Global Illumination Effects
Final Report

Tom Flanagan
CPSC 502
University of Calgary
thomas.m.flanagan@ucalgary.ca

ABSTRACT

Current advances in GPU hardware architecture and processing power in recent years have allowed more and more general purpose techniques to be used in real time graphics applications, where previously only fixed-function programs could be written. These advances have allowed developers the freedom to implement more photorealistic effects, by bypassing or augmenting the standard graphics pipeline. It also allows the development of algorithms for the GPU (Graphics Processing Unit) that could previously only be run off-line on the CPU. This report presents the results of investigating and developing a photon mapping and ray tracing algorithm for photorealistic effects and global illumination lighting that was previously restricted to offline CPU-based algorithms, now using modern GPU technologies.

Categories and Subject Descriptors

I.3.7 [**Three-Dimensional Graphics and Realism**] Color, shading, shadowing, and texture, Raytracing.

General Terms

Algorithms, Measurement, Performance, Experimentation, Verification.

Keywords

Global Illumination, Photon Mapping, GPGPU, KD-Tree, Real-Time, Post-processing.

1. Introduction/Motivation

Photorealistic rendering is the computer graphics field of producing images that look as realistic as possible, as if taken from a camera or perceived by the human visual system. Rendering techniques simulate the way light propagates within the environment and interact with different objects and materials. Some of these features include global illumination, diffuse interreflection, colour bleeding, soft shadows, caustics, and imaging artifacts of real-world cameras (e.g. depth of field, exposure control, motion blur, chromatic aberration, lens flare, bloom, and film grain) (Figure 1).

Figure 1: Examples of computer-generated global illumination effects

Many algorithms for solving different photorealistic features exist, but most, especially global illumination algorithms, are very computationally expensive and can only be used in offline rendering situations. A fundamental problem in computer graphics research is to create more accurate photorealistic images, and to be able to produce those images in real-time [29], [9].

Realistic images are in high demand from different industry domains in entertainment (films, games), science and engineering, but often many parts of these the photorealistic images used in such applications take long times to render and must be done off-line. Due to the recent improvement in programmable graphics hardware opportunities exist to develop algorithms for the GPU that may be able to render more photorealistic effects in real-time.

2. Problem Description

In traditional real time graphics applications, the graphics hardware has a fixed pipeline which consists of applying transformations to vertices, rasterizing polygons to pixels, and shading pixels to apply lighting and textures. The algorithms of this pipeline were fixed, and could not be modified beyond its inputs and parameters. This restriction allowed the algorithm to be implemented in hardware, for performance reasons to allow real-time operation.

Vertex and fragment shaders now allow software programs to be written that can replace the fixed vertex transformation process and the pixel shading/lighting algorithms, but the rasterization process is still fixed.

General-Purpose GPU (GPGPU) technologies like nVidia's Compute Unified Device Architecture (CUDA) and the Open Computing Language (OpenCL) allow general purpose programs to be written for the GPU as a massively parallel processor.

2.1 Background and Definitions

This section describes the key rendering effects related to the project.

Reflection and **refraction** are effects when the camera ray strikes an object, but instead of seeing that object, the material redirects the camera ray to some other part of the scene, and you see that instead. This is commonly seen in such things as mirrors, glass, crystal, and water.

Depth of field is an effect caused by lenses with a nonzero aperture in which only objects of a certain distance are in focus, and everything else is blurred. Motion blur is similarly caused when a picture is exposed for a nonzero time, and light from a moving object will be blurred across its motion.

High Dynamic Range (HDR) refers to the fact that human eyes can see a very large range of light intensities, but computer displays can only output a limited range of brightness's. Tone mapping and bloom are examples of effects that simulate having a high dynamic range image on a low dynamic range device.

Global illumination is a group of algorithms that try to simulate a more realistic lighting environment by taking into account visibility and light reflections from all surfaces in the scene. This is in contrast to direct illumination, which only accounts for the distance and direction of the shading point to the light source itself, and does not consider other objects in the scene. Global illumination can be further broken down into three fundamental components and effects: **(1) Diffuse interreflection** accounts for the reflection of light from one diffuse surface to another, possibly over multiple bounces. **(2) Caustics** account for the distribution of light after it reflects or refracts from a specular surface onto a diffuse surface, such as light through a magnifying glass, bounced off of a mirror, or refracted light at the bottom of a pool. **(3) Colour bleeding** accounts for the fact that the properties of the light itself may change after reflecting or refracting off of a surface.

