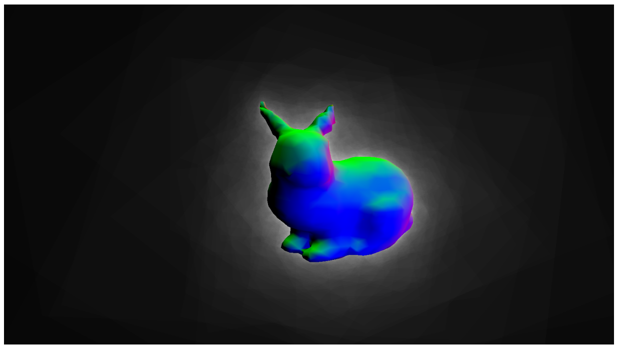
Agglomerative Building and Traversal of an Object Aligned Bounding Volume Heirarchy

Dan Aston 300406690

October 2018



Abstract

This report comprises and investigation of the topic of acceleration structures, specifically bounding volume heirarchies for real time raytracing, and covers an implementation of a provided example using a simple bottom up agglomerative technique based on object aligned bounding boxes (OABBs).

The design challenge is discussed in two aspects: designing a structure and algorithm for an efficient ray traversal, and designing an efficient algorithm for building this structure. For both aspects, precedents are discussed, solutions are proposed, and our implementation is summarised.

We will compare the performance of this technique to prior work, and discuss how they perform differently. We then propose directions for further research, such as alternative abstraction hulls, multi trees, and different heuristics to assist the build process. We will propose and discuss a parallelised version of the agglomerative build process.

Design Discussion

Traversal

Raytracing is a rendering technique that is agnostic to screen space, and the view frustrum.

A ray may be cast in any direction, originating from any point. In the screenspace world, one knows an eye ray intersects a surface at depth t by sampler the z-buffer only after all primitives have been rasterised. This is convenient if the primitives had to be rasterised anyway.

What this tells us is that the length of the ray depends on some organisation of the scene that depends on the ray's origin. This means that for each ray origin, there is a different set of salient scene features to be processed in order to find the depth of its closest intersection.

The direct analogy to rasterisation in the context of ray tracing is checking the intersection against all primitives. For a single ray, the scene cost is O(n). With rays requiring multiple samples, and sampling algorithms budgeted to tens of steps, this is prohibitive for a scene with an expressive level of detail.

The performance constraints should be fine, given that a single ray will only have a handful of intersections with the scene.

Acceleration structures

For a given ray, we only care about the primitives it intersects. What is needed is a means by which to organise a scene in a way that abstracts non salient primitives, based on the origin and orientation of the ray.

We want to eliminate as many non intersecting primitives as possible with as small a number of checks as possible. If a ray is processing something that will not provide a termination event, we want to know that it wont as soon as possible.

Topology

Scene objects are often thought of as a heirarchy of features. This higher level description is more useful to a ray than a rasteriser. The projection of geometry onto a surface is not useful to a ray, it is only interested in the events that occur along its path.

The higher level descriptions of geometry may take a range of different forms and be grouped in different ways. However, these descriptions introduce a range of possible cases, and ray tracing is all about eliminating possibilities.

On the one hand, we want a data structure to be as short a tree as possible. If multiple cores are working on one ray, a multi tree would allow an earlier dispatch of threads.

On the other hand, in the context of a ray cast by a single thread , the depth of the tree is irrelevant. We just want as few checks as possible, with minimal branching.

For the investigative purposes of this project, fragment shaders already have acceptable thread occupancy. We propose an acceleration structure with the topology of a simple binary tree, with a single branch to distinguish leaf nodes and inner nodes.

Abstractions

We introduce the concept of an "abstraction hull". An abstraction hull (or in this context just, "hull") is something that is easier to compute an intersection with than the set of its contents.

Surface Area and Emptiness

Hulls only save computation when they are missed, and when testing them is cheaper than testing their contents. The more "emptiness" there is within a given node, the more likely it is that it will escape without more checks. A good BVH maximises ray thoroughfare.

While checking within an intersected containing hull hA, the likelyhood that a child hull hC will intersect the ray is proportional to the area of the child's profile as projected onto the image plane divided by the area of the profile of the parent.

A BVH that contains all scene geometry can be simpler but not emptier than the scene itself. A good BVH simplifies the scene while filling it out as little as possible.

The literature on BVHs makes frequent reference to the surface area heuristic (SAH). The idea is that the best possible hull to enclose a set is the one with minimum surface area. It is worth noting that this is not the same concept as maximising the likelyhood of early escape, but it is a useful concept that balances the requirements of simple abstractions with conservation of emptiness.

Slab, k-DOPs, Spheres

The presented implementation uses slab, which is a well known technique for defining bounding boxes. It can also be generalised to arbitrary convex hulls.

