MASTER IN COMPUTER SCIENCE PROFESSIONAL FOCUS IN SOFTWARE ENGINEERING

Study of Data Structures for Ray Tracing Acceleration

DEVILLERS, Hugo

Award date:
2020

Awarding institution:
University of Namur

UNIVERSITÉ DE NAMUR

FACULTÉ D'INFORMATIQUE

Study of Data Structures for Ray Tracing Acceleration

Hugo Devillers

Acknowledgments

1. Prof. Bruno Dumas from the University of Namur, for having greenlit this unusual MSc thesis topic and allowing me to work on what I was interested in, even though it meant going outside of his usual research area.
2. The Computer Graphics group at the University of Saarland in Saarbrücken Germany, for having so kindly welcomed me, including during lockdown and beyond the original time frame of this thesis.
3. Arsène in particular, for being a pretty kick-ass supervisor and making himself so available to my questions.
4. My family for their continued support during not only the writing of this thesis, but all of my education.

Abstract (In English)

In computer graphics, we use specific data structures to accelerate the search of intersections between rays and the 3D models that make up scenes (Ray tracing). Over the years, numerous techniques have been developed to accelerate both the computation of ray casts, but also the build times of the acceleration structure itself, a factor that becomes important in real-time applications. In this thesis, we will discuss the theory behind such structures and explore the consequent prior work on the topic. Beyond merely doing a comparative study, we will provide insight into the performance factors behind the real-world performance of such structures.

Résumé (En Français)

Dans le domaine de l'imagerie par ordinateur, on utilise des structures de données spécifiques pour accélérer la recherche d'intersections entre des rayons et les modèles 3D d'une scène (Lancer de rayon). De nombreux efforts ont étés consacrés pour développer les structures plus optimales possibles, maximisant non seulement la vitesse de calcul des lancers de rayon, mais aussi minimisant les coûts de construction des structures elle-mêmes, un facteur important lorsqu'on est dans une application interactive. Dans ce mémoire, nous explorerons la théorie de ces structures et nous étudieront la littérature disponible sur le sujet. Au delà d'un simple comparatif de performances, nous fourniront une analyse en profondeur des facteurs derrière l'efficacité de telles structures.

Contents

Chapter 1

Thesis Introduction

1.1 Motivation

Tracing (casting) rays through a scene and looking for the intersections with the geometry is a big area of interest in the field of Computer Graphics. Among other applications in collision detection for example, the most common use case of ray tracing is in realistic rendering to simulate the behavior of light. Performing these ray intersection queries naively in linear time is inefficient and so tremendous work has been put into making those operations as fast as possible.

Accelerating the computation of those ray queries is done by using data structures specifically designed and optimized for that purpose. In the literature, those are colloquially called Acceleration Structures, though it might be more correct to use the terminology Acceleration Data Structures to reflect the fact these are *data structures* for *accelerating* ray tracing operations.

1.2 The research question

In this thesis, we will attempt to answer the following question: *What makes a good acceleration data structure ?*

In other words, we are interested in acceleration data structures that have real-world applicability with great measured performance, not just the best theoretical complexity. We are also interested in the factors that drive that real-world performance, to explain *why* and *how* a structure achieves its results.

We argue that answering this question comprehensively is especially important in 2020 as we are currently at the start of a paradigm shift: With the release of NVidia Turing GPUs [8] featuring dedicated hardware units and the accompanying mainstream API support [53], new use cases for ray tracing are rendered viable, challenging the established status-quo for rendering in real-time [43] but also providing useful acceleration that can be used in other areas [42] [51].

This hardware acceleration however represents a shift away from programmers interacting directly with the ray tracing acceleration data structures themselves. In the current implementations the APIs to use that acceleration presents the user with a black box, this means that the choice made by the hardware vendor carries a lot of weight and performance implications, and the fixed-function hardware implementation makes this choice even more important to get right.

Putting together the considerable existing interest in ray tracing prior to 2018 [27], the additional use cases for it enabled by hardware acceleration, the finality of putting the decision of which structure to use in hardware and the potential for new structures that can be designed in concert with their fixed-function accelerators, we believe now is a very interesting time to be studying acceleration data structures.

1.3 Structure of this text

In Chapter 2 we will examine the theory behind acceleration data structures, what are the characteristics of the queries we have to answer, the paths we can take when designing such a structure, as well as the hardware context they must work within. Next, in Chapter 3, we will proceed to review a number of relevant prior works and discuss how they fit in with the previous discussion.

Afterwards, in Chapter 4, we will proceed to describe our implementation of the previously reviewed techniques in our own framework, which we will then use to analyze their behavior against a variety of scenes. In Chapter 5 we will take the data we obtained previously to build out our understanding of acceleration data structures and answer the original question qualitatively. Finally in Chapter 6 we will present our final conclusions and possible future work.

1.4 Limitations and scope of this thesis

Given the scope of a master's thesis, this work could not consider every paper written on the topic, and we had to put a limit to the scope of our work.

1. We settled on studying only the *single ray-tracing* case, and purposefully did not investigate packet tracing, frustrum tracing or any other generalized ray tracing method.

2. We did all our testing on the CPU, with algorithms written in C++17, without using vector intrinsics. We did not port our traversal algorithms to the GPU.

3. While we have numbers for the build times, we did not stress about dynamic rebuilds, and our code only performed acceleration data structure builds at initialization. We mostly focused on the traversal performance and the factors underlying it.

4. Lastly, we ran out of time to implement some of the more interesting structures [38] and some of our original ideas. We believe this will be best addressed in future work.

Nevertheless, we believe this work to be an useful reference for anyone interested in the topic of acceleration data structures.

Chapter 2

Principles of Acceleration Data Structures

2.1 Ray tracing operations

Tracing (or casting) a ray is a geometric query in the form of:

> *Given this starting location x and this direction ω, what object in the scene will we hit, if any, and at what point along the ray ?*

In this context, object means some geometrical shape, a mathematical primitive that we can intersect with a line, for example a triangle or a sphere. A scene is a collection of such objects, and can be organised in multiple so-called models, often representing real-world objects. For example, a scene containing a chair and a table where each can be modelled with a few boxes.

A mathematical notation for this is given below. (2.1) is the distance function to the closest solid point along the ray, and (2.2) yields the point itself. The letter t is usually used for the distance traveled along the ray, referred to as *time* along the ray.

$$d(x, \omega) = \inf\{t | x + \omega t \in S\} \tag{2.1}$$

$$c(x, \omega) = x + \omega d(x, \omega) \tag{2.2}$$

Where S is the set of points belonging to the surfaces of the objects in the scene.

In general, we are only interested in a restricted range of possible t distances, so a ray tracing routine typically takes a $[t_{min}, t_{max}]$ range for t. Any intersection that does not fit inside that range will not be reported. Typically t_{min} is zero or a strictly positive number, since negative t would correspond to intersections *behind* the ray origin.

2.1.1 Nearest-hit and Any-hit ray queries

By default, we want to know the nearest object a ray will hit, this is known as a "nearest-hit" ray.

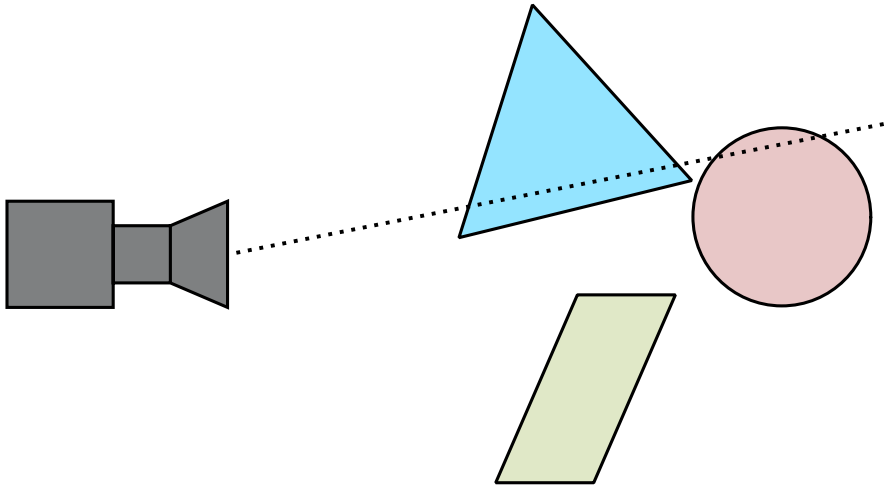


Figure 2.1: A camera ray intersecting objects in a scene

Sometimes, we do not care about what we hit, or the precise value of t. In those cases the ray query is essentially just a binary test to see if two points are mutually unoccluded, or if a ray can escape the scene without touching anything. These are known as "any-hit" rays. In the rendering literature those rays are commonly referred to as *shadow* rays due to their common use to calculate if an area is in the shade from a light.

2.1.2 Rays coherence

An important consideration when tracing many rays, is whether or not the rays are *coherent*. Coherent rays have a similar origin and direction, a classic example in rendering are primary rays, rays that go from a camera (a point) into a general direction (forming a frustum). Coherent rays enjoy better performance, because they are more friendly to caches and vectorization, which we will go into in more detail below. We give an example of coherent rays in Figure 2.3.

2.2 Applications of ray tracing

Easily the most popular application of ray tracing is for rendering photorealistic images. Algorithms such as Path Tracing [22] simulate the behavior of light in a scene accurately, allowing for not only good looking but physically accurate images to be created. In other, non-photorealistic, forms of rendering like information visualization and computer aided design, we can also use rays to generate images. Ray tracing has also other applications in simulating for example the behavior of sound, or in collision

detection.

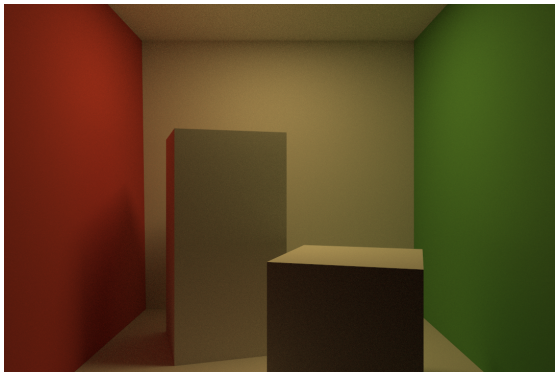


Figure 2.2: A path traced render of a classic scene known as a “cornell box”

In this thesis we focused on rendering as the main application and benchmark for ray tracing performance. For the purposes of studying the ray tracing operation itself, we can distinguish three type of rays:

- Primary rays, which are the coherent rays going from the camera position to the scene, passing through the viewport.
- Secondary or bounce rays, which are the non-coherent rays resulting from primary rays bouncing off objects in many directions
- Shadow rays, which are also incoherent but are any-hit rays.

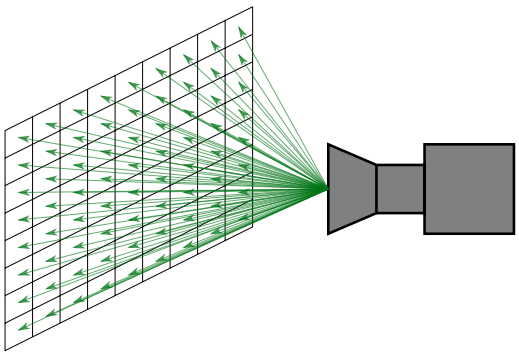


Figure 2.3: Example of primary rays in rendering. Rays (in green), go from the camera to the center of each pixel in the to-be rendered image. These are very coherent rays: Each ray will more or less hit what its neighbor will.

2.2.1 Going off on a tangent: Rasterization

Performing ray-tracing against a scene is solving a specific variant of the wider visibility problem [4], specifically the "visibility along a line" subproblem. In certain cases, rendering tasks can be mapped to another subproblem, the "visibility from a point" variant, using the technique known as rasterization.

Rasterization is an ubiquitous method for performing primary visibility in real-time graphics. It can be seen as a very efficient alternative to ray-tracing for primary rays. In a way it is like ray tracing, but done in a different order: while in ray-tracing you normally iterate first over the rays (and thus the pixels) then iterate over the object to find what hit each ray, here you instead start by iterating over the objects and check which rays (pixels) it will hit. This only works when the rays are all parallel, or in the pattern like Figure 2.3, if the rays are not arranged like so they need to be traced individually.

In a more intuitive sense, rasterization works by "drawing" the objects on the screen in the spots they should occupy. Figure 2.4 shows us how this looks.

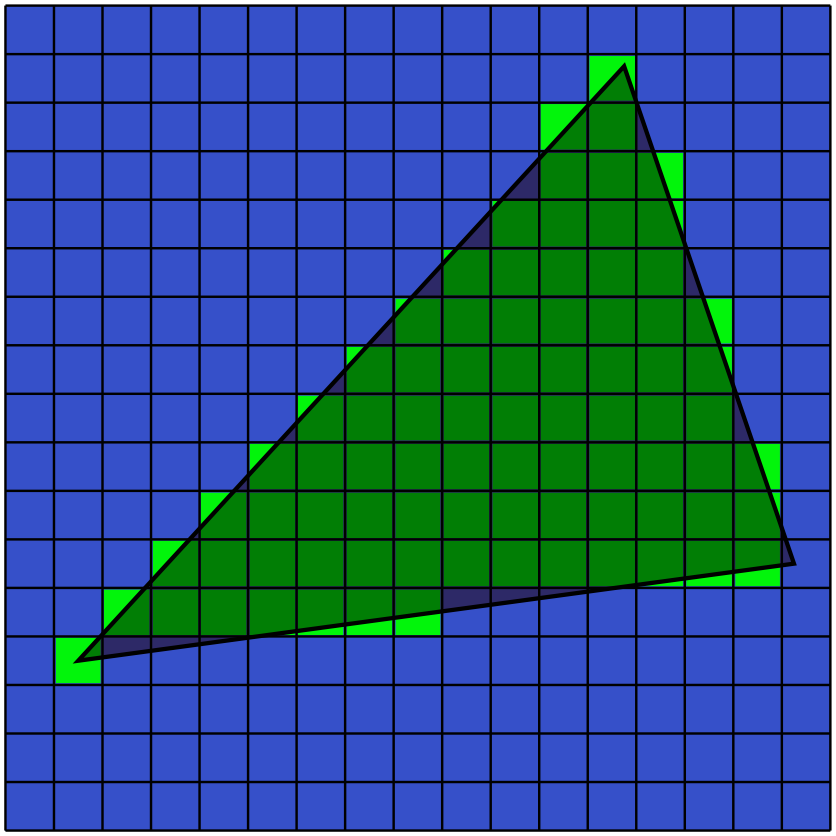


Figure 2.4: A rasterized triangle. In this instance the green pixels are the ones where we consider their ray hit the triangle.

However, drawing the objects in no particular order is equivalent to doing an

"any-hit" ray cast. In order to know what is the *closest* object for each ray (pixel), we need to perform *Visible Surface Determination*, to avoid drawing the further objects on top of the closer ones.

To perform this work we can rely on algorithms that sort the objects based on their proximity to the camera, such as the Painter's algorithm, which simply draws the objects back to front, overwriting the data of occluded objects. However this is wasteful computationally, and worse cannot deal with some configurations of objects such as in fig. 2.5.

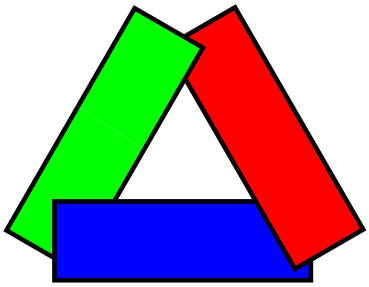


Figure 2.5: There is no order of drawing these 3 shapes that capture this overlapping configuration without splitting them.

Rather than object-sorting approaches, the most widely used hidden surface removal technique for rasterization is the *z-Buffer* or depth buffer method, where you add an additional buffer responsible for storing depth values. Each time a pixel of the image is written to, the distance from the surface and the camera (the "depth" of the pixel) is compared against the value stored in the buffer. If it is less (therefore closer to the camera), the new pixel data overwrites the old one and updates the depth buffer, otherwise it is dropped and the image remains unchanged.

While we will not use rasterization in this work directly, it is important to point it out as an alternative to tracing rays, and we will mention it again in Chapter 5 when we discuss real-time rendering applications.

2.3 Motivation for acceleration data structures

Answering nearest-hit and any-hit queries naively by checking the ray against every object would scale linearly with the amount of geometric complexity in our scenes. In big O notation, this is written as $O(N)$. We can intuitively see this as sub-optimal: picture a bathroom scene with the bathtub filled with detailed, computationally expensive to intersect, rubber ducks (Figure 2.6).

A ray that is going by the tub without touching it has no reason to perform computations against any of the ducks. In that case, we could program the ray intersection routines such that if a ray isn't going to touch the bath tub itself, none of

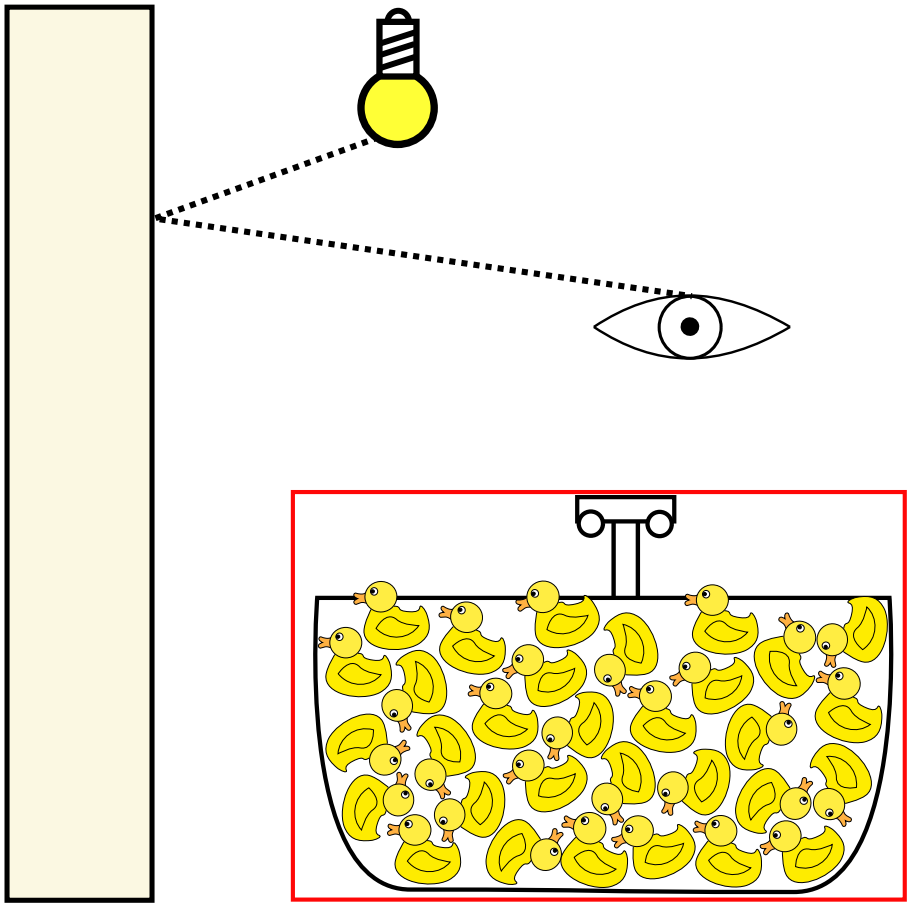


Figure 2.6: The two rays (dotted lines) making up the light path does not intersect with the tub (bounded by the red square), therefore it is unnecessary to intersect any of the ducks within it.

the geometrically complex little ducks contained within should be intersected, saving a lot of time.

When designing an acceleration data structure, we are trying to generalize those optimizations programmatically and creating a data structure to store the scene in such a way that those efficiency-boosting culling decisions can be taken the most effectively. Beyond just culling, such structures are in fact about lowering the overall cost of intersecting, by reformulating the linear process of checking each object for intersections, into a sub-linear search for intersections.

2.4 Performance factors in modern hardware

Achieving the lowest time complexity in terms of big O notation is not a sufficient or even strictly required condition to performing best in real world scenes on real hardware. While its a good indicator of asymptotic performance and can give some context to the observed performance characteristics, it also fails to capture many traits

that are very relevant to real implementations, in particular the effects of memory latency and cache behaviors.

In this section we will thus discuss the performance factors that affect real-world renderers and more generally high performance applications.

2.4.1 Memory access

It is a widely known fact that memory speeds did improve at the same rate as CPUs performance did [7], already in the 1980s operations could be carried out faster than memory could fetch and store the associated data. Two approaches exist to cope with this latency issue: historically we have used caching to reduce the average latency, but with the advent of GPUs that approach has been complemented with hiding the latency through massive parallelism.

Caching

The idea of a cache in computing is to be a comparatively smaller and faster piece of storage, placed in between a storage consumer and the storage itself. A cache is to store a sub-set of the information contained and act as the middle-man: when a memory request is made, it actually goes to the cache, which might have the relevant address currently loaded and thus be able to serve the request faster. If the data for a given address does not currently reside in the cache, the request is then forwarded to the underlying, slower storage.

In modern computer systems, cache actually forms a neat hierarchy [30]. A typical memory address may be backed by up to three or four physical levels of SRAM caches inside the CPU, then main work memory, then persistent storage and possibly even cold storage, with typically an order of magnitude more latency to access each successive level.

Caching is omnipresent in hardware and software, and absolutely essential to performance. Making effective use of caches is a requirement to unlock the full power of modern processors: while they can issue billions of instructions per second, any main memory access missing the cache completely can come with a stall equivalent to that of hundreds of instructions, far eclipsing even the worst scenarios of branch miss-predictions.

Caches work best when used in an obvious way, such as reading a linear block of memory. Doing so plays to the strengths of caches: after the first few accesses the cache will become populated with the data needed ahead of time, such that no further latency penalty is to be paid. In general the more coherent the memory accesses are, the most likely the caches are to cover the majority of requests and reduce overall latency.

Latency hiding

Latency hiding as found in GPUs is a different way of approaching the issue of memory latency [46]. Rather than putting huge stress on reducing the average latency with caches (though the two approaches are not mutually exclusive) the idea is to accept the latency and design a processor that has always enough work to keep itself occupied for however long the data takes to arrive.

On a GPU this is done by having a relatively big piece of on-board memory called a register file, used to quickly swap from executing one thread to another as soon as a

memory stall occurs. When a long memory operation is issued, the GPU scheduler will look in the register file for another thread that is not currently waiting for anything to arrive, and switch execution to that thread.

For this to be effective, the number of in-flight threads must be a lot greater than the actual execution capabilities per cycle. That number of in-flight threads is bounded by the size of the aforementioned register file, usually in order of magnitude of a couple kilobytes per core capable of executing one thread (Keeping in mind GPU threads are typically scheduled together in waves of 16-64 threads at once).

This means there is a trade-off when writing GPU kernels (programs): The more registers a thread needs, the less of them can be scheduled at once and the worse the latency hiding capabilities become. In more than a few cases it makes sense to trade a few extra instructions to save a couple registers, to maximize the *occupancy* of the GPU.

