Bandwidth and Memory Efficiency in Real-Time Ray Tracing

Pedro Lousada
INESC-ID/Instituto Superior Técnico, University of Lisbon
Rua Alves Redol, 9
1000-029 Lisboa, Portugal
pedro.lousada@ist.utl.pt

Vasco Costa
INESC-ID/Instituto Superior Técnico, University of Lisbon
Rua Alves Redol, 9
1000-029 Lisboa, Portugal
vasco.costa@ist.utl.pt

João M. Pereira
INESC-ID/Instituto Superior Técnico, University of Lisbon
Rua Alves Redol, 9
1000-029 Lisboa, Portugal
jap@inesc-id.pt

ABSTRACT

Real time ray tracing has been given a lot of attention in recent years in the academic and research community. Several novel algorithms have appeared that parallelize different aspects of the ray tracing algorithm through the use of a GPU. Among these, the creation of Bounding Volume Hierarchies (BVHs). We believe that recent approaches have failed to consider the performance impact of memory accesses in GPU and how their cost affects the overall performance of the application. In this work we show that by reducing memory bandwidth and footprint we are able to achieve significant improvements in BVH traversal times. We do this by compressing the BVH and the triangle mesh in a parallel manner after its creation, in each frame, and then decompressing it as needed while traversing the BVH.

Keywords

Ray-tracing, bounding-volume-hierarchy, parallelization, gpu, quantization, memory.

1 INTRODUCTION

Real-time rendering typically concerns itself with the generation of synthetic images at a rate fast enough that the viewer can interact with a virtual environment. As an image appears on screen, the viewer acts or reacts, and this feedback affects what is generated next. Two of the most popular approaches to synthetic image generation are Rasterisation and Ray-Tracing. Both have been used in computer graphics for the past decades. Each method allow us to generate 2D images from 3D scenes composed of virtual objects.

Real-time ray-tracing received little attention outside the academic world mainly due to its high computation costs which made it a much more expensive and slow approach compared to rasterisation. Ray tracing offers a fairly long list of advantages over rasterisation. Ray-tracing can easily simulate non-local effects such as shadows, reflections and refractions. In Rasterisation reflections and shadows are hard to compute; refractions are very hard.

Due to its mathematical correctness, ray tracing can make generated images look more realistic. The issue is being able to generate them at a rate fast enough as required by these real-time applications.

1.1 Problem

At its core, the ray tracing algorithm follows the following logic: for each pixel of the display, we cast rays that propagate in a straight line until they intersect an element of the scene being rendered. The color of the pixel is computed as a function of the material at the intersected element's surface, the incident light (radiance) function at the intersection point, and of the viewing direction (observer's position). For a simple scene with no secondary rays (i.e. reflections, shadows, refractions, etc) this means having at least NxM intersection tests (N being the number of rays and M the number of polygons in the scene). For its nature, ray tracing inherently leads to a high number of expensive floating point operations. This, together with an irregular and high bandwidth memory access pattern, means performance will be an issue when trying to achieve real time rendering.

One way to optimize the standard ray tracing algorithm is through the use of acceleration structures. Acceleration structures allow us to lower the number of intersections tests needed to render an image. Currently Bounding Volume Hierarchies (BVHs) are the most popular solution. The state of the art in this kind of structure

has been targeted towards generating the structure at a rate fast enough to be usable in a real time application, typically creating a new BVH or refitting an existing one at each frame.

Most modern algorithms use a GPU to achieve the performance they need. Despite their capability of performing a high number of floating point operations per second, GPUs suffer from slow memory access. Yet most algorithms that focus on constructing a BVH in parallel manner tend to overlook this and are careless with both memory accesses and bandwidth, resulting in poorer overall performance than otherwise possible.

1.2 Contributions

We believe that by optimizing GPU memory footprint and bandwidth efficiency we will be able to improve render performance. In this work we show how to develop a ray tracing application based on a state of the art algorithm in parallel BVH construction and then improve it further through memory compression techniques.

Our contribution is a novel algorithm, based on existing techniques for real time BVH construction, with focus on improvements in BVH and triangle mesh compression. We are able to reduce the total size of memory used to store both the BVH and the triangle mesh as well as reduce the memory bandwidth of the application. We saw improvements of up to 40% in occupied memory, and reductions of up to 20% in memory bandwidth, with our techniques.

2 BACKGROUND

First introduced by Whitted [Whi80], ray tracing is an algorithm for image synthesis where direct illumination (including shadows), and perfect reflections/refractions are simulated. It employs the use of ray casting to intersect eye rays with objects in a computer simulated scene. Eye rays which intersect objects lead to the creation of extra secondary rays: e.g. shadow, reflection and refraction rays.