If a shape is convex, each intersecting ray will have one entry and one exit point. Entry points will have normals facing towards the ray's origin, and vice versa. Therefore, faces may be divided into entry and exit planes depending on the dot product between their normals and the ray direction.

If a ray misses the hull, at least one pair of these planes will the "flipped", and the nearest exit point will be closer than the furtherest entry.

Otherwise, the furtherest entry point is a close valid hit.

AABB drawbacks

There are two things AABBs struggle with: large polygons, and diagonal surfaces.

Consider the case of a corrugated iron roof. There are many long triangles along each wave of the iron, and each pushes out the corners of their bounding boxes far beyond the plane of the roof's panel.

Even if they are subdivided, the resulting parent nodes will take up a needless amount of negative space in the direction normal to the roof.

Rays that glance along the surface of the roof tangentially will intersect bounding boxes as if they were participating media, which is far costlier than testing against an aligned abstraction.

Traversal pseudocode

The traversal loop is written here in a more OO friendly pseudocode.

There are two phases. No intersection tests are made until visiting the node that contains the origin, although siblings are pushed to the stack on the way.

```
bestHit = ESCAPE;
// PUSH TO ORIGIN PHASE /////////////////////////////////
    while( boxcontains (visit,r.o) ){
      //triangle case
      if (topo[visit-1] == -1)  {break;}
      //parent node case
      visitStack[stackCounter] = topo[visit*2];
      stackCounter ++;
      visit = topo[visit*2-1];
    }
```

Once the stack has been preloaded with the siblings along the path to the containing node, the node itself is added to the stack so that its children can be tested first. Then, the array tries to escape the stack through intersection tests.

```
   // INTERSECTION PHASE //////////////////////////////////
    for (int  STEP = 0; STEP < MAX_TRAVERSAL_STEPS; STEP++
      if (stackCounter<=0) break;
      visit = visitStack[-1+stackCounter--];

      slabBox(visit,r);
      if (!theBvhHit.hits) continue;
      if (theBvhHit.t > leafRay.bestHit.t){
        continue;
      }

      leafRay.boxAccum += 0.03; // Visualise the BVH

      // Primitive case
      if (topo[visit*2-1] < 0){
      testTriIndex(r,topo[(visit)*2]);
        if( theTri.hits
            && theTri.t < leafRay.bestHit.t //){
            && theTri.t > 0){
          leafRay.bestHit.hits=true;
          leafRay.bestHit = theTri;
        }
      continue;
      }
      unsigned int small = topo[visit * 2 - 1];
      unsigned int large = topo[visit * 2 ];//+ 1];
      visitStack[stackCounter++] = small;
      visitStack[stackCounter++] = large;
     }
} // Traversal escaped the tree. Got the best hit.
```

As you can see, the traversal algorithm is very simple. Most of the work here is going into intersection checks against the bounding volumes with slabBox(), and the primitive triangles with testTriIndex();

Memory usage

The current implementation requires significant state for traversal. The biggest part of it is the visit stack. Each stack destination uses an unsigned int, and there are many. There is low hanging fruit here. The binary topology will allow traversal using a bitstack instead of an unsigned integer stack.

Instead of absolute addresses, the traversal can be plotted as a string of binaries signifying directions on the stack. The stack counter can tell the visitor which bit to look up, and the bit means whether or not the second sibling has been visited yet.

Building

Top-Down vs Bottom-up

There are top down techniques, and there are bottom up techniques. Our implementation is a bottom up technique.

This is a useful dichotomy because both perspectives provide different intuitions for parallelisation.

A top down technique begins with the bounding hull of the entire scene and creates leaves through subdivision.

In the planning of this project, we identified the child hulls as dependencies for that of the parent. No hull can be finalised without its children's hulls also being finalised. Therefore, we build the heirarchy from the bottom up.

Pre sorting

The tradeoff with bottom up approaches is that you start off with each primitive having every other primitive as possible hull siblings. Efficient building neccessitates sorting the geometry into a structure to facilitate a faster nearest neighbor search, which is a whole different preprocessing step.

Morton coding

A fast way to do this is to put all primitives into a 3D space filling Z-order curve. The are well documented techniques for doing this with bitwise operators. The positions of primitives are quantised to an integer representation, and the bits of each dimensional component are simply interleaved to write the position in the curve.

When the geometry is sorted, primitives can safely eliminate most of the scene from checks, avoiding O(nˆ2)

Summary

The implemented BVH is an oriented bounding box volume heirarchy (OBBVH), built using an agglomerative technique.

Building happens on the CPU, and the OBBVH is sent to the gpu into a set of shader storage buffer objects.

The current builder implementation is very brute force. It uses a while loop with no nearest neighbor search.