Memory bandwidth

While less of an immediate concern than latency, memory bandwidth is absolutely a finite resource. Actually reaching the capacity of the theoretical memory bandwidth is possible, but it is rarely the bottleneck in most scenarios [2]. Being memory-bandwidth bound is often the sign of quite an optimal renderer, as memory and instruction latency troubles are much more common to run into.

2.4.2 Performance enhancements in modern CPUs

Contemporary microprocessors operate in very interesting ways. Though a CPU appear to act as a sequential machine executing instructions in order, the actual CPU core takes that traditional instruction flow and decodes it into its own internal representation, performing many and sometimes quite dramatic transformations to it.

Pipelining

First, all modern CPUs have a pipeline: they execute multiple instructions in flight, decomposed in independent stages. At any given clock cycle, different instructions may be concurrently fetched, decoded, executed and their results stored in a register, though in practice pipelines further decompose those steps depending on the specific micro-architecture.

This comes with a few hazards: if an instruction early in the pipeline is a branch, only one of its paths can be taken, and the processor has no choice but to make a guess and predict the result of that branch. If that assumption fails to hold, the pipeline will typically have to be flushed, resulting in a —depending on micro-architecture— significant stall of the CPU.

Out of order execution

Not only are modern CPUs pipelined, they also work out of order. The decoded instructions are analyzed to look for data dependencies: which registers are read and written to and in which order. The CPU is free to re-order instructions as long as it is transparent to the program, usually this enables easing a memory or instruction dependency and preventing stalls by spacing dependent instructions apart from each other.

Out of order execution and speculative execution can combine together to create surprising complications, with sometimes entire branches of speculated execution having to be dropped or rolled back. As an important side note, in recent history some security issues have been discovered where attackers could use the measured latency of memory accesses and speculative execution to access memory illegally.

Superscalar execution

Modern processors are also what is known as ”super scalar”: they can execute multiple instructions at once. Assuming no data dependency, two instructions can be issued at once: for example two integer operations or one integer and one floating point one. Modern processors are such that they will actively look for such occasions to overlap work, going as far as re-interpreting the execution flow to use different registers.

2.4.3 Data and Instruction-level parallelism

Data parallelism is the concept of working on the same piece of data from multiple points at once without dependencies. A data parallel algorithm is enticing because it has no or few requirements of synchronization, and so work can be easily split among many participants.

Instruction-level parallelism (ILP) describes the ability of a processor to execute multiple instructions simultaneously. ILP can also be achieved implicitly with a sequential instruction flow, as seen before with OOO and super-scalar execution.

SIMD

SIMD —Single Instruction Multiple Data— describes instructions that can apply at once the same operation to multiple operands. SIMD is a form of explicit ILP where every sub-operation is data-parallel. A good candidate for SIMD operations is vector and matrix maths we typically encounter in Computer Graphics. All modern CPU architectures now include some explicit SIMD instructions in their set (SSE, AVX, NEON, ...).

SIMT

SIMT - Single Instruction Multiple Threads - is the paradigm used by high performance GPU architectures today. In a SIMT processing unit, a wave of threads are executed together in lock-step and the GPU scheduler is able to fill its wide (16-64 typically) units by executing the same instruction for all threads in the currently executing wave.

Depending on the exact implementation, divergent control flow may require masking (some work will not be written back, effectively disabling execution for a thread), and the execution unit may also be able to deal with independent instruction pointers for each thread. In short, SIMT is able to feed large SIMD units with work from conceptually independent threads, but may suffer from too much divergence in the threads.

Footnote: VLIW

VLIW - Very Long Instruction Word - is an alternative way of achieving ILP once popular in desktop GPUs. Each instruction in a VLIW architecture actually encodes

work for multiple units in a parallel processor, and the burden of scheduling work to execute in parallel while respecting dependencies is placed on the compiler.

In high performance applications VLIW architectures have mostly fallen out of favour due to the difficulty of building such advanced compilers and the poor flexibility of the model, and GPUs have since moved on to SIMT models.

2.4.4 Concurrency

Unlike data parallelism, concurrency is the ability to execute a completely different piece of work in parallel to another.

Modern CPUs offer a number of independent physical cores, each one with its own computational resources. They also offer logical cores, a technique known as Simultaneous Multi Threading (SMT) where unused resources left over from the execution of one instruction flow can be opportunistically used by other logical threads.

GPUs are more oriented towards (massive) data parallelism, but some are also capable of concurrently executing different kernels at once, a capability branded as ”Async Compute” in consumer-level graphics hardware. At a smaller scale, depending on how well the SIMT implementation handles divergence, a GPU can start to look like a many-core processor.

2.4.5 Takeaways

For our purposes of high-performance computing a few things must be kept in mind at all times.

First is making good use of the memory: having a small footprint is better for caches, and the less spread out successive accesses are, the more effective the caches are. Secondly, the algorithms must be as data-parallel as possible: there has to be the fewest amount of mandatory synchronisation and waiting for dependencies, both in the large scale (CPU threads, GPU kernels) and in the small scale for instruction-level parallelism.

Finally, on GPUs the rules are a bit stricter: having too much state (used registers) in each thread limits occupancy and thus performance by making latency hiding less effective. Heavy divergence also hurts GPUs by wasting work and starving execution units for work.

These factors are often played against each other in practice, and it is necessary to make well-educated compromises to get the most out of a processor.

2.5 Qualification of acceleration data structures

All acceleration data structures studied in this thesis observe some form of the divide and conquer approach to search for intersections in sub-linear time. Not all structures divide the same things though, and the way to structure their data means traversing the structures will have different properties. We will also look at the important ones.

2.5.1 Approaches to scene subdivision

Object subdivision is the process of subdividing a scene in disjoint sets of objects. Intersecting primitives in an acceleration structure built on an object subdivision approach has a strong guarantee the same object will never be intersected twice (since

it is only present in one place in the structure). This also means no indirection has to take place, since there is only one copy of the data.

The possibility space for building such sets gets quite large, especially when you consider a recursive/hierarchical structure, but it is typically finite and so could theoretically be explored fully. In practice it never is and we instead rely on binning, greedy algorithms, heuristics and pre/post process steps as we will discuss later in this text.

Space subdivision is the opposite, subdividing a scene in disjoint regions of space that can contain multiple primitives - or none at all. The trade-off is those regions of space might duplicate references to the same primitives, and thus decrease overall efficiency with redundant intersection tests, indirection and more memory usage. The advantage is early ray termination (ERT) can always be applied - see below for an introduction to ERT.

As 3D space is continuous, the possibility space for space division is much larger, in fact infinite if it wasn't for the limited precision of floating point numbers in practice. Approaches to space subdivision typically revolve around discretization of space in a grid structure of some sort.

Hybrid approaches are possible, but require very careful consideration to avoid ending up with the worst of both worlds.

2.5.2 Top down or bottom up construction

When a hierarchical structure is used, the first question for a build algorithm is whether to use a top-down build, where the scene is recursively split, or a bottom-up one, where small portions of the scene are processed individually and are subsequently merged together.

Bottom-up builds are potentially more data-parallel: they are not dependent on any data to start out, but they also lack the global information you can have when starting from the top, so the quality may suffer.

2.5.3 Desirable traversal properties

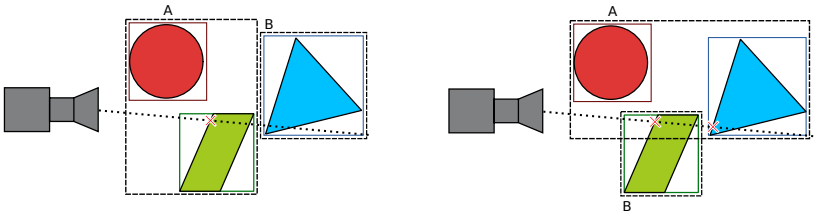
Low memory footprint

As discussed in an earlier section, memory is a huge factor in high performance computing, and in particular is a common bottleneck in ray-casting. In general the lowest the memory footprint for a given structure, the more of it will fit in the caches, and so the more hits will occur, meaning faster processing. Low memory footprint also tends to go hand in hand with less memory accessed, which is also ideal.

Early ray termination

Early ray termination refers to the ability of some acceleration data structures to ”skip” some remaining intersection candidates when an intersection is found that fills some criteria for being the closest to the camera, with no further contenders possible in the unexplored space. In particular, when the intersection is found within a region of space that lies entirely “before” the next one in terms of distance, we can deduce P is the closest hit full stop and end the traversal “early”.

To visualize this, let there be a hypothetical structure that clusters primitives together in named dotted boxes (A and B here). Let's say the traversal algorithm for this structure is to first sort the dotted boxes, and then intersect their contents starting with the closest box. Figure 2.7 shows us two scenarios for ERT applicability:



(a) The closest hit in the A region is eligible for ERT: we don't need to check B for better hits.

(b) The hit we found in A is not closer than the distance to reach B: therefore we must keep going and check for intersections in B.

Figure 2.7: Two scenarios that show when ERT is safe, and when it is not

ERT is applicable in the first case (a), since the distance of the closest hit point in A could not possibly be beaten by any point in B. However, in the second case (b), this is no longer the case, and both the primitives in A and B have to be fully intersected to know for sure where the closest hit is.

This class of optimizations is obviously extremely desirable for all non-shadow rays (shadow rays can always return on first hit by definition), and is commonly found in spatial acceleration structures.

Stackless traversal algorithm

For scheduling, some traversal algorithms employ a stack to store elements to traverse later and achieve ordered traversal, most notably traversal algorithms for tree-like acceleration structures rely on a stack. Having a stack can be quite costly, in particular on the GPU, where the relatively large amount of per-thread memory required for even a modest-sized stack will quickly limit the numbers of threads that can be kept in flux, and therefore limit the latency-hiding capabilities of the GPU.

Many strategies have been devised for minimizing and outright avoiding a stack when performing ray traversal. Three recurring classes of such strategies exist: Strategies that perform traversal in a fixed, non-ray dependent order, strategies that remember the progress made in the traversal and find their way to the last branch took to visit the other side, and strategies that follow "ropes" (pointers) to neighboring nodes taken upon exiting a node.

Sadly stackless traversal is a time/size complexity tradeoff: the stack space is exchanged for efficiency, with disadvantages such as having to intersect nodes in a less optimal order, intersecting nodes multiple times or increasing the memory footprint of the tree. For some acceleration data structures, for instance Uniform Grids, a stack is not needed to begin with, as the traversal is always ordered.

Data parallel construction algorithm

As the real-time graphics industry starts to adopt ray-tracing techniques, the need for fast (re)construction of the accelerating data structures for raytracing will only grow stronger. For such purposes as dynamically loading content, altering the world or procedural characters animation, there is a need to have an up-to-date structure, possibly every frame. Even in the offline case, very long build times can undermine the speedup of the acceleration data structure if we end up casting too few rays.

Ideally a build algorithm would be as data-parallel as possible, to allow it to run on the same accelerator as the intersections will, and to use the vast amount of parallel processing power available today. We will see later in this work that some acceleration data structures have an intrinsically data-parallel build, while some others have data-parallel builds but with less effective results than the non-data-parallel.

Graceful handling of the teapot in a stadium problem

The ”Teapot in a stadium” problem describes pathological cases where an intricately detailed, small object (for example a finely modelled teapot), is placed in a much larger object with comparatively less detail. In more general terms, this is the problem where you have large differences in scene complexity along its space.

To handle this problem neatly, a structure has to be adaptive: it must recognize high and low density areas and avoid ”hot spots”, matching the information density itself with that of the scene.

Chapter 3

Survey of existing Acceleration Data Structures

In this chapter we will discuss a selection of acceleration data structures, give the intuition for how they work, describe their traversal algorithm and discuss their (possibly multiple) construction algorithms. We will also provide illustrations and relevant insight we think might be useful to the reader for implementing it themselves.

3.1 Bounding Volume Hierarchies

Bounding Volume Hierarchies (BVHs) are possibly the most ubiquitous data structure for accelerating ray queries. They are object-partitioning trees, that is they organize the primitives of a scene in a tree, putting the primitives in the leaves and a bounding volume at each inner node. Those bounding volumes bound the whole sub-tree for each node, such that when a ray fails to intersect the volume, we can eliminate the entire sub-tree, which is quite effective in reducing complexity.

The typical bounding volume is the axis-aligned bounding box (AABB), often just called bounding box, with non-axis aligned bounding boxes specifically referred to as Oriented Bounding Boxes (OBB). A BVH can also use other geometric shapes for bounding volumes, such as spheres or tetrahedrons, but this is less common in practice. Figure 3.1 shows us a typical BVH in a 2D, together with the tree layout on the right.

An interesting property of a BVH is that it is always "valid": Assuming the bounding volumes are consistent with the hierarchy, any distribution of the primitives along the tree is valid. This has the neat consequence for animated scenes that as long as the topology of the scene doesn't change radically, a BVH can be re-fitted [33] to a new frame by recomputing the bounding volumes to match the updated primitives.

3.1.1 Traversal of a BVH

Traversing a BVH is conceptually simple too, it is analogous to a tree search. Starting at the root, tree is descended upon, and nodes are eliminated when their bounding volumes are either missed by the ray entirely, or the hit would be outside of the valid range $[t_{min}, t_{max}]$ of the ray. A stack[1] is used to store the nodes pending processing.

[1] Originally a priority queue was used [26], but a stack proved better in practice

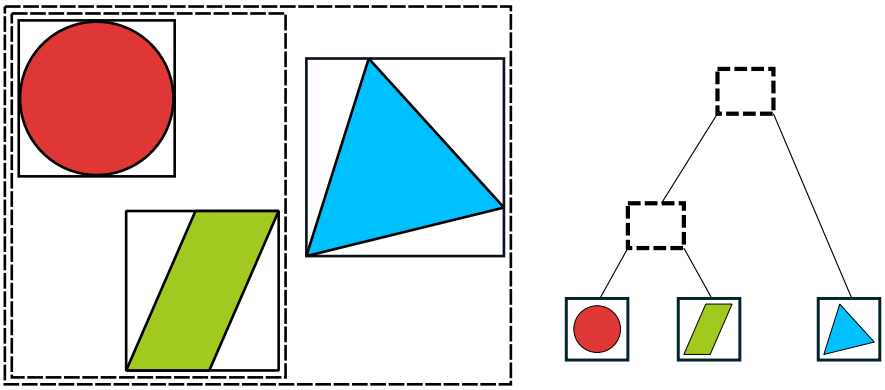


Figure 3.1: Example of a 2D BVH. Dotted boxes are inner nodes, plain boxes are leaf nodes that store actual the primitives (only one per leaf node in this example).

When a leaf node is finally reached, its primitives are intersected. The traversal ends when the stack is empty.

Whenever a hit is found, t_{max} is updated to reflect the fact we are not interested in rays further away from the camera, only closer hits. In the case of a shadow ray, we just return on first hit. We provide pseudocode for this algorithm in listing 3.1.

```
def traverse(tree, ray):
  hit = nil
  stack = [tree.root]
  while stack.not_empty:
    node = stack.pop
    if intersect(ray, node.bounding_volume) <> nil:
      fi node is Leaf:
        foreach primitive in node.prims:
            hit, ray.tmax = intersect(ray, primitive)
            if shadow and hit <> nil:
              return nil
      else if node is Inner:
        foreach children in node.children:
          stack.push(children)
  return hit
```

Listing 3.1: Simplistic BVH traversal using an explicit stack

In practice various optimizations are applied on this traversal loop. First we can evaluate the bounding volumes eagerly, before pushing on the stack. This allows culling them early, eliminating needless accesses to the stack for nodes that would be immediately culled. The node variable is usually lifted out and kept in a loop variable, reducing accesses to the stack further when only one children node passes the bounding volume check.

Together with early culling we can also sort the child nodes when intersecting their bounding volumes, with closer nodes being scheduled for intersection before the farther ones. Doing so will increase the chances of a intersection being found in the closer nodes, which means further candidates will be discarded afterwards.

It is possible to intersect the leaf nodes *eagerly*: Instead of pushing them on the stack, which is unnecessary since by definition a leaf node cannot require recursion, they can simply be intersected as soon as they are validly referenced by an inner node. For correctness this requires an additional check at the start of intersection if the root node is a leaf node itself.

To use explicit instruction level parallelism, both the bounding volume and the primitive intersections can happen in parallel. To get the most out of that it is possible to change the arity of the tree and the number of primitives per leaf, for example on a machine with 4-wide SIMD capability, the computations for 4 bounding boxes can happen in parallel [10].

3.1.2 The Surface Area Heuristic

The surface Area Heuristic [35] is a heuristic tool to estimate the expected performance of a given data structure for accelerating rays by computing a cost metric. The SAH starts from the assumption of uniformly distributed, infinite random rays that all intersect the scene but miss all primitives.

Under this, the probability of a ray intersecting a node is proportional to its surface area. The SAH ignores the possibility that an intersection is indeed found, and makes its prediction along the entire path of the ray through the scene structure.

The SAH was originally proposed in the context of spatial partitioning structures like Kd-Trees, but can be applied object partitioning approaches too. The global SAH cost for a ray can be written as (3.1), we note this is slightly different from MacDonald's original formulation because having a fixed cost attached with leaf nodes has been experimentally connected with build algorithms missing good candidates [49].

$$SAH = \frac{C_{inner} \times \sum_{i \in I} SA(i) + C_{prim} \times \sum_{l \in L} SA(l) * N(l)}{SA(root)} \tag{3.1}$$

With C_{inner} being the cost for intersecting a inner node and C_{prim} the cost of intersecting a primitive, $SA()$ standing for the surface area of a node, $N()$ being the primitive count of a node and I and L being the sets of inner and leaf nodes, respectively.

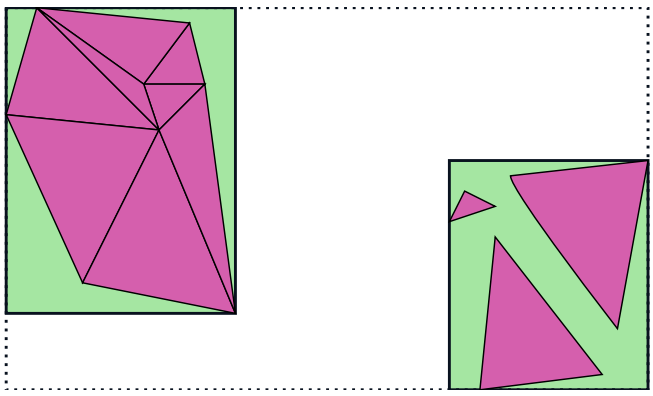


Figure 3.2: Within the parent node in dotted lines, the triangles (in purple) can be grouped in two sets. The SAH is optimized in this instance, the two clusters of triangle are bounded by separate, tight bounding volumes.

Criticism of the SAH

Even though the SAH is widely used and successful in practice, it is not without issues, as a 2013 HPG paper by Aila et al [1] points out: Its assumptions are questionable and more concerning is how there is sometimes a disagreement between the SAH cost of a structure and its actual performance.

Of the assumptions the SAH make, the random distribution of rays position and directions more or less hold, but the assumption of a ray spanning the entire scene without intersections is problematic. Aila et al identity two issues and provide two additional metrics that can be used in conjunction with the SAH to make its predictions more accurate.

End Point Overlap describes the additional searching work necessary to verify one of the ray ends (origin or hit point) is indeed unique, in the context of overlapping regions of space. Spatial partitioning structures are exempt from this issue by design since they don't have overlap. The EPO is given in (3.2) and can be mixed in with the SAH by (3.3)

$$EPO = \sum_{n \in N} Cost_n \frac{SA((S \setminus Q(n)) \cap n)}{SA(S)} \tag{3.2}$$

With S being the set of all surfaces, n being the volume of the node and $Q(n)$ being the surfaces in the subtree n.

$$(1 - \alpha)SAH + \alpha EPO \tag{3.3}$$

With $0 \leq \alpha \leq 1$

Leaf Count Variability The LCV is the standard deviation of the number of leaf nodes intersected by a ray:

$$LCV = \sqrt{E[N_l^2] - E[N_l]^2} \tag{3.4}$$

Its impact is felt in SIMT heavy processing on GPUs kernels, since those described by Aila et al [1] need to switch between intersecting leaf and inner nodes. The heavy variance can reduce vector units utilization. It is not presented as something to be optimized during construction, but rather as the missing factor to explain the experimental numbers obtained.

3.1.3 Insertion based construction

One of the earliest method for building a BVH automatically[2] is described by Goldsmith [16]. It is an insertion-based build: starting with one node, each primitive is added to

[2]Surprisingly, the first bounding volume hierarchies were created by hand.

the tree in succession, with a search to determine the most optimal place to insert it at. The builder is SAH-aware and tries to minimize the additional cost on insertion.

However a significant issue with this is how sensitive the top of the tree will be to the insertion order of primitives: if tightly clustered primitives are added first, the resulting hierarchy will then have to expand massively and the culling efficiency will be very poor.

3.1.4 Top down construction

A common approach to building BVHs is to start by creating a root node containing all the primitives, and then use heuristics to split each node recursively until some termination criteria is hit. The termination criteria is given by the SAH: we stop when the cost of creating more nodes is more than the cost of leaving everything as a leaf node. The number of primitives in a leaf node is typically between 4 and 16, for reasons already discussed above regarding SIMD efficiency.

The decision of *how* to split inner nodes is a much more important one. We want the tree layout to cluster as many primitives as we can in a minimal volume so we can cull most effectively. We use heuristics to capture those effectiveness characteristics, the most widely used being the Surface Area Heuristic detailed below.

Full sweep SAH build

Using the SAH as our cost model, Wald [48] describes an algorithm for building a binary BVH according to it. Starting with a leaf node containing the entire scene, the key idea for knowing where to split is to sweep the objects across the three axes, first sorting them by the position of their centroid according to the axis, then iterating through that list, moving objects from the right set to the left as they are iterated on.

At each step, the bounding box for both the left and right set is computed and in turn the SAH cost is derived. The minimal SAH cost is greedily selected after sweeping all objects along the three axes. A simple termination criteria is to stop splitting once the SAH cost for splitting a node in two fails to best the cost of leaving it as is. We provide pseudocode for the full sweep build in listing 3.2.

```
def try_split(node: Node):
  count = node.prims.size
  leaf_cost = sah_cost_leaf(node)
  best_cost = leaf_cost
  best_axis = -1, best_place = -1
  foreach axis in X, Y, Z:
    sorted_prims[axis] = node.prims.sortBy(axis)

    left_area = float[count]
    bbox = empty_bbox()
    foreach index in 0 .. count:
      left_area[index] = area(bbox)
      bbox.expand(sorted_prims[axis][index])

    bbox = empty_bbox()
    foreach index in count .. 0:
      right_area = area(bbox)
      bbox.expand(sorted_prims[axis][index])
```

```
      cost = sah_cost_inner(index, left_area[index],
        count - index, right_area)
      if cost < best_cost:
        best_cost = cost
        best_axis = axis
        best_place = index

  if best_axis == -1:
    return node
  else
    left = sorted_prims[best_axis][0 .. best_place]
    right = sorted_prims[best_axis][best_place .. count]
    return new InnerNode(left, right)

def build_tree(primitives):
  root = new LeafNode(primitives)
  return try_split(root)
```

Listing 3.2: Full sweep BVH build

Worthy of note is how the two passes over the sorted primitives are done each time, this allows for progressively growing the bounding boxes for each set linearly instead of having a nested loop. On the second pass we do the actual SAH cost computation, and we keep any beneficial split.

