Fast GPU Ray Tracing of Dynamic Meshes using Geometry Images

Nathan A. Carr
Adobe Corp

Jared Hoberock, Keenan Crane, John C. Hart
University of Illinois, Urbana-Champaign

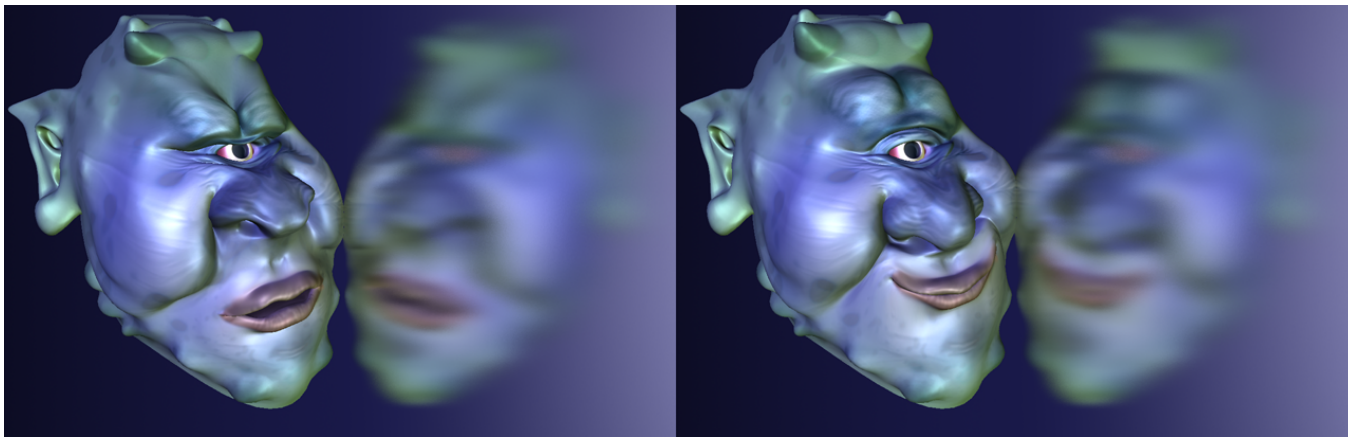

Figure 2: The two poses of the character above (a 128K-triangle blendshape model represented with a 256^2 geometry image) were rendered at a resolution of 1272×815 using naive ray sampling in about a half a minute each, including the construction of the threaded bounding box hierarchy.

ABSTRACT

Using the GPU to accelerate ray tracing may seem like a natural choice due to the highly parallel nature of the problem. However, determining the most versatile GPU data structure for scene storage and traversal is a challenge. In this paper, we introduce a new method for quick intersection of triangular meshes on the GPU. The method uses a threaded bounding volume hierarchy built from a geometry image, which can be efficiently traversed and constructed entirely on the GPU. This acceleration scheme is highly competitive with other GPU ray tracing methods, while allowing for both dynamic geometry and an efficient level of detail scheme at no extra cost.

Keywords: ray tracing, GPU algorithms, geometry images, mesh parameterization

1 INTRODUCTION

Ray tracing is often referred to as an "embarrassingly" parallel application, but the key to efficient ray tracing is to avoid intersecting every ray with every geometric primitive. The geometric partitioning needed for such optimization can be difficult to implement efficiently on data-parallel architectures such as the GPU. For instance, a GPU ray tracer accelerated by a uniform grid suffers from the overhead of state-based programming [12] and a GPU ray tracer that offloads partitioning to the CPU suffers from a CPU-GPU communication bottleneck [2].

The GPU is becoming a practical high-performance platform for ray tracing due to increased generality and rapid parallel performance growth relative to CPUs. CPU ray tracers have also gained parallelism due to SSE, distributed processing, and soon multi-core CPUs [19, 13]. However, visual simulation applications increasingly rely on the CPU for physics, AI, animation, and other tasks which diminish ray tracing performance. Special purpose hardware can also support efficient parallel ray tracing [21], but does not enjoy the same economy of scale as GPUs.

The main contribution of this paper is a new GPU ray tracer accelerated by a threaded bounding box hierarchy stored as a geometry image min-max MIP map. An input mesh is uniformly tessellated and stored as a *geometry image* [5], where each texel stores the coordinates of a mesh vertex. We then create a pair of MIP maps [20], one *min* and one *max*, whose texels correspond to the minimum (maximum) x, y and z coordinates of their corresponding vertices in the geometry image, as shown in Figure 3.

Traditionally, traversing a bounding box hierarchy makes use of conditional execution and a stack (though alternate representations are sometimes possible [17]), both of which are highly inefficient on current GPUs. We overcome these deficiencies by statically "threading" the hierarchy, adding two pointers to each node: a *hit* pointer leading to the first child and a *miss* pointer leading to the next sibling (or uncle). This threading scheme allows nodes of the hierarchy to be efficiently streamed through the GPU pipeline.

