# GPU Rendering of Secondary Effects

Kai Bürger, Stefan Hertel, Jens Krüger and Rüdiger Westermann

Technische Universität München
Email: {buergerk,hertel,jens.krueger,westermann}@in.tum.de

## Abstract

In this paper we present an efficient data structure and algorithms for GPU ray tracing of secondary effects like reflections, refractions and shadows. Our method extends previous work on layered depth cubes in that it uses layered depth cubes as an adaptive space partitioning scheme for ray tracing. We propose a new method to efficiently build LDCs on the GPU using geometry shaders available in Direct3D 10. The overhead of peeling the scene multiple times can thus be avoided. We further show that the traversal of secondary rays is greatly accelerated by exploiting a two level hierarchy and the adaptive nature of the LDC. Due to the computational and bandwidth capacities available on recent GPUs our method enables high-quality rendering of static *and* dynamic scenes at interactive rates.

## 1 Introduction and Related Work

Since the early years of computer graphics there has been interest in ray tracing due to its potential for the accurate rendering of complex light phenomena. Over the last few years, there was an ever growing interest due to the observation that interactive ray tracing can now be achieved on custom hardware [8, 23, 22], or by using a cluster of custom computers [19, 26]. Recent advances in hardware and software technology, including specialized ray tracing chips [25] as well as advanced space partitioning and traversal schemes [27, 30], have even shown that ray tracing is potentially suited for real time applications like computer games and virtual environments.

Simultaneously, considerable effort has been put into the implementation of ray tracing on programmable graphics hardware. Inspired by the early work of Purcell et al. [21] and Carr et al. [2], in a number of succeeding implementations it was shown that the capabilities of recent GPU stream architectures including parallelism across stream elements and low-latency memory interfaces can effectively be used for ray tracing [17]. While these approaches were solely based on uniform space partitioning schemes, recent work has also demonstrated the possibility to build and traverse adaptive spatial hierarchies on the GPU [15]. Hereafter, Foley and Sugerman [5] as well as Popov et al. [20] independently examined stack operations—a feature not well-supported by the GPU—and they reported a significant performance gain by using a stackless traversal algorithm for kd-trees. Alternatively, Carr et al. [3] represented surfaces as geometry images and introduced linked bounding volume hierarchies to avoid conditionals and stack operations. By taking advantage of the GPU to construct these hierarchies, for the first time the authors could demonstrate real-time GPU ray tracing of dynamic scenes.

Despite all the advancements in GPU ray tracing, including efficient approximations for ray-object intersection using pre-computed environment imposters [14] and ray-object penetration depths [31], it can still not be denied that high quality ray tracing using optimized CPU codes performs favorable or even faster than many GPU implementations. The main reason why rasterization hardware is not perfectly suited for ray tracing is the inability of current GPUs to efficiently determine ray-object intersections for rays others than view rays. This makes it difficult to accurately simulate secondary effects like reflections, refractions, and shadows, as such effects require parts of the scene to be rendered multiple times under different projections, i.e. onto different receivers. Although possible in principle, it was shown by Wand and Straßer [28] for the rendering of underwater caustics that this approach is not practicable in general.

On the other hand it can be observed that a significant part of the render time in typical 3D applications is shading, and it is well accepted that the GPU outperforms the CPU in this respect. Appar-


Figure 1: Method demonstration: Cubemap reflections (left), our method (middle) and software ray tracing (right). On a single Geforce 8800 GTX our method renders the scene into a 1280x1024 image at 3 fps.

ently, due to the GPUs inability to effectively exploit adaptive space partitioning schemes this advantage is entirely amortized in ray tracing. Our motivation is thus to overcome GPU limitations in finding ray-object intersections, at the same time exploiting the intrinsic strength of these architectures to shade billions of fragments in real-time.

