Rethinking Graphics and Gaming Courses Because of Fast Ray Tracing

Peter Shirley U of Utah
Kelvin Sung U of Wash., Bothell
Erik Brunvand U of Utah
Alan Davis U of Utah
Steven Parker U of Utah
Solomon Boulos U of Utah

Figure 1: Ray tracing can robustly and naturally support next generation visual effects not easily combined with GPU graphics including depth-of-field, motion blur, glossy and specular reflection, soft shadows, and correct refraction. More details on the system that generated of these images are available in Boulos et al. (2006).

Abstract

Almost all current games are implemented using the graphics processing units (GPUs) found on almost every PC. These GPUs use the z-buffer algorithm to do visibility calculations. Ray tracing, an alternative to the z-buffer algorithm, delivers higher visual quality than the z-buffer algorithm but has historically been too slow for interactive use. However, ray tracing has benefitted from improvements in computer hardware, and many believe it will replace the z-buffer algorithm as the visibility engine in games. If that replacement happens, it will imply fundamental changes in both the API to and capabilities of 3D graphics engines. This paper discusses the implications for games and graphics oriented classes should this switch to ray tracing occur.

1 Introduction

At present almost every personal computer has a dedicated processor that enables interactive 3D graphics. These graphics processing units (GPUs) implement the *z-buffer* algorithm introduced in Catmull's landmark University of Utah dissertation [Catmull 1974]. These GPUs can interactively display several million triangles with texture and lighting. The wide availability of GPUs has revolutionized how work is done in many disciplines, and has enabled the hugely successful video game industry. While the hardware implementation of the z-buffer algorithm has allowed excellent interactivity at a low cost, significant improvements in visual quality will require Whitted's ray tracing algorithm [Whitted 1980] or Cook's distribution ray tracing algorithm [Cook et al. 1984] which allow many advanced visual effects (Figure 1). The ray tracing algorithm is better suited to huge datasets than the z-buffer algorithm because it creates an image in time sublinear in the number of objects while the z-buffer is linear in the number of objects. It is ray tracing's larger time constant and lack of a commodity hardware implementation that makes the z-buffer a faster choice for data sets that are not huge. Ray tracing is better suited for creating shadows, reflections, and refractions because it directly simulates the physics of light. Ray tracing enables these forms of isolated visibility queries that are problematic (or impossible) for the z-buffer algorithm. Ray tracing also allows flexibility in the intersection computation for the primitive objects that allows non-polygonal primitives such as splines or curves to be represented directly. Unfortunately, computing these visual effects based on simulating light rays is computationally expensive, especially on a general purpose CPU. The ray tracing algorithm currently requires tens to hundreds of CPUs to be interactive at full-screen resolution.

Games have driven almost all desktop 3D graphics, and we believe that trend will continue. Because ray tracing is well-suited to support a qunatum leap in the ability of games to support higher-order surfaces, complex models, and high quality lighting, we believe games will migrate to using it once ray tracing is fast enough. In this paper we argue that it is feasible to make ray tracing fast enough, and that this implies that migration will take place. Because this implies a basic change of algorithm, it will have major effects on both future games and future graphics courses. The main purpose of this paper is to examine what those effects might be.

2 Ray tracing versus rasterization

In this section we review the ray tracing and z-buffer algorithms, the applications that use them. Current GPUs are based on the z-buffer [Catmull 1974] which is a straightforward algorithm. In hardware implementations it typically has two frame buffers for color and one for z (depth) values. While computing one of the color buffers (the "back" buffer), it displays the other (the "front" buffer). When all of the colors are computed in the back buffer, the two buffers are "swapped" (the front becomes the back and vice-versa) and the new set of colors are displayed. The z-buffer is only used while computing the new colors in the back buffer. Computing the back buffer is a loop over all triangles:

```
initialize all pixel colors to background color
initialize all pixel $z$ values to $\infty$
for all $N$ triangles do
    for each pixel $p$ that triangle might be seen through do
        compute color $c_{new}$ and depth $z_{new}$
        if $z_{new} < z_p$ then
            $c_p = c_{new}$
            $z_p = z_{new}$
```

