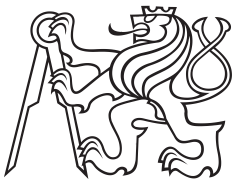
Master's Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Efficient ray tracing algorithms exploiting kd-trees on a GPU

Bc. Robert Papay

Supervisor: prof. Ing. Vlastimil Havran, Ph.D.
Field of study: Open informatics
Subfield: Computer Graphics
January 2025

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Papay** Jméno: **Robert** Osobní číslo: **492304**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačová grafika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Efektivní algoritmy s využitím kd-stromů pro vrhání paprsků na GPU

Název diplomové práce anglicky:

Efficient ray tracing algorithms exploiting kd-trees on a GPU

Pokyny pro vypracování:

Nastudujte literaturu ke stavbě a použití datových struktur na grafickém akcelerátoru pro metodu sledování paprsku se zaměřením na paralelizaci výpočtu a kd-stromy. Implementujte a porovnejte efektivní algoritmy traverzace paprsku kd-stromem na sadě testovacích 3D scén různé velikosti a prostorové distribuce. Dále se zabývejte rychlými algoritmy pro stavbu kd-stromu na grafické kartě pro vrhání paprsku. Vybrané algoritmy implementujte a otestujte pro alespoň deset testovacích scén o počtu trojúhelníků 100K až cca 100M.
Následně se zabývejte algoritmy pro slučování (merge) dvou či více kd-stromů do jednoho kd-stromu a reverzní operaci vyjmutí již vloženého kd-stromu, testovacími implementacemi těchto algoritmů na CPU a následně i jejich přenosem na grafický akcelerátor. Otestujte aplikovatelnost slučování kd-stromu pro jednoduché dynamické scény s jedním i více objekty o počtu trojúhelníků 1K až 50K . Změřte časovou a paměťovou náročnost algoritmů pro stavbu, traverzaci i slučování kd-stromů pro alespoň 5 testovacích scén obsahujících animaci..
Pro implementaci na grafické kartě použijte jazyk CUDA firmy NVIDIA, případně jazyk popsaný standardem OpenCL, případně jiným vhodným programovacím jazykem formou nadstavby nad těmito jazyky jako je HIP/SYCL.

Seznam doporučené literatury:

1) Zhou et al.: Real-Time KD-Tree Construction on Graphics Hardware,
SIGGRAPH ASIA 2008 and references to this paper.
2) M. Vinkler: Construction of Acceleration Data Structures for Ray Tracing, PhD thesis, Masaryk University 2014.
https://is.muni.cz/th/w0k6h/? kod=PV204
3) D. Horn, J. Sugerman, M. Houston, P. Hanrahan, Interactive k-D Tree GPU Raytracing, 2007.
4) Z. Wu, F. Zhao, X. Liu: SAH KD-tree construction on GPU, HPG 2011.
5) S. Chung, M. Choi M, D. Youn D and S. Kim S. (2019). Comparison of BVH and KD-Tree for the GPGPU Acceleration on Real Mobile Devices. Frontier Computing. 10.1007/978-981-13-3648-5_62. (535-540).
6) X. Liang X, H. Yang, Y. Zhang, J. Yin J and Y. Cao (2016). Efficient kd-tree construction for ray tracing using ray distribution sampling. Multimedia Tools and Applications. 75:23. (15881-15899).
Další literaturu dodá vedoucí práce.

Jméno a pracoviště vedoucí(ho) diplomové práce:

prof. Ing. Vlastimil Havran, Ph.D. Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **15.02.2024** Termín odevzdání diplomové práce: **07.01.2025**

Platnost zadání diplomové práce: **21.09.2025**

prof. Ing. Vlastimil Havran, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studenta

Acknowledgements

I would like to primarily thank my supervisor, prof. Ing. Vlastimil Havran, Ph.D., for his guidance and advice while working on this thesis. I would also like to express my deepest gratitude to my family and friends for their invaluable support for the whole duration of my studies. Lastly, I would like to thank my friends that helped me proofread the thesis.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

In Prague, 7. January 2025

Abstract

This work focuses on parallel algorithms for building and traversing k-d trees to accelerate ray tracing, mainly for dynamic scenes. It provides an introduction to k-d trees as well as GPU architectures. It also provides an overview and analysis of existing solutions for building k-d trees in parallel, for building k-d trees for dynamic scenes, and for traversing k-d trees on the GPU.

A game engine–like framework is designed and implemented to provide a basis for implementing the ray tracing related algorithms. An algorithm design and implementation for building k-d trees on the GPU using binning and exploiting a task pool is presented, with detailed description of all its steps. An algorithm design for merging k-d trees on the CPU and GPU is also presented.

Nine algorithms are implemented in total: two CPU single-threaded k-d tree building algorithms using exact and approximate split selection, one GPU k-d tree building algorithm using binning, one CPU k-d tree merging algorithm and four traversal algorithms designed for the GPU, in addition to the traditional stack-based traversal algorithm. The implemented algorithms are tested on ten static and five dynamic scenes, with tables for each scene presenting the final results.

Keywords: ray tracing, k-d tree, GPU, GPGPU, min-max binning, task pool, k-d tree merging

Supervisor: prof. Ing. Vlastimil Havran, Ph.D.
Katedra počítačové grafiky a interakce, ČVUT FEL

Abstrakt

Tato práce se zaměřuje na paralelní algoritmy pro stavbu a procházení k-d stromů za účelem zrychlení algoritmu sledování paprsku, převážně pro dynamické scény. Práce poskytuje úvod do k-d stromů a do architektury grafických karet. Dále poskytuje přehled a analýzu existujících řešení pro paralelní stavbu k-d stromů, pro stavbu k-d stromů pro dynamické scény, a pro procházení k-d stromů na grafické kartě.

Je navrhnut a implementován framework podobající se hernímu enginu, aby poskytl základ na implementaci algoritmů souvisejících s algoritmem sledováním paprsků. Je představen návrh a popis implementace algoritmu stavby na grafické kartě využívající binning a frontu úloh, s podrobným popisem všech kroků. Je také představen návrh algoritmů na vkládání k-d stromů na CPU a GPU.

Celkem je implementováno devět algoritmů: dva jednovláknové algoritmy stavby na CPU, používající přesný a přibližný výběr dělící roviny, jeden algoritmus stavby na GPU používající binning, jeden algoritmus na CPU pro vkládání k-d stromů, a čtyři algoritmy na procházení k-d stromů určené pro grafické karty spolu s tradičním algoritmem na procházení založeným na zásobníku. Implementované algoritmy jsou otestovány na deseti statických a pěti dynamických scénách. Výsledky jsou prezentovány v tabulkách pro každou scénu.

Klíčová slova: sledování paprsku, kd-strom, GPU, obecné výpočty na GPU, min-max binning, fronta úloh, vkládání k-d stromů

Překlad názvu: Efektivní algoritmy s kd-stromy pro ray tracing na GPU

Contents

Figures

Tables

Chapter 1

Introduction

Ray tracing and rasterisation are the two leading methods of image synthesis in computer graphics. Ray tracing simulates how light behaves by tracing the path of light particles (rays) shot from light sources, capturing the light's colour and intensity when they hit the sensor of the camera. Because most of the light rays would not hit the sensor, rays are traditionally shot from the camera through a pinhole instead (sometimes called backward ray tracing). The main disadvantage of ray tracing is its computational complexity. Even the simplest ray tracing algorithm with only primary rays (rays shot directly from the camera) needs to shoot as many rays as there are pixels to be displayed, which is, for instance, approximately two million rays for a Full HD screen. A more advanced ray tracing algorithm that includes light bounces and shadows is the Whitted ray tracing algorithm [Whi80]. It simulates real-world effects by reflecting and refracting on each bounce of each ray and shooting shadow rays (rays shot towards lights, used to determine whether the point is under direct light or not) to simulate shadows. This increases the amount of rays exponentially as the rays bounce on various surfaces. The large number of rays means that to get an interactive frame rate, we have to get a performance of several megarays per second (Mr/s) or even better gigarays per second (Gr/s). Due to this, the much less demanding rasterisation has been and still is the preferred rendering method for most real-time applications, even though physical effects are much harder to simulate with it. But as the performance of GPUs has been increasing over the years, ray tracing is slowly becoming more and more affordable.

To simulate ray bounces and shadowing, the rays need to intersect with scene geometry. The naive algorithm of intersecting each primitive with each ray has linear time complexity for a single ray. To get a reasonable performance, we have to implement acceleration data structures that improve it, most commonly to a logarithmic average case time complexity. The most popular ones for ray tracing are uniform and non-uniform grids, bounding volume hierarchies (BVHs) and k-d trees. There are many modifications to each of these base data structures, and there is generally no "best" data structure for all scenes. The main disadvantage of k-d trees is the memory requirements, which are unknown until the tree is built and require copies of primitive references to be distributed among leaf nodes (=leaves). This work

will mainly focus on k-d trees for ray tracing [Hav00].

Let us assume that we have built a k-d tree for the scene we want to display. As each primary ray is independent, ray tracing is inherently parallelisable. Therefore, we can accelerate it using a GPU (graphics processing unit), a computation unit designed to be a massively parallel system, but we have to be careful when porting algorithms designed for a CPU to the GPU. The traditional k-d tree traversal algorithm uses a stack. Each thread needs its own copy and the size of the stack is unknown, although we know it cannot exceed the depth of the tree. Because GPU memory and programmability was much more limited in the past, Foley et al. [FS05] developed two stack-less algorithms, kd-restart and kd-backtrack. Horn et al. [HSH07] then expanded the kd-restart algorithm with two optimisations, called push-down and short-stack. A part of this work focuses on implementing and comparing the traditional and stack-less traversal algorithms (excluding kd-backtrack) on modern graphics cards using GPGPU (General-Purpose computing on GPUs).

We have yet to mention how to build a k-d tree. For static scenes, the tree needs to be computed only once at the beginning or even precomputed, so it is not important for real-time interactive applications. But in most interactive applications, the scenes are rarely completely static, instead being composed of static and moving (dynamic) parts. When rendering a dynamic scene, at least some parts of the tree need to be rebuilt each frame, so the tree creation also becomes a critical part of the rendering process. The surface area heuristic (SAH, [GS87; MB90]) is a widely used heuristic for building good quality k-d trees. A popular algorithm that uses it, developed by Wald and Havran [WH06] with its $O(n \log n)$ time complexity, is unfortunately still too slow for interactive applications when run on a single thread. Large part of this work focuses on building algorithms, with the main focus being on parallelisation.

If something in this work is not explained clearly or the reader would like to learn more about the topic than what is explained in this work, we forward the reader to an overview of modern ray tracing by Li et al. [LDG17].

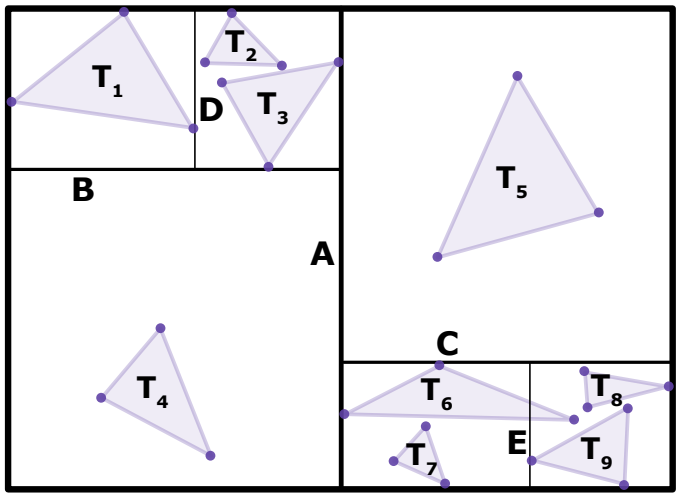
Chapter 2

Theoretical background

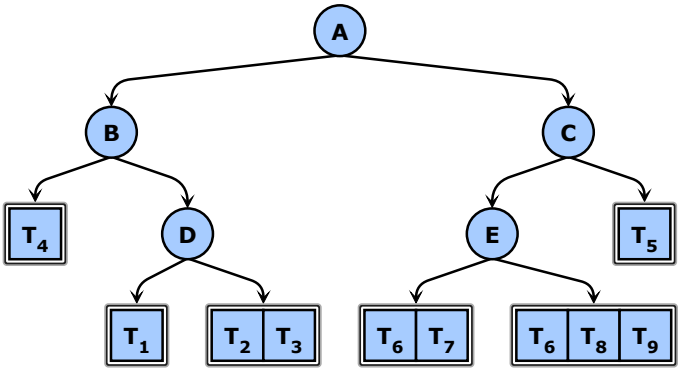
The k-dimensional tree is a space partitioning data structure, which extends binary trees to k dimensions [Ben75]. A 1-d tree equates to a classic binary search tree. Each node corresponds to an axis-aligned bounding box and contains the coordinate and axis of a partitioning plane. The plane partitions the node into two children, representing two smaller, neighbouring bounding boxes (see Figure 2.1a). When traversing the tree, we can decide whether to traverse the "left" and/or "right" sub-tree based on whether we are interested in the corresponding sub-space the sub-tree envelops. The left child corresponds to the one oriented to the negative infinity along the partitioning axis, right child to the positive infinity. It was originally developed for point sets and its most well-known use is the nearest neighbour search. Due to being a binary tree, it has expected logarithmic insertion, deletion, and query times. Although organisation of points is the most straightforward use of k-d trees, it can also be used to organise other primitives, such as triangles, according to their bounding boxes, which is also used in computer graphics.

2.1 Construction

The efficiency of tree data structures depends on their quality, which depends on the building algorithm. If a binary tree has only left children, the expected logarithmic times become linear. Different uses have different metrics that describe how the tree should be built. A simple metric for binary trees that ensures logarithmic times is how balanced it is. Balanced trees minimise the absolute difference of depths of left and right sub-trees for all nodes. When building a k-d tree, the selection of the coordinates and axis of the partitioning plane is crucial, as it is the only variable that influences tree quality. We can construct the tree incrementally in $O(n \log n)$ time by repeatedly inserting primitives, but when all primitives are known at the time of construction, it is easier to optimise the tree quality by considering all primitives at once. For a uniform distribution of points, the basic algorithm places the partitioning plane near the object median, which creates a balanced tree. The axis is chosen in a round-robin fashion (in 3D: x, y, z, x and so on, see Figure 2.2a for a 2D example). If we always chose the x axis, the tree would be balanced, but the space would be partitioned into narrow rectangles, hindering the



(a) Diagram of the 2-d tree

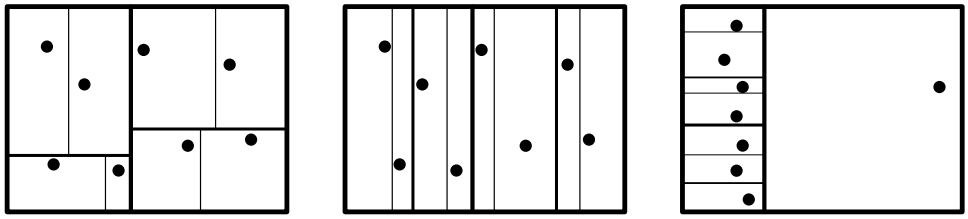


(b) Graph of the 2-d tree

Figure 2.1 An example of a 2-dimensional k-d tree. Notice how triangle T_6 crosses the splitting plane and thus has to be stored in both leaves of node E.

performance of nearest-neighbour searches (see Figure 2.2b). These heuristics work well for uniform distributions, but both fail in the example in Figure 2.2c. There, it would be best to place the first split near the second left-most point on the x axis, and then place the rest of the splits on the y axis. As we can see, general cases require more sophisticated heuristics to create more efficient trees.

As mentioned above, we can use k-d trees to organise other primitives such as triangles (see Figure 2.1). Primitives belong in a sub-tree if they intersect the corresponding bounding box of the node. The position of the splitting plane determines into which child or children a primitive belongs. There are cases where a splitting plane always crosses some primitives or where a split that crosses primitives will be considered optimal. This leads to increased memory requirements when duplicating primitive references in both sub-trees. It also causes the memory requirements to be unknown until the whole tree is built.



(a) Round-robin **(b)** Single axis **(c)** Smart

Figure 2.2 k-d trees for point sets built using different axis selection methods

2.2 Surface Area Heuristic

For ray tracing purposes, Surface Area Heuristic (SAH, [GS87; MB90]) has been considered the state of the art heuristic for building good quality k-d trees for over 25 years. Let us assume that the rays are uniformly distributed infinite lines, the costs of the traversal step K_T and the primitive intersection K_I are known, the cost of intersecting N primitives is linearly dependant on N and that we are intersecting bounding boxes. We can then derive a formula using geometric probability theory [Sol78] for the conditional probability that a ray will intersect box B_{sub} located inside box B if it hits B:

$$P(B_{sub}|B) = \frac{SA(B_{sub})}{SA(B)}, \tag{2.1}$$

where $SA(B)$ is the surface area of B. The cost function C for an inner node represented by its bounding box B and split by splitting plane π, having two child nodes represented by boxes B_L and B_R, is then equal to

$$C(B, \pi) = K_T + P(B_L|B) \cdot C(B_L) + P(B_R|B) \cdot C(B_R) \tag{2.2}$$

We choose the splitting plane so that the cost is at its minimum:

$$C(B) = \min_{\pi} C(B, \pi),$$

where the cost of a leaf node containing primitives T is defined as

$$C(B_{leaf}) = |T| \cdot K_I$$

The cost of a tree is equal to the cost of the root node. As we can see, the cost function is recursive. When computing the cost of an inner node, we need to compute costs for each position of the splitting plane to find the minimum. For each position of the splitting plane, we need to compute the costs of child nodes, which require the same process. That means that finding the true minimum cost of a tree has exponential time complexity.

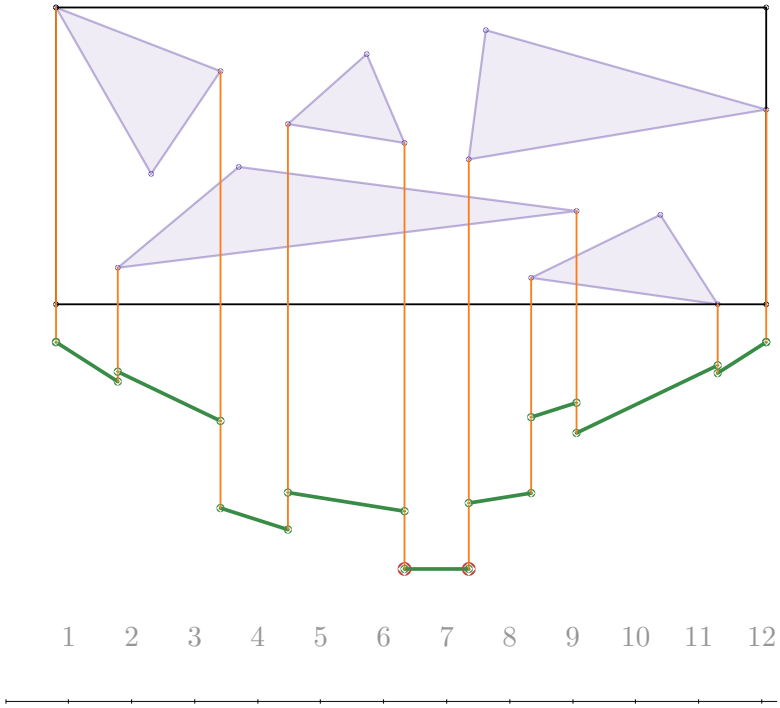


Figure 2.3 Visualisation of the SAH function along the x axis

To avoid this, we usually simplify the heuristic by using a locally greedy approximation that assumes that the cost of a child node is roughly equal to the cost of a leaf node containing the same primitives, meaning we compute $C(B_L)$ and $C(B_R)$ as

$$C(B_L) = |T_L| \cdot K_I, C(B_R) = |T_R| \cdot K_I \tag{2.3}$$

Substituting Eq. 2.3 together with Eq. 2.1 into Eq. 2.2 gives us the locally greedy approximation cost function that we will use:

$$C(B, \pi) = K_T + \frac{SA(B_L)}{SA(B)} \cdot |T_L| \cdot K_I + \frac{SA(B_R)}{SA(B)} \cdot |T_R| \cdot K_I \tag{2.4}$$

When moving the splitting plane, only $SA(B_L)$, $SA(B_R)$, $|T_L|$ and $|T_R|$ change, while the other values remain constant. $SA(B_L)$ and $SA(B_R)$ are linear and continuous with respect to the coordinate of the splitting plane. Whenever one value rises, the other value decreases, and vice versa. The numbers of primitives, $|T_L|$ and $|T_R|$, change only when the splitting plane starts or stops crossing a primitive. The resulting function is piecewise constant. When we combine these four values into Eq. 2.4, we get a piecewise linear function with points of discontinuity at the edges of bounding boxes of primitives. This function is represented in green in Figure 2.3. The figure also colours the function where the cost is at its minimum in red. In this

case, the minimum is anywhere along the line segment between the two red points. In general, there will always exist at least one point where the cost function will have its minimum and that will lie on the edge of a bounding box of some primitive. This means that it is necessary to compute the cost function only at the edges of the bounding boxes of primitives and to take the splitting coordinate where the cost function is at its minimum.

A naive approach of evaluating the SAH would be to iterate over all primitives in the node and their bounding boxes and compute the surface areas and primitive counts on the left and right. This approach leads to $O(N^2)$ time complexity for each node with N primitives, because we have to iterate over all primitives to count them. Because binary trees built over N primitives have $O(\log N)$ nodes, the overall complexity is $O(f(N) \log N)$, where $f(N)$ is the time complexity of processing a node. The time complexity of building the entire tree using the naive approach is thus $O(N^2 \log N)$.

A better approach would be to first sort the minimal and maximal coordinates of the bounding boxes of primitives. We can then sweep across the coordinates (called *events*) along the splitting axis, increasing and decreasing the primitive counts accordingly. Sorting (with $O(n \log n)$ time complexity can be done either at each node, leading to an $O(n \log^2 n)$ algorithm, or at the start, as the node partitioning maintains the event order, leading to an $O(n \log n)$ algorithm.

In later chapters, we talk about exact and approximate split selection. Evaluating the SAH at primitive bounding box edges leads to selecting the so-called *perfect split* and is the exact split selection. When approximating the splitting plane position, we are computing exact cost values, but not necessarily at the correct coordinates, which may lead to selecting a split that is not perfect.

2.3 Split clipping

A way to further improve the quality is to clip primitives to the node's bounding box [Hav00]. If we simply intersect primitive bounding boxes with node bounding boxes, we may include primitives in leaves where they do not intersect the leaf's bounding box (see Figure 2.4). To remove these false positives, we have to clip the primitive with the bounding box of the node. This may create a smaller primitive when the primitive is not entirely inside the bounding box, or even remove the primitive entirely. We can remove the false positives when splitting a node into two children by splitting primitives that are straddling the splitting plane with the splitting plane.

Because clipping can change the order of events, we have to sort the events again in each node, leading to the $O(n \log^2 n)$ time complexity. Wald and Havran [WH06] introduced an algorithm that allows for primitive clipping with better time complexity. Based on the assumption that only $O(\sqrt{n})$ primitives straddle the splitting plane and the observation that only these need to be sorted, they introduce an $O(n)$ algorithm for processing the nodes, leading back to the $O(n \log n)$ time complexity.

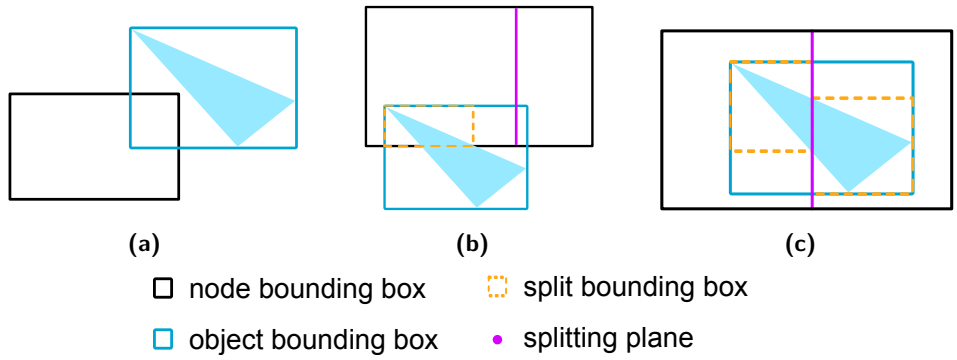
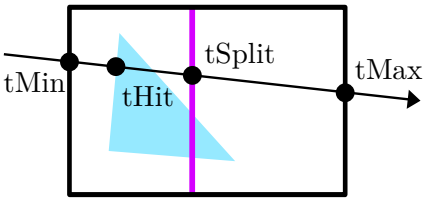
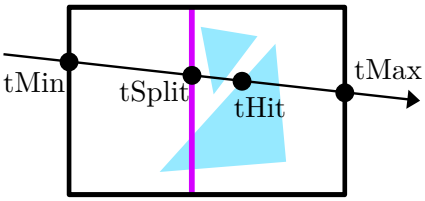


Figure 2.4 Issues when not using split clipping. In (a) the primitive bounding box intersects the node but the primitive is outside the node. In (b) the primitive bounding box intersects both children but the primitive belongs only to the left child. In (c) the primitive bounding boxes in both children without clipping are larger than with clipping.



