Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2013, Article ID 578269, 14 pages
http://dx.doi.org/10.1155/2013/578269

Research Article

The Brigade Renderer: A Path Tracer for Real-Time Games

Jacco Bikker and Jeroen van Schijndel

ADE/IGAD, NHTV Breda University of Applied Sciences, Monseigneur Hopmansstraat 1, 4817 JT Breda, The Netherlands

Correspondence should be addressed to Jacco Bikker; bikker.j@gmail.com

Received 20 September 2012; Accepted 30 December 2012

Academic Editor: Yiyu Cai

We present the Brigade renderer: an efficient system that uses the path tracing algorithm to produce images for real-time games. We describe the architecture of the Brigade renderer, and provide implementation details. We describe two games that have been created using Brigade.

1. Background

Historically, games have been an important driving force in the advance of graphics hardware and rendering algorithms. Effort has evolved from striving for abstract, visually pleasing results, to more plausible realistic rendering. In the former, a distinct visual style is chosen, which does not necessarily require realism. Instead, over-the-top animation styles and matching graphics are used. Examples of this approach are most early 2D computer games, but there are also more recent titles such as Super Mario Galaxy [1] and Okami [2] (Figure 1).

Many modern games strive for realistic graphics, where the goal is to convince the player that the result is (or could be) realistic. Examples are racing games such as the Gran Turismo series [3] and flight simulators such as Tom Clancy's H.A.W.X. [4] (Figure 2), which use rasterization-based renderers, augmented with various algorithms to add secondary effects such as shadows, reflections, and indirect illumination.

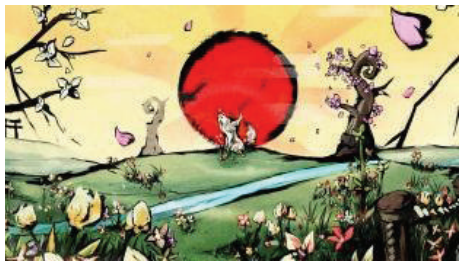
Recently, efforts are being made towards physically correct results. For static scenery and a static light configuration, this can be achieved by precalculating global illumination, or by coarsely calculating radiosity. Examples of this are games based on the Unreal 3 engine [5] (Figure 3). Games using the Frostbite 2 engine [6] support ray tracing of coarse level geometry for glossy reflections. The Unreal 4 engine [7] supports approximate global illumination using cone tracing [8].

Physically based rendering of virtual worlds has strong advantages. The obvious advantage is image fidelity (Figure 4). Perhaps of equal importance, however, is production efficiency. Whereas lighting for a scene in a rasterization-based engine typically requires a designer to work around technical limitations of the renderer to make the lighting look right, physically based rendering naturally leads to correct lighting. This limits the design effort to a creative process alone.

Of the available physically based rendering algorithms, stochastic ray-tracing based methods (path tracing and derived methods) are favored over finite element methods, due to their elegance and efficient handling of large scenes. Unlike rasterization-based approaches, path tracing scales up to photo realism with minimal algorithmic complexity: the only dependencies are compute power and memory bandwidth. Both increase over time. Moore's law states that the number of transistors that can be placed inexpensively on an integrated circuit rises exponentially over time [9]. Although the link between transistor count and application performance is complex, the latter follows the same pattern, with compute power increasing at 71% per year on average, and DRAM bandwidth at 25% per year [10].

Assuming that all other factors remain constant (e.g., scene complexity, screen resolution), it can thus be assumed

(a)



(b)

Figure 1: Two examples of modern games that use a nonrealistic visual style. (a) Super Mario Galaxy, (b) Okami.

(a)

(b)

Figure 2: Two examples of modern games that aim for a high level of realism. (a) Tom Clancy's H.A.W.X., (b) Gran Turismo 5.

that there will be a point where physically based rendering is feasible on consumer hardware.

2. Previous Work

Recently, Whitted-style ray tracing and distribution ray tracing have been shown to run in real-time, or at least at interactive frame rates, on CPUs (see e.g., [12–15] and GPUs [16–19], as well as the streaming processors of modern consoles [20, 21]).

Interactive path tracing was first mentioned in 1999 by Walter et al. as a possible application of their Render Cache system [22]. Using their system and a sixty-core machine, a scene can be navigated at interactive frame rates. During camera movement, samples are cached and reprojected to construct an approximation for the new camera view point. New samples are created for pixels with a relatively large error. The image converges to the correct solution when the camera is stationary.

Sadeghi et al. use ray packets for their path tracer [23]. Coherence between rays on the paths of block of pixels is obtained by using the same random numbers for all pixels in the block. This introduces structural noise but remains unbiased. The system is CPU based and achieves about 1.2 M rays per second per core of an Intel Core 2 Quad running at 2.83 Ghz.

In their 2009 paper, Aila and Laine evaluate the performance of various ray traversal kernels on the GPU [19]. Although they did not aim to implement a full path tracer, their measurements include a diffuse bounce, for which they report nearly 50 M rays per second on an NVidia GTX285, not including shading.

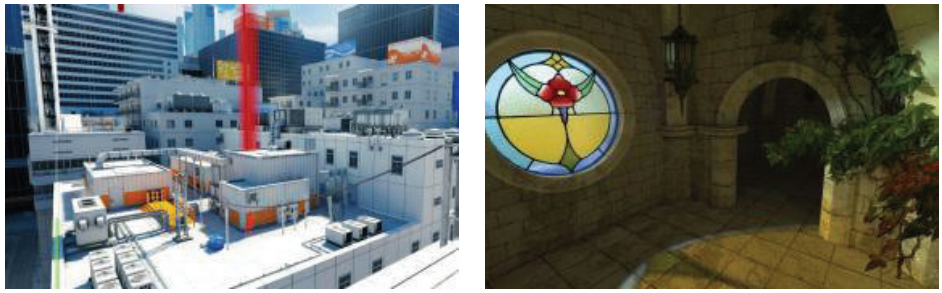
More recently, Novák et al. used GPU path tracing with path regeneration to double the performance of the path tracing algorithm on stream processors [24]. Their system is able to render interactive previews on commodity hardware, achieving 13 M rays per second on an NVidia GTX285 on moderately complex scenes, and is claimed to be "the first efficient (bidirectional) path tracer on the GPU." Van Antwerpen proposed a generic streaming approach for GPU path tracing algorithms and used this to implement three streaming GPU-only unbiased rendering algorithms: a path tracer, a bidirectional path tracer, and an energy redistribution path tracer [25].

Outside academia, several applications implement interactive path tracing. Examples are octane [26], smallpt [27], tokaspt [28], smallluxgpu [29], and nvidia's Design Garage demo [30].

