Karlsruhe Institute of Technology

Compressed Octrees and BVHs for Ray Tracing Acceleration Structures

Bachelor Thesis of

Marius Kilian

At the Department of Informatics
Institut für Visualisierung und Datenanalyse,
Lehrstuhl für Computergrafik

April 8, 2021

Reviewer:	Prof. Dr.-Ing. Corsten Dachsbacher
Second reviewer:	Prof. Dr. Hartmut Prautzsch
Advisor:	M. Sc. Addis Dittebrandt

Contents

Abstract

We compare the trade-off between reducing the memory footprint of ray tracing acceleration structures, by compressing the topology, against the increased computational resources and time needed to extract the resulting implicit data during ray traversal. We use an octree as an acceleration structure, as almost all information about its axis-aligned bounding boxes (AABBs) is implicitly encoded, allowing for a very efficient compression. We compress the topology into a bit field requiring only one bit per node. To counteract expensive traversal on a single large bit field, we divide these bit fields into chunks, that serve as subtrees of the original tree. Furthermore, we compress a BVH using this same technique to compare the same trade-off, as it is the most common ray tracing acceleration structure in use. Additionally, we quantize the axis-aligned bounding boxes (AABBs) of the BVH by storing their relative position to a larger AABB, and saving that with less data. The compression techniques succeed in reducing the size of the topology significantly, but have shortcomings due to the overall design, which are reflected in the increased intersection times of the compressed acceleration structures, as opposed to their uncompressed counterparts.

Zusammenfassung

Wir vergleichen ob ein reduzierter Speicherbedarf von Ray Tracing Beschleunigungsstrukturen durch Komprimierung der Topologie den dadurch erhöhten Rechenaufwand, der für die Extraktion der daraus resultierenden impliziten Daten benötigt wird, ausgleicht und die dadurch benötigte Rechenzeit aufwiegt. Wir benutzen einen Octree als Beschleunigungsstruktur, da nahzu alle relevanten Informationen implizit in der Topologie dargestellt werden können, und somit eine signifikante Komprimierung möglich ist. Wir komprimieren die Topologie in ein Bitfeld, in welchem jedes Bit einen Knoten des Baumes repräsentiert. Um der teuren Traversierung auf einem großen Bitfeld entgegenzuwirken, teilen wir es in Stücke (Chunks) auf, wo jedes Stück einen Teilbaum des original Baumes darstellt. Weiterhin komprimieren wir eine BVH auf die selbe Art und Weise, da sie häufige Verwendung in der Praxis findet, um den gleichen Vergleich machen zu können. Letztlich quantisieren wir die Hüllkörper (AABB) der BVH, indem wir ihre relative Position und Größe zu einem größeren, umfassenden Hüllkörper mit weniger Daten speichern. Die Komprimierung hat erfolgreich und bedeutend die Größe der Topologie reduziert. Allerdings wird die Rechenzeit der komprimierten Beschleunigungsstrukturen erhöht, da unsere Komprimierungen einige Einschränkungen mit sich bringen, welche sich als zu große Nachteile herausstellten.

1. Motivation

Photo-realistic rendering of complex scenes requires for the interaction of light to be accurately captured. Ray tracing is an algorithm predominantly used for this, to approximate photo-realistic lighting, reflections and shadow effects, among others. Millions of rays are shot out from the camera, with additional rays shot from the points where these rays intersect with the scene to the lighting sources. Each ray has an origin and a direction. In essence, it needs to be determined, whether the ray intersects with any primitive and where such an intersection takes place. To determine whether any ray hits a specific primitive, a so called primitive intersection test has to be performed. Because these tests are expensive on computational resources, so-called acceleration structures are used in order to minimize the number of primitive intersection tests that have to be performed per ray. These acceleration structures are traversed, and determine which primitives need to be tested for intersection, and which primitives can be skipped.

A common acceleration structure in ray tracing is the BVH [KK86]. This structure has its strengths with fast and precise traversal, meaning that the AABBs are very tightly wrapped around their content, thus resulting in less false positives, where the ray intersects with an AABB, but does not intersect with any of its content. However a BVH needs to save a lot of data explicitly, such as the topology and scene bounds, to do so. Because of this, its memory footprint is quite large. This means, that the CPU Cache is often be full while traversing the structure, and therefore needs to load data from memory, which takes a lot of CPU clock cycles [AK10].

This thesis explores avoiding wasting so many clock cycles when loading data from memory, by keeping the memory footprint of such an acceleration structure lower. This is first explored by using octrees as a ray tracing acceleration structure in chapter 4.1. Octrees are a space-partitioning data structure, as opposed to BVHs, which are an object-partitioning data structure. A space-partitioning data structure is one, where the partitioning takes place in regards to the space itself, rather than the contents of the space, such as the primitives. Of course, the primitives are considered, but only to provide information about whether or not to continue the partition, rather than how to continue partitioning. Octrees are not commonly used for ray tracing because of this. They have several disadvantages over an object-partitioning data structure, such as having to perform duplicate primitive intersection tests for some primitives, since the point in space where the partition is might directly go through such a primitive. However, an upside of their rigidity is that the bounding boxes at each level of the hierarchy are implicit in the topology, meaning that these bounding boxes do not have to be explicitly saved. Even though the octrees bounding

box is divided up a lot of times during construction, the only bounding box that needs to explicitly be saved are the world AABB. All other bounding boxes can be reconstructed during traversal.

Even though it is inherently disadvantageous to use an octree as an acceleration structure, due to its rigidity, this same rigidity allows for a very effective compression of the topology. We explore such a compression in chapter 4.2, and whether the severely smaller memory footprint of such a compressed octree makes up for its inherent disadvantages. If the resulting acceleration structure can keep up with other acceleration structures in the field in terms of rendering time, it would be a very useful alternative, since it requires so much less data. This could allow easier rendering of larger scenes, for example. Furthermore, we explore the compression of the topology of a BVH in chapter 5.2, which also results in less space being needed to store the data, therefore resulting in a smaller memory footprint.

The compression techniques tackle trying to store the topology of both octrees, as well as BVHs, with as little data as possible. This is done by storing the data in a breadth-first order, using a bit field for the node types (leaf node or interior node), where each bit represents one node. By doing this, the topology for both the octree, as well as the BVH, can be stored with a very low amount of bits per node. As seen later, each structure has some additional data that needs to be saved, with the BVH having more such data than the octree, but most of the data that is used in every step of a ray traversal can be compressed very tightly this way.

Both using an octree in general, as well as the compression techniques, have the trade-off of having to extract implicit data, which uses CPU cycles to compute. However, the idea is that the smaller memory footprint, as well as the fact that more data can be stored using less space in memory, results in more cache hits, and more data being readily in the CPU cache. In addition, when data is loaded from memory, more data can be loaded and stored in CPU cache at once, since the compressed data takes less space. The idea here is exploring whether the trade-off between loading less data into CPU cache, and loading data from memory less often makes up for the increased number of calculations that have to be performed during data extraction.

2. Background

2.1 Rendering Equation

A common goal of computer graphics is the photo-realistic rendering of digital scenes. Particularly lighting and shadow effects pose a problem, with several different solutions, each with varying strengths and weaknesses. The essence of the problem is the so-called rendering equation [Kaj86]

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i, \omega_o) L_i(x, \omega_i)(\omega_i \cdot \mathbf{n}) d\omega_i$$

where L_o is the total light at position x radiated towards direction ω_o. Often, more parameters are taken into consideration, such as the time, t, or the wavelength of the light, λ. The first term on the right side, L_e is the light emitted from point x with direction ω_o. Essentially, if the point x is a light source, the energy of that light source is determined by this term. The second term represents all light that hits point x and is reflected towards direction ω_o. This includes all light from the incoming directions ω_i, integrated over the unit hemisphere with center x, Ω. Here, L_i is the amount of light coming towards point x from direction ω_i. This light is calculated by finding more points, y, that emit or reflect light towards point x, through ray tracing. In other words, we can describe this as $L_i(x, \omega_i) = L_o(\text{raytrace}(x, -\omega_i))$. The term f_r is the bidirectional reflectance distrubance function (BRDF), and represents the proportion of light reflected from ω_i to ω_o at position x. $(\omega_i \cdot \mathbf{n})$ weakens the term in regards to the angle of incidence. A flatter angle means, that the same amount of light is distributed over a larger surface, meaning there is less energy exactly at point x.

2.2 Monte Carlo Integration

The first term, L_e, is easy to determine since it only depends on local information at point x. The second term, however, is an integral over any possible incoming direction ω_i in the unit hemisphere Ω. This integral is very complex and also recursive, due to the previously mentioned relationship, $L_i(x, \omega_i) = L_o(\text{raytrace}(x, -\omega_i))$. Hence, this integral has to be approximated. We do this numerically, using Monte Carlo integration to approximate the integral.

We do this by reducing the Integral to a finite Sum, using the Monte Carlo method of generating random directions ω_i from a probability distribution, $p(\omega_i)$, over a defined domain:

$$L_o(x, \omega_o) \approx L_e(x, \omega_o) + \frac{1}{N} \sum_N \frac{1}{p(\omega_i)} f_r(x, \omega_i, \omega_o) L_i(x, \omega_i)(\omega_i \cdot \mathbf{n})$$

This finite sum therefore has a finite number of directions ω_i to iterate over, allowing us to find an approximated value for the integral, therefore solving L_o. For each direction ω_i generated through the Monte Carlo method, a ray is shot with origin x and direction ω_i. To find the amount of light coming from a single such direction ω_i, we need to know what point, y_i, a ray shot in that direction intersects with, in order to determine how much light comes from that point. This equates to the aforementioned relationship $L_i(x, \omega_i) = L_o(\text{raytrace}(x, -\omega_i))$. We therefore apply the same rendering equation and Monte Carlo integration to each such point y_i.

This recursive way of solving the equation is called *distributed ray tracing*, which follows every $-\omega_i$ from every found point, usually up to a certain recursive depth. Distributed ray tracing has, as demonstrated, exponential growth, which is why, alternatively, some rendering algorithms use so-called *path tracing*, which only follows a single path, or direction $-\omega_i$, for each point. The single path determined at each point is the one with the highest probability according to the probability distribution p. Both these so-called integrator functions are not further explained in this thesis.

2.3 Acceleration Structures

As each ray has to determine its closest point of intersection to continue the traversal, a lot of primitives need to be tested for intersection against each ray. Not only is this usually a very large amount of primitives, which would make testing the ray against all of them very impracticable, but these intersection tests are also a very expensive operation. To reduce the amount of intersection tests each ray has to perform, we use so-called acceleration structures. These acceleration structures typically have some sort of hierarchy, starting with a single node, and dividing the scene more finely in the lower levels. A common acceleration structure in ray tracing is the BVH [KK86]. Building it from the bottom up, a small number of primitives are grouped together inside an AABB, which we call a leaf node. The next layer up groups two of these leaf nodes together into a larger AABB, which becomes the parent node of the leaf nodes. This is done until only one AABB remains, which is the root node of the hierarchy. These acceleration structures are traversed top-down, and can potentially quickly determine large regions that the ray misses in its entirety, therefore allowing many primitive intersection tests to be skipped. On top of this, the intersection tests against the structure itself are typically a lot cheaper than the intersection tests against primitives.

Ultimately, acceleration structures allow for a faster calculation of the Monte Carlo integration, by shortening the execution time of intersection queries considerably.

3. Related Work

This chapter discusses existing work, that relates to the topic of this thesis. We draw parallels to studies on general topology, as well as specific compression on BVHs. Furthermore, we discuss voxel-compression, as it also bases on octrees.

3.1 Topology and BVH Compression

[Jac89] compresses data structures by sorting them breadth-first and compressing them to individual bits. An efficient way of traversing the structure is then presented through so-called *rank* and *select* functions. rank(n) counts the amount of `1` bits that occurred up to the n^{th} bit, while select(n) finds the location of the n^{th} `1` bit. Through these functions, and the breadth-first structure, the location of the child and parent nodes can be easily determined. Even though a significant compression is achieved, these functions can become very expensive when they have to be applied to a very long such bit field, therefore making traversal slower, the deeper into the structure we are. This thesis uses this compression technique, but divide the bit field into chunks, and only apply these operations to the local chunk bit fields.

A compression technique of BVHs is presented in [MW06]. The nodes they compress consist of an AABB with six 32-bit values, and 2 further 32-bit pointers, totalling to 32 bytes of data per node. The paper acknowledges the large amount of data stored for the AABB of each node, and reduce the amount of data needed per node by "63%-75%". They achieve this by reducing the six 32-bit values of an AABB to six 4-bit or six 8-bit values, depending on the use-case, therefore reducing the total data needed per node to 8 bytes or 12 bytes respectively. This is done by quantizing the AABB of each node in relation to its parent node. This thesis uses this same approach of quantizing the AABBs as part of compressing a BVH, but we further use topology compression to also remove the pointers.

While GPU-based, [YKL17] compresses BVHs to achieve significantly faster ray traversal. This is done through the quantization of primitives, and by using wide BVHs, meaning that each interior node has more child nodes. In this case, each interior node has 8 child nodes. Because of this, the scene is divided more quickly in the hierarchy, and the resulting tree is less deep. Following this, the traversal stack can be significantly more compact, which is very important for GPUs. In addition, the wide BVHs allow one parent node, and therefore one AABB to be used as the quantization frame of many child nodes. While we do not use a wide BVH, we explore this same concept in a different way, where we quantize all nodes of a chunk of nodes relative to the same quantization frame, further compressing the data.

3.2 Sparse Voxel Directed Acyclic Graphs (SVDAGs)

A new acceleration structure is presented in [KSA13], called a *sparse voxel DAG*, where common subtrees in the hierarchy are eliminated by forming a directed acyclic graph. They are, however, limited to a single bit of information per leaf node, which only specficies whether the leaf node is empty or not. While the compression of this paper is very efficient, this limitation of a single bit of information per leaf node means it is unsuited for this thesis, since each leaf node in our acceleration structures is unique.

[VMG17] expands on [KSA13] by allowing the SVDAG to be aware of symmetrical subtrees. This means, that the subtrees are not identical, but are mirrored in relation to one another. They can therefore also be compressed, further reducing the total amount of data required to store the acceleration structure.

The paper discussed in [KSA13] is limited to a single bit of information (or geometry) per leaf node (voxel). [DKB+16] compresses arbitrary data, such as colors, and presents a new mapping scheme, allowing the aforementioned sparse voxel DAGs to be used with arbitrary information per voxel. It achieves drastic compression of the scene and improves the time a ray takes to traverse a scene when finding a point of intersection.

4. Compressed Octrees for Ray Tracing

An octree is a data structure which can be used to partition three-dimensional space. As such, it can be used as an acceleration structure in the field of ray tracing. An octree partitions a three-dimensional space by dividing an AABB into 8 evenly sized octants, by splitting it at its center for each of the 3 dimensions. This process is recursively repeated, until certain conditions are met. Therefore, octrees are a completely rigid structure and the only explicit information that needs to be saved are the world AABB, the topology, and information about what primitives are contained in which leaf nodes. The leaf node information can be saved using a pointer to a primitive list. All other AABBs are implicit in the structure of the octree, and can be implicitly reconstructed. This is also the most significant advantage of using an octree as opposed to, for example, a BVH or a k-dimensional tree (k-d tree), which need to explicitly save the AABBs and split-planes for each node, respectively. The topology is what we compress, ultimately making the explicitly saved information very small, hence reducing the memory footprint, especially when compared to its BVH and k-d tree counterparts. Saving less data, therefore resulting in smaller memory usage for the acceleration structure, hopefully results in more of the structure being saved in CPU cache, ultimately leading to faster traversal due to more cache hits.

There is also a significant disadvantage to octrees that should be kept in mind; As the subdivision process is rigid, primitives are not necessarily contained in only a single leaf node, which can lead to multiple disadvantageous situations. Firstly, this can lead to multiple intersection tests between a ray and the aforementioned primitives, which results in unnecessary intersection tests, lengthening the time the ray takes to traverse. As intersection tests are expensive, the goal is to have to do as few of them as possible. This may therefore counter the effects of any compression that we perform. Secondly, when a ray hits a primitive within a leaf node, it is not guaranteed that the point of intersection is also contained in that leaf node. In such a case, the ray would have to further traverse the octree as other Primitives might still get hit closer to the origin of the ray. Both of these scenarios ultimately result in the ray traversal taking longer. We explore whether the very significant compression that can be performed on an octree due to its rigidity makes up for the inherent disadvantages and problems that this same rigidity poses.

Now that the basics of an octree are established, we first take a look at an implementation of such an octree in chapter 4.1. We refer to this as the *Basic Octree*, and explore how its data can be stored in chapter 4.1.1, the different ways we can construct such an octree in chapter 4.1.2, and how a ray traverses it in chapter 4.1.3. before comparing a few values

for several parameters that are used for construction, which result in a slightly different structure, in chapter 4.1.4.

After the basic octree is established, we apply compression techniques and in chapter 4.2. Due to the type of each node being stored in a single bit, we refer to this octree as the *1-bit Octree*. In chapter 4.2.1 we look at how its data is stored, and discuss some problems, as well as a solution to these problems in chapter 4.2.2. Chapter 4.2.3 discusses ray traversal of the 1-bit octree, before comparing the performance differences of a few values for several parameters of the compressed data structure in order to determine the best such values to choose.

