Adaptive Collapsing on Bounding Volume Hierarchies for Ray-Tracing

A. Susano Pinto

Abstract

Ray tracing is a computationally intensive process, several tree data structures and heuristics have been developed to optimize it. This paper presents a new heuristic in the area, based on collapsing some nodes in order to achieve a smaller expected number of node-tests. Two ways of using this heuristic in Bounding Volume Hierarchies are presented as well as the cost-model used to drive the heuristic development and measure it's efficiency. Some procedures on integrating this heuristic with other optimizations are also discussed.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

There has been a lot of research on optimizing tree data structures for making ray tracing faster, such as Bounding Volumes Hierarchies (BVH) which speed-up ray trace queries by using an object space partition tree.

On BVH trees, each node has a bounding volume that encloses the bounding volumes of all its childrens. Those bounding volumes are often axis aligned to reduce the computational cost of testing ray hits. The hierarchic tree structure is used to reduce the amount of primitive and bounding volume tests needed in each trace, and has a great impact on the pruning ability of the tree. The structure is usually constructed by top-down and bottom-up greedy methods. No optimal method is know for this generic construction problem [NT03].

One of the methods used to build a BVH is the top-down greedy construction with surface area heuristic (SAH [GS87]) which can be built in $N \log N$. The final result of that construction method is a binary search tree, which is sometimes flattened to a quad [DHK08] [EG08] or to 16-way trees [WBB08] to take advantage of SIMD.

In this paper, a new heuristic method to further optimize already constructed BVH trees will be presented. It is based on a generic version of BVH where each node can have a variable number of childrens and uses that to achieve a tree structure expected to work better than it's original. Nonetheless the methods and heuristics developed can also be used to enhance fixed-width trees as shown in Section 4.3 and can be adapted to take advantage of SIMD.

1.1. RayCast on Bounding Volume Hierarchies

The process of ray tracing, or determining ray hits on a BVH can be seen as a simple recursive process:

Algorithm 1 *raytrace*(*ray*, *node*)

```
1: if test_hit(ray, node.bounding_volume) then
2:     test_primitives(ray, node)
3:     for all node.childs do
4:         raytrace(ray, child)
5:     end for
6: end if
```

The core methods of this recursive transversal are *test_primitives*() and *test_hit*(). The presented heuristic focuses on the *test_hit*() and tries to optimize the usefulness of each call to that method by deciding which of those calls can be removed.

2. Cost model

A cost function is used to measure the expected computational cost and assess the efficiency of a given structure. To easy that process it was assumed that all rays hit the scene enclosing bounding volume; only complete ray hit queries are used, that is: all primitives hitting a given ray have to be

found; and all primitives are seen as enclosed by a bounding volume, disallowing any deletion or modification on leaf level nodes.

Let $P_{hit}(A|A_{parent})$ denote the probability of a ray hitting node A knowing it hits A_{parent}, which in the case of a BVH structure represents the probability of needing to test childrens of node A for ray hits, knowing that ray hits A_{parent}. This function is expected to scale up to grandparents under the form: $P_{hit}(A|C) = P_{hit}(A|B).P_{hit}(B|C)$.

Let $T_{cost}(A)$ denote the computational cost to test if a given ray hits node A, ie: number of instructions needed to test ray hits against A bounding volume.

Using the previous functions it's possible to define a ray trace cost function, which can be used to analyze the expected cost of ray tracing on a given node A:

$$R_{cost}(A) = \quad (T_{cost}(child) + P_{hit}(child|A).R_{cost}(child))$$

Considering only complete ray hit queries are used and P_{hit} is exact with a given set of N rays then $N * R_{cost}(root)$ would measure the exact cost of casting the set of rays on *root*. Therefore any method capable of reducing $R_{cost}(root)$ would speed-up the tracing of the given set of rays on *root*.

It is desired that P_{hit} exactly matches the probability obtained if the set of rays that is going to be traced is known in advance. But that is impractical, so heuristics have to be used.