(a) Closest intersection found is inside the first node, we can end traversal.



(b) Closest intersection found is outside the first node, we have to check the other node (where there is a closer intersection).

Figure 2.5 Ray intersecting an inner node with two leaf child nodes. First, we traverse the left child, then the right child.

2.4 Traversal

The goal of ray tracing is to find a primitive p that intersects a given ray r with origin $\mathbf{o}$ and direction $\mathbf{d}$ at point $\mathbf{I}$, parametrised as $\mathbf{I} = \mathbf{o} + t\mathbf{d}, t \geq 0$. We either want to find the closest intersection (= find I with lowest t, used for primary and secondary rays) or find if there is *any* intersection (= find any I, used for shadow rays). t may also be limited by t_{max} as $0 \leq t \leq t_{max}$. This is used, for example, by point lights, because without t_{max}, the shadow ray would get blocked by primitives behind the light.

Calculating intersections is usually expensive. The purpose of acceleration data structures is to prune as many primitives as possible, so that the ray has to compute as little intersections as possible. k-d trees achieve this by dividing primitives into nodes, as mentioned previously. The ray then has to traverse only the nodes whose bounding boxes it intersects, computing primitive intersections only in the leaves it reaches. Acceleration data structures are designed so that traversing a node is much cheaper than intersecting a

primitive. Because k-d trees divide the space without overlapping, we can also end primary and secondary ray traversal early if the intersection we found is inside the leaf (see Figure 2.5a). When the closest intersection is outside the leaf, there might be another, closer primitive in the other node (see Figure 2.5b).

The classic algorithm (see Algorithm 1) uses a stack to store nodes that need to be traversed later. It begins with intersecting the scene bounding box and calculating the distances sceneMin and sceneMax, which is not shown in the algorithm. If the ray does not intersect the scene bounding box, the traversal ends. There are three distances we need when traversing the k-d tree: the distance from the ray origin to the entry point of the node's bounding box tMin, the distance to the exit point tMax, and the distance to the intersection with the splitting plane tSplit. There is also the distance to the intersection tHit, which is the output of the algorithm.

In each step, the algorithm determines which child node to traverse. If both children need to be traversed, it orders them so that the first one is closer to the ray origin. The second node then goes onto the stack and the algorithms continues traversing the first node. When it reaches a leaf, it intersects all primitives inside that leaf with the ray. If it does not find an intersection that can end the traversal, it pops a node from the stack and uses it in the next traversal step. This is repeated until a traversal–ending intersection is found or the stack is emptied.

Algorithm 1: Stack traversal

```
1  begin
2      stack.push(root, sceneMin, sceneMax)
3      tHit = infinity
4      while tMax < sceneMax do
5          (node, tMin, tMax) = stack.pop()
6          while not node.isLeaf() do
7              a = node.axis
8              tSplit = (node.value - ray.origin[a]) / ray.direction[a]
9              (first, second) = order(ray.direction[a], node.left, node.right)
10             if tMax ≤ tSplit then
11                 node = first
12             else if tSplit ≤ tMin then
13                 node = second
14             else
15                 node = first
16                 tMax = tSplit
17                 stack.push(second, tSplit, tMax)
18             end
19         end
20         for p in node.primitives do
21             tHit = min(tHit, intersect(ray, p))
22         end
23         if tHit < tMax then
24             return tHit
25         end
26     end
27     return tHit
28 end
```

Chapter 3

Massively parallel architectures

GPUs are massively parallel architectures, focused on data throughput rather than complex flow control. The GPU is not suitable for executing whole programs, because it would not be able to effectively do many needed tasks such as processing input, executing complex algorithms not designed for parallel computation and so on. Rather it is designed to work together with the CPU, where the CPU offloads highly parallelisable work to the GPU, which amortises the computation cost needed for large amounts of data with its high throughput, all while the CPU can work on other tasks. Instead of having several more complex processing units like the CPU that can run only a few threads in parallel, it has many simpler, but powerful processing units (called Streaming Multiprocessors in CUDA, Compute Units in OpenCL) with many threads. Modern GPUs have two main concepts which differ from the standard execution on a CPU, its execution model and the memory model. There are other concepts like bank conflicts, latency masking and so on, but we will not go into detail about them in this work. For more information about GPGPU, refer to [HHS23]. The models described in this chapter are based mainly on NVIDIA GPUs and the CUDA language [Cor] and may not be accurate for other GPUs.

3.1 Execution model

Modern desktop CPUs have several cores (e.g. eight). Let us assume that each core can only run a single thread at a time. Each thread is usually executing different instructions on different data (the concept is called MIMD, or multiple instructions, multiple data). MIMD can be used for almost anything, but is best used for independent tasks, for example a main thread rendering UI, while a secondary thread processes data so that the UI is responsive. CPUs usually also have something called vector instructions, which execute the same instruction on multiple data (this concept is called SIMD, or single instruction, multiple data). Each thread can execute its own vector instructions. SIMD can be used, for example, for vector addition or matrix multiplication.

GPUs operate on a model called SIMT (single instruction, multiple threads), where all threads execute the same program called the *kernel*, but not all

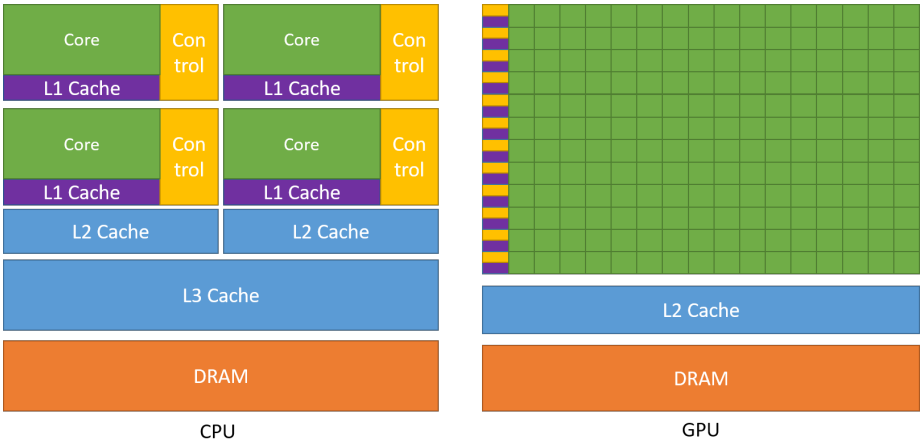
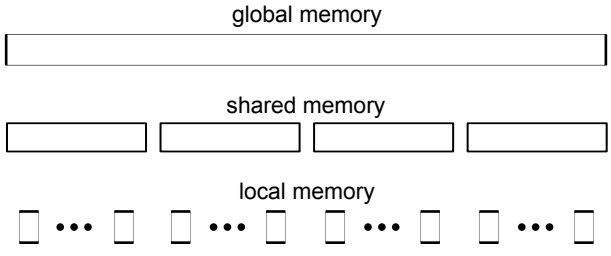


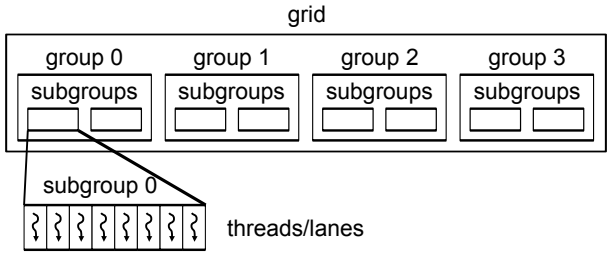
Figure 3.1 Comparison of the CPU and GPU architectures from [Cor]

threads have to be executing the same instruction at the same time. Instead of several threads, GPUs usually operate with several thousand threads, and GPU cores can run multiple threads at the same time. The threads are grouped into three layers (see Figure 3.2b), depending on how they interact with each other. We use the following terminology (with equivalents from CUDA and OpenCL in parentheses):

- **Thread** (CUDA thread, OpenCL work-item) - The base building block. Each thread has its own stack and executes given instructions.
- **Subgroup** (CUDA warp, OpenCL sub-group, AMD wavefront) - The first layer, groups threads in a similar manner to vector instructions on the CPU. On older architectures, all threads in a subgroup were always executing the same instruction. It is the smallest synchronisation primitive and exchanging data is most efficient at this layer. NVIDIA warps consist of 32 threads, AMD wavefronts of 64 threads.
- **Group** (CUDA block, OpenCL work-group) - The second layer, consisting of several subgroups. The number of subgroups in a group is usually user-defined. Each group runs on the same processor, giving the threads in a group access to faster, shared memory (explained more in the next section). Synchronisation in this layer is usually still fairly cheap.
- **Grid** (CUDA grid, OpenCL nd-range) - This layer consists of groups and includes all launched threads. Communication between groups must happen through global memory and synchronisation is expensive (for example, it is not even built-in in CUDA).
- **Lane** (CUDA lane, OpenCL ×) - What we call threads when talking about them in the context of subgroups. Thread 0 can mean the first thread in the grid or a group, but lane 0 is always the first thread in a subgroup.



(a) Memory model



(b) Execution model

Figure 3.2 GPU memory and execution models with 4 groups, 2 subgroups and 8 lanes

It is common for GPGPU language standards to provide methods for retrieving indices in the context of at least some of the layers. The grid can also be multi-dimensional. For example, we can run 64 threads in a 1D block of 64 threads or a 2D block of 8×8 threads. This is mainly used for easier indexing, for example when we are processing 2D arrays such as an image or a matrix.

3.2 Memory model

The memory model is also different from the standard CPU memory model with single global memory and several cache layers. The first differentiation is between host and device memory. Host memory is the standard computer memory, device memory is the GPU memory. Device memory has at least three layers in concept - in CUDA it is the global memory, shared memory and local memory (see Figure 3.2a). OpenCL names them global, local, and private memory, respectively. Global memory is accessible by all threads in a grid and is the only memory accessible from the host. It is also the slowest memory to access (if we do not count accessing host memory from kernels), so we have to be careful about using it when designing algorithms. Shared memory is shared between threads in a group and is much faster to access because it is generally located in a shared cache. It is best used to cache data from global memory and operate on the data before writing them back, instead of operating on them in global memory. Local memory is the memory of each thread and can only be accessed by the owning thread.

Chapter 4

Previous work on building k-d trees

There have been several attempts at building the k-d tree in parallel, both on the CPU and the GPU. GPU methods have become more prevalent due to GPGPU becoming more accessible and large scenes requiring a lot of work. In this chapter, we introduce research done on the topic with a focus on dynamic scenes.

Parallelising the construction process is difficult due to several reasons. The first reason is that almost all algorithms need to build the tree in a top-down manner. We can choose between a depth-first (DFS) or breadth-first (BFS) approach. When we choose DFS, we assume that the goal is to assign a sub-tree to each thread, so that the sub-trees can be built independently. The most prevalent problem with this approach is processing the first layers, where the number of tree nodes is lower than the number of available cores. We will call these first layers the *top layers*, and the rest the *bottom layers* (see Figure 4.1). Processing the top layers without parallelisation can prove to be a significant bottleneck.

In the BFS approach, we build one level of the tree at a time. If done correctly, we should not have problems with any part of the process, but we have to synchronise the threads for each level. In both approaches, we also have to realise that the k-d tree is not a balanced tree and the number of primitives in different nodes can vary greatly. Without proper load balancing, some threads may be assigned significantly more work than others, leading to wasted parallel potential.

The second reason is the unknown memory requirements. Dynamic allocations from running kernels hinder the performance on GPUs and were not even possible in the early days of GPGPU. The final difficulty is evaluating the SAH while split clipping, because we need to be constantly sorting the event lists for each node. This also coincides with the previous reason, as each node needs to hold its own sorted primitive list, whose size can grow the deeper we go due to primitive reference duplication.

4.1 Early approaches

One of the first attempts to parallelise the construction process was made by Shevtsov et al. [SSK07] on the CPU. They suggest a fully parallel algorithm

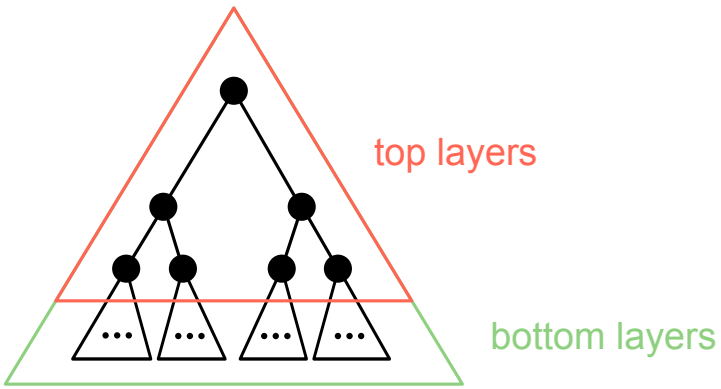


Figure 4.1 Top and bottom layers of a k-d tree when building using the DFS approach with four cores

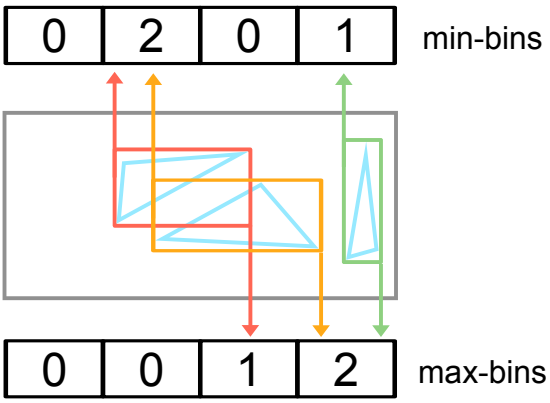


Figure 4.2 Min-max binning for the x axis

for a multi-core CPU architecture (they used four cores). The main idea is to use min-max binning for both the initial decomposition into sub-trees in the top layers and the approximate split selection.

Min-max binning works by creating two arrays for each dimension, essentially splitting the space into equally sized bins. Each min-bin is used to store the number of primitives whose bounding boxes start inside the space the bin represents. Each max-bin does the same for the ends of primitive bounding boxes (see Figure 4.2). The bin boundaries are taken as candidates for the splitting plane. To calculate the number of primitives on the left of the left candidate plane of bin b, we sum all min-bin values preceding bin b excluding the value of b itself. To calculate the number of primitives on the right, we sum all max-bin values following bin b, including value of b. So, for example, the splitting plane candidate in the middle from Figure 4.2 has two primitives on the left and three on the right. Using this information we can easily

and precisely compute the cost for each candidate. To eliminate repeated calculations and allow for parallelisation, we can perform an exclusive scan on the min-bins and inclusive scan on the max-bins and use the new bin values as the numbers of primitives on the left and right.

The parallel version of the whole algorithm consists of three stages - the initial clustering phase, the approximate local building phase, and the exact local building phase.

The initial clustering phase handles the top layers. They use low resolution parallel min-max binning to repeatedly find the approximate object median and split the primitives according to it. They argue that using the object median should result in natural load balancing for scenes with uniform primitive distribution. The phase ends when the number of clusters is equal to the number of available cores. The created sub-trees are distributed among threads running on the cores.

The approximate local building phase uses min-max binning to find the best split candidate. It uses a fixed number of bins for each node, which exponentially increases the total number of bins in each layer. It ends when the number of primitives is less or equal to the number of bins.

The exact local building phase uses exact split selection to find the perfect split. To ensure good load balancing for general scenes, the local building phases create tasks for new nodes. The tasks are put in a shared task pool with a limited size, which distributes the work among the running threads.

Popov et al. [DPS10] also use min-max binning, but their implementation runs on the GPU. They construct the tree in a BFS manner, where processing a level is called a step. Each step consists of three phases: binning with cost computation and split selection (Compute-Cost), creating the children (Split), and split clipping (Triangle-Splitter). Steps are processed in different stages. The specific stage is chosen based on the number of primitives in nodes.

The algorithm runs almost entirely on the GPU, but requires CPU synchronisation. From what we understand, each combination of a stage and a phase has its own kernel. For each level, they launch as many blocks as the GPU can run. When a block finishes processing a task, it fetches another one to work on from a pool, which is based on the idea of persistent warps [AL09], but with blocks.

Zhou et al. [ZHW08] presented an algorithm that uses the GPU for parallel computations with CPU coordination and synchronisation. They build the tree in a BFS manner, but not strictly layer by layer due to their empty space maximising heuristic. The main idea is to recognise two different types of nodes during the building process based on the number of triangles they currently contain - *large* and *small* nodes. If the node contains more than T triangles, which is a user set parameter (equal to 64 in their implementation), the node is recognised as large, otherwise it is considered small.

The algorithm first processes all large nodes using spatial median splitting and empty space maximising, and then all small nodes using exact SAH computations. The large nodes are processed in a loop, where in each iteration, the current active list of nodes is processed. The newly created

nodes are saved to either a list for the next iteration if the node is considered large, or to a list of small nodes if it is considered small. At the end of the iteration, the list for the next iteration is assigned to the active list and is cleared. When only small nodes remain, the algorithm moves onto processing them in a similar manner.

The implementation requires a lot of CPU synchronisation and many kernel calls. Chang et al. [CSI15] tried to modernise the algorithm by using modern enhanced intrinsic CUDA functions and calling fewer kernels. Their version succeeded in improving the original, both in the number of kernel calls and total build time.

4.2 Exact split selection

There were also attempts to parallelise algorithms with exact split selection by using events for the whole construction. Some parts of the algorithm are trivial to implement in a multi-threaded environment in isolation. For example, the initial sorting can be performed by existing parallel sorting algorithms [AR14] and the evaluation of the SAH is independent and can also be parallelised. The main problems arise from the primitives straddling the splitting plane, which increase memory requirements and create the need to re-sort event lists in each node when split clipping.

The algorithms presented by Choi et al. [CKL10] forego the clipping, allowing them to create two parallelisations, which they call "*nested*" and "*in-place*". The nested version works as a parallelised version of the sequential algorithm. It uses two levels of parallelism, node and geometry level parallelism. The geometry level parallelisation uses parallel scans on event lists to evaluate the SAH. It also uses a parallel scan to distribute primitives into child nodes. Similarly to [SSK07] the algorithm employs parallelism in both the top layers by using the above-mentioned techniques and in the bottom layers by building independent trees when the number of nodes reaches the number of cores.

The in-place algorithm tries to alleviate memory consumption problems that arise from nodes having duplicated primitives by operating on triangles instead of nodes in a BFS manner. Each triangle gets pointers to its events and an array of nodes it belongs to at the current level. The idea builds on the assumption (which was experimentally backed up) that while a node can contain many triangles with duplicates, a triangle mostly belongs to only a few nodes at each level. As with the previous algorithm, they suggest switching to local k-d tree construction when the number of nodes exceeds the number of cores.

Wu et al. [WZL11] attempted to parallelise the exact split selection algorithms with split clipping. They use the same idea for SAH evaluation as [CKL10], using parallel scans and reductions, but include split clipping. Their main contribution is a bucket-type sorting algorithm that is better suited for GPU computations (mainly because it is easily parallelisable) than the sequential sort-and-merge algorithm presented by Wald and Havran [WH06]. They also deduce that the sorting has to be done only on axes that are not

the splitting axis. They claim that their algorithm runs entirely on the GPU, but it seems that it runs mostly on the GPU with some CPU synchronisation. Evidence for this is the `ParGenTree` function, which uses dynamic memory allocation (first added with compute capability 2.0 on Fermi architecture in 2010 [Cor]) and calls functions that have the parallel keyword in them, which the authors use to signify spawning multiple threads, hinting at it meaning running a kernel.

Zhou and Meng [ZM11] introduced an algorithm for a hybrid CPU-GPU architecture. They process nodes on the CPU in parallel, and when the number of events is larger than a constant parameter, they offload the split selection onto the GPU. To provide good load balancing, work-stealing is allowed. Each CPU thread has a local queue of nodes to process. When one of the threads empties its queue, it "steals" half the work (measured in total number of events, not nodes) from another, so-called "victim" thread.

4.3 Modified k-d trees

There were also attempts to slightly modify the basic structure of k-d trees to improve its parallelisation. Some use the scene graph to build a multi-level data structure. Kang et al. [KNP13] propose a new data structure they call "gkDtree", or group-based k-d tree. It uses the scene graph to build a two-level k-d tree, where scene nodes are translated into "group" nodes that form a k-d tree with scene objects as their primitives. Each group leaf node then has an attached local, classic k-d tree built for the scene object. The base tree is built for the whole scene, but is updated only for dynamic groups (groups with dynamic objects), for which the article provides a parallel load balanced algorithm. Wang et al. [WGD14] propose a similar data structure. Instead of using the scene graph, they build a BVH directly over the scene objects. The leaves of the hierarchy again contain local data structures built for the scene objects and the base tree is updated only for parts that are dynamic.

Another approach is taken by Li et al. [LDG17]. They use Morton codes to uniformly divide space and construct a fully balanced binary tree. This tree resembles an octree, but exists only conceptually and is used to construct a k-d tree by collapsing nodes without primitives and minimising empty space. They call this step "path compression". By computing Morton codes for all primitives, they determine which nodes are essential (that is nodes whose Morton code is included in the list of primitive Morton codes) and run a thread for each essential node to compress the path up to a parent essential node. This allows the algorithm to run in parallel without the need to build the tree layer by layer.

4.4 Other relevant work

Research has also been carried out in areas related to k-d tree construction besides construction algorithms themselves. Tillmann et al. [TPK16] use an online auto-tuner library to dynamically search for build parameters, such as the cost of traversal or intersections, while the scene is raytraced each frame. They report an improvement of 1× to 2× depending on the scene, and a slower convergence for dynamic scenes.

Chapter 5

Building k-d trees for dynamic scenes

As mentioned in Chapter 1, dynamic scenes are much more difficult to raytrace than static scenes. For static scenes, the acceleration data structure has to be built only once when the scene gets loaded, and can even be pre-built and only loaded into memory together with the scene. This is because the data structure does not depend on the position or rotation of the camera. When we introduce dynamic objects that change their position, rotation or scale, essentially moving the primitives that define them, the acceleration data structure has to be rebuilt or somehow modified to reflect this change. The following sections will introduce different (not necessarily all) approaches to solving this problem.

5.1 Rebuilding the whole tree

The simplest option is to rebuild the entire k-d tree each frame. This approach can work for smaller scenes with mostly dynamic objects, but should perform the worst when the number of static primitives is many times larger than the number of dynamic primitives. Scenes that are mostly static are common for example in the video game industry, where most of the environment is static. The dynamic objects like the player, vehicles, interactable objects and so on consist of only a small fraction of primitives compared to the whole scene. This is also partly because static objects allow for a variety of precomputations that can be used at runtime for better performance, so most developers try to make most of the scenes static.

5.2 Building two separate trees

An approach that should lead to the best build time performance is precomputing the static tree (= tree containing static objects), and then rebuilding only the dynamic tree (= tree containing dynamic objects) [SBS03]. We can also build separate trees for each dynamic object. The main drawback is that when we render the scene, we have to traverse two (or all) trees instead of only one and combine the traversal results into one. This essentially means that we trade off render time performance for build time performance.

5.3 Two level trees

To improve on the previous method, we can define a data structure built over the existing data structures, as mentioned in Section 4.3. We rebuild the data structures for each dynamic object separately and update the higher level data structure. This allows us to combine the benefit of not needing to rebuild the static tree while also traversing only a single acceleration data structure, with the cost of updating the higher level data structure. This should improve build time performance without hindering render times too much. There is even another optimisation we can do for dynamic objects with rigid transformations. As the acceleration data structure built over them does not change, only moves, rotates and scales in space, we can use it without rebuilding it. The only change we need to make is to transform the ray we are tracing into the local space of the dynamic object.

5.4 Merging trees

Another approach is to build separate trees for each object and then merge them into one tree. This approach is studied by Eleftheriades [Ele10]. While this eliminates the need to traverse multiple acceleration data structures, it does not solve the problem of rebuilding the data structure for static primitives. That can be solved by merging the dynamic tree(s) into the static tree in a way that can be reversed, so it can be done again the next frame while only recomputing the dynamic tree(s). This approach is studied in Chapter 10.

Chapter 6

Specialised traversal algorithms

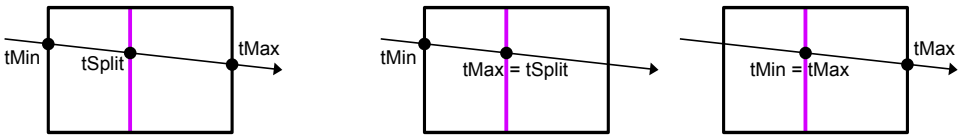
In this chapter, we present two different algorithms: the stack-less restart algorithm and the short-stack algorithm. We also present the push-down modification. All algorithms are presented in pseudo-code form with highlighted differences. The differences are always relative to the previous algorithm. The kd-restart algorithm is compared to Algorithm 1. The code is taken from [HSH07].

6.1 kd-restart

The restart algorithm (Algorithm 2, [FS05]) alleviates the need for a stack by cleverly using the tMin variable and simply "restarting" from the root node when a leaf node is reached. With the classic stack-based algorithm, when we reach a node whose both children need to be traversed (we will call this a *dual node*), we would push the farther one onto the stack. The node on the stack would be traversed later. With this algorithm, we ignore the farther node but take advantage of the fact that the condition to traverse the second, farther node (tSplit $\leq$ tMin) uses the variable tMin. After we reach a leaf, we "restart" the traversal instead of popping a node and set tMin to tMax. This ensures that when the same path is taken again and we reach the last dual node, the farther node will be traversed this time (note that tMin is now equal to tSplit, see Figure 6.1). The obvious drawback of this approach is that the tree needs to be traversed from the root up to a leaf as many times as there are dual nodes. This can significantly increase the time complexity and the number of uncached memory reads.

