Two GPU Algorithms for Raytracing

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

Master of Science (by Research)
in
Computer Science and Engineering

by

Srinath.R
201107625
srinath.ravichandran@research.iiit.ac.in

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

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled ‘Two GPU Methods for Raytracing‘ by Srinath.R (201107625), has been carried out under my supervision and is not submitted elsewhere for a degree.

Date	Adviser: Prof. P.J.Narayanan

To Amma and Appa

Acknowledgments

This work would not have been possible without the guidance of my advisor Prof.P.J.Narayanan. He was instrumental in helping me go the extra mile even when things looked bleak. He guided me when I failed and helped me develop a healthy attitude towards research. I hope to put the things that he has taught me to good use in the future.

I would like to thank my Amma and Appa for giving me the wings to fly high in search of what I truly desire. I dedicate this thesis to them. I would like my Akka for being the best buddy out there. I would like to thank my fellow lab mates Rajvi, Parikshit, Revanth, Saurabh, Mihir, Aditya who were the main reason for keeping the lab a fun place for me. The Risk, DOTA and other banter sessions are something that I will cherish for life. I've made some amazing friends here at IIIT who were there to hold me up when I was low and made me not miss home. Harsha, Divyesh, Manoj, Ziaul, Ajay, Bharath, Suryansh, Karthik and Siva - I thank you guys for all that you've done. Thank you Navya for being there when I needed you the most. And finally my gratitude to the Almighty who was always there as a gentle guiding force providing me with confidence and strength when I fumbled many times.

Abstract

Raytracing has been primarily used for generating photo-realistic imagery. Once considered as an offline algorithm, raytracing has primarily become realtime or near realtime with the advancement in computer hardware power. GPU computing has enabled the general purpose usage of graphics cards which were primarily designed for for graphics rendering purposes. Raytracing could be considered as a very embarrassingly parallel process in which each pixel's contribution to the final image is computed independent of one another. The tremendous power of GPUs has been been utilized for vastly improving the performance of almost all the functions in a raytracing pipeline.

The raytracing pipeline for rendering images fundamentally consists of three phases. The first is an acceleration structure construction phase which involves creating a data structure. It provides very fast results for intersection queries for the rays traced within the scene and all the objects contained within the scene. The second is the traversal phase in which each ray is traced within the scene employing the aforementioned data structure. The final optional phase is shading in which each hit point is shaded based on a shading algorithm. The shading algorithm determines the color of the pixel of the image for which a particular ray was traversed and the shading performed for all the rays for all the pixels computes the final rendered image. In this thesis we examine two problems associated with the GPU raytracing pipeline and provide two new approaches for them.

In the first work, we develop a novel parallel algorithm for performing raytracing without constructing any explicit acceleration structure on the GPU. This decouples the long standing notion of creating and storing a separate acceleration structure followed by tracing rays through the structure. Our algorithm creates an implicit hierarchical acceleration structure and traverses the implicit structure at each phase of the algorithm in tandem. Parallel construction algorithms for acceleration structures are difficult to implement on the GPU and also have a large memory footprint to store the resulting structure. Compared to CPUs the amount of memory available to GPUs is limited and hence methods that work on very small memory footprints are of utmost importance. Our algorithm is conceptually very simple, utilizing efficient parallel GPU primitives such as sort and reduce. Further our algorithm has small memory requirements compared to methods that construct acceleration structures thereby making it a suitable candidate for the GPU. Since our method employs a *traverse-while-construct* method, it is particularly very useful for animated scenes in which the acceleration structure has to be created for each frame in a traditional raytracing pipeline. We implement our algorithm on a CUDA enabled machine and show that our algorithm can perform much better than the serial CPU version of the algorithm.

Our second work is targeted towards the problem in the shading phase of the raytracing pipeline. Traditional renderers employed in the production rendering are primary unidirectional path tracers which traces rays from the camera into the scene. However the path tracing algorithm has difficulty in rendering scenes with complicated lighting scenarios which provide very good aesthetic value to certain kinds of scenes. Bidirectional path tracing on the other hand traces rays from both the camera and the lights within the scene and hence able to render scenes with complicated lighting scenarios much more effectively than path tracing. Current production renderers are primarily CPU based and only a few have started to employ GPUs for the entire pipeline. Limited memory compared to CPUs was the original factor for not employing GPUs. However with current generation GPUs having much more memory than earlier generations, GPUs have been slowly adopted for path tracing based pipelines. However bidirectional path tracers that are GPU based are yet to be fully employed in a full production rendering scenario. Our work is aimed providing a solution that tries to bridge the gap that enables bidirectional path tracing to be used in the production rendering scenario which involves complex materials. We specifically provide a new connection mechanism employed in the light vertex cache - bidirectional path tracing algorithm (LVC-BDPT) for improving shading efficiency as well as a sort based pipeline for improving the runtime performance of the algorithm on the GPU. We show that we perform much better than the unoptimized version of the algorithm as well as providing much better quality for the same amount computations performed.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer graphics has revolutionized the entertainment industry. Today's games and movies are graphically very advanced compared to earlier generations. Viewer immersion is the single most important driving factor in the entertainment industry and computer graphics has become its main enabler. The entertainment industry is also a very important commercial element with movies and games having budgets in range of millions of dollars and revenues in the range of hundreds of millions of dollars. Computer graphics is also a very important component of the visualization, architectural and automotive industry. The automotive industry is dependent on photorealistic rendering of its products in the design stage before moving on to the actual production state. Further predictive rendering algorithms have enabled architects and engineers to build better buildings by helping them visualize how the building would look like in the design stage. Computer graphics is also used in the medical field for applications such as 3d MRIs and CT scans. Further, fields such as particle and nuclear physics make use of computer graphics algorithms for visualizing many phenomena. Rapid advancement in computer graphics technology coupled with an equally tremendous growth in computer hardware power has been the major reasons for the growth in the above fields. One of the most important and related developments in the computer industry has been the advent of GPU computing.

1.1 GPU Computing

The Graphics Processing Unit (GPU) was first designed for rendering with the goal of accelerating the rasterization graphics pipeline. Rasterization is the process of converting a three dimensional representation of objects into a two dimensional image consisting of pixels. The rasterization pipeline is well defined and contains many functional stages to convert the input three dimensional object representation into the final image. Early generation GPUs had fixed a function pipeline in hardware each designed for a particular stage in the rasterization pipeline. However with the advent of the programmable functional units in the GPUs, programmers were able to execute small programs called 'shaders' in certain stages of the pipeline. Programmers utilized these programmable shaders to do general purpose matrix and vector computations much more faster than on the CPU. This acceleration was made possible due to the

fact that matrix and vector operations were very fundamental operations in the rasterization pipeline and the GPUs were designed specifically for accelerating such work. With the advent of the unified compute architecture in GPUs and programming APIs like CUDA (Compute Uniform Device Architecture) [32], DirectCompute and OpenCL (Open Computing Language) [42], programmers were able to effectively utilize the GPU for writing parallel general purpose code.

GPU computing has grown by leaps and bounds since then with each generation of GPUs becoming more powerful than the earlier generations packing in more cores as well as becoming more power efficient. GPU computing has resulted in extremely fast parallel primitive like sort [39], reduce, scan [40], etc. via dedicated libraries like Thrust [17], CUDPP, etc. GPU computing has been embraced by many fields like Computational Fluid Dynamics (CFD), Molecular Biology, Astrophysics, etc., which have utilized the immense parallelism available in the GPUs to accelerate the algorithms by many fold. Several top high performing super computers in the world today employ GPUs to provided added program acceleration capabilities. GPU computing has also become extremely fundamental for the recent advances in deep learning for training very large and deep neural networks.

1.2 Rendering

Rendering is the process of converting a three dimensional virtual representation of a scene into a two dimensional image as viewed by a virtual camera present within the scene. Traditionally rendering is viewed as either being realtime or offline. Realtime rendering is targeted with the goal of producing the final image with a target frame rate which is normally 30 or 60 frames per second (fps). Speed is of the highest priority in realtime rendering and hence the algorithms employed sacrifice some visual fidelity for performance. Video games, virtual reality (VR) rendering are example scenarios where realtime rendering is employed. Offline rendering time for a single frame can extend from a few minutes to a couple of hours. Quality of the rendered image is the single most dominant requirement in offline rendering. Production rendering employed in movies is an example of offline rendering. Rasterization and raytracing [46] are the two main fundamental rendering algorithms. Rasterization is used typically in games for realtime rendering. Raytracing was originally an offline rendering algorithm used to generate very realistic imagery. Raytracing is fundamental to production rendering because of the sheer variety of effects that it is capable of producing. However with recent developments in GPU computing and computer hardware, raytracing has become realtime for simple to moderately complex scenes. However raytracing employed to render complex scenes used in production rendering still remains offline.

1.3 Raytracing

Raytracing is generally aimed at generating photo-realistic images of 3-dimensional scenes. Raytracing proceeds by first taking as its input a description of the virtual scene to be rendered containing all the geometry and material information. The description also contains all the light sources present in

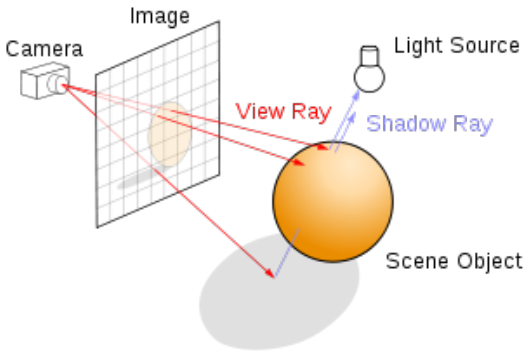


Figure 1.1: Illustration of raytracing (courtesy Wikipedia)

the scene and the virtual viewpoint within the scene from which the image to be rendered is computed. Raytracing proceeds by generating rays for each pixel in the image (Figure 1.1) and traces the ray into the scene. The algorithm then computes the nearest intersection surface point among all the scene objects as described in Line 2 in Algorithm 1. It then evaluates a shading algorithm at the intersected point to compute the final color of the pixel. The algorithm proceeds ahead to compute the color of each pixel and computes the final composite image.

Algorithm 1 Basic baytracing algorithm

```
1: procedure BASIC RAYTRACING ALGORITHM
2:     for each pixel in image do
3:         Compute view ray from camera center through pixel.
4:         Compute nearest surface intersection point.
5:         Shade intersection point according to shading parameters.
6:     end for
7: end procedure
```

The raytracing pipeline in general can be described using Figure 1.2. The algorithm begins with the input scene description containing all the objects. Direct brute force computation of intersections between all the rays and the scene objects is computationally inefficient and hence acceleration data structures are created to compute the ray-object intersections faster. Ray traversal using the acceleration structure follows the construction phase followed by shading of the intersected point. Depending upon the acceleration structure constructed, the ray traversal performances differ. We will discuss more about acceleration structures in Section 1.4. Depending upon the type of algorithm employed to compute shading, the type of effects that can be simulated and the quality of the image also vary. Light transport algorithms model the flow of light within a scene and are generally used to shade the intersected point. We will discuss about light transport algorithms in Section 1.6.

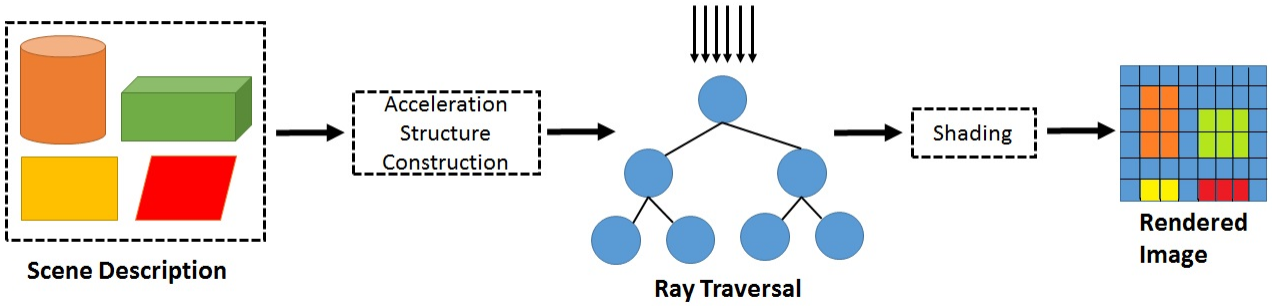


Figure 1.2: Raytracing pipeline illustrated.

1.4 Acceleration Structures

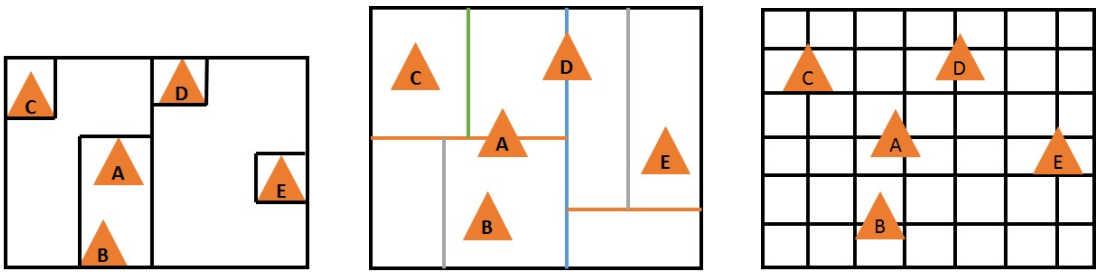
The raytracing algorithm is a very compute intensive process. Ray-object intersection routines are the heart of any raytracing application and the majority of rendering time is normally spent in computing intersections. Traditionally the scenes are modelled using polygons particularly using triangles because triangles have comparatively minimal storage requirements and ray-triangle intersections can be computed very quickly [29]. Current day raytracing pipelines employ images that have very large pixel resolutions like 1K, 2K, or 4K resolutions. Further inorder to render images with practically no noise, a large number of rays are traversed through each pixel and the results are filtered to reduce aliasing in the image. Hence in modern renderers the number of rays in flight can be in the range of a few millions to even billions. Further the scenes rendered are getting more complex with scenes having triangle count ranging from a few hundred thousands to a couple of million triangles to present day movies employing scenes that have several billion triangles. This is due to the fact that as hardware power increases artists are able to create more and more complex scenes with increased polygon count to increase the visual fidelity.

A naive ray-objection intersection routine used for computing intersections between M rays and N objects has a complexity of $\mathcal{O}(M \times N)$. This can become computationally very expensive and even impractical even with the most advanced hardware available today. Hence in order to reduce the computational complexity a variety of acceleration data structures have been developed. These acceleration structures organize the scene primitives by virtue of their positions such that a ray executes intersection queries only with those objects in its path of traversal. Hence a current day raytracing pipeline will have an acceleration structure construction step as a fundamental step as shown in Figure 1.2. Generally the acceleration structure is constructed only once for a scene and can be used to render multiple times from any viewpoint in case of static scenes. However in case of animated scenes where scene geometry changes from frame to frame frame, the acceleration structure has to be computed every time thereby making the costs incurred on a per frame basis rather than a per render basis.

The raytracing algorithm is an inherently parallel process in which each pixel's color can be computed independently of one another. With the advent of programmable GPUs, the raytracing algorithm

has been modified to run in parallel using the graphics pipeline [37]. With the advances in GPU hardware power and dedicated GPU programming technologies, raytracing which was once an offline process has become realtime [3]. Earlier parallel raytracing methods required the acceleration structure to be pre-built in an offline process with only the tracing step in realtime. However a variety of very fast parallel algorithms have been developed since then that construct the acceleration structures very quickly in CPUs and GPUs.

1.5 Different Acceleration Structures



(a) BVH (b) Kdtree (c) Grids

Figure 1.3: Various kinds of acceleration structures used for raytracing.

An acceleration structure is a data-structure that is constructed over scene geometry with the primary goal of reducing the time taken to compute the intersection queries for any ray within a scene. An acceleration structure can also be used to compute nearest neighbour queries, occlusion queries and many other similar queries encountered. Acceleration structures can be broadly categorized as *object partitioning* and *space partitioning*. Object partitioning acceleration structures divide the the scene elements object wise traditionally with a parent-child hierarchical relationship. For example, a 3d model involving a room will have the model as the parent and the individual parts of the room such as walls, floor, roof, tables, etc as the children. Any ray that is tested against this model for intersection proceeds down the object hierarchy. Spatial partitioning structures divide scene objects by splitting the enclosing space of the objects.

