Compression of the BoxTree data structure

Alexander Schwochow

Supervisor: M.Sc. Toni Tan

Bachelor Thesis
Computer Science

Computer Graphics And Virtual Reality Research Lab

Computer Graphics and Virtual Reality Research Lab
University of Bremen
Germany
October 2021

Abstract

This thesis aims to present multiple ways to compress and optimize the amount of memory a BoxTree uses. For this, different techniques will be shown and evaluated on a theoretical level. The changes most likely to be applicable, will then be grouped based on their characteristics and put into three main implementations, that in some cases get sub-implementations for comparisons. Each of these will then be tested in dept and based upon the results, use cases for each implementation will be recommended.

Contents

CHAPTER 1

Introduction

Collision detection is deeply linked with virtual reality and with that graphics as a whole. Everything, that tries to simulate a virtual world, has to simulate the collision of objects, to give it a realistic appearance. Otherwise objects could just clip into each other and any kind of gravity would result in objects falling indefinitely. This does not just apply to virtual worlds used for games, or simulations, this is also applicable for real life processes, like CNC milling [1], or robots [2] that require moving parts not to collide. The exact requirements in terms of resource usage and performance can thus differ for each use case.

Overall, the idea of collision detection is to see, if any of the polygons, consisting of a few points of one object, are within a polygon of a second object. Checking this manually for every possible combination of polygon-pairings can be incredibly inefficient. For this reason, a more performant concept, called Bounding Volumes was developed. A Bounding Volume is simple geometric object, like a boxes, or a spheres, that wraps around every collision-object, most commonly splitting into gradually smaller Bounding Volumes, forming a Bounding Volume Hierarchy. When checking for collisions, it is then possible to check if the biggest Bounding Volumes

intersect and then proceed accordingly, by either stepping down to the finer layer of Bounding Volumes, checking the underlying polygons, or immediately returning the result. This might seem, like it is just an extra step to get to the polygons, but it can massively speed up the process by eliminating all polygons inside of Bounding Volumes, that are not overlapping. As such, this approach has become very popular for collision detection. [3]

1.1 The BoxTree

For this thesis, the Bounding Volume and corresponding algorithm used is the BoxTree. As the name implies, it uses boxes as its Bounding Volumes. It starts of with one box around the entire object, that recursively splits of into two new boxes, until a box only spans one polygon. At that point, the box stores information about the polygon, that is required to do polygon intersection with them. This type of structure basically represents a binary tree, with the normal boxes, being nodes and the polygon boxes being leafs. [4] There is a bit more to this, but details will be added to this, when needed.

1.2 Motivation

Typically the main concern, when it comes to collision detection, is the performance of the algorithm. Having the required result available for the next frame of a video game, or other similar requirements, are typically more important, than the memory usage, which can often be neglected. However, there are use cases, where these requirements can be flipped, or weighted more similarly. As an example, in the IoT department, you can imagine vacuum, or service robots calculating paths for their movement, to avoid hitting other objects and robots. For such a case, typical performance requirements don't necessarily hold, because it does not make to much of a difference, if a collision can be found a few milliseconds fewer or later. Another factor, the production cost on the other hand, can be crucial. Being able to reduce the amount of installed RAM can reduce the price to produce these machines, which

either results in higher profit margins or lower costs to the consumer.

Even usually performance oriented domains, like gaming could profit from this. Modern game consoles don't have dedicated video memory, instead using the same physical RAM as graphics and system RAM. [5] Reducing the RAM used for collision detection, can thus allow the memory to be used as V-RAM and enable higher quality, or just more textures. Especially, since the limiting factor for visual fidelity is the GPU, which gives the collision detection on the CPU a bit of room to work with. As such, the main focus of this thesis will be to reduce the memory footprint as much as possible, with a second focus on trying to remain as close as possible to the original performance as possible.

CHAPTER 2

Structure optimizations

The original BoxTree implementation is focuses on speed and is not optimized to yield the lowest possible memory footprint. As a result, it is possible to gain significant memory gains, by changing key parts of the original structure.[1]

2.1 The original structure

As explained in the Introduction, the BoxTree consists of boxes, that encompass some object. These are implemented more or less straight forward, with each box being a separately allocated heap object, that represents either a node, or a leaf. Nodes store the information about how and where the next boxes are positioned. This is represented with two floats and one integer for the plane, in which to cut. In addition, nodes store two pointers to the next boxes, for which a Null-pointer represents an empty sub-box. What an empty sub-box is and why it is useful will be explained later. Leafs only need to store the information about the corresponding polygon. For this, they store the index of the polygon, as well as 4 possible indexes of

[1] The implementation referenced can be found at `https://gitlab.informatik.uni-bremen.de/cgvr/CollDet`

the points, that the polygon consists of. The actual 3D point-coordinates are stored in the geometry, so a pointer to this array will also be stored. With the pointer and the indexes it is then possible to read the coordinates when needed. A leaf also has an integer for the cutplane, however it is not used as such. To differentiate between nodes and leafs, as both are just BoxTree objects with a union for both sub-types, the upper bits of the cutplane are used to mark the type.

With every number being represented by 4 bytes, this should theoretically add up to 32 bytes. However in practice, there is also an unused virtual destructor, padding, as well as the decision by malloc to just allocate a few more bytes, then necessary. The last of these is technically implementation specific behaviour and may not happen with every compiler, or Standard-Library, however it is a noticeable side effect of this style of implementation and reproducible on multiple machines. This over-allocation has been found by observing an unexpectedly high overall memory usage and has been confirmed by the malloc_usable_size() function, that prints the actually allocated bytes per allocation. All of these increase the size of a BoxTree box by almost a factor of two, resulting in a size of 56bytes. The resulting BoxTree box looks like this:

BoxTree			
virtual destructor	8		
cutplane	4		
padding	4		
Union			
Node		Leaf	
left-node ptr	8	polygon-index	4
right-node ptr	8	padding	4
cutp	4	4x point index	16
cutr	4	points ptr	8
padding	8		
malloc overhead	8		

Table 2.1: Layout of the Boxtree on a x64 system with sizes in bytes

2.2 Moving constant members

Every BoxTree leaf, of the same geometry, stores the same 64 bit pointer to the points of its geometry. This is required, as every original BoxTree object, must hold all the information necessary for a check, against another BoxTree. To save the wasted space for these duplicated pointers, a header is added to the overall BoxTree. This header will act as the first point of contact, when checking two BoxTrees against each other, which allows us to only store constant information once and pass them to the checking method, when collision detection is done. In addition to saving already existing information, this also allows us to abstract away from the original way the BoxTree was implemented, as long as a top-level interface is presented, that behaves, like the original.

2.3 Indexing

As described at the start, the BoxTree is based on allocating every node and leaf, as a separate BoxTree heap-object. This requires every node to store two 8 byte pointers to the next BoxTree, as well as wasting 8 additional bytes for every allocation. To stop malloc from wasting memory, the BoxTree boxes have to be stored in an array, or vector. This also allows to just store 32bit indexes into this array, instead of 64bit pointers. The resulting array will be stored in and read from the header.

2.4 Merge leafs

In addition to minimizing the size of references inside of BoxTrees, it is also possible to reduce the number of references them self. To do so, the leafs will be merged with the nodes above them, removing the need to store references to them. This results in four different types of nodes, depending on the amount and placement of the merged leafs.