Binned SAH construction

The full sweep BVH build is high quality, but expensive. It requires sorting the entire scene many times —or keeping it sorted with a stable partitioning algorithm—, and going through $3N$ split positions per processed node (N being the total number of objects in the subtree of that node).

It is possible to generate trees of comparable quality, but in significantly less time by using a "binning" approach, originally a technique used for building Kd trees. Wald describes one such method [47]. In the described method only a fixed number of split planes will be considered, along a single axis.

First, the split axis is chosen, a possible heuristic is taking the bounding box containing all the centroids of the primitives in the currently considered subtree. It is also possible to try them all.

Along this axis, the range between the node spatial extents is split in k equally sized intervals, slicing the current volume in k *bins*. Each primitive is sorted in exactly one bin.

After all primitives were assigned into a bin, the bins bounding volumes are computed based on the primitives they ended up containing. Finally, a regular SAH sweep is performed on the bins themselves as if they were primitives.

3.1.5 Bottom-up construction

Linear Bounding Volume Hierarchies

The Linear BVH [34] is an extremely data-parallel and fast approach for creating a low-quality BVH. It involves fitting the scene in a cube and filling that cube with

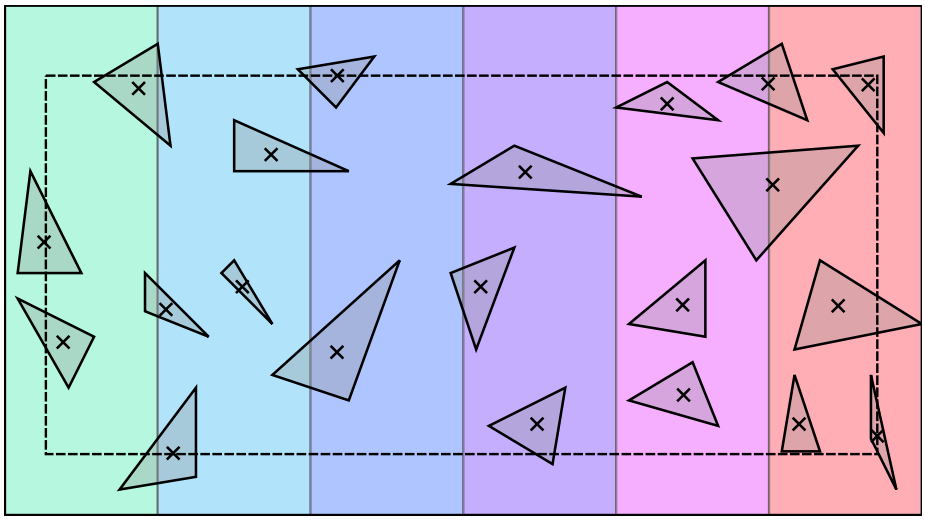


Figure 3.3: Binning with k=6 bins, the centroids volume is the box with dotted lines.

a space-filling curve, in this case the Z-order or "Morton curve". Along this one-dimensional curve every primitive can be placed according to its centroid (Figure 3.4).

This curve encodes spatial properties: We can see how segments of the curve encode for regions of space. The LBVH build uses a discrete grid where coordinates are hashed into so-called "Morton codes". We can assign each primitive a Morton code (say, with 30 or 63 bits of precision to fit in a typical integer value) and sort the primitives by the lexicographic order of their code.

Once that is done, we can notice that the bits of the code are actually equivalent to addresses in an octree (quad tree in the 2D case). By looking at discontinuities in the list of sorted codes, we can find the bounds of this implicit hierarchy and recover its structure. Once the structure is obtained all that is left to do is compute the bounding boxes for the inner nodes.

The LBVH build has interesting properties: First as mentioned, it is extremely data parallel: Generating the codes can be done in parallel for every primitive, efficient parallel algorithms for sorting like radix sort can be used and finally recovering the hierarchy can also be done in parallel.

The algorithm overall is subjectively simple and elegant, with a few distinct steps and simple memory requirements. It features no recursion and no free parameters to tweak. All these properties make it very easy to implement, even on a GPU, where the abundance of power coupled with the data parallelism can easily handle dynamic scenes in realtime.

However the resulting trees are not of a very high quality, and are handily beaten by almost any other method in terms of speedup if build time is not a consideration. Lauterbach [34] suggested using it in combination with a SAH-aware parallel build in a hybrid build, using the LBVH "bootstrap" the upper levels of a top-down build.

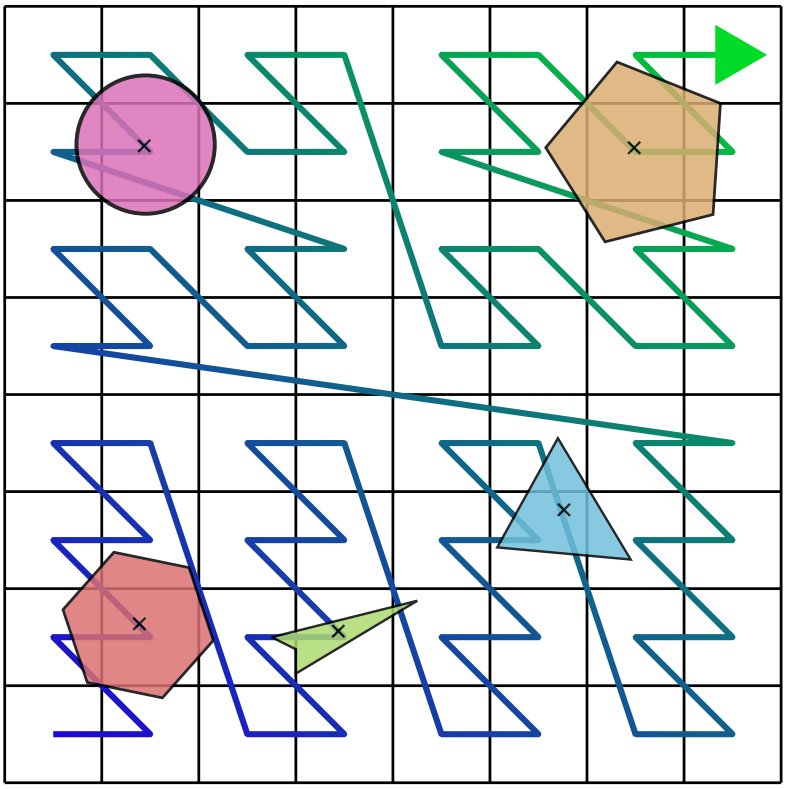


Figure 3.4: A grid and Morton curve fitted to a scene

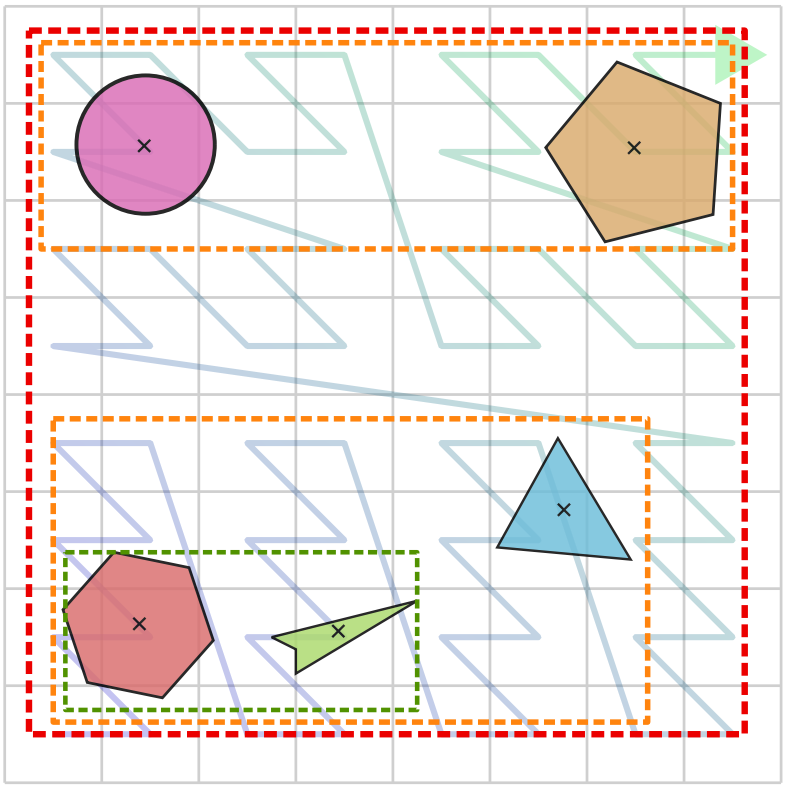


Figure 3.5: The recovered hierarchy from using the Morton codes as a spatial ordering

Parallel Locally Ordered Clustering

Similar to the LBVH in that it employs a Morton curve, PLOC [36] is a recently published algorithm for building a tree by agglomerating primitives together in clusters. Just like the LBVH build it is massively data-parallel by nature, but it produces far better trees, both in SAH costs and actual measured speedup, while retaining a very low amount of complexity.

At the core of PLOC is a nearest-neighbor search: At every step, pairs of clusters that have mutually the lowest cost according to a specific distance function are merged together. Clusters are initialized as leaf nodes containing one primitive each, each merge operation creates an inner node. The algorithm keeps merging clusters until only one remains, corresponding to the root node.

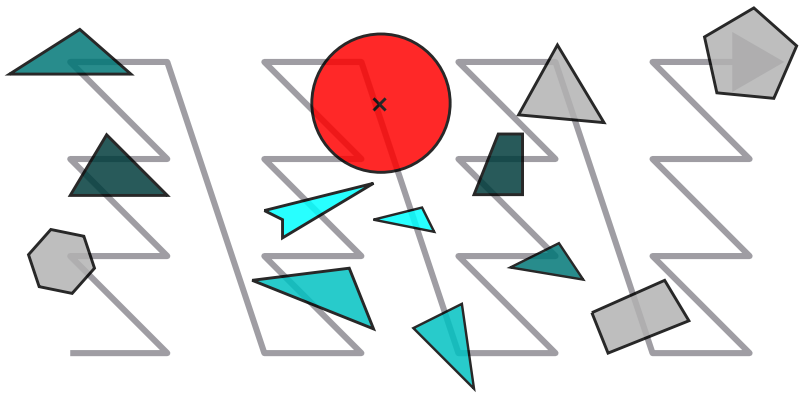


Figure 3.6: Nearest neighbor search for $r = 3$ with search candidates in shades of teal

To enable fast, sub-linear nearest-neighbor searches, all the primitives are stored in a list sorted by the Morton order of the primitives centroids, just like shown in Figure 3.4. The nearest-neighbor search uses a r parameter for controlling the window of primitives that will be considered (Figure 3.6).

The existence of this bounded search window renders the search complexity independent of the size of the scene, and was found to not miss important candidates due to the requirement of the nearest neighbor relation being mutual for two pairs of nodes.

In fact, the implicit spatial structure encoded with such a curve is said to be a positive factor, as the separation of clusters leads to less overlap (See section 3.1.2) and in turn a better correlation between SAH and actual trace performance as discussed in *On Quality Metrics of Bounding Volume Hierarchies* [1].

3.1.6 Pre Splitting

Unlike a spatial data structure, a BVH is not allowed to split a primitive. This can cause issues with outliers, such as unusually large objects or objects whose bounding box is mostly empty space, with figure 3.7 serving as an example. A common strategy to alleviate this issue is to subdivide these problematic triangles in advance, as a pre-process step.

Subdividing comes with the overhead of either creating more primitives, or having an indirection level by storing references to primitives instead of the primitives

themselves.

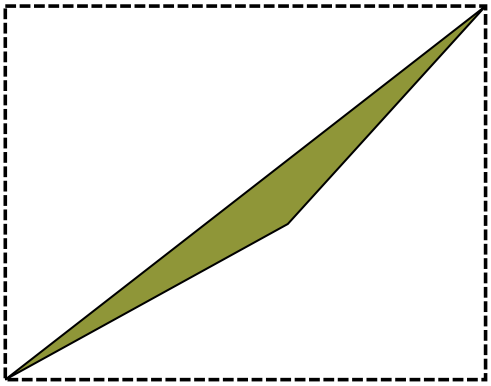


Figure 3.7: This triangle has next to no area, yet the bounding box is large

The Edge Volume Heuristic

The Edge Volume Heuristic (EVH) [11] proposes splitting triangles by considering the bounding boxes of their edges. Each of the three edges of a triangle is considered, with its two endpoint acting as the bounds of a box. The volume of the largest box is compared against a threshold value, and the triangle is split in two along that edge if it is over the threshold.

Split triangles are considered again, such that the process can be recursive. Because this heuristic is based only on considering edges, and makes a deterministic decision, it does not introduces numerical precision issues issues in the mesh. Any shared edge that is a candidate for splitting will be split identically on both sides, leaving the resulting meshes watertight.

Spatial Median Splits Heuristic

In *Fast Parallel Construction of High-Quality Bounding Volume Hierarchies* [25] another heuristic for pre-splitting triangles was introduced. Unlike the EVH, this heuristic will attempt to split triangles analogously to how a spatial data structure would, in particular it uses the splitting planes of a regular median space subdivision to split primitives up, as shown in Figure 3.8.

This is done so primitives crossing such planes do not end up causing massive overlap once the bounding boxes are refitted at the end of LBVH construction. For any considered candidate, the split axis is the one where the extents of the primitive bounding box are the largest, and the split position is the most important split plane crossed by the box, as highlighted in Figure 3.8.

To make the memory requirements predictable, a factor for the number of allowed references (for example, 1.5x the number of primitives for 50% extra split primitives) is picked. This allows allocating everything needed in advance.

Primitives are all assigned a priority modeling how well the bounding box area matches the primitive area in accordance to the following equation:

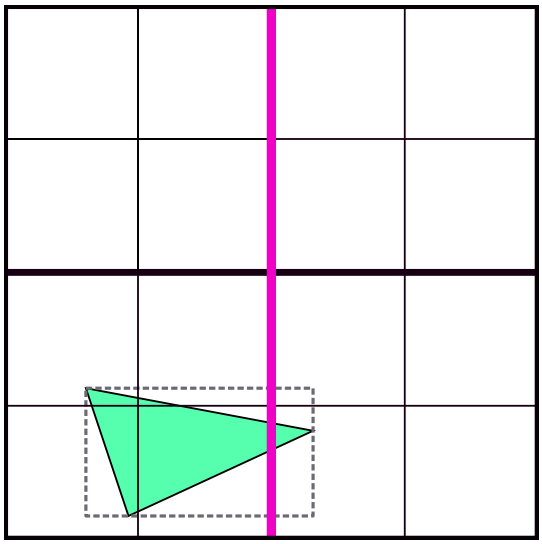


Figure 3.8: Primitives are split where their bounding box intersects a major region boundary, highlighted in pink in this case.

$$Priority_t = (X^i.(Area_{BoundingBox} - Area_{Primitive}))^Y \tag{3.5}$$

In (3.5), X and Y are free parameters, the values 2 and 1/3 respectively were suggested as they produced good empirical results. i is the depth of the node in the virtual spatial splits tree that would correspond to the plane.

From these priorities a factor D is computed to distribute the number of available splits (max references - number of primitives) for each primitive, making sure to not make more splits than memory was allocated for.

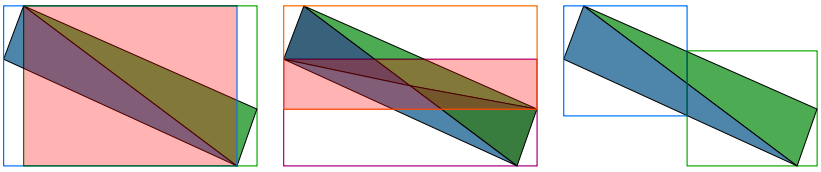
Finally each primitive is split as many times as budgeted for using a stack, each time a split occurs the results are put on the stack to be split again until we run out of split budget for a specific primitive.

3.1.7 Split BVHs

Split BVHs [45] are a special kind of BVHs that allow splitting a node *spatially* rather than by partitioning the set of objects contained inside. As discussed in the pre-splitting section, some primitives have problematic surface area to bounding volume area ratios, and considering spatial splits during construction, in accordance to the SAH cost, can produce better quality trees than pre-splitting, as seen in Figure 3.9.

The major difference between a SBVH builder and a regular greedy SAH-aware top-down builder is that in addition to object partitioning candidates we also search for advantageous spatial splits. At each step, the best option of the two schemes is picked according to SAH cost. For object splits, both full sweep BVH and smarter approaches like binning can be used.

For spatial splits, unlike regular Kd-Trees, the SAH cost along a primitive is not linear, because the overall bounding volume for the entire bin depends on the clipped



(a) No splitting (b) EVH (c) Spatial splits

Figure 3.9: The red region is the overlap between the two volumes.

primitives in all directions, not just along the split axis. Therefore the already costly method of only evaluating the bounding boxes extents is insufficient here. To answer this challenge, the authors propose using a binning approach.

The scene is split into N bins. For each bin, a bounding box is grown based on the bounding boxes of every primitive contained, clipped against the split plane. Each bin also has an entry and exit counter to record the primitives that start and end respectively in each bin. Once the bins are full, they are swept and the SAH cost is evaluated by using the bins bounding boxes and the entry/exit counters to know how many references lie on each side.

To keep the amount of additional references under control, a user-defined parameter $\alpha \in [0, 1]$ is used. The parameter controls the threshold of when to *consider* spatial splits with regards to the ratio of surface area in the overlapping regions of an object split to the region of their parent:

$$\frac{SA(B_L \cap B_R)}{SA(B_{root})} > \alpha \tag{3.6}$$

Note that α does not control the actual choice of using spatial splits, the SAH does. It merely gives a requirement for considering them, and essentially says some spatial splits are not interesting. It was experimentally found that very low values of α (10^{-5}) give good results with limited reference duplication.

Advantages of SBVHs include their better raytracing performance, better SAH cost and robustness over pure spatial partitioning methods: when a good spatial split cannot be found, the algorithm can simply fall back to object partitioning. Empty nodes are also not an issue, since bounding boxes of nodes still tightly enclose their contents there is no need for empty nodes.

Two disadvantages are their relatively higher complexity and the loss of ability to re-fit a SBVH if the scene geometry shifts. More accurately re-fitting would still produce a correct tree, but the spatial splits would be lost as they are only encoded in the bounding boxes of the split nodes.

3.1.8 Post Processes Tree Layout Optimizations

A popular approach in the literature is applying modifications to existing trees as a post process in order to improve their characteristics. These techniques can allow for improving the SAH cost and traversal performance beyond what a greedy SAH build can do, partly because they can use the full form of the SAH instead of the greedy form.

Metaheuristics and tree rotations

Kensler studied tree rotations [28] for optimizing binary BVH trees.

The idea is to perform *rotations* on sub-trees, analogously to tree rotations in other kinds of trees such as sorted binary trees. Tree rotations for BVHs are slightly different however, since leaf nodes cannot take the place of inner nodes, and children of a node are not ordered in general. The 4 rotations allowed in Kensler's work are shown in Figure 3.10.

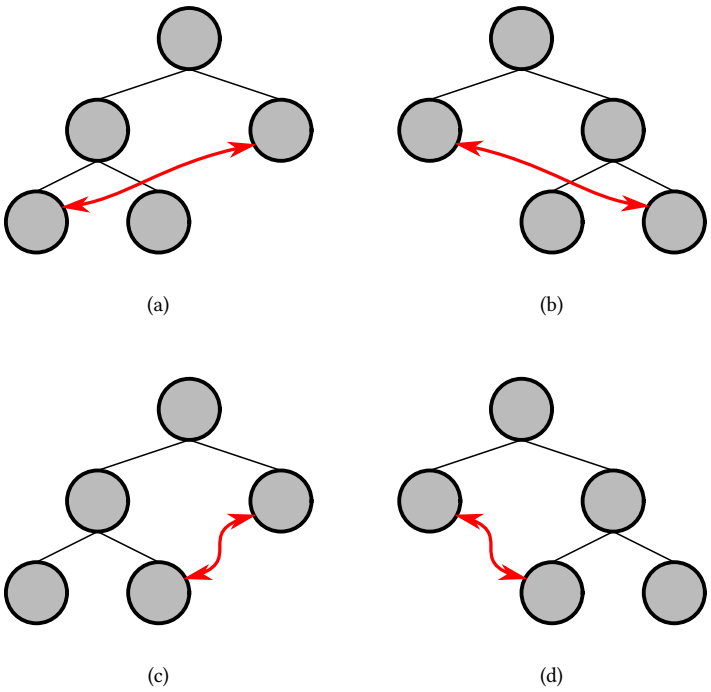


Figure 3.10: The 4 basic rotations available in a BVH

Additionally, swaps between grandchildren of a node were also used. The effect of these swaps could also be obtained by two rotations, but allowing those directly can allow jumping outside of a local minima.

With these transformations and the SAH heuristic as a cost model, we can use general-purpose optimization heuristics to guide the trees towards the optimum cost.

Hill Climbing First, a simple hill climbing algorithm was proposed. The tree is visited recursively, computing the cost when a leaf node is reached. Inner nodes first compute the cost of their children, then consider all possible rotations and swaps. The one that yields the best improvement (if any) is applied. The process is repeated on the entire tree until the SAH cost no longer improves.

Noteworthy is no matter how many rotations or swaps happen in a subtree, the bounding box for that subtree will not change, such that the ascendant nodes will not need to be updated. This means there is no risk of adverse effects to optimizing a

subtree on the rest of the tree, the impact of the modifications is simply a SAH cost reduction for that portion of the tree.

Simulated Annealing Starting from the hill climbing algorithm, Kensler described another one based on simulated annealing. It is structured similarly to the hill climbing optimizer, but this time non-beneficial rotations and swaps have a chance to be considered. The probability of allowing a SAH-worsening move scales with the cost of such a move and global temperature.

Global temperature is cycled many times, with progressively less heat each time. After annealing is over, a final hill climbing phase begins to reach the local minima in the final location.

The trees obtained by hill climbing and simulated annealing had better SAH cost and up to around 15% improvements in path tracing time in architectural scenes with highly non uniform primitive sizes.

Treelets restructuring

A fast, parallel algorithm for a post-process tree optimization is described by Karras et al in *Fast Parallel Construction of High-Quality Bounding Volume Hierarchies* [25].

This approach achieves parallelism by considering sub-trees in isolation (treelets) and re-structuring those treelets from scratch. The treelets used in the paper are relatively small, with $n = 7$ leaves. Treelet leaves do not have to be leaves in the overall tree, as shown in Figure 3.11, they can be formed from any subtree within the BVH.

