An Improved Multi-Level Raytracing Algorithm

Joshua Barczak

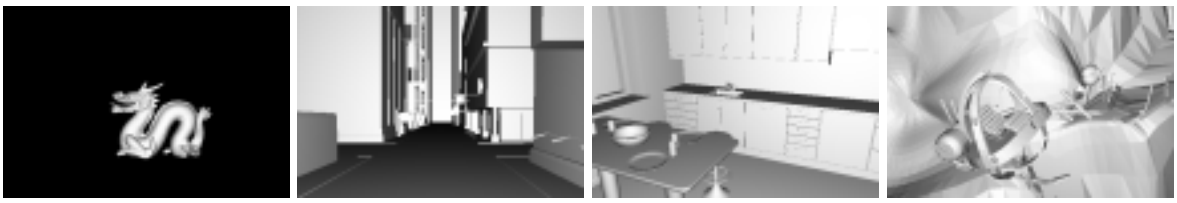

Figure 1: Renderings of our test scenes. On average, our method eliminates 6%, 69%, 63%, and 56% of BVH traversals for these scenes.

Abstract

Previous work has used entry point search (EP search) to accelerate raytracing by avoiding redundant traversal of nodes in a KD tree. EP search works by identifying a node which may be used as a starting point for tracing a particular group of coherent rays, thus bypassing all nodes above it. This is most effective when all rays can be shown to be occluded, since a leaf or low inner node can often be used as the entry point. Unfortunately, occlusion detection is difficult in scenes with complex geometry, and is particularly difficult for BVH data structures, for which ordered traversal cannot be guaranteed. If occlusion cannot be considered, EP search alone becomes considerably less effective. We present an alternative to EP search which eliminates traversals above *and below* the entry point node. Compared to EP search, our method has a similar runtime cost, generally eliminates more nodes, never eliminates fewer nodes, and is more effective in the absence of occlusion. Although designed for BVHs, our algorithm can be easily applied to other tree structures.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism, Ray Tracing—; I.3.3 [Graphics Systems]: Picture/Image Generation Display algorithms—;

Keywords: Ray Tracing, Frustum Culling, Entry Point Search, Path Compression

1 Introduction

Raytracing is a very important technique in computer graphics, which has seen a surge of exciting research activity in recent years. It provides a framework for simulating many diverse visual phenomena, and it can easily accomodate a wide variety of surface representations. The fundamental operation in raytracing is determining the first intersection point between a ray and a geometric primitive in a 3D scene. This is often done by traversing a hierarchical data structure containing the scene geometry. Raytracing in this manner scales logarithmically with scene complexity, making it a favorable choice for rendering scenes containing large numbers of primitives.

Wald et al. [2001; 2003] introduced the use of SIMD instructions to simultaneously trace multiple rays through a KD tree, which made interactive raytracing feasible on multi-processor desktop machines. Subsequent work [Dmitriev et al. 2004; Reshetov et al. 2005; Boulos et al. 2006; Wald et al. 2007] yielded a rich class of *packet tracing* techniques, whereby coherent groups of rays are simultaneously traversed through an acceleration structure. In addition to the use of SIMD instructions to vectorize calculations, current techniques employ packet-level tests to *eliminate* calculations, which are often most effective when applied to packets much larger than the SIMD width.

Whereas packet tracing seeks to amortize or cull *ray-level* operations, a more recent idea is to cull *packet-level* calculations using a pre-pass through the tree. This was introduced in the form of *entry point search*, (EP search) [Reshetov et al. 2005], and the basic idea is to start traversal at a node other than the root, thus traversing fewer nodes. Our work began as an attempt to apply this idea to BVH data structures, but has resulted in an entirely different, and more effective algorithm.

2 Prior work

Although KD-trees [Fussell et al. 1988] were the preferred raytracing data structure in earlier times, the bounding volume hierarchy or BVH [Rubin and Whitted 1980] has become increasingly popular. Recent results demonstrate that BVH packet tracing acheives comparable performance to a KD-tree, and is much more flexible in its ability to handle dynamic scenes [Wald et al. 2007; Lauterbach et al. 2006; Yoon et al. 2007]. Packet traversal schemes for BVHs were presented by Wald et al. [2007] and Lauterbach et al. [2006] at roughly the same time. Although BVH traversal requires more computation at each node, the trees generally are shorter and contain fewer nodes than the equivalent KD trees. A key drawback of a BVH is that the possibility of overlapping bounding volumes makes it is impossible to guarantee that the first intersection point that is found is the first one struck by the ray.