The main difference between rasterisation and raytracing is the ability to simulate complex light effects such as shadows, reflections and refractions. Whereas rasterisation engines often simulate these effects through the use of texturing, ray tracing takes a more accurate analytical approach. Since these effects are highly geometry dependent, simulated methods can look unrealistic when changing object position or viewer position. The difference in ability to simulate secondary visual effects between rasterisation and ray tracing can be seen in Figure 1.

Figure 1: Rasterisation vs ray tracing (source: Intel)

2.1 Bounding Volume Hierarchies

One of the major problems of ray tracing is the sheer number of intersection tests one must perform in order to check if a certain ray intersects an object of the scene. Acceleration structures help us reduce the number of intersections to test by organizing the geometry of the scene into a data structure that can easily be explored. The organization of an acceleration structure is typically hierarchical, loosely meaning that the topmost level encloses the levels below it, and so on.

Bounding Volume Hierarchies (BVHs) are a tree-like structure that subdivides a scene into smaller portions. A BVH partitions a scene's objects. Each geometric primitive object is wrapped with an individual bounding volume. These form the leaf nodes of the tree. Bounding volumes are then recursively merged together until we are left with a single bounding volume wrapping the entire scene.

In a typical object hierarchy data structure it is easy to update the data structure as an object moves, because an object lives in just one node, thus the bounds for that node can be updated with relatively simple and localized update operations. For deformable scenes, just refitting a BVH - i.e., recomputing the hierarchy node's bounding volumes, but not changing the hierarchy itself - is sufficient to produce a valid BVH for the new frame. BVHs also allow for incremental changes to the hierarchy [WMG+09].

In BVHs, primitives are referenced exactly once, allowing us to save GPU memory and bandwidth. Empty cells, that frequently occur in spatial subdivision, do not exist in object hierarchies either. The effectiveness of a Bounding Volume Hierarchy for a particular scene depends on the characteristics of the hierarchy the build algorithm produces.

2.2 GPU Computing

Modern GPUs are SIMD (Single Instruction Multiple Data) devices [GPKB12], meaning that they can per-

form the same operation on multiple data points simultaneously. Even though a modern CPU core can dispatch more operations per second than a GPU core, the GPU as a whole can vastly outperform a CPU by having thousands of cores running the same operations versus the 4-8 cores present in a CPU. The massive parallelism of programmable GPUs lends itself to inherently parallel problems such as ray tracing. A given compute kernel executes a single program across many parallel threads. Typically, each kernel completes execution before the next kernel begins, with an implicit barrier synchronization between kernels. GPUs have support for multiple, independent kernels to execute simultaneously, but many kernels are large enough to fill the entire device. Threads are decomposed into thread blocks; threads within a given block may efficiently synchronize with each other and have shared access to per-block on-chip memory.

3 RELATED WORK

Generating an acceleration structure is a necessary step when trying to achieve real-time performances. One can take two approaches: To construct a new structure every frame or to reuse the same structure and adjust it at each frame. The latter option has been explored [KA13] but lacks performance when processing large trees or large modifications to the data structure are required.

Approaches that explore a construction of new hierarchy at each frame tended to rely on serial algorithms running on the CPU to construct the necessary hierarchical acceleration structures [GPM11]. While this was once necessary due to architectural limitations, modern GPUs provide all the facilities necessary to implement hierarchy construction directly. Doing so should provide a strong benefit, as building structures directly on the GPU avoids the need for the relatively expensive latency introduced by copying data structures between CPU and GPU memory spaces.

The first to explore such a method were Lauterbach et al. [LGS+09]. Lauterbach et al. introduced a novel algorithm using *spatial Morton codes* [Mor66] to reduce the construction of BVHs to a sorting problem. *Morton codes* are used to determine a primitive's order along a space filling curve. They can be computed directly from a primitive's geometric coordinates. The algorithm encloses each input primitive with an Axis-Aligned minimum Bounding Box (AABB) and determines the enclosing AABB of the entire input geometry. By taking the barycenter of each primitive's AABB as its representative point, and by quantizing each of the 3 coordinates of the representative points into k-bit integers, a $3k$-bit *Morton code* is constructed by interleaving the successive bits of these quantized coordinates. Figure 2 shows a 2D representation of this. Sorting the *Morton codes* will automatically lay the associated points in order along a *Morton spatial curve*. It will also order the corresponding primitives in a spatially coherent way. Because of this, sorting geometric primitives according to their *Morton code* is used to improve cache coherence since a ray that hits a certain primitive will likely also hit the primitive adjacent to it.

The main problem of this family of algorithms [LGS+09, PL10, GPM11] is that, in order to build the resulting BVH, a series of sequential steps must be taken. Lauterbach et al. create their BVH by sequentially observing each bit of each *Morton code* and grouping the primitives according to the value of the bit. At each level if said bit has value *0* then the primitive is placed in a group, if the bit has value *1* then it is placed in the opposite group. Garanzha et al. take a different approach by generating one level of nodes at a time, starting from the root. They then process the nodes of the BVH on a given level in parallel. For this they use binary search to partition the primitives contained within each node. The resulting child nodes are then enumerated using an atomic counter, and subsequently processed on the next round.