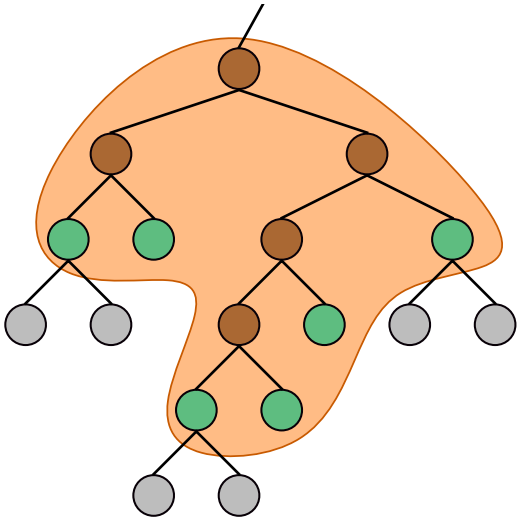


Figure 3.11: Example of a treelet with $n = 6$ leaf nodes.

The algorithm starts at each leaf node, and performs a bottom-up traversal, using a locking scheme so that only one thread processes each node. Whenever an inner node is found, the algorithm attempts to grow a treelet (starting with the node) to the desired size. Treelets are grown by repeatedly selecting the leaf node of the treelet with the biggest surface area and turning that node into an inner node, adding its two children as new treelet leaf nodes.

Once a treelet is grown, it is *restructured.* This means the existing inner nodes are dropped and the possibility space for possible binary trees with those leaves is searched exhaustively for the one with the best SAH cost. Even for small values of n doing so is computationally impractical without some smart algorithm design.

Karras et al explain how the space can be searched effectively. First they represent partitions of the set of leaves using a bitmask. Doing so is both practical for their algorithm but also space-efficient, with $n = 7$ allowing for storing a subset of leaves in a single byte, which played a significant role in enabling their efficient implementation on a GPU with limited shared memory.

Secondly they remove redundant computations by computing the SAH cost of all sub-sets in a deterministic order: Starting at computing costs for singletons and moving on to subsets with one more element each time. For each possible sub-set of the leaf nodes with $n > 1$, all possible partitions of that sub-set are evaluated, and the best one is stored as a bitmask and associated SAH cost.

To recover the optimal structure for a treelet one looks at the best partitioning for the bitmask corresponding to the set of all leaves in the treelet. From this bitmask, two children sets can be recovered, and those are again recursively queried for their optimal partitioning until leave nodes are obtained. The memory of the inner nodes of the original treelet can be recycled for the new one, since the number of nodes in the treelet does not change.

Reinsertion methods

The paper *Fast Insertion-Based Optimization of Bounding Volume Hierarchies* [5] presents a method for optimizing binary object partitioning trees *globally* by moving nodes in the hierarchy. Unlike with tree rotations or treelet restructuring, this is a global process optimization: while a long sequence of rotations can theoretically yield the same effect as a reinsertion, doing so is both costly and unlikely.

Nodes are selected for removal based on three heuristics for node inefficiency combined into one. The first one, M_{Sum} is the ratio of the surface area of the node to the average of the surface area of its children, targeting nodes with too much empty space. The second one, M_{Min} is the ratio of the surface area of the node to the *minimum* surface area of its children. The idea of this heuristic is to account for nodes with non-uniform node sizes. The last heuristic M_{Area} is simply the surface area of the node itself. All three are combined into (3.7).

$$Inefficiency(N) = M_{Sum}(N) * M_{Min}(N) * M_{Area}(N) \tag{3.7}$$

When a node N is selected for reinsertion (See Figure 3.12), both it, its parent P and its two immediate children (L and R) are removed from the tree. The parent is removed because in a binary tree removing one of its children (N) renders it redundant. L and R are placed in a reinsertion queue, while N and R storage is kept for reuse as new nodes on the reinsertion side.

To reinsert a node L a branch and bound search is performed in the entire tree, starting at the root. To guide the search two cost metrics for reinsertion are used: a direct cost $C_D(L, X)$ which denotes the surface area for a node containing L and X, and an induced cost $C_I(L, X)$ representing the enlargement of the ascendance of X to

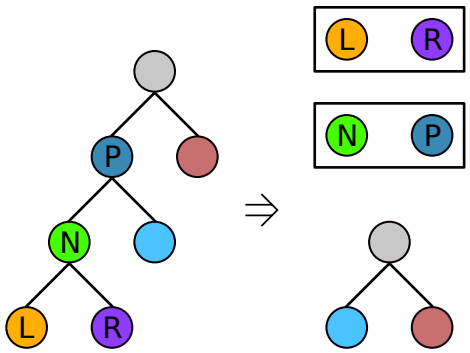


Figure 3.12: The N node is found to be inefficient, it is removed with its parent and kept in an unused node list. The nodes L and R are queued for reinsertion.

fit L inside. The branch and bound algorithm uses a priority queue according to the inverse of the induced cost.

$$C_D(L, X) = SA(L \cup X) \tag{3.8}$$
$$C_I(L, X) = C_I(L, parent(X)) + SA(L \cup parent(X)) - SA(parent(X)) \tag{3.9}$$
$$C_I(L, Root) = 0 \tag{3.10}$$

Figure 3.13: The induced cost for the root node is always going to be zero, the global scene bounding box has no reason to change.

When searching through the tree the lower bound of the cost reinserting the node at X is given by the induced cost for adding L to X, which we can easily track as we traverse the tree, and the surface area of L which is the lower bound for the surface area of $L \cup X$. Any potential branch which has a higher lower bound than the current best code found is discarded.

Once the best reinsertion position is found, each removed node is added back in the tree by reusing one of the redundant nodes from the removal as a new parent (Figure 3.14). The reinsertion process is applied to a small number of nodes per step (i.e. 1%), and runs until the SAH cost of the tree starts to converge. A threshold p_T is used to control after how many rounds of no improvements is the algorithm to stop.

Node reinsertion can improve the SAH metrics and the trace speed of trees constructed from any method (bottom-up, top-down or insertion-based like in [16]), whether or not those builders were SAH-aware. On top of that, it is close to two orders of magnitude faster than hill climbing and simulated annealing methods while yielding comparable or better results. This makes reinsertion a very strong technique to complement a builder of any type.

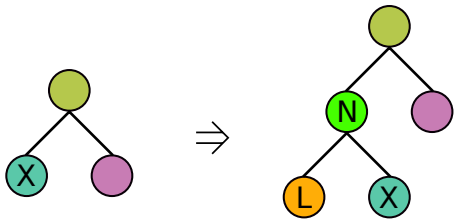


Figure 3.14: The node X has been identified as an appropriate reinsertion point for L, a new node is added in place of X with both it and L as children, reusing the storage for the previously removed node N

Parallel reinsertion One disadvantage of the previous method is that the sequential nature of the algorithm makes it unfit for parallel processors. Meister and Bittner proposed a variant of reinsertion designed for parallel processors [37], implemented on NVidia GPUs.

The key insight for parallel reinsertion is observing that it is unnecessary to actually remove a node from the hierarchy to search for a reinsertion point. Instead, for every considered node for removal, a pre-order traversal is performed on the entire tree, which also drops the need to maintain a priority queue.

The pre-order traversal (See figure 3.15) works with a pivot node, always visiting the sibling of the pivot node that does not contain the input node. When traversing down, the left side of the tree is always visited first, and after exploring the right subtree we go up one level, exploring its right subtree, up to the pivot. Once the pivot is reached, it is bumped to its parent and the search continues in its sibling.

When searching through the tree in this manner, we also track the total reduction in surface area along the path to the new location candidate. This reduction in surface area might not be monotonic, it may locally get worse especially after the pivot since that is a region of space where bounding boxes will have to account for our new node.

To ensure efficient search, branches are pruned when the lower bound for their surface area reduction plus the tracked surface area reduction is not as good as the one for the best candidate output node found so far. This lower bound is given by the difference in surface area between the parent of the input node and the input node itself.

Since the search happens in parallel for different nodes, there are many opportunities for conflicting insertions. Two class of conflicts exist: *Topological conflicts* where two reinsertions would modify the same nodes, jeopardizing the structure of the tree, and *path conflicts*, where the structure of the tree will remain valid but the computed reductions in surface area will not be accurate anymore.

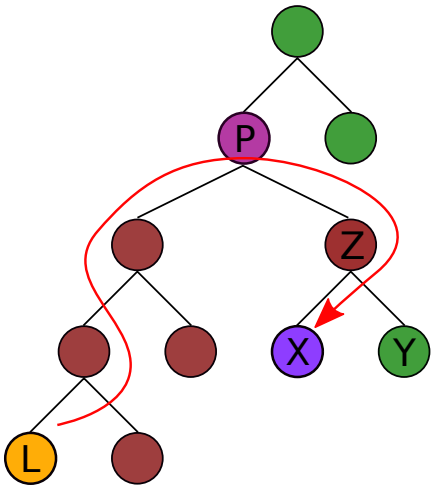


Figure 3.15: Input node L is looking for the best reinsertion candidate. Given the pivot node P and node Z having been previously explored, node X is currently being evaluated. The next step in traversal will back up to Z and upon realizing its the parent node of the last visited node, traverse its sibling node Y.

To prevent topological conflicts, the input node, its sibling, their parent and grandparents are locked, as well as the output node and its parent. If there is contention, the input node with the biggest surface area reduction wins out. A reinsertion only goes through if all six involved nodes are locked successfully.

It turns out that only performing to avoid topological conflicts and letting path conflicts happen is beneficial, to the surprise of researchers. This is explained by how little influence any one node has to the bounding box of an ascendant node, typically only defining one face of it at most. By removing two or more nodes at once, the total surface area reduction is higher than anticipated, and uncovers more fresh nodes on the edge of the bounding box. Conversely the reverse happens when adding to a node, the bounding box increase is lower than predicted for the sum of individual reinsertions sharing a path.

3.1.9 Stackless BVHs

Standard BVH traversal requires a stack for remembering nodes queued for traversal. As discussed in section 2.5.3 this can be considered a drawback and approaches have been formulated to minimize or outright avoid the use of a stack.

Skip nodes and Ordered Traversal

If the traversal order for BVH is fixed for all rays, that is, there is a total ordering on which nodes to visit assuming all bounding volume tests pass, then the traversal can be done without a stack. Instead, each node of the tree is augmented with a "skip pointer" [44] that points to next node to visit in case the current subtree should be skipped.

The rather major drawback of ordered traversal is that the scene is traversed in a potentially random order, and pathological cases can come up where many intersection tests will be performed in a part of the scene that is clearly occluded from the ray origin. An extension of the technique [6] is to store different orderings better suited for different ray traversal directions, but this comes with a losing memory trade-off as more directions get added.

Restart Trails

Laine proposed restart trails [32] to retain the good culling lost with a fixed traversal order. As the name indicates, restart trails are used in a *restart* strategy, where the tree search is reset back to the root when a leaf node is reached. For this to work with a BVH, we need a way to know which parts of the tree have been visited already.

Assuming the traversal order is well-defined (not to be confused for a fixed traversal order *in general* as discussed in section 3.1.9), the children of any given inner node can be strictly ordered into a near and a far child. It is possible for both to be culled by adjusting the ray t_{max} during traversal, but the near child will never be culled without the far children.

The restart trail tells the traversal which branch to take at every level. A value of one for any given level says "please explore the far child", while zero means we are still exploring near side. Note that in case only one children node remains after culling, that child is also considered the far child. When we reach the end of current tree branch, we do a "pop" by flipping the last '0' in the trail to a one, and filling the remainder of the trail with zeroes. The traversal resumes at the start, see Figure 3.17

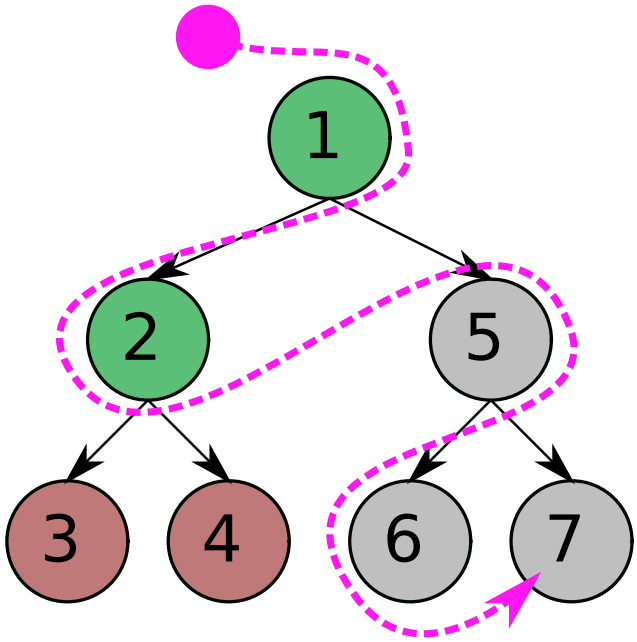
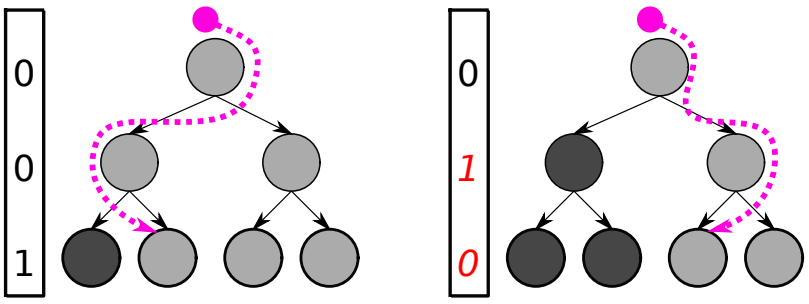


Figure 3.16: Ordered traversal in action: every node has a pre-determined traversal order. Green nodes have been visited already, red nodes have been culled. Because 2's bounding volume is missed, the traversal process (in dashed pink) skips straight from 2 to 5.



(a) (b)

Figure 3.17: Effects of a "pop" operation on the restart trail. Note that the tree is shown in traversal order for a specific ray: the closer nodes are always to the left.

Some extra attention needs to be taken for single children, so we also store the level at which the last "pop" occurred. This enables realizing when a near node becomes the far node as its sibling got culled, and in those cases we pop once more to avoid exploring that sub-tree again. The top bit of the trail is a sentinel bit for the entire tree: when it flips to one, the search is over.

Smart back tracing

Efficient Stack-less BVH Traversal for Ray Tracing [19] proposes a stackless algorithm for traversing trees that does not require intersecting more bounding volumes or more leaf nodes, and maintains the same traversal order as a normal stackful traversal.

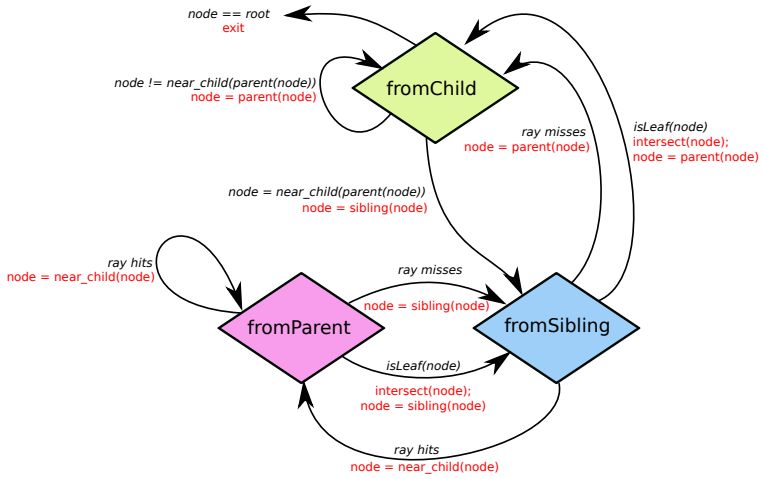


Figure 3.18: The state machine for stackless BVH traversal

Under the following assumptions:

1. We are traversing a binary BVH where each inner node has exactly two children.
2. Pointers to the parent node are available and the sibling node can easily be found.
3. The order of traversal (in which order to visit the two children of an inner node) is stable for any given ray and cheap to compute.

We can describe the traversal process can be modeled by a simple 3-state automaton with 3 transitions per state, as shown in Figure 3.18. Using the parent nodes, the tree is traversed downwards but also upwards when a subtree has been fully explored.

This is not so unlike the pre-order search used in section 3.1.8. The key trait to this method is using the traversal order for two nodes as implicit information to recover whether the current subtree was the first or second choice for traversal scheduling.

Hapala et al also provide insight into transforming the state machine in Figure 3.18 into an efficient implementation for a SIMT machine such as a GPU. The pseudocode for this is provided in listing 3.3.

```
def traverse(tree, ray):
  node = tree.root
  while true:
    def go_up():
        if node == tree.root:
          break
        last = node
        node = parent(node)
        continue

    if isInner(node):
      near, far = traversal_order(node.children, ray)
      if(last == far):
        // subtree was already traversed, go up
        go_up()
      descend_into = (last == parent(node)) ? near : far
      if intersect(node.bbox, ray):
        last = current
        current = descend_into
      else:
        if descend_into == near:
          last = near // descend into far next time
        else:
          // nothing to intersect here, go up
          go_up()
    else:
      // process leaf nodes here
```

Listing 3.3: GPU-friendly translation of Figure 3.18

While this appears as an elegant formulation of a stackless traversal for BVHs, this technique did not actually yield better performance than the stackful approach in testing. Hapala et al. reported that their CUDA implementation performed worse than the reference stackful algorithm on Tesla and Kepler GPU architectures, losing by about 30%.

Some of that difference was attributed to the less effective formulation of the loop that intersects only one children node per iteration, but the stack-less approach had to do more memory accesses and re-evaluate the traversal order often.

It is therefore proposed that the memory savings may be beneficial in different settings where the size of the incomplete ray state matters, such as hardware implementations of ray traversal, ray reordering or when the scene data is distributed across different machines (requiring the ray state to be transferred over some bus).

3.2 Kd-Trees

A Kd-tree is a spatial data structure used for storing k-dimensional points. In order to store non-point primitives to perform ray tracing, we store references to primitives in the leaves of the tree. Any primitive straddling multiple leaf nodes will be present in both. Kd-trees are a sub-set of Binary Space Partitioning (BSP) trees, with the restriction that each inner node is split in two by an axis-aligned plane. Figure 3.19

shows an example of a very simple kd-tree with 3 inner nodes and 4 leaf nodes.

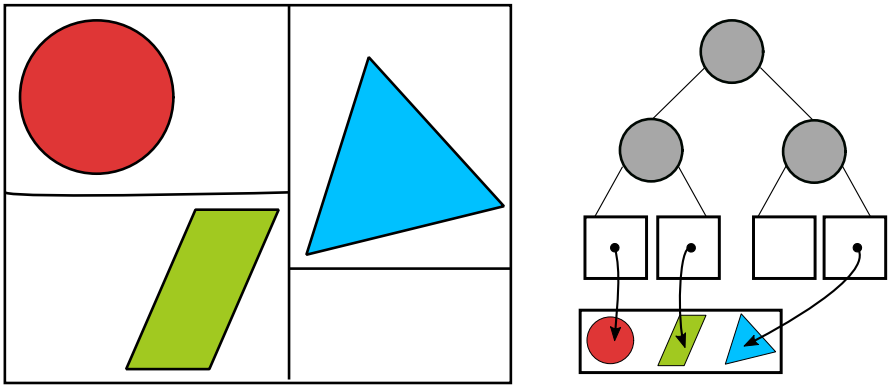


Figure 3.19: Example of a Kd-tree. Nodes are all placed in a general bounding volume, that is successively split into several sub-volumes. Leaf nodes contain references to the actual primitives but can also be empty.

Kd-trees and other forms of space partitioning trees like quad and octrees (which are just trees of arity 4 and 8, respectively) can be seen as the dual of bounding volume hierarchies, testing and eliminating regions of space rather than groups of primitives. They are most often studied alongside them and many approaches have been successfully applied to both, as we will see below.

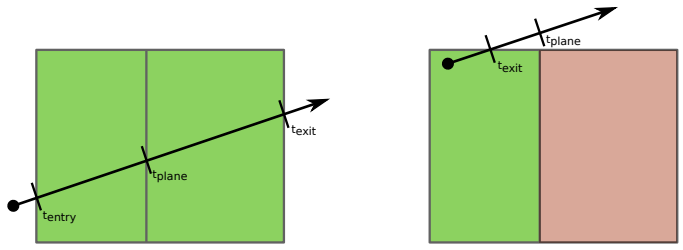
Comparing them to a "standard" BVH (one built without allowing spatial splits), building them can be harder and they often have to store multiple references to primitives and deal with, but on the other hand they never suffer from overlapping regions, allowing for effective use of Early Ray Termination and easier stackless traversal than in BVHs.

3.2.1 Kd-Tree traversal

Standard stack-based Kd-tree traversal [18] performs a recursive search through the tree, intersecting the ray with each inner node to make a decision on which children to continue the search in. When the two children are scheduled for visit, the search proceeds to the nearest one first and puts the furthest one on the stack. When a leaf node is reached, it is intersected and one node is popped from the stack to resume the search at. If the stack is empty, the traversal is over.

Figure 3.20 shows an example of a strategy for sorting and culling the children, using a strategy based on the relative position of the ray origin with regards to the split plane, the near node is said to be the one the ray origin resides in. Another strategy is to use the sign of the ray direction along the split axis. Both of these have robustness issues, respectively when the ray origin is on the splitting plane and when a direction component is zero.

A third strategy which does not suffer from these robustness issues was described by Harvan et al [21], but it requires additional storage for storing the exact position of the ray-split planes intersections.



(a) The intersection with the plane lies *between* t_{entry} and t_{exit}, both the near and far nodes need to be traversed.

(b) The intersection with the split plane lies *after* t_{exit}, so the entire far node is off-limits and can be culled.

(c) The intersection with the split plane is *behind* the ray, so the non-near node will never be visited.

(d) The intersection with the split plane lies *before* t_{entry}, therefore the near node needs not be visited.

Figure 3.20: The 4 possible outcomes for intersecting a node in a Kd-tree. Children in green will be visited, red ones will be culled.

Early ray termination

Since kd-trees are strictly spatial partitioning structures, early ray termination is very effective and essential for the best performance. Intuitively we could stop as soon as an intersection is found in a leaf, but since primitives are not clipped to the leaf nodes, it is possible for the hit to actually not belong to the space the leaf node covers, as shown in Figure 3.21.

To perform ERT while accounting for this case, after traversing a leaf, we check if the best hit so far (or t_{max} if none was found already) lays before t_{exit} for the current node. If it is, we can stop the traversal immediately without having to unroll the stack or do any further processing.

