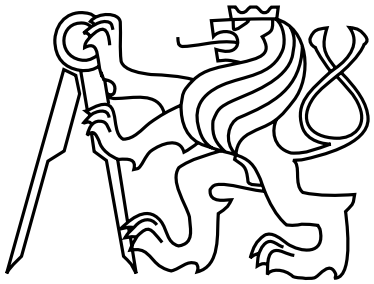
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Graphics and Interaction



Master's Thesis

Methods for Fast Construction of Bounding Volume Hierarchies

Bc. Daniel Meister

Supervisor: Ing. Jiří Bittner, Ph.D.

Study Programme: Open Informatics

Field of Study: Computer Graphics and Interaction

May 12, 2014

Aknowledgements

I would like to thank Ing. Jiří Bittner, Ph.D. for his kind attitude and for supervising this thesis. Also I would like to express my gratitude to RNDr. Marek Vinkler for many valuable advices.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 12, 2014 .....

Abstract

The bounding volume hierarchy is one of the most common acceleration data structures used in computer graphics. This thesis is about a fast parallel construction of bounding volume hierarchies on the GPU. Two construction algorithms were implemented the CUDA technology. The first algorithm is based on Morton codes and spatial median splits. The second algorithm uses the surface area heuristic to minimize the cost of the hierarchy. Both algorithms were tested in the real-time rendering system. The core of the rendering system is based on an efficient ray tracer.

Abstrakt

Hierarchie obálek je jedna z nepoužívanějších akceleračních datových struktur v počítačové grafice. Tato práce se zabývá rychlou paralelní konstrukcí hierarchie obálek na grafických procesorech. Implementoval jsem dva konstrukční algoritmy v technologii CUDA. První algoritmus je založen na dělení prostorovým mediánem a Mortonových kódech. Druhý algoritmus využívá SAH heuristiku k minimalizaci ceny výsledné hierarchie. Oba algoritmy jsem testoval v zobrazovacím systému, který je založený na algoritmu zpětného sledování paprsku.

Contents

1 Introduction **1**
1.1 Thesis Structure . . . 1

2 Theoretical Background **3**
2.1 Ray Shooting and Visibility . . . 3
2.2 Image Synthesis . . . 4
2.2.1 Ray Casting . . . 4
2.2.2 Ray Tracing . . . 5
2.2.3 Path Tracing and Rendering Equation . . . 5
2.3 Acceleration Data Structures . . . 6
2.3.1 Object Partitioning Structures . . . 7
2.3.2 Space Partitioning Structures . . . 8
2.4 Parallel Computing . . . 9
2.4.1 Reduction . . . 10
2.4.2 Prefix Scan . . . 10
2.4.3 Radix Sort . . . 12

3 Compute Unified Device Architecture **15**
3.1 Programming Model . . . 15
3.2 Memory Model . . . 16
3.3 Execution Model . . . 18

4 Construction of Bounding Volume Hierarchies **19**
4.1 Cost Model . . . 19
4.2 Top-Down Construction . . . 20
4.2.1 Surface Area Heuristic . . . 20
4.2.2 Spatial and Object Median . . . 21
4.3 Bottom-Up Construction . . . 22
4.4 Incremental Construction . . . 22

5 Bounding Volume Hierarchies on the GPU **23**
5.1 Morton Curve and Spatial Median Splits . . . 23
5.2 Linear Bounding Volume Hierarchies . . . 24
5.2.1 Spatial Median Splits . . . 25
5.2.2 SAH Splits . . . 25
5.3 Hierarchical Linear BVH . . . 26

5.4 Hierarchical Linear BVH with Work Queues . . . 27
5.4.1 Spatial Median Splits . . . 28
5.4.2 SAH Splits . . . 28
5.5 Compact Prefix BVH . . . 29
5.6 BVH with Task Pool . . . 30
5.6.1 Spatial Median Splits . . . 30
5.6.2 SAH Splits . . . 32

6 Design and Implementation **33**
6.1 Construction of Bounding Volume Hierarchies . . . 33
6.1.1 Spatial Median Splits . . . 33
6.1.2 SAH Splits . . . 35
6.1.3 Radix Sort . . . 37
6.2 Rendering System . . . 38
6.2.1 Used Technologies . . . 38
6.2.2 CUDA Management . . . 39
6.2.3 Scene Management . . . 40
6.2.4 BVH Management . . . 41
6.2.5 Rendering Pipeline . . . 41
6.2.6 User Interface . . . 42

7 Results and Discussion **45**
7.1 Static Scenes . . . 45
7.1.1 Spatial Median Split with 30-bit Morton Codes . . . 46
7.1.2 Spatial Median Split with 60-bit Morton Codes . . . 46
7.1.3 SAH Splits with 30-bit Morton Codes and 15-bit Clusters . . . 46
7.2 Dynamic Scenes . . . 47
7.3 Discussion . . . 47

8 Conclusion **59**
8.1 Future Work . . . 59

Bibliography **61**

A List of Abbreviations **65**

B Installation Guide **67**

C User Manual **69**
C.1 Configuration . . . 69
C.2 Controls . . . 70

D DVD Content **71**

Chapter 1

Introduction

Rendering algorithms based on ray shooting are an important part of the image synthesis. The basic idea is to shoot a ray through each pixel of the virtual camera to the virtual scene. The closest intersections between rays and the scene are computed. Pixels are shaded according to material properties of intersected primitives, light sources and a given lighting model. The major obstacle of these algorithms is the time complexity. Thousands of rays are tested against thousands of scene primitives. The time complexity can be reduced using various acceleration data structures. In real-time interactive applications the scene geometry changes in each frame and the acceleration structure is no longer valid. The most natural update of the data structure is a reconstruction from scratch. Therefore, the goal is to minimize the construction time and maximize the efficiency of the data structure. The bounding volume hierarchy fulfills both criteria.

CUDA is a technology providing general purpose processing on graphics cards. Mapping construction algorithms of hierarchical data structures to massively parallel architecture of modern graphics cards is not trivial due to the parent-children dependencies. Subtle parallel algorithm design and careful implementation is a key to the efficiency. This thesis is about a fast parallel construction of bounding volume hierarchies on graphics cards.

1.1 Thesis Structure

The first part of this thesis is rather theoretical. We will provide a theoretical background to ray shooting and acceleration data structures. We will summarize well-known parallel algorithms. We will introduce the CUDA technology with a focus on terminology. We will describe general principles of construction of bounding volume hierarchies. We will present a survey on parallel construction algorithms of bounding volume hierarchies on graphics cards. The second part of this thesis is purely practical. We will describe the design of the rendering system and the implementation of the construction algorithms. We will present and discuss our results.

Chapter 2

Theoretical Background

In this chapter we will introduce the problem and summarize the related work. In the first section we will describe the visibility problem and the ray shooting algorithm. In the second section we will present a survey on the image synthesis algorithms based on ray shooting. In the third section we will summarize data structures used to accelerate ray shooting algorithm. In the last section we will describe some well-known parallel algorithms.

2.1 Ray Shooting and Visibility

A visibility problem is a fundamental issue of computer graphics. A visibility can be defined in terms of mathematical relations. The visibility for two points is a binary relation V on $\mathbb{E}^d$, where $\mathbb{E}^d$ denotes the d dimensional Euclidean space. Two points $\mathbf{x}$ and $\mathbf{y}$ in d dimensional Euclidean space are mutually visible if and only if the line segment $\overline{\mathbf{xy}}$ with endpoints $\mathbf{x}$ and $\mathbf{y}$ does not intersect any geometrical primitive [13]. The relation is trivially reflexive and symmetric. Using this definition we simply define a visibility function v.

$$v(\mathbf{x}, \mathbf{y}) = \begin{cases} 1 \text{ if } (\mathbf{x}, \mathbf{y}) \in V \\ 0 \text{ otherwise} \end{cases} \tag{2.1}$$

The ray shooting problem is similar to the visibility problem. For a given ray we want find the closest geometric primitive intersecting the ray. The ray $\mathbf{r} = (\mathbf{o}, \mathbf{d})$ is a semi-infinite line specified by its origin $\mathbf{o} \in \mathbb{E}^d$ and its direction $\mathbf{d} \in \mathbb{E}^d$ such that $\|\mathbf{d}\| = 1$. The parametric form expresses the ray $\mathbf{r}$ as a function of scalar value t [24].

$$\mathbf{r}(t) = \mathbf{o} + t\mathbf{d}, \ 0 \leq t < \infty \tag{2.2}$$

Using the parametric form we can simply derive ray-primitive intersection algorithms, e.g. ray-sphere, ray-triangle, etc. The geometric primitive is a compact subspace in $\mathbb{E}^d$ with $d-1$ dimensional continuous boundary, e.g. polygons, implicit surfaces, etc. Using the previous definitions we define the ray shooting problem. Formally, for a given ray $\mathbf{r} = (\mathbf{o}, \mathbf{d})$ and the set of geometric primitives P we want find the primitive $p \in P$ such that $p \cap \mathbf{r}(t) \neq \emptyset$ and t is the minimum. A primitive intersected the ray may not exist. A ray shooting algorithm is an algorithm providing a solution to the ray shooting problem. The visibility problem is

trivially reducible to the ray shooting problem, see figure 2.1. A naïve ray shooting algorithm sequentially tests all geometric primitives against a given ray. The time complexity of the naïve algorithm is $\mathcal{O}(n)$, where n is the number of primitives. For thousands of rays and thousands of geometric primitives the naïve algorithm is inapplicable. We can reduce the time complexity using acceleration data structures. The acceleration data structures are discussed further.

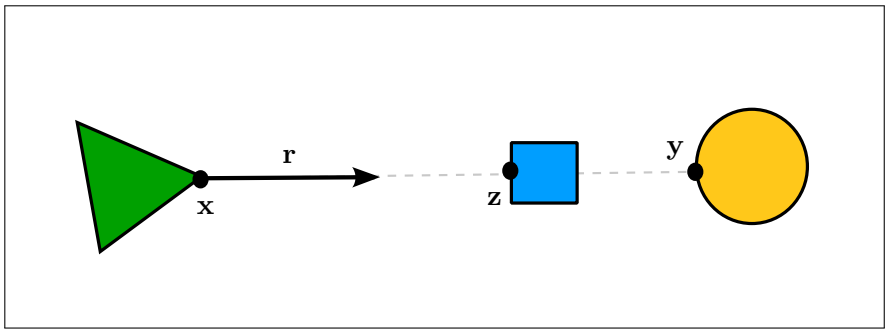


Figure 2.1: A scheme illustrating the visibility problem and the ray shooting algorithm. Points **x** and **y** are not mutually visible. Points **x** and **z** are mutually visible. The solution of the ray shooting problem for the ray **r** is the blue rectangle.

2.2 Image Synthesis

An image synthesis is a process of creating images. A photorealistic image synthesis tries to create images of a scene that are indistinguishable from the photograph of the same scene. To achieve this goal it is necessary to simulate the physics of light and its interaction with matter [24]. The geometric optics omits the wavelength of light and describes light propagation in terms of rays. The geometric optics assumes that the light travels instantaneously through the medium in straight lines and the light is not influenced by external factors, e.g. gravity or magnetic field [6]. The geometric optics model is sufficient for the image synthesis.

There is a class of image synthesis algorithms based on the ray shooting algorithm sometimes inaccurately termed as ray tracing. The fundamental idea is to shoot the rays through a virtual camera to the scene and trace them back to the light sources. Using the ray shooting algorithm we find the point of the interaction between the light ray and the surface. In the context of ray shooting the virtual scene is described as a set of geometric primitives. In this section we will present a brief survey on these image synthesis algorithms with the focus on the ray shooting algorithm.

2.2.1 Ray Casting

The first ray shooting based rendering algorithm was introduced by Appel [2] in 1968. The idea is to shoot a ray from the camera center through each pixel of the image plane and to find the closest primitive intersecting the ray. An intensity value of the pixel is then computed according to the material properties, light sources, and a given lighting model. Shadows can

be computed by shooting rays from the point of intersection to the light sources. If there is any primitive intersecting the ray then the point is occluded. Today this approach is known as ray casting.

2.2.2 Ray Tracing

This approach was extended by Whitted [29] in 1979. Primary rays are shot from the camera center to the scene and the closest intersections are computed. Shadow rays are shot to the light sources and the occlusion is computed. Secondary rays, reflected or refracted rays, are shot from the point of intersection. Each secondary ray is recursively traced. This process is depicted in figure 2.2. Today this approach is known as Whitted-style ray tracing. The disadvantage is that this approach is restricted to the perfect reflection and the perfect refraction. In 1984 Cook *et al.* [5] extended ray tracing with some realistic effects including depth of field, motion blur, glossy reflections, etc. This approach is called distributed ray tracing and it is based on modeling probability distributions of phenomena and stochastic sampling [27].

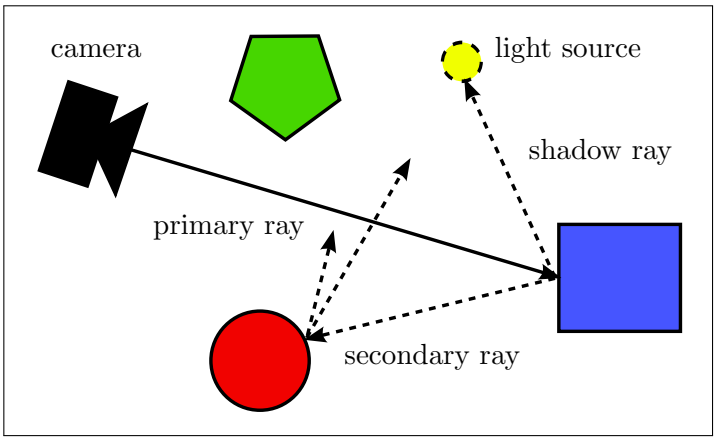


Figure 2.2: A scheme illustrating the ray tracing algorithm in 2D.

2.2.3 Path Tracing and Rendering Equation

Kajiya [15] introduced the rendering equation and a path tracing algorithm in 1986. The rendering equation expresses the scattering of light in the scene, which is the fundamental phenomenon in the photorealistic image synthesis.

$$L(\mathbf{x}, \boldsymbol{\omega}_o) = L_e(\mathbf{x}, \boldsymbol{\omega}_o) + \int_{\boldsymbol{\omega}_i \in \Omega} f_r(\mathbf{x}, \boldsymbol{\omega}_i, \boldsymbol{\omega}_o) L(\mathbf{x}, \boldsymbol{\omega}_i) \cos\theta_r d\boldsymbol{\omega}_i \tag{2.3}$$

The angle between the normal vector $\mathbf{n}$ of the point $\mathbf{x}$ and the direction vector $\boldsymbol{\omega}_i$ is denoted by θ_i. The set of all unit direction vectors $\boldsymbol{\omega}_i$ such that $\boldsymbol{\omega}_i \cdot \mathbf{n} > 0$ is denoted by Ω. The *bidirectional reflectance distribution function*, denoted by f_r, expresses how light is reflected on a surface. The total spectral radiance $L(\mathbf{x}, \boldsymbol{\omega}_o)$ leaving the point $\mathbf{x}$ in direction

$\boldsymbol{\omega}_o$ is sum of the emitted and reflected spectral radiance. The emitted spectral radiance is denoted by $L_e(\mathbf{x}, \boldsymbol{\omega}_o)$. The reflected spectral radiance is the integral over all vectors $\boldsymbol{\omega}_i$. The integrand is the spectral radiance reflected in the direction $\boldsymbol{\omega}_i$ attenuated by the term $\cos\theta_i$ and weighted by the value of the function f_r. The situation described by the rendering equation is depicted in figure 2.3.

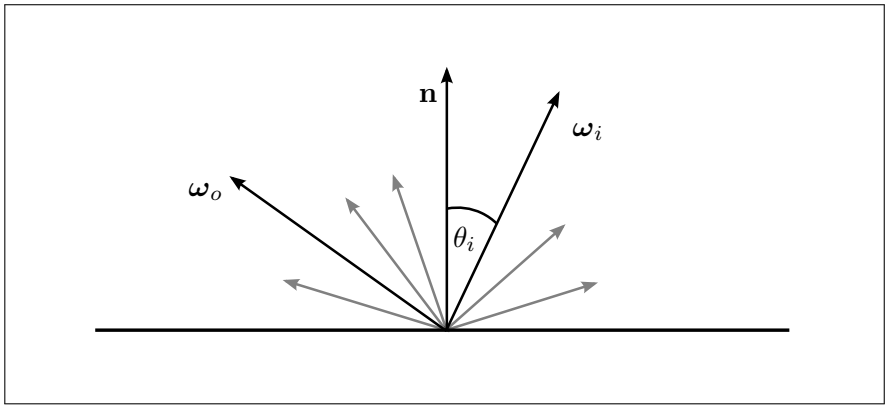


Figure 2.3: A scheme illustrating the rendering equation.

In a general case the path tracing algorithm is based on the rendering equation. It is impossible to solve the rendering equation analytically. The rendering equation must be solved numerically using Monte Carlo integration. The difficulty of the rendering equation is the recurrence relation within the integral. The idea is to generate paths from the camera to the light sources. A path is a sequence of surface points and it is generated by shooting a ray through the camera to the scene. At the nearest point of intersection a reflected ray is shot in a randomly sampled direction and the ray is traced recursively. Each path accumulates the partial spectral radiance according the rendering equation. The final spectral radiance of the pixel is computed by the averaging the partial results of the paths. One of the paths must hit a light source, otherwise the pixel is black. The presence of a noise depends on the number of samples per pixel.

2.3 Acceleration Data Structures

A naïve ray tracing algorithm has the time complexity $\mathcal{O}(mn)$ in the worst case, where m is the number of rays and n is the number of primitives. Whitted noted that up to 95% of the execution time is spent on ray-primitive intersection computations [29]. Therefore, it is desirable to decrease the amount of ray-primitive intersection tests.

Acceleration data structures exploit spatial coherency of objects and split the scene into small local coherent parts. During intersection computation are tested only parts spatial coherent with the ray, others are pruned. The scene can be split in a space or object domain. Thus, acceleration structures consist of two major classes: object and space partitioning structures.

2.3.1 Object Partitioning Structures

Object partitioning structures split the scene objects into disjoint subsets. Thus, no object has more than one entry in the structure and a memory footprint of the data structure is bounded. This approach is typically based on bounding volumes, whose associated space subsets may overlap.

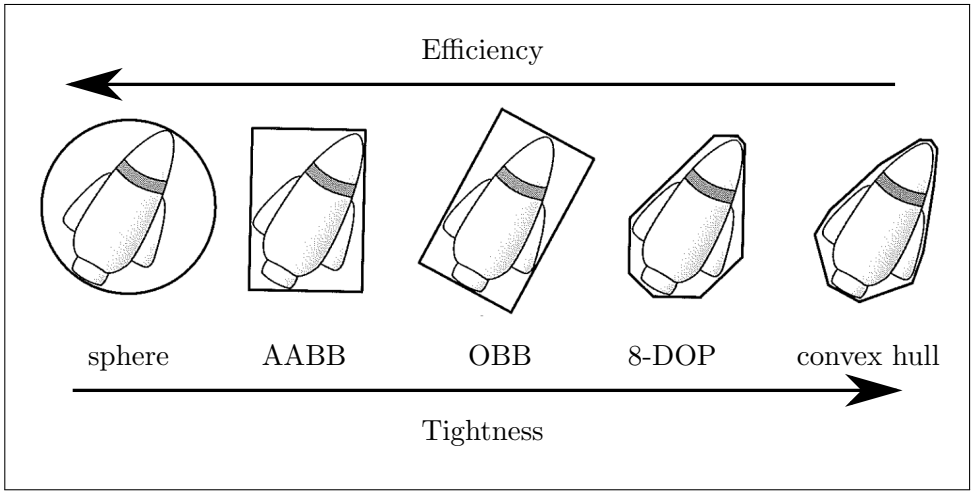


Figure 2.4: Bounding volumes, courtesy of Ericson [7].