4.1 Basic Octree

The basic octree is constructed with the rules that are described in chapters 4.1.1 and 4.1.2, but does not yet implement the compression techniques that this thesis explores. It is meant as an entry point to how exactly an octree might be constructed, and as a basis for later evaluation.

4.1.1 Data Layout

```
class Octree:
    Bounds world_bounds
    List<uint32> nodes_list
    List<uint32> sizes_list
    List<Primitive> primitives_list
```

To construct a basic, functioning octree, let us first take a look at how the data is stored. As previously established in chapter 4, the data that needs to be explicitly stored are the world AABB, the topology, and the information about what primitives are contained in which leaf nodes. The world AABB are six 32-bit floating point values, one `min` and one `max` value for each of the 3 axis, x, y and z. The information about which primitives are contained within which leaf node is spread out in 2 lists. One list of primitives, called `primitives_list`, includes all primitives from all leaf nodes, with primitives from the same leaf node always having consecutive indices. Another list, with the same length as the list of nodes, saves the integer value of the amount of primitives within each leaf node. This list is called `sizes_list`. The topology is a list called `nodes_list`, containing all nodes, with each node having 32 bits of information, divided up as follows:

Figure 4.1: Data Layout of the Nodes List in the Basic Octree

The lowest bit signifies the type of node, either being `0` for a leaf node or `1` for an interior node. The remaining 31 bits are an offset. In the case of an interior node, they point to the location of the beginning of its child nodes in `nodes_list`. The offset of a leaf node points to a location inside `primitives_list`. The amount of primitives within any given leaf node with index n inside the `nodes_list` is stored also at index n inside the `sizes_list`. The reason why the `nodes_list` and `sizes_list` are split into 2 is, that the entries of `nodes_list` need to be accessed a lot more frequently, when traversing the octree, than the entries of `sizes_list`.

To know which primitives are contained within each node, the AABB of each node should be calculated and saved for building the octree only. They can later be discarded, as they are reconstructed when needed.

4.1.2 Construction

An important decision to make, when considering the structure of an octree, is the decision of when a node should be an interior node, therefore being further subdivided, or a leaf node, containing a list of primitives. Since all primitives of any given leaf node need to be checked for ray intersection, whenever the ray intersects with the AABB of said leaf node, it is important to keep in mind the following two factors, in order to reduce the number of primitive intersection tests.

Firstly, to a certain extent, a leaf node should contain as few primitives as possible. This is a direct consequence of the previously mentioned intersection policy. Secondly, any primitive should ideally be contained in as few leaf nodes as possible, since it would otherwise potentially need to be tested for intersection several times for a given ray. This follows from the rigidity of the octree, that might subdivide a node directly in a space where Primitives, as described in the Introduction to this chapter. There are methods of preventing these repeated intersection tests for the same primitive, such as mailboxing [AW87], but they are not discussed within this thesis. A primitive contained in more than one leaf node is henceforth referred to as a duplicate primitive.

Following these rules, we can create 2 parameters, one for each rule. The first determines the minimum number of primitives per interior node. In other words, if any node contains less primitives than this primitive threshold parameter, it instantly becomes a leaf node.

The other parameter is used to limit the amount of duplicate primitives that making the current node an interior node would create. In other words, if making the current node an interior node would make each one of its primitives be contained in exactly 2 of its child nodes, then making it an interior node would mean we essentially double the amount of primitives. Of course that amount could further increase, if the created child nodes also become interior nodes. Referring to the total amount of primitives in the potential child nodes (therefore counting any as often as they occur) as `num_dupl_prims`, and the amount of primitives in the base node as `num_prims`, we can create a parameter `mult_thresh`, so that the following rule has to uphold, in order for a node to become an interior node: `num_dupl_prims` $<$ `mult_thresh` $*$ `num_prims`. Algorithm 1 summarizes these rules.

Algorithm 1 Determine whether or not to make a Node a Leaf Node

```
1:  procedure MakeLeafNode(node)
2:      if node.numPrims < primThresh then
3:          return true
4:      end if
5:      numDuplPrims := 0
6:      for all octant in node.boundingBox do
7:          for all Primitive prim in node.primitives do
8:              if octant contains prim then
9:                  numDuplPrims := numDuplPrims + 1
10:             end if
11:         end for
12:     end for
13:     if numDuplPrims > multThresh * numPrims then
14:         return true
15:     end if
16:     return false
17: end procedure
```

In chapter 4.1.4 we compare how several values for these parameters change the performance of the octree.

4.1.2.1 Alternative Algorithms for Deciding the Type of a Node

Two alternative algorithms for deciding the type of node were also tested, before settling on the simple duplicate primitive threshold that was described in the previous section. Both of these alternative deciders always included the aforementioned primitive threshold.

The first decider was a simple maximum depth of the octree. When a maximum depth was reached at any path, that node would become a leaf node. This worked decently well

when working with a specific scene. The ideal maximum depth for that scene had to first be tested manually, however, and did not carry through to other scenes.

The second decider, and the more interesting one, tried to find clusters of many tightly packed primitives that the octree would have troubles subdividing due to its rigidity. To do this for a given node (we refer to this node as the *base node*), the node would be subdivided into 8 octants, as if it were an interior node. The number of primitives in each of those octants would then be counted, and if they exceeded a certain threshold, relative to the number of primitives in the base node, that octant would be considered for the cluster. For example, if this threshold were 80%, then any octant that contained 80% of the primitives in the base node would be considered for the cluster. Then we would make a list of all primitives that are contained in *all* of these octants that were considered for the cluster. This list of primitives would be a dense cluster of primitives. Lastly, we would find the AABB of this cluster, and compare the volume to the AABB of the base node. If the volume exceeded a certain threshold, relative to the AABB of the base node, then the node would become a leaf node.

However, this ran into several problems. The biggest such problem was, that if a single very large primitive (such as a flat floor) was contained in the cluster, the size of the AABB of the cluster would always take up the whole AABB of the base node, even if the rest of the cluster was very small relative to the base node. This would result in leaf nodes that, firstly, still contained many primitives and, secondly, have a cluster, but one that is in reality still very small compared to the AABB of said leaf node. Hence, many rays would intersect with the AABB of the leaf node, test for intersection against the large amount of primitives, but not intersect with any of them, since they only made up a very small volume of the AABB of the leaf node.

Ideally, this algorithm would find a cluster of primitives within a node, then determine how much space in the node this cluster took up. If it took up most of the AABB of the base node, then it would be unwise to divide the cluster further, since most of the primitives would be in multiple octants. This ultimately led to the idea of having a simple duplicate primitive multiplication threshold, which is easier to manage and test for different values, as well as more intuitive.

4.1.3 Ray Traversal

Traversing this Octree requires a few rules and steps. When a ray intersects with an interior node, the order of traversal of the child nodes of that node needs to be determined. Once it has been determined, those child nodes are fully traversed in that order. Here, *fully* means that if one of these child nodes is an interior node as well, its children are first traversed before continuing with the rest of the original child nodes. The following algorithm helps clarify this.

This, however, still leaves two unknowns, namely how the traversal order is determined and how a leaf node is handled.

For the traversal, order, an important characteristic of Octrees comes into play; If a ray intersects with an interior node, it also intersects in the same place with one child of that node. This is due to the fact, that every part of the AABB of the node is covered by the AABBs of its child nodes, when put together. Ultimately, this also implies that that specific child node is the first node the ray should traverse. We call the current node being traversed the *base node*, while the specific child node that is implied by the previously mentioned characteristic is referred to as the *entry child node*. Any other child nodes are referred to simply as *child nodes*. Assuming that we know the point of intersection of the ray and the base node, we can determine the entry child node by comparing this point of

Algorithm 2 Octree Ray Traversal

```
1: procedure TRAVERSE(ray, node)
2:     if ISLEAFNODE(node) then
3:         HANDLELEAFNODE(ray, node)
4:     else
5:         traversal := FINDTRAVERSALORDER(ray, node)
6:         for all childNode in traversal do
7:             if childNode.t < ray.t then
8:                 TRAVERSE(ray, childNode)
9:             end if
10:        end for
11:    end if
12: end procedure
```

intersection with the center point of the base node. Comparing which has a bigger x, y and z value determines the Octant of the entry child node. To determine which other child nodes are traversed, we can simply intersect the ray with all three axis-aligned planes that halve the AABB of the base node [AGL89]. The distance from the rays origin to these three plane intersection points, together with the information whether these three plane intersection points are inside the AABB of the base node determines the full traversal order, by sorting the intersection points by the t parameter. Figure 4.2 demonstrates this in 2D-Space, with a Quadtree. It should be noted, that the hereby calculated points of intersection, or rather, the `t` parameters, should then also be saved, as they are needed to further traverse the child nodes.

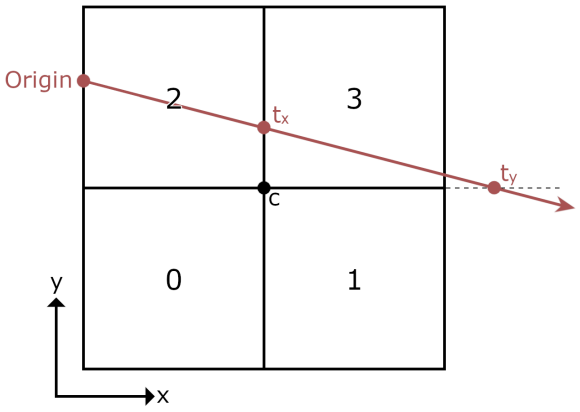


Figure 4.2: Determining the traversal order of child nodes in a quadtree

In this figure, the origin of the ray starts at octant 2. We can test this by simply comparing the coordinates of the origin to the center point c of the AABB. In the octree, we have established each octant as having an index 0-7, or $000_2 - 111_2$, interpreted as $b_z b_y b_x$, with b_i being the bit for axis i. If b_i is `0`, we take the octant smaller than the center point for that axis, if it is `1`, we take the octant larger than the center point for that axis. With the quadtree of figure 4.2, this base principle remains the same, except that we have indices 0-3, or $00_2 - 11_2$, interpreted as $b_y b_x$. This is important, because we can easily determine the traversal order by checking the order in which the ray passes the center point for each axis. As can be seen in the example, the ray first passes the center point along the x-value of the center point at t_x, before passing the y-value of the center point at t_y. Knowing that the starting quadrant is quadrant 2, or quadrant 10_2, we can find the next quadrant

by swapping the bit of the axis where the center point is crossed. Since the x-value of the center point is crossed first, we flip b_x, so $10_2 = 2_{10}$ becomes $11_2 = 3_{10}$. Hence, we know that quadrant 3 is the next one. Lastly, we can see that the ray passes the y-value of the center point at t_y. Since t_y lies outside of the original AABB, this ray does not intersect with any more quadrants inside the original AABB.

We now consider this in more general terms. To start off the traversal, the ray is tested for intersection with the world AABB. On a hit, the point of intersection should be saved. Since the origin o and direction d of the ray are given, it is enough to save a Parameter t, with the point p being calculated as $p = o + t * d$. Using this, the root node can then recursively be traversed using the above method, with every step of the recursion receiving the corresponding t parameter, which is either, in the case of the entry child node, the same as the t parameter of the parent node, or, for other child nodes, calculated while finding the traversal order, as seen in Figure 4.3. Here, t_i is the t parameter for quadrant i, and t_{i-j} is the t parameter for quadrant j inside quadrant i. The quadrants can then be traversed in ascending t parameter order.

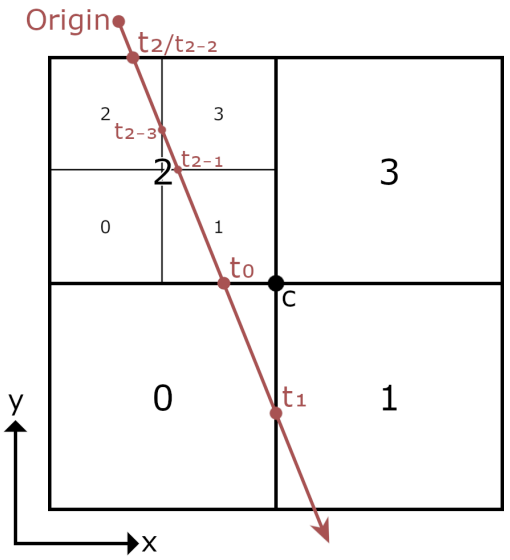


Figure 4.3: Traversing a quadtree in ascending t parameter order

Lastly, there is the case of a node being a leaf node. In such a case, all primitives in that leaf node should be tested for intersection, since there is no information about which of these primitives is closer or further. As discussed in the beginning of chapter 4, a hit between the ray and a primitive of a certain leaf node does not guarantee the point of intersection to be inside said leaf node. Therefore, after intersecting all primitives of that leaf node, when a hit has been found, a simple test should be performed, that checks whether the found point of intersection is within that leaf node. If it is, that point is definitely the closest primitive, and the traversal can stop. If it is not the traversal should continue. The point that was hit is definitely in another leaf node, that is later traversed, so even if that point were to be the closest point of intersection to the ray, it is found again at a later time during traversal. However, it is quite possible, that there are other primitives between the leaf node and the point of intersection, which is why the traversal needs to properly continue.

Since the AABB of each node is only implicitly saved, we need to keep a stack of all traversed AABBs, therefore allowing access to the AABB of the parent node. However, this also quickly increases the memory footprint of the octree, since each bounding box is 24 bytes (192 bits) big.

4.1.4 Comparison of Construction Parameters

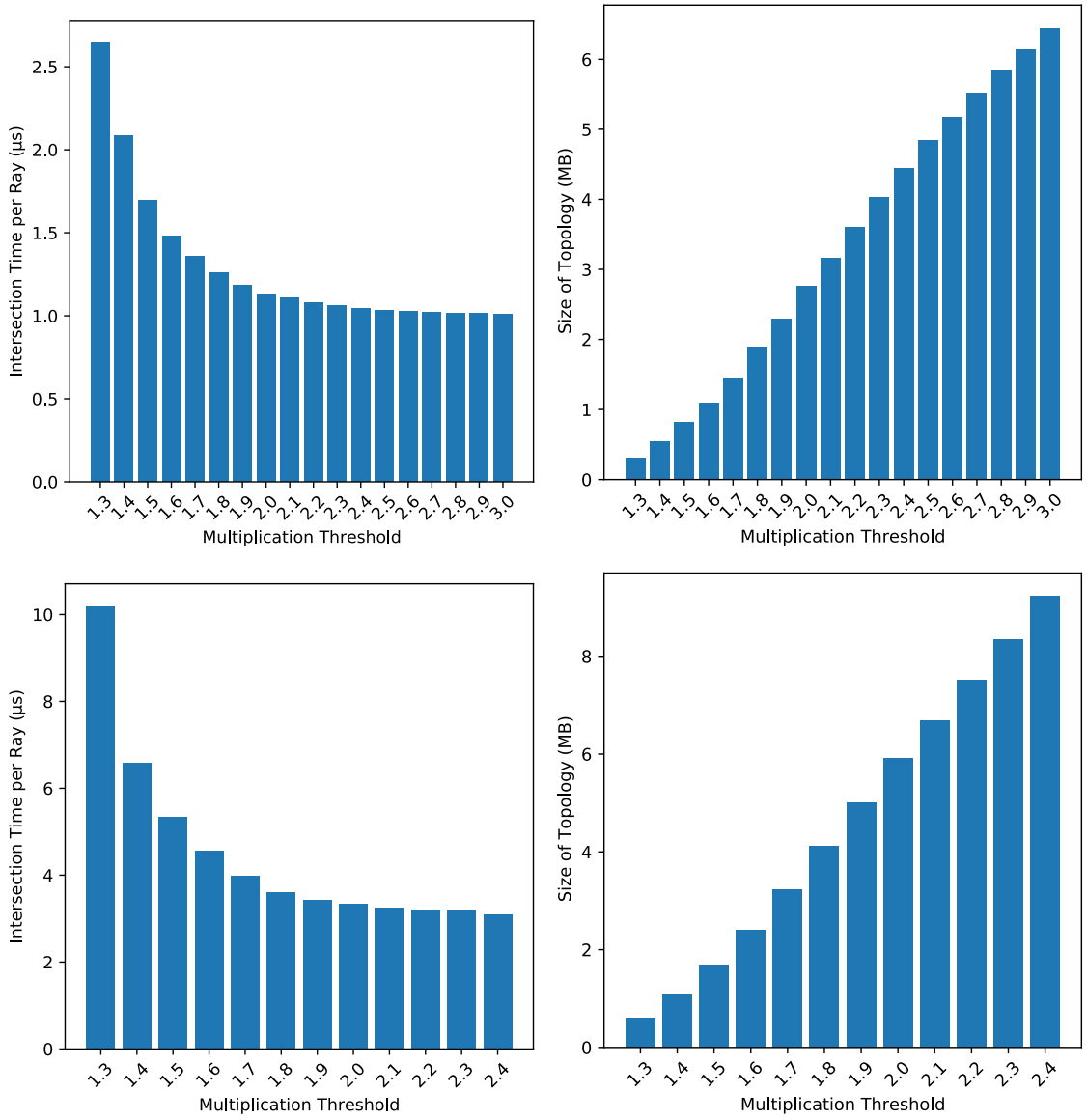


Figure 4.4: Intersection time and topology size for different multiplication thresholds
Scenes: *Crown* (top), *Measure-one* (bottom)

As previously discussed, there are two parameters that can be adjusted for how our octree is constructed, which both affect the structure and depth of the octree. The goal is, to compare different values in order to find the ones where ray traversal is fast and the size of the topology is reasonably small. To do this, we use two scenes. One is called *Crown* and is a relatively small scene with many areas densely packed with primitives, while *Measure-one* is a large scene of a room with many small details. The depicted sizes are the size of the topology, which entails all data that is needed for every step of the traversal. This is further described in chapter 6.1, along with more details about the testing environment, including the full renders of the used scenes.