1.5.1 Bounding Volume Hierarchies

Bounding Volume Hierarchies [38] form a disjoint hierarchical division of scene objects for ray object intersection queries. BVH structures are attractive due to the fact that all scene elements appear only once in the entire hierarchy resulting in compact structures. Further BVHs are easier to construct compared to other hierarchical acceleration structures such as kd trees. The enclosing volume used in the hierarchy is traditionally an axis aligned bounding box (AABB). However spheres are also used as the bounding volumes due to having minimal storage requirements than AABBs. Tracing of rays through the hierarchy proceeds in a top down fashion starting from the root node. Only if the intersection test

between the incoming ray and the parent AABB succeeds, the child AABBs are tested with ray. Else the ray is discarded since the parent AABB always encloses the AABBs of the its children. The decision to determine which of the two children to be traversed first is normally decided by performing the traversal with the node whose AABB has the minimal intersection distance with the ray. In this way, the ray traverses through the hierarchy with increasing orders of depth from its starting point. All the nodes that were not taken for traversal are generally pushed down a stack and are then popped up once the ray comes out of the intersection routine. Stackless implementations also exist that are better suited for parallel traversal [2, 16].

BVH construction algorithms on the GPU are extremely fast due to the fact that the entire construction process has been reformulated as a sorting step using 3d morton codes as illustrated in Figure 1.4. Each scene primitive is assigned a 3d morton code based on its position within the scene. The morton code or the z-order curve code is composed by interleaving the bits of the x, y, z coordinates of each point. Sorting all these morton codes arranges all the scene objects in an implicit hierarchy. An explicit acceleration structure is then constructed over these codes [28]. With the advent of very fast GPU sorting algorithms, the construction process of BVH on the GPUs has tremendously decreased. Further work by [35, 13, 24, 25, 41] have increased the quality of the BVH produced as well reduced the construction times even further.

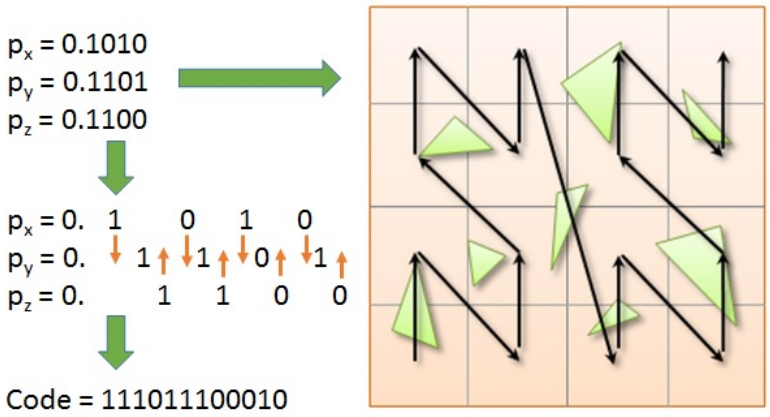


Figure 1.4: BVH construction using morton codes. Image adapted from the 'PARALLEL FORALL' website.

1.5.2 Kd Trees

Kd trees [4] employ an axis aligned spatial division scheme to partition scene primitives. The construction algorithms partitions scene elements by splitting the space in one of the three coordinate axes as indicated in Fig 1.3. Compared to BVH, kd trees are generally very efficient in partitioning scene objects since they cull empty space very effectively. Kd trees also have better intersection query performance due to various optimizations such as early ray termination added to the ray traversal routine.

Early ray termination is available only in spatial acceleration structures due to the fact that space is split and not objects and hence an object which is intersected first will always be the first object that was encountered. This property does not hold true in BVHs. However kd tree construction algorithms are more complicated than BVH construction algorithms. Hence they incur a longer runtime cost. Further kd trees require more storage space than that required by BVH. This is due to the fact that a scene primitive that straddles a split plane will have its reference in both the nodes created by the split plane and hence kd trees can have substantial duplication of scene primitive references.

The kd tree construction algorithms expose a lot of data parallelism that has been used to develop fast parallel construction algorithms on the CPU and the GPU. Choi et al. [6] developed a parallel kd tree construction algorithm on the CPU where as Zhou et al. [48] developed the first parallel kd tree construction algorithm on the GPU. Wu et al. [47] developed an improved parallel kd tree construction algorithm on the GPU that was SAH optimized for the best performance.

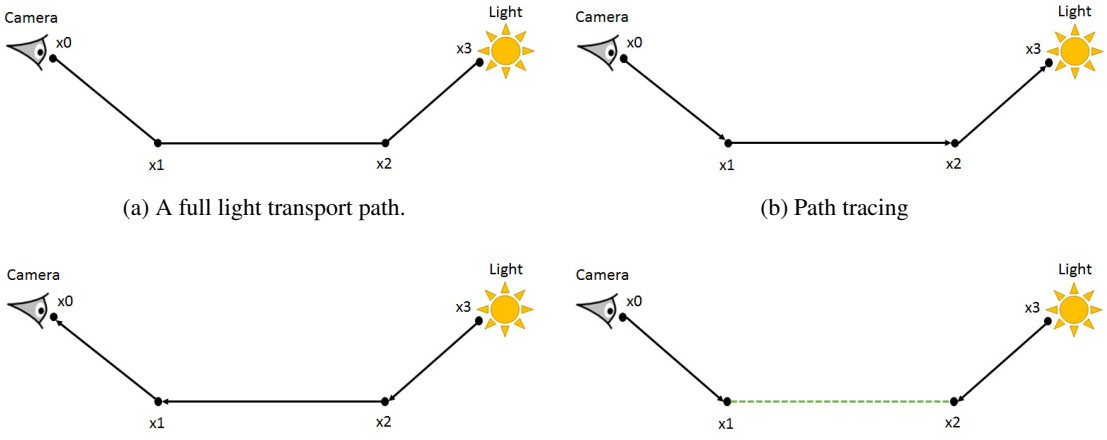
1.5.3 Grid Based Structures

Grid based structures [12] traditionally divide an axis aligned region of space equal regions of space called voxels (Figure 1.3). Each voxel stores the scene objects that overlap it. A ray object intersection query basically traces the ray through the grid voxel by voxel checking for intersections with all the objects within each voxel. Ray traversal routines using grids are also extremely fast compared to other acceleration structures due to the simple nature of grids.

Grid acceleration structures can be computed extremely quickly on GPU hardware using the very fast parallel primitives such as scan, sort and reduce [40, 39, 14]. An extremely fast grid computation algorithm using the GPU was presented by Kalojanov et al. [22]. However grid acceleration structures can result in very poor performance in certain scenarios when the scene objects are not distributed evenly throughout the enclosing space but rather concentrated in only certain parts of the scene. This is called 'teapot in a stadium' scenario and can cause performance loss because most of the grid cells will have no objects to perform intersection queries with whereas only certain cells will a lot of scene objects to be considered for intersections. This scenario generally appears due to not having enough fine subdivision for the entire region. Having large voxels will result in scene objects being confined to only a few cells where as a very fine subdivisions would result in lots of empty space. However this problem has been tackled with employing adaptive grids that does not split the entire region uniformly but rather splits regions based on the geometric complexity of the objects within that space [21]. With grids being able to be computed extremely fast, GPUs have been utilized for real time raytracing of dynamic scenes [15].

1.6 Shading Algorithms

In the raytracing pipeline, shading algorithms executed at the intersection points of rays determine the color of the point which is then accumulated in the final image. Different shading algorithms capture



(a) A full light transport path.

(b) Path tracing

(c) Light tracing

(d) Bidirectional path tracing

Figure 1.5: Types of light transport algorithms. The full light transport path consists of one point in the camera, one point on the light and two points on scene surfaces. The direction of the ray indicates the direction in which the path was constructed to model the flow of light. The dashed line indicates a visibility segment that is valid only if the two points are mutually visible to one another.

different phenomena effectively. Traditionally a physical model of light is employed by the shading algorithms to determine the color of the object being visualized. Light transport algorithms assume a particulate nature of light which assumes light to move in straight lines and hence raytracing is the de-facto tool for modelling the transport. These light transport algorithms model the flow of light within a scene and are able to visualize the scene as viewed through a virtual camera placed within the scene. A variety of light transport algorithms have been formulated that are effective for rendering different kinds of scenes effectively. Light transport algorithms are broadly classified as unidirectional and bidirectional depending upon the flow of light is modelled.

Unidirectional light transport algorithms model the flow of light from a single source which can either be the camera or the light with the destination being either the light or camera respectively. Path tracing (Figure 1.5b) [8] proceeds from the camera and tries to construct light carrying paths within the scene towards the light source. Light tracing (Figure 1.5c) [11] constructs paths from the light source towards the camera. Depending upon the properties of the camera, light source and objects within the scene, one algorithm can render scenes effectively than the other.

Bidirectional light transport algorithms start from both the camera and the light. The algorithm then connects both the subpaths to form visible and valid light paths. Since paths are created from both the light and camera at the same time, bidirectional algorithms are traditionally very efficient in capturing difficult to render lighting conditions by unidirectional methods. However bidirectional methods are generally more complex and require more memory than unidirectional methods. Bidirectional Path Tracing (Figure 1.5d) [44, 26], Photon Mapping [19] are examples of bidirectional methods.

The rendering equation [20] models the flow of light as an integral equation. Generally an analytical solution for the rendering equation is computable only under very simple assumptions about the scene model. When the scene conditions don't favor computing an analytical solution, Monte Carlo methods have been employed to solve the rendering equation. Monte Carlo integration [23] methods employ random numbers to compute the solutions and hence very simple. However Monte Carlo methods are computationally very expensive and require lots of samples to arrive at a reasonably good estimate of the solution. Monte Carlo light transport algorithms provide ample scope for parallelization and with the advent of GPU programming, the graphics community has embraced the notion of employing GPUs for accurate light simulation. We will discuss more about light transport algorithms and the Monte Carlo integration method in Chapter 3.

1.7 Summary of Contributions

Our contributions in this thesis are two fold addressing two different problems in the raytracing pipeline on the GPU. In our first work we develop a new parallel raytracing method that performs tracing without constructing any explicit acceleration structure on the GPU. The method employs a divide and conquer approach akin to quicksort to construct an implicit hierarchy while traversing the structure at the same time. The method was developed with the goals of being efficient, simple and needing very less memory. Acceleration structures traditionally have large memory requirements and memory capacity available on GPUs is far less than CPUs. Our method is particularly attractive for GPUs since the memory required is very less compared to methods that construct an explicit acceleration structure on the GPU. Further our algorithm is also simple compared to other parallel acceleration structure construction algorithms on the GPU thereby making it attractive for quick adoption into existing programs. We also show that our algorithm performs better than the serial CPU algorithm for a variety of standard test scenes.

In our second work, we develop a new method called Coherent and Importance Sampled LVC-BDPT (CIS-LBDPT) which addresses the issue of improving the connection phase of bidirectional path tracing (BDPT) in the Light Vertex Cache (LVC) framework as well as improving the algorithm's performance on the GPU in the presence of complex layered materials. We work on the idea that the connection phase in BDPT is a significant source of variance in the rendered image and hence we develop a new method to reduce variance by choosing intelligent connections. Further the presence of complex layered materials decreases the code execution coherency on GPUs thereby resulting in poor performance of the algorithm without any optimizations. We present a sort based approach that improves code execution coherency as well as coalesced data access and hence improves the runtime performance of the algorithm on the GPU. We show that our algorithm is faster and produces qualitatively better images than the original GPU LVC-BDPT algorithm.

1.8 Thesis Organization

The thesis is organized as follows. In Chapter 2 we present our new parallel divide and conquer raytracing method. In Chapter 3 we provide an overview of light transport algorithms with a particular focus on a bidirectional path tracing. In Chapter 4 we describe our CIS-LBDPT algorithm. We conclude the thesis by summarizing the contributions in Chapter 5.

Chapter 2

Parallel Divide and Conquer Ray Tracing

In this chapter, we present our work on devising a fundamentally new way of performing raytracing on the GPU without constructing any explict acceleration structure. We begin the chapter by providing some information that sheds light on acceleration structure usage in the raytracing pipeline followed by explaining the recent method of divide and conquer raytracing (DACRT) on the CPU. The DACRT algorithm does not construct any explicit acceleration structure and traces and traverses an implicit acceleration structure in tandem. We describe the original serial CPU algorithm followed by indicating the areas where the algorithm can be effectively parallelized thereby leading to better performance. We then proceed ahead to present our new parallel divide and conquer raytracing (PDACRT) running entirely on the GPU followed by results and comparisons against the serial CPU algorithm.

2.1 Background Work

Raytracing is a fundamental and well researched area in the field of computer graphics. Raytracing is the defacto algorithm employed to generate photorealistic imagery. With the advent of GPU programming GPUs have been employed to accelerate the raytracing pipeline as a whole i.e the acceleration structure construction step as well as the tracing phase. While the basic naive raytracing algorithm which computes the intersections between each ray and all the scene primitives is very simple it can become computationally infeasible when the size of rays or scene primitives becomes very large which is a common these days. Hence the naive raytracing algorithm was discarded in favor of methods which reduced the intersection computation complexity by employing acceleration structures such as Bounding Volume Hierarchies (BVH) [38], kd trees [4] and Grids [12].

However the compulsory requirement of having an explicit acceleration structure can become burdensome for developers of raytracing applications making software engineering complex. Further acceleration structure construction algorithms on the GPU are inherently very complex due to their parallel nature. Further the memory requirements of acceleration structures can become very large which can pose a problem on GPUs where memory is traditionally limited when compared to CPUs. Hence a new

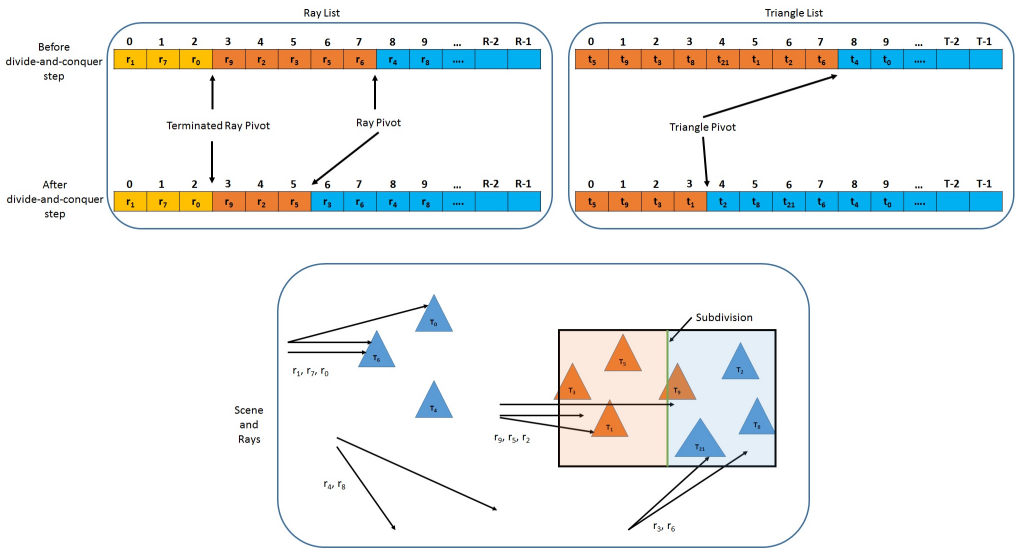


Figure 2.1: Figure showing serial DACRT in action for a particular scenario of triangles and rays. The variable *terminatedRayPivot* indicates those rays for which the nearest intersection has been computed and further processing is not necessary. Only elements from the start of the list or terminated pivots till the current ray/triangle pivot is processed in the current recursion level. Once a nearest intersection for a ray is computed the terminatedRayPivot is moved forward. Image adapted from Mora et al. [30]

method that computes the intersections between all rays and scene primitives in a naive manner while having very less memory requirements and is also very simple to implement is very much interesting.

