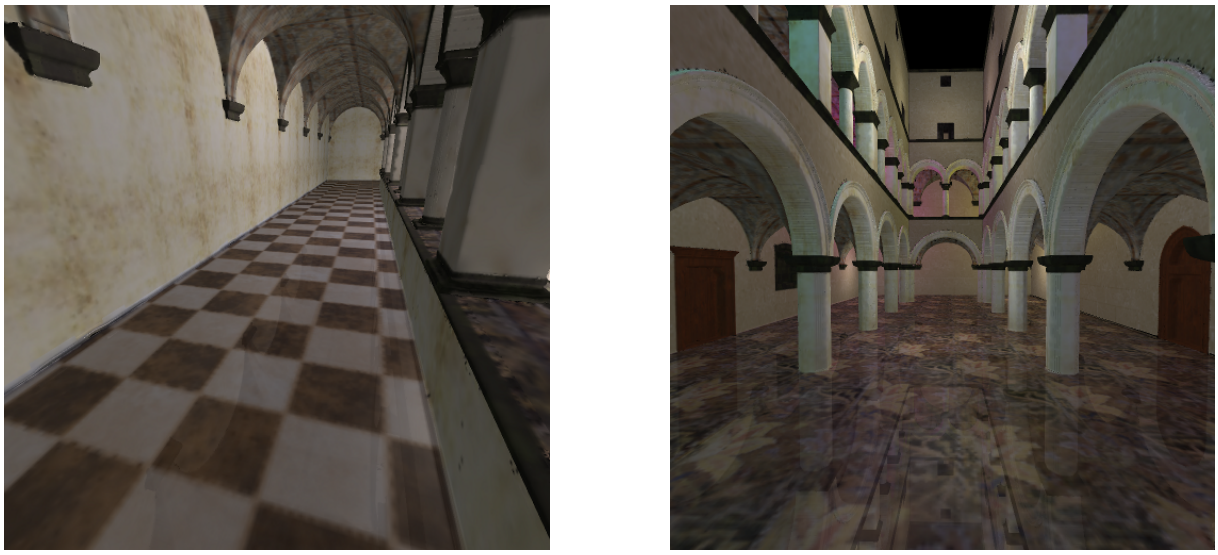

Figure 5.5: The Sponza Atrium, a 12 million point model, rendered with textures at 5 fps. The model is rendered using an ISO with 7.8 Million leaf nodes (276 seconds precomputation time). Texture mapping is performed using a diffuse gather that takes 156 seconds. The scene contains 16 non-shadow casting light sources and consumes 720 MB of GPU memory.

Chapter 6

Conclusion

We have evaluated two approaches for raytracing point models, Splat Based Raytracing, and Implicit Surface Octrees. We have shown that the ISO outperforms the splat based approach in terms of both speed and quality. We now have a framework for fast raytracing of point models on the GPU, which achieves framerates of 5 to 40 fps on various scenes. The system is modular and can be easily extended. It supports large point models, dynamic reflections, refractions and shadows, and precomputed light maps for indirect illumination and caustics. We also demonstrate how we can accelerate the computation of caustics using our raytracing framework. Finally, we develop a simple texturing algorithm for point models and integrate texture mapping into our framework.

It has been said that the killer application of general purpose GPU (GPGPU) programming, is graphics. This is turning out to be true, as more and more people use new GPU programming APIs such as CUDA, OpenCL and DirectCompute, to implement advanced graphics algorithms, that were previously restricted to slow CPU based renderers(REYES [ZHR+09]). With future improvements in graphics hardware, customized GPU renderers such as our raytracer will be increasingly used in production environments such as games and movies.

6.1 Future Work

This section details how our work on GPU raytracing can be extended further to achieve better speed and quality.

1. Primary ray casting optimizations: Primary ray casting can be accelerated by using the rasterization hardware ([CNLE09]) or using beam optimizations ([LK10]). This is similar to the packet traversal approach followed in CPU raytracers, and reuses the work done while traversing nearby rays.

