Fast Ray Tracing and The Potential Effects on Graphics and Gaming Courses

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

The modern graphics processing units (GPUs), found on almost every personal computer, use the Z-buffer algorithm to compute visibility. 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 benefited from improvements in computer hardware, and many believe it will replace the z-buffer algorithm as the graphics engine on PCs. If this replacement happens, it will imply fundamental changes in both the API to and capabilities of 3D graphics engines. This paper overviews the backgrounds in Z-buffer and ray tracing, presents our case that ray-tracing will replace Z-buffer in the near future, and discusses the implications for graphics oriented classes should this switch to ray tracing occur. Since computer gaming is one of the most important industry driving graphics hardware and the fact that recently there are many computer science courses related to games and games development, we also describe the potential impact on games related classes.

Categories and Subject Descriptors

K.3.2 [Computers and Education]: Computer and Information Science Education – *computer science education*

Keywords: ray tracing, z-buffer, curriculum, graphics courses, computer gaming

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, there are three classes of applications that have not significantly benefited from this revolution:

- those that have datasets much larger than a few million triangles;
- those that have non-polygonal data not easily converted into triangles;
- those that demand high quality shadows, reflection, and refraction effects.

These classes of applications typically use Whitted's ray tracing algorithm [Whitted 1980; Glassner 1989; Shirley and Morley 2003] or Cook's distribution ray tracing algorithm (DRT) which is an order of magnitude more expensive than simple Whitted ray tracing but allows 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 quantum 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 such a

migration implies a basic change of algorithm, it will have major effects on both future future graphics and games related courses. The main purposes of this paper are to analyze these issues and examine how courses related to graphics and games might be effected.

Another reason we think ray tracing should have more of a place in computer graphics classes is that academics is well ahead of industry in ray tracing research. This enables an instructor to make the students feel like they are "in the club" in a way that their cohorts in industry are not. It becomes increasingly difficult to convince a college-age student that they can learn things in the academic environment that is unavailable in industry, it is important to take advantage of such opportunities and ray tracing presents a major one.

We first give an overview of the z-buffer algorithm found in graphics processing units (GPUs), and contrasts this with the ray tracing algorithm. We then argue, in detailed, why we believe ray tracing is coming to the desktop, and probably sooner than is commonly believed. Finally, we discuss what the implications of this change are for graphics and games education.

2 Ray tracing versus rasterization

In this section we review the ray tracing and z-buffer algorithms, the applications that use them, the performance such applications demand, and the various ways such performance might be delivered.

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 ∞
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**
      $c_p = c_{new}$
      $z_p = z_{new}$

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

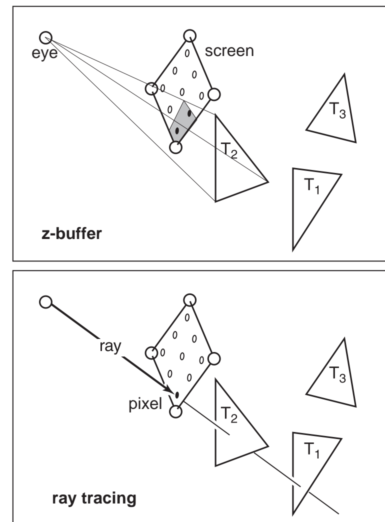

Figure 2: Top: the z-buffer algorithm projects a triangle toward the nine pixel screen and writes all pixels with the distance to the eye (the "z" value) and its color unless a smaller distance is already written in the z-buffer. Bottom: the ray tracing algorithm sends a 3D half-line (a "ray") into the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

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].

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

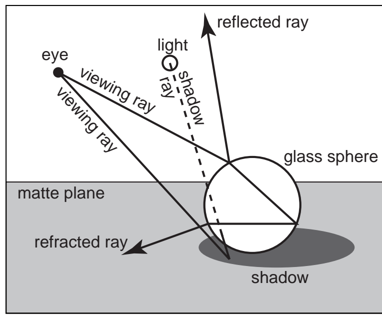

Figure 3: Ray tracing can easily generate shadow rays, reflected rays, and refracted rays. These rays need not have a shared origin so they are difficult to duplicate for a z-buffer algorithm.