2.4.1 Minimize Leafs

As explained earlier, the BoxTree stores a pointer into the original geometry used to create the BoxTree. This pointer points to an array of 3D coordinates, or points of the geometry, that each polygons refer to using its indexes. These polygons are once each in the geometry, but also copied into the leafs, as every leaf represents a polygon. There is however also the index of the polygon in each leaf, that refers to the polygon in the geometry. With this, the whole polygon can be read at runtime from the geometry, as long as the pointer to the polygon array in the geometry is stored. This is basically the the same way the original implementation reads the points, that also just get read dynamically. The only difference between the polygons in the geometry and the ones in the leafs, is that during construction the last of the indexes in the leafs would be marked, when the polygon is a triangle. This information can however also just be read from the polygon pointer, reducing the size of a leaf to just the one polygon index-integer. As nodes now either have an index to refer to another node, or refer to a polygon with an index, these two can just be represented by one union, resulting in the same size for all node-types. For this to work, the type of the index has to be stored, which will be done in the cutplane integer, that is not fully used. Having the same size allows the use of only one array to hold all nodes, without any space wasted for padding.

2.5 Cuts

The only part of the BoxTree, that has so far has not been optimized is the cut-information, that every node contains. This consists of the cutplane orientation (x,y,z) and two floats, that mark the position of where the box is split. As this information is exactly the new information, that is obtained, when building the BoxTree and that is definitely needed for every check, it is considerably harder to optimize. The cutplane could be shrunken down to just one byte, however due to memory alignment and possible padding, optimizing this properly will be done in the next chapter.

The best strategy to compress the floats is dependant on the type and size of the geometry. As such, multiple options were implemented and evaluated. The easiest option is to just use 16 bit floats. These will be converted from and to 32 bit floats, when they get stored and read, as working with anything else, then 32 or 64 bit floats is generally not intended. Converting from 16 bit floats uses just 3 instructions[A.5] and otherwise remains at the same performance level. A more sophisticated option to store the floats will be described in the next chapter.

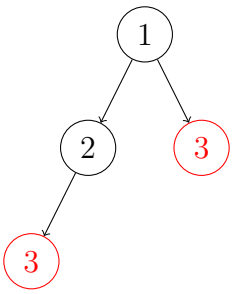
2.6 Automatic index calculation

The main part, that is remaining, are the indexes to the next nodes. These have already shrunken to halve their size, it would however be nice to eliminate them completely. For this, the children would have to be placed in a predictable way, so that their index can be calculated based on the index of their parent. The first idea would be to just store the children of a node directly behind the node.

$$index(child_{left}) = index(parent) + 1 \tag{2.1}$$

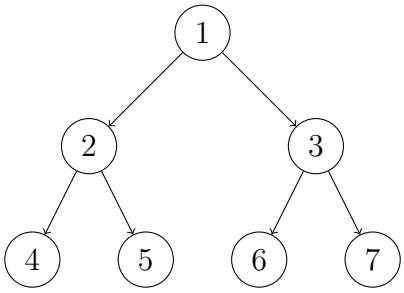
$$index(child_{right}) = index(parent) + 2 \tag{2.2}$$

Sadly this does not work, as the children of the first child would have to overwrite the second child.



This means the children have to be stored at an index, that is guaranteed to not be used by another child. The way to do this, is to double the parents index and store the two children there. To hint at the correctness, one could just imaging

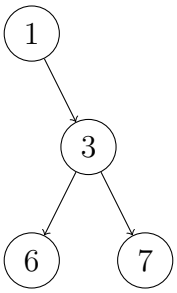
enumerating a full, complete binary tree from top to bottom, left to right.



$$index(child_{left}) = index(parent) \times 2 \tag{2.3}$$

$$index(child_{right}) = index(parent) \times 2 + 1 \tag{2.4}$$

The problem, that this approach has, is that it can only efficiently store full, complete binary trees. If the binary tree is not full, or complete, this method still has to allocate the memory for a theoretical maximum. As an example, take a look at this binary tree:



Even though node 2 is not present, the indexes of the children of node 1 still have to start at 3. This continues down to the next level, where indexes 4 & 5 remain unused. To store this, an array of size $7 \times$ (node-size) would have to be allocated, despite only four nodes being present. For actual geometries, this tree would obviously extend dozents of layers down, effectively wasting half of the array, assuming the right part remains full, complete and extends down to the same level. These properties are also called a perfect binary tree. To estimate, if this approach could save memory, only the lowest depth of a BoxTree from typical geometries has to be determined to calculate the size of a perfect BoxTree, which would have to be allocated for. This can be done using the formula [6]: $2^{depth} - 1$ and would result in the following table,

assuming the size of a node shrinks to 8bytes:

	max-depth	slots	used slots	current size	new size
ATST-4252	31	2.15×10^9	7419	116KB	17GB
Castle-14871	41	2.20×10^{12}	25011	391KB	17TB
Raptor-400000	43	8.80×10^{12}	772783	12MB	70TB
ATST-152944	54	1.80×10^{16}	266924	4MB	144PB

Table 2.2: Size calculation for using implicit indexing

The resulting size would be orders of magnitude larger, than the current implementation, with most of the array being empty. This makes sense, as the BoxTree uses implicit empty boxes. Typically a box splits into two Boxes with roughly the same amount of polygons. In many cases, it can however be faster, to just cut of an empty part of the original box and store this as an invalid index for one of the children. Any object, that collides with this empty child, can immediately return, instead of otherwise traversing down continually halving boxes. Each of these however adds one level to the binary tree, that continues with just one child. This is basically the worstcase for this method, as the the size estimates show. Using a map to avoid the cost for empty slots might seem sensible, however this map would have to store the index as a key, negating the attempt at removing them.

There exists another solution though. As seen before, placing both children behind the node, results in overlapping placements. Placing just one child behind a node is viable though, as long as that child is allowed to place its own children, before the index for the second child is determined. This means only one index would need to be stored. That does however result in another problem. Currently, each node of the BoxTree is the same size and as such, they can be stored in just one array. If only one node-index would be stored, the size of only some node-types would decrease. More specifically, only when the node-index is not reused as a polygon index it can be removed. That would mean, the nodes would have to be stored in different arrays, as otherwise they would just have padding up to the largest type

in the array for every node. Different arrays would however mean, that the same indexes cannot be used across the whole BoxTree, effectively invalidating this whole approach.

2.7 Overview

All of these optimizations result in a new BoxTree, that is 16 bytes in size, which means about 71% smaller, than the original to represent a single node. In addition, the final size is memory aligned and can fit multiple nodes into a single cache line, without overlap. A cache line typically represents 64 bytes of consecutive memory, that gets read and more importantly for the purpose of performance, cached as a whole, whenever any point in memory is read [7]. The exact layout of the four resulting BoxTree types is as follows[2]:

	Cutplane	1. Type	2. Type	2x Indexes	2x 16b floats	Sum
Two leafs	4	-	-	8	4	16
Left leaf	4	0	-	8	4	16
Right leaf	4	-	0	8	4	16
No leafs	4	0	0	8	4	16

Table 2.3: Overview of the remaining BoxtTree members with sizes in bytes

[2]The new implementation can be found at: `https://gitlab.informatik.uni-bremen.de/cgvr/CollDet/-/tree/boxtree-compressed`

CHAPTER 3

Dynamic compression

The aforementioned reduction in size, has been achieved by minimizing the amount of variables per box, resulting in some changes to how the BoxTree has to be traversed, but no change to how each box has to be read. This means performance overhead is minimal, whilst achieving big improvements for memory usage. It is however possible to compress this structure.