Bounding Volumes: The idea is to enclose complex scene objects in a simpler bounding volume. Bounding volume should be efficient and tight [13, 17]. Efficiency means that the intersection test between the bounding volume and a ray is cheap. Tightness is a measure to hit the bounded object by a ray when the bounding volume is hit by the same ray. These two criteria are opposite and must be balanced. The idea of bounding volumes is not only restricted to the ray tracing algorithm. The concept of bounding volumes is used in collision detection or occlusion culling algorithms. Examples of bounding volumes are depicted in figure 2.4.

Bounding Volume Hierarchies: A bounding volume hierarchy is a natural extension of the bounding volumes. The bounding volume hierarchy was originally introduced by Rubin and Whitted [25] in 1980. In general the bounding volume hierarchy is a rooted tree of an arbitrary branching factor. Usually the bounding volume hierarchy has the branching factor two. Leaf nodes contain geometric primitives. Interior nodes contain references to descendants and bounding volume enclosing all descendants. A root node contains a bounding volume enclosing the whole scene.

A ray query is evaluated by traversing the tree from the root node to the leaf nodes. Branches whose bounding volumes are not hit are pruned. Primitives in leaf nodes are tested sequentially. An example of bounding volume hierarchy is depicted in figure 2.5.

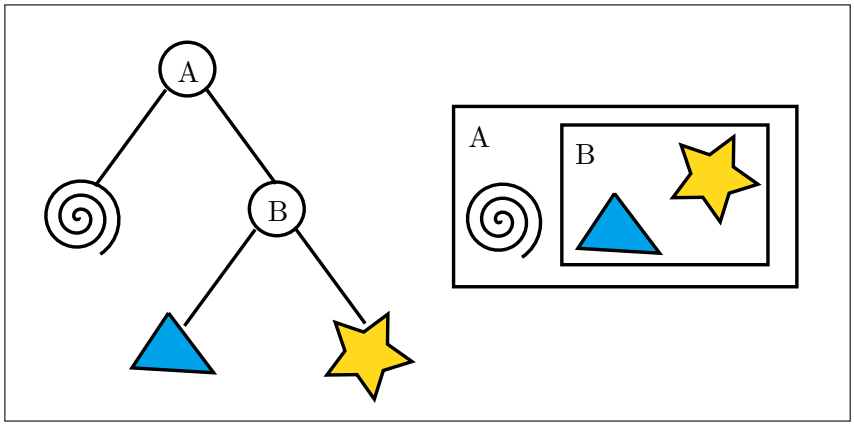


Figure 2.5: An example of the bounding volume hierarchy in 2D.

2.3.2 Space Partitioning Structures

Space partitioning structures subdivide the scene space box into disjoint subsets. Scene objects are split together with the scene space. Thus, a geometric primitive may have more than one entry in the data structure and the memory footprint of the data structure is unbounded.

Uniform Grids: The simplest space partitioning structure is an uniform grid. The idea is to subdivide the scene by axis aligned splitting planes with the uniform spacing [13]. The scene space is split into equally sized space elements. Efficiency of the uniform grid is heavily dependent on the scene data distribution. In dense scenes all space elements are uniformly occupied and the efficiency is good. On the other hand, in sparse scenes almost all space elements are empty and few elements contain many geometric primitives and the efficiency is poor. A ray traversal algorithm for the uniform grid is known as 3D-DDA [8], which is similar to well-known DDA algorithm. An example of the uniform grid is depicted in figure 2.6.

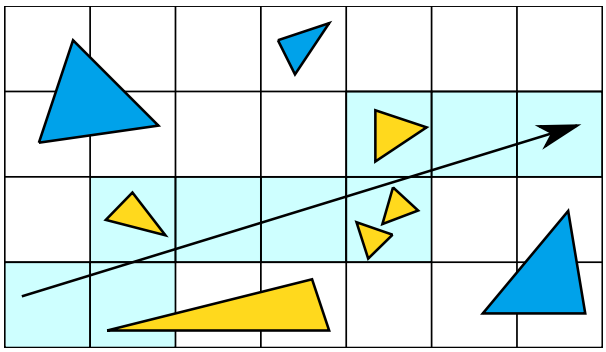


Figure 2.6: An example of the uniform grid in 2D, yellow objects are tested.

kD-Trees: A kD-tree was designed by Bentley [3] in 1975. The kD-tree is a rooted binary tree, which recursively subdivides the scene space by axis aligned splitting planes. Leaf nodes contain geometric primitives. Interior nodes contain the position of the splitting plane, the axis of the plane and references to descendants. The kD-tree is a special case of a BSP tree, which subdivides the space by planes in general positions.

A ray traversal algorithm for the kD-tree proceeds in similar manner as the ray traversal algorithm for the bounding volume hierarchies. The kD-tree is traversed from the root node to leaf nodes, but pruning is done more efficient. The ray traversal algorithm distinguishes up to thirteen different cases based on the node splitting plane position, the node bounds, the origin and the direction of the ray. Detailed analysis of the kD-tree can be found in the Havran's Ph.D. thesis [13]. An example of the kD-tree is depicted in figure 2.7.

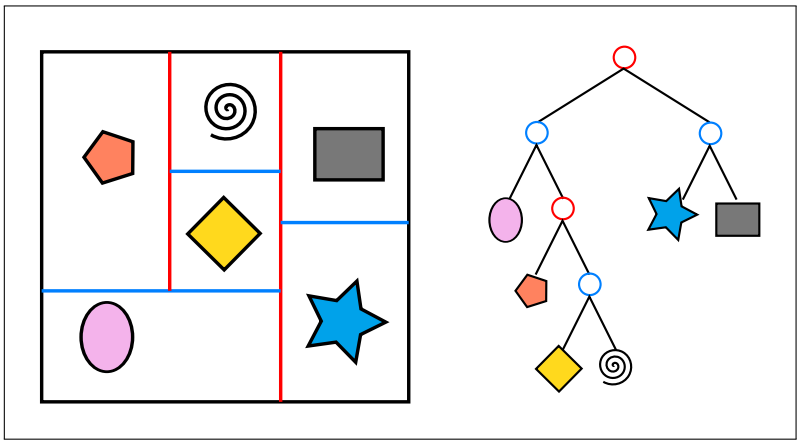


Figure 2.7: An example of the kD-tree in 2D.

Octrees: An octree is well-known data structure used for modeling 3D solid objects. Glassner [10] introduced the octree in the context of ray tracing. The octree is a rooted tree with the branching factor eight. Thus, all interior nodes have exactly eight children. Leaf nodes contain geometric primitives. The octree recursively subdivides the scene space by three orthogonal planes which are placed in the middle of a subdivided space element.

A ray traversal algorithm for the octree is similar to the algorithm for the kD-tree, but it is more complicated. At most four of eight children are visited if the parent node is hit [13]. It is due to the linear nature of the ray and the orthogonality of the splitting planes. An example of the octree is depicted in figure 2.8.

2.4 Parallel Computing

In the first part of this section we will describe two the most common parallel algorithms: parallel reduction and parallel prefix scan. In the second part of this section we will explain the parallel radix sort algorithm.

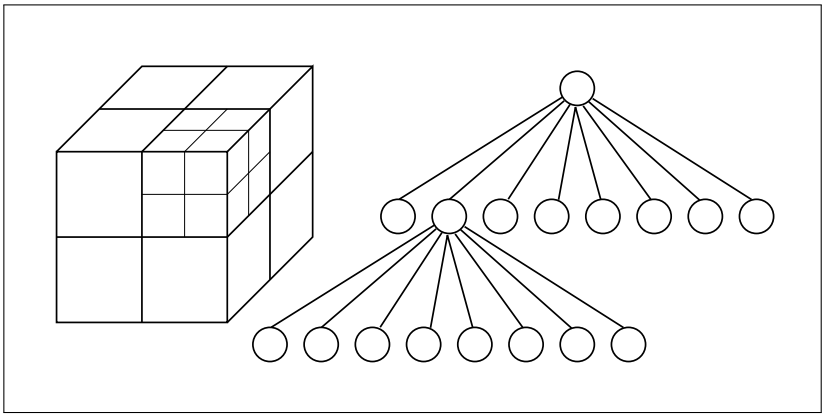


Figure 2.8: An example of the octree.

2.4.1 Reduction

The reduce operation takes a binary associative operator $\oplus$ and the ordered sequence

$$a_0, a_1, \ldots, a_{n-1} \tag{2.4}$$

of n elements and returns the value

$$a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}. \tag{2.5}$$

The sequential algorithm of the reduction is straightforward. We scan the sequence element by element and simultaneously update the partial result of processed elements. The time complexity of the sequential algorithm is $\mathcal{O}(n)$. The parallel algorithm is based on the divide and conquer paradigm. The sequence is recursively divided into halves until a single element is left. In interior nodes of a recursion tree partial results are merged using the $\oplus$ operator. The pseudocode of the algorithm is shown in algorithm 1. The depth of the recursion tree is $\lceil \log_2 n \rceil$, and hence the time complexity for $\frac{n}{2}$ processor is $\mathcal{O}(\log n)$. The time complexity for p processors such that $p < \frac{n}{2}$ is $\mathcal{O}(\left\lceil \frac{n}{p} \right\rceil + \log p)$. Each processor requires $\left\lceil \frac{n}{p} \right\rceil$ time steps and merging partial results of each processors takes another $\log_2 p$ steps.

Algorithm 1 Parallel Reduction, courtesy of Blelloch [4].

1: **for** $d \leftarrow 0$ **to** $\log_2 n - 1$ **do**
2: **for** $i \leftarrow 0$ **to** $n - 1$ **by** 2^{d+1} **in parallel do**
3: $a_{i+2^{d+1}-1} \leftarrow a_{i+2^d-1} \oplus a_{i+2^{d+1}-1}$
4: **end for**
5: **end for**

2.4.2 Prefix Scan

There are two types of prefix scans: inclusive and exclusive. Both prefix scans take a binary associative operator $\oplus$ and the following ordered sequence of n elements.

$$a_0, a_1, \ldots, a_{n-1} \tag{2.6}$$

Inclusive prefix scan: The inclusive prefix scan returns the following ordered sequence.

$$a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}) \tag{2.7}$$

Exclusive prefix scan: The exclusive prefix scan requires an identity element e such that $(a_i \oplus e) = (e \oplus a_i) = a_i$ for arbitrary a_i. The exclusive prefix scan returns the following ordered sequence.

$$e, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2}) \tag{2.8}$$

The exclusive prefix scan can be constructed from the inclusive prefix scan by removing the last element and inserting the identity element at the beginning. The inclusive prefix scan can be constructed from the exclusive prefix scan by removing the identity element and inserting the sum of the last element of the input sequence and the last element of the exclusive prefix scan at the end. The sequential algorithm is also straightforward. We scan the input sequence element by element and simultaneously write the partial results of the output sequence. The time complexity of the sequential algorithm is $\mathcal{O}(n)$. There are two parallel prefix scan algorithms.

Algorithm 2 Hillis-Steele algorithm, courtesy of Nvidia [21].

```
1: for d ← 0 to log2(n) − 1 do
2:     for i ← 0 to n − 1 in parallel do
3:         if i ≥ 2^d then
4:             a_i ← a_{i−2^d} ⊕ a_i
5:         end if
6:     end for
7: end for
```

Hillis-Steele Algorithm: This algorithm was designed by Hillis and Steele [14] in 1986. This algorithm computes inclusive prefix scan. The pseudocode of the algorithm is shown in algorithm 2. Steps of the algorithms are depicted in figure 2.9. This algorithm requires $\mathcal{O}(n \log n)$ computational steps. The time complexity for $\frac{n}{2}$ processors is $\mathcal{O}(\log n)$. The time complexity for p processors such that $p < \frac{n}{2}$ is $\mathcal{O}(\left\lceil \frac{n}{p} \right\rceil + \log p)$. This algorithm is suitable for small inputs.

Blelloch's Algorithm: This algorithm was designed by Blelloch [4] in 1990. This algorithm computes exclusive prefix scan. The algorithm consists of two passes. In the first pass the parallel reduction is computed. The first pass is sometimes called up-sweep and

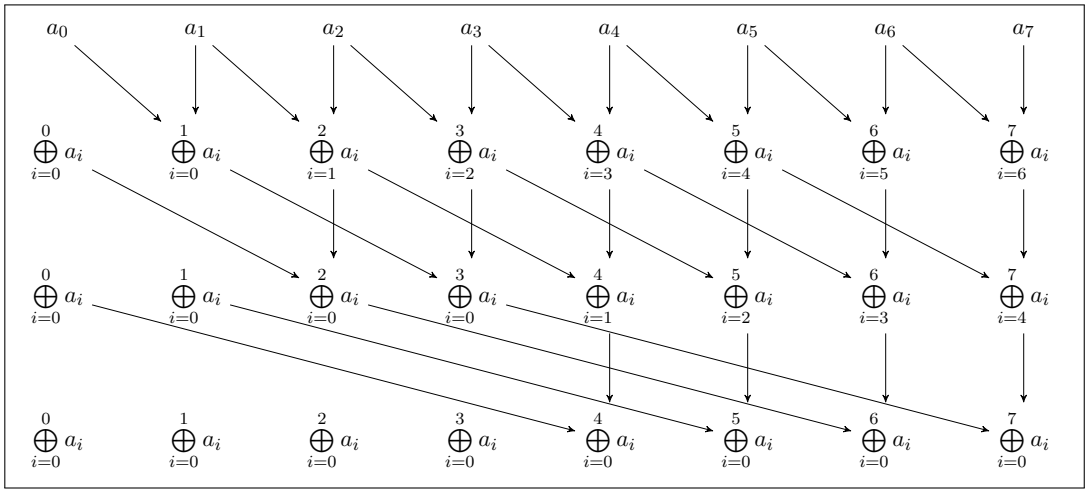


Figure 2.9: Hillis-Steele algorithm for 8 elements.

it is depicted in figure 2.10. The second pass is called down-sweep. The pseudocode of the second pass is shown in algorithm 3.

Steps of the algorithm are depicted in figure 2.11. A tree structure can be resembled in the figure. Each interior node of the tree contains sum of all leaves preceding the node in the preorder traversal. The root value is set to e, because there are no leaves preceding the root. Each left child node has the same number of preceding leaves as its parent node, hence each left child node has the same value as its parent node. A value of each right child node is the sum of the parent value and the left sibling value.

The advantage is that this algorithm requires only $\mathcal{O}(n)$ computational steps. The time complexity is the same as the time complexity of Hillis-Steele algorithm. The algorithm is suitable for large inputs.

Algorithm 3 Blelloch's algorithm, down-sweep, courtesy of Blelloch [4].

1: $a_{n-1} \leftarrow 0$
2: **for** $d \leftarrow \log_2(n) - 1$ **downto** 0 **do**
3: **for** $i \leftarrow 0$ **to** $n - 1$ **by** 2^{d+1} **in parallel do**
4: $t \leftarrow a_{i+2^d-1}$
5: $a_{i+2^d-1} \leftarrow a_{i+2^{d+1}-1}$
6: $a_{i+2^{d+1}-1} \leftarrow t \oplus a_{i+2^{d+1}-1}$
7: **end for**
8: **end for**

2.4.3 Radix Sort

A radix sort is well-known sorting algorithm. Keys are represented as a sequences of digits. The idea is to sort keys successively according to all digits from the least significant to the most significant ones. Using a stable sort algorithm we get a lexicographic ordering of

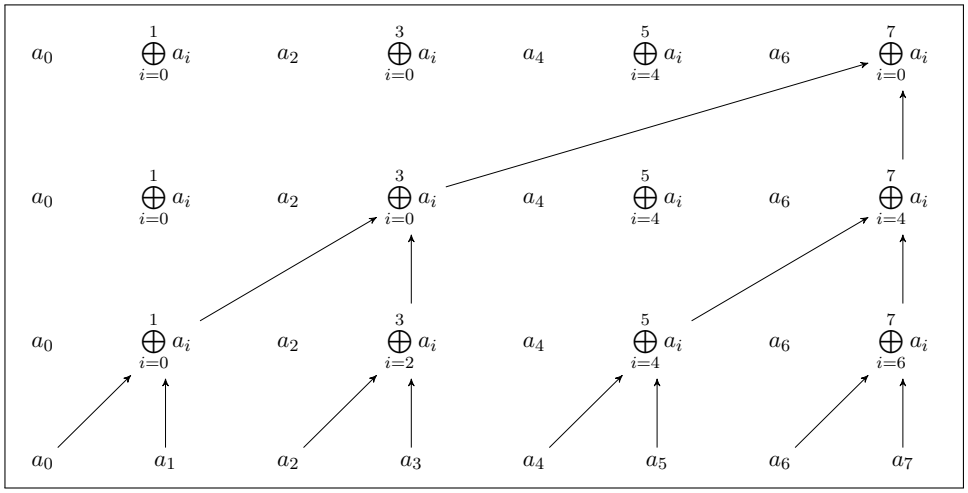


Figure 2.10: Blelloch's algorithm, up-sweep phase for 8 elements.

the keys. In each pass of the algorithm a d-bit digit of k-bit keys is processed. Keys are distributed into 2^d buckets according to the current digit. The total number of passes is $\lceil \frac{k}{d} \rceil$ [20]. The advantage is the linear time complexity $\mathcal{O}(n)$.

The distribution of keys can be easy parallelized via exclusive prefix scan. The algorithm requires 2^d flag vectors of length n. The current digits of keys are examined. For i-th key the i-th flag of the particular vector is set to 1. Performing exclusive prefix scan on the concatenated vectors we obtain output indices for each keys. The parallel time complexity for n processors is $\mathcal{O}(\log n)$. The time complexity for $p < n$ processor is $\mathcal{O}(\left\lceil \frac{n}{p} \right\rceil + \log p)$. An example of the algorithm is depicted in figure 2.12.

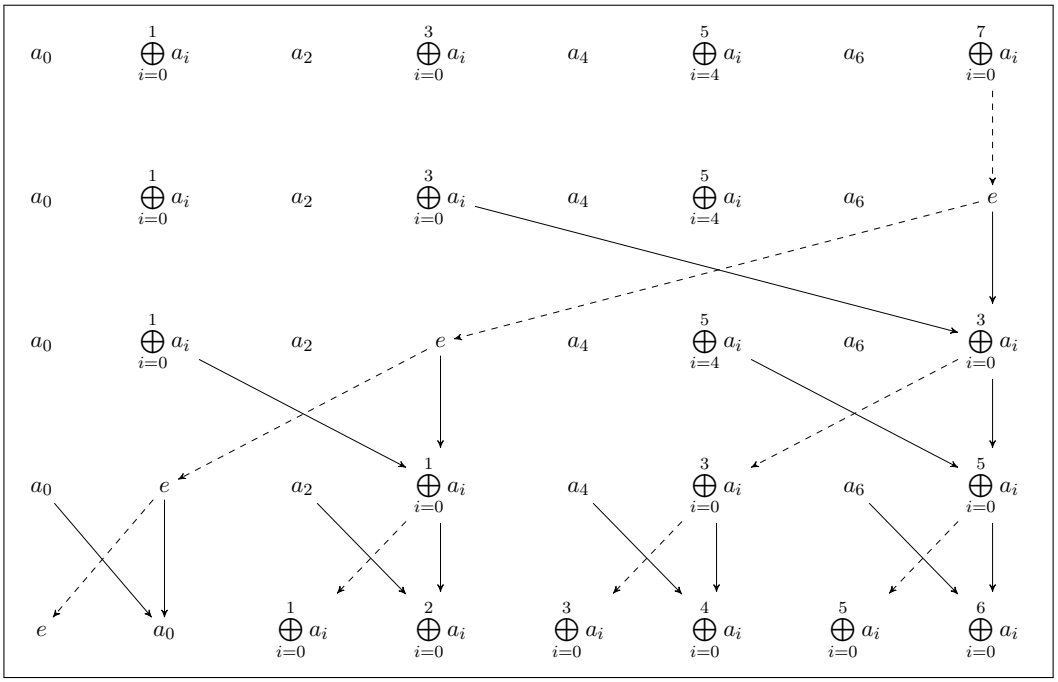


Figure 2.11: Blelloch's algorithm, down-sweep phase for 8 elements.

