The Hierarchical Ray Engine [1]

László Szécsi

Dept. of Control Engineering and Information Technology
Budapest University of Technology and Economics
Magyar Tudósok krt. 2., H-1117, Hungary
szecsi@iit.bme.hu

ABSTRACT

Due to the success of texture based approaches, ray casting has lately been confined to performing preprocessing in realtime applications. Though GPU based ray casting implementations outperform the CPU now, they either do not scale well for higher primitive counts, or require the costly construction of spatial hierarchies. We present an improved algorithm based on the Ray Engine approach, which builds a hierarchy of rays instead of objects, completely on the graphics card. Exploiting the coherence between rays when displaying refractive objects or computing caustics, realtime frame rates are achieved without preprocessing. Thus, the method fills a gap in the realtime rendering repertoire.

Keywords: Ray tracing, GPU programming.

1 INTRODUCTION

With high performance hardware designed to support scan conversion image synthesis, most research aims to eliminate time consuming ray casting from illumination algorithms, or to move it to a preprocessing step computing texture maps. However, there are some light transport effects that exhibit inherently recursive behavior, most prominently visible refractive objects, or caustics via multiple reflections or refractions. Accurate maps or transport factor matrices cannot be constructed with a feasible storage requirement. On the other hand, these problems are effectively handled by recursive raytracing or photon tracing, both based on ray casting. Moreover, if we consider eye rays or light rays from small light sources, hitting reasonably smooth objects, the rays to be traced will be coherent, even after multiple reflections of refractions.

In order to make ray casting feasible in realtime applications, it is imperative to make use of the immense computing power of the GPU. One delivering research direction has spawned from the approach of Purcell et al.[6], the impact of which we will briefly evaluate in Section 6. If we are looking for a solution which does not rely on a pre-built acceleration structure, the most important milestone we find is the Ray Engine[2]. Based on the recognition that ray casting is a crossbar on rays and primitives, while scan conversion is a crossbar on pixels and primitives, they have devised a method for computing all possible ray-primitive intersections on the GPU. On contemporary hardware they could achieve processing power similar to the CPU's.

2 PREVIOUS WORK

2.1 The ray engine

As the ray engine serves as the basis of our improved approach, let us reiterate its working mechanism in current GPU terminology. Every pixel of the render target is associated with a ray. The origin and direction of rays to be traced are stored in textures that have the same dimensions as the render target. In every pass, a single ray casting primitive is taken, and it is rendered as a full-screen quad, with the primitive data attached to the quad vertices. Thus, pixel shaders for every pixel will receive the primitive data, and can

[1] Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. FULL Papers conference proceedings ISBN 80-86943-03-8. WSCG'2006, January 31-February 4, 2006 Plzen, Czech Republic. Copyright UNION Agency - Science Press

also access the ray data via texture reads. The ray-primitive intersection calculation can be performed in the shader. Then, using the distance of the intersection as a depth value, a depth test is performed to verify that no closer intersection has been found yet. If the result passes the test, it is written to the render target and the depth buffer is updated. This way every pixel will hold the information about the nearest intersection between the scene primitives and the ray associated with the pixel. The pitfall of the ray engine is that it implements the naive ray casting algorithm of testing every ray against every primitive. On the other hand, ray casting research has been directed on building effective acceleration hierarchies to minimize the number of actual intersection tests performed [1]. Unfortunately, these results cannot be easily ported to the graphics hardware.

2.2 Exploiting coherence on the CPU

It is relevant to mention that besides hierarchies, methods making use of image-space coherence were also exhaustively researched[8]. Most importantly, we will make use of the motif of decomposing the image space into tiles containing assumedly coherent rays in our algorithm.

2.3 Acceleration hierarchies for the GPU

Foley and Sugerman[3] implemented the architecture foreseen by Purcell et al.[6]. The kd-tree acceleration hierarchy, formerly confined to the CPU, is traversed on the GPU using algorithms not optimal in the worst-case algorithmic sense, but eliminating the need of a stack. This offers a competitive alternative to CPU ray tracing, but it is not directly targeted on real-time applications, and it is ill-suited for highly dynamic scenes because of the construction cost of the kd-tree.