Karras et al. [Kar12] further developed this line of thought by describing an algorithm to build a BVH in a totally parallel manner. Karras et al. introduce an in-place algorithm for constructing a binary radix tree (also called a *Patricia* tree) which can directly be converted into a BVH.

Their approach is based on the fact that for a scene with N primitives we know we can make a *Patricia* tree with $N-1$ internal nodes to represent it. The similarities between each *Morton code* and its neighbours are analyzed to determine the position of each internal node in the Patricia Tree. The child-parent association is then calculated based on the range of *same-value-bits* with the rest of the *Morton codes*.

BVHs have a large memory footprint due to the need to store the bounding boxes. Hence several approaches have been used, over the CPU, to allow the visualization of large models without paging data from disk. Mahovsky et al. [MW06] and Bauszat et al. [BEM10] quantize the bounding box data, with significant memory and bandwidth savings, at the expense of extra computations. These approaches degrade rendering performance for models that fit uncompressed within main memory, even with a ray-bundle scheme, due to limited available CPU math performance. However a GPU is more bandwidth than math constrained so this conclusion needs to be revisited in that case.

4 APPROACH AND ARCHITECTURE

In order to achieve a high frame rate one must reduce the time it takes to generate an image at each frame.

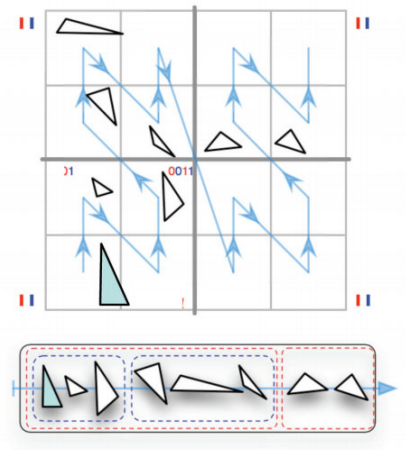


Figure 2: Example 2-D Morton code ordering of triangles with the first two levels of the hierarchy. Blue and red bits indicate x and y axes, respectively. (source: NVIDIA)

For this we make use of a GPU to help us accelerate all the floating point operations required to compute the intersections in a ray-tracer. We aim to reduce the memory bandwidth and memory footprint of our application. We make use of research, made in the area of parallel BVH construction in GPUs, to reduce the number of intersection tests performed (thus reducing memory bandwidth as well). We then explore BVH and triangle meshes compression techniques. With a compressed BVH and triangle mesh we expect our rendering kernels to achieve better rendering times since the data transferred between the GPU's global memory and the kernel's local memory will be smaller.

This is described in further detail in the following sections.

4.1 Binary Radix Tree Properties

Before we describe our algorithm there are a few Binary Radix Tree properties we should cover as these are important to understand our work.

A radix tree is a space-optimized tree often used for indexing string data, although it can also be used to index any data divisible in smaller comparable chunks such as characters, binary numbers, etc. For simplicity, assume that from now on our radix tree only contains binary values and that they are in a lexicographical order as in our application each key will correspond to a sorted *Morton code*.

Given a set of keys $k_0, ..., k_{n-1}$ represented as bits, a radix tree can be seen as a hierarchical representation of the common bits of each key. The keys are represented in the leaf nodes of the tree, and each internal node corresponds to the longest common prefix shared between the keys in that subtree.

Assume the example referenced in Figure 3. As one would expect the root of the tree covers the full range of keys. At each level the keys are partitioned according to their first differing bit. The first difference occurs between keys k_3 and k_4, thus, the left child of the root node contains keys k_0 to k_3 and the right child contains k_4 to k_7. We continue this process to essentially get a hierarchical representation of the common prefixes between each key. At the bottom level of the tree we will find that each child references a key.

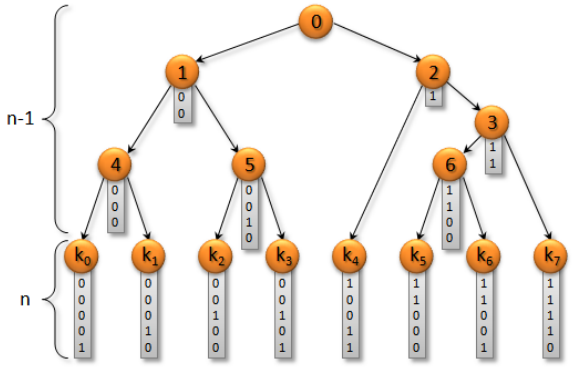


Figure 3: A visual representation of an ordered radix tree. The numbers 0-7 act as keys (leaf nodes) of the tree. Each internal node represents a common range prefix of the binary value of all the leaf nodes bellow it. Notice we have $N-1$ internal nodes for N keys.