2. Out-of-core rendering: The main issue with raytracing on any platform is that scenes that do not fit in memory are very hard to render. Each part of the scene can have complex interactions with any other part of the scene (due to the various paths that rays can take after reflection and refraction). This calls for a completely different approach to raytracing where parts of the scene that a ray wants to visit are not present in memory. The ray should then request for this scene, and go into a waiting phase. Performing this dynamic scene swapping is referred to as out-of-core rendering. On the GPU, this approach can be quite

challenging since memory swapping operations should be interleaved with computations to mitigate the performance hit ([CNLE09]).

3. Compression and dynamic decompression of scene data. Most production level graphics pipelines incorporate some sort of compression/decompression scheme to offset the memory load caused by today's massive datasets. Such a scheme will allow us to handle larger models and higher levels of detail than currently possible.

4. Full Path tracing on GPU: Currently, we have a system to generate caustics (`LS+D` paths) on the GPU. This system can be extended to support other possible light paths (`LD+S+D` paths). This will predominantly enable diffuse inter-reflections.

5. Better sampling in ISO: The implicit surface octree currently performs a sub-optimal sampling of the isovalues in space. The Body Centered Cubic (BCC) lattice has been shown to be a better sampling strategy [FEVM10]. This involves adding an additional sample to the center of each leaf node, and a more complex interpolation routine.

Chapter 7

Appendix

7.1 GPU Architecture and CUDA

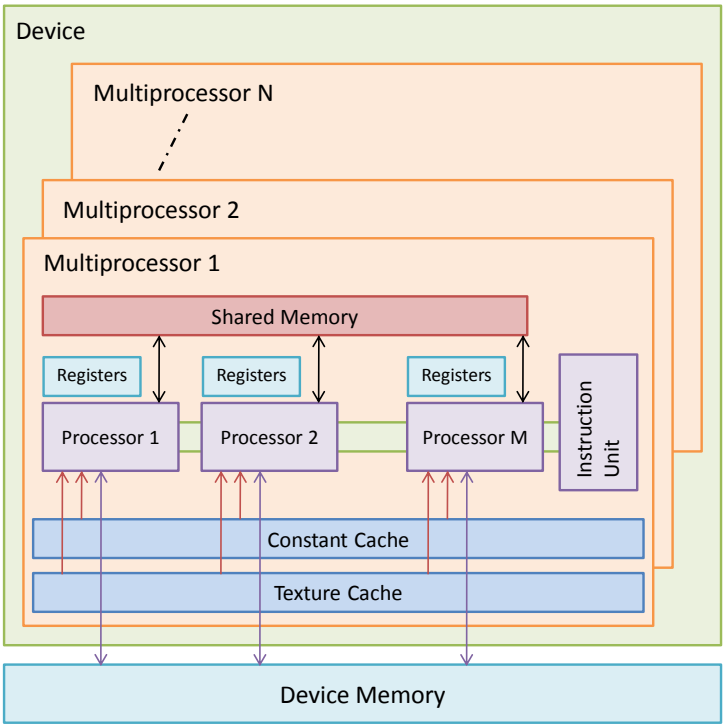

Figure 7.1: Hardware Model of GPU [Gor09]

We have implemented the GPU raytracer on NVIDIA's CUDA (Compute Unified Device Architecture) framework. This section provides a brief overview of the GPU architecture and the features and limitations of the CUDA programming model.

The GPU is a highly parallel device that executes thousands of threads at a time. Programs written for GPUs are called Kernels. Each thread executes the same kernel at a time. Although it seems like all threads are performing the same task, each thread has access to a thread id which we can use to assign different data for each thread to work on. Further, it is possible to perform data dependent branches in the kernel so that different threads can execute different pieces of code. Threads are organized into blocks and grids, for easier management.