3.1 Traditional compression

A simple solution would be to use a typical compression algorithm, to compresses the entire BoxTree after building and to decompress it, when needed. This is possible, because nodes don't use pointer based referencing anymore, instead relying on index based referencing. With this, it is possible to move and compress the BoxTree in any way, that is needed, without worrying about bad pointers. Having the data in arrays also means, that implementing such an approach can be quite simple, as this is a well known use case for compression and has such been implemented in libraries like the LZMA SDK, which the 7z format uses for compression. [8] This approach in general has a few advantages, as well as disadvantages:

Pros	Cons
Dynamic compression rate, based on the properties of each BoxTree	Requires decompression before starting to check for collisions
Decompressed result can be reused for tests against other Boxtrees	The entire BoxTree will be decompressed, even though only a small part, might actually be necessary
Tried and tested existing implementations	Will have a higher memory footprint during and after decompression, than not compressing at all

Table 3.1: Pros and cons of traditional compression

The best use case for such an implementation, seems to be a system, where collision detection is only required for a small, rarely changing selection of distinct BoxTrees at any given time. You could imagine a sandbox world, in which a user is able to choose between a broad selection of items to spawn in and interact with, as a possible scenario.

This is however not the typically expected use case and in conjunction with the possibility, to use even more RAM during checking, than before, this option was not chosen to explore further.

3.2 Domain specific compression

To achieve a more preferable result, decompression should be possible at the per node basis, whilst still being able to adapt to the properties of each BoxTree. One of these properties, that is known at the start of the construction, is the amount of polygons, that the geometry holds. Because every leaf holds exactly one polygon, the amount of polygons is equivalent to the amount of leafs. With this information, the size of a full tree and with that the maximal amount of nodes in the tree can be estimated. Both the polygon index in the leaf, as well as the node index in nodes, have so far been represented with an 32 bit Integer. In most cases, this is way more, than is ever needed, so the idea would be to just shrink down the size of the

datatypes.

One option would be to have different pregenerated versions of the nodes, with different sizes (int, short, byte) for these indexes, that get chosen at runtime. This could be implemented with inheritance from a base-node, to keep changes to the collision algorithm relatively low. Figuring out the concrete type of this node, would be the only performance downside, apart from some minor inefficiencies, if the structure has to be packed using "#pragma pack" to prevent padding. There is only one downside to this, in terms of memory usage. As an example, if the required bits for the two node indexes, as well as the two polygon indexes are determined to be 9 bits, each of these would get 2 bytes, resulting in 4 bytes for indexes per node. Optimally, only $9 \times 2 = 18$ bits $= 3$ bytes would be necessary. Recall from the optimization phase, that the cutplane only has to represents 3 different values, which means 2 bits. Typical data structures would have to store this in at least one byte, whilst in our example it could still fit inside the already existing 3 bytes ($9 \times 2 + 2 = 20$ bits $= 3$ bytes).

To handle bit-sized datatypes, instead of byte-sized ones, C++ offers so called Bit Fields. The size of these however have to be determined at compile time. This means, to achieve optimal results, these would have to be pregenerated for every possible combination of sizes using constexpr, macro combinations, as well as template compiling every function to use all possible versions, or combination of versions, when two BoxTrees are expected as input. It is unclear however, if the preprocessor would be able to handle this massive amount of classes and functions. In addition, the size of the generated binary would be massive, whilst understanding what the code does, would probably be an order of magnitude harder, than before, as for example, choosing the correct function would probably have to be done using a multidimensional function pointer array. Not to mention, that all of this complexity would be a performance overhead in and of itself. Because of all of this, a simpler approach was chosen.

3.2.1 Runtime size handling

Instead of preparing every possible variation at compile-time and then choosing the correct version of code to use, it is also possible to write the code to dynamically handle different sized objects. To achieve this, the size of each variable in bits, has to be stored inside of the BoxTree. When trying to read a variable from a node, the offset of this variable has to be calculated. This means summing up the sizes of all variables before the requested one. If the offset is larger then 8, the pointer is going to get moved one byte, until the offset is smaller. Then, a chunk of data is read, that is larger, than the requested variable. This chunk is then shifted, by the remaining offset and masked.

All of this seems like a lot of work, but in practice this can be implemented quite efficiently into a single line of code, assuming the offset is known:

```
auto value = (*(POINTER + OFFSET / 8)) >> (OFFSET % 8) & MASK
```

In a modern compiler (Clang 11), this compiles to just five assembly instructions:

```
mov     eax, esi
and     eax, -8
and     sil, 7
sarx    rax, qword ptr [rdi + rax], rsi
and     eax, edx
```

This leaves the offset and mask to be determined. The mask can just be calculated once during construction and then just read, when needed. For the offset, a similar strategy could be implemented. If the variables would be read one after another however, the size of the last variable can just be added to the current offset, adding just one instruction of work to get the offset. Both approaches result in the same performance, so the second one was chosen.

A slightly modified version of this offset reading can also be used to write to any position. As the performance of the construction is not the main focus of this thesis, this part is not explained in detail. It should however be noted, that despite no at-

tempts at performance optimizations on that part, the build time of the compressed BoxTree is in some cases even slightly faster, than the original. This is due to the original BoxTrees reliance on heap-objects, the allocation of which takes longer, than compressing and writing into an array for the compressed BoxTree.

Overall, this results in a relatively performant way, to decompress nodes, that were compressed to the almost lowest possible amount, whilst keeping the same size for all nodes of the same type. In theory the last of these requirements could be dropped for nodes, that have a small indexes. By storing the size information about each node, just like the size of each entry in an ext2 directory [9], each node could have the minimal possible size. This would however require to traverse the entire array, node by node, to find a node with a specific index. In addition, this would require an even more branched and complex decompression functions, so the expected improvement does not seem proportional to the additional performance overhead. With different sized nodes, the nodes also have to be stored in different arrays, so the type flag is now used to describe the type of the next node type, not just to differentiate between polygon and node index. That also means, that nodes no longer know their own type, but as long as that information is passed on from the parent, this is not a problem. All of these compression-changes result in the following BoxTree:

	Cutplane	1. Type	2. Type	2x Indexes	2x 16b floats	Sum
Two leafs	2	-	-	TBD	32	34+?
Left leaf	2	2	-	TBD	32	36+?
Right leaf	2	-	2	TBD	32	36+?
No leafs	2	2	2	TBD	32	38+?

Table 3.2: Overview of the compressed BoxtTree members with sizes in bits

3.2.2 Float improvements

During the Structure optimizations of the BoxTree, the floats have been reduced to 16 bit each. This may not be wanted, if the full precision is desired. In addition, the library even supports a high precision mode, that utilizes doubles (64 bit floats). Going from this explicit wish for precision to 16 bit seems like it may not be the

expected behaviour. This means another way to reduce the size taken up by the floats has to be proposed.