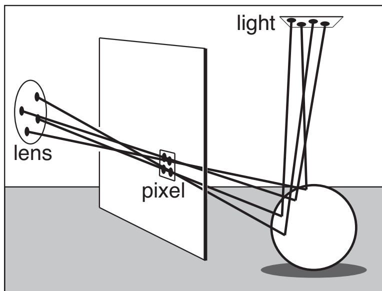

Figure 4: In distribution ray tracing, multiple samples are taken per pixel and these are used both for antialiasing and other effects such as soft shadows as is shown here by sampling different points on an area light source.

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.

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. As a reference, we target a HDTV-type 1080p monitor (e.g., displays by game console) refreshed at 60Hz, or “real time” frequency. This is about two million pixels per update.

Because visual quality is important, we must also consider “secondary” rays. These can be for shadows, reflections, or refractions. The individual viewing ray and secondary rays are sometimes called *ray segments* [Hurley 2005], and we will use that terminology. We note that for a general scene many ray segments per viewing ray are typically required. For example, a headlight model will have five segments for the simplest path through the assembly (one viewing ray, two refracted rays before the reflector, one reflected ray, and two refracted rays through inner and outer surfaces of the glass to leave the headlight). Other paths will have more. An architecture space will typically have dozens up to hundreds of lights, each generating a shadow ray segment.

The dominant visual artifact of interactive ray tracing programs is flickering caused by subpixel objects of different colors moving over the pixel center when the objects or eye moves [Martin et al. 2002]. This is best alleviated by sending multiple viewing rays per pixel and averaging the result, which not only alleviates flickering but also provides antialiasing. Current z-buffer systems use 4-9 samples per pixel on object edges to accomplish this. A high quality ray traced image must have at least 4 viewing rays per pixel.

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.

A general rule of thumb in system design is that system generality and system performance are inversely correlated. That is, a system that is good in general is not excellent at anything in specific. This is exactly the insight that caused the GPU revolution in the 1980’s [Clark 1982; Poulton et al. 1985; Fuchs et al. 1985; Deering et al. 1988]. A carefully crafted computational pipeline for transforming triangles and doing depth checks combined with an equally carefully crafted memory system to feed those pipelines combined to make our current generation of z-buffer GPUs possible. Current GPUs have up to 192 floating point units on a single GPU [ati] and aggregate memory bandwidth of 15-30Gbytes per second from their on-chip memories [nVidia Corporation]. These combine to achieve graphics performance that is orders of magnitude higher than could be achieved by running the same algorithms on a general purpose processor.

Virtually all of these performance gains of GPUs are due to architectural improvements. In fact, GPUs generally run with quite a bit slower clock than high-performance CPUs in the same process generation. A state of the art GPU might run at 300-450MHz whereas a CPU might run at close to 4GHz. The architectural advantage comes mainly from custom floating point pipelines designed to execute the operations required for z-buffer graphics and from wide, fast, on-chip memory structures. Pipelines of this sort reduce the need to bring data in and out of a register file as is done on a

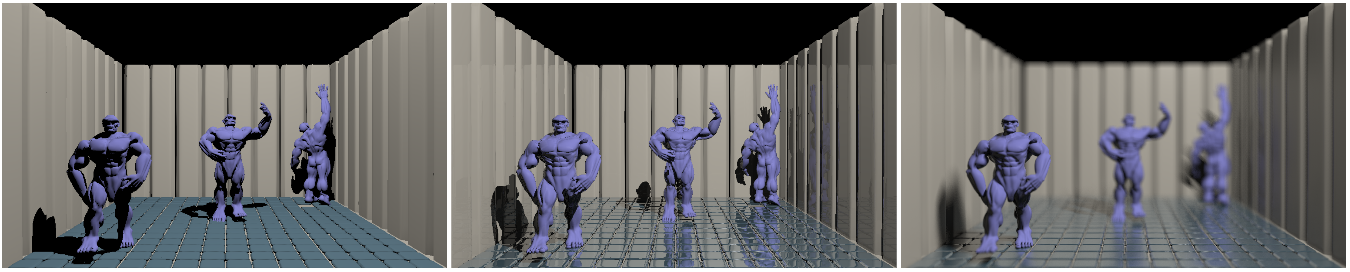

Figure 5: Left: ray tracing with shadows only with 1 sample per pixel. Middle: Whitted-style ray tracing with 1 sample per pixel. Right: distribution ray tracing with 64 samples per pixel. We expect Whitted-style ray tracing to soon be on the desktop, and distribution ray tracing to later follow.