3. Efficient GPU Path Tracing

The unbiased path tracing algorithm with russian roulette is shown in Algorithm 1. The algorithm aims to find a number of paths that connect the camera to light sources, via zero or more scene surfaces, by performing a random walk. The expected value of the average energy transported via these paths is the solution to the rendering equation [31]. To reduce the variance of the estimate, two extensions are commonly used. Russian Roulette is used to reduce the number of very long paths (which generally contribute little to the final image), and at each nonspecular surface interaction, direct light is explicitly sampled.

The path tracing algorithm can be efficiently implemented as on the GPU, using a single kernel per pixel. The kernel loops over the samples for a pixel and outputs the final color. This limits memory access to read-only scene



(a) (b)

Figure 3: Precalculated global illumination, calculated using Unreal technology. (a) Mirror's edge, lit by Beäst. (b) Scene lit by Lightmass.

(a) (b)

Figure 4: Ray tracing versus path tracing. (a) was rendered using the Arauna ray tracer [9], which supports direct illumination from point lights only. (b) uses path tracing for direct and indirect illumination of area light sources.

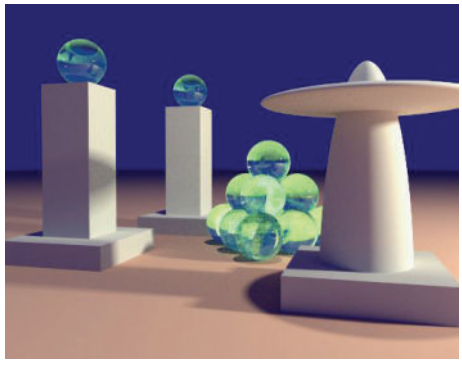


Figure 5: Scene from Kayija's paper, rendered using our CUDA path tracing algorithm.

access, and a single write for the final pixel color. CUDA code for this is provided in Appendix Section. Example output is shown in Figure 5. For this scene, ray/scene intersection uses a hardcoded scene consisting of axis aligned rectangular cuboids, spheres, and ellipsoids. Materials are limited to diffuse and dielectric. Using this setup, a single NVidia GTX580 achieves 750 M rays per second, which results in an almost converged image at real-time frame rates.

For more general scenes, we can replace the hardcoded ray/scene intersection by the BVH traversal code proposed by Aila and Laine [19].

4. The Brigade System

A renderer for games has specific requirements, which differ significantly from other applications. Of these, the requirement of real-time performance probably has the greatest overall impact on the design of a renderer. A modern game runs at 60 fps or more. For certain genres, a lower frame rate is acceptable. For the game Doom 4, a fixed frame rate of 30 fps is enforced by the renderer [32].

Frame rate translates to a strict millisecond budget, which must be divided over all subsystems. Note that if we chose to run the subsystems in order, the budget available to rendering decreases. If, on the other hand, we run the subsystems and rendering in parallel, we introduce input lag: in a worst-case scenario, user input that occurred at the beginning of frame N will be rendered in frame $N + 1$ and presented to the user just before frame $N + 2$ starts.

```
for each pass
  for each pixel
    c_rgb ← 0, scale_rgb ← 1
    hitDiffuse ← false
    D⃗,O ←ray_through_pixel()
    do
      // find material, distance and normal along ray
      m, I, N⃗ ← find nearest(O, D⃗)
      if (isempty(m))
        break //path left scene
      else if (is_light(m))
        if not hitDiffuse
          c_rgb ← c_rgb + scale_rgb * getEmissive(m)
        break // path hit light
      else
        O ← I
        if is_diffuse(m)
          c_rgb ←c_rgb + sampleDirect()
          hitDiffuse←true
        D⃗, scale_rgb ← evalBRDF(m, I,D⃗,N⃗)
        p ← RR(m)
        if rnd() < p) break // russian roulette
          scale_rgb ← scale_rgb * (1 − p)
    while (true)
    pixel[x, y] ← pixel[x, y] + c_rgb
  endfor
endfor
```

Algorithm 1: The path tracing algorithm with Russian roulette and explicit light sampling, in a format suitable for sequential execution. The final image is scaled by 1/passes.

Apart from real-time performance, rendering for games requires dynamic scenery. Scene elements may undergo complex movement due to physics as well as hand-crafted animations and procedural effects such as explosions. Contrary to popular belief, global changes to scenery are uncommon in games. Typically, large portions of the scenery are static, to avoid game states in which the user cannot progress.

Tightly coupled to the real-time requirement is the fact that games are interactive applications. The renderer must produce correct results for all possible user input and cannot predict any scenery changes that depend on user interaction.

On top of the generic requirements, there are requirements that evolve over time, most notably rendering resolution and scene complexity. At the time of writing, a typical game renders at a resolution of at least 1280 × 720 (HD 720). A typical scene consists of hundreds of thousands of polygons.

The Brigade rendering system is designed specifically for games and applies and encapsulates the technology of Section 3 in this context. Brigade renders scenes consisting of static and dynamic geometry, consisting of millions of triangles. It uses a fixed-function shading pipeline and supports diffuse and specular surfaces with textures and normal maps, as well as dielectrics with absorption. The animation system supports rigid animation and skinned meshes. Scenes are illuminated using emissive surfaces, of which an unlimited amount may be present.

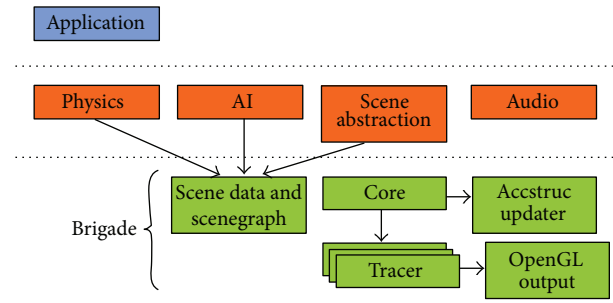


Figure 6: Functional overview of the Brigade renderer, combined with a generic game engine.

The rendering system achieves high performance by fully utilizing all compute devices in a heterogeneous architecture (Section 4.2). It implements a synchronization-free balancing scheme to divide the workload over these compute devices (Section 4.3). Adaptive converging (Section 4.5) and dynamic workload scaling (Section 4.7) are used to ensure a real-time frame rate at high-definition resolutions.

4.1. Functional Overview. Figure 6 provides a functional overview of the Brigade renderer. In a typical setup, Brigade is combined with a game engine that provides components not specific to the rendering algorithm, such as artificial intelligence and physics libraries. In terms of abstraction, the functionality provided by Brigade is thus similar to the functionality implemented by OpenGL and DirectX.

The main components of Brigade are as follows.

4.1.1. Scene Graph. The scene and hierarchical scene graph contain all data required for rendering. This includes the object hierarchy, mesh data, materials, textures, cameras, and lights. The object decomposition represented by the scene graph is used to steer acceleration structure construction, which makes the scene graph an essential data structure within the system. For convenience, the scene graph object implements keyframe and bone animation.

4.1.2. Core. The core implements the Render () method, initiates acceleration structure updates, synchronizes scene data changes with the compute devices, and divides work over the tracers, if there is more than one.