3.2.2 Stackless Kd-Tree traversal

Unlike BVHs, stackless traversal comes quite naturally for kd-trees thanks for the lack of overlapping nodes. By shortening the ray to start at t_{exit} in the last visited leaf node,

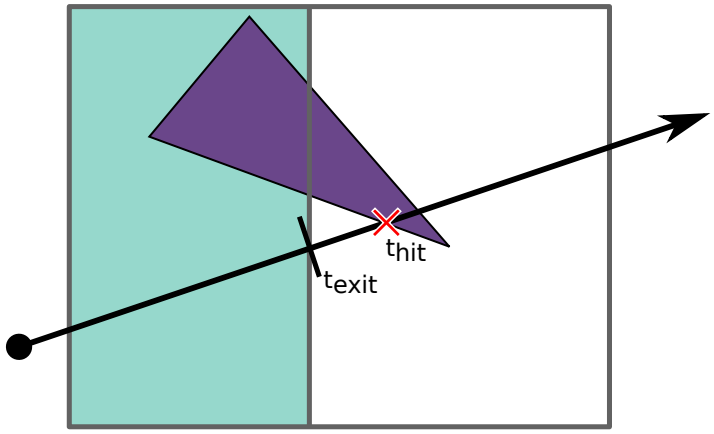


Figure 3.21: Example of an intersection found in a leaf, that does not actually belong in the space the leaf covers.

we cull all the nodes in the subtree starting at the last decision point (the last node that had two children to visit). The only remaining thing to do is resume the traversal to go back to that last decision point and take the alternative path.

Two variants of this ray-shortening traversal exist: *kd-restart* and *kd-backtrack* [14]. In *kd-restart*, we go back to the root of the tree and traverse it all over again until we arrive at the last decision point, which is no longer a decision point since the ray was shortened, and we immediately take the far node path.

In *kd-backtrack*, nodes are augmented with a parent pointer to avoid re-traversing the entire tree every time. After shortening the ray, we backtrack through parent nodes up to the point where we encounter a node that intersects this new shortened ray. The downside is that in order to perform the backtracking, we also need to store the bounding box for each node, and intersect it with the ray for each back tracking step.

3.2.3 Kd-Tree construction

Kd-tree construction is also similar to BVH construction, and build techniques evolved together, with advances in BVHs often reflecting earlier work in kd-trees. Kd-trees are built in a top-down fashion, which is achieved similarly to top-down builds for BVHs, with the biggest difference being the split operation itself, with objects able to end up on both sides.

As spatial data structures, kd-trees split *space*, so the decision on where to split is critical. In the 3d case this means 3 possible split axises, and theoretically an *infinite* amount of ways to split objects[3].

The first and most obvious approach is to split the nodes at the spatial median [15]. This approach is flawed however, since the decision of where to stop splitting was down to a threshold on the number of primitives left in a node, which did not capture the behavior of ray traversal.

When MacDonald and Booth introduced the Surface Area Heuristic [35], they

[3]In practice the implementation detail of fixed-size floating point datatypes renders those *technically* finite but this is irrelevant for our purposes and wildly impractical to explore.

presented it as a way to govern the construction of a tree. They showed that the optimal splitting plane lies somewhere between the object median (the plane that puts as many primitives on either side) and the spatial median.

Further, they derived that the optimum cost had to be along one of the extents of one of the objects of the scene, which means there are only at most $2N$ interesting split planes to try for each axis to find the best one according to the SAH. With the computational limits at the time, MacDonald and Booth proposed sampling the split space at regular intervals. Alternatively, they proposed to use the average of the spatial and object medians as a quick heuristic.

Evaluating all the possible split positions still requires considering $6N$ possibilities for each step of the top-down build, for each of those we have to count how many primitives are on either side of the split. Doing so naively takes $O(N^2)$ time, but a smarter algorithm can evaluate all splits iteratively [39], achieving $O(Nlog^2N)$. Wald and Havran [49] report on a variant of said iterative algorithm that can asymptotically achieve $O(NlogN)$ time.

Perfect splits

When considering split planes candidates, it is essential to account for the notion of "perfect splits", as discussed in [20]. Perfect splits are the split planes that lie along the faces of *clipped* primitives. These splitting planes can often be extremely advantageous, and ignoring them can have serious repercussions on the quality of the resulting tree.

Figure 3.22 shows a situation where considering perfect splits is relevant. (a) gives the setup: we are considering the right node after the first horizontal split that cuts across the left triangle. (b) shows what happens if we *don't* consider such splits: the bounding box for the brown triangle is left unclipped and we thus miss the pink split position in (c) that has the best SAH cost, with a significant area of empty space singled out.

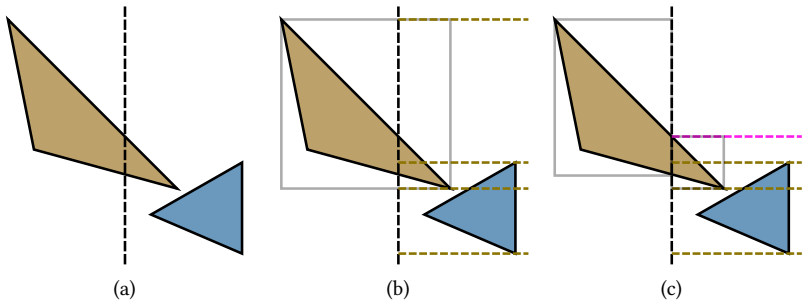
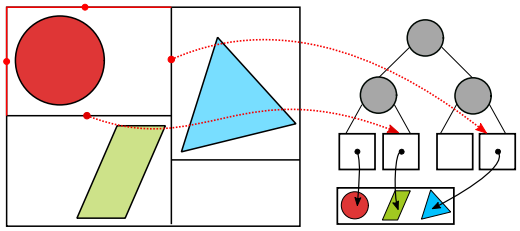


Figure 3.22: "Perfect Splits"

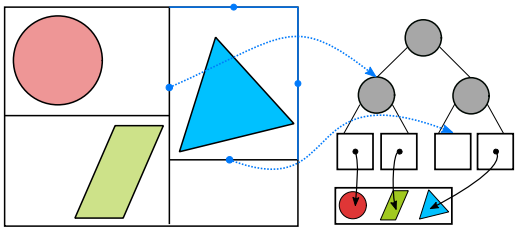
3.2.4 Kd-trees with ropes

Another strategy for stackless traversal that does not require backtracking or otherwise restarting the traversal in some way, is to use *ropes* [40], also known as neighbor links. The principle of those is to augment leaf nodes with pointers on each on their faces. These pointers get the traversal to the smallest sub-tree such that exiting the node through that face would always place it in that subtree.

In other terms, each leaf node gets six (in the 3d case) ropes to the node at which the traversal should continue upon exiting the leaf node. The resulting traversal tries to locate the entry point of the ray in the current subtree. When it finds a leaf node, it intersects it and follows the rope for the face the ray will exit by. When a nil rope is encountered, the traversal is over.



(a) Ropes for the node containing the red circle



(b) Ropes for the node containing the blue triangle

Figure 3.23: Example of ropes in a kd-tree. Note how the ropes at the edges of the tree point to nothing, and how the left rope for the blue node points to an inner node.

Construction of a kd-tree with ropes requires only a simple post-process recursive pass over the tree. First, the root node is given *nil* ropes for every side. All children nodes inherit the ropes of their parent, except along the splitting plane, where they get a rope to their sibling.

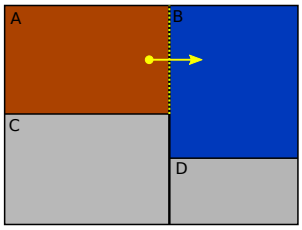
Lastly, the ropes can be further optimized by being recursively refined when conditions allow, see Figure 3.24. We keep applying either case until neither is applicable. This last optimization is optional and was noted to be harmful in the packet traversal scheme described by Popov et al [40] as it reduces coherency.

3.3 Grids

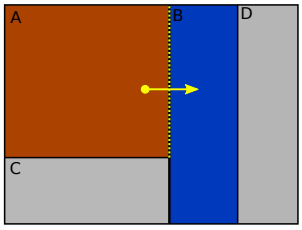
Uniform grids [41] —also known as regular grids— are one of the simplest spatial data structures possible. The scene is simply placed inside a grid, and each cell of the grid contains a list of references to the primitives that touch the cell.

Building a grid is conceptually trivial: Simply compute a global bounding volume, settle on a grid resolution and then fill the grid cells, intersecting the primitives with the cells to decide on the references. Such a build can be implemented in a data-parallel way and is very fast, as long as reasonable grid sizes are chosen [23].

Regular grids are known for being pretty ineffective when it comes to dealing with any irregular structures because they have no degree of freedom for adapting their density to the density of primitives in the scene. Which means choosing a resolution is often choosing between wasting time and memory with long deserts of empty cell,



(a) First case: A's bounding box is entirely in one region of F (the upper one in this case, above the split position), so we can point to its children B instead)



(b) Second case: The exit face is parallel to the split plane of F, so we can point to children closer to A, which is B in this case.

Figure 3.24: We are computing the rope for A's right exit face. We initially start with F as the rope target, which we inherited from E.

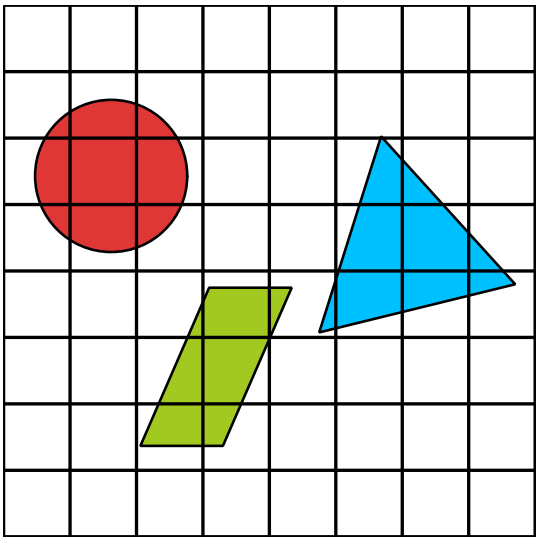


Figure 3.25: Example of a uniform grid. Each cell contains pointers to zero or more primitives.

or ineffective culling and no effectively no acceleration in denser parts of the scene. We will see some strategies for improving those characteristics below, and discuss the topic further in Chapter 5.

3.3.1 Grid traversal

Traversing against a grid is quite simple and elegant, it can be viewed as coloring the cells touched by a line drawn on the grid. Since grids are strictly spatial data structures, Early Ray Termination can be applied. Since grids are flat structures, there is no hierarchy, so there is no need for a stack, only a handful of values are necessary to track the position on the grid and known in which direction to move to the next cell.

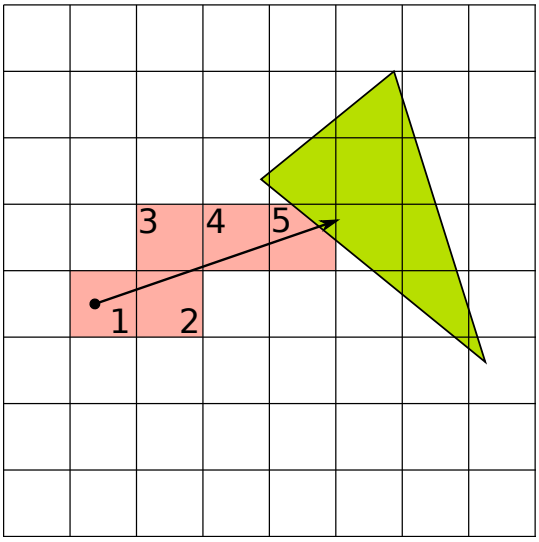


Figure 3.26: Cells touched by a ray, enumerated by order of traversal.

Figure 3.26 gives an example of how this traversal looks. The standard algorithm for achieving this pattern is a variant of the DDA algorithm [3], we provide it in Listing 3.4. They principle of this is to create and maintain knowledge of the "time" (distance along the ray) to the next voxel boundary in all dimensions, and always advance in the dimension for which this time is lowest.

```
def traverse(grid, ray):
    best_hit.t = ray.tmax
    grid_hit = intersect(ray, grid.bbox)
    if grid_hit.t2 > grid_hit.t1:
        return no_hit
    grid_pos = to_grid(ray.org + ray.dir * grid_hit.t)
    foreach a in axises:
        step[a] = ray.dir[a] > 1 ? 1 : -1
        next_edge[a] = grid_pos[a] + step == 1 ? 1 : 0
    t_edge = to_world(next_edge - ray.org) * ray.inv_dir
    while true:
        intersect_cell(grid_pos, best_hit)
```

```
        min_t, c = select_min_component(t_edge)
        if best_hit.t < min_time:
            return best_hit
        grid_pos[c] += step[c]
        t_edge[c] += to_grid(ray.inv_dir)[c]
        if !grid.in_bounds(grid_pos):
            break
    return best_hit
```

Listing 3.4: Grid traversal with improved DDA algorithm

3.3.2 Fast skipping of empty space

One popular approach to improve the qualities of uniform grids is to add some mechanism for skipping the regions of empty space fast. Generally the idea is to identify *regions* of empty space and mark them as such, then skip over them during traversal. As seen in Figure 3.27, this is extremely beneficial in some cases.

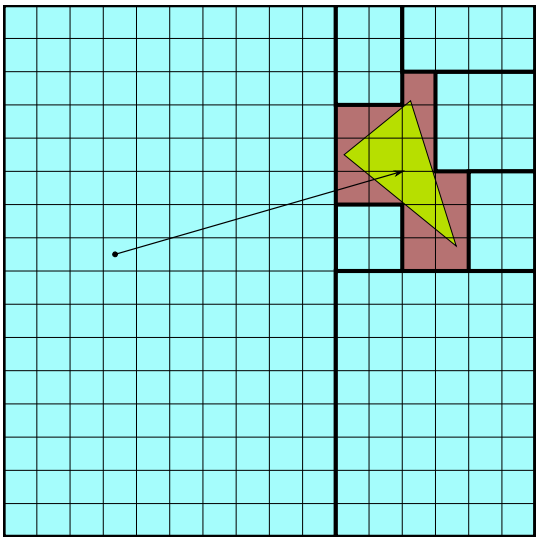


Figure 3.27: Example of macro regions

Typically to achieve this we store a distance field, giving us a metric to the nearest non-empty cell in empty cells, possibly multiple ones for different quadrants or axises. Es et al [13] detail their construction of an early GPU-based raytracing renderer while evaluating different approaches for storing those distance fields.

Devillers [12] proposed storing macro-regions separately and storing pointers to them in the empty cells of the grids. This approach gives maximum flexibility for describing empty regions of space, but comes at the expensive of slow build times and extra memory costs for both separate storage and the indirection during traversal.

3.3.3 Two-level grids

It is possible to nest grids within grids [9], which provides some answers to the major downside of grids which is their lack of adaptive detail.

In this thesis we studied two-level grids as presented by Kalojanov et al [23]. Figure 3.28 gives an example of one such structure. There is a top-level grid containing N bottom-level grids. Unlike say, a visually similar octree, this approach does not require a deep hierarchy to adapt with sudden changes in geometry density. Please note there is no requirements that sub-grids have power-of-two dimensions.

By restricting themselves to only two levels, they were able to generalize a previous fast data-parallel build for building one-level (uniform) grids [24] into a fast data-parallel build for two level grids. We give pseudo-code for this algorithm in Listing 3.5.

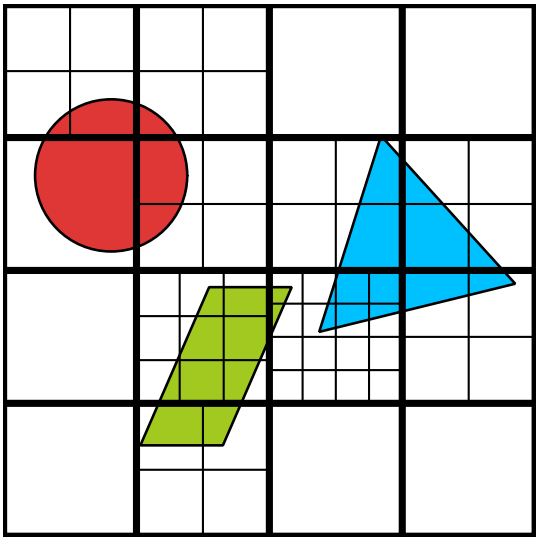


Figure 3.28: Two level grid in practice: The top grid has the thicker lines, while sub-grids use thinner grids. Note the lack of restriction on using non power-of-two grid sizes.

```
def volume(vec):
    return vec.x * vec.y * vec.z

def build_uniform_grid(primitives):
    bbox = compute_global_bbox(primitives)
    dims = pick_resolution(bbox, primitives.count)
    grid = allocate_cells(volume(dims))

    rough_pairs_count = allocate_int[primitives]
    foreach primitive in primitives:
        foreach cell in primitive.extents_in(dims):
            rough_pairs_count[primitive.index]++
    pairs_insert = prefix_sum(rough_pairs_count ++ [0])
    pairs = allocate_pairs(pairs_insert.last, (-1, -1))
    foreach primitive in primitives:
        foreach cell in primitive.extents_in(dims):
            if primitive touches cell:
                pair = (cell.index, primitive.index)
                i_pt = pairs_insert[primitive.index]++
```

```
                pairs[i_pt] = pair
        pairs.sort_by_first()
        foreach pair, i in pairs.indexed:
            if i == 0 || pairs[i-1].cellId <> pair.cellId:
                grid[pair.cellId].start = i
            if i == pairs.size - 1
                || pairs[i+1].cellId <> pair.cellId:
                grid[pair.cellId].end = i

        pairs.resize(pairs.find(-1,-1))
        return (bbox, dims, grid, pairs)

        //references = pairs.keep_second()
        //return (bbox, dims, grid, references)

    def build_second_level_grid(primitives):
        bbox, dims, tg, tp = build_uniform_grid(primitives)
        foreach cell in tg:
            cbbox = bbox / top_dims
            pcount = cell.end - cell.start
            cell.dims = pick_resolution(cbbox, pcount)

        grids_size = map(tg, {cell => volume(cell.dims)})
        grids_start = prefix_sum(grids_size ++ [0])
        l2_grids = allocate_cells(grids_start.last)
        foreach cell, i in tg.indexed:
            cell.first = grids_start[i]

        rough_pairs_count = allocate_int[tp.size]
        foreach top_cell_id, prim_id in tp:
            top_cell = tg[top_cell_id]
            primitive = primitives[prim_id]
            foreach cell in primitive.extents_in(top_cell):
                rough_pairs_count[prim_id]++

        pairs_insert = prefix_sum(rough_pairs_count ++ [0])
        pairs = allocate_pairs(pairs_insert.last, (-1, -1))
        foreach top_cell_id, prim_id in tp:
            top_cell = tg[top_cell_id]
            primitive = primitives[prim_id]
            foreach cell in primitive.extents_in(top_cell):
                if primitive touches cell:
                    global_index = grids_start[top_cell_id]
                    global_index += cell.index
                    pair = (cell.index, primitive.index)
                    i_pt = pairs_insert[primitive.index]++

        pairs.sort_by_first()
        foreach pair, i in pairs.indexed:
            if i == 0 || pairs[i-1].cellId <> pair.cellId:
```

```
            l2_grids[pair.cellId].start = i
        if i == pairs.size - 1
            || pairs[i+1].cellId <> pair.cellId:
            l2_grids[pair.cellId].end = i

    pairs.resize(pairs.find(-1,-1))
    references = pairs.keep_second()
    return (bbox, dims, tg, l2_grids, references)
```

Listing 3.5: Pseudocode for the two level grid build, can also be used for one level grids with light modifications

Two level grids were found to be fast to construct and thus suitable for dynamic scenes, but unfortunately they offer mediocre traversal performance compared to other contemporary approaches. We implemented them ourselves, and will analyze the benefits of two-level grids further in Chapter 5 where we discuss their performance relative to other grid-based techniques.

Chapter 4

Implementation work

We implemented an array of the acceleration structures discussed in the previous chapter in the Arty renderer, used for the *Realistic Image Synthesis* course at Saarland University. It features single traversal, a path tracer with multiple importance sampling [22], a bidirectional path tracer [31] and progressive photon mapping [17].

4.1 Implementation details for data structures

4.1.1 BVHs

For Bounding Volume Hierarchies, we modified Arty to use libbvh[1], a high-quality C++17 BVH header-only library. This was done despite the fact Arty already had a full-sweep BVH build with pre-splitting, because libbvh provides implementations for many other types of builds, in addition to various pre and post-processes. Thanks to this, we have a comprehensive library of BVH techniques to get concrete data on.

In order to give an interesting yet manageable amount of variants from all the features in libbvh, we settled on 4 typical builders configurations:

1. LBVH: A Linear BVH builder (see Section 3.1.5), with pre-splitting (Section 3.1.6) and a leaf collapsing post-process.
2. PLOC: Agglomerative builder (see Section 3.1.5), also with pre-splitting and leaf collapsing.
3. Binned SAH: Top-down build based on the SAH cost approximated with bins (see Section 3.1.4), with pre-splitting.
4. SBVH: Top-down build based on the SAH with spatial splits, considered state of the art for static scenes (See Section 3.1.7).

Additionally, we applied node re-insertion to all four to create four more variants, marked with a star $*$ suffix in our results.

[1](https://github.com/madmann91/bvh)

4.1.2 Kd-Trees

We added Kd-Trees to Arty based on their implementation in the book *Physically Based Rendering* [39]. Those feature a $O(Nlog^2N)$ build by sweeping an event list, as described by Wald et al [49], and takes into consideration perfect splits.

We implemented the classification strategies based on position and direction, as well as a stackful and kd-restart variant. For the rope-based kd-tree, we created the ropes and adapted our traversal based on Popov's et all paper on stackless kd-trees [40].

In the end we benchmarked the following configurations:

1. KD_no_ps: Stackful traversal, direction-based strategy for visiting the nodes, perfect splits off.
2. KD_basic: Stackful traversal, direction-based strategy for visiting the nodes, perfect splits on.
3. KD_positional: Stackful traversal, position-based strategy for visiting the nodes, perfect splits on.
4. KD_restart: Stackless KD-restart traversal, direction-based strategy for visiting the nodes, perfect splits on.
5. KD_ropes: Rope-based stackless variant, perfect splits on.

4.1.3 Grids

We implemented regular and two-level grids based off the work by Kalojanov et al [24] [23].

For the regular grid variant, we implemented the macro regions space skipping approach by Devillers [12], as well as another space-skipping scheme storing the size of the largest cube centered on the cell, into the cell itself, that we refer to as the distance field variant.

Additionally, we added another space-skipping scheme based on the idea of mip-maps. A grid storing binary information on whether a cell is empty or not was stored in multiple resolutions, and we used a multi-level traversal that could both traverse horizontally (along the 3d coordinates of the grid) and vertically (along the vertical levels).

4.2 Traversal data collection

Using C++ templates, we added an instrumentalized path for the traversal algorithms, making it return an additional `TraversalStats` structure. This structure tracks the following events on a per-ray basis:

1. Number of traversal steps. This tracks the number of main and secondary loop iterations in the traversal in order to find nodes/cells. This metric is not too useful for comparing between very different acceleration structure types as not all structures have the same loop organization.
2. Number of *visited* nodes (or cells for grids). Note than this is not necessarily a bounding box intersection, as we've seen in section 3.1.9, some traversal schemes will traverse nodes twice.