### 1.1 Contribution

The main contribution of this paper is a new GPU approach for ray tracing of secondary effects like reflections, refractions and shadows. This is achieved by using an adaptive spatial data structure at extreme resolution, and by providing novel methods to construct and traverse this data structure on the GPU. Just as proposed by Lischinski and Rappoport [13], our data structure represents the scene as a set of layered depth images (LDI) [24, 16, 6] along three orthogonal projections. According to Lischinski and Rappoport [13] we will refer to this structure as the layered depth cube (LDC). We extend LDCs in the following ways:

- LDIs are constructed via depth peeling [4], but we employ Direct3D 10 functionality so that the scene only has to be peeled once to generate LDIs along multiple viewing directions. To accelerate the LDC construction every polygon is rendered only into the LDI capturing the scene from the direction most perpendicular to this polygon (see Figure 5). To accommodate deferred shading the LDC stores not only fragment depth, but also interpolated colors and texture values as well as normals. The process leads to a view independent scene representation using a minimal number of samples.
- For each LDI a two level hierarchy is built. This method is similar to the empty space skipping structure used by Krüger and Westermann [11]. Entries in this low resolution representation of one LDC direction store for each bundle the minimum and maximum distances from a reference plane perpendicular to the direction of projection.
- We present an efficient ray-object intersection test using LDCs. As in each LDI the samples lie on a 2D raster we perform ray traversal in these rasters alternatively. This method is similar to screen-space ray tracing described by Krüger et al. [10], but it has a major advantage: The hierarchical representation is used to skip regions in these rasters not containing any structures a ray could intersect with.

As both the LDC construction and the traversal are performed on the GPU the rasterization capacities of recent architectures can effectively be exploited. In particular, since our approach does not require any pre-process to modify the initial scene representation it can be used in the same way to render dynamic scenes or scenes created or modified on the GPU. Some results of our approach together with a comparison to cubemap reflections and software ray tracing are shown in Figures 1 and 2.

The remainder of this paper is organized as follows: In the next chapter we will discuss the LDC construction in particular the use of Direct3D 10 features to speed up this process. We will then describe how this structure can be efficiently traversed on the GPU. Next, we explain the integration of the LDC construction and the ray traversal into the rendering algorithm. Finally, we analyze the perfor-

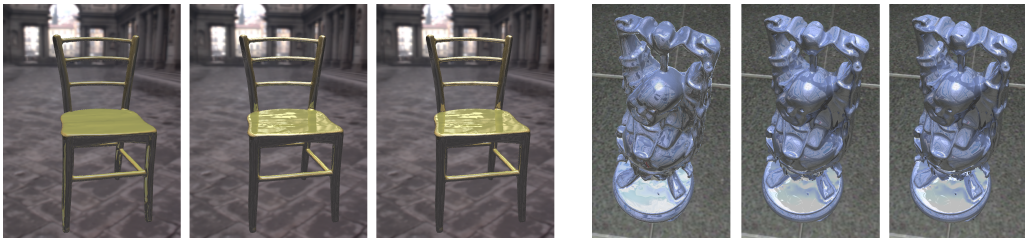

Figure 2: Method demonstration: Cubemap reflections (left), our method (middle) and software ray tracing (right). On a single Geforce 8800 GTX our method renders the scene into a 1280x1024 image at 5 and 3 fps, respectively.

mance of the major components of our system, and we conclude the paper with some remarks about future research in this field.

## 1.2 LDC Construction

To efficiently construct LDCs on the GPU we first employ depth peeling [4] to generate LDIs along three orthogonal viewing directions. Depth-peeling requires multiple rendering passes. For each pixel, in the $n$-th pass the $(n-1)$-th nearest fragments are rejected in a fragment program and the closest of all remaining fragments is retained by the standard depth test. A floating point texture map—the depth map—is used to communicate the depth of the surviving fragments to the next pass. The number of rendering passes is equal to the objects depth complexity, i.e. the maximum number of object points falling into a single pixel. This number is determined by rendering the objects once and by counting at each pixel the number of fragments falling into it during rasterization. The maximum over all pixels is then collected in a log-step reduce-max operation [12].

Although more efficient depth peeling variants exist, for instance the method proposed by Wexler et al. [29] showing linear complexity in the number of polygons compared to quadratic complexity of standard depth peeling, in the current work we favor the more complex approach. This is because it does not require any pre-processing and thus can be used for the processing of dynamic scenes and scenes modified or generated on the GPU.

### 1.2.1 Construction using Geometry Shader

In this chapter we describe how to efficiently generate an LDC that captures the entire scene. From the text it should become clear that the same approach can of course be used to generate LDCs for separate objects in the scene. In particular for rigid objects this allows us to pre-compute the LDC once and to exclude it from depth peeling in successive frames. LDC construction greatly benefits from latest graphics APIs and hardware as explained in the following paragraphs.

