Eurographics Symposium on Rendering (2007)
Jan Kautz and Sumanta Pattanaik (Editors)

Whitted Ray-Tracing for Dynamic Scenes using a Ray-Space Hierarchy on the GPU

D. Roger[1] U. Assarsson[2] N. Holzschuch[1]

[1] ARTIS/INRIA Grenoble University [2] Chalmers University of Technology

Abstract

In this paper, we present a new algorithm for interactive rendering of animated scenes with Whitted Ray-Tracing, running on the GPU. We focus our attention on the secondary rays *(the rays generated by one or more bounces on specular objects), and use the GPU rasterizer for primary rays. Our algorithm is based on a ray-space hierarchy, allowing us to handle truly dynamic scenes without the need to rebuild or update the scene hierarchy. The ray-space hierarchy is entirely built on the GPU for every frame, using a very fast process. Traversing the ray-space hierarchy is also done on the GPU; one of the benefits of using a ray-space hierarchy is that we have a single shader, and a fixed number of passes. After traversing each level of the hierarchy, we prune empty branches using a stream reduction method. We present two different stream reduction methods, a fast one using a hierarchical algorithm, and an easy one using the Geometry shaders. Our algorithm results in interactive rendering with specular reflections and shadows for moderately complex scenes (*$\sim 700K$ *triangles), handles any kind of dynamic or unstructured scenes without any pre-processing, and scales well with both the scene complexity and the image resolution.*

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics processors I.3.7 [Computer Graphics]: Raytracing

1. Introduction

Global illumination methods for image synthesis start with the description of a virtual scene (its geometrical and material properties) and compute a picture of the scene as seen from a virtual observer. Among global illumination methods, Whitted ray-tracing [Whi80] uses ray-tracing [App68] to simulate reflections on specular objects. Ray-tracing and Whitted ray-tracing have been the object of many research, and are now widely used for production of photorealistic images. Several papers describe real-time or interactive ray-tracing.

Specular reflections is one area where ray-traced images typically differ from images generated with rasterization. These effects depend on *secondary rays*, as opposed to the primary rays (rays originating from the viewpoint of the camera). Secondary rays include reflected rays (after one or several reflections), refracted rays and shadow rays. While primary rays are highly coherent, making it easier to optimize the computations, *e.g.* using caching schemes together with the scene hierarchies, secondary rays exhibit much less coherence: two secondary rays generated from neighbouring points can intersect with objects that are far away in the scene. As a consequence, efficient computation of secondary rays is a harder problem than for primary rays.

Most ray-tracing algorithms use a scene hierarchy to speed up the computation of the intersection between each ray and the objects in the scene. But with animated scenes, the scene hierarchy must be rebuilt or updated at each frame, a process that is slowing down the computations.

In this paper, we present a new algorithm for interactive Whitted ray-tracing of dynamic scenes, using a ray-space hierarchy that is generated and processed on the GPU at each frame. Although our algorithm can handle all kinds of rays, including primary rays, we have elected to focus on the secondary rays, as they are both more *interesting* in terms of pictures generated and more *difficult* to compute. We let the GPU handle the primary rays, using rasterization, Z-buffer and per-pixel lighting. We also use the GPU to generate the first set of secondary rays (the rays caused by the first bounce on the scene), and to build a ray-space hierarchy. We then traverse this hierarchy on the GPU, starting at the root node and descending towards the leaves, corresponding to individual rays. For each node of the hierarchy (corresponding to a bundle of rays), we maintain the set of triangles intersected by this node. After traversing each level of the hierarchy, the stream of triangles is pruned of its empty nodes, using a stream reduction technique. We present two different stream reduction methods: one that is faster and relies on a hierarchical algorithm, and one that is easier to implement, using the Geometry shaders.

Our algorithm runs entirely on the GPU, without any communication to or from the CPU. Our experiments show that it runs interactively, with specular reflections, for moderately complex scenes. We can handle any kind of dynamic or unstructured scenes without any pre-processing. Finally, our algorithm scales well with both the scene complexity and the image resolution.

Our paper is organized as follows: in the next section, we review previous work on interactive rendering of specular reflections, interactive ray-tracing, GPU ray-tracing and ray-space hierarchies. We then present our algorithm for ray-tracing (section 3). A key step in our algorithm is the stream-reduction pass, for which we have designed two possible implementations; both will be discussed in section 4. Finally, we present our results in section 5, then conclude and present directions for future work.

2. Previous work

Ray-casting was formally introduced by Appel [App68] as a technique for visible surface determination. Whitted [Whi80] used ray-casting for the generation of photorealistic images (including recursive specular reflections). Ray-tracing has been the subject of intensive research, dealing with efficient acceleration methods: scene hierarchies, efficient parallel implementations or caching schemes.

We focus here on the previous work related to real-time or interactive ray-tracing, ray-space hierarchies and GPU ray-tracing. We also review methods for approximate specular reflections on the GPU.

Real-time and Interactive ray-tracing For a state of the art in interactive ray tracing, we refer the reader to Wald and Slusallek [WS01]. In recent years, several papers used a k-D tree for the scene hierarchy [RSH05, HSHH07]. For dynamic scenes, however, Wald *et al.* [WIK*06] showed that construction and updates of k-D trees can significantly slow down the rendering process. To overcome this limitation, they used a grid-based hierarchy; Ize *et al.* [IRWP06] studied different parallel algorithms for the efficient construction of this grid hierarchy.

Lauterbach *et al.* [LYTM06] and Wald *et al.* [WBS07] used a *Bounding Volume Hierarchy* (BVH) for interactive ray-tracing of dynamic scenes. BVHs are easier to compute and update than k-D trees for dynamic scenes. Eisemann *et al.* [EGMM07] presented two methods for fast updates of BVHs. Wächter and Keller [WK06] presented a new data structure, the *Bounding Interval Hierarchy* (BIH), that is both fast to rebuild or update on dynamic scenes and efficient for traversal.

Ray-space hierarchies Amanatides [Ama84] suggested grouping rays together for faster rendering and more realistic effects; he traced cones instead of rays, and used them for soft shadows and glossy reflections. He also used a hierarchy of cones for faster tracing of primary rays. Hanrahan and Heckbert [HH84] used beam tracing for more accurate ray-tracing and anti-aliasing, but without a hierarchical representation. Arvo and Kirk [AK87] created a complete hierarchy in 5-dimensional ray-space; rays were grouped in 5D hypercubes, resulting in faster ray-tracing. Ghazanfarpour and Hasenfratz [GH98] used hierarchical polyhedral beams of rays for faster tracing of primary and shadow rays. Nakamaru and Ohno [NO97, NO02] introduced *breadth-first ray-tracing*, where they keep the rays in memory and process the objects sequentially.

Chung and Field [CF99] have combined a ray-space hierarchy with a scene hierarchy for faster rendering. Similarly, Reshetov *et al.* combined a ray-space hierarchy on the primary rays with a k-D tree scene hierarchy.