2.4 Approximate ray tracing

Szirmay-Kalos et al. [7] use an improved version of environment mapping to take the origin of reflected or refracted rays into account when reading a cube map of the environment. They have also extended the method to handle the rear refraction on transparent objects. To be accurate, they require the environment and refractive object geometry to be star-shaped, so that all the geometry can be represented in a single cube map. As rendering these cube maps is costly, it has to be amortized to be real-time, so the method is applicable for scenes with low dynamism, or with a single moving object in a static environment.

2.5 Memory-Coherent Ray Tracing

Pharr et al.[5] have developed algorithms that use caching and lazy creation of texture and geometry to manage scene complexity and assure coherent access. The approach is also very effective at improving the performance of the ray engine[2]. However, the focus and possible areas of application are very different from our approach. They discuss how complex scenes can be effectively managed over various levels of conventional storage architectures, where rendering times are in the magnitude of several hours. On the other hand, our method is tailored for current graphics hardware, scenes not more complex than in typical interactive environments like computer games, but achieving real-time rendering frame rates.

3 ACCELERATION HIERARCHY FOR THE RAY ENGINE

CPU-based acceleration schemes are spatial object hierarchies. Although considerable research has dealt with exploiting the coherence between neighboring rays, including longest common traversal sequences [4] and image space interpolation [9], these have not altered the basic approach. That is, for a ray, we try to exclude as many objects as possible from intersection testing. This cannot be done in the ray engine architecture, as it follows a per primitive processing scheme instead of the per ray philosophy. Therefore, we also have to apply an acceleration hierarchy the other way round, not on the objects, but on the rays.

In typical applications, realtime ray casting augments scan conversion image synthesis where recursive ray tracing from the eye point or from a light sample point is necessary. In both scenarios, the primary ray impact points are determined by rendering the scene from either the eye or the light. As nearby rays hit similar surfaces, it can be assumed that reflected or refracted rays may also travel in similar directions, albeit with more and more deviation on multiple iterations. If we are able to compute enclosing objects for groups of nearby rays, it may be possible to exclude all rays within a group based on a single test against the primitive being processed. This approach fits well with the ray engine. Whenever the data of a

primitive is processed, we should find a way not to render it on the entire screen as a quad, but invoke the pixel shaders only where an intersection is possible. An obvious solution is to split the render target into tiles, render a set of tile quads instead of a full screen one, but make a decision for every tile beforehand whether it should be rendered at all. At a first glimpse, this may appear counterproductive, as, apparently, far more quads will be rendered. However, there is a set of issues that disprove concerns.

- The ray engine is pixel shader intensive, and makes practically no use of the vertex processing unit. The number of pixel shader runs, which remains crucial, is by no means increased.
- The high level test of whether a tile may include valid intersections can be performed in the vertex shader. If the intersection test fails, the vertices of the quad are transformed out of view, and discarded by clipping. Moving the vertices out of view does not require any computation, they are simply assigned an outlying extreme position.
- Instead of small quads, one can use point primitives, described by a single vertex. This eliminates the fourfold overhead of processing the same vertex data for all quad vertices, and needlessly interpolating values.

In order to implement this idea, we have to solve two problems. First, for rays grouped in the same tile, an enclosing object should be computed, for which an intersection test is fast. This computation should be performed on the GPU. Second, the data describing these enclosing objects should be accessible to the vertex shader.

The latter problem can be resolved by texture reads in the vertex shader. If data is rendered to a texture, where every texel corresponds to an enclosing object, it can be accessed when processing the appropriate tile. If we reorganize the rendering process, even this vertex texture fetch can be eliminated. Remember that we are rendering a tile for every ray casting primitive, for every possible tile position. Now if we take a tile position, and render all the primitives there at once, the enclosing object information is static. It can be passed to the vertex shader in uniform parameters. In order to do this, we have to read back the texture holding the enclosing object data from the graphics card. However, as it contains only as many texels as many tiles are used (16×16 is typical), this is not an expensive operation.