Reshetov et al. [2005] introduced entry point (EP) search, which is a very important precursor to our work. EP search uses the bounding frusta for large groups of rays in order to locate an entry point node, which is then used as a starting point for tracing the rays in the tile. The node selected as the entry point is the lowest common ancestor of all (non-empty) leaf nodes which are intersected by the bounding frustum. By starting traversal at this node, traversal calculations on its ancestors can be avoided.

Quite recently, Fowler et al. [2009] proposed a modification to EP search in which the candidate entry points are collected into a queue and then tested in top-down, rather than bottom-up order. This re-

duces the number of nodes visited while locating the entry point, while still returning the same answer (given the same set of visited leaves).

Havran and Bitner [2000] used a longest common traversal sequence (LCTS), which is a list of non-empty leaf nodes in a BSP tree which are always traversed in the same order by all rays in a frustum. Rays are tested against the LCTS first, and need not traverse the full BSP tree if an LCTS hit is found. Our technique is a distant relative of this approach, in that we also compute subsets of the main acceleration structure during rendering.

3 Exposition

We will begin by identifying some important limitations of EP search. These are most obvious in the context of BVH data structures, but they can apply to KD trees as well. Using EP search as a starting point, we will then derive an improved algorithm which is more robust with respect to these limitations.

3.1 Limitations of Entry Point Search

EP search is most effective when it can be determined that the entire set of rays is completely covered by large triangles, or when its bounding frustum is completely inside a closed object. In these situations, little or no traversal need be performed, because the majority of the nodes can be shown to be occluded and leaf nodes are often selected as the entry points. Unfortunately, the former test is only effective in a limited class of scenes (those containing large, flat triangles), and the latter requires a more involved tree construction process, which complicates the implementation and limits its utility for dynamic scenes. It is also problematic to apply either test to BVH data structures, which cannot guarantee depth-ordered traversal, and which do not explicitly represent the empty space inside of hollow objects.

In the absence of reliable occlusion detection, all nodes which intersect the frustum must be considered when choosing the entry point. This severely compromises the effectiveness of EP search, because it tends to confine the set of possible entry points to the uppermost levels of the tree. Previous work [Benthin 2006; Fowler et al. 2009] has observed this effect at work in complex scenes. In the worst case, any frustum which includes geometry on both sides of the topmost splitting plane will *never* benefit from EP search, despite incurring the overhead of a tree walk. Unfortunately, this worst case can be very common, as it occurs whenever the viewer looks towards the center of the model.

Even when occlusion can be considered, there will still exist reasonable scenarios in which an EP search yields no benefit. This can happen if rays in the frustum strike geometry on opposite sides of the root splitting plane. A hypothetical example is shown in Figure 2. In this example, both sides of the root node contain intersected geometry, which forces the root to be chosen as the entry point.

3.2 Re-Thinking Entry Point Search

Paradoxically, in our effort to improve upon EP search, it is most natural to begin by making it worse. Figure 3 shows a recursive formulation of EP search. Besides being recursive, this formulation will visit many more nodes than necessary, and should NOT be used in practice. We are deliberately describing a suboptimal variant of EP search, for reasons which will soon be made clear. (For actual implementation, we follow Reshetov et al. [2005] with a top-down candidate evaluation inspired by Fowler [2009]).

At each node, we first perform a frustum test against the node's bounding box, returning a NULL in the event of a miss. If the node is hit, and it is a leaf, it is returned as an entry point. Otherwise, a search is performed recursively on each subtree. If neither subtree contains an entry point, a NULL will be returned. If only one subtree contains an entry point, then it is returned. If both subtrees contain entry points, then the parent node is returned, as it is their lowest common ancestor. Rather than first filling a bifurcation stack and evaluating the nodes it contains, this approach simply traverses the tree and propagates entry point nodes up the call stack.

