Applying Parallel Processing Approach for Interactive Global Illumination

Mahmoud Zeidan

Computer Science Department
Faculty of Computer and Information Sciences

Ain Shams University

A thesis submitted for the degree of

Master of Science (M. Sc.)

2011 July

Abstract

One of the ultimate goals of computer graphics is to produce realism in simulated scenes. Global illumination algorithms are based on efficiently simulating the complicated physical details of light transport inside $3D$ scenes. But such techniques still to some degree miss the interactive and real-time rendering rate due to the complexity of its core algorithms. At the core of most global illumination algorithms is the ray tracing algorithm. However, it was noticed that the main bottleneck of any ray tracing algorithm is ray traversal. So in order to enhance the rendering time we have to target the ray tracing and its core primitives including; hierarchy construction, ray traversal, and shading calculation.

With the new parallel architecture offered by modern graphics hardware we can reformulate global illumination algorithms into efficient and scalable parallel implementations on graphics processing unit (GPU). In this thesis, we develop a parallel Whitted style ray tracer on GPU, and map all the stages of the photon mapping algorithm onto GPU employing recent and efficient parallel algorithm for construing a point based KD-tree.

We introduce a new parallel algorithm for building binned SAH BVH and evaluate the resulting tree hierarchy using parallel ray tracing techniques. We also compare our BVH construction algorithm with recent state of the art KD-tree and BVH construction algorithms on GPU. Our new algorithm for binned SAH BVH construction gives better or comparable rendering performance to recent state of the art hierarchical tree construction algorithms.

Finally, in this thesis we show that most recent parallel tree construction algorithms can be reduced to a small and general set of parallel primitives and can be easily explained with a small set of parallel utilities.

Contents

List of Figures **v**

List of Tables **ix**

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 2

1.3 Thesis Organization 3

2 Introduction to Ray Tracing and Global Illumination **5**

2.1 Ray Tracing 5

2.1.1 Classic Ray Tracing -*Whitted Style Ray Tracing*- 6

2.1.2 Distributed Ray Tracing -*a Ray Tracing Extension*- 7

2.2 Global Illumination 8

2.2.1 The Rendering Equation 9

2.3 Previous Work in Rendering Techniques 11

2.3.1 Rasterization 11

2.3.2 Physically Based Rendering 12

2.3.3 Reyes Rendering Architecture 13

2.3.4 Precomputed Radiance Transfer 13

3 Introduction to GPU Parallel Computing **15**

3.1 GPU and Parallel Computing 15

3.2 The CUDA Programming Model 16

3.2.1 Host and Device 16

3.2.2 Thread Hierarchy 16

3.2.3 Memory Hierarchy 16
3.2.4 SIMD/SIMT Execution 18
3.3 Data Parallel Primitive Algorithms on GPU 19
3.3.1 Parallel Reduction and Segmented Reduction 20
3.3.2 Parallel Scan and Segmented Scan 20
3.3.3 List Compaction 22
3.3.4 List Split and Segmented List Split 22

4 Parallel Hierarchical Tree Construction Algorithms on GPU 25
4.1 Motivation and Previous Work 25
4.1.1 Spatial Partitioning Data Structures 25
4.1.2 Parallel Tree Construction on GPU 26
4.1.3 Data Parallel Primitive Algorithms on GPU 28
4.2 Data Parallel Operator and Data Parallel Utilities 28
4.2.1 Data Parallel Operator 28
4.2.2 Frequently Used Data Parallel Utilities 29
4.2.3 Extensions to Data Parallel Operators and Utilities 30
4.3 BFS Tree Construction Algorithms on GPU 31
4.4 Parallel SAH KD-tree Construction Algorithm 32
4.4.1 Large Node Stage 33
4.4.2 Small Node Stage 39
4.5 Parallel SAH BVH Construction 43
4.5.1 Large Node Stage 44
4.5.2 Small Node Stage 49
4.6 Proposed Parallel Algorithm for Building Binned SAH BVH 50
4.6.1 Filtering Next/Further Nodes 56
4.6.2 Modifications and Extensions 57
4.6.2.1 Reducing Scan Passes for Triangles Projection. 57
4.6.2.2 Projecting Triangles using Parallel Sorting. 59
4.7 Linear Bounding Volume Hierarchy (LBVH) 60
4.7.1 Hybrid binned SAH BVH Algorithm 63
4.8 Analysis and Discussion 63
4.9 Evalutions and Comparisons of Proposed Proposed Tree Construction Algorithms 67

5 Ray Tracing on GPU 75

5.1 Parallel Ray Tracing on GPU 75

5.1.1 Parallel Rays Generation and Bouncing 77

5.1.2 Building Shade Tree 80

5.1.3 Accumulating Shading and Rendering 84

5.2 Results and Dicussion 87

5.2.1 Ray Tracing Performance 87

6 Photon Mapping on GPU 91

6.1 Introduction to Offline Photon Mapping 91

6.1.1 The First Pass -Building Photon Map(s)- 91

6.1.1.1 Photon Emission 91

6.1.1.2 Photon Tracing and Storing 93

6.1.2 The Second Pass - Rendering- 93

6.1.2.1 Radiance Estimate 94

6.1.2.2 Final Gather for Indirect Lighting 94

6.1.2.3 Irradiance Caching 94

6.2 Parallel Photon Mapping on GPU 95

6.2.1 Parallel Photon Tracing 95

6.2.2 Building Photons KD-Tree on GPU 99

6.2.2.1 Initialization Stage 101

6.2.2.2 Large Node Stage 103

6.2.2.3 Small Node Stage 109

6.2.3 Irradiance Estimation 112

6.2.3.1 Selecting Irradiance Samples 112

6.2.4 Final Gather on GPU 116

6.2.5 Rendering on GPU *-Putting All Together-* 116

6.3 Results and Dicussion 116

6.3.1 Photon Mapping Performance 116

7 Conclusion and Future Work 119

7.1 Conclustion 119

7.2 Future Work 120

A Frequently Used Utilities on GPU **121**

B Parallel Chunking Primitive and Utility **129**

Bibliography **133**

List of Figures

2.1 Ray Tracing Concept. 6

2.2 Whitted scene indicating light interaction with reflective and refractive materials, this image first appeared in [Whitted, 1980], image from [Parker, Bigler, Dietrich, Friedrich, Hoberock, Luebke, McAllister, McGuire, Morley, Robison, and Stich, 2010]. 8

2.3 Examples of visual effects generated using distributed ray tracing, from left to right motion blur, soft shadows, and depth of field. Images from [Boulos, Edwards, Lacewell, Kniss, Kautz, Shirley, and Wald, 2007]. 8

2.4 Image on the left is rendered with local illumination and image on the right is rendered with global illumination. In the right image notice natural appearance due to the indirect lighting effect on the ceiling around the light source and the color bleeding of the red and blue walls on the white ceiling and the caustics of the sphere ball on the floor. The two images from [Jensen, 2001]. 9

2.5 Example of such effects generated using global illumination algorithms a, b from [Zhou, Hou, Wang, and Guo, 2008], and c, d from [Jensen, 2001]. 10

2.6 Rendering equation relate outgoing lighting by integrating incoming lighting form all direction over upper hemisphere. 10

3.1 GPU internal organization and parallel code execution. 17

3.2 Segmented reduction example using sum and maximum operations. 20

3.3 Scan and segmented using the sum $(+)$ operator. 21

3.4 List compaction example. 22

3.5 List split example. 22

3.6 Segmented List split example. 24

4.1 Active nodes and associated triangles indices. 32
4.2 Dividing node's triangles into fixed sized chunks of at most 4 triangles. 33
4.3 Empty space calculation. 34
4.4 Large nodes splitting. 35
4.5 Sorting KD-tree triangles to child nodes using two flags arrays. 36
4.6 Large/Small nodes filtering, in this example we assume that the node thrshold T equals 4. 37
4.7 Clipping and distributing triangles to child nodes. 38
4.8 Small roots and related splits. 41
4.9 Small nodes modified structure, small node size is at most 4 primitives. 41
4.10 Small nodes splitting. 42
4.11 Splitting BVH nodes. 45
4.12 Sorting BVH triangles to child nodes. 46
4.13 Filtering large/Small nodes. 47
4.14 Distributing triangles to (child) nodes. 48
4.15 Projection triangles into corresponing bins. 51
4.16 Bins SoA calculation. 51
4.17 Sorting triangles locally to their respective bins. 52
4.18 Splitting BVH Node. 54
4.19 Distributing triangles to (child) nodes 56
4.20 Filtering Next/Further nodes, in this example we use node Next/Further threshold equals 2. 57
4.21 Sorting triangles into 4 bins using 2 sequential scans 58
4.22 Sorting triangles into 4 bins using radix sort 59
4.23 Morton codes generation. 60
4.24 LBVH heirarchy emission. 61
4.25 Sorted splits bounds and correspoinding nodes bound in the hierarchy. 62
4.26 LBVH pruning. 63
4.27 Benchmark scenes used to evaluate the construction and rendering performance of our construction algorithms. 68
4.28 Rays assignment to CUDA threads, in each cell at checkerboard numbers at the top left corner defines the sequential pixel order, and number in bottom right corner defined the z-order. 70

5.1 Shading trees on Cornel box scene: (Right) various forms of shade trees corresponding to different screen positions (left). 81
5.2 Shade tree node structure 83
5.3 Shade tree post order traversal 87
5.4 Our test scenes rendered on a 1024 X 1024 window using a GTX 285 device . 89

6.1 Main building blocks of the photon mapping algorithm 92
6.2 Main building blocks of GPU photon mapping algorithm 96
6.3 Photon positions and photons array 101
6.4 Initializing the association list 102
6.5 Empty space calculation 104
6.6 Large Nodes Split 105
6.7 Photons sorting to child nodes 106
6.8 Large/Small Nodes Splitting 106
6.9 Photons relocation 108
6.10 Small roots creation 110
6.11 Small nodes modified structure 110
6.12 Small nodes split 111
6.13 Our test scenes rendered using photon mapping at 2.5, 1.8, 2 fps on a 512 X 512 window using a GTX 285 117

A.1 Scan tail extraction for both inclusive and exclusive scan 122
A.2 Scan tails extraction for a segmented scan 124
A.3 Key value sorting example 126
A.4 Extracting sorted values bounds 126

B.1 Chunks creation using segmented scan, for illustration purpose we set the chunk size T equals 4. 130
B.2 Chunks creation using standard scan, chunk size T equals 4. 131

List of Tables

LIST OF TABLES

1

Introduction

1.1 Motivation

Computer graphics is the process of creating images by simulating light interaction with its surrounding environment. One of the ultimate goals of computer graphics is to create realistic-looking images; to reach such goal computer graphics researchers try to explore algorithms that accurately simulate the complicated physical nature of light. Over decades researchers have succeeded in achieving such goal and presented to the community a plethora of algorithms to synthesis photo-realistic images that can be used in movies, virtual reality, cinematic effects, scientific visualizations, and other applications.

The term global illumination refers to the large class of algorithms that accurately or approximately simulate the physics of light and render photo-realistic images, examples of such algorithms include but not limited to ray tracing [Whitted, 1980], path tracing [Kajiya, 1986], radiosity [Goral, Torrance, Greenberg, and Battaile, 1984], bidirectional path tracing [Veach, 1997], and photon mapping [Jensen, 2001]. Most global illumination algorithms first appeared as offline solutions and the real-time and interactive performance was governed to a large degree by the hardware progress. In the past few years hardware companies offered powerful machines that increased the computation power in both the vertical and horizontal directions [1] allowing computer graphics researchers to develop complex and parallel global illumination

[1]We mean by the vertical growth the increase in a single processing core power, and the horizontal growth the increase in the number of processing cores.

algorithms and reach the main goal of photorealism in a faster way (see e.g. [Wald, 2004]).

Most global illumination algorithms use the basic ray tracing techniques in the core rendering process. Ray tracing in its abstract form can be viewed as an intersection query followed by shading evaluation and it has been noticed that most time consumed in ray tracing is due to intersection query [Whitted, 1980]. This is because every ray searches in all scene primitives for the nearest intersection. To accelerate ray-primitive query; ray tracers always make use of indexing data structures which organize $3D$ scene objects in a structured form suitable for fast ray-primitives search. Indexing data structures are also referred in literature as spatial partitioning data structures and the most widely used indexing structures are grids, KD-trees, bounding volume hierarchies (BVHs), and Octrees. A vast amount of research spent in exploring the best spatial partitioning technique for ray tracing on both CPUs and multi-core CPUs and is has been noticed that each technique offer the best way form a different perspective and in some cases solutions are scene dependent or rely on certain assumptions of input scene (see e.g. [Garanzha, 2009]), but generally the major issue in all spatial partitioning techniques is the processing time including construction time and traversal time. Recently SIMD wide machines like GPUs offered a new parallel architecture which allowed fast parallel construction of spatial partitioning data structures including grids [Kalojanov and Slusallek, 2009], BVHs [Lauterbach, Garland, Sengupta, Luebke, and Manocha, 2009] , and KD-trees [Zhou et al., 2008] and also allowed an interactive frame rate of the ray tracing [Zhou et al., 2008] and photon mapping [Wang, Wang, Zhou, Pan, and Bao, 2009; Zhou et al., 2008]. But the GPU architecture to some degree remains unexplored (see [Tzeng, Patney, and Owens, 2010] and [Aila and Laine, 2009]) and compared to CPU; the GPU toolbox for global illumination algorithms is far from being complete.

In this thesis we try to fill in the gap at the interactive and real-time rendering techniques based on ray tracing and try to reach the goal of interactive global illumination by developing parallel algorithms for ray tracing and photon mapping on modern GPU. We also explore new parallel algorithms for constructing spatial partitioning data structures on GPU.

1.2 Contributions

In this thesis we present several contributions for interactive global illumination algorithms; we present new parallel algorithms for constructing binned surface area heuristics (SAH) bounding volume hierarchy (BVH) and compare our proposed algorithms to the up-to-date parallel algorithms for constructing both BVHs and KD-trees, our analysis of hierarchy construction allowed us to extend the data parallel primitives on GPU to include primitive operations for constructing hierarchal trees. We also propose a simplified API to write and explain parallel programs on GPU. The proposed API includes a new operator to express parallel code fragments and data parallel utilities which are used as wrappers for data parallel primitives on GPU. Then we employ the proposed API to reformulate the most recent parallel algorithms for hierarchal tree construction. Finally, we develop a complete parallel global illumination solution using both Whitted style ray tracing and photon mapping on GPU.

1.3 Thesis Organization

This thesis is organized as follow:

Chapter 1 presents a brief introduction about the interactive global illumination problem and a general outline of the thesis.

In Chapter 2 we explain the basic ray tracing algorithm and its extensions followed by an introduction to global illumination algorithms and the analytical form behind them, and introduce a taxonomy of the most dominant techniques for photo-realistic rendering.

Chapter 3 gives a brief review about GPU parallel architecture and CUDA runtime and a review the frequently used data parallel primitives on GPU.

Chapter 4 introduces the main contributions of this thesis. We begin by introducing our proposed parallel API which includes a data parallel operator and data parallel utilities. Then we use the proposed API to describe the up to parallel hierarchical tree construction algorithms on GPU for both BVH [Lauterbach et al., 2009] and KD-tree [Zhou et al., 2008], and present our new parallel binned SAH BVH construction algorithm followed by a performance comparison between all these algorithms.

In Chapter 5 we present a parallel implementation for a Whitted style ray tracer on GPU [Zhou et al., 2008] employing indexing structures presented in Chapter 4. We explain in details how to build rays tree on GPU using our data parallel notation and present parallel ways for shading evaluation.

In Chapter 6 we build a complete global illumination solution on GPU using photon mapping: firstly, we explain briefly the original photon mapping algorithm; secondly, we explain in details the mapping of the entire photon mapping algorithm on the GPU [Wang et al., 2009; Zhou et al., 2008] using parallel algorithms for ray tracing, photon tracing, irradiance estimation, final gathering, and shading evaluation. In this chapter, we also explain a parallel algorithm for building a point based KD-tree on GPU [Zhou et al., 2008] using our proposed API.

In Chapter 7 we present final concluding remarks and directions for future work.

2

Introduction to Ray Tracing and Global Illumination

2.1 Ray Tracing

Using computers to produce images is called image synthesis. Realistic image synthesis is increasingly important in areas such as entertainment (movies, special effects, and games), design, architecture, scientific visualizations and many more. A common trend in all these areas is to request more realistic images of increasingly complex models within a small amount of time.

Ray tracing [Whitted, 1980] is powerful rendering technique that has been used for image synthesis and gained much interest in the past decades. The basic idea of ray tracing is to follow light rays form light source and propagate them throughout the scene until they finish at the viewer (see Figure 2.1). Ray tracing was proposed to accurately simulate the physical light transport inside its surrounding environment.

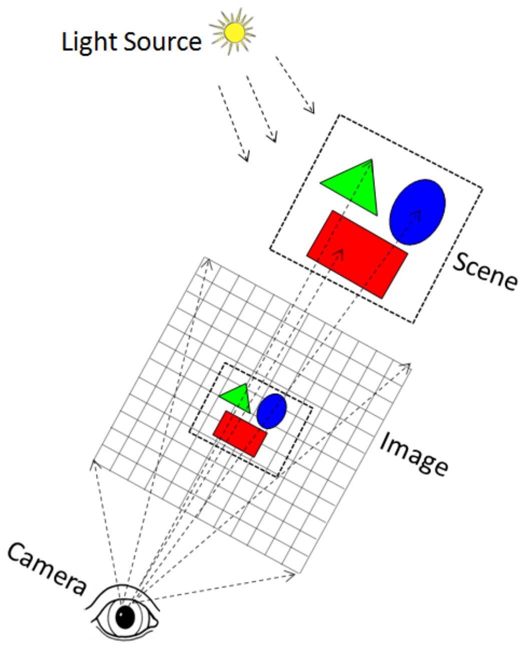


Figure 2.1: Ray Tracing Concept.

2.1.1 Classic Ray Tracing -*Whitted Style Ray Tracing*-

A naïve ray tracing algorithm [Whitted, 1980] begins by casting (primary) rays form each pixel toward a virtual viewing plane and calculates the shading color at each pixel based on the interaction between the primary ray, the surface materials, and light sources. Primary rays are created using a virtual camera that can be oriented in the virtual $3D$ space in a way similar to orienting a physical camera on a tripod. After shooting primary (visibility) rays we seek for the nearest intersection inside the scene. The brute force approach tests every ray against each primitive [1] in the scene and returns the nearest hit point. Several approaches have been used to decrease the traversal cost and avoid testing each primitive. The main idea it to partition the $3D$ scene space into sub regions [Havran, 2000] using spatial partitioning data structures and restrict the ray traversal to a small subspace and perform ray-primitive intersection with a small subset of scene primitives. Most commonly used spatial partitioning data structures include; uniform grids [Wald, Ize, Kensler, Knoll, and Parker, 2006], adaptive grids [Klimaszewski

[1]We mean by a primitive any geometric object that has a parametric representation like a sphere, a line, a cube, a triangle and even a parametric surface such as a NURBS and a Bezier curve.

and Sederberg, 1997], hierarchal grids [Ize, Shirley, and Parker, 2007; Reinhard, Smits, and Hansen, 2000], bounding volume hierarchies (BVHs) [Wald, 2007], Octrees, and KD-Trees [Wald and Havran, 2006].

Once the primary ray finds a hit we calculate the shading color using information about the incoming ray, the surface material at the hit point, and the light sources. First, we check the surface material; if it is a diffuse material then we trace a shadow ray originating from the intersection point toward each light source to check the light source visibility, where we mean by the visibility a boolean function indicating that light source is directly seen from the intersection point and no other object blockades the shadow ray. If the light source is directly visible; we calculate the shading color using a suitable shading model (e.g. Phong or Torrance-Spraw shading). On the other hand, if the surface material is reflective, or refractive we recursively trace secondary ray(s) in the reflection and/or the refraction direction and accumulate the returned shading color to the pixel color. We continue the recursive ray traversal until we end with a diffuse surface or reach a maximum traversal depth (e.g. 5).

As shown in Figure 2.2 we can use the ray tracing algorithm to simulate several visual effects such as shadows, reflection and refraction. We refer interested readers to the book [Glassner, 1989] which presents a classical introduction to ray tracing and the excellent books [Morley and Shirley, 2003; Pharr and Humphreys, 2010] which present more information about the theory behind ray tracing and the state-of-the-art rendering techniques, and the book [Akenine-Möller, Haines, and Hoffman, 2008] which is considered an excellent reference [1] for primitives intersection algorithms.

2.1.2 Distributed Ray Tracing *-a Ray Tracing Extension-*

Although classic ray tracing algorithm allows us to simulate various visual effects such as shadows, reflection and refraction; it still has limited capabilities for simulating these effects accurately. For example, classic ray tracing is limited to sharp shadows, perfect (100%) reflective and refractive materials and may produce aliasing artifacts. In [Cook, Porter, and Carpenter, 1984] Cook et al. introduced the idea of distributed ray tracing by sending more than one primary or secondary ray through each pixel using stochastic methods and integrating the returned

[1]The book's homepage `http://www.realtimerendering.com/` contains a detailed listing for most intersection algorithms and related source code on the web, and the book's blog is a rich source for up-to-date information about the computer graphics field.

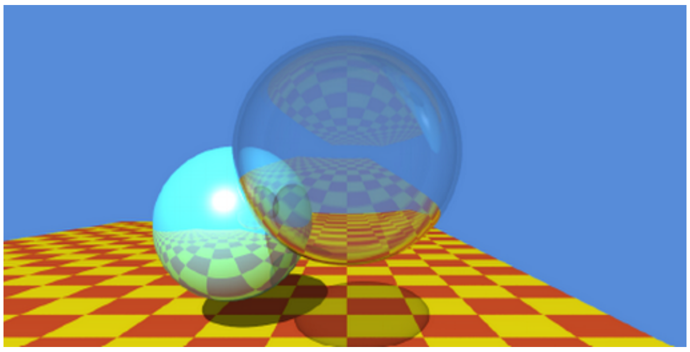


Figure 2.2: Whitted scene indicating light interaction with reflective and refractive materials, this image first appeared in [Whitted, 1980], image from [Parker et al., 2010].

results. With distributed ray tracing we can remove the aliasing artifacts generated by the naïve ray tracing algorithm and add several visual effects such as motion blur, depth of field, soft shadows, and glossy reflections (see Figure 2.3).

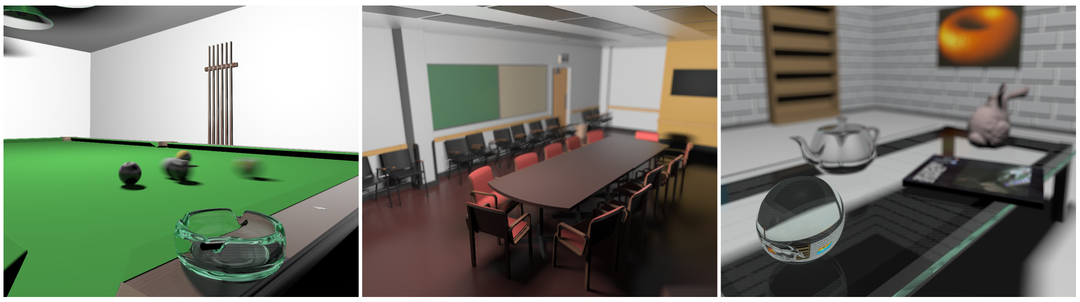
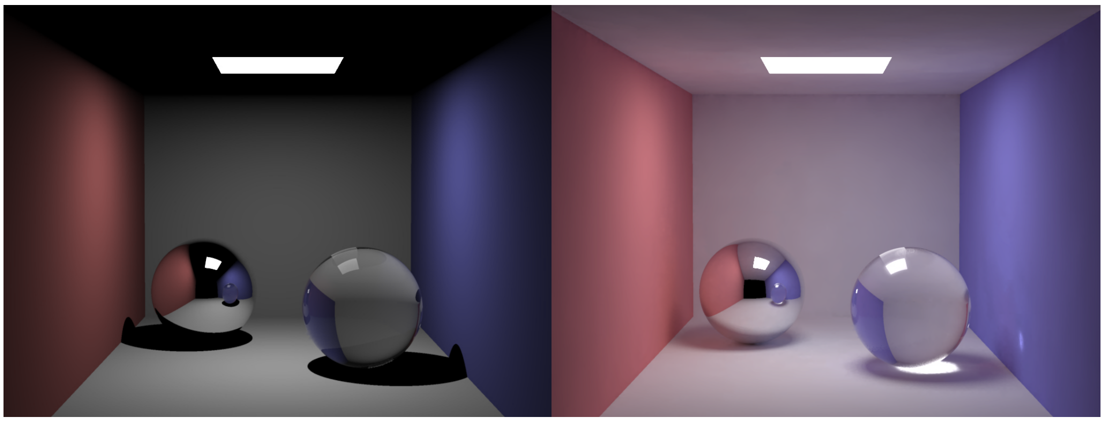


Figure 2.3: Examples of visual effects generated using distributed ray tracing, from left to right motion blur, soft shadows, and depth of field. Images from [Boulos et al., 2007].

2.2 Global Illumination

The term global illumination (GI) refers to the class of algorithms that accurately simulate the physical light transport in $3D$ scenes. A global illumination algorithm considers both the light that comes directly from the light source (local illumination), and the indirect light that is reflected and/or transmitted from other surfaces in the scene (global illumination) (see Figure 2.4).

Global illumination algorithms came to existence to efficiently simulate all natural light effects including those generated by ray tracing and to overcome the limitations of ray tracing



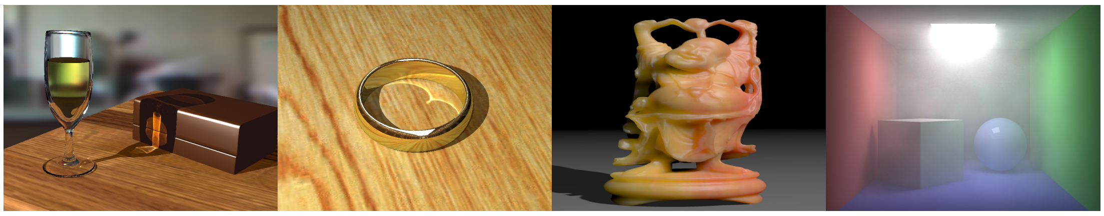
(a) Cornell box illuminated by local illumination **(b)** Cornell box illuminated by global illumination

Figure 2.4: Image on the left is rendered with local illumination and image on the right is rendered with global illumination. In the right image notice natural appearance due to the indirect lighting effect on the ceiling around the light source and the color bleeding of the red and blue walls on the white ceiling and the caustics of the sphere ball on the floor. The two images from [Jensen, 2001].

algorithms. Examples of such algorithms include path tracing [Kajiya, 1986], radiosity [Goral et al., 1984], bidirectional path tracing [Veach, 1997], and photon mapping [Jensen, 2001]. In many rending frameworks global illumination algorithms require integration with ray tracing algorithms. However, effects generated by global illumination algorithms may be hard or even impossible to be simulated by a naïve ray tracer or its extensions. Examples of such effects that can be efficiently simulated using global illumination algorithms include: color bleeding generated by diffuse surface inter-reflection; caustics produced by concentrated light rays refraction and reflection through specular materials; subsurface scattering through translucent materials like the skins and marbles; and volumetric rendering through participating media as in fog and smoke (see Figure 2.5).

2.2.1 The Rendering Equation

In his seminal paper [Kajiya, 1986] Kajiya introduced the rendering equation; a unified theory for all global illumination algorithms including ray tracing. The rendering equation is an integral equation that relate the reflected light at position x_o in a specific direction to the incoming light from all directions. It involves the integration of incident radiance over a hemispherical surface centered at each surface location.



(a) Caustics effects through refraction **(b)** Caustics effects through reflection **(c)** Subsurface scattering through marble **(d)** Participating media effect

Figure 2.5: Example of such effects generated using global illumination algorithms a, b from [Zhou et al., 2008], and c, d from [Jensen, 2001].

The rendering equation is defined using the formula:

$$L_r(\vec{x}, \vec{w_r}) = L_e(\vec{x}, \vec{w_r}) + \int_\Omega L_i(\vec{x}, \vec{w_i}) f_r(x, \vec{w_i} \leftarrow \vec{w_r}) \cos\theta_i d\omega_i$$

The reflected radiance L_r (see Figure 2.6) is computed by integrating the incoming radiance L_i over a hemisphere centered at certain surface point and oriented such that its north pole is aligned with the surface normal. The term fr represents the Bidirectional Reflectance Distribution Function (BRDF); a probability distribution function that describes the probability that an incoming ray of light is scattered in a specific outgoing direction. The term L_e refers to emitted light and is used for energy conservation and the cosine term is used to account for surface orientation relative to incoming direction.

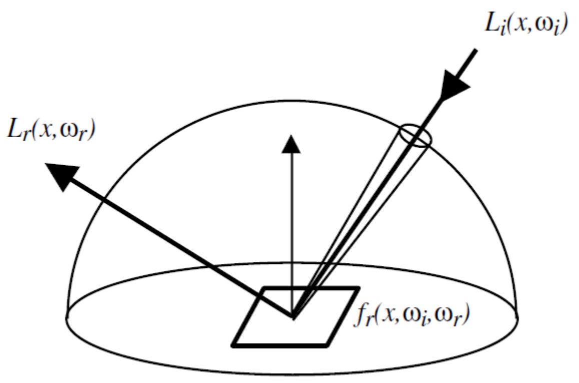


Figure 2.6: Rendering equation relate outgoing lighting by integrating incoming lighting form all direction over upper hemisphere.

The integration term in the rendering equation is evaluated using numerical methods and a large body of solutions to the rendering equation uses Monte Carlo techniques [Veach, 1997] [1]. The term Monte Carlo refers to algorithms that use the sampling theory and random variables in the physical simulations and function evaluation.

2.3 Previous Work in Rendering Techniques

The past decades have witnessed a significant advancement in computer graphics research and several techniques and algorithms have been proposed that either add more realism to rendered scenes and/or decrease the rendering time. Due to the diverse rendering techniques in computer graphics literature; proposing any taxonomy about past and current research directions in rendering techniques will be to a large degree inaccurate or incomplete. So we will just give a brief (and relatively biased) view about most popular rendering techniques which are still in use today.

2.3.1 Rasterization

Early graphics systems used rasterization techniques (like OpenGL and DirectX) [Hearn and Baker, 1994] in which $3D$ input primitives are rendered onto $2D$ displays after being processing in fixed parallel pipelined stages (e.g. modeling, projection, culling, and shading) on GPU. Although rasterization misses a lot of realism; it is still commonly used in games, CAD systems, modeling tools and other applications due its simplicity and its real-time frame rate. However, the limitation of the rasterized output of the fixed pipeline and the increased computation power of modern GPUs, allowed GPU chips companies to introduce the programmable pipeline [Pharr, Lefohn, Kolb, Lalonde, Foley, and Berry, Pharr et al.] in which input primitives are processed in a flexible parallel pipelined stages, and accordingly new high level languages called shading languages (e.g. GLSL, HLSL, Cg) have appeared. This new technology allowed researchers to render new effects and produce nearly physically-looking output [Engel, 2009]

[1]In his PhD [Subr, 2008] Subr presented a detailed survey about Monte Carlo methods in a chronological order, and the SIGGRAPH courses [Jensen, 2004; Jensen, Arvo, Dutre, Keller, Pharr, , and Shirley, 2003] are excellent references for Monte Carlo rendering techniques which explain both the theoretical and practical details of the method.

in a fast way by writing small programs called shaders that are applied in parallel on input primitive at certain processing steps in the programmable pipeline.

2.3.2 Physically Based Rendering

Physically based rendering techniques aim at creating naturally-looking synthetic images by solving the rendering equations numerically. Most of these technique can be grouped into two main categories; those use finite elements methods such as radiosity [Goral et al., 1984], and those use point sampling such as ray tracing methods [Cook et al., 1984; Whitted, 1980]. Due its ability to efficiently and simplicity to simulate various optical phenomena; point sampling methods attracted computer graphics researchers to develop many rendering techniques based ray tracing [Jensen, 2001; Veach, 1997].

Most ray tracing techniques used Monte Carlo methods [Jensen et al., 2003; Veach, 1997] to solve the high dimensional rendering equation numerically. And under certain assumptions Monte Carlo methods can render highly realistic images which are indistinguishable from images exist in real life. However, these methods first appeared as offline solutions due to many reasons including: the high dimensionality of the rendering equation; the low convergence rate of Monte Carlo solutions; and the relative complexity of simulated scenes (i.e. scene size).

Through decades, several techniques have been proposed to decrease the rendering time and achieve an interactive or even real-time frame rate. A large body of research work was focused on decreasing the number of traced rays by efficiently sampling the rendering equation [Hachisuka, Jarosz, Weistroffer, Dale, Humphreys, Zwicker, and Jensen, 2008], other directions of research where focused on efficiently tracing rays by developing fast and high quality data structures [Havran, 2000; Wald, 2004, 2007; Wald and Havran, 2006] for scene primitives or by developing faster ray traversal algorithms [Havran, 2000; Wald, 2004], or by tracing rays together in packet [Wald et al., 2006; Wald, Slusallek, Benthin, and Wagner, 2001]. On the other hand, recent advances in graphics hardware motivated new directions of research that make efficient use of both the increased computation power and the effective register SIMD (e.g., SSE, and GPU warp) instructions [Aila and Laine, 2009; Wald, Gnthery, and Slusalleky, 2004; Wald et al., 2006, 2001] and allowed them to propose fast parallel algorithms for hierarchy construction [Lauterbach et al., 2009; Zhou et al., 2008] and ray tracing [Garanzha and Loop, 2010].

2.3.3 Reyes Rendering Architecture

The Reyes (Renders Everything You Ever Saw) [Cook, Carpenter, and Catmull, 1987] rendering architecture was developed in 1987 by Cook et al. as a fast way to render photo-realistic images. Since that time, Reyes gained its popularity in movie industry [1] due the high quality and the relative speed of its output. Several stages of the Reyes architecture are amenable to parallelization; that inspired many researchers to implement a specific stage [Patney and Owens, 2008] or all stages [Zhou, Hou, Ren, Gong, Sun, and Guo, 2009] of the entire rendering pipeline on modern parallel machines. One limitation of the Reyes architecture is its inability to model many optical phenomena such as reflection and reflection, but recently Zhou et al. [Hou, Qin, Li, Guo, and Zhou, 2010] incorporated a ray tracing engine into Reyes pipeline which is fully implemented on GPU. Recent work in [Hou et al., 2010; Zhou et al., 2009] highlights several visual effects that have not been explored yet inside the Reyes architecture and present an interesting point for further investigation on massively parallel GPU.

2.3.4 Precomputed Radiance Transfer

Signal processing techniques allowed researchers to present new mathematical representations of reflection equation (i.e. the rendering equation under certain assumptions) [Ramamoorthi and Hanrahan, 2001] and introduce real-time rendering algorithms based on precomputed radiance transfer (PRT) [Ng, Ramamoorthi, and Hanrahan, 2003; Ramamoorthi and Hanrahan, 2002; Sloan, Kautz, and Snyder, 2002]. Most of PRT techniques assume distant environment lighting and simplifies the heavy integration term in the rendering equation to a matrix-vector multiplication which can be carried out efficiently on the graphics hardware (e.g. using shader programming). However, one of the drawbacks of the PRT method is that it requires a relatively complex and offline precomputation phase. Other drawbacks of this method include the large amount of precomputed data [Ng et al., 2003; Sun, Hou, Ren, Zhou, and Guo, 2011], and the relative complexity in rendering high frequency effects [Ng et al., 2003]. Interested readers can find a good start and an excellent recap about this area in the recent survey by Ramamoorthi [Ramamoorthi, 2009] [2].

[1]One successful Reyes rendering system which is commonly used in the movie industry is Pixar's Photorealistic RenderMan (PRMan)

[2]Novice readers can find a simple and detailed explanation about the PRT method in the technical report by Green [Green, Green] and the author's homepage contains a simple tutorial about the method `http://www.`

paulsprojects.net/opengl/sh/sh.html.

3

Introduction to GPU Parallel Computing

3.1 GPU and Parallel Computing

In 1965 Gordon E. Moore came up with his law [Moore, 2000] that the number of transistors would double every 18 months. For decades Moore's law successfully applied to everything including processors, memory chips, and wireless devices and was considered one of the main driving forces behind hardware progress and accordingly the related software technology. On the other, the demands for more real-time and high definition $3D$ graphics motivated hardware chips companies to produce highly parallel, scalable and efficient GPU (graphics processing unit). The good news was that GPU technology broke Moore's law and has shown to move faster than Moore's law timeline in the past few years.

Although GPUs were first designed to support shader programming in which a small program runs in parallel to process geometry and draw pixels, it was later used as a general parallel architecture for non-graphics applications. These applications were referred as GPGPU (General-Purpose computation on Graphics Processing Units) [1] and many researchers have shown promising success and impressive results by using GPU as a massively parallel processor for solving different scientific problems.

[1]The GPGPU term was coined by Mark J. Harris [Harris, Owens, Sengupta, Zhang, Davidson, and Tseng, 2007].

3.2 The CUDA Programming Model

CUDA [NVIDIA, 2010] stands for Compute Unified Device Architecture. It is a C like programming language designed to provide an abstraction for general purpose parallel computing over GPU. The CUDA programming involves running code on two different platforms: a host system that relies on one or more CPUs, and a card (frequently a graphics adapter) with one or more CUDA-enabled NVIDIA GPUs (the device).

3.2.1 Host and Device

As illustrated in Figure 3.1; a parallel program start with a serial code on CPU that launches many copies (up to millions) of simple function (kernel) that executes in parallel on the GPU device. The host program is also responsible for memory allocation and de-allocation on the device, as well as data transfer between host and device.

3.2.2 Thread Hierarchy

GPU can schedule millions of parallel threads running on different cores and executes the same kernel on different data elements (Single Instruction Multiple Data Stream -SIMD-). Threads are organized into $1D$, $2D$, or $3D$ groups called thread blocks, and thread blocks are organized into $1D$, $2D$ groups called grids. This organization has a tight correspondence to the memory organization inside GPU device, the communication way between various kernels, and kernel execution and scheduling.

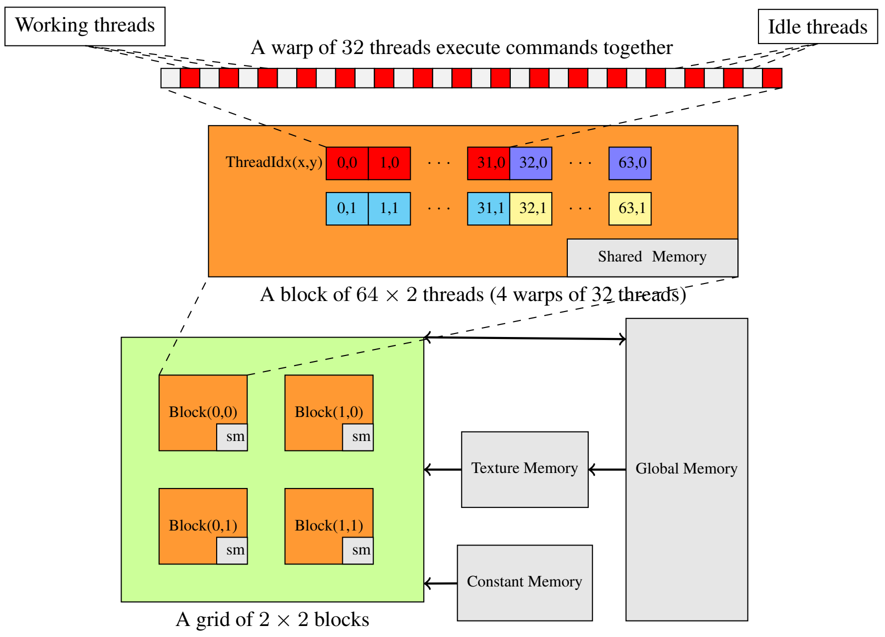
3.2.3 Memory Hierarchy

CUDA assumes that both the host and the device maintain their own DRAM, referred to as host memory and device memory respectively. Inside the device CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 3.1. A CUDA device memory space can be classified as global, local, shared, texture, constant, and register memory.