The build has three stages:

- Primitive node creation
- Topology creation loop
- Topology string rendering

First, a vector for leaf nodes is filled with a new node containing each primitive in the scene. In contrast to the subdivision of the axis aligned techniques, here each primitive occupies only one leaf.

Then the topology is built. The loop has two phases:

- Picking phase
- Matching phase

In the first, each loose node checks every other loose node for the best potential match. The best match is one with whom the resulting bounding box containing both nodes would be the smallest.

Nodes who propose hulls hold pointers to the nodes they would like to partner with in a tiny home and save copies of the specifications of the hull to share. Proposees recieve proposals during their turn in the loop and update their match pointer and copy of the hull plan to the best proposition. Nodes make and recieve proposals in the same pass.

Once nodes are satisfied that they have chosen the best candidate, a matching pass iteratively goes through the list, constructing new nodes from the indices of mutually matching pairs, and the bounding hull that they agree on.

Matching nodes are marked for two reasons. It means that the loop will not create multiple copies of the same parent node upon subsequent visits to matched nodes. It also means prospective testers can skip them.

The number of free primitives is known at the beginning of the build. This count is decremented upon each union, and the loop ends when it is equal to 1.

Results

Test Hardware

These results were recorded on this system:

i5 4690k Gigabyte z-97n Gaming 5 16GB DDR3 PC3-10600 Zotac GTX 660 3GB

All hardware running standard voltages and clock speeds.

Building

The single threaded brute force build is, as expected, comically slow.

It builds the 80 triangle ball mesh in 2 seconds. To load a 4000 triangle mesh, it takes several minutes.

BVH Quality

Enough encapsulation is happening to facilitate a consistent order of magnitude performance benefit over the n^2 brute force case with no culling.

It is obvious that improvements can be made to the efficiency of OBB construction. For one thing, the builder needs to find better bases for inner nodes. Currently all parent nodes are in the tree are aligned to the basis of some particular leaf node, instead of the overall basis of the containing set. Where unused far corners push out the centres of the faces of their parents, where their parents bases' are relatively diagonal.

This eats up a lot of negative space. Thankfully, finding a better basis for each parent will be trivial, and there are plenty of ways to cull corners.

Traversal

Framerates:

Rendering performs at a usable interactive rate. Framerates drop when the camera is very close to an object. This may be related to the average escape time

Further investigation

The building algorithm can be parallelised. There is probably a range of ways to do this. Stages of the matching process could be divided into seperate dispatches to make synchronisation simple, or a single pass algorithm could be implemented.

The conceptual simplicity afforded by splitting the build phases into seperate dispatches, introduces code complexity for managing buffers. Moving data back to the cpu just coordinate matching would introduce unneccessary overhead. The matching stage as it currently exists rewrites the list of free nodes after each pass, which would neccessitate pausing all threads to wait for a linear operation, whether on the cpu or the gpu.

We propose a single pass algorithm. With some careful synchronisation, the process could turn out to be simpler.

We know that creation of inner nodes decrements the amount of unmarried subtrees. Therefore, we know that if an algorithm associates one thread per node for the purpose of match testing, never will there need be more than N such threads.

A seperate matching thread may match and mark nodes, and add nodes to the list.

Once a test thread has have checked all candidates to match its node and picked a match, they may go into a spinner loop, where they wait for the wanted match to check their node.

If it is then rejected, the thread can continue looking for the next best match until it is certain it has found the next best thing.

If the match is mutual, the matching thread marks them both and constructs their parent node. It also decides that one thread terminates and the other gets to operate the parent.

The matching thread can asynchronously search for mutually matched pairs to mark and construct from. Both parties must have picked one another from the full set of free nodes that existed when they started looking.

Because new nodes can only be composed of nodes in the current list, they will be at least as large as the smallest hull in the list. After a node has measured the size of the list and commenced testing, no nodes appended to the set in that time will be better than the best one it finds in the original set.

However, new nodes may still be better than many of the older ones.

Reasoning about this is made coherent by ranking favourites.

If a favourite choice f(n), and the next favourite choice f(n+1) match, its certain that the resulting hull will be no smaller than f(n+1), but its possible that their containing hull might still be a better candidate than favourite choice f(n+2) from the original list.

Without storing favourites in memory, upon rejection, nodes must begin their searches from the start. They will be able to skip nodes which have been marked as matched but will not recognise nodes that are still single but have already rejected them, and they will waste time testing one another. If they maintain a sorted list of favourites from a given list up to a certain index, upon rejection, they will only to test the newly created nodes against the favourites, while updating the favourites.

It will be a detail for investigation to figure out the optimum window size, and method of sorting favourites.

Conclusion

We have discussed the purpose, and design challenge of acceleration structures.

We have implemented a simple object aligned bounding volume heirarchy with a single threaded build process.

We have shown how there are a range of possible convex abstraction hulls, which perform differently, and have discussed the diagonal case.