4.1.3. Acceleration Structure Updater. The acceleration structure updater maintains the BVH, by selectively rebuilding parts of the acceleration structure based on changes in the scene graph.

4.1.4. Tracers. A tracer is an abstract representation of a compute device or group of similar compute devices. A "compute device" in this context can be a GPU, the set of available CPU cores, or a compute device connected over a network. The tracer holds a copy of the scene data and the acceleration

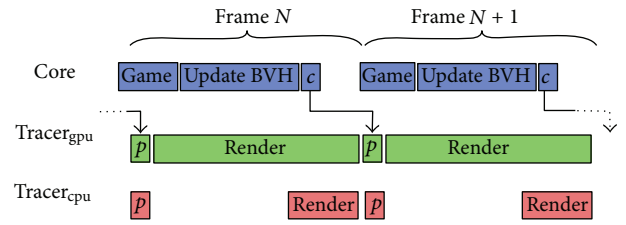


FIGURE 7: Double buffering the BVH. The CPU updates the BVH and sends changes to the tracers. Each tracer processes the changes in a commit buffer before rendering the next frame.

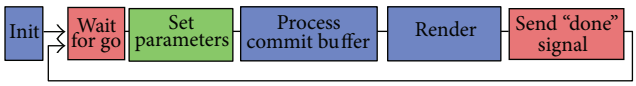


FIGURE 8: Tracer thread initialization and main loop.

structure and implements the path tracing algorithm with the next event estimation and multiple importance sampling. Tracers are assumed to produce identical output for identical input (except for nondeterministic aspects of the rendering algorithm).

The acceleration structure used by the tracers is the only cached data structure that is derived from scene data. All other data can be modified on-the-fly. This includes (all properties of) materials and lights.

In this system, the governing processes run on the CPU, and tracers (which in a typical setup primarily run on the GPUs) function as workers.

4.2. Rendering on a Heterogeneous System. A modern PC is a heterogeneous architecture, which typically consists of a CPU with multiple cores, and at least one GPU.

To efficiently use the available compute power, several options are available.

(1) The rendering algorithm is implemented completely on either the CPU or the GPU.

(2) The rendering algorithm is implemented on both the CPU and the GPU.

(3) Tasks are divided over CPU and GPU.

Each of these options has advantages and disadvantages. A renderer that runs entirely on the CPU or GPU may result in underutilization of the other compute device. An algorithm that is implemented on both the CPU and the GPU will use all resources but requires a greater implementation effort. Dividing tasks over CPU and GPU seems the most attractive option. This is, however, only efficient when CPU and GPU spend equal amounts of time on their assigned tasks.

A fourth option is to use a hybrid solution, where the CPU has specific tasks and uses the frame time that remains to assist the GPU. This is the approach implemented in our system. The CPU is responsible for game logic and acceleration structure maintenance, while the tracers perform the actual rendering. Assuming a CPU tracer is available, this system is able to keep compute devices fully occupied. The process is illustrated in Figure 7.

For each frame, the CPU updates the game state. The resulting changes to the scene graph are then used to update the BVH. The changes to the BVH, as well as any other scene changes, are sent to the tracers, where they are placed in a commit buffer, which the tracers use to iteratively update a local copy of the scene.

Parallel to these activities, the tracers render using the data that was prepared in the previous frame. A tracer starts a frame by processing the changes in the commit buffer, and then renders a part of the frame. CPU tracers are handled slightly differently than GPU tracers, by postponing rendering until the acceleration structure has been updated. This prevents rendering interferes with acceleration structure maintenance.

When no CPU tracer is available, the CPU can execute game code that does not affect the scene graph after copying scene changes to the commit buffers of the tracers.

4.3. Workload Balancing. The tracer flow is shown in Figure 8. Upon instantiation, the tracer spawns a thread that executes the worker loop. This loop waits for a signal from the core, renders a number of pixels, and signals the core, before going to sleep until the next frame.

When more than a single tracer is available, the core estimates an optimal workload division prior to rendering each frame. The advantage of this approach is that no communication between the tracers and the core is required once rendering has commenced, which greatly reduces communication overhead for GPU and network tracers. Dividing the work is nontrivial; however, not every compute device may have the same rendering capacity, and not every line of pixels has the same rendering cost (see Figure 10).

In a game, a typical camera moves in a somewhat smooth fashion. A good workload division for one frame will thus be at least reasonable for the next frame. We exploit this by adjusting the workload balance in an iterative manner.

We implemented four schemes to divide work over the tracers.

4.3.1. Do Not Balance. In this naive scheme, all workers are assigned an equal share of the screen pixels; no balancing is performed. This scheme is included for reference.

4.3.2. Robin Hood. This scheme starts with an equal distribution of the work for each tracer. After completing each frame, the tracer that finished last passes one work unit (one work unit equals four rows of pixels) to the tracer that finished first. When the work is poorly distributed, it may take a large number of frames to properly balance.

4.3.3. Perfect. Calculates the exact amount of work a tracer can handle based on the previous frame, but without considering differences in cost between lines of pixels. This may result in hickups, when many expensive lines are assigned to a tracer at once. The perfect balancer uses the following

Table 1: Average percentage of summed rendering time for all GPUs spent idling due to early completion, for the four balancing schemes, over 128 frames, for a slow and a faster moving camera. Measured for the Aztec scene.

	2 GPUs		3 GPUs	
	Slow	Fast	Slow	Fast
None	46.4	30.2	45.1	47.2
Robin Hood	2.1	8.2	4.9	20.7
Perfect	2.8	2.4	12.2	8.0
Perfect Smooth	1.4	3.4	2.8	6.2

formula to determine the workload for worker w for frame $i + 1$ based on the unit count and render time of frame i:

$$P_{w,i+1} = \frac{\text{units}_{w,i}\ \text{time}_{w,i}{}^{-1}}{\sum\left(\text{units}_i\ \text{time}_i{}^{-1}\right)}. \quad (1)$$

Perfect Smooth. Same as "Perfect", but this time, the workload per tracer is smoothed over multiple frames, using the following formula:

$$S_{w,i+1} = \alpha P_{w,i} + (1 - \alpha)\, S_{w,i}, \quad (2)$$

where $\alpha \in (0, 1)$.

Figure 9 shows the efficiency of the four schemes, for a spinning camera in the Aztec scene. For a slow moving camera, the workload in two subsequent frames is similar. All schemes except the overcompensating Perfect balancer work well. The Robin Hood balancer exhibits poor efficiency for the first frames. For a faster camera, Robin Hood is not able to keep up. For this situation, the aggressive Perfect balancer outperforms even the Perfect Smooth balancer. When more GPUs are used, Perfect Smooth is clearly the optimal scheme.

Table 1 shows the average efficiency of the four balancers over 128 frames, for a slow and a faster moving camera. This table confirms that the Perfect and Perfect Smooth schemes are similar in terms of average efficiency. The table does not, however, show the spikes that are visible in the graphs.

