https://doi.org/10.2352/ISSN.2470-1173.2017.3.ERVR-095

Distributed VR Rendering Using NVIDIA OptiX

Dylan McCarthy and Jürgen P. Schulze, University of California San Diego, La Jolla, CA, USA

Abstract

Virtual reality is rapidly becoming a pervasive component in the field of computing. From head mounted displays to CAVE virtual environments, realism in user immersion has continued to increase dramatically. While user interaction has made significant gains in the past few years, visual quality within the virtual environment has not. Many CAVE frameworks are built on libraries that use rasterization methods which limit the extent to which complex lighting models can be implemented.

In this paper, we seek to remedy this issue by introducing the NVIDIA OptiX real-time raytracing framework to the CAVE virtual environment. A rendering engine was first developed using NVIDIA OptiX before being ported to the CalVR virtual reality framework, which allows running OptiX in CAVE environments as well as modern consumer HMDs such as the Oculus Rift.

Introduction

Before delving into the solution, it is important to understand the problem. Equally as important, we must know how we have arrived at this problem.

Advancements have been made in the field of computer graphics driven by different motivations. On one end, there is interactive rendering. Rendering in which a user is able to move around the scene, move objects within the scene, or move the scene itself. The advancements in this region have been largely motivated by the gaming industry, visualization, and computer-aided design to name a few. In these applications, interactive frame rates are integral to a usable product. On the other end, there is photorealistic rendering, a field which has been largely motivated by the special effects industry as well as advertisement. It is often more possible to create a good looking view of a product by using photorealistic rendering rather than having to deal with complex lighting systems and cameras to capture a photograph in a physical environment. Similarly, it is often easier (and sometimes necessary) to create a frame of a movie through rendering techniques to allow for unreal appearances or to cut cost of production.

In recent years though, there has been an increased need to have realism in virtual environments. As virtual reality becomes more a part of our every day lives, it too becomes integrated with other fields within academia as well as in industry. Virtual reality has become a key part of training within the military. Service men and women are currently being trained using head mounted displays, omni-directional treadmills, and virtual cockpits. In this sense, it is imperative to create a sense of realism for the individual in training. A lack of realism could lead to a disconnect to one's training when out in the field. Another field in which virtual reality applications have seen a marked increase in quantity is the field of medicine. Whether it is for surgical training or phobia treatment, the desire for realism stands.

In this publication we describe how we integrated the NVIDIA Optix [12] real-time raytracing library into our virtual reality framework CalVR [11] so that we can render VR environments interactively.

Related Work

As previously mentioned, knowing where to start building a solution requires knowledge not only of the problem itself, but why the problem exists in the first place and how it developed.

Before the 1970s, shading methods were performed on a coarse-grained polygon level until the introduction of shading models which more realistically modeled lighting and materials within the real world. Diffuse shading on the polygon level - a method developed by Gouraud [5], was the method of choice for rendering at this time until 1974 when Phong [6] and Blinn [7] introduced specular shading on the fragment (pixel) granularity.

By the 1980s and 1990s, a lot more focus had been dedicated to the implementation of complex lighting models. Various methods such as ray tracing introduced by Whitted [8], Radiosity by Goral/Torrance [9], and path tracing developed by Kajiya [10], were able to achieve new levels of lighting complexity that existing methods could not. The development of physically accurate lighting models has been an important factor which has brought us to the visual results we see in graphics applications every day.

Over the years, there has been a great divergence between the interactive and photorealistic sectors of the field of Computer Graphics. Today, there are various libraries to choose from for development of interactive graphics applications. Some examples include OpenGL, a cross platform API, and DirectX, a Windows-specific graphics API. Over the years, much of the focus has been on increasing the amount of geometry within a scene while maintaing interactive frame rates as well as efficient texturing techniques. Unfortunately, very few tweaks for realism have been made over the years. Real-world phenomena such as shadows have been poorly addressed within the world of interactive graphics. Current methods only allow for hard shadows to be rendered at interactive frame rates within these graphics APIs.