A radix tree is considered a compact data structure, as it omits nodes which only have one child, thus removing redundant information and decreasing the overall size of the tree in memory.

One property of a binary radix tree is that any given tree with N keys will have $n-1$ internal nodes. This allows us to know, even before we construct the tree, how many nodes, and thus how much memory, we will require. Assuming that we have a lexicographically ordered tree allows us to express $[i, j]$ as the range of keys covered by any given internal node. We use $\delta(i, j)$ to denote the length of the longest common prefix between the keys k_i and k_j. The ordering of the keys automatically implies that $\delta(i', j') \geq \delta(i, j')$ for any $i', j' \in [i, j]$ [Kar12]. We can then determine the common prefix shared between a key under a given node by comparing the first and last key it covers - all the other keys in between are guaranteed to share the same or a larger prefix.

In practice each internal node partitions the keys under it on their first differing bit, the one following $\delta(i, j)$. We can safely assume that in the range $[k_i, k_j]$ part of the keys will have said bit set to 0 and others will have it set to 1. Since we are working with an ordered tree all of the keys with the bit set to 0 will be presented before the keys with the bit set to 1. We call the position of the last key, where this bit has value 0, the split position denoted by $\gamma \in [i, j-1]$. Since the split position is where the first bit differs between the keys in the range we can say that

$\delta(\gamma,\gamma+1) = \delta(i,j)$. The ranges $[k_i,k_\gamma]$ and $[k_{\gamma+1},k_j]$ will be subdivided at the next differing bit. We can thus say for sure is that $\delta(i,\gamma) > \delta(i,j)$ and $\delta(\gamma+1,j) > \delta(i,j)$. Taking Figure 3 once more as reference, we can see that the first differing bit happens between keys k_3 and k_4, the range is then split at $\gamma = 3$ resulting in the subranges $[k_0,k_3]$ and $[k_4,k_7]$. The left child then splits its range $[k_0,k_3]$ at the third bit, $\gamma = 1$. The right child splits the range $[k_4,k_7]$ at $\gamma = 4$, at the second bit, and so on.

4.2 Morton Codes

We start our process with a series of primitives represented by 3D points. In order to generate our BVH we first start by deciding in which order each leaf node will be represented in the tree. A good approach is to sort the leafs according to their position, generally we will want primitives close to each other in 3D space to appear close to each other on the tree. In order to do this we sort them via a space-filling curve, more specifically a Z-order curve. For this we take the centroid of each triangle and express it relative to the bounding volume of the entire scene, known after loading the scene's data.

Let bvh_{min} be defined as the minimum extents of the scene's bounding volume and bvh_{max} as its maximum. If we define c as the centroid point of a given triangle then we can express q, the same point, but now in coordinates relative to the bounding volume of the scene through the following formula:

$$q = \frac{c - bvh_{min}}{bvh_{min} - bvh_{max}} \tag{1}$$

q's coordinate values now vary between 0 and 1. We can think of it as a point within the 3D space delimited by the scene's bounding volume. The closer each coordinate is to 0 the closer the point will be to the minimum point of the bounding volume, and, the closer it is to 1 the closer it will be to its maximum point.

We now want to express this 3D point as a *Morton code*. The first step in this process is to transform our point from a continuous space into a discrete one. We achieve this by quantizing each floating point coordinate into a range created by the difference between the scene's bounding volume maximum and minimum extremes. Morton codes are most efficiently expressed as a single integer so to represent a 3D point in a single integer value we will have to make some precision sacrifices. We assume a machine architecture with 32 bit sized integers, meaning that in order to represent 3 distinct values in 32 bits we will have 10 bits for each of the 3 Cartesian coordinates. We are thus left with the following quantization equation:

$$q = \frac{(c - bvh_{min}) \times 2^{10}}{bvh_{min} - bvh_{max}} \tag{2}$$

Where q is now a 3D point whose coordinates are composed of 10 bit integers. To correctly represent this point along a Z-order curve we interleave the bits of all three coordinates together to form a single binary number. We take the value of each coordinate, expand the bits by inserting 2 bit "gaps" after each bit and then interleave them. Beyond this point it is simply a matter of calculating the Morton codes for each primitive and sorting them via a parallel sorting algorithm, we used radix sort for this operation.

4.3 Parallel Construction of BVH

This section follows the same line of thought described by Karras in [Kar12]. The idea is to assign indexes for the internal nodes in a way that enables finding their children without depending on earlier results, this way we can fully parallelize the construction of the BVH.

We create two separate arrays to store our nodes, **L** for the leaf nodes and **I** for the internal nodes. We define the layout of **I** as having the root node at the index 0, denoted by $\mathbf{I_0}$. The index of each internal node will be defined by its split position. For any given node the left child will be defined in $\mathbf{I}_\gamma$ if it covers more than one key and at $\mathbf{L}_\gamma$ if it doesn't. Similarly the right child will be located either at $\mathbf{I}_{\gamma+1}$ or at $\mathbf{L}_{\gamma+1}$. This layout has an important property, the index of every internal node will either coincide to the first or the last key it covers. Take for example the root node, it covers the entire range of keys $[0,n-1]$ and is located at position $\mathbf{I_0}$. A node covers the range $[i,j]$; its left child will be located at the end of the range $[i,\gamma]$ and its right child located at the beginning of the range $[\gamma+1,j]$.

