Approaching Arbitrary Scene Size for Ray Queries
Asher T. Norland

Figure 1: A frame of an error prone result near the end of the project.

Abstract
In this paper, a process to translate a user defined scene graph into a acceleration structure is explored.

Keywords: bounding volume hierarchy, scene graph

1 Introduction

Any graphics program that has multiple objects or deals with very large objects has to consider the coordinate system in which it resides. Whether it be a multi-hour render or a real-time application, any errors due to floating point precision limit both the design and scope of a project. Increasing the datatype resolution with double floating point variables does not solve the problem, it only delays misrepresentation as the scene size increase. Doubles additionally decrease GPU performance to an unacceptable level. Solutions in a modern workflow involve partitioning space by hand into multi-layered grids or compositing multiple renders into one scene. These processes are undesirable as they involve oversight and time from a human to setup and interface with, and are not automated.

By extending the acceleration structure of a scene, an automatic scene graph can be composed that minimizes the accumulated error between a ray origin and its intersection points. The user defines a scene graph which represents the desired layout of the scene, and then the acceleration structure uses that data to produce a functionally equivalent scene graph with the nodes of its internal structure. Outside of adding new geometry to the scene through a scene graph, the process is automated and parallelized on a GPU with modern construction algorithms, allowing large scenes to exist with minimal overhead to the construction process. Rays can then query the data structure with the minimal amount of floating point error. Section 3 highlights the steps when modifying an acceleration structure with the preconditions brought forth in section 2.

2 Background and Related Work

Path tracing involves heavy use of acceleration structures to maintain efficient rendering speeds. Almost all performance benefits come from decreasing the construction and traversal times of such structures and any ray queries. Therefore to represent arbitrary scenes, a low overhead must be maintained compared to a vanilla implementation.

Acceleration structures and scene graphs have similar layouts over faces. They differ in what the inner nodes represent and in their creation process of the inner nodes. Scene graph nodes represent a transformation from one space into another with no restrictions and are created to be a minimum span over a needed space by a user. On the other hand, acceleration structure nodes represent a partial coordinate space constrained by its parent space and are automatically bounded to optimally produce fast intersection tests. By constraining the input scene graphs transforms to follow the rule that a parent must encapsulate its children, so that each child space is a subset of its parent space, only the representation of the contents must be explored before a connection between the two is made. Transition between nodes becomes an operation between a parent set and a child set, so by connecting the datatype that each node transitions between, the scene graph and acceleration structure can be unified.

The acceleration structure of choice is the bounding volume hierarchy (BVH). Compared to other acceleration structures, its benefits include lending well to insertions and deletions, the speed of its creation, and the parallel nature of its construction. Specifically,
the basis for a parallel linear BVH(LBVH) described by Lauterbach et al. [1] is used because it linearizes 3D space with a space filling curve. Given a coordinate, a computer system with limitless memory could accurately represent any point in a bounded space. The Z-order curve, or Morton code, is one of the faster space filling curves to generate and is created by interleaving the bits of n-dimensional integers into n offsets. Much like an octree, every set of n bits recursively defines a smaller space to represent.

Like Lauterbach et al. [1] before them, Garanzha et al. [2] and Pantaleoni and Luebke[3] also used a LBVH and implemented a bottom-up approach for its construction, which increased speed even further. Parallelization became so independent that the switch to the GPU with Karras[4] produced a massive performance increase with a bottom-up construction method. With the modified construction algorithm, fast tree creation came at the cost of traversal speed. This is described by Aila[5], solidifying the LBVH as the fastest BVH construction available with the traversal speeds nearing 50 to 60 percent of the golden standard, surface area heuristic construction. Apetrei[6] continued this trend and simplified the model to the maximum extent possible, making it even faster. This is the version of the algorithm used in this paper because with all this speed another important aspect of the LBVH construction emerged, the radix trie over bits. The radix trie contains a string of information that is inserted prior to each of its children, decreasing memory costs.

The LBVH expects the all the input geometry to be sorted by their Morton code. The sorted buffer is then the basis for choosing splits for the BVH. The splits themselves are chosen on the neighboring Morton codes and the highest differing bit that they have in common with the current node. The resulting tree thus respects the spatial locality of one triangle compared to all other triangles. The reason for the traversal performance disparity with the golden standard is that traversal performance is affected by tree balancing and not spatial locality. However, by retaining the property of spatial locality, the Morton code for a given face is now in relation to all other triangles across all n bits. This projects implementation expands upon the radix trie Apetrei[6] implements for their LBVH construction. The construction was implemented prior to this project.