Figure 4.4 shows a comparison of the multiplication threshold parameter that was established in chapter 4.1.1. The used primitive threshold for these comparisons is 32, as that is one that established itself as a decent value during testing. The reason for the improved intersection times as the multiplication threshold increases is, that several nodes that were

leaf nodes at a lower multiplication threshold become interior nodes at a higher multiplication threshold. This leads to more detail in the acceleration structure, therefore skipping more unnecessary primitive intersection tests than the octree with a lower multiplication value would. The size of the topology goes up for the same reason of more detail in the octree, which results in a larger octree, and more duplicate primitives. The graphs show a clear improvement in intersection time, as the multiplication threshold is increased from 1.3 to 2. For values higher than 2, the intersection time becomes only marginally smaller for the *Crown* scene, and almost stagnates for the *Measure-one* scene. As the size of the topology increases linearly for both scenes, for multiplication thresholds between 1.3 and 2.4, the ideal multiplication threshold is a value around 2. Therefore, we henceforth use a value of 2.0 for the multiplication threshold for the construction of octrees.

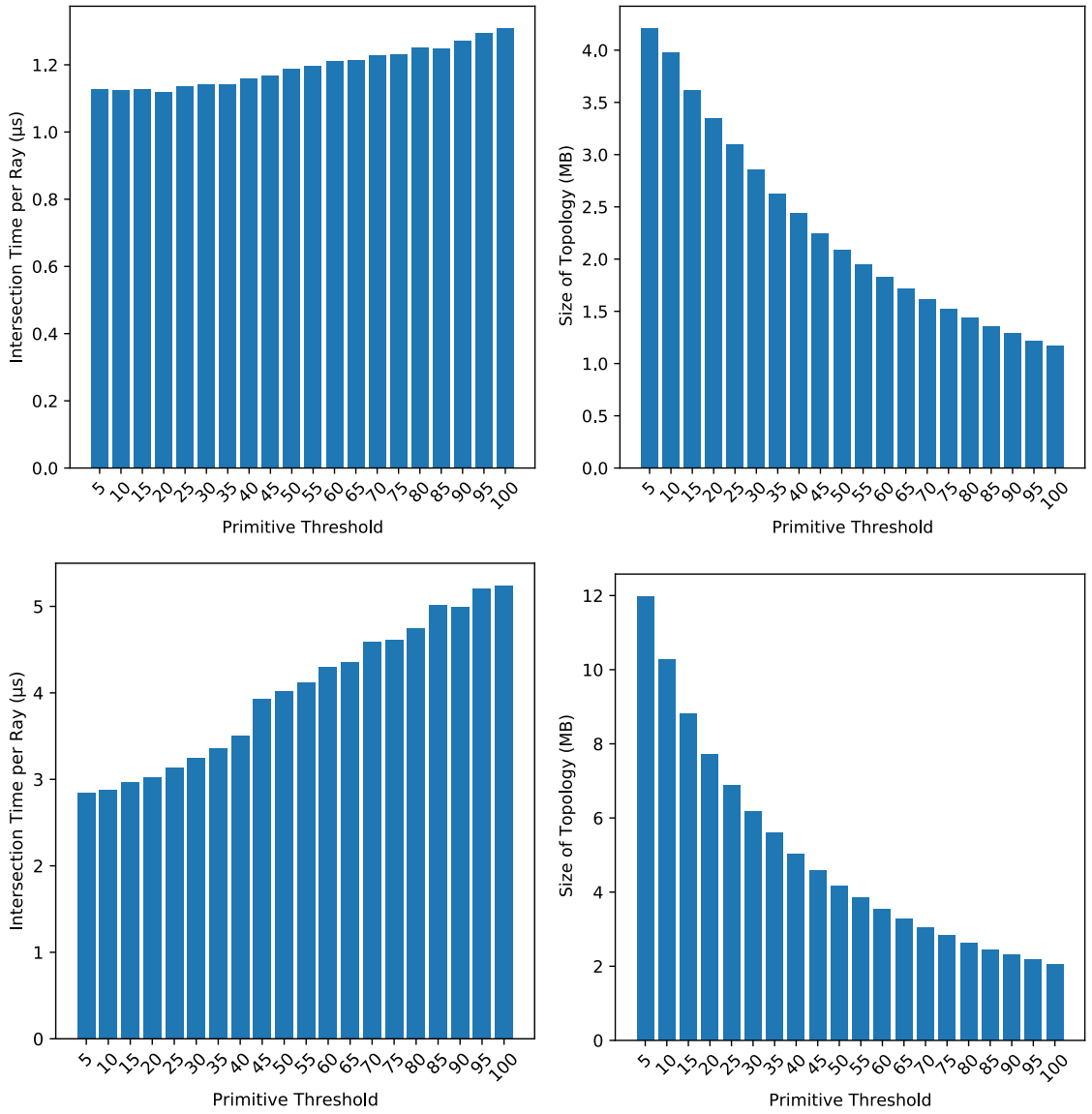


Figure 4.5: Intersection time and topology size for different primitive thresholds
Scenes: *Crown* (top), *Measure-one* (bottom)

The second parameter we compare is the primitive threshold. During the construction of the octree, any node with less than or equal the amount of nodes of the primitive threshold becomes a leaf node. Figure 4.5 shows how different values for the primitive threshold affect the intersection time per ray, as well as the size of the topology. It should

be noted, that there might be slight variations for the intersection time at each primitive threshold, as the results base on a single test for each value, due to time constraints.

As figure 4.5 shows, the intersection time goes up as the primitive threshold goes up. This is due to the fact, that nodes become leaf nodes sooner in construction, as the amount of primitives they contain is more quickly below the primitive threshold. This results in a more coarse structure of the octree, as the primitive threshold increases, resulting in more unnecessary primitive intersection tests. The size of the topology goes down, however, for the same reasons. The graphs show a very significant decrease in size of topology, as we increase the primitive threshold from 5 to 35, at which point this decrease starts mitigating. The intersection time per ray starts increasing in a meaningful way at values higher than 20, for both scenes. Because our main objective is faster execution times, we choose the value 20 for the primitive threshold, as it still reduces the size of the topology by around 19% for the *Crown* scene, and around 34% for the *Measure-one* scene, as compared to a lower value of 5.

4.2 1-Bit Octree

With the term 1-bit Octree, we describe an octree that gets close to storing all information for the topology in only 1 bit per node. It is theoretically possible to store the full topology of the octree in 1 bit per node, as is also discussed in this chapter, but it has severe disadvantages. Hence, we create a structure where some additional information is stored besides the otherwise 1 bit per node. The structure described in section 4.2.1 was never implemented, but acts as a stepping stone for the next section, which describes the used implementation.

4.2.1 Data Layout

```
class Octree_1Bit:
    Bounds world_bounds
    List<Primitive> primitives_list
    List<uint32> offset_list
    List<uint64> bitfield
```

As discussed in chapter 4.1, the data that needs to be explicitly saved are the world AABB, as six 32-bit floating point values, the information about which primitives are contained in which leaf nodes, and lastly, the topology. The topology is what we compress. Currently, using the structure established in figure 4.1, the node list of a basic Octree looks something like figure 4.6.

Figure 4.6: Data Layout of the Nodes List in the Basic Octree

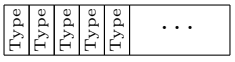
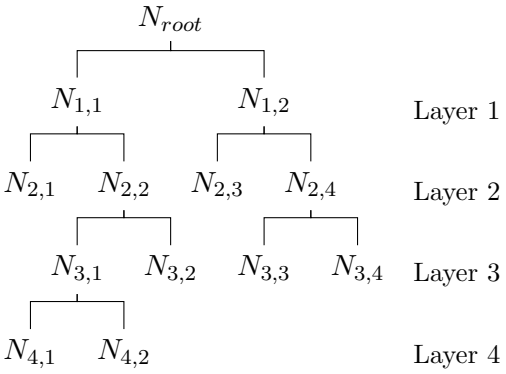


Figure 4.7: Data Layout of the Nodes List in the 1-Bit Octree

The basic octree discussed in chapter 4.1 only followed a few rules for constructing such a node list, such as each set of 8 sibling nodes having to be consecutive in the list, so that only one child pointer needs to be stored per interior node. However, to implement the compression from [Jac89], we need to reconstruct this list in breadth-first order. We compress this structure by removing the per-node offset, creating a structure shown in figure 4.7. Each individual bit in each entry of the node list is a single node. As such, a list of type `uint64`, for example, contains up to 64 nodes per list entry. This can be done, as seen later, because every non-leaf node has the same amount of child nodes. To demonstrate this, consider the binary tree in figure 4.8 for easier visualization. The same principle can be applied to an Octree.

Looking at the graph of figure 4.8, N_{root} is the root node, and every other node is identified by its unique layer number and position within that layer, like such: $N_{l,p}$ with l being the layer, and p the position within that layer (form left to right). By inserting the nodes into a list layer by layer (breadth-first), the list looks like the bit field in figure 4.8. It should be noted, that N_{root} is not in this bit field, as it is not necessary to be explicitly saved. If the list is empty, we can imply that the root node is a leaf node. Otherwise it is an interior



Layer 1		Layer 2				Layer 3				Layer 4	
$N_{1,1}$	$N_{1,2}$	$N_{2,1}$	$N_{2,2}$	$N_{2,3}$	$N_{2,4}$	$N_{3,1}$	$N_{3,2}$	$N_{3,3}$	$N_{3,4}$	$N_{4,1}$	$N_{4,2}$

Figure 4.8: Data Layout of the Nodes in a 1-Bit Binary tree

node. As can be seen from the tree, Nodes $N_{2,1}$, $N_{2,3}$, $N_{3,2}$, $N_{3,3}$, $N_{3,4}$, $N_{4,1}$ and $N_{4,2}$ are leaf nodes, while all others are interior nodes. Using the same system as in chapter 4.1.1, we assign `0` to the leaf nodes and `1` to the interior nodes, making the resulting bit field look like this:

0	1	2	3	4	5	6	7	8	9	10	11
1	1	0	1	0	1	1	0	0	0	0	0

Figure 4.9: Example Data Layout of the Binary Tree from figure 4.8

To show that this bit field holds all necessary information for the tree, we reconstruct the tree from this bit field. For this, we need the previously mentioned rule of a binary tree, which is that every interior node has exactly 2 children. In an octree, the same principle can be applied, since every interior node of an octree has exactly 8 children. Looking at the list, we know that N_{root} is an interior node, since the list is not empty. Therefore, the first two entries are the nodes of layer 1, or, in other words, the child nodes of N_{root}. We know that layer 1 contains exactly 2 nodes, since the implicit layer 0 has only one node, N_{root}.

We can use the *rank* function from [Jac89] to count the amount of `1` bits, or interior nodes, before a certain index. Therefore, rank(2) gives us the amount of `1` bits in layer 1, which is 2. This means, that the size of layer 2 is 4 bits, since there are 2 layer 1 interior nodes and 2 child nodes per layer 1 interior node. Therefore, the bits at position 2-5 are nodes from layer 2. We also know which pair of nodes belongs to which layer 1 node, since they are ordered. The first pair of layer 2 nodes (indices 2-3) belong to the first interior node (therefore the first `1`-bit) of layer 1, while the second pair of layer 2 nodes (indices 4-5) belongs to the second interior node of layer 2.

We can describe the starting position of the child nodes of an interior node with index i as

$$\text{child}(i) = 2(\text{rank}(i) + 1)$$

The term rank is incremented by 1, since the node at position i is an interior node, but the rank function only counts the preceding bits. Using the rank function in this way, we can continue these same steps to figure out which nodes belong to which layer, as well as the parent node to each of the nodes of the lower layers, therefore reconstructing the original tree from figure 4.8.

As previously mentioned, we can use this same principle with an octree, since every interior node has exactly 8 children. If the bit field is not empty, we know that root node is an interior node, making the first 8 nodes of the bit field the layer 1 nodes. Therefore, assuming n amount of `1`-bits in a layer, we know that the next layer has the size $8n$. The child function also has to be slightly adjusted to $\text{child}(i) = 8(\text{rank}(i) + 1)$.

One question, that has so far been left unanswered, is, how to determine which primitives belong to which leaf nodes. For this, two lists need to be saved. One list of primitives, called `primitives_list`, includes all primitives from all leaf nodes, with primitives from the same leaf node always having consecutive indices. This is essentially the same list as the `primitives_list` from chapter 4.1.1. A separate list, called `offset_list` determines where each primitives leaf nodes start, and where they end. For this, we have to first determine an order for the leaf nodes. For this, we use the exact order in which they are arranged within the previously established structure (see figure 4.9). This means, that the first `0` bit is the first leaf node, the second `0` bit is the second leaf node and so on. Ultimately, the n^{th} `0` bit is always the n^{th} leaf node. We can find the amount of `0` bits that precede a leaf node at index i as $i - rank(i)$.

Using this order, each element in `offset_list` determines the starting offset for the primitives of that leaf node in `primitives_list`. Therefore, the consecutive element in `offset_list`, which determines the starting offset for the next leaf node, simultaneously determines the ending offset for the current leaf node. For clarification, consider the example in figure 4.10.

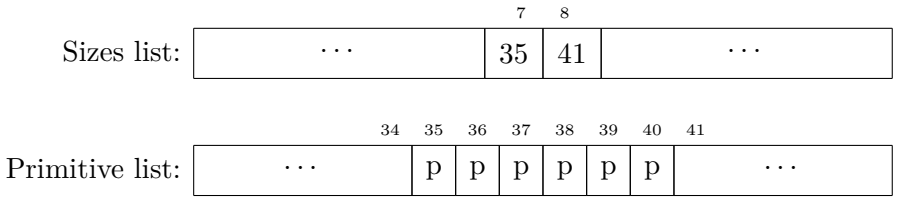


Figure 4.10: Example of how the primitives of a leaf node can be determined

In this example, we examine the 7^{th} `0` bit, therefore the 7^{th} leaf node. Looking at `offset_list`, we find, that its primitives in `primitives_list` start at index 35. By looking at the next entry in `offset_list`, we see that the next, therefore the 8^{th} node, has its primitives start at index 41. Therefore, the primitives of the 7^{th} leaf node are the primitives shown, from index 35-40, marked as p.

The `offset_list` and `primitives_list` have the same size as the basic octree, and are not compressed. This is fine, however, because they are only accessed for leaf nodes, and not for every node during traversal. Most nodes that are accessed during traversal are interior nodes, which is why these two lists are relatively rarely accessed. Therefore, they do not contribute a lot to the memory bandwidth, even if they are not more compressed than in the basic octree.

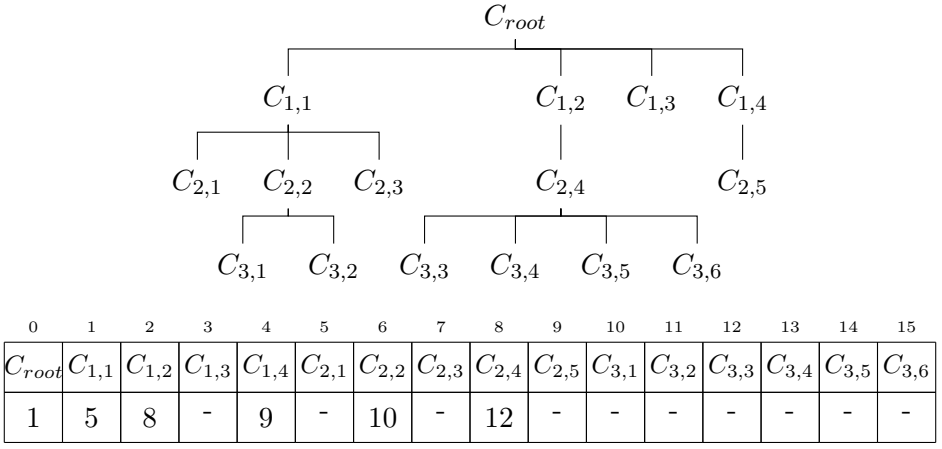
4.2.2 Chunks

```
class Octree_1Bit:
    Bounds world_bounds
    List<Chunk> chunk_list
    List<uint32> offset_list
    List<Primitive> primitives_list

    struct Chunk:
        uint32 child_chunks_offset
        uint32 leaf_offset
        List<uint64> bitfield
```

The process described in chapter 4.2.1 always creates a single bit field, containing all nodes of the octree. In a big enough scene, this can become a problem, since ranking requires touching all previous bits, which becomes very expensive with large bit fields. To avoid this, we can divide the octree up into chunks. These chunks are subtrees with a specified maximum size. By dividing our tree into subtrees, certain interior nodes become leaf nodes of their subtree. These nodes are nonetheless referred to as interior nodes, and, more specifically, as interior nodes with children in another chunk or interior nodes pointing to another chunk. Each chunk can contain an arbitrary number of these types of nodes.