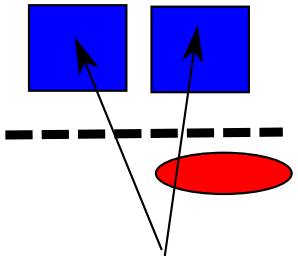

Figure 2: An example in which EP search is ineffective. Arrows denote the boundaries of a bundle of rays. Dashed line denotes location of topmost splitting plane.

```
EPSEARCH( Node N, Frustum F )
    isect = FrustumTest( N, F )
    IF isect == MISS THEN
        return NULL;
    ELSEIF isect == FULLY_INSIDE THEN
        return N;
    ELSEIF IsLeaf(N) THEN
        return N;
    ELSE
        L = EPSEARCH( LeftChild(N), F )
        R = EPSEARCH( RightChild(N), F )
        IF L != NULL and R != NULL THEN
            return N;
        ELSEIF L != NULL THEN
            return L;
        ELSE
            return R;
        END
    END
END
```

Figure 3: A recursive formulation of EP search

As an aside, we note that one can stop traversing immediately if a node's bounding volume is completely contained within the frustum. In this case, it is not necessary to traverse all the way to the leaves, because we know that every leaf node will eventually intersect the frustum. A test for this case is given by [Assarsson and Moller 2000] and it may be vectorized as in [Reshetov et al. 2005]. We have not seen this test used in prior work, but we found that it gives a notable improvement in our EP search implementation.

Consider what happens when only one subtree contains an entry point. This entry point is propagated up the call stack, bypassing its ancestors. This will continue to occur until another entry point is located in a different subtree, at which time their common ancestor will replace them as the entry point. These common ancestors are a superset of the *bifurcation nodes* used in iterative EP search. If we examine the set of nodes visited by a traversal of the tree (as illustrated in Figure 4), what we will generally find is a small number of bifurcating nodes which contain reachable leaves in both

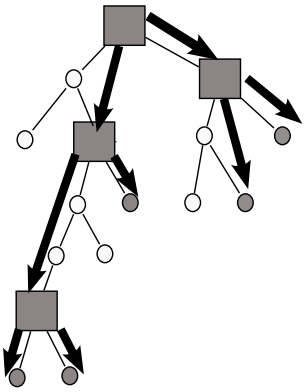

Figure 4: Sample tree topology. Thin lines indicate parent-child links in the BVH. Grey circles indicate visited leaves. Grey squares indicate bifurcation nodes. Hollow circles indicate bypassed nodes. Bold arrows indicate parent-child links resulting from path compression.

subtrees, seperated by groups of *bypassed* nodes, which have reachable leaves in only one subtree. Each bypassed node has at most one bifurcating ancestor and descendant, and there are generally many more bypassed nodes than bifurcating ones.

3.3 Path Compression

The key insight behind our work is that the bypassed nodes are essentially dead weight. For rays in the frustum, only one child of a bypassed node will ever contain any intersections. Furthermore, any ray which hits the bypassed node will also hit its bifurcating parent, and any ray which misses it will also miss its bifurcating descendant. For rays in the frustum, it is therefore possible to traverse *directly* from the node's bifurcating ancestor to its bifurcating descendant without compromising correctness. The bypassed nodes in the middle do not need to be tested. In fact, they do not even need to be accessed.

This observation leads us to our new algorithm. Rather than simply locating the highest possible entry point, we instead construct a temporary BVH on the fly, optimized for a particular frustum. This BVH is a subset of the original which stores only the bifurcating nodes, with direct links to their bifurcating descendants. The root of this BVH is the same node that would have been returned by an EP search, and its leaves correspond either to leaves in the original tree, or to inner nodes whose bounding volumes are fully contained by the frustum. The paths from root to leaves in this tree are shorter versions of the corresponding paths in the original tree. We call this technique *path compression* (overloading the name of a classical optimization for union-find data structures [Cormen et al. 2001]). We refer to the resulting tree as a *path compression tree* (PC tree).

PC tree construction is surprisingly easy to implement. It can be done using a simple modification to the recursive EP search, as shown in Figure 5. At each bifurcating node, rather than discarding the entry points found in the subtrees, we instead return a new PC node, with the two entry points as its children. For bypassed nodes, there is at most one entry point, so we simply propagate the result upwards. An iterative implementation of path compression is also possible, and is considerably more complex. We briefly outline the iterative implementation in Appendix A.