4.4. Double-Buffering Scene Data. For acceleration structure maintenance, we use the following assumptions.

(1) A game world may consist of millions of polygons.

(2) A small portion of these polygons is dynamic.

(3) Several tracers will use the same acceleration structure.

Based on these assumptions, a full rebuild of the BVH for each frame is neither required nor desired, as it would put a cap on maximum scene complexity, even when very few changes occur. We reuse the system described by Bikker [13], where each scene graph node has its own BVH, and a top-level BVH is constructed per frame over these BVHs. Each changed scene graph node is updated, using either full reconstruction or refitting.

Brigade uses a double-buffered approach for BVH maintenance. During a single frame, the CPU updates the BVH based on modifications of the scene graph. The resulting changes to the BVH are sent to the tracers, where they are placed in a commit buffer. At the start of the next frame, the commit buffer is processed, which results in an up-to-date BVH for each of the tracers. This process is illustrated in Figure 7.

Each frame is thus rendered using the BVH constructed during the previous frame. Acceleration maintenance construction thus only becomes a bottleneck when the time it requires exceeds the duration of a frame.

4.5. Converging. To reduce the noise in the final rendered image, several frames can be blended. Each pixel of the final image is calculated as $C_{\text{final}} = (1 - f)C_{\text{prev}} + fC_{\text{new}}$, where $f \in (0, 1]$. Value f is chosen either manually, or automatically, for example, based on camera speed. For stationary views, this approach results in a higher number of samples per pixel. For nonstationary views, this results in an incorrect image. The result can be improved by linking f to camera movement. For a stationary camera, a small value of f allows the renderer to blend many frames. For a moving camera, a value of f close to 1 minimizes ghosting.

Note that even though the camera may be static, objects in the scene may not be. It is therefore important to limit the minimum value of f to keep the ghosting for dynamic objects within acceptable bounds.

4.6. CPU Single Ray Queries. Brigade exposes a CPU-based synchronous single ray query that uses the BVH from the previous frame, to provide the game engine with a fast single-ray query. This query is useful for a number of typical game situations, such as line-of-sight queries for weapons and AI, collision queries for physics, and general object picking. The single-ray query uses the full detailed scene (rather than, e.g., a coarse collision mesh), including animated objects.

4.7. Dynamically Scaling Workload. Maintaining a sufficient frame rate is of paramount importance to a game application. In this subsection, we propose several approaches to scale the workload.

4.7.1. Adjusting Samples per Pixel. The relation between frames per second and samples per pixel is almost linear. Brigade adjusts the rendered number of samples per pixel when the frame rate drops below a specified minimum and increases this value when the frame rate exceeds a specified maximum.

4.7.2. Balancing Primary Rays and Secondary Rays. By balancing the ratio of primary and secondary rays, the quality of antialiasing and depth of field blurring can be traded for secondary effects. The primary rays are faster; increasing their ratio will also improve frame rate.

4.7.3. Scale Russian Roulette Termination Probability. Changing the termination probability of Russian roulette does not introduce bias, although it may increase variance [33]. Altering the termination probability affects the number of deeper path segments, and thus frame rate. Unlike the

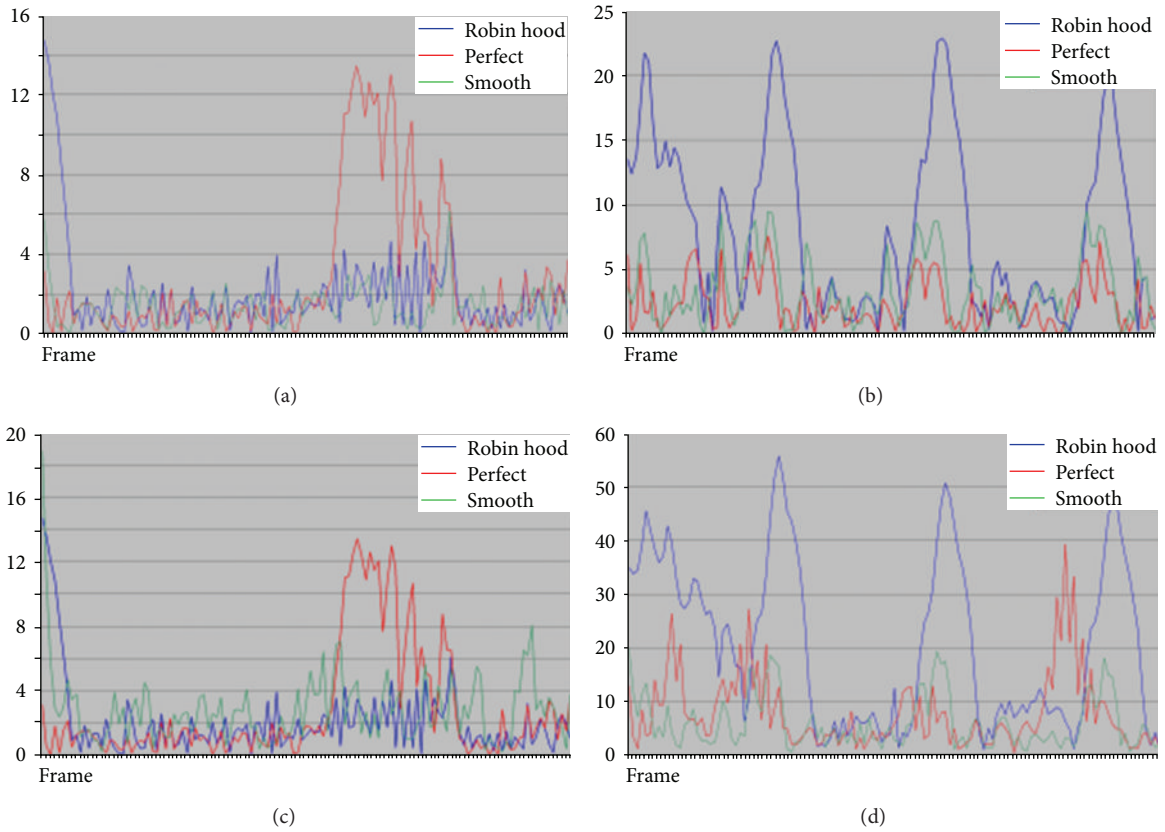


Figure 9: Efficiency of three workload balancing schemes, for two GPUs ((a)-(b)) and three GPUs ((c)-(d)), small camera movements ((a)–(c)) and larger camera movements ((b)–(d)). Values are percentages of rendering time spent idling due to early completion. Measured for the Aztec scene.

Figure 10: Render cost visualized: pixels representing the sky dome or light sources have a significant lower cost than other pixels. Cost is represented by greyscale values (brighter is higher cost), per 32 pixels (a full warp). Measured using a tracer implemented in NVidia's CUDA [11] for two scenes: Aztec (a) and Sibenik Cathedral (b).