2.1. Surface Area Heuristic

A simple method to approximate $P_{hit}(A|parent_A)$ is given by the Surface Area Heuristic [GS87], where $SA(n)$ denotes the bounding box surface area of a node:

$$P_{hit}(child|parent) = \frac{SA(child)}{SA(parent)}$$

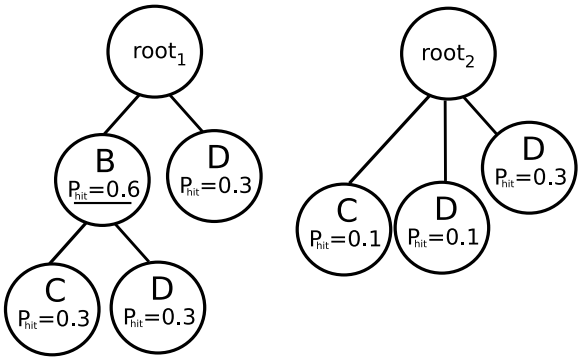
3. Ray/Node hit test

When traversing a BVH (Section 1.1) the function *test_hit*() tests whether a ray hits a bounding volume. This function is often intensive in terms of floating point operations and memory access and is responsible for pruning ray traces when it returns false.

False positives on that function can be tolerated as that does not affects the final result. And although a false positive leads to unnecessary test on a node's childrens, it can actually be used to reduce the expected number of *test_hit* calls. As shown on example at Figure 1, where $R_{cost}(root_1) = 3.2$ and $R_{cost}(root_2) = 3$.

That type of case is of great interest to optimize the tree structure and in the following sections the reason for their existence is discussed and a condition to determine those cases according to the presented model is showed.

Figure 1: *Collapsing node example, showing removing nodes can reduce the expected number of ray-nodes tests. The tree on the right was obtained by removing node B from the left one.*



3.1. Collapsing nodes

Often the value of $P_{hit}(A|A_{parent})$ for a given node is large, showing that the tree organization is introducing a *test_hit* call in which the transversal algorithm rarely can deduce useful information for pruning the search.

This type of case especially happens in binary BVHs, where the primitives belonging to a node are only split in two, generating large child volumes and as a consequence a large P_{hit}. Incorrectly chosen split points can also lead to this effect by creating large child bounding volumes.

3.2. Collapse-node condition

Based on the presented cost model, the nodes that can be collapsed with the objective of decreasing the expected R_{cost} (Section 2) can be detected by the condition:

$$R_{cost}(A) + T_{cost}(A) > \quad (T_{cost}(c) + P_{hit}(c|A_{parent}).R_{cost}(c))$$

4. Method

After identifying an operation that can reduce a tree R_{cost}, an automatic process of applying that operation until a tree no longer can be optimized can be found. Two such optimization algorithms are presented.

The first, a greedy approach, reachs a local minimum in a single pass over the tree. The second tries to find the global minimum for R_{cost} that can be achieved by only applying collapsing nodes, which can be seen as merging its childs on the node's parent. The efficiency of both methods is later compared in Section 5.

4.1. Greedy top-down method

Assuming T_{cost} is constant the condition presented on Section 3.2 can be reduced into:

$$1 + P_{hit}(A|A_{parent}).A_{nchilds} >= A_{nchilds}$$

Being possible to calculate this condition locally a top-down greedy method for optimizing a given BVH tree can be created (Algorithm 2). This algorithm recursively collapses nodes matching the presented condition. Although non-optimal, its linear run-time and reduced memory usage may be desirable in many cases.

Algorithm 2 $pushup(node)$

```
1: q ← queue()
2: append all node.childs on q
3: clear node.childs
4: while not q.empty do
5:    child ← q.pop()
6:    subchilds ← child.nchilds
7:    if 1 + P_hit(child|node).subchilds >= subchilds then
8:       append all child.childs on q
9:    else
10:      pushup(child)
11:      append child on node.childs
12:   end if
13: end while
```

4.2. Optimal Collapse-Node method

This method tries to find the minimum R_{cost} possible to reach on a tree by only collapsing nodes on it. The optimized tree can then be built by back-tracking.