6.2 Push-down and short-stack

Push-down (Algorithm 3, [HSH07]) is a simple but logical extension of the restart algorithm. It builds on the idea that we do not always need to restart from the root. When we traverse the left or right child of the root without the need to traverse the other child later, we can "push down" the root and restart from it later. We can keep pushing it down until we reach a dual node. After restarting, the dual node that stopped the root from being pushed



(a) Inner node is being traversed.

(b) The closer node is traversed first. tMax is set to tSplit.

(c) After restarting, tMin is set to tMax and the second node is traversed.

Figure 6.1 Traversing a k-d tree using kd-restart

down will now be able to be pushed down, so we can repeat this each time we restart.

The short-stack algorithm (Algorithm 4, [HSH07]) is a hybrid between the stack and restart algorithms. It assumes that when using the restart algorithm, most time is lost when a dual node is located deep in the tree near the leaves. The long path needs to be traversed again after restarting only to check a neighbouring node, assuming the root cannot be pushed down too deep. To help with these cases, it uses a small stack with limited capacity. When the stack is not full or empty, it functions in the same way as the stack in the classic stack-based traversal. When it is full and a new item should be pushed to it, we pop the oldest item, throw it away, and push the new item to the top. When it is empty and an item should be popped from it, we restart instead of ending traversal. [HSH07] also integrate push-down into the traversal, but restarting without push-down can also be used.

Algorithm 2: Restart traversal

```
1  begin
2      ~~stack.push(root, sceneMin, sceneMax)~~
3      tMin = tMax = sceneMin
4      tHit = infinity
5      while tMax < sceneMax do
6          ~~(node, tMin, tMax) = stack.pop()~~
7          node = root
8          tMin = tMax
9          tMax = sceneMax
10         while not node.isLeaf() do
11             a = node.axis
12             tSplit = (node.value - ray.origin[a]) / ray.direction[a]
13             (first, second) = order(ray.direction[a], node.left, node.right)
14             if tMax ≤ tSplit then
15                 node = first
16             else if tSplit ≤ tMin then
17                 node = second
18             else
19                 node = first
20                 tMax = tSplit
21                 ~~stack.push(second, tSplit, tMax)~~
22             end
23         end
24         for p in node.primitives do
25             tHit = min(tHit, intersect(ray, p))
26         end
27         if tHit < tMax then
28             return tHit
29         end
30     end
31     return tHit
32 end
```

Algorithm 3: Push down traversal

```
1  begin
2      tMin = tMax = sceneMin
3      tHit = infinity
4      while tMax < sceneMax do
5          node = root
6          tMin = tMax
7          tMax = sceneMax
8          pushdown = true
9          while not node.isLeaf() do
10             a = node.axis
11             tSplit = (node.value - ray.origin[a]) / ray.direction[a]
12             (first, second) = order(ray.direction[a], node.left, node.right)
13             if tMax ≤ tSplit then
14                 node = first
15             else if tSplit ≤ tMin then
16                 node = second
17             else
18                 node = first
19                 tMax = tSplit
20                 pushdown = false
21             end
22             if pushdown then
23                 root = node
24             end
25         end
26         for p in node.primitives do
27             tHit = min(tHit, intersect(ray, p))
28         end
29         if tHit < tMax then
30             return tHit
31         end
32     end
33     return tHit
34 end
```

Algorithm 4: Short stack traversal

```
1  begin
2      tMin = tMax = sceneMin
3      tHit = infinity
4      while tMax < sceneMax do
5          if stack.empty() then
6              node = root
7              tMin = tMax
8              tMax = sceneMax
9              pushdown = true
10         else
11             (node, tMin, tMax) = stack.pop()
12             pushdown = false
13         end
14         while not node.isLeaf() do
15             a = node.axis
16             tSplit = (node.value - ray.origin[a]) / ray.direction[a]
17             (first, second) = order(ray.direction[a], node.left, node.right)
18             if tMax ≤ tSplit then
19                 node = first
20             else if tSplit ≤ tMin then
21                 node = second
22             else
23                 node = first
24                 tMax = tSplit
25                 pushdown = false
26                 stack.push(second, tSplit, tMax)
27             end
28             if pushdown then
29                 root = node
30             end
31         end
32         for p in node.primitives do
33             tHit = min(tHit, intersect(ray, p))
34         end
35         if tHit < tMax then
36             return tHit
37         end
38     end
39     return tHit
40 end
```

Chapter 7

Analysis

In this chapter, we analyse the building algorithms introduced in previous chapters and provide reasoning behind which ones were chosen. We also introduce the design philosophy behind the base application and present the requirements.

7.1 Algorithm comparison and selection

When choosing an algorithm, we have to consider two things: how fast the building algorithm is and the quality of the tree it produces. The quality of the tree directly affects the ray tracing performance. Usually, the easier and faster it is to build the tree, the worse quality it has. The speed of the building algorithm increases in importance as the size of the scene increases. The quality of the tree increases in importance as more rays per second are traced per frame. Modern applications usually have large scenes and also use algorithms that shoot many rays, requiring balance between tree build times and their quality.

Our goal was to design an algorithm that:

- could be parallelised
- ran with minimal or ideally no CPU synchronisation
- produced good quality k-d trees

Due to the complexity of the GPU implementation, we decided to implement only one building algorithm.

Algorithms that do not use the SAH usually suffer from worse quality. [ZHW08] use spatial median splitting in the large node stage and require CPU synchronisation. [LDG17] essentially also use spatial median splitting, although their fast build times might accommodate for the worse quality, making it an interesting candidate for comparison with algorithms with slower build times but better output quality.

Because a part of this work focuses on building k-d trees and then merging them into a single data structure, multi-layered data structures were considered only for performance comparison. Algorithms using exact split

selection with events were considered and would be the base for a second algorithm design.

For the one GPU building algorithm we implemented, we decided to use min-max binning like [SSK07] and [DPS10], as binning is suitable for parallelisation and the quality of the k-d tree it produces is not much worse than using exact split selection.

From what we understand, none of the solutions described in Chapter 4 run entirely on the GPU, mainly because of dynamic memory allocation. In the following section, we describe the idea of a task pool, which is the solution we used.

7.2 Task pool on the GPU

As mentioned in Chapter 4, we face several issues when parallelising the construction algorithm. With the DFS approach, we must handle the initial decomposition when processing the top layers. With the BFS approach, we must handle the layer-by-layer synchronisation and load balancing when some subtrees end much earlier than others. We must also take into account dynamic memory allocation. One way to solve these problems is to use a task pool, as suggested by Vinkler [Vin14]. It allows the entire implementation to run on the GPU and solves parallelisation of processing both the top and bottom layers.

The task pool consists of a list of tasks, where each task represents some work to be carried out. The amount of work is represented by an atomic integer that corresponds to the number of *chunks* that needs to be processed before the task is completed. Chunks are processed by subgroups, and each subgroup can take multiple chunks to process at once (see Figure 9.2). When working on the top layers, it allows subgroups to cooperate on processing a single large node represented by one task with many chunks. As the depth of the tree increases, so does the number of tasks, but the number of chunks in each task decreases. This naturally distributes the work evenly between subgroups, until each subgroup will essentially work independently on its own task. Assuming there are no chunks that would require significantly more processing than others, there are no load-balancing issues because the task pool is shared among all subgroups and subgroups can work on any task in the pool.

The main issue is that to synchronise the whole kernel, most of the work must be done in the slowest memory layer, the global memory. Most algorithms that optimally utilise the GPU work in shared memory whenever possible, which is hard to achieve with the task pool approach. An issue the task pool does not solve is dynamic allocation. We also cannot predict the required size of the task pool and not increasing the size of the task pool when it is full could lead to a deadlock, so it in itself requires dynamic allocation. The deadlock would happen after the task pool fills up, and each subgroup is waiting for an empty slot to store a newly created task. The full design and implementation details will be given in Section 9.3.

7.3 GPGPU language selection

There are several language standards for implementing code that runs on the GPU. The oldest is the shader language. Originally used for vertex, fragment and other shaders that are part of the rasterisation pipeline, it has recently added compute shaders, which are shaders designed for GPGPU and run independently using the same libraries that define the other shaders (OpenGL [Wol18], Vulkan (vulkan.org), etc.). Another language standard is OpenCL [MGM11], developed by the Khronos Group. It focuses on general parallel computations and its code can also be run on other hardware besides GPUs. CUDA [Ans22] is a language developed by Nvidia and it uses the proprietary Nvidia CUDA compiler, so it is limited to Nvidia graphics cards. Another standard developed by the Khronos Group is SYCL (khronos.org/sycl), a more modern version of OpenCL based on C++17 (where OpenCL is based on C).

We decided to use CUDA because it uses C++, has many features including support for using some C++ classes from the STL in device code, and mainly because it is the only GPGPU language we found to have dynamic memory allocation from device code. The biggest disadvantage of CUDA is that it cannot run on GPUs from other companies like AMD, at least not directly (see [Adv]).

7.4 Software design philosophy

The application has two objectives. The primary objective is to provide an environment for implementing the chosen algorithms, testing them, and measuring their performance. The secondary objective is to create a reusable implementation that could serve as a basis for implementing and testing other algorithms connected to ray tracing, complete with support for visual debugging and an interface hierarchy that would allow for injecting code wherever would be needed.

7.5 Functional requirements

- The application shall be able to load and display a scene defined in a structured file.
- The scene specification shall allow for static, dynamic and animated meshes.
- The application shall be able to render the scene using ray tracing.
- The application shall have an interactive mode with a user controlled camera used for viewing the scene and testing functionality.

7.6 Non-functional requirements

- The application shall be responsive and run at a stable frame-rate while not ray tracing.
- The code shall be structured such that implementing new ray tracing algorithms and/or acceleration data structures requires minimal changes in existing code.

Chapter 8

Ray tracing application framework design and implementation

To simulate an application we are familiar with and allow for further extending the functionality, we decided to design the application as a simple game engine with a scene hierarchy containing *game objects* housing *components*. The main application class is called RTApp. It is a singleton class and contains methods for running the main application loop, which is entered after processing command line arguments. The main application loop is described in Algorithm 5. The information described in this chapter is hand-picked and does not necessarily contain all the information about the implementation. All details are available in the documentation and comments in the implementation.

Algorithm 5: Main application loop

```
1  begin
2      while not ShouldExit() do
           // Switch scene if requested
3          SwitchScene()
           // Call all pre-render methods
4          PreRenderUpdate()
           // Update UI
5          UIUpdate()
           // Render the scene, either with OpenGL or the current
              raytracer
6          Render()
           // Call all post-render methods
7          PostRenderUpdate()
           // Render UI elements
8          RenderUI()
           // Swap window buffers to display rendered scene
9          SwapBuffers()
           // Process input events
10         PollEvents()
11     end
12 end
```

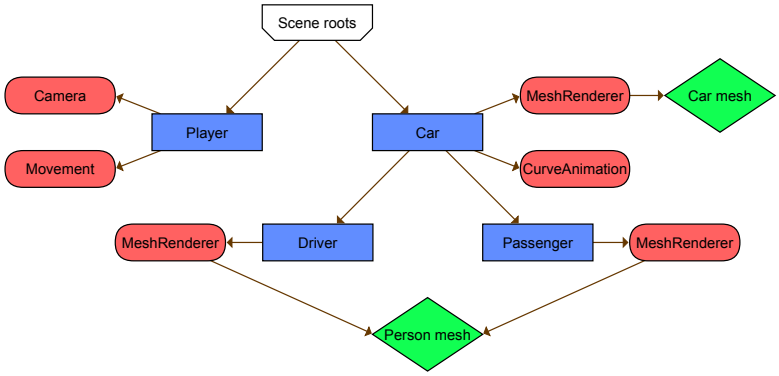


Figure 8.1 An example scene graph with game objects (blue rectangles), components (red ovals), and models (green diamonds). Transform components are not displayed (every GO has one).

The code is compiled with the help of CMake with the goal of being compatible with multiple platforms. External libraries that are not included in the source code itself are included using CMake's `find_package` function, which uses the currently available package manager to retrieve them. A list of used libraries can be found in Section 8.9.

8.1 Scene definition

The scene is represented as a tree graph, where the nodes of the graph are called *game objects* (= GO). Game objects are simply containers designed to hold a list of components, which give them functionality. Each GO has a mandatory Transform component, which gives it a position, rotation and scale in the scene. They also have a list of child GOs, that create the scene hierarchy. To be precise, the scene is a forest and not a tree in reality, as there is no root game object, instead there is a list of root game objects (see Figure 8.1). This design was modelled after the Unity game engine (unity.com).

We have two options if we want to create our scenes. We can either implement a scene editor, similar to ones that already exist as parts of other popular game engines, or define a human-readable and editable text format. Both options mean that we have to be able to serialize the scene. Due to the complexity of implementing our own scene editor, we chose to serialize the scenes in a simple JSON format, which allows for both hierarchies (the scene graph) and lists of objects (components and GO children).

8.2 Component system

A game object without components is just an empty scene graph node. The mandatory Transform component gives each game object a position in space. Each transform has its local position, rotation and scale, which are

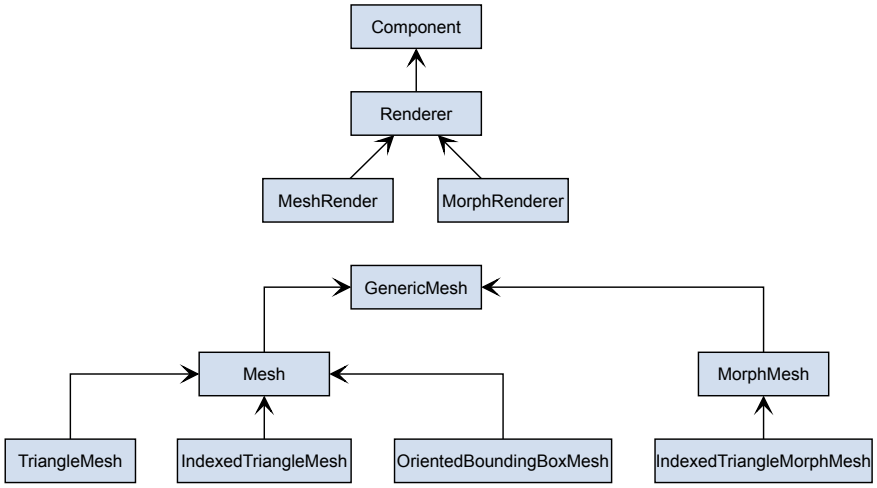


Figure 8.2 Hierarchy of the renderer components and mesh classes. Arrows represent inheritance.

relative to the parent game object's transform. This creates a hierarchy of transformations, where multiple game objects can move by moving a single common parent, which does not change their local positions, but changes their world positions. World position (and rotation and scale) is the position relative to the root of the scene and it is the position we perceive. For example, by moving the Car GO in Figure 8.1, that is changing its local position, we would also be moving the driver and the passenger. Their local positions would not change, but their world position would.

Components allow for the functionality of game objects to be implemented as composition over inheritance [Wika]. Each component can load its properties from the scene JSON file on scene load. On certain events, such as the scene update that happens every frame, after rendering the scene, etc., a scene updater calls the relevant virtual method for each component. The components then execute their functionality inside these methods. For example, an animator component would update its animation time based on the time since the last frame, or a movement component would move the transform of the GO it is attached to based on user input.

We have implemented several components needed for running the application and also others to implement relevant application requirements. The complete list of components and their descriptions can be found in the attached source code.

8.3 Models

The main feature of our application is displaying models. To do that, we first define the Asset class and the AssetID. Asset represents anything that can be loaded from the disk using an AssetID (for example a mesh, texture, etc.), which is currently represented as a path relative to the Assets folder that must be located next to the executable. Models consist of meshes and

materials. Meshes consist of lists of primitives, usually triangles. Because ray tracing allows to easily incorporate other types of primitives like spheres, we have prepared a system that allows for other types of primitives, as long they have implemented their mesh class and methods needed for ray tracing, such as intersecting with a ray. Materials contain properties used when shading and determining the colour of the mesh surface. Each submesh corresponds to a material.

To display the loaded meshes, we define a MeshRenderer component that holds a shared pointer to the Mesh class and a list of submeshes it is supposed to display. The subdivision into submeshes allows for a single mesh to have multiple materials and hold multiple objects, each rendered with a different MeshRenderer with possibly different location in the scene hierarchy (see Figure 8.1 and Figure 8.2).

8.4 Animations

Our focus is on dynamic scenes, which differ from static scenes by having animations in them. We have implemented two types of animations. The first type is transform animation, where we change the position, rotation, and scale of the object. This type of animation comes naturally after defining a scene hierarchy with transforms and is implemented by changing the game object's or its parent's position and/or rotation each frame using a custom component.

The second type of animation has different names, but we will refer to it as morph (target) animation [Wikb]. With this type of animation, each primitive in a mesh can change their shape (without changing into another type of primitive). It is defined as a set of key frames, which are meshes with primitives in different positions, and the animation is created by linearly interpolating between these frames. To enable this type of animation, we define a MorphRenderer component that holds a shared pointer to a MorphMesh class, similarly to MeshRenderer and Mesh. The MeshRenderer and MorphRenderer components both inherit from the Renderer component, and the Mesh and MorphMesh classes both inherit from the GenericMesh class. The Renderer and GenericMesh classes are purely virtual and hold common methods of their children (see Figure 8.2).

Each morph animation asset consists of a text file with .animation extension containing a list of mesh file names (one per line) that represent the key frames. The animation is controlled by the MorphController component, to which each MorphRenderer holds a reference. This allows multiple renderers to have the same animation time. The controller has an animation length property, which defines how fast the animation will play. Key frames are assumed to spaced equidistantly in time and the animation is looped.

8.5 Rendering

The application has two rendering modes, OpenGL mode and Raytracing mode. The first mode is meant for displaying the scenes (even without functional ray tracers) and debugging and visualising built acceleration data structures. It displays the scene using standard rasterisation with OpenGL.

The second mode is meant for testing ray tracing performance and should be used after implementing a functional ray tracer. For automating performance testing, it can also be run together with no-gui and stop-frame switches. The no-gui switch disables any OpenGL calls and does not open any application windows. The stop-frame switch with an integer argument stops the application after running for the specified number of frames.

The rendering is handled by two renderer instances located in the RTApp class. The first instance is an instance of Rasterizer class, which handles rendering using OpenGL. It is created only when in OpenGL mode. The second instance is an instance of any class implementing the IRaytracer interface, which handles rendering by ray tracing the scene and gets created in both modes. IRaytracer has the following pure virtual methods (not all methods are listed):

- **Init** - Called once after loading the first scene. This method exists in place for a constructor, because the ray tracer gets constructed before loading the scene and other things, which might be needed for initialisation.
- **BuildStatic** - Called once every time a new scene gets loaded, used for building acceleration data structures for static objects.
- **BuildDynamic** - Called every frame before rendering. Used for building acceleration data structures for dynamic objects.
- **RaytraceScene** - Called after every call to BuildDynamic (not necessarily immediately after). Used for rendering the scene.
- **GetRender/GetRenderTexture** - Returns the rendered image.

The IRaytracer interface allows for implementing custom ray tracers, but for standard acceleration data structures that do not need special handling, we have implemented StructureRaytracer. It implements most of the IRaytracer methods including the ray tracing algorithm. It only needs a class that implements the IAccelerationStructure interface, which it uses to intersect rays with the scene. IAccelerationStructure has the following pure virtual methods (not all methods are listed):

- **Init** - Same as IRaytracer.
- **BuildStatic** - Same as IRaytracer, but takes a list of static renderers to build the data structure over.

- **BuildDynamic** - Same as IRaytracer, but takes a list of dynamic active renderers to build the data structure over.
- **Traverse** - Takes a ray and its length, outputs an empty optional if the ray does not hit any primitives, or an optional with a structure with info about the closest hit primitive.
- **TraverseShadow** - Takes a ray and its length, outputs true if the ray hits any primitive, false otherwise.

For ray tracing on the GPU using k-d trees, we have implemented the KDTGPURaytracer class.

8.6 Frame tracer

There is also a special class, FrameTracer, that allows ray tracers to be run during OpenGL mode. It has a UI window with its controls displayed automatically in OpenGL mode, while being disabled in Raytracing mode. To use it, the user must first pause the scene to pause animations. When the scene is paused, only components with AllowDuringSnapshot set to true are updated, and three buttons are enabled - Build static, Build dynamic and Raytrace (see Figure 8.3). These buttons correspond to calling the similarly named methods of the current IRaytracer.

8.7 Visualisation and validation

Each implementation of IRaytracer/IAccelerationStructure can implement a navigator (INavigator) and a validator (IStructureValidator). The navigator is used by the StructureNavigator component to navigate and visualise the current data structure, which is useful for seeing what the algorithm does in the context of 3D space and can help find issues with the data structure that would be hard to see just from data. The component is accompanied by a UI window with navigation controls (see Figure 8.4). The buttons again call the similarly named methods of the navigator of the current IRaytracer. If a ray tracer does not have a navigator implemented, the buttons are disabled. An implementation of INavigator can add its own navigation UI to the main navigation window. The INavigator has the following virtual methods:

- **StartNavigating** - Initialises or resets navigation.
- **Navigate** - Renders the data structure specific navigation UI and processes input.
- **CanNavigateInto** - Returns true if this navigator has another internal navigator and it is possible to navigate into it, for example navigating inside a tree node.

- **NavigateInto** - Returns the pointer to an allocated and initialised internal navigator.
- **NavigateOut** - Called when navigating out
- **Render** - Renders the visualisation.

Validators are used by the StructureValidator component to test if the currently built data structure is built correctly. The component displays a UI window with a button that calls the Validate method, which returns an optional string (see Figure 8.4). If the optional value is empty, the validation is treated as successful, otherwise the string is treated as the error message and is displayed in the UI window next to the Validate button.

8.8 Scene configuration

To allow changing some parameters without recompiling the program and ease testing automation, we have implemented a simple configuration (=config) scheme. Each scene can have a config file, defined as a file next to the scene file with the same stem but with a .config extension instead of .scene extension. The config file is a JSON and is loaded together with the scene. Config values can be accessed using two different methods with a string key. The GetValue method takes an additional argument, which serves as the default value if the key is not found in the config. The TryGetValue method takes an additional reference argument and if the key is found, it writes the requested value to the reference and returns true. Otherwise, it only returns false. For values that are needed before a scene is loaded or do not depend on the scene, an additional app config file can be passed as a command line argument which is used when a value is not found in a scene config.

8.9 External libraries

In this section, we list all external C++ libraries we used, provide a link to their homepage, and briefly describe what they were used for.

- Assimp (assimp.org) - Used for loading meshes.
- stb (github.com/nothings/stb) - Used for saving rendered images to disk.
- OpenGL (opengl.org), glfw (glfw.org), vkfw (github.com/Cvelth/vkfw), glbinding (glbinding.org) - Used for the graphics.
- nlohmann/json (json.nlohmann.me) - Used for loading JSONs.
- cxxopts (github.com/jarro2783/cxxopts) - Used for parsing cmd line arguments.
- Dear ImGui (github.com/ocornut/imgui) - Used for UI.

- CUDA (developer.nvidia.com/cuda-toolkit) - Used for GPGPU programming,
- CMake (cmake.org) - Used for project compilation.