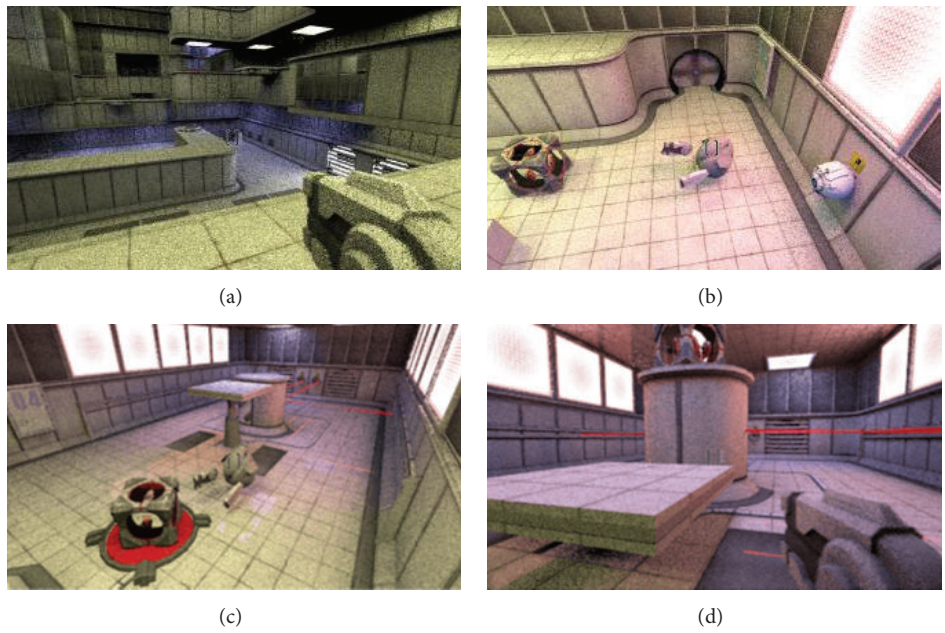
previous approach, scaling the termination probability using a factor which is based on frame rate does not distinguish between primary and secondary rays and allows smooth scaling of performance.

Alternatively, the workload can be reduced by reducing rendering resolution, or limiting trace depth. Limiting the maximum recursion depth of the path tracer introduces bias but also improves performance. In practice, due to Russian roulette, deep rays are rare, which limits the effect of a recursion depth cap on performance.

For game development, the scalability of a renderer based on path tracing is an attractive characteristic. A relatively slow system is able to run the path tracer at an acceptable frame rate, albeit perhaps not at an acceptable level of variance. Faster systems benefit from the additional performance by producing more samples per pixel, and thus a smoother image.

5. Discussion

The rendering system described in the previous section is relatively simple. To a large extend, this simplicity is the result of the chosen rendering algorithm. The path tracer does not rely on any precalculated data, which greatly reduces



(a) (b) (c) (d)

Figure 11: Two views from the "Reflect" game, rendered at 448 × 576 pixels using 16 spp, scaled up to 896 × 576.

data dependencies. There are two exceptions, and these are also the most complex parts of the system. The first is the acceleration structure, which is cached and updated iteratively, in a double-buffered fashion. As a result, games cannot make arbitrary changes to the scene graph. The second is the data synchronization between the renderer core and the tracers, which generally run on the GPU(s). Using a commit buffer system, Brigade makes this virtually invisible to the application, and few restrictions apply.

Apart from the tracers, Brigade is a platform-independent system. The tracers abstract away vendor-specific APIs for GPGPU and allow the implementation of networked tracers and CPU-based tracers. When using a CPU tracer, the system is able to achieve full system utilization, with little overhead.

6. Applied

To validate our results, we have applied the renderer to two-student game projects. Both games have been produced in approximately 14 working days.

6.1. Demo Project "Reflect". The Reflect game application is a student game that was developed using an early version of the Brigade engine. The game scenery is designed to simulate actual requirements for game development, and purposely mimics the graphical style of a well-known modern game (Portal 2 [34]).

The scenery has the following characteristics:

(i) scenery consists of approximately 250 k triangles, divided over multiple, separated rooms;

(ii) the scene is illuminated by thousands of area light sources, many of which are dynamic;

(iii) the game world is populated by dozens of dynamic objects.

Art assets for the game were created in Alias Wavefront Maya 2011 and were directly imported into the game.

Like Portal 2, Reflect is a puzzle game, where the player advances by activating triggers that in turn open doors or activate elevators. A "mirror gun" is available to the player to transform flat and curved wall sections into mirrors. These mirrors, as well as glass cube objects, can be used to divert lasers that block the way.

Configuration. Reflect was developed for a dual-CPU/dual-GPU machine (2 hexacore Intel Xeon processors, 2 NVidia GTX470 GPUs). We implemented a CPU tracer as well as a CUDA GPU tracer. For performance reasons, we limited the path tracers to a single diffuse bounce.

Game-Specific Optimizations. The scenery of the game consists of many rooms, separated by doors. A common optimization in rasterization-based renderers is to disable geometry that is known to be invisible. For a path tracer this does not significantly improve performance. We did find, however, that turning off lights in those rooms reduces variance, as the path tracer will no longer sample those light sources. This optimization is implemented at the application level: a system of triggers in the scene enables and disables sets of lights.

Performance and Variance. Figure 11 shows two scenes from the game running on a dual-CPU/dual GPU machine. At 16 spp, the game runs at 10–12 fps. At this sample count, brightly lit scenes are close to acceptable. Darker regions, such as the area under the platform in the right image, show significant temporal noise. Careful level layout helps to reduce objectionable noise levels. To the visual artist, this is