In addition, we present a method for quickly building the threaded hierarchy entirely on the GPU. The cost of construction is negligible with respect to total frame rendering time, and is done using geometry resident on the card. By updating the hierarchy every frame, the method can ray trace dynamic geometry such as skinned characters or blend shapes (previous GPU ray tracing methods were limited to static geometry only). A dynamic hierarchy also allows for dynamic level of detail, which can significantly reduce the cost of rendering distant or unimportant objects.

Two components of our method were presented in similar contexts. Geometry image MIP maps were first used for level of detail management [7] and collision detection [1]. The threading of a bounding box hierarchy was applied to GPU ray tracing by Thrane and Simonsen [18]. However, the combination of hierarchy thread-

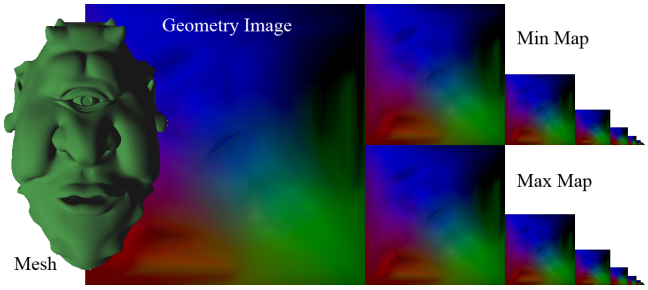

Figure 3: The GPU ray tracer described by this paper relies on the retesselation of a scene into a regular mesh, stored as a geometry image of its vertices along with a bounding box hierarchy constructed as a min/max pair of MIP maps.

ing with a geometry image bounding box layout is novel, and leads to a memory layout that is more economical and cache-coherent than previous GPU-side ray tracers. We also offer a new algorithm for maintaining this bounding box hierarchy on dynamic geometry. This enables effects like dynamic interreflection between a moving object and its surroundings, lending the method towards real-time applications.

2 Previous Work

The long history of high-performance ray tracing on the CPU is nicely summarized by Wald [19]. While the GPU outperforms the CPU on streaming kernels such as ray intersection, the CPU is much more efficient at maintaining and traversing the complex data structures needed to trace rays efficiently. The recent RPU paper [21] gives a similar overview of ray tracing on special-purpose hardware. The RPU resembles a GPU with SIMD processing of four-vector operations, but also includes a stack and separate traversal and shading processors for better load balancing. It can hence easily outperform a GPU ray tracer, but is currently expensive to build and commercially unavailable.

The Ray Engine [2] implemented a streaming ray-triangle kernel on the GPU, fed by buckets of coherent rays and proximate geometry organized by a CPU process. However, this division of labor required frequent communication of results from the GPU to the CPU over a narrow bus, negating much of the performance gained from the GPU kernel.

The performance degradation due to GPU-CPU communication can be avoided by implementing other components of ray tracing on the GPU. Purcell *et al.* [12] decomposed ray tracing into four separate GPU kernels: traversal, intersection, shading, and spawning. The performance of this faster state-based approach was nevertheless limited by SIMD scheduling because the GPU fragment processors had to run identical kernels. As a consequence, some processors ready for intersection had to wait for others to complete traversal. This type of delay limited the performance of state-based ray tracing to 10% of peak GPU efficiency. The method also utilized a uniform 3-D grid partition of scene geometry which was cache-coherent and accessed in constant-time. Unfortunately, this scheme resulted in a large number of ray steps through empty space, further delaying the rest of the fragment processors from reaching their next state.

State-based GPU ray tracing runs up to 50% faster when the cells of the uniform grid hold a precomputed distance transform (called a "proximity cloud") that allows the traversal kernel to skip large empty regions of space [8]. Even with proximity clouds, state-based GPU ray tracing is unable to outperform CPU ray tracers such as *Mental Ray*, largely due to the latter's support of a hierarchical kd-tree spatial partition, able to accommodate the varying level of geometric detail found in typical scenes [3].

Many hierarchy traversal algorithms rely on a stack, a feature not well-supported by the GPU (though a small four-element call stack is available). Foley and Sugerman [4] implemented two variations on a stackless traversal of a kd-tree: kd-restart, which iterates down the kd-tree, and kd-backtrack, which finds the parent of the next traversal node by following parent pointers up the tree, comparing current traversal progress with per-node bounding boxes. While this decomposition showed that GPU hierarchy traversal is feasible, it achieved only 10% of the performance of comparable CPU implementations, citing the need for further work in load balancing and data management.

Our GPU ray tracer uses a threaded bounding box hierarchy which does not rely on conditional execution (another feature poorly supported by the GPU) to determine the next node in a traversal. This method was developed concurrently with the one presented in Thrane and Simonsen's Master's thesis [18]. The method "threads" a bounding box hierarchy with a pair of pointers per node, indicating the next node to consider given that the ray either intersects or misses the node's bounding box. These threads allow the GPU to efficiently stream through the hierarchy without maintaining a stack. Because our hierarchy representation is stored in a structured 2D texture as opposed to a less structured linear stream, we expect better performance from the GPU's texture cache which operates on 2D blocks. Although a dynamic update of the linear structure may be possible, the structured 2D arrangement yields a straightforward implementation, again with potentially fewer cache misses.