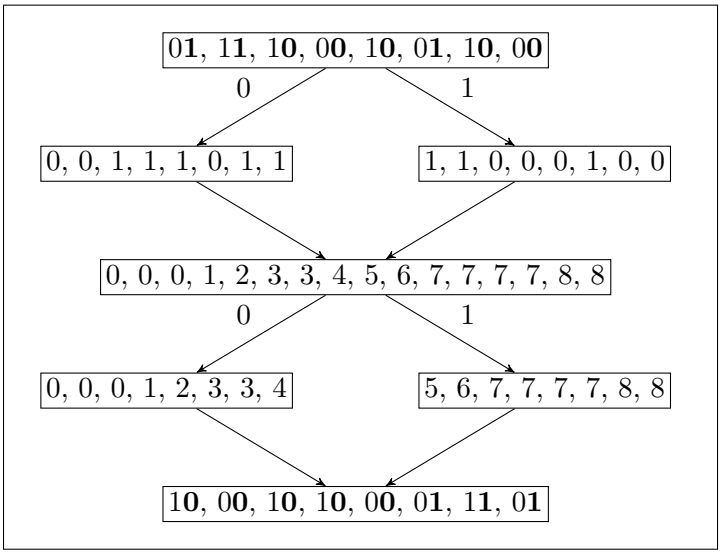


Figure 2.12: An example of radix sort algorithm such that $d = 1$. The current bold digits are examined. According to the digit value the corresponding flag of the particular vector is set to 1. Performing exclusive prefix scan on the concatenated vectors we obtain output indices.

Chapter 3

Compute Unified Device Architecture

The graphics card is a standard component of modern computers. Graphics processing units are based on massively parallel architecture. Thousands of threads can be executed simultaneously on the GPU, e.g. each pixel is processed in parallel. Therefore, the GPU is a potential source of tremendous computational power [22]. Compute Unified Device Architecture is a technology providing a general purpose processing on the GPU introduced by Nvidia in 2007. Thus, complex general tasks, e.g physics simulation or numerical computing, can be significantly accelerated. The disadvantage is that the CUDA technology is only implemented by GPUs manufactured by Nvidia. The computational power of GPUs is expressed as a *compute capability* value. The higher compute capability generally means more computational resources.

3.1 Programming Model

The communication between the CPU and the GPU is based on well-known client-server model. In the context of CUDA the CPU is termed as *host* and the GPU is termed as *device*. A kernel function is a function executed by threads in parallel on the GPU. The kernel function must be declared with the `__global__` attribute. Threads are organized in a two-level hierarchy. Threads are organized into blocks and blocks are organized into a grid [18]. A Block is a 1D to 3D array of threads and a grid is a 1D to 3D array of blocks. On GPUs with low compute capability the third dimension is limited. Each thread has a unique index within the block and each block has a unique index within the grid. According to these indices each thread identifies the data to be processed. Before launching the kernel the programmer specifies the extent of the blocks and the grid. The thread hierarchy is depicted in figure 3.1. Threads within the block can be efficiently synchronized using the function `__syncthreads()`. All threads in the block meet up at the point of call this function. The global synchronization is possible using atomic operations.

CUDA supports many programming languages. We will focus on C and C++ programming languages. CUDA C is a minimal extension of the C and C++ programming languages [22]. Almost all features of the C++ programming language are supported in kernels. The

kernels are launched and managed from the CPU via either the runtime or driver API. The runtime API is a high-level API that compiles and links kernel code into executables. The management of a CUDA context and CUDA modules is implicitly done by the runtime API. The driver API is a low-level API. A CUDA context and CUDA modules must be managed explicitly by the programmer. The driver API is harder to program, but it provides more control.

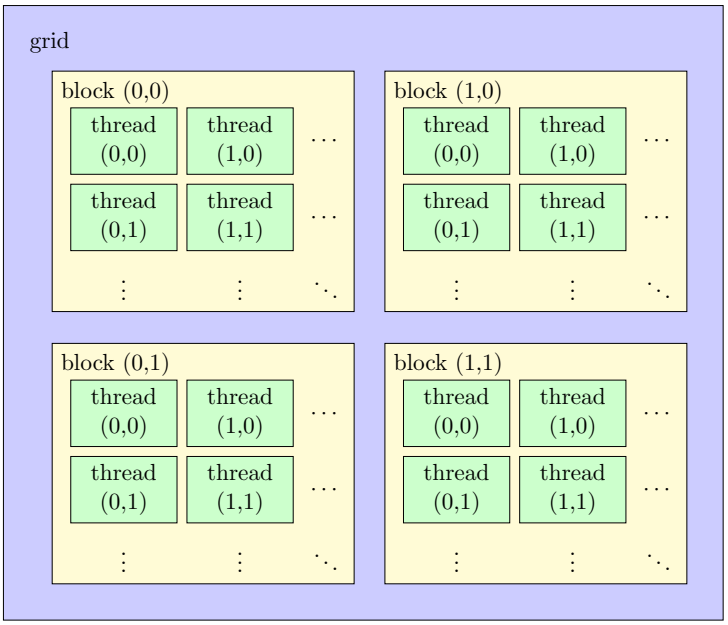


Figure 3.1: The CUDA thread hierarchy.

3.2 Memory Model

The CUDA memory model consists of six types of memories: registers, local memory, shared memory, constant memory, global memory, and texture memory. Different memory types have different access latency, different scope, and are suitable for different usage. The choice of the appropriate memory type is crucial for the efficiency. The memory model is depicted in figure 3.2.

Registers: The registers are the fastest on-chip memory. The registers are allocated to individual threads and each thread can access its own registers [18]. Unfortunately the number of registers per block is limited. The size of each register is 32 bits. Local variables in kernels are usually stored in registers.

Local Memory: The local memory is a part of the global memory and its access latency is the same as the access latency of global memory. The local memory is cached only on

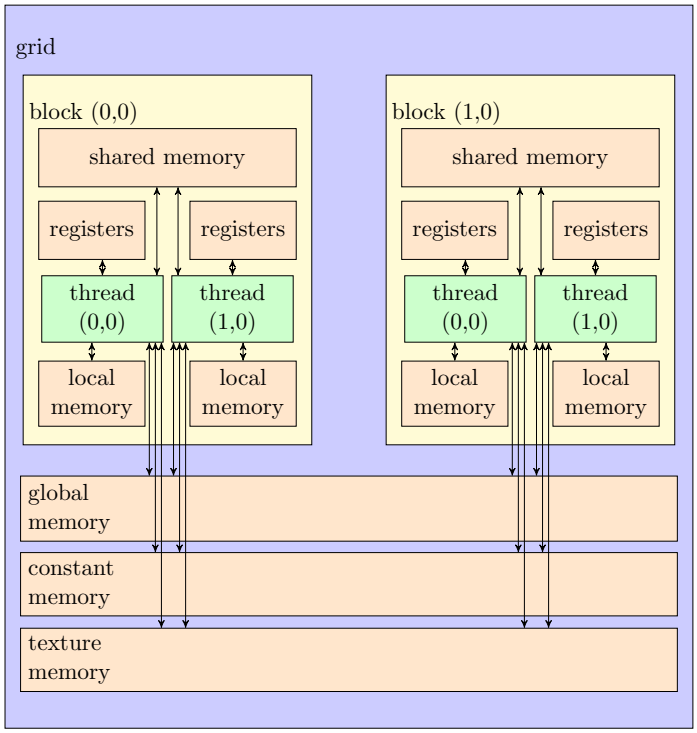


Figure 3.2: The CUDA memory model.

GPUs with the compute capability 2.0 or higher. All local variables which cannot fit into registers are stored in the local memory [22].

Shared memory: The shared memory is a very fast on-chip memory. Efficiency of shared memory access is comparable with the efficiency of registers. The scope of shared memory is the block of threads. Variables allocated in the shared memory must be declared with `__shared__` attribute within the kernel function. The block synchronization and the shared memory provides an efficient communication between threads in the block [18, 22].

Global Memory: The global memory is inefficient off-chip memory. The global memory access takes 400 to 600 cycles. The latency can be reduced using coalesced access pattern. The global memory is cached only on GPUs with the compute capability 2.0 or higher. The global memory is accessible to all threads. Variables allocated in the global memory must be declared with the `__device__` attribute outside of the kernel function. In global memory are usually stored large input or output data.

Constant Memory: The constant memory is a part of the global memory. The constant memory is read-only and it is cached. Variables allocated in the constant memory must be declared with the `__constant__` attribute outside the kernel function. The size of data stored in the constant memory must be known at the compile time [18]. The constant

memory is suitable for read-only random accesses. The disadvantage is that the size of the memory is limited to 64 kB.

Texture Memory: The texture memory is a part of the global memory. The texture memory is read-only and it is cached. The texture memory is accessible via special hardware texture units. The advantage is that the size of the texture memory is not limited as the size of the constant memory. The texture memory is suitable for spatially coherent data. The size of a texture element is 4, 8, or 16 bytes. The texture space is 1D to 3D. The values between samples can be efficiently interpolated using the special texture units.

3.3 Execution Model

GPUs consist of device memory, caches and streaming multiprocessors. The streaming multiprocessors consist of shared memory, registers and streaming processors, also known as CUDA cores. The simplified GPU architecture is depicted in figure 3.3. An exact structure of the GPU depends on the compute capability. The CUDA execution model defines how the blocks of threads are mapped to streaming multiprocessors. Threads are scheduled in warps, i.e. group of 32 concurrent threads, and executed on streaming multiprocessors. This architecture is termed as SIMT which means *single instruction multiple threads* [22].

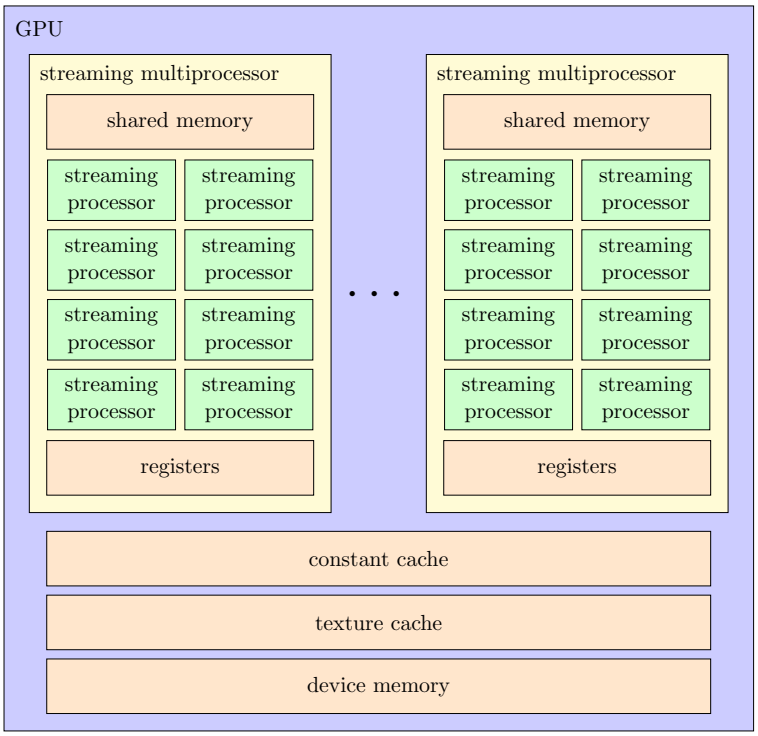


Figure 3.3: A simplified GPU architecture.

Chapter 4

Construction of Bounding Volume Hierarchies

We already know that the bounding volume hierarchy is a rooted tree of an arbitrary branching factor. Interior nodes contain bounding volumes enclosing all descendants and leaf nodes contain geometric primitives. Further we will focus on bounding volume hierarchies with branching factor two and with axis aligned bounding boxes.

The advantage of the bounding volume hierarchies, unlike space partitioning structures, is that can be constructed in three different manners: top-down, bottom-up and incremental.

4.1 Cost Model

Before we will describe various construction methods we will define a simple cost model. The cost of a bounding volume hierarchy can be expressed by the following recurrence equation.

$$c(N) = \begin{cases} k_T + P(N_L|N)c(N_L) + P(N_R|N)c(N_R) & \text{if N is interior node} \\ k_I|N| & \text{otherwise} \end{cases} \tag{4.1}$$

Where $c(N)$ is the cost of a (sub)tree with the root node N, k_T is the average cost of a traversal step, k_I is the average cost of a ray-primitive intersection test, N_L is the left child of the node N, N_R is the right child of the node N, $P(N_L|N)$ is the probability of traversing the node N_L, $P(N_R|N)$ is the probability of traversing the node N_R, and $|N|$ is the number of primitives in the node N.

We assume that the distribution of the rays is uniform and that are not blocked. Under these assumptions the conditional probabilities in equation 4.1 can be expressed as geometric probabilities. The geometric probability in our case is the ratio between the surface area of the child node and the parent node [27].

$$P(N_L|N) = \frac{SA(BV(N_L))}{SA(BV(N))} \tag{4.2}$$

$$P(N_R|N) = \frac{SA(BV(N_R))}{SA(BV(N))} \tag{4.3}$$

$SA(Y)$ denotes the surface area of the bounding volume Y and $BV(X)$ denotes the bounding volume of the node X. After substituting equations 4.2 and 4.3 into equation 4.1 we get equation 4.4.

$$c(N) = \begin{cases} k_T + \frac{SA(BV(N_L))}{SA(BV(N))} c(N_L) + \frac{SA(BV(N_R))}{SA(BV(N))} c(N_R) & \text{if N is interior node} \\ k_I |N| & \text{otherwise} \end{cases} \tag{4.4}$$

Unrolling the recurrence relation in equation 4.4 we get equation 4.5.

$$c(N) = k_T \sum_{N_i} \frac{SA(BV(N_i))}{SA(BV(N))} + k_I \sum_{N_l} \frac{SA(BV(N_l))}{SA(BV(N))} |N_l| \tag{4.5}$$

N_i denotes an interior node of a (sub)tree with the root node N, N_l denotes a leaf node of a (sub)tree with the root node N and $|N_l|$ denotes the number of primitives in the leaf node N_l.

4.2 Top-Down Construction

The most common way to construct a bounding volume hierarchy is a recursive division of the scene primitives. At the beginning we have a root node which contains the scene bounding box and all primitives. In each step of the construction one leaf node is processed. The leaf node is split into two new leaves which are further processed. The primitives of the split node are reordered into its children and bounding volumes of the children are computed. The division continues until one of the termination criteria is reached. The common termination criteria are maximum primitives in a leaf, maximum tree depth or maximum memory used. The advantage is that this approach is fast and can be parallelized.

The number of different trees is exponentially proportional to the number of primitives [11]. For each node there are $\mathcal{O}(2^n)$ ways how to reorder primitives into its children, where n is the number of primitives in the node. Some hierarchies are better than others. The question is how to split the node to build a good hierarchy.

There are three basic approaches how to split the node: object median, spatial median and *surface area heuristic* cost minimization. All three approaches try to separate the primitives by an axis aligned plane. The separation is not always possible, hence the primitives are approximated to points, e.g. a centroid of a primitive. The set of points can be always separated by a plane. All three splitting axes are used or one axis is chosen according to some heuristic. The greatest extent of the bounding box or round-robin are the most common ones.

4.2.1 Surface Area Heuristic

A surface area heuristic was originally proposed by Goldsmith and Salmon [11]. The number of possible splits by an axis aligned plane is only $\mathcal{O}(n)$, where n is the number of primitives in the node. The idea is to express a cost of each splitting plane and choose a plane with the lowest cost. The following cost function is derived from equation 4.4.

$$c(N, N_L, N_R) = k_T + k_I \left(\frac{SA(BV(N_L))}{SA(BV(N))} |N_L| + \frac{SA(BV(N_R))}{SA(BV(N))} |N_R| \right) \tag{4.6}$$

Where $|N_L|$ denotes the number of primitives in the node N_L and $|N_R|$ denotes the number of primitives in the node N_R. The recurrence relation has been replaced simply by the number of primitives in the child nodes. Surface area heuristic is a greedy local approximation of the cost function in equation 4.1. The expression in equation 4.6 assumes that the child nodes are leaves. Another termination criterion can be derived from equation 4.6. The splitting is terminated if the minimum cost of the split is greater than $k_I|N|$ [27]. There are two basic methods how to sample splitting planes.

Sweeping: This method is known as full SAH evaluation. All splitting planes are evaluated and a plane with the lowest cost is chosen. This approach requires three sorted arrays of primitive indices ordered according to the three coordinates. This method produces trees of higher quality, but it is rather slow.

Binning: This method was introduces by Ingo Wald [28]. It is known as SAH approximation. This method is limited to hierarchies of axis aligned bounding boxes. The bounding box of the current node is uniformly subdivided into k spatial regions. Each region is associated with a bin. Primitives are projected to the bins according the following equation.

$$bin(\mathbf{c}, j) = \left\lfloor \frac{k(1-\varepsilon)(BB_j^{max} - \mathbf{c}_j)}{BB_j^{max} - BB_j^{min}} \right\rfloor \tag{4.7}$$

Where $bin(\mathbf{c}, j)$ is the index of the bin, $\mathbf{c}$ is a centroid of the projected primitive, j is splitting dimension, BB_j^{min} is j-th component of the minimum bound vector and BB_j^{max} is j-th component of maximum bound vector. The expression $1 - \varepsilon$ ensures that the primitive lying on the maximum bound will be projected to the last bin. Each bin accumulates the number of projected primitives and a bounding box of the projected primitives. Two prefix scans of bins are performed and exactly $k - 1$ splitting planes are evaluated. Sometimes all primitives may fall into the same bin, then the object median split must be done. The advantage is scalability of the quality. If k is high then the construction is slow, but the quality is better. If k is low then the construction is fast, but the quality is rather low.

4.2.2 Spatial and Object Median

Spatial median splits were introduced by Kay and Kajiya [17]. The bounding box is split in the middle. The advantage is that this splitting method is very fast. The quality of the hierarchy is dependent on the spatial coherency of the scene.

An object median method splits the primitives into halves. This method is fast, but produces hierarchies of low quality. The advantage of this method is that it never fails until the node contains only one primitive. This method is used if binning or spatial median fails.

4.3 Bottom-Up Construction

This approach is also known as agglomerative clustering. At the beginning all primitives are considered as clusters. In every step of the construction the closest clusters are merged together based on a given distance function. At the end of algorithm there is only one cluster. The advantage is that this approach produces hierarchies of higher quality. The disadvantage is worse time complexity. A naïve algorithm has the time complexity $\mathcal{O}(n^3)$ in the worst case [12].