2.2 Rendering Techniques

There are many techniques that are used in graphics algorithms to simulate global illumination, and photorealistic effects in general.

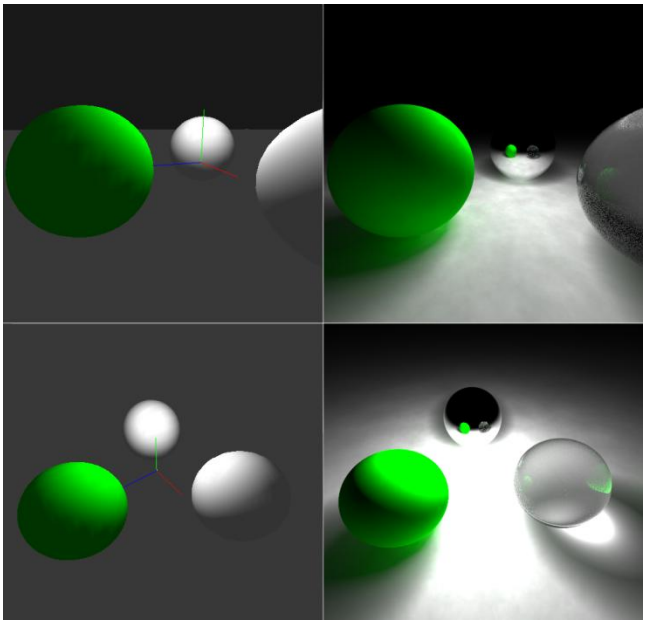

Figure 2. Direct illumination (left) compared to global illumination (right)

2.2.1 Ray Tracing

Ray tracing, ray casting, and path tracing are alternative algorithms from rasterization that use ray-object intersection tests to compute visibility of objects, usually with rays originating from the camera (Dutré et al, 2006). These algorithms are much more robust than rasterization, as they can handle reflection and refraction by recursively tracing new rays from the first intersection point, in a direction determined by the type of material it hit. Ray tracers produce images by casting one or more rays out from the camera point in the direction of each pixel on the screen. Shadow rays can be cast from a shading point to a light to determine if that point is in shadow or not. Distributed ray tracers shoot multiple rays per pixel, each with different properties (i.e. time, location on camera lens) to simulate effects such as motion blur or depth of field.

KD tree, BSP trees, bounding volume hierarchies (BVHs), oct trees, and uniform grids are all divide-and-conquer algorithms that spatially subdivide a set of points or polygons for efficient searching for items in nearby space (Akenine-Möller et al, 2008). These algorithms are commonly used to speed up ray-object intersection tests, and in photon mapping for looking up nearby photons.

2.2.2 Photon Mapping

Photon mapping [17] is a two-pass algorithm where in the first pass photons are recursively ray traced from the light sources into the scene, recording their intersections with scene objects into a spatial data structure (usually a KD-tree). In the second pass, rays are traced from the camera as in standard ray tracing, but to find the colour at each pixel, the lighting is computed by looking up nearby photons at that location and using their recorded incoming light direction, colour, and density to compute the shading for that pixel. Photon mapping is a very robust algorithm that can handle

both diffuse interreflection and caustics well. Photon mapping has two important extensions: final gathering and irradiance caching.

Final gathering [16] is an extension to photon mapping where instead of looking up nearby photons on the first intersection from the camera ray, multiple secondary rays are cast on a hemisphere from the intersection point to find what other object contribute to the lighting. Photons are then looked up at the intersections of those rays, and their contribution to the final pixel colour is calculated. Final gathering allows fewer photons to be traced for an equivalent quality image, and trades off low-frequency variance in the photon map density for high-frequency noise. Note that final gathering cannot be used for caustic lighting lookups, because of the very low probability of a random ray on the hemisphere connecting to a light source through a specular material.