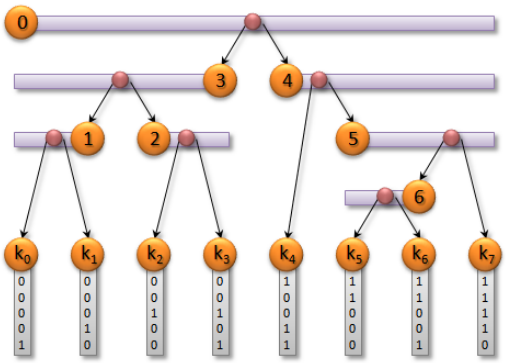


Figure 4: Bar representation of the keys covered by each internal node.

In Figure 4 we present a visual example of this property. Node 0 covers the entire range $[k_0,k_7]$ and is therefore located at $\mathbf{I_0}$. Its children, node 3 and 4 cover the ranges $[k_0,k_3]$ and $[k_4,k_7]$ and are placed at $\mathbf{I_3}$ and $\mathbf{I_4}$, respectively. Interestingly, this process will never result in gaps or duplicates when populating the internal node array. An advantage of using this scheme is that each internal node is conveniently placed next to a sibling

node in memory. And since internal nodes run from 0 to $n-1$ we can use them to directly address the nodes in memory.

In order to construct the BVH, we need to know more than simply which nodes cover which keys, we need to know how the nodes connect amongst themselves, i.e. the parent-child relations.

Lets say we want to determine the direction in which the range of keys covered by node $\mathbf{I_i}$ extends in. We call this direction d. In order to determine d we compare the length of the common prefix between the keys k_{i-1}, k_i, and k_{i+1}. If $\delta(i-1,i) > \delta(i,i+1)$ then $d=-1$. And if $\delta(i-1,i) < \delta(i,i+1)$, then $d=+1$.

Knowing this we can say that k_i and k_{i+d} both belong to node $\mathbf{I_i}$, and that k_{i-d} belongs to node $\mathbf{I_{i-d}}$.

We now need to know how far the range of each node extends. Since k_{i-d} does not belong to the range of keys covered by node $\mathbf{I_i}$, we can safely assume that the keys which are share amongst themselves a larger common prefix than k_i and k_{i-d} do. We call this common prefix δ_{min} so that $\delta_{min} = \delta(i,d-1)$.

$\delta(i,j) > \delta_{min}$ for any k_i belonging to $\mathbf{I_i}$. This being said all we need to do to find the other end of the range is to search for the largest l that satisfies the equation $\delta(i,i+ld) > \delta_{min}$. The fastest way to do this is to start at k_i and in powers of 2 increase the value of l until it no longer satisfies $\delta(i,i+ld) > \delta_{min}$. Once this happens we know that we have gone too far and we have left the range of keys covered by $\mathbf{I_i}$. Lets call this upper bound l_{max}. We know for sure the correct value of l is somewhere in the range $[l_{max}/2, l_{max}-1]$. Now it is only a matter of using a binary search to find the value of l under which $\delta(i,d(l+1)+i) \leq \delta_{min}$.

After finding the value of l we can use $j = i+ld$ to specify the other end of the range.

Lets call δ_{node} the length of the common prefix shared between k_i and k_j, given by $\delta(i,j)$. We use δ_{node} to search the split position γ that partitions the keys covered by $\mathbf{I_i}$. Now we perform a search for the largest $s \in [0, l-1]$ that satisfies $\delta(i,i+sd) > \delta_{node}$. i.e., we need to find the furthest key which shares a larger prefix with k_i than k_j does.

Discovering γ allows us to determine the ranges covered by each children. The left child will have a range covering $[min(i,j]), \gamma]$ and the right child will cover $[\gamma+1, max(i,j)]$.

For the final step we analyze the values of i, j and γ. If $i = \gamma$ we know $\mathbf{I_i}$'s left child is the leaf node $\mathbf{L}_\gamma$, otherwise it's the internal node $\mathbf{I}_\gamma$. Correspondingly if $j = \gamma+1$ we say $\mathbf{I_i}$'s right child is the leaf node $\mathbf{L_{\gamma+1}}$, otherwise it's internal node $\mathbf{I_{\gamma+1}}$. Pseudocode for this algorithm can be found in Algorithm 1.