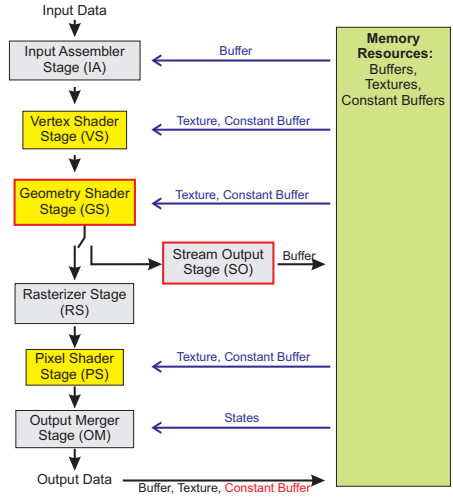


Figure 3: The Direct3D 10 rendering pipeline. Programmable stages are drawn in yellow. Note in particular the new programmable Geometry Shader stage after the Vertex Shader.

One of the key novelties of Direct3D 10 capable hardware [1] is a new programmable stage in the rendering pipeline. This stage—the Geometry

Shader—is placed directly after the Vertex Shader stage (see Figure 3). In contrast to the Vertex Shader the Geometry Shader takes as input an entire graphics primitive (e.g. a triangle or a triangle and its neighbors) and outputs zero to multiple new primitives. This is achieved by letting the Geometry Shader append the new primitives to one or multiple output streams. The LDC construction algorithm makes use of this feature and binds multiple output streams called render targets (MRTs) to the Geometry Shader. Note that these MRTs are different from the ones known in the Pixel Shader stage.

While MRTs in the Pixel Shader are used to output multiple color values at the very end of the pipeline, to each Geometry Shader MRT its own rasterizer, Pixel Shader stack, depth and color buffers are associated. One can even assign another stack of Pixel Shader MRTs to each of these separate pipelines (see Figure 4).

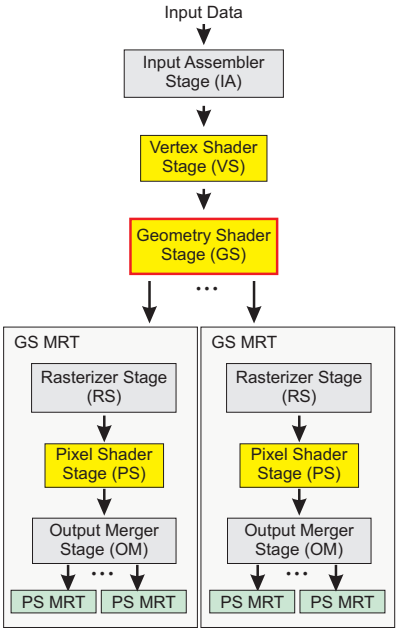


Figure 4: This figure depicts the difference between MRTs used in the Geometry Shader and in the Pixel Shader stage.

Without the functionality provided by the Geometry Shader LDC construction requires the entire scene to be rendered three times. By using the Geometry Shader MRTs we only need to process and rasterize the scene once. This is achieved by letting the Vertex Shader perform the geometry transformation excluding the viewing transformation. Transformed vertices are then combined to triangles and sent to the Geometry Shader stage. Here, face normals are computed for every triangle and compared against the three directions along which peeling is performed. Triangles are then rasterized into the MRT which captures those parts of the scene most perpendicular to its projection direction (see Figure 5). In this way every triangle has to be processed and rasterized only once. This method not only reduces GPU load to one third, but it also reduces the depth complexity along all three directions since lesser primitives appear in either target.

Furthermore, as no primitive is rasterized into more than one MRT the process avoids storage of redundant samples in different projection directions thus leading to an optimal view independent scene representation. As mentioned earlier, each Geometry Shader MRT pipeline can have multiple Pixel Shader MRTs at its very end (see Figure 4), we use this additional possibility to capture multiple values at once. In particular we use two Pixel Shader MRTs, one target to store the normal and depth information of the fragments and a second target for texture and color information. In the final rendering step we use the later values to perform deferred shading of the reflected and refracted surfaces.