general CPU. High-performance CPUs generally also use a large number of custom and dynamic circuits in their critical paths that are hand designed and hand tuned for speed. In general, custom circuits run between 3x to 8x faster than even semi-custom cell-based ASICs [Chinnery and Keutzer 2002], but these highly sensitive and difficult custom circuits are not typically required in GPUs, other than as part of their highly-replicated floating point arithmetic units. This is encouraging because it means that if similar architectural improvements are possible for a ray tracing chip, extremely high-performance custom circuit techniques should not be necessary, other than for a few highly replicated arithmetic subsystems.

Fundamentally ray tracing, like GPUs, will require a special purpose pipeline of floating point arithmetic units tuned to a specific set of operations. There are three fundamental operations that must be supported: intersecting a ray with the spatial data structure that encapsulates the scene objects, intersecting the ray with the primitive objects contained in the selected data structure leaf nodes, and computing the illumination and color of the pixel based on the intersection with the primitive object and the collection of the contributions from the secondary ray segments. These operations are different enough from the GPU pipeline that it is unlikely that a GPU will ever support ray tracing in an effective way. However, it is certainly feasible to design suitable high-performance floating point units and connect them in ways suitable to the ray tracing algorithm. More promising are architectures such as Intel's Terascale 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 one can argue that it is more important to understand and practice the mathematics behind interactive camera control[Angel et al. 2006]. 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 [Cunningham 2000; Wolfe 2000] 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 only recently demonstrated significant results (e.g., [Coleman et al. 2005; Parberry et al. 2005; Maxim 2006; DXFramework 2006; MUPPET 2006; Parberry 2006; Sung et al. 2007]). 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]. Although there are dedicated curriculum designed to educate next generation games developers (e.g., [Coleman et al. 2005; Zyda 2006; Murray et al. 2006; Fullerton 2006; Argent et al. 2006]), these programs are new, 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. [Sung et al. 2007], where we discuss the effects of interactive ray tracing in each of the: *game programming*, *game development*, and *game clients* categories.

Game programming classes. These are classes that study general technical issues related to programming computer games (e.g., Kuffner's CMU course [Sweedyk et al. 2005]). For example, topics covered may include path planning algorithms, terrain representation, etc. The topics covered in these classes are general and typically can be applied to different domains. These classes concentrate on covering the games programming topic area in the IDGA [IDGA 2003] curriculum framework.

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 occur 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.

Game 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. When evaluated against the IDGA curriculum framework we see that these classes cover all the major core topic areas. 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.

Game client classes. These are non-game related CS classes that creatively integrate games into their existing curriculum. Typically, games are used as programming assignments (e.g., [Adams 1998; Faltin 1999; Giguette 2003; Sung and Shirley 2004; Sweedyk et al. 2005; Bayliss and Strout 2006; Lewis and Massingill 2006]) or to teach abstract concepts (e.g. [Becker 2001; Giguette 2003; Hansen 2004; Dann et al. 2006; MUPPET 2006; Sung et al. 2007]), or as an example application area to teach the concepts involved in an entire topic area (e.g., [da Silva 2006b; da Silva 2006a; Chen and Cheng 2007]). These are *traditional* CS classes that exist independent from game programming. These classes are actually *clients* of game development where they *use* game development as a vehicle to deliver specific abstract concepts. After these classes, students are expected to understand the abstract concepts, and not about game development. 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

ADAMS, J. C. 1998. Chance-it: an object-oriented capstone project for cs-1. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, ACM Press, New York, NY, USA, 10–14.

ANGEL, E., CUNNINGHAM, S., SHIRLEY, P., AND SUNG, K. 2006. Teaching computer graphics without raster-level algorithms. In *Proceedings of SIGCSE*, 266–267.

ARGENT, L., DEPPER, B., FAJARDO, R., GJERTSON, S., LEUTENEGGER, S. T., LOPEZ, M. A., AND RUTENBECK, J. 2006. Building a game development program. *Computer 39*, 6, 52–60.

ATI technologies inc. www.ati.com.