Photorealistic rendering has seen many more techniques introduced since the 1990s such as bi-directional path tracing, Metropolis Light Transport, and Photon Mapping. These techniques allow for high levels of realism by allowing global illumination, realistic soft shadowing, and refraction through multiple media. Historically, though, these methods are very slow. To generate a single frame using these complex algorithmic methods, it may take minutes, hours, or even days.

Motivation

In comparing the two methods of polygonal rendering and ray tracing, a distinguishing factor emerges. In interactive, polygonal graphics, much of the time spent generating a frame for display on the screen is spent sending data to the GPU. Interactive graphics relies on a method called rasterization; an inherently se-

rial process. For each object in a given scene, all data must be directed to the GPU serially. Current frameworks implement the aforementioned Z-buffer algorithm for determining object depth with respect to a specific viewer. As such, generation of visual data in an interactive rendering context is not conducive to multithreading.

Ray tracing, a photorealistic rendering technique, is an inherently parallel workload. This allows implementation to take advantage of multiple processing units in parallel. This is especially attractive when considering scalability. Rasterization techniques find performance increases in determining what is able to be avoided whereas photorealistic rendering is driven by the desire to achieve better results based on what is even possible with lighting models that have yet been discovered.

Normally, CAVE environments are mechanized by a cluster of nodes, where each node is dedicated to generating display data for a specific screen or set of screens within the viewing region. In the context of photorealistic rendering, this lends itself perfectly to the idea of parallelization. For a given scene, the workload can easily be divided amongst various nodes within the cluster to generate various parts of the frame which can be stitched together to form a complete image.

CalVR

CAVEs are immersive virtual reality environments which are typically a video theater situated within a larger room. The viewing area is comprised of multiple screens, often times oriented in such a way that the viewer can be completely surrounded. The viewable area can be made of screens which are rear projection, or LCD displays oriented in a tiled fashion.

At UCSD, several CAVE environments exist. Namely, the StarCAVE (rear projection screens), NexCAVE, TourCAVE, and WAVE (all tiled LCD monitors). All of these CAVEs run on a common framework called CalVR. CalVR is a portable virtual reality framework developed at the Qualcomm Institute of UCSD. CalVR implements typical VR functionality of middleware such as window and viewport creation, viewer-centered perspective calculations, multiple graphics channels displays, multiprocessing, multithreading, cluster synchronization, and stereoscopic 3D viewing. Additionally, it supports non-standard VR systems such as autostereoscopic displays, 3D menu system, among others.

CalVR uses OpenSceneGraph, a graphics API written in standard C++ which uses OpenGL. As a result of this, CalVR uses rasterization methods which allows it to achieve reliable interactive frame rate. With the introduction of new graphics APIs which harness the parallelization capabilities of GPUs, it was our goal to integrate such a tool into the CalVR framework.

OptiX

One such framework which provides the capability to harness the parallelization capabilities of GPUs is the OptiX rendering framework developed by NVIDIA. OptiX itself is not a renderer, but rather a general purpose ray tracing API which facilitates development of applications built for the purposes of rendering, baking, collision detection, A.I. queries, etc. OptiX is based on NVIDIA CUDA, a parallel computing language platform with syntax and compilation very similar to that of the C language. The OptiX framework works by offloading computation to GPUs based on kernels written by the developer and passed into the OptiX engine. Within these kernels, the developer is free to perform any computations necessary for a given algorithm. This is especially helpful when developing for the CalVR framework due to the flexibility for implementation of various photorealistic rendering techniques.

As of the release of OptiX 3.5, a new addition was added to the API which handled the construction and traversal of acceleration structures as well as ray-triangle intersection routines, responsible for roughly 90% of the computation within a ray tracing application.

Ray Tracing

The development of the CalVR Plugin began with the creation of the rendering engine written in OptiX. The first task was to build a ray tracer that was capable of producing soft shadows with a movable area light, as well as structured importance sampling of an HDR environment map.