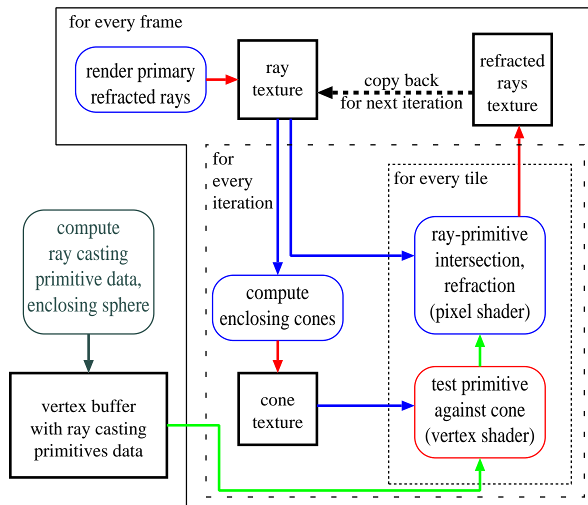

Figure 1: Block diagram of the hierarchical ray engine. Only the initial construction of the vertex buffer is performed on the CPU.

Figure 1 depicts the data flow in an application for tracing refracted rays, using the proposed method. Ray casting primitives are encoded as single vertices, and can be channeled to the shaders as a vertex buffer of point primitives. As a result of the intersection tests, the ray defining the next segment of the refraction path is written to the render target. The process is repeated for pixels, in which the path has not yet terminated. In the beginning of every iteration, an enclosing cone for rays is built for every tile, and stored in a texture. This texture is used in consequent vertex shader runs to carry out preliminary intersection tests.

Note that the cones have to be reconstructed for every new generation of rays, before the pass computing the nearest intersections. They cannot be precomputed on the CPU, and their construction must also be fast and possible to implement on the GPU.

4 CONSTRUCTION OF AN ENCLOSING CONE

We need to be able to perform the intersection test between the enclosing object and the ray casting primitive as fast as possible. At the same time, representation of both the primitives and the ray-enclosing objects must be compact, because primitive data has to be passed in a very limited number of vertex registers, and enclosing

objects must be described by a few texels. One rapid test is the intersection test between an infinite cone and a sphere. Enclosing spheres for all ray casting primitives can easily be computed, and described by a 3D position and a radius. Enclosing infinite cones of rays are described by an origin, a direction and an opening angle.

The infinite enclosing cones must be constructed in a pixel shader, in a pass before rendering the intersection records themselves. Note that in a practical application, the rays to be traced will be different for every frame, and for every level of refraction, so the reconstruction of the cones is also time critical. Therefore, a fast incremental approach is preferred over a tedious one, which could possibly produce more compact results, via, for instance, linear programming. The algorithm goes as follows:

1. Start with the zero angle enclosing cone of the first ray.
2. For each ray
   (a) Check if the direction of the ray lies within the solid angle covered by the cone, as seen from its apex. If it does not, extend the cone to include both the original solid angle and the new direction.
   (b) Check if the origin of the ray is within the volume enclosed by the cone. If it is not, translate the cone so that it includes both the original cone and the origin of the ray. The new cone should touch both the origin of the ray and the original cone, along one of its generator lines.

Both steps of modifying the cone require some mathematics. Let $\vec{x}$ be the axis direction of the cone, $\vec{a}$ its apex, φ the half of the opening angle, $\vec{r}$ the direction of the ray, and $\vec{p}$ its origin.

First, if the solid angle defined by the cone does not include the direction of the ray, the cone has to be extended (See Figure 2). This is the case if $\vec{x} \cdot \vec{r} < \cos\varphi$. Then, the generator direction $\vec{e}$, opposite to the ray direction, has to be found. If $\vec{r}$ is projected onto $\vec{x}$, the direction from $\vec{r}$ to the projected point defines $\vec{q}$:

$$\vec{q} = \frac{(\vec{x} \cdot \vec{r}) \cdot \vec{x} - \vec{r}}{|(\vec{x} \cdot \vec{r}) \cdot \vec{x} - \vec{r}|}.$$

Then $\vec{e}$ is found as a combination of $\vec{x}$ and $\vec{q}$:

$$\vec{e} = \vec{x} \cdot \cos\varphi + \vec{q} \cdot \sin\varphi.$$

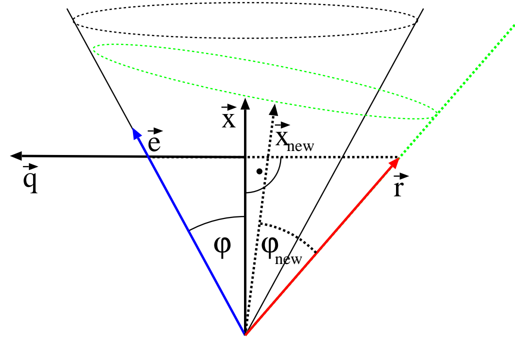

Figure 2: Extending the cone.