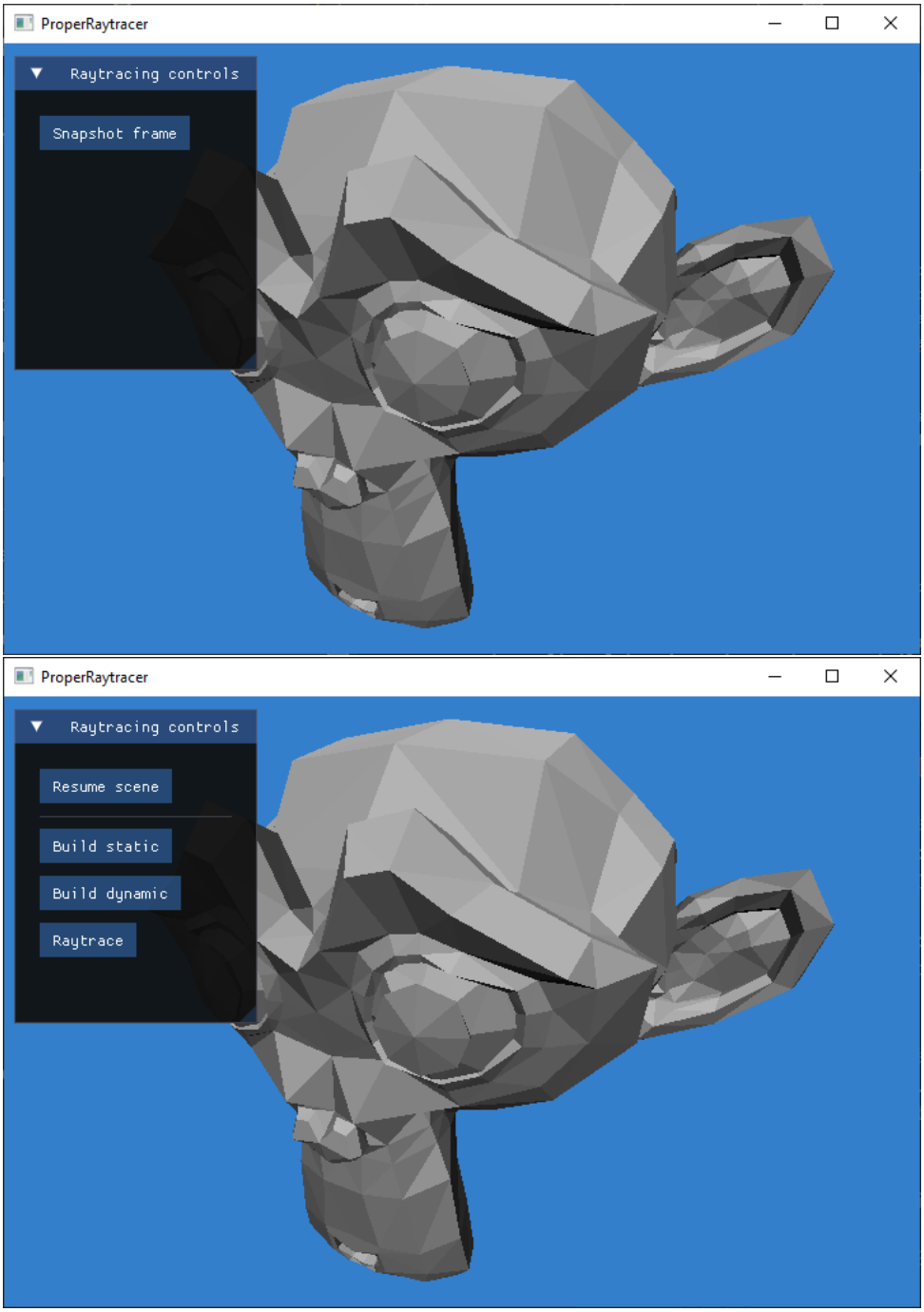


Figure 8.3 Application screenshots with frame tracer UI window (unpaused and paused)

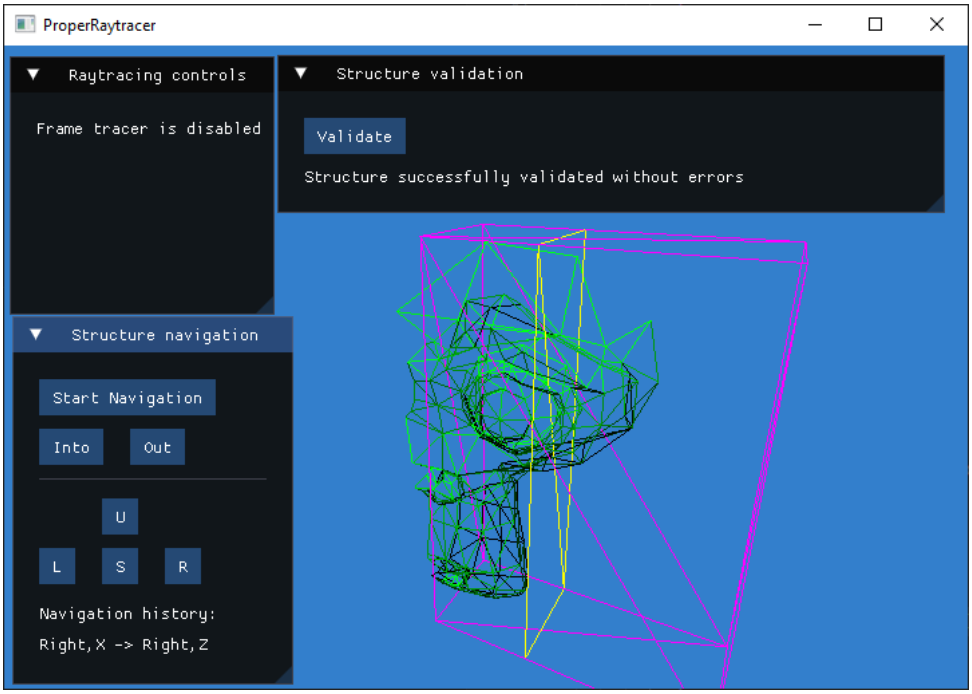


Figure 8.4 Application screenshot with structure navigator and validator UI windows. The currently built k-d tree was successfully validated and is being navigated. The node bounding box is coloured pink, the splitting plane yellow, and the model in green. Because the application does not support transparency, everything is rendered in wireframe mode.

Chapter 9

k-d tree building algorithm design and implementation

Our work consists of three implementations of k-d tree building algorithms: two reference CPU implementations for correctness testing purposes and one GPU implementation of the proposed min-max binning k-d tree building algorithm. All implementations use the same data structures for k-d tree nodes (see Figure 9.1).

```
enum class NodeType {
    InnerX = 0, InnerY, InnerZ, Leaf
};
struct KDTInnnerNode
{
    NodeType tag;
    float splitValue;
    int32 leftChildIdx;
    int32 rightChildIdx;
};
struct KDTLeafNode
{
    NodeType tag;
    int32 primitiveN;
    PUID* primitives;
};
struct Node
{
    union {
        KDTInnnerNode inner;
        KDTLeafNode leaf;
    };
};
```

Figure 9.1 Structures used in the kd-tree implementation in C++ language

9.1 Primitive references

The k-d tree leaf nodes contain a list of references to primitives. The reference is a data structure called PUID, or Primitive Unique IDentifier. It consists of two integers, an instance index and the offset of the primitive inside the instance. Instances are essentially references to submeshes. On the CPU, they consist of a pointer to a Renderer and the index of the submesh they reference. On the GPU, they consist of a pointer to a list of primitives of the submesh they reference, the transformation matrix, and other information about the submesh, such as its material.

To build a k-d tree, we need a mesh manager and a builder. The mesh manager creates and manages instances, and PUIDs are unique to the mesh manager that created them. The builder takes instances from a mesh manager and builds the tree over them. This structure allows us to have a single mesh manager implementation with multiple different builder implementations inheriting from a single abstract builder. The specific builder can then be selected at runtime based on user parameters.

9.2 Reference CPU implementations

We have implemented two reference solutions on the CPU. One uses min-max binning and is used to check the correctness of the GPU implementation. The second implementation uses events to select the perfect split and can be used to compare against the use of approximate split selection.

When a primitive is parallel to one of the three main axes, it causes its bounding box to have zero volume and it generates a start and end events at the same coordinate on one axis. This special case (that happens often in practice, for example in walls and floors) has to be handled separately or can be solved by slightly expanding the bounding box of planar primitives by a small ϵ. The expansion does not cause any issues, the only thing it can cause is a primitive being included in a leaf where it does not belong, similar to what happens when not split clipping. We chose to expand the bounding boxes. To avoid creating nodes with zero volume, we also forbid splitting planes to be placed at the borders of the bounding boxes of nodes.

For both the CPU and GPU implementations, we use similar termination criteria and parameter values as [VHB16]. The algorithms terminate when (a) the maximum depth d_{max} is reached, (b) the number of primitives in the leaf is less then n_{max}, or (c) the cost of making the node a leaf is less then splitting at the perfect split. We limit the maximum depth with $d_{max} = k_1 + k_2 \log_2(N)$, $k_1 = 2.0, k_2 = 1.2, N =$ number of primitives. n_{max} is based on the number of primitives in the scene, 2 for small scenes (less than 700 thousand primitives), 8 for medium scenes (around million triangles), 12 for large scenes (more than 3 million triangles).

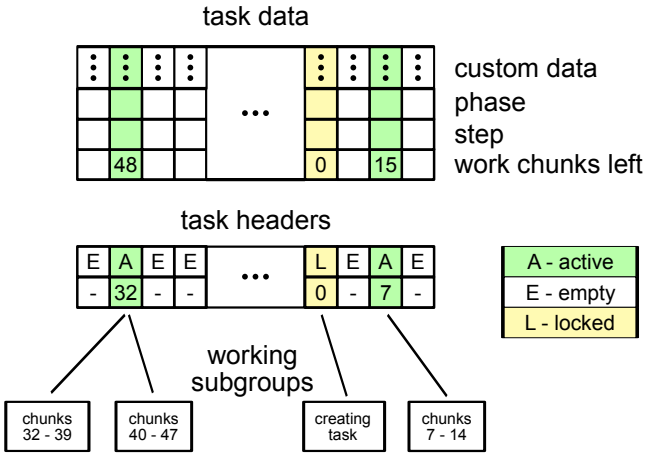


Figure 9.2 Task pool structure example

9.3 Task pool

As mentioned in Section 7.2, the task pool consists of a list of tasks and a list of atomic integers called the headers. A header can have one of three distinct types of value - empty, locked, and active. We adopt the same values as [Vin14], that is:

- **Locked**, value = 0: Either means that a task is being prepared or that the task's chunks have all been retrieved.
- **Active**, value > 0: The positive number represents the number of chunks that have yet to be retrieved.
- **Empty**, value = INT_MIN: Theoretically signified by any negative value. In practice, we may have a scenario where multiple subgroups read a positive header (for example equal to one) and they all atomically subtract some number, which can take the header quite deep below zero. Of course, only one subgroup will retrieve and process the chunk, but to avoid needing to set the header to zero afterwards, we ignore as many negative values (basically leaving them as undefined values) as possible and set the value that signifies the task is empty as the largest negative representable number.

The task itself consists of the task phase, task step, number of chunks left and algorithm specific data. The concept is that each task can have multiple task phases, and each task phase can have multiple task steps. The task steps serve as a synchronisation primitive for a specific task phase, as shown in the next section. The atomic number of chunks left tells us how many subgroups have finished processing their chunks, which is needed for post-processing (explained later). This is different from the header, which tells us how many chunks have yet to be retrieved and start being processed. Intuitively, these

Algorithm 6: Main task pool loop

```
1  begin
2      while RetrieveTask(task, C′, R′) do
3          ProcessTask(task, C′, R′))
4          if laneIdx == 0 then
5              task.workChunksLeft -= R′
6          end
7          if task.workChunksLeft == 0 then
8              TaskPostProcess(task)
9          end
10     end
11 end
```

numbers will be the same when the task is created, and the header will always be less than or equal to the number of chunks left. The task pool structure is illustrated in Figure 9.2.

The general loop for working with the task pool is described in Algorithm 6. The loop is expected to be executed by a subgroup, but can also work for groups with some modifications to the called functions. First, the subgroup retrieves R' chunks from a single task. The chunks are then processed. Finally, R' is subtracted from the remaining number of chunks needed to complete the task step or phase. If the subgroup is the last subgroup working on the task, that is, the remaining number of chunks is zero after subtraction, it does some additional processing we call post-processing.

The retrieval algorithm is described in Algorithm 7. We need to search the whole task pool for a task that can be retrieved, which happens in a loop. In one step of the loop, each lane in the subgroup loads its own value from the header. We then find the lane that has loaded a positive value and atomically subtract R. If the value C equal to the header value at the time of subtraction (returned by the atomic subtraction) is still positive, we have successfully retrieved R' chunks to process starting at chunk C', where $R' := R, C' := C - R$ if $(C - R) >= 0$, otherwise $R' := C, C' := 0$. To distribute the work equally, we define the size of the pool as a multiple of the total number of work threads and make each subgroup start the search at an offset based on the current number of active tasks in the pool and the index of the subgroup (see Algorithm 7). When the task pool is full, each subgroup starts at a different offset, not interfering with each other. When it is mostly empty, each subgroup will start the search at the front of the task pool, where the few tasks will be located.

To define R, [Vin14] suggest a value proportional to the number of chunks left in the task. Instead, we use a fixed value to save time on additional computations. When creating a new task or updating an existing one, we can skip the retrieval algorithm, and keep working on the task we updated/created. When the number of chunks is less than or equal to (some multiple of) R, we can also skip updating the task in global memory, if the subgroup has a local copy of it. Skipping retrieval saves a lot of time, because even if we

Algorithm 7: Task retrieval

```
1  begin
2      failedOnce = false
3      while task not retrieved do
4          if workingSubgroups == 0 then
5              return False
6          end
7          startIdx = (activeTasks / subgroupN) · subgroupIdx
8          startIdx = startIdx - (startIdx % subgroupSize)
9          startLane = subgroupIdx % subgroupSize
10         poolIdx = startIdx
11         repeat
12             taskIdx = poolIdx + laneIdx
13             C = atomicGet(taskHeaders[taskIdx])
14             if C > 0 then
15                 for i = 0 to subgroupSize do
16                     lane = (i + startLane) % subgroupSize
17                     if lane == laneIdx and C > 0 then
18                         C' = atomicSub(taskHeaders[taskIdx], R)
19                         if C' + R > 0 then
20                             R' = R
21                             if C' < 0 then
22                                 R' = C
23                                 C' = 0
24                             end
25                             task = tasks[taskIdx]
26                             if failedOnce then
27                                 atomicAdd(workingSubgroups, 1)
28                             end
29                             return True
30                         end
31                     end
32                 end
33             end
34             poolIdx = (poolIdx + subgroupSize) % taskPoolSize
35         until poolIdx != startIdx
36         if not failedOnce then
37             failedOnce = True
38             if laneIdx == 0 then
39                 atomicSub(workingSubgroups, 1)
40             end
41         end
42     end
43 end
```

retrieved a new task immediately, at least $w + 1$ atomic operations would be performed, with several additional computations and global memory accesses (w = subgroup size).

Processing depends on the algorithm that utilises the task pool. However, it should not modify non-atomic task variables and should generally do work that is local to the work chunk it is processing. The work it should do, for example in an array, can be computed from work chunk indices C' to $C' + R'$ obtained during task retrieval.

Post-processing is always performed only once by a single subgroup for each task after all other subgroups working on the task have finished. This allows the post-process function to change any variables associated with the task without atomic operations. Usual post-processing consists of changing the task phase or task step, finishing the task and unlocking the slot, or creating new tasks.

The task pool can be used for multiple algorithms. We group the common functionality into the GPUTaskPoolProcessor class and the ProcessTasks kernel. The ProcessTasks kernel implements the basic loop from Algorithm 6. Its argument is an instance of the processor class, on which it calls RetrieveTask, ProcessTask, and TaskPostProcess. Because we are transferring instances between the CPU and GPU, we did not want to use virtual methods. Fortunately, CUDA supports templates, so we used a type of static compile-time polymorphism called CRTP [Cop95] instead.

Our class also provides several other utility methods for working with the task pool, including a method for locking a task slot. Locking a task works in a similar manner to the retrieval algorithm, just instead of looking for an active task and subtracting chunks to be processed, we look for empty slots and change their header to the locked state.

9.4 Parallel scan using a task pool

A scan, also called a prefix sum [Ngu07], is an operation where an input array $[x_0, x_1, ..., x_n]$ is transformed such that every element equals the sum of its previous elements, or $[x_0, \sum_{j=0}^{1} x_j, ..., \sum_{j=0}^{i} x_j, ..., \sum_{j=0}^{n} x_j]$. If the element at position i is included, it is an inclusive sum/scan, if not, it is called an exclusive sum/scan. We can get the exclusive scan from the inclusive scan by shifting all elements of the inclusive scan right and vice versa, supplementing the first element with zero when shifting right and last element with the total sum when shifting left. We can also get the reverse scan by subtracting the elements of a forward scan of the other type from the total sum. The operation can also be substituted for any other binary associative operation, for example multiplication or maximum/minimum.

The scan operation is extremely useful in parallel applications when distributing array elements. When selecting elements from an array and creating a new array from them sequentially, we can simply carry the index of the last inserted element and insert the next element at the next index. When we try to do this in parallel, we realise that we do not know to which index the

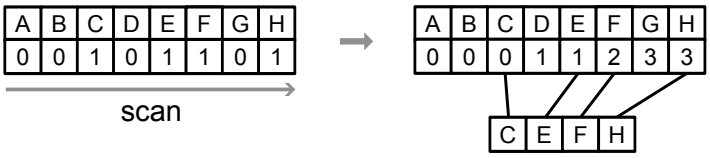


Figure 9.3 Scanned filter array used as indices for inserting filtered elements

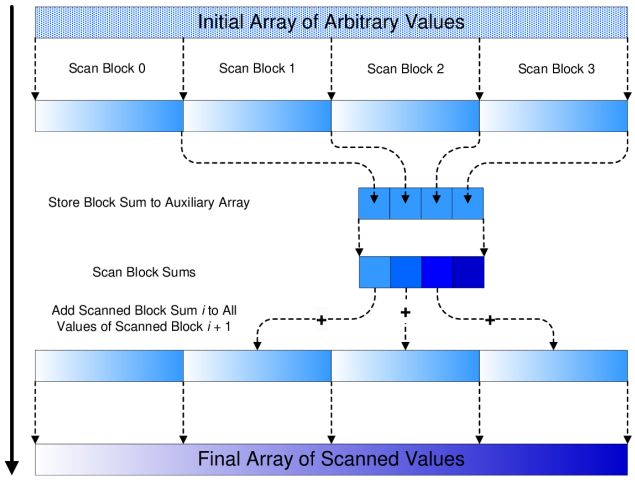


Figure 9.4 Parallel scan for larger arrays, taken from [Ngu07]

current thread should write, as it does not know how many elements where selected before it. This problem can be solved by creating an array of binary values, where zero means the element is not selected and one means it is, then doing an exclusive scan and using the result as the insertion index (see Figure 9.3). The scan also tells us the total number of selected elements so that we can allocate an appropriately sized array. All of these steps can be done in parallel. Because we need the scan when building the tree, we need to implement it using a task pool. Let us first talk about how it is done without a task pool.

First, we launch enough groups to process the whole input array. Each thread group loads its assigned memory block into shared memory, where it performs a parallel scan on it, using either the Hillis and Steele algorithm or the Blelloch algorithm (both described in [Ngu07]). The scanned elements are then copied back into the global array. When the total number of elements does not exceed the number of elements one group can process, the algorithm is finished. If it does, we have to save the sum of each memory block into an intermediate array and end the kernel, as we do not have a synchronisation primitive for the whole kernel (or it is unnecessarily expensive). We then run the same process over the intermediate array. If it can be processed by a single group, we do not end the algorithm, but instead distribute the scanned sums from the intermediate array into the global array, where what was previously the sum of memory block 1 gets added to all elements of memory block 2, sum of block 2 gets added to elements of block 3 and so on.

If the intermediate array cannot be processed by a single group, we repeat the same process, saving the group sums into a second intermediate array with the first intermediate array acting as the global array (see Figure 9.4).

By using the task steps, we have the advantage of being able to synchronise the whole kernel, so we do not have to split the computation, no matter how big the input array is. A big disadvantage, though, is that we cannot sensibly use shared memory for the computation, because we do not know which subgroups from which groups will be working on the task. We also have to be careful about the intermediate result arrays, because we must use dynamic memory allocation for them, which is expensive. We have three options:

(a) Allocate the next intermediate result array after processing a task step.

(b) Precompute the required number and allocate memory for all intermediate arrays together with the input array.

(c) Use the input array to save the intermediate results.

Option (a) is the easiest, but most inefficient. Both options (b) and (c) reduce the number of required allocations to a constant one. Option (b) should have better performance due to the elements being processed always being next to each other in memory resulting in better cache performance. Option (c) trades the performance benefit for requiring less memory.

We decided to use option (c). The task pool computation consists of two task phases, the up-sweep and the down-sweep, corresponding to the phases of the standard parallel scan algorithm. In the following explanation of the task phases, we provide the formulas for these variables:

- The number of task steps n_{steps} for the phase, based on the total number of elements N and subgroup size w
- The stride $stride$, based on the current task step s
- The number of chunks n_{chunks} for each step
- The index into the global array gid for each lane in the subgroup, based on the chunk index it is processing c and index of the lane $laneIdx$

9.4.1 Up-sweep

$$
\begin{aligned}
n_{steps} &= \log_{2^{\log_2(w)}}(N) \\
stride &= w^s \\
n_{chunks} &= \left\lceil \frac{N}{w^{s+1}} \right\rceil \\
gid &= stride - 1 + chunk \cdot stride \cdot w + laneIdx \cdot stride
\end{aligned}
$$

In the up-sweep phase, each subgroup processes w elements with increasing stride. Each lane loads an element based on gid, then the whole subgroup performs a scan over the elements and saves them back into the array (see Figure 9.5).

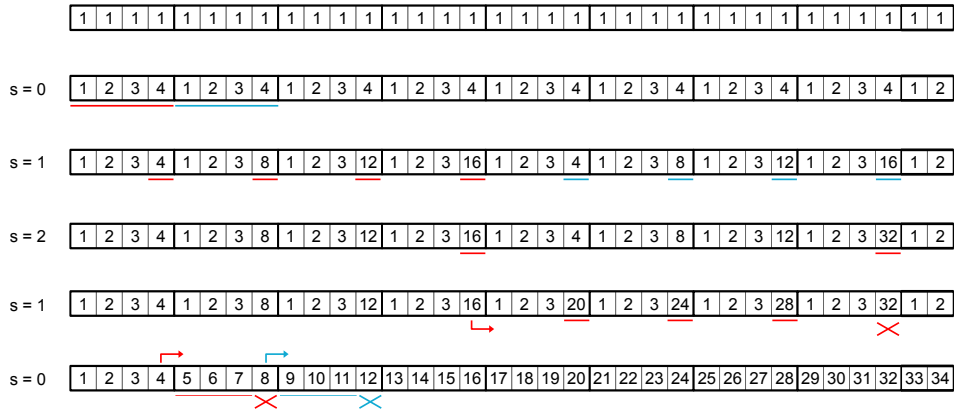


Figure 9.5 Visualisation of the parallel scan using the task pool for $w = 4, N = 34$. The first three task steps are part of the up-sweep phase, the last two are part of the down-sweep phase. The red underlining highlights elements that belong to the first chunk in task step s, blue highlights elements of the second chunk.

9.4.2 Down-sweep

$$
\begin{aligned}
n_{steps} &= \log_{2^{log_2(w)}}(N) - 1 \\
stride &= w^s \\
n_{chunks} &= \left\lceil \frac{N}{w^{s+1}} \right\rceil \\
gid &= stride - 1 + (chunk + 1) \cdot stride \cdot w + laneIdx \cdot stride
\end{aligned}
$$

In the down-sweep phase, the task steps and stride decrease instead of increasing, and the task step starts at $n_{steps} - 1$ instead of 0. Each chunk corresponds to the same w elements from the up-sweep phase for the same task step, just shifted $w \cdot stride$ elements to the right. Each lane except the last one reads its corresponding element and adds the sum of the previous chunk to it. The sum is saved in the last element of the original w elements without the shift. The element corresponding to the last lane is the sum of the current chunk used by the next chunk, computed by the previous task steps. This process distributes the chunk sums and completes the scan (see Figure 9.5).

9.5 GPU binning algorithm

In this section, we describe the GPU k-d tree building algorithm we implemented that uses min-max binning for split selection. The whole building algorithm runs by using the task pool, thus we will describe the algorithm in terms of its task phases. Except for the scan phases and the make leaf phase, the number of task steps in each phase is one, the number of chunks is N/w rounded up, and the global id is computed as $chunk \cdot w$. The custom task data contain: list of node primitives, lists for child primitives, classification

array, node pointer, array stride and number of task steps, bins, primitive counts (parent and children), node bounding box and depth, best split axis and coordinate.

9.5.1 Binning

The first phase is binning. Because we decided to use the min-max binning from [SSK07] (see Figure 4.2), we need two bin arrays per axis, so six arrays in total. A chunk represents w consecutive elements in the primitive array. Each lane takes a primitive from the chunk and computes in which bins the primitive belongs. The number of bins is static, and to make computation easier, we limit it to a multiple of w. Each primitive belongs in one min bin and one max bin on each axis.

Because the task bins are in global memory and any thread can access them, they need to be atomic. Each subgroup processes multiple chunks, so we have two options, either atomically increase the bin counter with each primitive, or compute the bin counters locally and then do the global binning at the end.

After binning all primitives, we move onto the binning post process, where we compute the best splitting plane for the current k-d tree node. The candidates are defined by the bins except the bin edges, that is $n_{bins} - 1$ splitting planes position candidates spaced uniformly (s_{bin} units) apart. We compute the cost using the SAH at all candidates and take the candidate with the lowest cost as the splitting plane of the node.

To evaluate the SAH, we need to know the surface areas and number of primitives to the left and right of the splitting plane. As explained in [SSK07], we can get the primitive counts by performing an exclusive scan on the min-bins and a reverse inclusive scan on the max-bins. Because the number of bins is static and small, this work can easily be performed by a single subgroup in post-process. By limiting the number of bins to a multiple of w, we can use subgroup-level primitives like the `__shfl_up_sync` [Cor] and others to perform a quick scan with only a single subgroup. Each lane computes a single cost value, after which we use subgroup-level reduction to find the minimum cost. Using a combination of `__ballot_sync`, which creates an integer with bits set to 1 where a statement is true, and `__ffs`, which returns the index of the first bit set to 1 in an integer [Cor], we can find the lane index of the lane with the minimum cost, which we use as the lane that does the remaining post-process work. If we have more than w bins, we iterate over them. In each iteration, we compute the cost for the current bins. Each lane than saves the minimum of the current value and the previous value. After iterating over all the bins, we perform the final reduction.