GPU-based Ray Tracing Carr *et al.* [CHH02] made the observation that ray-casting is a crossbar on rays and primitives, while pixel shading is a crossbar on pixels and primitives. They devised a method to use the pixel shading crossbar to compute ray-triangle intersections. Purcell *et al.* [PBMH02] ported the entire ray-tracing algorithm, using a grid for the scene hierarchy, tracing one ray per pixel. Because of the complexity of the ray-tracing algorithm, they had to use four different pixel shaders: for ray spawning, ray traversal, ray-triangle intersection and shading. Combined with the fact that the rays are in different phases, this limits their peak GPU performance to 10 % [CHCH06]. Further research have extended this work [Chr04, KL04, TS05], but all suffer from the same drawback and do not exploit the full GPU performance.

Ernst *et al.* [EVG04] used a scene hierarchy based on a k-D tree, but they required a fixed maximum stack depth. Foley and Sugerman [FS05] extended this algorithm to a stack-less traversal; they report 20 % GPU efficiency. Horn *et al.* [HSHH07] ported [FS05] to run in a single shader pass, using GPU branching and looping.

Thrane and Simonsen [TS05] and Carr *et al.* [CHCH06] used a Bounding Volume Hierarchy instead of a k-D tree. Carr *et al.* [CHCH06] stored their BVH as a hierarchical geometry image.

Szécsi [Szé06] used a two-level ray-space hierarchy to trace refraction rays on the GPU. The first level of the hierarchy is processed by the vertex shader, and the second level by the fragment shader.

Our work shares common points with several of these previous work. Like Horn *et al.* [HSHH07], we use the programmable pixel shader to handle the primary rays, and we only trace the secondary rays, but we use a ray-space hierarchy instead of a k-D tree scene hierarchy. Like Szécsi [Szé06], we use a ray-space hierarchy, but we build the complete hierarchy, as opposed to only the bottom two levels.

Specular reflections on the GPU Roger and Holzschuch [RH06], Estallela *et al.* [EMDT06] and Szirmay-Kalos *et al.* [SKALP05] compute approximate specular reflections on the GPU, searching for optical paths of extremal length. Our work differs from these, as we are computing a full Whitted ray-tracing solution, without any approximation.

3. Algorithm

3.1. Overview

Our algorithm works the following way:

1. Render the scene, with non-specular direct lighting effects;
2. Generate the first set of secondary rays;
3. Build the ray-space hierarchy from these rays;
4. Intersect the ray-space hierarchy with the scene:
   a. maintain a stream of (hierarchy nodes, triangles).
   b. recursively subdivide the nodes,
   c. discard irrelevant triangles,
5. Final ray-triangle intersection and shading.

The first step is done using a standard rasterizer, with pixel-based lighting (using fragment shaders). The same shader also outputs the first set of secondary rays in a separate render target, with their starting point and direction. The rays are indexed by the corresponding fragment position. Building the ray-space hierarchy is then a fast step, entirely done on the GPU (see Section 3.2), for each frame, at a cost of ≈ 2 ms for a resolution of 1024×1024.

Intersecting this ray-space hierarchy with the scene is the core of the algorithm (see Section 3.3). Each node in the hierarchy represents a bundle of rays. We compute the set of triangles whose bounding sphere intersects this bundle. We start with the triangles intersecting the root node, and descend along the hierarchy.

At the end of the hierarchy traversal, for each ray in the original set, we have the set of triangles whose bounding sphere it intersects. In a final pass, we compute the actual ray-triangles intersection, keep the closest intersection, compute its shading and output the corresponding fragment.

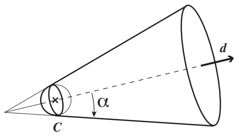


Figure 1: *We use a cone-sphere structure for our ray-space hierarchy. Each node is defined by a sphere and a cone.*

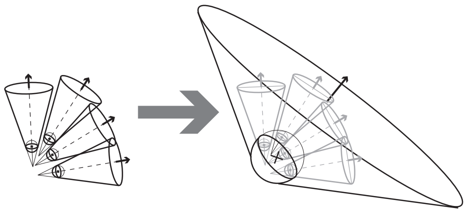


Figure 2: *The parent node is constructed as the enclosing cone-sphere for the four children.*

GPUs are not well adapted to hierarchical data structures. They are, in essence, SIMD machines and for optimal results, neighbouring fragments should run in the same branching conditions, in contradiction with the nature of hierarchical computations. We resolved this issue by separating the hierarchy traversal in two passes: the first pass runs the same shader on all data entries, with a fixed number of operations and a fixed number of outputs. In a second pass, we delete irrelevant outputs, reducing the size of the working buffer. This deletion pass is called a *stream reduction* pass, and it is essential to our algorithm. We have designed two different stream reduction methods (see Section 4): a fast, hierarchical method using fragment shaders, and an easy-to-implement method using geometry shaders.

3.2. Building and storing the ray hierarchy

The first step in our algorithm is building the ray hierarchy. Our algorithm can work with any kind of ray hierarchy, such as polyhedral beams or cones of rays. For practical reasons,

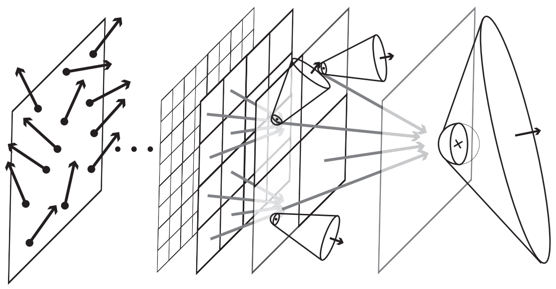


Figure 3: *We start with the set of secondary rays, and recursively build the enclosing cone-sphere for each hierarchical level.*

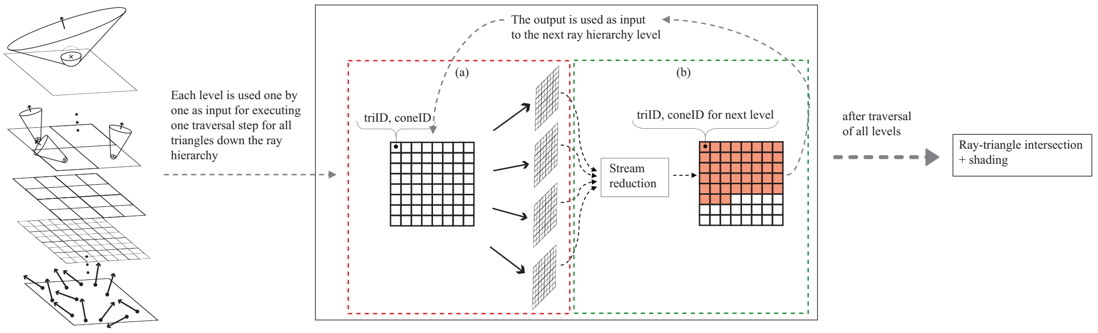

Each level is used one by one as input for executing one traversal step for all triangles down the ray hierarchy
The output is used as input to the next ray hierarchy level
(a)
(b)
triID, coneID
triID, coneID for next level
Stream reduction
after traversal of all levels
Ray-triangle intersection + shading