The new axis direction should be the average of $\vec{e}$ and $\vec{r}$, and the opening angle should also be adjusted:

$$\vec{x}_{\text{new}} = \frac{\vec{e} + \vec{r}}{|\vec{e} + \vec{r}|}, \qquad \cos\varphi_{\text{new}} = \vec{x}_{\text{new}} \cdot \vec{r}.$$

Given the information we had, which does not include any knowledge of rays already within the cone, we can state that this method computes the cone of minimum opening angle necessary to hold the given infinite semi-line and the cone.

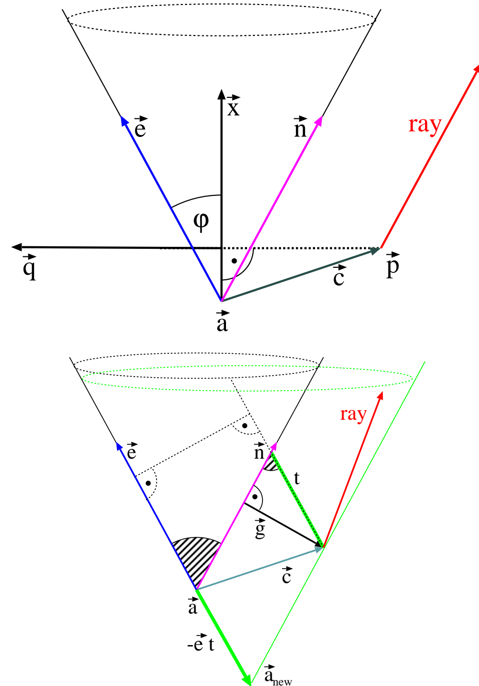

Figure 3: Finding the near and far generators, and translating a cone.

Translating the possibly extended cone to include the origin is somewhat more complicated, but follows the same trail (See Figure 3). First the *near-*

est generator direction $\vec{n}$ and the *farthest* generator direction $\vec{e}$ are found just like before. The vector $\vec{c} = \vec{p} - \vec{a}$ plays the role what $\vec{r}$ had in the previous computation;

$$\vec{q} = \frac{(\vec{x} \cdot \vec{c}) \cdot \vec{x} - \vec{c}}{|(\vec{x} \cdot \vec{c}) \cdot \vec{x} - \vec{c}|}.$$

Like before,

$$\vec{e} = \vec{x} \cdot \cos\varphi + \vec{q} \cdot \sin\varphi, \qquad \vec{n} = \vec{x} \cdot \cos\varphi - \vec{q} \cdot \sin\varphi.$$

We want to translate the cone along generator $\vec{e}$ so that the generator $\vec{n}$ moves to cover the ray origin $\vec{p}$ (Figure 3, on the right). The distance vector $\vec{g}$ between the origin and the nearest generator is found as:

$$\vec{g} = \vec{c} - (\vec{n} \cdot \vec{c}) \cdot \vec{n}.$$

The translation distance t and the new apex position are:

$$t = \frac{\vec{g}^2}{\vec{e} \cdot \vec{g}}, \qquad \vec{a}_{\text{new}} = \vec{a} - \vec{e} \cdot t.$$

Using the two steps in succession, we find a new cone that includes both the previous cone and the new ray, has a minimum opening as a priority, and was translated by a minimum amount as a secondary objective. Of course, knowing nothing about the rays already included in the cone, we cannot state that the computation achieves an optimal result in any way. However, it is conservative and mostly needs vector operations, fitting well in a pixel shader. Furthermore, cone construction is only performed once for every iteration of ray casting.

The computation of enclosing spheres of ray casting primitives is done on the CPU, at the same time when all the scene data is processed. The result is a vertex buffer, in which all ray casting primitives are encoded as vertices. Note that the position value slot can be used for passing the enclosing sphere data, as it will simply be exchanged with the tile position in the vertex shader. The cone intersects the sphere if

$$\varphi > \arccos[(\vec{v} - \vec{a}) \cdot \vec{x}] - \arcsin[r / |\vec{v} - \vec{a}|],$$

where $\vec{a}$, $\vec{x}$ and φ describe the cone as before, $\vec{v}$ is the center of the sphere and r is its radius.