Algorithm 1 Pseudocode for the parallel construction of a BVH [Kar12].

```
for all internal node with index i ∈ [0, n−2] do
    // Determine direction of the range (+1 or -1)
    d ← sign(δ(i,i+1) − δ(i,i−1))
    // Compute upper bound for the length of the range
    δ_min ← δ(i,i−d)
    l_max ← 2
    while δ(i,i+l_max·d) > δ_min do
        l_max ← l_max·2
    // Find the other end using binary search
    l ← 0
    for t ← l_max/2, l_max/4, ..., 1 do
        if δ(i,i+(l+t)·d) > δ_min then
            l ← l+t
    j ← i+l·d
    // Find the split position using binary search
    δ_node ← δ(i,j)
    s ← 0
    for t ← {⌈l/2⌉, ⌈l/4⌉, ..., 1} do
        if δ(i,i+(s+t)·d) > δ_node then
            s ← s+t
    γ ← i+s·d
    // Output child pointers
    if min(i,j) = γ then left ← L_γ else left ← I_γ
    if max(i,j) = γ+1 then right ← L_{γ+1} else right ← I_{γ+1}
    Ii ← (left, right)
```

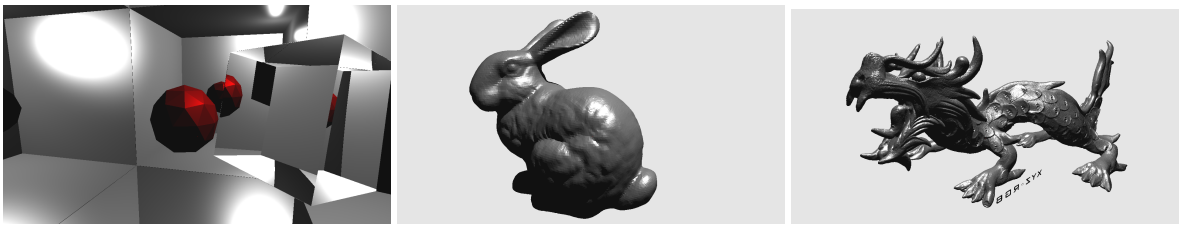
4.4 Compression

Reducing memory footprint and bandwidth is our goal. A certain amount of bandwidth is reduced by simply using a BVH. We can further improve our gains by compressing both the BVH and the triangle mesh.

4.4.1 BVH Compression

We start a thread at each internal node and make our way up to the top of the tree. Once we reach the top of the tree we start walking the same path in the opposite direction, that is, from the root node to the internal node in question. As we descend each level we take the bounding box of the previous parent and subdivide each dimension in 1024 segments. We treat this 1024^3 grid as a voxel space and map the minimum and maximum points of the current level's node bounding box into the nearest corresponding voxels.

It becomes clear that we lose some precision as we descend each level since we are going from floating point coordinates into integer indexes. Each index will be contained in the range $[0, 1024[$, so we can store 3 of these indexes in a single 32 bit integer. Each bounding box can then be stored as a single 2 integer data structure instead of a 6 float data structure, reducing its size down from 24 bytes to 8 bytes. It is obvious that with this method our BVH will become more loosely coupled since we lose precision when going from a continuous space (world coordinates) into a discrete one (voxel indexes). We round up when mapping the maximum point of a bounding volume to a voxel and round down when mapping the minimum so our bounding



(a) SHINY (152 triangles) (b) BUNNY (70K triangles) (c) ASIAN DRAGON (7.2M triangles)

Figure 5: Test Scenes.

volumes remain coherent. We predict this loss of precision will not have a big impact in our intersection tests, we will most likely intersect a few more bounding volumes, as they will be slightly larger in size, but we do not expect a significant increase in the number of tests. One thing to take into account is that in each thread we must quantize the entire path from the root to each internal node otherwise our BVH will become incoherent.

4.4.2 Mesh Compression

Karras [Kar12] uses each leaf node as a redirect to a primitive. We can remove an indirection level by having each leaf node envelop a single triangle and store said triangle in the node itself. We can also try and diminish the number of memory necessary to store a triangle by following the same line of thought from the BVH compression. By quantizing the position of each vertex of each triangle, into the bounding box that encapsulates it, we can go from having to store 9 floats to just 3 integers. This will of course have an impact on the model itself as we will be losing the original coordinates of each vertex. When recalculating each vertex back to world coordinates we will be changing the final world positions of each triangle, resulting in slight mesh deformation. Interestingly enough, this effect isn't noticeable unless examining models up close. In a practical application like a game or simulation where the camera and the objects are constantly moving this effect could pass up unnoticed. This compression step can be done in the same kernel as the quantization of the internal nodes of the tree, thus having little to no effect on the post-processing time of the BVH.

5 RESULTS

Tests were made using an NVIDIA GeForce GTX 970 with 4 GB of RAM. We chose to render our images in 1280x720p as this is one of the most commonly used resolutions for multimedia content.

We tested our algorithm with three different scenes, ASIAN DRAGON, SHINY and BUNNY.