Global memory space is not cached and has the life time of the application and is visible to all threads so it is used to store large blocks of data and it can be the best way to achieve global

Kernel code executes on device

```
__global void ProcessOnGPU(int*Data, int Size)
{
    extern __shared__ int sdata[];
    ...
    __syncthreads();//block's synchronization, all block's threads meet here
    if(threadIdx.x%2)
    {
        // do instruction x
    }
    else
    {
        // do instruction y
    }
    ...
}
```



```
int main()
{
    int size = 512;
    int *HostData, *DevicData;
    HostData = (int*)malloc(size*sizeof(int));
    cudaMalloc(DevicData, size*sizeof(int));
    cudaMemcpy(DevicData, HostData, size*sizeof(int), cudaMemcpyHostToDevice);
    int sharedMemSize = 128*sizeof(int);
    dim3 grid(2,2);
    dim3 block(64,2);
    ProcessOnGPU<<<grid, block, sharedMemSize>>>(DevicData, size);
    cudaMemcpy(HostData, DevicData, size*sizeof(int), cudaMemcpyDeviceToHost);
    cudaFree(DevicData);
    free(HostData);
    return 0;
}
```

Serial code executes on host and launches kennel

Figure 3.1: GPU internal organization and parallel code execution.

synchronization for a parallel program. Local memory is not cached and visible only inside a kernel so it is used for automatic variables. The constant memory is a small part of memory that is cached and visible to all threads so it is used for relatively small and frequently used constants. The texture memory space is a part of memory that is cached and visible to all threads and can be used to map relatively large and frequently used blocks of global memory to achieve better cache performance. The shared memory space is much faster than global memory space and visible to all threads inside a single block so it is used to copy small parts of global memory into block space for better cache performance inside the block. The register memory is visible inside a kernel and costs zero extra clock cycles per instruction. Table 3.1 summarizes the properties of memory spaces on NVIDIA GTX285 card which will be used in our experiments.

Memory Space	Scope	Lifetime	Size on GTX285
Global	Application	Application	1 GB
Local	Thread	Thread	–
Constant	Application	Application	64 KB
Shared	Block	Block	16 KB/Block
Register	Thread	Thread	16 K. Register/Block

table 3.1: Device memory spaces and corresponding size on NVIDIA GTX285 card.

3.2.4 SIMD/SIMT Execution

In SIMD machines the same instruction is executed on multiple data elements to achieve parallelism. Whereas the same concept applies to SIMT (single instruction multiple threads) machines like GPUs; SIMT machines differ in many respects: although SMID instructions were limited to very simple parallel instructions (e.g. additions and multiplications) on simple data elements; SIMT machines magnifies the scale of the instruction to include a complete program (kernel). To do so SIMT wide machines like GPU achieve parallelism in two ways: first, it schedules, fetches, and executes threads in groups of (32) threads called warps; then, inside the warp all the threads execute the same instruction in parallel. A warp executes one common instruction for all threads at a time; first, the device hardware calculates various data-dependent conditional branches and memory requirements for all threads in the warp, then it serially executes each branch separately, threads that do not follow certain branch remain idle. So full efficiency is realized when all 32 threads of a warp agree on their execution path.

We illustrate the concept of SIMT execution in Figure 3.1. As you notice the host code running on CPU calls the *ProcessOnGPU* kernel and launches 512 threads in total (a grid of 2×2 blocks, where each block is 64×2 threads). Each thread has its own copy of the kernel code and threads evaluate and execute certain commands in warps of 32 threads. This is illustrated by highlighting each warp with a color similar to the line of code it executes; for example, in block 1; the first warp executes line 6 and the second warp executes line 8. Inside the warp itself some threads may become idle since they don't pass through this execution path and this appears in odd numbered threads in warp 1 which don't pass through the *if* condition in line 5.

The device function call *__syncthreads* is the local synchronization point inside the block. All threads inside the block don't continue execution until all threads reach this point and evaluate this function call. In general we use the local synchronization in order to confirm read/write dependency, and this often occurs with shared and global memory transfers.

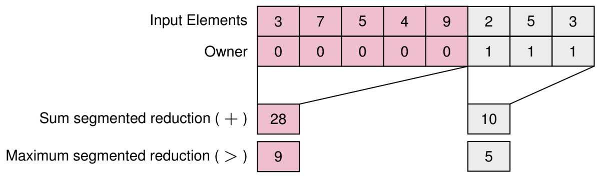
3.3 Data Parallel Primitive Algorithms on GPU

Data parallel primitive algorithms [Sengupta, Harris, Zhang, and Owens, 2007] are the main building blocks for most general purpose applications on GPU. The driving motivation behind these algorithms is the complex access pattern on input elements to produce output elements in parallel and their importance arise in transforming nested data-parallel programs into a flat data-parallel structure suitable for massively parallel GPU. Examples of such primitives include parallel reduction, parallel scan [Harris, Sengupta, and Owens, 2007; Shubhabrata Sengupta and Garland, 2008], parallel list split, parallel list compaction [Sengupta et al., 2007] and their segmented versions [Sengupta et al., 2007; Zhou et al., 2008], and parallel sorting [Satish, Harris, and Garland, 2009a], and compress-sort-decompress (CSD) [Garanzha and Loop, 2010]. Although most of these parallel primitives appeared earlier in classic references and implemented using other models (e.g. PRAM model) [Quinn, 1993]; importing these algorithms directly to GPU may not scale well with large inputs due to different architecture offered by GPU. And since we make heavy use of these algorithms in our work and later we extend the primitives toolbox for hierarchal tree construction algorithms (see Chapter 4), we will give a brief introduction about these algorithms in this section.

3.3.1 Parallel Reduction and Segmented Reduction

The reduction algorithm takes as input an array and an arithmetic operation $\oplus$ to produce a single value which is the result of applying the operator $\oplus$ to all elements in the input array. For example consider the array $[3, 7, 5, 4, 9, 2, 5, 3]$, if the operation $\oplus$ is an addition ($+$) then the reduction result is 32, and if the operation $\oplus$ is a maximum ($>$) then the result is 9.

The segmented reduction primitive [Zhou et al., 2008] performs the reduction operation to various partitions of the input array separately. The segmented reduction requires an additional array (owner array) as an input in which each element is set to an integer value representing the owner segment index of the corresponding input element. Figure 3.2 show the segmented reduction output using various associative operations on the array $[3, 7, 5, 4, 9, 2, 5, 3]$.



Input Elements
3
7
5
4
9
2
5
3
Owner
0
0
0
0
0
1
1
1
Sum segmented reduction ($+$)
28
10
Maximum segmented reduction ($>$)
9
5

Figure 3.2: Segmented reduction example using sum and maximum operations.

Although the segmented reduction algorithm requires the accessibility of all data elements in the input array to produce output values; an $O(N)$ work efficient algorithm [Zhou et al., 2008] can be implemented on GPU which requires only $\log(N)$ steps and single pass on array elements.

3.3.2 Parallel Scan and Segmented Scan

The scan operation takes an array of elements as an input and an associative binary function $\oplus$ with an identity value i and produces another array of the same size by performing the operation $\oplus$ on input elements preceding and possibly including each element. Scan is a critical algorithm which is used to transform nested data-parallel programs into a flat data-parallel structure. For the input array $[a_0, a_1, a_2, \cdots]$; an exclusive scan performs the operation on every element preceding the current one and returns the array $[i, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, \cdots]$, and an inclusive scan takes into consideration the current element and produces the array $[a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, a_0 \oplus a_1 \oplus a_2 \oplus a_3, \cdots]$. A backward scan performs the scan operation form the end to the beginning of the array and a segmented scan operates by scanning array partitions separately and requires an additional array for *Head Flags* marking the input array partitions, each element in the *Head Flags* array is set to 0 except at head locations of each segment which are set to 1.

Table 3.2 lists the most frequently used operations for the scan operation and related identity values, and Figure 3.3 shows an example of the scan and segmented scan using the sum operation on the array $[3, 7, 5, 4, 9, 2, 5, 3]$.

Operation	Operator ($\oplus$)	Identity Value (i)
Sum	$+$	0
Maximum	$>$	$-\infty$
Minimum	$<$	∞

table 3.2: Frequently used scan operations.

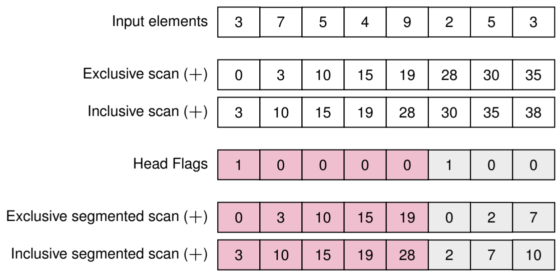


Figure 3.3: Scan and segmented using the sum $(+)$ operator.

As you notice the scan algorithm presents a complex access pattern that requires the accessibility of a variable number of input elements based on the relative position of each output element. Without an efficient parallel implementation we may stick with an inefficient memory usage or inefficient work load that causes the performance to reach the serial version of the algorithm. To the best of our knowledge; an efficient $O(\log(N))$ GPU parallel algorithms can be used to compute the scan [Harris et al., 2007] and segmented scan [Satish et al., 2009a; Sengupta et al., 2007] using only two passes over the input elements, these algorithms have a work complexity $O(N)$ which is the same work complexity [1] of the optimal serial scan algorithm.

[1]The work complexity refers to the total number of operations on input elements

3.3.3 List Compaction

The inputs to the list compaction primitive [Sengupta et al., 2007] are two arrays of the same size; one for input elements and the other for true/false flags so that we need to keep and compact elements corresponding to true values in the flags array.

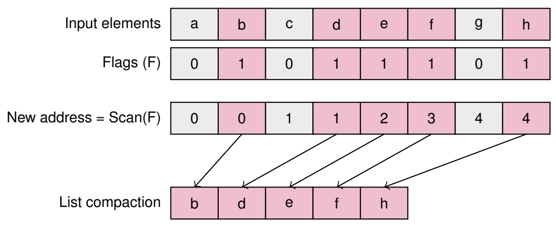


Figure 3.4: List compaction example.

List compaction is implemented by substituting the true and false flags by 1 and 0 respectively, then we apply an exclusive sum scan to the $1/0$ *Flags* array, after the scan we check each element at index i if its corresponding flag is 1; we store it at an address equals the value at the same index i in the scanned array (see Figure 3.4).

3.3.4 List Split and Segmented List Split

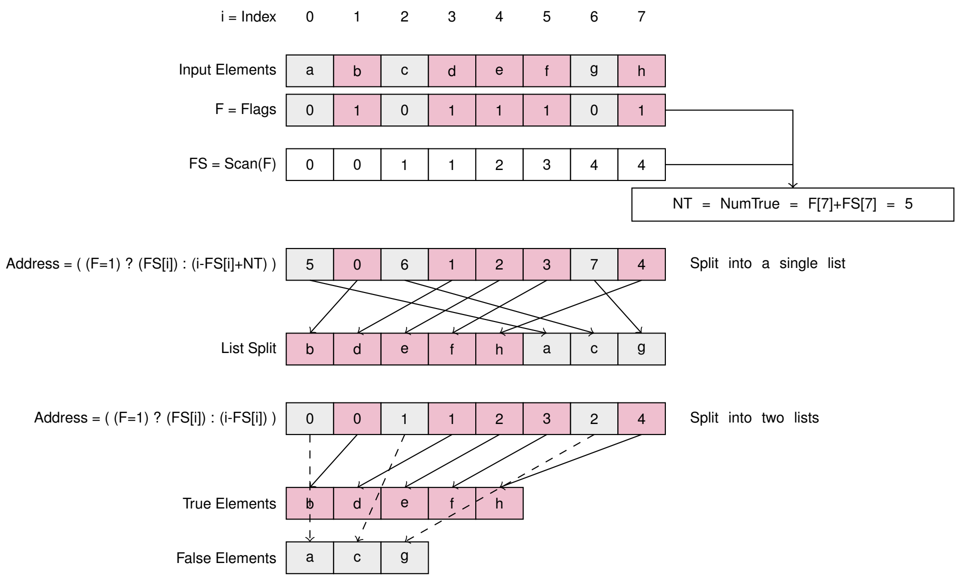


Figure 3.5: List split example.

In list split we divide an array of elements into two pieces based on another array of true/false flags of the same size. We can either separate the input elements into two arrays or move all elements marked true to the left of all elements marked false.

Similar to list compaction we first substitute the true and false flags by 1 and 0 respectively and perform an exclusive scan to the $1/0$ *Flags* array. Each input element at index i marked as true is scattered to an address equals to the corresponding value in the scanned array *(Scan(Flags)[i])*, and each element marked as false is scattered to an address equals to *(i - Scan(Flags)[i] + NumTrue)*. Where *NumTrue* is the total number of true elements which is calculated by summing the last value in both the *Flags* array and its scan. In some cases we are interested in scattering true elements to an array and false elements to another array in such case we keep the new address of true elements as in the previous case and modify the new address of false elements to be *(i - Scan(Flags)[i])* which represents the preceding number of false elements (see Figure 3.5).

The segmented list split can be implemented the same way we implement the nonsegmented version. But in fact we may need to know the start and size of each subarray at each segment. So we use the segmented scan to scan the flags array and get the scan tails of each segment to represent the size of left subarray of each segment and the consider the size of right subarray of each segment as the difference between the input segment size and left subarray size. We append the *right scan tails* to the *left scan tails* and scan this array if we want to split segments into a single array, or scan each *left scan tails* and *right scan tails* arrays separately if we need to split segments to two different arrays. We consider the scans of the scan tails as the start addresses of output segments and after that we scatter each elements; if the element goes to left subarray then we scatter it at an address defined by to the start address of its new output segment plus an offset defined by the local running scan of ones, and if the element goes to right subarray then we scatter it at an address defined by to the start address of its new output segment plus an offset defined by the local running scan of zeros.

Practically we can use a nonsegmented scan for the flags array but in such case we define the local running scan of ones in each segment as the difference in the scanned array between the values at corresponding element index and the value at the start index of the old parent segment, and define the local running scan of zeroes in each segment as the difference local offset of the element in the old parent segment and the local running scan of ones until this element.

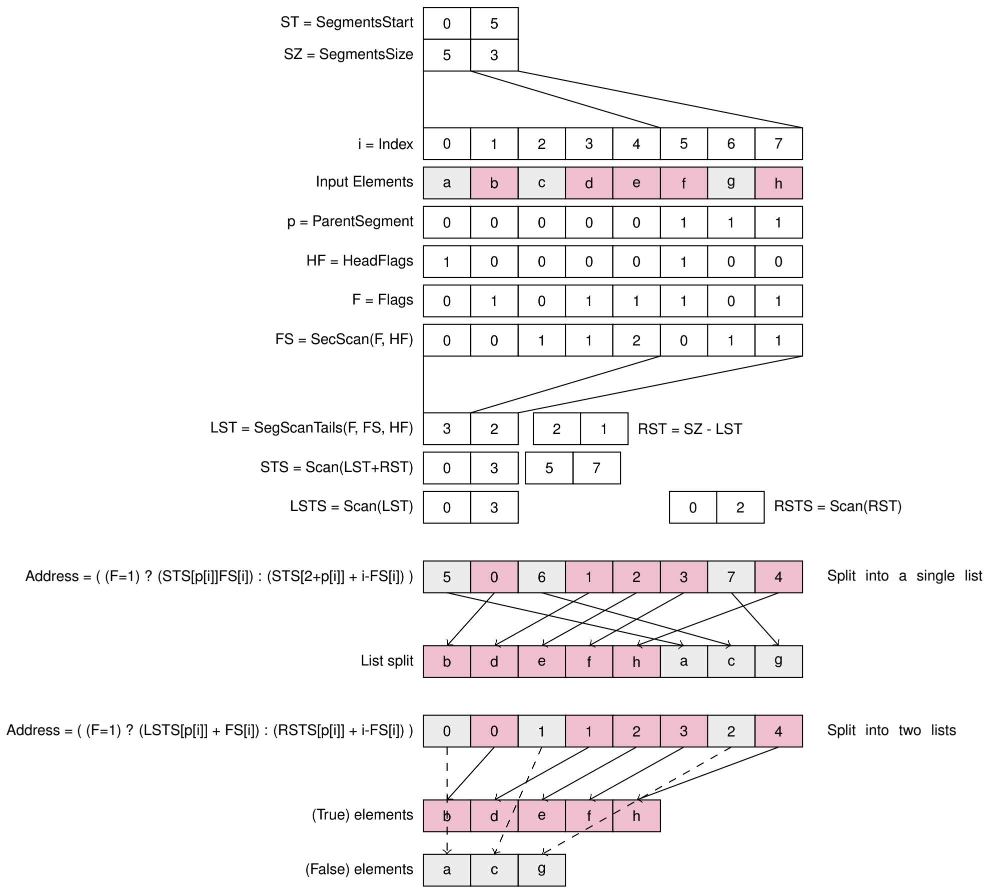


Figure 3.6: Segmented List split example.

4

Parallel Hierarchical Tree Construction Algorithms on GPU

4.1 Motivation and Previous Work

Closely related research to our work includes hierarchical tree construction on both CPU and GPU and parallel primitive algorithms on GPU. In this section we review the key techniques and algorithms in each of these areas.

4.1.1 Spatial Partitioning Data Structures

Dominant spatial partitioning data structures include grids, Octrees, bounding volume hierarchies (BVHs), and KD-trees [Havran, 2000]. The use of grids in interactive applications is motivated by their fast per-frame build time [Wald et al., 2006]; but due to the better quality produced by other structures; BVHs, and KD-trees are preferred in most applications even with their slower build time [Wald, 2004]. Efficient ways for building BVHs and KD-trees are used interchangeably because of their similar tree structures, and most optimization techniques try to make a tradeoff between the construction time and the resulting tree quality and accordingly the traversal time. The key idea behind these techniques is where to place internal node splits and when to stop node splitting. Simple and fast construction methods are based on median node splitting; either at the spatial median as in KD-trees, or at the object median as in BVHs. However, surface are heuristics (SAH) construction techniques [Goldsmith and Salmon, 1987] are widely used and known to be the best way to construct high quality spatial partitioning data structures for ray tracing and other applications [Havran, 2000; Wald, 2004].

SAH construction algorithms are known to perform slowly since they require the evaluation of the discreet SAH cost function at many split candidates in each node and efficient ways to evaluate full sweep SAH usually require one or more sorting passes over all primitives [Pharr and Humphreys, 2010; Wald et al., 2004; Wald and Havran, 2006]. To address this issue researchers often sample and approximate [Wald, 2007] or interpolate [Hunt, Mark, and Stoll, 2006; Popov, Günther, Seidel, and Slusallek, 2006] the SAH at a small number of candidates using projection and scanning and it was noticed that even with a small number of candidates the SAH algorithms still provide good tree quality compared to other techniques [Wald, Boulos, and Shirley, 2007]. Parallel extensions to the SAH construction algorithms on multi-core CPU have been addressed (e.g. in [Hunt et al., 2006; Popov et al., 2006; Wald, 2007]) and used to accelerate the construction process by up to orders of magnitudes. But since these parallel algorithms were designed to work with a small number of threads they are not expected to scale well on massively parallel architectures like GPUs.

4.1.2 Parallel Tree Construction on GPU

In [Lauterbach et al., 2009] Lauterbach et al. presented an efficient SAH BVH construction algorithm that runs at interactive rates, and presented a fast BVH construction algorithm (LBVH - Linear Bounding Volume Hierarchy) which favors the build time over tree quality and reduces the construction process to parallel sorting. Lauterbach et al. addressed the slow build time of the SAH BVH algorithm and the low tree quality of the LBVH algorithm by combining the two algorithms into a hybrid one that builds the higher tree levels by the LBVH algorithm and the lower tree levels by the SAH BVH algorithm. The resulting hierarchies from the hybrid algorithm have shown to be of comparable quality to those produced by the SAH BVH algorithm while their build time was relatively faster.

Zhou et al. [Zhou et al., 2008] presented a fast algorithm for building KD-tree on GPU in breadth first search (BFS) order where the algorithm classifies processed nodes into two categories -large nodes and small nodes- according to the number of triangles in each node. To balance between the build time and tree quality Zhou et al. used the spatial median and “empty space" maximization strategies to guide large nodes splits, and approximate SAH strategy to guide small nodes splits.

In this thesis we extend the idea of large/small nodes categorization to construct binned SAH BVH [Wald, 2007] by classifying nodes into more than two categories. We also employ the approximate SAH to guide node splits at all tree levels and use the LBVH algorithm to construct higher tree levels and accelerate the build time in a hybrid BVH algorithm. In contracts to the work done by citeauthor Danilewski2010 [Danilewski, Popov, and Slusallek, Danilewski et al.] which uses atomic operations for nodes classification; our interest is to show that this operation and other operations required for parallel tree construction can be reduced to a standard set of data parallel primitives that allow us to avoid the serialization issues incurred by atomic operations.

One limitation of BVH and KD-tree construction algorithms on GPU [Lauterbach et al., 2009; Zhou et al., 2008] is the excessive memory used to store temporary data which imposes a limitation on applications involving large and complex models. To cope with this problem; Hou, Sun, Zhou, Lauterbach, and Manocha [Hou et al., 2010] presented a lazy algorithm that builds hierarchies on GPU using partial breadth first search (PBFS) manner by adjusting the maximum work load according to the available device memory. Thus, to allow an efficient use of fixed device memory they employed a simple but efficient scheme for anti-fragmentation dynamic buffer management to store static and dynamic data on GPU. Although the PBFS approach can be trivially extended to our binned SAH algorithms; we choose to build the hierarchies in BFS order because our interest is to exploit the GPU streaming architecture for real-time and interactive per-frame hierarchy construction in ray tracing applications. However, our algorithms employ the buffer management strategies used in [Hou et al., 2010] for better use of device memory.

Very recently Pantaleoni and Luebke [Pantaleoni and Luebke, 2010] presented the Hierarchical Linear Bounding Volume Hierarchy (HLBVH) algorithm which constructs a hierarchy typical to that produced by LBVH while enhancing the build time and memory bandwidth overhead on GPU. To further enhance the tree quality produced by HLBVH Pantaleoni and Luebke [Pantaleoni and Luebke, 2010] presented a slower hybrid algorithm which uses the binned SAH [Wald, 2007] for building the higher tree levels and the HLBVH for building the lower tree levels. However, we choose LBVH to create our coarse grained hierarchy in our hybrid algorithm since at higher tree levels the GPU memory bandwidth overhead is not a major limitation in LBVH. On the other hand our hybrid algorithm is the opposite to their algorithm since they build the higher levels of the hierarchy using SAH and the lower levels using HLBVH.

4.1.3 Data Parallel Primitive Algorithms on GPU

Data parallel primitive algorithms [Sengupta et al., 2007] are the main building blocks for most general purpose applications on GPU. The importance of these algorithms arises in transforming nested data-parallel programs into a flat data-parallel structure suitable for GPU. Examples of such primitives include parallel reduction, parallel scan [Harris et al., 2007; Shubhabrata Sengupta and Garland, 2008], parallel list split, parallel list compaction [Sengupta et al., 2007] and their segmented versions [Sengupta et al., 2007; Zhou et al., 2008], parallel sorting [Satish et al., 2009a], and compress-sort-decompress (CSD) [Garanzha and Loop, 2010]. In our study we found that such primitives can be extended to include new primitives that wrap the general parallel tree construction operations such as node partitioning, node splitting, and triangles sorting, we also show that a simplified API can serve as a wrapper for most of these primitives on GPU.

4.2 Data Parallel Operator and Data Parallel Utilities

In this section we introduce the data parallel operator which we use as a synonym of a parallel kernel, and define frequently used parallel utilities which represent wrappers for data parallel primitives [Harris et al., 2007; Sengupta et al., 2007] on GPU. These definitions introduce a lot of terminologies used in explaining the details of our algorithms for the rest of this thesis.

4.2.1 Data Parallel Operator

A parallel operator defines a function that operates on a single data element and called to process multiple data elements in parallel. It represents a parallel code fragment inside a ***foreach*** $\cdots$ ***in parallel*** construct and often reflects a kernel definition/launch on GPU.
For example to define the distribute primitive [Sengupta et al., 2007] which copies a single value to all array elements; we define the *Distribute* operator as shown in Listing 4.1.

```
operator Distribute (O, I)
{
    i = ThreadIndex
    O[i] = I
}
```

Listing 4.1: Distribute operator definition

Then given an array *Data* of size *N* and an element *E* we call the *Distribute* operator as: `Distribute<N>(Data, E)` to copy *E* to all elements of *Data* in parallel.

So far we assumed that we have a direct access to a global thread index *(ThreadIndex)* in the operator definition and considered GPU as a general multi-threaded machine which performs the same operation *N* times on different data elements in parallel.

4.2.2 Frequently Used Data Parallel Utilities

In order to allow better parallel code readability and explanation we present parallel utilities in Appendix A to serve as a wrappers to most frequently used parallel primitives appeared in literature. Table 4.1 lists and explains the parallel unities which will be used throughout the thesis.

Utility	**Operation**
Reduce	Performs standard reduction
SegReduce	Performs segmented reduction
Scan	Performs standard scan
SegScan	Performs segmented scan
ScanTail	Get scan tail (e.g. last value of the scan result)
SegScanTails	Get scan tails of various array segments
Append	Appends an array to another one
Compact	Performs parallel compaction of non-null elements
Split	Splits an array into two subarrays
SegSplit	Splits an array segments into two subarrays segments
Sort	Performs key-value sorting
FindSortedBounds	Finds the start index of each sorted cluster, and the size of a cluster
Rand	Generates random numbers

table 4.1: Parallel utilities which will be used in all subsequent sections.

4.2.3 Extensions to Data Parallel Operators and Utilities

Extending Operations to Higher Dimensions. In the former example of the *Distribute* operator we ignored the internal grid/block organization of GPU threads; to allow the parallel operator to reflect this organization we have two options:

1. Either by nesting the operator definition and call.
   For example given the operator *Distribute* which we defined earlier; we define an operator *HighDisitrbute* as show in Listing 4.2.

```
operator HighDisitrbute(O, I, N)
{
    i = ThreadIndex
    Disitrbute<N>(O[i], I[i])
}
```

Listing 4.2: HighDistribute operator definition

   And then given a $2D$ array *Data2D* of size $M \times N$ and a array *E1D* of size M we call this operator as: HighDisitrbute<M>(Data2D, E1D, N) to copy each element in *E1D* to the corresponding row in *Data2D* array in parallel.

2. Or by reflecting grid/block organization in the operator call such as:
   Operator < (GridDimX, GridDimY, GridDimZ) (BlockDimX, BlockDimY, BlockDimZ) > (Params) , In such case we assume that we have a direct access to variables *GridIdX, GridIdY, GridIdZ, BlockIdX, BlockIdY, BlockIdZ* in the operator definition which represent block and grid indices for each respective axis. Using the same example to copy elements in the array *E1D* to the rows of the $2D$ array *Data2D*; we define the *Disitrbute2D* operator as show in Listing 4.3

```
operator Distribute2D(O, I)
{
    row = GridIdX
    col = BlockIdX
    O[row][col] = I[row]
}
```

Listing 4.3: Distribute2D operator definition

   And then we call it as: Distribute2D< (M), (N) >(Data2D, E1D).

Vector Parameters. In many cases we may need to apply the same parallel operation on multiple parameters, in such case instead of writing many calls of the same operator or utility for different data elements we allow the input/output parameters to take the vector format *(<Param$_1$,Param$_2$,...>)*, for example the call ***Distribute<N>(<Data$_1$, Data$_2$>, <E$_1$,E$_2$>)*** will apply the same distribute operation between each pair (Data$_i$, E$_i$), and the call ***Scan(<O$_1$, O$_2$>, <I$_1$,I$_2$>,$\cdots$)*** will apply the scan operation between each pair *(O_i,I_i)*.

Standard Numerical and Logical Operators. We also allow an implicit parallel extension to standard numerical and logical operators on arrays. For example, given two arrays ***Data$_1$, Data$_2$***, the command ***Data$_1$+Data$_2$*** will ***sum*** the two arrays element-wise in parallel, and the command ***Data$_1$ & Data$_2$*** will ***AND*** the two arrays element-wise in parallel.

4.3 BFS Tree Construction Algorithms on GPU

A general breadth first search (BFS) algorithm begins with a root node which encloses the scene bounding box and contains all the primitives in the scene and builds the hierarchy in two main steps: (1) for N tree nodes the algorithm tries to find the best split plane for each node according to some heuristics and then either splits the node or keeps it unsplit; (2) for each split node the algorithm distributes its triangles to its child nodes and returns to step 1 to process new child nodes. The input to each iteration is an array of nodes and a triangle-node association array which stores triangle indices contained in the nodes array sorted by node index. Each node store the scene space it occupies, while each internal node stores pointers to its child nodes and each leaf node stores the index of its first triangle in the triangle-node association array and the number of triangles it contains.

On GPU some operations are parallelized on the nodes level and have to access child triangles while some operations are parallelized on the triangles level and have to access parent nodes. For the node-level parallelization we can use the node's start triangle index and the number of triangles to access the its triangles in the triangle-node association array, but for the triangle-level parallelization in which we need to access parent nodes; we have to maintain an indirection array that store for each triangle its parent node index in the nodes array.

Assumptions:
For better cache performance on GPU; all tree nodes data structures are stored in separate arrays using structure of array *(SoA)* format [Wald, 2004]. Practically, we create and store all

tree nodes in a single array structure; but during certain algorithm steps we may be interested to process or store a specific subset of nodes which may not conserve a contiguous block in the nodes array; so we use an array of indices to reference such subset of nodes. For example: the *ActiveNodes* array is used to refer to currently processed nodes indices in the nodes SoA. As shown Figure 4.1 the *NodeID* represents the node index in the nodes SoA, and the *Index* represents the index to the *ActiveNodes* array at which we store the *NodeID*; that is why we always assume that *ActiveNodes* array is referenced by a consecutive integer sequence starting from 0. The *ActiveTriangles* is the triangle-node association array at which we store triangles indices of active nodes and the *ActiveParents* array stores indices of the *ActiveNodes* array at which we store triangles parents *NodeID*.

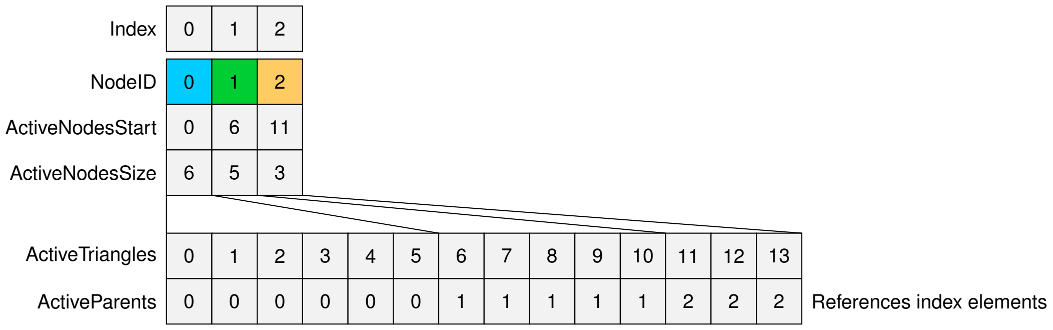


Figure 4.1: Active nodes and associated triangles indices.

In next sections we review recent state of the art BFS tree construction algorithms on GPU presented in [Lauterbach et al., 2009; Zhou et al., 2008] using our proposed API, and present our new algorithms for building binned SAH BVH.

4.4 Parallel SAH KD-tree Construction Algorithm

In [Zhou et al., 2008] Zhou et al. presented a fast parallel KD-tree algorithm which builds tree hierarchy on GPU in BFS order. The algorithm classifies tree nodes into large and small nodes, and uses different split strategies for each category. A node is considered large if its triangles are greater than a predefined threshold T, otherwise, it is considered a small node. The main construction pipeline starts with the root node and builds the hierarchy in two main stages: (i) large node stage in which we split all large nodes using simple and inexpensive split cost evaluation methods; and (ii) small node stage in which we split small nodes after splitting all large nodes employing an approximate SAH cost evaluation for node splits. In this section we explain the main processing steps of this algorithm.

4.4.1 Large Node Stage

In this stage we start with the root node and process all large nodes in six main steps: (1) divide node's triangles into fixed size chunks; (2) compute the bounding box of all nodes using standard and segmented reduction on the chunks data; (3) split nodes; (4) sort triangles to child nodes; (5) filter child nodes into large and small nodes; (6) clip and distribute triangles to child nodes. Then we return to step 1 if we still have large nodes or go to next stage if not. Since large nodes appear at the higher tree levels; it is better to use simple and inexpensive heuristics based on "empty space" maximizing [Havran, 2000] and spatial median [Wald, 2007] for node splits.

Step 1: Dividing Node's Triangles into Fixed Size Chunks.

To divide triangles of each node into fixed dized chunks of at most T triangles we apply the *CreateChunks* utility explained in Appendix B on *ActiveNodes* data. Figure 4.2 shows the resulting chunks data structure of our running example.

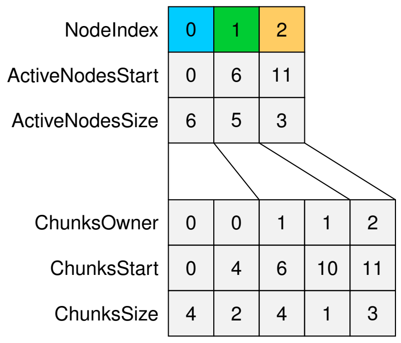


Figure 4.2: Dividing node's triangles into fixed sized chunks of at most 4 triangles.

Step 2: Computing Nodes Bounding Box.

First, we customize the *Reduce* utility to reduce the bounding boxes of all triangles in a single chunk in parallel. This utility takes as an input the chunk range from *ChunksStart*, and *ChunksSize* arrays and consider the identity element for an Axis Aligned Bounding Box (AABB) *(B_t)* as $[\infty, -\infty]$ and the reduction operation between two AABBs *(B_1,B_2)* as *[min(B_1.min,*

B_2.min), max(B_1.max, B_2.max)]. Then we invoke a nested parallel call for this utility to calculate the bounding boxes of all chunks in parallel into array *ChunksAABBs*, and call a customized version for AABBs of the *SegmentedReduction* utility to calculate nodes AABBs by considering *ChunksAABBs* as the input elements and *ChunksOwner* as the *Owner* array.

Step 3: Splitting Nodes.
To account for both "empty space" maximization and spatial median as the splitting criteria for large nodes we keep two bounding boxes for each node; the tight bounding box *(B_t)* calculated in the previous step, and the inherited bounding box *(B_i)* which recursively inherits split planes from its parent nodes starting with the scene AABB at the root node.

We calculate the "empty space" as the difference between the two bounding boxes at each of their 6 side planes. If the "empty space" at certain side is greater than a predefined threshold C_e relative to the corresponding axis then we split this node at the tight bounding box position into two child nodes; an empty node and an inherited node which inherits both the parent node primitives and its tight bounding box B_t.

To parallelize this step we make an operator that processes all nodes in parallel. In this operator we count for each node the number of sides that pass the "empty space" threshold and store this count into array *NumEmptyNodes*, we also make a 6 bits mask for each node into which we set bit i if the corresponding side passes the "empty space" threshold, where $i \in [0-5]$ and corresponds to min and max sides for the *x*, *y*, and *z* axes respectively, we store these masks into array *EmptySides*. After calling this operator we perform an exclusive scan to the *NumEmptyNodes* array using the scan utility (see Figure 4.3).

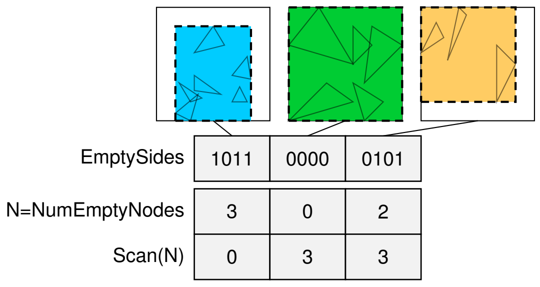


Figure 4.3: Empty space calculation.

Once we scanned the number of empty nodes we make an operator that splits all nodes in parallel. In this operator, given a node N at index i in the *AvtiveNodes* array we get the start address of its children using the scan of the *NumEmptyNodes* array as $2Scan(NumEmptyNodes)[i]+2i$ and use the corresponding bit mask in the *EmptySides* array to create empty and inherited nodes and then we split the lowest inherited node at the spatial median into two child nodes and store them into positions $2i$, $2i+1$ in the *ChildNodes* array (see Figure 4.4).

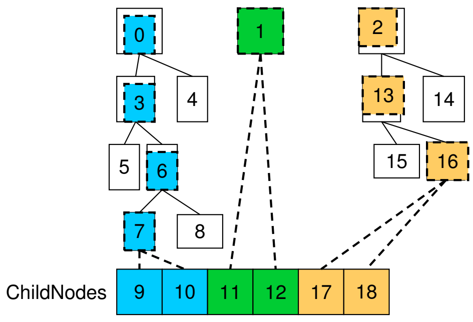


Figure 4.4: Large nodes splitting.

Step 4: Sorting Triangles to Child Nodes.

We prepare two arrays of flags *(Left, and Both)* equal in size to the *ActiveTriangles* array and make an operator which sorts triangles of a single chunk to the new child nodes in parallel. This operator takes as an input the chunk data from *ChunksStart*, *ChunksSize*, and *ChunksOwner* arrays and fills in the *Left* and *Both* array elements corresponding to the chunk range. In this operator we check the triangle against its parent node split plane (a triangle can access its parent node using the *ChunksOwner* element) and store into the corresponding *Left*, and *Both* array elements: 1, and 0 if the triangle lies to the left of the split plane; 0, and 0 if it lies to the right of split plane; and 1, and 1 if it is straddling the split plane.

We invoke a nested parallel call to this operator to sort triangles of all chunks, and call the *Scan* utility to perform a exclusive scan for the *Left*, and *Both* arrays. Then we call *SegScanTails* utility employing *ActiveNodesStart* and *ActiveNodesSize* as the *HeadIndices* and the *Size* arguments respectively and store the scan tails of *Left*, and *Both* arrays into *LeftScanTails*, and *BothScanTails* arrays respectively (see Figure 4.5).