For that, $R_{cost}(node)$ function is changed into $R_{cost}(node, size)$ which represents the minimum cost needed for a sub tree rooted in *node* with *size* entry points, or in order words, it finds a tree-cut not greater than *size* where $P_{hit}(cut\_node|node).R_{cost}(cut\_node)$ is smallest. This transformation allows the use of a dynamic-programming approach to efficiently find a global minimum value for R_{cost} as showed on Algorithm 3.

Due to efficiency reasons, the maximum number of childrens a given node can have has to be fixed at *MAX_CHILDS*, but that is believed to not have a big impact. Under this the algorithm can run in $2N.MAX\_CHILDS^2$.

To allow the algorithm to adapt to group testing, a useful feature under SIMD instructions, $T\_cost(N_{nodes})$ is used to represent the computational cost of simultaneously testing if a ray hits N different nodes. As an example, on a 4-SIMD machine $test\_group4$ coud be used as cost function.

$$test\_group4(N_{nodes}) = \left\lceil \frac{N_{nodes}}{4} \right\rceil$$

The presented Algorithm 3 can be extended to work with non-binary trees as input, but that is outside the scope of this paper, as most build methods generate binary trees. Nonetheless, a working version for optimizing variable width trees and with back-tracking to reconstruct the optimal tree is available for download (`http://andresp.no-ip.org/page/EG2010/`).

Algorithm 3 $R_{cost}(node, cutsize)$

```
1: if is_leaf(node) then return T_cost(1)
2: if not calculated cost_memoization[node] then
3:    cost ← array[]
4:    cost[1] ← ∞
5:    for t in (2..MAX_CHILDS) do
6:       cost[t] ← ∞
7:       for i in (1..t − 1) do
8:          r = R_cost(node.left, i) + R_cost(node.right, t − i)
9:          cost[t] ← min(cost[t], r)
10:      end for
11:      cost[1] ← min(cost[1], T_cost(t) + cost[t])
12:   end for
13:   cost_memoization[node] ← cost
14: end if
15: return min(cost_memoization[node][1..cutsize])
```

4.3. Easy adaptation to fixed-width BVH

One of the negative sides of applying this heuristic by erasing a node is the creation of nodes with a variable number of childrens. This can increase the complexity of implementing optimizations tightly related to the concept of a binary tree, for example, a search order heuristic. It's suggested that in those cases, a flag can be used on each node indicating where a hit test should be performed or should be skipped by assuming it would return true, thus allowing the tree to keep it's original structure.

5. Results

To aid measurement of the efficiency of the developed methods, 4 different types of BVH and 3 variables were used.

5.1. Data structures

default binary BVH tree built using a SAH heuristic
STgreedy BVH built from *default* using method in Section 4.1
QBVH 4-ways BVH tree built from *default* by colapsing odd depth-levels of the tree
STdp BVH built from *default* using method in Section 4.2 with $MAX\_CHILDS = 15$.

5.2. Variables

Tree size the number of nodes needed for the tree
Rcost the value of $R_{cost}(root)$ using $T_{cost}(N) = N$
BB tests/ray the number of BB-tests per ray

The results were obtained by rendering two scenes, using ray trace only on secondary rays. On all trace queries, all hits were found even when only one or the closest hit was needed. It's important to recall that according to the original constraints, a node could only have one primitive, this leads to none of leaves BB be a candidate to collapse.