BAYLISS, J. D., AND STROUT, S. 2006. Games as a "flavor" of cs1. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, ACM Press, New York, NY, USA, 500–504.

BECKER, K. 2001. Teaching with games: the minesweeper and asteroids experience. *J. Comput. Small Coll. 17*, 2, 23–33.

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.

CHEN, W.-K., AND CHENG, Y. C. 2007. Teaching object-oriented programming laboratory with computer game programming. *IEEE Transctions on Education 50*, 3 (August), 197–203.

CHINNERY, D., AND KEUTZER, K. 2002. *Closing the Gap Between ASIC & Custom*. Kluwer Academic Publishers Group.

CLARK, J. H. 1982. The geometry engine: A vlsi geometry system for graphics. In *SIGGRAPH '82: Proceedings of the 9th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 127–133.

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.

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.

CUNNINGHAM, S. 2000. Powers of 10: The case for changing the first course in computer graphics. *Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Eductaion* (March), 293–296.

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*.

DANN, W., COOPER, S., AND PAUSCH, R. 2006. *Learning to Program with Alice*. Prentice Hall, Upper Saddle River, NJ.

DEERING, M., WINNER, S., SCHEDIWY, B., DUFFY, C., AND HUNT, N. 1988. The triangle processor and normal vector shader: a vlsi system for high performance graphics. In *SIGGRAPH '88: Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 21–30.

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

FALTIN, N. 1999. Designing courseware on algorithms for active learning with virtual board games. In *ITiCSE '99: Proceedings of the 4th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education*, ACM Press, New York, NY, USA, 135–138.

FUCHS, H., GOLDFEATHER, J., HULTQUIST, J. P., SPACH, S., AUSTIN, J. D., FREDERICK P. BROOKS, J., EYLES, J. G., AND POULTON, J. 1985. Fast spheres, shadows, textures, transparencies, and imgage enhancements in pixel-planes. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 111–120.

FULLERTON, T. 2006. Play-centric games education. *Computer 39*, 6, 36–42.

GIGUETTE, R. 2003. Pre-games: games designed to introduce cs1 and cs2 programming assignments. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, ACM Press, New York, NY, USA, 288–292.

GLASSNER, A., Ed. 1989. *An introduction to ray tracing*. Academic Press, London.

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.

Hansen, S. 2004. The game of set®: an ideal example for introducing polymorphism and design patterns. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, ACM Press, New York, NY, USA, 110–114.

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

Hurley, J. 2005. Ray tracing goes mainstream. *Intel Technology Journal 9*, 2, 99–108.

IDGA, 2003. IGDA curriculum framework report version 2.3 beta, February. International Game Developer's Association, *http://www.igda.org/academia*.

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

Lewis, M. C., and Massingill, B. 2006. Graphical game development in cs2: a flexible infrastructure for a semester long project. In *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, ACM Press, New York, NY, USA, 505–509.

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.

Martin, W., Shirley, P., Parker, S., Thompson, W., and Reinhard, E. 2002. Temporally coherent interactive ray tracing. *Journal of Graphics Tools 7*, 2, 41–48.

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*.

Murray, J., Bogost, I., Mateas, M., and Nitsche, M. 2006. Game design education: Integrating computation and culture. *Computer 39*, 6, 43–51.

nVidia Corporation. www.nvidia.com.

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.

Poulton, J., Fuchs, H., Austin, J. D., Eyles, J. G., Heineche, J., Hsieh, C., Goldfeather, J., Hultquist, J. P., and Spach, S. 1985. PIXEL-PLANES: Building a VLSI based raster graphics system. In *Chapel Hill Conference on VLSI*.

Shirley, P., and Morley, R. K. 2003. *Realistic Ray Tracing*. A. K. Peters, Natick, MA.

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., and Shirley, P. 2004. Returning adult students for algorithm analysis. *The Journal of Computing Sciences in Colleges 20*, 2 (December), 62–72. Proceedings of the Sixth Annual CCSC-NW Conference.

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.

Wolfe, R. 2000. Bringing the introductory computer graphics course into the 21st century. *Computers and Graphics 24*, 1, 151–155.

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*.

Zyda, M. 2006. Guest editor's introduction: Educating the next generation of game developers. *Computer 39*, 6, 30–34.