The divide and conquer ray tracing (DACRT) [30] method performs raytracing without constructing any explicit acceleration structure. It does so by constructing and traversing an implicit acceleration structure in tandem on a CPU. The algorithm always works on a list of scene geometry elements and rays with implicit nodes represented by a range of elements within the list. It proceeds with the goal of reducing the search space for computing naive intersections between ray and scene primitives by employing a divide and conquer strategy similar to quicksort. The pseudocode for the serial CPU algorithm is given in Algorithm 2. The serial DACRT method takes as its input an array of scene primitives, an array of rays and the enclosing space or the bounding box of all primitives. The algorithm divides the enclosing space into a number of smaller ones. This creates a parent-child relationship between the split box and the resulting boxes. It then performs filtering operations to identify the rays and primitives that intersect the smaller child boxes. Rays which don't intersect any of the child boxes are eliminated. If the triangle-count or the ray-count of a box is small enough, the box along with its contents is sent for direct ray-triangle intersection evaluations. The variables *rLimit* and *pLimit* defined in Line 3 indicate the threshold condition for the number of rays and primitives and *NaiveRT* defined in 3 refers to the brute force intersection routine between the current set of rays and primitives. The ray-list and the triangle-list are updated to reflect their intersection relation to the remaining smaller

Algorithm 2 Serial DACRT

```
1:  procedure DACRT(Space E, SetOfRays R, SetOfPrimitives P)
2:      Begin
3:      if R.size() <rLimit or P.size() <pLimit then then NaiveRT(R, P)
4:      else
5:          E_i = SubdivideSpace(E)
6:          for each E_i do
7:          SetOfRays R' = R ∩ E_i
8:          SetOfPrimitives P' = P ∩ E_i
9:          DACRT(E_i, R', P')
10:     end if
11: end procedure
```

boxes. This process is recursively applied in a depth-first manner for each of the new bounding boxes in each recursion level until none remains.

The algorithm never creates any node structure like methods that construct an explicit acceleration structure do. Instead the implicit acceleration structure hierarchy is formed by considering ranges of elements within the list. The ranges of elements considered for a particular node within the implicit hierarchy is maintained by two *pivots* (Figure 2.1). Thes pivots act as markers to indicate positions in the ray and triangle lists where elements till the pivot point intersect the current bounding box whereas others don't. The elements between the start of the list and pivot indicate an implicit node within the implicit hierarchy. At each recursive step, elements are tested for intersection serially starting from index 0 till the pivot for all the child nodes computed by splitting the parent node. The elements that don't satisfy the intersection test are moved out and the pivot is advanced. The new pivot position indicates the elements of the child node created by splitting the parent node. When a pivot position indicates a small list, the box, rays, and triangles are sent for direct intersection computation.

The DACRT algorithm as such can be computationally expensive since the filtering operations involves a lot of data movement operations as well as repeated intersection tests. Hence a variety of optimization strategies were employed to accelerate the original algorithm. Conic packets capitalized on the coherent nature of primary rays to reduce number of filtering operations. Primary rays from the camera are coherent with respect to the direction of their traversal and hence a collection of rays can be approximated by a single cone which encapuslated all the rays. With this encapsulation, the number of rays were reduced to very small number of cones which was used in the filtering step. Once a threshold condition was reached, the contents of the cones were unpacked and a second run of the serial DACRT algorithm was run with this reduced set of triangles and rays. Rays that had their nearest intersection computed were masked and further processing of cones that contained them neglected these rays. Conic packets provided an order of magnitude improvement in speed to the original algorithm. However conic packets were utilized only for primary rays and not secondary rays due to their incoherent nature. Another optimization strategy was early ray termination that removed a ray from the list immediately after a nearest intersection was computed for it. This basically advanced the start of the list from which rays

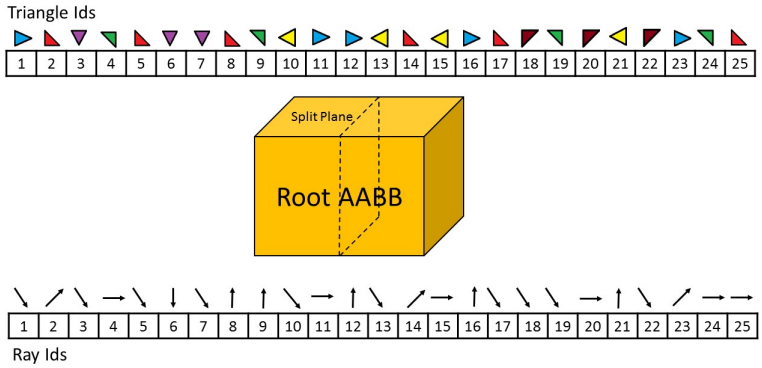


Figure 2.2: Implicit root node is defined over the input triangle and ray index lists. Each triangle and ray is handled by a thread to determine if it intersects the child AABBs afer split

were considered for filtering. Simplified triangle-AABB intersection tests were employed to quickly approximate filtering results and SSE capabilities of the CPU were employed to perform filtering in a SIMD manner. These optimizations were critical to the reported performance. Further Mora et al. presented a simple multithreaded variant of the algorithm in which each CPU core handled a separate list of ray and triangle indices. Each core went through the entire recursive step independently. Results from each were aggregated at the very end. The performance characteristics of the algorithm were also studied in a totally incoherent setting in path tracing [1]. The performance of the algorithm was also improved in a recent work [31]. They proposed an efficient DACRT method that exploited the distribution of rays by sampling the rays to construct an acceleration data structure. Further the ray traversal phase was accelerated by deriving a new cost metric which was used to avoid inefficient subdivision of the intersection problem where the number of rays was not sufficiently reduced.

2.2 Parallel Divide and Conquer Ray Tracing

The serial CPU DACRT algorithm exposes parallelism that can be employed to accelerate the algorithm as a whole. The filtering step basically computes ray/object intersection tests with an implicit node represented by an enclosing bounding box. All the intersection tests between the rays and objects with the bounding box can be done in parallel. This is called *intra-node parallelism*. The serial algorithm proceeds in a depth first manner using recursive calls. The depth first recursive call tree can be converted to a breadth first processing of each level of recursion where each recursion node represents a filtering or a naive intersection computation. In the breadth first view of the implicit hierarchy, all the implicit sibling nodes in the level can have their filtering operations done in parallel. We term this as *inter-node parallelism*. All the naive intersections between rays and objects within an implicit node can be computed in parallel once the implicit node's contents reach the threshold condition. Our parallel algorithm is designed to utilize all these sources of parallelism.

Algorithm 3 Parallel DACRT

```
1: procedure PARALLELDACRT(TriangleArray △, RayArray ↑, Scene AABB □, Global mints array,
   Global hitid array)
2:     Initialize level with root info
3:     while true do
4:         Split all current level nodes into child nodes in parallel
5:         Perform Triangle & Ray Filtering in parallel
6:         Calculate child-node pivots; mark terminal-nodes
7:         if (terminal-node buffer almost full) then
8:             Process all terminal-nodes
9:             Update global mints and global hitid
10:        end if
11:        Compute next level node sizes in parallel
12:        Gather next level triangle and ray index data in parallel
13:        Copy terminal-nodes to buffer in parallel
14:        Re-index next level pivots in parallel
15:        Create next level and assign it to current level
16:        Free memory for current level
17:    end while
18:    if (terminal-node buffer not empty) then
19:        Process all remaining terminal nodes
20:        Update global mints and global hitid
21:    end if
22: end procedure
```

We now present our parallel version of the algorithm running on the GPU. The pseudo code for our algorithm is given in Algorithm 3. A *node* represents a node in the implicit hierarchy. It contains a number of triangles and an enclosing AABB, along with a number of rays that intersect the AABB. A *child-node* is created from a parent node by splitting its AABB according to some condition, and contains a subset of its parent's triangles and rays. A *level* represents the group of nodes at a fixed depth from the root of the implicit hierarchy. A *pivot* is a two component vector that stores the start and end points of a group of elements belonging to a node within a linear list. Every ray stores the distance *mints* to the nearest intersected object indicated by *hitid*. *Global mints* and *Global hitid* arrays store parameter information for all input rays. *Terminal-nodes* have triangle or ray count below a fixed threshold. *Terminal-node buffer* represents a buffer in which all terminal nodes are stored temporarily before processing. The buffer also contains a separate *mints* and *hitid* arrays to store temporary ray parameters.

The Parallel DACRT algorithm starts with a linear list of rays and triangles along with the scene bounding box as shown in Figure 2.2. The algorithm iteratively constructs an implicit hierarchical structure over the list of triangles and traverses it level by level at the same time. The data is always kept in linear lists though referred to as *nodes* and *levels*. Nodes are implicitly defined over a linear list (*level*) using pivots that indicate the range of elements belonging to it. Nodes have pivots for both

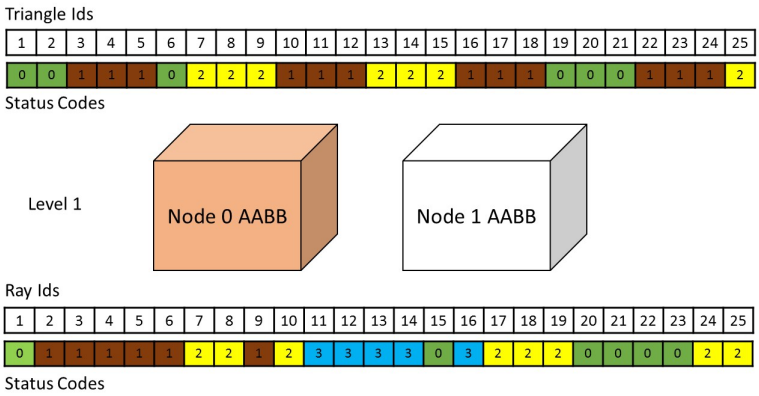


Figure 2.3: Child AABBs created after splitting. Parallel candidate tests on triangles and rays yield the status codes as shown. Elements have their status codes coloured to indicate left (green), both (brown), right (yellow) or none (blue)

triangle and ray lists. The implicit hierarchy is created by splitting nodes into children. The splitting scheme defines the type of hierarchy constructed. Spatial splitting of nodes results in a k-d tree while splitting of objects results in a BVH. We follow a spatial splitting scheme which splits a node's AABB into two child AABBs at the midpoint of the largest extent dimension. Once a node is split, the filtering operation determines the triangles and rays that intersect the child AABBs. Ray filtering works similar to a breadth first ray tracing where groups of rays are traced together down the implicit hierarchy. This scheme prevents the use of early ray termination as construction and building are performed together. However, the ray and triangle filtering offer fine-grained parallelism as every ray and triangle within every node can be handled independently. We exploit this to accelerate the process as a whole.

2.2.1 Parallel Filtering

The pseudo code for parallel filtering operation is given in Algorithm 4. *Element list* corresponds to either the ray list or triangle list and *Element id list* corresponds to respective id list.The filtering operation defined in Line 5 in Algorithm 3 is a per element intersection test with an AABB. We refer to the result of the filtering operation as a *status code* (Figure 2.3). The *status codes* returned by the filtering operations are used to rearrange the elements of the list so that elements with the same code occur contiguously as segments within the list (Figure 2.4). The *segments* correspond to nodes and are identified uniquely using *pivots*.

Consider a level (`Root` in Figure 2.2) in the hierarchy containing n nodes each with t triangles and r rays that needs filtering. When a parent node is split, its triangles can be in one, both, or none of the child nodes. A ray of the parent's AABB can intersect one, both, or no children's AABB. Hence, the output of the candidate tests for triangles can be *left*, *right* or *both*, encoded for each triangle as 1, 3, or 2, respectively. For rays, the possible results are *left*, *right*, *both* or *none*, encoded respectively as 1,

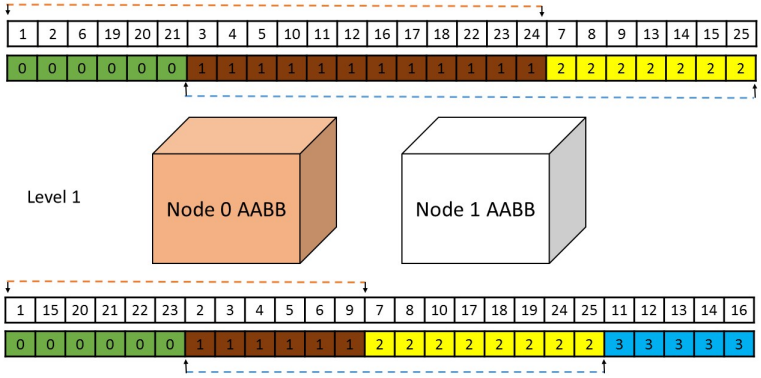


Figure 2.4: Sort-by-key rearranges elements of the two child nodes maintaining pivot property. The pivot ranges are marked with coloured dashed lines. Pivot ranges can substantially overlap between sibling nodes at any level.

3, 2, and 4 (Figure 2.3). The status code is encoded as an unsigned integer with 30-bits for the node number and right-most 2 bits for the intersection result. We then sort with the status code as the key and the element id as the value. The elements that straddle both children will be between those that intersect the left or right child after the sort (Figure 2.4). We can then extract the pivot ranges for the left and right children of each node easily from this representation. The left pivot's range would cover elements with codes *left* and *both* and right pivot's range would cover *both* and *right* (Figure 2.4). A segmented parallel reduce(sum) operation on the status codes would yield values that are multiples of their respective status codes. Dividing each value by its status code would return the number of elements for each node considered (Figure 2.5).

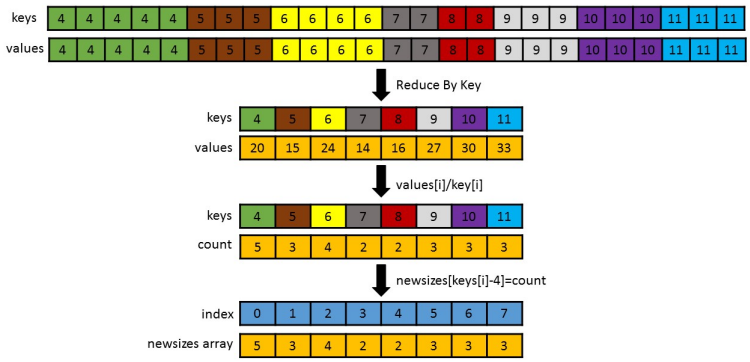


Figure 2.5: Parallel size calculation for child nodes at a level

After parallel filtering, a kernel marks all nodes that have low triangle ($< \alpha$) or ray ($< \beta$) counts as terminal-nodes (Figure 2.6). We then compute the sizes required for the next level nodes to store the ids.

Algorithm 4 Parallel Filter Algorithm

```
1:  procedure PARALLELFILTER(Aabb List, Element List, Element id List, Segment List, Pivot List)
2:      for each segment i ∈ (0, numsegments] in parallel do
3:          for each element j ∈ (0, numelements] in parallel do
4:              Split parent AABB into Left and Right
5:              Compute element intersection with AABB splits
6:              Compute status code and update keys
7:          end for
8:      end for
9:      parallel ReduceByKey(keys, keys, newkeys, newvals)
10:     for each key i in newkeys do
11:         Compute count of each status code
12:     end for
13:     for each i ∈ (0, numsegments] in parallel do
14:         Compute left, both, right and none values
15:         Update child pivot values
16:     end for
17: end procedure
```

We allocate enough memory for the next level nodes and a parallel gather operation copies nodes into their respective locations. The terminal nodes have their ids and data copied to a buffer and processed as described in section 2.2.2. We then re-index the pivot elements to account for the terminal nodes that have been removed from the list. We free memory used for the current level and proceed ahead to the new level created in a similar manner, till we have no more levels to process. We use $\alpha = 256$ and $\beta = 256$ for best performance.

Algorithm 5 Parallel Segmented Ray-Triangle Intersections

```
1:  procedure NAIVEINTERSECTION(TerminalNodeBuffer, Triangles List, Rays List)
2:      for each segment i ∈ (0, numsegments] in parallel do
3:          for each ray j ∈ (0, numrays] in parallel do
4:              Batch load triangle data into shared_memory[256]
5:              for each triangle k ∈ shared_memory do
6:                  Compute ray-triangle intersection
7:                  Update ray intersection parameters
8:              end for
9:          end for
10:     end for
11:     // Using buffer values
12:     parallel SortByKey(rayid, mints, hitids)
13:     parallel ReduceByKey(rayid, mints, hitids)
14: end procedure
```

2.2.2 Segmented Naive Ray-Triangle Intersections

The pseudocode for processing the terminal nodes is given in Algorithm 5. In order to fully utilize the resources of the GPU, we follow a buffered approach where each terminal node's triangle and ray ids are copied into a pre-allocated buffer that has a fixed size of η elements. When the buffer contents cross some fixed threshold $\lambda \leq \eta$ (Line 7 in Algorithm 3), we launch a GPU kernel that computes naive intersections between all the rays and triangles in every stored node. Every ray is handled by one thread and blocks of N threads handle every segment (node). Triangles for each segment are loaded in batches of size T into shared memory (Line 5 in Algorithm 5). We then proceed to compute the minimum *mints* and corresponding *hitid* value of every ray in the buffer by sorting the buffer *mints* and *hitid* value by using the ray-id and then performing a segmented minimum reduction using the ray-id as key. We then launch a kernel that updates the *global mints* and *hitid* arrays appropriately. In our experiments we empirically set the value of $N = 256$ and $T = 256$.