Figure 4: *Traversing the ray hierarchy: after the construction of the ray hierarchy, we store each scene-triangle in a texture, so that each texel contains a triangle ID and the cone ID of the root node. In step (a), we send this information to the fragment shader, which computes the intersections of the bounding sphere of the triangle with the four children cones, each one being bound to a separate render target. If the intersection is not empty, the shader outputs the cone index of the child together with the triangle ID, and otherwise a null node. In step (b), we remove all the null nodes using stream reduction and merge the results into a single texture, used as input for the next level. Steps (a) and (b) are repeated once for each level down the hierarchy. The final output contains, for each ray, the ID of all triangles whose bounding sphere it intersects.*

we have elected to use a combination of a cone and a sphere (see Figure 1). We define the sphere so that in encloses the starting points of all the rays in the ray bundle, and the cone so that it contains the sphere and includes all the rays in the ray bundle. The efficient part of the ray bundle contains the sphere and the part of the cone that is in front of the sphere. The remaining part of the cone is not used for the intersections.

This structure can be stored in a very compact way, each node requiring just 8 floats: 4 for the sphere (center and radius) and 4 for the cone (direction and spread angle, α). Note that the 3D point we store is the center of the sphere and not the apex of the cone. At the upper levels of the hierarchy, the ray bundles group rays with very different directions, so the spread angle of the cone can be larger than $\frac{\pi}{2}$, allowing a cone to enclose the entire space.

The ray hierarchy is constructed bottom-up. We start with the first set of secondary rays (rays reflected by visible specular objects). These rays are generated while rendering the scene, using a fragment shader to output the origin and direction of the ray in a separate render target. This forms the bottom layer of the hierarchy, with the sphere radii and the cones' spread angle, α, equal to zero to represent the exact rays. Each parent node is then created by computing the union of the child nodes (see Figure 2). This hierarchy construction is done on the GPU, in a fixed number of passes: for each node, we access its four children and compute the enclosing node (see Figure 3). This process is very similar to generating mip-maps.

Our ray-space hierarchy is indexed by the screen position of the rays. The lowest level has the same size as the screen: each ray corresponds to a single pixel (since the specular reflectors usually do not cover the entire screen, some pixels in the screen do not correspond to an actual ray). We keep this structure for the upper levels: each node in the hierarchy corresponds to an area of the screen, and groups together the secondary rays underlying this area. This spatial localisation of the nodes gives us the parent-children relationship without having to store it explicitly.

By construction, each node in our ray-space hierarchy encloses its four children. If a triangle does not intersect the current node, it will not intersect any of the children either, and can safely be discarded.

3.3. Traversing the ray hierarchy

Once we have built the ray-hierarchy, we traverse it for computing the intersections of the rays with the scene triangles. We do this in a top-down manner, for each node in the hierarchy creating the set of triangles potentially intersected by this node. We start with the triangles intersected by the root node, then descend the hierarchy. Each pass updates this information for the current level of the hierarchy, then sends the result to the next pass, working on the next level of the hierarchy (see Figure 4).

The required information is stored in a texture, so that each element contains the node ID and the triangle ID. Initially, the texture contains one element for each triangle in the scene, with the ID of the triangle and the root node ID.

Each level of the hierarchy is processed in a single pass, running the same shader on all these texture elements: for each element, we retrieve the four cone-sphere children corresponding to the cone ID, the bounding sphere of the triangle corresponding to the triangle ID and check their intersection. For each children, we output in a separate render target either the children ID and the triangle ID if there is an intersection, or an empty element otherwise.

After this traversal pass, we have four textures, each of

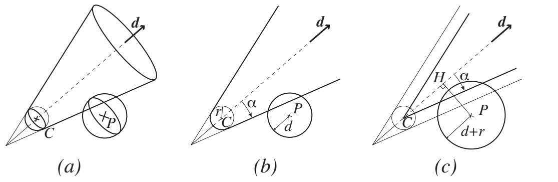


Figure 5: *Testing the intersection between a node and the bounding sphere of a triangle (a) reduces to a 2D problem (b). It is equivalent to testing the intersection between a reduced cone and an enlarged sphere (c).*

them with the same number of elements as the input texture, but with many empty elements. A stream-reduction pass (described in Section 4) removes the empty elements and packs the textures in a single texture, used as input for the next step.

The number of traversal passes is equal to the depth of the hierarchy, $\log_2 X$, where X is the width of the picture in pixels (*i.e.* 9 passes for a 512×512 picture).

At the end of the hierarchy traversal, we have the ID of each initial ray, with the ID of the triangles potentially intersected by this ray. We compute the actual ray-triangles intersections, select the closest intersection point, compute the shading and illumination and display the result.

3.3.1. Intersection between a node and a bounding sphere

The most frequent operation in our algorithm is computing whether the bundle of rays corresponding to a node is intersecting with the bounding sphere of a triangle. Given the symmetry of revolution, this is actually a 2D operation (see Figure 5).

We assume that we have a ray hierarchy node defined by a sphere (C, r) and a cone $(\boldsymbol{d}, \alpha)$, and we want to check the intersection with the bounding sphere of a triangle, defined by its center P and its radius d.

The problem is equivalent to testing the intersection between the cone of apex C, direction $\boldsymbol{d}$ and spread angle α with the sphere of center P and radius $d + r$:

$$\text{return} \quad \left(\overline{CH}\tan\alpha + \frac{d+r}{\cos\alpha} \geq \overline{HP}\right) \qquad (1)$$

3.4. Memory considerations

As we traverse the ray hierarchy, our algorithm stores all the pairs (hierarchy node, triangle) for which there is a potential intersection. We store these pairs in a large texture (2048×2048), where each texel contains two `int16` for the indices of the hierarchy nodes and two `int16` for the indices of the triangle.

During refinement, the total number of pairs (hierarchy node, triangle) can get larger than the number of texels. This happens when the ray-space hierarchy contains nodes with a large spatial or angular extent at the lower levels. Each of these node intersects with a large number of triangles. Large nodes at the upper levels of the hierarchy do not cause this problem, simply because there is a smaller number of nodes. A single discontinuity between two different reflectors will not cause this issue, but an irregular, bumpy or fractal reflector will.

When this happens, we implemented a simple workaround: the scene is subdivided into batches, each batch is processed independently and then the results are combined. Our experiments show that the rendering time for each batch is proportional to the number of triangles it contains, so subdividing the scene into batches will actually result in almost the same rendering time, except for the extra cost for processing each batch: in our experiments, ≈ 30 ms. With this technique, memory overflow is not predictable, but the system can react to it at the next frame: one possible strategy is to divide in two the batches that led to overflow. This way, any problem disapears within a few frames on still images.