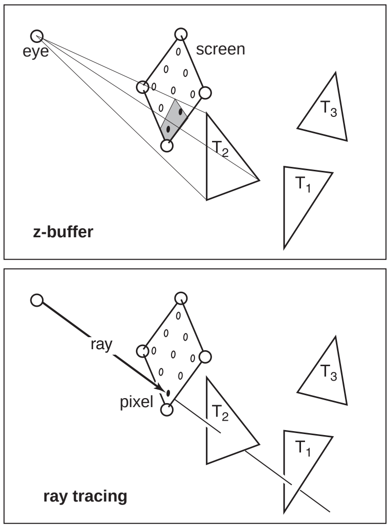

Figure 2: Top: the z-buffer algorithm projects a triangle toward the nine pixel screen and writes all pixels with the distance to the eye (the "z" value) and its color unless a smaller distance is already written in the z-buffer. Bottom: the ray tracing algorithm sends a 3D half-line (a "ray") into the set of objects and finds the closest one. In this case the triangle T_2 is returned.

The *if* statement is how the hidden surface elimination remains simple; overlapping polygons settle their order by means of storing the closest depth value seen so far in the triangle loop. This algorithm's runtime is proportional to the number of triangles N. While the algorithm can be made sub-linear through occlusion culling [Bittner et al. 2004], randomized culling [Wand et al. 2001], and by level-of-detail management [Luebke et al. 2002], these techniques add complexity and data restrictions to the implementations.

The z-buffer algorithm has difficulty in three main areas: rendering images with shadows and mirror-like reflections and refractions, rendering images with extremely large data sets, and rendering images with primitives that are not simple triangles. The ray tracing algorithm loops over pixels rather than objects:

```
for all $P$ pixels do
    find the nearest object seen through that pixel
```

This loop "finds the nearest object" by doing a line query in 3D (Figure 2). Some implementations do "double-buffering" like the z-buffer above, but the algorithm above is inherently "frameless" in that a pixel can be immediately updated when computed. Frameless implementations have some advantages in responsiveness [Bishop et al. 1994; Parker et al. 1999; Woolley et al. 2003], and are problematic for the z-buffer which does not know the color of any pixel until the end of its main loop.

There are five key advantages of ray tracing:

1. for preprocessed models, ray tracing is sub-linear in N [Cleary et al. 1983], so for some sufficiently large value of N it will always be faster than a z-buffer which is linear in N;

2. ray tracing can process curved surfaces such as splines in their native form [Martin et al. 2000];

3. ray tracing can naturally support volume data with partial transparency [Upson and Keeler 1988];

4. because a ray tracing program must perform a 3D line query, it can reuse this line query to easily generate shadows and reflections that also depend on a 3D line query (Figure 3) [Whitted 1980];

5. the ray tracing algorithm is highly parallel, and has been demonstrated to have over 91% parallel efficiency on 512 processors [Parker et al. 1999].

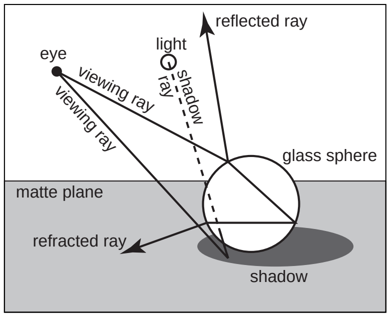

Figure 3: Ray tracing can easily generate shadow rays, reflected rays, and refracted rays. These rays need not have a shared origin so they are difficult to duplicate for a z-buffer algorithm.

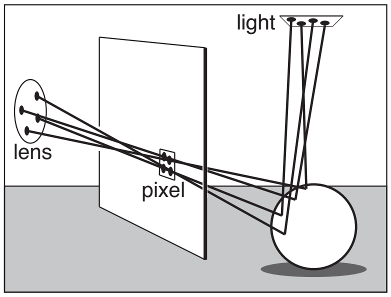

Figure 4: In distribution ray tracing, multiple samples are taken per pixel and these are used both for antialiasing and other effects such as soft shadows as is shown here by sampling different points on an area light source.

While the z-buffer can be made to do specular reflections and shadows [Stamminger and Drettakis 2002; Wyman 2005], the underlying techniques are neither general nor robust. However, the z-buffer algorithm does currently have two important advantages over ray tracing: although it is linear in N, it has a very low time constant so it can render scenes with moderate N very quickly; it has a mass-produced hardware implementation available that has lowered the time constant even further. Because the N for real applications is increasing exponentially for most applications, the time constant advantage is of decreasing importance. Further, the ubiquitous special purpose hardware exaggerates the time constant difference between ray tracing and the z-buffer approaches.