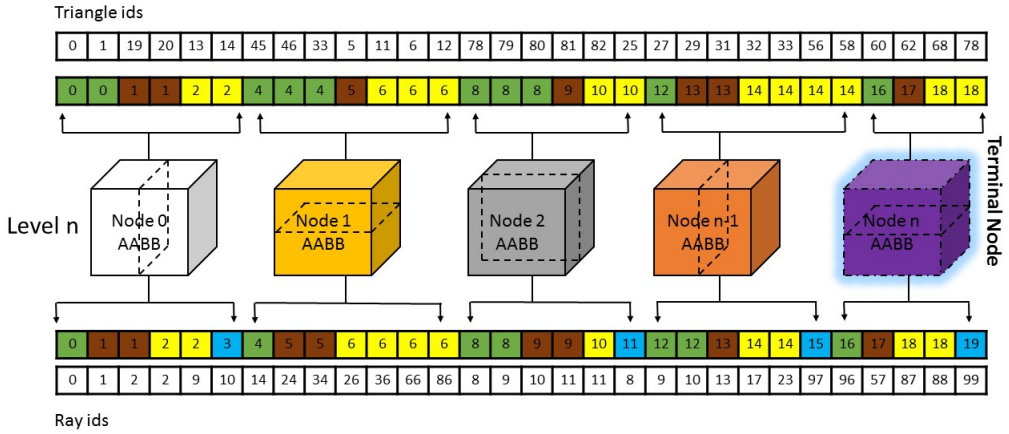


Figure 2.6: Nodes at level n. All nodes indicate their pivot ranges within a triangle and ray list maintained on a per level basis. The highlighted node is marked for naive intersections and its contents are copied directly into the terminal node buffer at the end of the iteration

2.3 Results

We have implemented our algorithm using CUDA on a machine having an Intel Core i7-920 CPU and a Nvidia GTX 580 GPU. All reported results are averages of values obtained during rendering from 12 different viewpoints. The scenes employed are standard test scenes (Figure 2.7) used to test the performance of raytracing algorithms.

2.3.1 Performance

Scene	CPU DACRT	PDACRT
Bunny	105 ms	87 ms
Conference	99 ms	148 ms
Angel	n/a	143 ms
Dragon	n/a	155 ms
Buddha	238 ms	165 ms
Turbine	285 ms	223 ms

Table 2.1: Comparison of ray casting time between CPU DACRT[30] and Parallel DACRT (PDACRT). CPU DACRT results indicate ray casting time in a single core 3Ghz Core 2 machine. All scenes were rendered at 1024x1024 resolution

Dynamic scenes change every frame. DACRT builds an implicit hierarchy and traces the hierarchy simultaneously. Table 2.1 compares the performance of our PDACRT method with the CPU DACRT work [30]. The timings reported are for primary rays along with the implicit structure creation. Our algorithm performs better than CPU DACRT for all the scenes except the Conference scene. The early ray termination using cones makes the sequential CPU version fast for Conference, as the viewpoint is inside the bounding box. Our PDACRT does a lot of work lacking early termination. Inspite of this, we are 1.2 to 1.5 times faster than the CPU implementation on other scenes with viewpoints outside the bounding box. The Conference scene needs 344 ms without cone tracing on the CPU [30]. The effect of parallel acceleration is thus much higher than is indicated in Table 1 as a result. PDACRT also performs much better compared to the best parallel CPU k-d tree method, which needs 654ms and 835ms respectively to build the k-d tree for Dragon and Buddha models on a 32 core machine [6]. The best GPU k-d tree construction method also needs 511ms and 645ms respectively for Dragon and Buddha [47]. PDACRT is 4-5 times faster to construct-and-trace these models. We also tested the performance of our algorithm for secondary rays with path tracing. Our path tracing performance with a 1024×1024 resolution and 7 diffuse bounces for the Conference and Sponza scenes are 5.9 MRays/sec and 4.8 MRays/sec respectively, compared to 5.5 and 4.6 MRays/sec reported by Mora et al. [30]. Table 2.2 shows performance of PDACRT for shadow, specular reflection, and ambient occlusion rays. Shadow rays generated towards a point light source have little coherence. As can be seen from Table 2.2, PDACRT is a little slower on the secondary ray pass than the primary pass, with higher difference on small object models when the viewpoint is inside the bounding box due to the highly incoherent nature of the rays. The performance for ambient occlusion (AO) rays (Table 2.2) show that even large number of incoherent rays is handled well by PDACRT.

Scene	PDACRT Shadow rays	PDACRT Specular Refl. rays	PDACRT AO rays
Bunny	67 ms	96 ms	149 ms
Conference	197 ms	222 ms	240 ms
Sponza	220 ms	246 ms	280 ms
Angel	102 ms	163 ms	182 ms
Dragon	128 ms	177 ms	192 ms
Buddha	150 ms	190 ms	204 ms
Turbine	213 ms	252 ms	287 ms

Table 2.2: PDACRT performance results for shadow rays, specular reflection rays and ambient occlusion rays. All results were generated at 1024×1024 resolution. Shadow rays were generated with one point light source. Ambient occlusion rays were generated with 8 AO rays per primary ray intersection.

2.3.2 Memory Requirements

A key advantage of DACRT is its low memory footprint. The CPU implementation by [30] follows a recursive depth first approach. Their memory requirements were for the two pivots (triangle and ray) and for the recursion stack only. Parallel DACRT implementation needs to store and maintain auxiliary information about all the nodes at any level during each iteration. More memory is needed as a result, but much lower than more traditional acceleration structures. Table 2.3 compares our peak memory usage with the GPU Kd-Tree [47] for primary rays at 1024×1024 resolution. All scenes were rendered with a buffer of size 24 MB except Bunny (8 MB buffer). In case of larger scenes or scenes with a large amount of rays, more memory would have to be allocated but still significantly lower than memory required by traditional acceleration structures.

Scene	PDACRT	GPU SAH KD-Tree
Bunny	47.66 MB	33.96 MB
Fairy	82.25 MB	80.33 MB
Exploding	82.68 MB	86.58 MB
Conference	85.82 MB	159.98 MB
Angel	82 MB	218.26 MB
Dragon	96.87 MB	417.33 MB
Buddha	107.89 MB	512.65 MB

Table 2.3: Comparison of peak memory usage for various scenes between Parallel DACRT and GPU SAH KD-Tree [47] construction method. Parallel DACRT (PDACRT) values are for scenes rendered at 1024 x 1024 resolution. Parallel DACRT values include memory requirements for buffer storage, triangle and ray data (32MB) also. Values for GPU SAH KD-tree construction include memory for triangle data only and not ray data.

2.3.3 Limitations

The level-wise processing of nodes results in duplication of ray ids which can be quite considerable at deeper levels, as a ray can intersect multiple nodes in a level. The duplication can be large for incoherent rays since rays can traverse deeply before they are culled. This behaviour was seen particularly for indoor scenes (like Conference) where the camera is inside the bounding box. All rays test positive for intersections at the upper levels of the hierarchy and have their ids duplicated. This duplication can lead to increased memory requirements when enough rays and triangles are not culled effectively. This behaviour is visible with relatively higher memory requirements for Fairy and Conference scenes (Table 2.3). Another issue is the lack of optimization strategies such as ordered traversal that is available when tracing individual rays through an acceleration structure. Parallel DACRT always processes rays in batches across all nodes in a level and hence ordered traversal is not possible.

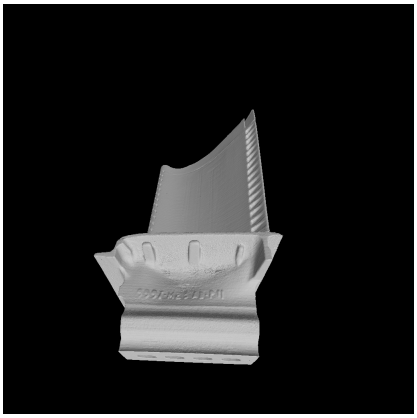
2.4 Conclusion

We have presented a fully parallel DACRT algorithm on the GPU. Ours is the first GPU implementation of this promising method. The DACRT approach has a low memory footprint that makes it attractive for GPU processing. We transform the original algorithm using a level-wise processing to fit the data-parallel model of the GPU. Several steps are mapped to standard primitives for which efficient implementations are available. We achieve better performance than reported on high-end CPUs so far. Future work would be to modify the algorithm that harnesses the benefits provided by the hardware modifications in the upcoming families of GPUs.

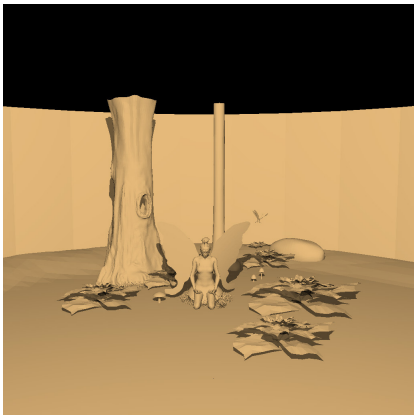
(a) Buddha (1.08M, 315 ms)

(b) Dragon (871k, 283 ms)

(c) Angel (474k, 143 ms)



(d) Turbine (1.7M, 223 ms)



(e) Fairy (174k, 276ms)

(f) Budda - 8 AO rays per pixel (410 ms)

Figure 2.7: Parallel DACRT results. All images were rendered at 1024x1024 resolution.

Chapter 3

Light Transport Algorithms

This chapter provides an introduction to the light transport problem. Light transport algorithms are employed in the shading stage of the raytracing pipeline and are responsible for determining the color of the surface being shaded. Light transport algorithms are generally physically based and they model the flow of light within the scene. Given the scene description along with accurate physical representations of surface properties of the objects, light transport algorithms can generate extremely realistic visualiztions of the scene since they capture the physics behind light flow. Hence inorder to understand the concepts of light transport algorithms, we need to have some understanding of the physical concepts employed in them. We begin the chapter by introducing the various radiometric terms that are essential to understand the light transport equation. We then present the light transport equation that is derived from these radiometric quantities. We then present the path integral framework of light transport [44] in which the light tranport equation is modelled as an integral equation over light carrying paths. We then present the different approaches employed to solve the path integral formulation particularly in the unidirectional and bidirectional frameworks. We then present a small introduction to Monte Carlo integration that is fundamental for solving integration problems and how its employed to solve the light transport problem. Most of the equations described in this chapter have been adapted from Veach et al. [44].

3.1 Radiometric Quantities

Radiometry is a set of techniques to measure electromagnetic radiation. Light is a form of electromagnetic radiation and since light transport algorithms generally work under the assumption of employing visible light, a working knowledge of radiometric terms is very essential. We now discuss some of the most important radiometric quantities that will be used in defining the light transport equation. Each of these quantities is defined by measuring the distribution of energy with respect to one or more parameters.

Radiant power is defined as energy per unit time:

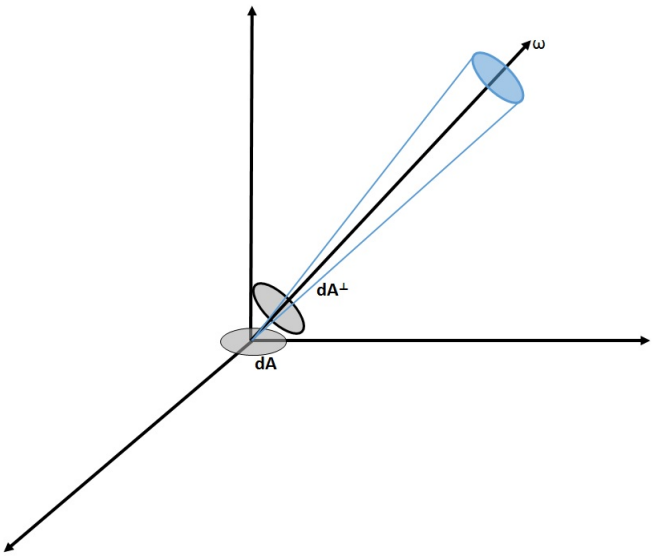


Figure 3.1: Radiance L is defined as the flux per unit solid angle per unit projected area. The cone represents the differential solid angle containing the direction considered. The circle at the base represents the differential projected area. Image adapted from Pharr et al. [36]

$$\phi = \frac{dQ(t)}{dt}, \tag{3.1}$$

and is measured in watts [$W = J.s^{-1}$]. Power is used to describe the rate at which energy is absorbed or emitted by a finite surface.

Irradiance is defined as power per unit surface area:

$$E(x) = \frac{d\phi(x)}{dA(x)}, \tag{3.2}$$

and is measured in units of $W.m^{-2}$. Irradiance is always defined with respect to a point x on a surface S with a specific normal $N(x)$. The term irradiance also generally implies the measurement of the incident radiation on one side of the surface only. However when light is leaving the surface, through either emission or scattering, the term *radiant exitance* (denoted by the symbol M) is used.

Radiance is the most important quantity for light transport calculations and is defined by

$$L(x, \omega) = \frac{d^2\phi(x, \omega)}{dA^{\perp}_{w}(x)d\sigma(\omega)}, \tag{3.3}$$

where $A^{\perp}_{\omega}$ is the projected area measure (Figure 3.1), which measures area on a hypothetical surface perpendicular to ω. In order to measure the radiance at (x, ω), we count the number of photons passing

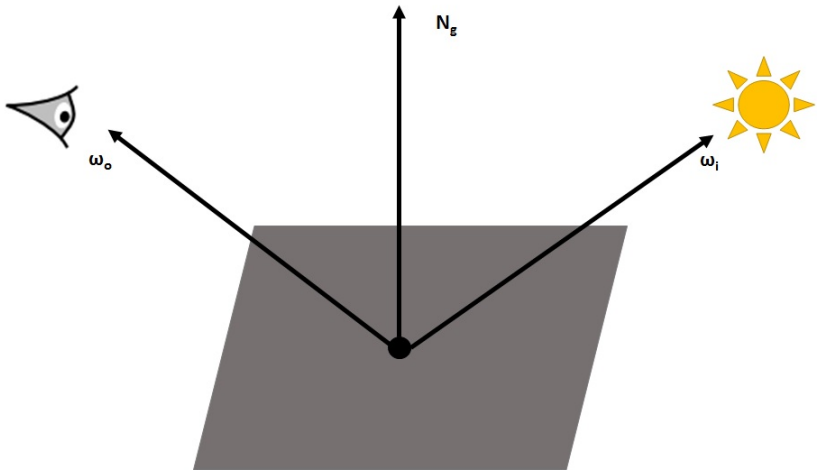


Figure 3.2: Geometry for defining the BSDF at a surface point. Image adapted from Veach et al. [44]

per unit time through a small surface $dA^{\perp}_{\omega}(x)$ perpendicular to ω, whose directions are contained in a small solid angle $d\sigma(\omega)$ around ω. The units of radiance are $[W.m^{-2}.sr^{-1}]$.

3.2 Bidirectional Scattering Distribution Function

The bidirectional scattering distribution function (BSDF) is a mathematical description of the light scattering properties of a surface. The BSDF at a point x is defined by:

$$f_s(\omega_i \rightarrow \omega_o) = \frac{dL_o(\omega_o)}{dE_i(\omega_i)} = \frac{dL_o(\omega_o)}{L_i(\omega_i)d\sigma^{\perp}(\omega_i)} \tag{3.4}$$

In other words, the BSDF at a surface point is the ratio of the outgoing radiance to the incoming irradiance for a given pair of directions i.e ω_i and ω_o. The notation $\omega_i \rightarrow \omega_o$ symbolizes the direction of light flow (Figure 3.2).

3.2.1 Scattering Equation

Given the definition for the BSDF in Equation 3.4 we can now proceed ahead to define the scattering equation. By integrating the relationship

$$dL_o(\omega_o) = L_i(\omega_i)f_s(\omega_i \rightarrow \omega_o)d\sigma^{\perp}(\omega_i), \tag{3.5}$$

over all the incoming directions, we can predict the radiance leaving a surface point. This is the surface scattering equation and is given by

$$L_o(\omega_o) = \int_{S^2} L_i(\omega_i)f_s(\omega_i \rightarrow \omega_o)d\sigma^{\perp}(\omega_i). \tag{3.6}$$

The Equation 3.6 can be used to predict the appearance of the surface, given a description of the incident illumination.