After finding the best splitting plane, we either move onto the next phase called classification, or decide that the minimum cost is worse than the cost of making the node a leaf instead, setting the next phase to make the node a leaf.

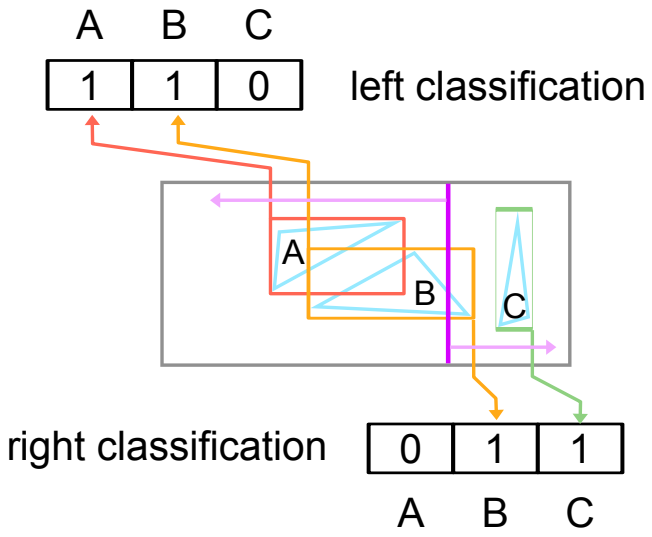


Figure 9.6 Primitives classified based on the bounds of their bounding boxes

9.5.2 Classification

The goal of this phase is to prepare the primitives for distribution to the left and right child nodes. The first step is marking which primitives belong to the left child and which to the right child. For that we need two additional arrays of integers of size equal to the number of primitives. The first array belongs to the left child, the second to the right child. Each integer corresponds to a primitive in the same position and we fill the arrays so that there is a 1 if the primitive belongs to the left/right child, and 0 if it does not (see Figure 9.6).

Each chunk again represents w consecutive elements in the primitive array, and to figure out to which child a primitive belongs to, we simply look at its bounding box in relation to the splitting plane. If the min bound is less than the splitting coordinate, it belongs to the left child. If the max bound is greater than the splitting coordinate, it belongs to the right child. A primitive can also belong to both children (see Chapter 2). Notice for future reference that we need the primitives' bounding boxes again.

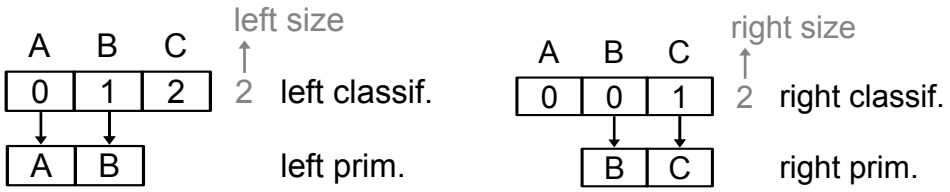
There is no notable post-processing performed for this phase besides changing the phase and updating the work chunk count.

9.5.3 Classification scan

Next, we need to perform a scan of the classification. A description of the scan algorithm using the task pool can be found in Section 9.4.

9.5.4 Distribution

The final phase for an inner node is distribution. Processing consists of distributing primitive references into their respective arrays belonging to child



(a) Only A and B have a different next element

(b) Only B and C have a different next element

Figure 9.7 Primitive distribution into left and right child nodes according to scanned classification

nodes. Chunks once again represent w consecutive elements. By looking at the current element in the scanned classification arrays (and subtracting one if the scan was inclusive), we get the index into which we should copy over the reference (see Figure 9.7). To decide whether we should copy it or not, we must simply look at the next (or previous for an inclusive scan) element in the classification arrays. If the element is different from the current one, we know that the current element was a 1 after classification (before scanning) and thus should be copied.

The final step is to create the next two tasks for the child nodes, which is done in the post-process. We update the current task with the task for the left child and we lock a new task slot for the task for the right child. The starting phase for the child tasks is either binning if the child node should be an inner node, or make leaf if it should be a leaf.

9.5.5 Make leaf

This phase always consists of a single work chunk and empty processing, with a simple post-process of assigning the variables for a leaf node.

9.6 Caching bounding boxes

We may notice that we need bounding boxes of primitives in both the binning and classification phases. Computing them might get expensive, as computing the bounding box itself, especially with split clipping, is not cheap itself. On top of that, although primitive references are located next to each other in memory, the primitives they reference will mostly not be. This leads to bad cache coherency, decreasing performance. To improve performance and coherency for the cost of memory, we can cache the primitive bounding boxes along with the primitive references.

When distributing primitive references into the left and right child nodes, we now also have to distribute their bounding boxes. Bounding boxes of primitives that lie entirely to the left or right of the splitting plane do not change. For primitives straddling the splitting plane, we have to either split the bounding box at the splitting plane coordinate if we are not clipping

them perfectly, or recompute them if we are. The algorithm we used for split clipping triangles is described in Algorithm 8 and visualised in Figure 2.4. It computes all intersections between the triangle and the bounding box, including the splitting plane. First, it computes the intersections of the edges of the triangle with the sides of the bounding box and the face defined by the splitting plane inside the bounding box. Then, it computes intersections of the edges of the bounding box and the edges of the splitting plane face with the the face of the triangle. It then adds vertices of the triangle to the intersection list. The final step is to create a bounding box of all intersections to the left of the splitting plane, and the bounding box of all intersections to the right.

9.7 Creating nodes

The last part related to dynamic memory allocation is creating nodes. We create nodes during distribution post-process, where we reserve space for two nodes and pass a pointer to them to the created tasks. We decided that each subgroup will have its own list of nodes a_s that will get copied into a single large array a_n at the end of the algorithm. Because we did not want to waste performance on copying old elements when reaching capacity for the current array (see any standard implementation of a dynamic array, e.g. std::vector), we implemented an array linked list hybrid called DeviceLinkedList. We took advantage of the fact that we only add nodes, not remove them. It has a head DeviceLinkedListHead with pointers to the first and last list. Each DeviceLinkedList consists of a pointer to the actual array, the size (number of current elements), capacity (maximum number of elements) and a pointer to the next list. Whenever the size exceeds the capacity, the next list is created with its capacity double of the current one and linked to the current one. No additional work such as copying all elements to a new array is needed. Another advantage is that direct pointers into memory are preserved.

The inner nodes created during this algorithm are not the same as the common KDTInnerNode. Instead of an index as the pointer to its children, its child pointer consists of two integers - the index of the subgroup that created the child i_s and the offset in that subgroup's list of lists o_s. The number of nodes created by each subgroup is kept in a separate array a_c. Whenever creating a new node, this number becomes the offset. When time comes to copy the nodes to the single large node array, we perform a parallel scan of the node counts, which become offsets into the large array. The total sum also tells us how much space to allocate for a_n. The code for copying would than be $a_n[a_c[i_s] + o_s] := a_s[o_s]$. When copying the nodes, we also transform the special inner build node into the standard KDTInnerNode.

9.8 Dynamic memory allocation

As mentioned before, one of the main reasons for choosing CUDA was its ability for dynamic memory allocation, and we used `new` in device code for all allocations during prototyping. After profiling the algorithm, we noticed that one of the main bottlenecks were these dynamic allocations. This lead us to creating our own, very simple, but functional allocator.

First, let us collect and analyse all uses of dynamic allocation:

(a) Extending the task pool

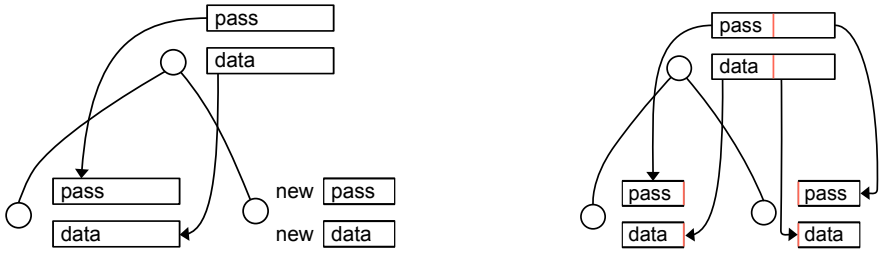
(b) Creating new linked node lists

(c) Creating room for primitive references and bounding boxes for child nodes

(d) Creating classification arrays

(a) and (b) are simple and only need allocation of new memory. (c) and (d) are a bit more complicated. First of all, the simplest way to implement them is to allocate a new array, fill it, and then deallocate it after not needing it any more. Even though this approach makes sense, we may notice that the children will need the arrays again, just smaller. For (d), if there was no primitive reference duplication, we could simply divide the array into two parts and pass it directly to the children for their own use, instead of deleting it and allocating two new arrays for each child. But because we are building a k-d tree, the duplication is inevitable. We could still divide it without any allocations if there was enough extra empty room, but we do not know the amount of extra room needed. This thought lead us to developing the MemoryPool class.

The MemoryPool class consists of two pointers and an integer. The first pointer is the start of the memory pool and the integer is its capacity, which cannot be exceeded (`__trap` [Cor] is executed when the capacity is exceeded). The second pointer is an atomic pointer to a 1-byte type that signifies the top of the pool. Each allocation increases the pool top pointer by the amount of bytes needed for the allocation padded to conform to the pool's memory alignment (e.g. if the pool has 8-byte alignment and we are allocating 4 bytes, the pool allocates 8 bytes). The allocation method returns the pointer to the first element and the size of the allocation, wrapped in a struct called AllocatedMemory. The class also has a method called AllocateOrSplit that takes AllocatedMemory together with two sizes as arguments. It tries to split the memory in AllocatedMemory into two, each with its size at least as large as the corresponding size from the arguments. If it can be split, it does not perform any atomic operations and returns the result of the split. If it cannot be split, it allocates new memory for the second requested allocation and assigns the AllocatedMemory from the argument to the first one, resulting in a single atomic increment. Because the first AllocatedMemory is much larger



(a) Passing without splitting

(b) Passing without splitting

Figure 9.8 Passing memory to children for copying primitive references

than it needs to be, there is a high change it will allow for a successful split next time it gets passed into this function.

Now what about case (c)? The issue is that we need both the parent's and children's arrays at the same time, as we would otherwise overwrite the data we need when copying from parent to child. Notice that after we distribute the data from the parent to the child, we do not need the parent data any more. That means we can split the parent array into two and pass each part to each child. The children can then use the memory as the memory for their child data. To summarise: Each node keeps two pieces of AllocatedMemory. The first piece holds valid data from the parent such as the primitive references of the current node. The second piece is split and used for data of child nodes, becoming the first piece of AllocatedMemory for the children. When creating the next task, the first piece also gets split and is passed down to the children as their second piece. This is illustrated in Figure 9.8.

The implementation focuses on requiring minimal synchronisation with other subgroups. With the knowledge that we only need to allocate new or split old memory, we achieved an allocation algorithm that only needs a single atomic increment, and only when allocating new memory. The disadvantage is that it wastes space for a chance to be split. Also the inability to deallocate becomes a slight issue when reaching a leaf node, as a leaf node does not need more splitting, but we cannot return the memory back to the pool, wasting even more memory.

Algorithm 8: Split clipping of a triangle

```
1  def SplitClip(triangle : Triangle, clippingBox : AABB, split):
2      leftBox = {∞, ∞, ∞, −∞, −∞, −∞}
3      rightBox = {∞, ∞, ∞, −∞, −∞, −∞}
4
       // Intersect triangle with the bounding box and splitting
          plane
5      I = Intersect(triangle.edges, clippingBox.sides + split.face)
6      I += Intersect(clippingBox.edges + split.face.edges, triangle.face)
7      for vertex in triangle.vertices do
8          if clippingBox.Includes(vertex) then
9              I += vertex
10         end
11     end
12
       // Make bounding boxes from intersections
13     for intersection in intersects do
14         if intersection[split.axis] ≤ split.coord then
15             leftBox.Include(intersection)
16         end
17         if split.coord ≤ intersection[split.axis] then
18             rightBox.Include(intersection)
19         end
20     end
21
       // Expand bounding boxes if their volume is zero
22     if Volume(leftBox) == 0 then
23         Expand(leftBox, ϵ)
24     end
25     if Volume(rightBox) == 0 then
26         Expand(rightBox, ϵ)
27     end
28
29     return (leftBox, rightBox)
```

Chapter 10

k-d tree merging algorithm design and implementation

[Ele10] propose two algorithms for merging multiple trees into a single one, namely `MergeBSPTrees` and `PartitionTrees`. Both algorithms are built upon the `PartitionTreeWithPlane` method, which splits the tree into two parts along a (axis-aligned) splitting plane, one part for each side (see Algorithm 9). We will briefly describe both algorithms, but for further explanation and illustrations, we recommend reading Chapter 3 from their work. Algorithms 9 and 10 were taken and modified from [Ele10].

The first algorithm recursively uses the partition method to split one of the trees into two parts using the splitting plane of the root of the other tree. In each level of recursion, it splits one of the trees and creates two sets of trees, the first set having the left partition of the split tree and the left child of the other tree, and the second set having the right partition and right child. It then repeats the process for both sets. After returning to the current level of recursion, there is a single root for each set and the two roots can be set as new children of the current node. Finally, it returns the current node as the root.

The second algorithm (see Algorithm 10) is also recursive and works in a similar manner. It looks at all bounds of the bounding boxes of the input trees and chooses one of them as the splitting plane. It then splits the trees into two sets, one for trees to the left of the splitting plane, one for trees to the right. For trees whose bounding boxes intersect with the splitting plane, it splits them using the partition method. It then repeats the process for both sets, once again returning with a single root for each set. Finally, it creates a new inner node, sets the returned roots as its children and returns it. When the splitting plane overlaps trees, it tries to generate better candidates using their `Dissolve` method, which essentially just removes the root of each tree, splitting each tree into two smaller trees and generating a single candidate per tree (that is not a leaf). This can happen only once in a row and it turns the algorithm into the `MergeBSPTrees` algorithm for trees that completely overlap.

The PartitionPrimitives method takes the primitives of a leaf node and a splitting plane and splits the primitive references in the leaf according to it, creating two new leaves. The IsTightlyBounding method takes a bounding

Algorithm 9: Partitions a tree with a plane [Ele10].

```
1  def PartitionTreeWithPlane(root, split):
2      if root is leaf then
3          return PartitionPrimitives(root, plane)
4      end
5
6      if root.split.axis == split.axis then
7          leftPar = PartitionTreeWithPlane(root.left, plane)
8          rightPar = PartitionTreeWithPlane(root.right, plane)
9          leftNode.left = leftPar[0]
10         leftNode.right = rightPar[0]
11         rightNode.left = leftPar[1]
12         rightNode.right = rightPar[1]
13         return leftNode, rightNode
14     end
15
16     if split.coord < root.split.coord then
17         left, right = PartitionTreeWithPlane(root.left, plane)
18         leftNode = left
19         rightNode.left = root.left
20         rightNode.right = right
21     else if root.split.coord < split.coord then
22         left, right = PartitionTreeWithPlane(root.left, plane)
23         leftNode.left = root.left
24         leftNode.right = right
25         rightNode = right
26     else
27         leftNode = root.left
28         rightNode = root.right
29     end
30     return leftNode, rightNode
```

box and a list of trees and returns true if the bounding box of all the trees is equal to the input bounding box. The MakeTreeFromLeaves method merges all primitive references into a single leaf node. It could also try to further subdivide them.

The first algorithm is good for merging two overlapping trees. As [Ele10] mention, the algorithm creates bad-quality trees if the trees do not overlap. For that reason, the second algorithm was developed, as it should put separate non-overlapping trees on the same level, with a new splitting plane separating them. The second algorithm is also better suited for merging more than two trees.

We assume that the static part of the scene is many times larger than the dynamic one, thus building the static tree requires much more time than building the dynamic tree(s). We also assume that most of the dynamic objects do not overlap with other objects or have minimal overlap. Both algorithms presented by [Ele10] have the disadvantage of destroying the original trees, with the need to rebuild the static tree each frame. We might

Algorithm 10: Creates a single k-d tree from input k-d trees [Ele10]

```
1  def PartitionTrees(trees, bb, comingFromDissolve):
2      if trees is empty then
3          return CreateEmptyLeafNode()
4      end
5
6      if length(trees) == 1 and IsTightlyBounding(bb, trees) then
7          return trees[0].root
8      end
9
10     split = FindBestPlane(trees, bb)
11     if (not split.isClean and not comingFromDissolve) or split.axis is None
        then
12         allLeaves = Dissolve(trees)
13         if allLeaves then
14             return MakeTreeFromLeaves(trees)
15         else
16             return PartitionTrees(trees, bb, True)
17         end
18     end
19
20     for tree in trees do
21         if tree.boundingBox.maxBounds[split.axis] ≤ split.coord then
22             leftTrees.add(tree)
23         else if split.coord ≤ tree.boundingBox.minBounds[split.axis] then
24             rightTrees.add(tree)
25         else
26             leftTree, rightTree = PartitionTreeWithPlane(tree.root, split)
27             leftTrees.add(leftTree)
28             rightTrees.add(rightTree)
29         end
30     end
31     leftBB, rightBB = Split(bb, split)
32     leftRoot = PartitionTrees(leftTrees, leftBB, False)
33     rightRoot = PartitionTrees(rightTrees, rightBB, False)
34     return CreateInnerNode(leftRoot, rightRoot)
```

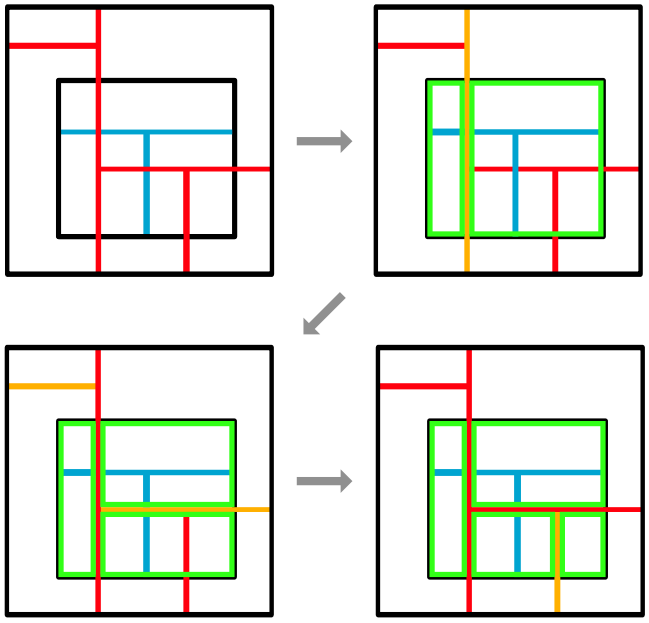


Figure 10.1 Step by step merging of a dynamic tree into a static tree

get better performance if we were able to merge the dynamic objects into the static tree while preserve it in a way that allowed it to be restored and the restoration did not consume too much time. This is the focus of our research on k-d tree merging.

10.1 Merging a single dynamic tree

Let us begin with a case where we either build a single tree over all dynamic objects, or we have a single dynamic object. The main idea is to preserve the static tree by destroying only the dynamic tree. First, we use the `PartitionTreeWithPlane` method to recursively partition the dynamic tree with the static tree nodes' splitting planes until we reach the leaves of the static tree (see figure Figure 10.1). We end up with a set of trees, where each tree belongs in a single leaf of the static tree. Let us call this method `PartitionTreeWithTree` (see Algorithm 11). Then we go over leaves that have a dynamic root associated with it, and replace each leaf with the root, while inserting primitives from the leaf into the dynamic tree. To preserve the static tree, we just need to store the leaf node somewhere else, together with a reference to its parent. When restoring the static tree, we go over all stored leaf nodes and replace the corresponding children with them, based on the attached parent references.

Algorithm 11: Partitions a dynamic tree with the static tree

```
1  def PartitionTreeWithTree(dRoot, sRoot, split):
2      if sRoot is leaf then
3          Associate(sRoot, dRoot)
4      else
5          if dRoot.boundingBox.maxBounds[split.axis] ≤ split.coord then
6              PartitionTreeWithTree(dRoot, sRoot.left, sRoot.split)
7          else if split.coord ≤ dRoot.boundingBox.minBounds[split.axis]
             then
8              PartitionTreeWithTree(dRoot, sRoot.right, sRoot.split)
9          else
10             left, right = PartitionTreeWithPlane(dRoot, split)
11             PartitionTreeWithTree(left, sRoot.left, sRoot.split)
12             PartitionTreeWithTree(right, sRoot.right, sRoot.split)
13         end
14     end
```

10.2 Merging multiple dynamic trees

We now have an algorithm that merges the dynamic tree into the static tree, while being able to restore the static tree later. For situations where we build separate dynamic trees for each object and there is more than one dynamic object, we have one obvious option - repeat the same process with each dynamic tree, but each time a new tree is merged, the tree with which we split the next dynamic tree includes all previous merged dynamic trees besides the static tree. For preservation purposes, we only store the replaced leaves if they belong to the original static tree. This strategy should work when each dynamic tree falls into a separate leaf of the original static tree, or when the trees that do fall in the same leaf overlap. This algorithm is akin to repeating a modified version of the `MergeBSPTrees` algorithm.

Algorithm 12: Merging multiple dynamic trees into the static tree

```
1  begin
2      for dynamicTree in dynamicTrees do
3          PartitionTreeWithTree(dynamicTree.root, staticTree.root,
             staticTree.root.split)
4      end
5      for leaf, data in associations do
6          if leaf is not empty then
7              data.trees += leaf
8          end
9          data.SwapLeaf(PartitionTrees(data.trees, AABB(data.trees), False)
10     end
11 end
```

Let us recall the second assumption we made – most of the dynamic objects do not overlap or have minimal overlap. If two dynamic trees fall

into the same leaf of the original static tree, and the assumption is they do not overlap, merging them one by one would have the same disadvantage as the `MergeBSPTrees` algorithm, worsening the quality of the final tree. To alleviate this issue we propose the following algorithm: First, we call `PartitionTreeWithTree` for each dynamic tree. This cases some leaves to have multiple dynamic roots associated with them. We then go over all leaves that have at least one dynamic root associated with them, store them for preservation and call the `PartitionTrees` method on the dynamic roots together with the static leaf if it is not empty (see Algorithm 12).

[Ele10] mention that the method `FindBestPlane` used in `PartitionTrees` is an equivalent for the same method used when building trees with the exact split selection. For evaluating the SAH, we need the number of primitives to the left and to the right of the splitting plane. Keeping track of the number of primitives in each tree after partitioning them would either require more computations or more memory. We decided to use our own simple heuristic that prefers splitting at the tree median where the splitting plane crosses the least number of trees. For a node B and a splitting plane candidate π, which crosses o trees and has l trees to the left and r trees to the right, the cost is calculated as:

$$C(B, \pi) = 2 \cdot o \cdot (1 + |l - r|) + o$$

We choose the split candidate with the lowest cost. The less trees the splitting plane overlaps and the closer it is to the tree median, the lower the cost. More weight is put on the number of overlapping trees, because we think it is more important.

The presented algorithm merges the static tree with trees built over dynamic objects and takes advantage of the `PartitionTrees` algorithm from [Ele10], all while preserving the static tree so that it can be restored back to its original form.

10.3 Merging on the GPU

[Ele10] mention trying another algorithm: inserting primitives from dynamic objects into static leaves and subdividing further whenever needed. They state that it was too slow, so they abandoned it. We suggest that it would be well suited for GPUs, because of its potential for being parallelised.

Let us assume we have built the static tree. First, we need to associate each dynamic primitive with all static tree leaves it intersects (intersect with leaf = intersect with the bounding box associated with that leaf). All such primitives need to be put into lists, one list per leaf. This can be performed in parallel and there are two methods of computation. The first method is that each thread takes a leaf and iterates over all primitives, saving primitives that intersect with the leaf to a list owned by the thread. This method has linear time complexity and all threads must iterate over all primitives, even for leaves with which no primitives intersect.

The second method is that each thread takes a primitive and traverses the tree to determine which static tree leaves the primitive intersects. This method has expected logarithmic time complexity if we assume the primitives intersect with a constant number of leaves. There are also no wasted computations, as all primitives must intersect at least one leaf. The issue with this method is that it requires synchronisation. When two threads determine that their primitives belong in the same leaf, they need to avoid a data race.

Next, we need to build a tree for each static tree leaf that does not have an empty list of dynamic primitives. Let us call such leaves marked leaves. This part is solved easily with the use of the GPU task pool. For each marked leaf, we first add the static primitives that belong to it to the list of dynamic primitives associated with the leaf. We also store the original leaf so that it can be restored later. We then create a task for each marked leaf in the task pool. The tasks are created in the same way as in the normal building process, we just start with multiple tasks and each task has the appropriate leaf node set as the root.

After processing the whole task pool, the algorithm is complete. After rendering the image, we can restore the static tree from the stored leaves.

Chapter 11

Results

The application was tested with the implemented algorithms and the performance and other statistics were measured. In this chapter, we present the hardware used to test the application and the test results.