Area Light

The first step was to implement shadow ray casting. The ray generation program shoots a ray into the scene from the location of the camera through the image plane and intersects the objects within the scene. Upon a successful intersection, a shadow ray is shot from the location of the intersection to the light source itself. If there is an intersection on the path from the point of intersection to the light source, the object is in shadow. The second step was to create the area light itself. The area light is represented simply as an anchor point and two vectors which extend from the anchor to form a parallelogram. Once the area light was created, the algorithm for shooting shadow rays needed to be modified. The modification made first was to shoot 100 samples along the parallelogram which formed the area light.

The method of sampling was uniform jitter in which for the 10x10 grid of samples along the area light, the same sample point within each of the 100 squares was chosen (Figure 2). The algorithm for uniform jitter can be seen in Algorithm 1.

Algorithm 1 Uniform jitter algorithm

```
1:  procedure AREA LIGHT
2:  let offset = rand_float2()
3:  let irradiance = float3(0)
4:      for each sample i ∈ N do
5:          Shoot ray from intersection to light with offset
6:
7:          if no occlusion from hit point to light sample then
8:              Add contribution to irradiance
9:
10:         end if
11:     end for
12: end procedure
```

As seen in Figure 1, the uniform jitter sampling technique creates banding due to repetition in the way the area light is sample by nearby points in the scene. Due to this complication, it was evident that the sampling technique needed to be altered again. In order to remove the banding, random numbers needed to be used. A buffer was created on the host end and passed down to the device code. By doing this, each pixel in the window has different

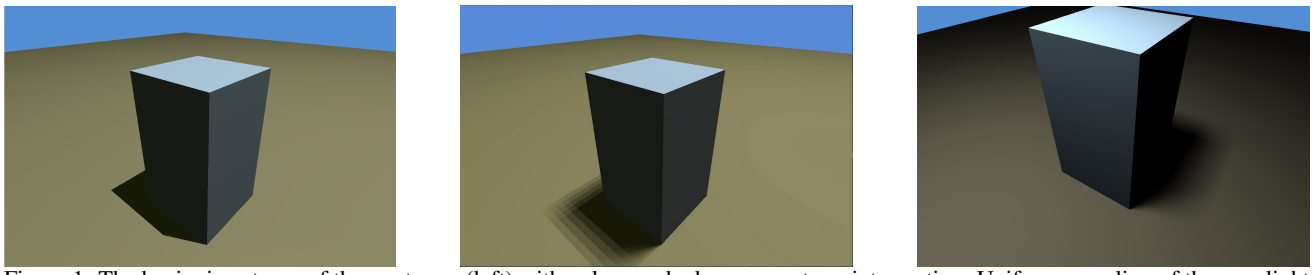

Figure 1: The beginning stages of the ray tracer (left) with only one shadow ray cast per intersection. Uniform sampling of the area light source (middle) creates visible banding due to a lack of randomness in the sampling pattern. Stratified area light source sampling (right) creates a much more pleasant soft shadow.

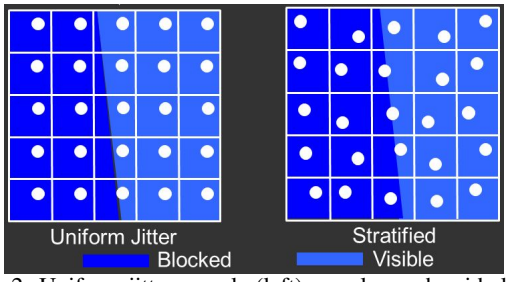

Figure 2: Uniform jitter sample (left) samples each grid element in the same relative location. Stratified sampling (right) samples a random position within each respective grid element[2].

offset values to lessen any sort of repetition within the sampling of the area light. The algorithm for this stratified technique can be found in Algorithm 2. Results showing the improvement over the uniform jitter algorithm can be seen in Figure 1.