4.4 Incremental Construction

An incremental construction was first introduced by Jeffrey Goldsmith and John Salmon [11]. Objects are piece by piece inserted into the leaves, which are consequently split into new leaves. All bounding boxes from the new leaves to the root must be refitted. The advantage of the incremental construction is that we don't need to know a whole input at the beginning of the construction. The disadvantage is that constructed hierarchies suffer from lack of quality.

Chapter 5

Bounding Volume Hierarchies on the GPU

Since GPGPU was introduced researchers have made a remarkable effort to map traditional sequential algorithms to a massive parallel GPU architecture. Many topics are still open. Efficient construction of bounding volume hierarchies is still a challenge. In this chapter we will present a survey on the construction algorithms on the GPU.

It is natural to seek inspiration in parallel algorithms on the CPU. Ingo Wald [28] noted that a tree construction can be parallelized in horizontal or vertical way. The vertical way means a parallel processing of subtrees. The horizontal way means parallel processing of splitting a single node. All further presented algorithms are based on vertical or horizontal processing.

5.1 Morton Curve and Spatial Median Splits

The Morton curve is a space filling curve. The order along the curve is given by $3k$-bit Morton codes. The space filled by the curve is subdivided into a grid of $2^k \times 2^k \times 2^k$ space elements. Each $3k$-bit Morton code corresponds to one space element in the grid. The advantage is that the mapping between the Morton codes and space elements coordinates is very simple. A $3k$-bit Morton code can be computed by interleaving successive bits of the corresponding space element coordinates. The following two equations define the relation between space element coordinates and the Morton code.

$$\mathbf{x} = [x, y, z] = [x_k \dots x_1, y_k \dots y_1, z_k \dots z_1] \tag{5.1}$$

$$m = z_k y_k x_k \dots z_1 y_1 x_1 \tag{5.2}$$

A space element $\mathbf{x}$ corresponds to a $3k$-bit Morton code m, where z_i, y_i, x_i are i-th bits of the coordinates. The Morton curve can be generalized to an arbitrary dimensions. An example of the Morton Curve in 2D is depicted in figure 5.1.

Spatial median splits can be efficiently done using the Morton curve. Each geometric primitive is represented by one point, e.g. a centroid of the primitive. The coordinates of

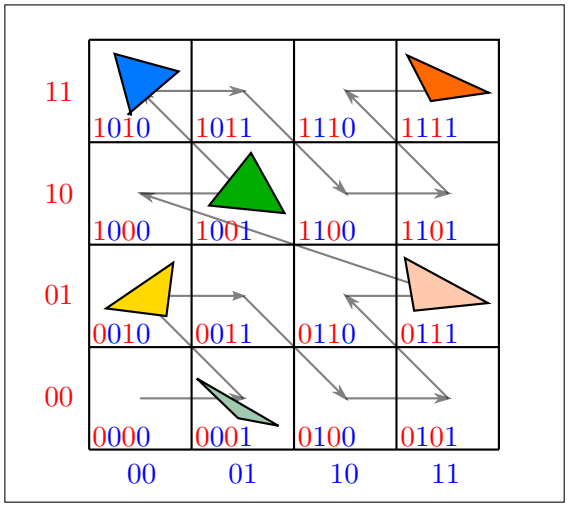


Figure 5.1: An example of the Morton curve and 4-bit Morton codes in 2D.

the points are quantized into k bit integers with the respect to the scene bounding box. These coordinates are mapped to the $3k$-bit Morton codes. Primitives are ordered along the Morton curve. The hierarchy is constructed by examining the most significant bits of the Morton codes and bucketing primitives to the appropriate children. Primitives with the current bit 0 are placed to the left child, others are placed to the right child.

5.2 Linear Bounding Volume Hierarchies

In 2009 Lauterbach *et al.* [19] introduced two novel construction algorithms on GPU. The first algorithm is based on the Morton curve and spatial median splits. The second algorithm is based on the SAH cost minimization splits.

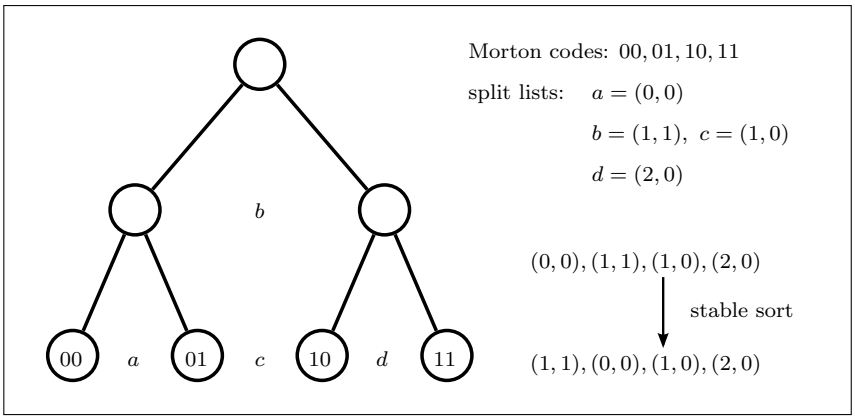


Figure 5.2: An example of the construction of split lists with four 2-bit Morton codes.

5.2.1 Spatial Median Splits

Assume that we have a sorted sequence of the Morton codes corresponding to the primitives. Bits of Morton code encode the path from the root to the leaf containing the primitive. The most significant different bit of two consecutive Morton codes identifies the least common ancestor, the farthest node from the root containing both primitives. The idea is based on determining the least common ancestors and on the construction of split lists.

All consecutive Morton code pairs are examined in parallel. Each thread determines the most significant different bit and constructs a split list. If the h-th bit is the most significant different bit, then the split list of the i-th primitive contains pairs $(i, h), (i, h-1) \dots (i, 0)$. A global split list is constructed by concatenation of individual split lists. The global split list is an ordered sequence of the first component of pairs. Using a stable sort algorithm on the global split list we get an ordered sequence of the second component of pair in decreasing order. The global list encodes all information about hierarchy topology. The hierarchy is constructed level by level using the global list. The construction of split lists is depicted in figure 5.2.

The disadvantage is that singletons, interior nodes with a single child, may occur. The singletons have to be removed with an additional post-process. The disadvantage is also the global list sorting. The number of global list items might be significantly larger than the number of primitives.

5.2.2 SAH Splits

The algorithm works with two task queues. Each task corresponds to the node which will be processed. At the beginning the input queue contains only a task corresponding to the root node. Each task is processed by a block of $3t$ threads. The bounding box of the node is uniformly subdivided by $3t$ axis align planes, t planes for each axis. Each thread evaluates the SAH cost of the corresponding splitting plane. A parallel reduction using minimum operator is performed to find a plane with the lowest cost.

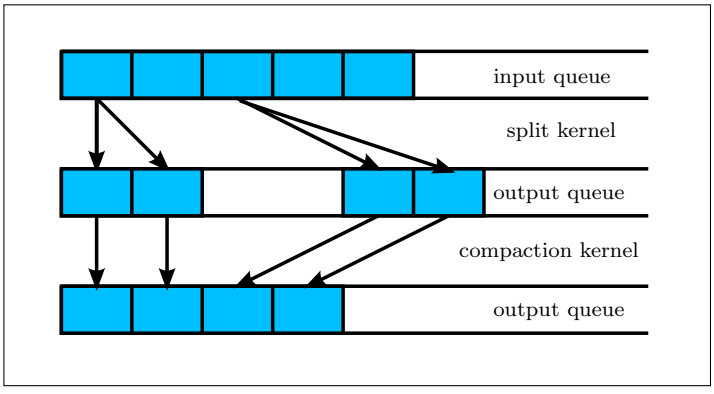


Figure 5.3: A scheme illustrating the queue system. The blue elements represent tasks. The split kernel processed tasks from the input queue and new tasks inserts to the output queue. The compaction kernel removes holes in the output queue.

Primitives have to be reordered into either the left or the right child node. An auxiliary array of size $2t$ is allocated in the shared memory, where t is the number of threads in the block. The first half corresponds to the left node and the second half corresponds to the right node. Each primitive is processed by a single thread. A thread reads the corresponding primitive and according to the position of the splitting plane decides whether it belongs to the left node or to the right node. If it belongs to the left node then the corresponding flag in the first half of the auxiliary array will be set to 1. If it belongs to the right node then the corresponding flag in the second half of the auxiliary array will be set to 1. After all flags are set a parallel exclusive prefix scan on the auxiliary array is performed. The output index consists of some global offset and the corresponding value of the prefix scan.

At the end of the iteration new tasks are generated and placed to the output queue. Indices of the tasks are $2i$ and $2i + 1$, where i is the index of the parent task. In the output queue holes may occur, which must be removed by an additional compaction pass, see figure 5.3. This process continues until the output queue is empty.

5.3 Hierarchical Linear BVH

In 2010 Pantaleoni and Luebke [23] improved the previous Morton code based construction algorithm. The authors observed that the Morton code defines a hierarchical structure similar to the octree. The first three bits of the code define the eight child voxels of the root voxel. The next three bits define the structure within the child voxels, and so on.

The sorting is divided into two levels. The top level sorting is done according to the first $3m$ bits. If m is relatively small, e.g. 5 or 6, two consecutive primitives will likely fall into the same voxel. Therefore, the Morton codes are compressed with the well-known run length encoding algorithm. The efficiency of the compression depends on the coherency between consecutive primitives. Efficiency of the compression is proportional to an acceleration of the top level sorting. After the top level sorting is done the codes are decompressed. Primitives are then sorted according to the remaining $3(k - m)$ bits.

The construction of the hierarchy is quite different from the previous case. In each iteration a treelet of depth p is constructed according to the examined p consecutive bits. The treelet can be encoded into an array of size $2^p - 1$. The index of the treelet root node is 0. The indices of the child nodes are $2i$ and $2i + 1$, where i is the index of the parent node. An example of the treelet encoding is depicted in figure 5.4.

The algorithm requires three auxiliary arrays. The number of primitives is denoted by n.

- *head_to_node*: An array of size n mapping a head of each leaf interval to the corresponding node index. If i is the head of an interval corresponding to the leaf with index j, then holds $head\_to\_node[i] = j$. Otherwise holds $head\_to\_node[i] = -1$.
- *primitive_to_interval*: An array of size n mapping a primitive index to the corresponding interval index.
- *interval_to_head*: An array mapping an interval index to a head of the corresponding interval.

At the beginning there is a single leaf interval and all primitives belong to this interval. The first entry of the array *head_to_node* is set to 0, other entries are set to -1. All entries of the array *primitive_to_interval* are set to 0. The first entry of the array *interval_to_head* is set to 0. In each iteration a binary vector of size n is created. If $head\_to\_node[i] \neq -1$, then the i-th component of the binary vector is set to 1, other components are set to 0. Interval indices are obtained performing a parallel inclusive prefix scan on the binary vector. Treelets are generated from the auxiliary arrays. The number of nodes is equal to twice the number of splits in each treelet. Thus, offsets for each treelet nodes are obtained performing a parallel exclusive scan on the numbers of nodes. At the end of each iteration all treelets are emitted and arrays *head_to_node* and *interval_to_head* are updated.

The advantage is that there are no singletons. Thus, the memory layout is better. The number of kernel launches is reduced due to the treelet processing. The input of sorting operation is dramatically reduced. The global split list in the previous case is larger than n. In this case the sorting operation is performed on a compressed array of size at most n.

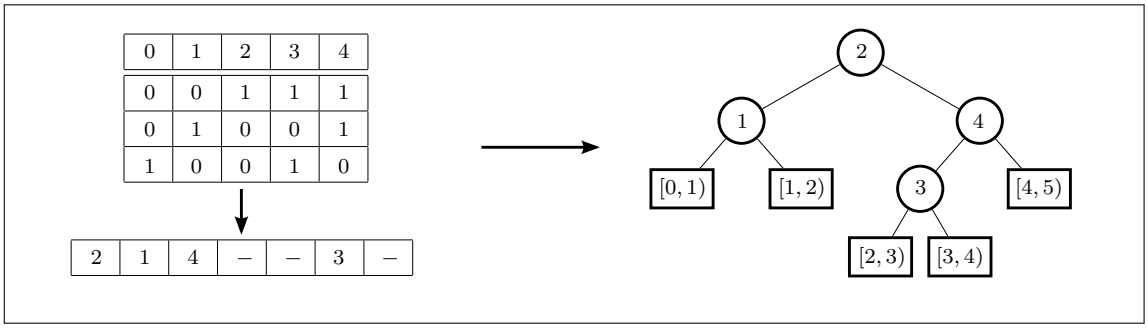


Figure 5.4: A scheme illustrating the treelet encoding such that $p = 3$. Indices of child nodes are $2i$ and $2i + 1$, here i is the index of the parent node.

5.4 Hierarchical Linear BVH with Work Queues

In 2011 Garanzha *et al.* [9] proposed two novel and simpler algorithms based on efficient task-queue system. The task-queue system is similar to the one proposed by Lauterbach *et al.* [19]. There are some significant differences. A task corresponds to a single split node and each task is processed by a single thread. In each iteration the tasks are fetched from the input queue. Each task will produce 0 or 2 new tasks. Performing warp-wide parallel prefix scan on these counts we obtain an offset for each task within the warp. Atomically adding the sum of new tasks to the global counter we obtain the global offset in the output queue. At the end of iteration the queues are swapped. This process continues until the output queue is empty. This process is summarized in figure 5.5.

Both algorithms are based on Morton curve. The two level sorting is replaced by efficient radix sort [20]. The first algorithm is simpler and the hierarchy is constructed by spatial median splits. The second algorithm constructs several levels of the hierarchy by SAH splits and the remaining levels by spatial median splits.

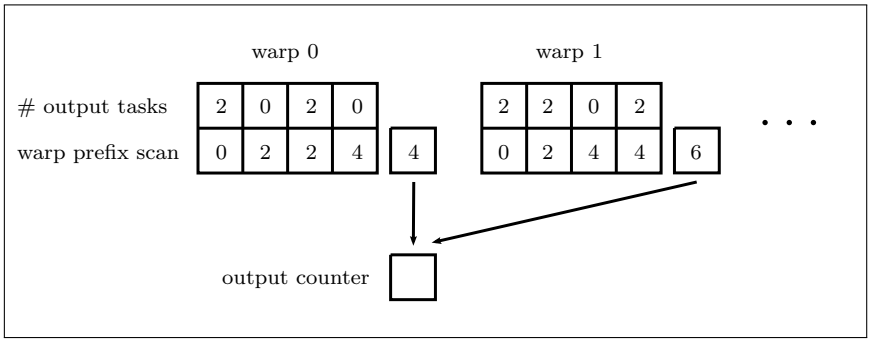


Figure 5.5: A scheme illustrating the queue system. Warps performs exclusive prefix scan on the number of output tasks. The first thread in the warp atomically adds the number of the output tasks to the global counter.

5.4.1 Spatial Median Splits

Each thread examines the current Morton code bit of the first and the last primitive in the corresponding node. The same bits are skipped. Using the binary search algorithm we find the last zero bit in the interval. Using the index of the last zero bit we split the corresponding to interval. The left part of the interval is assigned to the left child node. The right part of the interval is assigned to the right child node. The number of output tasks is computed performing parallel prefix scan. The created nodes are emitted and new tasks are put in the output queue. An additional refitting post-process is required. The hierarchy is built level by level. The post-process updates bounding boxes level by level from the leaves to the root.

5.4.2 SAH Splits

This algorithm is based on the surface area heuristic and the binning procedure. The first $m < k$ bits define coarse clusters. The primitives with the same m bits belong to the same cluster. Further we treat clusters as primitives. Each cluster has three attributes: a bounding box aggregating its primitives, an index of the corresponding task, and an index of the bin. Each task has five attributes: a bounding box of the corresponding node, an index the corresponding node, an index of the best splitting plane, an index of the first child task, and the set of bins. The iteration consists of three steps. At the beginning all clusters belong to the root task.

Binning: Parallel processing of clusters. A cluster is projected to the corresponding set of bins using atomic operations, i.e. minimum, maximum, add. For each axis the bin index is computed.

SAH Evaluation: Parallel processing of tasks. The result of binning is evaluated and the splitting plane with the lowest cost is chosen. The number of output task is computed and the global offset is determined.

Cluster Distribution: Parallel processing of clusters. In this step the clusters are distributed to new tasks. If the index of the splitting plane is less or equal to the bin index, then the cluster is assigned to the left child node. Otherwise the cluster is assigned to the right child node. The index of a task corresponding to the left child node is i and the index of a task corresponding to the right child node is $i + 1$, where i is the offset computed in the previous step.

These two algorithms seem to be simple and efficient. We decide to implement these two algorithms. The more details will be discussed in chapters 6 and 7.

5.5 Compact Prefix BVH

In 2012 Karras [16] introduced very efficient construction of binary trees including bounding volume hierarchies. The construction is based on spatial median splits and Morton codes. Assume that we have a sorted sequence of Morton codes corresponding the primitives. The algorithm splits nodes according to the current bit of Morton codes. The part of the interval containing zeros is assigned to the left child node. The part of the interval containing ones is assigned to the right child node.

The algorithm uses a very special data layout. Leaf nodes and interior nodes are stored in separate arrays. Leaf nodes are stored in array L. Interior nodes are stored in the array I. The root node is the first element of the array I. Children indices are γ and $\gamma + 1$, where γ is the index of the last zero bit in the interval. An important property of this layout is that one of the endpoints of the interval associated with the particular internal node is encoded in the index of the node. An example of the layout is depicted in figure 5.6.

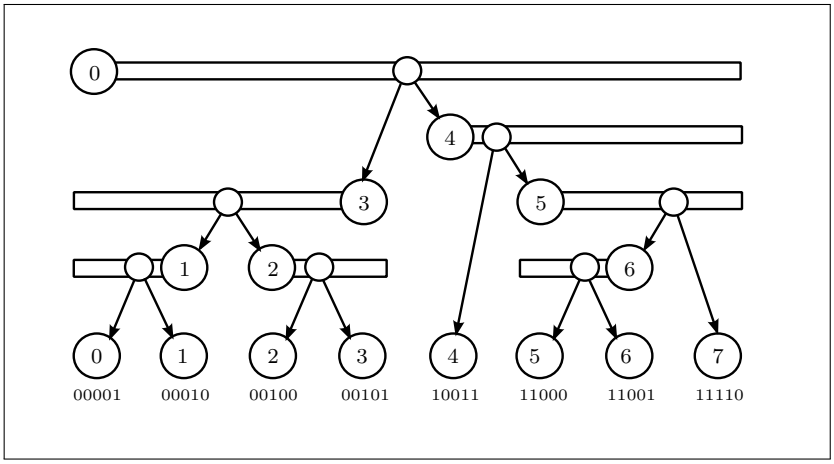


Figure 5.6: An example of the compact prefix BVH layout.

The algorithm efficiently determines the second endpoint by examining the neighborhood of the first endpoint. Searching for the second endpoint is based on the longest common prefixes of Morton codes. We define a function δ such that $\delta(i, j)$ returns the length of the longest common prefixes of the Morton codes in the interval $[i, j]$.

First the algorithm determines the direction d of searching by checking prefixes of the neighboring elements. Assume that the index of the first endpoint is i and the index of the second endpoint is j. If $\delta(i,i+1) > \delta(i,i-1)$ then the first endpoint is the first element of the interval and we will search for the second endpoint on the right side. If $\delta(i,i+1) < \delta(i,i-1)$ then the first endpoint is the last element of the interval and we will search for the second endpoint on the left side. Therefore, the direction $d = \text{sgn}(\delta(i,i+1) > \delta(i,i-1))$. The lower bound for $\delta(i,j)$ is given by $\delta_{min} = \delta(i,i-d)$. Using the lower bound δ_{min} and direction d we simply find the second endpoint by the binary search algorithm. In the same manner we find the split position γ. The lower bound is given by $\delta_{node} = \delta(i,j)$. Using the lower bound δ_{node} and the direction d we find the split position by the binary search algorithm. The pseudocode of this process is shown in algorithm 4.