3.2.2 BRDF and BTDF

The concept of BSDF as defined in Section 3.2 is not standard in radiometry. The scattered light from a surface point is subdivided into reflected and transmitted components. Each of these components are treated separately. This separation provides us with the definitions for the bidirectional reflectance distribution function (BRDF) and the bidirectional transmittance distribution function (BTDF) denoted by f_r and f_t generally used in the scattering equations. The BRDF is defined for the hemisphere containing both the incoming and outcoming directions whereas the BTDF is defined for directions contained in opposite hemispheres of the surface point.

3.2.2.1 BRDF Properties

The BRDFs used to define the surface properties have certain basic properties which are defined as below.

- Symmetry: The BRDFs are symmetric in the aspect that swapping the incoming and outgoing directions in the BRDF definition does not change the value of the BRDF. Hence

$$f_r(\omega_i \rightarrow \omega_o) = f_r(\omega_o \rightarrow \omega_i) \quad \text{for all} \quad \omega_i, \omega_o. \tag{3.7}$$

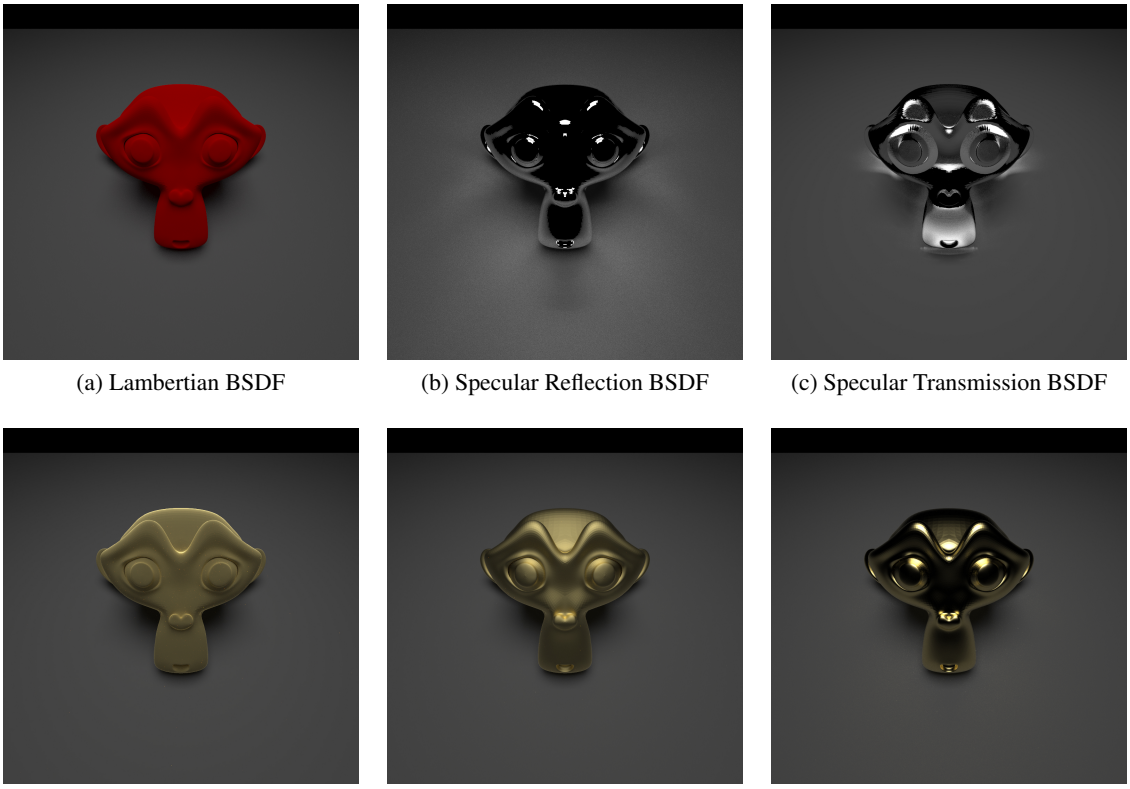
- Energy Conservation: BRDFs generally conserve energy. This is fundamentally important because a surface cannot reflect more energy than it receives unless and until it is an emitter which is traditionally modelled as a separate self emission term in the equations. The conservation of energy of the BRDF is given by the condition

$$\int_{\mathcal{H}_o^2} f_r(\omega_i \rightarrow \omega_o) d\sigma^{\perp}(\omega_o) \leq 1 \quad \text{for all} \quad \omega_i \in \mathcal{H}_i^2. \tag{3.8}$$

  where the terms $\mathcal{H}_o^2$ and $\mathcal{H}_i^2$ represent the hemispheres containing the incoming and outgoing directions.

3.2.3 Surface Reflection Models

Surface reflection models describe how light is reflected from scene objects. Many surface reflection models have been developed that can result in rendering very convincing looking objects. Surface reflection models have been developed from a variety of sources. The reflection distribution properties of many surfaces have been closely measured in laborataries to yield tables that are used while rendering. Phenomological models employ equations that attempt to describe the qualitative properties of



(a) Lambertian BSDF (b) Specular Reflection BSDF (c) Specular Transmission BSDF

(d) Microfacet BSDF - rough (e) Microfacet BSDF - medium smooth (f) Microfacet BSDF - very smooth

Figure 3.3: Different Types of BSDFs

real world surfaces. These models are very easy to use and have intuitive parameters that modify the appearance of the object. Low level information about the composition of surfaces might be known and they can be used to simulate the appearance of the object. For example, we might know that car paint is comprised of colored particles of some size suspended in a medium and we can use this to simulate how this particular paint would like. Some reflection models have been derived using a detailed model of light, treating light as a wave and computing the solutions to Maxwell's equations. Geometric optical models enable modelling closed form reflection models when the surface's low level scattering and geometric properties are known.

3.3 Types of BRDFs

We now describe a few of the common BRDFs that are widely used to model how light reflects off a surface. The BRDFs generally used are modelled using a variety of the aforementioned techniques such as measured data, phenomenological models and geometric optics.

(a) 2 layer glossy paint composed by adding a diffuse layer with a specular reflective layer

(b) 2 layer metallic foil composed by adding a microfacet conductor layer with a specular reflective layer

(c) 3 layer material composed of two specular reflective layers and a microfacet conductor material

Figure 3.4: Layered Materials

3.3.1 Lambertian

The Lambertian model is one of the simplest BRDF models available. It models a perfect diffuse surface that scatters incident illumination equally in all directions. Although this model is not physically plausible, it is a good approximation to many real-world surfaces like matte paint.

3.3.2 Specular Reflection

Specular reflection models the reflection of light from an intersection point in a mirror direction. Generally BRDFs of specular reflection type are modelled using dirac-delta functions since for an incoming direction, there is only one out going direction. Hence probabilistically choosing the mirror direction is zero and hence special sampling routines are required to handle such BRDFs.

3.3.3 Specular Transmission

Specular transmission models the flow of light as it crosses a boundary at the intersection point. Similar to specular reflection, for an incoming direction there is exactly only one transmitted direction. The transmitted direction is computed using Snell's law given the refractive indices of the two layers across which light crosses. The BRDF is modelled using a dirac delta function since there is only transmitted direction for the incoming direction.

3.3.4 Microfacet Models

Microfacet models [9] describe surface reflection under the assumption that rough surfaces can be modelled as a collection of small microfacets. A surface comprised of microfacets is essentially a heightfield, where the distribution of the facets is described statistically. The two main components

of microfacet models are an expression for the distribution of facets and a BRDF that describes how light scatters from the individual microfacets. Perfect mirror reflection is used typically however the Oren-Nayar [33] model treats the microfacets as Lambertian reflectors.

3.3.5 Layered Materials

The single layered materials that can be represented with the BSDFs presented above might lack visual variety and hence a different system in-order to represent rich varied materials is required. Layered material models are aimed at creating rich material variety by compositing different BSDF types together to form a material stack. A variety of methods have been developed to represent layering in materials [36, 45, 18]. Depending upon the method the variety of effects that can be captured using layering of BSDFs can differ. Although the layered models are able to provide rich material representations, they can be very expensive to evaluate since all the individual BSDF layers have to be evaluated and accounted for. This can put significant strain on the rendering algorithm's runtime performance.

3.4 Light Transport Equation

We now present the full light transport equation using the quantities and equations described earlier. Generally, we are most interested in the steady state or equilibrium radiance for a given scene. It is conventional to solve for the exitant version of the quantity L_o, from which the incident radiance L_i can be obtained using

$$L_i(x, \omega) = L_o(x_{\mathcal{M}}(x, \omega), -\omega), \tag{3.9}$$

where $x_{\mathcal{M}}(x, \omega), -\omega)$ is the ray-casting function that returns the first point of scene surfaces $\mathcal{M}$ visible from x in direction ω.

We can express express L_o as the sum of the emitted radiance L_e and the scattered radiance $L_{o,s}$ as

$$L_o = L_e + L_{o,s}. \tag{3.10}$$

The emitted radiance function is provided as part of the scene description and represents all the light sources in the scene. On the other hand, $L_{o,s}$ is determined using the scattering equation, according to which

$$L_{o,s}(x, \omega_o) = \int_{S^2} L_i(x, \omega_i) f_s(x, \omega_i \to \omega_o) d\sigma_x^{\perp}(\omega_i). \tag{3.11}$$

By putting these two equations together, we can derive the full equation for the light transport equation as:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_{S^2} L_o(x_{\mathcal{M}}(x, \omega), -\omega) f_s(x, \omega_i \rightarrow \omega_o) d\sigma_x^{\perp}(\omega_i). \tag{3.12}$$

This formulation of light transport is recursive in nature as the incoming radiance at point is given by the outgoing radiance at another point. The outgoing radiance from the second point again can be computed using the incoming radiance from a third different point. Each level of recursion provides us with a particular lighting effect and it represents increasing bounces that light has taken to reach the point under consideration. For example a one bounce of light from light source to camera represents direct lighting of a surface as seen by the camera. Full global illumination solutions try to capture all the lighting information as much as possible. Different light transport algorithms for the sake of computational efficiency clamp the maximum recursion depth or employ certain techniques to intelligently reduce the bounce limit.

The ultimate goal of light transport is to compute a set of real valued measurements $I_1, ..., I_M$. For example an algorithm that computes an image directly, each measurement I_j represents the value of a single pixel and M is the number of pixels in the image. Each measurement corresponds to the output of a hypothetical sensor that responds to the radiance $L_i(x, \omega)$ incident upon it. The response of the sensor may vary depending upon the position and direction of the incoming light. This is characterized by the sensor responsitivity $W_e(x, \omega)$ [7]. Hence the total response is computed by integrating the product $W_e L_i$, which is defined as the measurement equation and is given by

$$I = \int_{\mathcal{M} \times S^2} W_e(x, \omega) L_i(x, \omega) dA(x) d\sigma_x^{\perp}(\omega). \tag{3.13}$$

The incoming radiance L_i in Equation 3.13 can be computed in a recursive manner by shooting a ray from the sensor (using the raycast function defined in Equation 3.9) in the incoming direction and finding the first intersected point and computing the outgoing radiance from that point towards the sensor using Equation 3.11. The outgoing radiance from the first intersected surface point can also be computed in the same way in a recursive manner till some arbitrary condition such as depth of recursion is met or no scene object is intersected. These recursive methods of solving for the light transport equation can be done in a unidirectional way or a bidirectional manner.

Unidirectional methods generally try to solve the light transport equation by shooting rays and estimating the various quantities either from the camera sensor or the light. The method is called path tracing [8] when the procedure begins from the camera and moves towards the light. It is called as light tracing [11] when the procedure starts from the light and moves towards the sensor. Both these methods have advantages and disadvantages over one another depending upon the scene parameters.

Bidirectional methods construct light transport paths from both the sensor and the light at the same time and generally involve a connection phase which completes the full path [26]. Bidirectional methods can be considered to be the culmination of path tracing and light tracing. Bidirectional methods are particularly effective at solving difficult lighting scenarios such as highly occluded lights. This is because unidirectional path tracing will have only a very few or no paths that reach the occluded light

source at all. Light tracing however proceeds from the light source and might come out of the occluded region but might have a difficult time finding a good path to the camera sensor. Bidirectional methods explicitly connect these two subpaths to form a valid complete light path and therefore able to render such difficult scenes far more effectively.

3.5 Path Integral Framework

The path integral formulation of light transport [44] models how light moves within a scene and is recorded by a sensor within the scene. Each measurement recorded by the sensor can be written in the following form

$$I_j = \int_{\Omega} f_j(\overline{x}) d\mu(\overline{x}), \tag{3.14}$$

where Ω is the set of all light transport paths of all possible lengths, μ is the measure on this space of paths, and f_j is called the measurement contribution function. Each path $\overline{x}$ of length k is composed of vertices $(x_0, x_1, ..., x_{k-1})$. Vertex x_0 sits on a light source, the vertex x_{k-1} on the camera sensor, and the intermediate vertices on other surfaces of the scene.

The measurement contribution function f_j defined for a particular path $(\overline{x})$ of length k can be computed from the measurement equation defined in Equation 3.13 by recursively expanding the incoming L_i term upto k times. The expanded equation can be defined as:

$$\begin{aligned} f_j(\overline{x}) = {} & L_e(x_0 \rightarrow x_1) G(x_0 \leftrightarrow x_1) W_e^j(x_{k-1} \rightarrow x_{k-2}) \\ & \prod_{i=1}^{k-1} f_s(x_{i-1} \rightarrow x_i \rightarrow x_{i+1}) G(x_i \leftrightarrow x_{i+1}), \end{aligned} \tag{3.15}$$

where L_e is the radiance emitted from x_0 to x_1, G the geometry term between vertices, f_s the BSDF term and W the sensor importance function.

The path integral formulation for light transport has several advantages. Since the expression for each measurement is an integral, general purpose Monte Carlo integration methods can be applied to solve the equation. Second the path integral formulation has a simpler form rather than the light transport equation which has a recursive form. Further by dealing with paths, the framework also provides a complete description of light transport. Solving the light transport equation is traditionally done by sampling the equation in a recursive manner leading to paths starting entirely from the sensor or the light. With the path integral approach it is possible to construct paths of arbitrary lengths by starting from practically anywhere in the scene. A path can be constructured by starting from a vertex in the middle of the path and proceeding in both ways towards the camera and the light source or can be constructed in a unidirectional manner from either the camera or the light source. The only thing required is that vertices so chosen must be mutually visible to other vertices they share the path segment with.

3.6 Basic Monte Carlo Integration

The basic idea of Monte Carlo integration is to evaluate any integral of the form

$$I = \int_{\Omega} f(x) d\mu(x) \tag{3.16}$$

using random sampling. In the most basic form of Monte Carlo integration, this is done by independently sampling N points $X_1, ..., X_N$ according to some convenient probability density function p, and then computing the estimate

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}, \tag{3.17}$$

where the estimator's result F_N is a random variable and its properties depend on how many sample points were chosen. We can show that the estimator F_N gives the correct result on average. This is shown by

$$\begin{aligned} E[F_N] &= E\big[\frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}\big] \\ &= \frac{1}{N} \sum_{i=1}^{N} \int_{\Omega} \frac{f(x)}{p(x)} p(x) d\mu(x) \\ &= \int_{\Omega} f(x) d\mu(x) \\ &= I, \end{aligned}$$

provided that $f(x)/p(x)$ is finite whenever $f(x) \neq 0$. Monte Carlo method of solving the light transport equation generally involves sampling random points on scene surfaces, sampling random directions for next bounce ray computation, sampling random points on sensor or light and such. The dimensions considered for solving depend upon the kind of effects that are required to capture. For example, accurate motion blur of moving objects can be computed by sampling a time variable that would be used to compute the location of the object. Depth of focus can be simulated by sampling rays from locations on a realistic lens model. Spectral rendering can be simulated by sampling light rays of particular wavelength and capturing the contribution for that particular wavelength and accumulating the results. Antialiasing of the final image is performed by sampling a lot of points on the sensor and accumulating the results of rays through each sensor point. Hence inorder to capture any new effect, we just need a way of sampling a random variable from the domain and evaluating the function with that random variable and scaling it appropriately.