Once we have constructed a PC tree for a given frustum, we can raytrace against it by traversing it in the same manner as an ordinary BVH, using any traversal algorithm. When a leaf is reached in the

```
PATHCOMPRESS( Node N, Frustum F )
    isect = FrustumTest( N, F )
    IF isect == MISS THEN
        return NULL;
    ELSEIF isect == FULLY_INSIDE THEN
        return new PCNode(N);
    ELSEIF IsLeaf(N) THEN
        return new PCNode(N);
    ELSE
        L = PATHCOMPRESS( LeftChild(N), F )
        R = PATHCOMPRESS( RightChild(N), F )
        IF L != NULL and R != NULL THEN
            ret = new PCNode(N);
            ret.left = L;
            ret.right = R;
            return ret;
        ELSEIF L != NULL THEN
            return L;
        ELSE
            return R;
        END
    END
END
```

Figure 5: Pseudo-Code for Path Compression

PC tree, we begin traversing the original data structure, starting with the node indicated by the PC leaf. Note that path compression can, in principle be applied to other data structures than BVHs. In this case, the original data structure is simply overlaid with a BVH.

3.4 Memory Layout and Management

If the original data structure is a BVH, the PC nodes need only store a pointer to the original BVH node, along with a pair of child links. The bounding volume of the original node can be re-used for the PC node. For other data structures such as KD trees, path compression can be applied, but the bounding boxes of the bifurcation nodes must be tracked during PC tree construction and stored explicitly in the PC nodes.

For high performance, it is critical to use an efficient memory management strategy for PC tree nodes. All nodes in our implementation are stored in one contiguous array, and are assigned in sequential order, as needed. A count is maintained of the number of nodes in use. Because each tree is assigned a contiguous range of the array, deallocation of an entire tree can be done by simply decrementing the node count (provided the trees are destroyed in reverse order of construction). If the array becomes full during PC tree construction, a new one is allocated, and used nodes are copied. The array size is doubled with each re-allocation, and eventually stabilizes. In order to enable this reallocation, it is best to store all node references as relative offsets, instead of pointers. This memory management scheme all but eliminates allocation overhead, and it also ensures good cache behavior for the PC nodes.

3.5 Hierarchical Path Compression

Once a PC tree has been built for a particular bounding frustum, it is possible to refine it further to suit a subset of the frustum. To refine an existing PC tree, we can simply apply the same algorithm to the PC tree, creating a new tree which is a subset of the previous one. If a leaf is reached during PC refinement, path compression should be re-run on the original BVH, beginning with the node indicated by the leaf. Note that the test for fully intersected nodes can be omitted when testing inner nodes in the PC tree. Since these nodes

were not fully contained in the original frustum, they cannot be fully contained in a subset of it.

Refinement can be done as often as desired, but it eventually becomes counter-productive. In our experiments with primary rays, we tried numerous permutations of initial tile count and splitting factor. We achieved the best results for 1024x1024 pixel images by building one PC tree for the entire screen, and refining it by splitting into an 8x8 tile grid (256x256 pixels per tile). Each of the 8x8 "super-tiles" are then split directly into 16x16 pixel ray packets (which we found to be optimal in our implementation). Different image resolutions or traversal schemes will naturally benefit from different tiling strategies, but an 8x8 grid of super-tiles seems to be a good baseline.

3.6 Multiple Threads

Extending our technique to multiple threads is straightforward. For multi-threaded rendering, one can simply use a separate node pool for each thread (dividing super-tiles among threads). Alternatively, one could also serialize rendering between tiles, using only one PC tree which is shared among all threads. If memory is plentiful, yet another option is to pre-compute and store the trees for all tiles, and have each thread select the appropriate PC tree for the ray packet being worked on.

4 Results

We have implemented both path compression and EP search, and present experimental data for fly-throughs in a variety of test scenes, shown in Figure 1. Unless otherwise noted, all data presented are averages over a complete fly-through of each scene. Our scenes include the ubiquitous dragon, two of the BART environments (excluding the dynamic geometry), and a complex environment from 'Outbound', a student video game project based on ray tracing.

All experiments were conducted by rendering at 1024x1024 pixels, using one core of a 1.86GHz Core2 Duo laptop. For BVH traversal, we use our own implementation of [Wald et al. 2007], with a geometric frustum intersection test [Reshetov et al. 2005] in lieu of interval arithmetic. We use 16x16 ray packets, which we have found to be the optimal size in our implementation. To simplify our code, we use C++ template functions to implement BVH traversal, using a policy based mechanism to control the behavior at leaf nodes. This makes it possible to traverse both the PC tree and original BVH using nearly the same traversal code.