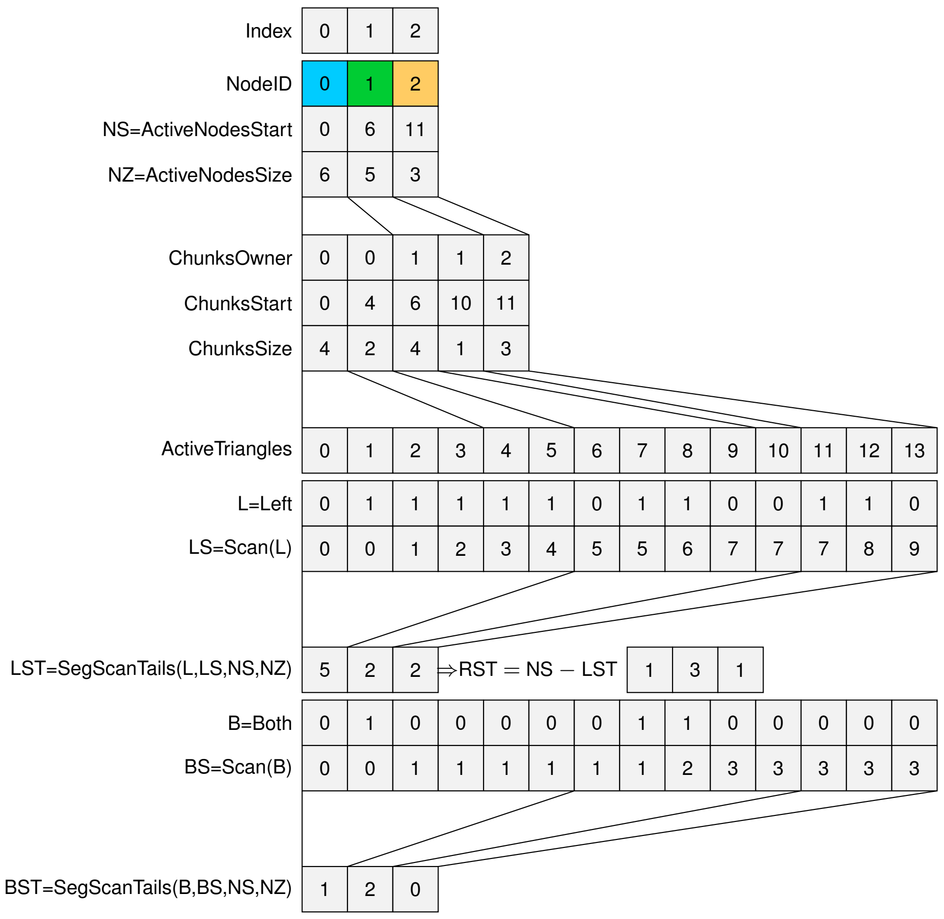

Index 0 1 2
NodeID 0 1 2
NS=ActiveNodesStart 0 6 11
NZ=ActiveNodesSize 6 5 3
ChunksOwner 0 0 1 1 2
ChunksStart 0 4 6 10 11
ChunksSize 4 2 4 1 3
ActiveTriangles 0 1 2 3 4 5 6 7 8 9 10 11 12 13
L=Left 0 1 1 1 1 1 0 1 1 0 0 1 1 0
LS=Scan(L) 0 0 1 2 3 4 5 5 6 7 7 7 8 9
LST=SegScanTails(L,LS,NS,NZ) 5 2 2
⇒RST = NS − LST 1 3 1
B=Both 0 1 0 0 0 0 0 1 1 0 0 0 0 0
BS=Scan(B) 0 0 1 1 1 1 1 1 2 3 3 3 3 3
BST=SegScanTails(B,BS,NS,NZ) 1 2 0

Figure 4.5: Sorting KD-tree triangles to child nodes using two flags arrays.

Step 5: Filtering Large/Small Nodes.
In this step we make an operator to process nodes of *ActiveNodes* array in parallel and fill both the *ChildNodesSize* array into which we store the size of each child node and the *Large* flags array into which we store 1 if the corresponding child node size is greater than triangles threshold *T*, and 0 otherwise (both arrays have length equals *2N*, where *N* is the length of *ActiveNodes* array). Given a parent node at index i in the *ActiveNodes* array we fill its left child node size at index $2i$ as *LeftScanTails[i]*, and fill its right child node size at index $2i + 1$ as *ActiveNodesSize[i] - LeftScanTails[i] + BothScanTails[i]*, we also fill the *Large* flags array by comparing the child node size against *T*.

We call the *Split* utility employing *Large* flags array as the *Flags* array to split *ChildNodes* into *NextNodes* and *SmallNodes*, and split *ChildNodesSize* into *NextNodesSize* and *SmallNodes-*

Size. Then we perform an exclusive scan for both *NextNodesSize* and *SmallNodesSize* arrays to get *NextNodesStart* and *SmallNodesStart* arrays respectively (see Figure 4.6). We update the *SmallNodesStart* array by adding to each element the number of previously stored small triangles in order to reflect the correct position of their triangles after appending new small triangles to previously stored ones. Then we append the *SmallNodes* SoA to the previously stored small nodes for later processing.

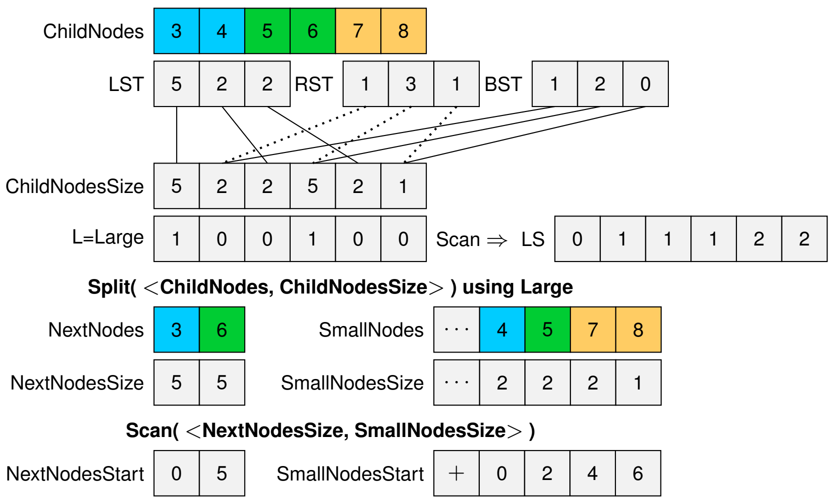


Figure 4.6: Large/Small nodes filtering, in this example we assume that the node thrshold T equals 4.

Step 6: Clipping and Distributing Triangles to Child Nodes.
Consider a triangle t at index i in a parent node N with a start index $Satrt_N$, and a size $Size_N$ which is split into a left child node N_L with a start index $Start_L$, and a right child node N_R with a start index $Start_R$. Using *Left*, and *Both* arrays a triangle t may be: (1) sorted to left child node only if *Left[i]* = *1*, and *Both[i]=0*; (2) sorted to right child node only if *Left[i]* = *0*, and *Both[i]=0*, or (3) clipped and sorted to both child nodes if *Left[i]* = *1*, and *Both[i]=1*. And the child node *(N_L, or N_R)* may be classified as a large node or a small node and accordingly its triangles must be stored in its corresponding triangles indices arrays.

To sort a triangle t we start by finding its new parent node N_L, and/or N_R, then we sort the triangle into the corresponding triangles array at an address defined by the child node start address plus a local offset calculated using the scans of *Left*, and *Both* arrays . We define 3 local offset addresses for a triangle t : (1) $Offset_L$ which represents the number of *ones* preceding it in *Left* array and is calculated as *Scan(Left)[i] - Scan(Left)[Srart$_N$]*; (2) *Offset$_R$* which represents the number of *zeros* preceding it in *Left* array and is calculated as *i - Start$_N$ - Offset$_L$*; and (3) *Offset$_B$* which represents the number of *ones* preceding it in *Both* array and is calculated as *Scan(Both)[i] - Scan(Both)[Srart$_N$]*.

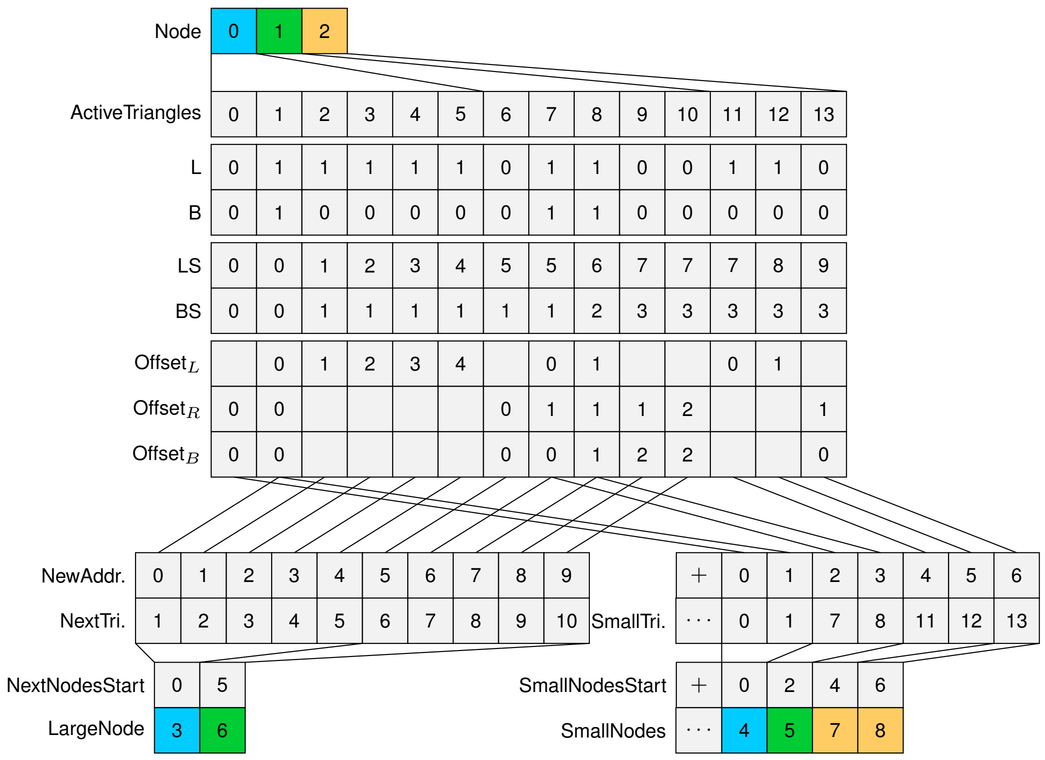

Node 0 1 2
ActiveTriangles 0 1 2 3 4 5 6 7 8 9 10 11 12 13
L 0 1 1 1 1 1 0 1 1 0 0 1 1 0
B 0 1 0 0 0 0 0 1 1 0 0 0 0 0
LS 0 0 1 2 3 4 5 5 6 7 7 7 8 9
BS 0 0 1 1 1 1 1 1 2 3 3 3 3 3
Offset$_L$ 0 1 2 3 4 0 1 0 1
Offset$_R$ 0 0 0 1 1 1 2 1
Offset$_B$ 0 0 0 0 1 2 2 0
NewAddr. 0 1 2 3 4 5 6 7 8 9
NextTri. 1 2 3 4 5 6 7 8 9 10
+ 0 1 2 3 4 5 6
SmallTri. · · · 0 1 7 8 11 12 13
NextNodesStart 0 5
LargeNode 3 6
SmallNodesStart + 0 2 4 6
SmallNodes · · · 4 5 7 8

Figure 4.7: Clipping and distributing triangles to child nodes.

We distinguish 8 distinct cases for a triangle distributing (see Figure 4.7):

1. Triangle *t* goes to the left child node N_L only where:

   (a) N_L is a large node; then we store *t* at index *Start$_L$* + *Offset$_L$* in the *NextTraingles* array.

   (b) N_L is a small node; then we store *t* at index *Start$_L$* + *Offset$_L$* in the *SmallTraingles* array.

2. Triangle *t* goes to the right child node N_R only where:

(a) N_R is a large node; then we store *t* at index *Start*$_R$ + *Offset*$_R$ + *Offset*$_B$ in the *NextTraingles* array.

(b) N_R is a small node; then we store *t* at index *Start*$_R$ + *Offset*$_R$ + *Offset*$_B$ in the *SmallTraingles* array.

3. Triangle *t* goes to both child nodes N_L*, and* N_R where:

(a) N_L is a large node and N_R is a large node; then we clip and store *t* at indices *Start*$_L$ + *Offset*$_L$ and *Start*$_R$ + *Offset*$_R$ + *Offset*$_B$ in the *NextTraingles* array.

(b) N_L is a large node and N_R is a small node; then we clip and store *t* at index *Start*$_L$ + *Offset*$_L$ in the *NextTraingles* array and at index *Start*$_R$ + *Offset*$_R$ + *Offset*$_B$ in the *SmallTraingles* array.

(c) N_L is a small node and N_R is a large node; then we clip and store *t* at index *Start*$_L$ + *Offset*$_L$ in the *SmallTraingles* array and at index *Start*$_R$ + *Offset*$_R$ + *Offset*$_B$ in the *NextTraingles* array.

(d) N_L is a small node and N_R is a small node; then we clip and store *t* at indices *Start*$_L$+ *Offset*$_L$ and *Start*$_R$ + *Offset*$_R$ + *Offset*$_B$ in the *SmallTraingles* array.

Similar to triangles sorting we create an operator which handle the previously stated cases for a triangle and distribute all triangles of a single chunk in parallel, and then we make a nested parallel call to this operator to distribute triangles of all chunks.

After finishing this step we check if we still have large nodes in the *NextNodes* array then we swap *NextNodes* SoA and *ActiveNodes* SoA and their triangles association arrays and return to step 1 of the large node stage, otherwise we go to the small node stage after swapping *SmallNodes* SoA and *ActiveNodes* SoA and their triangles association arrays and proceed to next stage.

4.4.2 Small Node Stage

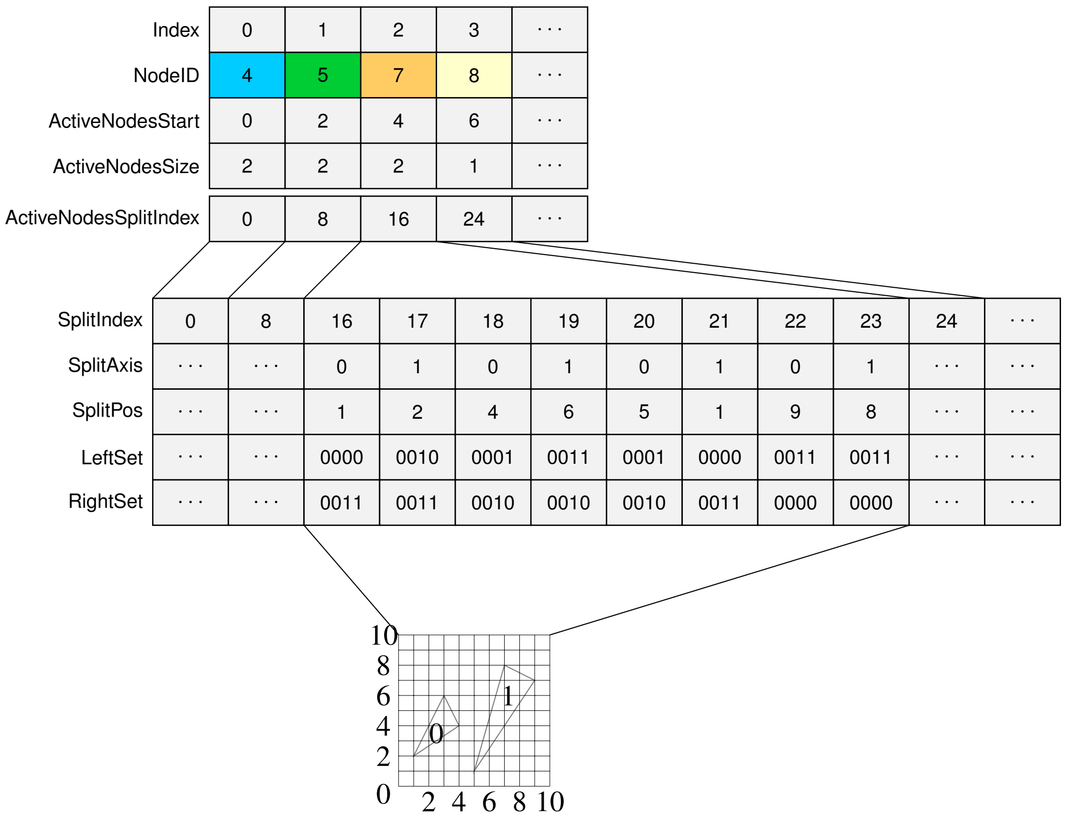
In this stage we still process nodes in BFS order but we employ an approximate SAH without triangles clipping as cost estimation for node splits. We begin with a preprocessing step in which we enumerate all split candidates in each node. Then we process nodes in two main steps: (1) evaluate the SAH at all split candidates in each node, then select the minimum SAH using standard reduction and compare the minimum SAH with the cost of not splitting the node to either split the node or mark it as a leaf; (2) for each split node we sort triangles to child nodes. We return to step 1 of the small stage if we still have new child nodes.

Preprocessing Small Nodes.

In this step we prepare small roots which correspond to all small nodes with large parent nodes. Each small root defines the start triangle address index in the *ActiveTriangles* array and the number of the triangles N_t (where $0 < N_t \leq T$) and lists all split candidates defined by the 6 side planes of each triangle AABB. The split candidate is defined by split axis, split position, left and right triangles sets where each set is represented using a *T* bits mask [Zhou et al., 2008].

To create small roots we prepare struct of array (SoA) for splits which consists of three arrays: *SplitPosition* in which we store the split positions, *LeftSet* in which we store the *T* bits masks of the left triangles sets, and *RightSet* in which we store the *T* bits masks of the right triangles sets and each of these arrays are of size equals $6N$ where N is the total number of triangles in all small nodes.

We make an operator that processes the triangles of each small node in parallel. In this operator, given a triangle t at index i we store its 6 side positions, left, and right triangles sets staring at index $6i$ into arrays *SplitPosition*, *LeftSet*, and *RightSet* respectively. The left and right triangles sets are created by comparing the candidate position against all other triangles in the node. We make a nested parallel call to this operator to prepare the split candidates for all small roots (see Figure 4.8).



Index	0	1	2	3	···
NodeID	4	5	7	8	···
ActiveNodesStart	0	2	4	6	···
ActiveNodesSize	2	2	2	1	···
ActiveNodesSplitIndex	0	8	16	24	···

SplitIndex	0	8	16	17	18	19	20	21	22	23	24	···
SplitAxis	···	···	0	1	0	1	0	1	0	1	···	···
SplitPos	···	···	1	2	4	6	5	1	9	8	···	···
LeftSet	···	···	0000	0010	0001	0011	0001	0000	0011	0011	···	···
RightSet	···	···	0011	0011	0010	0010	0010	0011	0000	0000	···	···

Figure 4.8: Small roots and related splits.

We modify the small nodes structure a little bit (see Figure 4.9). Instead of defining the triangles of a node by the start triangle index and number of triangles we define them by an index to the small root and a T bits mask *NodeTrianglesSet* representing the triangles set in the node relative to small root start triangle index, for a node with N_t triangles this mask is initialized to $(1 << N_t) - 1$.

Index	0	1	2	3
NodeID	4	5	7	8
SmallRootIndex	0	1	2	3
NodeTrianglesSet	0011	0011	0011	0001

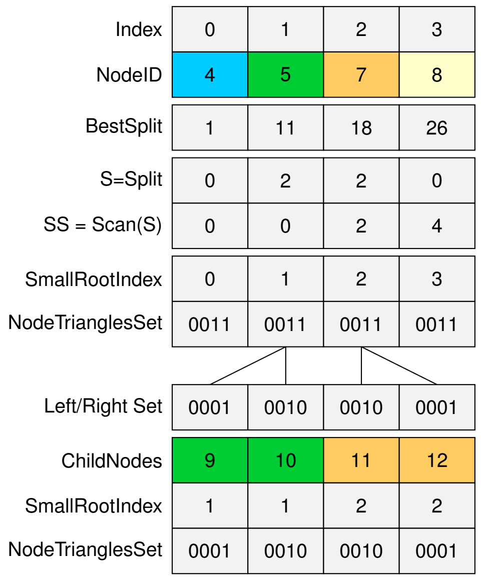
Figure 4.9: Small nodes modified structure, small node size is at most 4 primitives.

Step 1: Evaluating Node SAH Cost.

We make an operator that processes *6T* split candidates (s_i) for each small node in parallel where ($s_i \in [0, 6T)$). In this operator we get the node's triangles set mask from the *(NodeTrianglesSet)* array and if a split s_i exist in the triangle set (bit $s_i/6$ is set to 1 in the mask) then we get split position, left triangle set mask *(LeftSet)* and right triangle set mask *(RightSet)* form the small root, the split axis is defined implicitly by the split index $s_i\%3$, and evaluate the SAH cost as:

$$SAH_{si} = C_T + \frac{C_I}{SA_N}(N_L(s_i)SA_L(s_i) + N_R(s_i)SA_R(s_i))$$

Where C_T is the cost of node traversal, C_I is the cost of triangle intersection, $N_L(s_i), N_R(s_i)$ are the number of triangles lying to the left and to the right of the split respectively; $N_L(s_i)$ is calculated as the bit count of the *LeftSet anded* with the *NodeTrianglesSet* and $N_R(s_i)$ is calculated as the bit count of the *RightSet anded* with the *NodeTrianglesSet*, SA_N is the surface area of node AABB, and $SA_L(s_i), SA_R(s_i)$ are the surface area of the child nodes AABBs resulting from splitting the parent node AABB with the plane defined by the split candidate. We make a nested parallel call to this operator to process all nodes in parallel. We also customize the reduction utility to reduce the $6T$ values of each node and return the minimum SAH value and its corresponding split index, and make a nested parallel call to this utility to select the minimums SAH cost for each node.



Index	0	1	2	3
NodeID	4	5	7	8
BestSplit	1	11	18	26
S=Split	0	2	2	0
SS = Scan(S)	0	0	2	4
SmallRootIndex	0	1	2	3
NodeTrianglesSet	0011	0011	0011	0011
Left/Right Set	0001	0010	0010	0001
ChildNodes	9	10	11	12
SmallRootIndex	1	1	2	2
NodeTrianglesSet	0001	0010	0010	0001

Figure 4.10: Small nodes splitting.

We prepare an array for *Split* flags equal in size to *ActiveNodes* array, into this array we store 2 if the corresponding node will be split and 0 otherwise. Once we calculated the minimum SAH for each node; we compare it with the cost of not splitting the node and store the flags into *Split* array. Then we perform an exclusive scan to this array to use it in creating new child node.

Step 2: Node Split and Triangles Sorting.
This step is relatively simple; we make an operator that processes all node in parallel and create child nodes. In this operator, given a node at index i in the *ActiveNodes* we check the flag in the *Split* flags array and if it is set then we use the minimum cost split index calculated in the previous step to get split position, axis, *LeftSet* and *RightSet* form the small roots arrays and create the two child nodes starting at index *Scan(Split)[i]*. For the child nodes we set the *NodeRootIndex* as of the parent node value, and set the triangles set of the left child node as the bitwise *AND* between *LeftSet* and *NodeTrianglesSet* and set the triangles set of the right child node as the bitwise *AND* between *RightSet* and *NodeTrianglesSet* and split the parent node AABB at the split plane into two AABBs for the left and right child nodes, we also mark the parent node as an internal node and set its child references to refer to the newly created child nodes (see Figure 4.10).

Finally, we use the new child nodes as the active nodes for the next step; if we have no child node then the tree construction is complete.

4.5 Parallel SAH BVH Construction

In [Lauterbach et al., 2009] Lauterbach et al. presented an efficient parallel algorithm for constructing BVH in BFS order using approximate SAH cost. This algorithm uses two different split strategies for large and small nodes splits that we refer as the small and large node stages in this section. Once all large nodes are split; the algorithm runs a special kernel that processes all small roots and makes use of local processor cache (i.e. shared memory) and wide SIMD instruction set to creates a complete sub-tree under each small root using full sweep SAH cost evaluation.

4.5.1 Large Node Stage

This stage begins with the root node and processes all large nodes in five main steps: (1) evaluates the SAH cost at all split candidates in each node and select the minimum SAH using standard reduction; (2) compare the minimum SAH with the cost of not splitting the node and either we split the node or mark it as a leaf node; (3) sort triangles to child nodes; (4) calculate child nodes size and filter them into large and small nodes; (5) distribute triangles to child nodes, and either we return to step 1 if we still have large nodes or go to next stage if not.

Step 1: Evaluating Nodes SAH Cost.
In this step we evaluate an approximate SAH cost at k (e.g. 64) uniformly sampled split positions inside node's AABB in each of the three axes. We make an operator that evaluates each of the $3k$ splits for each node in parallel employing fast shared memory of GPU to load primitive's data, and make a nested parallel call to this operator to process all nodes. Then we customize the reduction utility to work on a single node and select the minimum SAH and the corresponding split candidate from the $3k$ values, and make a nested parallel call to this utility to process all nodes in parallel. We prepare another operator that works for all nodes and compares the minimum SAH with the cost of not splitting the node. If the minimum SAH cost is lower than the leaf cost; then we store 1 at the corresponding node index in the *Split* flags array, otherwise we store 0 to indicate that this node will be a leaf in the hierarchy.

Step 2: Splitting Nodes.
We scan the *Split* flags array and use it to split the *ActiveNodes* and *ActiveNodesSize* arrays and get the right side of the split operation in the *LeafNodes* and *LeafNodesSize* arrays respectively. We scan the *LeafNodesSize* into *LeafNodesStart* array, and update the *LeafNodesStart* array by adding to each element the number of previously stored leaf triangles in order to reflect the correct position of their triangles after appending new leaf triangles to previously stored ones. Then we append the *LeafNodes* SoA to the previously stores leaf nodes. To create new child nodes we create an operator that check for each node at index i the corresponding value in the *Split* array if it is set; then we create two child nodes in the *ChildNodes* array starting at address $2Scan(Split)[i]$ (see Figure 4.11).

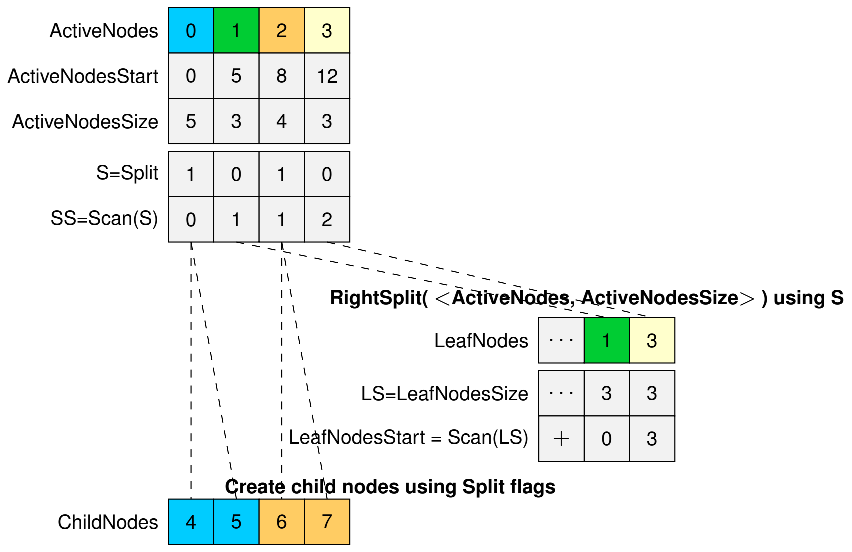


Figure 4.11: Splitting BVH nodes.

Step 3: Sorting Triangles to Child Nodes.

To parallelize this step over triangles we use the indirection array *(ActiveParents)* (see Section 4.3) and prepare an operator that can sort all triangles in parallel by filling *Left* flags array at which we store 1 at the corresponding triangle index if the triangle classified to left child node and store 0 otherwise. In this operator we check the value at the parent node index in the *Split* flags array; if it is 1 then the node is split and we have to compare the triangle centriod against the split plane and sore 1 in the *Left* flags array if the triangle lies to the left of the split plane or 0 otherwise. We scan the *Left* flags array and call the *SegScanTails* utility employing *ActiveNodesStart* array as the *Start* argument and *ActiveNodesSize* array as the *Size* argument and store the results into *LeftScanTails* array (see Figure 4.12).

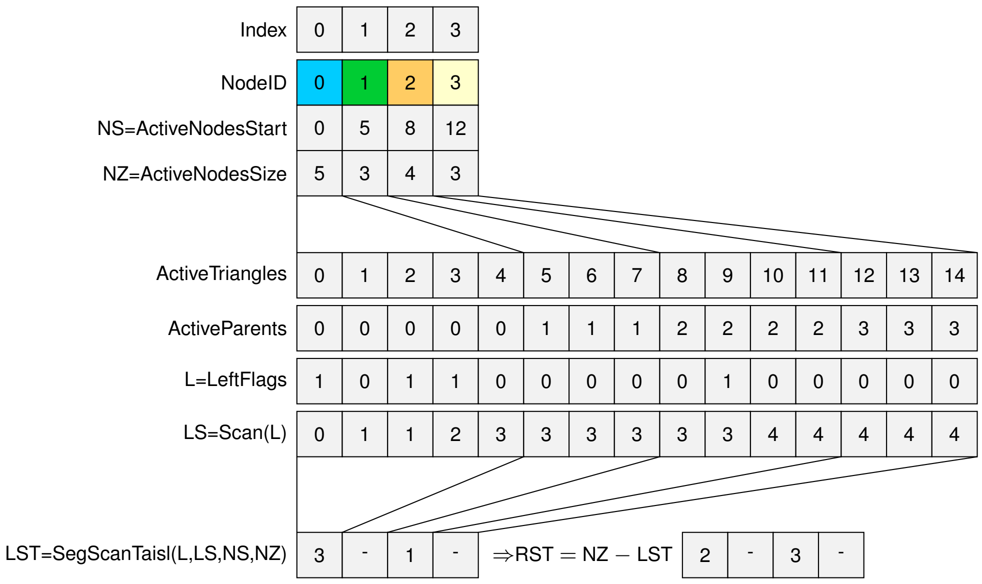


Figure 4.12: Sorting BVH triangles to child nodes.

Step 4: Filtering Large/Small Nodes.

In this step we make an operator to process *ActiveNodes* in parallel and fills two arrays; *ChildNodesSize* array which store the size of each child node, and *Large* flags array into which we store 1 if the child node size is greater than *T*, and 0 otherwise. Given a node *N* at index *i* with a size $Size_N$, we check the corresponding index in the *Split* flags array; if it is set then we store *LeftScanTails*[i] as the left child node size at index $2Scan(Split)[i]$ in the *ChildNodesSize* array, and store $Size_N$ - *LeftScanTails*[i] as the right child node size at index $2Scan(Split)[i]+1$ in the *ChildNodesSize* array. We fill the corresponding indices in the *Large* flags array by comparing child node size against *T*.

We call the split utility employing *Large* flags array and its scan to split *ChildNodes* into *NextNodes* and *SmallNodes*, and to split *ChildNodesSize* into *NextNodesSize* and *SmallNodesSize*. Then we perform an exclusive scan on *NextNodesSize* to get *NextNodesStart* and on *SmallNodesSize* to get *SmallNodesStart* (see Figure 4.13). We update the *SmallNodesStart* array by adding to each element the number of previously stored small triangles in order to reflect the correct position of their triangles after appending new small triangles to previously stored ones. Then we append the *SmallNodes* SoA to the previously stored small nodes for later processing.

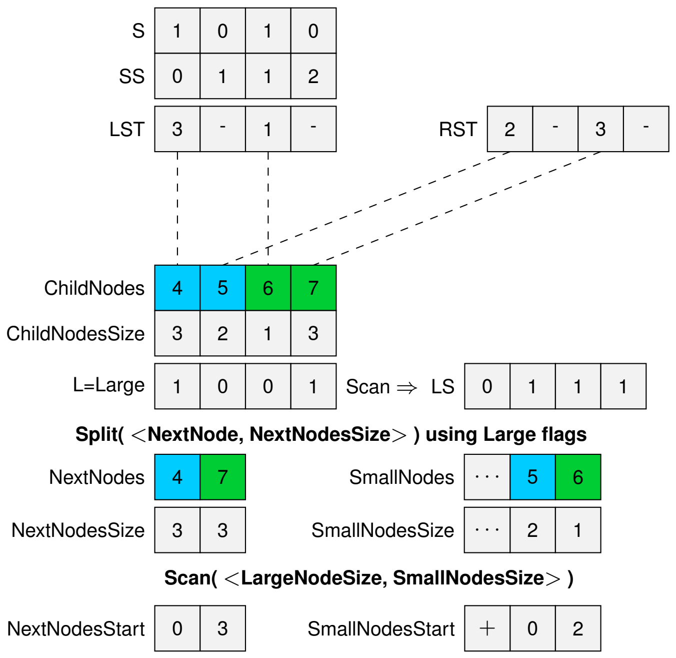


Figure 4.13: Filtering large/Small nodes.

Distributing Triangles to (Child) Nodes.

To sort a triangle we need to know its (new) parent node and the corresponding association list and store it at an address defined by the node start address and a local offset inside this node. Similar to the triangles sorting we make an operator that distributes all triangles in parallel. Consider a triangle *t* at index i in a parent node *N* at index N_i with a start index $Start_N$ and a size $Size_N$. We define 3 local offset addresses for a triangle *t* : (1) the number of *ones* preceding it in *Left* array *Offset*$_L$ which is calculated as *Scan(Left)[i] - Scan(Left)[Srart*$_N$*]*; (2) the number of *zeros* preceding it in *Left* array *Offset*$_R$ which is calculated as *i - Start*$_N$ *- Offset*$_L$; and (3) the relative position inside its old parent node *Offset*$_N$ which is calculated as *i - Srart*$_N$. Using the *Split* flags array a triangle *t* may be sorted to: (1) a leaf node *(Splits[*N_i*] = 0)*; or (2) goes to a new child node *(Splits[*N_i*] = 1)*, hence, if the triangle is to be stored in a new child node we use the *Left* array to determine whether it will be sorted to: (1) left node child *(Left[i] = 1)*; or (2) right child node *(Left[i] = 0)*.

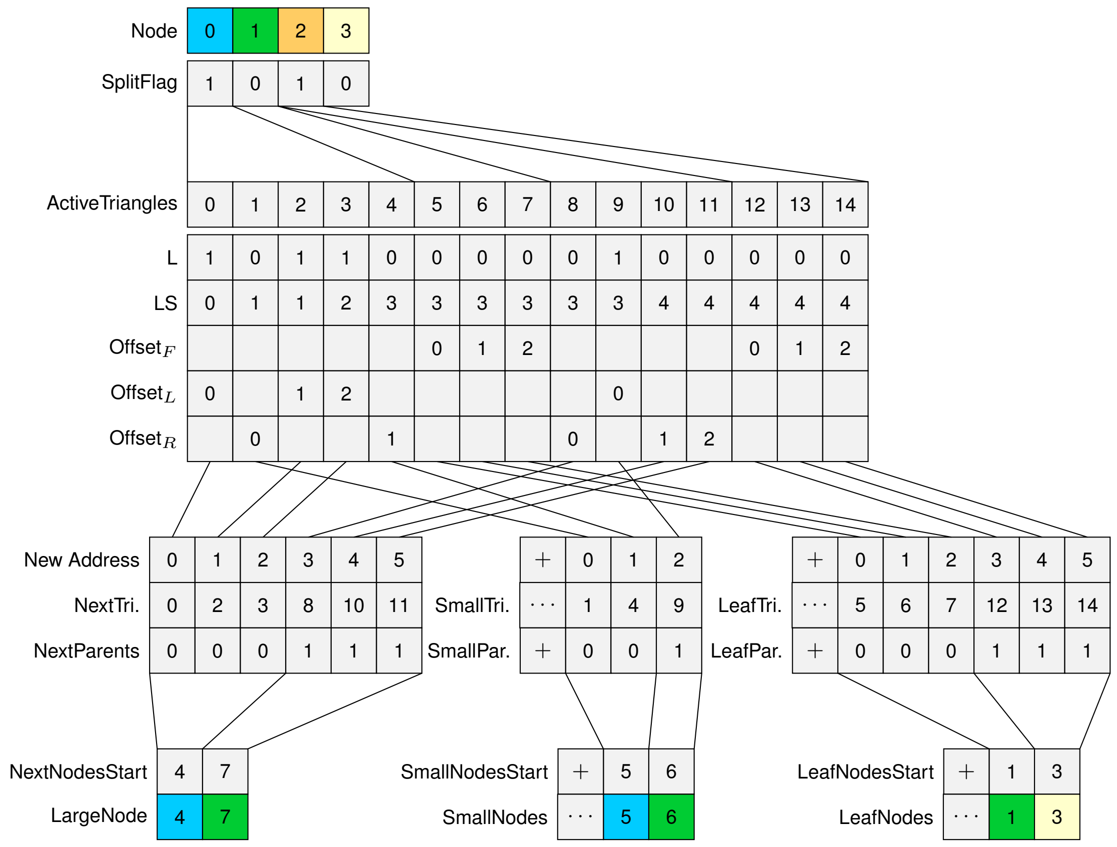


Figure 4.14: Distributing triangles to (child) nodes.

We distinguish 5 distinct cases for a triangle (see Figure 4.14):

1. Triangle *t* goes to leaf node, then we store *t* at index $Start_F + Offset_N$ in the *Leaf-Traingles* array, where $Start_F$ is the start index of the leaf node.

2. Triangle *t* goes to the left child node N_L where:

   (a) N_L is a large node; then we store *t* at index $Start_L + Offset_L$ in the *NextTraingles* array.

   (b) N_L is a small node; then we store *t* at index $Start_L + Offset_L$ in the *SmallTraingles* array.

3. Triangle *t* goes to the right child node N_R where:

   (a) N_R is a large node; then we store *t* at index $Start_R + Offset_R$ in the *NextTraingles* array.

(b) N_R is a small node; then we store *t* at index $Start_R + Offset_R$ in the *SmallTrain-gles* array.

Similar to triangles sorting we create an operator that processes all triangles in parallel. In this operator we just handle the previously stated cases for a triangle and update the triangles parents arrays accordingly. After finishing this step we check if we still have large nodes in the *NextNodes* array then we swap the *NextNodes* and *ActiveNodes* SoAs and their triangles association arrays and return to step 1 of the large node stage, otherwise we go to the small node stage after swapping *SmallNodes* SoA and *ActiveNodes* SoAs.

4.5.2 Small Node Stage

In this stage we split all small nodes with size at most *T* (e.g. 32 threads). We make an operator that takes a small node and constructs a complete sub-tree rooted at this small node using full sweep SAH cost in breadth first search (BFS) order. In this operate we maintain a local shared queue for non-split tree nodes which is initialized with a small root. As long as we still have nodes in the local queue we pop a tree node *N* form the queue, for a node with size *k* we let each thread *i* evaluates the 3 SAH cost values for each axis at primitve *i* centroid and then we select the minimum of them of these three values. Then we select the minimum SAH value and corresponding split plane form the *k* values using the standard reduction and compare this cost by the cost of not splitting the node; if the minimum SAH cost is less than the leaf cost then we split the node by local thread 0, and sort and distribute node's triangles in a way similar to the large node stage using the k threads, and push the two child nodes in the local nodes queue by local thread 0. We call this operator with *T* threads for each small root and make a nested call for this operate to process all small roots in parallel.

In the small stage we have to make use of the effective register SIMD (i.e. warp size of 32 threads) to evaluate the SAH cost in parallel and make use of local shared memory to load primitives data in local cache.