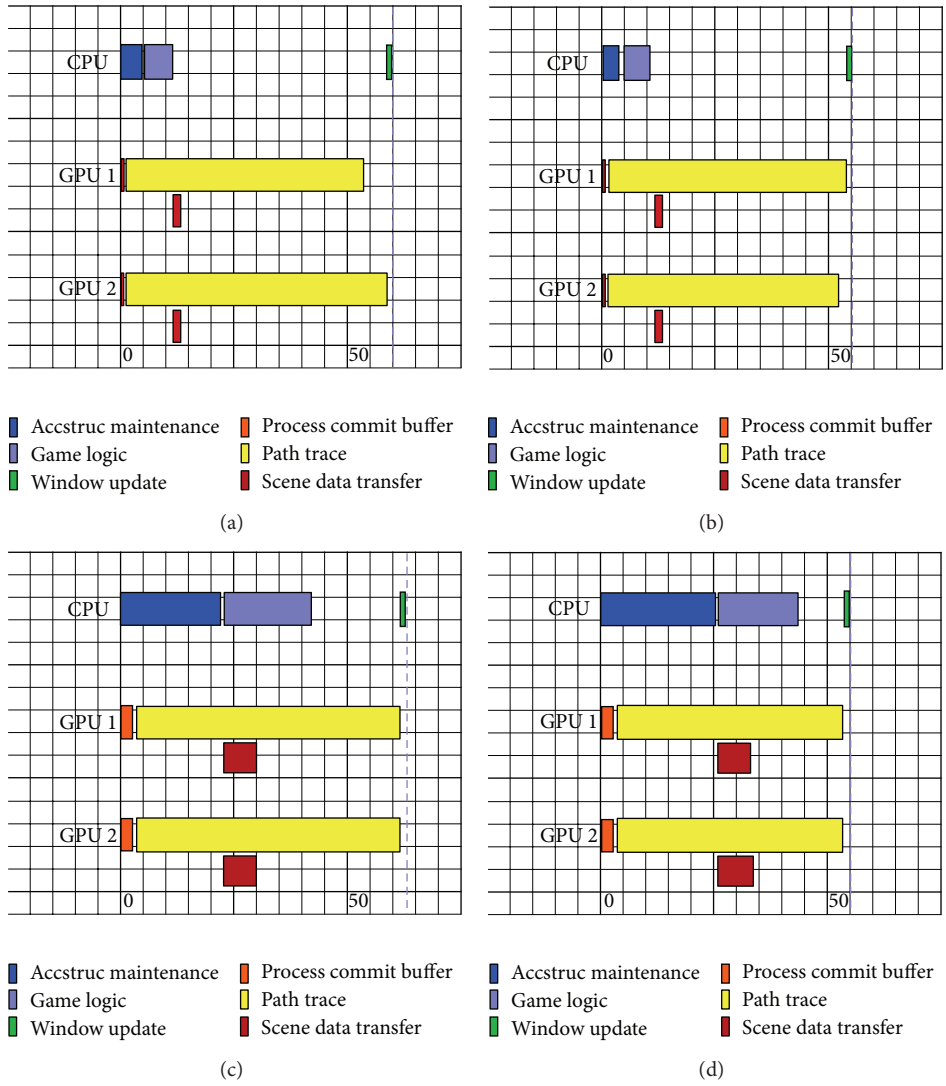


Figure 12: System utilization for the four views shown in Figure 13.

counter-intuitive, where rasterization-based renderers tend to use small amounts of point light sources, a path tracer benefits from large area lights, and incurs no slowdown when those lights are animated.

Materials in the levels are deliberately simple. Although specular surfaces are supported by this version of the renderer, specularity significantly increases noise, making this impractical in most situations.

Observations. The Reflect game struggles to achieve an acceptable frame rate, at a low resolution, on a high-end system. The project does, however, show the potential of path tracing for games. The art for this game was produced in Maya 2011 and was directly imported into the game, leading to very short development cycles, and usable art on the second day of the project. Within the same time span, the programmers implemented a basic physics engine using ray queries that allowed them to navigate the rooms.

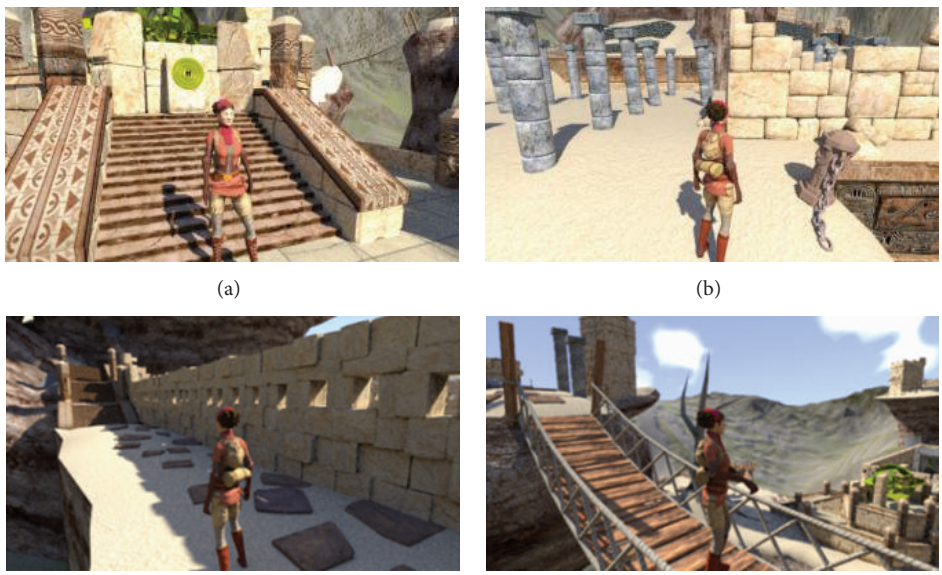
The freedom in lighting setup led to a final level that contains approximately 10 k light emitting polygons. Direct and indirect illumination simply works and results in subtle global illumination, both for static and dynamic objects.

The CPU tracer that was implemented for this project proved to be problematic: keeping the CPU and GPU tracers in sync required significant engineering effort, while the overall contribution of the CPU is quite small.

6.2. Demo Project "It's About Time". The student game "It's About Time" was created using a recent version of the Brigade renderer. Four views from the game are shown in Figure 13.

"It's About Time" is a third-person puzzle game that takes place in deserted Aztec ruins. The player must collect a number of artifacts by solving a number of puzzles, located in several areas in an open outdoor world.

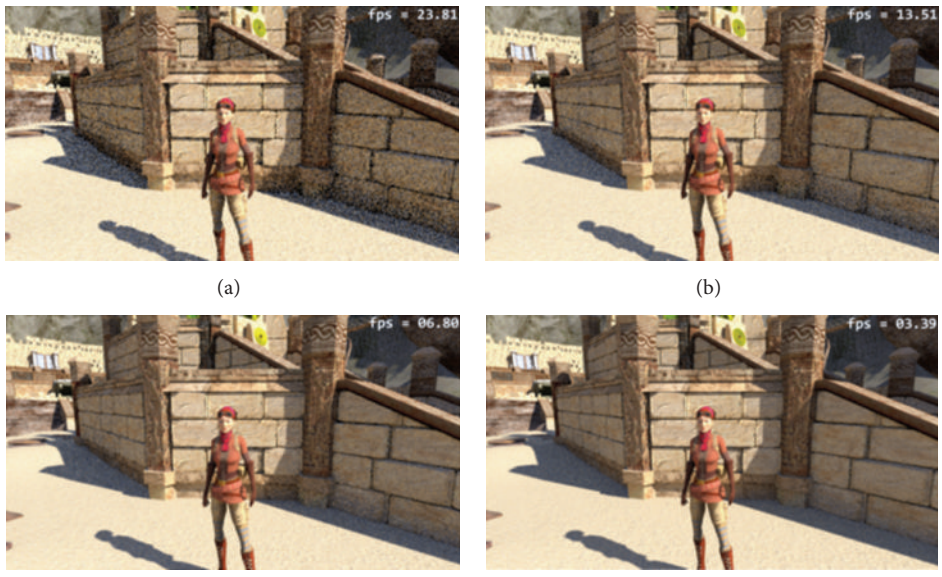
6.2.1. Configuration. "It's About Time" is designed to run on a typical high-end system, using a single hexacore CPU and one or more current-generation NVidia or AMD GPUs. The game renders to standard HD resolution. This resolution can



(a) (b)

(c) (d)

Figure 13: Four views from "It's About Time."