Algorithm 2 Stratified algorithm

```
1:  procedure AREA LIGHT
2:  let irradiance = float3(0)
3:      for each sample i ∈ N  do
4:          let offset = offset_buffer(i)
5:
6:          Shoot ray from intersection to light with offset
7:
8:          if no occlusion from hit point to light sample then
9:              Add contribution to irradiance
10:
11:         end if
12:     end for
13: end procedure
```

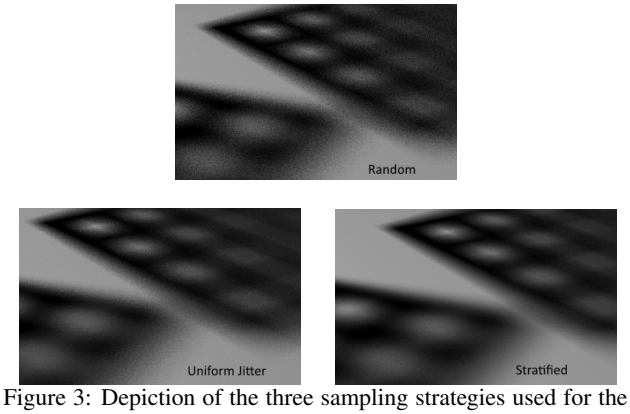

Figure 3: Depiction of the three sampling strategies used for the grids scene. The random sampling (top) technique causes unwanted noise as a result of potential clumping of sample points. Uniform jitter (left) has less noise than the random sampling results but still has some noticeable noise especially within the grid spaces. Stratified sampling (right) has the best results with smooth soft shadows and gradual transition from shadowed regions to lit regions.

The ray tracer really was the rendering engine in its nascency. At the time the soft shadow implementation was developed, it was novel. Until that point, image processing solutions had been used to approximate soft shadows to maintain interactive frame rates.

Progressive Photon Mapping

While ray tracing allows for decent performance and far superior image quality to rasterization methods, ray tracing does not adequately model more complex lighting. For example, ray tracing is unable to model real world phenomena such as caustics. A caustic is a result of a curved surface directing light to a focal point. It is the process in which the trajectory of photons from a light source is bent in a similar fashion such that photons meet at specific areas in a scene which appear to the viewer as very well lit.

Despite the success of the ray tracer developed, GPU technology is improving rapidly enough to where more complex algorithms can be implemented in preparation for improved hardware capability. Because of this, the next step was to improve the rendering engine to encompass progressive photon mapping.

GPUs are very capable computing devices when fed data quickly. They have great potential for providing marked performance improvement in workloads that allow for parallelization. Where GPUs fall short is in memory bandwidth as well as mem-