3. Number of intersected bounding boxes.

4. Number of intersected primitives.

Additionally, we track an estimation of global memory accesses by tracking the nodes/cells accesses, the primitive references and data accesses, and the stack accesses. These accesses are tracked with a bytes counter, incremented on every access with the size of the dereferenced data.

We consider other forms of memory accesses (register spills, small per-scene structures) to have enough locality for caching to be effective enough so they don't play a major role. Nodes, primitives and stack memory accesses tend be over larger pieces of data and be quite incoherent, so we expect those to be the cause of eventual memory-related issues.

4.3 Visualization

We added a debug view to visualize the traversal complexity for various types of rays. Such visualizations are common for primary rays but we also provided visualization for secondary rays. We added the ability to visualize ambient occlusion rays, as well as diffuse secondary bounces and shadow rays for direct illumination using next even estimation.

We use the data obtained in the instrumentalized ray traversal. Since the traversal statistics come with many counters, we have separate visualizations for each, plus some useful aggregates. The selected and aggregated values are mapped on a multi-point color gradient to create heatmaps, with a logarithmic scale. We provide such heatmaps and analyze them in Chapter 5. The scale of such heatmaps is uniform for all of them, and rather than duplicate it many times we provide it here in Figure 4.1.

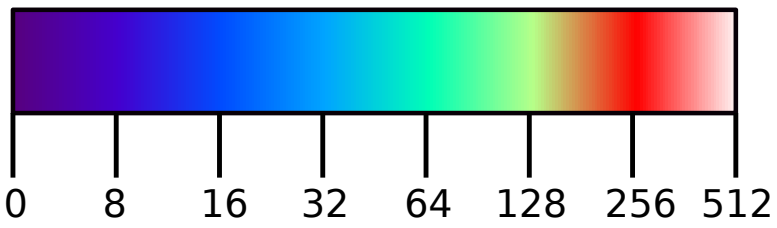


Figure 4.1: The scale for all heatmaps in this work

4.4 Benchmarking setup and procedure

We used a machine equipped with an AMD Ryzen R7 1700 with 32 gigabytes of DDR4 memory running in dual-channel mode at 2400Mhz for all our testing. The CPU was left running at default clocks with a stock cooler and the XMP profiles on the memory were not used. The machine was booted off a Fedora 31 solid-state drive connected via USB3.

To benchmark ray traversal, we first generated three sets of rays file for each scene. We ran Arty's path tracer in a special mode that recorded every ray traced, and stored

all the primary rays, bounce rays and shadow rays in separate files. We collected rays as we asked Arty to render 64 samples at the default resolution of 1080 by 720 pixels.

We then re-played these rays in another executable that only traced the rays in parallel without performing any shading. We compiled the replay program with GCC 9.2.1 with $-O3$ in CMake's *Release* mode. We ran three passes of this replay program:

1. The first pass collects the traversal statistics for this ray type/scene/acceleration data structure combination. This pass is quite slow because of the heavy instrumentation, therefore we only keep the statistics and do not benchmark its execution speed.

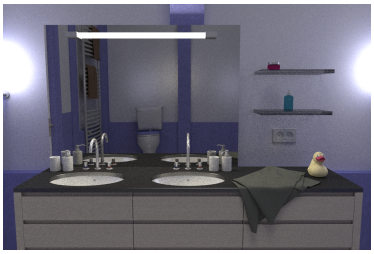
2. The second pass runs the traversal replay again with the *perf* profiling tool. This pass is a dummy pass, we will load the scene and build the acceleration data structure, but exit just as we should start to trace the rays.

3. The final pass runs the traversal again with *perf* but performs the full run at normal speed (The instrumentation of *perf* uses hardware performance counters and does not slow down execution).

Thanks to the dummy run, we can estimate the *perf* metrics for ray traversal only by subtracting the counter values of the final pass with the ones from the dummy pass. These numbers are imperfect, however they should still give us a better estimate than using the combined numbers directly.

4.4.1 Test scenes

In this work we used scenes from the set shown in Figure 4.2.



(a) “Bathroom”

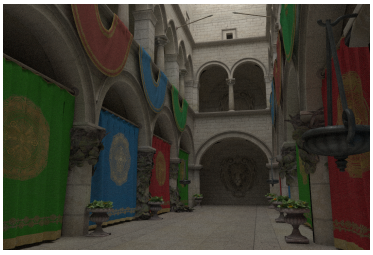
(b) “Bedroom”

(c) “Wooden Staircase”

(d) “Dining Room”

(e) “Kitchen”

(f) “Living Room”



(g) “Crytek Sponza”

Figure 4.2: The test scenes used for benchmarking

Chapter 5

Results and discussion

5.1 Benchmark results analysis

Our benchmarks yielded a considerable amount of information, letting us compare different data structures and variants of those, in different scenes, for different types of rays and with possibly different parameters. We start this section by providing our raw traversal figures in Section 5.1.1 then move on to discuss more specifics on certain type of structures, rays or scenes and finally talk about other metrics.

5.1.1 Raw ray traversal performance

Table 5.1: Traversal rate for primary rays (Mrays/s)

Structure	Staircase	Sponza	Living room	Kitchen	Dining room	Bedroom	Bathroom
KD_no_ps	34.4	20.9	31.9	45.2	33.7	44.7	28.2
KD_positional	42.7	24.0	37.4	46.1	40.7	44.1	33.5
KD_basic	46.5	25.5	38.4	47.6	43.5	47.9	35.3
KD_restart	23.3	9.7	DNF	24.3	20.5	22.7	DNF
KD_ropes	39.1	23.2	31.8	40.7	37.8	42.4	31.3
Regular grid	9.4	2.2	5.1	5.1	7.3	6.0	8.0
Grid+occl mipmap	12.2	2.1	14.7	13.5	13.2	13.4	15.3
Grid+dist field	12.8	2.1	16.0	17.2	14.3	15.4	18.8
Macro regions	21.6	2.2	15.1	22.0	16.7	18.7	19.1
Two-level grid	17.5	10.0	12.2	11.4	13.4	12.1	18.8
LBVH	22.1	11.3	26.6	21.3	27.5	20.9	27.6
PLOC	23.1	12.2	30.3	28.7	35.5	25.3	32.8
SAH	24.8	12.4	31.3	27.7	29.3	25.9	33.5
SBVH	29.6	15.4	33.0	35.5	35.5	26.5	38.6
LBVH*	28.7	15.9	33.4	29.3	37.7	29.3	33.4
PLOC*	27.9	15.5	33.3	31.2	38.0	30.2	34.2
SAH*	27.5	17.2	34.2	32.7	37.0	28.0	34.7
SBVH*	30.6	16.7	38.8	38.8	40.7	31.6	39.6

5.1.2 Breakdown by type

Understanding the previous tables is not very easy without some visual aid. There is a lot to these numbers, and we have more than the traversal rate to look at. First we

Table 5.2: Traversal rate for bounce rays (Mrays/s)

Structure	Staircase	Sponza	Living room	Kitchen	Dining room	Bedroom	Bathroom
KD_no_ps	21.6	10.4	18.6	22.4	16.1	19.0	16.9
KD_positional	24.4	11.3	20.3	23.1	19.5	19.1	17.7
KD_basic	26.0	12.0	20.9	24.3	20.8	20.5	19.0
KD_restart	17.9	7.2	13.9	16.5	12.9	13.6	DNF
KD_ropes	24.7	9.6	17.3	21.9	18.1	17.1	17.3
Regular grid	7.8	2.6	2.4	2.6	3.2	2.6	2.8
Grid+occl mipmap	9.2	2.5	7.6	7.1	6.3	6.7	7.1
Grid+dist field	8.4	2.6	5.7	5.4	5.4	5.0	5.5
Macro regions	14.2	2.7	9.6	10.8	7.3	8.7	8.9
Two-level grid	13.6	7.8	9.7	10.5	10.2	9.4	10.3
LBVH	18.2	5.9	14.2	15.1	15.1	12.0	15.7
PLOC	20.0	7.1	17.6	18.5	19.4	14.8	19.1
SAH	21.6	8.1	17.9	19.7	18.6	16.7	21.2
SBVH	28.7	9.9	22.0	27.2	23.5	20.0	26.1
LBVH*	23.6	9.3	19.7	21.4	21.1	17.5	20.7
PLOC*	23.1	9.0	20.1	21.6	21.4	17.6	21.8
SAH*	23.9	9.4	20.0	22.2	21.6	17.8	22.5
SBVH*	30.3	10.4	23.7	28.5	24.5	21.9	27.1

Table 5.3: Traversal rate for shadow rays (Mrays/s)

Structure	Staircase	Sponza	Living room	Kitchen	Dining room	Bedroom	Bathroom
KD_no_ps	14.1	12.4	30.0	17.7	14.2	16.2	13.2
KD_positional	16.6	13.0	31.5	17.8	17.7	15.5	14.9
KD_basic	17.5	13.7	32.4	19.0	18.8	16.7	15.9
KD_restart	10.1	8.4	20.3	11.5	11.1	10.4	10.1
KD_ropes	15.0	10.8	27.6	16.8	16.3	13.6	14.7
Regular grid	2.2	3.1	1.9	2.7	2.7	2.5	2.5
Grid+occl mipmap	2.4	3.2	8.3	5.3	5.4	5.2	4.7
Grid+dist field	2.3	3.3	5.4	4.3	4.4	3.7	4.2
Macro regions	2.6	3.5	8.7	6.8	6.1	6.7	6.2
Two-level grid	7.2	9.0	7.3	7.8	8.4	7.1	7.1
LBVH	12.1	11.6	22.6	14.2	16.4	11.8	20.3
PLOC	14.5	15.3	27.5	20.6	22.0	17.8	26.5
SAH	17.5	14.2	27.4	20.0	20.1	17.1	27.9
SBVH	20.5	16.6	36.7	25.4	24.4	20.6	31.8
LBVH*	18.3	14.6	28.7	22.0	22.6	17.5	26.7
PLOC*	18.3	16.5	30.2	23.4	23.6	18.6	29.0
SAH*	19.2	16.2	30.3	22.7	22.3	19.1	31.0
SBVH*	21.7	16.6	40.3	27.8	26.8	21.3	33.4

will perform a breakdown by type, sorting our acceleration data structures in three different categories: BVHs, Kd-trees and Grids.

BVHs

We begin by taking a look at the performance of bounding volume hierarchies in our test scenes in Section 5.1.2. We can first observe that the Sponza scene is unusually hard, we attribute this to the fact it features more geometric complexity and occlusion, rather than being mostly a single room like our other scenes. We will come back to this later.

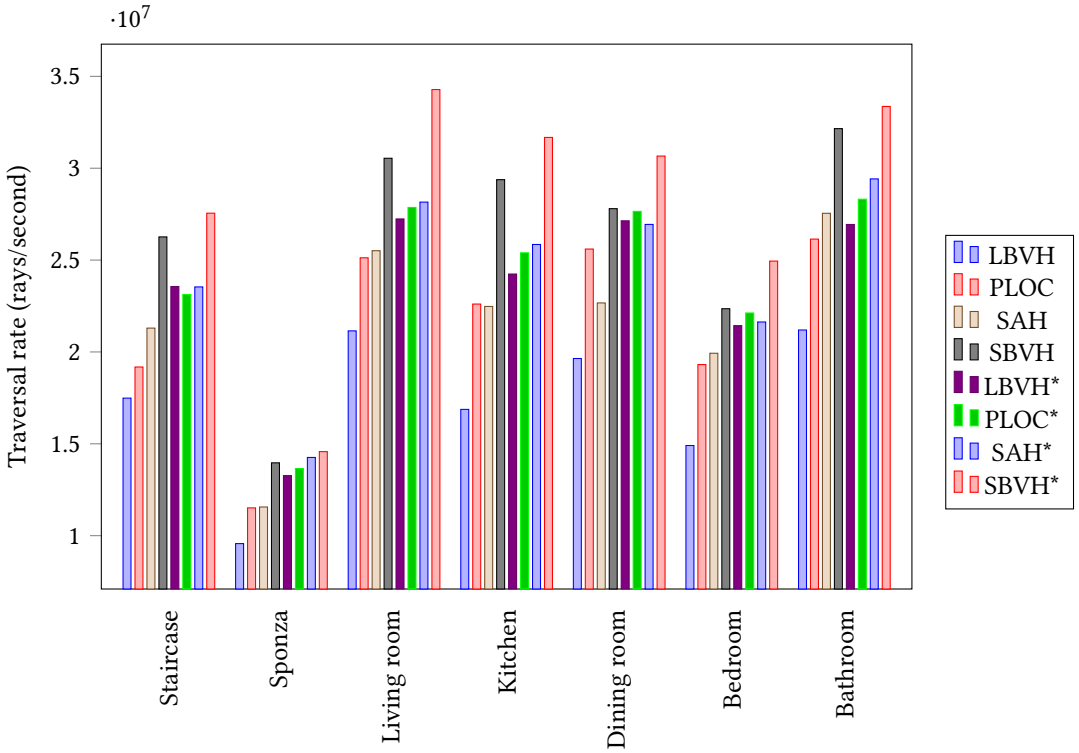


Figure 5.1: Traversal rate of scenes for different BVH variants (average rate for all ray types)

We can see that the SBVH builder clearly leads the pack, both with and without reinsertion applied. The reinsertion gives a boost to every builder type, but the LBVH one clearly gets the most out of it. We observe that the PLOC builders gets similar performance to the binned SAH one, which given PLOC is a data-parallel build, renders it more desirable overall.

In Figure 5.4a, we break out the performance of different ray types for all our BVH variants. Once more we can report a similar performance scale as in Section 5.1.2.

Kd-trees

For kd-trees, Section 5.1.2 tells a simple picture: the most "vanilla" kd-tree (stackful, with spatial splits) is the best. The direction-based intersection order is slightly better than the positional one.

Kd-restart performed very poorly, and actually failed to finish some runs because of what we believe to be an issue in our implementation, letting some rays enter into an infinite loop. The results for these failed runs were ignored in Figure 5.4b.

The stackless traversal using ropes fared much better, staying closer in performance to the stackful variants, more or less matching the variant with no perfect splits (kd_no_ps).

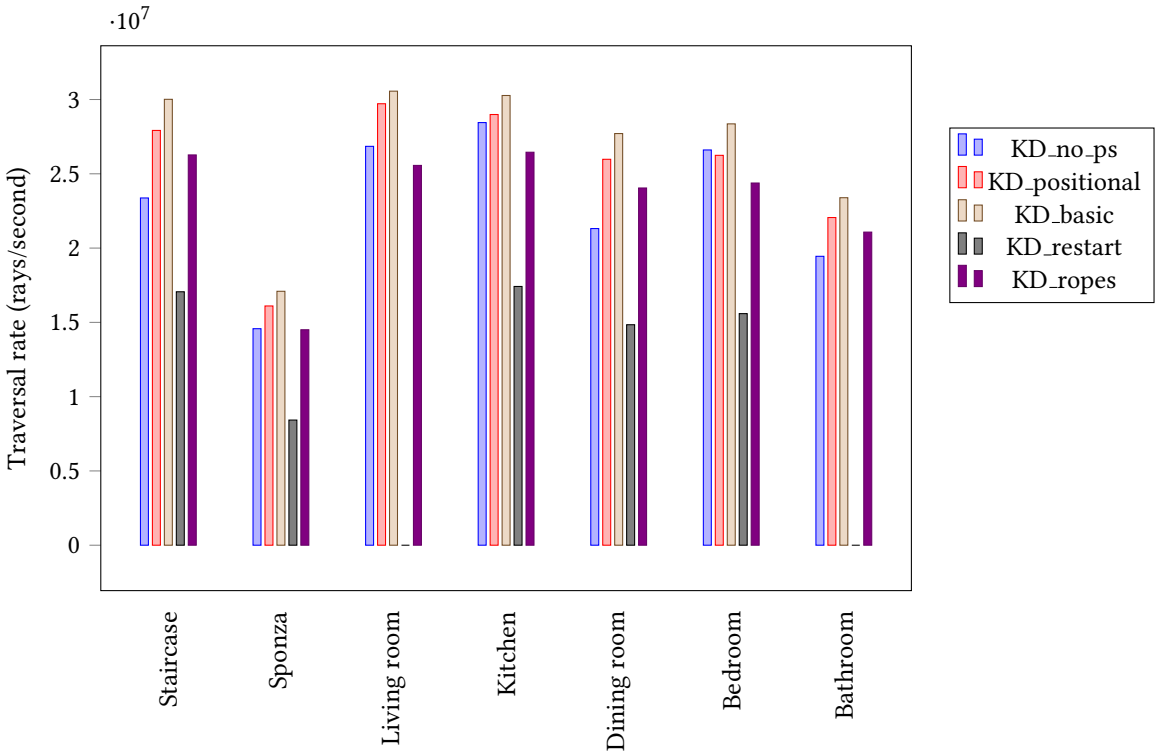


Figure 5.2: Traversal rate of scenes for different kd-tree variants (average rate for all ray types)

Grids

In Section 5.1.2 two types of grids emerge as being remotely competitive: Macro regions and two-level grids. In particular we can see that in the sponza scene, all grids perform extremely poorly, save for two-level grids which manages to stay in the same order of magnitude as other BVH and Kd-tree methods. Every other method yields awful performance.

Figure 5.4c sheds further light on the situation. The space-skipping methods (mipmaps, distance field, macro regions) all improve performance significantly and

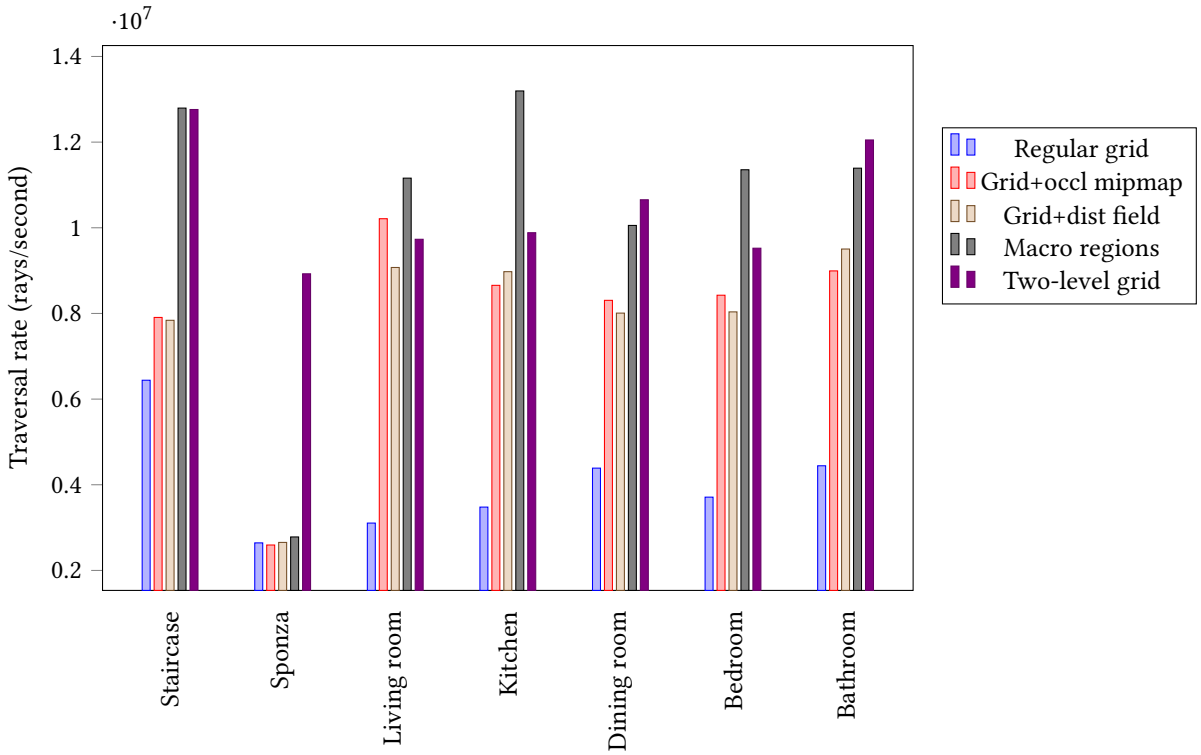


Figure 5.3: Traversal rate of scenes for different grids variants (average rate for all ray types)

work especially well for primary rays. Two-level grids have slightly more modest primary ray performance but dominate in the bounce rays and shadow rays metric.

5.1.3 Overall results

To keep the following graphs readable and informative, we picked the two best performing variants or otherwise relevant forms in each category of acceleration data structures:

1. For BVHs we used SBVH and PLOC*, the first because it is one of the best performers, the second for being a viable approach for interactive scenes.
2. For kd-trees we compare the stackful traversal with the direction-based strategy and the stackless one using ropes. Both made use of perfect splits in their builds and were in fact using the same base trees.
3. For grids, we kept macro regions and two-level grids as not only they were the best performers, but they improve on regular grids each in a different direction.

We plot their results in Figure 5.5 and Figure 5.6 for reference. Overall BVHs and kd-trees work best, with the best efforts of grids only rarely getting close (notably in the Sponza scene).

5.1.4 Memory accesses

We used our memory accesses tracking to generate rough bandwidth metrics for our acceleration structures (Figure 5.7, Section 5.1.4). The graphs feature a dotted line for the theoretical limit of the main memory of our system: 38.4 gigabytes/second for dual-channel DDR4-2400. We can see some structures get above that rate, that can be explained by the action of caches.

5.1.5 Other metrics

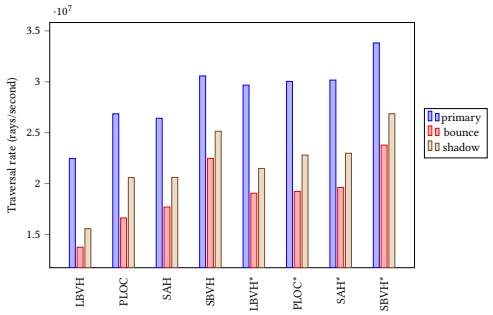
Since we ran all our benches with the "perf" tool, we have many metrics on those as well. We compiled a few interesting ones in this section: Figure 5.9 shows us the rate of L1 data cache loads (considering a 64 bytes cache line size on x86_64). Figure 5.10 gives us the rough rate of memory accesses as recorded by our instrumentation. Finally Figure 5.11 gives us the rate of primitives intersected.

5.2 Build times

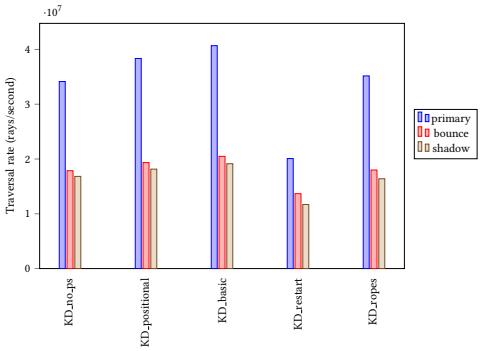
We collected the build times in Table 5.4. There was seemingly an issue with the reinsertion optimization when applied to the LBVH and PLOC builders, leading to anomalously high build times. Since many of those algorithms are meant to run on a massively data-parallel processor like a GPU, these numbers are not very insightful.