Fortunately, when analyzing the distribution of these floats, a property has been observed, that can be used to do so. Especially for simple geometric shapes, like spheres and boxes, these floats are not all distinct values. This makes sense, as the same form of polygon-placement repeats multiple times in these structures. Even for geometries, that have a more sophisticated structure, like a model of a raptor with 400000 Polygons, only ∼8% of floats are actually unique. This figure can rise depending on the properties of the geometry. Generally speaking, the more polygons, the lower the percentage of unique polygons. As such, geometries like an ATST with 4252 Polygons has 70% unique floats, while a more detailed version with 152944 Polygons only has ∼28%. Testing this on a broad selection of different geometries shows, that an average of only around 10-60% of floats are actually unique.[A.1]

It has to be noted, that this property only holds for the individual floats, not the pairs of floats, that are being stored in a node.

This opens up the option to store the distinct floats in one place and then only reference these with an index. Yet again, the size of this index can be dynamically estimated, which can reduce its size to just a few bits in most cases. There are however two worst cases, that may lead to this approach exceeding the original size. Either the geometry is structured so unique, that most floats are distinct, or the geometry is so large, that the size of the index is so large, they overshadow the reduction from deduplicating the floats.

These problems could be mitigated, by changing the building process, to a two step process. The first step would calculate all the values, which would be used to build the BoxTree. The second would determine the actually required bits for all variables and choose the optimal compression strategy for the floats. At this point however, it is basically just the same idea, as compressing an existing BoxTree, as all the values

have to be stored somewhere, while it is being built. This could have its applications and is a nice point of comparison, but the substantial additional memory and time overhead during construction seem to make this less usable in actual applications.

	Cutplane	1. Type	2. Type	2x Node Indexes	2x float Indexes	Sum
Two leafs	2	-	-	TBD	TBD	2+?
Left leaf	2	2	-	TBD	TBD	4+?
Right leaf	2	-	2	TBD	TBD	4+?
No leafs	2	2	2	TBD	TBD	6+?

Table 3.3: Overview of the deduplicated BoxTree members with sizes in bits

Overall, this compression leaves only the bare minimum of the BoxTree to be constant in size, allowing for incredibly small base-nodes. This reduction will of course be reduced, when taking into account the size of the float-array, but having nodes the size of only one integer for very small geometries is a massive drop nonetheless.

CHAPTER 4

Memory savings

The previous chapters have described the optimizations and compression used to minimize the size of the BoxTree. Simple reductions in the size of each box, could be viewed as a reduction in the overall size of the original BoxTree. There have however been changes, that make this calculation more difficult. First of all, moving the leafs into the nodes, have reduced the overall amount of nodes, as well as having created three different types of nodes. These have then been gotten custom sizes, during compression, depending on the size of the BoxTree. In addition, the properties of the geometry can have a strong influence on the deduplication of floats. For all these different combinations, the effects upon different models thus need to be evaluated in order to estimate the actual memory savings. This will be done in this chapter.

In addition to the BoxTree, the library, that has been modified, also implemented a so called DopTree to handle collision detection. This has however shown to already be worse in both performance and memory usage, when compared to the original BoxTree implementation and has as such not been used as a point of further comparison.

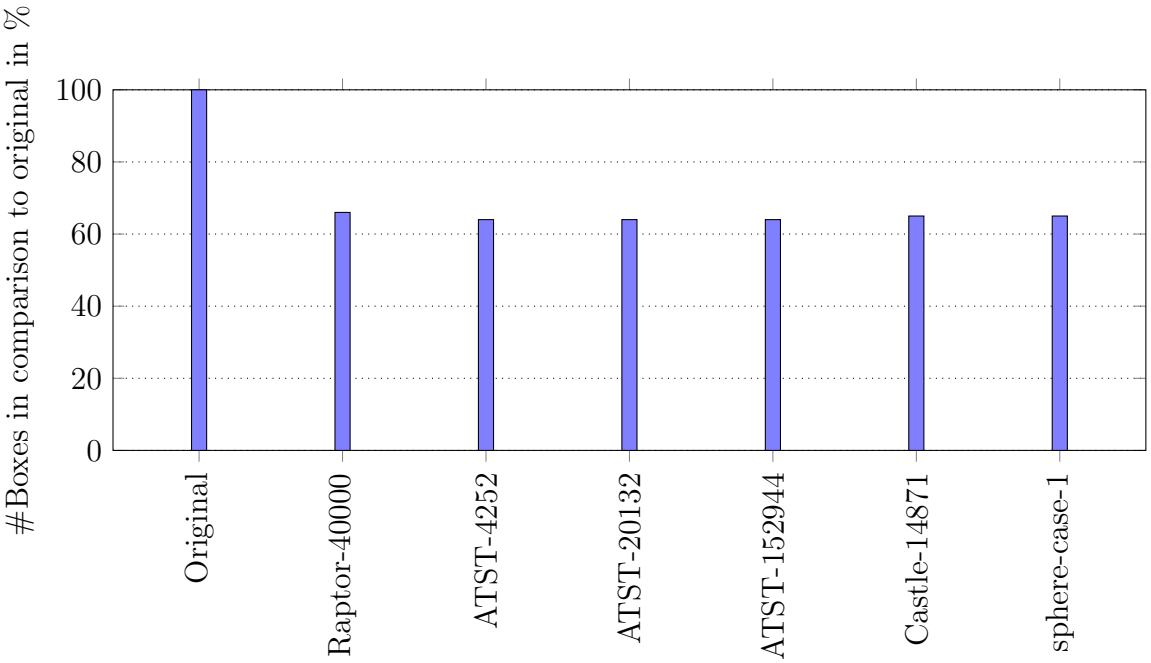
Testing the overall memory usage with external measuring tools has resulted in in-

accurate and varying results. Especially already freed resources, like reallocated arrays, would often not, or only at a random later time actually be given back to the system by malloc. As a result, the actually allocated sizes have been calculated within the program and printed out for these results.

4.1 Optimized BoxTree

To begin with, the results from merging the leafs into the nodes will be shown.

Figure 4.1: Effects of merging leafs into nodes [A.2]



Despite having a wide range of models[1] and levels of detail for the same geometries, the ratio of original leafs to inner nodes is almost the same across the board. This is surprising, as the generated binary trees are not at all complete, nor full and thus generally not easily predictable. With this percentage and the ratio of original, to optimized node size, the expected size of the optimized BoxTree, can be relatively

[1]The models can be found here: `https://cgvr.cs.uni-bremen.de/research/colldet_benchmark/` or in the original implementation itself. The new models did require conversion using [10]

trivially estimated.

$$Size_{Optimized} \approx Size_{Original} * 0.64 * (16/56) \approx Size_{Original} * 0.18 \tag{4.1}$$

This means, even before doing any compression, the size has already been reduced by around 82%. Recall, that this optimized version, uses 16bit floats, which is a divergence from the original. Storing 32 bit floats will result in almost the same amount of reduction however.

$$Size_{Optimized} \approx Size_{Original} * 0.64 * (20/56) \approx Size_{Original} * 0.23 \tag{4.2}$$

Using these two results as the new baseline, the geometry dependant compression can be compared.

4.2 Compression

As noted before, the size of the indexes will be estimated during construction, to ensure a fast and memory efficient build-process, not necessarily the optimal results. To see, how close this approach is to the optimum, the optimal values for each model have been manually determined and tested. This should give a good indication, of how good this approach is in practice.