Irradiance caching [16] is another extension to photon mapping where the final lookup of nearby photons is cached at gather points, and then nearby cached gather points are interpolated for future queries instead or performing a full photon lookup.

2.2.3 Radiosity

Radiosity is an algorithm that divides the scene into patches of surfaces, and then computes the visibility between all pairs of patches. It uses these visibilities to compute the brightness of each patch by starting from the light source, and iteratively applying the brightness of one patch to all its visible neighbours. This algorithm works very well for finding diffuse interreflection, but cannot handle effects where the viewing angle is an important parameter in the lighting, as specular reflection, refraction, or caustics.

2.2.4 Other Strategies

Post-processing algorithms take as input a rendered image, and apply kernels or programs to each pixel. Common post processing algorithms are exposure control, HDR, bloom, depth of field, and motion blur.

Depth of field and motion blur can created directly via distributed ray tracing using multiple rays for different locations on a lens's aperture or in time, or it can be simulated using post-processing effects that blur the final image depending on the depth or velocity of each pixel. Tone mapping simulates HDR images by changing the intensity of a pixel to be relative to its local neighbours to improve contrast, and bloom makes bright areas appear more intense by having light leak into nearby areas.

2.3 Related Work

Much work has been done in graphics algorithms since the introduction of shader programs and more general purpose frameworks like CUDA and OpenCL.

[10] presents a method of choosing final gather rays of all ray traced pixels in a uniform manner, which enables all final gather rays of the same direction to be clustered and rendered using the GPU's rasterization hardware in a single pass, due to the coherent nature of the rays. This can compute the final gathering of an arbitrary number of visible points in a fixed number of passes on the GPU. [11] has implemented this in the Parthenon Renderer project. [26] present a similar algorithm for using the rasterization hardware on the first (from light) and last (into camera) segments of photon paths. [23] also use rasterization for the final gather portion of their algorithm. This technique has been used in Radiosity algorithms for calculating patch-patch visibilities for some time ([7], , [32]).

[12] presents a way of adaptively sampling camera rays in a distributed ray tracing algorithm to achieve high quality images using less rays. Their algorithm uses a KD tree to store camera rays based on their initial parameters, and uses this tree to find nodes of greatest variance where more camera rays will be added to improve quality. This results in much higher image quality than using the same number of rays in a uniform manner because the rays are cast where they will have the greatest effect on the final image, and can work in an arbitrary number of dimensions, so can also reduce temporal aliasing in motion blur, or artifacts in depth of field or soft shadowing effects.

[15] and [14] change the regular order of photon mapping passes to trace camera rays first, and then iteratively trace photons in many secondary passes, each increasing the accuracy of the image by refining the radiance estimate at ray trace points. This reduces the memory requirements of photon mapping in that photons from previous passes need not be stored. In [13] they improve this algorithm by adding a distributed ray tracing pass after each photon mapping pass to enable the algorithm to handle effects such as motion blur or depth of field, and implement it using fragment shaders on the GPU. Hachisuka demonstrates this in his Stochastic Progressive Photon Mapping project.

[19] and [42] propose algorithms that can create spatial subdivision trees entirely in GPGPU programs that are faster then the best CPU algorithms. Javor proposes a uniform grid structure, and Zhou creates KD-trees. [40] uses this algorithm to compute photon mapping and real-time ray tracing entirely in the GPU using irradiance caching.

[35] develops a CPU-based real-time ray tracer called OpenRT for his PhD thesis, and briefly discusses GPU implementations and global illumination. Aruana [5] is another implementation of a high-performance CPU-based ray tracer.

Recently, attempts are being made to port smallPT [3] and LuxRender (a derivative of pbrt, [29]) to use GPGPU algorithms (SmallptGPU, SmallLuxGPU,[6]) to compute the path tracing components using the OpenCL GPGPU environment.

The RenderCache [37] uses image-space caching and re-projection of shaded pixels to reduce the shading costs for proceeding frames of images, based on the fact that sequential frames will likely be very similar, and can reduce the shading cost of new frames by 10 to 100 times while preserving image quality.

[34] improve this algorithm to run on vertex and pixel shaders on the GPU.