3.6.1 Advantages of Monte Carlo Integration

Monte Carlo integration has some major advantages compared to other numerical integration techniques. First the rate of convergence of Monte Carlo methods is $O(N^{-1/2})$ in any dimension, regardless of the smoothness of the integral. This makes it very useful for computer graphics where we are required to compute multidimensional integrals of discontinuous functions.

Second, Monte Carlo integration is very simple. The only two basic basic operations are required are sampling and point evaluation. This allows software to employ Monte Carlo integration modules as black boxes with greater flexibility. Hence it is very easier to implement a variety of effects such as motion blur, depth of field, etc. in computer graphics algorithms.

Third, Monte Carlo is very general because it stems from the idea of random sampling. Finally Monte Carlo methods are better suited than quadrature methods for integrands with singularities.

3.7 Sampling Random Variables

There a variety of routines for sampling random variables. One of the methods is the *transformation* or *inversion* method. Assume that we have to sample from a one dimensional density function p. We let P be the cumulative distribution function of p. The inversion method consists of letting $X = P^{-1}(U)$ where U is a uniform random variable drawn on $[0, 1]$. It can be verified that X has the required density p. This technique can also be extended to multiple dimensions either by computing the marginal or conditional distributions and inverting each dimension separately or more generally by deriving a transformation $x = g(u)$ with an appropriate Jacobian determinant (such that $|det(J_g(x))|^{-1} = p(x)$, where J_g indicates the Jacobian of g).

The main advantage of the inversion method is that it allows samples to be stratified easily by stratifying the parameter space $[0, 1]^s$ easily and mapping these samples to the required domain. Another advantage of the method is that it has a fixed cost per sample which can be easily estimated. However the main disadvantage of the method is that the density function $p(x)$ must be capable of being integrated analytically which is not possible always.

Another sampling method is the *rejection method* given by von Neumann. The idea is to sample from a convenient density function q such that

$$p(x) \leq Mq(x) \tag{3.18}$$

for some constant M. Generally the samples for q are generated by the transformation method. We then follow the following procedure rejection sampling procedure.

This procedure generates a random sample X whose density function is p. The main advantage of the rejection method is that it can be used even when the function from which samples are to be drawn is not analytically integrable. However we still need to be able to integrate some function Mq that is an upper bound for p. Furthermore this bound should be reasonably tight since the average number of samples

Algorithm 6 Rejection Sampling

```
1: procedure REJECTION SAMPLING
2:     for i = 1 to ∞ do
3:         Sample X_i according to q.
4:         Sample U_i uniformly on [0, 1].
5:         if U_i ≤ p(X_i)/(Mq(X_i)) then
6:             return X_i
7:         end if
8:     end for
9: end procedure
```

drawn before acceptance is M. Thus the efficiency of the rejection sampling method can be very low if applied naively. Another disadvantage of this method is that it is difficult to apply stratification.

3.8 Importance Sampling

Since the estimator in the Monte Carlo integration process is a random variable, properties such as variance, expected value, etc., can be defined for it. In the context of image generation, variance in the estimator manifests itself as noise. Noise is perhaps the most important factor that image generation algorithms try to reduce in an image to improve its quality. A variety of techniques have been employed to reduce the noise in the estimator. Perhaps the most important of all the variance reduction techniques is the importance sampling method which we discuss as follows.

Importance sampling refers to the method of choosing a density function p similar to the integrand f used in the Monte Carlo integral. It is a well know fact that by setting $p(x) = cf(x)$, where the constant of proportionality is

$$c = \frac{1}{\int_\Omega f(y)d\mu(y)}. \tag{3.19}$$

This leads to an estimator with zero variance since

$$F = \frac{f(X)}{p(X)} = \frac{1}{c}, \tag{3.20}$$

for all sample points X. However this technique is not practical since the procedure expects random variables drawn proportional to the original function that is being estimated in the first place. However good samples can be drawn by choosing functions that have similar shape to the original function being integrated.

Importance sampling in employed in the Monte Carlo integration method of the light transport equation. When solving the light transport equation random samples may be drawn based upon the function

being integrated rather than employing uniform random samples. For example, while solving Equation 3.11 it is advantageous to sample either the incoming radiance term L_i or the BSDF term f_s or both. The idea behind this method is the fact that samples that contribute most i.e important samples should be sampled with a high probability. Hence major source of illumination in case of incident radiance sampling or most contributing direction in case of BSDF sampling is preferred. Further in case of multi layered materials it might be advantageous to sample all the layers of the material to find the next bounce direction. However this might be computationally expensive since the number of rays would grow exponentially with each bounce and hence for computational purposes normally only one layer is chosen at random and a corresponding ray is sampled. However the BSDF of the layer chosen in employed for importance sampling the next bounce direction thereby retaining the benefits of importance sampling.

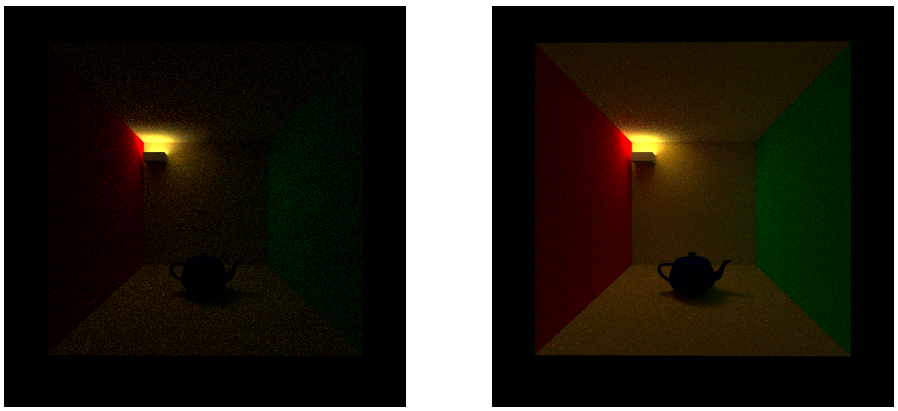
Chapter 4

Coherent and Importance Sampled - LVC BDPT (CIS-LBDPT)

In this chapter we describe a new method that addresses a particular problem in the shading stage of the raytracing pipeline. Specifically our method improves the performance and rendering quality of the Light Vertex Cache (LVC) formulation of bidirectional path tracing on the GPU. We begin by providing a brief overview about the bidirectional path tracing (BDPT) algorithm and how it is aimed at solving the path integral. We then provide a brief overview about the approaches taken to implement BDPT on a GPU and the complexities involved. We then describe the original Light Vertex Cache (LVC) approach of employing bidirectional path tracing on the GPU. We then proceed ahead to explain our new approach to improve the connection phase of the LVC formulation of the BDPT algorithm. We also provide information about our new sort based framework that is aimed at improving the performance of the algorithm in the presence of complex layered materials. We conclude the chapter by presenting results of our algorithm and comparing it against the original LVC algorithm.

4.1 Bidirectional Path Tracing

Light transport algorithms are aimed at modelling the flow of light within a scene. The rendering equation [20] describes light transport as an multi-dimensional integral equation. Monte Carlo methods have been employed to solve this equation since computing analytic solutions is extremely hard. Monte Carlo methods employ random sampling to compute the value of the integral and a large number of such samples are required to converge to the correct solution. The path space formulation [44] models light transport as an integral over light carrying paths rather than a recursive integral equation. In the path space formulation, a light transport path is generally created using local path sampling and can be sampled starting from a light source (light tracing), the camera (path tracing), or both (bidirectional path tracing). Although unidirectional methods are very capable of capturing most of the lighting effects in a scene, under certain conditions such as scenes having strong indirect lighting or lights inside fixtures, they fail to capture lighting effectively. Usually unidirectional methods require a large number of samples to converge to a relatively noise free solution. Bidirectional methods on the other hand are able to handle these very difficult lighting conditions easily compared to unidirectional methods (Figure 4.1).



(a) Path tracing (b) Bidirectional path tracing

Figure 4.1: Simple 'Cornell Box' like scene having an area light fixed inside an enclosure that is kept at the far end of the box. Both images were generated with fixing max path length to be 5. This is a very difficult lighting setup as most of the paths in path tracing never test for a positive connection to the light source thereby exhibiting lots of noise. BDPT on the other hand is able to handle this difficult lighting condition with ease.

Consider a light carrying path $\overline{x}$ consisting of k vertices from the light to the camera. Bidirectional path tracing first creates samples from both the camera and the light to create this light path. It then proceeds with generating the camera and light subpaths individually by tracing the sample rays. Assume that the light subpath to consist of s vertices $y_0, y_1, ..., y_{s-1}$ and the camera subpath to consist of t vertices $z_0, z_1, ..., z_{t-1}$. The vertex y_0 lies on the light source while the vertex z_0 lies on the camera sensor. The measurement contribution function 3.15 therefore can be split into three terms representing the values accumulated as a part of the camera, light subpath and a connection component. The light subpath's partial contribution terms are given by

$$A(\overline{x}) = L_e(y_0 \to y_1) \prod_{i=0}^{s-2} G(y_i \to y_{i+1}) \prod_{i=0}^{s-3} f_s(y_i \to y_{i+1} \to y_{i+2}) \tag{4.1}$$

and the camera subpath's partial contribution term is given by

$$B(\overline{x}) = W_e(z_0 \to z_1) \prod_{i=0}^{t-2} G(z_i \to z_{i+1}) \prod_{i=0}^{t-3} f_s(z_{i1} \to z_{i+1} \to z_{i+2}). \tag{4.2}$$

When the paths terminate due to Russian roulette or fixed depth clamping, the two subpaths are connected by shadow rays that connect a camera path vertex to a light path vertex. A connection between two vertices is termed valid only if the two vertices are mutually visible to each other, i.e, there is no occluder between them. Only valid connections contribute to the final pixel colour. The connection term between the two vertices is given by

$$C_{s,t}(\overline{x}) = f_s(y_{s-2} \rightarrow y_{s-1} \rightarrow z_{t-1}) f_s(z_{t-2} \rightarrow z_{t-1} \rightarrow y_{s-1}) \\ G(y_{s-1}, z_{t-1}) V(y_{s-1}, z_{t-1}). \tag{4.3}$$

With these partial terms, the final contribution term $f(\overline{x})$ can be computed as

$$f(\overline{x}) = A(\overline{x}) C_{s,t}(\overline{x}) B(\overline{x}). \tag{4.4}$$

In the Monte Carlo method, Equation 4.4 is solved by sampling a large number of random paths in path space and averaging the results of evaluating the value of the measurement contribution function and scaling it by the probability of sampling that path. Each path of length k indicates the number of bounces that light has taken to reach the camera sensor. It is generally inefficient to stochastically generate different paths of lengths k. Generally a multitude of paths of varying lengths are created together by connecting each vertex of the camera path with all the vertices of the light path thereby creating a family of different length paths. All these paths are weighted appropriately and their contributions added to the pixel.

The BDPT algorithm exposes parallelism at different stages of its execution: ray traversal, connection computation and shading. GPUs have been used to accelerate the BDPT algorithm by capitalizing on these sources of parallelism. BDPT was first explored in a hybrid combinatorial CPU-GPU setting in which all the camera and light subpaths were generate in the CPU [34]. Once all the subpaths were created, all the vertices were transferred to the GPU after which the GPU computed the visibility between all possible pairs of vertices. Once the visibility tests were complete, the results were again moved back to the CPU after which shading was completed and the image buffer updated. The first fully GPU BDPT implementation was done van Antwerpen et al [43]. They performed BDPT in a streaming manner in which a stream of samples from both the eye and the camera was kept alive at all points of time. Memory was preallocated for both the camera and light subpath vertices by clamping the maximum allowable length of subpaths. Once the number of active light and camera subpaths fell below a threshold limit, the algorithm computed the connections between the subpaths and updated the image. The dead samples were then removed from the stream using stream compaction and a new samples were regenerated in the free positions in the stream. This method posed a memory utilization problem since all the space allocated for each path would not be effectively utilized since the path were terminated in a stochastic manner.

4.2 GPU Performance Issues

High performance computing performance on the GPU is dependent on a variety of factors. The most fundamental performance determining factors include code execution coherency, coalesced data access and occupancy. Light transport algorithms by their very nature can be very GPU unfriendly

under certain conditions. We look at the problems that each phase of light transport algorithms pose when employed to run on a GPU.

4.2.1 Megakernels

Traditional path tracers on the CPU have a single function that computes the final color of a pixel. This single function normally encapsulates all the logic associated with ray traversal, next event estimation, shading computation, next bounce computation, etc. Direct mapping of the CPU function to a GPU kernel is possible. However such a direct mapping would result in an bloated megakernel. On a GPU, each thread is allocated some fixed amount of registers for all the variables and computations required. Each SM on the GPU can have more than one block of a thread simultaneously running on it. However with increased register usage, the hardware becomes unable to schedule concurrent blocks on thereby reducing the performance of the kernel. Further when the number of registers required by each thread increases the threshold, all the variables are spilled into global memory on the GPU. Hence any access of the variable requires a global memory read which can be very expensive. These problems were analysed by Laine et al. [27] and they proposed splitting the larger kernels into smaller kernels each doing a specific job such as traversal or shading. Since the kernels are small enough the hardware is effectively able to schedule them simultaneously thereby increasing the performance.

4.2.2 Code Execution incoherency

A GPU follows a SIMT (Single Instruction Multiple Thread) execution strategy. In this method all the threads in the warp execute in step. However all the threads execute the same instruction only if the instruction is the same. If there is any divergence in the code paths taken by the threads in a warp, the hardware masks all the threads that don't execute the code and executes only those threads that take that particular code path. In essence the amount of parallelism is reduced due to the serial execution of the group of threads in a warp one after another. With increase in code divergence, the performance drops drastically. In a raytracing application, shading is a very important source of code execution divergence. This is because each thread normally handles one particular ray which hits different objects in the scene thereby requiring access to different material data. This would result in each thread handling a different shading routine thereby increasing code execution divergence. This problem is further aggravated in the presence of complex layered materials since each thread can be executing code that handles each layer separately.

4.2.3 Coalesced Data Access

Another issue that decreases the performance of any raytracing algorithm on the GPU is uncoalesced data access. As discussed earlier, each ray traditionally handles a separate ray which might be traversing a different part of the scene and intersecting different objects. The shading and next bounce computation

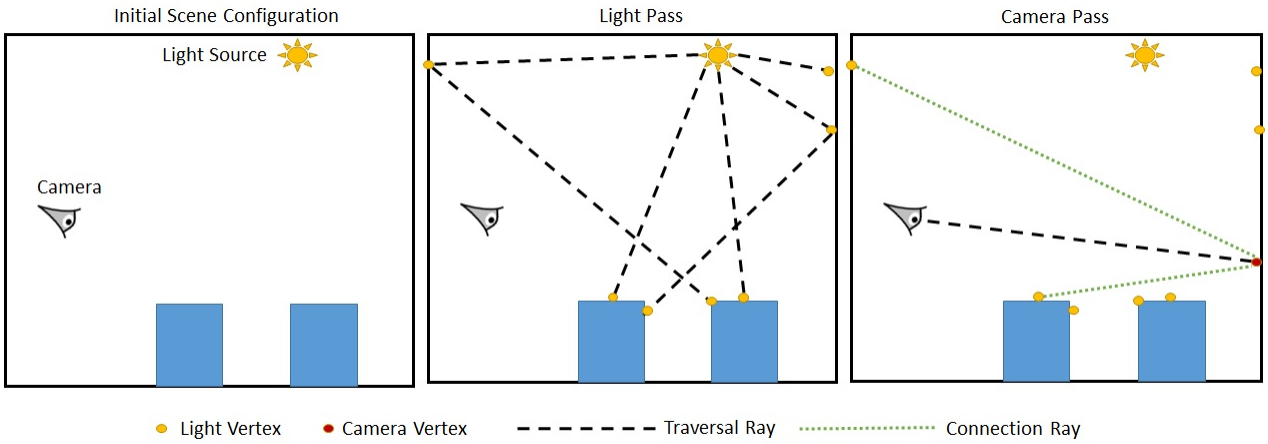


Figure 4.2: Light Vertex Cache BDPT. Assuming a prepass was performed that allocated enough space for the LVC. The algorithm first performs a light pass and stores the vertices. During camera pass N vertices are chosen from LVC for camera vertex connection. In the above figure $N = 2$.

routines require access to geometry and material data from all the individual objects. On a GPU memory requests are handled on a per warp basis. The memory controller is able to fetch 32, 64, 128, 256 contiguous bytes from global memory per memory read cycle. Hence if all threads access contiguous data from the global memory, the memory controller is able to serve all these requests in the least amount of time possible. However if the data requested is not coalesced i.e scattered in global memory then the hardware serves all memory requests in a serial manner thereby severely reducing performance. Hence threads handling rays that request different geometry and material data have uncoalesced memory accesses thereby reducing program efficiency.