Ultimately, we expect *distribution ray tracing* (Figures 4 and 5) to end up on the desktop because of its higher image quality. However, because it requires tens of samples per pixel, it is intrinsically more expensive than traditional Whitted-style ray tracing.

Simulation and games demand interactivity and currently use z-buffer hardware almost exclusively. However, they spend a great deal of effort and programmer time creating complicated "hacks" to fake lighting effects and reduce N by model simplification and in the end they have imagery of inferior quality to that generated by ray tracing. Those industries would use ray tracing if it were fast enough.

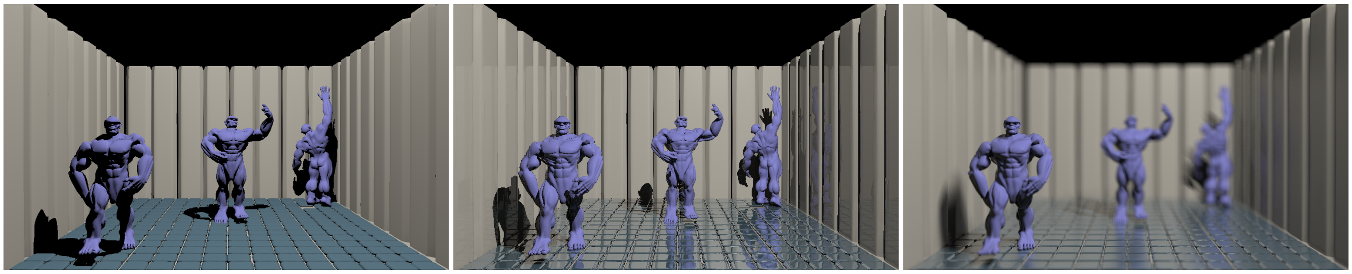

Figure 5: Left: ray tracing with shadows only with 1 sample per pixel. Middle: Whitted-style ray tracing with 1 sample per pixel. Right: distribution ray tracing with 64 samples per pixel. We expect Whitted-style ray tracing to soon be on the desktop, and distribution ray tracing to later follow.

3 Why we think ray tracing is coming

In this section we examine exactly what the gap is between current ray tracing performance and that needed for games, and speculate on how that gap will be closed.

Because the console market is so influential in the games market, we assume a 1080p monitor (an HDTV screen with about two million pixels) at 60Hz. Our own software implementation of Whitted ray tracing runs at approximately 1 million pixels per second per core on complex scenes (hundreds of thousands of primitives) on a 2GHz Opteron 870. For distribution ray tracing with 64 samples per pixel we are about 100 times slower than that. The slowdown is more than a factor of 64 because in addition to using 64 times as many samples, the spread of rays resulting from sampling areas such as lights decreases coherence and adds some per-sample computation. So in software on one core with current CPU technology we are approximately a factor of 100 away from our 60Hz 2 million pixel goal, and for distribution ray tracing we are approximately a factor of 10,000 away. So we believe we need improvements of about 100 times over current one-core software systems for ray tracing to be fluid for games, and a further factor of 100 for distribution ray tracing to become fluid.

For adequate, as opposed to ideal, performance, we could also assume 480p30 (300k pixels, 30Hz). Further, we could use 16 samples per pixel for distribution ray tracing. This puts us less a factor of ten away for one core for Whitted-style ray tracing and less than a factor of 200 for distribution ray tracing for one core. Given that eight core CPUs are on the horizon, and dual CPU quad-core systems are available now, Whitted-style ray tracing in software should be fluid quite soon on the desktop with a 480p30 display.

Because clock speeds and process sizes are unlikely to shrink much more [sia 2004], to gain a factor of 100 over current one core systems, a change of architecture is probably needed. Current GPUs have impressive raw performance, but so far they have not been shown to be efficient for ray tracing [Purcell et al. 2002]. More promising are architectures such as Intel's Tera-scale prototype that use tens of simple cores with good floating point capabilities. Such chips would surely be better suited to ray tracing than current general-purpose CPUs, and would probably be sufficient for 1080p Whitted-style ray tracing. For distribution ray tracing, a special purpose ray tracing chip may be needed. A prototype ASIC design indicates that this is probably a practical route even if improvements in process are slow from now on [Woop et al.].