Before we look at how such chunks are created and structured, let us first look at the tree that these chunks create. Specifically, when we later divide our octree into such a chunk structure, several interior nodes will have their child nodes in a different chunk, and will therefore act as a pointer to another chunk. Due to the fact that each chunk can contain nodes with child nodes in another chunk, therefore pointing to a different chunk, the chunks form a sort of hierarchy. Figure 4.11 provides an example for this.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C_{root}	$C_{1,1}$	$C_{1,2}$	$C_{1,3}$	$C_{1,4}$	$C_{2,1}$	$C_{2,2}$	$C_{2,3}$	$C_{2,4}$	$C_{2,5}$	$C_{3,1}$	$C_{3,2}$	$C_{3,3}$	$C_{3,4}$	$C_{3,5}$	$C_{3,6}$
1	5	8	-	9	-	10	-	12	-	-	-	-	-	-	-

Figure 4.11: An example tree of chunks

In this example, chunk C_{root} contains 4 interior nodes that point to other chunks. To store these chunks as data, we make a list of all the chunks sorted in breadth-first order. Thus, C_{root} is the first entry of the list, then come the chunks $C_{1,i}$, sorted in ascending order for i, then the $C_{2,j}$ chunks, etc. Each chunk then contains an offset value, denoting the index of its first child chunk within that list. In figure 4.11, the list shows how such a layout would look, with the bottom row being the index of the first child chunk of each chunk. Any value denoted as "-" means, that the corresponding chunk does not contain any interior nodes pointing to other chunks. In practice, this value would be set, but never

used. It should be noted, that each chunk has the same size, even if some might not be completely filled with nodes. This is necessary, in order to store them contiguously.

4.2.2.1 Chunk Construction

Now that the structure of the chunks is established, let us take a look at how each chunk is constructed. As previously noted, a chunk is simply a subtree of the original octree, with a given maximum size that is a multiple of 8. To construct the layout of such a subtree, we look once again at figure 4.9. The reason we are able to reconstruct the graph from figure 4.8 from this bit field is, that all child nodes of any interior node are always grouped together, and these groups appear in the same order as their parent interior nodes. Therefore, when constructing the chunks from an octree, it is important to keep these sibling nodes together in the bit field. This means, that any set of 8 siblings needs to be always grouped together and within the same subtree. On top of this, in the same way that in figure 4.8 we could leave out explicitly writing down the value of the root node, we can do the same within each chunk. This is, because the root node of a subtree always corresponds with an interior node of the original octree. This is also a direct consequence of the previously mentioned policy of keeping sibling nodes within the same subtree. Hence, the size of such a subtree is always a multiple of 8.

As only a limited number of nodes fit into each chunk, the child nodes of some interior nodes can not fit into the same chunk as those interior nodes. These child nodes are later put into a different chunk. To be able to determine which child nodes in a different chunk belong to which interior nodes, we use the breadth-first ordering of the bit field. Assuming our chunk can fit $8n$ nodes, and knowing that child nodes of an interior node are always grouped together, we know that the children of n interior nodes can fit into a chunk. The first set of 8 nodes is taken up by the implicit root node. This means, that the child nodes of the first $n-1$ interior nodes, or `1`-bits, can fit in the same chunk, while the child nodes of any interior node thereafter will have to be in a different chunk.

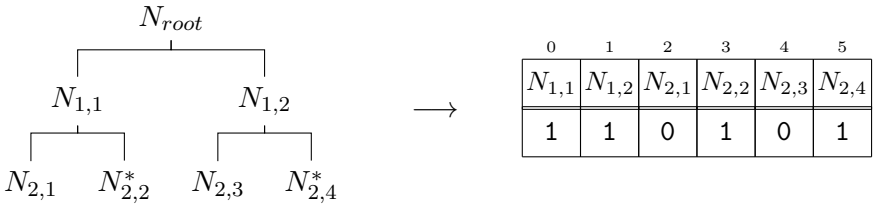


Figure 4.12: The first subtree of binary tree from figure 4.8

Figure 4.12 shows what such a chunk would look like for a binary tree, using the binary tree from figure 4.8 as an example. This chunk has a maximum node size of 6, meaning its bit field is 6 bits long. The way the chunk is constructed, is by starting at the root node, N_{root}, and traversing the tree in breadth-first order until all nodes of the tree are in the chunk, or the chunk is full, meaning 6 nodes have been traversed, excluding the root node. In this case, this traversal ends when node $N_{2,4}$ is traversed, since it is the 6^{th} node to be traversed in breadth-first order. The chunk is therefore fully filled, including all nodes from layer 1 and layer 2. We assign a `1`-bit to all interior nodes of the original tree, and a `0`-bit to all leaf nodes of the original tree. This means, that any node marked as N^* is treated as an interior node, and is therefore saved as a `1`-bit.

4.2.2.2 Interior Nodes with Child Nodes in a different Chunk

We can then use the child function described in chapter 4.1.1, that tells us the starting position of the child nodes of a given interior node with index i. In this example, child(0) =

2 and child(1) = 4. However, child(3) = 6, which is an index outside of the chunk. Because of this, we know that the two child nodes of this interior node are actually in another chunk. Knowing this, let us take a look at how the other chunks for the binary tree from figure 4.8 would be constructed. Figure 4.13 shows the way this binary tree would be partitioned into chunks, using this breadth-first traversal method.

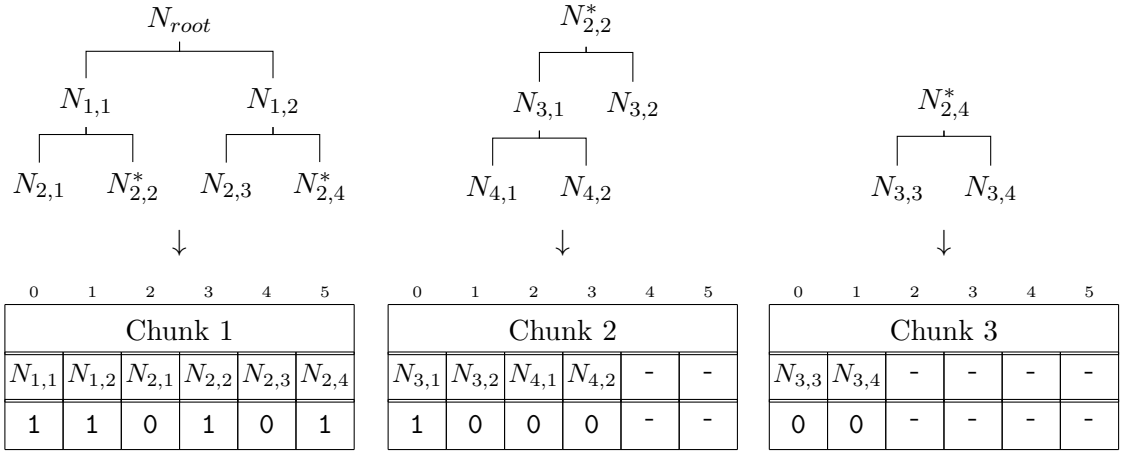


Figure 4.13: Subtree division on binary tree from figure 4.8

If we extract the subtree that we already filled into a chunk in figure 4.12, we are left with several segregated subtrees. Repeating this same process of filling chunks for each of these subtrees, until all nodes are in chunks results in our final chunk structure. Figure 4.13 shows the result of this on the example from figure 4.9.

The first 2 spots in the bit field for a chunk are always filled by the child nodes of the root node of that subtree. As the chunks in figure 4.13 have 6 bits available, this leaves 4 bits to be filled. Since any 2 consecutive bits are always filled as a pair, because sibling nodes need to be grouped and within the same chunk, there are 2 available slots (of 2 bits each) for these sibling groups. Ultimately, this means that the first 2 interior nodes must be interior nodes in the subtree too, meaning their child nodes are part of the same subtree or chunk. Any subsequent interior node is therefore an interior node with children in a different subtree or chunk.

To determine which chunk such a subsequent interior node refers to, we just consider the how many-th such interior node with child nodes in a different chunk it is. We describe this function as the *childchunk* function, where childchunk(i) returns the index of the chunk that contains the child nodes of the interior node i. Using i_j to return to the node with index i in chunk j, for the chunk in figure 4.13, childchunk(3_1) = 2 and childchunk(5_1) = 3, referring to chunk 2 and chunk 3, respectively.

More specifically, with the offset to starting position of the child chunks of a given chunk as $\text{offset}_{\text{chunk}}$ and the number of interior nodes per chunk that can have child nodes in the same chunk n, the exact function is

$$\text{childchunk}(i_j) = \text{offset}_{\text{chunk}} + rank(i_j) - n,$$

with i being an the index of an interior node in chunk j, with child nodes in another chunk.

4.2.2.3 Reconstruction

Figure 4.14 shows the same chunk structure as figure 4.13, but with an extra row added to the bottom. This extra row contains an entry for every interior node. If this entry is

of the form B_i, the child nodes of that interior node are within the same chunk, starting at index i. If the entry is of the form C_j, then the child nodes of that interior node are in Chunk j.

	0	1	2	3	4	5
Chunk 1	1	1	0	1	0	1
	B_2	B_4	-	C_2	-	C_3

	0	1	2	3	4	5
Chunk 2	1	0	0	0	-	-
	B_2	-	-	-	-	-

	0	1	2	3	4	5
Chunk 3	0	0	-	-	-	-
	-	-	-	-	-	-

Figure 4.14: Determining the position of the child nodes of each interior node

In more general terms, and now in regards to an octree, a chunk with space for n groups of siblings (therefore for $8n$ bits), has, at the time of its creation, $n-1$ slots for these sibling groups, since the first slot (the first 8 bits) is taken by the child nodes of the implicit root node. Because of the breadth-first policy when filling these chunks, the first $n-1$ `1`-bits have children in the same chunk, while any `1`-bit thereafter has its child nodes in a different chunk.

4.2.2.4 Leaf Node Data

Lastly, we have to determine how leaf nodes access their primitives. In the version of this octree without chunks, as discussed in chapter 4.2.1, the primitive information of the n^{th} node is contained in the n^{th} and $(n+1)^{th}$ entry of `offset_list`, which then lets us know which entries of `primitives_list` to access. The nodes in this chunk structure, however, should only know directly about other nodes within their chunk, meaning that we can determine the how many-th leaf node a node is within that chunk, but not the how many-th leaf node it is in total. To keep using the `offset_list` and `primitives_list` structure as previously, we add a leaf node offset value to each chunk, that essentially contains the sum of the number of leaf nodes in the previous chunks. In other words, adding the local position of a leaf node in a chunk (the how many-th leaf node it is within that chunk) to the leaf node offset results in the global position of the leaf node in the octree (the how many-th leaf node it is in the whole octree).

4.2.2.5 Comparison to non-Chunk Octree

Using this chunk structure, we have addressed the problem mentioned at the beginning of the chapter, which was, that the dependence of each node on all nodes before it could result in very expensive calculations as the traversal goes deeper into the octree. Using the chunk structure, all information about a node is contained within each chunk, using the previously established offset to child chunks of a given chunk as well as the leaf node offset. Furthermore, we can still store each node type using a single bit, even though an extra type of node was added. Specifically, the node that is an interior node, but whose child nodes are in another chunk. Even though this adds more overhead, meaning we are further away from actually storing each node fully with 1 bit, it can still remain relatively close.

The size of a CPU cache line is most commonly 64 bytes. Therefore, we want the chunk size to be in such a way, that multiple chunks fit perfectly into one cache line, or one chunk fits perfectly into multiple cache lines. We test for chunk sizes of 16, 32, 64 and 128 bytes. Imagine a chunk structure, where the two offsets are 32 bits, 4 bytes, each, and the bit field is 448 bits, or 56 bytes. This results in a chunk of 64 bytes. The chunk uses a total of 512 bits to store 448 nodes, meaning that the chunk uses an average of $\frac{512}{488} = \frac{8}{7} \approx 1.14$ bits per node. Increasing the chunk size to 128 bytes, this number goes

down to $\frac{1024}{960} = \frac{16}{15} \approx 1.07$ bits per node. We can also explore a chunk size of 32 bytes, where two chunks can fit into a cache line. For such a chunk, we use 256 bits to store 192 nodes, resulting in $\frac{256}{192} = \frac{4}{3} \approx 1.33$ bits per node. Lastly, we consider a chunk size of 16 bytes. Even though this is not a multiple of 32 bytes, two chunks can fit into a cache line side by side, ultimately also filling up the cache line. A chunk of 16 bytes has 128 bits of data, 64 of which are used for the offsets. It therefore results in $\frac{128}{64} = 2$ bits per node. These are, however, best case values, which only apply when a chunk is fully filled with nodes. In a worst case scenario, each chunk contains only the 8 child nodes of the implicit root node of the chunk. For a chunk size of 16 bytes, 32 bytes, 64 bytes and 128 bytes, we would use $\frac{128}{8} = 16$, $\frac{256}{8} = 32$, $\frac{512}{8} = 64$ and $\frac{1024}{8} = 64$ bits per node, respectively. In chapter 4.2.4 we compare these different values for the chunk size, as well as other parameters for a chunk, and compare the memory footprints and traversal speeds, as well as how filled the chunks are on average.

A big disadvantage to storing each node type with a single bit in this way is that the original octree structure quickly becomes split into many smaller subtrees. Figure 4.13 demonstrates this concept for a binary tree. Even though the original binary tree contains 12 nodes (excluding N_{root}), and each can hold 6 nodes, the structure requires 3 chunks, because of the way the subtrees are divided. On average, each chunk is only $\frac{2}{3}$ filled. This happens, because each chunk is limited to a single subtree. As soon as a part of the original octree is cut off to create a subtree with less nodes than can fit into a chunk, that chunk can no longer be fully filled. Because of the breadth-first traversal, and creating the chunks top-down, each chunk leads to the tree being divided into multiple, smaller subtrees. These are often smaller than the chunk size, which can potentially lead to a very small portion of the bit field of a chunk being filled. On top of this, many chunks might also be filled by the bare minimum of nodes, meaning all child nodes of the implicit root node of the chunk node are leaf nodes, thus wasting a lot of space in the chunk. One solution to this could be to add a second bit per node type, that specifies, for interior nodes, whether their child nodes are within the same chunk or not. That way, much more flexibility can be achieved when choosing the fill method of the chunks, allowing chunks to be filled in a more efficient way. However, this of course adds more explicit data stored per node. While considered, such a method is not explored in this thesis.

4.2.3 Ray Traversal

Since this octree has the same topology as the octree constructed in chapter 4.1, a lot of the traversal is the same. The traversal order of any child nodes is determined in the same way described in chapter 4.1.3, and the general algorithm is the same as algorithm 2. The only difference is the way that information is accessed, since all data, including the topology, is stored differently than in the basic octree. Of course, when understanding how this data is stored, as described in chapters 4.2.1 and 4.2.2, it is not very far-fetched to understand how to traverse that data. Therefore, we will go through the rough steps. Henceforth, any time a nodes rank or index is mentioned, it refers to the local rank and local index within the chunk, respectively.

If any given node is an interior node, we simply apply the child or childchunk function, according to the amount of interior nodes that came before it. Assuming n to be the number of interior nodes that can have child nodes in the same chunk, any given interior node with index i has children in the same chunk if $\text{rank}(i) < n$. In such a case, we use the child function, otherwise the childchunk function. If any given node at index i is a leaf node, we can figure out which primitives are contained within the AABB of that node using the same methods described in chapters 4.1.1, 4.2.1 and 4.2.2.

With this, as well as the traversal discussed in chapter 4.1.3, the 1-bit octree can be fully traversed in any order required by a ray.

4.2.4 Comparison of Chunk Parameters

Since the topology of the 1-bit octree is the exact same than that of the basic octree, we do not have to test the multiplication and primitive thresholds again, and can assume the same values for both as for the basic octree. Instead, we compare different parameters for the construction of the chunks. Figure 4.15 shows a comparison of different chunk sizes affecting intersection time and topology size. As established, each chunk has a 32-bit offset for its child chunks, as well as a 32-bit offset for the number of leaf nodes in the previous chunks. Therefore, reducing the chunk size increases the number of bits per node.

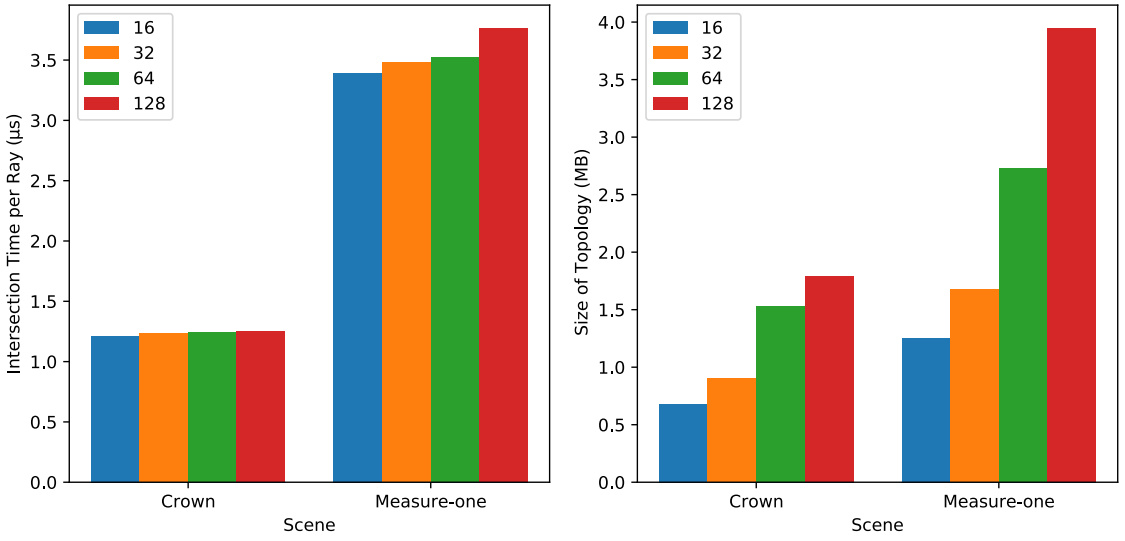


Figure 4.15: Intersection time and topology size for different chunk sizes

As the graphs in figure 4.15 shows, both the intersection time, as well as the size of the topology are better for lower chunk sizes. For the *Crown* scene, the differences in intersection time are relatively small, but still favor a smaller chunk size. For the Measure-one scene, the differences are more obvious. The reason for this improvement when using a smaller chunk size is the breadth-first order of the bit field. As shown in chapter 4.2.2, specifically in figures 4.12 and 4.13, the division of the subtree into chunks in breadth-first order splits the remaining tree into many subtrees. Since each chunk of our specified layout can only contain a single such subtree, many chunks are not filled to their fullest. Additionally, many chunks only contain 8 leaf nodes, further adding to this inefficiency. The bigger the first chunk, the more severely it divides the tree into small subtrees. Each of these subtrees do not contain nearly enough nodes to fill these bigger chunks. Figure 4.16 clearly demonstrates this.