Another straight forward workaround would be to read back the overflowing part of the stream to the CPU and process it in a separate batch after the current batch is fully processed. This can create a maximum of d batches, where d is the depth of the tree, temporarily stored on the CPU-side. The GPU-CPU bandwidth should not be a major problem here, since over a hundred 1024×1024 frames can be sent per second over the PCI Express bus.

This subdivision into batches can also be used to run our algorithm on very large scenes: as long as a single batch can be processed by our ray-tracing engine, there is no limit on the size of the scene.

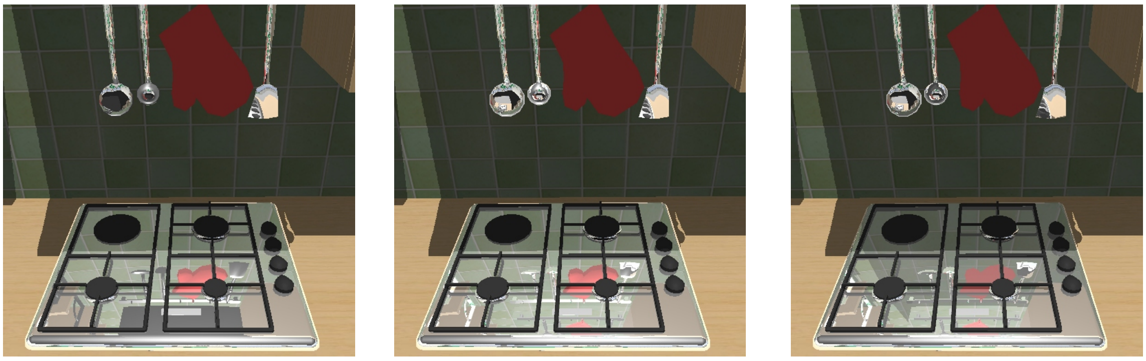
3.5. Other secondary rays

3.5.1. Shadow rays

Our ray-tracing engine is generic, and can handle any kind of rays, not just the first bounce of secondary rays. We have also used it for shadow rays (see Figure 6(c)). We know that all shadow rays share a common termination (the point light source). For a better efficiency, we revert the directions of the shadow rays before computation, so that the light source is now their common starting point. Thus, we build a very tight ray-space hierarchy, with a null dimension in space.

3.5.2. Further light bounces

We also use our engine for further bounces (see Figure 6(b)). When a ray hits a specular surface, we generate the reflected ray for this pixel. We then send the set of reflected rays to our ray-intersection engine, with the same steps as for the rays



(a) 1 specular reflection (302 ms). (b) 2 specular reflections (674 ms). (c) 2 specular refl.+shadows (993 ms).

Figure 6: *Our ray-tracer handles multiple reflections and shadow rays (Kitchen scene, 83K polygons,* 512×512 *pixels). Please note that the entire scene is visible in the reflection. See also the color plate.*

generated by the first bounce: building the ray hierarchy, intersecting it with the scene. Each further bounce of light has a computational cost, making the overall algorithm slower, but increases the realism of the images generated.

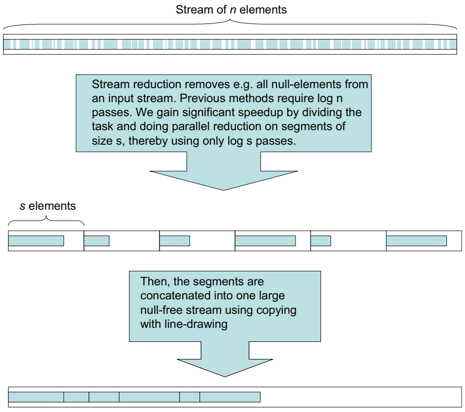
The rays generated by further light bounces have even less coherency than the rays generated in the first pass, making the ray-space hierarchy looser. However, our algorithm is robust enough to handle such hierarchies. Also, the rays corresponding to further light bounces are usually less frequent in the picture, which compensates the looseness of the hierarchy.

4. Stream reduction pass

After each traversal of a level of the ray-space hierarchy, we need to remove the empty elements in each stream. This is called a *stream reduction* pass, and it is essential to our algorithm: otherwise, the number of elements, including the empty ones, will grow to $\#pixels \times \#triangles$ and fill the available memory. Stream reduction is also an essential element of many GPGPU algorithms [Hor05]. For a better efficiency of our ray-tracing algorithm, we have designed two new stream reduction methods; both of them are faster than Horn's [Hor05], as well as Sengupta *et al.* extension [SLO06]. The first method is a hierarchical version of [Hor05], that is faster and works on older GPUs, while the second methods requires Geometry Shaders, currently available only on GeForce 8800 cards, but is easier to implement.

4.1. Hierarchical stream reduction

Our first stream reduction method is a hierarchical version of Horn's [Hor05]. We split the streams to be reduced into smaller components of a fixed size, *s*. On each component, we run a standard stream reduction method, such as the one described by [Hor05]. We then concatenate the results of each pass using line drawing. This concatenation step is easier because we know the number of non-empty elements for each component. See Figure 7 for an outline of the algorithm.



Stream reduction removes e.g. all null-elements from an input stream. Previous methods require log n passes. We gain significant speedup by dividing the task and doing parallel reduction on segments of size s, thereby using only log s passes.

Then, the segments are concatenated into one large null-free stream using copying with line-drawing

Figure 7: *Our hierarchical stream reduction algorithm removes all null-elements from the input stream. We split the stream into small components of a fixed size, s, then run a standard parallel stream reduction on each component (in log s passes). Finally, we concatenate the resulting segments in one render pass using line drawing.*

Our hierarchical stream reduction method is five times faster than the non-hierarchical method in [Hor05] for streams of $4k \times 4k$ elements. The non-hierarchical version has complexity of $O(n \log n)$, while our method has a complexity of $4n + n \log s$. Experimentally, we have found that we get the best results for $s = 64$ elements, for streams compatible with current graphics cards (up to 16 M elements).

4.2. Using Geometry Shaders for stream reduction

Recent graphics cards, such as the NVidia GeForce 8800, offer the ability to program the geometry engine, using *geometry shaders*. The geometry shader is the only programmable part of the graphics pipeline where the number of outputs is not necessarily equal to the number of inputs, making it a good candidate for a simple stream reduction.

We have implemented a stream reduction pass using geometry shaders: each stream is mapped into a vertex-buffer object, which we render using points. The geometry shader discards the empty nodes (no primitives are output) and sends non-empty nodes into another vertex buffer object, ready for the next iteration. With this method, we have to run the hierarchy traversal on the vertex shader, instead of the fragment shader. The output of the hierarchy traversal (through vertex shaders) is sent as input to the geometry shader. This makes sense because on the GeForce 8800 architecture, the vertex processors and the fragment processors are identical, and therefore run at the same speed.

Stream reduction using the geometry engine is easier to implement, but hierarchical stream reduction is 40 % faster for a stream of 4 M elements.