In summary, Whitted-style ray tracing at 480p30 is already fluid for complex models on 8 core systems. Improvements to hardware should allow a gradual migration to 1080p60 and later distribution ray tracer. The basic issues faced by users and programmers will not be influenced much by that migration. Because of ray tracing's high visual quality, it should have a market, and games programmers will need to understand ray tracing.

4 Impact on Computer Graphics Courses

Over the past two decades computer graphics courses [Cunningham et al. 1988; Grissom et al. 1995; Hitchner et al. 1999; Angel et al. 2006] exhibit a pattern of curriculum refinement driven by the changes in graphics hardware. Although the foundations in the field remain the same, the subset of these concepts covered and the context for which these concepts are presented have evolved significantly. For example, 20 years ago learning and practicing raster line drawing algorithms were relevant, while with current GPUs it is more important to understand and practice the mathematics behind interactive camera control. If we are correct that ray tracing will replace z-buffer then computer graphics courses will once again need to evolve.

Just as real time z-buffer hardware did not alter the *core* concepts in computer graphics, we do not expect real time ray tracing hardware to fundamentally impact the discipline. The topics covered in an introductory course will continue to be foundational concepts [Angel et al. 2006] such as transformation, hierarchical modeling, illumination models, camera modeling. However, as in the case of real time z-buffer, we expect the priority and context to evolve. We now list the biggest likely changes in emphasis.

Illumination models. Ray tracing integrates visibility and illumination computations. For example, shadows are computed as part of visible energy received from the light source, and reflection is computed as visible energy received from the mirror reflection direction. In this way, many common physical effects that are currently referred to as "special effects" (e.g., transparency, reflection, etc.) will evolve back into their natural illumination computations.

Perspective transform and homogeneous coordinate. Ray tracing simulates perspective naturally. For this reason, the mathematics model that simulates foreshortening and the associated homogeneous coordinate system will become less important. This will simplify the traditional transformation pipeline, where the last stage of the pipeline, projection transform, will not be needed anymore.

Higher-order surfaces. Ray tracing computes visibility by mathematically intersecting a line and a primitive. It is straightforward to ray trace mathematically defined higher-orders surfaces such as trimmed NURBS. Without tessellation, higher-order primitives have much more compact representations while retaining all the original geometric integrity. For example, an implicit sphere can be represented by a handful of variables while the mathematic expression maintains quadratic continuity throughout the surface. The current heavier emphasis on object models based on tessellated triangle meshes will shift towards modeling based on surfaces' native representations.

Volumetric effects. Ray tracing supports volumetric data naturally. Volumetric effects like arial fog or more general participating media (e.g, smoke) can be modeled as semitransparent volumetric primitives with dedicated illumination models. The needs for special case shaders, *tricks*, and *hacks* for such effects will greatly diminished.

5 Impact on Computer Gaming Courses

The front end visualization of interactive computer games depends on computer graphics. Conversely, it is also true that computer gaming is the single most important factor driving the development of computer graphics hardware. For these reasons, when discussing impacts of interactive ray tracing, we should also examine the effects on computer games. Future computer games will take advantage of the new functionality: the faster ray tracing hardware, and the drastically improved realism. As educators, our tasks are to analyze and understand how to evolve computer gaming courses/curricula accordingly.

The development of courses/curricula associated with computer gaming has lagged behind the industry quite significantly. While computer gaming has been around in different forms for many decades, the first classes dedicated to *games development* has become available only in the early 1990s [Parberry et al. 2006]. During these early times there were very few computer gaming related classes. Most of the efforts in incorporating gaming into computer science classes/curricula had begun only recently (e.g., [Coleman et al. 2005; Parberry et al. 2005; Maxim 2006; DXFramework 2006; MUPPET 2006; Parberry 2006]). In addition, many of these efforts represented strategies to increase interest and enthusiasm for the discipline to counter the drastic downturn in enrollments [Vegso 2005]. Computer gaming as a discipline in computer science is a work-in-progress.

The following discussion is organized based on the framework described in Sung et al. [2007], where we discuss the effects of interactive ray tracing in each of the: *games programming*, *games development*, and *games clients* categories.

Games programming classes. These are classes that study general issues related to programming computer games (e.g., Kuffner's CMU course [Sweedyk et al. 2005]). The following are games related programming issues associated with fast ray tracing.