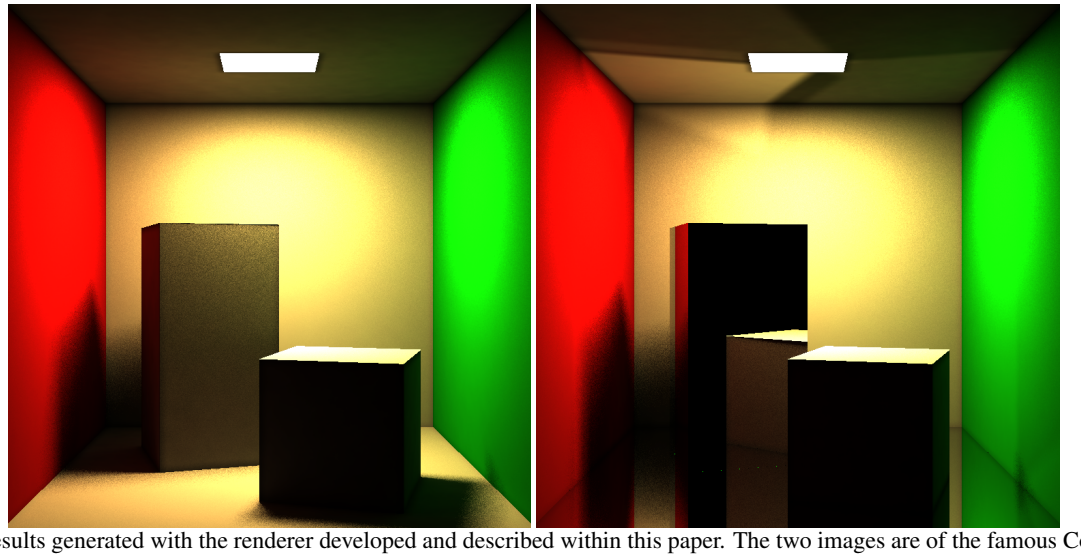

Figure 4: Results generated with the renderer developed and described within this paper. The two images are of the famous Cornell Box. Purely diffuse surfaces (left) show noticeable color bleeding on the sides of the two boxes from the red wall on the left and the green wall on the right. Specular mirrored surfaces (right) for the large box as well as the floor show reflection of photons within the scene creating a reflection of the smaller box on the front face of the larger box. Notice the reflected light off the top of the tall box onto the ceiling as it bounces directly from the light source and is absorbed by the diffuse ceiling material.

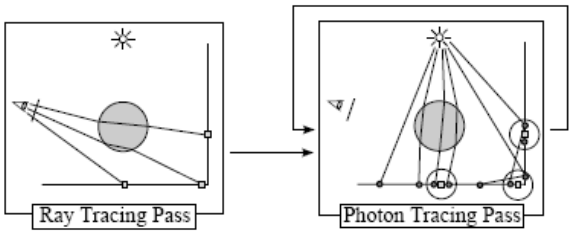

Figure 5: Progressive photon mapping diagram illustrating the iterative process of continually refining the results at the hit points by shooting more photons from the light source[3].

ory capacity. Photon mapping is a rendering algorithm which is known to require large quantities of memory for storage of photons that have been absorbed around the scene. An extension of photon mapping is a method called progressive photon mapping in which smaller quantities of photons are traced from the light sources at a time, but are gathered at hit points throughout the scene. Photons are continually traced from the light source and the hit points are refined to have more accurate irradiance values. This is a three pass algorithm, all three passes are discussed in the subsequent sections.

Algorithm 3 PPM Trace

```
1: procedure TRACE
2:     if camera position has changed then
3:         ray_tracing_pass()
4:     end if
5: photon_tracing_pass()
6: Generate kd-tree
7: gather_photons()
8: end procedure
```

Ray Tracing Pass

The first of the three passes is the ray tracing pass. In this pass, primary rays are traced from the cameras location into the scene to find hit points which are visible within that pixel's volume. During this pass, material properties of hit points are queried to determine whether to store the point of intersection as a hit point or continue by shooting more rays. This qualification is tested based on a given materials specularity value. If a material is specular, a hit point is not recorded at the point of intersection, but rather another ray is recursively traced to find the object being reflected at the hit point. Hit points are stored in a buffer allocated on the GPU from the host side.

Algorithm 4 Ray tracing pass

```
1: procedure RTPASS
2:     if material is diffuse then
3:         Store hit point in output buffer
4:     else
5:         Recursively trace a reflected ray
6:     end if
7: end procedure
```

Photon Tracing Pass

The second of the three passes is the photon tracing pass. This is the pass in which photons are traced from the light source itself and stored into an output buffer if it hits a diffuse material. The tracing algorithm is quite similar to that of the first ray tracing pass in that specular materials result in a reflected ray. However, in the photon tracing pass, photons are stored upon hitting a diffuse surface and a recursive ray is traced. Unlike the reflected ray for specular materials, the direction of the recursively traced ray from a diffuse surface is generated along the unit hemisphere. That is, the hemisphere that resides on top of the normal at the point of intersection. Random number generation is very impor-

Figure 6: The image on the left is an image generated using the renderer described in this paper. It addresses the SDS light path problem and clearly shows the caustics within the rings being reflecting in the mirror. On the other hand, the image on the right (not rendered by the renderer described in this paper) which employs bidirectional path tracing fails to capture the caustic from the case in the mirror. Notice the caustic appears as a shadow when reflected from the mirror[4].

tant in ray tracing applications for exactly this reason. If there is any bias or repetition, there will be unwanted visual effects. Ideally, the direction of the recursive ray would be completely random, since this is how light travels off of diffuse surfaces in the real world. Recursive ray tracing from diffuse surfaces allows for color bleeding between the wall and tall block as seen in Figure 4.