11.1 Hardware

- OS: Windows 10 Pro, 22H2
- RAM: 32 GB, 2400 Mhz
- CPU:
  - AMD Ryzen 5 5600 6-core
  - Base frequency: 3.50 GHz
  - L1 cache: 384 KB
  - L2 cache: 3 MB
  - L3 cache: 32 MB
- GPU:
  - NVIDIA GeForce RTX 4080
  - Dedicated GPU memory: 16 GB
  - Base clock speed: 2205 MHz
  - Architecture: Ada Lovelace
  - Compute capability: 8.9
  - Streaming Multiprocessors: 76
  - L1 cache (per SM): 128 KB
  - L2 cache: 64 MB
- Compilation:
  - C++ compiler: MSVC cl.exe
  - Cuda Toolkit version: 12.6
  - Configuration: x64 Release

11.2 Statistics

For testing, we need to save various measurements and later compute statistics from them. The statistics could be processed by the same application that creates them or by a separate application. Since we think Python is better suited for data processing and analysis than C++, we decided to choose the second approach.

To create measurements that could be processed by another application, we save them in JSON files with a .stats extension. The statistics file is a JSON object with two root keys: info and data. Both are JSON objects with keys that represent the name of the measurement. Saving measurements is performed by interacting with the TableStatistics class in C++ code. When a measurement is saved to data, it is added to a list (JSON array) of values, from which we can then take averages, medians, etc. When it is saved to info, only the first input value for a specific key is added. We also wanted to allow running the application multiple times while using the same statistics file. Instead of overwriting it, if the specified config file exists, it is opened, parsed, and the values are used as the initial values for the TableStatistics class. This way, the data from the previous run are not lost, but extended.

To automatically process the statistics and make testing easier, we implemented three helper scripts using Python. Script *create_benchmark_configs.py* has methods to generate different configurations based on a template config. Script *TableUtil.py* has classes and methods for retrieving values from statistics files and creating tables from them. Script *benchmarker.py* has methods for running tests using the generated configs. *table_maker.py* has dictionaries to retrieve and calculate specific statistics using TableUtil and methods to create tables from the results of the benchmarker script.

11.3 Testing

Testing was performed on ten static scenes and five dynamic scenes, where one dynamic scene has three variations. Four of the static scenes are the dynamic scenes with dynamic objects removed. The only exception is the FairyForest scene, where the whole scene is dynamic, so the static version of the scene consists of the first animation frame. Animation is achieved by using morph animations and InterpolateTransform components. InterpolateTransform takes the original transform the object started with and a target transform and interpolates between them.

Each test consisted of running the application two times for 20 frames without GUI with 3840×2160 resolution, for a total of 40 frames. Because we feel that 40 frames is not enough to adjust for external influences, such as OS scheduling, we chose to take the median as the final value of time-related statistics. The scenes were shaded using primitive normals and material colours, without textures, and with differing numbers of lights (mostly two to three lights).

Each test is represented by a global configuration file. Each scene also has its own configuration file, with a custom R (see Section 9.3) and n_{max} (see Section 9.2). The configurations with building algorithms use a GPU ray tracer with the short-stack with push-down traversal algorithm. k-d trees built on the CPU were transferred to the GPU before ray tracing.

11.3.1 Static scenes

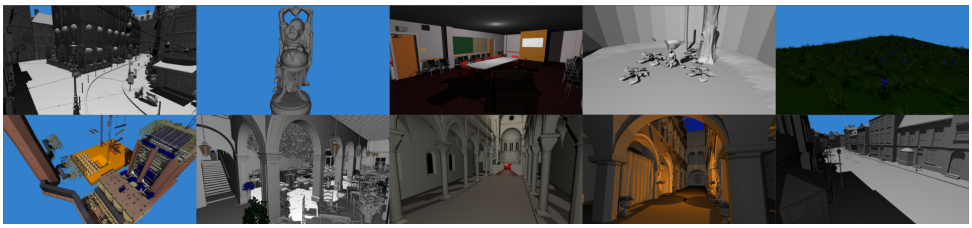


Figure 11.1 Renders of static scenes. From left to right: Bistro, Buddha, Conference, Fairy forest, Field, Power plant, San Miguel, Sibenik, Sponza, Street.

Both CPU implementations with exact and approximate split selection and the GPU implementation were tested on each static scene. The different traversal algorithms were also tested on each scene. We present some of the results in this subsection in the form of graphs, but all statistics, including some not mentioned, and their values, can be found in Appenix D.1 in the form of tables. Each set of tables has a caption at the top of the page with the scene name and some common statistics. For the final renders for each scene, see Figure 11.1.

The following is a list of configurations with explanations for each configuration name:

- **CPU exact**: exact split selection algorithm on the CPU
- **CPU approx**: approximate split selection (binning) algorithm on the CPU
- **GPU approx**: approximate split selection (binning) algorithm on the GPU
- **Stack**: stack-based traversal algorithm
- **Restart**: kd-restart traversal algorithm
- **PushDown**: push-down traversal algorithm
- **RestartShortStack**: short-stack traversal algorithm with kd-restart, stack capacity in parentheses
- **PushDownShortStack**: short-stack traversal algorithm with push-down, stack capacity in parentheses

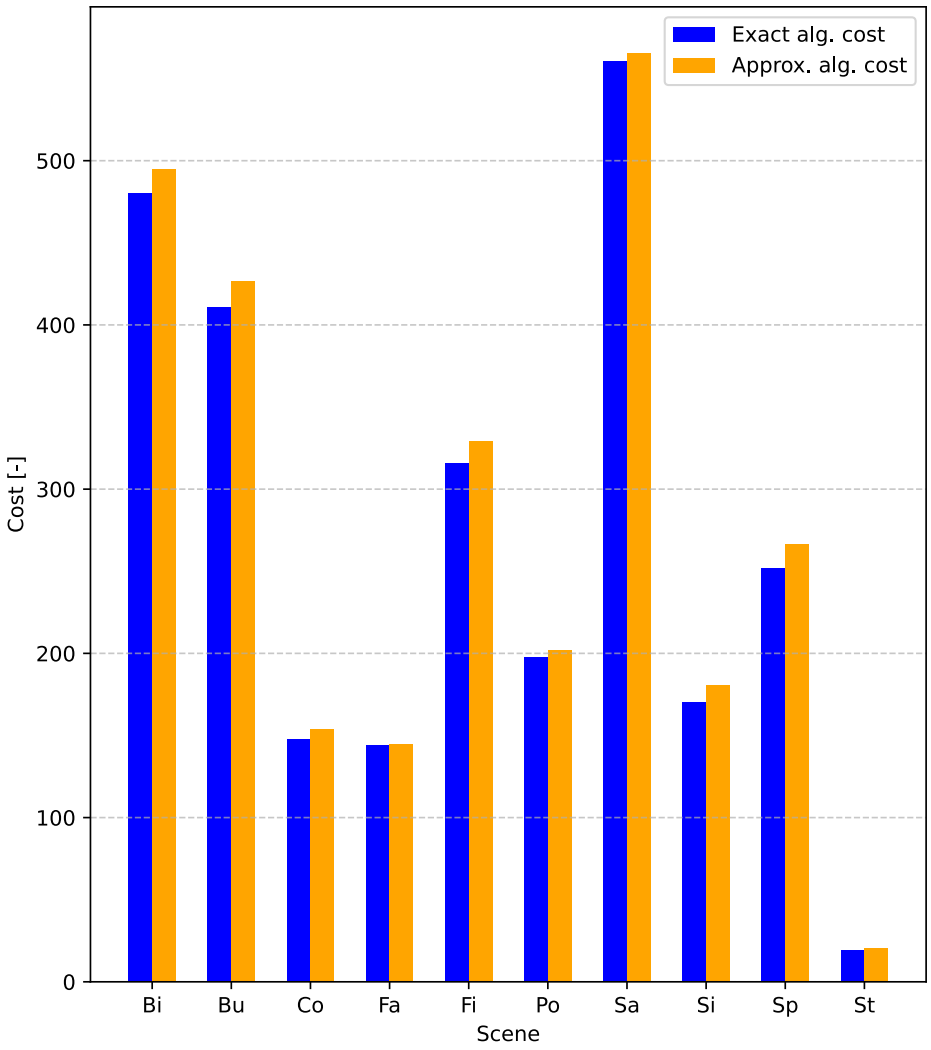


Figure 11.2 Comparison of costs of the algorithms with exact and approximate split selection

We can draw several conclusions based on the results. The algorithm with exact split selection has better cost then the approximate split selection in all scenes, but surprisingly does not always have better ray tracing performance (see Figure 11.2 and Figure 11.3). The approximate split selection algorithm on the CPU and on the GPU have similar tree statistics, but they are not identical. The difference can be introduced when performing computations with floating point numbers in parallel. Due to rounding errors, some operations, such as addition and multiplication, lose their associative property under those circumstances (see tables in Appenix D.1).

The build performance of the GPU algorithm is significantly better than that of the CPU algorithms (see Figure 11.4). Because we expect the GPU algorithm to have some performance overhead, which should be more notice-

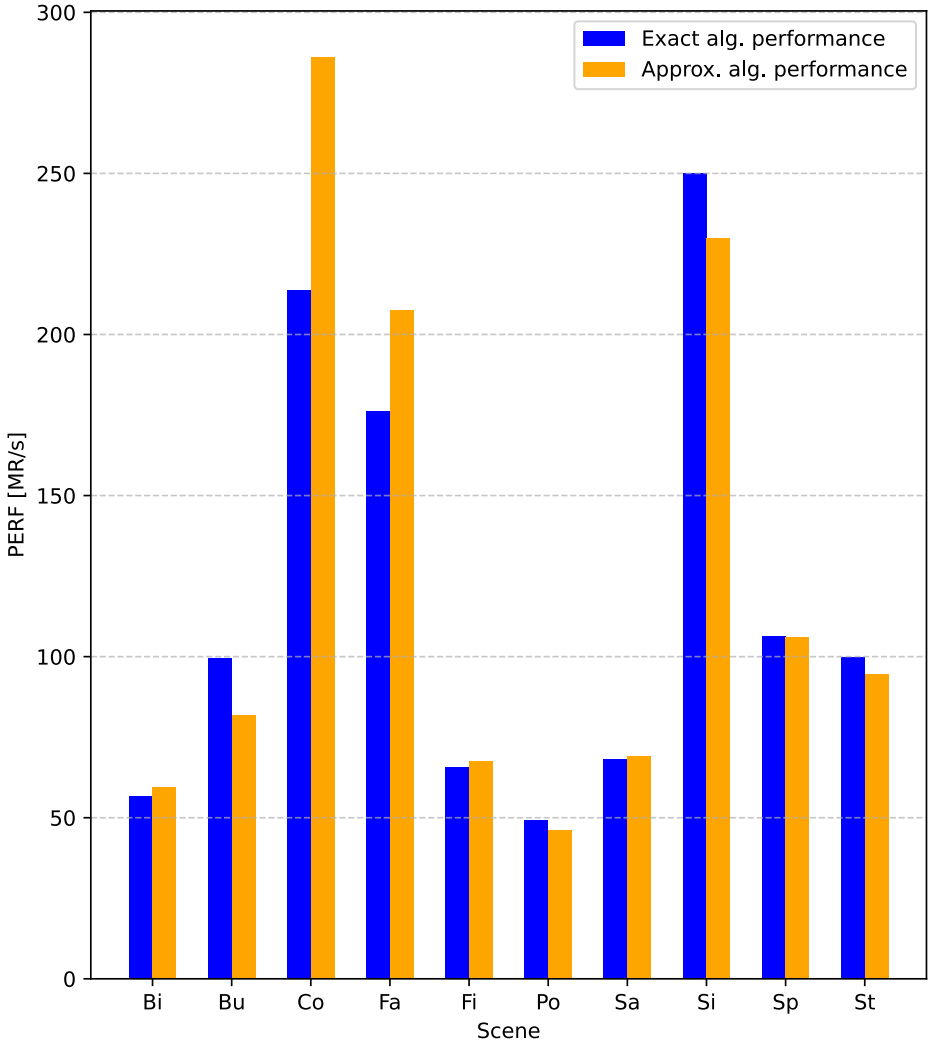
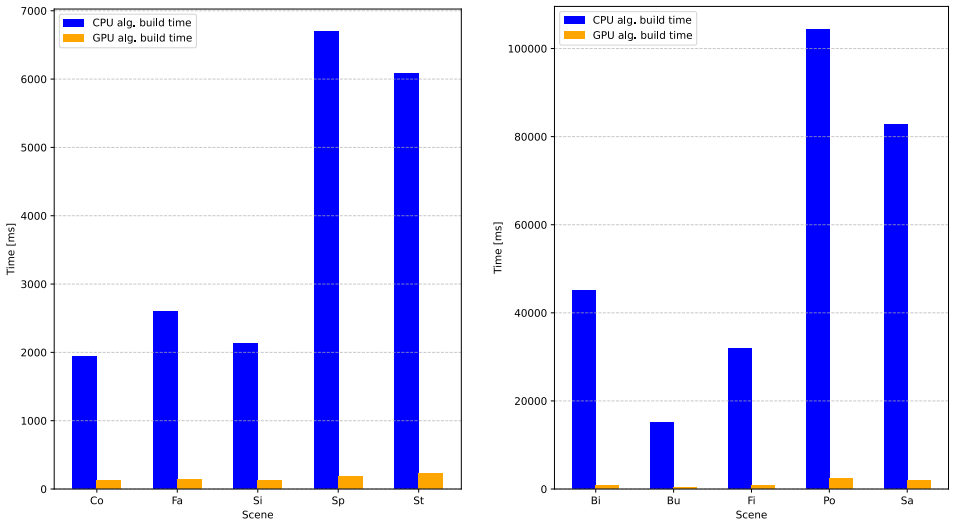


Figure 11.3 Comparison of ray tracing performance of the algorithms with exact and approximate split selection

able in smaller scenes, we divided the time statistics based on the number of primitives in the scene. In large scenes (more than one million primitives), the GPU building algorithm is at least 38× faster, and at most 53× faster than the CPU building algorithm with approximate split selection. On small scenes, it is at least 15×, and at most 36× faster, which confirms our hypothesis. We also included statistics for the time it took to complete processing the task pool, because that is the most optimised part of the algorithm. As we can see from Figure 11.5, the task pool takes about half of the total build time. The total time includes operations like copying nodes after the tree is built, and transforming them from build nodes to tree nodes (see Section 9.7). This suggests that the GPU algorithm could be noticeably improved by optimising node copying.



(a) Build times for small scenes **(b)** Build times for large scenes

Figure 11.4 Comparison of build times for the CPU and GPU algorithms, both using approximate split selection. The left graph is for scenes with less than a million primitives, the right is for the rest.

The k-d trees built on the GPU take up more memory than the ones built on the CPU, which can be seen in Figure 11.6. The presented amounts of memory do not include the total amount of memory needed for building the tree (for example the memory needed for cached bounding boxes and classification arrays), which is much higher. For example, to build the Power plant scene with more than 12 million primitives, we needed to allocate 2 GB for the primitive memory pool (which stores primitive references) and 8 GB for the data memory pool (which stores other data, such as the bounding boxes, additional task pools, build nodes, etc.).

The performance of the different traversal algorithms can be seen in Figure 11.7. The stack-based traversal has the worst performance across the board, presumably because the stack was located in global memory. The short-stack traversal algorithms have better performance with lower stack capacities, even though the number of traversal steps is higher (see tables in Appenix D.1). Based on the results, we cannot decide which traversal algorithm is the best, as both the stackless and the short-stack traversal algorithms were the fastest for some scenes.

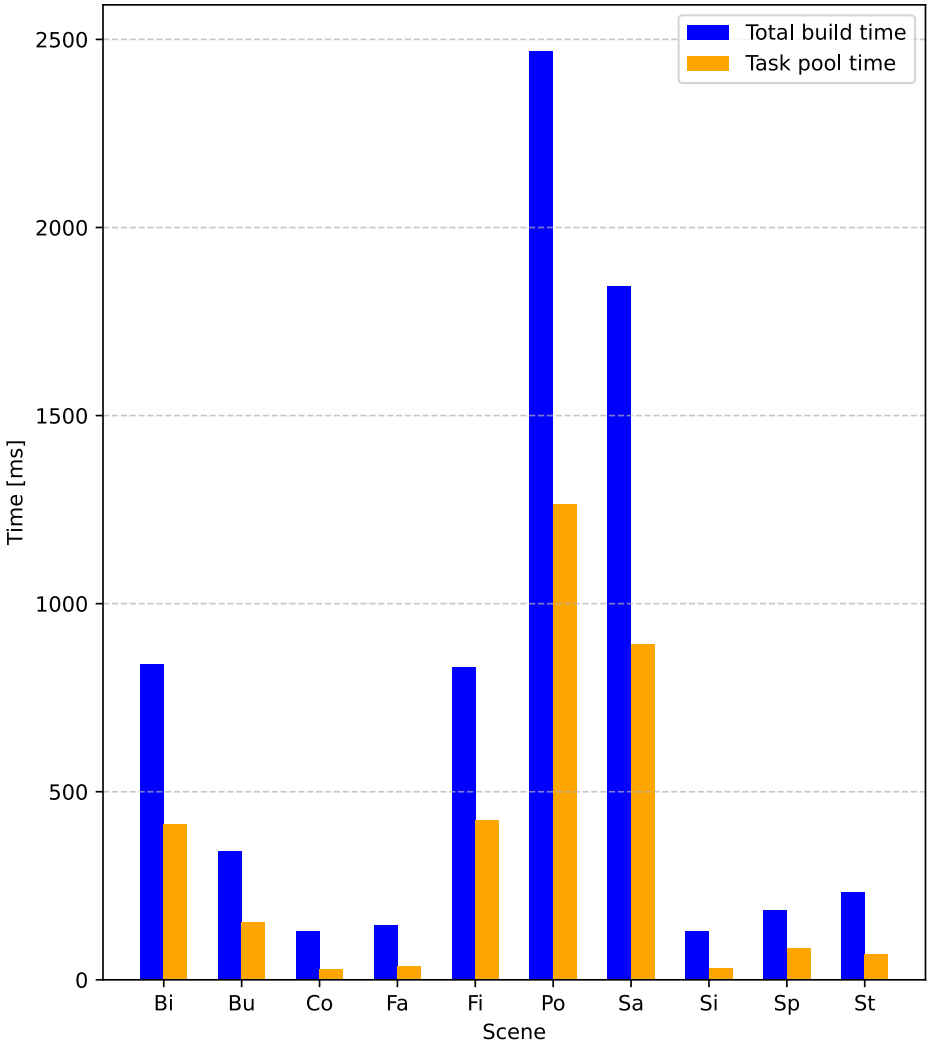


Figure 11.5 Comparison of the total build time of the GPU algorithm and the time it took for the task pool to finish

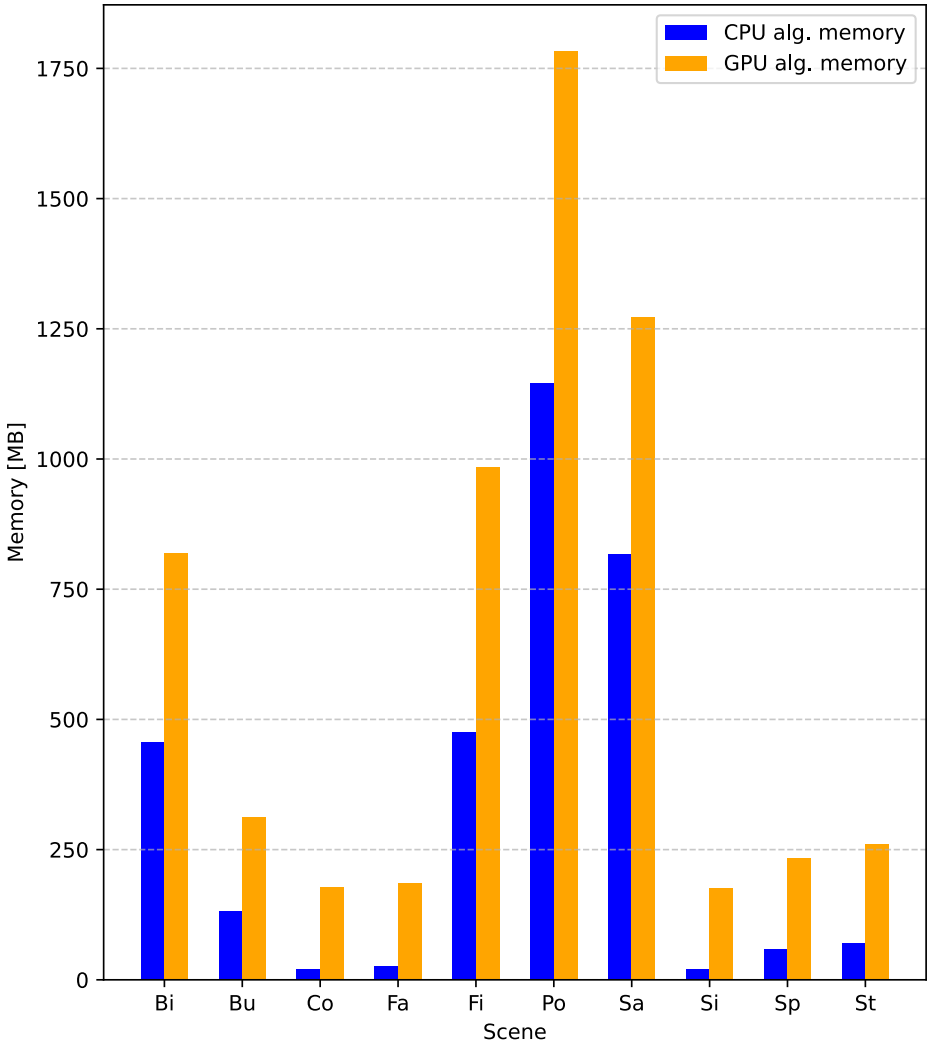


Figure 11.6 Comparison of the memory consumption of the CPU and GPU algorithms using approximate split selection

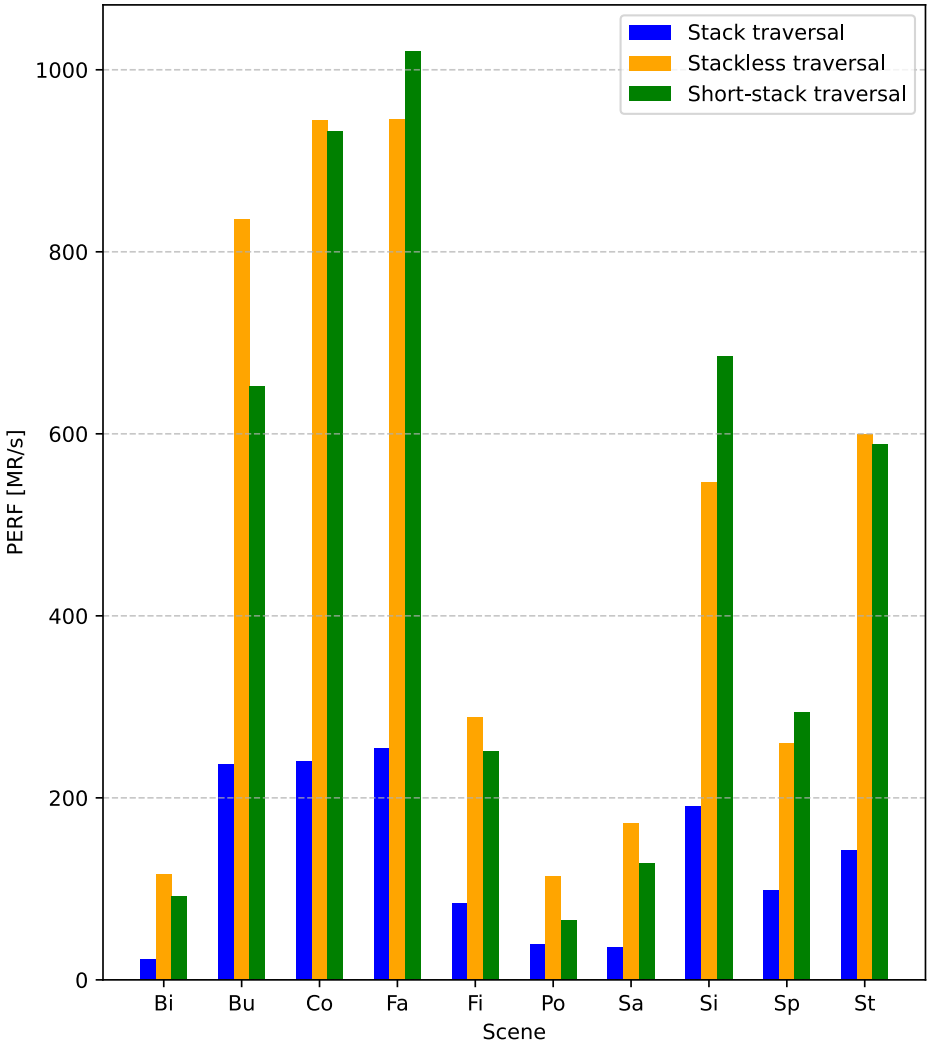


Figure 11.7 Comparison of the ray tracing performance for the different traversal algorithms. Only the stackless and short-stack traversal algorithms with push-down and stack capacity of four are displayed.