3 Algorithm Details

While the hierarchy is dynamic, it does rely on two precomputed data structures: a mesh parameterization and a *link hierarchy*. Note that this limits us to meshes of constant topology. The parameterization is used every frame to build a geometry image and bounding box hierarchy for a deforming mesh. The link hierarchy (which is independent of specific geometry) controls the traversal order of a ray's walk through the bounding box hierarchy. In this section, we detail each step of this process.

3.1 Offline Precomputation

We parameterize our input geometry offline using standard techniques. Each surface is cut into a number of charts and mapped to rectangular regions of a square texture. Our implementation employs the area-preserving L^2 geometric stretch metric proposed by Sander *et al.* [16] and iteratively relaxes an initial linear solution into a local minimum of distortion. Chart packing was done by hand, but can also be done using automated packing tools.

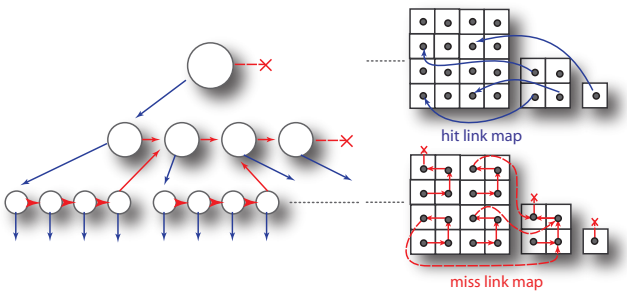

Figure 4: Link map hierarchy.

Link Hierarchy Creation To stream bounding volume traversal, we require a static set of traversal links for our rays which we call a *link hierarchy* (Figure 4). At each traversal step we have two primary states to consider. (1) A ray misses the bounding volume of a given tree node n. In such a case, the ray follows the *miss* link to find the next bounding volume in the tree to test against. Miss links point to either a sibling or uncle of n. (2) A ray hits the bounding volume of node n. In such a case, the ray follows the node's *hit* link to a child node of n. A zero-valued link indicates an overall miss (termination), and a negative-valued link signals a leaf node requiring triangle intersection (see Figure 5), pointing to a texel corner in the geometry image. By following precomputed hit or miss links, a ray streams through the tree without the need for state or complex branching.

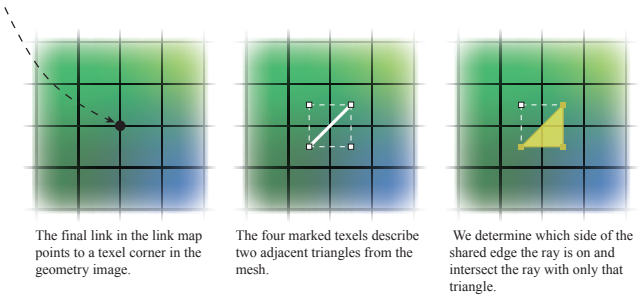

Figure 5: Intersecting triangles from a geometry image.

3.2 Online Ray Tracing

Ray tracing is a three step process that involves rasterizing an object's current geometry image, building its bounding box hierarchy, and traversing the hierarchy to determine ray-mesh intersections.

Geometry Image Rasterization If a mesh has undergone deformation, we must update its geometry image in order to maintain a correct bounding volume hierarchy. To do this, we rasterize the parameter-space of the mesh into a texture, using world-space positions as vertex colors. This gives us a texture in which adjacent texels represent nearby points on the mesh surface. These points will be used during ray tracing to construct triangles which also lie on the surface. Additionally, we render the corresponding normal map at this time.

Bounding Volume Hierarchy Construction After geometry image rasterization, we construct a hierarchy of axis-aligned bounding boxes. The process is conceptually similar to MIP map construction. We start by creating two textures which represent the finest level of a *min* image pyramid and a *max* image pyramid, and recursively build the coarser levels. The rgb channels of this pyramid store the minimum and maximum extents of an axis aligned bounding volume around the four corresponding texels in the level below. AABBs in the bottom level of the pyramid bound the four corresponding texels from the geometry image. A bounding box is placed around every quad in the geometry image, so the resolution of the lowest level of the pyramid is equal to the resolution of the geometry image minus one.

Rather than using MIP maps for storage, we pack the entire hierarchy into a single texture, since MIP map texture reads currently fail to produce correct results under divergent conditions[1]. The layout of the bounding box hierarchy mirrors the layout of the link map hierarchy (Sec. 3.1), so a single uv coordinate can be used to reference the hit link, miss link, and bounding box information for a given node in the tree.

Ray Traversal During traversal, we maintain a link location specifying where we are in the tree. Initially, this link location is the uv location of the root of the tree. We enter the traversal loop and intersect the ray against the bounding box at the current link location. Depending on the outcome of the intersection, the link location is updated by following either the miss link or hit link. Upon completion of the shader, we return the uv value of the closest hit location in the geometry image. We return a negative uv value when the ray misses the geometry entirely.