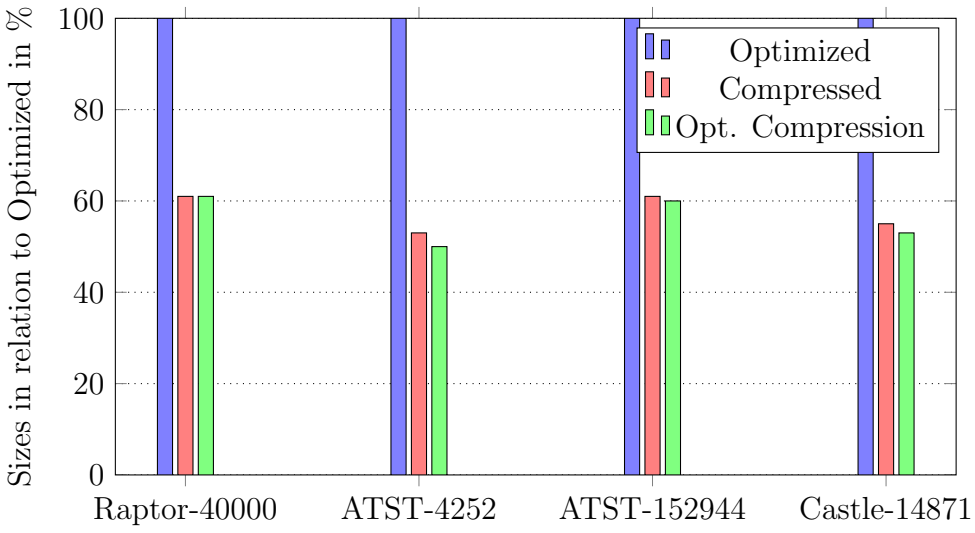


Figure 4.2: Memory results of dynamic compression [A.3]

This shows, that the size of the BoxTree has been almost halved yet again. Almost independent of geometry, or polygon count, this reduction comes out to 42% on average. Compared to the original, this means a ∼90% reduction. In addition, it is visible, that the estimated compression is almost on par, with the lowest possible compression. This can be attributed to the ratio of leafs to nodes of the original BoxTree. With about 64% of all boxess being nodes and 36% leafs, this means the amount of compressed boxes is around double the amount of leafs. The amount of leafs is however just the amount of polygons, thus giving a good estimate for the amount of compressed nodes of:

$$\#CompressedNodes = \#Polygons * 2 \tag{4.3}$$

To ensure a one-pass build-process, the constant has even been increased to 3, for the actual implementation, that has been tested, in case of a theoretical worst-case model. With this, the size for node indexes can be calculated with maximal 1-2 unused bits, thus giving us almost optimal compression.

4.2.1 Deduplication of floats

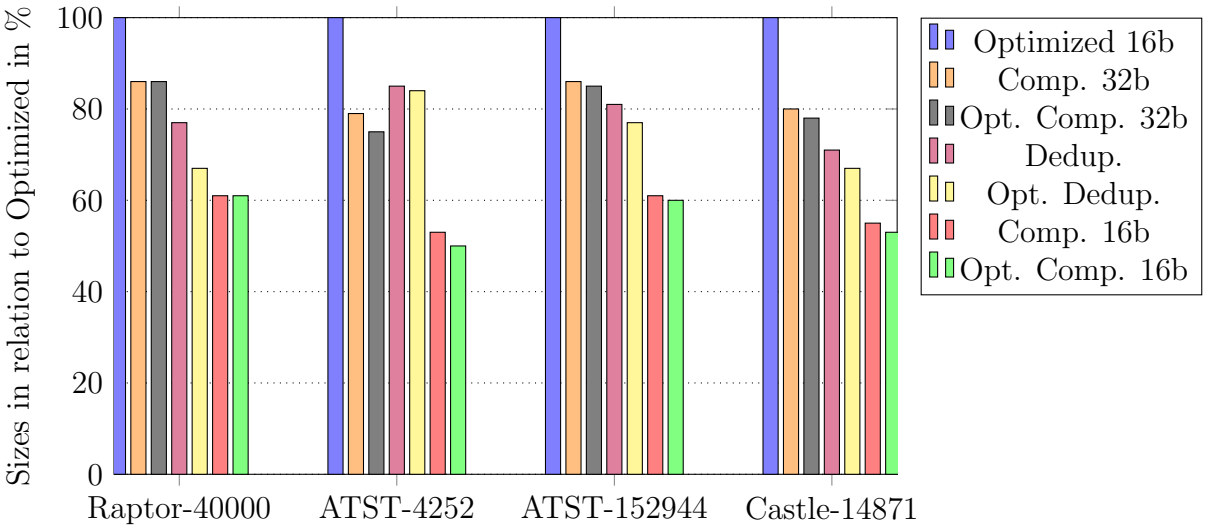
So far, the compression has always been expected to result in lower sizes and only the range of this has had be be determined. With the deduplication of floats, this is not the case any more. By moving from floats, to indexes to floats, the reduction is dependant on both the size of the indexes, as well as the amount of unique floats. Just as with the node-indexes, the size of the indexes will be estimated to expect the worst case. This means every node could have two unique floats, and thus this has to be expected:

$$\#MaxUniqueFloats = \#CompressedNodes \times 2 \tag{4.4}$$

With this equation, the size of the float-indexes already matches the size of the 16 bit-floats with just around ten thousand polygons. Combined with the stored floats, it should therefore not be expected, that the compression matches that of the

previous version. As the float array stores actual 32 bit floats, in comparison to the 16 bit floats, this would also not really be a fair comparison. When comparing node sizes to a compressed 32-bit float version, around 716 million polygons would be necessary to reach that size for the deduplicated version. For any geometry, that is smaller, which should be almost always the case, the node-size can thus be reduced in comparison to the simple version and depending on the uniqueness of the floats, even as a whole. To see how much this loses by having to estimate the amount of unique floats for index sizes, a ideal version has also been tested and plotted against the previously mentioned approaches.

Figure 4.3: Comparison of all different compression variants [A.4]



This time, the different geometries make a huge difference. For the raptor and the Castel models, the size of the deduplicated BoxTrees are smaller, than just having the 32bits in the node and larger, than storing 16 bit floats. The BoxTree for the small ATST in comparison exceeds the simple approach for storing 32 bit floats. Increasing the detail of that model, makes the compression worth while again. As noted in the last chapter, this can be explained by the percentage of unique floats. Unsurprisingly, the small ATST model has the highest at 69%, with the others at around 8-34%. This suggests, that this approach is only worth while, when dealing with large geometries, even though the size of the indexes gets reduced the most for small geometries.

4.3 Conclusion

Switching between deduplication and in-node storing depending on the geometry size, should be the approach to take, when 32bits of precision are needed and the goal is maximal space efficiency. As this could be done by just one branch in the build and collision check functions, that lead to different implementations, there should be no performance overhead from the switching itself. For general use cases, the 16 bit compressed variant remains the most space efficient. When build time is not a relevant factor, the calculations for the optimal variants of both approaches can also be done after the building of a temporary BoxTree, that is then compressed in the optimal way and with optimal index-sizes. As the memory usage during the build-process already exceeds the actual BoxTree size by a factor of 10, this is not unrealistic in terms of extra memory usage.