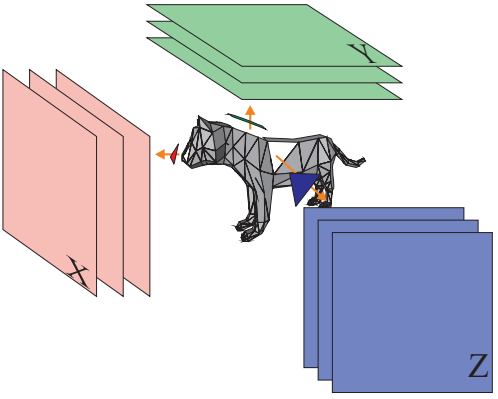


Figure 5: This figure shows how triangles are projected into only one LDI stack depending on their face normal orientation.

By using the Geometry Shader MRTs the complexity of our algorithm to construct the LDC is reduced to $\max(\text{depthComplexity})$ passes. How-

ever, with the advent of layered depth/color buffers at the end of the rendering pipeline we expect our construction procedure to become even more efficient. For a long time such a functionality is present on GPUs (e.g. ATI/AMDs FBuffer technology [9]) though it has not yet been exposed. On the other hand we perceive an ever growing demand for depth peeling in a number of applications ranging from order independent transparency [4] and CSG rendering [7] to volume rendering applications [18] and real-time rendering [10]. As a consequence we expect graphics APIs to support these features in the foreseeable future. Then, the generation of LDCs could even be performed in one single rendering pass.

### 1.3 Two Level LDC

An LDC exhibits an extremely adaptive and compact representation of the scene, yet it lacks the hierarchical nature of other space partitioning schemes such as octrees or kd-trees. It is on the other hand well known that such schemes can greatly accelerate ray tracing in that they allow for an efficient determination of ray-object intersections.

To generate a two-level LDC we first compute the LDC as described above. Then a single low resolution data structure is generated for every LDI. This structure is built only on the depth values whereas the the normal and texture stacks remain unchanged. For the generation of an $n \times m$ reduced empty space skipping texture we employ a custom pixel shader, that performs a max/min computation on a grid of $n \times m$ fragments from all LDI layers in one direction (see Figure 6). Such a texture can be seen as an axis aligned bounding box, which will be used later on to efficiently skip empty space in the scene.

Equipped with the acceleration texture as described we will now show how to exploit this data structure for GPU ray tracing.

### 1.4 Ray Traversal

To traverse rays through a two level LDI we employ an approach similar to the one proposed by Krüger and Westermann [11]. We perform a modified DDA algorithm to find the coarse level grid cells intersected by the ray. Therefore the ray is tested against the depth range stored in the cells of the acceleration texture being hit. If an intersection is found we

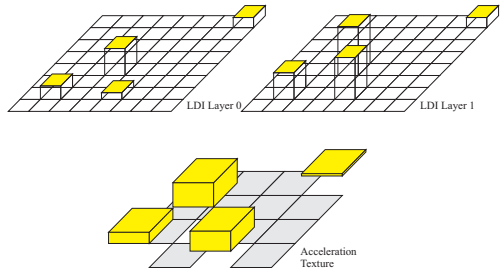


Figure 6: The image shows how two LDI layers (top) are combined into a single acceleration texture (bottom). This texture effectively stores bounding boxes around all LDI layers for a given $n \times m$ grid in this case $2 \times 2$.

step down to the LDIs and use the same DDA algorithm within the $n \times m$ block. If an intersection with one of the values stored in the LDIs is found we terminate the ray traversal, otherwise we continue at the next block in the coarser empty space skipping texture.

In the way described we subsequently process one peel direction after another. Note that if an intersection in one direction is found this intersection does not necessarily has to be the first intersection along the ray. However, in this case we can change the end point of the ray to the position of that intersection, continually reducing the remaining distance to be considered along the ray. After all layers have been traversed the intersection point between the ray and an object in the scene is found, or just the intersection with the LDCs bounding box. This information is used later on for shading the reflected sample point.

## 2 Rendering

After having described both the two level data structure used to represent a scene on the GPU and an efficient ray traversal scheme for this data structure we will now present the rendering algorithm that finally generates an image of the scene. This algorithm renders the scene in three passes. In the first pass—which itself consists of multiple subpasses—the LDC capturing the scene is generated. In the second pass “primary ray-object intersections” are determined by rasterizing the scene under the current viewing transformation. In the third

pass the secondary rays are traced in the LDC to simulate reflections, refractions and shadows.