The GPU consists of a large number of simple processors (ranging from 112 in the GeForce 9800GT to 240 in the GTX 275). These processors consist of floating point arithmetic units and special units to perform transcendental operations. Since it is extremely complex to manage instruction streams for such large numbers of processors, they are grouped into Multiprocessors, which contain 8 processors each. Each processor inside a multiprocessor shares the same instruction stream. nVidia GPUs execute 32 threads on each multiprocessor at a time, and this is called a warp. Each thread in a warp takes the exact same branches as every other thread. This is because they share the same instruction stream. If even one thread takes a branch, then the remaining 31 threads are also required to take that branch and execute all instructions present in it. To prevent wrong results, the outputs of operations are not written to registers in threads that execute these spurious branches. Therefore it is advisable for multiple adjacent threads to take the same branches. Otherwise a large performance hit will be incurred.

GPU memory is organized into registers (8192 registers per multiprocessor on a 9800GT, 16384 on GTX 275), shared memory (16 KB per multiprocessor), and global memory (the full size of the RAM on GPU, in several hundred MegaBytes). Registers are used to execute code and cannot be directly accessed by the programmer. The existing registers are shared between the executing threads, and are also used to store the context of waiting threads. Therefore, to ensure high occupancy(large number of threads), we need to ensure that each thread requires as few registers as possible. At around 10 registers per thread, the code runs at near optimal efficiency. If a thread uses around 32 registers, the theoretical throughput is halved. At close to 50 registers, the occupancy falls to around 25 percent.

Shared memory is fast, low latency memory local to the multiprocessor. It can be thought of as a user managed cache. Data stored in shared memory cannot be accessed by other multiprocessors. Shared memory is optimized for multiple accesses at the same time. Global memory is high bandwidth (60-170 GBps), high latency (around 400 clock cycles) memory. Coalesced accesses to global memory are quite fast, while random accesses are slow. To speed up random global memory access, a texture cache is provided. This is an automatically managed cache which is optimized for 2D spatial locality in accesses.

Programming in the CUDA framework is similar to writing C code. The major differences between CUDA and C++ code are:

1. No support for recursion

2. No dynamic memory allocation

3. Limited support for classes

Further, a CUDA kernel(device code) cannot launch another CUDA kernel. All kernels must be launched from the CPU(host code).

7.2 Raytracing equations

1. Reflection

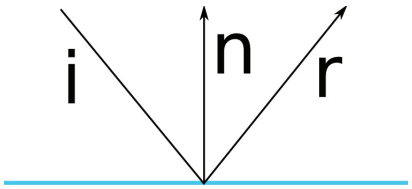

Figure 7.2: Reflection

$$r = i - 2 \times n \times n.r \tag{7.1}$$

2. Refraction

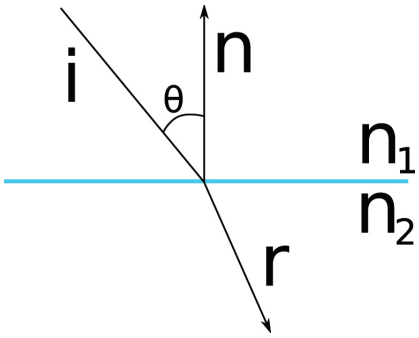

Figure 7.3: Refraction

$$r = (-\frac{n_2}{n_1} \times (i.n) - \sqrt{1 - \frac{n_2}{n_1} \times \sqrt{1 - (i.n)^2}} * n + \frac{n_2}{n_1} \times i) \tag{7.2}$$

where n_1 and n_2 are the refractive indices of the source and destination media respectively.

Bibliography

[AA03] A. Adamson and M. Alexa. Ray tracing point set surfaces. In *SMI '03: Proceedings of the Shape Modeling International 2003*, 2003.

[Bal03] Kavita Bala. Combining edges and points for interactive high-quality rendering. *ACM Trans. Graph*, 22:631–640, 2003.

[CHCH06] Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast gpu ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface 2006*, pages 203–209, 2006.