Traversal Algorithm

```
link = location of root node
t = infinity; // par. distance along ray
while(1) {
  BV = boundingVolume[link]
  if rayHitsBoundingVolume(ray,BV,t)
    link = hitLink[link]
  else
    link = missLink[link]
  if( link == 0 )
    break; // we're done so exit
  if( link < 0 )
    // we're at a leaf
    // intersect ray with geometry image
    ...
    link = missLink[link]
}
```

Figure 6: Pseudo-code for GPU bounding volume hierarchy traversal.

3.3 GPU Ray-Intersection

The traversal kernel handles ray intersection for both axis-aligned bounding volumes and triangles. The slabs method is used to intersect against axis-aligned bounding boxes [9]. This method computes the distance t' between the ray's entry point into the bounding box and the ray origin. By maintaining the distance t to the currently known closest hit, we are able to avoid traversal into subtrees that are beyond this intersection point (i.e. $t < t'$).

Our ray-triangle intersection routine is based on that of Möller and Trumbore[11]. This routine is enhanced using the knowledge that triangles will always be intersected in pairs. Assuming backface culling, the key observation is that a ray may intersect at most one of two triangles sharing an edge (except in the rare degenerate case). We first determine on which side of the shared edge the ray passes, and then compute the intersection with the appropriate triangle. As a result, intersection is accelerated by a factor of nearly two. Assuming no texture fetch latency, an NVIDIA G70 GPU can execute this code in twenty cycles achieving in excess of 1 billion ray-triangle intersection tests per second[2].

4 Results

Our implementation used OpenGL version 1.5 and Cg version 1.4. Dynamic branching in the traversal shader required the use of Shader Model 3.0. Data was stored in half-precision (16-bit per

[1]The GL_NV_vertex_program3 TXL instruction returns texels from the same MIP level for 2×2 fragment blocks instead of allowing a different level for each fragment (this is a known issue).

[2]Performance as measured by NVIDIA NVshaderPerf

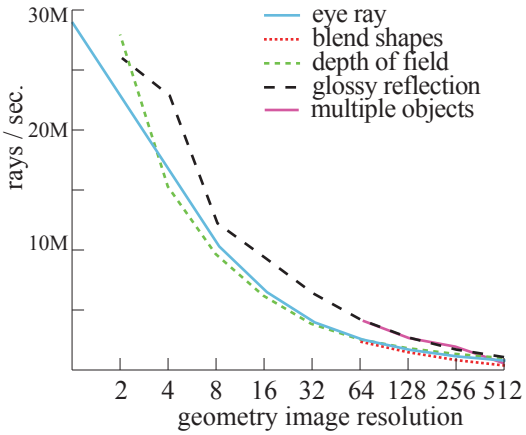

Figure 7: Performance (in millions of rays per second) versus geometry image resolution for four ray tracing test applications. A 512^2 geometry image corresponds to 524,288 triangles.

LOD	Tris.	Eye	Blend	DOF	Glossy	Mult.
2	8			28.2	26.5	
4	32	29.4		15.1	23.3	
8	128	10.1		9.31	12.5	
16	512	6.25		5.71	9.56	
32	2K	3.70		3.30	6.56	
64	8K	2.25	2.23	1.97	4.32	4.35
128	32K	1.37	1.35	1.21	2.80	2.85
256	128K	0.818	0.802	0.706	1.78	1.83
512	512K	0.462	0.453	0.348	1.14	1.08

Table 1: Performance rates (in millions of rays per second) for eye rays, the animated blendshape, depth of field, glossy reflection, and eye rays for multiple objects measured per level of detail (in geometry image resolution).

component) textures, with a separate texture for each of geometry image, min map, max map, link map, ray origins, and ray directions. Results were rendered via an FBO into a half-precision deferred shading buffer. Performance was measured on a single NVIDIA G70 (GeForce 7800 GTX) GPU running at 430 MHz core clock and 1.2 GHz memory clock with release 80 drivers on a Windows XP system with an AMD Athlon 3500+ 2.2 GHz CPU. We did not do extensive tuning of shader parameters such as loop unrolling.

Eye rays Our first test shoots rays from the eye through a mesh, computing only local illumination at the intersection. Eye rays are not a particularly interesting application of ray tracing, but are needed to compare performance directly against other methods which use similar tests [19, 4]. (Rasterization generally provides a more efficient way to compute the first hit.) This test also gives us an idea of peak performance, since eye rays are highly spatially coherent (and therefore highly cache coherent). Figure 7 plots ray intersection performance versus geometry image resolution.

Dynamic Geometry Our next test shoots eye rays at a mesh animated with blend shapes. Figure 1 shows different expressions of our model. Figure 7 confirms that performance is not substantially worse for dynamic geometry (render times differed by 2 percent on average).