11.3.2 Dynamic scenes

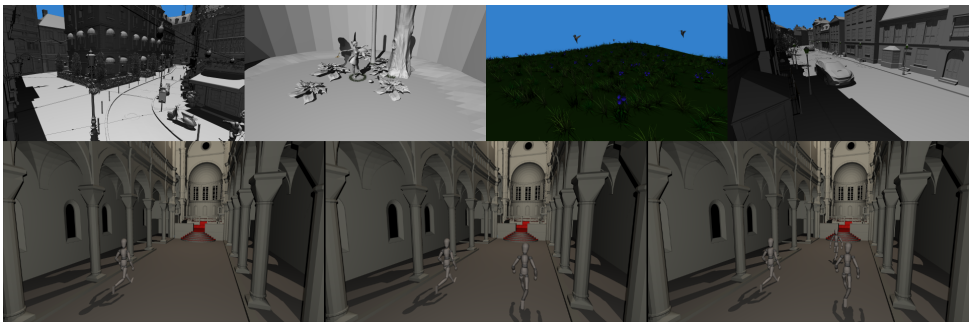


Figure 11.8 Renders of dynamic scenes. From left to right: Bistro, Fairy forest, Field, Street, and Sibenik with one, two and three dynamic objects.

The dynamic scenes were used to test the k-d tree merging implementation. We have only implemented the CPU algorithm. For comparison, we have also included the CPU exact split selection and the GPU binning implementations. We also tested building the whole tree versus building a separate k-d trees for the static and dynamic objects. The Sibenik scene was tested with one, two, and three dynamic objects to determine the effect of varying the number of objects on the tree build times and quality. Once again, some results are presented in the form of graphs, but the complete results can be found in Appenix D.1. For the final renders, see Figure 11.8.

The setup was the same as for the static scenes, only with different configuration files. The following is a list of configurations used for dynamic scene testing with explanations for each configuration name.

- **CPU single**: approximate split selection algorithm on the CPU, k-d tree rebuilt each frame for the whole scene
- **CPU both**: static tree built using exact split selection, dynamic tree built using approximate split selection, both on the CPU
- **CPU merge**: static tree built using exact split selection, dynamic trees built using approximate split selection and merged into the static tree on the CPU
- **GPU single**: approximate split selection algorithm on the GPU, k-d tree rebuilt each frame for the whole scene
- **GPU both**: static tree and dynamic tree built using exact split selection separately on the GPU

As we can see from Figure 11.9, Figure 11.10, and Figure 11.11, the merging algorithm is never better in our tests than building two separate trees for the static and dynamic part of the scene. Although building two separate trees means traversing two trees instead of one, the performance is not much different than when traversing a single tree. The total time to render a frame

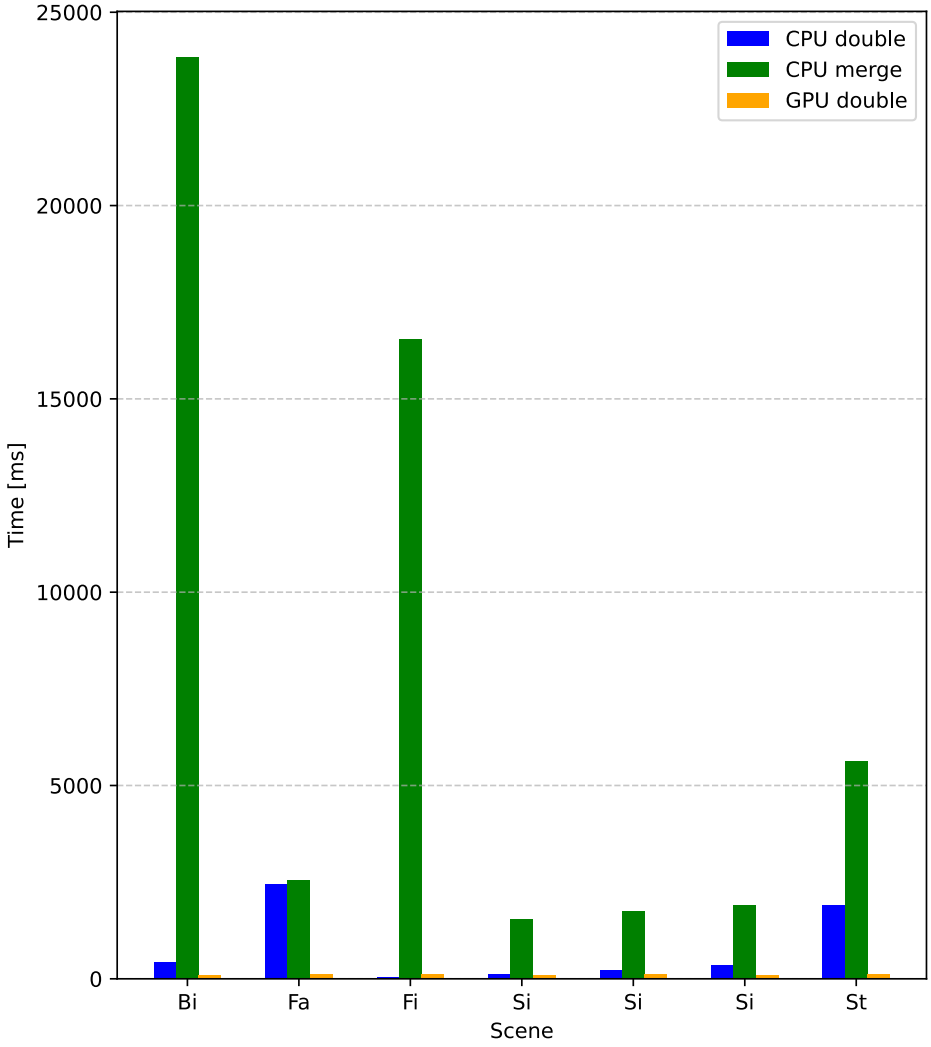


Figure 11.9 Comparison of build times of the merging algorithm and the build times of the standard algorithms for only the dynamic part of the scene

is about two times lower than when rebuilding the whole tree, which could be improved by further optimising the merging algorithm, but it does not seem better than building two separate trees. The merging times also depend on the number of dynamic objects. Adding a dynamic object increases the total time to render a frame by around 10% for the merging algorithm, and by around 50% for the algorithm with two trees, which means that with increased number of dynamic objects, the merging algorithm might get better results.

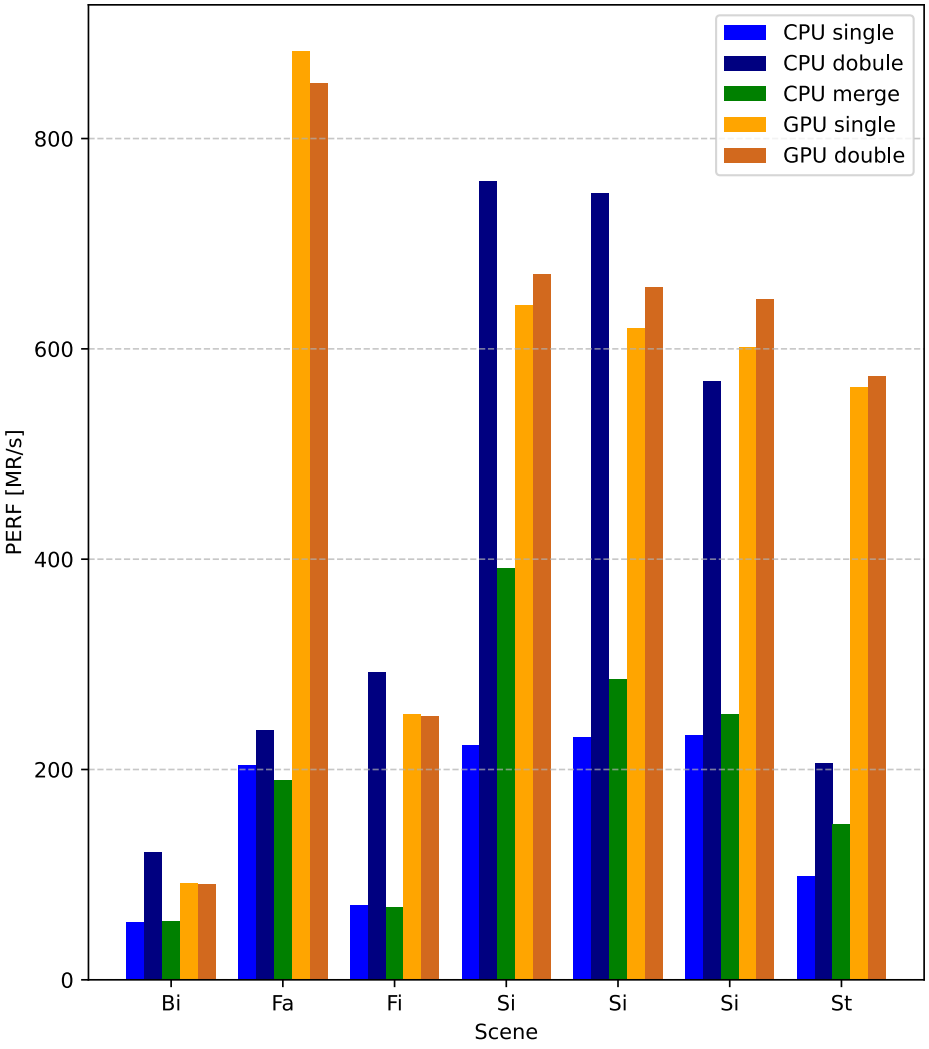


Figure 11.10 Comparison of ray tracing performance for all dynamic scene configurations

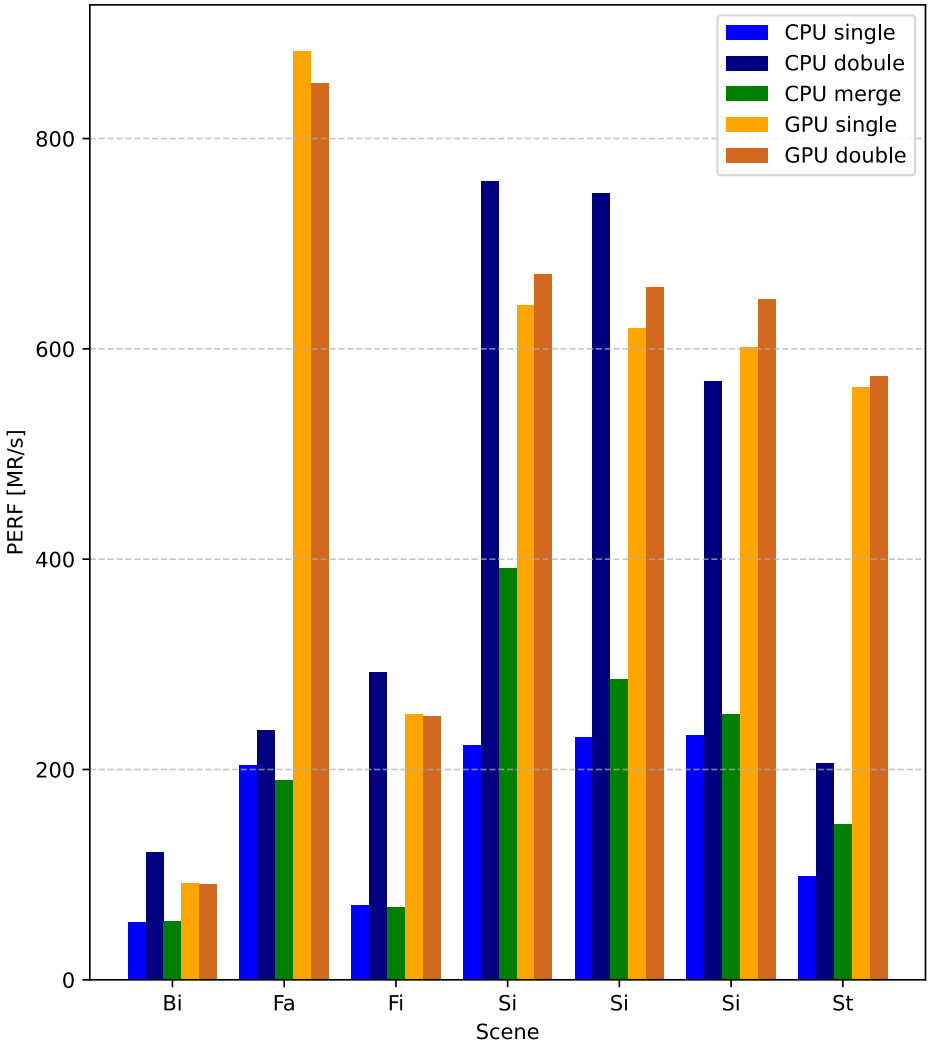


Figure 11.11 Comparison of total time it took to render a frame for all dynamic scene configurations

Chapter 12

Conclusion

The goal of this thesis was to study, implement, and test algorithms for building and traversing k-d trees to accelerate ray tracing. First, an overview of the theory related to the discussed topics was provided. Then, a study of existing solutions for building and traversing k-d trees was presented, focused on parallel algorithms. The analysis of the existing solutions concluded with proposing one GPU algorithm for building k-d trees and selecting two CPU k-d tree building algorithms for comparison with the parallel algorithm. It was decided that the GPU algorithm should run entirely on the GPU and therefore a task pool with persistent warps was chosen as the main component of the algorithm.

Then, an application framework was proposed to provide a basis for implementing the proposed algorithms. The application was designed as a game engine with support for debugging and visualising ray tracing algorithms, with the intention of providing an environment for their implementation and testing with extensive flexibility.

Next, the design and implementation of the proposed algorithm for building k-d trees on the GPU were thoroughly examined. The algorithm utilises a task pool to build nodes, and uses binning to evaluate the cost function. An algorithm for merging k-d trees was also proposed. It is aware of the existence of a static k-d tree that is assumed to be considerably larger than any k-d trees built over dynamic objects and that does not change over time. It is designed to preserve this static tree when merging all trees into a single tree and restore it to its original form, so that the static tree can be used again without the need to rebuild it. An algorithm for merging k-d trees on the GPU was also proposed, but not implemented.

Finally, the implemented algorithms were tested and compared. Our implementation of the GPU building algorithm was proven to be significantly faster than the CPU implementations. The main drawback is the amount of memory needed to build the k-d tree on the GPU, which could be improved with a more sophisticated dynamic memory allocator. The algorithm still has more potential, as the build time itself, signified by the time needed to process the task pool, is around two times lower than the total build time. The algorithm is also scalable, with better speed-ups for larger scenes.

The specialised traversal algorithms were proven to be better than the stack algorithm, although we could not determine which of them is the best. The k-d tree merging algorithm was discovered to be a valid option for dynamic scenes, but not the best one in our tests.

Bibliography

[Adv] I. Advanced Micro Devices. *HIP documentation*. URL: https://rocm.docs.amd.com/projects/HIP/en/latest/ (visited on 03/01/2024).

[AL09] T. Aila and S. Laine. 'Understanding the efficiency of ray traversal on GPUs'. In: *Proceedings of the Conference on High Performance Graphics 2009*. HPG '09. New Orleans, Louisiana: Association for Computing Machinery, 2009, pp. 145–149. ISBN: 9781605586038.

[AR14] S. Akl and W. Rheinboldt. *Parallel Sorting Algorithms*. Notes and reports in computer science and applied mathematics. Academic Press, Inc., 2014. ISBN: 9781483268088.

[Ans22] R. Ansorge. *Programming in Parallel with CUDA: A Practical Guide*. Cambridge University Press, 2022. ISBN: 9781108855273. DOI: 10.1017/9781108855273.

[Ben75] J. L. Bentley. 'Multidimensional Binary Search Trees Used for Associative Searching'. In: *Commun. ACM* 18.9 (1975), pp. 509–517. ISSN: 0001-0782. DOI: 10.1145/361002.361007.

[CSI15] B. Chang, W. Seo and I. Ihm. 'On the efficient Implementation of a real-time kd-tree construction algorithm'. In: *GPU Computing and Applications* (2015), pp. 207–219.

[CKL10] B. Choi et al. 'Parallel SAH K-D Tree Construction'. In: *Proceedings of the Conference on High Performance Graphics*. HPG '10. Saarbrucken, Germany: Eurographics Association, 2010, pp. 77–86.

[Cop95] J. O. Coplien. 'Curiously recurring template patterns'. In: *C++ Rep.* 7.2 (Feb. 1995). ISSN: 1040-6042.

[Cor] N. Corporation. *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 18/12/2024).

[DPS10] P. Danilewski, S. Popov and P. Slusallek. *Binned SAH Kd-Tree Construction on a GPU*. Tech. rep. Saarland University, Computer Graphics Lab, June 2010.

[Ele10] E. Eleftheriades. 'Accelerating ray tracing for dynamic scenes using kd-tree merging'. Bachelor thesis. University of Cyprus, 2010.

[FS05] T. Foley and J. Sugerman. 'KD-tree acceleration structures for a GPU raytracer'. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. HWWS '05. Los Angeles, California: Association for Computing Machinery, 2005, pp. 15–22. ISBN: 1595930868. DOI: 10.1145/1071866.1071869.

[GS87] J. Goldsmith and J. Salmon. 'Automatic Creation of Object Hierarchies for Ray Tracing'. In: *IEEE Computer Graphics and Applications* 7.5 (1987), pp. 14–20. DOI: 10.1109/MCG.1987.276983.

[Hav00] V. Havran. 'Heuristic ray shooting algorithms'. PhD thesis. Czech Technical University in Prague, 2000.

[HHS23] P. Hijma, S. Heldens, A. Sclocco, B. van Werkhoven and H. E. Bal. 'Optimization Techniques for GPU Programming'. In: *ACM Comput. Surv.* 55.11 (Mar. 2023). ISSN: 0360-0300. DOI: 10.1145/3570638.

[HSH07] D. R. Horn, J. Sugerman, M. Houston and P. Hanrahan. 'Interactive k-d tree GPU raytracing'. In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. I3D '07. Seattle, Washington: Association for Computing Machinery, 2007, pp. 167–174. ISBN: 9781595936288. DOI: 10.1145/1230100.1230129.

[KNP13] Y.-S. Kang, J.-H. Nah, W.-C. Park and S.-B. Yang. 'gkDtree: A group-based parallel update kd-tree for interactive ray tracing'. In: *Journal of Systems Architecture* 59.3 (2013), pp. 166–175. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2011.06.003.

[LDG17] Z. Li, Y. Deng and M. Gu. 'Path Compression Kd-Trees with Multi-Layer Parallel Construction a Case Study on Ray Tracing'. In: *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '17. San Francisco, California: Association for Computing Machinery, 2017. ISBN: 9781450348867. DOI: 10.1145/3023368.3023382.

[MB90] D. J. MacDonald and K. S. Booth. 'Heuristics for ray tracing using space subdivision'. In: *Vis. Comput.* 6.3 (May 1990), pp. 153–166. ISSN: 0178-2789. DOI: 10.1007/BF01911006.

[MGM11] A. Munshi, B. Gaster, T. G. Mattson, J. Fung and D. Ginsburg. *OpenCL Programming Guide*. 1st. Addison-Wesley Professional, 2011. ISBN: 0321749642.

[Ngu07] H. Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007. Chap. 39. ISBN: 9780321545428. DOI: 10.5555/1407436.

[SSK07] M. Shevtsov, A. Soupikov and A. Kapustin. 'Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes'. In: *Computer Graphics Forum* 26.3 (2007), pp. 395–404. DOI: 10.1111/j.1467-8659.2007.01062.x.

[Sol78] H. Solomon. *Geometric Probability*. CBMS-NSF Regional Conference Series in Applied Mathematics. Society for Industrial and Applied Mathematics, 1978. ISBN: 9780898710250.

[SBS03] L. Szécsi, B. Benedek and L. Szirmay-Kalos. 'Accelerating animation through verification of shooting walks'. In: *Proceedings of the 19th Spring Conference on Computer Graphics*. SCCG '03. Budmerice, Slovakia: Association for Computing Machinery, 2003, pp. 231–238. ISBN: 158113861X. DOI: 10.1145/984952.984990.

[TPK16] M. Tillmann, P. Pfaffe, C. Kaag and W. F. Tichy. 'Online-Autotuning of Parallel SAH kD-Trees'. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2016, pp. 628–637. DOI: 10.1109/IPDPS.2016.31.

[Vin14] M. Vinkler. 'Construction of Acceleration Data Structures for Ray Tracing'. PhD thesis. Masarykova univerzita, 2014.

[VHB16] M. Vinkler, V. Havran and J. Bittner. 'Performance Comparison of Bounding Volume Hierarchies and Kd-Trees for GPU Ray Tracing'. In: *Computer Graphics Forum* 35.8 (2016), pp. 68–79. DOI: 10.1111/cgf.12776.

[WH06] I. Wald and V. Havran. 'On building fast kd-trees for ray tracing, and on doing that in O (N log N)'. In: *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE. 2006, pp. 61–69.

[WGD14] Y. Wang, P. Guo and F. Duan. 'A fast ray tracing algorithm based on a hybrid structure'. In: *Multimedia Tools and Applications* 75.4 (2014), pp. 1883–1898. ISSN: 1573-7721. DOI: 10.1007/s11042-014-2378-3.

[Whi80] T. Whitted. 'An improved illumination model for shaded display'. In: *Commun. ACM* 23.6 (June 1980), pp. 343–349. ISSN: 0001-0782. DOI: 10.1145/358876.358882.

[Wika] Wikipedia. *Composition over inheritance*. URL: https://en.wikipedia.org/wiki/Composition_over_inheritance (visited on 03/01/2024).

[Wikb] Wikipedia. *Composition over inheritance*. URL: https://en.wikipedia.org/wiki/Morph_target_animation (visited on 03/01/2024).

[Wol18] D. Wolff. *OpenGL 4 Shading Language Cookbook: Build high-quality, real-time 3D graphics with OpenGL 4.6, GLSL 4.6 and C++17, 3rd Edition*. Packt Publishing, 2018. ISBN: 9781789342253.

[WZL11] Z. Wu, F. Zhao and X. Liu. 'SAH KD-Tree Construction on GPU'. In: *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*. HPG '11. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2011, pp. 71–78. ISBN: 9781450308960. DOI: 10.1145/2018323.2018335.

[ZHW08] K. Zhou, Q. Hou, R. Wang and B. Guo. 'Real-Time KD-Tree Construction on Graphics Hardware'. In: *ACM Trans. Graph.* 27.5 (2008). ISSN: 0730-0301. DOI: 10.1145/1409060.1409079.

[ZM11] P. Zhou and X. Meng. 'SAH Based KD Tree Construction on Hybrid Architecture'. In: *2011 Workshop on Digital Media and Digital Content Management*. 2011, pp. 185–189. DOI: 10.1109/DMDCM.2011.14.

Appendix A

Attachment list

Attachments:

- source.zip - Full project source code (CMake project and Python scripts) with a link to a gitlab repository with the latest version of the project and other documents
- imgs.zip - Full scale (4k) renders of the test scenes
- scenes.zip - Four smaller scenes (static Conference, Sibenik and Sponza, and all three versions of the dynamic Sibenik) with a link to the rest of the test scenes
- manual.pdf - Manual describing how to build and use the project (also included in Appenix B)

Appendix B
Manual

The project is meant to be portable, but was tested only on Windows 10. The following information is therefore mainly for Windows users.

B.1 Build instructions

First, we assume the following prerequisites have been fulfilled:

- Installed CMake version $\geq$ 3.25.2
- Installed package manager compatible with CMake for the current OS (we used vcpkg on Windows)
- NVIDIA GPU with compatible compute capability
- C++ compiler, CUDA Toolkit 12.6

To build the project, follow these instructions:

1. Download and unpack source.zip.
2. Use a program such as Visual Studio Code to open the directory ProperRaytracer.
3. Select a configuration: Debug for an unoptimised build with debug info, RelWithDbgInfo for an optimised build with debug info, or Release for an optimised build.
4. Build the project using CMake.

B.2 Usage instructions

After building the CMake project, the application executable should be located in the build folder, called *ProperRaytracer.exe*. The application takes several arguments, with a mandatory scene name argument that specifies the scene to start on. Scenes, configs, models, and other assets can be provided with both an absolute path, or a path relative to the Assets folder. The

Assets folder next to the project source code is copied to the current build folder on each build.

When no arguments are provided to the application or the -h/--help argument is provided, the list of arguments with descriptions of what they do is printed to the console. We refer the reader to this list for more information on arguments.

The main feature of the application is viewing scenes and testing implemented ray tracers. Scenes are in the JSON format and can be created and edited manually. There is a simple test scene with some test objects in the project's Assets folder. The scenes used for testing are attached in a separate file (scenes.zip). The description of all implemented components and other scene properties is beyond the scope of this manual, but can be deduced from reading the commented source code.

After starting the application, the scene is loaded and displayed. All attached scenes have the camera movement component. Hold the right mouse button and use WASD for horizontal movement, Space and CTRL for vertical movement (relative to the orientation of the camera). Use Shift to double the speed. Speed and look sensitivity can be modified in the scene files.

If opened in the OpenGL mode, the frame tracer window can be used to pause the scene and run the current ray tracer. The built data structure can then be viewed using the Structure navigator window, if the StructureNavigator component is in the scene. If opened in the Raytracing mode, the application will build the specified data structure and ray trace the scene automatically.

Configs used for testing are located in the StaticBenchmarks and DynamicBenchmarks folders in next to the test scenes. To run the tests, run *benchmarker.py* with the correct build path set in the script.