While passes one and two are clear, pass three needs some further explanations.

### 2.1 Ray Tracing and Shading

After the second pass the scene as seen from the current view point, appropriately shaded but without any secondary effects, has been generated. In this image we need to find for every specular receiver the points in the scene from where to receive an additional radiance contribution. Therefore we proceed in two steps. In the first step we render reflective/refractive objects by employing a pixel shader program that computes for every fragment the following values:

- the reflection vector
- the intersection of this vector with the LDC
- the parallel projection of the reflection vector into the three orthogonal LDIs

The later two quantities are stored in three render targets, of which we use the first to store the intersection point and the remaining two to store projected directions (see Figure 7). In the second step we use these values to perform the LDC traversal as described in the previous section. This results in either an intersection point with an object in the scene or with the scene's bounding box. In the latter case we simply perform a lookup into an environment map to compute the color of the reflection. In the former case we lookup the normal and the color generated for the intersected point in the LDC construction, and we use this information to shade the reflected point. Finally, the color seen along the primary rays is combined with the color of the secondary ray and rendered into the color buffer.

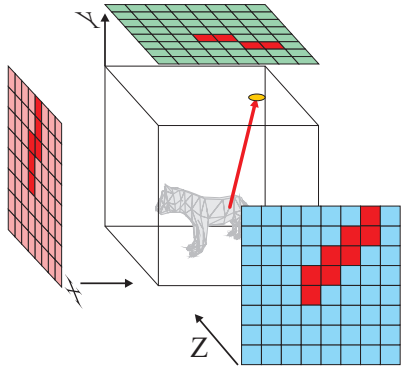


Figure 7: This image illustrates the four values generated prior to the ray traversal. Firstly, the intersection point between the reflected ray and the bounding box yellow) is computed. Next, the ray is projected into the three orthogonal LDIs (red, green, blue).

## 3 Results

We have tested the proposed GPU ray tracing technique in a number of different scenarios consisting of several thousands up to hundreds of thousands triangles. Images of such scenes together with ground truth images generated by a software ray tracer are given in Figures 1 and 2. At the end of this chapter some additional examples including shadowing and refractions are shown. In contrast to the first four examples, in the fifth example a dynamic object is rendered using vertex shader skinning. In this case the LDC is constructed on-the-fly in every frame of the animation (see Figure 10).

All of our tests were run on a single core Pentium 4 equipped with a NVIDIA Geforce 8800 GTX graphics card with 768 MB local video memory. In all of our tests $1K \times 1K$ LDIs were used to sample the objects along three orthogonal viewing directions. Note that this corresponds to a resolution of the spatial data structures of $1K^3$. However, as we store 16 Bit floating point depth values in every LDI this resolution is in fact significantly higher. As we only capture those areas in space where some objects are present, our approach requires considerably less memory than other techniques using a uniform grid structure to represent the scene.

All objects are encoded as indexed vertex arrays stored in GPU memory. In all our experiments an intersection was determined if the distance between a point on the secondary ray and a fragment coded in the LDC was less than $0.001$ in world space. This tolerance was also used in all other examples throughout this paper. It can be seen in all our examples that the scene is adequately sampled by the LDC and GPU ray tracing produces almost exactly the same results as the software ray tracer running in double floating point precision on the CPU.

Representative timings in milliseconds (ms) for GPU ray casting of secondary effects in the four example scenes are listed in Table 1. The number of

LDIs required to capture the scenes adequately are 8, 12, 7, 6 and 6 respectively. The values in columns labeled (A) show the amount of time spent by the GPU for LDC construction. Columns labeled (B) show the time spent for ray tracing including rendering of the final result. Performance was measured using LDIs at $512 \times 512$ and $1K \times 1K$ resolution. All tests were rendered into a $1280 \times 1024$ viewport (see Figures 8 to 10) .


Figure 8: Two reflective teapots above a mirroring plane and the EG07 Phlegmatic Dragon with a reflective surface. Note the reflection of the dragons face on its nose.