3. Solution to Problem

This project proposes using a modified photon mapping algorithm to compute global illumination in a real-time graphics application. This algorithm must be able to handle large scenes and dynamic geometry. Due to time constraints and challenges learning the CUDA architecture, this project was scaled back from its previous goals of implementing a progressive algorithm that could cope with dynamic geometry, and will only implement photon mapping for static scenes.

3.1 Implementation Framework

This project was implemented in a combination of Python (using the PyCUDA, pyglet, and PIL libraries) for the main application, nVidia CUDA for the ray tracing and photon mapping algorithms, and C++ for the KD tree building algorithms. It was implemented on Windows 7, using a computer with an Intel Core2 Quad Q8200 CPU and an nVidia GeForce GTX 275 GPU. The implementation should however be cross platform and run on any CUDA Compute 2.3 capable device.

3.2 Overview

Python was used for the main application, and manages the GUI, loading scene files, and calling all of the other modules. First, the application loads and parses the scene files containing the geometry and materials. The application also loads the scene Surface Area Heuristic (SAH) KD tree which is created offline by a separate tool. Using a CUDA algorithm in the GPU, photons are traced through the scene and their hits recorded. The hits are then compiled into a Voxel Volume Heuristic (VVH) KD tree using a CPU algorithm in C++. The main loop of the program consists of shooting camera rays (one per screen pixel) in a CUDA algorithm, using the geometry KD tree to find candidate triangles, and the Moller triangle intersection test [27] to find the nearest intersection triangle. The ray hits are then passed to another CUDA program that uses the photon KD tree to look up the k-nearest photons to the hit position, using a max-heap to keep track of the photon distances. The pixel's color is then computed as the sum of the bidirectional reflectance distribution function (BRDF) of each of the photons, divided by the area of the circle that contains them. The final image is then post-processed (currently, this is just gamma correction) and displayed.

The **Surface Area Heuristic** [25] Proposes keeping a geometry KD tree intentionally unbalanced because the probability of a ray entering nodes in the trees is not equal for all nodes. The cost C(V) of a node V is estimated as:

$$C(V) = C_{trav} + P(V_L)C(V_L) + P(V_R)C(V_R)$$

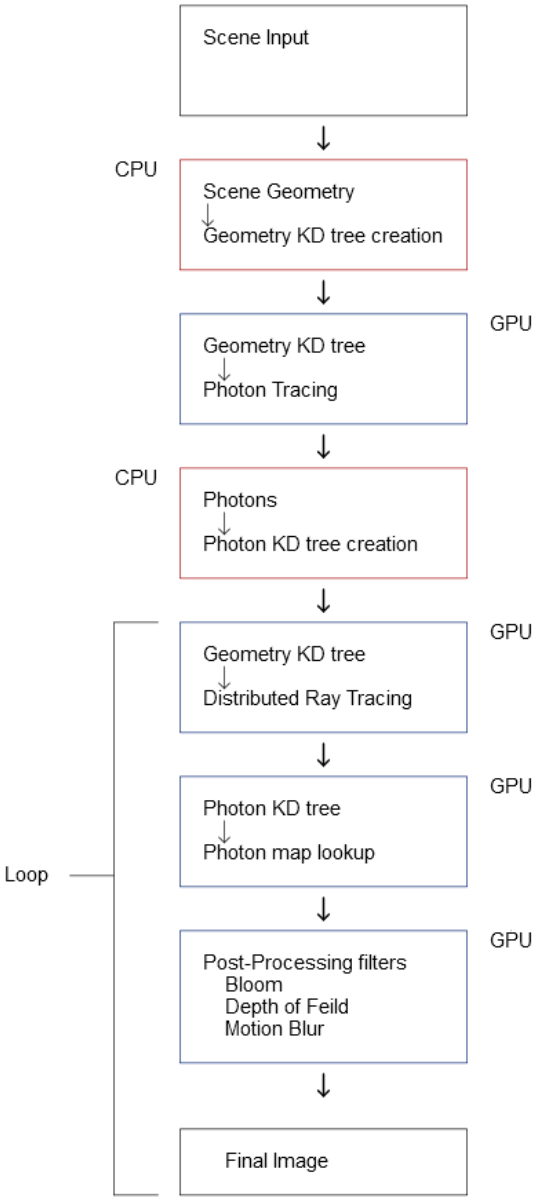

Figure 3: Application components

where C_{trav} is the cost of traversing a node, and $P(V_L)$, $P(V_R)$ is the probability that a ray will intersect with the left and right child, respectively. $P(V_L)$ and $P(V_R)$ are computed as

$$P(V_L) = \frac{SAH(V_L)}{SAH(V)} \quad P(V_R) = \frac{SAH(V_R)}{SAH(V)}$$

Where $SAH(V)$ is the surface area of a voxel.