The disadvantage of this algorithm is that the Morton codes must be unique. The author solved this problem using augmented Morton codes. An augmented Morton code is an original Morton codes concatenated with its index. The advantage is that the algorithm is not limited by the construction by levels and the parallelization of the construction is maximized.

5.6 BVH with Task Pool

In 2013 Vinkler *et al.* [26] designed a general framework based on a task pool with persistent warps. The idea is to launch a single kernel with as many warps as possible. Then all the work takes place on the GPU. The task pool is the core component of the framework. A task is a computational job associated with the given range of data. The task consists of phases and phase consists of steps. A phase is a logical algorithmic block of a task and a step is an algorithmic block of the phase. The task is associated with work chunks, which are the smallest unit of work. The work chunk consists of 32 data items and each thread in warp processes a single item. Warps fetch the work chunks of active tasks. Unlike queue based algorithm the advantage is that the task pool provides communication between warps, e.g. some tasks may be dependent on others. The communication is realized via atomic operations. At the beginning initial tasks are put in the task pool. Some task may spawn new tasks. Warps continue fetching the work chunks until there is no active tasks in the task pool.

Authors mapped the construction of bounding volume hierarchies to this framework. Both spatial median and SAH splits are possible. A task corresponds to a split node. At the beginning the root task is put to the task pool.

5.6.1 Spatial Median Splits

The splitting plane is trivially determined. In the reordering step each warp processes an interval of 32 primitives and computes the number of primitives to the left and to the right. Atomically adding the number of primitives on the left side to a global left counter we obtain a global offset for primitives belonging to the left side. Similarly atomically subtracting the number of primitives on the right side from the global right counter we obtain a global offset for primitives belonging to the right side. The global counters are initialized with first

Algorithm 4 Compact prefix BVH construction algorithm, courtesy of Karras [16].

Input: An index i of the split node.

$d \leftarrow \text{sgn}(\delta(i, i+1) - \delta(i, i-1))$
$\delta_{min} \leftarrow \delta(i, i-d)$
$l_{max} \leftarrow 2$
while $\delta(i, +i + l_{max}d) > \delta_{min}$ **do**
    $l_{max} \leftarrow 2l_{max}$
end while
$l \leftarrow 0$
for $k \leftarrow 1$ **to** $\log_2 l_{max}$ **do**
    $t \leftarrow \frac{l_{max}}{2^k}$
    if $\delta(i, i + d(l+t)) > \delta_{min}$ **then**
        $l \leftarrow l + t$
    end if
end for
$j \leftarrow i + dl$
$\delta_{node} \leftarrow \delta(i, j)$
$s \leftarrow 0$
for $k \leftarrow 1$ **to** $\log_2 l_{max}$ **do**
    $t \leftarrow \lceil \frac{l}{2^k} \rceil$
    if $\delta(i, i + d(s+t)) > \delta_{node}$ **then**
        $s \leftarrow s + t$
    end if
end for
$\gamma \leftarrow i + sd + \min(d, 0)$
if $\min(i, j) = \gamma$ **then**
    $left \leftarrow L[\gamma]$
else
    $left \leftarrow I[\gamma]$
end if
if $\max(i, j) = \gamma$ **then**
    $right \leftarrow L[\gamma + 1]$
else
    $right \leftarrow I[\gamma + 1]$
end if

and the last index of the interval. At the end left and right bounding boxes are computed performing parallel reduction on the left and the right intervals.

5.6.2 SAH Splits

In each axis 32 splitting planes are uniformly sampled. Each warp processes a distinct interval of primitives associated with the node. Each thread processes a single primitive and determines its position with respect to one of splitting planes. The number of primitives and bounding boxes on both sides are atomically updated. The last finished warp chooses the best splitting plane. The reordering step is done in the same way as in the previous case.

Chapter 6

Design and Implementation

In this chapter we will describe design and implementation details. In the first section we will describe the implementation of the construction of bounding volume hierarchies. In the second section we will describe our rendering system.

6.1 Construction of Bounding Volume Hierarchies

In this section we will describe details of implemented algorithms. We implemented both algorithms according to Garanzha *et al.* [9]. We are using the ray tracer from Timo Aila and Samuli Laine's framework [1]. The ray tracer support three types of data layout: an array of structures, a structure of arrays or a compact layout. The compact layout stores only interior nodes of the hierarchy. An interval associated with the leaf node is directly referenced from a parent interior node by the index of the first element. Intervals associated with leaf nodes are separated by special values, known as terminators. Thus, the end of the interval is detected via the terminator. The compact layout and structure of arrays layout are inconvenient for the parallel construction. Thus, we use the array of structures layout.

The construction algorithms are implemented in the `HLBVHBuilder` class. The size of each node is 64 bytes. Interior nodes contain bounding boxes of the child nodes and indices to the child nodes. Indices to leaf nodes are encoded using bitwise negation. Leaf nodes contain an interval of the triangle indices and bounding box of the triangles. A termination criterion is a maximum triangles in a leaf.

6.1.1 Spatial Median Splits

The first algorithm is simpler and it is based on Morton codes and spatial median splits. The parameters of this algorithm are maximum triangles in a leaf and the length of Morton codes in bits. In the following paragraphs we will describe particular steps.

Scene Bounding Box: First we compute the bounding box of the scene. Each thread processes several triangles of the input scene and accumulates the bounding box. We compute an unrolled parallel reduction within warps and within blocks using `__syncthreads` function. The first thread in the block atomically updates the global bounding box using `atomicMin` and `atomicMax` functions

Morton Codes: Each thread processes several triangles of the scene. For each triangle a Morton code is computed according to equation 5.2 using the scene bounding box from the previous step. We use `unsigned long long` data type for Morton codes. The length of Morton codes up to 60 bits is supported. Each thread also writes an appropriate index of the triangle to the index array.

Sorting: We use the Duane Merrill and Andrew Grimshaw's radix sort algorithm [20]. This algorithm is implemented in a `RadixSort` class and will be discussed further. We sort triangles using Morton codes as keys and triangle indices as values.

```
while(queue.getInSize() > 0) {

    // Split kernel.
    CudaKernel splitKernel = module->getKernel("split");

    // Set parameters.
    splitKernel.setParams(...);

    // Launch split kernel.
    splitKernel.launch(...);

    // Add new nodes.
    bvh.numberOfNodes += queue.getOutSize();

    // Resize node buffer if is necessary.
    if (size_t(bvh.nodes.getSize()) < size_t(bvh.numberOfNodes) * sizeof(Node))
        bvh.nodeBuffer.resize(2 * bvh.numberOfNodes * sizeof(Node));

    // Ping - pong queues.
    queue.swap();

    // Reset counter.
    queue.resetOutSize();

}
```

Listing 6.1: A sample code of the construction using spatial median splits.

Triangle Transformation: The ray tracer uses a unit ray-triangle intersection test [30]. The intersection is computed in the coordinate system of the triangle. An origin of the system is located in the first vertex of the triangle. The first basis vector is a vector pointing from the first vertex to the second vertex of the triangle. The second basis vector is a vector pointing from the first vertex to the third vertex of the triangle. The third basis vector is a normal vector of the triangle anchored in the first vertex. In this step the transformation matrices are computed. Each thread processes several triangles. For each triangle the transformation matrix to its coordinate system is computed. The matrix is written to the appropriate place using the indices from the previous step.

Splits: In this step we construct the hierarchy topology. We use two task queues: an input and an output queue. Both queues and operations associated with the queues are encapsulated in the `TaskQueue` class. A task is a four tuple including an index of the associated node, an index of the current bit, start and end indices of the associated triangle

interval. At the beginning input queue contains only a root task including a triangle interval of the whole scene. The index of the root node is 0 and the index of the current bit is an index of the most significant bit. The split kernel is being launched in a loop, see listing 6.1.

In the kernel each input task is processed by a single thread. The thread reads the task and checks the termination criteria and decides whether the node should be split. If so the thread splits the interval using the current bit of the task and Morton codes. If current bit of Morton codes associated with the task are all zeros or all ones then the next different bits are used. If all bits of Morton codes are consumed then the object median split is done. We find the splitting index using the binary search algorithm searching for the first index with the bit equal to 1. Threads within the warp write the number of output tasks to an auxiliary array allocated in the shared memory. Within the warp the unrolled Hillis-Steele prefix scan is performed on this array. The first thread in the warp atomically adds the sum of output tasks to the global counter using the `atomicAdd` function. This function returns the original value of the counter. Adding the original value to the prefix scan values each thread obtains an offset to the output queue for its output tasks, see figure 5.5. Node indices for new tasks are obtained by adding the offset to the number of nodes. After the kernel is finished the queues are swapped and this procedure is repeated until the output queue is empty. Additionally for each kernel launch we store the number of outputs tasks.

Refitting: In the previous step the hierarchy was constructed level by level in a top-down manner. The refitting procedure is similar. The bounding boxes of the hierarchy are refitted level by level from leaves to the root. We use the number of output tasks from the previous step as bounds of interval in node array corresponding to the currently processed level.

6.1.2 SAH Splits

The second algorithm is partially based on the first algorithm. The first levels of the hierarchy are constructed by SAH splits and the remaining levels are constructed by the spatial median splits. There is another parameter of the construction, the length of Morton code prefix in bits used for SAH splits, denoted by m. Up to the split procedure the process is the same as in the first algorithm.

Clusters: We use the first $m < k$ bits of Morton code to generate clusters. Triangles with the same first m bits belong to the same cluster. The cluster generation is similar to the splitting procedure using spatial median splits. We split the intervals until all m bits are consumed. Additionally for each triangle we store an index pointing to the associated cluster. At the beginning all triangles belong to the root cluster. After each iteration triangles are distributed to new clusters. Once all clusters are identified the clusters' bounding boxes are computed. Each thread processes several triangles. The thread computes bounding boxes of these triangles. The thread identifies the clusters associated with the triangles and atomically updates their bounding boxes. Morton codes of clusters are mapped back to quantized coordinates using equation 5.1. Therefore, the m must be multiple of 3. Further we treat the clusters as primitives. The cluster data are stored in a `Clusters` class.

Top Levels: This step is similar to the splitting procedure using spatial median splits. We work in two domains: cluster domain and task domain. Therefore, we need auxiliary arrays providing mapping between these two domains. Each cluster has a task index, bin index and node index. The task index is an index of the associated task. The bin index is the result of binning procedure. The node index is an index of the associated node. Each task has a bin set, an index of the plane and an index of the first child task. We use a `ClusterTaskQueue` to managed the cluster task data. At the beginning all cluster belongs to the root task. The splitting procedure is executed in a loop. The iteration of the loop consists of three kernels, see listing 6.2.

```
while (clusterQueue.getInSize() > 0) {

        // Reset bins.
        CudaKernel resetKernel = module->getKernel("resetBins");
        resetKernel.setParams(numberOfBins, clusterQueue.getBinsBuffer());
        resetKernel.launch();

        // Binning.
        CudaKernel binKernel = module->getKernel("binClusters");
        binKernel.setParams(...);
        binKernel.launch(...);

        // Splitting.
        CudaKernel splitKernel = module->getKernel("splitSAH");
        splitKernel.setParams(...);
        splitKernel.launch();

        // Distribute clusters.
        CudaKernel distributeKernel = module->getKernel("distributeClusters");
        distributeKernel.setParams(...);
        distributeKernel.launch();

        // Add new nodes.
        bvh.numberOfNodes += clusterQueue.getOutSize();

        // Ping - pong queues.
        clusterQueue.swap();

        // Reset counters.
        clusterQueue.resetOutSize();

}
```

Listing 6.2: A sample code of the construction using SAH splits.

In the first kernel each cluster is processed with a single thread. Each cluster can access a bin set of the associated task. Quantized coordinates of the cluster are used as a bin index. Bins are updated using atomic operations. Therefore, the number of bins for one axis must be at least $2^{\frac{m}{3}}$. This concept guarantee that all clusters of the associated task will never be projected into the same bin.

In the second kernel each task is processed with a single thread. Each thread evaluates costs of splitting planes according to equation 4.6 using the result of binning. For each axis two sequential prefix scans are performed on bins using the local memory. The splitting plane with the lowest cost is chosen. Using the same approach as previously we obtain indices of the child tasks. If the task contains only one cluster then we insert an appropriate splitting

task to the input queue used for lower levels of the hierarchy. For a single cluster we mark the index of the first child task with -1.

In the third kernel clusters are distributed to new tasks. Each cluster is processed by a single thread. The thread compares the bin index with the index of splitting plane and determines the new task index. From the previous step we know the index of the first child task. The index of the second child task is the index of the first child task incremented by one. If the index of the first child task is equal to -1 we know that the node was not split. For this case we mark the task index with -1 and the cluster is not processed further. We update the interval bounds of the corresponding task and the node index of the associated cluster. This process is repeated until the cluster input queue is empty.

Lower Levels: The remaining levels are constructed by spatial median splits. We use the input tasks from the previous step. After the topology is constructed we have to run the refitting procedure. Nodes corresponding to the clusters are not refitted because they don't resides in the continuous interval and must be processed further.

Cluster Refitting: During SAH evaluation bounding boxes of interior nodes are implicitly computed. Interior nodes contain bounding boxes of the left and the right children. We have to only refit interior nodes associated with clusters. We launch a single kernel. Each cluster is processed by a single thread. Using node indices we can simply map clusters to the corresponding nodes.

6.1.3 Radix Sort

We reimplemented the Duane Merrill and Andrew Grimshaw's radix sort algorithm [20]. The algorithm follows the the radix sort scheme discussed in chapter 2. In each iteration 4-bit or 5-bit digits of input keys are processed. The iteration of the sorting algorithm consists of three kernel launches.

The first kernel decodes and buckets the keys. Each thread processes several keys. Each thread decodes its keys and according to the current digit locally increments the corresponding flag in the corresponding flag array. Performing sequential reductions on these flag arrays we obtain the number of keys in each bucket. Performing parallel reductions within the thread block on partial reductions of threads we obtain the number of keys in each bucket within the block. These values are written to a global auxiliary array. The first part of the auxiliary array contains reduction values corresponding to the first digit. The second part of the auxiliary array contains reduction values corresponding to the second digit, and so on. The second kernel performs parallel prefix scan on the global auxiliary array using a single thread block. The third kernel again decodes and buckets the keys. Each thread again computes the flags according to the current digit. Reductions are replaced by prefix scan. Each thread performs local sequential prefix scan on its flag arrays. Threads within the block then perform parallel prefix scan on these partial local scans. Each thread determines the output indices of its keys and values using global prefix scan values and block prefix scan values.

6.2 Rendering System

In this section we will describe implementation details of the rendering system. The application is written in ANSI C++ and in CUDA. We parallelized almost all computations using the CUDA technology including the BVH construction, rendering, and an interpolation of key frames. The architecture of the application is object oriented. The kernels are encapsulated by classes. The kernels are scalable according the compute capability of the particular device.

The application works in the following way. At the beginning the application is configured from a text file. A scene is loaded from a file and a BVH is constructed. In this point the applications enters the main loop. Rays are generated and intersections are computed using the BVH. Pixels of the final image are shaded and the image is displayed via graphical user interface. In this point the main loop returns to the ray generation and the whole process is repeated. In the main loop the application handles the user inputs and changes appropriate parameters. The application also supports animated scenes. The animated scenes additionally require an interpolation of key frames and the BVH reconstruction in each frame.

6.2.1 Used Technologies

In this subsection we will briefly discuss the used technologies and libraries and the purpose of the choice.

CUDA: The CUDA technology was discussed in chapter 3. In our application we use the CUDA driver API because it provides more control over the program. Using the driver API we manage the CUDA modules explicitly. The host code and the device code is separated. We compile and load the CUDA modules in runtime. The management of the modules is similar to the management of shader programs.

Qt: Qt framework is a cross-platform application and UI framework. The framework is distributed into modules. In our application we use a core module, a GUI module and an OpenGL module. The core module provides many useful classes, e.g. container classes, timer classes, file manipulation classes, etc. The GUI module provides a graphical user interface elements, e.g widgets, dialogs, etc. The OpenGL module provides a basic OpenGL functionality and an OpenGL context management.

GLM: GLM is a header only mathematics library for graphics software based on OpenGL Shading Language. The library provides various classes of vectors and matrices with overloaded operators. The advantage is that the library is partially supported in the CUDA kernels. We work with the vectors and matrices across the whole application.

GLEW: GLEW is a cross-platform library. The GLEW library manages OpenGL extensions and provides the functionality of new versions of OpenGL. In our application we use OpenGL to display the result images.

Assimp: Assimp is a cross-platform library. The library provides 3D model import and export operations with the uniform interface. Various well-known formats are supported. The library provides many useful post-processing options. In our application we use only the import operation.

DevIL: DevIL is a cross-platform image library. The library has a very intuitive API similar to the OpenGL API. The library provides image import and image export operations in different formats. In our application we use both operations.

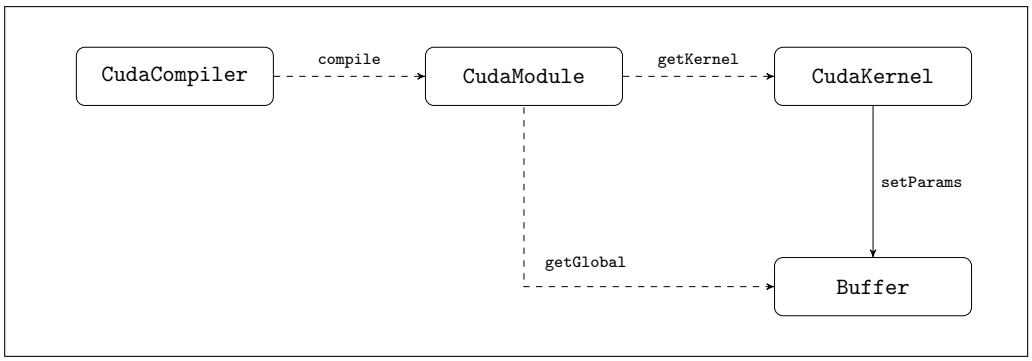


Figure 6.1: A diagram illustrating the CUDA management.

6.2.2 CUDA Management

We need to manage modules, kernels, and data. The management is based on Timo Aila and Samuli Laine's framework [1]. Unfortunately this framework depends on the Windows API. We reimplemented four classes from the framework replacing the Windows API with the Qt framework. Classes `CudaModule`, `CudaKernel` and `CudaCompiler` encapsulate CUDA functions. A diagram of the classes is depicted in figure 6.1.

A `CudaCompiler` class is based on the factory pattern. The `CudaCompiler` class creates instances of a `CudaModule` class. First a source `cu` file is compiled to a `cubin` file. The `cubin` file is a binary form of the CUDA module. Then the CUDA module is loaded from the `cubin` file using an appropriate CUDA function and the associated `CudaModule` instance is created. The compilation of the `cu` files is quite expensive. Therefore, the compiler uses a hashing scheme. The source `cu` files are hashed and the result hash value is used as a name of the `cubin` file. If the `cubin` file with the name exists then the module is loaded directly and the compilation is skipped.

The `CudaModule` class encapsulates the manipulation with the CUDA modules and the `CudaKernel` class encapsulates the manipulation with the kernel. Via a `CudaModule` instance we can retrieve `CudaKernel` instances associated with the module or we can access the global variables of the module. The management of the CUDA context is implemented in static members of the `CudaModule` class. Static methods also provide useful information about the device, e.g. free memory or the compute capability. Via a `CudaKernel` instance we can set the parameters of the kernel or we can launch the associated kernel.