Perfect Specular Reflection Figure 9, *top* shows three Utah Teapots exhibiting specular reflection. Ray traced reflections produce sharp detail and self-reflection, which are difficult to achieve

Figure 8: A 598×634 image of an animated character rendered with depth of field using a 256^2 ($131,072$ effective triangles) geometry image. $97,057,792$ rays were evaluated in under 2 minutes.

with environment maps. Although secondary rays are potentially less coherent than eye rays, this did not seem to have an effect on performance for this particular scene. Note that the first hit was computed via rasterization for all applications involving secondary rays.

Monte Carlo Ray Tracing Effects such as glossy reflections, depth of field, and motion blur are robustly handled by Monte Carlo ray tracing, which takes a sparse set of samples over a large domain. Depth of field in figure 8 and glossy reflections in figures [1] 9,*left* are generated by shooting uncorrelated random rays into the scene, examining performance under poorer cache coherence and worse divergent branching than for eye or perfect specular rays. (However, many of these rays may miss the geometry, traversing only the coarsest levels of the hierarchy.) Figure 7 shows performance similar to eye rays for our test scenes.

Multiple Objects The scene in Figure 9 tests the "teapot in a stadium" scenario by placing three Utah Teapots at the center of relatively large, sparse geometry. This type of scene demonstrates the ability of our algorithm to handle scenes with geometry at widely varying scales. We have observed that packing multiple objects into a single parameterization incurs no appreciable degradation in performance, as indicated by the average eye-ray performance for this scene shown in Figure 7.

Visual Error Finally, we analyze the image space error resulting from the geometry image representation. Error is calculated by computing the average per pixel difference of a ray traced image versus a rasterized reference. Figure 11 plots average pixel error versus resolution and illustrates decreasing visual error as geometry image resolution increases. Incongruous silhouettes contribute significantly to error at lower resolutions.

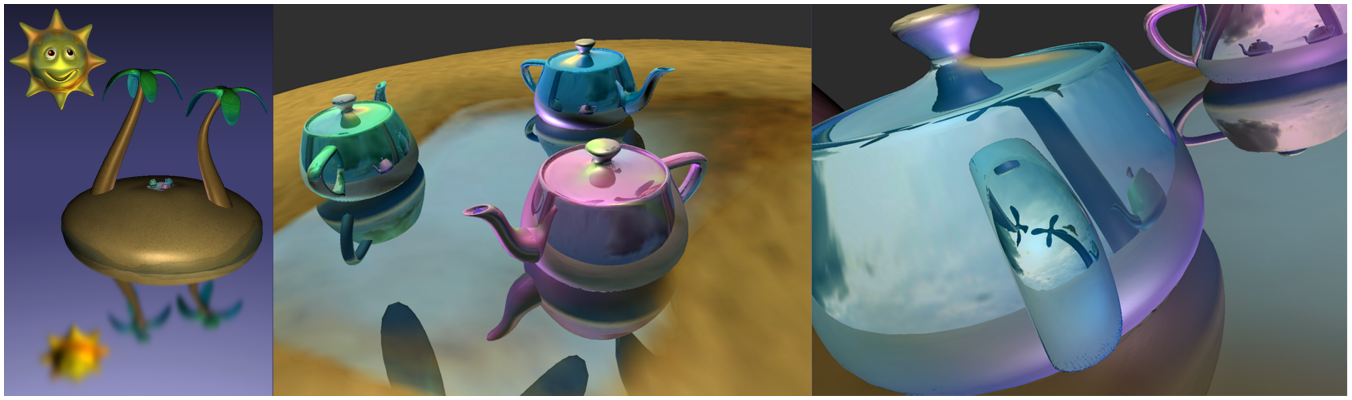

Figure 9: *Left.* Ray traced scene with multiple charts and heterogenous detail exhibiting both glossy and specular reflection. *Center.* Closeup of teapots exhibiting self- and inter-object reflection. *Right.* Closeup of teapot handle showing crisp, curved reflection of distant trees.

5 Discussion

In this section we analyze performance of our method for common ray tracing applications. Direct, exact comparison of various GPU implementations is difficult due to a large number of dependent variables among hardware generations and vendors. We aim to identify and discuss overall benefits and tradeoffs versus other ray tracing alternatives.

5.1 Intersection Performance

A common test of performance shoots eye rays at the 70K triangle Stanford Bunny to generate a 512^2 image. For this test, BVH takes 257 ms (GeForce 6800 Ultra) [18], uniform grid 357 ms, kd-backtrack 690 ms, and kd-tree restart 701 ms (X800 XT PE) [4]. We achieve between 180 ms for a 128^2 geometry image and 675 ms at 256^2 (GeForce 6800 Ultra). As indicated in Figure 10, our approach does as well as a uniform grid on a uniform mesh (and should outperform it otherwise), but is not quite as optimized as BVH, which also threads a bounding volume hierarchy.