[CHH02] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 37–46, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[CNLE09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, Boston, MA, Etats-Unis, feb 2009. ACM, ACM Press.

[FEVM10] Bernhard Finkbeiner, Alireza Entezari, Dimitri Van De Ville, and Torsten Mller. Efficient volume rendering on the body centered cubic lattice using box splines. *Computers and Graphics*, In Press, Corrected Proof, 2010.

[GL10] Kirill Garanzha and Charles Loop. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *Eurographics*, 29(2), May 2010.

[Gor09] Rhushabh Goradia. Global illumination of point models, 2009.

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 167–174, New York, NY, USA, 2007. ACM.

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK, 1996. Springer-Verlag.

[KGCC10] Sriram Kashyap, Rhushabh Goradia, Parag Chaudhuri, and Sharat Chandran. Real time ray tracing of point-based models. In *I3D '10: Proceedings of the 2010 ACM*

SIGGRAPH symposium on Interactive 3D Graphics and Games, pages 1–1, New York, NY, USA, 2010. ACM.

[KL04] Botsch M. Kobbelt L. A survey of pointbased techniques in computer graphics. In *Computers & Graphics 28*, volume 6, pages 801–814, 2004.

[KWPH06] Aaron Knoll, Ingo Wald, Steven G Parker, and Charles D Hansen. Interactive Isosurface Ray Tracing of Large Octree Volumes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 115–124, 2006.

[LK10] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *Proceedings of ACM SIGGRAPH 2010 Symposium on Interactive 3D Graphics and Games*, pages 55–63. ACM Press, 2010.

[LMR07] L. Linsen, K. Mller, and P. Rosenthal. Splat-based ray tracing of point clouds. In *Journal of WSCG*, pages 51–58, 2007.

[LPC+00] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3D scanning of large statues. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 131–144. ACM Press, 2000.

[LW85] Marc Levoy and Turner Whitted. The use of points as a display primitive. Technical report, University of North Carolina at Chapel Hill, 1985.

[PZvBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 335–342. ACM Press, 2000.

[RL00] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In Kurt Akeley, editor, *Computer Graphics Proceedings*, pages 343–352, 2000.

[SJ00] G. Schaufler and H. Jensen. Ray tracing point sampled geometry. In *Eurographics Rendering Workshop Proceedings*, pages 319–328, 2000.

[WK04] Jianhua Wu and Leif Kobbelt. Optimized sub-sampling of point sets for surface splatting. *Computer Graphics Forum*, 23:643–652, 2004.

[WS03] Michael Wand and Wolfgang Straer. Multi-resolution point-sample raytracing, 2003.

[WS05] Ingo Wald and Hans-Peter Seidel. Interactive Ray Tracing of Point Based Models. In *Proceedings of 2005 Symposium on Point Based Graphics*, 2005.

[ZHR+09] Kun Zhou, Qiming Hou, Zhong Ren, Minmin Gong, Xin Sun, and Baining Guo. Renderants: interactive reyes rendering on gpus. In *SIGGRAPH Asia '09: ACM SIGGRAPH Asia 2009 papers*, pages 1–11, New York, NY, USA, 2009. ACM.

[ZK07] Haitao Zhang and Arie E. Kaufman. Point-and-edge model for edge-preserving splatting. *Vis. Comput.*, 23(6):397–408, 2007.

[ZPK+02] Matthias Zwicker, Mark Pauly, Oliver Knoll, Markus Gross, and Eth Zrich. Pointshop 3d: An interactive system for point-based surface editing. pages 322–329, 2002.

[ZPvBG01] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 371–378, New York, NY, USA, 2001. ACM Press.

[ZRB+04] Matthias Zwicker, Jussi Räsänen, Mario Botsch, Carsten Dachsbacher, and Mark Pauly. Perspective accurate splatting. In *GI '04: Proceedings of Graphics Interface 2004*, pages 247–254, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society.