The vertex data describing a triangle primitive may be composed as:

position enclosing sphere center and radius

normal the normal and offset of the triangles plane

texture 0-2 pre-processed triangle vertex position data for fast intersection computation

further texture registers normals at the triangles vertices

further texture registers texture coordinates at the triangles vertices

5 Excluding terminated ray paths

In a typical recursive raytracing problem, we may divide the geometry into ideally refractive or reflective surfaces, and locally shaded ones. We will refer to these groups as ideal and non-ideal surfaces. We start off with a texture of eye and light rays. Our method should follow these rays through multiple ideal reflections and refractions, until the exiting rays do not hit an ideal surface any more. These final rays we get as a result can either be traced against the non-ideal geometry of the scene, or used in any other method like environment mapping or caustics generation.

Firstly, there may be pixels in which no refractive or reflective surface is visible. Furthermore, in every iteration replacing rays with their reflected or refracted successors, there will be rays not arriving on any reflective of refractive surface, producing no output. It is desirable that for those pixels where there is no ray to trace, the pixel shader is not invoked at all. This is achieved using the stencil buffer and early stencil testing. When rendering intersections, every bit of the stencil serves as a flag for a specific iteration. The stencil read and write masks select the flag bit of the previous and current iterations, respectively. Should ray casting fail to hit any object, the stencil bit will not be set, and in the next iteration the pixel will be skipped. With an eight bits deep stencil buffer, this allows for eightfold reflection or refraction to be traced. Further iterations are possible without excluding further terminated paths.

6 RELATION TO OTHER APPROACHES

The ray engine was designed to be a general purpose raytracer, with no preconceptions on what rays there are to be traced. While this generality could be considered a great advantage, current advancements in technology and research make

the ray engine approach obsolete in all but few areas of application. Firstly, real time global illumination is better supported by texture-based methods making use of precomputed maps or rendered environments. Secondly, in offline computations, where ray casting is still essential, ray casting acceleration schemes well known on the CPU will very soon be adopted for the graphics hardware [6]. The acceleration structure must still be built on the CPU, rendering the approach incapable of realtime rendering of dynamic scenes. That practically leaves those two areas for realtime ray casting which our hierarchical ray engine specializes in: visible refractive objects and caustics. Our algorithm can outperform the ray engine by making use of the coherence of rays in these particular problems.

7 RESULTS

We have implemented the ray engine for tracing refracted eye rays, without the CPU-GPU load sharing scheme, and our algorithm (Figure 5), to compare their performance. Both algorithms also performed a refracted direction calculation for every intersection test.

7.1 Ray coherence

A basic assumption of our approach is that rays within a tile will remain more or less coherent after multiple refractions, and in most cases they still can be enclosed by tight cones. Images in Figure 4 shows how the opening angle of the cones increases with consequent refractions. Obviously, the cones are extremely accurate in the first few steps, and larger openings only appear near contours. Later on some cones expand as new refraction contours appear, but the main yield in this phase is that homogenous or already terminated areas have already been identified, and they are discarded without the CPU rearranging the remaining rays into a new array, as in the Ray Engine approach.

For the hierarchical ray engine implementation, we divided the 256×256 ray array into 16×16 tiles. The size of the tiles is somewhat arbitrary, but limited by the maximum size of point primitives the hardware can render. The optimum may be dependent on the complexity of the scene. For our test scenes, we found that further decreasing the tile size resulted in performance loss, meaning that the overhead of more processed vertices and larger textures became more significant than the gain from better coherence.

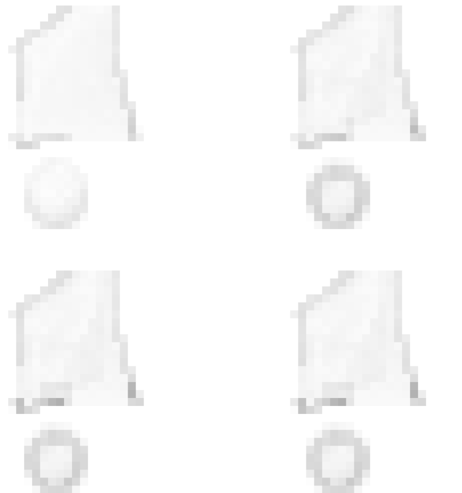

Figure 4: Cone angles for 1, 2, 4 and 8 iterations, respectively. Brighter texels indicate a lower angle.