Figure 2: *Bottle Collection: 1077K faces, 77M rays, ∼16 primitive tests/ray.*

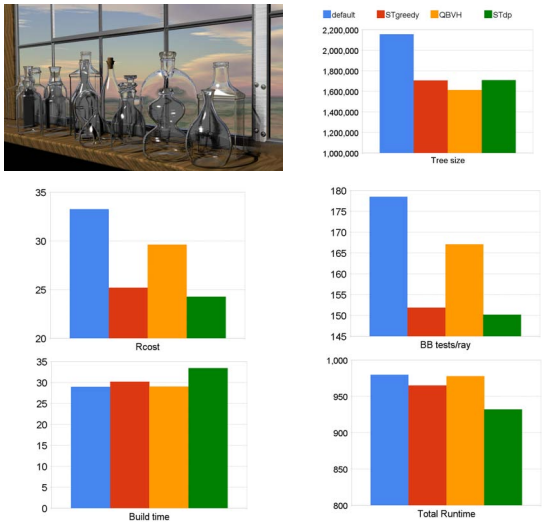
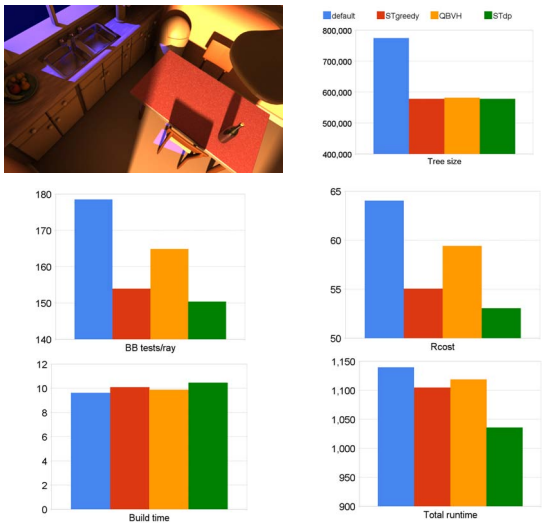


Figure 3: *Kitchen: 387K faces, 91M rays, ∼20 primitive tests/ray.*



The results in Figure 2 and Figure 3 show that reducing R_{cost} does have an effect on the number of hit tests needed during a ray trace. It also shows that part of the *QBVH* performance does not come from taking advantage of *SIMD* but by actually using wider trees, being that the type of advantage that $STgreedy$ and $STdp$ are able to better explore. Besides the presented algorithms also lead to small trees, bringing both smaller memory bandwidth and memory footprint.

6. Conclusion and Future Work

This paper introduces a new heuristic that achieves a reduced expected number of node tests on ray trace queries by detecting which nodes on a tree should be collapsed. It was already known that collapsing nodes could lead to an efficiency increase but there were lack of methods to detect and apply that heuristic. Two such methods were presented.

This study shows that according to a cost model, an optimal selection of nodes to be collapsed can be found in polynomial time and memory and that in practice that algorithm is linear in the number of nodes. With the proper T_{cost} function this method allows exploiting *SIMD* instructions.

The impact of using a heuristic more exact than SAH is still unknown, although that can be studied by using an exact heuristic (ie: by performing ray trace twice), and if worthwhile some time can be spent on developing more accurate or live adaptable P_{hit} heuristics.

Although this heuristic has been presented with the main focus on BVH data structures it's expected that the general idea can be implemented on other hierarchical search problems.

Finally, precautions were taken and discussed to make sure the introduced methods can be easily applied with other algorithms, and in that sense the author expects to have contributed to an easy integration with them.

7. Acknowledgments

The author would like to thank the *Google Summer of Code™ 2009* project, during which he was able to study, develop and implement the ideas presented in this paper. And also to the *Blender Foundation* and it's community which provided the scenes to test and a code base to develop in.

References

[DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. In *Computer Graphics Forum (Proc. 19th Eurographics Symposium on Rendering)* (2008), pp. 1225–1234.

[EG08] ERNST M., GREINER G.: Multi bounding volume hierarchies. In *IEEE Symposium on Interactive Ray Tracing, 2008. RT 2008* (2008), pp. 35–40.

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5 (1987), 14–20.

[NT03] NG K., TRIFONOV B.: Automatic bounding volume hierarchy generation using stochastic search methods. In *CPSC532D Mini-WorkshopâĂİ Stochastic Search Algorithms* (2003), Citeseer.

[WBB08] WALD I., BENTHIN C., BOULOS S.: Getting rid of packets-Efficient SIMD single-ray traversal using multi-branching BVHs. In *IEEE Symposium on Interactive Ray Tracing, 2008. RT 2008* (2008), pp. 49–57.