5. Results and Analysis

Unless otherwise specified, all the timings in this section were recorded on a Pentium 3.2 GHz with a NVidia GeForce 8800 GTS, with 640 Mb of memory.

In all our timings, we have used the rendering time, expressed in milliseconds. We measured the time it takes to render a complete picture (including both rasterization and ray-tracing). We used this value because it makes it easier to detect linear or sub-linear behaviour. The number of frames per second is simply equal to 1000 divided by this rendering time.

5.1. Test scenes

We have tested our algorithm on four different test scenes (see Figure 8). The Patio and Alley scenes are scenes with variable complexity, where we add or subtract objects to create scenes where we change the polygon count without changing the general nature of the scene. The Kitchen scene has a fixed complexity, but several different specular reflectors. We used two BART Museum scenes [LAAM01] to study the behaviour of our algorithm with unstructured specular reflectors. The Alley scene is an example of a large scene (up to 2.3 million triangles), for which we run our algorithm in several batches.

We have also tested our algorithm when we change the nature of the specular reflector. For this we used several LOD versions of the Stanford Bunny, and a statue model (see Figure 8, top row).

In all our tests, we computed the direct lighting using the GPU: per-pixel lighting with a pixel shader and shadows using a shadow map. For shadows inside reflections, however, we traced shadow rays. The number of shadow rays is thus smaller than the number of reflection rays by one unit.

5.2. Analysis of the algorithm

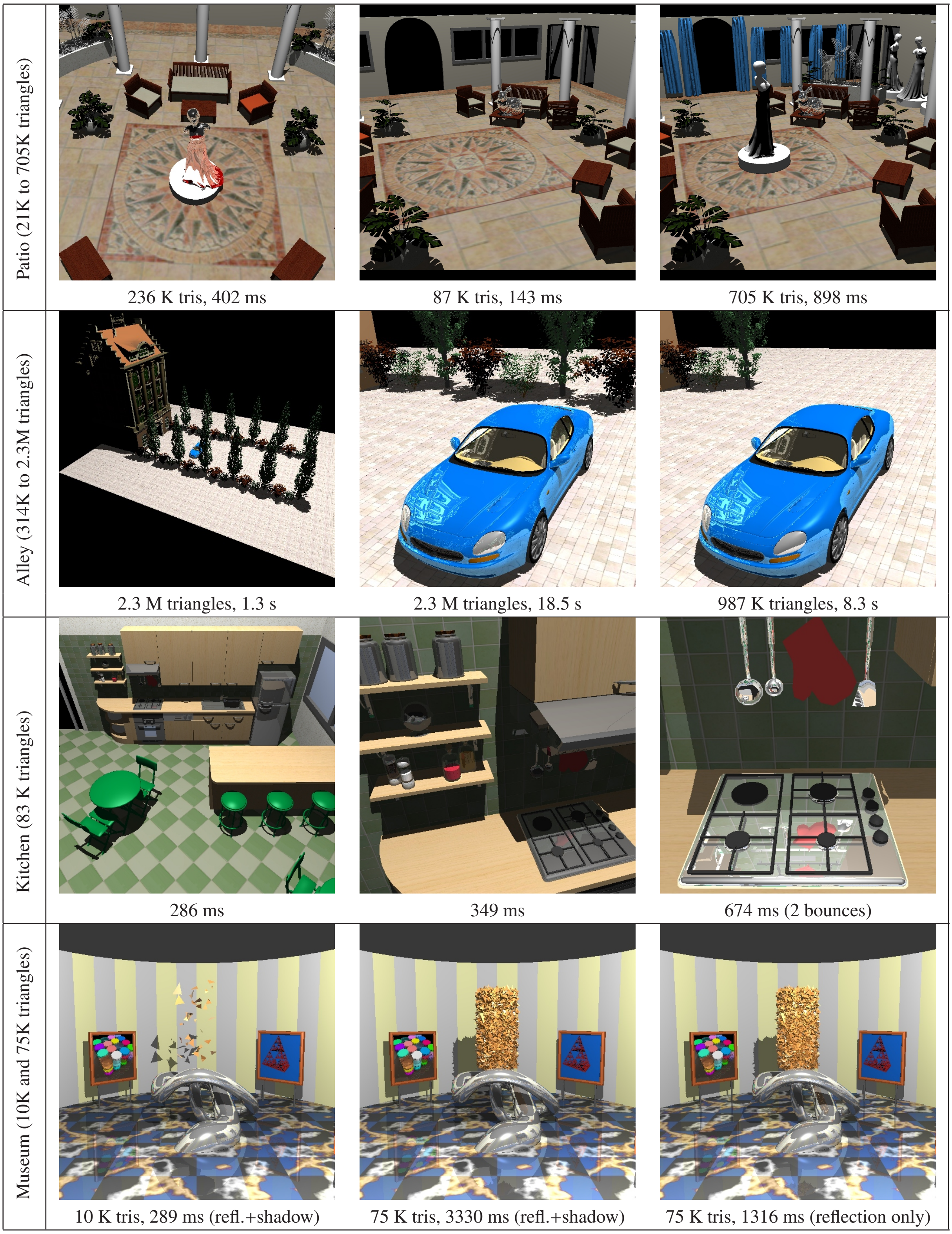
Costs for each step of the algorithm Our ray-tracer runs in four main phases: building the ray hierarchy, computing cone-spheres intersections for the hierarchy traversal, a stream reduction phase, and finally computing the actual ray-triangle intersection and shading the result.

We have found that the most important phase is the stream-reduction pass, taking roughly 60 % of computation time for each light bounce (see Figure 9(a)). The second most important step is the cone-sphere intersection, taking roughly 15 % of computation time. For the first light bounce, where there are lots of rays, the ray-triangle intersection pass takes a non-negligible part of computation time. For further light bounces, the number of rays decreases, and the time used for the ray-triangle intersection pass becomes much smaller. The cost of building the ray hierarchy itself is negligible, below 1 % (less than 2 ms for 1024×1024 resolution).

Variation with scene and reflector We are interested in the behaviour of our algorithm in the presence of varying scenes and reflectors. We placed a model of the Stanford Bunny, with 4 different levels of detail, ranging from 948 to 69000 polygons, inside the Kitchen and Patio scenes. The rendering time increases with the polygonal complexity of the Bunny (see Figure 9(b)): a more complex Bunny model means more spatial irregularities on the surface, and therefore a ray-space hierarchy with looser levels. We also placed a specular reflector with controlled irregularities (a surface of equation $\cos(nx)\cos(ny)$) inside the Kitchen scene (see Figure 11). The rendering time increases with n (see Figure 9(c)).