We can see however that kd-trees are quite slow to build in all cases, but adding the ropes was practically free. We did not benchmark the other kd-tree variants since they use the exact same algorithm, with no alterations other than commenting out for primitive clipping for the variant without perfect splits.

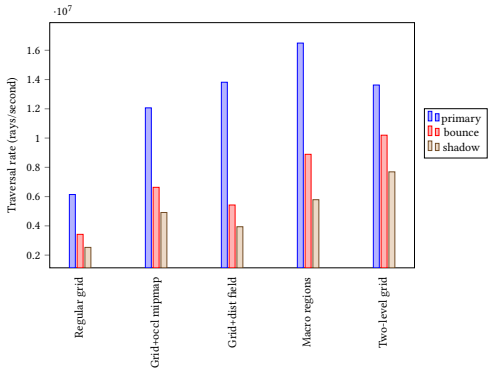
In terms of grids, they were all extremely fast to build, but computing the distance field and especially macro regions turned out to be very slow.



(a) Averaged traversal rate different BVH variants for each ray type across all scenes



(b) Averaged traversal rate different kd-tree variants for each ray type across all scenes



(c) Averaged traversal rate different grids variants for each ray type across all scenes

Figure 5.4: Performance for each ray type in different acceleration data structures

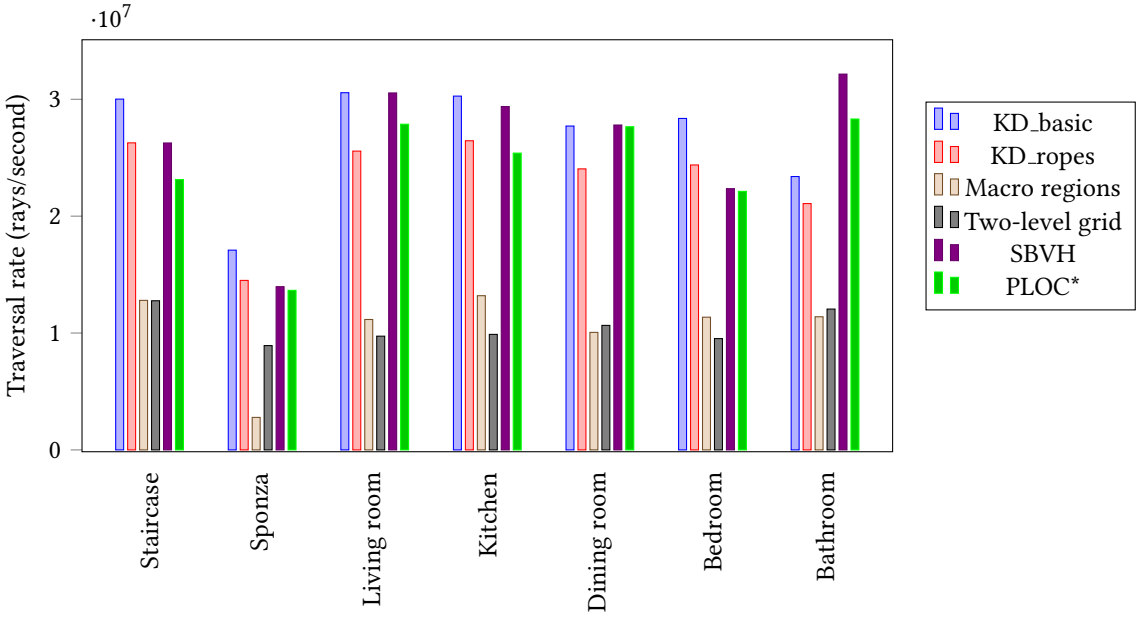


Figure 5.5: Traversal rate of scenes for the best-performing variants of all three categories

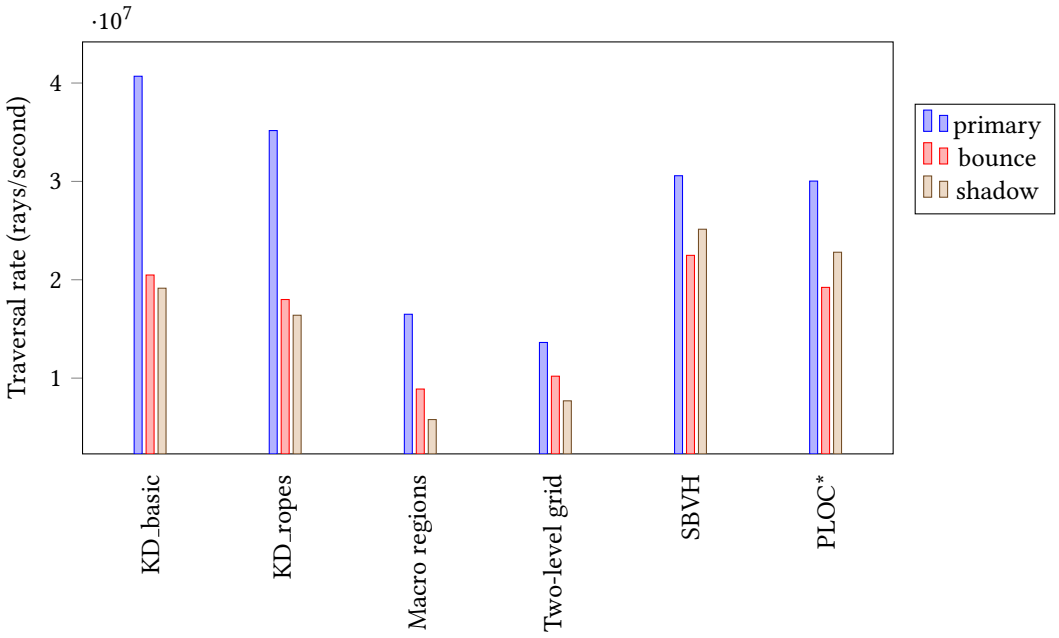


Figure 5.6: Averaged traversal rate for each ray type across all scenes

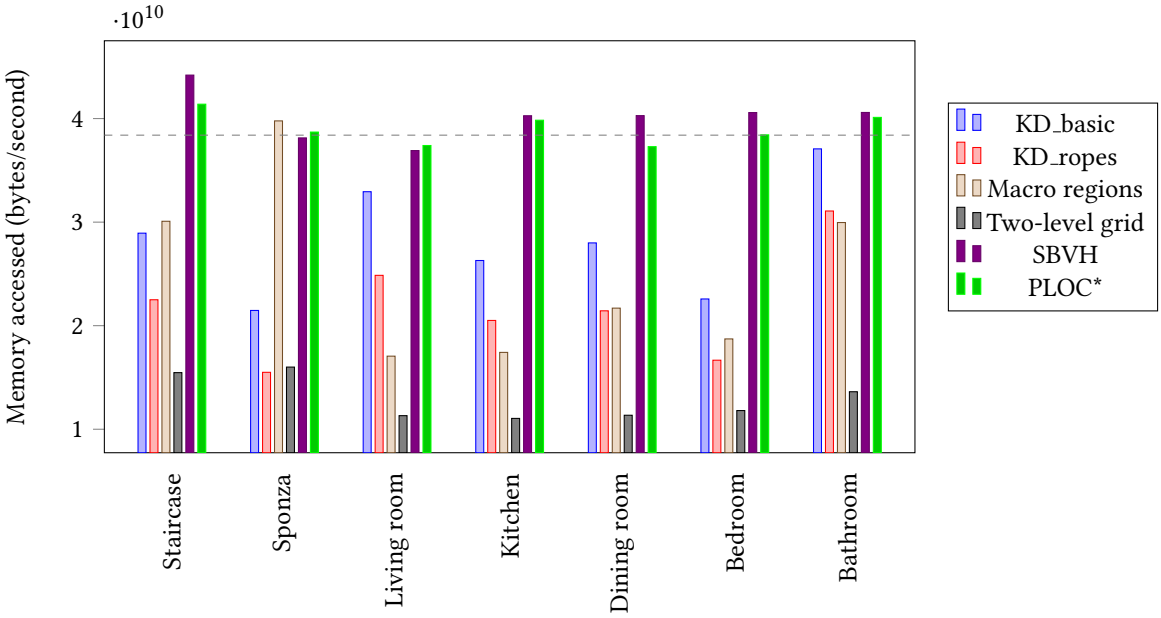


Figure 5.7: Memory rate of scenes for the best-performing variants of all three categories

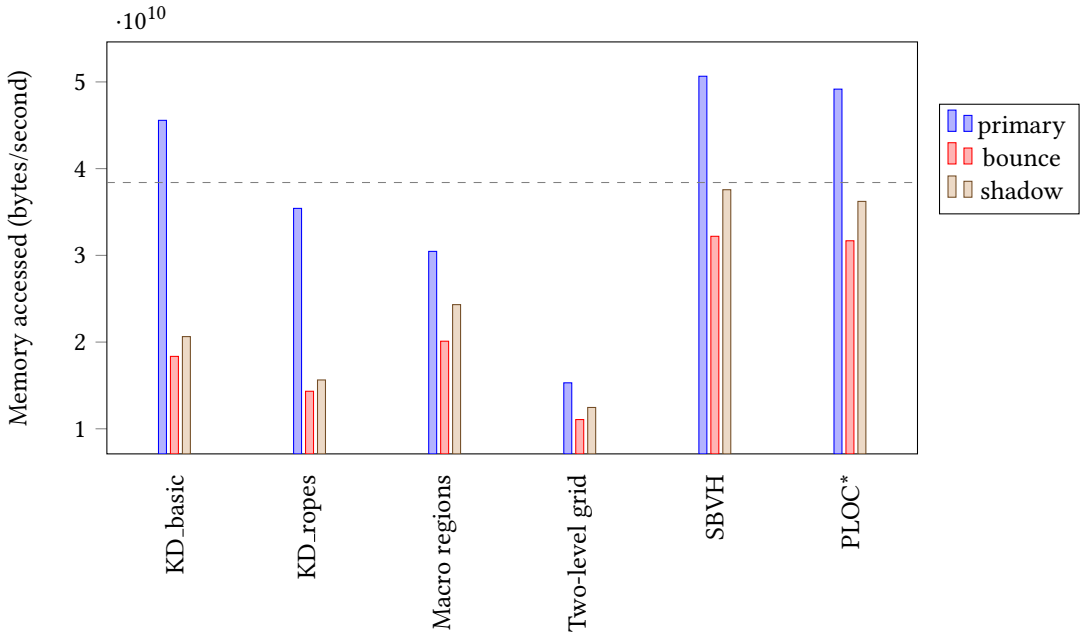


Figure 5.8: Averaged memory rate for each ray type across all scenes

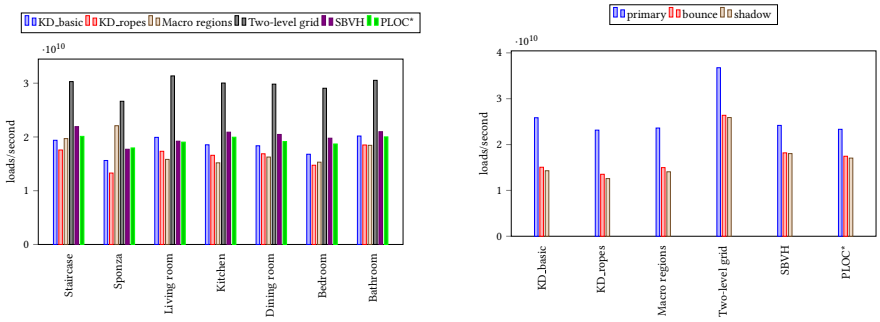


Figure 5.9: L1 data loads rate

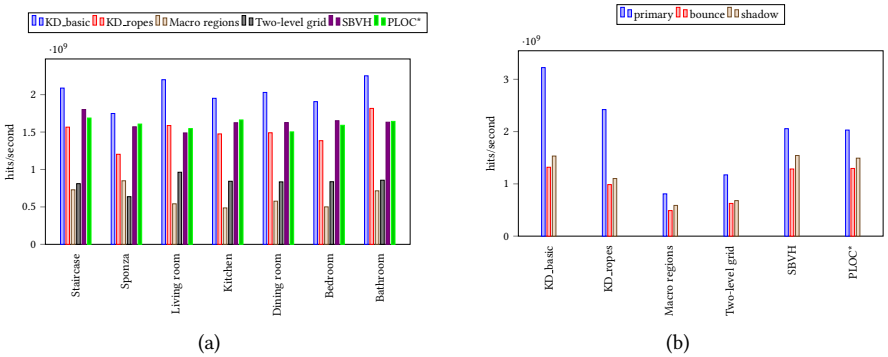


Figure 5.10: Memory hits rate

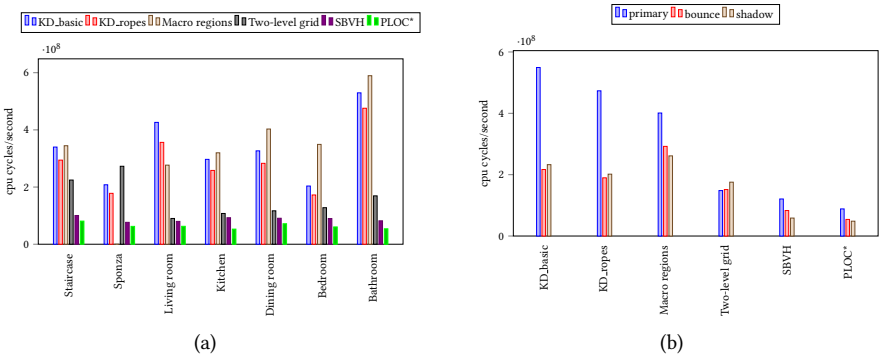


Figure 5.11: Primitives intersection rate

Structure	Staircase	Sponza	Living room	Kitchen	Dining room	Bedroom	Bathroom
LBVH	74	85	213	175	222	152	136
PLOC	85	87	235	204	258	170	162
SAH	96	98	257	217	288	181	173
SBVH	393	442	1421	816	1421	875	688
LBVH*	470	41868	5467	1408	13958	16247	1047
PLOC*	392	76809	5862	1870	11949	13423	932
SAH*	213	294	653	556	652	433	395
SBVH*	478	554	1740	1072	1728	1075	894
KD_basic	4247	3852	11613	9314	12521	8554	7344
KD_ropes	4160	3858	11714	9316	12663	8771	7390
Regular grid	73	55	102	122	175	153	90
Grid+occl mipmap	102	81	170	185	264	203	161
Grid+dist field	8272	9259	66815	31108	62578	22390	43840
Macro regions	654	279	80817	3667	4760	3873	4254
Two-level grid	205	185	506	417	564	465	369

Table 5.4: Build times for all scenes and data structures (milliseconds)

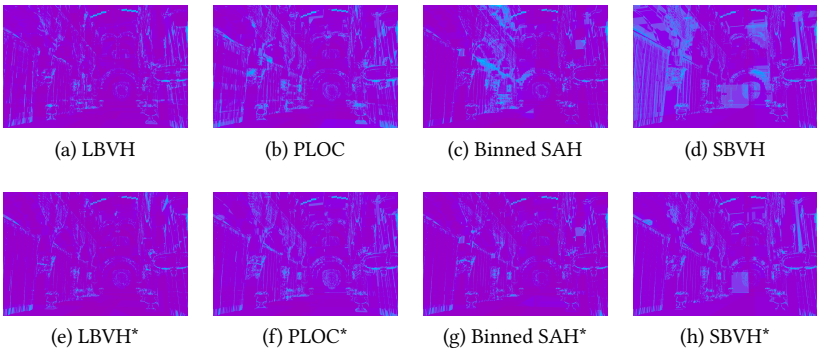
5.3 Heatmaps analysis

As described in Chapter 4, we have the ability to generate heatmaps for various metrics. Once again giving figures for all possible combination of ADS and scenes would not be practical so we will use them sparsely to discuss what the raw numbers don't show so well.

5.3.1 Intersected primitives

Probably the most important job for any acceleration structure, as it is the motivation for having them in the first place, is to minimize how many ray-primitive checks we do. We used the Bedroom and Sponza scenes, and generated images for different structures. This allows us to compare the quality of different variants of acceleration structures.

BVHs



(a) LBVH (b) PLOC (c) Binned SAH (d) SBVH

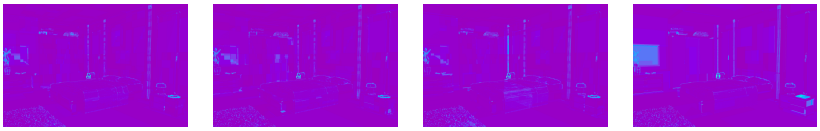
(e) LBVH* (f) PLOC* (g) Binned SAH* (h) SBVH*

Figure 5.12: Intersected primitives heatmap for the Sponza scene with BVH variants

In general we can say BVHs do a very good job of avoiding primitive intersection,

even the very low quality ones. In fact we observe the warmest results in the best-performing Split BVH builder, which is mildly surprising.

(a) LBVH (b) PLOC (c) Binned SAH (d) SBVH

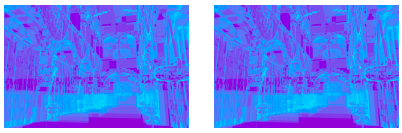


(e) LBVH* (f) PLOC* (g) Binned SAH* (h) SBVH*

Figure 5.13: Intersected primitives heatmap for the Bedroom scene with BVH variants

Kd-trees

(a) KD_no_ps (b) KD_basic (c) KD_positional



(d) KD_restart (e) KD_ropes

Figure 5.14: Intersected primitives heatmap for the Sponza scene with kd-tree variants

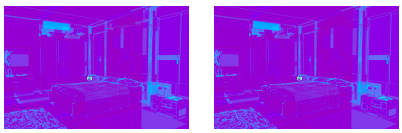
Figures 5.14 and 5.15 remind us of one very simple and important fact about our KD-trees variants and that is they all use the same top-down builder SAH-aware. Therefore, they will all have the same efficacy in culling primitives, because they will visit the same leaves.

In fact the one outlier is the KD_no_ps variant which sole specificity is that it does not consider perfect splits, resulting in worse performance, and hotter heatmaps.

Grids

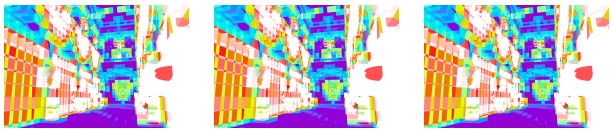
Figures 5.16 and 5.17 tells us something interesting about our grids variants and their performance: Regular grids are awful at culling primitives and all empty space-skipping variants directly inherit that property. Only two-level grids manage an improvement in primitive intersections, because those actually have the ability to adapt their resolution to the scene.

(a) KD_no_ps (b) KD_basic (c) KD_positional

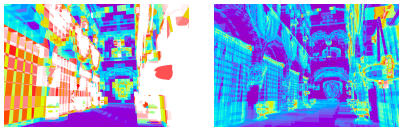


(d) KD_restart (e) KD_ropes

Figure 5.15: Intersected primitives heatmap for the Bedroom scene with kd-tree variants



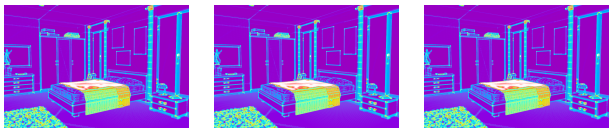
(a) Regular grid (b) Mip grid (c) Distance field



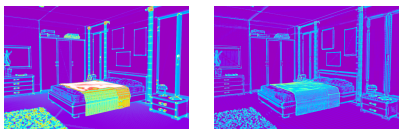
(d) Macro regions (e) Two-level grid

Figure 5.16: Intersected primitives heatmap for the Sponza scene with grids variants

The Sponza scene is particularly susceptible to this because its geometry is enclosed in a very large cube serving as the sky. This specificity makes it extra hard for grids to cope with, and largely explains the massive performance cliff suffered in Section 5.1.2, and showcases the better handling of these “teapot in a stadium” scenarios with two-level grids.



(a) Regular grid (b) Mip grid (c) Distance field



(d) Macro regions (e) Two-level grid

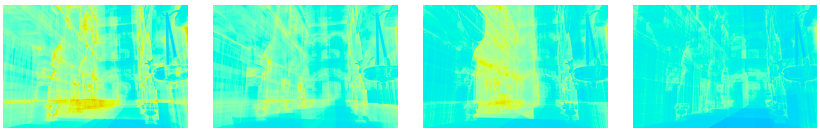
Figure 5.17: Intersected primitives heatmap for the Bedroom scene with grids variants

5.3.2 Traversal steps

Having looked at the heatmaps for intersected primitives, we know the raw amount of intersected primitives is not all there is to an acceleration data structure: Kd-trees have quite good performance even though their intersected primitives maps are resolutely hotter, and grids do not perform nearly as badly as the earlier heatmaps would suggest.

What those were missing in some kind of indication for the time spent traversing the structure itself. Since benchmarking the traversal time for singular rays would be impractical, we simply used the number of traversal steps from Section 4.2 as an approximation. While the results won't be apple-to-apples between categories of data structures, they are still informative and allow further discussion.

BVHs

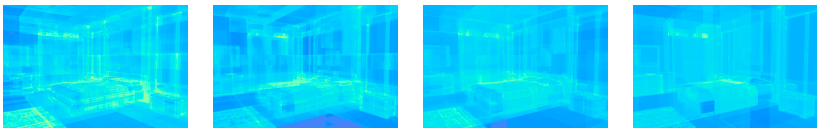


(a) LBVH (b) PLOC (c) Binned SAH (d) SBVH

(e) LBVH* (f) PLOC* (g) Binned SAH* (h) SBVH*

Figure 5.18: Traversal steps heatmap for the Sponza scene with BVH variants

The heatmaps in figures 5.18 and 5.19 do not tell us a whole lot of new things from what we learned in Section 5.1.2. We can see that the reinsertion is able to clear up the anomalously hot regions in the center of the frame for the Sponza scene when using the binned SAH builder.

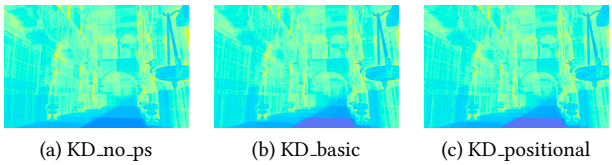


(a) LBVH (b) PLOC (c) Binned SAH (d) SBVH

(e) LBVH* (f) PLOC* (g) Binned SAH* (h) SBVH*

Figure 5.19: Traversal steps heatmap for the Bedroom scene with BVH variants

Kd-trees



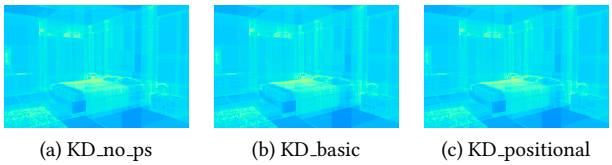
(a) KD_no_ps (b) KD_basic (c) KD_positional

(d) KD_restart (e) KD_ropes

Figure 5.20: Traversal steps heatmap for the Sponza scene with kd-tree variants

In Figure 5.20 and to a lesser extent fig. 5.21 we can witness how poorly the KD_restart variant fairs in terms of traversal steps. By its very nature kd-restart can and will significantly increase the numbers of traversal steps since it can mean traversing the tree from the top many times when an intersection is not immediately found.