As the timings show, by means of the proposed technique secondary effects in very complex scenes can be simulated at interactive rates and convincing quality. In particular it can be observed that the

| | LDI resolution | | | |
|---|---|---|---|---|
| | 512 x 512 | | 1024 x 1024 | |
| | A | B | A | B |
| Bunny (8192 tris) | 7 | 61 | 19 | 115 |
| Car (20264 tris) | 9 | 82 | 25 | 250 |
| Dragon (120K tris) | 12 | 205 | 31 | 441 |
| Max Planck (300K tris) | 19 | 160 | 53 | 346 |
| Tiny (animation) (1628 tris) | 5 | 44 | 16 | 95 |

Table 1: Timing statistics (in ms) for different scenes.

LDC construction is fast enough to allow for on-the-fly capturing of reasonable scenes. This property makes it possible to render dynamic objects at high frame rates and quality.

## 4 Conclusions and Future Work

In this work we have described a technique for GPU ray tracing of secondary effects. By using a view-independent, two level LDI representation in combination with an adaptive ray traversal scheme on the GPU interactive rendering of such effects is possible. We have shown how to construct LDCs efficiently on the GPU by exploiting recent functionality on Direct3D 10 capable graphics hardware together with the intrinsic strength of these architectures to shade millions of points in real-time. As our timings indicate, the proposed techniques enable interactive ray tracing of complex scenes at high accuracy. In comparison to software ray tracing visual artifacts are marginal.

Due to the efficiency of the LDC construction it is in particular possible to integrate this process into the rendering pass. This enables the rendering of dynamic objects without any additional modifications of the proposed algorithm. As the construction process only requires objects to be available in any renderable format our method can also deal with objects being modified or constructed on the GPU.

In the future, we will investigate how to further improve the performance of the proposed rendering technique. In particular we will focus on the problem how to effectively exploit the internal RGBA pipeline on current GPUs. In principle this can be done in two ways: Firstly, by storing four LDIs into one RGBA texture target and thus by providing the

possibility to trace every ray in four depth layers simultaneously. Secondly, by tracing four rays simultaneously in one fragment program. As both optimizations are greatly influenced by the ability of recent GPUs to effectively exit shader programs, a detailed analysis of the performance gains for different scenes and architectures is required. Furthermore, we plan to compare our two-level acceleration structure to fully octree or kd-tree hierarchies.

## 5 Acknowledgments

The authors wish to thank the AIM@SHAPE project and all of its partners for providing most of the meshes used in this paper as well the Academy of Sciences of the Czech Republic, and CGG, Czech Technical University in Prague for providing the Phlegmatic Dragon data set.

## References

[1] David Blythe. The direct3d 10 system. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 724–734, New York, NY, USA, 2006. ACM Press.

[2] N. Carr, J. Hall, and J. Hart. The ray engine. In *Proceedings of the ACM SIGGRAPH /EUROGRAPHICS conference on Graphics hardware*, pages 37–46, 2002.

[3] Nathan A. Carr, Jared Hoberock, Keenan Crane, and John C. Hart. Fast gpu ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of the 2006 conference on Graphics interface*, pages 203–209, 2006.

[4] Cass Everitt. Interactive order-independent transparency. Technical report, NVIDIA White papers, 2001.

[5] 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, 2005.

[6] Steven J. Gortler, Li-Wei He, and Michael F. Cohen. Rendering layered depth images. Technical Report Technical Report MSTR-TR-97-09, Microsoft Research, Redmond, WA, 1997.

[7] John Hable and Jarek Rossignac. Blister: Gpu-based rendering of boolean combinations of free-form triangulated shapes. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1024–1031, 2005.

[8] Daniel Hall. The ar350: Today's ray trace rendering processor. SIGGRAPH / Eurographics Workshop On Graphics Hardware - Hot 3D Session, 2001.

[9] Mike Houston, Arcot Preetham, and Mark Segal. A hardware f-buffer implementatio. Technical report, Stanford University Computer Science Technical Reports, 2006.

[10] Jens Krüger, Kai Bürger, and Rüdiger Westermann. Interactive screen-space accurate photon tracing on GPUs. In *Rendering Techniques (Eurographics Symposium on Rendering - EGSR)*, pages 319–329, June 2006.

[11] Jens Krüger and Rüdiger Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003*, 2003.

[12] Jens Krüger and Rüdiger Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3), 2003.

[13] Dani Lischinski and Ari Rappoport. Image-based rendering for non-diffuse synthetic scenes. In *Proceedings, Ninth Eurographics Workshop on Rendering*, pages 301–314, 1998.