3 Implementation

There are five different operational changes to the BVH pipeline necessary in order to allow arbitrary scene sizes to exist with low overhead. The first is the premise of the BVH traversal itself where distant objects are not accurately represented when performing any intersection routines. The following four deal with the BVHs representation between nodes and affect all facets of its use, Morton generation, sorting, tree construction, and traversal.

3.1 Ray Intersection Tests

As the error accumulates while traversing the scene graph, each ray is transformed into the current nodes coordinate space, eliminating any error from further traversal. The scene graph cannot exactly represent any point is space, but it represents a version of space that is indistinguishable from the desired partition of space. The structure uses locality to group triangles, and because humans use spatial cues to orient themselves in space, there is no perceptual difference between the two. However, there are errors in the data the rays collect through missed triangle hits. The classic Miller-Trumbore[7] technique is not watertight on triangle edges, so if the ray hits an edge exactly, it will continue as if no intersection occurred. This leads to situation where rays will pass through a triangle mesh, especially at distance, without an intersection, depositing light on the meshes inwards.

The proposed solution by Woop[8] lends well to each node having its own coordinate system because it translates the ray to the origin and aligns it to an axis, shearing the triangle in the process. The resulting intersection almost always hit. Two cases where it is not are perpendicular rays to the triangle face, and axis aligned bounding boxes(AABB) that don’t fit the translated triangle.

AABB intersection errors are solved by implementing a corrected AABB intersection routine that accounts for IEE754 floating point accuracy. Much like the triangle which is transformed for the aligned ray, the box is transformed. However, the box transformation additionally rounds each transformed coordinate to the next representable floating point number in its respective direction. Any errors in the transformation should thus be conservatively eliminated by the slightly larger than needed box. There are still a few misses in this projects implementation which are discussed further in section 6.

Figure 3: Edge intersection errors from the inside of a mesh. The fixed image is completely dark.

3.2 Morton Code Generation

A scene graph is initially given where each parent node encapsulates its children. The leaves of the scene graph each represent objects. In general, objects are created by an artist or are procedurally generated, leaving them in a coordinate space where they are represented as intended. The transformations between the root of the graph and its leaves must all be representable in their respective coordinate system. Apetrei[6] does not discuss their method for creating a radix trie from data so a method must be created to extract one from a scene graph.

Each of the scene graphs transformations are given a 64 bit integer to store the Morton code and created in a bottom-up fashion. The current node exists in the unit box its parent represents, so the Morton code can be calculated per face on the GPU for all the leaves.

Figure 2: A sphere too far away to be intersected with a pure floating point model. This leads to incorrect light emittance from the sphere.
in parallel. The remaining nodes are calculated on the CPU to easily avoid memory races. Each of the following nodes then use their transformation on the unit box’s min and max. The transformed min and max now reside at specific points inside the current nodes unit box. A Morton code is generated for both of these points and minimized by using an XOR operation and then removing the bits after the most significant bit that is set. As the two codes are guaranteed to represent all codes in between them bitwise, the common code between these two represent all between spaces conservatively. A transform then should be calculated from the expected and given bounds and then sent down the trie for more conservative restructuring until the trie leafs are processed, but it is left out of this implementation. This repeats for each parent node until the scene root is reached.

There are many errors introduced with the implement method. To fix this, the 64 bit integer would need to be condensed only to the bits it uses, and expanded when more are needed. A binary radix trie data structure and more are discussed in section 6.