We have tested our algorithm on the Patio scene, changing its complexity from 21K to 705K triangles, and using three different specular reflectors: a smooth sphere, a 69K polygons Bunny, and a 25K polygons statue model (see Figure 10). We computed values for the sphere and the statue with the scene fitting in a single batch, and used a variable number of batches for the Bunny (from 1 to 3). The rendering time depends on the number of polygons in the scene, but the rate of variation is linked to the surface irregularities on the reflector. Irregular reflectors result in faster variations than smooth reflectors. The worst case corresponds to the statue, with a viewpoint where the scene is reflected in the folds at the bottom of the dress.

Judging from this data, the most important parameter in our algorithm is the shape of the specular reflectors. A smooth reflector results in a tight hierarchy, especially at the lower levels, while a reflector with many irregularities and discontinuities results in a looser hierarchy, and a rapid increase in the rendering time. Looseness at the upper levels of the hierarchy has less consequences: with a sphere, the top level of the hierarchy covers the entire space in the angular domain (see Figure 12(a)). We also tested our algorithm on a scene where all the walls are specular reflectors (see Figure 12(b)): the top level of the hierarchy covers the entire



Patio (21K to 705K triangles)

236 K tris, 402 ms

87 K tris, 143 ms

705 K tris, 898 ms

Alley (314K to 2.3M triangles)

2.3 M triangles, 1.3 s

2.3 M triangles, 18.5 s

987 K triangles, 8.3 s

Kitchen (83 K triangles)

286 ms

349 ms

674 ms (2 bounces)

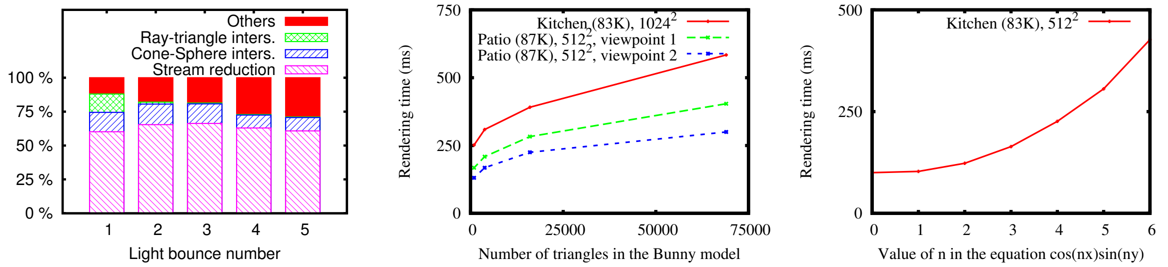
Museum (10K and 75K triangles)

10 K tris, 289 ms (refl.+shadow)

75 K tris, 3330 ms (refl.+shadow)

75 K tris, 1316 ms (reflection only)

Figure 8: *Our test scenes. All timings correspond to pictures with* 512 × 512 *pixels, and a single light bounce, with no shadow rays inside the reflection (unless otherwise specified). In the museum scene, the floor, stand and triangle soup are specular. Some of these scenes are shown in the color plate.*



(a) Relative time used by each step of our ray-tracer, for several light bounces.

(b) Several LODs for the specular Bunny.

(c) Specular surface with controlled curvature

Figure 9: *Analysis of the behaviour of our ray-tracer.*

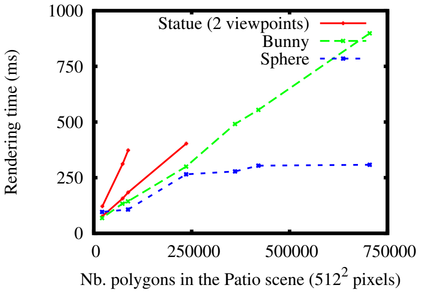
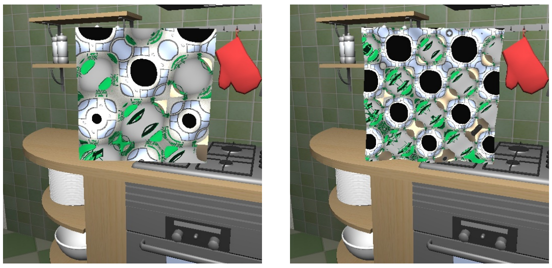


Figure 10: *Rendering times with our ray-tracer on the Patio scene, with different specular reflectors.*



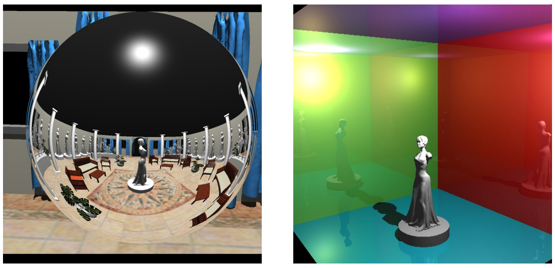
(a) $n = 2$

(b) $n = 3$

Figure 11: *Two scenes from our curvature experiment. The reflective surface has equation* $\cos(nx)\cos(ny)$ *(Kitchen scene, 83K polygons,* 512×512 *pixels).*

spatial domain. In both cases, the looseness of the hierarchy at the upper levels did not slow down the algorithm. Similarly, the large number of specular reflectors in the Kitchen scene does not hinder the algorithm (see Figure 8), even though the spatial extent of the hierarchy covers all visible specular reflectors.

Our explanation is that the top levels have a small number of nodes, so even if a node covers a large spatial and angular extent, it has small consequences on memory costs or computations. On the contrary, there is a large number of nodes at the bottom levels, so their spatial or angular extent has a strong impact on the algorithm.



(a) 705K tris, 308 ms

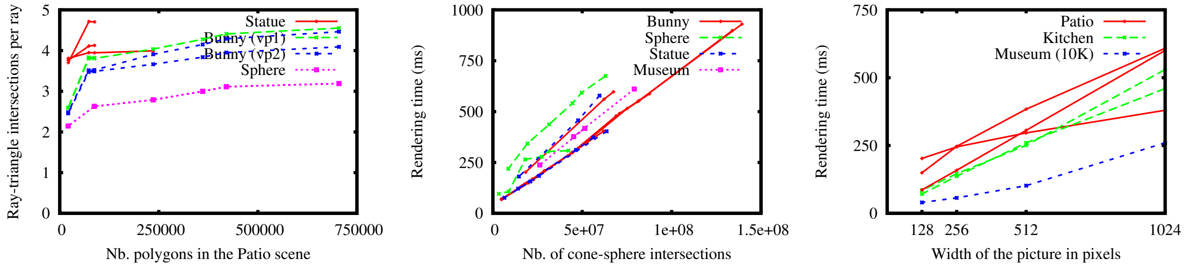
(b) 30K tris, 136 ms for one bounce, 1035 ms for 3 bounces + 2 shadow rays

Figure 12: *Our algorithm performs nicely even when the top levels have a large extent in the angular (a) or spatial (b) domain* (512×512 *pixels).*

Note: For all the curves in this section (Figures 9 and 10), we used the number of polygons in the reflected scene, *not* including the number of polygons in the specular reflector. This allows a better comparison between the different specular reflectors.