4.6 Proposed Parallel Algorithm for Building Binned SAH BVH

In this section we explain the main processing steps of our binned SAH BVH algorithm. The algorithm consists of n processing stages, where each stage differs only in the nodes size and accordingly the number of bins for SAH cost evaluation. In each stages we perform 5 main processing iterative steps: (1) project triangles into K bins; (2) relocate node's triangles into contiguous sets according to bin number; (3) divide each bin into Fixed-Sized chunks of triangles and calculate the AABB for each chuck using standard reduction and the bins AABB using segmented reduction on chunks data; (4) evaluate the SAH at $K - 1$ split candidates in each node using data parallel primitives on chunks data then we either split the node or mark it as a leaf, and filter new child nodes into large nodes belong to current stage and small nodes belong to further stages; (5) sort triangles to their (new) parent nodes. After finishing each iteration we return to step 1 if we still have nodes for current stage, otherwise we filter small nodes to these ready for next stage and those belong to further stages and advance processing to next stage.

Step 1: Projecting Triangles into Bins.

For each triangle *t* we calculate its bin number b_t using the formula [Wald, 2007]:

$$b_t = \frac{K(1-\epsilon)(c_{t,a} - cb_{min,a})}{(cb_{max,a} - cb_{min,a})}$$

Where *K* is the number of bins, c_t is the triangle centroid, *cb* is the the centroid bounds, and *a* is the binning axis (i.e. x, y, or z). We prepare *K* arrays for bin flags (each array of length equals the number of triangles in the association array). Then we make an operator that projects all triangles to their bins in parallel. In this operator, given a triangle *t* at index *i* which is projected to bin *k* we store 1 at index *i* into bin flags array number *k* and store 0 at this index in all other *K-1* arrays. Once we projected all triangles we scan all the bin flags arrays and get the scan tails of each array using the *SegScanTails* utility employing nodes as the scan segments (see Figure 4.15).

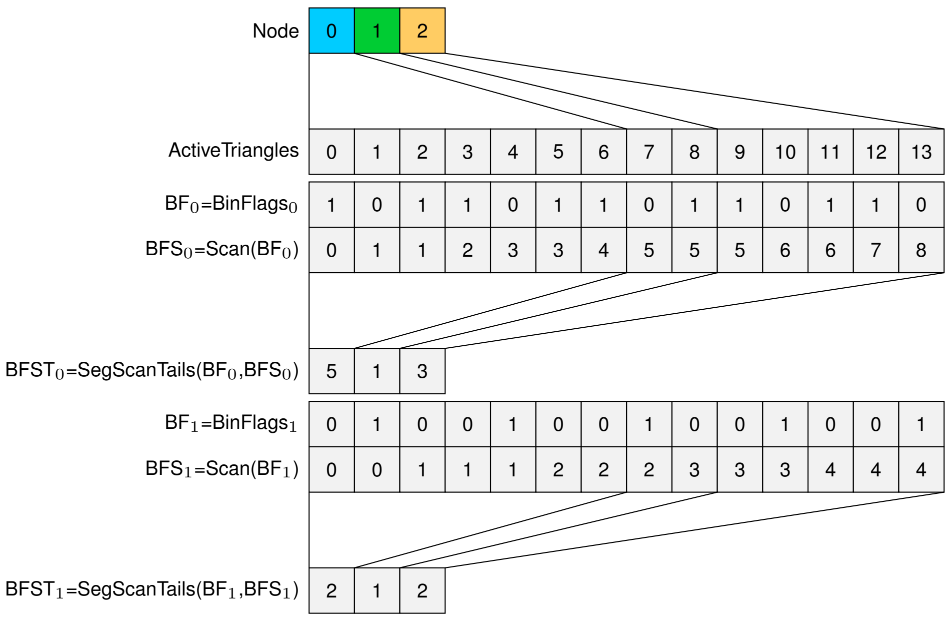


Figure 4.15: Projection triangles into corresponing bins.

We store all the scan tails into array *BinSize*, this array store the number of triangles projected into each bin stored by node index and has length equals *NK* where *N* is the current number of active nodes. Then we scan the *BinSize* array to use it to sort node's triangles into contiguous blocks sorted by bin index(see Figure 4.16).

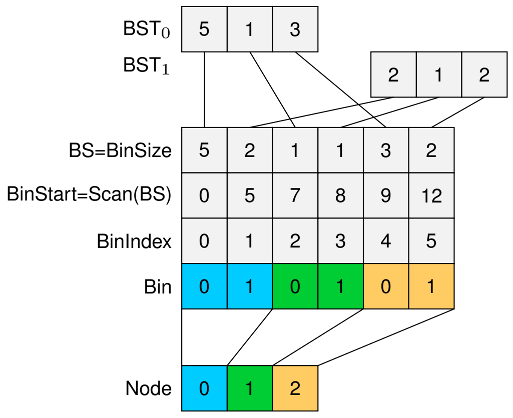


Figure 4.16: Bins SoA calculation.

Step 2: Sorting Triangles to their Bins.

We make an operator that sorts all triangles into a contiguous sets sorted according to bin number in each node. In this operator given a triangle t at index i which is projected into bin k and belongs to a parent node N at index N_i with a start address $Start_N$; we get triangle start bin address as $Scan(BinSize)[N_i \times K + k]$, and get local triangle offset in this bin as $Scan(BinFlags_k)[i]$ - $Scan(BinFlags_k)[Start_N]$, then we store the triangle at an index defined by its bin start address plus its local offset (see Figure 4.17).

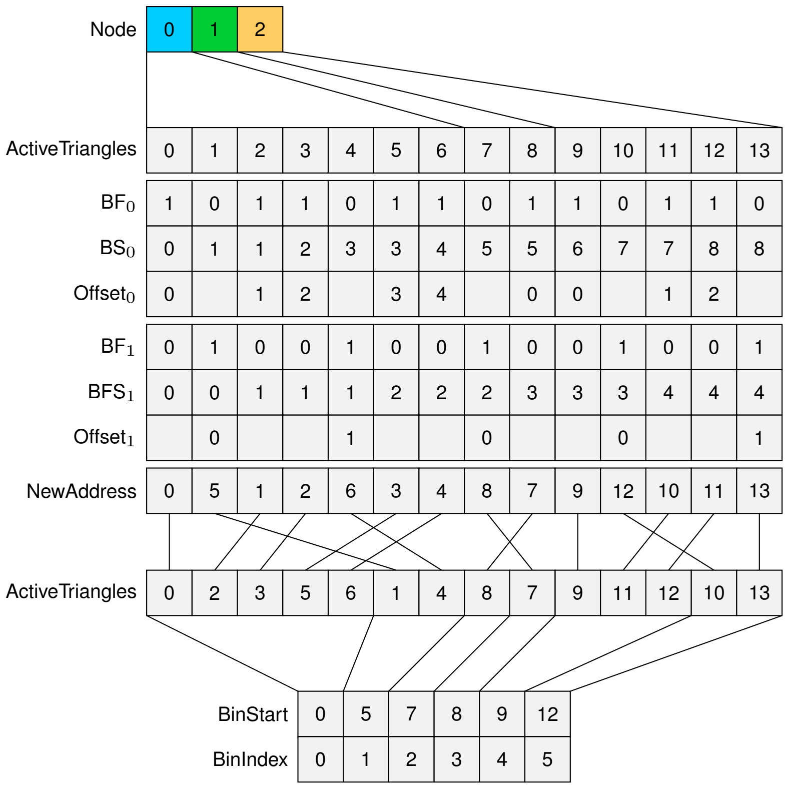


Figure 4.17: Sorting triangles locally to their respective bins.

Step 2: Calculating per Bin AABB

We divide each bin in each node to Fixed-Sized chunks of triangles. First, we call the *CreateChunks* utility to fill the *ChunksStart* and *ChunksSize*, and *Owner* arrays employing *BinStart* and *BinSize* arrays as the *Start* and *Size* arguments respectively. Then we calculate each chunks AABB using standard reduction and bins AABB using segmented reduction in a way similar to the KD-tree algorithm.

In the last processing stage in which we process internal nodes with the smallest size we avoid the use of chunks data structures and perform the segmented reduction process directly on the triangles array. First, we pass on all triangles and make the *Owner* array that map each triangle into its bin; for a triangle *t* at index *i* in a parent node at index N_i which is projected to bin k_t we fill its owner as $N_iK + k_t$.

Step 4: Splitting Nodes

We compute for each node the minimum split cost by evaluating the SAH cost at *K-1* uniformly sampled positions in the longest node side. This step is relatively simple as the computation is parallelized over nodes. For each node we run a left and right scans for both bins' AABBs, and evaluate the *K-1* SAH values in parallel and choose the lowest bin cost using standard reduction in each node. We compare this cost with the cost of not splitting the node and if the minimum SAH cost is lower we store 1 in *Split* flags array, otherwise we store 0.

Then we scan the *Split* flags array and employ it to split the *Nodes* and *NodesSize* arrays and the get the right side of the split operation as the *LeafNodes* and *LeafNodesSize* arrays respectively. We scan the *LeafNodesSize* array to get the *LeafNodesStart* array (see Figure 4.18). We update the *LeafNodesStart* array by adding to each element the number of previously stored leaf triangles in order to reflect the correct position of their triangles after appending new leaf triangles to previously stored ones. Then we append the *LeafNodes* SoA to the previously stores leaf nodes.

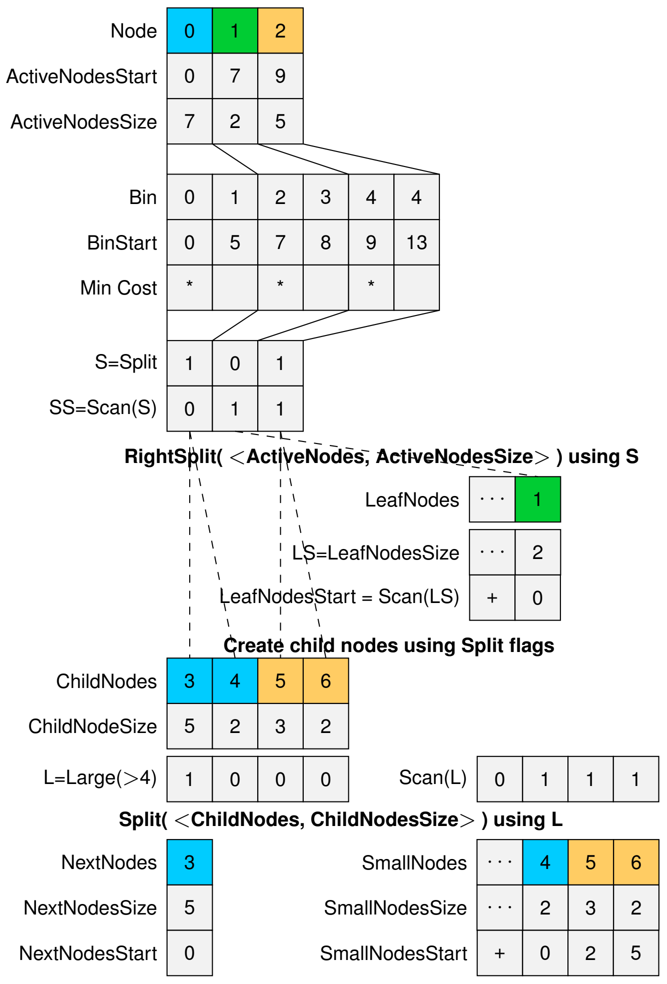


Figure 4.18: Splitting BVH Node.

Then we make an operator to create new child nodes. This operator processes all active node in parallel, for each node we check the split flag; if it is set then we get the left and right child data (AABB and triangles count) from the best bin and store them at indices *2Scan(Flags[i])*, and *2Scan(Flags[i])* + 1 respectively. We also fill an array for *Large* flags array into which store 1 if the child node passes the current stage threshold and 0 otherwise. We scan this array and use it in splitting child nodes into large nodes ready for the next step and small node which are stored for later processing.

Step 5: Distributing triangles to (Child) Nodes

We make an operator that sorts all triangles in parallel. To sort a triangle *t* at index *i* we begin by finding its parent node; if the parent node is not split then we sort it at index defined by the leaf node start plus its local offset in the parent node, and if it is split then we find its new parent node and its start address in the corresponding association list and sort it at an index defined by child node address plus its local offset in the child node. For a triangle *t* at index *i* in the triangles array and which belongs to node *N* that has a start address *Start*$_N$ we define the local offset (*Offset*$_t$) of *t* as *i-Start*$_N$. If the node si not split we define its new start address in the *LeafNodesStart* array as *Start*$_F$, and if it is split we define its two child nodes start addresses as textitStart$_L$ and textitStart$_R$ for left and right child nodes respectively. If triangle *t* goes to left child node we define its local offset in the left child node as *Offset*$_L$ which si calculated as *i-Start*$_N$, and if it goes to right child node we define its local offset in the right child node as *Offset*$_R$ which is calculated as *i-Split*$_N$, where *Split*$_N$ is the first triangle index best bin at which we split the node.

We distinguish 5 distinct cases for a triangle sorting (see Figure 4.19):

1. Triangle *t* goes to leaf node, then we store *t* at index $Start_F + Offset_t$ in the *LeafNodesTraingles* array, where $Start_F$ is the start index of the leaf node.

2. Triangle *t* goes to the left child node *(*N_L*)* where:

   (a) N_L is a large node; then we store *t* at index $Start_L + Offset_L$ in the *NextNodesTraingles* array.

   (b) N_L is a small node; then we store *t* at index $Start_L + Offset_L$ in the *SmallNodesTraingles* array.

3. Triangle *t* goes to the right child node *(*N_R*)* where:

   (a) N_R is a large node; then we store *t* at index $Start_R + Offset_R$ in the *NextNodesTraingles* array.

   (b) N_R is a small node; then we store *t* at index $Start_R + Offset_R$ in the *SmallNodesTraingles* array.

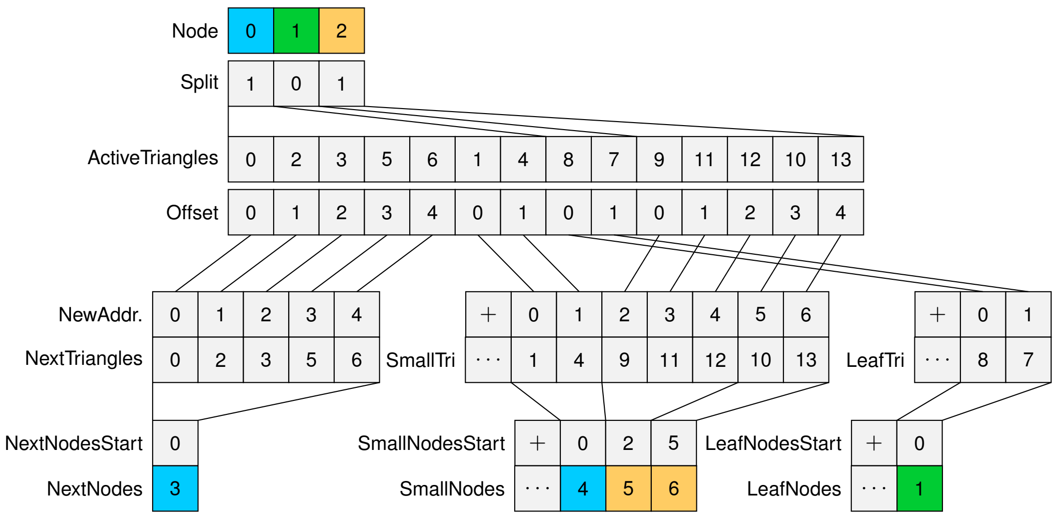


Figure 4.19: Distributing triangles to (child) nodes

4.6.1 Filtering Next/Further Nodes

After finishing the current stage we filter small nodes and their corresponding triangle into nodes for the next stage and nodes for further stages. This operation can be done easily by first filtering the nodes using a *Large* flags array into which we store 1 in the corresponding element if the node pass the next stage threshold and 0 otherwise. Then we split the *SmallNodes* array using the *Large* flags array into *NextNodes* and *FurtherNodes*, we also apply the same split operation to the *SmallNodeSize* to create *NextNodesSize* and *FurtherNodesSize* arrays, then we scan the *NextNodesSize* and *FurtherNodesSize* into *NextNodesStart* and *FurtherNodeStart* arrays respectively.

Triangles are also sorted into two association arrays (*NextNodeTriangles* and *FurtherNodeTriangles*), given a triangle *t* at index i in a parent node *N* with a start index $Satrt_N$, we store the triangle at an address defined by its new parent address $Satrt_{New}$ plus a local triangle offset in *N* which is calculated as *i-Start*$_N$ (see Figure 4.20).

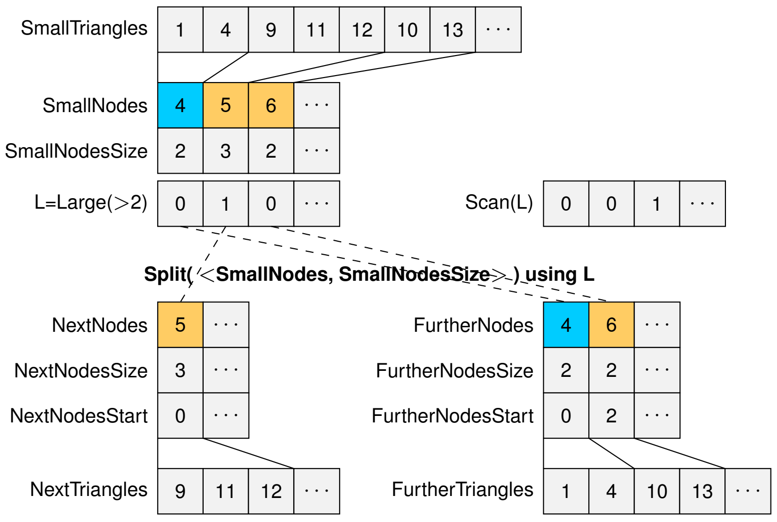


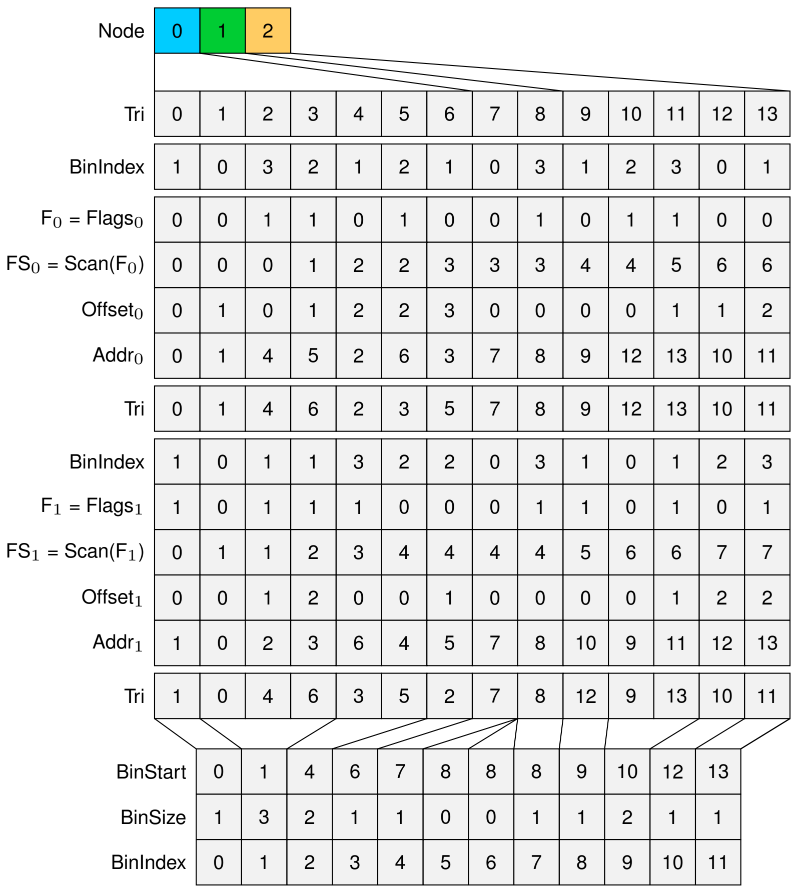
Figure 4.20: Filtering Next/Further nodes, in this example we use node Next/Further threshold equals 2.

We consider *NextNodes* SoA as the *ActiveNodes* SoA for the new stage and keep *FurtherNodes* SoA as the *SmallNodes* SoA for further processing stages.

4.6.2 Modifications and Extensions

4.6.2.1 Reducing Scan Passes for Triangles Projection.

During triangles projection we apply K simultaneous scans on K arrays to perform a segmented K split of the triangles array. However, we can perform this split operation by applying $\log_2(K)$ sequential scans on a single array of flags [Sengupta et al., 2007]. We begin by projecting each triangle t into corresponding bin b_t, then we perform a segmented split operation $\log_2(K)$ times employing the bits of bin index (b_t) as the split flags starting form the most to the least significant bit (see Figure 4.21).



Node	0	1	2											
Tri	0	1	2	3	4	5	6	7	8	9	10	11	12	13
BinIndex	1	0	3	2	1	2	1	0	3	1	2	3	0	1
F_0 = Flags_0	0	0	1	1	0	1	0	0	1	0	1	1	0	0
FS_0 = Scan(F_0)	0	0	0	1	2	2	3	3	3	4	4	5	6	6
Offset_0	0	1	0	1	2	2	3	0	0	0	0	1	1	2
Addr_0	0	1	4	5	2	6	3	7	8	9	12	13	10	11
Tri	0	1	4	6	2	3	5	7	8	9	12	13	10	11
BinIndex	1	0	1	1	3	2	2	0	3	1	0	1	2	3
F_1 = Flags_1	1	0	1	1	1	0	0	0	1	1	0	1	0	1
FS_1 = Scan(F_1)	0	1	1	2	3	4	4	4	4	5	6	6	7	7
Offset_1	0	0	1	2	0	0	1	0	0	0	0	1	2	2
Addr_1	1	0	2	3	6	4	5	7	8	10	9	11	12	13
Tri	1	0	4	6	3	5	2	7	8	12	9	13	10	11

BinStart	0	1	4	6	7	8	8	8	9	10	12	13
BinSize	1	3	2	1	1	0	0	1	1	2	1	1
BinIndex	0	1	2	3	4	5	6	7	8	9	10	11

Figure 4.21: Sorting triangles into 4 bins using 2 sequential scans

This split operation should be stable which means that splits on lower bits must respect the order produced by higher bits and the most significant bit split must respect the node partitions. We achieve this stable segmented split by incrementally doubling the node partitions after each bit split and calculating triangle offset needed for primitive relocation locally as the difference between the corresponding index in the scanned flags array and the start index of the owner partitions of the previous partition. The drawback of this method is the excessive memory bandwidth consumed by the consecutive $\log_2(K)$ primitives relocations, however form our experiments we found that the net enhancement of the construction time is about 1 order of magnitude compared to original method.

4.6.2.2 Projecting Triangles using Parallel Sorting.

Since the $\log_2(K)$ sequential scans is analogue to $\log_2(K)$ stable radix sort, we can avoid these scans and the memory bandwidth overhead incurred by primitives relocating using a single radix sort operation [Satish, Harris, and Garland, 2009b]. For each triangle *t* we make a binary code consisting of the parent node index in the highest significant bits and the bin index (b_t) in the lowest $\log_2(K)$ significant bits, then we perform parallel sorting on these codes which then reflect the correct primitive projection order (see Figure 4.22). We fill the *BinStart* array by examining the sorted pattern of the bin indices and filling the array upon finding two consecutive different bin indices, this operation is easily done by creating an operator that test every primitive in parallel. For empty bins we just interpolate the missing values in the *BinStart* array from neighboring elements using an operator that processes nodes in parallel, and the *BinSize* array is filled by the difference between each two neighboring elements in the *BinStart* array.

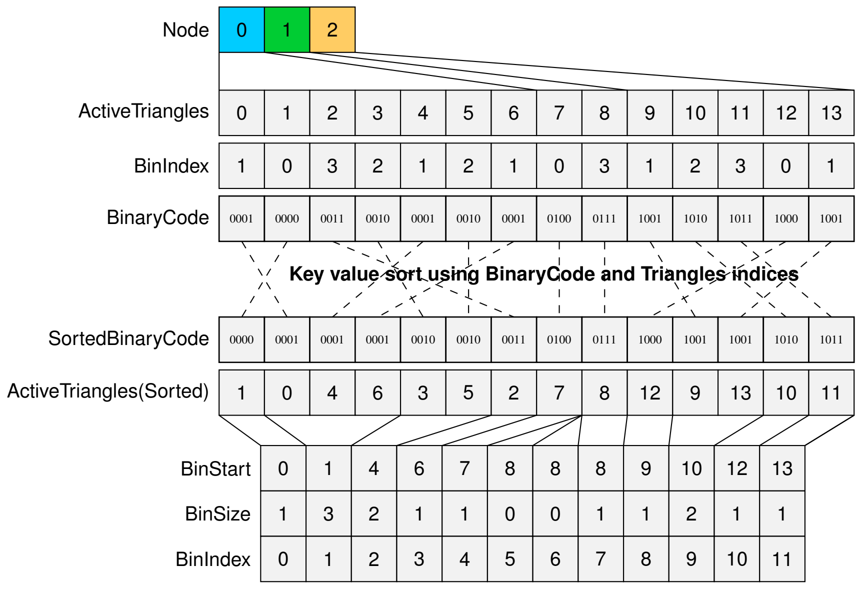


Figure 4.22: Sorting triangles into 4 bins using radix sort

4.7 Linear Bounding Volume Hierarchy (LBVH)

In [Lauterbach et al., 2009] Lauterbach el at. presented a fast BVH construction algorithm in which he employs a linear ordering based on Morton code and parallel radix sort to build the hierarchy. To build a hierarchy of depth $3k$ we create an operator that calculate for each primitive a $3k$ Morton code based on primitive centroid inside the scene bounding box, and call this operator to process all primitives in parallel. Then we sort these keys using a parallel sorting algorithm (e.g. radix sort). Upon a key observation by Lauterbach el at. [Lauterbach et al., 2009], if two adjacent keys differ in the most significant bit h then the final tree hierarchy will have $3k-h$ splits between these two primitives in all levels $h, h+1, \cdots, 3k$, So we prepare another operator that examine each adjacent pair of keys and just count the number of splits between them into array *NumSplits*. We perform an exclusive scan to the array *NumSplits* and get the total number of splits *nSplits* using the scan tail utility. Then we make an operator that fill the split pairs $[(i, h), (i, h+1), \cdots (i, 3k)]$ into the splits array, where each split store the primitive index in the sorted array and the split level in the hierarchy. For each primitive at index i we employ the *NumSplits* array to create the pairs and its scan to find the starting address for primitive pairs. We sort the splits array using table key-value parallel sorting employing the split level as the key. Now the sorted list records the splits sorted by tree level followed by primitive index (i.e. recording splits level by level).

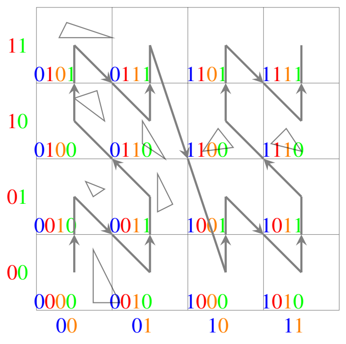


Figure 4.23: Morton codes generation.

We notice that for n splits al level l splits we will have $n+1$ nodes. So if the total number of splits equals *nSplits* then we will have *nSplits + 3k + 1* nodes in the hierarchy, where the last term in the previous expression corresponds to the root node. Each split corresponds to two nodes in the hierarchy; one to the right and one to the left. Thus, to create the tree nodes we map each split to a single tree node (the one to the left) except the last split in each level which is mapped to two nodes (the left and right ones). Each split at index i (with level l) in the sorted splits list will create its node at index $i + l$. It is trivial to note that splits with level $3k$ will correspond to the nodes at the leaf level and all other splits will correspond to internal nodes.

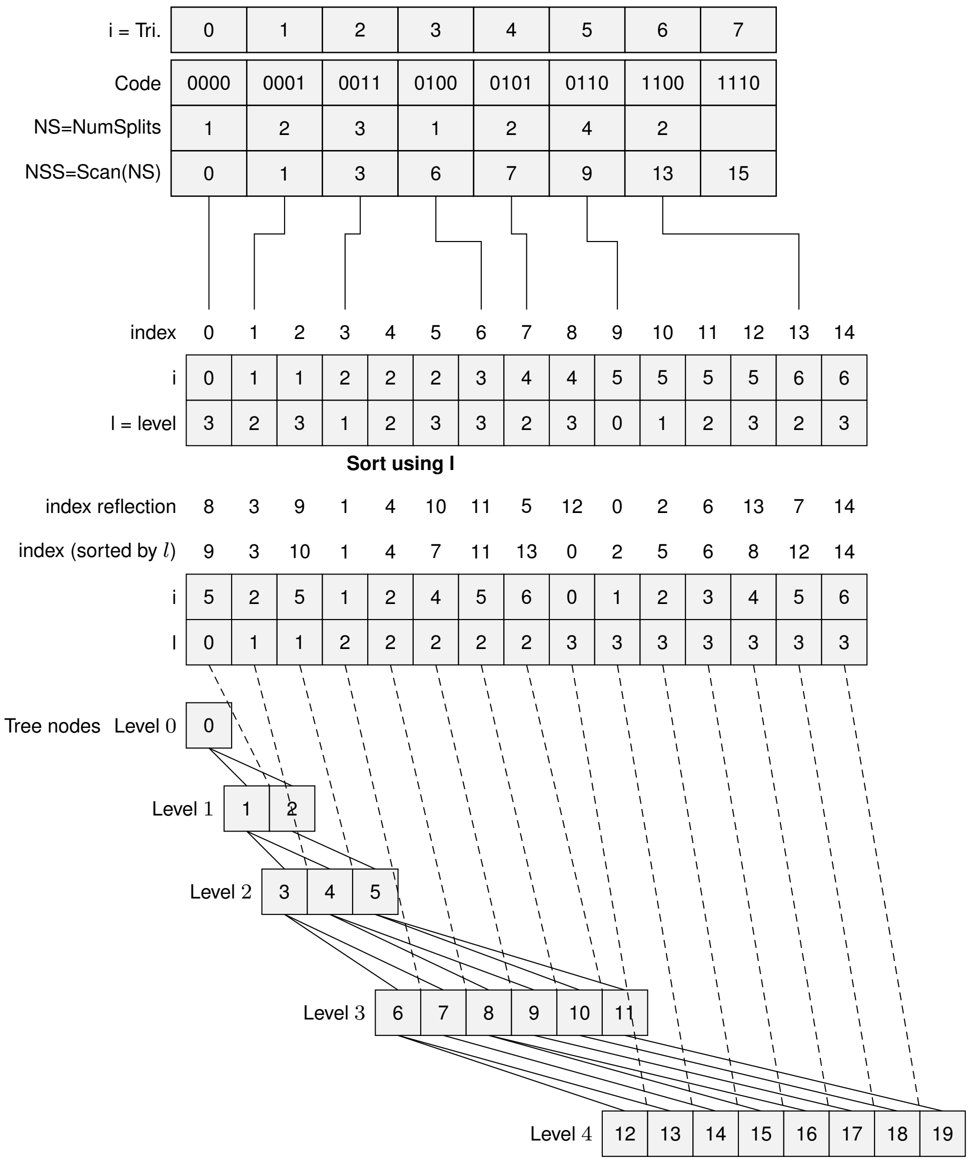
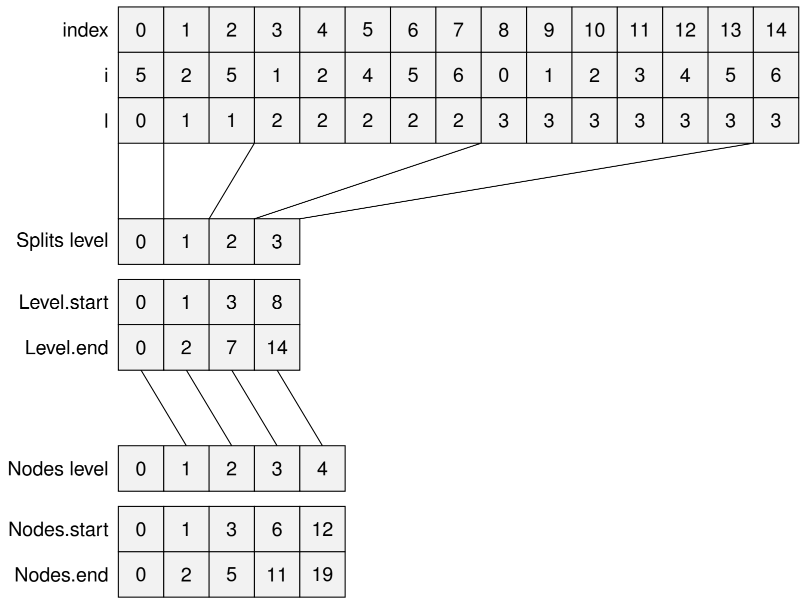

i = Tri. 0 1 2 3 4 5 6 7
Code 0000 0001 0011 0100 0101 0110 1100 1110
NS=NumSplits 1 2 3 1 2 4 2
NSS=Scan(NS) 0 1 3 6 7 9 13 15
index 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
i 0 1 1 2 2 2 3 4 4 5 5 5 5 6 6
l = level 3 2 3 1 2 3 3 2 3 0 1 2 3 2 3
Sort using l
index reflection 8 3 9 1 4 10 11 5 12 0 2 6 13 7 14
index (sorted by l) 9 3 10 1 4 7 11 13 0 2 5 6 8 12 14
i 5 2 5 1 2 4 5 6 0 1 2 3 4 5 6
l 0 1 1 2 2 2 2 2 3 3 3 3 3 3 3
Tree nodes Level 0 0
Level 1 1 2
Level 2 3 4 5
Level 3 6 7 8 9 10 11
Level 4 12 13 14 15 16 17 18 19

Figure 4.24: LBVH heirarchy emission.

Building the parent child relation is straightforward given the sorted list of splits; consider a split (h, l) at index i in the sorted splits array which corresponds to two child nodes one to the left (N_L) and one to the right (N_R), we get its next split $(h+1, l)$ in the sorted splits array and the two corresponding child nodes (C_L, C_R) and assign C_L as the right child of N_L, and C_R as the left child of N_R. Getting the next split from the current one is easily done using the sorted list first we get the index of split (h, l) before sort i_{us}, then we increment it by 1 and get the value in the reflection array at index $i_{us} + 1$ which corresponds to the position of the split $(h+1, l)$ in sorted splits array. To complete the hierarchy, starting form level 1 we assign first node in each level as the left child of first node the previous level, and the last node in each level as the right child of the last node in the previous level.



index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
i	5	2	5	1	2	4	5	6	0	1	2	3	4	5	6
l	0	1	1	2	2	2	2	2	3	3	3	3	3	3	3

Splits level	0	1	2	3
Level.start	0	1	3	8
Level.end	0	2	7	14

Nodes level	0	1	2	3	4
Nodes.start	0	1	3	6	12
Nodes.end	0	2	5	11	19

Figure 4.25: Sorted splits bounds and correspoinding nodes bound in the hierarchy.

To compute the BVH of the tree nodes we begin by reducing the AABB for all leaf nodes and traversing the tree in bottom-up order computing the internal node AABBs as the merge of the left and right child nodes. During this step we also delete singleton nodes by checking the left and light child nodes if they equal then we skip the child pointer to the parent node directly. This step is done using the information of the sorted array bounds so that the leaf nodes at levels $3k$ are store at positions $(3k + bounds[3k].start - 3k + bounds[3k].end + 1)$ and internal nodes at level $(l + bounds[l].start - l + bounds[l].end + 1)$.

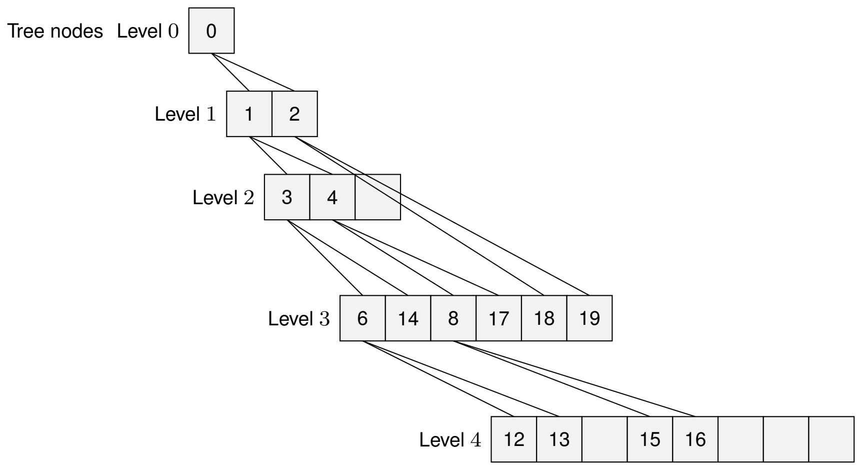


Figure 4.26: LBVH pruning.

4.7.1 Hybrid binned SAH BVH Algorithm

Similar to [Lauterbach et al., 2009] we employ the LBVH builder to construct the first n tree levels and use the binned SAH BVH builder to construct the remaining levels. We noticed that the leaf nodes bounding box calculation is the main bottleneck the LBVH construction and since at higher tree levels the number of triangles may be large and unbalanced between nodes, we use the *CreateChunks* utility to divide triangles of leaf nodes to Fixed-Sized chunks and use the reduction operator to calculate the AABB of each chunk followed by a segmented reduction step on the chunks AABBs to create leaf nodes AABBs. The resulting leaf nodes are first filtered and feed for the binned SAH BVH algorithm builder.

4.8 Analysis and Discussion

We have figured out that the BFS KD-tree and BVH construction algorithms can be reduced to a small set of parallel primitives on GPU. We assume that we have *N* nodes to be split which are stored in an array and that each node references a contiguous block of primitive indices in another array. The mission for any split procedure is to: (1) split or unsplit each nodes; (2) distribute the triangles to child nodes; (3) prepare the nodes and their corresponding primitives' indices for the next processing steps together.

We have discovered that this split operation can be generalized into a set of main consecutive parallel steps:

1. Partition large nodes into smaller ones.
2. Split nodes and optionally filter split nodes form unsplit nodes.
3. Sort (i.e. classify) primitives at every split node to their new parent nodes and use information form triangles sorting to calculate the child nodes size.
4. Optionally filter child nodes to their respective category (e.g. large or small).
5. Use information about parent, split, unsplit, and child nodes to distribute triangles to their new or old parents.

We explain in more details how each of these steps is mapped to the previously stated algorithms for KD-trees and BVHs. *Partitioning Nodes.*
To partition nodes into equally-sized sub-nodes we use the create chunks utility. *Splitting Nodes.*
For node splitting operation we have to distinguish two distinct cases:

1. Splits that always result in *two* child nodes as in the KD-tree large node stage.
2. Splits that results in *zero* or *two* child nodes as in BVH algorithms.

For the first case that always results in two splits we always prepare an array for child nodes that has double size of the input nodes array and given an input node at index *i* in the input array we store its two children at indices *2i* and *2i+1* in the output array. But for second case which results in *zero* or *two* child nodes, we begin by filling a 0/1 flags array aligned with input nodes array, at the index corresponding to each node we store 1 in flags array if the node will be split, and store 0 if it will not be split. Then we scan the flags array and the use the flags and the scan result to extract nodes corresponding to 0 flags as the right part of the split operation and store them to the leaf nodes. We also use the flags array and it scan we can create the child nodes; we pass again in parallel on each node if the corresponding flags equals 1 we create the two child node at indices *2Scan(flags)[i]* and *2Scan(flags)[i]+1* in output nodes queue.