The `Buffer` class encapsulates the manipulation with data. The class provides mapping data between various memory spaces. The supported memory spaces include the CPU memory space, the CUDA memory space and the OpenGL memory space. We can map data between arbitrary two memory spaces without explicitly copying. Typically a buffer instance is passed to a kernel as a parameter or global variables are retrieved from modules in the form of a buffer instance.

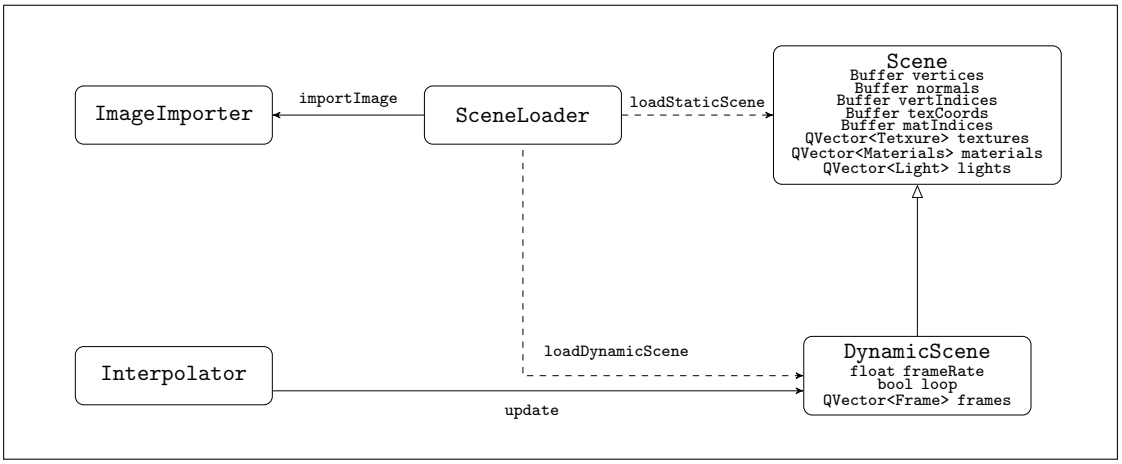


Figure 6.2: A diagram illustrating the scene management.

6.2.3 Scene Management

A `Scene` class is a container class for scene data including vertices, normal vectors, vertex indices, texture coordinates, textures, materials, material indices, and lights. The scene data except lights and materials are stored in `Buffer` instances. Lights and materials are mapped to constant memory. Texture data are encapsulated in a `Texture` structure. Only directional or point lights are supported. The `Scene` class has no public constructor and its instances can be only created via a `SceneLoader` friend class. The `SceneLoader` class encapsulates import functions of the Assimp library. The `SceneLoader` class is based on a factory pattern. The scene data are loaded from a given file. Texture data are loaded via a helper `ImageImporter` class which encapsulates import functions of the DevIL library.

A `DynamicScene` class is a derived class from the `Scene` class. A `DynamicScene` class is a container class for animated scenes with key frames. Each key frame contains vertices and normal vectors, the rest scene data are shared. Vertices and normal vectors of key frames are encapsulated in a `Frame` structure. The `Scene` class is extended with a vector of `Frame` instances. The `Interpolator` class linearly interpolates vertices and normal vectors of two key frames according to a given time parameter and a frame rate. The result of the interpolation is stored in buffers of the ancestor class. The interpolation is implemented on GPU using the CUDA technology. Instances of `DynamicScene` can be created via the `SceneLoader` friend class from a given list of filenames. A diagram illustrating the scene management is depicted in figure 6.2.

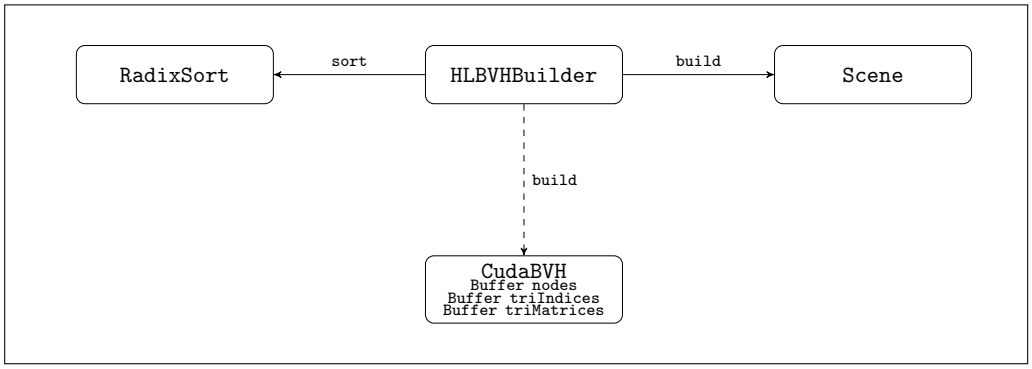


Figure 6.3: A diagram illustrating the BVH management.

6.2.4 BVH Management

The ray tracer requires the BVH in form of a `CudaBVH` class. The ray tracer uses a unit ray-triangle intersection test [30]. Thus, the `CudaBVH` class contains hierarchy nodes, triangle indices and triangle transformation matrices. The `CudaBVH` class has no public constructor and its instances can be created only via a `HLBVHBuilder` class. The BVH management is depicted in figure 6.3.

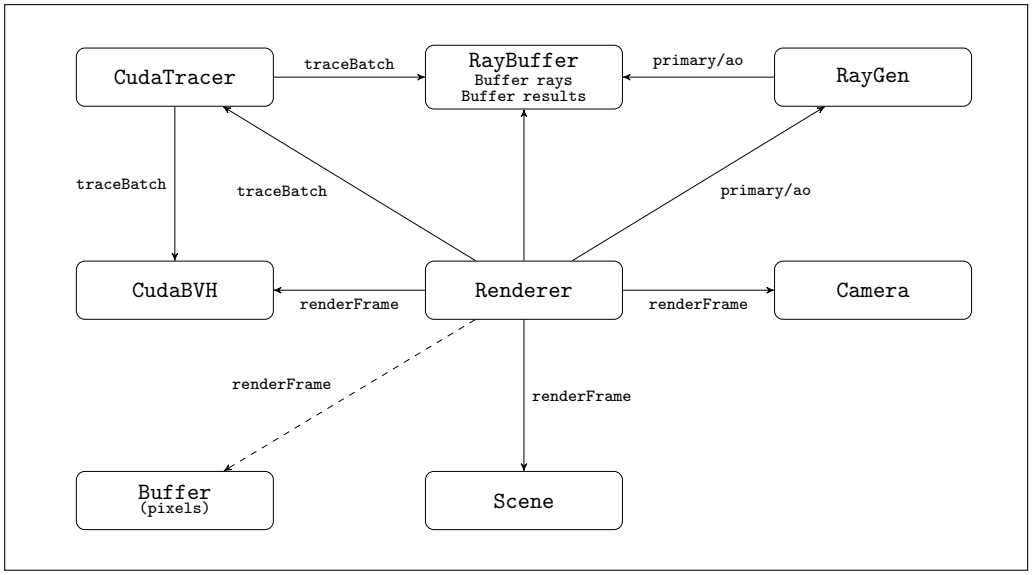


Figure 6.4: A diagram illustrating the rendering pipline.

6.2.5 Rendering Pipeline

A `Renderer` class and its method `renderFrame` is an entry point of the rendering pipeline. Input parameters of the `renderFrame` method are a `Camera` instance, a `Scene` instance and

a `CudaBVH` instance. An output parameter is a final image stored in a `Buffer` instance. The renderer supports three types of rays: primary rays, ambient occlusion rays, and diffuse rays. Images computed with different ray types are depicted in figure 6.5. First renderer generates the appropriate type of rays using a `RayGen` class. Ray data including ray origin and ray direction vector are encapsulated by a `Ray` structure. The `Ray` structure also contains bounds of the rays, i.e. minimum and maximum t. Using a `CudaTracer` class we compute the intersections. The result of ray tracing for each ray is stored in a `RayResult` structure. The `RayResult` structure contains the index of the intersected triangle, a parameter t, and barycentric coordinates of the hit. Both ray results and rays are encapsulated by a `RayBuffer` class. Pixels of the final image are shaded according to a given ray type. The image is displayed via a graphical user interface or exported to a file using an `ImageExporter` class. The `ImageExporter` class encapsulates export functions of the DevIL library.

Primary Rays: There are two shading modes using primary rays. The first mode uses flat shading without materials. The second mode uses smooth shading with materials and textures. The second mode interpolates normal vectors and texture coordinates using barycentric coordinates of the intersection obtained from the ray tracer. The final value of the texel is obtained using bilinear interpolation of neighbor values. The color of a pixel is computed according to Phong illumination model.

Ambient Occlusion Rays: Ambient occlusion rays are generated after the primary rays. Points of intersection of primary rays are used as origins of ambient occlusion rays. For each primary we generate several ambient occlusion rays. Direction vectors of ambient occlusion rays are randomly sampled. An ambient occlusion radius limits the length of the rays. Only triangles within the radius are considered during intersection computation. The final color of a pixel is computed as the sum of ambient occlusion rays that hit nothing per primary ray divided by the number of ambient occlusion rays per primary ray.

Diffuse Rays: Diffuse rays are similar to the ambient occlusion rays, but there are some differences. First the diffuse rays are not limited by a radius. Second a pixel value is computed in different way. If a diffuse ray hits some triangle then the value is computed as the diffuse component of Phong illumination model. Otherwise the value is equal to the white color. The final color of a pixel is sum of these values divided by the number of diffuse rays per primary ray and multiplied by a diffuse color of the triangle hit by the associated primary ray.

6.2.6 User Interface

In this subsection we will briefly describe user interfaces of the application. The application can be configured via a text file. The file may contain named blocks with keys and values. We use an `Environment`[1] class to parse the configuration file. Configurable parameters include renderer parameters, camera parameters, etc. All configurable parameters are listed in appendix C.

[1] An author of the `Environment` class is Tomáš Kopal.

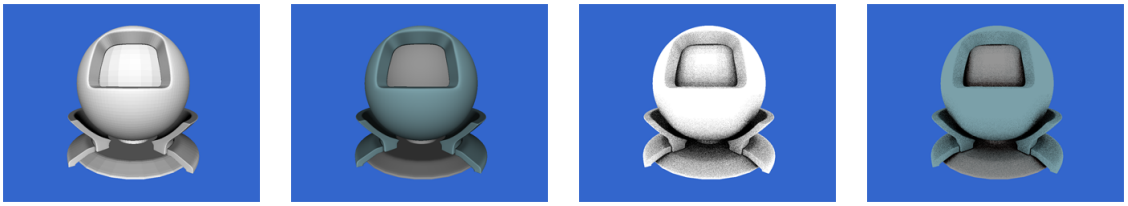


Figure 6.5: Mitsuba model traced with different ray types: primary rays with flat shading, primary rays with smooth shading and materials, ambient occlusion rays, and diffuse rays.

A graphical user interface is important component of the application. The main purpose of the GUI is that we need to display final images. The GUI is based on Qt framework. The main window is implemented in a `MainWindow` class. To display images we use a `RenderWidget` class derived from the `QGLWidget` class. The `RenderWidget` is a central widget of the main window. The `RenderWidget` handles mouse and keyboard events. Using mouse and keyboard users can rotate or translate the camera. There is a menu bar in the main window. Using the menu bar users can interactively change a ray type or a BVH construction algorithm. Users can open a settings window and configure the application in runtime.

Chapter 7

Results and Discussion

In this chapter we will present our results. We tested two types of scenes: static and dynamic scenes. We use all three ray types: primary rays, ambient occlusion rays, and diffuse rays. Sample images are depicted in figure 7.12. The image resolution of all tested scenes is 1024×768 pixels. We use one directional light source. We use one primary ray per pixel. We use four secondary rays, ambient occlusion or diffuse rays per primary ray. The radius for ambient occlusion rays was set to 1.0. We use 8, 16 and 32 maximum triangles in leaves as a termination criterion.

We used two types of timers. We used very precise CUDA events to measure kernel times. We use less precise QT timers to measure a whole process including CPU time. The CUDA events have microseconds precision. The QT timers have only milliseconds precision. For testing we used a PC with the following equipment.

- CPU: Intel Xeon E5-1620, 3.6 GHz, 10 MB
- GPU: GeForce GTX TITAN, 6144 MB RAM, Compute Capability 3.5
- Memory: 24 GB RAM
- OS: Windows 7 64 bit
- Compiler: The Visual C++ Compiler 16.00, Nvidia NVCC 5.0

7.1 Static Scenes

Static means that the geometry of a scene does not change in time. Therefore, the hierarchy is built only once. We tested fourteen static scenes of various triangle distributions. Images of static scenes are depicted in figure 7.1. We tested both implemented algorithms. For testing we use three configuration of the algorithms: spatial median splits with 30-bit Morton codes, spatial median splits with 60-bit Morton codes and SAH splits with 30-bit Morton codes 15-bit clusters.

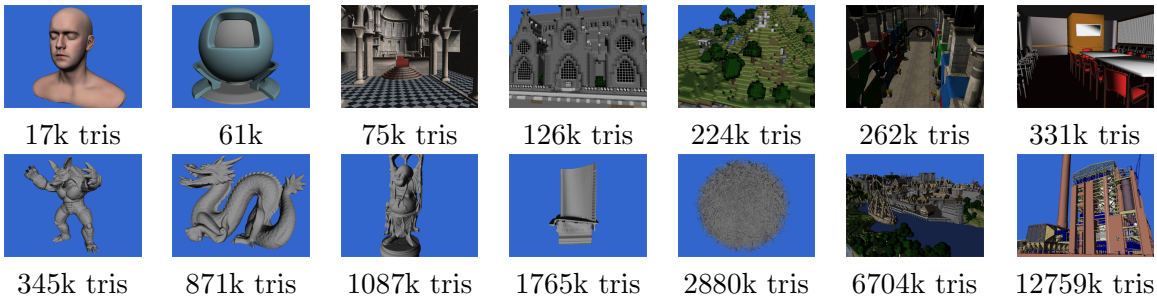


Figure 7.1: Images of the static scenes: Head, Mitsuba, Sibenik Cathedral, Rungholt House, Lost Empire, Crytek Sponza, Conference, Armadillo, Stanford Dragon, Happy Buddha, Turbine Blade, Hairball, Rungholt, Power Plant.

7.1.1 Spatial Median Split with 30-bit Morton Codes

The results for this configuration are shown in tables 7.1 and 7.2. The construction performance in MTris/s is depicted in figure 7.4. Construction time scales rather logarithmically than linearly with depth of the hierarchy. It is due to the level by level construction. Therefore, it pays off to construct deeper hierarchies with lower costs and higher ray tracing performance. Spatial coherent scenes such as Armadillo or Happy Buddha have lower cost than complex scenes such as Crytek Sponza. The size in MB is not direct proportional to the number of nodes. As mentioned the hierarchy structure contains also triangle indices and triangle transformation matrices. Therefore, the size of the hierarchy is dependent rather on the number of triangles than the number of nodes. Auxiliary memory used for construction is relatively small.

7.1.2 Spatial Median Split with 60-bit Morton Codes

The results for this configuration are shown in tables 7.3 and 7.4. The construction performance in MTris/s is depicted in figure 7.5. The results are similar to the results of the previous case. There are two possible explanations. The first explanation is that the termination criteria are applied before the first 30 bits are consumed. This is reasonable especially for small scenes. When all bits of Morton codes are consumed then the algorithm uses object median splits. Thus, the second explanation is that the object median splits are very close to the spatial median splits. Only the construction time differs from the previous case especially for larger scenes. The additional time is consumed in the sorting phase. The radix sort algorithm requires twice number of iterations for double-length keys.

7.1.3 SAH Splits with 30-bit Morton Codes and 15-bit Clusters

The results for this configuration are shown in tables 7.5 and 7.6. The construction performance in MTris/s is depicted in figure 7.6. The algorithm is more complex. Therefore, the construction times and auxiliary memory consumption are higher. Times of particular kernels are shown in table 7.7. The most time consuming are splitting procedure using SAH

splits and cluster computations for small scenes. The most time consuming are sorting, cluster computation and refitting procedure for large scenes.

The algorithm splits nodes until the only one cluster remains regardless of the termination criteria. The termination criteria are applied after all clusters are split. Therefore, hierarchies of small scenes are deeper than should be. This fact corresponds to the number of nodes and construction times. As expected hierarchies constructed by SAH splits have lower costs and higher ray tracing performance. The quality improvement is significant especially for spatial incoherent scenes such as Crytek Sponza or Sibenik.

29 key frames
5k tris

246 key frames
11k tris

30 key frames
78k tris

21 key frames
176k tris

16 key frames
252k tris

Figure 7.2: Images of the dynamic scenes: Wood Doll, Toasters, Ben, Fairy Forest, Exploding Dragon.

7.2 Dynamic Scenes

Dynamic means that the geometry of the scene changes in time. Thus, the hierarchy must be updated in every frame. We tested five dynamic scenes of various complexity. Images of dynamic scenes are depicted in figure 7.2. For testing we use two configurations of the algorithm: spatial median splits with 30-bit Morton codes and SAH splits with 30-bit Morton codes and 15-bit clusters. We use 8 maximum triangles in leaves as a termination criterion.

The results for both configurations are shown in tables 7.8 and 7.9. We averaged all values across all key frames of each scene. FPS values include both construction time and render time. We compared costs of all key frames for both configurations for each dynamic scene. The results of comparison are depicted in figures 7.7, 7.8, 7.9, 7.10, and 7.11. We measured cost of simple refitting. Thus, the hierarchy is built from the first key frame using the SAH configuration. In each other key frames bounding boxes are only refitted from leaves to the root. Hierarchies constructed via SAH splits have the lowest costs in all cases. Interestingly, for scenes Wood Doll, Toasters and Fairy Forest the refitted hierarchies have lower costs than the reconstructed hierarchies by spatial median splits.

The special case is the Exploding Dragon scene. The geometry of this scene changes completely in time. The Bunny falls on the Dragon and both models explode, see figure 7.3. In this case simple refitting absolutely fails.

7.3 Discussion

The results imply some interesting observations. The construction performance in MTris/s is higher for large scenes. This is the property of massively parallel processing. Large scenes

Figure 7.3: Sample key frames of the Exploding Dragon scene. The geometry of the scene changes completely.

keeps the GPU more occupied. Similarly, it pays off to construct deeper hierarchies, because the construction time scales logarithmically with the depth. This is again a property of massively parallel processing. Reconstruction of the hierarchy is reasonable because simple refitting may be unstable for some distributions. The efficiency of the refitting depends on how much the geometry changes.