Thus the minimum cost of a node can be found by evaluating all possible splitting planes, and choosing the one with lowest cost.

The **Voxel Volume Heuristic** [36] Proposes a similar measure for photon KD trees, but weighted by the volume of the node instead of the surface area.

$$P(V_L) = \frac{VOL(V_L \pm R)}{VOL(V \pm R)} \quad P(V_R) = \frac{VOL(V_R \pm R)}{VOL(V \pm R)}$$

Where R is the maximum radius of a photon lookup.

4. Results

Evaluation was done primarily on two scenes, the Sponza Atrium model (Fig. 5), and the Conference Room model (Fig. 6). The Sponza Atrium has 76,154 triangles while the Conference room has almost four times more geometry with 282,655 triangles. In both scenes, 262,144 initial photons were shot from an area light source in the sky, with a maximum of 4 bounces each. approximately 675,000 total photon hits were recorded in the photon map (photons were shot randomly, and are sometimes absorbed or leave the scene without hitting anything). All images were rendered at 512x512 pixels.

Figure 4: Left: Sponza Atrium, Right: Conference Room

4.1 Sample Images

Some images of the Sponza Atrium using the 32-nearest photons for lighting. Notice diffuse interreflection (all ears not in direct sunlight), soft shadows around columns, and colour bleeding of the brown ceiling onto the floor. Also note the artifacts created by the photon mapping algorithm, such as low-frequency noise in areas that are evenly lit (walls in all images), and where the photon mapping algorithm considers photons that are nearby the shading point, but should not contribute to the lighting (light circles on floor around chair legs).

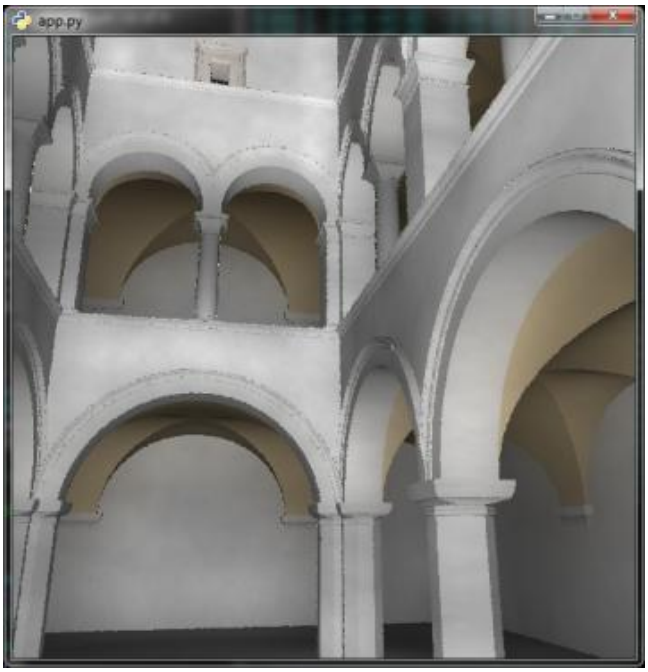

Figure 5: Sponza Atrium

Figure 6: Conference Room

4.2 Performance

This algorithm is was able to achieve real time performance on both scenes, with the rendering time mostly dependant on the number of photons looked up per pixel.