Algorithm 1 Compute a 64 bit Morton code

```
1: procedure ENCODEMORTON(coord)
2:     max = 2097151
3:     x = coord.x * max
4:     y = coord.y * max
5:     z = coord.z * max
6:     answer = Split(x)|Split(y)<<1|Split(z)<<2
7:     Return answer
8: end procedure
9: procedure SPLIT(x)
10:     x = (x<<32)&0 fiz 20000000 f f f f
11:     x = (x<<16)&0 fiz 20000000 f f f f
12:     x = (x<<8)&0 fiz 10000000 f f 000 f
13:     x = (x<<4)&0 fiz 10000000 f f 000 f
14:     x = (x<<2)&0 fiz 30c30c30c30c30c3
15: Return x
16: end procedure
```

3.3 Morton Code Sorting

The fastest GPU sort available is the radix sort which is non-comparative and operates on keys with the same amount of bits. Unfortunately, the amount of bits used in the radix trie are variable. If this implementation used a shared trie, then only the childrens first bit would have to be compared for every node. Instead, as the trie operates on 64 bit blocks accounting for this problem at the cost of generation accuracy, a prefix sum can be used to isolate all the faces of the maximum depth and their respective indices in the global array of faces. Once they are isolated, the radix sort can be called upon the sub-array which sorts the key, the Morton code block, and its value, the face and its index. After the sub-array is sorted, the faces are scattered to their new position in the global array. Much like the radix sort which sorts a key by its least significant bit to its most significant bit, the process iterates from the least significant block to the most significant block, the root. Once this is completed all the faces are now sorted by their Morton code represented by the trie.

3.4 LBVH Construction

As described in section 2, the splits of the LBVH are created by the highest differing bit for a nodes neighbors. The final tree is constructed by keeping track of the left and right spans of each node and iterating the construction process until a single node spans the entire range of faces. As each split is recorded, the parent node constructs a bounding box over the boxes of each of its children. In the modified construction kernel, it is not a AABB which needs to be calculated, but a transform to each of the children. The radix trie describes the highest differing bit, but it also describes the bit sequence needed to get the next node. If the node being considered represents the unit cube, the Morton code can be decoded and mapped to the cube. The resulting box describes the output of the transformation which is then calculated and stored by the node.

```
Figure 4: The implemented radix trie
```

```
Figure 5: The desired radix tree
```

3.5 LBVH Traversal

At this point, the extracted scene graph embedded within the LBVH can be traversed. As each ray traversal starts with an origin, the camera needs to be presented as a face and be sorted along with the rest of the faces via insertion in the scene graph. Once traversal starts, all the rays know the location of the camera node and they traverse the BVH backwards until they reach the root. Backwards traversal does not modify the ray origin and precision will be lost at each traversed node. During backwards traversal, the opposite child from the one just visited is added to the stack with the current transformed array for visitation later. Once the root is reached, forward traversal can occur starting with the first node on the stack, the one closest to the camera. If the intersection is a success, the origin of the ray is moved to the boundary of the intersected plane so that all further transformations happen in local space. Traversal then continues as normal.

4 Results

Two test scenes are used to put the stability of the system under stress. The first has three objects which are an unscaled tree, a sun scaled and transformed a trillion units away from the tree, and a low polygon version of the Stanford Lucy model which is a nanometer wide and placed on a plane the tree rests on. This is the simplest test case as the sphere provides a light that should affect the visible geometry, and a smaller object which should not be visible. The second scene consists of five coloured spheres, of progressively smaller scales and translates, nearly aligned above the camera. Nearly aligned spheres, all larger than the cameras transform, allow for spatial cues to exist for the user, but each sphere should still be distinct from the others with no perceptual size difference between them.

Performance of the algorithms can be split into three categories, Morton generation, the creation of the LBVH which incorporates the sorting and construction algorithms, and the traversal stage. Morton code generation is done on the CPU after the initial face calculation on the GPU. The resulting time then accounts for both the memory transfers and the actual calculation. If the implementation for Morton generation was stable and implemented on the GPU,
Figure 6: The latest visible results of trying to run scene one. The transforms for each BVH node are the identity matrix showing that the algorithm works for no modified transforms. Yet there is still an accuracy error in the intersection code. All four models are clearly present.

Table 1: Performance results. Traversal times are only given for 720*1080 primary rays, not representing secondary rays and any foling traversals. Timings are ordered by Morton generation and LBVH Construction followed by Traversal speeds

<table>
<thead>
<tr>
<th>Scene</th>
<th>Unmodified</th>
<th>0.04ms</th>
<th>1.2ms</th>
<th>18.2ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scene One</td>
<td></td>
<td>0.97ms</td>
<td>1.8ms</td>
<td>42.6ms</td>
</tr>
<tr>
<td>Scene Two Unmodified</td>
<td></td>
<td>0.04ms</td>
<td>0.9ms</td>
<td>13.2ms</td>
</tr>
<tr>
<td>Scene Two</td>
<td></td>
<td>0.89ms</td>
<td>1.7ms</td>
<td>39.1ms</td>
</tr>
</tbody>
</table>

generation times would decrease even further. As the algorithm progressed up the scene graph and then down the radix tree, the length of generation is dependant on the depth of each data structure.