(a) (b)

(c) (d)

Figure 14: Noise level and performance at 2 spp, 4 spp, 8 spp, and 16 spp. Rendering resolution is 640 × 360. Measured on a system with a 6-core CPU and two NVidia GTX470 GPUs.

be halved to improve frame rate. We developed an updated CUDA tracer that roughly doubles the performance of the first iteration (as used in Reflect), as well as an OpenCL tracer, which produces identical images. A CPU tracer was not developed; the CPU is reserved for acceleration structure maintenance and game logic. The implemented path tracers are unbiased.

6.2.2. Project-Specific Features. One of the puzzles features an animated water surface that responds to the player, consisting of tens of thousands of polygons. For the player character, a detailed set of skinned animations is used. The puzzles make extensive use of rigid animation. As a result, acceleration structure maintenance requires considerable processing. A detailed day-night cycle and an animated cloud system (with shadowing) were implemented to emphasize the strength of the path tracer for dynamic lighting. A standard physics engine was integrated to provide high quality physics simulation. The level is detailed and consists of 1.4 M triangles. The artists used a small set of sand stones to construct most of the buildings and ruins.

6.2.3. Game-Specific Optimizations. The game world is illuminated by a sun (or the moon), and some smaller light sources. To reduce variance, we modified the path tracer to always sample two light sources per diffuse surface interaction. One of these rays always probes the primary light source. This significantly reduces variance in most areas. Adaptive converging is used to improve image quality when the camera is (almost) stationary. These application-specific optimizations where implemented in the GPU tracer code.

6.2.4. System Utilization. Figure 12 shows system utilization for the four views of Figure 13, rendered at 4 spp.

For the first two views, the CPU is underutilized, as both acceleration structure maintenance and game logic require little processing time. For the other two views, the camera is near a simulated water surface that consists of 18 k polygons. Both the simulation itself and the resulting acceleration structure maintenance require considerable processing time. This also affects the GPU tracers, which use more time to transfer and process the modified scene data.

6.2.5. Memory Use. The Brigade renderer is an in-core rendering system, which stores multiple copies of the scenery. The host system stores a full copy of the scene and synchronizes this data with each of the tracers.

For the 1.4 M triangle scene of "It's About Time," memory use is 737 MB, which consists of 175 MB triangle data, 42 MB for the acceleration structure, and 520 MB texture data. The size of other data structures is negligible, except for the commit buffer, which must be large enough to store per-frame changes to scene data and the acceleration structure. For "It's About Time," we used a 2 MB commit buffer.

6.2.6. Performance and Variance. Figure 14 shows a single scene from the game, rendered using varying sample counts. As in Reflect, areas that are directly illuminated converge quickly, while shadowed areas exhibit more noise. For the outdoor scenery of "It's About Time," an acceptable quality for most camera views is obtained with 8 or 16 spp. On a system with two NVidia GTX470 GPUs, we achieve 2 to 4 spp at real-time frame rates, at a quarter of 720p HD resolution (640×360). This lets us quantify the remaining performance gap: real-time frame rates at 720p require 8 to 16 times the achieved performance.

7. Conclusions and Future Work

We have investigated the feasibility of using physically based rendering in the context of real-time graphics for games. We implemented a renderer based on the path tracing algorithm, and used this to develop two proof-of-concept games. We have shown that real-time path tracing is feasible on current generation hardware, although careful light setup is required to keep variance levels acceptable.

The development of a game using path tracing for rendering simplifies game development. This affects both software engineering and art asset development. Since Brigade does not distinguish static and dynamic light sources and does not impose any limitations on the number or size of light sources, lighting design requires little knowledge beyond discipline-specific skills. The fact that polygon counts and material properties have only a small impact on rendering performance provides level designers and graphical artists with a high level of freedom in the design of the game. This reduces the number of iterations level art goes through, and allows a team to have game assets in the engine early on in the project.

Despite these positive experiences, real-time path tracing in commercial games is not feasible yet on current generation high-end hardware. Acceptable variance at HD resolution and real-time frame rates requires 8x to 16x the performance that can be achieved on our test system. Without further algorithmic improvements, this level may be reached in a few years. We do believe this can be accelerated. Already GPU ray tracing performance is benefiting from architectural improvements, on top of steady performance improvements. Another way to partially solve the rendering performance problem is to use cloud rendering, where dedicated servers are used for rendering images, which are then transferred over the internet to the client. At the time of writing, the Brigade system is being integrated into the OTOY cloud service, specifically for this purpose. The cloud rendering service will be made available to indie game developers in the near future and will allow them to use path tracing without the need of owning sufficiently powerful hardware.

Apart from raw performance, we should address the issue of variance. While low sample rates already result in reasonably converged images in our experiments, this will not be sufficient for more complex materials. Techniques like bidirectional path tracing (BDPT) and energy redistribution path tracing (ERPT) may solve this to some extent. However, not all of these techniques produce acceptable images at low sample rates; therefore, a minimum performance level is required before this can be considered for real-time graphics.

A temporary solution to the rendering performance problem is to use postprocessing on the path traced image. Although some work has been conducted in this area, it typically does not consider all the data that is available in a path tracer, which leaves room for improvement. Note that any form of postprocessing will introduce bias in the rendered image. For the intended purpose, this is, however, not an objection.

Appendix

Efficient CUDA implementation of the path tracing algorithm, using a single kernel per pixel (see Algorithm 2).

```
extern "C" __global__void TracePixelReference()
{
  // setup path
  int numRays = context.width * context.height;
  int idx0 = threadIdx.y + blockDim.y *
    (blockIdx.x + gridDim.x * blockIdx.y) +
    ((context.firstline * context.width) >> 5);
  int tx = threadIdx.x & 7, ty = threadIdx.x >> 3;
  int tilesperline = context.width >> 3;
  int xt = idx0 % tilesperline;
  int yt = idx0/tilesperline;
  int px = (xt << 3) + tx, py = (yt << 2) + ty;
  int pidx = numRays -1 -
    (px + py * context.width);
  RNG genrand(pidx, (clock() * pidx *
    8191) ^140167);
  int spp = context.SampleCount;
  float rcpw = 1.0f/context.width;
  float u = (float)px * rcpw -0.5f;
  floatv = (float)(py + (context.width -
    context.height) * 0.5f) * rcpw -0.5f;
  float3 E = make_float3(0, 0, 0);
  // trace path
  for(int sample = 0; sample < spp; sample++)
  {
    // construct primary ray
    float3 O, D;
    CreatePrimaryRay(O, D);
    // trace path
    float3 throughput = make_float3(1, 1, 1);
    int depth = 0;
    while (1)
    {
      int prim = 0;
      float2 BC, UV = make_float2(0, 0);
      float dist = 1000000;
      bool backfaced = false;
      intersect(O, D, dist, BC, prim, backfaced);
      O += D * dist;
      if (prim == -1)
      {
        E += throughput * GetSkySample(D);
        break;
      }
      Triangle& tri = context.Triangles[prim];
      TracerMaterial mat =
        context.Materials[tri.GetMaterialIdx()];
      if (mat.flags & Material::EMITTER) // light
      {
        E += throughput * mat.EmissiveColor;
        break;
      }
      else // diffuse reflection
      {
        float3 matcol = tri.GetMaterialColor(
          mat, BC, UV);
        float3 N = tri.GetNormal(mat, BC, UV) *
          (backfaced ? -1: 1);
```