- Ray tracing is capable of computing line/primitive intersection extraordinarily fast. The intersection computation is the foundation for supporting collision and selection. From the games programming perspective, the challenge would be to efficiently integrate this functionality into the core of games engine. In addition, the entire collision subsystem should be reevaluated. For example, with the ray tracing collision support, the merit of support collision primitives becomes questionable.
- Ray tracing naturally supports partial redraw where redraws only need to occurr in regions that changed from previous frame. This means polygon/primitive count will not be the only factor affecting the bottom-line frame rate. In this case, the frame-to-frame coherency may be even more important. From games programming perspective, it is important to understand and take advantage of temporal coherence.

Games development classes. These are classes that study how to design new games as an end product (e.g., [Jones 2000; Coleman et al. 2005; Parberry et al. 2005]). Students in these classes must be concerned with all aspects of a real game production including entertainment value, visual quality, audio effects, etc. For these classes, the challenges are in game design to improve the aesthetic and the general game play experience based on the new functionality.

- As discussed, many existing *special* effects will become *common* and *natural* illumination effects with ray tracing. Not being special means there will be no special restrictions associated with the effects (e.g., no restriction on mirror must be planar). However it is also true that the computations of these effects will not be free or cheap. For example, although any and every object in a scene can be reflective, it is always faster to compute the frame with no reflections. In games development classes, students must learn to balance the new found flexibility, against the associated cost to achieve the specific aesthetic needs of their games.
- As discussed, the ray tracing hardware supports efficient collision detection, and the ray tracing paradigm supports partial redraw. These two characteristics suggest that as long as we limit the changes in consecutive frames, we can design and interact with scenes of high complexity. For example, walking into a room full of objects with detailed geometry with the ability to interact (e.g., pickup and open) with any object.

Games clients classes. These are classes that use gaming as vehicles for delivering specific concepts (e.g., [da Silva 2006b; da Silva 2006a; Sung et al. 2007]). These classes are *applications* of computer gaming, we expect the impact on classes in this category to be indirect and minimal.

Discussion. Our outline of changes to courses above is by no means exhaustive. Our areas of expertise are in computer graphics and computer architecture. Based on our knowledge, we predict interactive ray tracing will become more prominent in the near future. Because of our background, and because computer graphics is a better established field, we have some understanding on the impact of this upcoming change with respect to computer graphics. We believe ray tracing will also significantly impact computer gaming and computer gaming related classes. In this last section, we presented our speculations, but we are not as confident in these ideas as games classes are much less well established than graphics classes.

Acknowledgments

This work is supported in part by the National Science Foundation grants DUE-0442420, NSF-0541009, and NSF 03-06151; and a grant from the Microsoft Research Gaming RFP award number 15871. All opinions, findings, conclusions, and recommendations in this work are those of the authors and do not necessarily reflect the views of the National Science Foundation or Microsoft.

References

Angel, E., Cunningham, S., Shirley, P., and Sung, K. 2006. Teaching computer graphics without raster-level algorithms. In *Proceedings of SIGCSE*, 266–267.

Bishop, G., Fuchs, H., McMillan, L., and Zagier, E. J. S. 1994. Frameless rendering: Double buffering considered harmful. In *Proceedings of SIGGRAPH*, 175–176.

Bittner, J., Wimmer, M., Piringer, H., and Purgathofer, W. 2004. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum 23*, 3, 615–624.

Boulos, S., Edwards, D., Lacewell, J. D., Kniss, J., Kautz, J., Shirley, P., and Wald, I. 2006. Packet-based whitted and distribution ray tracing. Tech. Rep. UUCS-06-013, School of Computing, University of Utah.

Catmull, E. 1974. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD thesis, University of Utah.

Cleary, J., Wyvill, B., Birtwistle, G., and Vatti, R. 1983. A Parallel Ray Tracing Computer. In *Proceedings of the Association of Simula Users Conference*, 77–80.

Coleman, R., Krembs, M., Labouseur, A., and Weir, J. 2005. Game design & programming concentration within the computer science curriculum. In *Proceedings of SIGCSE*, 545–550.

Cook, R. L., Porter, T., and Carpenter, L. 1984. Distributed ray tracing. In *Proceedings of SIGGRAPH*, 165–174.