Operation	*Time (ms)*	
	Sponza Atrium	Conf-erence Room
Ray tracing	26	26
Ray tracing + 2 nearest photon lookup	33	50
Ray tracing + 4 nearest photon lookup	36	54
Ray tracing + 8 nearest photon lookup	47	68
Ray tracing + 16 nearest photon lookup	67	107
Ray tracing + 32 nearest photon lookup	121	197
Ray tracing + 64 nearest photon lookup	264	399
Ray tracing + 128 nearest photon lookup	634	856
Ray tracing + 256 nearest photon lookup	1541	1977

Note that ray tracing performance is equal for both scenes, even though the conference room scene has 4 times more geometry. Performance is mainly affected by the number of photon samples required per shading point. One reason for this may be that less time was spent optimizing the photon lookup algorithm and its KD tree, and the photons are being stored as AoS instead of SoA, so memory bandwidth on the GPU may be the limiting factor.

Figure 7: four, 32, and 256 photon samples per pixel, showing variance in lighting due to random photons

5. Conclusion

Advances in hardware technology present an excellent opportunity for creating high-quality graphics applications. Realistic computer generated images are in high demand, and there has been much interest in the research community in the past years in the topic of real-time applications.

This project demonstrates that it is feasible to create ray traced and photon mapped images in real time using current graphics hardware.

6. Acknowledgments

Thanks to Bartosz Fabianowski for assisting with CUDA programming and real time photon mapping concepts, and for providing test scenes.

7. Bibliography

[1] Aila, T., Laine., S., Understanding the efficiency of ray traversal on GPUs , High-Performance Graphics 2009.

[2] Akenine-MöLler, T., Haines, E., Hoffman, N., Real-time rendering, third edition, A.K. Peters Ltd. (2008)

[3] Beason, K., Smallpt: global illumination in 99 lines of c++, http://kevinbeason.com/smallpt/, accessed Jan 13 2010

[4] Bikker, J., Real-time ray tracing through the eyes of a game developer , Interactive Ray Tracing, 2007

[5] Bikker, J., Aruana: realtime ray tracing, http://igad.nhtv.nl/~bikker/, accessed Jan 17, 2010

[6] Bucciarelli, D., Smallptgpu and smallluxgpu, http://davibu.interfree.it/, accessed Jan 17, 2010

[7] Cohen, M. F., Greenberg, D. P., The hemicube: a radiosity solution for complex environments, In Proceedings of Siggraph (1985), pp. 31–40.

[8] Dingliana, B. F. A. J., Interactive global photon mapping, Eurographics Symposium on Rendering 2009

[9] Dutré, P., Bala, K., Bekaert, P., Advanced global illumination, second edition, A K Peters Ltd. (2006)

[10] Hachisuka, T., High-quality global illumination rendering using rasterization, GPU Gems 2: Programming Techniques for High Performance Graphics and General-Purpose Computation, 2005

[11] Hachisuka, T., Final gathering on gpu, GP2, ACM Workshop on General Purpose Computing on Graphics Processors, Los Angeles, August 2004

[12] Hachisuka, T., Jarosz, W., Weistroffer, R. P., Dale, K., Humphreys, G., Zwicker, M., Jensen, H. W., Multidimensional adaptive sampling and reconstruction for ray tracing, ACM Transactions on Graphics (SIGGRAPH 2008), 2008

[13] Hachisuka, T., Jensen, H. W., Stochastic progressive photon mapping, ACM Transactions on Graphics (SIGGRAPH Asia 2009), 2009

[14] Hachisuka, T., Ogaki, S., Jensen, H. W., Progressive photon mapping, ACM Transactions on Graphics (SIGGRAPH Asia 2008), 2008

[15] Havran, V., Herzog, R., Seidel, H.-P. 2005., Fast final gathering via reverse photon mapping, Computer Graphics Forum (Proceedings of Eurographics 2005) 24, 3 (September).

[16] Jensen, H. W., Realistic image synthesis using photon mapping, A. K. Peters, Ltd. (2001)

[17] Jensen, H. W., Global illumination using photon maps, Rendering Techniques '96. Eds. X. Pueyo and P. Schröder. Springer-Verlag, pages 21-30, 1996

[18] Jozwowski, T.R. 2002., Real time photon mapping, MSc, Michigan Technological University

[19] Kalojanov, J., Slusallek, P., A parallel algorithm for construction of uniform grids, High Performance Graphics 2009, New Orleans

[20] Karlsson, F., Ljungstedt, C. J., Ray tracing fully implemented on programmable graphics hardware, Master's Thesis, Chalmers University of Technology, Göteborg 2004