Sorting Primitives at Every Split Nodes and Calculating Child Nodes Size.
As for primitives sorting we have to distinguish two distinct cases:

1. Primitives which are sorted to only one of the child node as in BVHs and point-based KD-trees.
2. Primitives which are sorted to one or both child nodes as in geometry KD-trees.

For the first case in which a triangle is sorted to only one of child nodes we use a single array of $1/0$ flags where we store 1 in corresponding index to each triangle if the triangle goes to left child node and store 0 if triangle goes to right child node. This flags array will be used to split primitives locally in each node by using a standard scan or segmented scan on these flags employing nodes triangle ownership as the segments of the flags. Such split operation is called a disjoint binary segmented split. The segmented scan tails of the flags array corresponding to left child nodes size, and for each parent node the size of its right child node is calculated as difference between its size and its corresponding scan tail.

In some cases primitives may be already sorted as in the binned SAH algorithm where the primitives were sorted to their respective bins and in point based KD-tree as we will see later, thus in such case we have no need to use flags array to classify primitives. Instead we just need to determine the splitting primitives in such sorted sequence.

For the second in which a triangle is sorted to only one or two of child nodes we use a two array of $1/0$ flags where we store $[1, 0]$ in corresponding index to each triangle in the two arrays if the triangle goes to left child node only and store $[0, 0]$ in the two arrays if triangle goes to right child node only, and $[1, 1]$ in the two arrays if triangle goes to both child nodes. This flags array will be used to split primitives locally in each node by using a standard scan or segmented scan on these flags employing nodes triangle ownership as the segments of the flags. Such split operation is called an overlapped binary segmented split. For each parent node the size of its left child node is calculated as the corresponding value at segmented scan tail of first flags array and the size of its right child node is calculated as sum of corresponding segmented scan tail of second flags array and difference between its size and its corresponding scan tail of the first array.

Filtering Child Nodes to their Node Category.

For two node categories as in large/small node filtering of KD-tree and BVH large stage, and binned BVH stage node filtering we have to divide this step into several parallel steps; First we use a single flags array to classify nodes and their corresponding size to large or small category; then, we scan each of size array corresponding to each category which will be used to locate triangles; Finally, we distribute a triangle by first checking its parent node category in the flags array to determine the corresponding association array and finding its parent node start and store it at index defined by the sum of its new parent start index plus the its local offset in its parent node.

For more than two node categories we can k flags arrays for k nodes categories to filter the nodes and the triangles sorting is identical to the two categories case. This is identical to what was supposed in the our proposed binned SAH BVH algorithm, but, since node filtering operation will require k parallel scans which may be a bottleneck we chose to filter the node using a single flags array into nodes belong to category 1 which are read for current processing stage and nodes belong to other category, and then after finishing each processing stage we again split nodes into nodes belong to next stage and nodes belongs to all further stages.

Distributing Triangles to (Child) Nodes.
Similar to primitives step sorting we have to distinguish two distinct cases:

1. Primitives which are distributed to only one of the child node as in binned BVH next/-further node filtering and point-based KD-tree and primitives by be sorted to leaf nodes as in BVHs node splitting step.
2. Primitives which are distributed to one or both child nodes as in geometry KD-trees.

In the first case we first check whether the parent node is split or node by examining the corresponding split flag; if the node is not split then we find leaf node and its corresponding start index using the scan of the split flags, and sort the primitive at an index defined by the leaf node start index plus its local offset in its old parent node. On the other hand, if the parent node is split and the primitive is sorted to one child node we use the flags array prepared in the triangle sorting step to find the parent child node category, then we check the corresponding flag of this node in the category flags array prepared in node filtering step to find the corresponding primitives association array and use the scan of the category flags array to find the start node index, then we sort a primitive at an index defined by the sum of the new parent node index and the scan of the classification flags; where left-child triangles use the local running scan of ones and the right-child triangles use the local running scan of zeros.

In the second stage where we sort primitives one or two child nodes we use the two arrays of falgs prepared in the triangles sorting step to find the new parent nodes: [1,0] means left child node only; [0,0] means right child node only; and [1,1] means both child nodes, then for every primitives sorted to left child node we store it at index defined by its parent node index plus an offset defined by the local running scan of ones the first flags array, and for every primitive sorted to right child node we store it at index defined by start node index plus the local running scan of zeros the first flags array and the local running scan of ones the second flags array.

4.9 Evalutions and Comparisons of Proposed Proposed Tree Construction Algorithms

Machine specifications. All algorithms were implemented using CUDA programming language on a machine with an NVIDIA Geforce 285 GTX with 1 GB memory and a Core 2 Due processor running at 2.66 GHz. In all algorithms we use the parallel primitives as explained earlier.

Device memory management. Similar to [Hou et al., 2010] we allocate a large block of memory for the entire algorithms, and allocate a conservative block for the persistent data structures which can be calculate in advance such as the final BVH tree structure in the SAH BVH algorithm which is bounded by $2n-1$ (where n is the number of primitives) due the fact that the worst case structure result in a leaf node with only one primitives and internal nodes in binary tree with n leaves is $n-1$. During intermediate steps and for temporary date such as the ping-pong lists we always able to allocate a fixed memory block size for the main poll which is freed as soon as is not in further use, and for final persistent data structures we can distinguish two cases; cases in which we know the required memory block size in advance such as the final tree in the KD-Tree algorithm which impose no constraint in our memory management, and case in which we can't calculate the required memory block size in advance such as the geometry array in the KD-tree algorithm, in such case we reserve a conservative block of memory relative to the initial scene size and large enough for the entire algorithm.

Model	**Size**	**Properties**
Toasters	11 K	Sparse geometry
Bunny	69 K	Uniform geometry
Dragon	100 K	Uniform geometry
Fairy Forest	178 K	Sparse geometry
Exploding Dragon	252 K	Sparse geometry

table 4.2: Benchmark scenes used in our experiments.

Benchmark models. In our implementation we use several publicly available benchmark models which cover the variability of relative scene size, and primitives' distribution (see Table 3.1) and to allow the evaluation of our new algorithms and compare it to recent published work (see Figure 4.27).

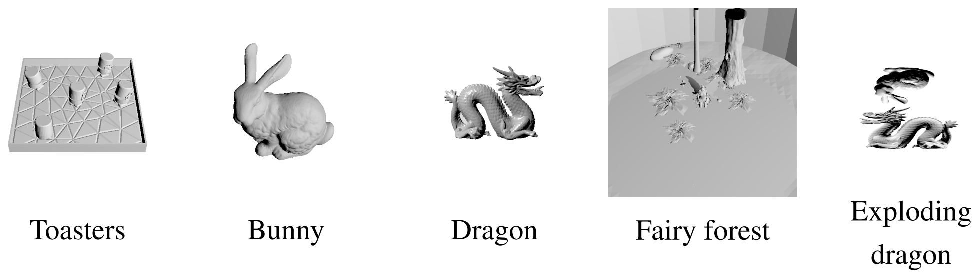


Figure 4.27: Benchmark scenes used to evaluate the construction and rendering performance of our construction algorithms.

Evaluation metrics. We evaluate and compare our algorithm using the construction time, note that all timings cover the entire construction process starting with building the initial bounding boxes, excluding initial CPU-GPU upload of geometry. Although the construction time is an important factor, its important is closely coupled with resulting tree quality on the ray tracing performance, so we calculate the heuristic SAH cost as explained in [Pantaleoni and Luebke, 2010] to evaluate the resulting tree quality, note that the smaller value of the SAH the better tree quality, and vice versa. But since the SAH cost represent the expected cost for a ray to traverse the entire tree and that SAH is a local greedy measure it is no strict correlation to the ray tracing time; so we measure both the ray tracing time for the camera ray and the accumulated construction and tracing time. We also compare the final tree size and the final memory footprint and the memory peak for all algorithms.

Model	**Const. time / fps**	**R. Cast time / fps**	**Const.+Trace time / fps**	**SAH**	**Tree Size # nodes/mem. size**
Toasters	41 ms / 24	38 ms / 27	79 ms / 13	128	1789 / 0.5 MB
Bunny	74 ms / 14	71 ms / 14	145 ms / 7	150	133703 / 2.3 MB
Dragon	104.6 ms / 9.5	48.3 ms / 20.7	152.8 ms / 6.5	184	169693 / 3.2 MB
Fairy Forest	195.1 ms / 5.1	94.9 ms / 10.5	290 ms / 3.4	149	309379 / 7 MB
Exploding Dragon	195.1 ms / 5.1	76.2 ms / 13.1	271.3 ms / 3.7	171	435973 / 8.09 MB

table 4.3: KD-tree build time, ray casting, and tree quality statistics for our test scenes.

Model	# small roots	Initial memory size
Toasters	701	3.5 MB
Bunny	2457	13.1 MB
Dragon	4399	21.5 MB
Fairy Forest	10751	53.9 MB
Exploding Dragon	10231	52 MB

table 4.4: Number of small roots and initial memory size.

Primary Rays Generation and Assignment to GPU Threads. In SIMT machines each warp of 32 threads follow the same execution path which means that the machine first find various execution branches for the warp and sequentially execute each branch separately. Threads that does not follow a certain branch remains idle while other threads execute their own branches. Ray traversal present a complex pattern of execution branching in which the ray oscillate between hierarchal traversal and primitive intersection in an unpredictable sequence. So in a single warp unless all rays needs to do the same action (hierarchal traversal or primitive intersection), some threads (e.g. traversing the hierarchy) have to remain idle until other threads (e.g. testing intersection with primitives) finish their execution branch. Since spatially coherent rays [Aila and Laine, 2009; Gunther, Popov, Seidel, and Slusallek, 2007; Wald et al., 2001] most probably will follow similar traversal path in a $3D$ scene; Morton order (or Z-order curve) [BIALLY, 1969] [1] ray assignment will be an efficient way for rays assignment to CUDA threads. For a screen of width *W* and height *H* we generate *N* rays, where $N = WH$ and assign each ray to a single thread. As shown in Figure 4.28 we assign rays to threads according to z-order where spatially coherent rays traverse neighbor pixels in the screen which are marked by similar colors.

Since each ray has a one to one correspondence to screen pixels, we assume that each ray *(R_i)* has an index *i* equal to screen location *where ($i \in [0, N)$).* To efficiently assign rays to threads we pre-create an array of integer values *(IdxToPos)* that maps each thread index *($tid \in [0, N)$)* to a ray index *i.* So given a thread *(tid)* we get its ray index as *i = IdxToPos[tid]* and then we calculate the corresponding screen position *($y = i/w, x = i\%w$)* and generate a ray through this pixel using our camera. We also pre-create anther array *(PosToIdx)* to be sued later in the shading step, and which do the reverse mapping and translate a screen position to a ray index

[1] Also known as space filling curve

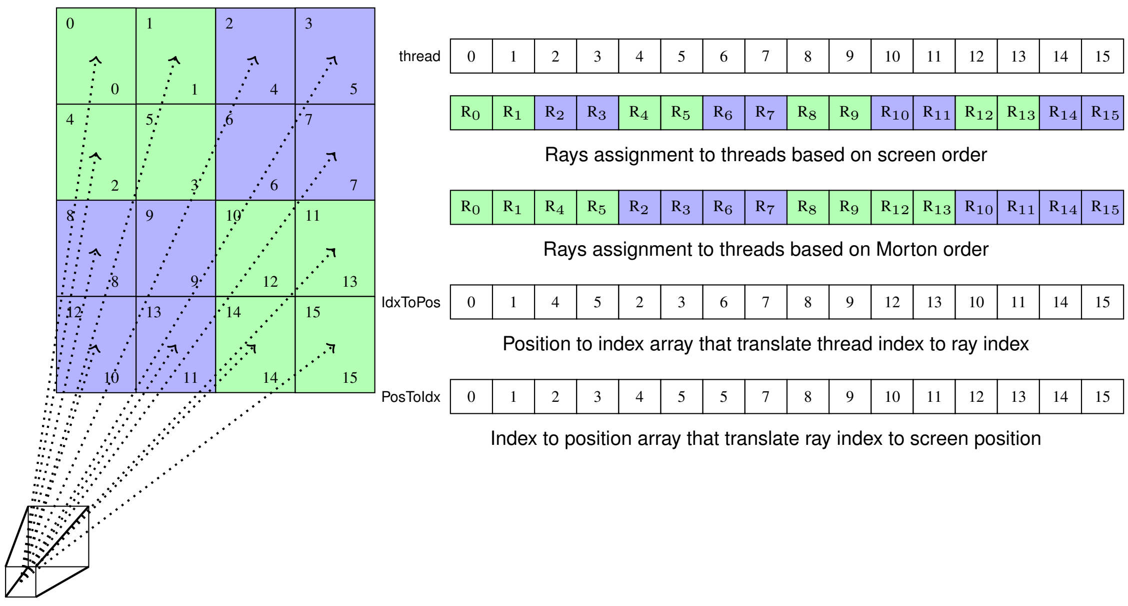

thread
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
R0 R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15
Rays assignment to threads based on screen order
R0 R1 R4 R5 R2 R3 R6 R7 R8 R9 R12 R13 R10 R11 R14 R15
Rays assignment to threads based on Morton order
IdxToPos
0 1 4 5 2 3 6 7 8 9 12 13 10 11 14 15
Position to index array that translate thread index to ray index
PosToIdx
0 1 2 3 4 5 5 7 8 9 10 11 12 13 14 15
Index to position array that translate ray index to screen position

Figure 4.28: Rays assignment to CUDA threads, in each cell at checkerboard numbers at the top left corner defines the sequential pixel order, and number in bottom right corner defined the z-order.

in the rays array [1].

Hierarchal traversal and primitives intersection. We uses stack based traversal algorithms for both KD-Tree [Foley and Sugerman, 2005; Horn, Sugerman, Houston, and Hanrahan, 2007; Wald, 2004] and BVH [Wald, 2004] and for triangles intersections we used Woop's unit triangle intersection test [Woop, Woop]. For the traversal kernel we used the *while-while* trace method [Aila and Laine, 2009] since it gives the best performance for the non persistent threads.

Algorithms evaluation. The LBVH has the best construction time (see Table 3.5) for all scenes but since the LBVH divides the space uniformly the resulting tree quality is comparable to other methods only in scenes with uniformly distributed primitives such as the Bunny model and fails in sparsely distributed scenes such as the Toasters model. While the LBVH final tree size is relatively large due to the many singletons in the hierarchy, recent work by Pantaleoni and Luebke [Pantaleoni and Luebke, 2010] overcome this issue using a new hierarchy emission.

The KD-Tree algorithm has a relatively fast construction time and good ray tracing performance (see Table 3.2) due the fast median node splits as the higher tree levels and the fast SAH

[1]Out implementation uses this method as it was implemented in the source code of [Aila and Laine, 2009] paper which is available at the author homepage

evaluation using the bit masks for node primitives but our implementation failed to approach the performance appeared in the original publication [Zhou et al., 2008], we believe that this due that we avoid primitive clipping and only perform AABB clipping. Wald et al. [Wald and Havran, 2006] noted that the number of clipped triangles is order of $\sqrt{N}$ while we found that after the large stage the number of clipped triangle is about $2N$. The KD-Tree algorithm has a heavy memory foot print mainly due to the intermediate data structures for the small roots since each small root with n nodes requires two words to store start primitive index and the size of the node and $6n$ splits where each split requires 24 byes (4 bytes of the split axis, 4 bytes for the split position and 8 bytes for each of the left and right sets) (see Table 3.2). Keeping in mind that the total splits always greater than the scene due to triangles splitting, this memory cost is relatively high.

The SAH BVH algorithm is relatively slow due to the many SAH cost evolution in each node, while the tree quality the best for ray tracing performance (see Table 3.4), to overcome the construction time limitations the LBVH can be used to quickly build the higher tree levels in a hybrid algorithm as explained in [Lauterbach et al., 2009] which gives a good cumulative construction and ray tracing performance (see Table 3.6). Since the original SAH BVH algorithm is not work efficient; our new binned SAH algorithm take this into consideration and only evaluate the SAH cost once using chunks and share the chunks information using scans and results in a construction time which is at least an order of magnitude faster that the SAH BVH algorithm in most scenes (see Table 3.7). To also allow faster build time we use the LBVH to construction the higher tree levels resulting in a better build time (see Table 3.8). The memory footprint of the SAH BVH and the binned SAH BVH algorithms and their hybrid version are relatively small since scene primitives are not split and all intermediate date are always conservative in input the scene size.

Our binned SAH BVH algorithm give the worst performance in Toasters scene, we believe that this is due the large triangles in the model which always results in large sub trees and lead to false split positions in next splits, such problem can be avoided by selecting split candidates from the bounding box around triangles centroids.

We presented fast and efficient parallel algorithms for building BVH algorithms on GPU. We compared our new algorithms with the recent state of the art algorithms for building both KD-tree and BVH on GPU and showed that our algorithm in most scenes outperform these algorithm in the construction time and the cumulative build time and trace time for the visibility

Model	**Const.** **time / fps**	**R. Cast** **time / fps**	**Const.+Trace** **time / fps**	**SAH**	**Tree Size** **# nodes/mem. size**
Toasters	36.9 ms / 27.1	21.3 ms / 46.9	58.2 ms / 17.1	67	17803 / 0.68 MB
Bunny	144.3 ms / 6.9	28.8 ms / 34.7	173.1 ms / 5.8	61	115719 / 4.4 MB
Dragon	219.4 ms / 4.5	23.1 ms / 43.1	242.6 ms / 4.1	82.9	185971 / 7.1 MB
Fairy Forest	522.4 ms / 1.9	48 ms / 20.1	570.4 ms / 1.8	55	262161 / 10 MB
Exploding Dragon	554 ms / 1.8	38 ms / 26.3	593 ms / 1.7	76	464725 / 17.7 MB

table 4.5: SAH BVH build time, ray casting, and tree quality statistics for our test scenes.

Model	**Const.** **time / fps**	**R. Cast** **time / fps**	**Const.+Trace** **time / fps**	**SAH**	**Tree Size** **# nodes/mem. size**
Toasters	3 ms / 330	28.3 ms / 35.3	31.3 ms / 31.9	101.962	114297 / 4.3 MB
Bunny	8.5 ms / 117.2	32.9 ms / 30.4	41.4 ms / 24.1	72.6	660731 / 25.2 MB
Dragon	10.9 ms / 91.5	28.3 ms / 35.2	39.3 ms / 25.4	104	854540 / 32.6 MB
Fairy Forest	10.9 ms / 91.1	63.8 ms / 15.6	74.8 ms / 13.4	61.4	351502 / 13.4 MB
Exploding Dragon	19.5 ms / 51.2	46.7 ms / 21.3	66.3 ms / 15	100	1808614 / 68.9931 MB

table 4.6: LBVH build time, ray casting, and tree quality statistics for our test scenes.

Model	**Const.** **time / fps**	**R. Cast** **time / fps**	**Const.+Trace** **time / fps**	**SAH**	**Tree Size** **# nodes/mem. size**
Toasters	10.9 ms / 91.6	27.3 ms / 36.6	38.2 ms / 26.2	99	26038 / 0.99 MB
Bunny	29.2 ms / 34.3	32 ms / 31.1	61.3 ms / 16.3	71	129154 / 4.9 MB
Dragon	56.5 ms / 17.7	27.7 ms / 36.1	84.1 ms / 11.9	102	202401 / 7.7 MB
Fairy Forest	195.8 ms / 5.1	52.3 ms / 19.1	248 ms / 4	59.4	269846 / 10.3 MB
Exploding Dragon	127.4 ms / 7.8	43.9 ms / 22.8	171.3 ms / 5.8	96.7	481906 / 18.3 MB

table 4.7: Hybrid SAH BVH build time, ray casting, and tree quality statistics for our test scenes.

Model	**Const.** **time / fps**	**R. Cast** **time / fps**	**Const.+Trace** **time / fps**	**SAH**	**Tree Size** **# nodes/mem. size**
Toasters	37.7 ms / 26.4	28.3 ms / 35.7	66 ms / 15	36	4077 / 0.16 MB
Bunny	47.4 ms / 21	31 ms / 32.1	78.5 ms / 12.7	63	42849 / 1.6 MB
Dragon	63.5 ms / 15.7	27.3 ms / 36.6	90.8 ms / 11	85.6	68567 / 2.6 MB
Fairy Forest	94.6 ms / 10.6	147.1 ms / 6.8	241.7 ms / 4.1	58.6	106515 / 4.06 MB
Exploding Dragon	84 ms / 11.9	38.5 ms / 26	122.6 ms / 8.16	79	171529 / 6.5 MB

table 4.8: Binned SAH BVH build time, ray casting, and tree quality statistics for our test scenes using five stages.

Model	**Const.** **time / fps**	**R. Cast** **time / fps**	**Const.+Trace** **time / fps**	**SAH**	**Tree Size** **# nodes/mem. size**
Toasters	29.6 ms / 33.8	20 ms / 50	49.6 ms / 20.1	40	3530 / 0.13 MB
Bunny	36.4 ms / 27.4	31 ms / 32.3	67.4 ms / 14.8	73	63879 / 2.4 MB
Dragon	43.2 ms / 23.1	27.3 ms / 36.6	70.5 ms / 14.2	92.6	79537 / 3.0 MB
Fairy Forest	76.3 ms / 13.1	153.2 ms / 6.5	229.5 ms / 4.3	64.6	126502 / 4.9 MB
Exploding Dragon	74 ms / 13.5	42.5 ms / 23.5	116.5 ms / 8.6	89	191437 / 7.3 MB

table 4.9: Hybrid binned SAH BVH build time, ray casting, and tree quality statistics for our test scenes.

text. We also showed that most of these algorithms can be reducede to a small and standard set of parallel primitive algorithms. In next chapter we will evaluate our new hierarch in both Whitted style ray tracing and photon mapping.

5

Ray Tracing on GPU

5.1 Parallel Ray Tracing on GPU

As explained earlier a ray tracing algorithm is a recursive procedure that is called to calculate the shading color reflected at each pixel. Listing 5.1 presents the pseudo code for a general ray tracer in which we call the trace procedure for each pixel.

```
foreach pixel
    Pick a ray from the eye through this pixel
    Pixel color = Trace( ray )
```

Listing 5.1: Recursive Ray Tracing Algorithm

The trace procedure traverses the ray through the scene and finds the nearest intersection; when we find an intersection we calculate the shading due to direct lighting using surface properties, and light source(s). If we hit a specular surface we recursively trace another bounce of reflected and/or refracted ray(s) and accumulate the returned shading color to the final pixel value. We refer the readers to the these references [Jensen, 2004; Jensen et al., 2003; Morley and Shirley, 2003; Pharr and Humphreys, 2010] which contain more details about ray tracing and its variations ray tracing algorithms and their variations [1] .

```
Procedure Trace ( ray )
{
    // search the nearest hit point with scene primitives
    hit point = FindNearestIntersection()
```

[1]The reference [Pharr and Humphreys, 2010] includes source code for a complete physically based renderer which is available at the book homepage

```
    color = RGB(0,0,0)
    foreach (light source)
    {
        Trace shadow ray form hit point to light source
        if ( shadow ray intersects light source )
            color +=  direct illumination
        if( specular )
            color += Trace (reflected/refracted ray(s))
    }
    return color
}
```

Listing 5.2: Tracing Ray

Since current versions of CUDA does not support recursion we have to reformulate the recursive trace procedure into an iterative implementation. A naïve iterative trace procedure can be implemented on GPU by substituting the recursion with a local stack. But such implementation is expected to perform poorly on GPU since the trace procedure then will result in a large branching divergence between GPU threads. We found that two important properties of the trace procedure can help us to build an efficient parallel implementation on GPU: (1) since the shading color returned form the next bounce(s) is irrelevant from the shading color at the current hit, then we can make the shading calculation in a separate step after traversing all primary and secondary rays; (2) In practice we recurs the trace procedure N times, where N is a relatively small number *(e.g.* $N < 6$*)* which means that we can trace each bounce in parallel in a breadth-first search order.

A general parallel pipeline for ray tracing on GPU employing the above observations was introduced by Zhou et al. in [Zhou et al., 2008], where we perform the following steps:

1. Generate and trace primary rays *in parallel*.
2. Compact and append hits on non-specular surfaces using *parallel* list compaction.
3. Collect hits on specular surfaces using *parallel* list compaction and *in parallel* trace reflection and/or refraction rays from them.
4. Repeat Step 2 and Step 3 for N bounces.
5. Generate and trace shadow rays and accumulate shading *in parallel*.

In this section we will explain in more details how we can implement each of these steps efficiently on GPU.

Choosing the reflectance model. We use Schlick approximation of the Fresnel reflectance to simulate refraction through transparent materials. Since we use Fresnel approximation we expect that every ray hitting a specular surface is expected to bounce at most two rays one in the mirror like reflection direction for which the returned shading color is weighted by the reflectivity *(Re)* coefficient and another ray in the refraction direction for which the returned shading color is weighted by the transmissive coefficient *(Tr = 1- Re)*.

5.1.1 Parallel Rays Generation and Bouncing

In this section we present a general parallel ray tracing algorithm and implementation details that support both reflection and refraction of physically based renderer.

```
//Global Data
N   // rays array, number of rays
Nodes, NodesLength // array of 4 bytes words to store tree nodes and its size
DiffuseNodesData{Tri, U, V} // array of 4 bytes words to store tree nodes and its size
ReflectionNodesData{Tri, U, V, Child} // array of 4 bytes words to store tree nodes and its size
RefreactionNodesData{Tri, U, V, Re, Child} // array of 4 bytes words to store tree nodes and its size
Rays{Org, Dir} // rays array, number of rays
Hits{T, Tri, U, V} //hit distance, triangle id, its barycenters (α,β) of the hit
DiffuseFlags, DiffuseFlagsScan //arrays for diffuse hits flags and its scan
NumBouncedRays, NumBouncedRaysScan //array for number of bounced ray at the hit and its scan
TwoBoncedRaysFlags, TwoBoncedRaysFlagsScan //array that store 1 if the hit bounce two rays and its scan
DiffuseNodes, DiffuseNodesLenght //array for diffuse node indices in the tree and its count
SpecularNodes //array for specular node indices in the tree
SpecularNodesPerLevel //array to store specular node count per level


Procedure RayTrace()
{
    //initialization}
    NumTracedRays = N // initial number of traced rays
    TreeNodesCount = 0
    in = out = 0  // address for input/output rays in rays array
    pp = 0  // Ping-Pong variable

    for(b=0;  b<NumBounces;  b++ )
    {
        in = out
        pp = 1 - pp   // flip the ping-pong variable
        out = pp * NumTracedRays

        //trace rays
        TraceRays<NumTracedRays>(&Ray[in])

        //scan diffuse and specular hits
        Scan(<DifHitFlagScan, NumBouncedRaysScan, TwoBoncedRaysFlagScan>, <DifHitFlag, NumBouncedRays,
      TwoBoncedRaysFlag>, NumTracedRays)

        ScanTail(<DifFlagsCount, NumBouncedRaysCount, TwoBoncedRaysFlagCount>, <DifHitFlagScan, NumBouncedRaysScan
      , TwoBoncedRaysFlagScan>, <DifHitFlag, NumBouncedRays, TwoBoncedRaysFlag>, NumTracedRays)

        //compact diffuse hits
       if(DifFlagsCount > 0)
       {
            CompactDiffuseHits<NumTracedRays>()
            DifNodeCount += DifFlagsCount
       }

        //update node count
```

```
            SpecNodesCountPerLevel[b] += NumTracedRays
            SpecNodesCountPerLevel[b+1] += NumSpecNodes[b]
            TreeNodesCount += NumTracedRays

        if(NumBouncedRaysCount == 0)
        {
            break
        }

        //compact specular hits
        CompactSpecularHits<NumTracedRays>(&Ray[in], b)
            ReflectionNodeCount += NumBouncedRaysCount - 2 * TwoBoncedRaysFlagCount
            RefractionNodeCount += TwoBoncedRaysFlagCount

        //create new rays
        if(i < NumBounces-1)
        {
            GenerateNewRays<NumTracedRays>(&Ray[in],&Ray[out])
        }

        NumTracedRays = NumBouncedRaysCount // new rays count
    }
}
```

Listing 5.3: Parallel ray tracing algorithm

Data Structures.
Output Data Structures. The output of the algorithm is the rays tree nodes and 3 arrays for their related data (see below). We also create 2 arrays *(DiffuseNodes, SpecularNodes)* for each of the diffuse and specular node indices in the nodes array. And for specular node we also prepare a small array *(SpecularNodesPerLevel)* that store the bounding indices of each bounce in the specular nodes array (this array has length equals the number of bounces of ray tracer).
Transient Data Structures. Initially we need N primary rays, and for secondary rays we have two options: (1) Either to allocate enough rays for the next bounce which means that we reserve at most two secondary rays for each primary ray. So given a primary ray R_i at index i we always reserve indices $2i$, $2i+1$ in the output buffer for its new bounced rays and later we remove *null* rays using the compact utility. (2) Or we count the number of bounced rays in an integer array *(NumBouncedRays)* (which has the same size of the rays array) and later we generate new rays in a compact form using the scan of *NumBouncedRays* array. We found that the second option is more efficient than the first from the memory storage perspective, at the same time we get a similar performance to first option since we keep intermediate values of ray hit and avoid recalculation of primitives' intersections. We pre-allocate rays array where each ray is represented using an origin *(Org)* and a direction *(Dir)* and use this array in a Ping-Pong fashion to store traversed and bounced rays. During ray traversal we temporary need an array *Hits* to store hit information including hit distance *(T)* and primitive data *(i.e. triangle id and barycenter coordinates of the triangle)* for each ray hit, this array has length equals N corresponding to the upper bound of traversed rays at the primary ray traversal stage. We create 3 arrays *(DiffuseFlags, NumBouncedRays, TwoBoncedRaysFlags)* that count the

number of diffuse and non-diffuse hits, and prepare another 3 arrays for the scan of each one, each of these arrays are of length equals *N*.

Rays Traversal and Bouncing. A typical pseudo code for a parallel ray tracing algorithm is presented in Listing 5.3, in which we iterate for an number of bounces *NumBounces* to simulate ray traversal, in each iteration we trace current rays in parallel, count diffuse hits using the array *DiffuseHits*, and count non-diffuse hits using two arrays: (1) the *NumBouncedRays* array which store the number of bounced rays at each hit; And (2) the *TwoBoncedRaysFlags* array in which store 1 if the current hit bounces two new rays and 0 otherwise. In next phases we can look at the two arrays *(NumBouncedRays, and TwoBoncedRaysFlags)* from different perspectives:

1. If we are interested only on the number of new bounced rays we use the array *NumBouncedRays* and its scan.
2. If we are interested in the number of specular hits we can use the difference between *NumBouncedRays, and TwoBoncedRaysFlags* and their scans.
3. If we are interested in the number of hits that bounces a single ray we use difference between *NumBouncedRays* array and the array defined by *2* $\times$ *TwoBoncedRaysFlags* and their corresponding scans.
4. If we are interested in the number of hits that bounces two rays we use an array defined by 2 $\times$ *TwoBoncedRaysFlags* and the corresponding scan.
5. If we are interested in the number of rays misses we use the difference between.

```
Operator TraceRays(Ray ray[])
{
    i = ThreadIndex
    iRay = ray[i]
    if(iRay hits a primitive)
    {
        Set RayT[i],TriId[i],TriAlpha[i],TriBeta[i] //hit distance, triangle id, $\alpha,\beta$ of the hit
        if(iRay hits a diffuse surface)
        {
            DifHitFlag[i] = 1
            NumBouncedRays[i] = 0
            TwoBoncedRaysFlag[i] = 0
        }
        else // ray hits a specular surface
        {
            DifHitFlag[i] = 0
            Store the number of bounced rays in NumBouncedRays[i]
            TwoBoncedRaysFlag[i] = NumBouncedRays[i] − 1
        }
    }
}
```

Listing 5.4: Parallel Trace Rays Operator

If we found valid diffuse hits using the scan tail of the *DiffuseHits* array, we append them to the shading tree using the operator *CompactDiffuseHits*. And if we found specular hits we make two steps: (1) We append non-diffuse hits to the shading tree using the operator *CompactSpecularHits*; (2) If we are not in the last bounce we generate new rays which will be traced in the next bounce using the operator *GenerateNewRays*.

In the procedure *GenerateNewRays* we check the *NumBouncedRays* array, if it is greater than 0 we get the hit information for that ray *(T, Tri, U, V)* to create new rays in the reflection and refraction directions and compact them in the *OutRays* array using the scan of *NumBouncedRays* array.

```
Operator GenerateNewRays(Rays,OutRays)
{
    i = ThreadIndex
    if(NumBouncedRays[i] == 2)
    {
        Get hit info from Hits (T[i], Tri[i],U[i],V[i])
        Create reflection ray (Ray1) and refraction ray (Ray2) using hit info and Rays[i]
        outAddress = NumBouncedRaysScan[i]
        < OutRays[outAddress], OutRays[outAddress+1] > = < Ray1, Ray2 >
    }
    else if(NumBouncedRays[i] == 1)
    {
        Get hit info from Hits (T[i], Tri[i],U[i],V[i])
        Create reflection ray (Ray1) using hit info and Rays[i]
        outAddress = NumBouncedRaysScan[i]
        OutRays[outAddress] = Ray1
    }
}
```

Listing 5.5: Generate New Rays Operator

5.1.2 Building Shade Tree

Shade tree is a tree which originates at each pixel and store ray traversal path inside the scene. In shade tree nodes represent traversed ray hit (or miss) and links corresponding to ray bouncing between surfaces. Shade trees are important since we perform the shading calculation after finishing ray traversal pass, so it must include all necessary information to perform the shading step. In this section we will explain several ways to efficiently organize, store and traverse shade trees on GPU.

Figure 5.1 represents various examples of shade trees, the tree rooted at node 1 represents the simplest form of the shading tree with only one node corresponding to a primary ray that misses the geometry, in such case we just store a state to indicate a ray miss at this node. The tree rooted at node 2 represents the case when we hit a diffuse surface, so we store in this node a state to indicate a diffuse ray hit and a reference hit information (e.g.; triangle id, and hit position). Tree rooted at node 3 represents a single ray bounce through specular reflection in

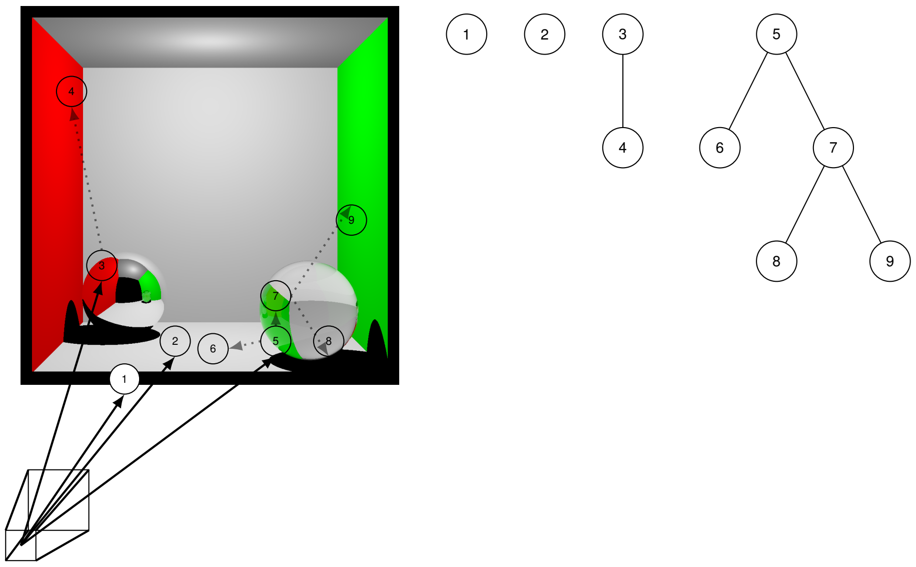


Figure 5.1: Shading trees on Cornel box scene: (Right) various forms of shade trees corresponding to different screen positions (left).

this case the root node contains a state to indicate a ray reflection, and store the primary ray hit information, and a reference to the next bounced ray (node 4) which itself stores a diffuse hit information corresponding to the secondary ray hit at the right red wall. The tree rooted at node 5 represents a more complex pattern in which we perform two bounces of ray traversal, primary ray hit on the refractive sphere results in two new rays, one in the ideal reflection direction (node 6) and the other in the refraction direction (node 7) through the sphere medium. Node 6 which corresponds to the reflected ray ends at the floor and node 7 bounces again two rays in the reflection (node 8) and refraction direction (node 9). As you notice the reflection in node 8 is inside the sphere and will result in two bounced rays again but since we limit the ray traversal depth to 3 we kill this ray and mark it as a ray miss.

Shade Node Storage. We need to store in each node (1) a state to indicate the hit type, (2) a reference to the hit information, and in case of ray reflection and refraction we have to store (3) a reference to child nodes corresponding to child nodes (bounced rays). Each ray can have one of 4 distinct cases for an intersection: (1) Ray misses geometry; (2) Ray hits a diffuse surface; (3) Ray hits a specular surface and bounces a single ray; (4) Ray hits a specular surface and bounces two new rays. So we just need two bits to store each these states which we refer form now as the *HitState*. For *HitState* 0 *(ray miss)* we needed not to store anything else in the node. For *HitState* 1 *(diffuse hit)* we need to store a reference to hit information (i.e. primitive id, and barycenter coordinates). For states 2 and 3 *(single and double ray bounces)* we need to store a single reference to next bounce(s) (i.e. another node(s) in the shade tree) and a reference to store a reference to(i.e. primitive id, and barycenter coordinates). Note that with state 3 *(double ray bounces)* we always store the child nodes beside each other so that a single reference at the parent node is enough to access the two children.

We store tree nodes in linear array so that the primary rays occupy the first *N* nodes in the array, and use the scan primitive to compact the tree nodes corresponding to secondary rays and to maintain references to hits information and child nodes.

Shade Trees Structure. As we explained earlier the tree node have to store hit state in the 2 MSB of the node and hit information and references to child nodes. So if *HitState = 1* we store the primitive id and barycenter coordinate of the primitive, if *HitState = 10* we create a new node for the new reflected ray and store the primitive id and barycenter coordinate of the primitive at and a reference to the child node, and if the *HitState = 11* we create two new nodes for the new rays and store the primitive id and barycenter coordinate of the primitive and Schlick reflectivity coefficient *(Re)* and a reference to the child nodes.

Initially we make four arrays:

1. *Nodes* array of 4 bytes words to store tree nodes.
2. *DiffuseHitData* array to stores the diffuse hit information (i.e. Tri, U, V), which is compacted using the scan of *DiffuseHits* array.
3. *ReflectionHitData* array which stores the hit information and an index to the child node corresponding to the bounced ray and this array is compacted using the difference between the scans of *NumBouncedRays, and TwoBoncedRaysFlags* arrays.
4. *RefractionHitData* which store the hit information and Schlick reflectivity coefficient *(Re)* and an index to the first of the two child nodes corresponding to the bounced rays and this array is compacted using the scan of *TwoBoncedRaysFlags* array.

And the tree node will be used as follow:

1. bits $[31 - 30]$ to store *HitState*.
2. bits $[30 - 0]$ will store a *Reference* which is translated based on *HitState*:
   (a) If *HitState = 00*, then it will be unused.
   (b) If *HitState = 01*, then it store an index referencing *DiffuseHitData* array in which we store the hit infomation.