As depicted in Figure 1, the enclosing cones for the tiles were computed in a shader. The results shown in Figure 7 were obtained for different geometries, with the maximum path length set to 5, on an NV6800GT. A refractive surface was visible in every pixel. Considering that it takes at least two iterations to discard a path that has entered an object, a minimum estimate for the throughput can be given. With 256×256 pixels, 5000 triangles, 2 iterations, and an FPS of 0.23 our ray engine implementation computed $150M$ intersections per second. With the hierarchical approach achieving 3 FPS, this figure reaches $2G$. The improvement over the naive approach is very much dependent on the complexity of the geometry. If there are few primitives, the ray casting does not take much time, and the constant overhead of constructing the enclosing cones does not pay off. However, as the number of primitives increases, the situation is reversed. Already at a triangle count of 100, the new algorithm runs twice as fast. Furthermore, for 256×256 eye rays, all hitting a refractive surface, frame rates enough for real time rendering have been achieved. This makes it possible to interactively and accurately render visible refractive objects, or, when using the technique to trace photon paths, correct caustic patterns.

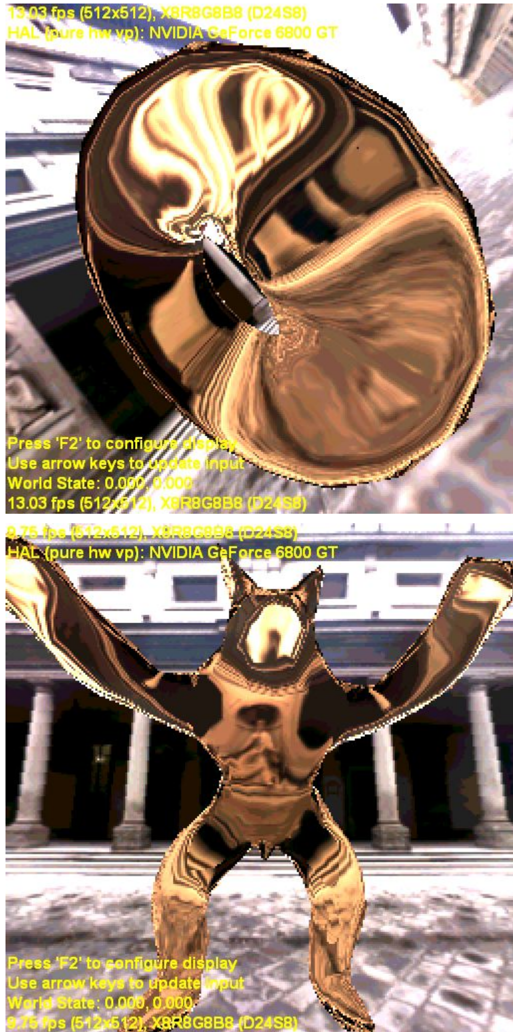

Figure 5: Images rendered using the hierarchical algorithm at 256×256 resolution.