All of our path compression results use hierarchical tiling as described in section 3.5. For comparison, we used the same hierarchical scheme with EP search. We found that finer tiling with EP search did not significantly improve the results.

4.1 Traversal Reduction

We begin by examining the number of node traversals required to trace primary rays in each scene (Table 1). Here we count the number of times a node is visited when tracing a ray packet, as well as the number of nodes accessed by EP search or path compression. As can be seen, PC tree construction will tend to visit many more nodes than a corresponding EP search would, but this is more than paid for by the reduction in node visits during raytracing. Neither path compression nor EP search is particularly effective for the dragon scene, since the camera is often viewing the entire model from a distance, and most leaves are reachable. Note that for the dragon, EP search is counter-productive.

4.2 Performance

Table 2 gives raw performance measurements. The numbers presented are pure raycasting rates only, and do not include shading or display. We see that path compression can yield speedups as high as 88 percent, and that speedups of over 20 percent are routine. In contrast, we see that simple EP search, while it is occasionally beneficial, is generally of little value.

While our speedups are correllated with the reductions in node traversals, their magnitudes are not very high. This is due to the effect of Amdahl's law. The maximum possible speedup is limited by the fraction of time spent in BVH traversal, which in our case is only about 50%. When path compression is used, this figure drops to about 34%.

Profiling indicates that the vast majority of our time is still spent on ray packet traversal and intersection testing. PC construction and EP search are done quite infrequently, and thus add very little overhead. In more distant viewpoints, where PC provides no benefit, it also tends to terminate very quickly. Our implementation of EP search exhibits the same characteristics.

We conducted our experiments using both recursive and iterative implementations of path compression. We found that while the iterative variant executes about 20% faster than the recursive, the relative cost of path compression is so small that there is no significant change in frame time.

4.3 Memory Consumption

An important question for our work is the memory footprint of the PC tree. In Table 3, we present the final sizes of the PC node pools for each scene, as well as the maximum and average memory requirements (that is, the amount of memory in the pool that was actually in use on any particular frame). The data are presented relative to the size of each scene's BVH, to help place our overhead in context. We find that the amount of memory consumed by the PC tree is typically very small, relative to the size of the main data structure. In addition, memory consumption remains stable across a variety of scene sizes.

There are two principle reasons why our memory consumption is so low. The first is the relative scarcity of bifurcating nodes, and the second is the fact that we terminate PC tree construction when an entire node is contained in the frustum. This early termination allows us to avoid the densely populated lower levels, and is important in keeping our memory consumption under control.

4.4 Partition Traversal

Overbeck et al. [2008] presented an alternative ray traversal technique for BVHs, which is more effective for incoherent rays. This approach divides the rays in a packet into subsets at each visited node, rather than using early-descent. This trades additional work at the top portion of the tree for a reduction in work at the lower levels, because missed rays are prevented from being 'dragged' to lower levels of the tree by rays which hit in the higher levels. We experimented briefly with partition traversal, and found that path compression yields a much higher performance gain in this case. Average performance and speedup are shown in Table 4.

Interestingly, we found that because partition traversal does so much more work per node, it favors a more extreme approach. In addition to the hierarchical tiling used in previous experiments, we found that it was also helpful to construct a PC tree for each individual ray packet prior to tracing it, and to omit the frustum test during packet tracing. We can reasonably expect to see similar results in a

	Traversals	Traversals(EP)	Traversals(PC)	EP Ratio	PC Ratio
Dragon	357	356 (2.9)	277 (55)	1.004	0.94
City	199	177 (2.5)	56 (5.7)	0.99	0.31
Kitchen	273	267 (2.2)	88 (12.9)	0.98	0.36
Outbound	394	374 (5.3)	147 (29.2)	0.96	0.44

Table 1: Average traversals, in thousands. Numbers in parenthesis are nodes visited during EP search or PC tree construction.