Algorithm 5 Photon tracing pass

```
1: procedure PPASS
2:     if material is diffuse then
3:         Store hit point in output buffer
4:         Recursively trace a reflected ray sampled along the
   unit hemisphere
5:     else
6:         Recursively trace a reflected ray
7:     end if
8: end procedure
```

Gather Phase

The third and final of the three stages of the progressive photon mapping algorithm is the gather phase. This is the phase in which a kd-tree is queried to find photons near the hit points generated from the ray tracing pass. This kd-tree is generated on the host side out of the photon hit points generated during the photon tracing pass. This process occurs between the photon tracing and gather phases. At a very high level, the idea of photon mapping is that in order to find the light paths converging on a singular point in space within the scene, we estimate based on the light paths of the parts of the scene around that point. The photons which hit the area surrounding a given hit point are likely to have the same light paths as the hit point in question (the view point generated from the ray trace pass). However, as the radius of photons which we take into consideration grows, so too does the disparity in the likeness of the light paths of all those absorbed photons. Conversely, as the radius shrinks and we continue to gather photons hitting nearer and nearer the hit point in question, the more similar the light paths are and the better the results. At the limit where the radius approaches 0, the result converges on all of the light paths reaching the given hit point.

SDS Path

A major topic of research within the Computer Graphics community within the past decade has been on creating algorithms which are capable of handling the SDS path. This is a phenomenon in which a light ray travels from the light source, hits a specular surface, a diffuse surface, another specular surface, and then reaches the camera. The ability to handle this complex light path has become a measurement of the effectiveness of a given rendering technique.

Existing methods such as path tracing struggle to converge on a correct result due to the low likelihood of tracing a ray from a diffuse surface which will reach the eye. Specifically, when a ray is traced and intersected, the reflected ray coming off of the diffuse surface is generated by randomly sampling the unit hemisphere surround the normal vector at the hit point. The number of directions that can be chosen when sampling the unit hemisphere is infinite. As a result of this, method such as path tracing and even bidirectional path tracing struggle to make the connections between the diffuse surface and the viewer.

Progressive photon mapping, on the other hand, handles this specific light path very well. As seen in Figure 6, the progressive photon mapping technique implemented in my renderer shows caustics in the mirror. Contrast this to the mirrored desk scene in Figure 6 where the caustic of the vase is unable to be properly reflected in the mirror.

CalVR Plugins

Applications are developed for CalVR in the form of plugins. Plugins in CalVR must implement a virtual base class which contains functions called by the CalVR core. The last part of this project was to create a plugin for CalVR. The biggest hurdle in integrating OptiX with CalVR was the compilation of the source files into a shared library. In order to use a specific plugin, CalVR needs to be rebuilt such that the core knows which plugin to load at runtime. Each plugin is compiled into a shared library, and its virtual functions specified by the CalVR Plugin based class are called by the core for rendering as well as for callbacks such as various forms of user input.

Config Files

It is also noteworthy that CalVR Plugins are initialized differently depending on the specific CAVE environment the application is running on at a given time. Each of the CAVE environment has differing hardware specifications, number of monitors, number of nodes, IP addresses, among other traits. These specifics are passed into the CalVR Plugin as well as the CalVR core through the use of config files. These files are files in XML format which specify the aforementioned traits of a CAVE environment as well as desired priority, tracking information (mouse, heads, hands), and menu system information.

Future Work

The renderer presented in this work has many more features it can include to more accurately represent phenomena within the real world. A possible improvement to the progressing photon mapping algorithm implemented would be the addition of a distributed rendering pass such that the method becomes stochastic progressive photon mapping. This alteration to the existing pipeline may prove to in fact be quite small. Visual improvement would take the form of improved representation of SDS lighting paths as stochastic progressive photon mapping performs more samples with a given pixel volume.