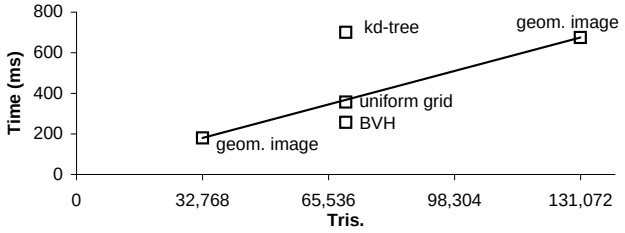

Figure 10: Comparison of GPU ray tracers on the Stanford bunny. Note that hardware used has slightly different performance characteristics (GeForce 6800 Ultra vs. X800 XT PE).

We were unable to test the Cornell box or the BART Robots scene [10] because our implementation does not yet handle sharp edges, which could be implemented through an extension [15]. We instead constructed a test scene with a similar non-uniform distribution of geometry, shown in Figure 9.

Table 1, *Mult*. gives eye-ray performance for the Teapot Island test. The BART Robots scene contains 71708 triangles. The original Teapot Island scene contains 36032 triangles, but requires a geometry image resolution between 256^2 and 512^2 geometry image or 128K to 512K effective triangles to render accurately due to poor parameterization (teapot geometry is not instanced and consequently the island is not given much texture area). Performance was between 1.1 and 1.8 million rays/sec on average, though far fewer rays descend the hierarchy than in the BART Robots. For the BART Robots, Foley & Sugerman report performance equivalent to about 0.5 million rays/sec for both kd-restart and kd-backtrack. Thrane & Simonsen report performance equivalent to about 0.7 million rays/sec with a BVH implementation for the same scene. Performance of our algorithm is generally competitive; more importantly, it compares well to its own performance for uniform geometry.

5.2 Acceleration Structures

The bounding volume hierarchies constructed by Thrane & Simonsen and Foley & Sugerman are more efficient than ours due to extensive optimization. However, the optimzation process is a lengthy, heuristic-based approach [6], and there is currently no known mapping of such a method to the GPU. Dynamic geometry may still be possible within Thrane & Simonsen's framework by applying a reduction operation on vertex positions to determine updated bounding box extents, though an analogous approach is not apparent for a kd-tree. Note that with any of the approaches updating node extents may reduce the optimality of the hierarchy.

A regular grid requires no optimization, and could be quickly constructed every frame (by rasterizing mesh triangles into every grid slice, for example). However, the inability to deal with geometry at a wide range of scales makes this structure unappealing.

Thrane & Simonsen use a linear array of data for their BVH implementation versus our 2D memory layout. This results in both smaller link addresses as well as a simpler way to compute the *hit* address, but does not take advantage of the GPU's ability to cache *blocks* of texture as in a 2D layout - the overall tradeoff is not clear. However, the true performance deficit in both implementations is the inability to sort traversal order: nodes are always traversed in the same order, often resulting in a huge number of unnecessary intersections.

5.3 Storage

Per-node storage, storage for intermediate state, and per-triangle storage is given for several hierarchy schemes in Table 2. For internal nodes, our implementation uses three floats for each of AABB *min* and *max* and two floats for each of *hit* and *miss* link pointers, all at 16-bit precision. One link pointer might be avoided by instead computing the address from the current link node and level in the hierarchy. Note that neither BVH implementation requires any intermediate state since traversal is done in a single pass. In addition to saving memory, eschewing state avoids the need to re-load state on every pass. Neither Karlsson & Ljungstedt nor Thrane &