For both of the scenes, the construction of the LBVH is roughly 50% slower than the unmodified construction. This can be easily attributed to the increased amount of arithmetic calculations done and increased sorting count. The sorting time would also increase if the depth of the radix trie increases because of the serial trie sorts.

The results for ray traversal do not produce the expected visual results. The timings for that algorithm do not accurately reflect the timings that would occur if the Morton code encoding and decoding worked as intended. If that were the case, more AABBs would be hit at the traversal times would increase slightly due to the additional triangle intersections.

5 Future Work

Ideas that are presented here are flawed in their implementation and not optimized for stable use. Beyond this projects time frame, work will be done to fix bugs and implement new features as described here and in section 6.

5.1 LBVH

There are further LBVH optimizations provided by Karras and Aila [8] that decrease the construction speed, but increase the traversal speed to 97 percent of the golden standard, providing a better construction/traversal ratio compared to overall efficiency than previous implementations. Some of these optimizations such as tree restructuring would remove the spatial relation property, but could further be considered for half-measures. Additionally, this LBVH implementation would drastically benefit from triangle splitting according to a heuristic. When a large triangle and many small triangles are close together, the error between the triangles transforms would be proportional to their sizes. Any reduction in size between two close triangles would therefore increase ray accuracy in addition to the traversal gains explored by the paper.

5.2 Intersection

Woop ray/AABB intersection tests are too slow to be considered a long-term solution to offset bounding boxes. Additional effort needs to be applied to CUDA kernel occupancy and divergence cases to mask the increased register usage that the method introduces.

5.3 Radix Trie

All of these changes consider a static scene only. To implement a dynamic scene and arbitrarily timed object removals and additions to the scene, the radix trie needs to be updated on a per tick basis. New scene graphs need to be blended with the previous ticks trie. To solve moving faces, a new kernel could be created and percolate a changed Morton code up and down the current trie.

Changing the radix trie into a true binary radix trie would increase the performance of all algorithms accessing its contents, and even
its generation. Errors caused by the original implementation would also change.

6 Errata

The creation of this project was during three weeks of undergraduate schooling and reflects 100+ hours of work and many more theorizing. One of the greatest challenges was coming up with a way to extract a scene graph with many more nodes from a user defined scene graph. With this in mind, there are many future changes that would create an optimal and stable solution to what is presented. Many implementation choices were forced by their ease of creation and not their efficiency, which ultimately ended up with code that did not perform as expected.

All the errors shown with scene one occur with scene two, but because all the spheres are translated away, there is only one uninteresting white sphere.

Figure 9: An error where only some transformed nodes are being selected and others are failing to be transformed properly. Additionally, the Lucy and sphere models are not being visited.

Figure 10: Expected scene one results

6.1 Intersection

There is an issue with the intersection implementation as, very rarely, there are missed hits and colour bleeding inside a mesh despite the source materiel stating that no hits were recorded in their implementation.

6.2 Radix Trie

The radix trie as implemented is not an optimal trie as each node is aligned to a 64 bit integer, reducing the amount of shared memory, and introducing error in the Morton code due to under-utilized or over-utilized bits. Additionally, each object duplicates this trie to avoid implementing another traversal algorithm over a new data structure. This limits radix trie traversal to a shared, per object operation, not a shared, global operation. Therefore there is no percolation of data down the radix trie as the differences in the conservative bounds and the original bounds represented by the scene graph transform are not properly being recorded between bit boundaries, leading to missed hits on some AABBs during traversal and also misrepresented sizes. The biggest factor in this is no doubt the 64 bit integer not being an arbitrarily sized array of bits as it should be. Indeed, any bit traversals that transition from one integer set into the next suffer from accuracy and implementation bugs due to a forced alignment for easy sorting and GPU traversal.

Acknowledgements

Thank you Professor Cutler for letting me approach a problem not feasibly solved in three weeks and for the joy and knowledge I gained from trying.

References