Additional improvements to this renderer could take the form of subsurface scattering as well as participating media. Subsurface scattering is a phenomenon in which photons traverse complex light paths within a medium itself rather than reflecting off. The most commonly used example of this is shining light through a human finger. A third and final potential improvement to the renderer is the ability to render light paths through participating media such as clouds, smoke, or fog. Both of these methods could also prove to be integrated with the existing rendering framework with ease as photon mapping lends itself very well to these two extensions.

Conclusion

With the release of various rendering frameworks such as NVIDIA's IRay and NVWorks, it seems very clear that the future holds much promise for distributed photorealistic rendering using GPUs. Although the common method for generating displayable data for a CAVE environment at present is scene graph OpenGL-based libraries, GPU-based ray tracing methods are become ever more attractive as dramatic improvements in performance continue to be made.

In this paper, the process by which a rendering engine was built using the OptiX ray tracing API was introduced, as well as the way in which it was integrated into the CalVR CAVE virtuality reality framework. Within the past few years, much modern technology has been released and improved to allow users a more visually pleasing immersive experience. On the whole, applications have done well to make use of the improved hardware capabilities with the existing methods. However, it is important to always consider the ways in which the improved hardware can be used differently than the way it is used by existing methods.

References

[1] Levoy Marc, History of Graphics, Stanford University CS 248 Introduction to Computer Graphics, September 25, 2003, `https://graphics.stanford.edu/courses/...03/History-of-graphics/History-of-graphics.ppt`.

[2] Ramamoorthi Ravi, A Theory of Monte Carlo Visibility Sampling Ravi , UC Berkeley (2010).

[3] Hachisuka Toshiya, Progressive Photon Mapping, UC San Diego, SIGGRAPH Asia 2008, Singapore, December 2008.

[4] Georgiev, Light Transport Simulation with Vertex Connection and Merging, Saarland University, 23rd International Conference on Transport Theory, September 2013

[5] Gouraud, Henri. "Continuous shading of curved surfaces". IEEE Transactions on Computers. C20 (6): 623629, 1971

[6] B. T. Phong, Illumination for computer generated pictures, Communications of ACM 18 (1975), no. 6, 311317

[7] James F. Blinn. "Models of light reflection for computer synthesized pictures". Proc. 4th annual conference on computer graphics and interactive techniques: 192198, 1977

[8] Turner Whitted. An improved illumination model for shaded display. Communications of the ACM 23, 6 (June 1980), 343-349

[9] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In Proceedings of the 11th annual conference on Computer graphics and interactive techniques (SIGGRAPH '84), Hank Christiansen (Ed.). ACM, New York, NY, USA, 213-222

[10] James T. Kajiya. The rendering equation. In Proceedings of the 13th annual conference on Computer graphics and interactive techniques (SIGGRAPH '86), David C. Evans and Russell J. Athay (Eds.). ACM, New York, NY, USA, 143-150, 1986

[11] J.P. Schulze, A. Prudhomme, P. Weber, T.A. DeFanti, "CalVR: An Advanced Open Source Virtual Reality Software Framework", In Proceedings of IS&T/SPIE Electronic Imaging, The Engineering Reality of Virtual Reality, San Francisco, CA, February 4, 2013, ISBN 9780819494221

[12] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. OptiX: a general purpose ray tracing engine. ACM Trans. Graph. 29, 4, Article 66 (July 2010), 13 pages

Author Biography

At the time this research was done, Dylan McCarthy was a graduate student in the computer science department of UC San Diego. He graduated in the summer of 2016 with the degree of Master of Science.

Jurgen Schulze is an Associate Research Scientist at UC San Diego's Qualcomm Institute, and an Associate Adjunct Professor in the computer science department, where he teaches computer graphics and 3D user interfaces. He holds an M.S. degree from the University of Massachusetts and a Ph.D. from the University of Stuttgart in Germany. After his graduation he spent two years as a post-doctoral researcher in the Computer Science Department at Brown University.