As the left graph shows, the total number of chunks does decrease for both scenes, as the chunk size increases, as is expected. However, the chunk size doubles, whereas the number of chunks does not quite halve. This results in a larger overall topology size. The increased execution time can come from multiple factors. Firstly, the increased size of the topology has a direct impact on the intersection time. The results of chapter 4.1.4 had an increased size of topology, but lead to better intersection times. This was due to the fact, that in that chapter, the increased size of the topology lead to a differently structured octree, with more nodes and more detail, resulting in a trade-off between size of topology and the structure itself. Here, however, the exact same octree is simply saved with more data, therefore increasing the memory footprint and causing increased intersection times. Secondly, as previously established, one of the reasons for using such a chunk structure at all is, that the rank function, introduced in chapter 4.1.1, becomes less efficient for larger

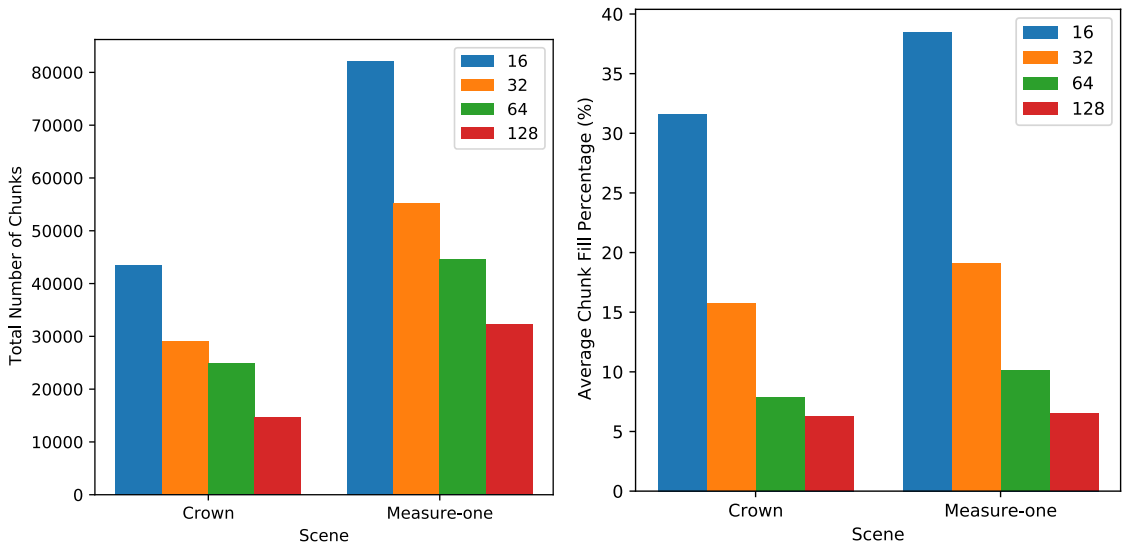


Figure 4.16: Number of chunks and chunk fill percentage for different chunk sizes

bit fields. The larger such a chunk is, however, the larger the bit field it contains, causing an increased calculation time of the rank of a node.

The graph on the right shows the average chunk fill percentage, or the average amount of nodes that are contained within a chunk relative to how many nodes the chunk has space for. As can be seen, this value has a clear downwards trend as the chunk size increases, further showing the inefficiency of the higher chunk sizes in combination with the breadth-first order of the bit fields.

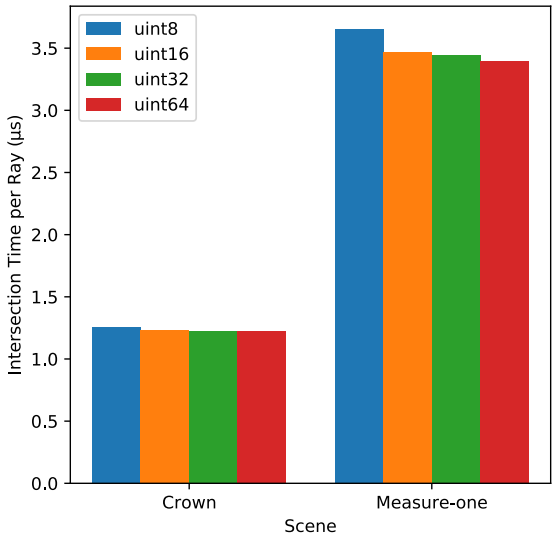


Figure 4.17: Intersection time and topology size for different bit field data types

The other parameter defining a chunk is the underlying data type of which a bit field is made. While this value has no effect on the number of chunks or how filled they are, we test the effect that changing this value has on the calculations that have to be performed during traversal. The bit fields of our implementation are arrays of this data type. In figure 4.17 we compare 8-, 16-, 32- and 64-bit unsigned integers for the bit fields. The graphs show a clear, albeit small, improvement in intersection time per ray for larger data types. The likely reason for this is the decreased amount of rank functions that need to be

performed, since more nodes can be ranked at the same time. The reason that the jump from 8- to 16-bits is more significant than the jump from 16 to 32 can be attributed to the fact that our rank function is implemented using a so-called *Popcount* intrinsic function. These functions did not have implementations for an 8-bit variable, meaning that any 8-bit variable is first converted into a 16-bit variable. This is, of course, very inefficient, and therefore results in such a big difference in intersection time between the 8-bit and 16-bit bit fields. Not only does each 8-bit integer of the bit field have to be converted to a 16 bit integer first, but nothing is gained either, since both the 8 bit, as well as the 16 bit versions have to perform a 16 bit rank operation.

Ultimately, these graphs clearly show, that the best chunk parameters are a chunk size of 16 bytes with bit fields of 64-bit unsigned integers. With these values, we achieve not only a faster intersect speed, but also a smaller size of topology, making them the clear choice for our goals. Therefore, any 1-bit Octree used henceforth uses these values for the chunks.

5. Compressed Bounding Volume Hierarchies for Ray Tracing

A bounding volume hierarchy (BVH) is, just as an octree discussed in chapter 4, a data structure that can be used to accelerate ray traversal. Unlike octrees, it does not directly partition the three-dimensional space, but rather the geometry within that space. At the lowest level, an axis-aligned bounding box (AABB) surrounds one or several primitives. These AABBs are then, at the next level, also grouped and packed into larger AABBs. Ultimately, there will only be one AABB encompassing all other AABBs and primitives. A BVH is the hereby created tree structure. Since every AABB inside a BVH is arbitrary, and not rigid like with the octree, all information about the AABB of each node needs to be saved. This is, as compared to an octree, a severe disadvantage, as suddenly a lot more data needs to be saved explicitly per node, resulting in a much larger memory footprint for the topology. Since the topology of a structure needs to be accessed a lot to find the point of intersection of a ray, having a topology that needs more data is disadvantageous, because different data needs to be constantly loaded in from memory.

However, it is also important to keep in mind that a BVH structure offers several advantages over an octree. For example, each primitive is contained in exactly one leaf node, meaning no primitives need to be tested for intersection more than once, and no other methods need to be applied to prevent this repeated testing. The AABBs also wrap more tightly around their primitives, increasing the chance that the AABB is missed by the ray, since it is smaller. As established, this is advantageous, as it allows skipping all primitive intersection tests within that AABB. Lastly, all the explicit saving of the data also means that no data needs to be computed from implicit information, but is readily available.

With all this in mind, we attempt to compress the topology of the established BVH. We first discuss a basic BVH from [PJH16] in chapter 5.1, then compress the AABBs by quantizing them using the methods from [MW06] in chapter 5.2. In chapter 5.2.1 we apply the same bit field techniques that we applied for the octree in chapters 4.2.1 and 4.2.2. Lastly, we compare the performance of two slightly different approaches to the quantization in chapter 5.2.3.

5.1 Basic Bounding Volume Hierarchy

```
class BVH:
    BVH_Node root_node
    List<Primitive> primitives

    struct BVH_Node:
        AABB aabb
        union:
            uint32 second_child_offset
            uint32 primitive_offset
        uint16 num_primitives
        uint8 split_axis
        uint8 padding
```

The specific implementation we will look at for a BVH comes from *Physically Based Rendering* [PJH16], and is also a binary tree. This means, that each node is either an interior node with exactly two child nodes, or a leaf node. Each node saves its AABB as six 32-bit floating point numbers, one min and one max for each axis, x, y and z. On top of that, each layer of the binary tree BVH is split among a certain axis, depending on the used BVH builder. While the axis that each interior node is split at does not need to be explicitly saved, it is advantageous, as it allows for much faster traversal [PJH16], as is discussed in chapter 5.2.2.

This BVH saves each child nodes of the interior nodes in depth-first order, so the first child directly follows the node and the second child is pointed to by `second_child_offset`. For leaf nodes, this same 32-bit pointer points to the position of the primitives in a list of primitives. There is also a 16-bit integer saving the number of primitives in a given leaf node. This value is 0, if the node is an interior node, meaning that this value also includes the information about what type of node the considered node is. The previously mentioned saved split axis only takes on three different values, meaning that 2 bits would be enough to save it. It is therefore saved as the smallest primary data type, taking up 8 bits.

All in all, each node needs 192 bits for its AABB, 8 bits for its split axis, 32 bits for its pointer, and 16 bits for its node type and number of primitives, adding up to 248 bits per node for the topology. To achieve the bits per node being a multiple of 32 bits and a multiple of 64 bits, allowing for guaranteed alignment in memory, an 8 bit padding is added, bringing up the total number of bits per node to 256.

5.1.1 Ray Traversal

For a ray to start traversing a BVH, it is first tested for intersection against the world AABB. If it does not intersect, we already know that the ray misses everything inside those AABB, and therefore does not return a hit. If it intersects with the world AABB, or the AABB of any other interior node for that matter, its two child nodes are traversed next. The order is dependant on the split axis of the considered interior node, as well as the direction of the ray in that axis. For example, if the split axis is the x axis, and the direction of the ray in the x axis is positive, then the ray first traverses the child node whose AABB has smaller x-values. In the case of the basic BVH, this would be the node that directly follows the interior node. If the direction of the ray in the split axis is negative, the second child node is traversed first, which, in the case of the BVH, is pointed to by the `second_child_offset`. This is done to increase traversal speed, as a ray going in a positive split axis direction passes the smaller split axis value before the larger ones.

Of course, depending on the point of origin of the ray, as well as the directions in its other two axes, it might miss the closer AABB, and hit the one further. However, if the ray does intersect with the closer AABB and, furthermore, with a primitive contained within this closer AABB, we can now know only to check for intersections closer than that primitive. If the ray then intersects with the farther AABB at a point behind said primitive, we can skip the farther AABB entirely. This enables a much faster traversal, since it allows entire regions to be excluded from intersection tests.

5.2 Quantized Bounding Volume Hierarchy

```
class BVH:
    BVH_Node root_node
    AABB world_aabb
    List<Primitive> primitives

    struct BVH_Node:
        Quantization_Key quant_key
        union:
            uint32 second_child_offset
            uint32 primitive_offset
        uint16 num_primitives
        uint8 split_axis
        uint16 padding

    struct Quantization_Key:
        uint8[3] min
        uint8[3] max
```

The above structure is a stepping stone for the final implementation, and was never implemented as shown. The construction and compression of the quantized BVH is based on the basic BVH, but also uses many of the same methods and idea used for the 1-bit octree, described in chapter 4.2. However, since the BVH needs to save much more data explicitly per node, compressing only the node type to a bit field, as done in chapter 4.2 for the octree, results only in a relatively small improvement. As established, the basic BVH needs 256 bits per node, with 192 of those being taken up by the AABB. We therefore compress the AABB by quantizing it similar to [MW06], before applying the bit field compression in chapter 5.2.1. It is important to note, that unlike the compression by using a bit field, this quantization does lose some data, and is not a perfect reconstruction of the original BVH. The quantized AABB requires each node to only explicitly save six 8-bit values, rather than six 32-bit values, hence saving $6 * (32 - 8) = 144$ bits per node, and reducing the total bits per node to $256 - 144 = 112$. To make each node a multiple of 32 bits, we add a 16 bit padding. This allows for guaranteed alignment in memory. Ultimately, the basic BVH with quantized AABBs requires 128 bits per node, half of the basic BVH without quantization.

This of course brings with it the advantage of reducing the memory footprint, but also brings notable disadvantages with it. As seen shortly, reconstructing the AABB of a node from the quantization requires extra calculations. Secondly, a quantized AABB is slightly larger than the original AABB, therefore resulting in some rays to intersect with the quantized AABB that would not intersect with the exact AABB, and therefore does not intersect with any primitives inside this AABB. This might result in unnecessary primitive intersection tests for such a ray. However, the trade-off between these larger quantized AABB with a lower memory footprint and the increased number of nodes that are traversed and primitives that are intersected is worth it, as shown by [MW06].

To quantize a the AABB of a node, a larger AABB is needed, relative to which the quantization is done. In a BVH, the AABB of any node is always fully encompassed by the AABB of its parent node. Hence, we quantize the AABB of each node relative to the parent AABB. For clarification, the node to be quantized is referred to as the *base node*, while its parent node is referred to as the *parent node* until chapter 5.2.1. To quantize the base AABB relative to the parent AABB, the parent AABB is divided into an evenly

spaced grid. The base AABB is then snapped to the lines of this grid, with the minimum of the base AABB being snapped to the closest smaller point in the grid, and the maximum being snapped to the closest bigger point in the grid. If we interpret each line on the grid as having an index, starting at 0 for the minimum, and going up by 1 for each line, the resulting quantized AABB of the base node can then be saved by knowing the base AABB, as well as the index of the grid line that the quantized AABB snapped to, for each axis. This results in six such indices, one for the minimum of and one for the maximum each axis. By choosing a $255 * 255 * 255$ grid to divide the parent AABB, these indices go from 0 to 255, therefore taking on 256 different values, and can be saved with 8 bits each, as $2^8 = 256$. Algorithms 3 and 4 show how this is calculated.

Algorithm 3 Determine the quantization key of a node

```
1:  procedure DetermineQuantizationKey(parentAABB, baseAABB)
2:      quantKey                          ▷ Stores six 8-bit integer values between 0 and 255
3:      for all axis do
4:          distance := parentAABB.max[axis] - parentAABB.min[axis]
5:          offset := parentAABB.min[axis]
6:          quantKey.min[axis] := ⌊255 * (baseAABB.min[axis] - offset) / distance⌋
7:          quantKey.max[axis] := ⌈255 * (baseAABB.max[axis] - offset) / distance⌉
8:      end for
9:      return quantKey
10: end procedure
```

Algorithm 4 Determine the quantized AABB of a node using its quantization key

```
1:  procedure DetermineQuantizedBounds(parentAABB, quantKey)
2:      quantAABB                                  ▷ The resulting quantized AABB
3:      for all axis do
4:          distance := parentAABB.max[axis] - parentAABB.min[axis]
5:          offset := parentAABB.min[axis]
6:          quantAABB.min[axis] := (distance * quantKey.min[axis] / 255) + offset
7:          quantAABB.max[axis] := (distance * quantKey.max[axis] / 255) + offset
8:      end for
9:      return quantKey
10: end procedure
```

We call the resulting six 8-bit integer values the *quantization key* of a node. Since the AABB of the parent node are also saved through a quantization key, the world AABB of the scene have to be explicitly saved. To calculate the quantized AABB of any given node, all previous quantized AABB need to be known.

5.2.1 Data Layout and Bit Field Compression

```
class QBVH:
    AABB world_aabb
    List<Primitive> primitives_list
    List<uint32> offset_list
    List<Chunk> chunk_list
    List<Node_Info> node_info_list

    struct Chunk:
        AABB chunk_aabb
        uint32 child_chunks_offset
        uint32 sizes_offset
        uint32 node_info_offset
        List<uint64> bitfield

    struct Node_Info:
        Quantization_Key quant_key
        uint8 split_axis

    struct Quantization_Key:
        uint8[3] min
        uint8[3] max
```

In this section, we apply bit field compression to the just established version of the BVH, which uses the quantized AABBs. This bit field compression is the exact same as the one for the octree, described in chapter 4.2, with some additions.