SHINY (see Figure 5a) is a scene representative of highly reflective and refractive scenes. It also represents a low polygon scene. It consists of an icosphere surrounded by five mirrors and a glass prism. This scene focuses on testing performance in scenes with a high number of secondary rays.

BUNNY (see Figure 5b) is what we would call a "medium" sized scene. It serves as a midterm between low polygon scenes (SHINY) and high polygon scenes (ASIAN DRAGON). This scene only features eye and shadow rays.

ASIAN DRAGON (see Figure 5c) is representative of complex objects with a high number of triangles. It is a dense model with a great number of triangles in a small space. Our intent with this scene is measure the performance gains of BVH compression for dense BVHs. This scene only features eye and shadow rays.

5.1 Ray-Box Intersection Tests

Despite SHINY being a small, enclosed scene we notice that lack of precision of the bounding volume areas causes an increase of 9% in the intersection tests (see Figure 6). We assume this is caused by reflected and refracted rays, that would normally pass tangent to the icosphere, but are instead tested and categorized as a hit. In BUNNY we notice an increase of 7.1% in the number of ray-box intersection tests, making it the less affected of all the 3 scenes. In ASIAN DRAGON we notice an increase of 12.38% in the number of ray-box intersection tests. Since this scene has a high tree we expect rounding "errors" to accumulate over the levels thus leading to this increase in the number of intersection tests.

5.2 Ray-Triangle Intersection Tests

SHINY has an increase of 11.1% in the tests performed. We believe this number is greater, in comparison to the other test scenes, because in this scene almost every ray will hit an object and thus it will have to traverse the entire BVH, making it more prone to the effects of loss of precision of the bounding volumes.

In BUNNY we notice an increase of 7.4% in the number of tests performed. This increase in the number of ray-triangle test seems to be similar to the increase of ray-box intersection tests.

ASIAN DRAGON has an increase of 10.75% tests performed. As in BUNNY this number seems to go in hand

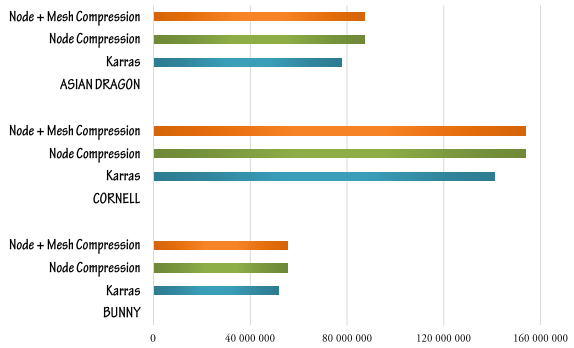


Figure 6: Amount of ray-box intersections.

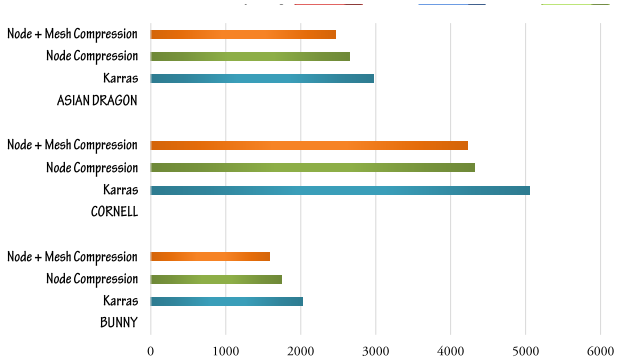


Figure 7: Memory read in each frame (MB).

with the increase of ray-box intersection tests, although slightly lower.

5.3 Global Memory Reads

The following results are based in the number of intersection tests performed and the size of each internal node, leaf node and primitive.

In SHINY we notice we are able to achieve a significant impact in the amount of memory read with node compression (14.5%), but see little to no gains when implementing mesh compression (see Figure 7). This is because, in this scene, the number of ray-box intersections dwarfs the number of ray-triangle intersections, hence optimizations made to the execution time of ray-triangle tests will have little impact.

BUNNY reduces its memory accesses by 14% when compressing internal nodes alone and 22% when compressing the mesh alike. These results go in hand with the results obtained in ASIAN DRAGON providing some insight that every reasonably complex model is positively affected by our improvements.

In ASIAN DRAGON we see a steady decrease of accessed memory with our algorithms. We reduce memory accesses by 11.9% with node compression and, by 20.3% with mesh compression.

5.4 Kernel Execution Time

In this section we measure the execution time for each of the most relevant kernels. KARRAS represents the basic algorithm, as described by Karras in [Kar12]. NODE is the same algorithm with BVH compression. NODE+MESH is the same algorithm with BVH and triangle compression.

SHINY does not benefit significantly from our modifications. We notice a reduction in rendering time of 6.7% when compressing internal nodes and 10.7% when compressing the mesh. BVH construction and compression time improvements are marginal at best. We believe our modifications do not have an impact on this scene since it's easy to fit in the GPU caches.