Number of ray-triangle intersections Figure 13(a) shows the average number of ray-triangle intersections for each ray in the Patio scene, for different viewpoints, picture resolutions and specular reflectors. It corresponds to the average number of triangles that are kept for each ray, at the end of the traversal of the hierarchy. Depending on scene complexity, we get values between 2.5 and 4.5, which corresponds to the number of polygons the ray intersects, thus showing that the ray-space hierarchy works properly. In several cases, the curves for picture resolutions of 512×512 and 1024×1024 are indistinguishable.

Number of Cone-sphere intersections The most important result we found is that the rendering time is closely correlated to the number of cone-sphere intersections. In Figure 13(b), we plot the former as a function of the latter, for all the tests we ran. The results are strikingly similar, for all test scenes: the number of cone-sphere intersections has a



(a) Average number of ray/triangle intersections for each ray, as a function of scene complexity.

(b) Rendering time as a function of the number of cone-sphere intersections, for all our test scenes.

(c) Rendering time as a function of the picture width for our test scenes. Rendering time is linear with the width, and thus *sub-linear* with the total number of pixels.

Figure 13: *Analysis of the behaviour of our ray-tracer.*

direct impact on the rendering time. This may seem surprising as we found that the bottleneck of the algorithm was the stream reduction pass and not the cone-sphere intersection (Figure 9(a)). But the computation time for the stream reduction pass depends on the number of non-empty elements in the stream, and is thus related to the number of cone-sphere intersections.

Picture resolution and pixels covered by the reflectors Unsurprisingly, we have found that the rendering time for our algorithm is linear with the percentage of the screen covered by the specular reflectors. More surprisingly, we have found that the rendering time is *sub-linear* with the total number of pixels in the picture (see Figure 13(c)). Both effects are a consequence of using ray hierarchies: for the first effect, the percentage of the screen covered by the specular reflector linearly affects the number of cones at each level of the hierarchy, and therefore the number of cone-sphere intersections at each level. For the second effect, doubling the width and height of the picture (thus quadrupling the number of pixels) only results in a single level added at the bottom of the hierarchy and, on average, only doubles the number of non-empty nodes in the hierarchy. The number of ray-triangle intersections is still quadrupled, but this step has a relatively small cost (see Figure 9(a)).

Large scenes We ran our algorithm on large scenes with many unstructured objects (more than 1 million polygons, see Figure 8, second row). For these scenes, the number of batches required is very high (between 20 and 140), and we do not aim at interactive rendering, but we found that our algorithm is robust enough to handle these scenes and scales well, even with many un-structured objects such as trees. See Figure 14(a) for the rendering time (in seconds) as a function of the number of polygons, for the Alley scene. We also included the Patio scene for comparison. Looking at the variation rate for both curves, we can see that the Alley scene is more difficult for our algorithm than the Patio scene, probably due to the large number of triangles intersected by each cone in the models of the trees.

5.3. Comparison with previous work

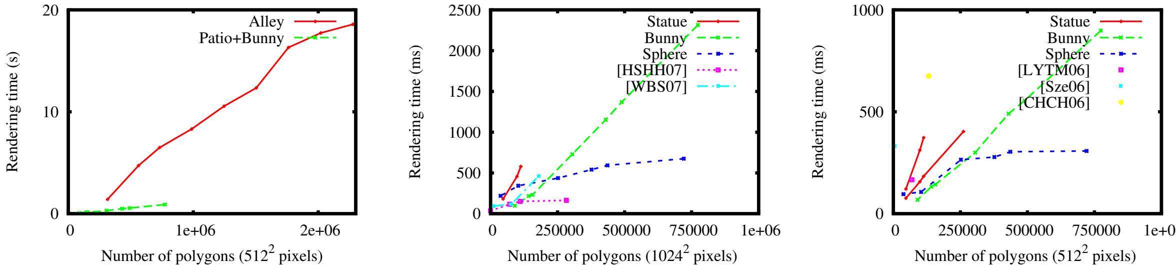
Comparing our algorithm with previous work is a difficult task, as most previous work trace primary rays in static scenes, while we trace secondary rays in dynamic scenes. Papers that report figures for animated scenes and/or secondary rays find that their algorithm is much slower than on static scenes and/or primary rays. We feel that it would not be useful to compare the rendering times for our algorithm with those for a ray-tracer computing primary rays in static scenes. The latter is bound to be faster, but the two algorithms are simply not solving the same problem.

In all our comparisons with previous work, we converted the framerates reported in the papers into milliseconds, and we used data corresponding to dynamic scenes or secondary rays (or both). We used the same picture size as the original papers.

5.3.1. CPU ray-tracing

Bounding Volume Hierarchies: Lauterbach *et al.* [LYTM06] used bounding volume hierarchies as an acceleration structure. Figure 14(c) shows a comparison of their rendering time for a dynamic scene with specular reflection (exploding Bunny) with our algorithm. Our algorithm runs approximately twice as fast, with a more complex scene being reflected in the Bunny.

Dynamic BVHs: Wald *et al.* [WBS07] used Dynamic Bounding Volume Hierarchies for ray-tracing of deformable scenes. They only report data for simple shading, without secondary rays. Figure 14(b) shows a comparison with our algorithm. Our algorithm runs at comparable speeds (slower for simple scenes, faster for more complex scenes), while computing specular reflections as well.



(a) Large unstructured scene (Patio included for comparison).

(b) Comparison with previous work [HSHH07, WBS07].

(c) Comparison with previous work [LYTM06, Szé06, CHCH06].

Figure 14: *Rendering time for our algorithm as a function of the number of polygons.*

Bounding Interval Hierarchy: Wächter and Keller [WK06] used a Bounding Interval Hierarchy as an acceleration structure. They computed specular reflections on the BART Museum scene (see Figure 8, bottom row). On the simple Museum3 scene (10412 triangles), we render the scene in 289 ms, compared to 1282 ms for [WK06]. On the complex Museum8 scene (75844 triangles) we are slower: we render the scene in 3330 ms, compared to 2040 ms in [WK06]. This slow rendering comes from the shadow rays: rendering the scene with specular reflections and no shadows in the reflection only requires 1316 ms.

5.3.2. GPU ray-tracing

Interactive k-D tree GPU raytracing: Horn *et al.* [HSHH07] traverse k-d trees using the GPU. Figure 14(b) shows a comparison of the rendering times for our algorithm with their figures for one level of specular reflection. Their algorithm is slightly faster than ours, but only handles static scenes.

The Hierarchical Ray Engine: Szécsi [Szé06] used a two-level hierarchy on ray-space for ray-tracing. Figure 14(c) shows a comparison of our algorithm with [Szé06]. We are able to handle much larger scenes, and we are approximately 4 times faster.

Geometry Images Carr *et al.* [CHCH06] constructed geometry images for fast ray-tracing of animated meshes. Figure 14(c) shows the rendering times reported in [CHCH06] for primary rays on a Bunny scene, compared with our algorithm. Using a specular Bunny as the reflector, we are approximately 5 times faster, while handling specular reflections.