Cunningham, S., Brown, J. R., Burton, R. P., and Ohlson, M. 1988. Varieties of computer graphics courses in computer science. In *Proceedings of SIGCSE*, 313–313.

da Silva, F. S. C., 2006. Artificial intelligence for computer games. University of Sao Paulo (USP/SP), Microsoft Academic Alliance Repository Newsgroup, Object ID: 6210, *http://www.msdnaacr.net/curriculum/pfv.aspx?ID=6210*.

da Silva, F. S. C., 2006. Software enginnering for computer games. University of Sao Paulo (USP/SP), Microsoft Academic Alliance Repository Newsgroup, Object ID: 6211, *http://www.msdnaacr.net/curriculum/pfv.aspx?ID=6211*.

DXFramework, 2006. Dxframework: A pedagogical computer game engine library. University of Michigan, *http://dxframework.org/*.

Grissom, S., Kubitz, B., Bresenham, J., Owen, G. S., and Schweitzer, D. 1995. Approaches to teaching computer graphics (abstract). In *Proceedings of SIGCSE*, 382–383.

Hitchner, L., Cunningham, S., Grissom, S., and Wolfe, R. 1999. Computer graphics: the introductory course grows up. In *Proceedings of SIGCSE*, 341–342.

Jones, R. M. 2000. Design and implementation of computer games: a capstone course for undergraduate computer science education. In *Proceedings of SIGCSE*, 260–264.

Luebke, D., Watson, B., Cohen, J. D., Reddy, M., and Varshney, A. 2002. *Level of Detail for 3D Graphics*. Elsevier Science Inc., New York.

Martin, W., Cohen, E., Fish, R., and Shirley, P. S. 2000. Practical ray tracing of trimmed nurbs surfaces. *Journal of Graphics Tools 5*, 1, 27–52.

Maxim, B., 2006. Game design and implementation 1 and 2. Microsoft Academic Alliance Repository Newsgroup, Object ID: 6227, *http://www.msdnaacr.net/curriculum/pfv.aspx?ID=6227*.

MUPPET, 2006. Multi-user programming pedagogy for enhancing traditional study. Rochester Institute of Technology, *http://muppets.rit.edu/muppetsweb/people/index.php*.

Parberry, I., Roden, T., and Kazemzadeh, M. B. 2005. Experience with an industry-driven capstone course on game programming: extended abstract. In *Proceedings of SIGCSE*, 91–95.

Parberry, I., Kazemzadeh, M. B., and Roden, T. 2006. The art and science of game programming. In *Proceedings of SIGCSE*, 510–514.

Parberry, I., 2006. Sage: A simple academic game engine. University of North Texas, *http://larc.csci.unt.edu/sage/*.

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

Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. 2002. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3, 703–712.

2004. International technology roadmap for semiconductors. www.itrs.net / Common / 2004Update / 2004Update.htm.

Stamminger, M., and Drettakis, G. 2002. Perspective shadow maps. In *Proceedings of SIGGRAPH*, 557–562.

Sung, K., Shirley, P., and Reed-Rosenberg, R. 2007. Experiencing aspects of games programming in an introductory computer graphics class. In *Proceedings of SIGCSE*. to appear.

Sweedyk, E., deLaet, M., Slattery, M. C., and Kuffner, J. 2005. Computer games and cs education: why and how. In *Proceedings of SIGCSE*, 256–257.

Upson, C., and Keeler, M. 1988. VBUFFER: Visible volume rendering. In *Proceedings of SIGGRAPH*, 59–64.

Vegso, J. 2005. Interest in cs as major drops among incoming freshmen. *Computing Research News 17*, 3 (May).

Wand, M., Fischer, M., Peter, I., auf der Heide, F. M., and Strasser, W. 2001. The randomized z-buffer algorithm: interactive rendering of highly complex scenes. In *Proceedings of SIGGRAPH*, 361–370.

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

Woolley, C., Luebke, D., Watson, B., and Dayal, A. 2003. Interruptible rendering. In *ACM Symposium on Interactive 3D Graphics*, 143–151.

Woop, S., Brunvand, E., and Slusallek, P. Estimating performance of a ray-tracing ASIC design. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 7–14.

Wyman, C. 2005. An approximate image-space approach for interactive refraction. In *Proceedings of SIGGRAPH*.