(c) If *HitState = 10*, then it store an index referencing *ReflectionHitData* array in which we store the hit information and a reference to the child node index.

(d) If *HitState = 11*, then it store an index referencing *RefractionHitData* array in which we store the hit information and Schlick reflectivity coefficient *(Re)* and a reference to the first of the two child nodes.

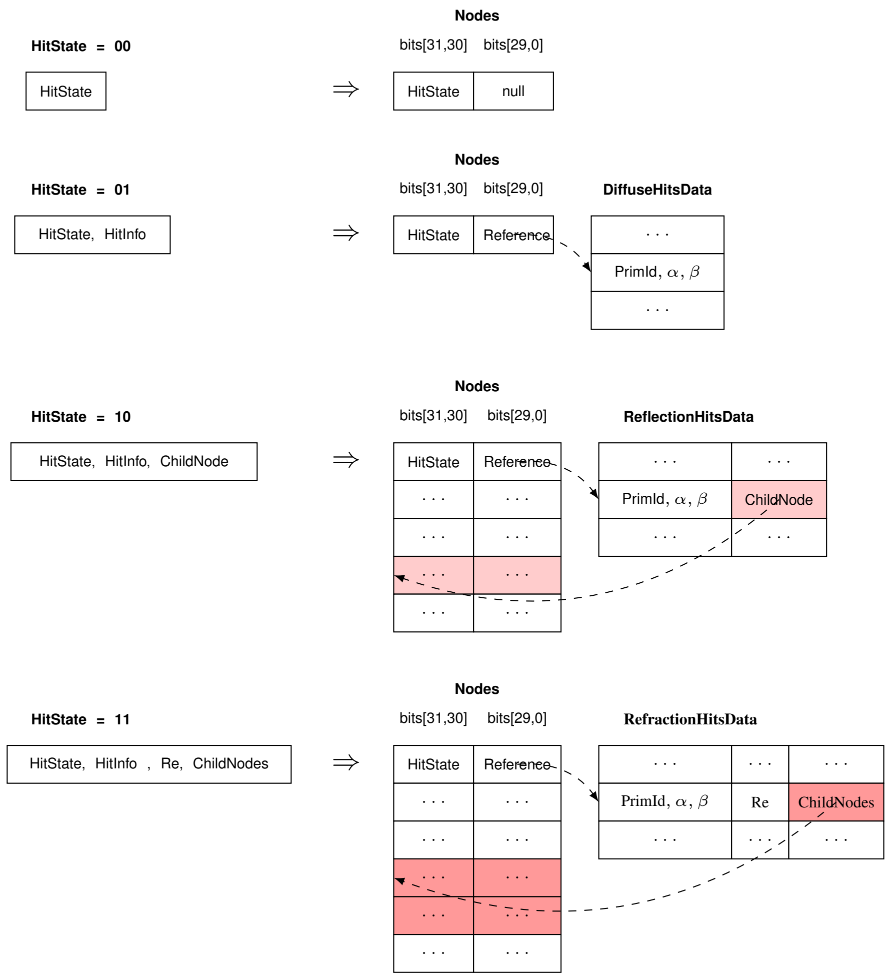


Figure 5.2: Shade tree node structure

To complete the tree structure we make an array *DiffuseNodes* to store the tree nodes indices with diffuse hit state and compact it using scan of *DiffuseHits* array, And make an array *SpecularNodes* to store the tree nodes indices with specular hit state and compact it using the difference between the scans of *NumBouncedRays* and *TwoBoncedRaysFlags* arrays. We make arrays *(SpecularNodesPerLevel)* to store the last index of *SpecularNodes* array for each ray traversal level. Values stored in *SpecularNodesPerLevel* are the accumlation of difference between the scan tails of *NumBouncedRays* and *TwoBoncedRaysFlags* arrays.

```
Operator CompactDiffuseHits()
{
    i = ThreadIndex
    if(DiffuseHits[i] == 1)
    {
        Get hit information (TriId[i],TriAlpha[i],TriBeta[i])
        dataAddress = DifNodeCount + DifHitFlagScan[i]
        DiffuseHitsData[dataAddress] = <TriId[i],TriAlpha[i],TriBeta[i]>
        RayTreeStart = TreeNodesCount
        nodeIndex = RayTreeStart + i
        Nodes[nodeIndex]$_{bits[30-0]}$ = dataAddress
        Nodes[nodeIndex]$_{bits[31-30]}$ = 01
        DiffuseNodes[dataAddress] = nodeIndex
    }
}
```

Listing 5.6: Compact Diffuse Hits Operator

```
Operator CompactSpecularHits(Ray ray, int Bounce)
{
    i = ThreadIndex
    if(NumBouncedRays[i] == 1)
    {
        Get hit information (TriId[i],TriAlpha[i],TriBeta[i])
        RayTreeStart = TreeNodesCount
        childNodeIndex = RayTreeStart + NumTracedRays + NumBouncedRaysScan[i]
        dataAddress = ReflectionNodeCount + NumBouncedRaysScan[i] - 2*TwoBoncedRaysFlagScan[i]
        ReflectionHitsData[dataAddress] = <TriId[i],TriAlpha[i],TriBeta[i], childNodeIndex>
        nodeIndex = RayTreeStart + i
        Nodes[nodeIndex]$_{bits[30-0]}$ = dataAddress
        Nodes[nodeIndex]$_{bits[31-30]}$ = 10
        SpecularNodesStart = SpecNodesCountPerLevel[Bounce]
        specularNodeIndex = SpecularNodesStart + NumBouncedRaysScan[i] - TwoBoncedRaysFlagScan[i]
        SpecularNodes[specularNodeIndex] = nodeIndex
    }
    if(NumBouncedRays[i] = 2)
    {
        Get hit information (TriId[i],TriAlpha[i],TriBeta[i])
        Calculate reflectivity cofficint (Re) uisng hit info and ray[i]
        RayTreeStart = TreeNodesCount
        childNodeIndex = RayTreeStart + NumTracedRays + NumBouncedRaysScan[i]
        dataAddress = RefractionNodeCount + TwoBoncedRaysFlagScan[i]
        ReflectionHitsData[dataAddress] = <TriId[i],TriAlpha[i],TriBeta[i],Re,childNodeIndex>}
        nodeIndex = RayTreeStart + i
        Nodes[nodeIndex]$_{bits[30-0]}$ = dataAddress
        Nodes[nodeIndex]$_{bits[31-30]}$ = 11
        SpecularNodesStart = SpecNodesCountPerLevel[Bounce]
        specularNodeIndex = SpecularNodesStart + NumBouncedRaysScan[i] - TwoBoncedRaysFlagScan[i]
        SpecularNodes[specularNodeIndex] = nodeIndex
    }
}
```

Listing 5.7: Compact Specular Hits Operator

5.1.3 Accumulating Shading and Rendering

With this structure shading can be done by traversing each of the first *N* nodes corresponding to primary rays using post order tree traversal (see next section). But in next section we will present an efficient parallel way to traverse the three level by level in bottom up manner. For this purpose we need to store the indices of tree nodes corresponding to diffuse hits and specular hits for each ray traversing level.

Parallel Post Order Shade Trees Traversal.

```
//Global Data
Nodes           //shade tree nodes
N               //screen size
Surface         //N size color buffer for screen
//Local Data
NodeStack       //node stack
ColorStack      //color stack

Procedure Shade()
{
    EvaluateShadingPO<N>()
}
```

Listing 5.8: Post order shade tree traversal

```
Operator  EvaluateShadingPO()
{
    i = ThreadIndex
    <NodeIndex, NodeState> = <osToIdx[i], 0> //get the primary ray index, state
    while(true)
    {
        <HitState, Reference> = <Tree[NodeIndex]$_{bits[31-30]}$, Tree[NodeIndex]$_{bits[30-0]}$>
        if(HitState == 0)     //case 1: ray miss
        {
            ColorStack.Push(black)
            if( NodeStack.Size>0 )
            {
                <NodeIndex,NodeState> = NodeStack.Pop() //pop node
            }
            else //nodes stack emtpy
            {
                break
            }
        }
        else if(HitState == 1)      //case 2: ray hits
        {
            Get HitInfo uisng Reference
            Calculate DirectLight using HitInfo
            ColorStack.Push(DirectLight)
            if(NodeStack.Size>0)
            {
                <NodeIndex,NodeState> = NodeStack.Pop() //pop node
            }
            else //nodes stack emtpy
            {
                break
            }
        }
        else if(HitState = 2)   //case 3: ray bounce a single ray
        {
            if(NodeState = 1)//next bounce(s) evaluated
            {
                Get HitInfo uisng Reference
                Calculate DirectLight using HitInfo
                Color = ColorStack.Pop()
                Calculate IndirectLight using DirectLight, HitInfo, and Color
                ColorStack.Push(IndirectLight)
                if(NodeStack.Size>0)
                {
                    <NodeIndex,NodeState> = NodeStack.Pop()   //pop node
                }
                else //nodes stack empty
                {
                    break
                }
            }
            else //next bounce(s) not evaluated
            {
                NodeStack.Push(<NodeIndex,2>)//Push parent node
                Get ChildNodeIndex uisng Reference
               <NodeIndex, NodeState> = <ChildNodeIndex,1> //move to child node
            }
        }
        else //case 4: ray bounce two rays
        {
            if(NodeState = 1)//next bounce(s) evaluated
            {
                Get HitInfo uisng Reference
                Calculate DirectLight using HitInfo
                Color_1 = ColorStack.Pop()
                Color_2 = ColorStack.Pop()
                Calculate IndirectLight using DirectLight, HitInfo, Color_1, and Color_2
                ColorStack.Push(IndirectLight)
                if(NodeStack.Size>0)
                {
                    <NodeIndex,NodeState> = NodeStack.Pop()    //pop node
                }
                else //nodes stack empty
                {
                    break
                }
            }
            else//next bounce(s) not evaluated
            {
                NodeStack.Push(<NodeIndex,1>)//Push parent node
                Get ChildNodeIndex uisng Reference
                NodeStack.Push(<Reference,ChildNodeIndex>)//Push left child
               <NodeIndex, NodeState> = <ChildNodeIndex+1,1> //move to right child
            }
        }
    }
    Surface[i] = ColorStack.Pop();
}
```

Listing 5.9: EvaluateShadingPO Operator

In this method we traverse the first *N* shade trees corresponding to screen pixels in a post order manner, the roots of these tress are stored in first *N* locations in the nodes array. Listing 5.9 present the pseudo code for the post order traversal method. We use two stacks *ColorStack* to store the color of child nodes that have been evaluated and *NodeStack* to store the nodes that have not been evaluated yet. Since we traverse the shade tree nodes in post order manner we store in each *NodeStack* entry both the node index in the nodes array and *ShadeState* to represent state of the traversed node.

Node *ShadeState* can be either 0 or 1, a 0 value is the start state of a node and means that the node children have not been evaluated yet and a 1 value means that node children have been evaluated the shading color of current node is ready to be accumulated using the *ColorStack* and hit information. So given a node to be tested we check both nodes children and *ShadeState*. If node has one child node we check the *ShadeState* if it is 0 then we Push the node index

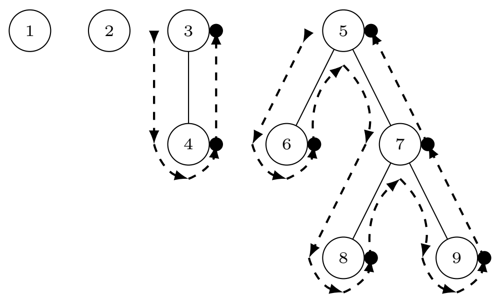


Figure 5.3: Shade tree post order traversal

and state 1 to the stack and if node state is 1 (which mean that the bounced ray hit has been evaluated) we evaluate its shading color as the surface reflectance multiplied by top color of *ColorStack* (which represents the color at the bounced ray hit). If the node has two children we check the *ShadeState* if it is 0 then we Push to the stack both the node index with state 1 and the left child node index with a state 0 and if the node state is 1 (which means that the two bounced rays' hits have been evaluated) we evaluate its shading color using the top two color in *ColorStack* weighted by Fresnel terms and multiplied by surface reflectance. If the node no children and represent a diffuse hit we calculate the shading color using the hit information of the node and Push it to the *ColorStack* and finally for nodes representing a ray miss we just Push a zero color to the *ColorStack*.

Final Rendering

This step is relatively simple, thanks to CUDA runtime which allow us to map device memory to OpenGL pixel buffer object (PBO). We just send the pixels final colors calculated in the shade function to the PBO which is rendered directly to screen.

5.2 Results and Dicussion

5.2.1 Ray Tracing Performance

Test Setup. All algorithms were implemented using CUDA programming language on a machine with an NVIDIA Geforce 285 GTX with 1 GB memory and a Core 2 Due processor running at 2.66 GHz. We store all data as dynamic lists in linear device memory allocated via CUDA. We also excluded the memory overhead time form our time computation. We store all data structures as structure of arrays (SoA). In our implementation we consider the number of threads per block is 256, and number of block varies according to processed data parallel primitives' size. All data parallel primitives were called form the CUDPP library [10].

Benchmark Scenes. We tested our algorithm on five publicly available scene which include small and moderate size scenes up to hundreds kilo triangles. The reference renderings are shown in Figure 6. Each scene has different complexity; the Fairy Forest model is 145 K. Triangles and represents a large scene with a sparse geometry, the Toasters model is 11 K. Triangles and represent a small scene with a sparse geometry, the Dragon model is 100 K. Triangles, the Cloth on Sphere model is 92 K. Triangles, and the Bunny/Dragon model is 92 K. Triangles, and each of the them represent an approximately uniformly distributed geometry. Our ray tracer support Phong shading and also support texturing.

Our rendered scenes represent the diverse light passes for a ray tracing algorithm. Figure 5.17-b represent a complex light pass through the Fairy Forest scene including reflection and transmission in three ray bounces. Figure 5.17-d represents a ray tracing through the Toasters model using reflection rays using two ray bounces and Figure 5.17-e represent a ray tracing using a transmission through a refractive surface in the Toasters scene using two ray bounces. Figures 5.17-a, 5.17-c, 5.17-f, 5.17-g, and 5.17-h represents a single bounce direct lighting of the Fairy, Dragon, Cloth, and Bunny/Dragon scenes.

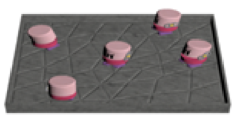
Model	**Average Tracing Time**	**Frame Rate**
Fairy Forest (Single Bounce)	50 ms	20 fps
Fairy Forest (Thee Bounces / Reflection, Transmission)	85 ms	11 fps
Toasters (Single Bounce)	26 ms	38 fps
Toasters (Two Bounces/Reflection)	40 ms	25 fps
Toasters (Two Bounces/Refraction)	41 ms	24 fps
Dragon (Single Bounce)	23.1 ms	43.1 fps
Cloth on Sphere (Single Bounce)	21 ms	30 fps
Bunny/Dragon (Single Bounce)	38 ms	26.3 fps

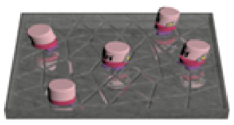
table 5.1: Ray tracing rendering time for our test scenes

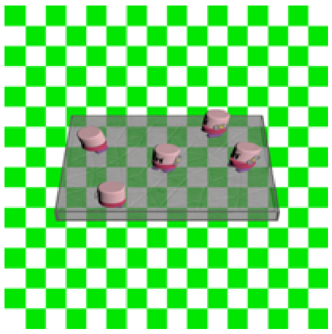
As illustrated in Table 5.1, all results are the average of 100 runs of the same model, the performance we achieve for primary and secondary rays for all benchmarks achieves real-time frame rates. We also noticed that the overhead for the secondary rays in both the Fairy Forest and Toasters models is low compared to the primary rays we believe that this is due our scenes contains a relatively small number of specular surfaces.

(a) Fairy Forest model

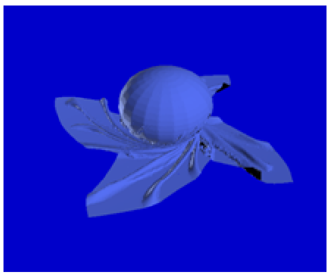
(b) Fairy Forest model (Reflection)


(c) Toasters model


(d) Toasters model (Reflection)


(e) Toasters model (Refraction)

(f) Dragon model


(g) Cloth on ball model

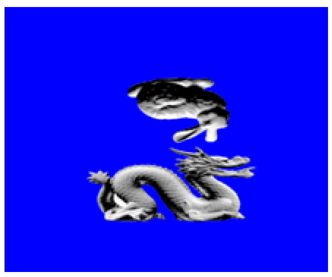

(h) Bunny/Dragon model

Figure 5.4: Our test scenes rendered on a 1024 X 1024 window using a GTX 285 device

6

Photon Mapping on GPU

6.1 Introduction to Offline Photon Mapping

Photon mapping (PM) [Jensen, 1996, 2001, 2004] is a two pass rendering algorithm that can efficiently produce a full range of global illumination (GI) effects including caustics and diffuse interreflections. In the first pass photon map(s) are constructed by tracing photons generated at each light source, and in second pass we use the photon map(s) to estimate the radiance at any point to produce various global illumination effects. In this section we will explain briefly the main rendering steps of the photon mapping algorithm.

6.1.1 The First Pass -Building Photon Map(s)-

In this pass a relatively large number of photons are emitted from each light source, traced throughout the scene, and stored in the photon map. Photons carry and propagate flux (packets of light energy) and are used simulate light energy transfer inside the scene. Technically a photon is represented by a position (3 floats), an incident direction represented as a sampled spherical direction inside a unit sphere (2 bytes), and a power represented as a color value (3 floats).

6.1.1.1 Photon Emission

Photons are emitted from the light source in directions similar to the PDF of light source emission. For example for a point light source photons are emitted uniformly in all direction. Each

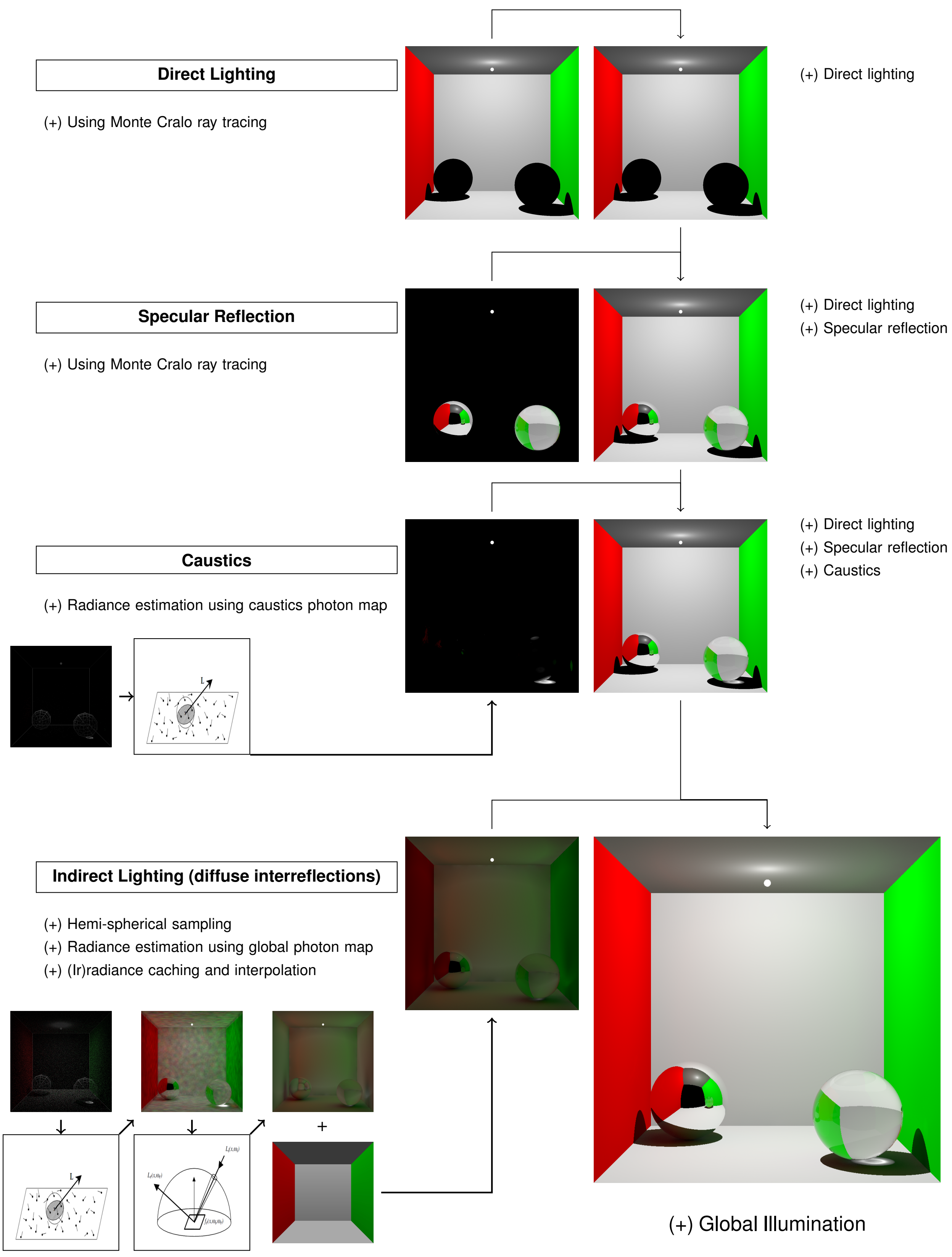

Direct Lighting
(+) Using Monte Cralo ray tracing
(+) Direct lighting
Specular Reflection
(+) Using Monte Cralo ray tracing
(+) Direct lighting
(+) Specular reflection
Caustics
(+) Radiance estimation using caustics photon map
(+) Direct lighting
(+) Specular reflection
(+) Caustics
Indirect Lighting (diffuse interreflections)
(+) Hemi-spherical sampling
(+) Radiance estimation using global photon map
(+) (Ir)radiance caching and interpolation
+
(+) Global Illumination

Figure 6.1: Main building blocks of the photon mapping algorithm

emitted photon carries the same power as of the light source and after photon tracing pass we scale the power of all stored photons by the total number of emitted photons. Since a photon may be stored several times during traversal, the total number of stored photons may differ from the number of emitted photons.

6.1.1.2 Photon Tracing and Storing

Once a photon is emitted it is traced into the scene in a way similar to ray tracing. When a photon hits a surface it can be reflected, refracted, or absorbed based on the surface material and a stochastic test using Russian roulette Arvo and Kirk [1990]; Jensen [2001]. A photon is stored in the photon map (e.g. a point based KD-tree) [1] only when it hits a diffuse surface. One advantage of the photon mapping algorithm is that we can use several photon maps to efficiently simulate various visual effects. For example we can use a global photon map that stores all photons to render indirect lighting due to diffuse interreflections, and we can use a caustic photon map which stores a photon when its immediate previous hit is a specular surface to simulate caustics effects, and to simulate participating media and volumetric rendering we use volume photon map together with ray marching. In our work we will ignore volumetric rendering and will concentrate on other global illumination effects produced using global and caustic photon maps on GPU.

6.1.2 The Second Pass - Rendering-

To render an image using photon mapping we use standard or distributed ray tracing and evaluate the reflected radiance at each intersection point as the sum of four distinct components (see figure 6.1):

1. **Direct lighting** which is calculated using standard Monte Carlo ray tracing.
2. **Specular reflection** which is calculated also using Monte Carlo ray tracing.
3. **Caustics** due concentrated reflected or refracted light rays and this term is calculated by radiance estimation in the caustic photon map.
4. **Indirect illumination** due to diffuse interreflections which is calculated efficiently using the global photon map in a more complex step called final gather.

[1]Other variations of the photon map structure include grid and hash grid see [Ma and McCool, 2002; Purcell, Buck, Mark, and Hanrahan, 2005] for more details.

6.1.2.1 Radiance Estimate

To estimate the reflected radiance at a given surface point we use the photon map to locate the nearest photons in a the upper hemisphere with a radius r centered at the this point. Then we calculate the reflected radiance $L_r(x, \vec{\omega})$ as the integration of the irradiance flux modulated by the surface BRDF using the equation:

$$L_r(x, \vec{\omega}) \approx \frac{1}{\pi r^2} \sum_{p=1}^{N} f_r(x, \vec{\omega_p}, \vec{\omega}) \Delta\phi_p(x, \vec{\omega_p})$$

Where f_r is the BRDF of the surface material (e.g. the alpedo ρ for lambertian surfaces). In this equation, the more photons in the radiance estimation, the better approximation at the cost of more computation effort. But later we will see that about $50 \sim 100$ photons is enough to get plausible results if we use a final gather step for indirect lighting.

6.1.2.2 Final Gather for Indirect Lighting

To evaluate incident radiance at certain point we can perform irradiance estimation directly in the global photon map but this will lead to blotchy spots in the rendering unless we use very large number of photons in the both the photon map and radiance estimation. Instead, a better way to calculate incident radiance is to perform a final gather step at each shading point. During this final gather step we sample a considerable number of rays (i.e. about $250 \sim 1000$ rays) in the upper hemisphere according to surface BRDF and trace these rays into the scene, once we found a hit [1] we perform irradiance estimation in the global photon map at this hit point. Finally, we calculate the incident irradiance as the average estimation returned from all rays hits.

6.1.2.3 Irradiance Caching

Even though with final gather the rendering time may be prohibitively impractical due to total number of final gather rays. Ward et al. [Ward, Rubinstein, and Clear, 1988] noticed that indirect lighting changes smoothly over surfaces and presented his key idea about radiance caching (RC) [2]. The main idea behind radiance caching is to store old values of indirect radiance in a hierarchical data structure (e.g. Octree) that allows fast range search and use interpolation when possible to calculate further new queries of radiance values when possible. Unsurprisingly the

[1] In this case we still search for the nearest hit in scene.

[2] In fact the radiance caching term was later coined after the original publication appeared.

same technique can be applied directly to the irradiance calculation with photon mapping which significantly reduces the render time making the whole photon mapping algorithm more practical [Gautron, Bouatouch, and Pattanaik, 2006; Jarosz, Donner, Zwicker, and Jensen, 2008; Jarosz, Zwicker, and Jensen, 2008; Křivánek, Bouatouch, Pattanaik, and Žára, 2008] [1].

6.2 Parallel Photon Mapping on GPU

In this section we explain the main steps of the parallel photon mapping algorithm on GPU. The entire pipeline is implemented on GPU using NVIDIA CUDA NVIDIA [2010]. Figure 6.2 shows the flow chart of the algorithm using the Cornell box scene. The input to the algorithm is the scene description including objects, materials and light sources and the output is a globally illuminated rendered scene.

Given an input scene we begin by building a hierarchical tree for the scene using any of the methods explained in chapter 4. We use the resulting scene tree for both ray tracing as explained in chapter 5 and photon tracing (Section 6.2.1). The output of the ray tracing pass are the rays trees , and the output of the photon tracing pass are two arrays of photons one for global photons and the other for caustics photons, each of these arrays is seeded to a point based KD-tree builder (Section 6.2.2). We use the rays trees structure to get all the shading points that are directly or indirectly seen form the viewer and select the initial seed points for irradiance samples positions (Section 6.2.3.1) and then we perform final gathering to calculate irradiance samples which are seeded to KD-tree builder to build the irradiance tree (Section 6.2.4). Finally we accumulate shading (Section 6.2.5) using ray tracing for direct lighting, irradiance estimation for caustics and irradiance interpolation for indirect lighting using the irradiance tree.

6.2.1 Parallel Photon Tracing

```
//Global Data
Photons(Pos, Dir, Flux, Length)
N                          // number of emitted photons
Rays(Org, Dir, Flux)       // traced rays
Hits(Dist, Tri, U, V)      // hit info
Store, StoreScan           // ray flag used to mark photon hit diffuse surface
Bounce, BounceScan         // photon flag mark a photon at the ray to be bounced
Scene                      //  scene tree and primitives

Procedure PhotonTracer()
```

[1]We refer the readers for the SIGGRAPH course notes ”Practical Global Illumination Using Irradiance Caching” [Křivánek, Gautron, Ward, Arikan, and Jensen, 2007] for more details about radiance caching with photon mapping.

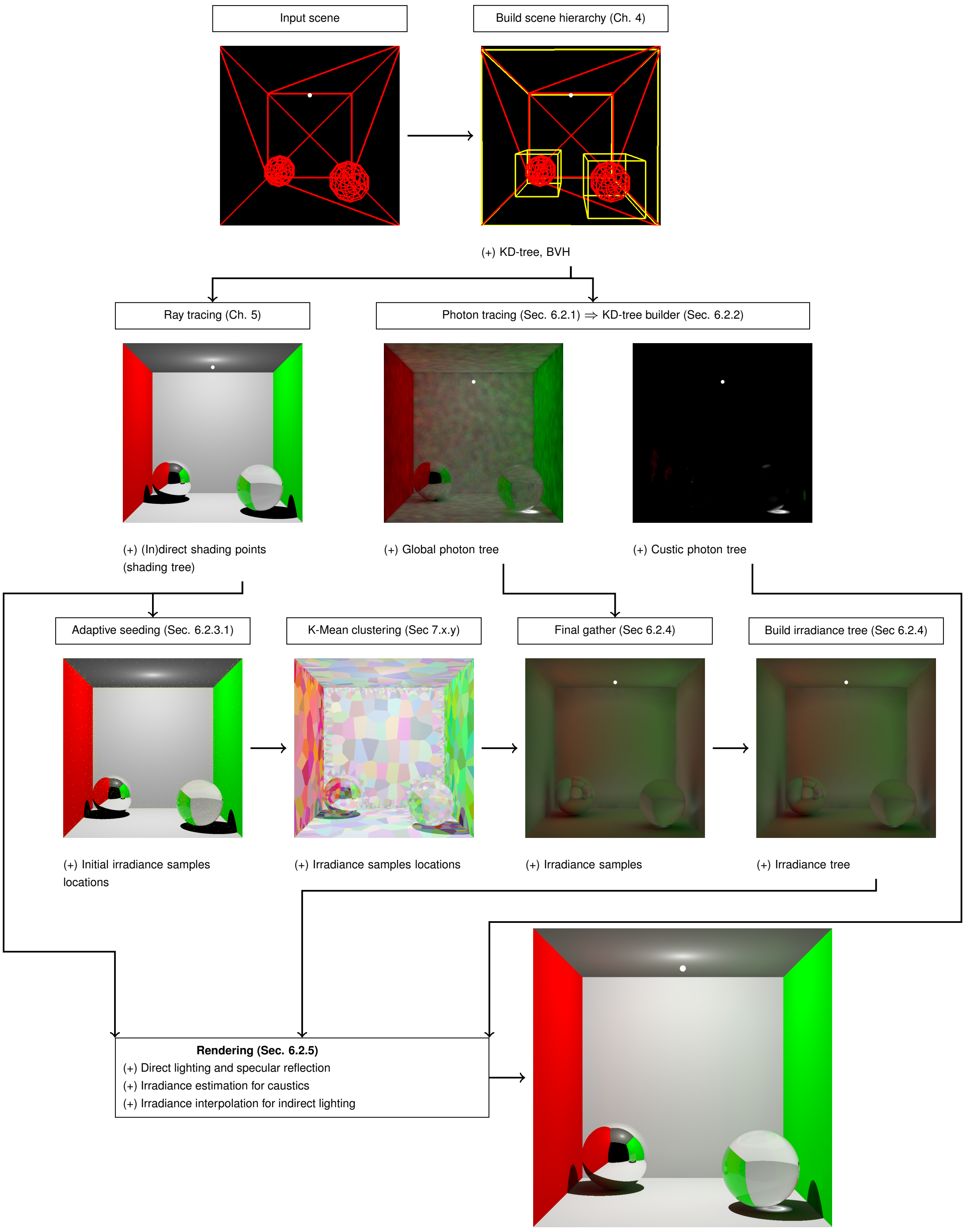

Input scene
Build scene hierarchy (Ch. 4)
(+) KD-tree, BVH
Ray tracing (Ch. 5)
Photon tracing (Sec. 6.2.1) ⇒ KD-tree builder (Sec. 6.2.2)
(+) (In)direct shading points (shading tree)
(+) Global photon tree
(+) Custic photon tree
Adaptive seeding (Sec. 6.2.3.1)
K-Mean clustering (Sec 7.x.y)
Final gather (Sec 6.2.4)
Build irradiance tree (Sec 6.2.4)
(+) Initial irradiance samples locations
(+) Irradiance samples locations
(+) Irradiance samples
(+) Irradiance tree
Rendering (Sec. 6.2.5)
(+) Direct lighting and specular reflection
(+) Irradiance estimation for caustics
(+) Irradiance interpolation for indirect lighting

Figure 6.2: Main building blocks of GPU photon mapping algorithm

```
{
    // emit photons
    EmitPhotons<N>()
    NumTracedPhotons = N

    in = out = 0 // address for input/output rays in rays array
    pp = 0 // ping-pong variable

    // simulate photon bouncing
    for(int i=0; i<NumBounces; i++)
    {
        pp = 1-pp
        in = out
        out = pp * NumTracedPhotons

        // tracing photons
        TracePhotons<NumTracedPhotons>(&Rays[in])

        Scan(<StoreScan, BounceScan>, <Store, Bounce>, NumTracedPhotons)
        ScanTail(<NumStore, NumBounce>, <StoreScan, BounceScan>, <Store, Bounce>, NumTracedPhotons)

        // storing photons
        StorePhotons<NumTracedPhotons>(&Rays[in])
        Photons.Length += NumStore

        // bouncing photons
        BouncePhotons<NumTracedPhotons>(&Rays[in], &Rays[out])
        NumTracedPhotons = NumBounce
    }
    // Scale all photons power
    Photons.Flux /= N
```

Listing 6.1: Parallel Photon tracing algorithm

The parallel photon tracer is shown in Algorithm 6.1. The algorithm takes as input scene tree and light sources and generates and traces N photons for a relatively small number of bounces (e.g. 5 bounces). In this algorithm and for simplicity we assume that we have only one light source and it is trivial to adapt the algorithm to consider more light sources. We also deal with point light sources in our work and dealing with other kinds of light sources such as area lights remains an open problem for future work.

Data Structures.
Output Data Structures. The output of the algorithm is *Photons* array including photon position, incident direction, and irradiance flux.
Transient Data Structures. To simulate photon tracing we need *Rays* array which stores photons positions and incident directions and irradiant flux, this array will be used as an input/output buffer in a ping-pong fashion and has length equals *2N*, where N is the initial number of traced photons. For the traced rays we temporary need an array *Hits* to store hit information inclding hit distance *(T)* and primitive data *(i.e. triangle id and barycenter coordinates of the triangle)* for each ray hit, this array has length equals *N*. We need an array of flags *(Store)* which store 1 at index i if the traced photon at index i hits a diffuse surface and 0 otherwise, and another array of flags *(Bounce)* which store 1 at index i if the traced photon at index i will

be reflected or transmitted and 0 if it will be absorbed. We store the type of photon reflection (diffuse reflection, mirror like reflection, or transmission) temporary in the higher 2 bits of the *Tri* field of the Hits struct. We prepare two other arrays *(StoreScan , BounceScan)* for the scans of the *Store* and *Bounce* arrays. Since each photon have only one choice of reflection or refraction each of the *Store* and *Bounce* arrays and their scans are of length equals N.

We create an operator **EmitPhotons** that fill in the *Rays* array. In this operator we store the light source position in the ray origin and set the ray direction according to the BRDF of the light source (for point light source it is uniformly sampled over a sphere) similar to Jensen [2001] and store in the flux the light source power. We call this operator to create N (e.g. 100k $\sim$ 400K) photons in parallel.

```
Operator EmitPhotons()
{
    Input: light source(s) // light source
    i = ThreadIndex
    Set Rays.Pos[i], Rays.Dir[i], Rays.Flux[i] using light source properties
}
```

Listing 6.2: Emit photons

Then we iterate for a number of bounces *(NumBounces)* and traverse photons, store photons hit diffuse surfaces in parallel, and generate new bounced photons in parallel to be used in the next iteration.

When tracing photons (see *TracePhotons* operator); if a photon hit a surface we temporary store hit information into *Hits* struct, and if the hit surface is diffuse we store 1 in *Store* array and 0 otherwise, and perform a stochastic test to specify the next event of the current photon [Arvo and Kirk, 1990; Jensen, 2001]; if the photon will be bounced we store 1 in the *Bounce* array and 0 otherwise, we also temporary store the next event in the 2 most significant bits in the *Tri* field *([00] for diffuse reflection, [01] for mirror like reflection, and [10] for transmission)*.

```
Operator TracePhotons(Rays rays[])
{
    input: SceneTree            // scene tree (e.g. BVH, KD-Tree)
    i = ThreadIndex
    Search between Rays[i] and SceneTree and store result in Hits[i]
    if(Hits[i].Tri != -1)
    {
        BRDF = Get surface BRDF of trainagle Hits[i].Tri
        Store[i] = (BRDF is diffuse)? 1 : 0
        P = Get averaage reflectance of trainagle HitInfo.TriId
        Bounce[i] = (RandD() < P)? 1 : 0
        if(Bounce[i] == 1)
        {
            Store next photon event in Hits[i].Tri$_{bits[31,30]}$
        }
    }
    else
    {
        Store[i] = 0
        Bounce[i] = 0
    }
}
```

Listing 6.3: Trace photons

After tracing the photons we perform an exclusive scan to each of the *Store*, and *Bounce* arrays into *StoreScan*, and *BounceScan* arrays respectively. We use the *Store* array and its scan to compact and append current photons into *Photons* array, we get stored photon data using information stored in both the *Hits* and *Rays* strcuts (see *StorePhotons* operator). We also use the *Bounce* array to generate and compact new bounced photons for the next iteration, first we check the flag in the *Bounce* array if it is set we get the next action of the photon form the 2 most significant bits in the *Tri* field of the *Hits* struct and according to this event we create the new photon in a compact form using the scan of the *Bounce* array, we get the new photon data using information stored in both the *Rays* and *Hits* arrays (see **BouncePhotons** operator).

```
Operator StorePhotons(in:Rays)
{
    i = ThreadIndex
    if(Store[i] == 1)
    {
        address = Photons.Length + StoreScan[i]
        Fill Photons.(Pos, Dir, Flux)[address] using Rays[i], Hits[i]
    }
}
```

Listing 6.4: Store photons

```
Operator BouncePhotons(in:Rays, out:RaysO)
{
    i = ThreadIndex
    if(Bounce[i] == 1)
    {
        address = BounceScan[i]
        Fill RaysO[address] using Rays[i], Hits[i]
    }
}
```

Listing 6.5: Bounce photons

6.2.2 Building Photons KD-Tree on GPU

The construction algorithm begins with an array of photons and ends with a point based KD-tree that is either stored in a linear array of nodes or a compact preorder form Zhou et al. [2008]. Similar to the primitives KD-tree each internal node store both the split plane and split position and references the indices of its left and right child nodes and each leaf node references a set of photons in the photons array. We allow an exception for empty nodes which are treated as leaf nodes reference nothing in the photons array. For efficient construction we follow the method used in Wald, Günther, and Slusallek [2004]; Zhou et al. [2008] and begin by sorting photons in each coordinate axis and use this sorted sequence to efficiently split nodes.

In this section we will explain the main processing steps of the parallel point based KD-tree construction algorithm on GPU, the construction pipeline is divided into four stages: (1) initialization stage in which we sort the photons and create the root node and initialize its related data; (2) processing large nodes stage in which we split large node; (3) processing small nodes in which we split small nodes; and (4) KD-tree output stage in which we reorganize the tree nodes storage to reflect a preorder traversal layout. Similar to the primitives KD-tree we use a photons threshold T (e.g. 32) to distinguish large and small nodes.