4.3 Light Vertex Cache - BDPT

Davidovic et al. presented a Light Vertex Cache Bidirectional Path Tracing (LVC-BDPT) on the GPU [10]. The LVC formulation was aimed at solving the original problem of having to preallocate enough memory for both the light and camera paths in the GPU [43]. Their algorithm began by performing a coarse light prepass within the scene by using a fraction of the intended light paths. The pass was performed till all the paths were terminated. The average path length was then computed from the prepass and was used to allocate enough memory for all the light paths for the actual light pass. During the light pass the intermediate subpath vertices were stored in the LVC and the paths were forcibly terminated when the LVC was full. When the LVC buffer was full and the light pass completed, the camera pass began by starting paths from the camera. During every camera path intersection point, a random number N vertices were chosen from the LVC and were used for connection with the current camera subpath vertex to form a complete light transport path (Figure 4.2). The connections were evaluated and the image buffer was updated appropriately. Then the camera pass then continued with generating the

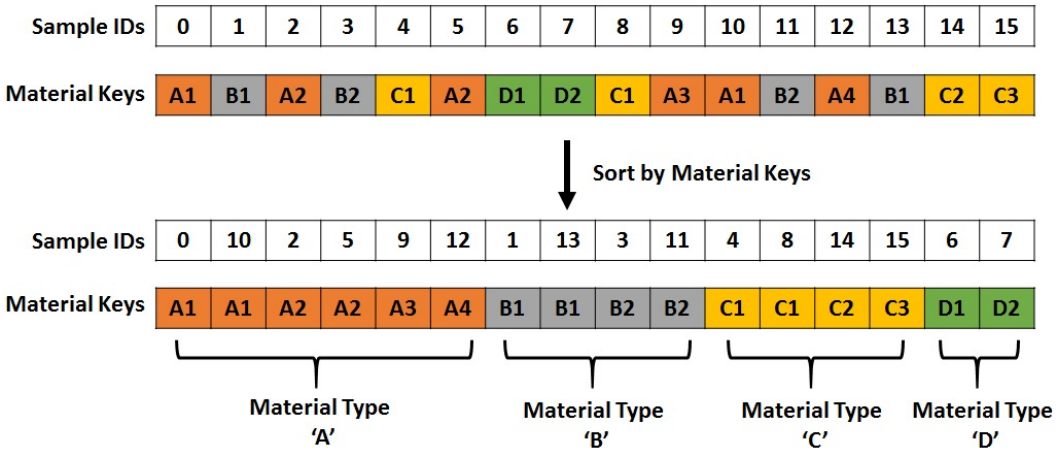


Figure 4.3: Sorting of samples based on material keys. Each 32 bit key is composed of the first 8 higher order bits representing the material type(represented by alphabets A,B,C and D) and the remaining bits indicating the id of a particular material applied to a primitive(represented by the numeral).

next bounce ray. This process was performed till all the camera samples were terminated. Since only the current set of active camera vertices were required to be stored in memory, the amount of memory required was very less compared to the strategies involving the entire camera and light subpath vertices. They presented a single kernel variant with regeneration (LVC-BDPTsk) and multikernel variant with streaming (LVC-BDPTmk).

4.4 Coherent and Importance-Sampled LVC BDPT

We propose two modifications to the LVC-BDPT scheme to improve quality and speed. The first involves material sorting to improve the coherence of material evaluations. The second involves sampling the LVC vertices more effectively to improve the quality of rendered images. We first describe our material evaluation.

4.4.1 Complex Materials

Materials traditionally used in production rendering can vary from simple single layered materials to extremely complex multi layered materials. We employ a layered material system as described by Pharr and Humphreys [36]. Materials in our system can have upto 4 independent layers. At each shading point, a material stack composed of the individual BSDFs is generated. The evaluation proceeds to determine which layer to sample for next bounce and computes the pdf associated with sampling the next bounce direction. We support a variety of BSDFs such as Lambertian, specular reflection, specular transmission, Oren-Nayar, and microfacet models with a variety of distribution function such as Blinn-Phong, Beckmann, GGX and various geometric functions.

4.4.2 Material Sorting

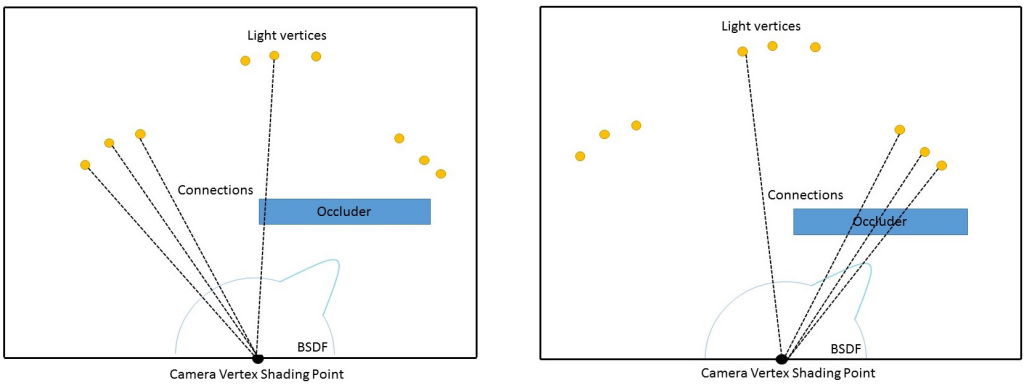
We employ a layered material system [36] to create materials of varied complexity. Materials in our system can have upto four independent layers. At each shading point, a material stack composed of the individual BSDFs is generated. We support a variety of BSDFs such as lambertian, specular reflection, specular transmission, Oren-Nayar, and microfacet models with a variety of distribution and geometric functions. Efficiency on the GPUs critically depends on coherent instruction execution and memory access. In a scene containing complex materials, rays in later bounces will hit different materials. This can result in thread divergence and degraded performance. This is more acute on layered materials with different types of BSDFs.

Laine et al. enhanced material coherence for path-tracing by placing each material-intersection request into a per-material queues [27]. We bring coherence to BDPT using material sorting in place of the use of queues. Queues have several disadvantages in a GPU setting. Queue management typically involves irregular computations that are not efficient on the GPUs. Queues also require atomic operations and can waste memory as space needs to be preallocated to each entry based on the potential maximum length. Queues may additionally be overrun if their sizes are not estimated properly and are in general underutilized. Queue buffer overrun presents a problem in a bidirectional path tracing scenario where the number of rays are typically much larger than path tracing.

We use in-place sorting of the intersection points based on material ids during each step of the algorithm to enforce execution as well as data coherence. Every primitive in the scene is assigned a unique 32 bit composite key composed of the first 8 higher order bits representing the material type with rest of the bits representing an unique material id. During traversal phase each thread handling a sample creates a tuple of the composite key and sample id in a global list. This list is sorted on composite key using an efficient Thrust sort primitive, before material evaluation is performed. Only the sample id list is sorted and not the actual ray data. Sorting has several advantages. The GPUs have fast radix sorting primitives. Sorting uses memory efficiently; space for only the total number of samples is needed in any step with no wastage. Sorting brings all samples of same material together (Figure 4.3). We then evaluate materials using a single kernel which operates with more execution coherence since threads in a warp now handle sorted material types. Our material evaluation kernel is kept lean currently. However should the need arise to handle extremely complex materials, we can add a separate kernel to handle only those materials within the current framework. We demonstrate the computational advantage brought by sorting on several situations in the results section.

4.4.3 Importance-Sampling of LVCs

In the LVC-BDPT, N LVC vertices are evaluated for connections for each camera vertex. In a scene with complicated geometry, several of the sampled LVC vertices may be shadowed, resulting in wasted visibility checks that could be expensive. Higher N values will give better quality images. The



(a) Scenario 1: Poor BSDF contribution at camera vertex for the light vertices chosen.

(b) Scenario 2: Visibility test fails for the chosen light vertices thereby adding no contribution to the final image.

Figure 4.4: Two scenarios indicating how a particularly bad sampling of light vertices might result in poor contribution.

connection term between a light path of length s and a camera path of length t can be given by equation 4.3.

Here, f_s terms are the BRDFs evaluated at the camera and light vertices for the connection direction, G the geometry term and V the visibility term. Depending upon the scene, estimation of these factors can be costly and a bad choice of LVC can result high variance. For example, a scene in which the light and camera subpath vertices both have high subpath contributions but have a failed visibility (Fig 4.4a) will result in the estimator being zero, while incurring an expensive shadow ray traversal. Even among visible LVC vertices, low product of BRDF values can also result in wasted computations (Fig 4.4b). The number N of samples will have to be increased to reduce the variance of the estimator in such scenarios.

We propose a different sampling scheme to reduce the number of visibility tests performed. Our method is similar to the sampling-importance-resampling method used to reduce visibility tests based on the product of the BRDF and the incoming radiance terms [5]. Our scheme starts with sampling a larger number M of vertices from the LVC. We then compute all the contribution of the selected vertices and create a distribution of these values. We then sample N vertices from this distribution are sent for visibility evaluation and image updation, if found visible. We thus evaluate materials first, and visibility later. On complex scenes, visibility tests can dominate the render time. Preferring more promising vertices for visibility and discarding less promising ones will enhance the quality of the rendered image. We compare our new connection method against the original LVC-BDPT method by comparing the RMSE values obtained by iterative rendering on different scenes. We show that our algorithm is able to provide lower RMSE values for the same number of samples per pixel traced. Our method also provides lower RMSE values for a given total rendering time, as can be seen in the results section.

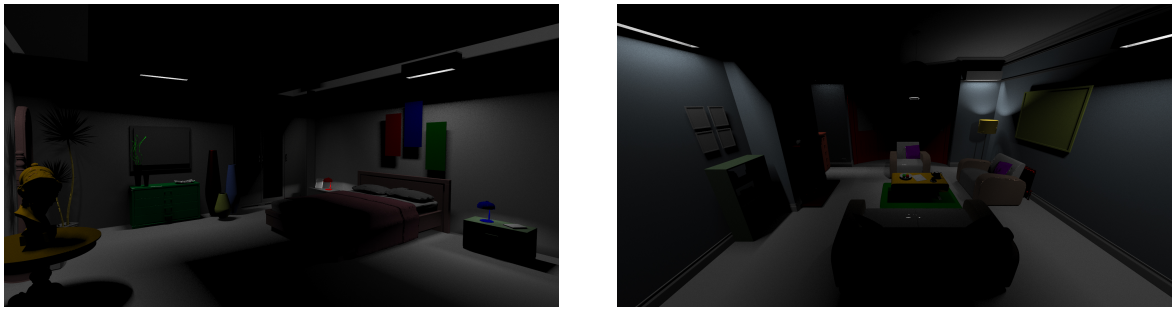
Algorithm 7 Light Pass

```
1: procedure LIGHT TRACE
2:     Create LightSamples in parallel
3:     while #LightSamples != 0 do
4:         Trace all rays in parallel
5:         Sort rays on intersected material id.
6:         Evaluate materials in parallel
7:         Store vertices in LVC in parallel
8:         Compute next bounce rays in parallel
9:         Compact rays to remove dead samples
10:    end while
11: end procedure
```

Algorithm 8 Camera Pass

```
1: procedure CAMERA TRACE
2:     Create CameraSamples in parallel
3:     while #CameraSamples != 0 do
4:         Trace all rays in parallel
5:         Sort rays on intersected material id.
6:         Stream0 - Evaluate direct lighting in parallel
7:         Stream1 - Evaluate modified connections in parallel
8:         Stream2 - Compute next bounce rays in parallel
9:         Trace connection shadow rays in parallel
10:        Compact rays to remove dead samples.
11:    end while
12: end procedure
```

The pseudocode for the light tracing and camera tracing schemes of our Coherent and Importance-Sampled LVC BDPT (CIS LBDPT) method is given in Algorithms 7 and 8. We employ a multikernel approach for different tasks such as direct lighting computation, ray traversal, material evaluation and modified connection evaluations (Line 6, 7, 8 in Algorithm 8). We employ the multi stream launch capabilities of the latest GPU hardware to launch all these kernels in parallel and collect their results. In our experiments, we use $M = 10$ (#samples generated by our scheme) and $N = 5$ (#samples used to update the image by our and LVC-BDPT schemes) unless otherwise specified. Given the available memory on the GPU, our implementation is able to generate and trace 4 samples per pixel simultaneously on 1980×1080 images in parallel (That is, it processes 8.5 million camera rays at a time.). With increase in GPU memory we can accomodate even more sample resolutions. An iteration of our algorithm involves tracing these samples to completion. Subsequent iterations will increase the number of samples per pixel but tracing another 8.5 million rays. This can be thought of as progressively adding 4 samples for each pixel in each iteration. Every subsequent iteration performs both the light pass followed by the camera pass from the beginning thereby effectively flushing the LVC and utilizing a new set of light vertices for connection.



(a) Bedroom scene

(b) Whiteroom scene

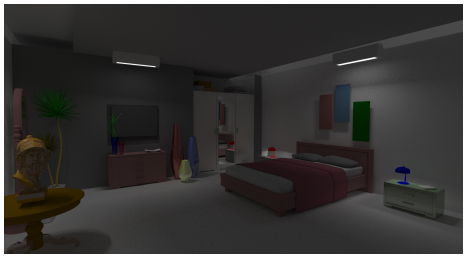
Figure 4.5: Direct lighting only visualization of two scenes used to test the performance of our algorithm. Both images were rendered at 128spp. Both images show that only very few parts of the scene are actually visible to the lights placed inside the fixtures with rest of the space clearly not illuminated. Bidirectional path tracing is able to render these scenes very effectively.

4.5 Results

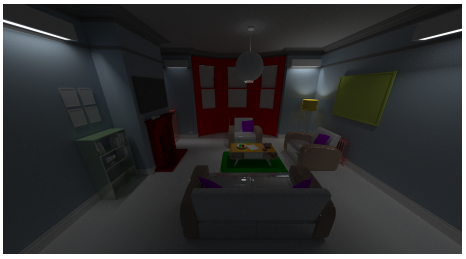
We have implemented all our algorithms on a machine containing a corei7-4790K processor and a Nvidia GTX Titan GPU having 6GB of video memory. We have used CUDA 7.0 and the Thrust libraries for sort and compaction. We have used primarily three scenes of varying geometric complexity and material complexity for testing the performance of our algorithm. Further all three scenes have difficult to render lighting scenarios with the lights inside deep fixtures (Figure 4.5). The first scene *Monkeybox* is a geometrically very simple scene that is similar to the *Cornell Box* scene. The scene contains three area lights which are primarily placed inside the fixtures hanging from the roof. The second scene *Whiteroom* is a fairly complicated scene with 6 area lights placed deep inside fixtures. The third scene *Bedroom* is a very complicated scene with a strong indirect lighting scenario that BDPT is able to capture very effectively. The ground truth results for comparing the RMSE values of images rendered by the algorithms were generated in a progressive manner by using a naive GPU-BDPT implementation till the images had very less visible noise in them.

4.5.1 Sorting for Coherence

We compare the runtime performance of our algorithm against LVC BDPTmk algorithm that does not employ sorting to handle all the materials scenes having varying material and geometric complexity. Table 1 shows the running times for different scenes as the number of samples per pixel increases. These numbers are a good indicator of the performance of the CIS-LBDPT algorithm in the presence of complex materials. Good speed up is achieved across the board. The benefits obtained by having coherent work chunks that can be handled by the GPU is very evident from the numbers themselves.



(a) Bedroom (1.8M tris)



(b) Whiteroom (574K tris)

(c) Monkeybox (196K tris)

Figure 4.6: Scenes of varying geometric complexity with complex lighting and multi-layered materials. All the scenes were modelled with lights inside fixtures. All the scenes were rendered using CIS-LBDPT. Images 4.6a and 4.6b were rendered at 1980x1080 resolution for 256 iterations with 4 spp per iteration. Image 4.6c was rendered at 1024x1024 resolution for 128 iterations using 8spp per iteration.