CHAPTER 5

Performance overhead and optimizations

Multiple options to compress and optimize a BoxTree have been shown, as well as their effects on memory usage. Now the question becomes, with what performance penalties, these improvements have been gained. With that information, it should be possible to determine the overall efficiency of these approaches and a recommendation, when each could be used, can be given.

5.1 Testing Methodology

In comparison to measuring or calculating memory usage, measuring performance is a more complex undertaking. Especially for Collision Detection, as the used geometry, the distance between objects and the rotation can have a huge influence on the result. To tackle this problem, there exists a set of predetermined distance and rotation values, that test different aspects of collision detection, upon a predetermined collection of models. [11] These values and geometries have in the following tests been used to give the best representation of the performance differences. [1]

[1]For this purpose, the original authors, have supplied a benchmarking program. The time commitment to get this working, greatly exceeded the time to re-implement the main functionality though. As such, the official way of handling these values has not been used.

To ensure correct results, multiple runs of each test have been done. The first of these has always been discarded, as it generally varied by a bit from subsequent runs and the time of the remaining ones have been averaged. For determining more specific performance metrics, perf has been used to monitor detailed program statistics and performance on a per instruction level.

During the runs, no other programs appart from the OS have been running on the System[2] to ensure a controled environment. In addition, a modern linux kernel 5.15-Manjaro and compiler (GCC11.1) on "-O3 -march-native" optimization settings have been used for realistic, modern-day performance setting.[3]

5.2 Optimized BoxTree

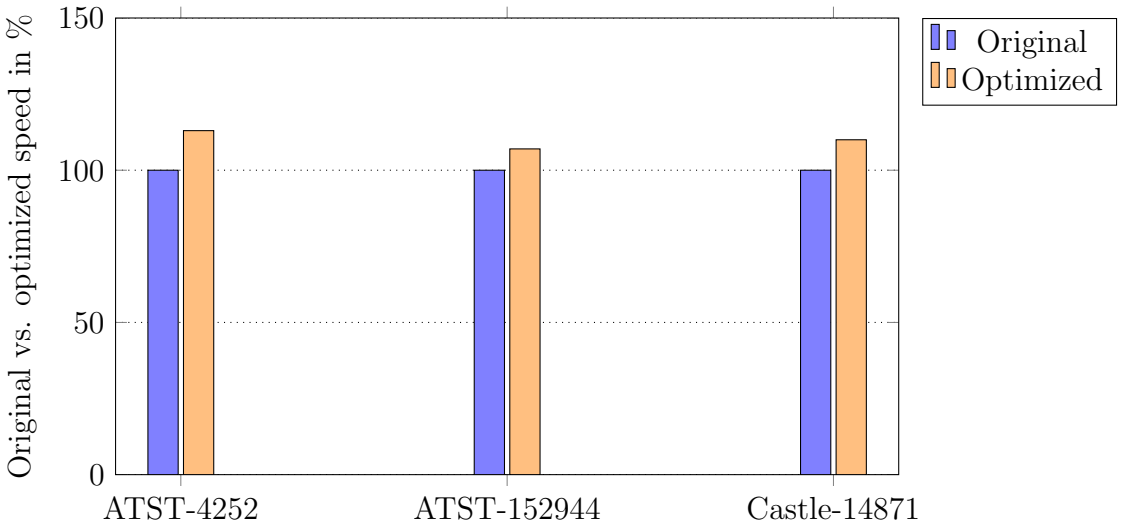
The optimized BoxTree implemented the changes, most likely to result in only minimal performance penalties. The biggest of these, can be expected to be the removal of the polygons from the leafs. To read these, they have to be read from the polygon vector in the geometry. This means a read from the already present node is traded for one to somewhere into the RAM. Converting a float from 16 to 32 bits is only 3 instructions using Intel Intrinsics, but these also add up, over the entire runtime. Moving constant information into a header, is almost free, as this information is almost certainly going to be cached after the first node. Having nodes in an array might even gain performance, as having all nodes in one place, multiple nodes can be on the same cache-line. That means the chance is high, that the next accessed node is already in cache and can be accessed considerably faster, than normal reading from memory. To maximize this, the first child of each node is placed right after that node. The checking function also starts checking that child, when possible. With a node-size of 16 and a cache-line size of 64 Byte, it not thus possible to have

[2]System: Intel i5-8250U, 8GB RAM

[3]The default Ubuntu Compiler GCC9.3 on -O2 optimizations massively favored the new implementations, as it was unable to optimize a slow loop in the original away. Because of this, the most recent and most optimized versions of everything have been used.

the three next nodes cached, by just reading the first.

Figure 5.1: Comparison of original speed vs. optimized [A.6]



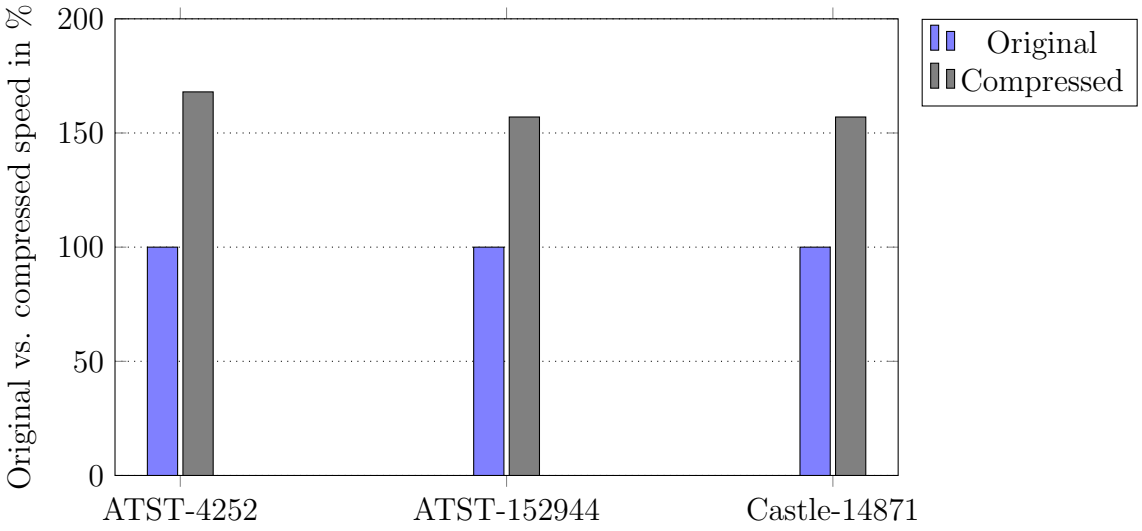
Despite the possible improvements, the memory optimized version is still around 10% slower, than the original. As expected, the average reduction in speed can be traced back to the converting of floats and accessing arrays, with each taking about ∼3-5% each of the total runtime. When it comes to caches, the optimized version has overall less cache-misses, than the original, despite the extra read into the geometry. This hints at the fact, that the multiple nodes per cache line do get extensively used and help to keep the overall performance relatively similar. [A.7] Despite not being able to catch up to the original implementation, this performance loss is minor, compared to the aforementioned ∼82% memory gain.