Algorithm 2: Continued.

```
                D = normalize(RandomReflection(
                    genrand, N) );
              throughput *= matcol * dot(D, N);
            }
            O += D * EPSILON;
            depth++;
            if (depth > 3)
            {
              if (genrand() > 0.5f) break;
              throughput *= 2.0f;
            }
          }
        }
        context.RenderTarget[pidx] =
          make_float4(E/(float)spp, 1);
      }
```

Algorithm 2: Path tracing implemented in CUDA.

Acknowledgments

Several scenes used in this paper were modeled by students of the IGAD program of the NHTV Breda University of Applied Sciences. The original of the scene shown in scene 4 was modeled by Jim Kajiya. The author wishes to thank Erik Jansen and the anonymous reviewers for proofreading and useful suggestions.

References

[1] S. Miyamoto, *Super Mario Galaxy*, Nintendo, 2007.

[2] H. Kamiya and A. O. Inaba, "Okami," 2006.

[3] K. Yamauchi, Gran Tourismo Series, 1997.

[4] T. Simon, Tom Clancy's H.A.W.X., 2009.

[5] T. Sweeny, Unreal Engine 3, 2008.

[6] J. Andersson, *Frostbite 2 Engine*, Nvidia, 2011.

[7] M. Mittring, *The Technology behind the "Unreal Engine 4 Elemental Demo"*, Epic Games, Inc., 2012.

[8] C. Crassin, F. Neyret, M. Sainz, S. Green, and E. Eisemann, "Interactive indirect illumination using voxel-based cone tracing: an insight," in *Proceedings of the ACM International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '11)*, Vancouver, Canada, August 2011.

[9] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, 1965.

[10] J. Owens, "Streaming architectures and technology trends," in *GPU Gems 2*, Addison-Wesley, 2005.

[11] NVidia, *Fermi: NVidia's Next Generation CUDA Compute Architecture*, NVidia, Santa Clara, Calif, USA, 2009.

[12] I. Wald and P. Slusallek, "State of the art in interactive ray tracing," in *State of the Art Reports, Eurographics*, pp. 21–42, Manchester, UK, 2001.

[13] J. Bikker, "Real-time ray tracing through the eyes of a game developer," in *Proceedings of the IEEE Symposium on Interactive Ray Tracing (RT '07)*, pp. 1–10, IEEE Computer Society, Ulm, Germany, September 2007.

[14] S. Boulos, D. Edwards, J. D. Lacewell et al., "Packet-based whitted and distribution ray tracing," in *Proceedings of the Graphics Interface (GI '07)*, pp. 177–184, ACM, Montreal, Canada, 2007.

[15] R. Overbeck, R. Ramamoorthi, and W. R. Mark, "Large ray packets for real-time whitted ray tracing," in *Proceedings of the IEEE/EG Symposium on Interactive Ray Tracing (RT '08)*, pp. 41–48, August 2008.

[16] T. J. Purcell, I. Buck, W. R. Mark, and P. Hanrahan, "Ray tracing on programmable graphics hardware," *ACM Transactions on Graphics*, vol. 21, no. 3, pp. 703–712, 2002.

[17] T. Foley and J. Sugerman, "KD-tree acceleration structures for a GPU raytracer," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '05)*, pp. 15–22, ACM, Los Angeles, Calif, USA, 2005.

[18] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-D tree GPU raytracing," in *Proceedings of the Symposium on Interactive 3D Graphics and Games (I3D '07)*, pp. 167–174, ACM, Seattle, Wash, USA, 2007.

[19] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on GPUs," in *Proceedings of the Conference on High-Performance Graphics (HPG '09)*, pp. 145–150, ACM, New Orleans, La, USA, August 2009.

[20] C. Benthin, *Realtime ray tracing on current CPU architectures [Ph.D. thesis]*, Saarland University, Saarbrücken, Germany, 2006.

[21] J. Sugerman, T. Foley, S. Yoshioka, and P. Hanrahan, "Ray tracing on a cell processor with software caching," in *Proceedings of the IEEE Symposium on Interactive Ray Tracing (RT '06)*, September 2006.

[22] B. Walter, G. Drettakis, and S. Parker, "Interactive rendering using the render cache," in *Rendering Techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering)*, D. Lischinski and G. W. Larson, Eds., vol. 10, pp. 235–246, Springer, New York, NY, USA, 1999.

[23] I. Sadeghi, B. Chen, and H. W. Jensen, "Coherent path tracing," *Journal of Graphics, GPU, and Game Tools*, vol. 14, no. 2, pp. 33–43, 2009.

[24] J. Novák, V. Havran, and C. Daschbacher, "Path regeneration for interactive path tracing," in *The European Association for Computer Graphics 28th Annual Conference: EUROGRAPHICS 2007, Short Papers*, pp. 61–64, The European Association for Computer Graphics, 2010.

[25] D. Van Antwerpen, *Unbiased physically based rendering on the GPU [M.S. thesis]*, Technical University Delft, Delft, The Netherlands, 2011.

[26] Refractive, Octane Renderer, 2010, http://www.refractive-software.com/.

[27] K. Beason, SmallPT, 2007, http://www.kevinbeason.com/.

[28] T. Berger-Perrin, "The Once Known As SmallPT," 2009, http://code.google.com/p/tokaspt/.

[29] Jromang, SmallLuxGPU, 2009, http://www.luxrender.net/wiki/SLG.

[30] NVidia, Design Garage, 2010, http://www.nvidia.com/.

[31] J. T. Kajiya, "The rendering equation," in *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '86)*, pp. 143–150, ACM, Dallas, Tex, USA, 1986.

[32] T. Bramwell, "Doom 4 "three times" Rage visual quality," 2011, http://www.eurogamer.net/articles/doom-4-three-times-rage-visual-quality.

[33] L. Szirmay-Kalos, G. Antal, and M. Sbert, "Go with the winners strategy in path tracing," in *Proceedings of the International Conference in Central Europe on Computer Graphics and Visualization (WSCG '05)*, pp. 49–56, 2005.

[34] G. Newell and J. Weier, Portal 2. Valve Corporation, 2011.