	Reference	EP	PC	EP Speedup (Max)	PC Speedup (Max)
Dragon	5.43	5.44	5.50	1.002 (1.06)	1.01 (1.08)
City	8.81	9.01	11.25	1.02 (1.08)	1.28 (1.46)
Kitchen	7.78	7.91	9.57	1.02 (1.23)	1.23 (1.51)
Outbound	6.02	6.04	8.30	1.003 (1.05)	1.38 (1.88)

Table 2: Average packet tracing rates (MRays/s), and speedups. Speedups in parenthesis are the highest observed for a given scene.

	Triangles(Thousands)	BVH Size (MB)	Avg. PC Bytes (%)	Max PC Bytes (%)	Pool Size (%)
Dragon	871	12.97	0.52	0.98	1.44
City	9.2	0.12	4.11	8.72	9.85
Kitchen	103	1.38	1.30	3.23	3.39
Outbound	644	7.97	0.35	1.22	2.35

Table 3: Memory consumption. PC Tree sizes are given as percentages of BVH size

	Partition	Partition(PC)	Speedup
Dragon	3.63	4.66	1.23
City	4.18	8.58	2.05
Kitchen	3.49	7.53	2.16
Outbound	2.44	5.86	2.40

Table 4: Average packet tracing rates (MRays/s), and speedups for partition traversal.

single-ray implementation. In spite of the impressive performance gains, early-descent traversal still defeats partition traversal when path compression is applied. However, there may be some merit to using PC in conjunction with partition traversal for reflection rays.

5 Future Work

5.1 Other Data Structures

As we have already indicated, path compression, suitably modified, is compatible with other raytracing data structures than BVHs. KD trees or bounding interval hierarchices [Wachter and Keller 2006] are both binary trees in which there is an implicit bounding box for every node. Path compression can thus be applied to these data structures as well, and a comparison across the various data structures would be interesting to study.

5.2 Shadow Rays

While we have shown PC to be beneficial for primary rays, this is arguably an unconvincing case. It would be desirable to examine ways that this approach could be applied for secondary rays. We conducted brief experiments with area light sources in the kitchen scene, firing one large ray packet at the area light for each rendered pixel. We had expected to see a performance benefit by performing a PC pass prior to shooting each ray packet, but this turned out to be counterproductive. It is clear from these results that path compression or EP search must be amortized over multiple traversals of the tree in order to be effective, which concurs with a prediction made in [Overbeck et al. 2007].

Despite these negative results, there are certain scenarios in which path compression might be beneficial for shadow rays. For very localized light sources, such as spot lights or point lights with falloff, it might be useful to perform path compression using the bounding volume of the light's region of influence (compressing away any nodes which do not lie inside). In this way, a tree could be computed once per frame and re-used for all shadow rays cast from the light. For shadows which are directly visible from the camera, the view-frustum partitioning methods which are used for shadow maps [Zhang et al. 2006] could also be used to build path compression trees.

5.3 Applications Beyond Raytracing

We find it exciting to consider that our ideas could be generalized to other algorithms which perform searches on tree structures. Any problem which involves repeated, coherent queries against a spatial data structure might benefit from a variant of our path compression technique. For example, one could imagine a photon mapping implementation which batches photon lookups into spatially coherent groups, and uses a loose bounding volume of the query points to compress a KD tree of photon positions. Applications could also be found in areas outside rendering, such as collision detection or database search.

6 Acknowledgements

The dragon model was obtained from the Stanford 3D Scanning Repository. The kitchen and city are from [Lext et al. 2001]. Outbound was created by students from NHTV Breda University of Applied Sciences, and is available at: http://igad.nhtv.nl/ bikker/downloads.htm.

References

ASSARSSON, U., AND MOLLER, T. 2000. Optimized view frustum culling algorithms for bounding boxes. *journal of graphics tools 5*, 1, 9–22.

BENTHIN, C. 2006. *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University.

BOULOS, S., WALD, I., AND SHIRLEY, P. 2006. Geometric and arithmetic culling methods for entire ray packets. Tech. rep., University of Utah.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill.

DMITRIEV, K., HAVRAN, V., AND SEIDEL, H.-P. 2004. Faster ray tracing with simd shaft culling. Tech. rep., Max Plank Institute Fur Informatik.

FOWLER, C., COLLINS, S., AND MANZKE, M. 2009. Accelerated entry point search algorithm for real time ray tracing. *ACM Spring Conference on Computer Graphics*.

FUSSELL, D., FUSSELL, D., SUBRAMANIAN, K. R., AND SUBRAMANIAN, K. R. 1988. Fast ray tracing using k-d trees. Tech. rep.