5.3 Compressed BoxTree

When applying compression on top of that, the performance loss can be expected to be considerably worse. As noted previously, the just about ∼12 instructions per check, necessary for converting all floats of two BoxTree Nodes in the optimized version, resulted in a ∼3-5% slowdown. Every read from the compressed BoxTree will result in at least 6 decompression instructions, which can quickly become incredibly costly. To reduce these effects to a minimum, all the decompression is handled at

the start of the comparison between two nodes. This way, every member only has to be decompressed once and can then be used like usual for the rest of the check. In addition, when reaching the leafs, the decompressed node-information is passed on to the next function call, that can reuse the data, limiting the amount of decompression needed to almost the minimum. The optimal solution might seem to be caching the results of all nodes and to store them, until the end of the comparison. That way, every node only has to be decompressed once and if a node is visited for the second time, it can just read from wherever the node has been decompressed to. This would however greatly increase the memory usage during checking. In addition, the overhead and added branches from this actually even resulted in a slower execution time, when experimenting with that approach. As such, only limited reuse of decompressed node-information was implemented.

Figure 5.2: Comparison of original speed vs. compressed [A.6]



The performance is considerably worse, which could probably have been expected. Especially when comparing memory gain, against the performance overhead, this becomes apparent. To halve the remaining size of the optimized version, the overhead increases by around a factor of 6 to ∼60%. Overall this means, to reduce the size by 82% requires a 10% performance overhead and to reduce the original size by 90% a 60% overhead. A huge amount of work has been put into implementing this compression as efficiently as possible, so this remaining overhead seems to be the

around the minimum required to use this memory reduction. The only remaining options would be to either implement multi-threading, or increase the branching factor of the BoxTree.

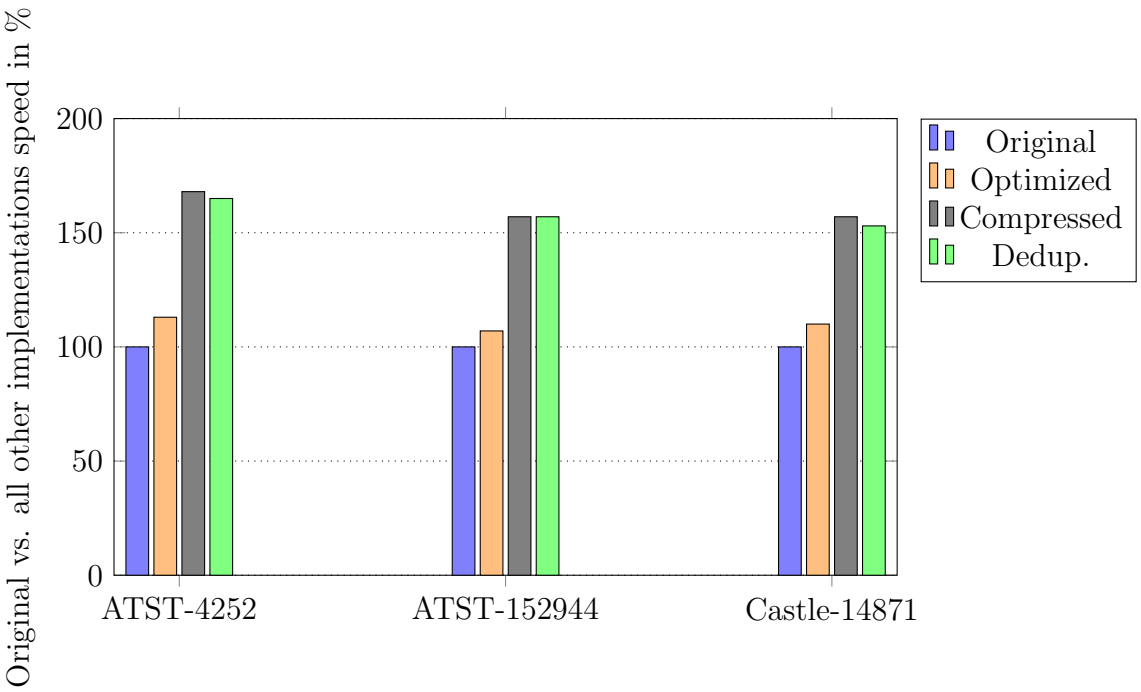
Multi-Threading might reduce the overhead, but it would also be possible and only fair to implement the same changes in the original, effectively not resulting in any change between the implementations. Otherwise, testing a single threaded application against one with in theory unlimited Threads would not really show anything other, than that more resources mean faster performance.

Increasing the branching factor means instead of having one box of the BoxTree split into two boxes, that one box splits into $n > 2$ many boxes. This could of course also just be implemented in the original, it does however have another problem. To represent more boxes, more data would have to be stored in each node. Effectively moving away from the best possible compression, at which point the optimized version should just be the preferred option.

5.3.1 Deduplication

Deduplication adds another layer of overhead on top of the existing compression. First the float index has to be decompressed and with that, the float array has to be read. This is most likely not cached and might even override cache, that could otherwise hold node information.

Figure 5.3: Overview of all performance comparisons [A.6]



Despite that, the performance can in some cases even be marginally faster, than the basic compressed version. It seems like the 3 instructions to convert a float in the compressed version is actually slower, than reading a value from an array. The overall performance does also seem to somewhat correlate with the unique amount of floats, that these geometries have. One explanation would be, that the same floats get accessed over and over again. As the model with the lowest amount of unique floats is the slowest, this does however not really make sense. A more likely explanation would be, that deciding factor is the overall size of the geometry. With a small geometry comes a small float-vector. This means the 4 floats, that can be stored per cache line account for a higher percentage of the overall amount of floats, making it more likely to access already cached results. Exactly this behaviour can also be observed, when looking at the cache-misses of these implementations, were compared to the compressed implementation, the cache-misses for the ATST-152944 is considerably higher, whilst staying relatively similar for the other models.[A.7]

CHAPTER 6

Conclusion

Looking at the performance results in comparison to the memory gain shows, that it is possible to achieve big memory gains with only minor performance penalties. This optimized implementation can be recommended for basically every use case, that values memory usage to some extend. When applying compression on top of that, the applicable use-cases shrink. With 60% overhead, this seems to be only be useful, when either the required RAM should be kept to a minimum for cost reasons, or the performance is not really a concern. For these, the question becomes, if deduplication of floats should be attempted. As seen during the performance tests, deduplication can be marginally faster, than the regular compressed version, it does however not reach the compression levels of that. With the performance differences almost falling into measuring inaccuracies, deduplicating the floats does overall not seem to be a viable strategy and remaining with 16bit floats seems to be the way to go.

With around 80-90% reduction in size, the size of the BoxTree seems to be almost at a minimum. As explained during the Automatic index calculation, the indexes are hard, if not impossible to optimize away. To reduce the size of the floats below

2 bytes each would either require implementing a custom conversion algorithm, or resort to some sort of approximation. To keep a comparable pace, this would have to be done with just a hand full of instructions, whilst the less precise result would most likely result in more unnecessary node-checks, further decreasing performance. For the time being, the current optimization and compression seems to be around the lowest possible.