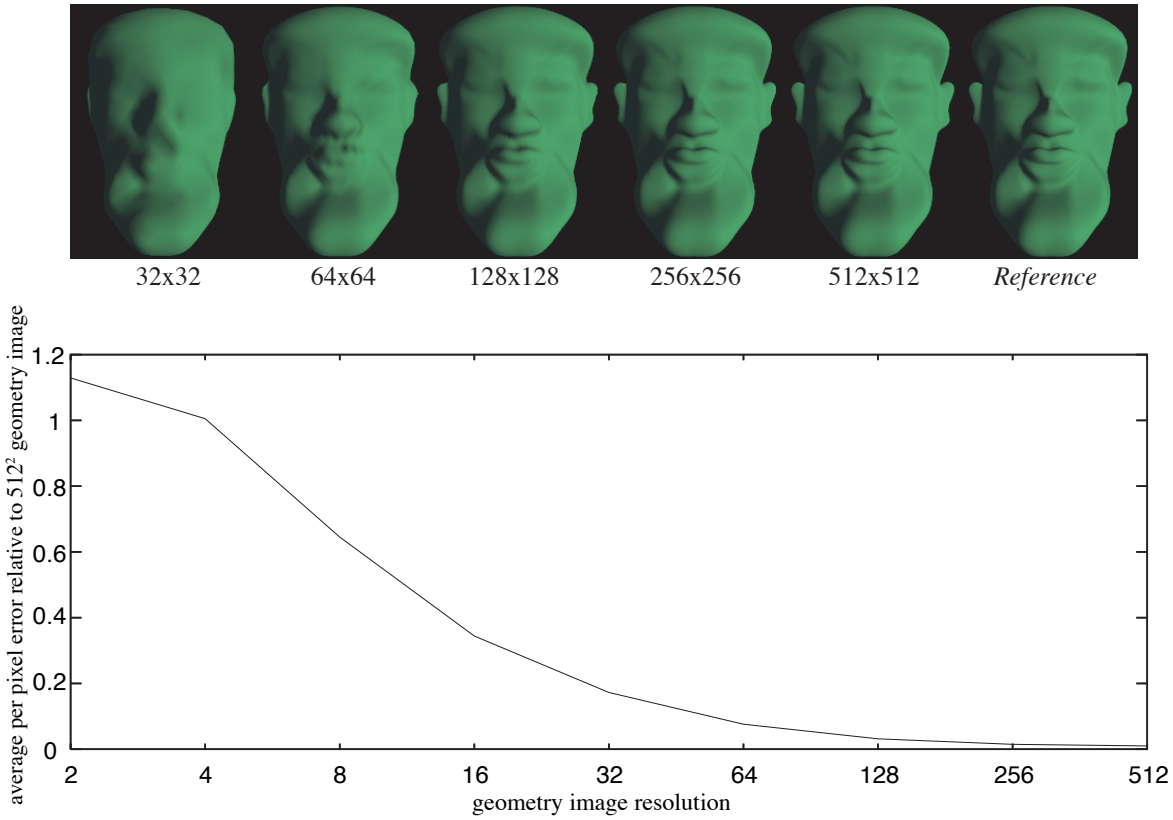

Figure 11: Image space error decreases with increasing geometry image resolution.

Method	**Node**	**State**	Triangle
Karlsson proximity cloud	64*	64*	320*
Foley backtrack	64	128	320
Foley extended split	320	128	320
Thrane uniform Grid	32*	256*	320*
Thrane kd restart	128*	128*	320*
Thrane kd backtrack	384*	128*	320*
Thrane BVH	128*	(*none*)	320*
BVH w/ GI (our method)	160	(*none*)	24

Table 2: Storage requirements for each hierarchy element, for intermediate state, and for each triangle in bits for various methods (* assuming 32-bit floats).

Simonsen use vertex-sharing for triangles (i.e., three independent vertices are stored for every triangle), whereas a geometry image naturally shares each vertex among six triangles (i.e., less than one vertex is stored per triangle). However, our algorithm requires a deeper tree and may require more triangles to represent the same mesh. A lower bandwidth fixed-point representation may improve efficiency, though past attempts have met with limited success [2].

5.4 Implementation

All of the algorithms mentioned are relatively straightforward to implement, but each requires some complexity in order to build a good hierarchy: Thrane & Simonsen's BVH and Foley & Sugerman's kd-tree require extensive balancing, while our method requires careful parameterization. Regular grids are simple to implement in all respects, but again are not a general-purpose solution.

6 Conclusion and Future Work

We have described a general-purpose GPU ray tracer accelerated by a bounding box hierarchy stored in a geometry image. Performance is comparable to other high-performance ray tracers, with the additional benefit of dynamic geometry. The hierarchy can be built on the GPU and is compatible with the GPU's streaming architecture.

Optimized Link Hierarchies Our current implementation uses a single precomputed link thread to determine ray traversal. As a result, traversal order is arbitrarily fixed. We might instead precompute several link hierarchies sorted on the six cardinal directions and choose the best one at traversal time by hashing on a ray's direction. This scheme may improve performance as a ray would be more likely to terminate early. We could also arrange nodes so that they would be accessed along a Hilbert curve, potentially increasing cache coherence.

Non-Power of Two Geometry Images Our implementation assumes that geometry images are square powers of two. However, there is a large jump in mesh quality between successive powers, and the optimal resolution may lie somewhere in between. Geometry images with irregular dimensions could be supported by simply modifying the hit/miss pointer patterns in the link hierarchy.

Signal Specialization The efficiency of the bounding box hierarchy and the quality of the ray traced model depend heavily on the parameterization used to construct the geometry image. Better parameterizations could be generated by using surface curvature to automatically guide selection of cuts on the mesh and distribution of area in the geometry image.

Adaptive Level of Detail Fast hierarchy construction makes adaptive level of detail possible by repartitioning objects' allocation in parameter space. Adaptive LOD could also be implemented by intersecting rays with coarser levels of a MIP map pyramid of the geometry image according to their *t* values.

Reparameterization. Our ray tracer supports the rendering of dynamic objects by rebuilding the bounding box hierarchy using an unchanging geometry image parameterization on a mesh with changing 3-D vertex positions. This approach precludes any change to mesh topology, and is insensitive to changes in the parameterization distortion caused by the addition or removal of geometric features. We expect that such distortion could be quickly reduced by applying a few relaxation steps to the parameterization before bounding box hierarchy reconstruction.

Instancing Instancing can be supported with an additional BVH where leaves bound meshes. Leaf nodes in the link map hierarchy would point to top-level nodes of a mesh in the current BVH.

Out-of-Core Streaming Production scenes are too large to fit into the core memory of the GPU. However, a fast hierarchy update makes it possible to quickly stream partial scenes into GPU memory. A REYES-like renderer can use this ability to ray trace a manageable working set. This scheme is a tradeoff between ideal ray tracing efficiency and scalability in a rasterization-based pipeline.