Appendix C

Assignment translation

MASTER'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Papay Robert** Personal ID number: **492304**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Graphics and Interaction**

Study program: **Open Informatics**

Specialisation: **Computer Graphics**

II. Master's thesis details

Master's thesis title in English:

Efficient ray tracing algorithms exploiting kd-trees on a GPU

Master's thesis title in Czech:

Efektivní algoritmy s využitím kd-stromů pro vrhání paprsků na GPU

Guidelines:

Study the literature on building and using data structures on a graphics accelerator for ray tracing with a focus on parallelization of computation and kd-trees. Implement and compare efficient kd-tree traversal algorithms on a set of 3D test scenes of varying size and spatial distribution. Next, study the fast algorithms for building a kd-tree on the GPU. Implement and test the selected algorithms for at least ten test scenes with the number of triangles from 100K to about 100M.
Then, address the algorithms for merging two or more kd-trees into one kd-tree and the reverse operation of removing an already inserted kd-tree. Test implementations of these algorithms on the CPU, and then transfer them to the GPU. Test the applicability of kd-tree merging for simple dynamic scenes with one or more objects with triangle counts of 1K to 50K. Measure the time and memory requirements of the kd-tree construction, traversal, and merging algorithms for at least 5 test scenes containing animation.
For implementation on the GPU, use the NVIDIA CUDA language, or a language described by the OpenCL standard, or another suitable programming language in the form of a common API exploiting these languages, such as HIP/SYCL.

Bibliography / sources:

1) Zhou et al.: Real-Time KD-Tree Construction on Graphics Hardware,
SIGGRAPH ASIA 2008 and references to this paper.
2) M. Vinkler: Construction of Acceleration Data Structures for Ray Tracing, PhD thesis, Masaryk University 2014.
https://is.muni.cz/th/w0k6h/? kod=PV204
3) D. Horn, J. Sugerman, M. Houston, P. Hanrahan, Interactive k-D Tree GPU Raytracing, 2007.
4) Z. Wu, F. Zhao, X. Liu: SAH KD-tree construction on GPU, HPG 2011.
5) S. Chung, M. Choi M, D. Youn D and S. Kim S. (2019). Comparison of BVH and KD-Tree for the GPGPU Acceleration on Real Mobile Devices. Frontier Computing. 10.1007/978-981-13-3648-5_62. (535-540).
6) X. Liang X, H. Yang, Y. Zhang, J. Yin J and Y. Cao (2016). Efficient kd-tree construction for ray tracing using ray distribution sampling. Multimedia Tools and Applications. 75:23. (15881-15899).
Další literaturu dodá vedoucí práce.

Appendix D

Tables

This appendix contains all tables made from the testing statistics. The following is a list of statistics used in the tables and their full description:

- N_S: Total count of static primitives in the scene
- N_D: Total count of dynamic primitives in the scene
- N_Q: Total count of queries (rays)
- N_n: Total count of nodes in the k-d tree(s)
- R_{I2L}: Ratio of inner nodes to leaf nodes
- R_E: Ratio of leaves with primitives to empty leaves
- N_{AP}: Average count of primitives in a leaf
- N_{MP}: Maximum count of primitives in a leaf
- D_{AVG}: Average depth of the k-d tree(s)
- D_{MAX}: Maximum depth of the k-d tree(s)
- C: Cost of the tree according to the full cost model
- N_{IT}: Average count of intersections per ray
- N_{TR}: Average count of traversal steps per ray
- $T_{B,S}$: Total amount of time needed to build the static k-d tree, including creating instances, copying nodes, etc.
- $T_{B,D}$: Total amount of time needed to build the dynamic k-d tree
- T_{TP}: Time taken for the task pool to finish
- $PERF$: Ray tracing performance in mega rays per second (MR = mega ray)
- T_Q: Average time taken to finish tracing a ray
- T_F: Average time taken to build the data structure and finish rendering the scene

D.1 Static scenes

Bistro ($N_S = 3878729, N_Q : 24602534, D_{MAX} = 29$)

Config	N_n	R_{I2L}	R_E	N_{AP}	N_{MP}	D_{AVG}	C
CPU exact	3740437	1.00	10.47	9.74	464	26.50	480.11
CPU approx	4171557	1.00	8.29	9.67	439	26.64	494.78
GPU approx	4171171	1.00	8.29	9.67	439	26.63	494.70

Table D.1 Bistro scene k-d tree statistics

Config	$T_{B,D}$ (ms)	T_{TP} (ms)	N_R ($\cdot 10^{-6}$)	N_{SR} ($\cdot 10^{-6}$)	M (MB)
CPU exact	61750.53	—	—	—	411.064
CPU approx	45133.57	—	—	—	456.219
GPU approx	837.59	413.21	11.483	9.785	818.211

Table D.2 Bistro scene k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU exact	56.57	17.68	183.58	224.18	62.19
CPU approx	59.48	16.81	195.13	239.69	45.55
GPU approx	91.82	10.89	195.11	239.67	1.11

Table D.3 Bistro scene ray tracing statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}
Stack	22.48	44.48	195.10	225.79
Restart	113.16	8.84	195.10	834.95
PushDown	115.65	8.65	195.11	766.18
RestartShortStack (4)	92.47	10.81	195.11	241.88
RestartShortStack (8)	69.56	14.38	195.11	226.52
PushDownShortStack (4)	91.33	10.95	195.11	239.67
PushDownShortStack (8)	67.76	14.76	195.11	226.46

Table D.4 Bistro traversal algorithm statistics

Buddha ($N_S = 1087474, N_Q : 9653466, D_{MAX} = 27$)

Config	N_n	R_{I2L}	R_E	N_{AP}	N_{MP}	D_{AVG}	C
CPU exact	753815	1.00	5.50	4.51	41	21.23	410.48
CPU approx	1998349	1.00	4.10	4.17	55	23.65	426.78
GPU approx	1998265	1.00	4.10	4.17	55	23.65	426.80

Table D.5 Buddha scene k-d tree statistics

Config	$T_{B,D}$ (ms)	T_{TP} (ms)	N_R ($\cdot 10^{-6}$)	N_{SR} ($\cdot 10^{-6}$)	M (MB)
CPU exact	8756.46	—	—	—	51.339
CPU approx	15092.81	—	—	—	130.692
GPU approx	342.71	152.17	1.405	4.237	311.510

Table D.6 Buddha scene k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU exact	99.39	10.06	9.80	26.87	8.85
CPU approx	81.84	12.22	10.09	29.67	15.21
GPU approx	723.06	1.38	10.09	29.67	0.36

Table D.7 Buddha scene ray tracing statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}
Stack	237.19	4.22	10.09	28.63
Restart	781.04	1.28	10.09	96.47
PushDown	836.12	1.20	10.09	79.94
RestartShortStack (4)	647.12	1.55	10.09	30.13
RestartShortStack (8)	557.86	1.79	10.09	28.66
PushDownShortStack (4)	651.91	1.53	10.09	29.67
PushDownShortStack (8)	552.04	1.81	10.09	28.65

Table D.8 Buddha traversal algorithm statistics

Conference ($N_S = 124619, N_Q : 24883200, D_{MAX} = 23$)

Config	N_n	R_{I2L}	R_E	N_{AP}	N_{MP}	D_{AVG}	C
CPU exact	337411	1.00	4.03	2.98	101	21.58	147.96
CPU approx	335355	1.00	2.98	3.44	123	21.58	154.03
GPU approx	336283	1.00	2.98	3.44	123	21.58	153.96

Table D.9 Conference scene k-d tree statistics

Config	$T_{B,D}$ (ms)	T_{TP} (ms)	N_R ($\cdot 10^{-6}$)	N_{SR} ($\cdot 10^{-6}$)	M (MB)
CPU exact	2722.47	—	—	—	18.843
CPU approx	1945.62	—	—	—	19.972
GPU approx	129.24	28.41	0.219	0.712	177.437

Table D.10 Conference scene k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU exact	213.68	4.68	15.41	51.16	2.84
CPU approx	286.12	3.49	16.22	58.00	2.03
GPU approx	923.18	1.08	16.14	58.13	0.16

Table D.11 Conference scene ray tracing statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}
Stack	240.06	4.17	16.14	56.06
Restart	773.57	1.29	16.14	172.10
PushDown	944.86	1.06	16.14	131.77
RestartShortStack (4)	939.31	1.06	16.14	58.65
RestartShortStack (8)	613.96	1.63	16.14	56.11
PushDownShortStack (4)	932.11	1.07	16.14	58.13
PushDownShortStack (8)	600.68	1.66	16.14	56.10

Table D.12 Conference traversal algorithm statistics

FairyForest ($N_S = 174117, N_Q : 24883194, D_{MAX} = 23$)

Config	N_n	R_{I2L}	R_E	N_{AP}	N_{MP}	D_{AVG}	C
CPU exact	355597	1.00	2.93	3.35	161	21.79	143.98
CPU approx	445717	1.00	2.85	3.22	138	21.64	144.58
GPU approx	445751	1.00	2.85	3.22	138	21.64	144.41

Table D.13 FairyForest scene k-d tree statistics

Config	$T_{B,D}$ (ms)	T_{TP} (ms)	N_R ($\cdot 10^{-6}$)	N_{SR} ($\cdot 10^{-6}$)	M (MB)
CPU exact	3008.70	—	—	—	20.905
CPU approx	2598.70	—	—	—	25.757
GPU approx	143.95	36.37	0.323	0.936	186.027

Table D.14 FairyForest scene k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU exact	176.03	5.68	11.64	52.94	3.15
CPU approx	207.43	4.82	10.83	58.64	2.72
GPU approx	997.78	1.00	10.81	58.70	0.17

Table D.15 FairyForest scene ray tracing statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}
Stack	254.28	3.93	10.81	56.87
Restart	845.45	1.18	10.81	161.07
PushDown	945.97	1.06	10.81	138.03
RestartShortStack (4)	1029.84	0.97	10.81	59.04
RestartShortStack (8)	650.53	1.54	10.81	56.91
PushDownShortStack (4)	1020.29	0.98	10.81	58.70
PushDownShortStack (8)	643.83	1.55	10.81	56.91

Table D.16 FairyForest traversal algorithm statistics

Field ($N_S = 3669924, N_Q : 19286028, D_{MAX} = 29$)

Config	N_n	R_{I2L}	R_E	N_{AP}	N_{MP}	D_{AVG}	C
CPU exact	4247837	1.00	2.34	9.70	755	26.91	315.52
CPU approx	4277555	1.00	2.13	9.86	825	26.93	328.92
GPU approx	4278041	1.00	2.13	9.86	825	26.93	328.82

Table D.17 Field scene k-d tree statistics

Config	$T_{B,D}$ (ms)	T_{TP} (ms)	N_R ($\cdot 10^{-6}$)	N_{SR} ($\cdot 10^{-6}$)	M (MB)
CPU exact	49109.90	—	—	—	465.652
CPU approx	31839.25	—	—	—	474.410
GPU approx	831.98	423.73	10.232	10.387	984.727

Table D.18 Field scene k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU exact	65.78	15.20	38.89	76.30	49.40
CPU approx	67.64	14.78	40.59	80.97	32.12
GPU approx	252.35	3.96	40.58	80.97	0.91

Table D.19 Field scene ray tracing statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}
Stack	83.66	11.95	40.58	77.50
Restart	273.10	3.66	40.58	275.96
PushDown	288.39	3.47	40.58	224.73
RestartShortStack (4)	255.54	3.91	40.58	82.21
RestartShortStack (8)	184.11	5.43	40.58	77.71
PushDownShortStack (4)	251.35	3.98	40.58	80.97
PushDownShortStack (8)	178.42	5.60	40.58	77.64

Table D.20 Field traversal algorithm statistics

Powerplant ($N_S = 12701147, N_Q : 17026812, D_{MAX} = 31$)

Config	N_n	R_{I2L}	R_E	N_{AP}	N_{MP}	D_{AVG}	C
CPU exact	8834903	1.00	9.58	8.13	504	27.99	197.32
CPU approx	10956153	1.00	5.29	9.08	324	28.71	201.84
GPU approx	10955423	1.00	5.29	9.08	324	28.71	201.79

Table D.21 Powerplant scene k-d tree statistics

Config	$T_{B,D}$ (ms)	T_{TP} (ms)	N_R ($\cdot 10^{-6}$)	N_{SR} ($\cdot 10^{-6}$)	M (MB)
CPU exact	132673.95	—	—	—	857.070
CPU approx	104345.23	—	—	—	1146.268
GPU approx	2468.47	1265.33	48.031	25.922	1782.876

Table D.22 Powerplant scene k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU exact	49.26	20.30	110.16	147.70	133.02
CPU approx	46.06	21.71	110.11	158.17	104.71
GPU approx	64.87	15.42	110.04	158.16	2.73

Table D.23 Powerplant scene ray tracing statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}
Stack	38.92	25.70	110.04	143.18
Restart	112.10	8.92	110.04	594.91
PushDown	113.71	8.79	110.04	575.84
RestartShortStack (4)	65.82	15.19	110.04	158.40
RestartShortStack (8)	64.87	15.41	110.04	144.18
PushDownShortStack (4)	65.88	15.18	110.04	158.16
PushDownShortStack (8)	64.35	15.54	110.04	144.17

Table D.24 Powerplant traversal algorithm statistics

SanMiguel ($N_S = 9963191, N_Q : 24883196, D_{MAX} = 30$)

Config	N_n	R_{I2L}	R_E	N_{AP}	N_{MP}	D_{AVG}	C
CPU exact	7318341	1.00	7.40	8.24	494	27.13	560.50
CPU approx	8270597	1.00	6.38	8.36	530	27.34	565.42
GPU approx	8270455	1.00	6.37	8.36	530	27.34	565.35

Table D.25 SanMiguel scene k-d tree statistics

Config	$T_{B,D}$ (ms)	T_{TP} (ms)	N_R ($\cdot 10^{-6}$)	N_{SR} ($\cdot 10^{-6}$)	M (MB)
CPU exact	117197.53	—	—	—	716.747
CPU approx	82663.75	—	—	—	817.566
GPU approx	1844.33	890.69	27.925	19.215	1271.952

Table D.26 SanMiguel scene k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU exact	68.20	14.66	125.95	155.66	117.56
CPU approx	69.05	14.48	125.76	160.06	83.02
GPU approx	129.41	7.73	125.75	160.04	2.04

Table D.27 SanMiguel scene ray tracing statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}
Stack	35.59	28.10	125.75	152.16
Restart	168.43	5.94	125.75	530.26
PushDown	171.71	5.82	125.75	484.37
RestartShortStack (4)	128.89	7.76	125.75	161.46
RestartShortStack (8)	99.87	10.01	125.75	152.54
PushDownShortStack (4)	128.30	7.79	125.75	160.04
PushDownShortStack (8)	98.54	10.15	125.75	152.48

Table D.28 SanMiguel traversal algorithm statistics

Sibenik ($N_S = 75284, N_Q : 24883144, D_{MAX} = 22$)

Config	N_n	R_{I2L}	R_E	N_{AP}	N_{MP}	D_{AVG}	C
CPU exact	304177	1.00	4.56	2.02	62	20.30	170.45
CPU approx	381183	1.00	3.09	2.44	54	20.63	180.73
GPU approx	381253	1.00	3.09	2.44	54	20.63	180.63

Table D.29 Sibenik scene k-d tree statistics

Config	$T_{B,D}$ (ms)	T_{TP} (ms)	N_R ($\cdot 10^{-6}$)	N_{SR} ($\cdot 10^{-6}$)	M (MB)
CPU exact	2180.09	—	—	—	14.655
CPU approx	2131.99	—	—	—	19.645
GPU approx	128.53	29.99	0.206	0.785	176.644

Table D.30 Sibenik scene k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU exact	249.92	4.00	15.89	92.51	2.28
CPU approx	229.71	4.35	14.91	100.85	2.24
GPU approx	673.02	1.49	14.91	100.86	0.17

Table D.31 Sibenik scene ray tracing statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}
Stack	190.26	5.26	14.91	96.44
Restart	458.96	2.18	14.91	312.91
PushDown	546.62	1.83	14.91	260.05
RestartShortStack (4)	682.08	1.47	14.91	102.10
RestartShortStack (8)	420.67	2.38	14.91	96.57
PushDownShortStack (4)	685.03	1.46	14.91	100.86
PushDownShortStack (8)	416.43	2.40	14.91	96.55

Table D.32 Sibenik traversal algorithm statistics

Sponza ($N_S = 262267, N_Q : 32845488, D_{MAX} = 24$)

Config	N_n	R_{I2L}	R_E	N_{AP}	N_{MP}	D_{AVG}	C
CPU exact	1050685	1.00	3.67	2.25	63	22.75	251.75
CPU approx	1137767	1.00	3.13	2.54	82	22.84	266.29
GPU approx	1137805	1.00	3.13	2.54	82	22.84	266.01

Table D.33 Sponza scene k-d tree statistics

Config	$T_{B,D}$ (ms)	T_{TP} (ms)	N_R ($\cdot 10^{-6}$)	N_{SR} ($\cdot 10^{-6}$)	M (MB)
CPU exact	8221.02	—	—	—	52.499
CPU approx	6690.59	—	—	—	59.485
GPU approx	184.01	83.80	0.737	2.354	233.674

Table D.34 Sponza scene k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU exact	106.29	9.41	30.66	143.49	8.53
CPU approx	105.98	9.44	34.02	159.62	7.00
GPU approx	286.41	3.49	34.04	159.63	0.30

Table D.35 Sponza scene ray tracing statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}
Stack	98.03	10.20	34.04	150.15
Restart	250.26	4.00	34.04	530.52
PushDown	259.85	3.85	34.04	489.74
RestartShortStack (4)	294.40	3.40	34.04	160.72
RestartShortStack (8)	194.10	5.15	34.04	150.39
PushDownShortStack (4)	294.06	3.40	34.04	159.63
PushDownShortStack (8)	192.92	5.18	34.04	150.38

Table D.36 Sponza traversal algorithm statistics

Street ($N_S = 961430, N_Q : 24176866, D_{MAX} = 26$)

Config	N_n	R_{I2L}	R_E	N_{AP}	N_{MP}	D_{AVG}	C
CPU exact	639655	1.00	6.61	7.74	132	24.31	19.40
CPU approx	654381	1.00	4.99	9.37	196	24.48	20.40
GPU approx	654481	1.00	5.00	9.36	196	24.48	20.40

Table D.37 Street scene k-d tree statistics

Config	$T_{B,D}$ (ms)	T_{TP} (ms)	N_R ($\cdot 10^{-6}$)	N_{SR} ($\cdot 10^{-6}$)	M (MB)
CPU exact	9263.74	—	—	—	60.054
CPU approx	6087.83	—	—	—	69.967
GPU approx	231.47	68.30	1.148	1.565	260.519

Table D.38 Street scene k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU exact	99.71	10.03	31.65	67.62	9.51
CPU approx	94.54	10.58	33.11	69.39	6.34
GPU approx	583.60	1.71	33.13	69.40	0.27

Table D.39 Street scene ray tracing statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}
Stack	142.04	7.04	33.12	67.38
Restart	542.52	1.84	33.12	212.27
PushDown	599.05	1.67	33.12	169.92
RestartShortStack (4)	588.02	1.70	33.13	70.37
RestartShortStack (8)	395.18	2.53	33.13	67.42
PushDownShortStack (4)	588.26	1.70	33.13	69.40
PushDownShortStack (8)	391.73	2.55	33.13	67.41

Table D.40 Street traversal algorithm statistics

D.2 Dynamic scenes

Bistro ($N_S = 3878729, N_D = 99037, N_Q : 24603957$)

Config	$T_{B,S}$ (ms)	$T_{B,D}$ (ms)	T_{TP} (ms)
CPU single	0.00	45684.82	—
CPU double	61711.01	439.41	—
CPU merge	37382.95	23830.65	—
GPU single	0.00	844.88	415.97
GPU double	788.12	98.36	11.47

Table D.41 Bistro scene dynamic k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU single	54.35	18.40	193.88	239.90	46.14
CPU double	121.69	8.22	184.01	228.14	0.64
CPU merge	55.73	17.94	183.77	227.31	24.27
GPU single	92.08	10.86	193.92	239.80	1.11
GPU double	90.80	11.01	195.52	243.44	0.37

Table D.42 Bistro scene dynamic ray tracing statistics

FairyForest ($N_S = 0, N_D = 174117, N_Q : 24883194$)

Config	$T_{B,S}$ (ms)	$T_{B,D}$ (ms)	T_{TP} (ms)
CPU single	0.00	2487.39	—
CPU double	0.36	2437.91	—
CPU merge	0.00	2542.29	—
GPU single	0.00	129.13	34.90
GPU double	0.06	128.16	35.09

Table D.43 FairyForest scene dynamic k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU single	203.80	4.91	11.11	59.46	2.61
CPU double	237.33	4.21	11.16	60.00	2.54
CPU merge	189.35	5.28	11.12	59.71	2.67
GPU single	882.99	1.13	11.11	59.55	0.16
GPU double	852.78	1.17	11.14	59.73	0.16

Table D.44 FairyForest scene dynamic ray tracing statistics

Field ($N_S = 3669924, N_D = 10494, N_Q : 19306559$)

Config	$T_{B,S}$ (ms)	$T_{B,D}$ (ms)	T_{TP} (ms)
CPU single	0.00	32054.33	—
CPU double	48785.57	49.44	—
CPU merge	31455.25	16546.50	—
GPU single	0.00	822.36	422.19
GPU double	767.71	111.78	5.85

Table D.45 Field scene dynamic k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU single	70.67	14.15	40.61	82.80	32.33
CPU double	292.12	3.42	38.92	77.19	0.12
CPU merge	69.05	14.48	38.90	78.49	16.83
GPU single	252.00	3.97	40.61	82.74	0.90
GPU double	250.98	3.98	40.60	81.85	0.19

Table D.46 Field scene dynamic ray tracing statistics

Sibenik1 ($N_S = 75284, N_D = 6658, N_Q : 24883144$)

Config	$T_{B,S}$ (ms)	$T_{B,D}$ (ms)	T_{TP} (ms)
CPU single	0.01	2097.86	—
CPU double	1733.11	127.81	—
CPU merge	529.51	1544.07	—
GPU single	0.00	113.84	30.19
GPU double	92.19	103.33	5.05

Table D.47 Sibenik1 scene dynamic k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU single	222.67	4.49	15.42	99.83	2.21
CPU double	758.95	1.32	15.93	92.80	0.16
CPU merge	391.50	2.55	16.10	94.86	1.61
GPU single	641.06	1.56	15.32	101.30	0.15
GPU double	670.49	1.49	15.04	101.88	0.14

Table D.48 Sibenik1 scene dynamic ray tracing statistics

Sibenik2 ($N_S = 75284, N_D = 13316, N_Q : 24883144$)

Config	$T_{B,S}$ (ms)	$T_{B,D}$ (ms)	T_{TP} (ms)
CPU single	0.00	2131.95	—
CPU double	1714.91	230.50	—
CPU merge	535.40	1732.95	—
GPU single	0.00	123.39	31.07
GPU double	92.22	106.80	6.67

Table D.49 Sibenik2 scene dynamic k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU single	230.41	4.34	15.32	102.55	2.24
CPU double	748.17	1.34	16.04	93.75	0.26
CPU merge	285.64	3.50	16.12	96.13	1.82
GPU single	619.64	1.61	15.38	103.58	0.16
GPU double	658.79	1.52	15.16	103.04	0.14

Table D.50 Sibenik2 scene dynamic ray tracing statistics

Sibenik3 ($N_S = 75284, N_D = 19974, N_Q : 24883144$)

Config	$T_{B,S}$ (ms)	$T_{B,D}$ (ms)	T_{TP} (ms)
CPU single	0.00	2136.12	—
CPU double	1743.84	351.98	—
CPU merge	533.34	1889.51	—
GPU single	0.00	127.93	31.04
GPU double	91.90	99.16	8.29

Table D.51 Sibenik3 scene dynamic k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU single	232.53	4.30	15.77	106.79	2.24
CPU double	569.49	1.76	16.14	95.17	0.40
CPU merge	252.29	3.96	16.16	98.61	1.99
GPU single	600.99	1.66	15.81	106.45	0.17
GPU double	647.29	1.54	15.22	104.03	0.14

Table D.52 Sibenik3 scene dynamic ray tracing statistics

Street ($N_S = 961430, N_D = 198312, N_Q : 24177507$)

Config	$T_{B,S}$ (ms)	$T_{B,D}$ (ms)	T_{TP} (ms)
CPU single	0.00	9067.82	—
CPU double	9078.73	1906.54	—
CPU merge	5209.54	5625.74	—
GPU single	0.00	277.62	98.97
GPU double	174.85	119.42	24.65

Table D.53 Street scene dynamic k-d tree build statistics

Config	PERF (MR/s)	T_Q (ns)	N_{IT}	N_{TR}	T_F (s)
CPU single	98.83	10.12	34.08	76.31	9.31
CPU double	206.10	4.85	32.28	70.41	2.02
CPU merge	147.54	6.78	33.20	73.90	5.79
GPU single	563.32	1.78	33.63	75.48	0.32
GPU double	574.16	1.74	33.76	72.03	0.16

Table D.54 Street scene dynamic ray tracing statistics