List of Abbreviations

CNC Computerized Numerical Control. 1

CPU Central processing unit. 3

GCC GNU Compiler Collection. 26

GPU Graphics processing unit. 3

IoT Internet of Things. 2

RAM Random-Access Memory. 2, 3, 13, 26, 31

TBD to be determined. 16, 18

V-RAM Video Random Access Memory. 3

Bibliography

[1] T. Duc Tang, "Algorithms for collision detection and avoidance for five-axis nc machining: A state of the art review," *Computer-Aided Design*, vol. 51, pp. 1–17, 06 2014. 1

[2] F. Schwarzer, M. Saha, and J.-C. Latombe, *Exact Collision Checking of Robot Paths*, pp. 25–41. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. 1

[3] S. Dinas and J. Ban, "A literature review of bounding volumes hierarchy focused on collision detection," vol. 17, pp. 49–62, 06 2015. 1

[4] G. Zachmann, "The boxtree: Exact and fast collision detection of arbitrary polyhedra," 03 2001. 1.1

[5] Sony, "Playstation 5 spec sheet." `https://www.playstation.com/en-us/ps4/tech-specs/`, 2021. Accessed: 2021-10-14. 1.2

[6] Yuming Zou and Paul E. Black, "perfect binary tree in dictionary of algorithms and data structures [online], paul e. black, ed. 27." `https://www.nist.gov/dads/HTML/perfectBinaryTree.html`, 11 2019. Accessed: 2021-10-25. 2.6

[7] U. Drepper, "What every programmer should know about memory," 2007. 2.7

[8] Igor Pavlov, "7-zip compression library." `https://www.7-zip.org/sdk.html`, 2021. Accessed: 2021-10-14. 3.1

[9] K. C. Wang, *EXT2 File System*, pp. 301–356. Cham: Springer International Publishing, 2018. 3.2.1

[10] P. Min, "meshconv." `http://www.patrickmin.com/meshconv` or `https://www.google.com/search?q=meshconv`, 1997 - 2019. Accessed: 2021-10-20. 1

[11] R. Weller, D. Mainzer, G. Zachmann, M. Sagardia, T. Hulin, and C. Preusche, "A benchmarking suite for 6-dof real time collision response algorithms," 01 2010. 5.1

APPENDIX

A.1 Distribution of unique floats

	Number of floats	Unique floats	Unique floats in %
ATST-4252	14838	10378	69,94
ATST_20132	70468	43085	61,14
ATST-152944	533848	148970	27,90
Castle_14871	50022	17075	34,13
Raptor-400000	1545566	126665	8.20
boxtree_sphere_case_1	3256	1673	51,38
sphere -1310720 (complexity 1000)	5351530	650426	12,5

A.2 Number of merged vs unmerged nodes

	% merged / orig	# orig-nodes	# merged nodes
Raptor-40000	66	1172783	772783
ATST-4252	64	11623	7419
ATST-20132	64	55033	35234
ATST-152944	64	417155	266924
Castle-14871	65	39630	25011
boxtree_sphere_case_1	65	2510	1628

A.3 Size of optimized vs (optimally) compressed

	optimized size in KB	compressed size in KB	optimal compressed size in KB
Raptor-40000	12074	7394	7394
ATST-4252	116	62	58
ATST-152944	4171	2548	2518
Castle-14871	391	214	211

A.4 Sizes of all optimizations/compressions

	Raptor-40000	ATST-4252	ATST-152944	Castle-14871
optimized size in KB	12074	116	4171	391
comp. f16 size in KB	7394	62	2548	214
opt. comp. f16 size in KB	7394	58	2518	211
comp. f32 size in KB	10413	91	3591	312
opt. comp. f32 size in KB	10413	87	3560	308
Dedup size in KB	9312	98	3391	281
opt. Dedup size in KB	8041	97	3189	262

A.5 f16-conversion

```
#include <immintrin.h>
float f16Tof32(unsigned short n){
    return _cvtsh_ss(n);
}

f16Tof32(unsigned short):
        vpxor    xmm0, xmm0, xmm0
        vpinsrw xmm0, xmm0, edi, 0
        vcvtph2ps       xmm0, xmm0
        ret
```

This has been compiled using "-march=skylake, -O3" using GCC 11.1.

A.6 Speed comparisons

	Orig. time in ms	Opt. time in ms	Compressed time in ms	Dedup. time in ms
Castle-14871	109835	120301	172537	168131
ATST-4252	49308	55841	82790	812318
ATST-152944	82396	87710	129701	130232

These have been gathered by running: "./bench -A ALGO -g cu PATH-TO-MODEL -f PATH-TO-POSITIONS" using the modified version of bench

A.7 Cache-misses comparison

	#cache-misses original	#cache-misses compressed	#cache-misses deduplicated
ATST_4252	22.722.822	19.604.482	20.363.871
ATST_152944	76.001.853	63.070.606	75.479.357
Castle_14871	59.524.581	26.210.776	27.545.390

Universität Bremen

Nachname _____ MatrikelNr. _____

Vorname/n _____

A) Eigenständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.
Alle Teile meiner Arbeit, die wortwörtlich oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht. Gleiches gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet.
Die Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht.
Die elektronische Fassung der Arbeit stimmt mit der gedruckten Version überein.
Mir ist bewusst, dass wahrheitswidrige Angaben als Täuschung behandelt werden.

B) Erklärung zur Veröffentlichung von Bachelor- oder Masterarbeiten

Die Abschlussarbeit wird zwei Jahre nach Studienabschluss dem Archiv der Universität Bremen zur dauerhaften Archivierung angeboten. Archiviert werden:

1) Masterarbeiten mit lokalem oder regionalem Bezug sowie pro Studienfach und Studienjahr 10 % aller Abschlussarbeiten
2) Bachelorarbeiten des jeweils ersten und letzten Bachelorabschlusses pro Studienfach u. Jahr.

☐ Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

☐ Ich bin damit einverstanden, dass meine Abschlussarbeit nach 30 Jahren (gem. §7 Abs. 2 BremArchivG) im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

☐ Ich bin nicht damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

C) Einverständniserklärung über die Bereitstellung und Nutzung der Bachelorarbeit / Masterarbeit / Hausarbeit in elektronischer Form zur Überprüfung durch Plagiatssoftware

Eingereichte Arbeiten können mit der Software *Plagscan* auf einen hauseigenen Server auf Übereinstimmung mit externen Quellen und der institutionseigenen Datenbank untersucht werden.
Zum Zweck des Abgleichs mit zukünftig zu überprüfenden Studien- und Prüfungsarbeiten kann die Arbeit dauerhaft in der institutionseigenen Datenbank der Universität Bremen gespeichert werden.

☐ Ich bin damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum Zweck der Überprüfung auf Plagiate auf den *Plagscan*-Server der Universität Bremen hochgeladen wird.

☐ Ich bin ebenfalls damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum o.g. Zweck auf dem *Plagscan*-Server der Universität Bremen hochgeladen u. dauerhaft auf dem *Plagscan*-Server gespeichert wird.

☐ Ich bin nicht damit einverstanden, dass die von mir vorgelegte u. verfasste Arbeit zum o.g. Zweck auf dem *Plagscan*-Server der Universität Bremen hochgeladen u. dauerhaft gespeichert wird.

Mit meiner Unterschrift versichere ich, dass ich die oben stehenden Erklärungen gelesen und verstanden habe.
Mit meiner Unterschrift bestätige ich die Richtigkeit der oben gemachten Angaben.

Datum Unterschrift