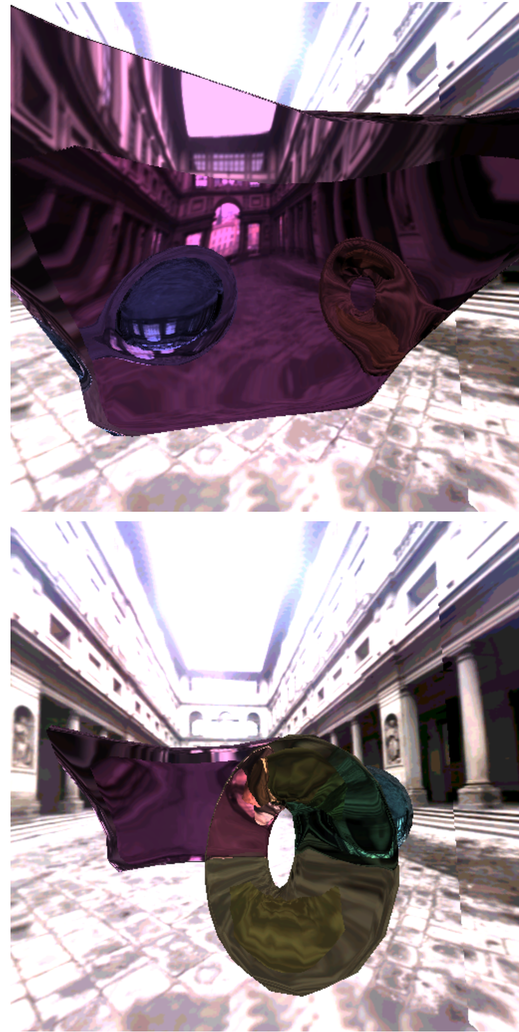

Figure 6: Images rendered using the hierarchical algorithm at 512×512 resolution.

8 COMPARISON WITH OTHER ALGORITHMS

Compared to the brute force algorithm, Foley and Sugerman[3] reported relative speedups for their grid and kd-tree GPU acceleration structures between a factor of 5 and 9. Their test scenes contained up to 100000 triangles. As complex acceleration structures have to be built using the CPU, the algorithms are very well suited for high triangle count static scenes, and impose no constraints on the coherence of the rays whatsoever.

With a quite different set of preferences, the hierarchical ray engine can achieve a relative speedup of 15, outperforming the spatial hierarchies. However, the algorithm is linear in the number of triangles, making it less suitable for scenes with more than several thousand triangles, and it also assumes strongly coherent rays. However, as stated in the introduction, the problem of caustics and visible multiple refractions, the only applications of ray tracing where there is no reasonable incremental alternative, require these features exactly.

The approximate ray tracing approach[7] has a very similar area of applications compared to our algorithm. Its main limitations are that it uses pre-computed distance impostor textures, and only two refractions can be handled. It is impossible to render a refractive object seen through a refractive object. Furthermore, the refraction computation is only accurate for a single star-shaped object, the geometry of which can be captured in a single cube map. Therefore, while the approxi-

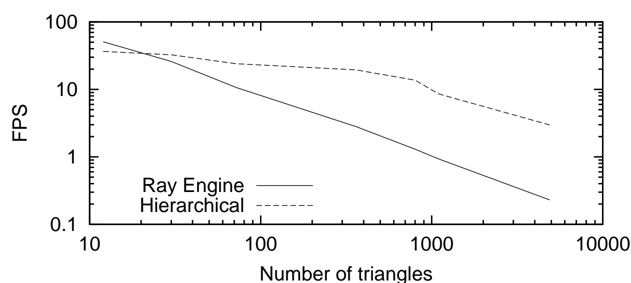

Figure 7: Frames per second rates achieved for rendering refractive objects.

mate ray tracing is extremely fast and convincing for some spacial cases, our algorithm offers accurate multiple refractions and a potential to handle dynamic objects or even liquids.

9 FUTURE WORK

The current implementation is only using the algorithm to query an environment map for rays that do not hit any refractive surface. Although it is theoretically straightforward, we still need to work on an implementation that demonstrates the method's applicability for tracing scenes with non-ideal objects, and, more importantly, for generating caustics.

REFERENCES

[1] J. Arvo and D. Kirk. A survey of ray tracing acceleration techniques. In *An Introduction to Ray Tracing*, pages 201–262. 1989.

[2] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. In *Proc. Graph. Hw. 2002*, pages 1–10, 2002.

[3] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 15–22, New York, NY, USA, 2005. ACM Press.

[4] V. Havran. *Heuristic Ray Shooting Algorithms.* Czech Tech. Univ., Ph.D. dissertation, 2001.

[5] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 101–108, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[6] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. on Graph.*, 21(3):703–712, 2002.

[7] László Szirmay-Kalos, Barnabás Aszódi, István Lazányi, and Mátyás Premecz. Approximate ray-tracing on the gpu with distance impostors. In *Computer Graphics Forum, Proceedings of Eurographics 2005*, volume 24, Dublin, Ireland, 2005. Eurographics, Blackwell.

[8] I. Wald, C. Benthin, P. Slussalek, and M. Wagner. Interactive rendering with coherent ray tracing. In *Eurographics '01*, 2001.

[9] Bruce Walter, George Drettakis, and Steven Parker. Interactive rendering using the render cache. In *Rendering techniques '99*, volume 10, pages 235–246, Jun 1999.