5.2.1.1 Quantization Key and Split Axis

Firstly, we need a way to associate nodes with their quantization key and split axis. For this, we save these two items together, in a structure like figure 5.1. We can then make a list, where each entry is of this structure, calling this list `node_info_list`.

Figure 5.1: Data Layout of Quantization Key and Split Axis

The list is saved in the same order that the nodes are saved in the node bit field, then by knowing the index of a given node (which is always known during traversal), we can access the quantization key and split axis of that node at that same index of the `node_info_list`. Figure 5.2 shows an example of this, with n_i as the node information, containing quantization key and split axis, for node at index i. For example, the node in the bit field at index 3, which is an interior node, has its node information n_3 stored also at index 3 in the node info list.

We then save this `node_info_list` outside of the chunk structure, because it is relatively large, and because it compensates for close to empty chunks.

5.2.1.2 Chunks

The chunk structure also looks slightly different to that of chapter 4.2.2. Firstly, we need an additional offset for the newly established `node_info_list`. Since this list has an

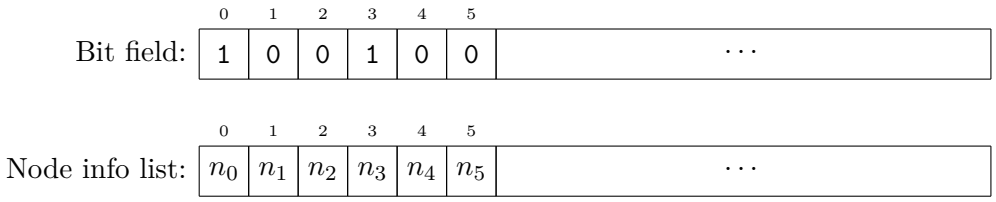


Figure 5.2: Accessing the node information n_i from a node at index i

entry for every node, we need the information of how many total nodes came in all chunks before a given chunk. This information is not available to us in the chunk structure of the octree. We therefore add a 32-bit `node_info_offset` to the chunk structure. Lastly, we add an exact AABB to each chunk, which is the AABB of the implicit root node of each chunk. This serves two purposes. The first purpose is a simple milestone for basing our quantization off of an exact AABB again, instead of the quantized AABB.

The second reason, is that this allows us to take a different approach to the quantization. Previously, the AABB of a node was quantized in relation to the AABB of its direct parent node. Instead, we can now use the AABB saved in each chunk as the quantization frame. For the lower level nodes of each chunk, this takes away a relatively large amount of precision. However, with the parent node as the quantization frame, we have to carry over the calculated quantized AABB of each node in a stack during traversal, since calculating a the quantized AABB of a node is dependant on every node on the direct path from that given node to the implicit root node of the chunk. With this new approach, however, every quantized AABB can be calculated independently of any other node. In chapter 5.2.3 we compare these two methods of quantization in number of nodes traversed and number of primitives tested for intersection, as well as overall traversal speed.

5.2.1.3 Comparison to Uncompressed BVH

Compared to the chunks of the octree, a lot more explicit information needs to be saved. Each chunk saves a full $6 * 32 = 192$ bit AABB, three 32-bit offsets and the bit field. Excluding the bit field, each chunk contains 288 bits. For the chunk size, we consider a chunk of size 64 Bytes, as well as 128 Bytes. This is, as with the octree, because a cache line is typically 64 Bytes long. We cannot consider a chunk size of 32 Bytes, since the chunk data excluding the bit field already requires 288 bits, which is 36 Bytes. The basic BVH needs 256 bits, or 32 Bytes, per node, as seen in chapter 5.1. The basic BVH with quantized AABBs needed 128 bits per node, as seen in chapter 5.2.

We first consider the best case scenario, where each chunk is fully filled. For a chunk size of 64 Bytes, the bit field can hold $3 * 64 = 192$ nodes. We therefore divide the 64 Bytes, or 512 bits, by the number of nodes, resulting in $\frac{512}{192} = \frac{8}{3} \approx 2.67$ bits per node. For a chunk size of 128 Bytes, the bit field can contain up to $11 * 64 = 704$ nodes. We divide the 128 Bytes, or 1024 bits, by the number of nodes, resulting in $\frac{1024}{704} = \frac{16}{11} \approx 1.45$ bits per node. However, we have to also take into consideration the quantization key of $6 * 8 = 48$ bits and split axis of 8 bits for each node, since they are part of the topology, even if they are not contained within the chunks themselves. This adds 56 bits per node, bringing the aforementioned values up to around 58.67 bits per node and 57.45 bits per node for a chunk of size 64 Bytes and 128 Bytes, respectively. These values are significantly lower than the 256 bits per node that the basic BVH needs, or the 128 bits per node that a basic BVH with quantized AABBs needs.

In the worst case scenario, a chunk contains only the two child nodes of the implicit root node of the chunk. With a chunk size of 64 Bytes, or 512 bits, we therefore require $\frac{512}{2} = 256$ bits per node. With a chunk size of 128 Bytes, or 1024 bits, we require $\frac{1024}{2} = 512$

bits per node. Again, we have to consider the quantization key and split axis, bringing these values up to 312 and 568 bits per node, respectively. Not only are these values larger than the 256 bits per node of the base BVH, but the compression also requires more calculations for extracting the implicit information during traversal. In such a worst case scenario, the memory footprint is increased, on top of those extra calculations.

We determine how filled these chunks are on average for different chunk sizes in chapter 5.2.3, evaluating the efficiency of our compression, as well as the effects on the traversal speed.

5.2.2 Ray Traversal

Traversal for the quantized BVH is analogue to traversal for the BVH. The data and the nodes are accessed slightly differently, as has been discussed in chapters 5.2.1, as well as 4.2.3.

5.2.3 Comparison of Chunk Parameters and Quantization Frames

As in chapter 4.2.4 with the 1-bit octree, we compare different chunk parameters for the chunks of our quantized BVH. Since the BVH chunk has 36 Bytes of offsets and other data, the smallest possible chunk size is 64 Bytes. Therefore, we only consider chunk sizes of 64 Bytes and 128 Bytes. As figure 5.3 shows, however, the results are identical as in chapter 4.2.4 with the 1-bit octree.

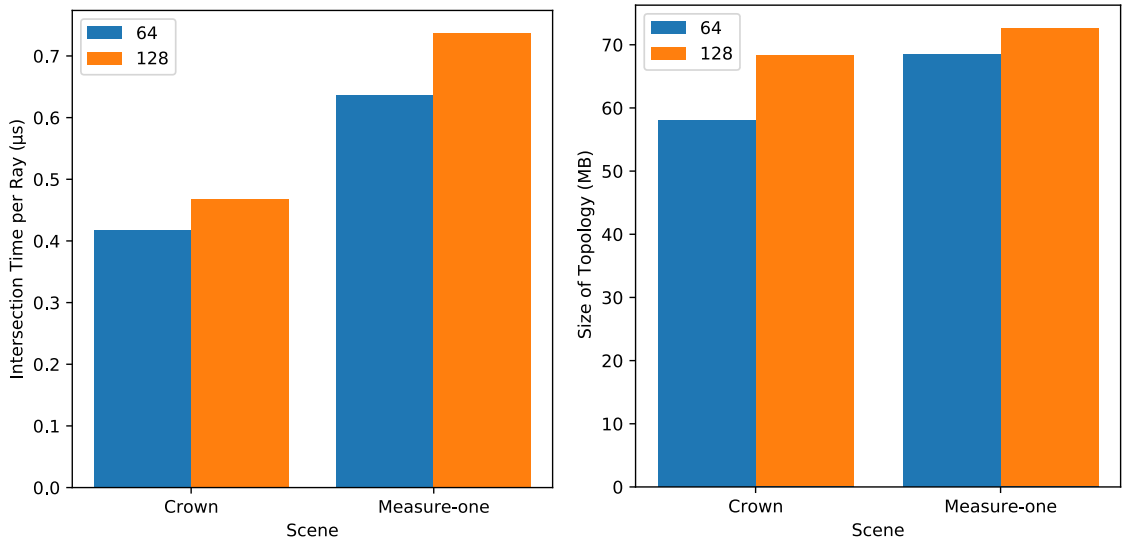


Figure 5.3: Intersection time and topology size for different chunk sizes

We can further use the same reasoning for why smaller chunk size is advantageous. As figure 5.4 shows, the total number of chunks is clearly more than half, as the chunk size is increased from 64 Bytes to 128 Bytes. This results, again in a much larger memory footprint. An interesting result here is, that with a chunk size of 128 Bytes, the *Crown* scene, containing around 3.5 Million primitives actually has more total chunks than the *Measure-one* scene, which contains around 4.5 Million primitives, so around 1 Million more. This very nicely exemplifies the inefficiency of the large chunk sizes in combination with the breadth-first sorting order. This is further demonstrated by the graph on the right, showing the average chunk fill percentage. In other words, this value shows the number of nodes that are inside a chunk, as compared to how many nodes the chunk has space for, on average. While all values are rather low, with the highest value being the 64

Byte chunks in the *Measure-one* scene at just under 10%, the average chunk fill percentage for the 128 Byte chunks is a significant amount lower than that of the 64 Byte chunks.

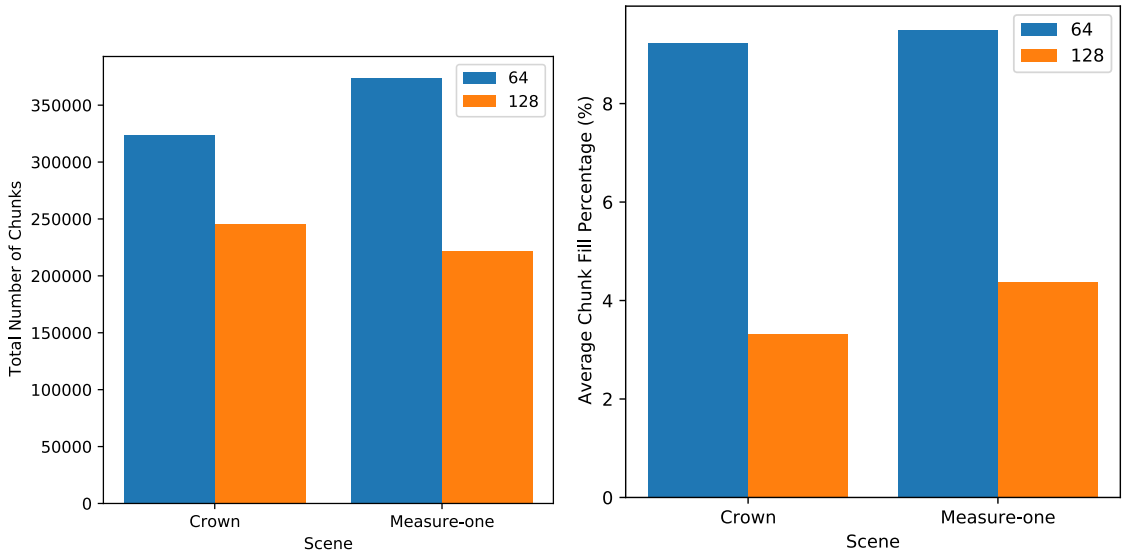


Figure 5.4: Number of chunks and chunk fill percentage for different chunk sizes

The chunks of our quantized BVH do, however, have another comparison to be made, which is unique to them, compared to the 1-bit octree. As discussed in chapters 5.2.1, the quantization of a the AABB of a node, which we refer to as the *base node*, always happens in relation to another node, which we refer to as the *quantization key root node*.

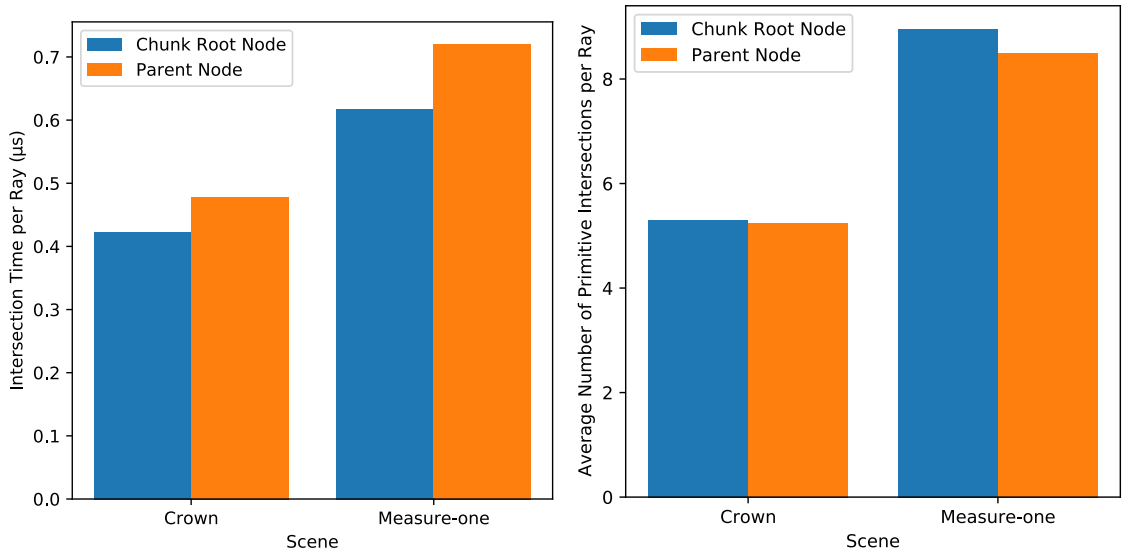


Figure 5.5: Comparison of the two different quantization key root nodes

We established two candidates for the quantization key root node. One is the AABB of the parent node of the node, while the other is the AABB of the implicit root node of each chunk. The parent node as a root node results in more precision during quantization, since the AABB of the base node can be quantized relative to an AABB closer to its own size. The advantage of using the AABB of the implicit root node of the chunk which contains the base node is, that this AABB is readily available, since it is explicitly saved in each chunk.

The left graph of figure 5.5 shows a clear improvement in intersection time when choosing the AABB of the implicit root node of the chunk as the quantization key root node, as compared to the parent node. The right graph shows, that using the parent node does lead to less primitive intersections per Ray. On top of this, no matter what node is used as the quantization key root node, that AABB has to be used along with the quantization key to determine the quantized AABB, to continue traversal. Therefore, there is also no increased amount of calculations. We can therefore conclude, that this result is simply due to fact that we have to keep a copy of the calculated quantized AABB for each node during traversal, when choosing the parent node, as this is the key difference between the two candidates for the quantization key root node. When choosing the implicit root node of the chunk, the AABB is already explicitly saved within the chunk, meaning that we do not need to save an extra stack of AABBs.

Ultimately, using the implicit root node of the chunk as the quantization key root node results in a faster intersection time. The different choices for the root node also do not have an impact on the size of the topology. Therefore this is the clear better choice.

6. Evaluation

To measure the performance of the compression techniques used, we look at how the compression reduces the memory footprint, the effects of the compression on the intersection time, as well as how the performance stacks up against existing solutions, such the BVH from [PJH16], as well as the *Embree* ray tracing acceleration structure by *Intel*, which also bases on a BVH [emb].

6.1 Testing Environment

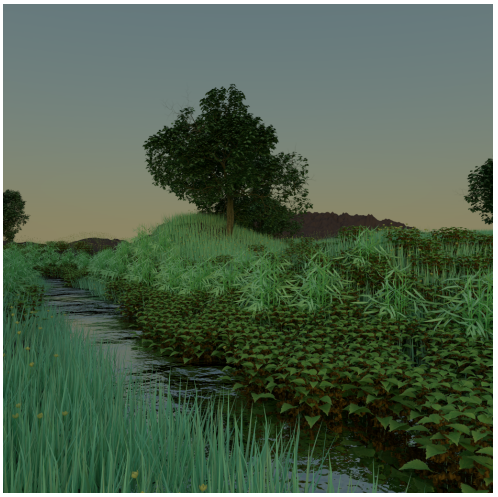
To put the results into context, the environment in which the results were produced are detailed in this section.

The framework that was used to implement the acceleration structures was the pbrt-v3 framework based on [PJH16]. The built in version of the BVH is the basic BVH described in chapter 5.1.

The *Embree* BVH acceleration structure by Intel [emb] was also integrated into the framework to compare the intersection times to a state of the art acceleration structure. For this integration, we could not use the internal primitives of *Embree*, but used user-defined primitives instead. Through callbacks, intersection tests are done with the primitives of *pbrt-v3*, which pass the results back to *Embree*. This is, of course, not as efficient as *Embree* could be. However, since our acceleration structures also use the primitives from *pbrt-v3*, the overhead is the same, and the comparison becomes fairer again, since we only care about the topology, and what difference the compression of it makes.