[14] Szirmay-Kalos Lszl, Aszdi Barnabs, Laznyi Istvn, and Premecz Mtys. Approximate ray-tracing on the GPU with distance impostors. *Computer Graphics Forum*, 24(3), 2005.

[15] Günther Greiner Manfred Ernst, Christian Vogelgsang. Stack implementation on programmable graphics hardware. In *Vision Modeling and Visualization*, pages 255–262, 2004.

[16] Nelson Max. Hierarchical rendering of trees from precomputed multi-layer z-buffers. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 165–174, London, UK, 1996. Springer-Verlag.

[17] Gabo Moreno-Fortuny and Michael McCool. Unified stream processing raytracer. Poster at GP2: The ACM Workshop on General Purpose Computing on Graphics Processors, 2004.

[18] Zoltan Nagy and Reinhard Klein. Depth-peeling for texture-based volume rendering. In *PG '03: Proceedings of the 11th Pacific Conference on Computer Graphics and Applica-*

*tions*, 2003.

[19] W. Martin Parker, P.-P. Sloan, P. Shirley, B. Smits, and C. Hansen. Interactive ray tracing. In *Symposium on Interactive 3D Computer Graphics*, pages 119–126, 1999.

[20] Stefan Popov, Johannes Günther, and Philipp Slusallek. Stackless kd-tree traversal for high performance gpu ray tracing. In *Proceedings of EUROGRAPHICS annual conference*, 2007.

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

[22] J. Schmittler, S. Woop, D. Wagner, W.J. Paul, and P. Slusallek. Realtime ray tracing of dynamic scenes on an fpga chip. In *Graphics Hardware 2004*. Eurographics Association, 2004.

[23] Jörg Schmittler, Ingo Wald, and Philipp Slusallek. SaarCOR: a hardware architecture for ray tracing. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 27–36, 2002.

[24] Jonathan Shade, Steven Gortler, Li wei He, and Richard Szeliski. Layered depth images. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 231–242, 1998.

[25] Jörg Schmittler Sven Woop and Philipp Slusallek. Rpu: A programmable ray processing unit for realtime ray tracing. In *Proceedings of ACM SIGGRAPH 2005*, July 2005.

[26] Ingo Wald, Carsten Benthin, and Philipp Slusallek. Distributed interactive ray tracing of dynamic scenes. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, 2003.

[27] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, pages 485–493, 2006. (Proceedings of ACM SIGGRAPH 2006).

[28] Michael Wand and Wolfgang Straßer. Real-time caustics. In P. Brunet and D. Fellner, editors, *Computer Graphics Forum*, volume 22(3), 2003.

[29] Daniel Wexler, Larry Gritz, Eric Enderton, and Jonathan Rice. Gpu-accelerated high-quality hidden surface removal. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 7–14, New York, NY, USA, 2005. ACM Press.

[30] Sven Woop, Gerd Marmitt, and Philipp Slusallek. B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes. In *Proceedings of Graphics Hardware*, 2006.

[31] Chris Wyman. Interactive image-space refraction of nearby geometry. In *Proceedings of GRAPHITE*, pages 205–211, 2005.

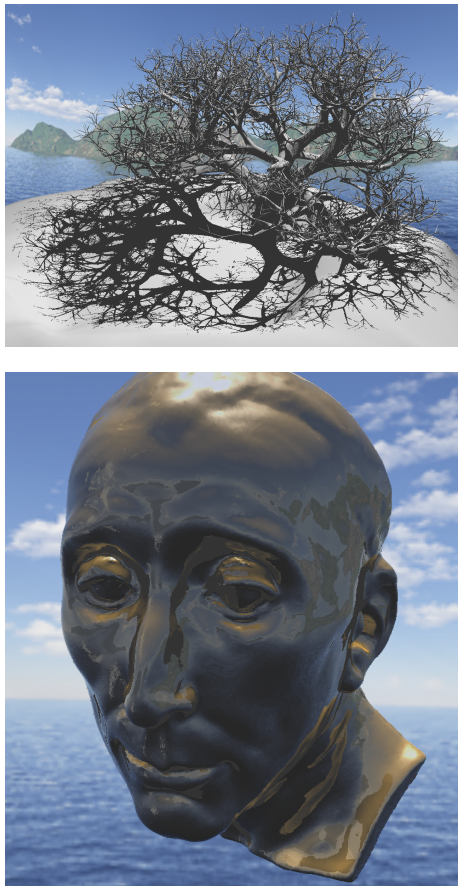

Figure 9: These Images depict from top to bottom. Shadows of a complex tree computed by our approach, and a reflective bust of Niccolo da Uzzano.