HAVRAN, V., AND BITTNER, J. 2000. Lcts: Ray shooting using longest common traversal sequences. *Computer Graphics Forum 19*, 59–70.

LAUTERBACH, C., YOON, S.-E., TUFT, D., AND MANOCHA, D. 2006. Rt-deform: Interactive ray tracing of dynamic scenes using bvhs. *Symposium on Interactive Ray Tracing 0*, 39–46.

LEXT, J., ASSARSSON, U., AND MLLER, T. 2001. A benchmark for animated ray tracing. *IEEE Computer Graphics and Applications 21*, 2, 22–31.

OVERBECK, R., RAMAMOORTHI, R., AND MARK, W. 2007. A real time beam tracer with application to exact soft shadows. In *Eurographics Symposium on Rendering*.

OVERBECK, R., RAMAMOORTHI, R., AND MARK, W. R. 2008. Large ray packets for real-time whitted ray tracing. In *IEEE Symposium on Interactive Ray Tracing*.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, ACM, New York, NY, USA, 1176–1185.

RUBIN, S. M., AND WHITTED, T. 1980. A 3-dimensional representation for fast rendering of complex scenes. *SIGGRAPH Comput. Graph. 14*, 3, 110–116.

WACHTER, C., AND KELLER, A. 2006. Instant ray tracing: The bounding interval hierarchy. In *In Rendering Techniques 2006 Proceedings of the 17th Eurographics Symposium on Rendering*, 139–149.

WALD, I., BENTHIN, C., AND WAGNER, M. 2001. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*, 153–164.

WALD, I., BENTHIN, C., AND SLUSALLEK, P. 2003. Openrt - a flexible and scalable rendering engine for interactive 3d graphics. Tech. rep., Saarland University.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1, 6.

YOON, S.-E., CURTIS, S., AND MANOCHA, D. 2007. Ray tracing dynamic scenes using selective restructuring. In *SIGGRAPH '07: ACM SIGGRAPH 2007 sketches*, ACM, New York, NY, USA, 55.

ZHANG, F., SUN, H., XU, L., AND LUN, L. K. 2006. Parallel-split shadow maps for large-scale virtual environments. In *VRCIA '06: Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications*, ACM Press, New York, NY, USA, 311–318.

A Iterative Path Compression

Here we briefly outline an iterative algorithm for building path compression trees. The iterative algorithm uses a bifurcation stack similar to that used in EP search. Each entry in the stack contains a pointer to a BVH node, and space for a pair of PC node references. The algorithm possesses two distinct phases: A *traversal* phase, which walks downwards through a section of the tree, and a *stack contraction* phase, which incrementally builds the PC tree from the bottom up. The algorithm begins in the traversal phase, and switches to stack contraction whenever a change is made to the PC tree.

During the traversal phase, stack entries may be added or removed, and PC nodes may be created and linked to the topmost node in the stack. There are several cases to consider:

- When a leaf node or fully enclosed inner node is reached, a PC node is created. If the stack is empty, then this PC node is the only one in the PC tree, and it is returned. Otherwise, the PC node is linked to the topmost stack entry, and stack contraction occurs.
- Otherwise, if exactly one child node is hit by the frustum, traversal proceeds to that child.
- Otherwise, if both child nodes are hit by the frustum, the node is tentatively considered a bifurcator, and a new entry is pushed onto the stack. Traversal proceeds to the left child.
- Otherwise, if neither child is hit, it means that the subtree being traversed does not contain reachable leaves. This means that the node at the top of the stack is not a "true" bifurcator, so it is popped from the stack, and its linked PC tree (if present) is moved up to the next stack entry. If the stack becomes empty, this linked PC tree is returned immediately, otherwise, stack contraction occurs.

Following any change to the stack, the algorithm enters a stack contraction phase. If the topmost entry in the bifurcation stack possesses two linked PC trees, then these trees are parented to a new PC node containing the BVH node from atop the stack. The stack is then popped, and the resulting PC tree is linked to the new stack top. This continues until one of the following conditions occurs:

- **The stack is empty**. In this case, the completed PC tree is returned.
- **There is only one PC tree**. In this case, the algorithm returns to the traversal phase, beginning at the right child of the node atop the stack.