4.5.2 Performance Results

Table 4.2 shows the quality comparison of rendered image as the rendering progresses. The plot of RMS error against iteration number shows for all the three scenes show the quality advantage of our improved sampling scheme. The error is lower for images rendered using our scheme than using LVC-BDPT which does only random sampling. The plot of RMS error against run time shows clearly that for a fixed time budget, the CIS-LBDPT achieves better rendering quality across the board. The quality gains are likely to be even more if larger values of M and N are used.

4.5.3 Limitations

Our method employs a new sampling scheme of choosing LVC vertices during the camera pass. It is performed with the goal of decreasing the variance by choosing good samples as well as reducing the number of shadow ray computations which would be required otherwise for sampling a larger pool of vertices for attaining the same quality. Even though method shown here handles fairly complex materials, the cost of evaluating extremely complex materials can become very high thereby increasing the computation time. This can hence reduce the performance gained in sorting the vertices for improved material coherence.

	Effective SPP	LVC-BDPTmk (ms)	CIS-LBDPT (ms)	Speedup
MonkeyBox	8	18237	14228	1.28x
	16	39534	31376	1.26x
	24	61867	46169	1.34x
	32	84471	64214	1.31x
	40	104630	78082	1.34x
Whiteroom	4	34890	24780	1.48x
	8	68011	48988	1.38x
	12	105419	76138	1.38x
	16	145022	106276	1.36x
	20	182498	126046	1.44x
Bedroom	4	31595	20374	1.48x
	8	67509	46053	1.46x
	12	103618	70717	1.46x
	16	141286	94269	1.49x
	20	179323	122053	1.46x

Table 4.1: Comparison of execution times for LVC-BDPTmk and CIS-LBDPT methods. The Monkeybox scene was rendered at 1024x1024 resolution with 8spp per iteration. The Whiteroom and Bedroom scenes were rendered at 1980x1080 resolution with 4spp per iteration.

4.6 Conclusion

We presented a Coherent and Importance Sampled approach to perform LVC based BDPT efficiently on the GPU. Sorting provided coherence to material evaluation and an improved sampling method enhanced the rendering quality. Together, impressive performance has been shown at impressive quality levels on scenes of varying material and geometric complexity. Our work should inspire further improvements that will eventually place the GPU in the middle of a production rendering pipeline in the near future. For future work we would like to investigate out of core rendering of scenes with complex materials using our method.

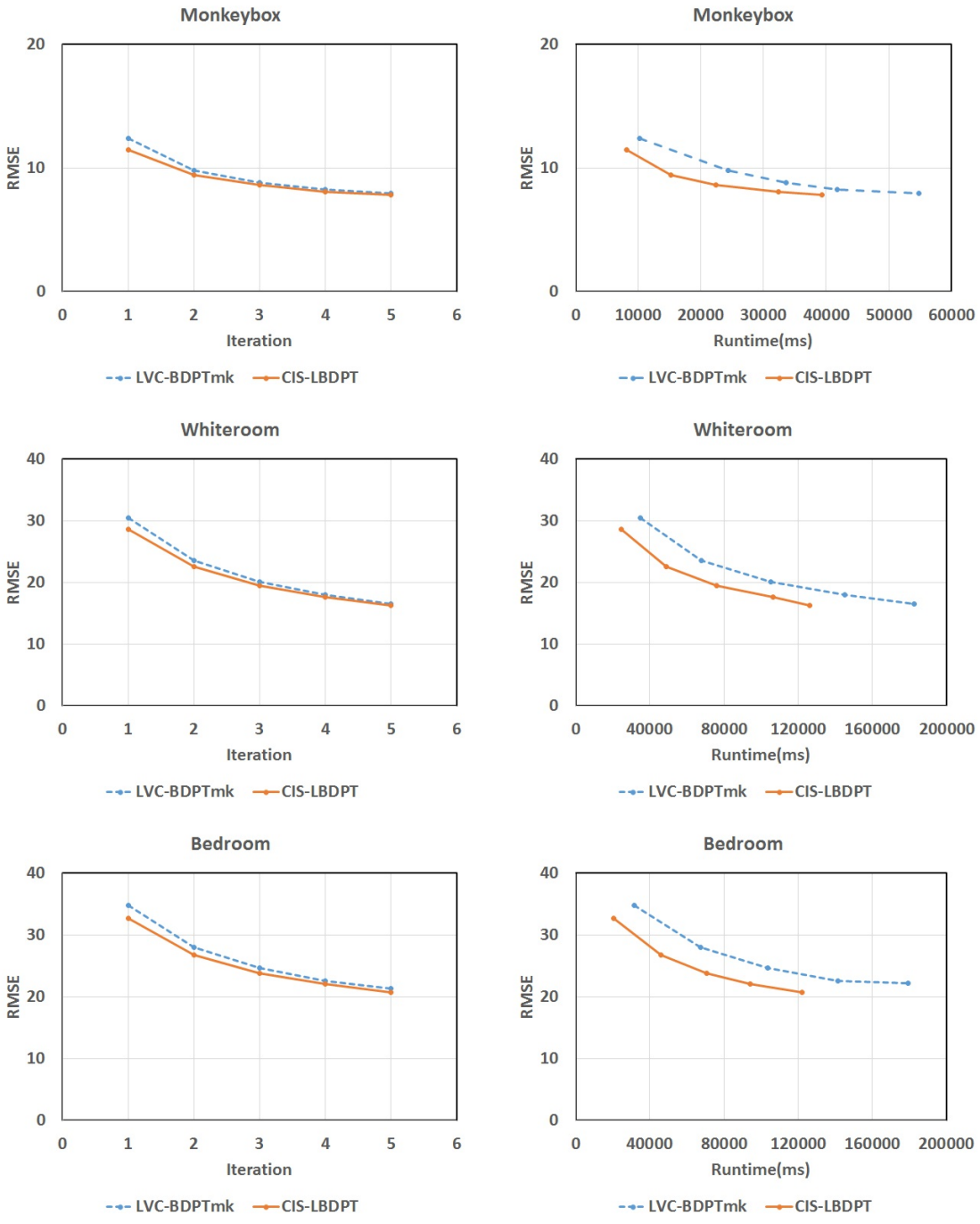


Table 4.2: Comparison of performance of LVC-BDPTmk and CIS-LBDPT methods for the three scenes. The graphs indicate how RMSE values reduce for images generated by both the methods as number of iterations and runtime of the algorithms increase.

Chapter 5

Conclusions and Future Work

We have presented in this thesis two works that are aimed at solving problems in different parts of the raytracing pipeline. Our first work presented a fundamentally new way of performing raytracing on the GPU without constructing any explicit acceleration structure. Our algorithm *Parallel Divide and Conquer Raytracing (PDACRT)* presented a fully parallel raytracing method on the GPU which traced and traversed an implicit hierarchy in tandem. Our algorithm is simple to implement and leverages upon well established parallel primitives on the GPU such as sort and reduce. Our results indicated that PDACRT required very less memory compared to acceleration structures constructed on the GPU. We also showed results where our algorithm performed well than the serial CPU DACRT algorithm.

Our second work presented a new algorithm *CIS-LBDPT* that was designed with the goal of improving the connection phase in the Light Vertex Cache (LVC) formulation of the Bidirectional Path Tracing (BDPT) algorithm. We presented an importance-sampling based of LVC vertices with the goal of reducing the number of shadow ray connections to get better quality images than the LVC-BDPT algorithm. Our work also addressed efficiency issues encountered while employing GPUs for BDPT in the presence of complex layered materials. We presented a sort based framework that enforced material execution coherence thereby increasing the runtime performance of the algorithm. Both our works are aimed at enabling GPUs to become mainstream for all the different phases of the raytracing pipeline at all levels.

Upcoming GPU architectures have more memory and more processing power in them and algorithms written for today's hardware can have some free lunch for sometime. However the best performance would result only when the algorithms are custom tuned for the newer architectures. Although GPUs have been designed with the goal of accelerating graphics applications, the major general purpose applications that have their performance accelerated due to GPUs are those that generally involve matrix or vector operations. General purpose applications that don't involve such operations can have only limited if not any acceleration by being run on the GPU. Even though raytracing and light transport algorithms expose a lot of parallelism on the outset, direct mapping of the operations to hardware is not that efficient due to the data access and execution patterns. Hence more research is required for improving the performance of raytracing algorithms on future hardware. Even though current generation GPUs

have more memory than earlier generations, it is not enough for production rendering where the data is mostly out-of-core. Hence out-of-core rendering algorithms for GPUs is also a very interesting research area.

Although current research trends are aimed at modifying or developing new algorithms for existing hardware, absolute true performance benefits could be attained with dedicated hardware for raytracing. As such the hardware should have dedicated units for different functional stages of the raytracing pipeline such as ray generation, ray traversal, shading, etc. The recent hardware architectures such as PowerVR show much promise and future parallel raytracing algorithms and research work should be more focussed on such dedicated hardware. Raytracing not only has applications in the graphics industry but also other very important industries which could benefit tremendously with these upcoming changes. Raytracing is considered as a mature research field by many computer graphics scientists. With these recent developments and upcoming generations of hardware, raytracing still presents much scope for exciting research for years to come.

Related Publications

- Parallel Divide and Conquer Ray Tracing - Siggraph Asia 2013 - Technical Briefs
- Coherent and Importance Sampled - LVC BDPT (CIS-LBDPT) - Submitted to Siggraph Asia 2015 - Technical Briefs (Submitted. Under Review)

Bibliography

[1] A. T. Áfra. Incoherent ray tracing without acceleration structures. In C. Andjar and E. Puppo, editors, *Eurographics (Short Papers)*, pages 97–100. Eurographics Association, 2012.

[2] A. T. Áfra and L. Szirmay-Kalos. Stackless multi-bvh traversal for CPU, MIC and GPU ray tracing. *Computer Graphics Forum*, 33(1):129–140, 2014.

[3] T. Aila and S. Laine. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of High-Performance Graphics 2009*, pages 145–149, 2009.

[4] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.

[5] D. Burke, A. Ghosh, and W. Heidrich. Bidirectional importance sampling for direct illumination. In *Proceedings of the Sixteenth Eurographics Conference on Rendering Techniques*, EGSR '05, pages 147–156, Aire-la-Ville, Switzerland, Switzerland, 2005. Eurographics Association.

[6] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart. Parallel SAH k-d tree construction. High Performance Graphics '10, pages 77–86, 2010.

[7] P. H. Christensen. Adjoints and importance in rendering: An overview. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):329–340, July 2003.

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

[9] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24, Jan. 1982.

[10] T. Davidovič, J. Křivánek, M. Hašan, and P. Slusallek. Progressive light transport simulation on the gpu: Survey and improvements. *ACM Transactions on Graphics (TOG)*, 33(3):29, 2014.

[11] P. Dutré, E. P. Lafortune, and Y. Willems. Monte Carlo Light Tracing with Direct Computation of Pixel Intensities. In *3rd International Conference on Computational Graphics and Visualisation Techniques*, pages 128–137, Dec. 1993.

[12] A. Fujimoto, T. Tanaka, and K. Iwata. Tutorial: computer graphics; image synthesis. chapter ARTS: accelerated ray-tracing system, pages 148–159. Computer Science Press, Inc., 1988.

[13] K. Garanzha, J. Pantaleoni, and D. McAllister. Simpler and faster HLBVH with work queues. High Performance Graphics '11, pages 59–64, New York, NY, USA, 2011. ACM.

[14] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. SIGMOD '06, pages 325–336. ACM, 2006.

[15] S. Guntury and P. J. Narayanan. Raytracing dynamic scenes on the gpu using grids. *IEEE Transactions on Visualization and Computer Graphics*, 18(1):5–16, 2012.

[16] M. Hapala, T. Davidovič, I. Wald, V. Havran, and P. Slusallek. Efficient stack-less bvh traversal for ray tracing. In *Proceedings of the 27th Spring Conference on Computer Graphics*, SCCG '11, pages 7–12, New York, NY, USA, 2013. ACM.

[17] J. Hoberock and N. Bell. Thrust: A parallel template library, 2010. Version 1.7.0.

[18] W. Jakob, E. d'Eon, O. Jakob, and S. Marschner. A comprehensive framework for rendering layered materials. *ACM Trans. Graph.*, 33(4):118:1–118:14, July 2014.

[19] H. W. Jensen. Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pages 21–30, London, UK, UK, 1996. Springer-Verlag.

[20] J. T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[21] J. Kalojanov, M. Billeter, and P. Slusallek. Two-Level Grids for Ray Tracing on GPUs. In O. D. Min Chen, editor, *EG 2011 - Full Papers*, pages 307–314, Llandudno, UK, 2011. Eurographics Association.

[22] J. Kalojanov and P. Slusallek. A parallel algorithm for construction of uniform grids. High Performance Graphics '09, pages 23–28, New York, NY, USA, 2009. ACM.

[23] M. H. Kalos and P. A. Whitlock. *Monte Carlo Methods. Vol. 1: Basics*. Wiley-Interscience, New York, NY, USA, 1986.

[24] T. Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-High Performance Graphics'12, pages 33–37, Aire-la-Ville, Switzerland, Switzerland, 2012. Eurographics Association.

[25] T. Karras and T. Aila. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, High Performance Graphics '13, pages 89–99, New York, NY, USA, 2013. ACM.

[26] E. P. Lafortune and Y. D. Willems. Bi-directional path tracing. In *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (COMPUGRAPHICS 93*, pages 145–153, 1993.

[27] S. Laine, T. Karras, and T. Aila. Megakernels considered harmful: Wavefront path tracing on gpus. In *Proceedings of the 5th High-Performance Graphics Conference*, High Performance Graphics '13, pages 137–143, New York, NY, USA, 2013. ACM.

[28] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast bvh construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[29] T. Möller and B. Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21–28, Oct. 1997.

[30] B. Mora. Naive ray-tracing: A divide-and-conquer approach. *ACM Trans. Graph.*, 30(5):117:1–117:12, Oct. 2011.

[31] K. Nabata, K. Iwasaki, Y. Dobashi, and T. Nishita. Efficient divide-and-conquer ray tracing using ray sampling. In *Proceedings of the 5th High-Performance Graphics Conference*, High Performance Graphics '13, pages 129–135, New York, NY, USA, 2013. ACM.

[32] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar. 2008.

[33] M. Oren and S. K. Nayar. Generalization of lambert's reflectance model. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 239–246, New York, NY, USA, 1994. ACM.

[34] A. Pajot, L. Barthe, M. Paulin, and P. Poulin. Combinatorial bidirectional path-tracing for efficient hybrid CPU/GPU rendering. *Computer Graphics Forum*, 30(2):315–324, 2011.

[35] J. Pantaleoni and D. Luebke. HLBVH: hierarchical lbvh construction for real-time ray tracing of dynamic geometry. High Performance Graphics '10, pages 87–95. Eurographics Association, 2010.

[36] M. Pharr and G. Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010.

[37] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan. Ray tracing on programmable graphics hardware. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pages 703–712, New York, NY, USA, 2002. ACM.

[38] S. M. Rubin and T. Whitted. A 3-dimensional representation for fast rendering of complex scenes. SIGGRAPH '80, pages 110–116. ACM, 1980.

[39] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore gpus. IPDPS '09, pages 1–10. IEEE Computer Society, 2009.

[40] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for GPU computing. Graphics Hardware '07, pages 97–106. Eurographics Association, 2007.

[41] M. Stich, H. Friedrich, and A. Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, High Performance Graphics '09, pages 7–13, New York, NY, USA, 2009. ACM.

[42] J. E. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *IEEE Des. Test*, 12(3):66–73, May 2010.

[43] D. van Antwerpen. Improving simd efficiency for parallel monte carlo light transport on the gpu. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, High Performance Graphics '11, pages 41–50, New York, NY, USA, 2011. ACM.

[44] E. Veach. *Robust Monte Carlo Methods for Light Transport Simulation*. PhD thesis, Stanford, CA, USA, 1998. AAI9837162.

[45] A. Weidlich and A. Wilkie. Thinking in layers: Modeling with layered materials. In *SIGGRAPH Asia 2011 Courses*, SA '11, pages 20:1–20:43, New York, NY, USA, 2011. ACM.

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

[47] Z. Wu, F. Zhao, and X. Liu. Sah kd-tree construction on GPU. High Performance Graphics '11, pages 71–78, New York, NY, USA, 2011. ACM.

[48] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):126:1–126:11, Dec. 2008.