Data Structures:

Input Data Structures. The input to the algorithm is an array of $3D$ floats resenting positions (i.e. photon positions) which are stored in a SoA format.

Output Data Structures. The output of the algorithm is linear array of tree nodes where each internal node stores the split plane and references to child nodes and each leaf node store references to the set of photons contained in this node implicitly by a *Start* index and a *Size* to reference a contiguous set of photons in the photons array, we optionally store in each node its bounding box. Following the efficient node storage presented in Wald [2004] the node structure is stored in 8 bytes (two 4 bytes integer words), and at any time during the algorithm we use the two most significant bits *([31,30])* of the first word of the node to distinguish internal and leaf nodes as follow:

- If the node is a leaf node, we store $[11]$ as a leaf node flag.
- If the node is an internal node, we store the split axis into these two bits.

The other bits are managed according to the current stage of the algorithm as follow:

- During the large nodes stage each internal node uses the 30 least significant bits of the first word to store the reference index of its left child node, and store the split position in the second word, and each leaf node uses the 30 least significant bits of the first word to store the index of the node's first photon index the in the photon array, and uses the second word store the number of photons in this node.
- During small nodes stage internal node remain as of large nodes stage, but for leaf node we use the 30 least significant bits of the first word as an index to another array of small roots nodes, where each small root store at most T photons references, the second word is used as a bitwise mask to indicate the node ownership of at most T photons referenced in the corresponding small root.

During large or small node stage we always store the two children of an internal node beside each other in the node array so that a single reference in the parent node is enough to access both children.

In the final tree layout we reorganize the tree nodes in linear integer array in which we store nodes and their related photons together. Each internal node data still stored in 2 words where bits $31, 30$ for the first word are used to store the split axis, and bits $[29 - 0]$ are used to store the index of the right child node, and the left child node is accessed implicitly since it is stored directly after the parent node. Each leaf node of size n is stored in $1+n$ words, in the first word we store $[11]$ in bits $[31, 30]$ as an leaf flag and use the remaining 30 bits to store the number of photons inside this nodes, then we follow this word by the indices of node's photons in the photons array.

Transient Data Structures. We will explain transient data structures as we explain the processing steps of construction the algorithm.

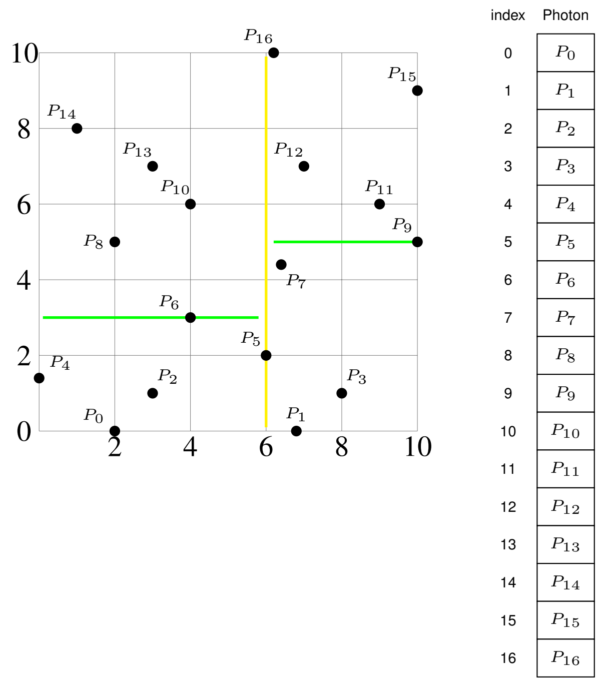


Figure 6.3: Photon positions and photons array

6.2.2.1 Initialization Stage

In this stage we create the root node and the *association lists* which reference all photons in a sorted order.

Creating Root Node. We create the root node as a leaf referencing all the photons by setting the leaf flag and storing 0, N in the start node index and size respectively (where N is the total number of photons), then we add this node to the *ActiveNodes* array. We also maintain an indirection array for all the photons that records the parent node index in the *ActiveNodes* array for each photon.

Creating Association List. First we make three association lists of size N one for each coordinate axis, we fill these lists in parallel using a sequential indices to refer to all non-sorted input photons . The we use the sort utility to sort the photons in each coordinate axis employing a temporary copy of the photon positions array as the values, and the indices array as the keys; now the indices arrays reflect a sorted sequence of the photons in each axis and at any time a leaf node can get its child photons by any of these arrays and for any node the three arrays always refer to the same set of photons but in a different order. We concatenate these indices arrays into a single array of size $3N$.

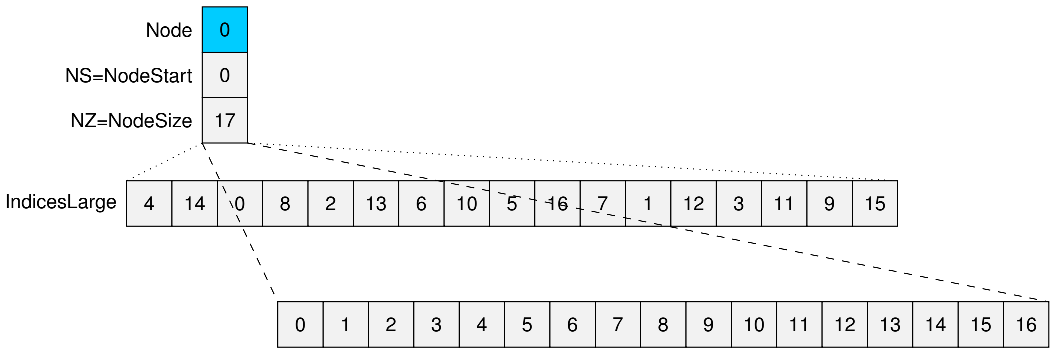


Figure 6.4: Initializing the association list

We choose to sort the indices using a temporary copy of the photons array for two reasons: first, this will save us a large number of value swaps during the entire algorithm to keep the photons sequence always in order; second, the photon structure always store other data like the incident direction and irradiance flux so it is better to keep the photons array always aligned and reflect the sorted other by the association lists.

During the algorithm it is easy for a leaf node to get its child photons using the *IndicesLarge* array by accessing the photon index and then reading the photon data from the photons array.

We prepare for the *IndicesLarge*, other two empty copies, the first is used as an output buffer for the next large nodes and used in a ping-pong fashion with the initial arrays and the second is used for small nodes which is accumulatively filled during large node stage.

As a final step in the initialization stage we initialize the inherited bounding box of the root node with scene AABB, and this is trivially done using the first and last values of the sorted sequence of photons for each axis, the minimum photon values are accessed by indices $0, N, 2N$ and the maximum photon values are accessed by indices $N-1, 2N-1, 3N-1$.

6.2.2.2 Large Node Stage

In this stage we perform the following main steps: (1) compute the bounding box of all nodes in an $O(1)$ step; (2) split nodes ; (3) sort photons to child nodes based on the median photon; (4) filter child nodes into large and small nodes; (5) distribute photons to child nodes and either we return to step 1 if we still have large nodes or go to next stage if not. As in the primitives KD-tree we use simple and inexpensive heuristics based on empty space maximizing and object median for all large node splits.

Compute Nodes Bounding Box. To create the AABB for current large nodes we make an operator that process each node in the *ActiveNodes*. In this operator we get the *start index* and the *size* of the nodes, and calculate the *lasts index* as *start index + size - 1*, we calculate the node AABB using the positions of the photons referenced by the *start index* and the *last index* in photons array with shift 0 for the x-axis, shift N for the y-axis, and shift $2N$ for the z-axis (where N is the number of phtotons in large nodes).

Nodes Splitting. To account for both "empty space" maximization and object median as the splitting criteria for large nodes we keep two bounding boxes for each node; the tight bounding box *(B_t)* calculated in the previous step, and the inherited bounding box *(B_i)* which recursively inherits split planes from its parent nodes starting with the scene AABB at the root node.

We calculate the "empty space" as the difference between the two bounding boxes at each of their 6 side planes. If the "empty space" at certain side is greater than a predefined threshold C_e relative to the corresponding axis then we split this node at the tight bounding box position into two child nodes; an empty node and an inherited node which inherits both the parent primitives and the tight bounding box B_t.

To parallelize this step we make an operator that processes all nodes in parallel. In this operator we count for each node the number of sides that pass the "empty space" threshold and store this count into array *NumEmptyNodes*, we also make a 6 bits mask for each node into which we set bit i if the corresponding side passes the "empty space" threshold, where $i \in [0-5]$ and corresponds to min and max sides for the *x*, *y*, and *z* axes, we store these masks into array *EmptySides*. After calling this operator we perform an exclusive scan to the *NumEmptyNodes* array using the scan utility (see Figure 6.5).

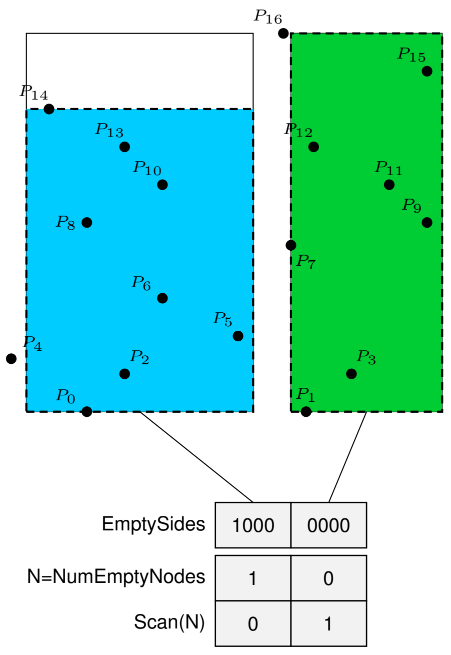


Figure 6.5: Empty space calculation

Once we scanned the number of empty nodes we make an operator that splits all nodes in parallel. In this operator, given a node N at index i in the *AvtiveNodes* array we get the start address of its children using the scan of the *NumEmptyNodes* array as $2Scan(NumEmptyNodes)[i]+2i$ and use the corresponding bit mask in the *EmptySides* array to create empty and inherited nodes and then we split the lowest inherited node at the object median of the longest axis into two child nodes and store them into positions $2i$, $2i + 1$ in the *ChildNodes* array (see Figure 6.6). In this step we fill both the *ChildNodeSize* array in which we store the size of each child node by the median values, and the *Large* flags array in which we store 1 if the corresponding child node size is greater than triangles threshold *(T)*, and 0 otherwise (both arrays equal in size to the *ChildNodes* array).

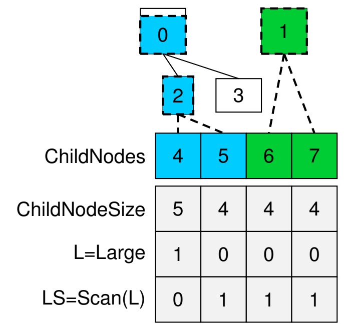


Figure 6.6: Large Nodes Split

Since we process nodes in BFS order an later in the tree output stage we will need access all the nodes level by level, we found that it is too hard to count the tree levels resulted from the incoherent empty nodes pattern. So we choose to record the nodes level by level in each iteration by considering the parent nodes and to record internal inherited and empty nodes we keep trace of two other variable for each node: first, we record the lowest inherited or empty node index; second, we record the number of inherited nodes and whether the inherited node is the left or the right child, this information can be packed into a single integer value where we store the number of inherited node N_h in the lowest significant 3 bits where $N_h \in [0, 6]$ and store in the next significant bits an N_h bits mask where bit i is set if the inherited node appear as a left child and cleared if it appear as a right child. Later, we will see that this information is enough to access all nodes in between parent and child nodes.

Photons Sorting. We create an arrays *PhtotonLeft* which is aligned with the photons SoA. We make an operator *SortPhoton* which classifies each photon to the left or the right child node. In this operator we get parent node split axis and object median and access the sorted photon at this access from the *IndicesLarge* array; if the photon is below than the object median then we store 1 in the corresponding value in the *PhtotonLeft* array, otherwise we store 0. Then we make another operator that processes each $3N$ elements in the *IndicesLarge* array and fill an array of flags of size $3N$; in this operator we read the photon index the *IndicesLarge* array and get the photon value in the *PhtotonLeft* array, and store this value in the corresponding value in the *Left* array.

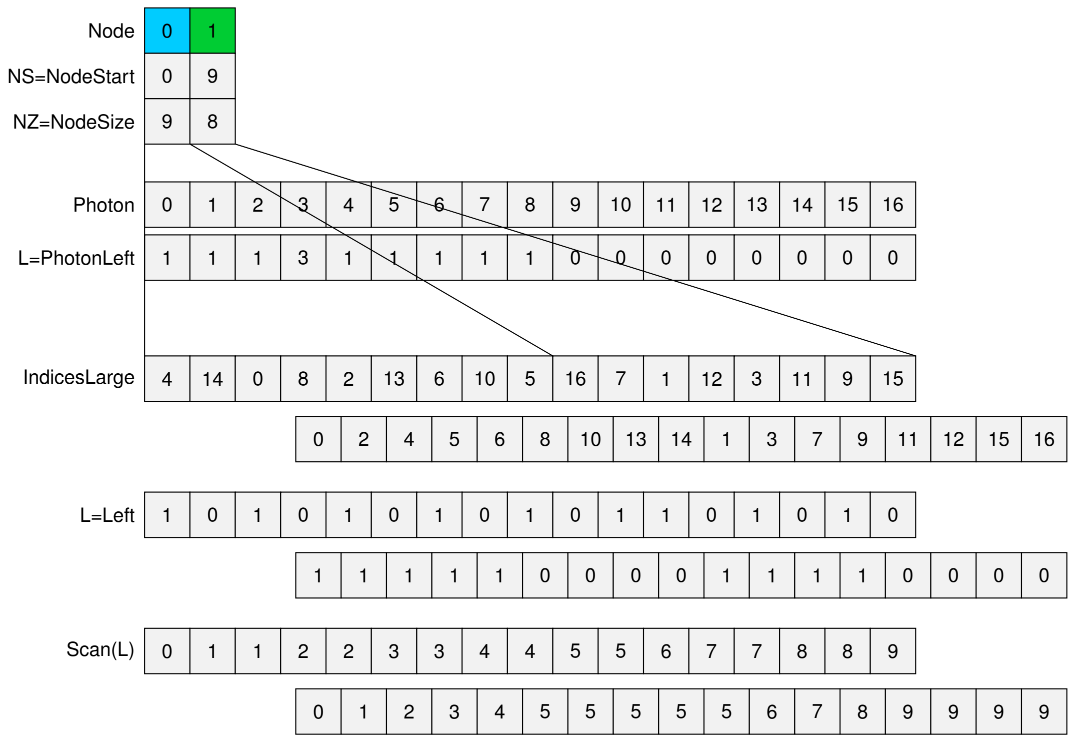

Node 0 1
NS=NodeStart 0 9
NZ=NodeSize 9 8
Photon 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
L=PhotonLeft 1 1 1 3 1 1 1 1 1 0 0 0 0 0 0 0 0
IndicesLarge 4 14 0 8 2 13 6 10 5 16 7 1 12 3 11 9 15
0 2 4 5 6 8 10 13 14 1 3 7 9 11 12 15 16
L=Left 1 0 1 0 1 0 1 0 1 0 1 1 0 1 0 1 0
1 1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0
Scan(L) 0 1 1 2 2 3 3 4 4 5 5 6 7 7 8 8 9
0 1 2 3 4 5 5 5 5 5 6 7 8 9 9 9 9

Figure 6.7: Photons sorting to child nodes

Large/Small Nodes Filtering.

We call the *Split* utility employing *Large* array as the *Flags* array to split *ChildNodes* into *NextNodes* and *SmallNodes*, and split *ChildNodesSize* into *NextNodesSize* and *SmallNodesSize*. Then we perform an exclusive scan for both *NextNodesSize* and *SmallNodesSize* arrays to get *NextNodesStart* and *SmallNodesStart* arrays respectively (see Figure 6.8).

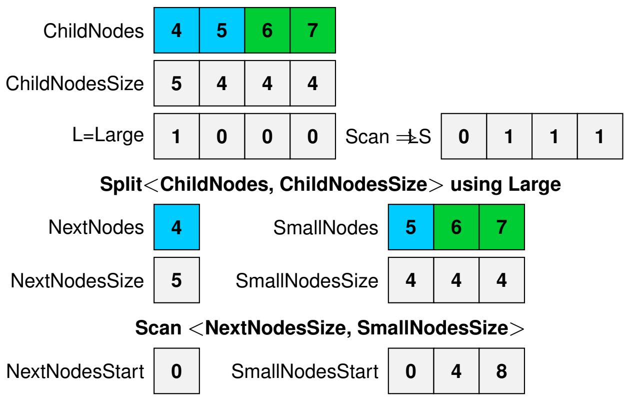


Figure 6.8: Large/Small Nodes Splitting

Photons Distribution. Consider a photon *p* at index i in the sorted association list, in parent node *N* with a start index $Satrt_N$, and a size $Size_N$ which split into a left child node N_L with a start index $Start_L$, and a right child node N_R with a start index $Start_R$. Using *Left* array a photon *(p)* may be: (1) sorted to left child node *(Left[i] = 1)*; (2) sorted to right child node *(Left[i] = 0)*. And the child node (N_L*, or* N_R) may be classified as a large or a small node and accordingly its photons must be stored in its corresponding photons list.

To sort a photon index we start by finding its respective axis A as $i\%N_{old}$ (where N_{old} is number of photons in the *IndicesLarge* array) and new parent node N_L, or N_R and the corresponding photons list and the node start address, then we sort the photon at an offset defined by the local scans of *Left* arrays relative to its new parent start address. We define the local node start for the respective access Start$_{AN}$ as axis * N+ Start$_N$ and 2 local offset addresses for a photon *p*: (1) the number of *ones* preceding it in *Left* array ($Offset_L$) which is calculated as *Scan(Left)[i]* - *Scan(Left)[*$Start_{AN}$*]*; (2) the number of *zeros* preceding it in *Left* array ($Offset_R$) which is calculated as *i* - $Start_{AN}$ - *Scan(Left)[i]* - *Scan(Left)[*$Srart_{AN}$*]*.

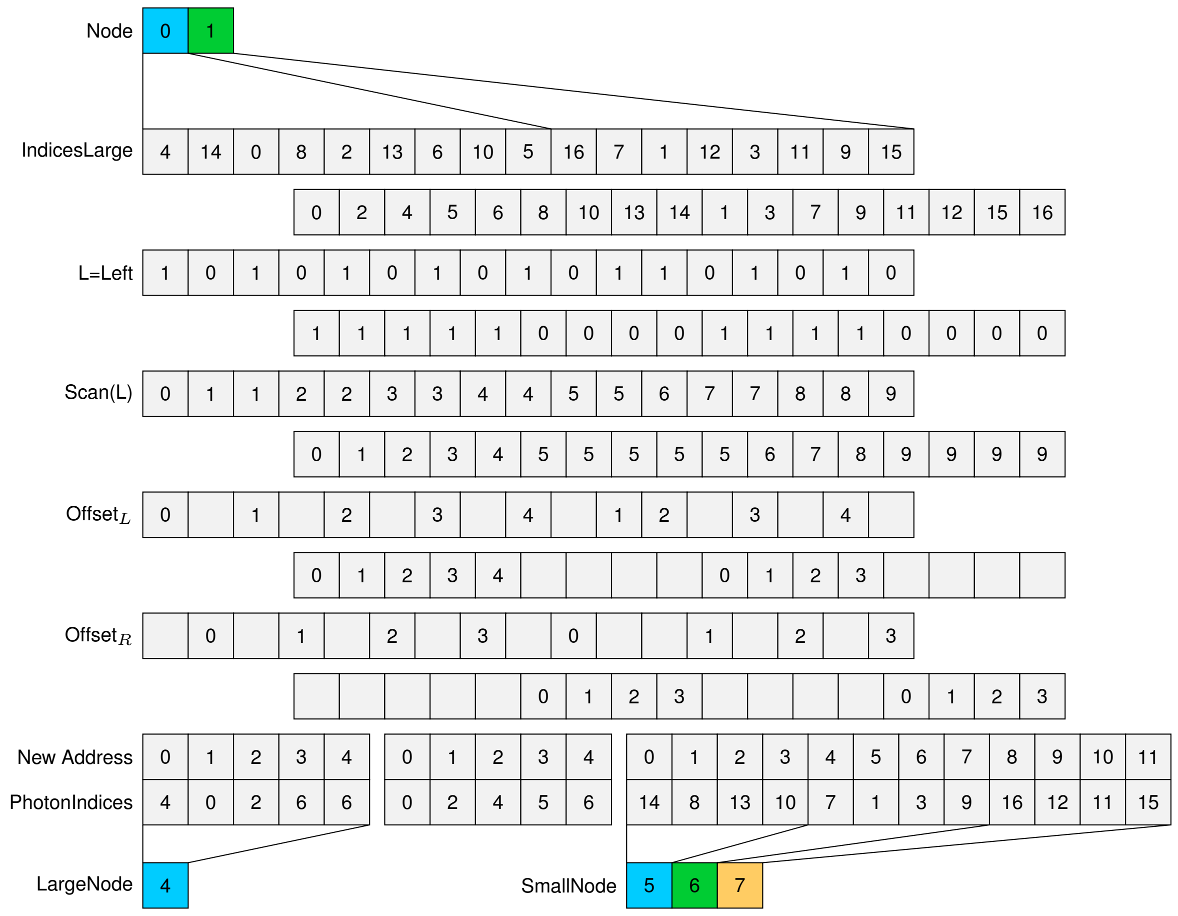


Figure 6.9: Photons relocation

We can distinguish 4 distinct cases for photon relocation, considering N_{new} is number of photon in the next array which is calculated as the scan tail of the *NextNodeStart* and the *NextNodeSize* arrays:

1. Photons p goes to Left child node (N_L) where:

   (a) N_L is a large node; then we store p at index $N_{new} * A + Start_{NL} + Offset_L$ in the *NextNodesPhotons* array.

   (b) N_L is a small node and A is 0; then we store p at index $Start_{NL} + Offset_L$ in the *SmallNodesPhotons* array.

2. Photon p goes to right child node (N_R) where:

   (a) N_R is a large node; then we store p at index $N_{new} * A + Start_{NR} + Offset_R$ in the

NextNodesPhotons array.

(b) N_R is a small node and A is 0; then we store *p* at index *Start*$_{NR}$ + *Offset*$_R$ in the *SmallNodesPhotons* array.

Similar to photons sorting we create an operator which handle the previously stated cases for photon relocation and sort all photons in parallel. In this operator we also fill in the corresponding *Refelction* arrays based on the new photon index and new parent node type and update the photon parent node index with the child node index, note that a photon is sorted to a small node we only need the indices of the x-axis since in small stage we have no need to keep the respective sorted order in all axes.

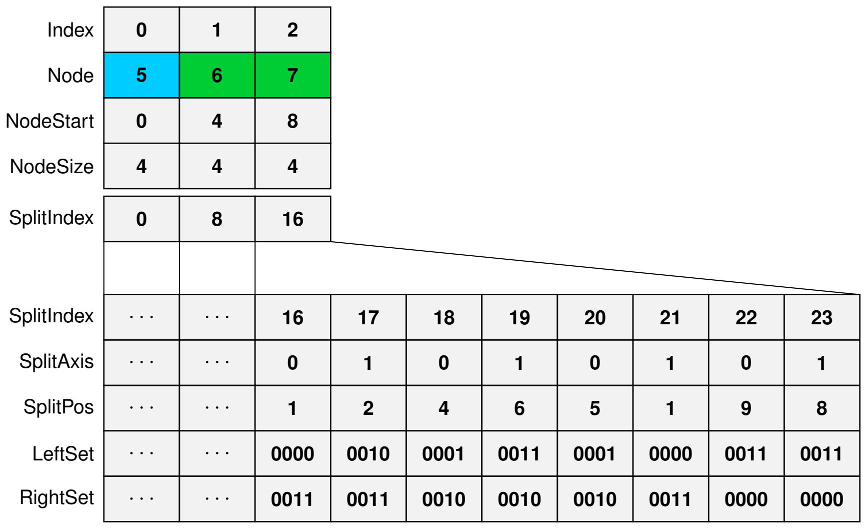
6.2.2.3 Small Node Stage

In this stage we still process nodes in BFS order and employ VVH as cost estimation for node splits. We begin with a preprocessing step in which we enumerate all split candidates in each node. Then we perform the following steps: (1) evaluate the VVH at all split candidates in each node and select the minimum value using standard reduction; (2) compare the minimum VVH with the cost of not splitting the node and either we split the node and sort photons to child nodes or mark the node as a leaf and return to step 1 if we still have new child nodes.

Preprocessing Small Nodes. In this step we prepare small roots which correspond to all small nodes with a large parent node. Each small root defines the start photon address index in the photons list and the number of the photons *N*$_p$ (where $0 < N_p \leq T$) and lists all split candidates defined by the 3 coordinates of each photon. The split candidate is defined by split axis, split position, left and right photons sets where each photon set is represented using a *T* bits mask Zhou et al. [2008].

To create small roots we prepare three arrays: *SplitPosition*, *LeftSet*, and *RightSet*, (each of these arrays are of size equals $3N$, where N is the total number of photons) into these arrays we store a split position, a *T* bits mask of the left set, and a *T* bits mask of the right set for each split candidate.

We make an operator that works on all photons in parallel. In this operator, given a photon p at index i we store its 3 coodinate positions and left and right triangles sets staring at index $3i$. The left and right triangles sets are created by comparing the candidate position against all other photons in the node.

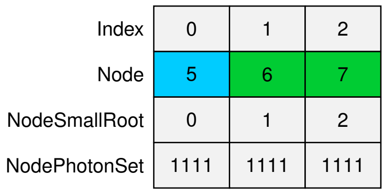


Index	0	1	2
Node	5	6	7
NodeStart	0	4	8
NodeSize	4	4	4
SplitIndex	0	8	16

SplitIndex	···	···	16	17	18	19	20	21	22	23
SplitAxis	···	···	0	1	0	1	0	1	0	1
SplitPos	···	···	1	2	4	6	5	1	9	8
LeftSet	···	···	0000	0010	0001	0011	0001	0000	0011	0011
RightSet	···	···	0011	0011	0010	0010	0010	0011	0000	0000

Figure 6.10: Small roots creation

We modify the small nodes structure a little bit; Instead of defining the photons of a node by the start photon index and number of photons, we define them by an index to the small root and a T bits mask *(NodeMask)* representing the photons set in the node relative to small root start index, similar to small roots, for a node with n photons this mask is initialized to $(1 << n) - 1$.



Index	0	1	2
Node	5	6	7
NodeSmallRoot	0	1	2
NodePhotonSet	1111	1111	1111

Figure 6.11: Small nodes modified structure

Evaluating Node Cost. We make an operator that processes *3T* splits candidates (s_i) for each small node in parallel where $(s_i \in [0, 3T))$. In this operator we get the node's photons set and if a split s_i exist in the photon set (bit $s_i/3$ is set to 1 in the *PhotonSet*) then we get split position *(p)*, left photon set *(LeftSet)* and right photon set *(RightSet)* form the small root, and the split axis *(a)* is defined implicitly by the split index *(s_i%3)*. Then we evaluate the VVH cost as:

$$VVH_{si} = C_{ts} + \frac{C_L(s_i)V_L(s_i)}{V} + \frac{C_R(s_i)V_R(s_i)}{V}$$

Where C_{ts} is the cost of node traversal, $C_L(s_i), C_R(s_i)$ are the number of photons lying to the left and the right of the split, $C_L(s_i)$, and $C_R(s_i)$ are calculated using the bit count of the *LeftSet ANDED* with *PhotonSet* and bit count of the *RightSet ANDED* with *PhotonSet* respectively, $V_L(s_i), V_R(s_i)$ are the volume of the child AABBs resulting from splitting the parent node AABB with the split plane defined by *(p,a)*. We customize the reduction utility to reduce $3T$ values of each node and return the minimum VVH value and its corresponding split index, then we make a nested parallel call to this utility to process all nodes in parallel.

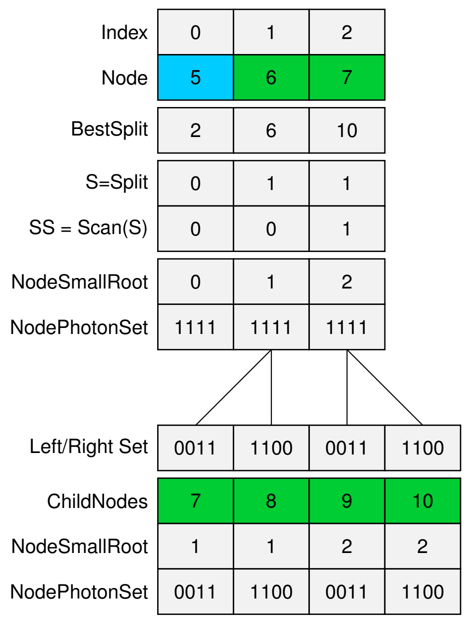


Figure 6.12: Small nodes split

We prepare an array *NodeSplit* (the size of this array equals current number of nodes), into this array we store 1 if corresponding node will be split and 0 otherwise. Once we calculated the minimum SAH for each node; we compare it with the cost of not splitting the node and fill the corresponding values in *NodeSplit* array. Then we perform an exclusive scan to this array to use it in creating new child node.

Node Split and Photons Sorting. This step is relatively simple; we check the flag in the *NodeSplit* array and if it is set then we use the split index calculated in the previous step to get split position, axis, *LeftSet* and *RightSet* form the small root. The two child nodes are created at an address defined by the scan of the *NodeSplit* array multiplied by 2. For the child nodes we set the *SamllRoot* index as of the parent node value, and set the photons set of the left child node as the bit wise *AND* between *LeftSet* and *PhotonSet* and set the photons set of the right child node as the bit wise *AND* between *RightSet* and *PhotonSet*. We also mark the parent node as an internal node and set its child references to refer to the newly created child nodes.

$$\eta = \alpha\|x_i - x_k\| + \sqrt{2 - 2(\vec{n_i} \cdot \vec{n_k})} \tag{6.1}$$

6.2.3 Irradiance Estimation

To calculate irradiance due to indirect lighting we perform the following steps:

- Estimate initial sampling locations for irradiance samples using normals and positions of all shading points.
- Refine sampling locations using k-means clustering algorithm.
- Estimate irradiance samples using final gather rays.
- Build a hierarchy (e.g KD-Tree) for irradiance samples to be used later for irradiance interpolation.

6.2.3.1 Selecting Irradiance Samples

We use the adaptive seeding algorithm presented in [Wang et al., 2009] [1] to select *K* initial sampling locations for irradiance caching by building a static screen space quadtree [Gargantini, 1982] for all shading points. The input to the algorithm is a list of all diffuse hits that are directly or indirectly seen form the viewer, this list if hits include hit position and hit normal and corresponding screen pixel. We used the *DiffueNodes* array to get all hits data from the rays trees, but with a simple modification to the ray tracing algorithm to add the pixel location to *DiffuseData* array (see chapter 5). Other inputs to the algorithm includes the number of levels in the quadtree *(L)*. The output of the algorithm is the initial *K* candidate positions and normals for irradiance caching samples.

We used the fact every hit has a one to one corresponding to screen pixels to build linkless a screen space quadtree for all hits. Note that our quadtree does not use any hashing methodology like [Choi, Ju, Chang, Lee, and Kim, 2009], instead we create a *2L* bits Morton code for each hit point extracted from the screen location of the hit. This code will be used to implicitly define the hit point to parent node relation in the quadtree. And to define the parent node to hit point relation first we use the parallel sort, and sorted range extraction to define the relation form hit points to leaf nodes, and then we use standard reduction on the quadtree nodes to define form hits points relation to internal nodes.

[1] The algorithm pseudo code was presented in a supplementary document with paper

```
L    //number of tree levels
Hits(Position, Normal, Code, Index, Pixel, Error[L], Length)//hit position,normal,Morton code,index,pixel,error
Nodes(Start, End, Position, Normal, Error, Size, Samples, Length)//node start,end,position,normal,error, number of samples
NodePerLevel, NodePerLevelScan//number of quadtree nodes per level (pre-filled), and its scan
K    //number of seed samples
Temp            //temporary array used for scan, size number of quadtree leaves
Seeds(Position, Normal) //seeds position, normal
Procedure AdaptiveSeeding()
 {
    //step 1: create and sort moron codes for hits
    start = NodePerLevel[L]      // get the start address for leav nodes
    GenerateMortonCode<Hits.Length>();


    Sort(Hits.Code,Hits.Index,Hits.Length)
    //fill reference range for leaves
    FindSortedBounds(&Nodes.Start[start],&Nodes.End[start],Hits.Code,Hits.Length)


    //step 2: calculate average normal and position for leaf nodes
    SegReduce(<&Nodes.Position[start] ,&Nodes.Normal[start]>, <Hits.Position, Hits.Noraml>,Hits.Code,Hits.Index,
      Hits.Length)


    //step 3: calculate average normal and position for inernal nodes using reduction
    for(level=L-1; level>=1; level--)
    {
        ReduceQuadNodes<NodePerLevel[level]>( NodePerLevelScan[level] )
    }

    //step 4: calculate the error per shading point at each level
    CalculateError<Hits.Length>()



    //step 5: calculate the error per nodes at each level
    for(level=L-1; level>=0; level--)
    {
        start = NodesPerLevelScan[level] //start of nodes at this level
        SegReduce(&Nodes.Error[start], Hits.Error[][level],Hits.Code_{bits[0-2level]} ,Hits.Index,Hits.Length)
    }

    //step 6: distrbiute number of seed samples per nodes
    Nodes.Smaples[0] = K
    for(level=0;  level<=L-1; level++) //for each level
    {
        DistributeSeeds<NodePerLevel[level]>( NodePerLevelScan[level] )
    }

    //step 7: select seed samples
    //scan the number of seeds at leaves
    Scan(Temp, &NSed[start], NodePerLevel[L])

    SelectInitialSeeds<NodePerLevel[level]>( NodePerLevelScan[level] )
}
```

Listing 6.6: Adaptive Sample Seeding

Algorithm 6.6 outlines the pseudo code for adaptive seeding algorithm which runs in 7 main steps;

In **step 1** We begin with the *Hits* struct and fill the *Index* array using a sequential order and the *Code* array using using the Morton code. Then we perform parallel sorting on *Index* array using values stored in *Code* array, now *Index* array references a screen space z-order of hit points. We use the utility **FindSortedBounds** to extract the parent node references to hits for all leave nodes using the sorted *Code* array. At this stage the hit point to parent leave node is defined by using the *2L* bits code and the leaf node to hit points relation is defined by node start *(Node.Start)* and node end *(Node.End)* arrays which reference the hit points indirectly through indices array *(Hits.Index)*.

```
Operator GenerateMortonCode()
{
    i = ThreadIndex
    <Hits.Index[i],Hits.Code[i]> = <i,Encode(Hits.Pixel[i])>
}
```

Listing 6.7: Generate Morton Code

In **step 2** we use the segmented reduction to calculate total normal and position vectors for every leaf node by summing both positions, normals of its child hit points.

In **step 3** we move the quadtree level by level in bottom-up manner starting at level preceding the leaf level. In each level we process nodes in parallel. For each internal node we begin by calculating internal nodes start address as the start of its first child and the its end address as the end of its fourth child, then we calculate its position and its normal as the sum of its 4 children position and normal vectors respectively, we also normalize the position of the child node using the node size and normalize its normal vector by dividing this vector by its magnitude.

```
Operator ReduceQuadNodes(in:Start)
{
    i = ThreadIndex
    address = Start + i
    childAdrress = 4*address
    Nodes.Start[address] = Nodes.Start[childAdrress]
    Nodes.End[address] = Nodes.End[childAdrress+3]
    for(i = 0; i<3; i++ )
    {
        Nodes.Position[address] += Nodes.Position[childAdrress]
        Nodes.Normal[address] += Nodes.Normal[childAdrress]
        Nodes.Size[childAdrress] = Nodes.End[childAdrress] - Nodes.Start[childAdrress]
        Nodes.Position[childAdrress] /= Nodes.Size[childAdrress]
        Nodes.Normal[childAdrress] /= Length(Nodes.Normal[childAdrress])
        childAdrress++
    }
}
```

Listing 6.8: Reduce Quadtree Nodes

In **step 4** we calculate L values for each hit point representing the geometric variation of the hit to its paten nodes in each level. This step is very efficient since we first calculate the start node index at every level using the pre-calculated array *(NodePerLevelScan)* and use the Morton code to give the parent node offset in each level. We begin the *2L* Morton code which reference a parent node offset at the leaf level and then move the tree upward level by level by shifting this code 2 bits to right for each up level. The function GetError calculate the geometric variation using equation 6.1

```
Operator CalculateError()
{
    i = ThreadIndex
    code = Hits.Code[i]
    for(level=L-1; level>=1; level--)
    {
        nodeIndex = NodePerLevel[i] + code //get node index form lvel start and morton code
        Calcuate error (e) using Node at nodeIndex and shading point at i
        Hits.Error[i][level-1] = e
        code>>=2 //shift the Morton code 2 bits right to move one level up
    }
}
```

Listing 6.9: Calculate Error Per Shading Point

In **step 5** we use the segmented reduction algorithm to calculate the total geometric variation for all node as the sum of all geometric variation of its child hits. The trick here is that we still use Morton code to give the *Owner* relationship for the segmented reduction utility. Similar to step 4 we get the start index of the parent node at each level using the pre-calculated array *(NodePerLevelScan)* and then at level l we use most significant bits $2l$ to get the parent node offset at that level.

In **step 6** we distribute the number of seeds adaptively to tree nodes level by level. We begin by initialing the number of seed samples at the root node by the total number K. Then we traverse the tree level by level starting from the root level. In each step we process nodes in parallel and distribute parent number of seeds to child nodes in proportional to normalized geometric variation term.

```
Operator DistributeSeeds(in:Start)
{
    i = ThreadIndex
    address = Start + i //get node index form level start and level offset
    childAddress = address*4 // fisrt child node index
    //distribute node's seeds to its children proportion to children error
    error = Nodes.Error[childAddress+0] + Nodes.Error[childAddress+1] + Nodes.Error[childAddress+2] + Nodes.Error[
      childAddress+3]
    for(c=0; c<3; c++)
    {
        Nodes.Samples[childAddress+c] = Nodes.Samples[address]*(Nodes.Error[childAddress+c]/error)
    }
    Make sure that $\sum_{c=0}^{3}$ Nodes.Samples[childAddress+c] = Nodes.Samples[address]
}
```

Listing 6.10: Distrbute seed in tree node in top down manner

In the **final step** we scan the number of seeds for all leaf nodes into array *Temp*, then we launch a parallel kernel for all leave nodes. We check if the node has S samples to distribute, we select S positions and normals randomly form leave node hits and store them in the *Seed* struct starting at an address defined by the scanned array *(Temp)*.

```
Operator SelectInitialSeeds(in:Start)
{
    i = ThreadIndex
    address = Start + i //get node index form level start and level offset
    nSamples = Nodes.Samples[address]
    if( nSamples > 0 )
    {
        outAddress = Temp[i]
        HitsStart = Nodes.Start[address]
        HitsEnd = Nodes.End[address]
        Use jittering to select initial nSamples seeds locations and normals form Hits[HitsStart ,HitsEnd) and
      store them into array Seeds at address outAddress
    }
}
```

Listing 6.11: Select Initial Seeds

6.2.4 Final Gather on GPU

Once we selected initial locations for irradiance samples then we perform the final gather step. For each sample location we sample $250 \sim 500$ rays on the upper hemisphere of the candidate position and trace all rays in parallel. For all final gather rays hits we perform irradiance estimation using the global photons KD-tree, then we average the radiance estimation for each candidate using reduction. And finally we build a KD-tree for all irradiance samples based on sample positions.