The ropes variant actually fairs very well here too, closely following the stackful variants.



(a) KD_no_ps (b) KD_basic (c) KD_positional

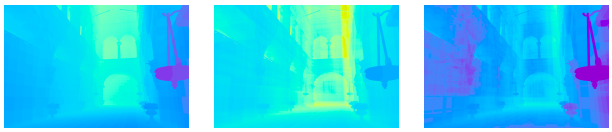
(d) KD_restart (e) KD_ropes

Figure 5.21: Traversal steps heatmap for the Bedroom scene with kd-tree variants

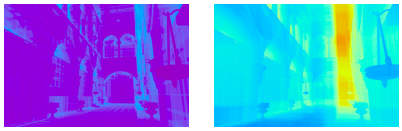
Grids

There are two main things going on with the traversal step heatmaps for grids. Firstly, we again see regular grids being the red lanterns, with fig. 5.23a being the worst looking heatmap so far. The problem here is that the resolution is too fine, and we make many redundant traversal steps. The situation is better in fig. 5.22, but only because the grid resolution is so low, resulting in way too many intersection tests as seen earlier.

This is where the empty space skipping variants get to really shine. By far the best performer is macro regions, which basically just jumps from a large block of empty

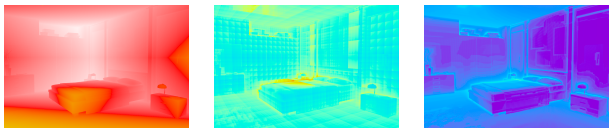


(a) Regular grid (b) Mip grid (c) Distance field

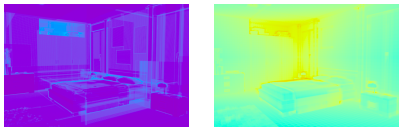


(d) Macro regions (e) Two-level grid

Figure 5.22: Traversal steps heatmap for the Sponza scene with grids variants



(a) Regular grid (b) Mip grid (c) Distance field



(d) Macro regions (e) Two-level grid

Figure 5.23: Traversal steps heatmap for the Bedroom scene with grids variants

space to another. Next comes the distance field, and then the mipmapped grid and the two level grid close the march.

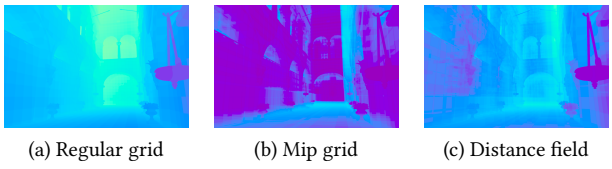
We'd like to point out that this traversal steps metric is very imperfect: we count a traversal step every time we change to another mip-level for the mipmapped grid, these vertical (through the stack of mip levels) traversal steps, while not free, are not as costly as horizontal (through the actual 3d space) steps, which inflates the perceived overhead of the mipmapped grid.

For grids specifically, we thus also include heatmaps for the number of *cells* visited, available in Figure 5.24 and Figure 5.25. This visualization gives the mip mapped approach to empty space skipping more credit, but now the distance fields look worse, even though (for primary rays anyways) it performs better.

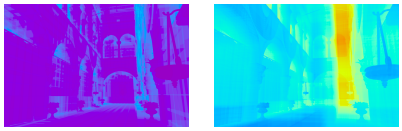
5.4 Discussion

5.4.1 On BVHs versus kd-trees

While BVHs and kd-trees did similarly well in the overall traversal rates (Figure 5.7), we can see that BVHs won the "real test", that is the bounce and shadow rays performance (Figure 5.6), and the kd-tree win was in primary rays. Considering that incoherent bounce and shadow rays are not only more numerous in a typical renderer, but also



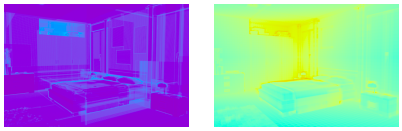
(a) Regular grid (b) Mip grid (c) Distance field



(d) Macro regions (e) Two-level grid

Figure 5.24: Visited cells heatmap for the Sponza scene with grids variants

(a) Regular grid (b) Mip grid (c) Distance field



(d) Macro regions (e) Two-level grid

Figure 5.25: Visited cells heatmap for the Bedroom scene with grids variants

not amenable to more the efficient rasterization method like primary rays are, this makes kd-tree performance less interesting that the overall numbers first suggest.

A major issue with kd-trees is how their only build algorithm is a top-down recursive build. This kind of build is much harder to parallelize, given the strong data dependencies to the last recursion step. Not only that but those builds are slow to begin with, and their best algorithms are either complex to implement [49] or rely on binning approximations, which BVHs can also do with better results [47]. The notion of "perfect splits" (See Figure 3.22) is known to be important, but creating a botton-up builder that takes them in consideration is an open question.

Furthermore, BVHs have other unique advantages: It is possible to quickly "re-fit" a BVH to changing geometry and keep reasonable performance [48], an operation that makes no sense in a kd-tree. Node reinsertion methods (See Section 3.1.8) can optimize BVHs further than a greedy build can achieve. Beating a purely greedy kd-tree build by considering even two split levels at once means trying an exponentially increasing set of possibilities.

Finally, the starting advantage of kd-trees (the efficacy of spatial splits) are no longer exclusive to them: Spatial splits have been applied to BVHs with great results (See Section 3.1.7), and other pre-splitting techniques can achieve a similar effect. In the end the kd-trees do conserve one advantage, and that is their superior ability to perform early ray termination thanks to the guaranteed lack of node overlaps.

In 2008 Ingo Wald reported that

> "by 2005 virtually all fast ray tracers were built on kd-trees" [50]

Now, 15 years later in 2020, we believe the opposite statement to be correct, with virtually all educational material and open source renderers referring to BVHs as *the* data structure for raytracing acceleration. In fact this is so significant that finding modern resources on kd-trees proved quite hard, and we had to rely on older literature.

This sentiment is further echoed in the de-facto standardization on BVHs for fixed-function raytracing units that made it in consumer hardware [8], and the accompanying real-time graphics APIs [53]. These real-time applications rely on fast (re)builds to be viable, and typically combine raytracing with rasterization for primary rays, leaving the choice between BVHs and kd-trees obvious.

5.4.2 Limitations and flaws with BVHs

There are two important remaining issues we see with BVHs today. First is their lack of ability to terminate the traversal instantly like in a grid or a kd-tree, leading to wasted time after hits unwinding a stack to check for a better one in the intersecting regions between the visited nodes and the nodes scheduled for traversal.

The reader might have picked up on the fact shadow rays are significantly faster to trace in BVHs than other types of incoherent rays. We argue this is a *flaw* with BVHs: it is not so much that their any-hit traversal is any faster, it is rather that nearest-hit rays are slower due to this larger overhead to prove a hit as being the nearest one.

A BVH with spatial splits and strict non-aliasing rules for object splits (rejecting partitions that cause two children of a node to share space) could restore this ability to terminate early, but we do not know of previous research in this direction.

The second issue, which all other structures also suffer from, is the difficulty in handling non-axis-aligned geometry. Because of the nature of the most common bounding volume used, i.e. *axis-aligned* bounding boxes, some primitives just cannot be efficiently bounded, leaving us with poor SAH costs. Using other volumes such as Oriented Bounding Boxes (OBBs) helps with this problem, but results in considerably slower ray-volume intersections because less can be pre-computed for the whole ray. There has been some work on reducing that cost [52], and we believe there is more to be done in that area.

5.4.3 Relevance of stackless traversal

We have seen that all the stackless traversal approaches we tried performed worse than their stackful equivalents. This isn't a new finding, but the utility of saving memory for a stack is decreased today thanks to improvement in both the flexibility of GPU hardware and memory sizes. We unfortunately did not explore GPU traversal ourselves to make a definitive claim about this, but we do believe the stackless variants of stackful algorithms to be of little interest for common applications given the tradeoffs involved, and especially so on CPU-like devices.

If stackless traversal is absolutely necessary, we strongly believe approaches based on ropes are the most elegant and fastest, even considering the memory overhead. Again, there are data structures that do not rely on a stack to begin with, namely grids. Depending on the mental model of the reader, it might be insightful to consider space-skipping methods in grids as "ropes". We believe that grids are under-explored as raytracing acceleration data structures, as we will discuss below.

5.4.4 Revisiting grids

Our grids showed poor performance, but interestingly the top two performers had very different bottlenecks. Macro regions were slow because of the poor resolution of their underlying grid leaving them unable to cull primitives very effectively. But two-level grids were slow because the traversal of the grid itself was slow and no empty space skipping was applied ! Otherwise the primitive culling performance of the two level grid, while not astounding compared to the tree-like structures, was a huge step-up from regular grids derivatives. The obvious thing to do here would to apply empty space skipping to a two-level grid to reduce its internal traversal cost.

A similar idea was the subject of a recent paper by Pérard-Gayot et al [38], that we unfortunately did not have time left to implement ourselves. While the authors did provide their implementation, we would have had to re-implement it on the CPU in our framework to have comparable numbers. Irregular grids start from a two-level grid and go one step beyond just empty space skipping: Merging cells with similar content, allowing to jump through not only empty space, but also non-empty space. Irregular grids are also built with regards to the SAH heuristic, which is not the case for other grids.

The reported performance of Irregular grids is meeting or exceeding that of SBVH trees, which is very encouraging for grids as a whole. We believe there is value in exploring new variants of grids and that their inherent lack of need for a stack is a huge bonus for any GPU implementation.

5.5 The perfect acceleration data structure

It is hard to say with certainty if the current state of the art is as good as it gets, if we are close to some optimum where there are only diminishing returns beyond. While by no means "perfect", it is interesting to note how *all* data structures studied here still managed to reduce the traversal time by many orders of magnitude from a naïve $O(N)$ brute-force approach.

We see two major components to the efficiency of any acceleration data structure: Their effectiveness in culling primitives (how many primitives had to be actually intersected to arrive at an answer), and the overhead of traversing the data structure itself. A good acceleration data structure needs to be strong on both counts, otherwise it will end up looking like our macro regions or two-level grids.

Going by everything we've seen, we can take a few guesses on how the ideal structure would look like:

1. First it should be a spatial data structure with no overlapping children for a given level, so we can always perform ERT.

2. Next it should be as flat as possible, but without giving up the ability to arbitrarily add detail where needed.

3. It should require either no stack or be content with a very small (4-8 entries) one, and computing the next cell should be easy and not involve complicated sorting.

4. Finally it should consider the SAH or a future heuristic like it, and be effective in culling as many primitives as possible with very tight volumes.

While many of the studied structures certainly have some of those characteristics, there is none that truly excels in all of them, so we believe there is still significant future work to be done in this field.

For the time being, we can report that SBVHs are a state of the art data structure as long as build times are unconstrained, and the parallel locally ordered clustering (PLOC) build is our pick for a good quality and GPU-friendly parallel build. Depending on how many rays are traced, it is possible that a slower to traverse but faster to build structure, like a linear BVH or even a regular grid, can outperform the "better" techniques when build times are added to the traversal times.

In such combined scenarios however, we believe kd-trees are never desirable, due to their long and non-parallel builds. We in fact question if kd-trees are ever desirable given all the cons outlined above.

Chapter 6

Conclusions

We have introduced the problem of tracing rays and finding intersections, and looked at its most popular applications in rendering. We have studied the question of data structures for accelerating those intersection searches, and outlined many of the important characteristics that they can offer. We have looked at the performance factors of modern computing, and related them to the challenges of tracing rays rapidly.

We have went over a significant body of previous work and reported on many build and traversal approaches for the three categories of data structures we identified: BVHs, kd-trees and grids. We provided informative descriptions of many of those, along with many diagrams, code samples and visualizations. We then implemented much of the discussed prior work in a (single ray) renderer, and computed performance figures for a selection of common scenes, and produced many graphs and complexity visualizations from those.

From these numbers we have been able to identify BVHs and kd-trees as the top performers, and grids as less performant. However we also found that BVHs are actually more desirable than kd-trees in many ways and explained the relative unpopularity of the latter in the 2010s and beyond. We also discussed the reasons for the poor performance of our grids variants, and argued that other grid variants could in fact be some of the best structures for real-time ray tracing.

Throughout the text, we picked at what makes a good acceleration data structure, and we came to a general conclusion and pointed at what the future structures could be like. We report that BVHs are the best structures today, but we believe grids have large untapped potential.

6.1 Future work

As discussed in chapter 5, we think one of the major rooms for improvements in BVHs is making them better at handling problematic primitives (As seen in Figure 3.7). One possible way to address that without resorting to more costly orientable bounding volumes would be to offer a few multiple *spaces* to encode axis-aligned bounding boxes in. Figure 6.1 shows an example of this in the 2D case.

The point of such an approach is that the bounding box intersection tests are not any more costly than regular axis-aligned boxes, since those are *still* axis-aligned boxes, with the only twist being there is now a handful of possible coordinate spaces

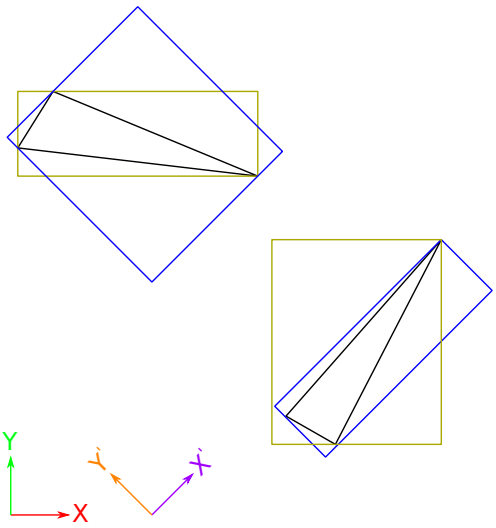


Figure 6.1: Each primitive gets two bounding boxes, a yellow and a blue one, one in each space. We can pick the best for each primitive.

for them to be aligned to. The overhead for each ray of computing shared constants for two or four sets of axises rather than just one is minimal compared to the overhead of doing that for possibly *every* bounding box we hit, like we do if they can be arbitrarily oriented. Not only that but those constants can be computed ahead of time at the start of the ray and for all coordinate spaces in parallel.

Building a tree with bounding boxes in multiple spaces would still be practical in a top-down fashion, since the number of new possibilities is just multiplied by a small constant. The real challenge for such a technique will be the points where an inner node will have to tightly bound children in different spaces. Going further we could look into applying existing research on k-dops [29] (volumes bounded by k intervals in arbitrary axises) in the collision detection field.

The other major future work area, as mentionned earlier in Chapter 5, is to study "smart" grids such as irregular grids [38] and potential variations on them. One interesting idea is to store the *difference* in primitives at each cell boundary, avoiding having to re-intersect primitives that were already present in the previous cell. In fact, such a concept was already investigated in the writing of this thesis, but has not yielded presentable results yet.

Bibliography

[1] Timo Aila, Tero Karras, and Samuli Laine. On quality metrics of bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, page 101–107, New York, NY, USA, 2013. Association for Computing Machinery. 23, 29

[2] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, page 145–149, New York, NY, USA, 2009. Association for Computing Machinery. 14

[3] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. *Proceedings of EuroGraphics*, 87, 08 1987. 49

[4] Jiri Bittner and Peter Wonka. Visibility in computer graphics. *Environment and Planning B: Planning and Design*, 30:729–755, 09 2003. 10

[5] Jiří Bittner, Michal Hapala, and Vlastimil Havran. Fast insertion-based optimization of bounding volume hierarchies. *Comput. Graph. Forum*, 32:85–100, 2013. 35

[6] Solomon Boulos. Notes on efficient ray tracing. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, page 10–es, New York, NY, USA, 2005. Association for Computing Machinery. 39

[7] Carlos Carvalho. The gap between processor and memory speeds. 01 2002. 13

[8] NVIDIA Corporation. Nvidia turing gpu architecture, 2018. 5, 76

[9] Vasco S. Costa and João Madeiras Pereira. Multi-level grid strategies for ray tracing - improving render time performance for row displacement compressed grids. In *GRAPP*, 2010. 50

[10] Holger Dammertz, Johannes Hanika, and Alexander Keller. Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. *Comput. Graph. Forum*, 27:1225–1233, 06 2008. 22

[11] Holger Dammertz and Alexander Keller. The edge volume heuristic – robust triangle subdivision for improved bvh performance. pages 155 – 158, 09 2008. 30

[12] Olivier Devillers. The Macro-Regions,an Efficient Space Subdivision Structure for Ray Tracing. In *Eurographics*, pages 27–38, Hambourg, Germany, 1989. 50, 55

[13] Alphan Es and Veysi Isler. Accelerated regular grid traversals using extended anisotropic chessboard distance fields on a parallel stream processor. *J. Parallel Distributed Comput.*, 67:1201–1217, 2007. 50

[14] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. 01 2005. 45

[15] A. S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–24, 1984. 45

[16] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, 1987. 23, 36

[17] Toshiya Hachisuka, Shinji Ogaki, and Henrik Wann Jensen. Progressive photon mapping. In *ACM SIGGRAPH Asia 2008 Papers*, SIGGRAPH Asia '08, New York, NY, USA, 2008. Association for Computing Machinery. 54

[18] M. Hapala and V. Havran. Review: Kd-tree traversal algorithms for ray tracing. *Computer Graphics Forum*, 30(1):199–213, 2011. 43

[19] Michal Hapala, Tomáš Davidovič, Ingo Wald, Vlastimil Havran, and Philipp Slusallek. Efficient stack-less bvh traversal for ray tracing. *Proceedings - SCCG 2011: 27th Spring Conference on Computer Graphics*, 04 2011. 41

[20] Vlastimil Havran and Jiri Bittner. On improving kd-trees for ray shooting. *Journal of WSCG, v.10, 209-216 (2002)*, 10, 04 2002. 46

[21] Vlastimil Havran, Tomás Kopal, Jiri Bittner, and Jiri Zara. Fast robust bsp tree traversal algorithm for ray tracing. 12 1998. 43

[22] James T. Kajiya. The rendering equation. In *Computer Graphics*, pages 143–150, 1986. 8, 54

[23] Javor Kalojanov, Markus Billeter, and Philipp Slusallek. Two-level grids for ray tracing on gpus. *Comput. Graph. Forum*, 30:307–314, 2011. 47, 51, 55

[24] Javor Kalojanov and Philipp Slusallek. A parallel algorithm for construction of uniform grids. pages 23–28, 01 2009. 51, 55

[25] Tero Karras and Timo Aila. Fast parallel construction of high-quality bounding volume hierarchies. pages 89–99, 07 2013. 30, 34

[26] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, page 269–278, New York, NY, USA, 1986. Association for Computing Machinery. 20

[27] A. Keller, L. Fascione, Marcos Fajardo, I. Georgiev, P. Christensen, J. Hanika, Christian Eisenacher, and G. Nichols. The path tracing revolution in the movie industry. pages 1–7, 07 2015. 6

[28] A. Kensler. Tree rotations for improving bounding volume hierarchies. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 73–76, 2008. 33

[29] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998. 80

[30] S. Kumar and P. K. Singh. An overview of modern cache memory and performance analysis of replacement policies. In *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, pages 210–214, 2016. 13

[31] Eric Lafortune and Yves Willems. Bi-directional path tracing. *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics'*, 93, 01 1998. 54

[32] Samuli Laine. Restart trail for stackless bvh traversal. pages 107–111, 01 2010. 39

[33] Thomas Larsson. Strategies for bounding volume hierarchy updates for ray tracing of deformable models. 2003. 20

[34] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David P. Luebke, and Dinesh Manocha. Fast bvh construction on gpus. *Computer Graphics Forum*, 28:375 – 384, 04 2009. 25, 26

[35] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166, May 1990. 22, 45

[36] D. Meister and J. Bittner. Parallel locally-ordered clustering for bounding volume hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics*, 24(3):1345–1353, 2018. 29

[37] D. Meister and J. Bittner. Parallel reinsertion for bounding volume hierarchy optimization. *Computer Graphics Forum*, 37(2):463–473, 2018. 37

[38] Arsène Pérard-Gayot, Javor Kalojanov, and Philipp Slusallek. Gpu ray tracing using irregular grids. *Comput. Graph. Forum*, 36(2):477–486, May 2017. 6, 77, 80

[39] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010. 46, 55

[40] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. *Cohen-Or, Daniel; Slavik, Pavel: Eurographics 2007, Blackwell, 415-424 (2007)*, 26, 09 2007. 46, 47, 55

[41] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, July 2002. 47

[42] Niklas Röber, Ulrich Kaminski, and Maic Masuch. Ray acoustics using computer graphics technology. *Proceedings of the 10th International Conference on Digital Audio Effects, DAFx 2007*, 10 0002. 5

[43] Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination. In *Proceedings of High*

Performance Graphics, HPG '17, New York, NY, USA, 2017. Association for Computing Machinery. 5

[44] Brian Smits. Efficiency issues for ray tracing. *Journal of Graphics Tools*, 3:1–14, 1999. 39

[45] Martin Stich, Heiko Friedrich, and Andreas Dietrich. Spatial splits in bounding volume hierarchies. pages 7–13, 08 2009. 31

[46] Vasily Volkov. Understanding latency hiding on gpus. 2016. 13

[47] Ingo Wald. On fast construction of sah based bounding volume hierarchies. pages 33 – 40, 10 2007. 25, 75

[48] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26, 01 2007. 24, 75

[49] Ingo Wald and Vlastimil Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). *Symposium on Interactive Ray Tracing*, 0:61–69, 09 2006. 22, 46, 55, 75

[50] Ingo Wald, Thiago Ize, and Steven G. Parker. Fast, parallel, and asynchronous construction of bvhs for ray tracing animated scenes. *Computers & Graphics*, 32(1):3 – 13, 2008. 76

[51] Ingo Wald, Will Usher, N. Morrical, Laura M. Lediaev, and Valerio Pascucci. Rtx beyond ray tracing: Exploring the use of hardware ray tracing cores for tet-mesh point location. In *High Performance Graphics*, 2019. 5

[52] S. Woop, C. Benthin, I. Wald, G. Johnson, and Eric Tabellion. Exploiting local orientation similarity for efficient ray traversal of hair and fur. In *High Performance Graphics*, 2014. 76

[53] Chris Wyman and Adam Marrs. *Introduction to DirectX Raytracing: High-Quality and Real-Time Rendering with DXR and Other APIs*, pages 21–47. 01 2019. 5, 76