	KARRAS TIME (MS)	NODE TIME (MS)	NODE+MESH TIME (MS)
BVH CREATION	2.17	2.17	2.17
NODE COMP	0	0.01	0.01
MESH COMP	0	0	0.01
RENDERING	48.93	45.68	**43.68**

Table 1: Kernel execution time for SHINY frame.

	KARRAS TIME (MS)	NODE TIME (MS)	NODE+MESH TIME (MS)
BVH CREATION	7.82	7.82	7.82
NODE COMP	0	0.24	0.24
MESH COMP	0	0	0.35
RENDERING	80.74	64.15	**46.83**

Table 2: Kernel execution time for BUNNY frame.

In the BUNNY scene we notice most of the time is spent executing the rendering kernel as in SHINY. The creation of the BVH has a slight impact on the time required to render each frame and remains constant across all 3 variations. Compression is quick but greatly impacts the time it takes to render the scene. Here we can see a reduction in rendering time of 20.55% when compressing the BVH's internal nodes and a reduction of 41.01% when compressing both internal nodes and triangles. We are able to achieve a 37.62% overall improvement in time to frame performance with a full rebuild.

	KARRAS TIME (MS)	NODE TIME (MS)	NODE+MESH TIME (MS)
BVH CREATION	424.64	424.64	424.64
NODE COMP	0	16.33	16.33
MESH COMP	0	0	24.43
RENDERING	246.36	194.44	**160.95**

Table 3: Kernel execution time for ASIAN DRAGON frame.

In ASIAN DRAGON we see a time reduction of 21.08% in rendering kernel execution when compressing just the internal nodes, and 34.67% when compressing both internal nodes and triangles. As we can see our algorithm has a significant impact in high poly models, we fetch a large number of BVH nodes in this scene. Un-

fortunately the BVH CREATION kernel takes up most of the time in each frame when dealing with such a high number of polygons. So overall we are only able to achieve a 6.6% improvement in time to frame performance with a full rebuild.

6 CONCLUSIONS

In our tests we used the vanilla version of Karra's algorithm as a control benchmark. Tests showed all complex test scenes benefit from our compression approach. As we hypothesized the number of intersection tests increases but the overall time to traverse the BVH and perform each intersection test decreases. Some scenes benefit more from our algorithm than others. We notice that in scenes with a high number of polygons the construction of the BVH becomes a bottleneck, taking most of the time in the frame. This affects our algorithm as much as the control algorithm. Gains from our approach were diluted by this bottleneck for this kind of scene.

As we initially expected real time ray tracing benefits from a reduction in memory footprint and bandwidth. Despite having to spend more time compressing and then decompressing both the BVH and the scene's primitives in each frame this penalty is compensated by the reduced time it takes to fetch the scene's data from global memory.

7 FUTURE WORK

BVH construction times are a bottleneck in high polygon scenes. Our compression techniques improve rendering times, but the construction times are still an issue, in particular in large scenes with full rebuilds, these could also benefit from working set minimization, compression, or both. Using 64 bit types to store quantized values could also be a solution worth exploring.

8 ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their insightful comments.

This work was supported by national funds through Fundação para a Ciência e Tecnologia (FCT) with reference UID/CEC/50021/2013.

REFERENCES

[BEM10] Pablo Bauszat, Martin Eisemann, and Marcus A Magnor. The Minimal Bounding Volume Hierarchy. In *VMV*, pages 227–234, 2010.

[GPKB12] Jayshree Ghorpade, Jitendra Parande, Madhura Kulkarni, and Amit Bawaskar. GPGPU processing in CUDA architecture. *arXiv preprint arXiv:1202.4347*, 2012.

[GPM11] Kirill Garanzha, Jacopo Pantaleoni, and David McAllister. Simpler and faster HLBVH with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 59–64. ACM, 2011.

[KA13] Tero Karras and Timo Aila. Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 89–99. ACM, 2013.

[Kar12] Tero Karras. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pages 33–37. Eurographics Association, 2012.

[LGS+09] Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. Fast BVH construction on GPUs. In *Computer Graphics Forum*, volume 28, pages 375–384. Wiley Online Library, 2009.

[Mor66] Guy M Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company New York, 1966.

[MW06] Jeffrey Mahovsky and Brian Wyvill. Memory-Conserving Bounding Volume Hierarchies with Coherent Raytracing. In *Computer Graphics Forum*, volume 25, pages 173–182. Wiley Online Library, 2006.

[PL10] Jacopo Pantaleoni and David Luebke. HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, pages 87–95. Eurographics Association, 2010.

[Whi80] Turner Whitted. An Improved Illumination Model for Shaded Display. *Commun. ACM*, 23(6):343–349, Jun 1980.

[WMG+09] Ingo Wald, William R Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G Parker, and Peter Shirley. State of the art in ray tracing animated scenes. In *Computer Graphics Forum*, volume 28, pages 1691–1722. Wiley Online Library, 2009.