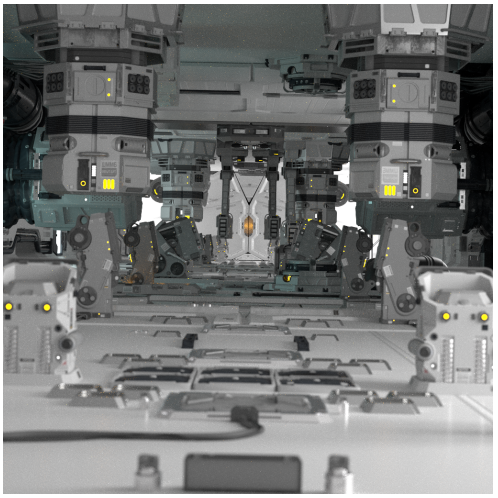
Figure 6.1 shows the rendered scenes that were used for evaluation [sce]. Below each image is the name of the scene, along with the number of primitives in the scene. If the scene had different renders from different angles available to it, the name of the chosen for evaluation is given in brackets. The scenes *Crown* and *Hair* both had a lot of primitives packed tightly together to create a lot of detail. These scenes were chosen, as the tightly packed primitives for the rigidity of the octree. The scenes *Ecosys* and *Landscape* both show an outdoor area, with the latter having an order of magnitude more detail than the former. These scenes were chosen as they require a very large volume of primitives to be covered, while still having details in areas that require the acceleration structure to adapt. The scene *Measure-one* is a room, and therefore a closed off space. It was chosen for similar reasons as the previous scenes, as it has a large volume to be covered, but still maintains a lot of detail all around said room. Lastly, the *Killeroo* scene was used

Killeroo (Killeroo-simple)
66,533 Primitives



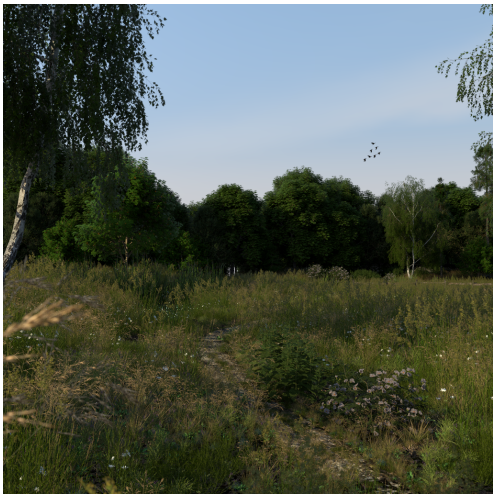
Ecosys
1,218,020 Primitives

Crown
3,540,215 Primitives



Measure-one (Frame25)
4,532,997 Primitives

Hair (Curly-hair)
26,332,640 Primitives



Landscape (View 0)
27,783,147 Primitives

Figure 6.1: Renders of the scenes used for evaluation

frequently in testing, as it is a rather small and simple scene, and allowed for quick testing. We do not evaluate it, as it does not offer anything that the other scenes do not cover, but it is mentioned because it provides a reference to what several decisions were based on.

All plots and graphs of any statistic are based on a single test result, not an average of multiple test results, due to time constraints. However, since the results of any intersection time are averaged out over millions of rays, the variance is kept relatively low. Furthermore, any results requiring more precision were performed with a higher number of pixel samples for the individual test, thus increasing the total number of rays, and further reducing the variance. Therefore, the test results should hold for an acceptable level of accuracy.

The tests were performed on a dedicated server on a CPU with 128 threads and more than 500 GB of memory. For the tests, only 8 of these threads were used per test, and only one test was run at a time. In extreme cases, around 7-8 GB of Memory were used during the construction of some data structures. However, most tests stayed well below that value.

The used rendering algorithm is a path tracing algorithm, with the common techniques, such as *Next Event Estimation* and *Russian Roulette*. The maximum depth for the path tracer was always set to 20.

6.2 Metrics

The *Size of Topology (MB)* metric used in several plots refers to the overall size of all data that is frequently accessed during ray traversal. In essence, this is data that we need to access for both interior and leaf nodes, therefore meaning we need to access it at every step of the traversal. This includes the entire node information for the BVH and all chunks for the quantized BVH or the 1-bit Octree. It does not include, for example, the `sizes_list` and `primitives_list` for any structure, as discussed in chapters 4.2.1 and 5.2.1, since those are only relevant for leaf nodes, and not for interior nodes. For the the octree, the number of leaf nodes traversed is around half the number of total nodes traversed. However, a lot of these nodes are empty, which is why the `primitives_list` is accessed infrequently. The `sizes_list` is accessed for about half the nodes of traversal, which is definitely relevant. We still leave it out of the size of topology, however, since it does not give information about the topology itself and we want to look at the compression of the topology only.

The metric *Intersection Time per Ray (µs)* describes the total time a ray spends intersecting the scene. This includes traversing the topology as well as doing intersection tests with the primitives in the scene. It includes both primary rays as well as rays for indirect lighting. However, shadow rays are not included.

The *(Average) Number of Primitive Intersections (per Ray)* metric, as the name suggests, counts the number of primitives that are tested for intersection by each ray. However, the scenes *Ecosys*, *Landscape* and *Measure-one* use instancing, meaning that they have the same structure of primitives in multiple locations of the scene, and create a separate acceleration structure for that structure, which is shared by the multiple instances of that structure of primitives. This acceleration structure is then regarded as a primitive for the rest of the scene. Therefore, whenever such a structure of primitives is entered by the ray, it is counted towards the number of primitive intersection stats. This is the same for all acceleration structures, and should furthermore be negligible for the results, but is still worth mentioning.

6.3 Effect of Topology Compression on Memory Footprint and Intersection

We assess the effectiveness of the compression by looking at the memory footprint of the topology, as well as the intersection time, for each acceleration structure.

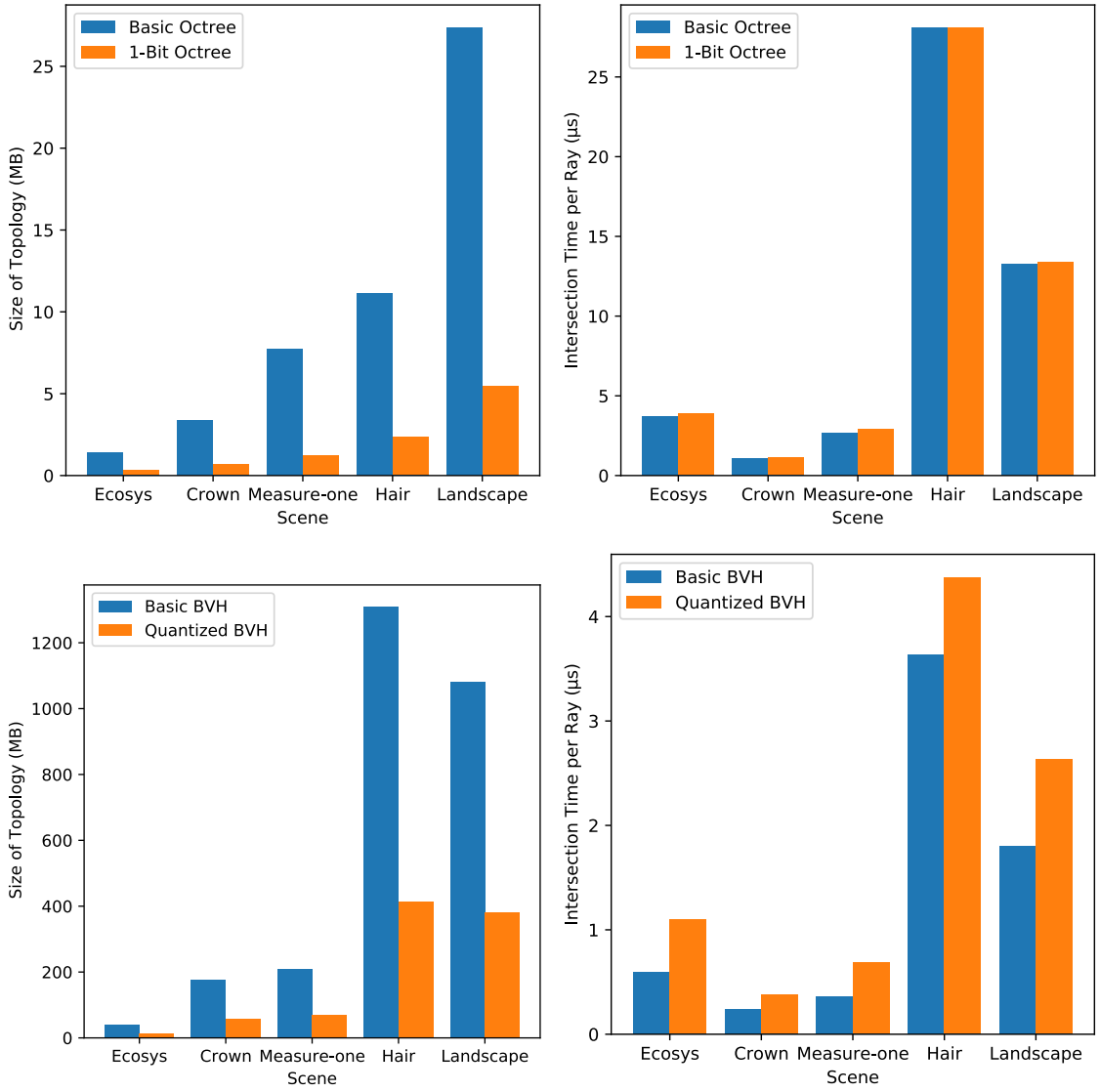


Figure 6.2: Compression of the topology and intersection time

As figure 6.2 shows, our compression techniques significantly reduce the memory footprint of the topology for both the octree and the BVH, for all scenes. However, the uncompressed structures almost exclusively outperform their compressed counterparts in terms of intersection time per ray. As can be seen on the graph for the octree, the intersection time increase from the bit field compression is only very small, when compared to the basic octree. In the case of the scene *Hair* it is even slightly faster. In this case, we get a very similar intersection time per ray while reducing the amount of data.

For the BVH, this difference in intersection time is quite significant. This is likely due to two main factors. Firstly, the extra quantization of the AABB results in even more implicit data that needs to be extracted, since the AABB of each node has to be calculated at each step of traversal from the quantization key, discussed in chapter 5.2. However, since [MW06] achieved similar performance in terms of rendering times when applying such a

quantization, it might be, that only our specific implementation of this quantization was not as effective. Another factor for this larger increase in intersection time could be that, due to the smaller amount of child nodes per interior node, all chunks that only contain leaf nodes therefore only contain 2 nodes, whereas the same situation for the octree would cause that octree chunk to contain 8 nodes. Since the chunks are filled in breadth-first order, this unfortunately happens regularly. On top of this, the chunks of the BVH are 64 bytes and cannot be smaller, due to the 40 bytes of offsets and other explicit data. This means that their bit field is 24 bytes, or 192 bits, large, while the bit field of the octree is only 8 bytes, or 64 bits, large. Because of these factors, the result is a much lower average chunk fill percentage for the quantized BVH as compared to the 1-bit octree, as figure 6.3 shows. This is also reflected in the larger relative topology size of the quantized BVH to the basic BVH, when compared to the relative topology size of the 1-bit octree to the basic octree in figure 6.2.

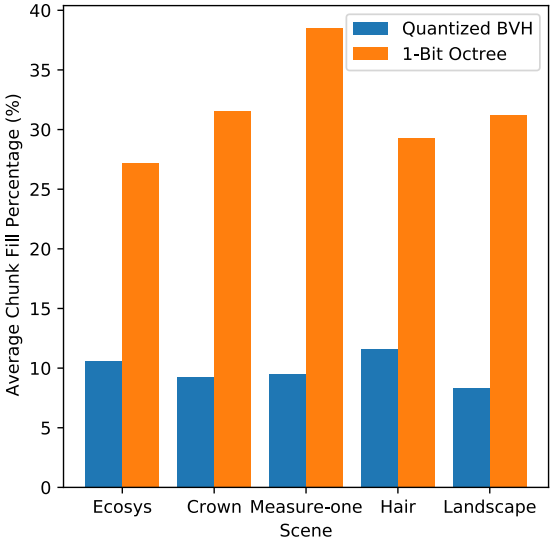


Figure 6.3: Average chunk fill percentage comparison

Ultimately, while the compression was very successful in reducing the memory footprint, our implementation results in a best case of similar intersection times per ray for the octree, and longer intersection times per ray for the BVH. Since our ultimate goal was reducing the intersection time through compression, the compression is unfortunately not as successful as we hoped.

The biggest reason for this, as seen in figure 6.3, is very likely the low average chunk fill percentage. This means, that the compression is not actually as effective as it could be. Making the size of the topology of the compressed acceleration structures even smaller by filling the chunks as much as possible would likely result in a performance increase as compared to the current compressed acceleration structures. Whether or not it would be able to compete with the intersection times of the uncompressed acceleration structures can not be identified without testing it.

The biggest problem that results in the small average chunk fill percentage is the way in which the chunks are currently filled. As has been discussed in chapter 4.2.2, due to the breadth-first order, the tree of the acceleration structure is divided into many subtrees and each chunk can only be filled with the nodes from one such subtree. Due to us only using a single bit for the node type, however, we can not easily change the way the chunks are filled. Since the chunk structure creates a third kind of node, which is an interior node with child nodes in another chunk, a second bit for the node type would be required, so that the

three types of nodes can be represented explicitly. This would remove the deterministic way the chunks are currently filled, and would allow for arbitrary filling algorithms. This would allow us to optimize not only the average chunk fill percentage, but also which nodes are in which chunk, so that as few chunks as possible need to be traversed per ray, further reducing the memory footprint. Unfortunately, time constraints did not allow for this to be tested.

6.4 The Effectiveness of Octrees as an Acceleration Structure

If we look at figure 6.2 again, another big difference between the graph for the size of the topology of the octree compared to the same graph for the BVH is the relative size of the topology of scene *Hair* compared to the other scenes. If we were to remove the scene Hair from the graph, the data for the octree would look fairly similar to the data from the BVH, barring the fact that it has a different scale. The scene *Hair* has just a slightly larger topology than *Measure-one* and is a lot smaller than *Landscape*, when looking at the graph of the octree. For the BVH, however, *Hair* has the largest topology.

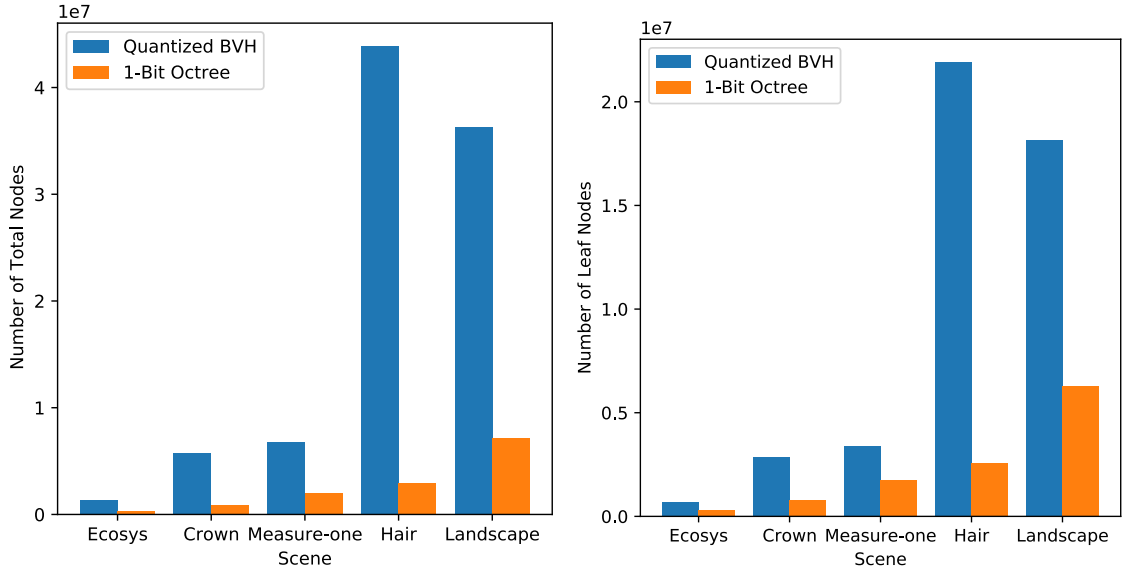


Figure 6.4: Number of Different Nodes for different Acceleration Structures

From this data, we can deduce that the octree had troubles in dividing the scene *Hair*, and made nodes into leaf nodes relatively early in the construction process, ultimately creating a relatively shallow tree structure, with a lot of primitives in the leaf nodes. This can also be seen in figure 6.4. The figure only shows the compressed versions of the acceleration structures, as these numbers are the exact same for the basic and compressed versions. For the BVH, the scene *Hair* has the most total nodes and the most leaf nodes, followed by *Landscape*, *Measure-one*, *Crown* and *Ecosys* in that order. For the octree the order is almost the same, except that *Hair* has significantly less nodes and leaf nodes than *Landscape*.

Depending on how the scene is constructed, the multiplication threshold used during the construction of the octree might limit the construction very early on, or allow for a very deep octree structure. Figure 6.5 further shows how inconsistently the octree divides the different scenes, by showing the different total primitive multiplication factor for each of the scenes. This divides the number of total duplicated primitives by the original number of primitives in the scene. In particular, the scenes *Ecosys* and *Landscape* stand out as having

an excessively large primitive multiplication factor. Considering that the scene *Landscape* has around 20 times as many primitives as *Ecosys* to begin with, this leaves *Landscape* with the most amount of total primitives by a large margin. This clearly underlines an inherent flaw of using an octree as an acceleration structure, which is its rigidity.