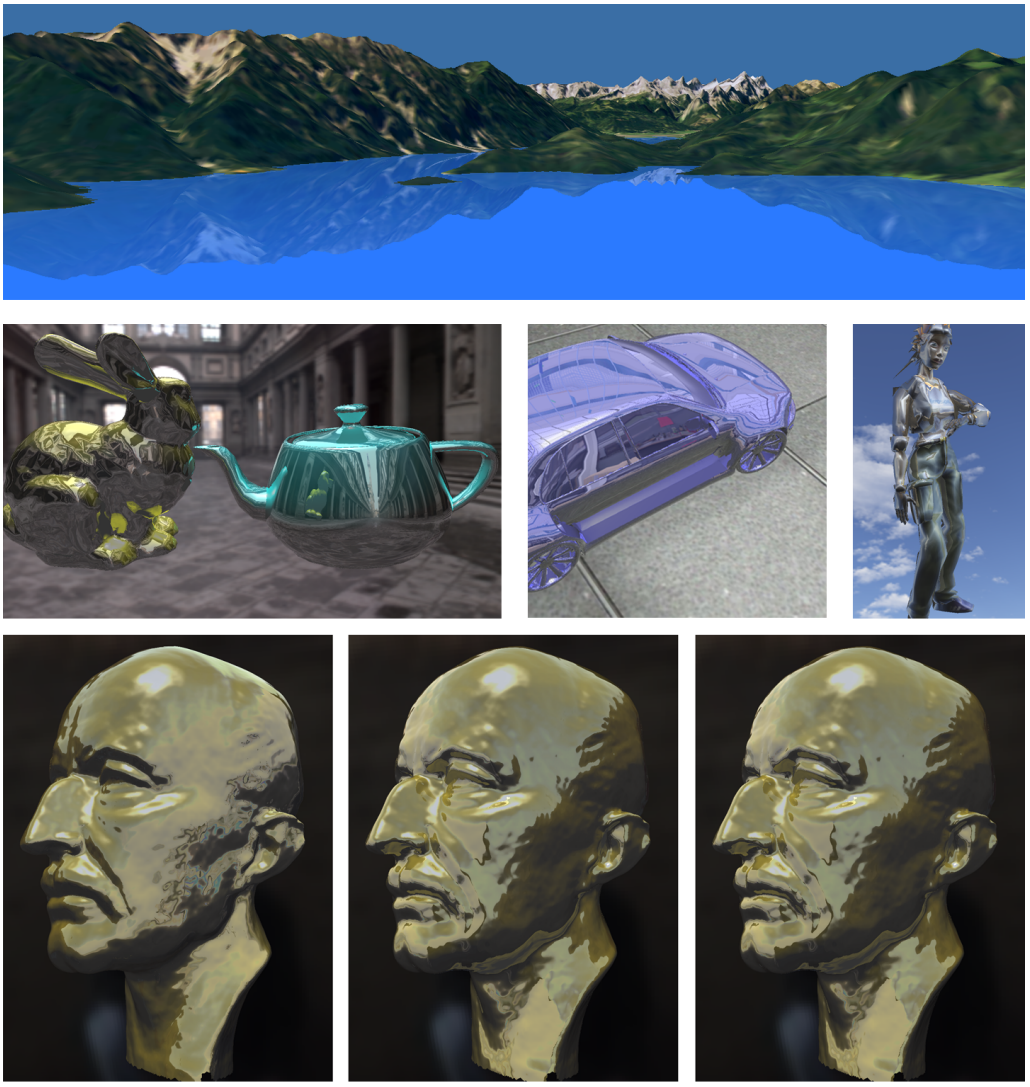

Figure 10: Various renderings using the proposed GPU ray casting are shown. Note that the rightmost figure in the middle row is one snapshot out of a GPU animation using vertex-skinning. In the last row we show cubemap reflections (left) and reflections rendered with our method (middle) and with a software ray tracer (right).