	Construction [ms]						Cost			#Nodes		
	Kernels			Overall								
Scene	8	16	32	8	16	32	8	16	32	8	16	32
Head	1.8	1.9	2.1	10.0	8.0	8.0	108	131	185	6969	3673	1941
Mitsuba	2.1	2.2	2.1	9.0	9.0	8.0	288	361	501	23655	12447	6299
Sibenik	2.2	2.3	2.6	10.0	12.0	10.0	247	300	413	30405	16321	8623
R. House	2.2	2.2	2.4	12.0	11.0	13.0	322	391	541	45445	23491	11947
Lost Empire	3.0	2.9	3.1	16.0	14.0	16.0	300	354	474	83429	42975	22121
Cr. Sponza	4.1	4.1	4.0	18.0	17.0	17.0	484	550	700	100249	52933	27713
Conference	4.1	4.0	5.1	18.0	20.0	18.0	245	290	407	118851	63165	33177
Armadillo	4.4	4.1	4.2	16.0	17.0	18.0	115	130	168	137939	67147	36093
St. Dragon	9.2	9.8	8.3	30.0	30.0	22.0	191	219	282	354299	183761	94193
H. Buddha	12.5	10.4	10.0	36.0	32.0	31.0	234	270	351	439141	226479	115951
Blade	18.2	17.2	14.9	48.0	45.0	41.0	267	308	404	668327	345247	172265
Hairball	26.4	24.8	27.4	71.0	65.0	73.0	1700	2011	2777	1151495	585639	288369
Rungholt	58.0	54.7	53.2	125.0	115.0	109.0	596	705	953	2491683	1311125	662163
Power Plant	100.4	95.0	93.8	215.0	197.0	190.0	215	244	322	4403547	2288377	1182525

Table 7.1: Results for the BVH construction using spatial median splits with 30-bit Morton codes. We use 8, 16 or 32 maximum triangles in leaves as a termination criterion.

	Performance [MRays/s]									Memory [MB]			
	Primary			Ambient Occlusion			Diffuse						
Scene	8	16	32	8	16	32	8	16	32	8	16	32	Aux.
Head	119	102	81	77	68	56	68	60	50	1.6	1.4	1.3	0.9
Mitsuba	91	78	71	70	61	51	60	52	43	5.6	4.9	4.5	3.2
Sibenik	59	50	31	112	96	73	36	29	20	7.0	6.1	5.6	3.9
R. House	89	76	59	129	113	88	100	87	69	11.4	10.0	9.3	6.5
Lost Empire	77	64	49	91	75	56	72	60	46	20.6	18.0	16.7	11.7
Cr. Sponza	19	15	11	37	29	21	17	14	10	24.2	21.2	19.5	13.6
Conference	41	34	22	72	62	45	35	28	20	30.2	26.5	24.6	17.2
Armadillo	97	86	69	59	55	46	47	43	36	32.3	27.9	25.8	18.0
St. Dragon	62	48	41	57	50	39	50	44	34	81.9	71.0	65.2	45.3
H. Buddha	69	65	50	50	45	36	43	39	32	102.0	88.4	81.3	56.5
Blade	113	98	88	61	56	50	50	45	39	162.8	142.1	131.0	91.8
Hairball	13	11	8	17	15	10	13	11	7	269.5	233.3	214.2	149.7
Rungholt	51	42	32	78	65	48	52	44	32	615.3	539.8	498.2	348.6
Power Plant	10	9	7	22	20	15	10	9	7	1149.4	1014.0	943.3	663.5

Table 7.2: Results for the BVH construction using spatial median splits with 30-bit Morton codes. We use 8, 16 or 32 maximum triangles in leaves as a termination criterion.

	Construction [ms]						Cost			#Nodes		
	Kernels			Overall								
Scene	8	16	32	8	16	32	8	16	32	8	16	32
Head	2.2	2.2	2.5	10.0	9.0	10.0	108	131	185	6975	3673	1955
Mitsuba	2.7	2.5	2.7	11.0	12.0	11.0	287	348	471	23507	12337	6239
Sibenik	2.9	3.3	3.9	12.0	12.0	14.0	247	299	409	30311	16257	8639
R. House	3.1	3.1	3.3	14.0	13.0	15.0	322	391	540	45533	23391	11921
Lost Empire	4.3	4.2	4.4	17.0	16.0	16.0	300	355	476	83441	42979	22097
Cr. Sponza	5.1	5.1	5.5	20.0	19.0	19.0	485	547	686	100323	52857	27639
Conference	7.1	6.3	7.7	27.0	22.0	23.0	245	290	408	131377	70813	38207
Armadillo	6.2	6.8	60.0	21.0	21.0	19.0	115	130	167	137655	68999	36815
St. Dragon	13.3	12.7	12.4	35.0	33.0	32.0	191	218	281	354173	183755	94363
H. Buddha	16.1	15.3	15.0	35.0	33.0	37.0	234	269	350	438805	226831	116037
Blade	27.2	26.1	22.7	58.0	55.0	47.0	266	307	404	667617	345791	172511
Hairball	39.3	37.6	38.5	86.0	81.0	81.0	1701	2010	2776	1154193	587259	288715
Rungholt	86.9	83.7	82.2	157.0	147.0	161.0	598	706	956	2489835	1312147	661669
Power Plant	161.9	155.2	152.4	278.0	260.0	252.0	214	243	324	4840783	2501507	1301027

Table 7.3: Results for the BVH construction using spatial median splits with 60-bit Morton codes. We use 8, 16 or 32 maximum triangles in leaves as a termination criterion.

	Performance [MRays/s]									Memory [MB]			
	Primary			Ambient Occlusion			Diffuse						
Scene	8	16	32	8	16	32	8	16	32	8	16	32	Aux.
Head	117	100	80	77	65	58	67	61	51	1.6	1.4	1.3	0.9
Mitsuba	94	88	64	73	64	52	60	53	43	5.6	4.9	4.5	3.2
Sibenik	59	44	36	112	96	73	36	29	20	7.0	6.1	5.6	3.9
R. House	90	76	58	129	112	86	100	87	68	11.5	10.0	9.3	6.5
Lost Empire	77	64	49	91	75	55	72	60	46	20.6	18.0	16.7	11.7
Cr. Sponza	19	16	11	36	29	21	17	14	10	24.2	21.2	19.5	13.6
Conference	40	34	22	72	61	45	35	29	20	30.9	27.0	24.9	17.2
Armadillo	100	79	69	60	54	46	47	43	36	32.3	27.9	25.8	18.0
St. Dragon	62	53	42	58	50	39	50	43	34	81.9	71.0	65.2	45.3
H. Buddha	79	66	51	51	45	37	44	39	32	102.0	88.4	81.3	56.5
Blade	116	100	88	62	56	50	50	45	38	162.7	142.1	131.0	91.8
Hairball	13	11	8	17	15	10	13	11	7	269.7	233.4	214.3	149.7
Rungholt	52	42	31	77	64	47	52	43	32	615.2	539.8	498.2	348.6
Power Plant	9	9	7	22	20	15	10	9	7	1177.4	1027.7	950.8	663.5

Table 7.4: Results for the BVH construction using spatial median splits with 60-bit Morton codes. We use 8, 16 or 32 maximum triangles in leaves as a termination criterion.

	Construction [ms]						Cost			#Nodes		
	Kernels			Overall								
Scene	8	16	32	8	16	32	8	16	32	8	16	32
Head	13.4	13.3	13.5	26.0	26.0	26.0	85	87	91	9965	7529	6355
Mitsuba	13.5	13.4	13.4	30.0	26.0	25.0	263	280	293	26783	19977	17611
Sibenik	16.8	14.7	14.8	31.0	29.0	30.0	188	203	225	33251	20749	14553
R. House	15.9	13.7	15.6	34.0	28.0	33.0	305	353	374	45805	27449	24041
Lost Empire	15.0	14.8	14.6	36.0	31.0	31.0	267	328	421	83619	44219	25675
Cr. Sponza	16.5	18.3	16.6	41.0	38.0	38.0	269	290	301	101601	54615	30905
Conference	16.6	16.2	14.7	43.0	36.0	37.0	146	160	183	120851	64209	34559
Armadillo	13.3	14.7	14.5	32.0	32.0	33.0	104	119	156	137941	69159	36949
St. Dragon	19.7	19.1	18.7	46.0	50.0	38.0	179	207	269	354309	183783	94245
H. Buddha	23.2	22.5	22.1	52.0	52.0	44.0	215	250	328	438487	226313	116343
Blade	33.3	31.9	28.2	62.0	66.0	60.0	315	360	387	663245	341519	170615
Hairball	46.6	43.9	43.1	91.0	90.0	79.0	1693	2004	2764	1151503	585743	288827
Rungholt	92.1	88.6	87.0	167.0	156.0	150.0	536	645	893	2491699	1311171	662321
Power Plant	152.8	147.0	145.6	274.0	258.0	249.0	197	225	301	4403557	2288421	1182613

Table 7.5: Results of BVH construction using SAH splits with 30-bit Morton codes and 15-bit clusters. We use 8, 16 or 32 maximum triangles in leaves as a termination criterion.

	Performance [MRays/s]									Memory [MB]			
	Primary			Ambient Occlusion			Diffuse						
Scene	8	16	32	8	16	32	8	16	32	8	16	32	Aux.
Head	126	119	108	81	77	72	70	67	63	1.8	1.6	1.6	9.6
Mitsuba	106	102	91	74	70	62	29	58	51	5.8	5.4	5.2	29.9
Sibenik	65	64	55	128	117	102	45	38	30	7.2	6.4	6.0	20.2
R. House	85	86	76	36	122	115	28	94	89	11.5	10.3	10.1	43.0
Lost Empire	78	63	50	95	78	60	74	62	49	20.6	18.1	16.9	42.5
Cr. Sponza	32	26	22	46	38	29	31	25	19	24.3	21.3	19.7	26.3
Conference	54	49	44	98	87	70	50	44	34	30.2	26.6	24.7	23.2
Armadillo	127	80	63	42	38	47	33	44	37	32.3	27.9	25.8	27.9
St. Dragon	64	55	42	58	51	39	51	44	35	81.9	71.0	65.2	62.1
H. Buddha	81	68	51	52	46	37	45	40	32	102.0	88.4	81.3	77.5
Blade	113	99	86	62	58	50	50	46	39	162.4	141.9	130.9	115.4
Hairball	13	11	8	18	15	10	13	11	8	269.5	233.3	214.3	200.6
Rungholt	53	43	32	81	67	49	55	45	33	615.3	539.8	498.2	388.5
Power Plant	15	13	10	26	23	16	14	12	9	1149.4	1014.0	943.3	666.9

Table 7.6: Results of BVH construction using SAH splits with 30-bit Morton codes and 15-bit clusters. We use 8, 16 or 32 maximum triangles in leaves as a termination criterion.

Scene	Sort	Clusters	SAH splits	Median splits	Refit	Other	Kernels	Overall
Head	0.3	4.5	7.5	0.2	0.4	0.5	13.4	26.0
Mitsuba	0.5	3.8	8.3	0.1	0.2	0.6	13.5	30.0
Sibenik	0.7	5.2	9.7	0.2	0.5	0.5	16.8	31.0
R.House	0.9	3.3	10.5	0.1	0.3	0.8	15.9	34.0
Lost Empire	1.0	3.4	9.0	0.2	0.5	0.9	15.0	36.0
Cr. Sponza	1.4	4.1	9.0	0.4	0.8	0.8	16.5	45.0
Conference	1.8	4.5	7.9	0.5	1.0	0.9	16.6	47.0
Armadillo	1.7	2.8	6.3	0.4	1.1	1.0	13.3	33.0
St. Dragon	3.9	3.6	7.4	0.9	2.4	1.5	19.7	67.0
H. Buddha	4.8	4.8	7.7	1.1	2.9	1.9	23.2	80.0
Blade	8.7	6.1	9.5	1.8	4.8	2.6	33.5	69.0
Hairball	13.2	9.6	10.2	2.8	7.0	3.8	46.6	91.0
Rungholt	28.5	24.9	9.7	5.4	14.1	9.5	92.1	167.0
Power Plant	53.0	47.9	5.2	7.7	26.0	13.1	152.9	274.0

Table 7.7: Times of particular construction phases in ms. SAH splits with 30-bit Morton codes and 15-bit clusters. We use 8 maximum triangles in leaves a as termination criterion.

Scene	Construction [ms]	Cost	FPS		
			Primary	Am. Occl.	Diffuse
Wood Doll	6.2	74	95	27	22
Toasters	6.8	205	61	21	14
Ben	12.2	93	53	17	14
Fairy Forest	14.1	129	37	11	6
Exploding Dragon	15.2	68	43	14	11

Table 7.8: Results for dynamic scenes, spatial median splits with 30-bit Morton codes. FPS includes both construction time and render time.

Scene	Construction [ms]	Cost	FPS		
			Primary	Am. Occl.	Diffuse
Wood Doll	21.5	51	39	11	9
Toasters	24.4	122	31	10	8
Ben	29.8	83	27	9	7
Fairy Forest	30.5	105	24	8	5
Exploding Dragon	31.2	59	26	9	7

Table 7.9: Results for dynamic scenes, SAH splits with 30-bit Morton codes and 15-bit clusters. FPS includes both construction time and render time.

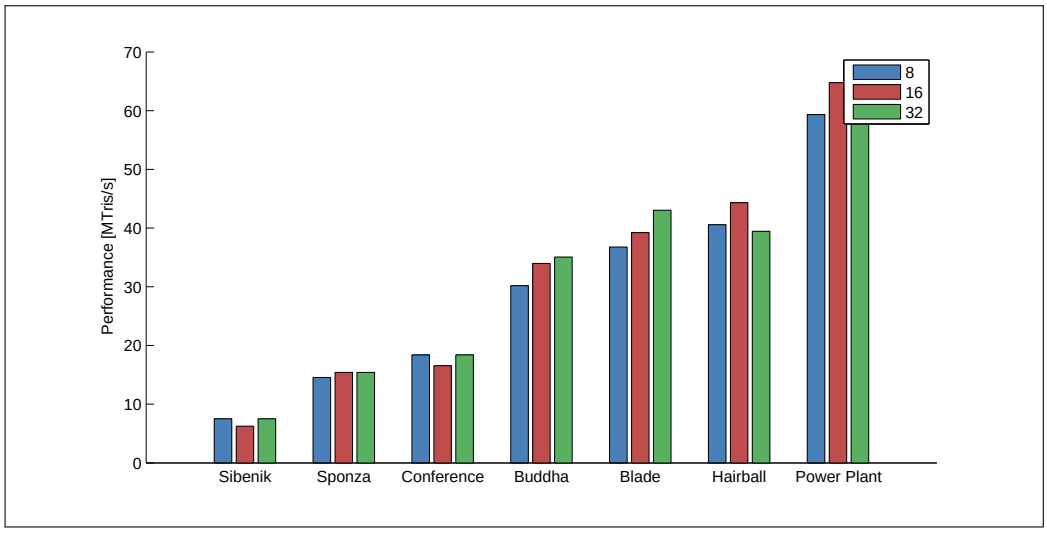


Figure 7.4: Construction performance in MTris/s using spatial median splits and 30-bit Morton codes for different termination criteria.

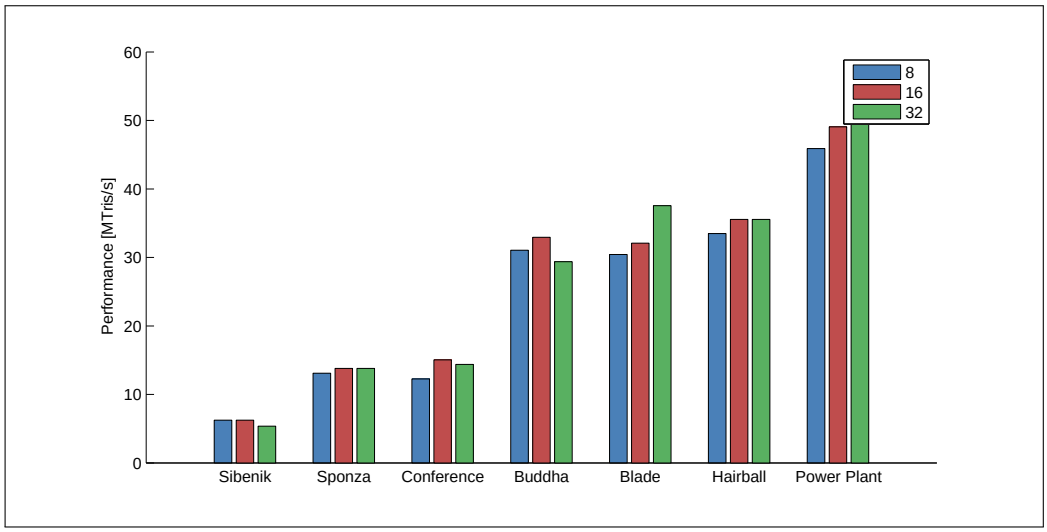


Figure 7.5: Construction performance in MTris/s using spatial median splits and 60-bit Morton codes for different termination criteria.

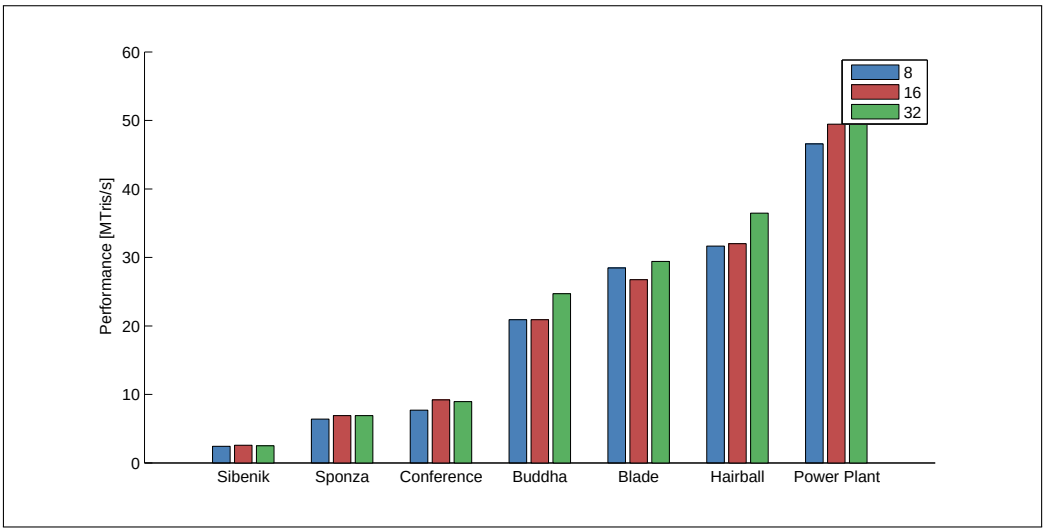


Figure 7.6: Construction performance in MTris/s using spatial SAH splits and 30-bit Morton codes and 15-bit clusters for different termination criteria.

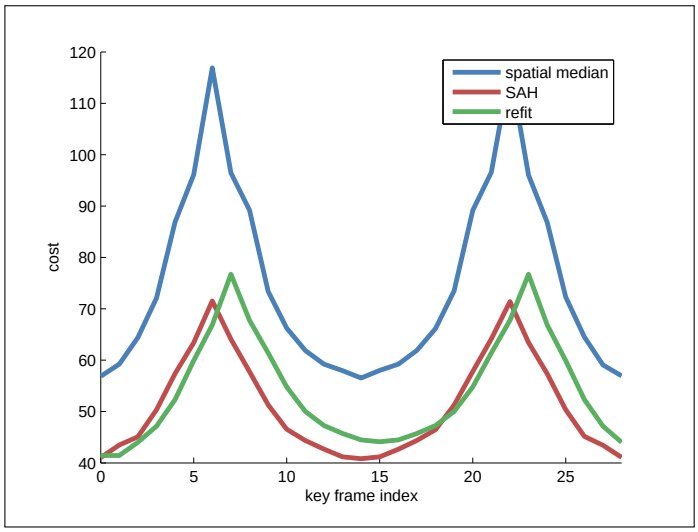


Figure 7.7: Costs of the Wood Doll scene for both construction algorithms and refitting procedure. We use 8 maximum triangles in leaves as a termination criterion.