5.3.3. Time to first picture

Like Wächter and Keller [WK06], we think that an important parameter for some applications is the *time to the first picture*: the time it takes until the user sees the first picture being computed (including the time for scene treatment and pre-processing). In our case, the time to the first picture is always *equal* to the rendering times reported, as our algorithm requires absolutely no pre-processing or scene structure.

6. Conclusion and future directions

In this paper, we have presented a new algorithm for fast GPU ray-tracing of secondary rays in dynamic scenes. Our algorithm uses a ray-space hierarchy, which we build on the fly for each rendering. The entire ray-tracing is done on the GPU, doing a hierarchical descent in ray-space and checking rays against the triangles in the scene. After each pass, we cull the empty sub-trees of the hierarchy using a hierarchical stream-reduction algorithm that is faster than previously published methods.

Using a ray-space hierarchy allows us to compute ray-tracing on the GPU without any conditional branches in the shaders, performing the same computations for each pixel at all steps. We stay closer to the SIMD architecture of the GPU, and we can thus exploit all its processing power. Our algorithm achieves interactive rendering on moderately complex scenes (up to ≈ 700K triangles, depending on the specular reflector), but can handle very large scenes. We also found that our algorithm scales sub-linearly with the total number of pixels in the picture, making it an interesting choice for the generation of high-definition pictures.

In future work, we would like to apply our ray-tracer to cone ray-tracing [Ama84]. Since we already work with cones, this could be an easy extension, and it would give us anti-aliasing, soft shadows and glossy BRDFs. We also want to combine our ray-space hierarchy with a scene hierarchy, traversing both hierarchies simultaneously for faster results.

References

[AK87] A J., K D.: Fast ray tracing by ray classification. *Computer Graphics (Proc. SIGGRAPH 87) 21*, 4 (1987).

[Ama84] A J.: Ray tracing with cones. *Computer Graphics (Proc. SIGGRAPH 84) 18*, 3 (1984).

[App68] A A.: Some techniques for shading machine renderings of solids. In *Spring Joint Computer Conf.* (1968), vol. 32, AFIPS.

[CF99] C A., F T.: Ray space for hierarchical ray casting. http://www.doc.ic.ac.uk/˜ajf/Research/-Papers/RaySpace/CGandA/rayspace-cga.ps.gz, 1999.

[CHCH06] C N. A., H J., C K., H J. C.: Fast GPU ray tracing of dynamic meshes using geometry images. In *Graphics Interface* (2006).

[CHH02] C N. A., H J. D., H J. C.: The Ray Engine. In *Graphics Hardware* (2002), pp. 37–46.

[Chr04] C M.: *Implementing Ray Tracing on GPU*. Diploma thesis, University of Applied Sciences, Basel, Switzerland, 2004.

[EGMM07] E M., G T., M M., M¨ S.: Automatic Creation of Object Hierarchies for Ray Tracing Dynamic Scenes. In *WSCG Short Papers Proceedings* (2007).

[EMDT06] E P., M I., D G., T D.: A GPU-driven algorithm for accurate interactive reflections on curved objects. In *Rendering Techniques 2006 (Proc. EG Symposium on Rendering)* (2006).

[EVG04] E M., V C., G G.: Stack implementation on programmable graphics hardware. In *Vision, Modeling, and Visualization 2004* (2004).

[FS05] F T., S J.: KD-tree acceleration structures for a GPU raytracer. In *Graphics Hardware* (2005).

[GH98] G D., H J.-M.: A beam tracing with precise antialiasing for polyhedral scenes. *Computer Graphics 22*, 1 (1998), 103–115.

[HH84] H P. S., H P.: Beam tracing polygonal objects. *Computer Graphics (Proc. SIGGRAPH 84) 18*, 3 (1984).

[Hor05] H D.: Stream reduction operations for GPGPU applications. In *GPU Gems 2*, Pharr M., (Ed.). Addison Wesley, 2005, pp. 573–589.

[HSHH07] H D., S J., H M., H P.: Interactive k-d tree gpu raytracing. In *Symposium on Interactive 3D Graphics and Games* (2007). (to appear).

[IRWP06] I T., R C., W I., P S. G.: An evaluation of parallel grid construction for ray tracing dynamic scenes. In *IEEE Symposium on Interactive Ray Tracing* (2006).

[KL04] K F., L C. J.: *Ray Tracing on Programmable Graphics Hardware*. Master's thesis, Chalmers University of Technology, Göteborg, Sweden, 2004.

[LAAM01] L J., A U., A -M¨ T.: A benchmark for animated ray tracing. *IEEE Computer Graphics and Applications 21*, 2 (2001), 22–31.

[LYTM06] L C., Y S.-E., T D., M D.: RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *IEEE Symposium on Interactive Ray Tracing* (2006).

[NO97] N K., O Y.: Breadth-first ray tracing utilizing uniform spatial subdivision. *IEEE Transactions on Visualization and Computer Graphics 3*, 4 (1997).

[NO02] N K., O Y.: Enhanced breadth-first ray tracing. *Journal of Graphics Tools 6*, 4 (2002).

[PBMH02] P T. J., B I., M W. R., H - P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics (Proc. SIGGRAPH 2002) 21*, 3 (2002), 703–712.

[RH06] R D., H N.: Accurate specular reflections in real-time. *Computer Graphics Forum (Proc. Eurographics 2006) 25*, 3 (2006).

[RSH05] R A., S A., H J.: Multi-level ray tracing algorithm. *ACM Transactions on Graphics (Proc. SIGGRAPH 2005) 24*, 3 (2005), 1176–1185.

[SKALP05] S -K L., A ´ B., L ´ I., P - M.: Approximate ray-tracing on the GPU with distance impostors. *Computer Graphics Forum (Proc. Eurographics 2005) 24*, 3 (2005).

[SLO06] S S., L A. E., O J. D.: A work-efficient step-efficient prefix sum algorithm. In *Workshop on Edge Computing Using New Commodity Architectures* (May 2006), pp. D–26–27.

[Szé06] S ´ L.: The Hierarchical Ray Engine. In *WSCG* (2006).

[TS05] T N., S L. O.: *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing*. Master's thesis, University of Aarhus, Denmark, 2005.

[WBS07] W I., B S., S P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1 (2007), 6.

[Whi80] W T.: An improved illumination model for shaded display. *Communications of the ACM 23*, 6 (June 1980), 343–349.

[WIK*06] W I., I T., K A., K A., P S. G.: Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics (Proc. SIGGRAPH 2006) 25*, 3 (2006).

[WK06] W¨ C., K A.: Instant ray tracing: The bounding interval hierarchy. In *Rendering Techniques 2006 (Proc. EG Symposium on Rendering)* (2006), pp. 161–166.

[WS01] W I., S P.: State-of-the-art in interactive raytracing. In *State of the Art Reports, Eurographics* (2001).