Acknowledgments

This research was supported in part by the NSF under the ITR grant #ACI-0113968 and by NVIDIA Corp. Thanks to Pete Shirley for advice and to David Gu for the bunny parameterization.

REFERENCES

[1] Bedřich Beneš and Nestor Goméz Villanueva. GI-COLLIDE: collision detection with geometry images. In *Proc. Spring Conference on Computer Graphics*, pages 95–102, 2005.

[2] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Proc. Graphics Hardware 2002*, pages 37–46, Sep. 2002.

[3] Martin Christen. Ray tracing on GPU. Master's thesis, Univ. of Applied Sciences Basel (FHBB), 2005.

[4] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a GPU raytracer. In *Proc. Graphics Hardware*, pages 15–22, 2005.

[5] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. In *Proc. SIGGRAPH*, pages 355–361, 2002.

[6] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Dept. of CSE, Fac. of EE, Czech Technical University in Prague, Nov. 2000.

[7] Junfeg Ji, Enhua Wu, Sheng Li, and Xuehui Liu. Dynamic LOD on GPU. *Proc. CGI*, 2005.

[8] Filip Karlsson and Carl Johan Ljungstedt. Ray tracing fully implemented on programmable graphics hardware. Master's thesis, Chalmers Univ. of Technology, 2004.

[9] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. In *Proc. SIGGRAPH*, pages 269–278, 1986.

[10] Jonas Lext, Ulf Assarsson, and Tomas Moller. BART: A benchmark for animated ray tracing. Technical Report 00-14, Dept. of CE, Chalmers U. of Tech., May 2000.

[11] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[12] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *Proc. SIGGRAPH*, 2002.

[13] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3):1176–1185, 2005.

[14] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. In *Proc. Sym. Geom. Proc.*, pages 146–155, 2003.

[15] Pedro V. Sander, Steven J. Gortler, John Snyder, and Hugues Hoppe. Signal-specialized parametrization. *Proc. Eurographics Rendering Workshop*, pages 87–100, 2002.

[16] Pedro V. Sander, John Snyder, Steven J. Gortler, and Hugues Hoppe. Texture mapping progressive meshes. In *Proc. ACM SIGGRAPH 2001*, pages 409–416, 2001.

[17] Brian Smits. Efficiency issues for ray tracing. *J. Graph. Tools*, 3(2):1–14, 1998.

[18] Niels Thrane and Lars Ole Simonsen. A comparison of acceleration structures for GPU assisted ray tracing. Master's thesis, University of Aarhus, Denmark, 2005.

[19] Ingo Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004.

[20] Lance Williams. Pyramidal parametrics. *Computer Graphics (Proc. SIGGRAPH 83)*, 17(3):1–11, July 1983.

[21] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: a programmable ray processing unit for realtime ray tracing. *(Proc. SIGGRAPH) ACM TOG*, 24(3):434–444, 2005.

7 APPENDIX A

Below are a few technical details useful in guiding an implementation of our algorithm.

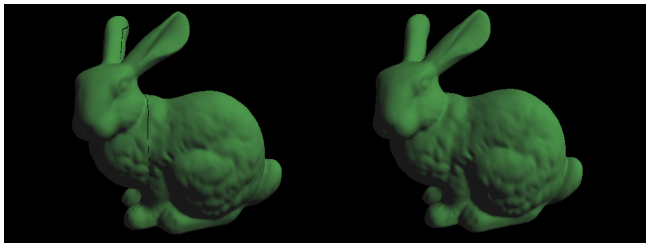

Figure 12: *Left.* Naively generating a geometry image may create cracks in the ray traced image. *Right.* Adding an additional border of texels prevents cracks.

Preventing Cracks Parameterizing a closed surface requires one or more cuts, resulting in potential cracks in the reconstructed surface. We draw an additional border of texels around chart boundaries to ensure that reconstructed boundary vertices are aligned along the cut [14]. Figure 12 demonstrates cracking on the Stanford bunny.

Multiple Objects For a scene with multiple meshes, we parameterize each object separately, then pack all objects into a single atlas. To avoid spurious triangles at chart boundaries, charts are separated by a one-pixel "gutter" of *NaN*s. *NaN*s have the special property of failing all numeric comparisons, and do not interfere with the construction of the bounding box hierarchy or ray-triangle intersection.

Precision In our implementation, all data is stored in a 16-bit floating-point format, which can sometimes cause tiny triangles to be rendered incorrectly. Re-scaling the scene is a simple way to remedy these artifacts.

Loop Counter Limits Current graphics hardware limits the number of times a loop may iterate. This limitation can result in premature termination of the *while* loop of figure 6, resulting in holes in the geometry (mainly for resolutions of 512^2 or larger). We dealt with this limit by doing more traversal steps per loop iteration, and could further break up traversal into multiple passes. Nesting the *while* in another *while* loop prevented holes in some cases by increasing the effective maximum iterations, but produced additional artifacts as well.