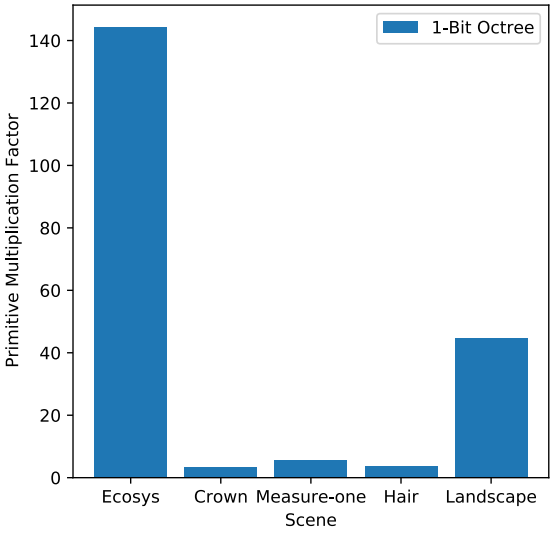


Figure 6.5: Total Primitive Multiplication Factor for the 1-bit Octree

The main problem with having so many duplicate primitives is the fact, that each of them is regarded as its own primitive during traversal, therefore resulting in the same primitive being tested for intersection several times during traversal. On top of this, while duplicating them in practice is only duplicating a pointer to the primitives, this still has a significant effect when considering the scale of the scenes. The scene *Landscape*, for example, has around 27 million primitives. Each one is pointed to by a pointer of 32 bits, or 4 bytes, resulting in a total of $4 * 27 * 10^6 = 108$ MB. With a primitive multiplication factor of around 40, that creates a total of $40 * 27 * 10^6 = 1.08 * 10^9$ primitives, resulting in $4 * 1.08 * 10^9 = 4.32$ GB of pointers. Therefore, even though only the pointers are duplicated, this creates a significant increase in data.

On the other end of the spectrum, the scenes *Crown* and *Hair* having such a low primitive multiplication factor can imply that the octree did not divide these scenes enough, although that is not a definitive conclusion. The primitive multiplication factor also seems arbitrary, and only depends on the structure of the scene, rather than the original size or complexity of it. While expected, it demonstrates the inconsistency of using such an acceleration structure, and partially explains its unimpressive performance, as it mostly either divides the scene too much, therefore creating too many duplicate primitives, or does not divide it enough, therefore not accelerating the traversal enough.

Figure 6.6 visualizes the total number of primitive intersections per pixel for the different scenes and the different acceleration structures. It becomes apparent, that while the *Embree* BVH, the basic BVH and the quantized BVH all perform similarly in this aspect, the octree struggles a lot in certain areas. For this figure, the 1-bit octree was omitted, since it represents the exact same topology as the octree.

One area that stands out as a difficulty for the octree are edges of objects. While this is a tricky area for any acceleration structure, the octree seems to have a much bigger problem with it. This is a direct result of the fact, that the AABB surrounding certain primitives are simply less precise and tight around those primitives, than the AABB of the other acceleration structures. For the scenes *Crown* and *Landscape*, the octree also struggles

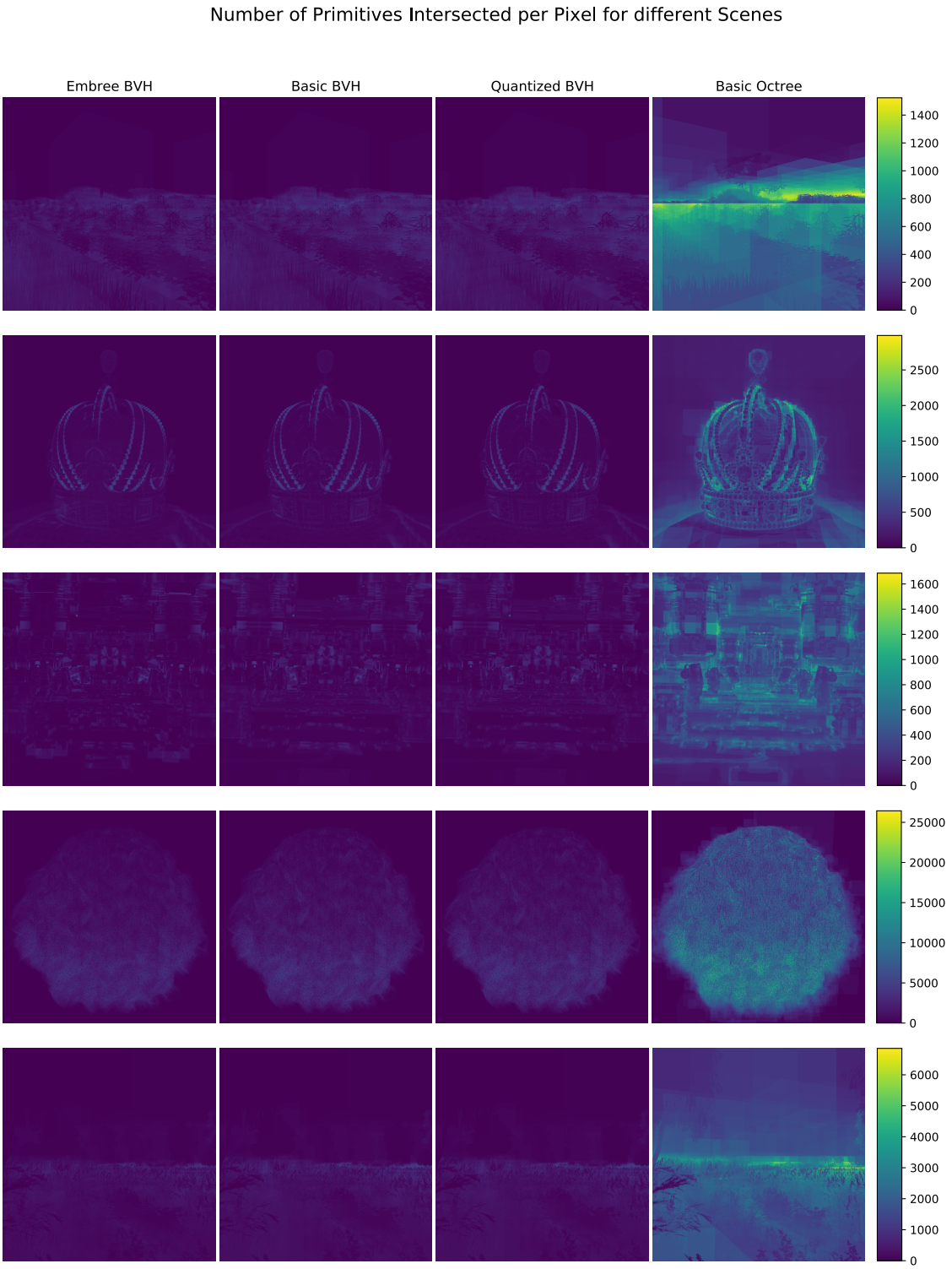


Figure 6.6: Scenes (in order): Ecosys, Crown, Measure-one, Hair, Landscape

with the backgrounds that do not contain any primitives, which are clearly brighter than those of the other acceleration structures. This is likely due to the fact, that there exist leaf nodes with very large AABBs, that contain a lot of primitives. Possibly, those leaf nodes were not further subdivided due to the multiplication threshold.

Ultimately, this data suggests that our implementation of the octree is not a suitable acceleration structure for ray tracing. It has troubles adapting to different scenes and complexities. Other times, when it divides the scene well, too many duplicate primitives are created. However, the possibility exists, that having a better deciding algorithm for when a node becomes a leaf node or an interior node might allow the octree to adapt to different scenes and complexities more consistently, while keeping the total primitive multiplication factor more under control.

6.5 Intersection Time of the different Acceleration Structures

Arguably the most important metric when comparing different acceleration structures is the intersection time. The reason acceleration structures are used is to reduce the time it takes to render a scene. An acceleration structure that reduces this time by a larger factor is generally more useful. Figure 6.7 shows the intersection time per ray for all the different acceleration structures and scenes discussed in this thesis.

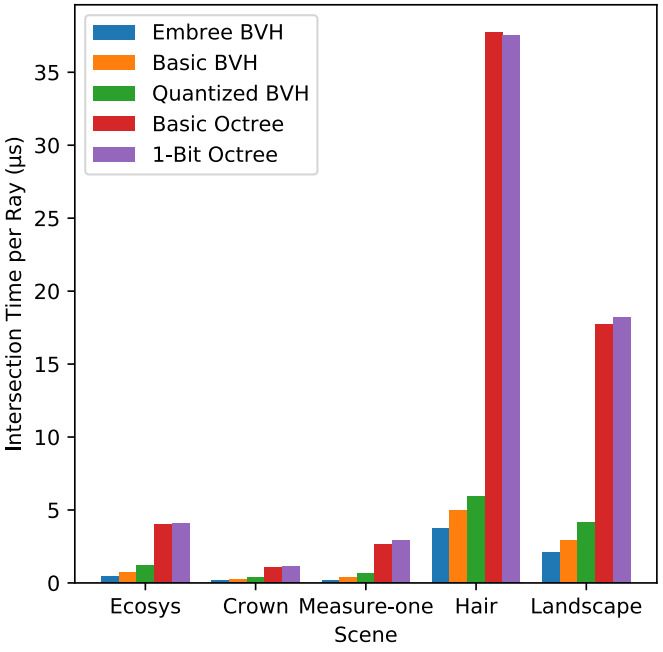


Figure 6.7: Comparing Intersection Time per Ray

As all other data leading up this point suggested, the basic octree, as well as the 1-bit octree, both have a significantly longer intersection time per ray as compared to the other acceleration structures. The quantized BVH can almost keep up, but still has a clear increase in intersection time per ray as compared to the basic BVH. However, the data also implies, that with more the optimizations of the quantized BVH discussed at the end of chapter 6.3, the intersection time could possibly be improved to match that of the basic BVH.

The *Embree* BVH is on the plot as a benchmark for a state-of-the-art acceleration structure. The BVH from [PJH16] is very close to keeping up with it, which means that, depending on

how significantly optimizations of the compression techniques can improve the quantized BVH, it could possibly even overtake the basic BVH, and have a comparable time to the *Embree* .

Ultimately, while the compression techniques clearly worked in terms of reducing the size of the topology, the specific and deterministic application of these techniques has too many problems that become apparent when looking at the data and, specifically, the intersection time. Allowing for more flexibility in the construction process of the octree, as well as the chunks for both the octree and the BVH, could potentially be worth it, even if it adds more explicit data per node and therefore per acceleration structure. Specific ideas for this are discussed in chapter 7.1.

7. Conclusion

Acceleration structures are an important tool for significantly reducing the time it takes for a ray to test a scene for intersection, ultimately allowing the photo-realistic rendering of scenes to be achieved using less computational resources. They organize the primitives of a scene into a hierarchy, allowing many primitives to be quickly skipped when ray tracing, by testing on aggregate information, such as AABBs, first.

We attempted to reduce this memory footprint by using several techniques. The primary idea was simply compressing the topology through a breadth-first bit-encoding scheme [Jac89]. We attempted to use an octree as an acceleration structure, since the AABBs of each node are implicit in the topology, thus allowing for a much more efficient compression of the data. The rank computation from [Jac89] is very expensive on large bit fields, which is why we divided the tree into smaller chunks in chapter 4.2.2, such that only small bit fields need to be consulted. Due to decisions of keeping the topology saved as a single bit per node, the construction of these chunks was very limited, which ultimately led to the chunks being filled very inefficiently, only reaching an average of about 30% of the bit fields in the chunks being filled, as seen in chapter 6.3. Even though the size of the topology, and therefore the memory footprint, was reduced significantly through this compression, the 1-bit octree performed slightly worse than the basic octree. However, through optimization of the compression techniques, for example increasing the average chunk fill rate and therefore also reducing the total number of chunks, the compression might very well be able to achieve a decrease in intersection time on top of a better reduction of the memory footprint. Some possible methods for these optimizations are discussed in chapter 7.1

Because of the shortcomings of the octree, we attempted to apply the same compression techniques on a BVH, in combination with quantization of the AABBs. We explained the basic construction of the BVH of [PJH16] in chapter 5.1. We first applied the quantization in chapter 5.2. This relative position is then saved in only six 8 bit values, rather than the six 32 bit values that are required for a full AABB. In chapter 5.2.1, we applied the same compression techniques that we already applied to the octree.

While the compression on the octree was more significant than the compression on the BVH, our implementation of it had too many difficulties for a task such as ray tracing. The compression techniques by themselves resulted in a clearly and significantly lower size of the topology, and therefore a much lower memory footprint. In terms of intersection time, however, while they showed promise, our specific implementations and choices ultimately resulted in too many problems and shortcomings, which were further amplified

when applied to the BVH. However, these problems are very obvious problems, that have clear potential to be improved and fixed. Such an improved version of the compression techniques could very well improve the intersection time to an extent that allows the quantized BVH to compete with the intersection times of the basic BVH, and possibly even overtake it.

7.1 Future Work

The most obvious way to improve the performance of the compression techniques is through filling up the chunks more efficiently, increasing their average fill percentage and, at the same time, reducing the total amount of chunks. This would, firstly, lead to a slightly better compression in terms of the size of the topology, but should lead to a better performance in terms of intersection time too, since more data is packed into a smaller memory footprint. The reason this should be strictly better is, that this better compression would be done without adding more calculations of extracting implicit data.

One way to optimize towards a higher average chunk fill percentage is to allow for more flexibility during construction. This could be achieved by adding an extra bit to each node that can be used for additional information, such as differentiating between interior nodes with children in the current or a different chunk. This would allow for flexible policies when filling the chunks, compared to the strict breadth-first policy of our implementation.

An alternative way overcome the problem of the low average chunk fill percentage is to remove the bit fields from the chunks entirely, and instead go back to storing the entire tree in a single large bit field. The only data that would be in the chunks would therefore be all the offsets they currently hold, and, in the case of the quantized BVH, the AABB of the implicit interior node. Each chunk would then be a milestone for a set amount of nodes, and could then hold only the data of how many interior nodes are in the bit field up to that point. This would alleviate the problems that applying a rank function on a large bit field has, while not wasting any bits by allocating them for nodes, but then not filling them.

These optimizations could very well fix the problems that we encountered during our implementation of the compression techniques, while not adding any new problems, as the optimizations do not change the compression itself, but rather, how it is stored and accessed.

Acronyms

AABB axis-aligned bounding box. 1–4, 6, 7, 9, 11, 13–15, 19, 26, 30–36, 38, 39, 43, 46, 48, 50, 51

BVH bounding volume hierarchy. ii, 1–4, 6, 7, 9, 30, 31, 33, 35–38, 40, 42–46, 48–51

k-d tree k-dimensional tree. 9

Bibliography

[AGL89] M. Agate, R. L. Grimsdale, and P. F. Lister, "The hero algorithm for ray-tracing octrees," in *Proceedings of the Fourth Eurographics Conference on Advances in Computer Graphics Hardware*, ser. EGGH'89. Goslar, DEU: Eurographics Association, 1989, p. 61–73.

[AK10] T. Aila and T. Karras, "Architecture considerations for tracing incoherent rays," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10. Goslar, DEU: Eurographics Association, 2010, p. 113–122.

[AW87] J. Amanatides and A. Woo, "A Fast Voxel Traversal Algorithm for Ray Tracing," in *EG 1987-Technical Papers*. Eurographics Association, 1987.

[DKB+16] B. Dado, T. R. Kol, P. Bauszat, J.-M. Thiery, and E. Eisemann, "Geometry and attribute compression for voxel scenes," *Computer Graphics Forum*, vol. 35, no. 2, pp. 397–407, 2016. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12841

[emb] Embree: A kernel framework for efficient cpu ray tracing. [Online]. Available: https://www.embree.org/related.html

[Jac89] G. Jacobson, "Space-efficient static trees and graphs," in *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*. Los Alamitos, CA, USA: IEEE Computer Society, nov 1989, pp. 549–554. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SFCS.1989.63533

[Kaj86] J. T. Kajiya, "The rendering equation," *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, p. 143–150, Aug. 1986. [Online]. Available: https://doi.org/10.1145/15886.15902

[KK86] T. L. Kay and J. T. Kajiya, "Ray tracing complex scenes," in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '86. New York, NY, USA: Association for Computing Machinery, 1986, p. 269–278. [Online]. Available: https://doi.org/10.1145/15922.15916

[KSA13] V. Kämpe, E. Sintorn, and U. Assarsson, "High resolution sparse voxel dags," *ACM Trans. Graph.*, vol. 32, no. 4, Jul. 2013. [Online]. Available: https://doi.org/10.1145/2461912.2462024

[MW06] J. Mahovsky and B. Wyvill, "Memory-conserving bounding volume hierarchies with coherent raytracing," *Computer Graphics Forum*, vol. 25, no. 2, pp. 173–182, 2006. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2006.00933.x

[PJH16] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*, 3rd ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016.

[sce] Scenes for pbrt-v3. [Online]. Available: https://www.pbrt.org/scenes-v3.html

[VMG17] A. J. Villanueva, F. Marton, and E. Gobetti, "Symmetry-aware sparse voxel DAGs (SSVDAGs) for compression-domain tracing of high-resolution geometric scenes," *Journal of Computer Graphics Techniques (JCGT)*, vol. 6, no. 2, pp. 1–30, May 2017. [Online]. Available: http://jcgt.org/published/0006/02/01/

[YKL17] H. Ylitie, T. Karras, and S. Laine, "Efficient incoherent ray traversal on gpus through compressed wide bvhs," in *Proceedings of High Performance Graphics*, ser. HPG '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3105762.3105773

Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt und von dieser als Teil einer Prüfungsleistung angenommen.

Karlsruhe, den April 8, 2021

(Marius Kilian)