[21] Krivanek, J., Gautron, P., Radiance caching for efficient global illumination computation, IEEE Trans. Visualization and Computer Graphics 11(5):550–561 (2005)

[22] Laine, S., Saransaari, H., Kontkanen, J., Lehtinen, J., Aila, T., Incremental instant radiosity for real-time indirect illumination, Eurographics Symposium on Rendering 2007

[23] Larson, B. D., Christensen, N., Simulating photon mapping for real-time applications, In Proc. Eurographics pp123-131 (2004)

[24] Luxrender, , Gpl physically based renderer, http://www.luxrender.net/, accessed Jan 17, 2010

[25] Macdonald, J. D., Booth, K. S., Heuristics for ray tracing using space subdivision., Visual Computer 6, 6 (1990), 153–65. 3

[26] Mcguire, M., Luebke, D., Hardware-accelerated global illumination by image space photon mapping, In Proceedings of the Conference on High Performance Graphics 2009 (New Orleans, Louisiana, August 01 - 03, 2009).

[27] Moller, T., Trumbore, B., Fast minimum storage ray triangle intersection , High Performance Graphics 2009, New Orleans

[28] Nijasure, M., Pattanaik, S., Goel, V., Real-time global illumination on gpu, U. Central Florida, Orlando, FL

[29] Pharr, M., Humphreys, G., Physically based rendering, Morgan Kaufmann (2004)

[30] Purcell, T. J., Donner, C., Cammarano, M., Jensen, H. W., Hanrahan, P., Photon mapping on programmable graphics hardware, In Proc. Graphics Hardware, 41–50 (2003)

[31] Purcell, T. J., I. Buck, W. R. Mark, P. Hanrahan. 2002., Ray tracing on programmable graphics hardware, ACM Transactions on Graphics (Proceedings of SIGGRAPH 2002) 21(3), pp. 703–712.

[32] Sillion, F., Puech, C., A general two-pass method integrating specular and diffuse refection, In Proceedings of Siggraph (1989), pp. 335–344.

[33] Tole, P., Pellacini, F., Walter, B., Greenberg, D. P., Interactive global illumination in dynamic scenes, ACM Trans. Graph. 21(3):537–546 (2002)

[34] VeláZquez-ArmendáRiz, E., Lee, E., Bala, K., Walter, B. 2006., Implementing the render cache and the edge-and-point image on graphics hardware, In Proceedings of Graphics interface 2006 (Quebec, Canada, June 07 - 09, 2006)

[35] Wald, I., Realtime ray tracing and interactive global illumination, PhD Thesis, Saarland University, 2004

[36] Wald, I., GüNther, J., Slusallek, P., Balancing considered harmful - faster photon mapping using the voxel volume heuristic, Eurographics Symposium on Rendering 2004

[37] Walter, B., Drettakis, G., Greenberg, D. P. 2002., Enhancing and optimizing the render cache, In Proceedings of the 13th Eurographics Workshop on Rendering (Pisa, Italy, June 26 - 28, 2002)

[38] Walter, B., Drettakis, G., Parker., S., Interactive rendering using the render cache, In Rendering techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering), volume 10, pages 235--246, Jun 1999

[39] Walter, B., Fernandez, S., Arbree, A., Bala, K., Donikian, M., Greenberg, D. P., Lightcuts: a scalable approach to illumination, ACM Trans. Graph. 24(3):1098–1107 (2005)

[40] Wang, R., Zhou, K., Pan, M., Bao, H. 2009., An efficient gpu-based approach for interactive global illumination, ACM Trans. Graph. 28, 3 (Jul. 2009), 1-8.

[41] Ward, G. J., Rubinstein, F. M., Clear, R. D., A ray tracing solution for diffuse interreflection, In Proc. SIGGRAPH pp85–92 (1988)Zhou, K., Hou, Q., Wang, R., Guo, B. 2008., Real-time kd-tree construction on graphics hardware, ACM Trans. Graph. 27(5) 126:1–11

[42] Zhou, K., Hou, Q., Wang, R., Guo, B. 2008., Real-time kd-tree construction on graphics hardware, ACM Trans. Graph. 27(5) 126:1–11