6.2.5 Rendering on GPU *-Putting All Together-*

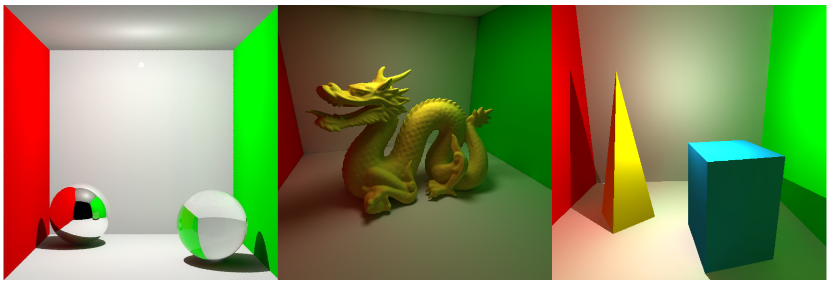
And as a final step we render the final image as the integration of three main rendering compenetent including:

1. Direct lighting and specular reflection using rays tracing as explained in chapter 5.
2. Caustics rendering using irradiance estimation in the caustic photon map.
3. Indirect lighting using irradiance interpolation in the irradiance tree.

6.3 Results and Dicussion

6.3.1 Photon Mapping Performance

Figure 6.13 show the results of parallel photon mapping algorithm on GPU for three test scenes, the first scene include both global photon map and caustics photon map rendering, the dragon and simple box models includes only global photon map. In all scenes we begin by emitting 200



(a) Cornel Box Model (14 primitives)

(b) Dragon Model (200 k triangles)

(c) Simple Box Model (28 triangles)

Figure 6.13: Our test scenes rendered using photon mapping at 2.5, 1.8, 2 fps on a 512 X 512 window using a GTX 285

k. photons and trace them in 5 bounces, we use iterative KNN search algorithm in irradiance estimation as explained in Zhou et al. [2008].

7

Conclusion and Future Work

7.1 Conclustion

This thesis presents several methods toward the goal of interactive and real-time global illumination based on ray tracing techniques. We introduced several theoretical contributions and technical details for parallel ray tracing and photon mapping algorithms on GPU. In Chapter 2 we summarized the state of the art techniques for interactive and real-time global illumination and presented a brief introduction about GPU architecture.

In Chapter 4 we introduced new parallel algorithms for constructing binned SAH BVH on GPU. Our algorithm is relatively fast compared to recent KD-tree and BVH algorithms on both CPU and GPU and the resulting tree is of comparable quality to these algorithms. We also extended the primitive algorithms toolbox to include efficient algorithms required for constructing hierarchal trees. We presented a deep comparison between our new algorithm and most recently algorithms for constructing both KD-trees and BVHs.

In Chapter 5 we employed our hierarchal tree construction algorithms in a Whitted style parallel ray tracing applications, and presented several ways to organize, structure, and store rays tree in a compact form suitable for GPU.

In Chapter 6 we implemented the entire photon mapping algorithm on GPU and evaluated a complete parallel global illumination solution for GPU.

7.2 Future Work

Our work allows further extensions for future work. The frequently used utilities on GPU presented in Chapter 4 may be a good start to implement a parallel template library for such primitives on GPU, other techniques such as hashing may also be include for such library.

We believe that the ideas presented and discussed for fast BVH construction algorithms on GPU such as Morton coding and primitive sorting can be extended to accelerate both the SAH KD-tree and point based KD-Tree construction algorithms.

Our plans for the near future is to extend the Whitted style ray tracing algorithm presented in Chapter 5 to include distributed ray tracing [Cook et al., 1984] effects. Very recently Garanzha and Loop [Garanzha and Loop, 2010] introduced soft shadow effect using rays sorting and frustum creation and traversal on GPU, such techniques may be promising for rendering other distributed rays tracing effects.

The global illumination solution presented in Chapter 6 introduces a complete global illumination solution; but several areas for future research still unexplored. For example we need to support area light sources and produce visual effects such as soft shadows, we believe that efficient ways for frustum creation and traversal on GPU [Garanzha and Loop, 2010] may be efficient for such case. In the radiance estimation step, we found that results produced by histogram based range search algorithm on GPU are of comparable quality to what produced by range search algorithm on CPU, we believe that other solutions such as heterogeneous work distribution on GPU Tzeng et al. [2010] will be efficient to implement the range search algorithm on GPU. The radiance caching implementation was based on the approximations that ignore the harmonic distance calculation and the adaptive nature of the original radiance aching algorithm [Ward et al., 1988], We believe that an adaptive algorithm for Octree construction may be efficient for implementing a better solution for radiance caching.

Appendix A

Frequently Used Utilities on GPU

These definitions represent wrappers for most frequently used parallel primitive algorithms on GPU.

1. Reduction

   ```
   void Reduce (O, I, N, Op=+)
   ```

   Performs a reduction to an array *I* of length *N* into *O* using operator *Op*.
   Parameters:
   1st : output, the reduction result.
   2^{ed} : input, an array to be reduced.
   3rd : input, array length.
   4th : input, reduction operation, valid values are $+$ for sum, $<$ for minimum, and $>$ for maximum, default value is $+$.
   Implementation:
   We implement this utility by calling the *cudppReduce* [Harris et al., 2007] function with the appropriate parameters.

2. Segmented Reduction

   ```
   void SegReduce (O, I, N, Owner, Op=+ )
   ```

   Performs a segmented reduction to an array *I* using an operator *Op* and an array for (Owner) values which indicates different array segments and returns the result into array *O*.
   Parameters:
   1st : output, the segmented reduction result.
   2^{ed} : input, an array to be reduced.
   3rd : input, array length.
   4th : input, the owner values for each segment and has the same array length N.
   5th : input, reduction operation, similar to reduce utility.

Implementation:
We implement this utility using the segmented reduction algorithm presented in [Zhou et al., 2008].

3. Scan

```
void Scan (O, I, N, Params )
```

Performs a scan to an arrays *I* of length *N* into *O* using options stored in the *Params* argument.
Parameters:
1st : output, the scan result.
2ed : input, an array to be scanned.
3rd : input, array length.
4rd : input, scan parameters, specifies the operator, the identity value, the type and the direction for the scan, and default values are {+*, 0, Exclusive, Forward*} which represent an exclusive forward sum scan.
Implementation:
We implement this utility by calling the *cudppScan* [Harris et al., 2007] function with the appropriate parameters.

4. Segmented Scan

```
void SegScan (O, I, Flags, N, Params )
```

Performs a segmented scan to an arrays *I*of length *N* using an array for *Head* flags to indicate different array segments and returns result into *O*.
Parameters:
1st : output, the segmented scan result.
2ed : input, an array to be scanned.
3rd : input, the head flags for each segment which has the same length N, each element of this array contains 0 except at the start locations of each segments in the input arrays it contains 1.
4th : input, array length.
5th : input, specifies the scan parameters similar to the scan utility.
Implementation:
We implement this utility by calling the *cudppSegScan* [Harris et al., 2007] function with the appropriate parameters.

5. Scan Tail (Last Value of a Scan)

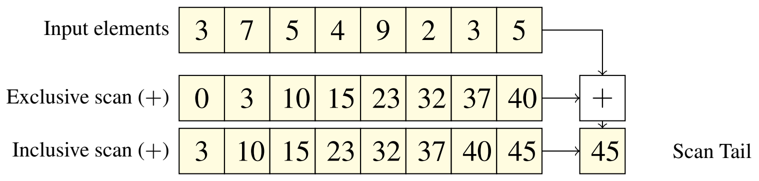


Figure A.1: Scan tail extraction for both inclusive and exclusive scan

Version 1: Exclusive Scan

```
void ScanTail (O, I, IScan, N, Params)
```

Get the last value(s) of the scan operation. This version corresponding to an exclusive scan so it requires the original array *I* and scanned array *IScan* to get the scan result. The value returned in *O* equals *I[N-1] Op IScan[N-1]* for forwrad scan [1].

Parameters:

1st : output, Last value of the scan.
2^{ed} : input, an array of elements of length N.
3rd : input, an array length N, representing the exclusive scan of I array.
4th : input, array length.
5th : input, specifies the scan parameters similar to the scan utility.

Implementation:

We launch a single thread of kernel that apply the operator to the last elements of the arrays *I* and *IScan* and copy the result to destination variables *O*.

Version 2: Inclusive Scan

```
void ScanTail (O, I, N, Params)
```

In this version we do not need both the original non-scanned array and the scan operator since the scan result already exist at the last element of the scanned array *IScan*. The value O equals IScan[N-1] for forward scan.

Parameters:

1st : output, Last value of the scan.
2^{ed} : input, an array length N, stores an inclusive scan result.
3rd : input, array length.
4th : input, specifies the scan parameters similar to the scan utility.

Implementation:

We launch a single thread of kernel that copy the last value in array *IScan* to destination variables *O*.

6. Segmented Scan Tails (Last Value of each Segment in a Scan)

Version 1: Exclusive Segmented Scan

```
void SegScanTails( O, I, IScan, SegmentsTail, NumSegments, Params)
```

Similar to ScanTail but get the tails of each segment in the scanned input.

Parameters:

1st : output, an array of length *NumSegments* which is filled by the last value of the scan of each segment.
2^{ed} : input, a non-scanned array of input elements.
3rd : input, an exclusive scan of the array *I*.
4th : input, an array of length *NumSegments* which contains the last index of each segment of the input arrays.
5th : input, the number of segments.
6th : input, specifies the scan parameters similar to the scan utility.

Implementation:

We launch *NumSegments* kernels to apply the operator to the last elements of input arrays *(I and IScan)* at

[1]For backward scan the value returned is corresponding to the first array elements, as a result the array length (N) has no use in this case.

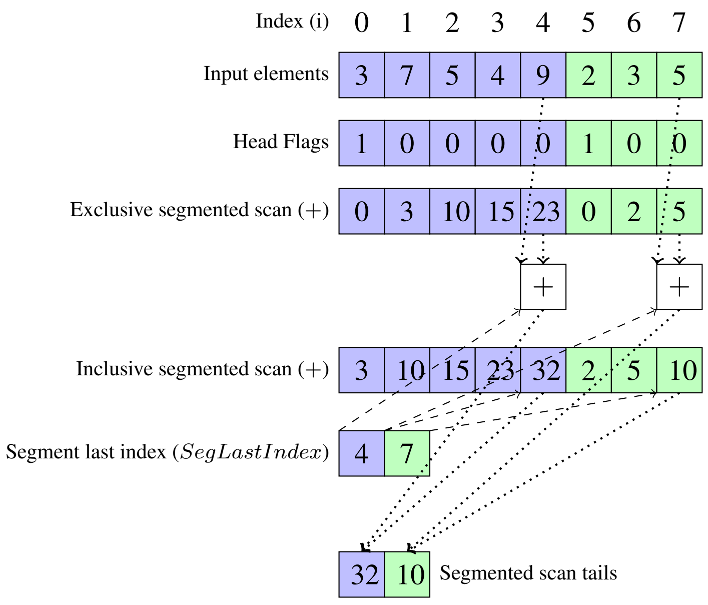


Figure A.2: Scan tails extraction for a segmented scan

each segment tail and copy the result to destination array *(O)* at corresponding segment index. The tail value of segment j will be calculated as *O[j] = I[SegLastIndex[j]]* $\oplus$ *IScan[SegLastIndex[j]]* for forward scan.

Version 2: Inclusive Segmented Scan

```
void SegScanTails (O, I, SegmentsTail, NumSegments, Params)
```

Like the non segmented scan we do not need both the original input arrays and the scan operator.

Parameters:

1^{st} : output, an array of length *NumSegments* which is filled by the last value of the scan of each sgement.

2^{ed} : input, an arrays represent an inclusive scan result.

3^{rd} : input, an array of length N which contains the last index of each segment of the input arrays.

4^{th} : input, the number of segments.

6^{th} : input, specifies the scan parameters similar to the scan utility.

Implementation:

We launch *NumSegments* kernels to copy to the last elements of each segment in the scanned array *IScan* to destination array *(O)* at corresponding segment index. The last of value of segment j of will be calculated as *O[j] = IScan[SegLastIndex[j]]*.

7. Append

```
void Append (O, I, N )
```

Append in input array *I* to the array *O* where each array is of length *N*.

Parameters:

1^{st} : output, destination array.
2^{ed} : input, source array.
3^{rd} : input, array length.
Implementation:
We launch N kernels where each kernel i copy the element at index i in the each input array *I* to the corresponding output array *O* at index i.

8. Compact

```
void Compact (O, I, Flags, FlagsScan)
```

Compact and append valid locations in input array *I* which are marked by 1 values in the *Flags* array.
Parameters:
1^{st} : output, destination array.
2^{ed} : input, source array.
3^{rd} : input, flags array to mark valid location in the input array to be appended.
4^{th} : input, exclusive sum scan of the flags array.
5^{th} : input, array length.

Implementation:
We launch N kernels where each kernel i check the corresponding flag if it equals 1 then we copy the element in input array *(I)* at index i to output array *(O)* at an address equals $FlagsScan[i]$ (see figure 3.4).

9. Split

Version 1: Split into a Single List
```
void Split (O, I, Flags, FlagsScan, N )
```
For an input array *I* we check the flags array and separate all elements marked as 1 to the left of all elements marked as 0.
Parameters:
1^{st} : output, destination array.
2^{ed} : input, source array.
3^{rd} : input, a flags array to mark true/false elements for the desired left or right side in the output.
4^{th} : input, exclusive sum scan of the flags array.
5^{th} : input, array length.

Version 2: Split into Two Lists
```
void Split (OTrue, OFalse, I, Flags, FlagsScan)
```
For an input array *I* we divide the elements into two array *OTrue* and *OFalse* according to the input flags in the array *Flags*.

1^{st} : output, destination array for elements marked as 1 in the in the flags array.
2^{ed} : output, destination array for elements marked as 1 in the in flags array.
3^{rd} : input, source array.
4^{th} : input, a flags array to mark true/false elements for the desired left or right side in the output.

5th : input, exclusive sum scan of the flags array.
6th : input, array length.

10. Sort

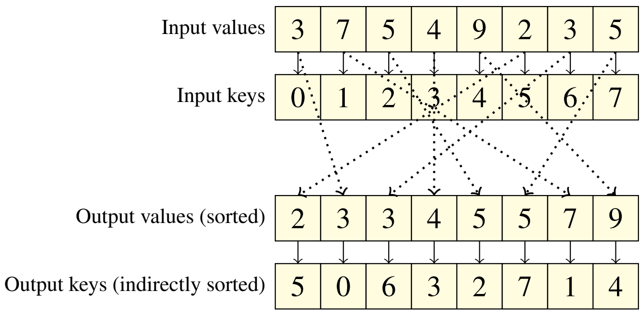


Figure A.3: Key value sorting example

```
void Sort (Values, Keys, N )
```

Sort an array of values and optionally relocate the corrsponding keys uisng the sorted sequence.
Parameters:
1st : input/output, array of values to be sorted.
2ed : input/output, array of keys to be orderd using the sorted sequence of values.
Implementation:
We call the radix sort routine *(cudppSort)* of the *CUDPP* [Harris et al., 2007] libarary.

11. Find Sorted Values Bounds

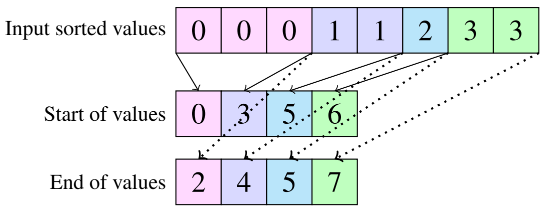


Figure A.4: Extracting sorted values bounds

```
void FindSortedBounds ( Start, End, I, N)
```

For a sorted array of *N* integers in the range from 0 to j it returns two arrays of $j + 1$ items where items in the first array indicate the start of each value and items in the second array indicate the end of each value in sorted array.
Parameters:
1st : output, an array that stores the start index of each value inclusivly.
1ed : output, an array that stores the end index of each value exclusively.
3rd : input, sorted array.

4^{th} : input, array length.
Implementation:
We use an efficient implementation of this utility which exits in NIVIDA CUDA samples [1], and this kernel works by launching *N* threads where each thread check neighboring values (V_1, V_2) and if it find they differ then it set the end bound of V_1 and the start bound of V_2. But we have to initialize the bound arrays to 0 values if it is expected that some values in the sorted range may not appear. We present this kernel in appendix.

12. Random Number Generator

```
uint[float] RandD ( )
```

Returns an unsigned integer or a float random number.
Implementation:
We pre-generate a large number of random numbers on GPU using **(cudppRand)** function which implement Mersenne Twister Matsumoto and Nishimura [1998] pseudorandom generator and later we access these numbers using thread index. Other option is to implement a simple quasi random number generator like Haltom sequence [2].

[1]This implementation exist in the particle sample of NVIDIA CUDA sample [NVIDIA, 2010]

[2]The NVIDIA CUDA samples [NVIDIA, 2010] also contains a sample implementing Mersenne Twister pseudorandom generator

Appendix B

Parallel Chunking Primitive and Utility

The input to the chunking primitive is the chunk size *T* and nodes SoA contains an array for nodes start index and an array for nodes size, and the output to such primitives is the chunks SoA that partition the nodes into chunks of primitives of at most *T* size. consisting of three arrays; the first is the *ChunksOwner* array which contains node indices to the nodes array; the second is the *ChunksStart* array which stores the start index of each chunk; and third is the *ChunksSize* array which stores the size of each chunk. We will explain two ways for preparing chunks data structures; the first method uses two segmented scan passes over a number of elements equals the total number of chunks to prepare the data structure, and the second uses a single standard scan pass and a parallel processing pass over a number of elements equals the total number of chunks to perform the same operator.

Method 1: Craeting Chunks Using Segemnted Scan. To divide triangles of each node into Fixed-Sized chunks of at most *T* triangles we use the *Decompression* scheme presented in Garanzha and Loop [2010]. First, we create an array *ChunksPerNode* equal in size to the *ActiveNodes* array, where *ChunksPerNode*$_i$ = $\lceil NodeSize_i/T \rceil$ and then scan this array and get total number of chunks *(TotChunks)* using the *ScanTail* utility. We create three arrays *HeadFlags* and *OwnerSkeleton* and *StartSkeleton* (each array of size equals to *TotChunks*). To fill these arrays we consider *Scan(ChunksPerNode)* as *HeadIndices* array and the number of nodes as the number of segments *(NumSegments)*. We fill the *HeadFlags* array by calling the *FillHeadFlags* utility, and fill the *OwnerSkeleton* array with node indices at head locations and *zeros* otherwise by calling the the *FillSkeleton* utility, and fill the *StartSkeleton* array with *NodeStart* at head locations and *T* values otherwise by calling the *FillSkeleton* utility employing *NodeStart* as the

HeadValues array , and *T* as the *ConstantValue*. Then we apply an inclusive segmented scan to *OwnerSkeleton* and *StartSkeleton* arrays using the *HeadFlags* array. After these scans we consider the *scan(OwnerSkeleton)* as the *ChunksOwner* array and the *scan(StartSkeleton)* as the *ChunksStart* array and fill the *ChunksSize* array with the differences between neighboring *ChunksStart* elements (see Figure B.1).

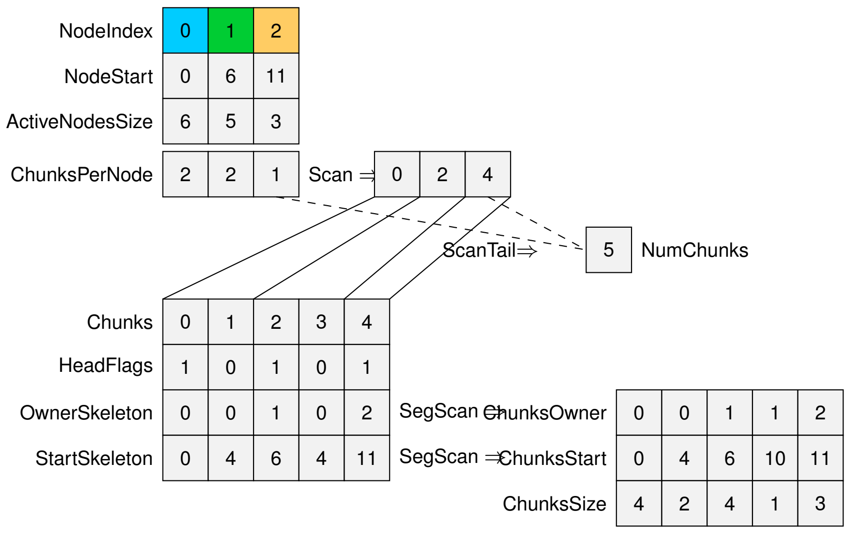


Figure B.1: Chunks creation using segmented scan, for illustration purpose we set the chunk size *T* equals 4.

Method 2: Craeting Chunks Using Standard Scan.

So far, we have created chunks data structure using two segmented scans over the *OwnerSkeleton* and *StartSkeleton* arrays. However, it was noticed that segmented scans are about three times slower than the unsegmented scans Sengupta et al. [2007]; so, we can enhance this process by replacing the two segmented scans with a single unsegmented scan. Like the original process we start by filling the *ChunksPerNode* array, scan it and get the total number of chunks *(NumChunks)*, then we fill the *HeadFlags* array in way similar to one used in the *FillHeadFlags* utility except that we set the first element to 0, then, we perform an inclusive scan this array and consider the scan result as the *ChunksOwner* array. We launch *NumChunks* threads of an operator that fills the *ChunkStart* array. In this operator using the thread index (i) we get the owner node *(O)* form the corresponds value in the *ChunksOwner* array, and get the segment start of the current chunk *(H)* as *Scan(ChunksPerNode)[O]*, and then we calculate the corresponding value in the start array as: *NodeStart[O] + (H-i)* $\times$ *T*. After calling this operator we fill the array *Size* using the difference between every two neighboring elements as in the original segmented chunk creation process(see Figure B.2).

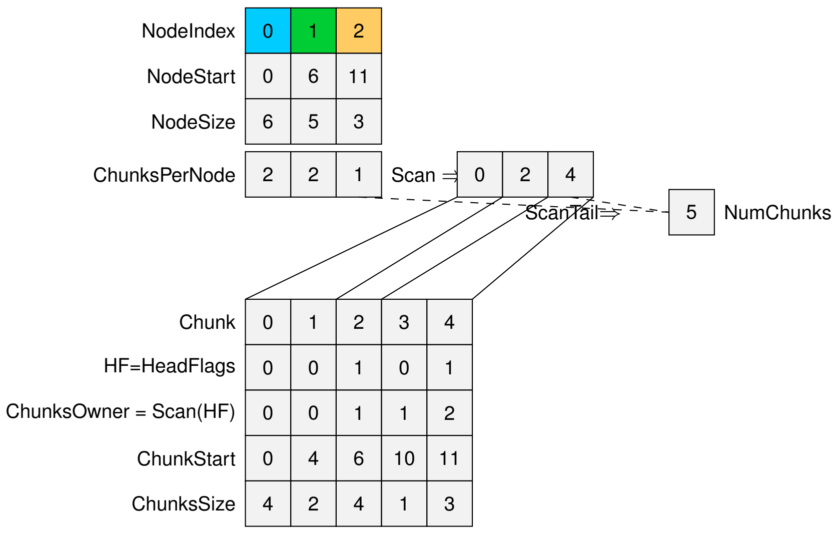


Figure B.2: Chunks creation using standard scan, chunk size *T* equals 4.

Parallel Chunking Utility.

We can define chunking primitive as follow:

```
void CreateChunks( T, NodesStart, NodesSize, N, ChunksOwner, ChunksStart,
ChunksStart, NumChunks )
```

Create chunks of primitives of approximately size *T* for the *N* nodes.

Parameters:

1st : input, the maximum size of a chunk.
2^{ed} : input, an array of length N that store the start index of each input node.
3rd : input, an array of length N that store the size of each input node.
4th : input, the number of input nodes.
5th : output, an array of length NumChunks that store the parent node index of each chunk.
6th : output, an array of length NumChunks that store the start index of each chunk.
7th : output, an array of length NumChunks that store the size of each chunk.
8th : output, the total number of output chunks.

Implementation:

We implemented this utility using method 1 which employs the segmented scan.

Bibliography

AILA, T. AND LAINE, S. 2009. Understanding the efficiency of ray traversal on gpus. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*. ACM, New York, NY, USA, 145–149. 2, 12, 69, 70

AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA. 7

ARVO, J. AND KIRK, D. 1990. Particle transport and image synthesis. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*. ACM, New York, NY, USA, 63–66. 93, 98

BIALLY, T. 1969. Space-filling curves: Their generation and their application to bandwidth reduction. In *IEEE Transactions on Information Theory*. Vol. 15. 658–664. 69

BOULOS, S., EDWARDS, D., LACEWELL, J. D., KNISS, J., KAUTZ, J., SHIRLEY, P., AND WALD, I. 2007. Packet-based whitted and distribution ray tracing. In *GI '07: Proceedings of Graphics Interface 2007*. ACM, New York, NY, USA, 177–184. v, 8

CHOI, M. G., JU, E., CHANG, J.-W., LEE, J., AND KIM, Y. J. 2009. Linkless octree using multi-level perfect hashing. *Computer Graphics Forum 28,* 7, 1773–1780. 112

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The reyes image rendering architecture. *SIGGRAPH Comput. Graph. 21*, 95–102. 13

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. *SIGGRAPH Comput. Graph. 18,* 3, 137–145. 7, 12, 120

DANILEWSKI, P., POPOV, S., AND SLUSALLEK, P. Binned sah kd-tree construction on a gpu. *Technical report*. 27

ENGEL, W. 2009. *ShaderX7: Advanced Rendering Techniques*. Charles River Media. 11

FOLEY, T. AND SUGERMAN, J. 2005. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. ACM, New York, NY, USA, 15–22. 70

GARANZHA, K. 2009. The use of precomputed triangle clusters for accelerated ray tracing in dynamic scenes. *Computer Graphics Forum 28,* 4, 1199–1206. 2

GARANZHA, K. AND LOOP, C. 2010. Fast ray sorting and breadth-first packet traversal for gpu ray tracing. *Computer Graphics Forum 29,* 2, 289–298. 12, 19, 28, 120, 129

GARGANTINI, I. 1982. An effective way to represent quadtrees. *Commun. ACM 25,* 12, 905–910. 112

GAUTRON, P., BOUATOUCH, K., AND PATTANAIK, S. 2006. Temporal radiance caching. In *ACM SIGGRAPH 2006 Sketches*. SIGGRAPH '06. ACM, New York, NY, USA. 95

GLASSNER, A. S. 1989. *An Introduction to Ray tracing*. Morgan Kaufmann. 7

GOLDSMITH, J. AND SALMON, J. 1987. Automatic creation of object hierarchies for ray tracing. *IEEE Comput. Graph. Appl. 7*, 14–20. 25

GORAL, C. M., TORRANCE, K. E., GREENBERG, D. P., AND BATTAILE, B. 1984. Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph. 18*, 213–222. 1, 9, 12

GREEN, R. Spherical harmonic lighting: The gritty details. 13

GUNTHER, J., POPOV, S., SEIDEL, H.-P., AND SLUSALLEK, P. 2007. Realtime ray tracing on gpu with bvh-based packet traversal. *Symposium on Interactive Ray Tracing 0*, 113–118. 69

HACHISUKA, T., JAROSZ, W., WEISTROFFER, R. P., DALE, K., HUMPHREYS, G., ZWICKER, M., AND JENSEN, H. W. 2008. Multidimensional adaptive sampling and reconstruction for ray tracing. *ACM Trans. Graph. 27*, 33:1–33:10. 12

HARRIS, M., OWENS, J. D., SENGUPTA, S., ZHANG, Y., DAVIDSON, A., AND TSENG, S. 2007. Cudpp library. 15, 121, 122, 126

HARRIS, M., SENGUPTA, S., AND OWENS, J. D. 2007. Parallel prefix sum (scan) with cuda. In *GPU Gems 3*, H. Nguyen, Ed. Addison Wesley. 19, 21, 28

HAVRAN, V. 2000. Heuristic ray shooting algorithms. Ph.D. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague. 6, 12, 25, 26, 33

HEARN, D. AND BAKER, M. P. 1994. *Computer Graphics*. Prentice Hall. 11

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN, P. 2007. Interactive k-d tree gpu raytracing. ACM, New York, NY, USA, 167–174. 70

HOU, Q., QIN, H., LI, W., GUO, B., AND ZHOU, K. 2010. Micropolygon ray tracing with defocus and motion blur. In *ACM SIGGRAPH 2010 papers*. SIGGRAPH '10. ACM, New York, NY, USA, 64:1–64:10. 13

HOU, Q., SUN, X., ZHOU, K., LAUTERBACH, C., AND MANOCHA, D. 2010. Memory-scalable gpu spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics 99,* PrePrints. 27, 67

HUNT, W., MARK, W., AND STOLL, G. 2006. Fast kd-tree construction with an adaptive error-bounded heuristic. *Symposium on Interactive Ray Tracing 0*, 81–88. 26

IZE, T., SHIRLEY, P., AND PARKER, S. G. 2007. Grid creation strategies for efficient ray tracing. *IEEE/EG Symposium on Interactive Ray Tracing 17*, 2732. 7

JAROSZ, W., DONNER, C., ZWICKER, M., AND JENSEN, H. W. 2008. Radiance caching for participating media. *ACM Trans. Graph. 27,* 1, 1–11. 95

JAROSZ, W., ZWICKER, M., AND JENSEN, H. W. 2008. Irradiance Gradients in the Presence of Participating Media and Occlusions. *Computer Graphics Forum (Proceedings of EGSR 2008) 27,* 4. 95

JENSEN, H. W. 1996. Global illumination using photon maps. *Rendering Techniques*, 21–30. 91

JENSEN, H. W. 2001. *Realistic Image Synthesis Using Photon Mapping*. AK Peters. v, 1, 9, 10, 12, 91, 93, 98

JENSEN, H. W. 2004. A practical guide to global illumination using ray tracing and photon mapping. In *ACM SIGGRAPH 2004 Course Notes*. SIGGRAPH '04. ACM, New York, NY, USA. 11, 75, 91

JENSEN, H. W., ARVO, J., DUTRE, P., KELLER, A., PHARR, M., , AND SHIRLEY, P. 2003. Monte carlo ray tracing. In *ACM SIGGRAPH 2003 courses*. SIGGRAPH '03. ACM, New York, NY, USA. 11, 12, 75

KAJIYA, J. T. 1986. The rendering equation. *SIGGRAPH Comput. Graph. 20*, 143–150. 1, 9

KALOJANOV, J. AND SLUSALLEK, P. 2009. A parallel algorithm for construction of uniform grids. In *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. ACM, New York, NY, USA, 23–28. 2

KLIMASZEWSKI, K. S. AND SEDERBERG, T. W. 1997. Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications 17*, 42–51. 6

KŘIVÁNEK, J., BOUATOUCH, K., PATTANAIK, S., AND ŽÁRA, J. 2008. Making radiance and irradiance caching practical: adaptive caching and neighbor clamping. In *ACM SIGGRAPH 2008 classes*. SIGGRAPH '08. ACM, New York, NY, USA, 77:1–77:12. 95

KŘIVÁNEK, J., GAUTRON, P., WARD, G., ARIKAN, O., AND JENSEN, H. W. 2007. Practical global illumination with irradiance caching. In *ACM SIGGRAPH 2007 courses*. SIGGRAPH '07. ACM, New York, NY, USA. 95

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast bvh construction on gpus. *Computer Graphics Forum 28,* 2, 375–384. 2, 3, 12, 26, 27, 32, 43, 60, 63, 71

MA, V. C. H. AND MCCOOL, M. D. 2002. Low latency photon mapping using block hashing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. HWWS '02. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 89–99. 93

MATSUMOTO, M. AND NISHIMURA, T. 1998. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul. 8*, 3–30. 127

MOORE, G. E. 2000. Readings in computer architecture. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Chapter Cramming more components onto integrated circuits, 56–59. 15

MORLEY, R. K. AND SHIRLEY, P. 2003. *Realistic Ray Tracing*. AK Peters. 7, 75

NG, R., RAMAMOORTHI, R., AND HANRAHAN, P. 2003. All-frequency shadows using non-linear wavelet lighting approximation. *ACM Trans. Graph. 22*, 376–381. 13

NVIDIA 2010. *CUDA programming guide 3.1, http://developer.nvidia.com/object/cuda.html*. NVIDIA. Last accessed Oct., 2010. 16, 95, 127

PANTALEONI, J. AND LUEBKE, D. 2010. Hlbvh: Hierarchical lbvh construction for real-time ray tracing of dynamic geometry. In *In Proceedings of High Performance Graphics'10*. 87–95. 27, 68, 70

PARKER, S. G., BIGLER, J., DIETRICH, A., FRIEDRICH, H., HOBEROCK, J., LUEBKE, D., MCALLISTER, D., MCGUIRE, M., MORLEY, K., ROBISON, A., AND STICH, M. 2010. Optix: a general purpose ray tracing engine. *ACM Trans. Graph. 29*, 66:1–66:13. v, 8

PATNEY, A. AND OWENS, J. D. 2008. Real-time reyes-style adaptive surface subdivision. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia) 27,* 5 (Dec.), 143:1–143:8. 13

PHARR, M. AND HUMPHREYS, G. 2010. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann. 7, 26, 75

PHARR, M., LEFOHN, A., KOLB, C., LALONDE, P., FOLEY, T., AND BERRY, G. Programmable graphics - the future of interactive rendering. *Neoptica Technical Report*. 11

POPOV, S., GÜNTHER, J., SEIDEL, H.-P., AND SLUSALLEK, P. 2006. Experiences with streaming construction of SAH KD-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*. 89–94. 26

PURCELL, T. J., BUCK, I., MARK, W. R., AND HANRAHAN, P. 2005. Ray tracing on programmable graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*. ACM, New York, NY, USA, 268. 93

QUINN, M. J. 1993. *Parallel Computing: Theory and Practice*. CMcGraw-Hill. 19

RAMAMOORTHI, R. 2009. Precomputation-based rendering. *Found. Trends. Comput. Graph. Vis. 3*, 281–369. 13

RAMAMOORTHI, R. AND HANRAHAN, P. 2001. An efficient representation for irradiance environment maps. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. SIGGRAPH '01. ACM, New York, NY, USA, 497–500. 13

RAMAMOORTHI, R. AND HANRAHAN, P. 2002. Frequency space environment map rendering. *ACM Trans. Graph. 21*, 517–526. 13

REINHARD, E., SMITS, B. E., AND HANSEN, C. 2000. Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*. Springer-Verlag, London, UK, 299–306. 7

SATISH, N., HARRIS, M., AND GARLAND, M. 2009a. Designing efficient sorting algorithms for manycore gpus. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. IEEE Computer Society, Washington, DC, USA, 1–10. 19, 21, 28

SATISH, N., HARRIS, M., AND GARLAND, M. 2009b. Designing efficient sorting algorithms for manycore gpus. In *23rd IEEE Intl Parallel & Distributed Processing Symposium*. 59

SENGUPTA, S., HARRIS, M., ZHANG, Y., AND OWENS, J. D. 2007. Scan primitives for gpu computing. In *Graphics Hardware 2007*. ACM, 97–106. 19, 21, 22, 28, 57, 130

SHUBHABRATA SENGUPTA, M. H. AND GARLAND, M. 2008. Efficient parallel scan algorithms for gpus. *NVIDIA Technical Report NVR-2008-003*. 19, 28

SLOAN, P.-P., KAUTZ, J., AND SNYDER, J. 2002. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Trans. Graph. 21*, 527–536. 13

SUBR, K. 2008. Sampling strategies for efficient monte carlo image synthesis. Ph.D. thesis, University of California, Irvine. 11

SUN, X., HOU, Q., REN, Z., ZHOU, K., AND GUO, B. 2011. Radiance transfer biclustering for real-time all-frequency biscale rendering. *IEEE Transactions on Visualization and Computer Graphics 17*, 64–73. 13

TZENG, S., PATNEY, A., AND OWENS, J. D. 2010. Task management for irregular-parallel workloads on the gpu. In *High Performance Graphics*, M. Doggett, S. Laine, and W. Hunt, Eds. Eurographics Association, 29–37. 2, 120

VEACH, E. 1997. Robust monte carlo methods for light transport simulation. Ph.D. thesis, Stanford University. 1, 9, 11, 12

WALD, I. 2004. Realtime Ray Tracing and Interactive Global Illumination. Ph.D. thesis, Computer Graphics Group, Saarland University. 2, 12, 25, 26, 31, 70, 100

WALD, I. 2007. On fast construction of sah-based bounding volume hierarchies. In *In Proceedings of the 2007 IEEE/EG Symposium on Interactive Ray Tracing. IEEE*. 33–40. 7, 12, 26, 27, 33, 50

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26,* 1, 6. 26

WALD, I., GNTHERY, J., AND SLUSALLEKY, P. 2004. Balancing considered harmful faster photon mapping using the voxel volume heuristic . *Computer Graphics Forum 23,* 3, 595–603. 12, 26

WALD, I., GÜNTHER, J., AND SLUSALLEK, P. 2004. Balancing considered harmful – faster photon mapping using the voxel volume heuristic. *Computer Graphics Forum 22,* 3, 595–603. (Proceedings of Eurographics). 99

WALD, I. AND HAVRAN, V. 2006. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*. 61–69. 7, 12, 26, 71

WALD, I., IZE, T., KENSLER, A., KNOLL, A., AND PARKER, S. G. 2006. Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph. 25*, 485–493. 6, 12, 25

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*. 153–164. 12, 69

WANG, R., WANG, R., ZHOU, K., PAN, M., AND BAO, H. 2009. An efficient gpu-based approach for interactive global illumination. In *SIGGRAPH '09: ACM SIGGRAPH 2009 papers*. ACM, New York, NY, USA, 1–8. 2, 4, 112

WARD, G. J., RUBINSTEIN, F. M., AND CLEAR, R. D. 1988. A ray tracing solution for diffuse interreflection. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*. ACM, New York, NY, USA, 85–92. 94, 120

WHITTED, T. 1980. An improved illumination model for shaded display. *Commun. ACM 23,* 6, 343–349. v, 1, 2, 5, 6, 8, 12

WOOP, S. A ray tracing hardware architecture for dynamic scenes. *Technical report, Saarland University*. 70

ZHOU, K., HOU, Q., REN, Z., GONG, M., SUN, X., AND GUO, B. 2009. Renderants: interactive reyes rendering on gpus. In *ACM SIGGRAPH Asia 2009 papers*. SIGGRAPH Asia '09. ACM, New York, NY, USA, 155:1–155:11. 13

ZHOU, K., HOU, Q., WANG, R., AND GUO, B. 2008. Real-time kd-tree construction on graphics hardware. *ACM Trans. Graph. 27,* 5, 1–11. v, 2, 3, 4, 10, 12, 19, 20, 26, 27, 28, 32, 40, 71, 76, 99, 109, 117, 122