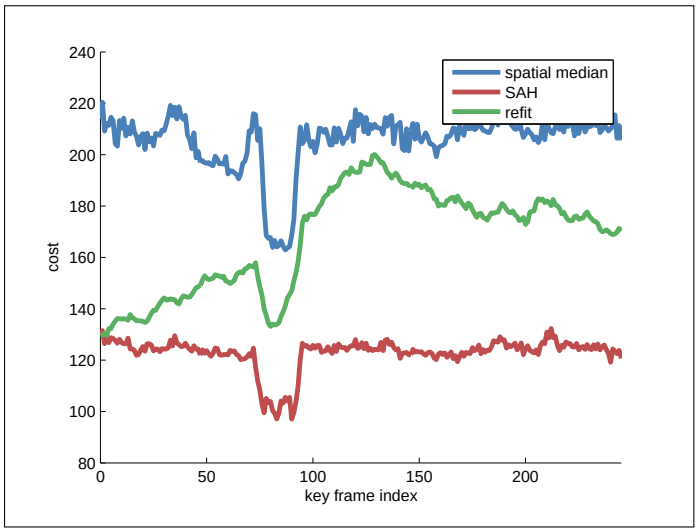


Figure 7.8: Costs of the Toasters scene for both construction algorithms and refitting procedure. We use 8 maximum triangles in leaves as a termination criterion.

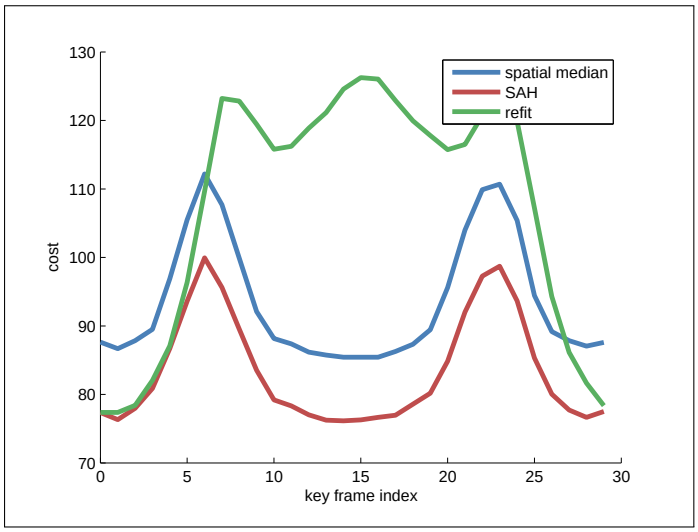


Figure 7.9: Costs of the Ben scene for both construction algorithms and refitting procedure. We use 8 maximum triangles in leaves as a termination criterion.

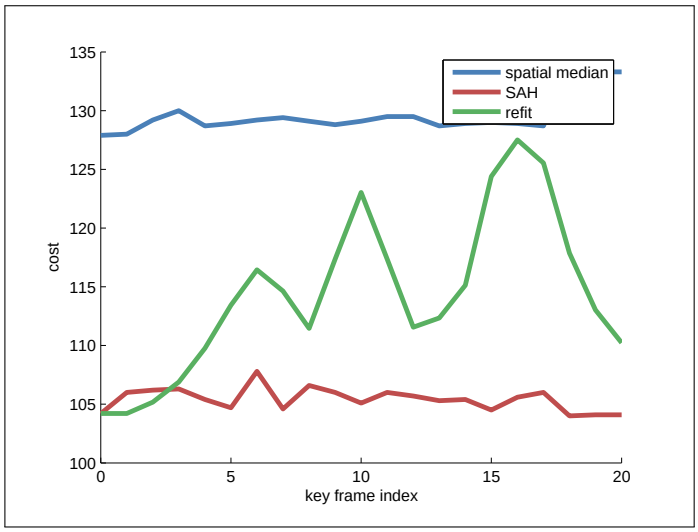


Figure 7.10: Costs of the Fairy Forest scene for both construction algorithms and refitting procedure. We use 8 maximum triangles in leaves as a termination criterion.

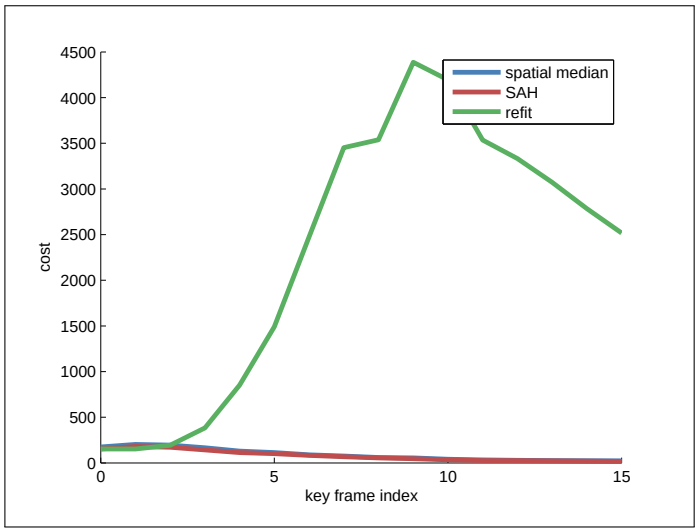


Figure 7.11: Costs of the Exploding Dragon scene for both construction algorithms and refitting procedure. We use 8 maximum triangles in leaves as a termination criterion.

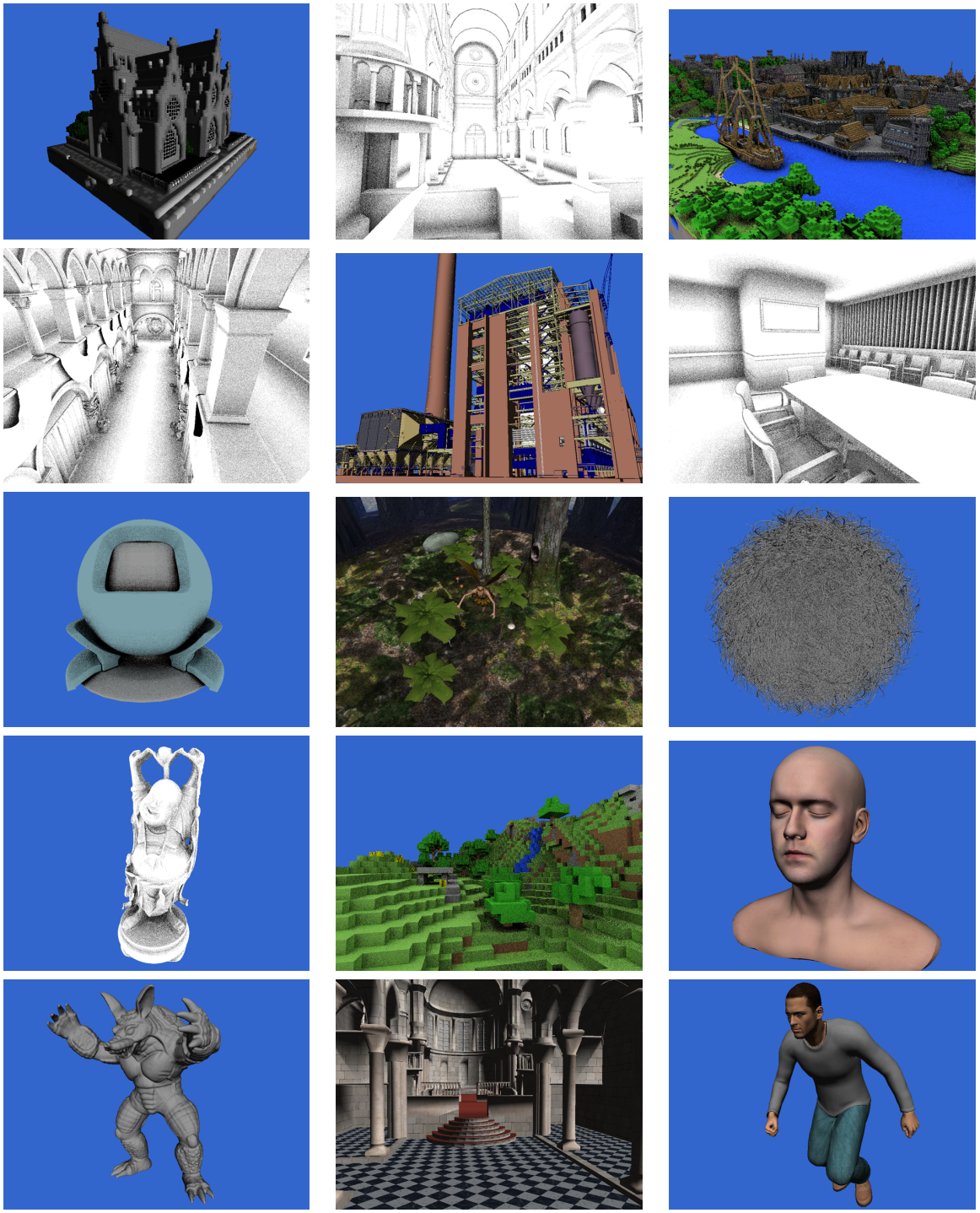


Figure 7.12: Images of scenes rendered with different ray types.

Chapter 8

Conclusion

This thesis addressed fast parallel construction algorithms of bounding volume hierarchies on the GPU. We presented a theoretical background of ray shooting and various acceleration data structures. We reviewed construction methods of bounding volume hierarchies and the BVH construction algorithms designed for massively parallel architecture of modern GPUs.

We implemented two construction algorithms in the CUDA technology according to Garanzha *et al.* [9]. Both algorithms are based on Morton codes and an efficient work queue system. Both algorithms were tested in our rendering system. Our rendering system is implemented in the CUDA technology. The rendering system is based on an efficient ray tracer. We tested both algorithms with different configuration. We tested fourteen static scenes and five dynamic scenes. The complexity of scenes varies between 5k triangles to 12.7M triangles.

The first algorithm constructs the hierarchy using spatial median splits. This algorithm is very fast, but it is suitable only for spatial coherent and symmetric scenes. Construction times vary between 8 ms (Mitsuba, 30-bit Morton codes, 32 maximum triangles in leaves) and 278 ms (Power Plant, 60-bit Morton codes, 8 maximum triangles in leaves). Ray tracing performance varies between 7 MRays/s (Power Plant, 30-bit Morton codes, 32 maximum triangles in leaves, diffuse rays) and 129 MRays/s (Rungholt House, 60-bit Morton codes, 8 maximum triangles in leaves, ambient occlusion rays).

The second algorithm optimizes top levels of hierarchy using SAH splits. This algorithm is slower than the first algorithm, but it is applicable for more complex scenes. Construction times vary between 25 ms (Mitsuba, 30-bit Morton codes, 15-bit clusters, 32 maximum triangles in leaves) and 274 ms (Power Plant, 30-bit Morton codes, 15-bit clusters, 8 maximum triangles in leaves). Ray tracing performance varies between 10 MRays/s (Hairball, 30-bit Morton codes, 15-bit clusters, 32 maximum triangles in leaves, diffuse rays) and 128 MRays/s (Sibenik, 30-bit Morton codes, 15-clusters, 8 maximum triangles in leaves, ambient occlusion rays).

8.1 Future Work

The first algorithm using spatial median splits is very fast, but the parallelism of the algorithm is limited. The first levels of the hierarchy contain only a small number of nodes and

the GPU is not fully occupied. The solution could be the algorithm proposed by Karras [16].

The second algorithm using SAH splits for top levels and spatial median splits is limited as well, but there is another issue. The algorithm constructs top levels of hierarchy using binning and SAH splits. The number of bins scales exponentially with the number of bits used for SAH splits. In hierarchies of large complex scenes the number of the top levels is much smaller than the number of the lower levels. The consequence is that the quality of these hierarchies is comparable with the quality of hierarchies constructed via spatial median splits. The parallel construction of bounding volume hierarchies using the surface area heuristic exclusively is still a challenge. The solution could be the framework introduced by Vinkler *et al.* [26]. The topic of this thesis certainly will discussed in the future.

Bibliography

[1] Timo Aila and Samuli Laine. Understanding the Efficiency of Ray Traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149, New York, NY, USA, 2009. ACM.

[2] Arthur Appel. Some Techniques for Shading Machine Renderings of Solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[3] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, September 1975.

[4] Guy E. Blelloch. Prefix Sums and Their Applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, November 1990.

[5] Robert L. Cook, Thomas Porter, and Loren Carpenter. Distributed Ray Tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145, January 1984.

[6] Philip Dutre, Kavita Bala, Philippe Bekaert, and Peter Shirley. *Advanced Global Illumination*. AK Peters Ltd, 2006.

[7] Christer Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[8] A. Fujimoto, Takayuki Tanaka, and K. Iwata. Tutorial: Computer Graphics; Image Synthesis. chapter ARTS: Accelerated Ray-tracing System, pages 148–159. Computer Science Press, Inc., New York, NY, USA, 1988.

[9] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. Simpler and Faster HLBVH with Work Queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 59–64, New York, NY, USA, 2011. ACM.

[10] A. S. Glassner. Tutorial: Computer Graphics; Image Synthesis. chapter Space Subdivision for Fast Ray Tracing, pages 160–167. Computer Science Press, Inc., New York, NY, USA, 1988.

[11] Jeffrey Goldsmith and John Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Comput. Graph. Appl.*, 7(5):14–20, May 1987.

[12] Yan Gu, Yong He, Kayvon Fatahalian, and Guy Blelloch. Efficient BVH Construction via Approximate Agglomerative Clustering. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 81–88, New York, NY, USA, 2013. ACM.

[13] Vlastimil Havran. *Heuristic Ray Shooting Algorithms.* Ph.D. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[14] W. Daniel Hillis and Guy L. Steele, Jr. Data Parallel Algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986.

[15] James T. Kajiya. The Rendering Equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[16] Tero Karras. Maximizing Parallelism in the Construction of BVHs, Octrees, and K-d Trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, pages 33–37. Eurographics Association, 2012.

[17] Timothy L. Kay and James T. Kajiya. Ray Tracing Complex Scenes. *SIGGRAPH Comput. Graph.*, 20(4):269–278, August 1986.

[18] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[19] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH Construction on GPUs. In *IN PROC. EUROGRAPHICS '09*, 2009.

[20] Duane Merrill and Andrew Grimshaw. High Performance and Scalable Radix Sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(02):245–272, 2011.

[21] Hubert Nguyen. *Gpu Gems 3.* Addison-Wesley Professional, first edition, 2007.

[22] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA. *Queue*, 6(2):40–53, March 2008.

[23] J. Pantaleoni and D. Luebke. HLBVH: Hierarchical LBVH Construction for Real-time Ray Tracing of Dynamic Geometry. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 87–95. Eurographics Association, 2010.

[24] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[25] Steven M. Rubin and Turner Whitted. A 3-dimensional Representation for Fast Rendering of Complex Scenes. *SIGGRAPH Comput. Graph.*, 14(3):110–116, July 1980.

[26] Marek Vinkler, Jiří Bittner, Vlastimil Havran, and Michal Hapala. Massively Parallel Hierarchical Scene Processing with Applications in Rendering. *Computer Graphics Forum*, 32(8):13–25, 2013.

[27] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination.* PhD thesis, Computer Graphics Group, Saarland University, 2004.

[28] Ingo Wald. On Fast Construction of SAH-based Bounding Volume Hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 33–40, Washington, DC, USA, 2007. IEEE Computer Society.

[29] Turner Whitted. An Improved Illumination Model for Shaded Display. *SIGGRAPH Comput. Graph.*, 13(2):14–, August 1979.

[30] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM Trans. Graph.*, 24(3):434–444, July 2005.

Appendix A

List of Abbreviations

1D One-Dimensional

2D Two-Dimensional

3D Three-Dimensional

AABB Axis Aligned Bounding Box

API Application Programming Interface

BSP Binary Space Partitioning

BV Bounding Volume

BV(X) Bounding Volume of the X

BVH Bounding Volume Hierarchy

CUDA Compute Unified Device Architecture

DDA Digital Differential Analyzer

DOP Discrete Oriented Polytop

FPS Frames Per Second

GPU Graphics Processing Unit

GPGPU General-purpose computing on Graphics Processing Units

GUI Graphical User Interface

HLBVH Hierarchical Linear Bounding Volume Hierarchy

LBVH Linear Bounding Volume Hierarchy

OBB Oriented Bounding Box

OpenGL Open Graphics Library

SAH Surface Area Heuristic

SA(X) Surface Area of the X

SIMT Single Instruction Multiple Threads

UI User Interface

Appendix B

Installation Guide

The application requires a graphics card manufactured by Nvidia with the CUDA technology. The compute capability of the graphics card must be at least 1.2. Further the application is dependent on several libraries. Windows 64 bit versions of libraries are included on the attached DVD. An overview of requirements is shown in table B.1.

Library	Min. version
CUDA	5.0
Qt	5.1 OpenGL version
GLEW	1.8
Assimp	3.0
DevIL	1.7.8
CMake	2.8.10
Visual C++	9.0
GCC	4.8.1

Table B.1: Requirements.

The application is compilable on both Windows and Linux. To build the application we require two other tools, CMake and a compiler. Using CMake user can generate either makefile or other project files. We will now describe one of many ways how to build the application.

1. Run the CMake GUI application.
2. Set source code as the *project root* directory.
3. Set build as the *project root*/build directory.
4. Configure and choose an appropriate project file.
5. Generate the project file.
6. Go to build directory where the project file resides.
7. Build the application via the project file.

Appendix C

User Manual

In this appendix we will describe the user interface of the application. First we will describe how to configure the application. Second we will describe the graphical user interface and the application controls.

Parameter	Block	Type	Description
filename	Scene	string	filename of a static scene
filefilter	Scene	string	reg. expression filtering filenames of a dynamic scene
scale	Scene	float	scene isotropic scale
numberOfSamples	Renderer	int	number of samples per primary ray
aoRadius	Renderer	float	radius of ambient occlusion rays
rayType	Renderer	string	primaryRaysFlat, primaryRaysSmooth, aoRays, diffuseRays
width	RenderWidget	int	width of the viewport
height	RenderWidget	int	height of the viewport
maxLeafSize	Bvh	int	maximum triangles in a leaf
mortonCodeBits	Bvh	int	size of Morton codes in bits
mortonCodeSAHBits	Bvh	int	number of bits used for SAH splits
splitMethod	Bvh	string	median, sah
position	Camera	vector	position of the camera
elevAngle	Camera	float	altitude angle of the camera in degrees
viewAngle	Camera	float	azimuth angle of the camera in degrees
nearPlane	Camera	float	near plane of the camera
farPlane	Camera	float	far plane of the camera
fieldOfView	Camera	float	field of view of the camera in degrees
step	Camera	float	move increment value
frameRate	Animation	float	frame rate of an animation scene
loop	Animation	bool	flag indicating animation loop

Table C.1: Configurable parameters.

C.1 Configuration

The application is configurable via a configuration file, i.e. a text file with the `env` suffix. The configuration file is passed to the application as a command line argument. If the configuration file is not specified then the `default.env` file is used, which should resides in the working directory. If the default file does not exist then the default values are used. The configuration file contains named blocks with keys and values. An example of the

configuration file is shown in listing C.1. The overview of all configurable parameters is shown table C.1.

```
RenderWidget {
    width 512
    height 512
}

Scene {
    filefilter ../data/scenes/fairy_forest/f*.obj
}

Renderer {
    rayType aoRays
}

Bvh {
    mortonCodeBits 60
    splitMethod median
}
```

Listing C.1: An example of the configuration file.

C.2 Controls

The application is fully interactive. Drag a mouse to rotate the camera or use the `W, A, S, D` keys to move the camera. Use the menu bar to open settings menu and configure the application in runtime. Use the menu bar to change a ray type or a BVH construction algorithm. Use the menu bar save a screenshot of the current image. Use the menu bar to close the current scene or open new one.

Appendix D

DVD Content