OptiX-based Raytracing with Volumetric Effects

Lawrence Sebald
University of Maryland Baltimore County

Abstract

In the past decade, the GPU has quickly become a powerhouse of computation. Even commodity machines have GPUs capable of hundreds of gigaflops of computation. Most of the computational power that exists in the GPUs of today is within the shading units of the device. These shading units are highly optimized SIMD floating-point engines that can be used for various types of computation, not just simply shading traditional Z-buffer rendered scenes as they were originally intended.

Nevertheless, traditional Z-buffer based rendering is still the most common operation performed on GPUs today. Computer games have taken advantage of GPUs by pumping greater amounts of polygons into scenes to produce more lifelike images. Recently, work has been done in the area of GPU-based real-time raytracing. Raytracing has many advantages over traditional Z-buffer based rendering in visual quality; however, its computational complexity has limited its applications in real-time rendering. This paper builds on the work done in real-time raytracing to add randomized volumetric effects to the scenes, in an attempt to add effects such as smoke and fog in a visually pleasing manner.

1. Introduction

The computing landscape has changed greatly over the past several years. As recent as five or so years ago, in the consumer market, only the higher end enthusiast machines would have multiple CPU cores, often provided by two separate CPUs. These machines, while they would have GPUs, had nowhere near the computational power seen today. Now, one would be hard pressed to find a machine that did not have at least a dual-core CPU, even in the fairly low end laptop space (ignoring the space of ultra-low power laptops and netbooks). This has prompted a paradigm shift away from single-threaded monolithic programs into multi- and many-threaded programs designed to use this additional CPU horsepower.

In line with the shift from single-core to dual-core general purpose CPUs, GPUs have, likewise, seen a shift to having more computational cores. The initial impetus for the expanding of the number of cores in GPUs was for real-time shading applications. In the past ten years, the landscape of real-time visual programs has gone from a fixed-function pipeline to a programmable pipeline with many options to be tweaked by programmers. Real-time shading led to vast improvements in the visual quality of games and other applications, without decreasing the frame rate.

As these shading cores are, at their root, highly optimized floating-point computational engines, it is only natural that they would be used for more general purpose computation. At first, general purpose computation on a GPU had to be molded to fit the shader-based pipeline through OpenGL or DirectX; however, since that time, frameworks have begun to spring forth that make matters simpler for general purpose computation. NVIDIA's CUDA framework was the first of these to gain mass acceptance in the field; however, other frameworks not limited to NIVIDA's GPUs, such as OpenCL and Microsoft's DirectCompute have recently been implemented. These frameworks can be used to create general purpose programs, but they can also be used to accelerate classic rendering techniques other than Z-buffer based rendering.

Various work has been done in the recent past on GPU-based raytracing. Raytracing is essentially an embarrassingly parallel algorithm that can benefit from the many-core nature of GPUs. While the images produced by such programs are visually pleasing and do run at interactive speeds, they often lack many effects that can add to their visual appeal and realism. Notably, volumetric effects like fog are all but completely absent from consideration.

One important factor in generating convincing volumetric effects is having appropriate noise backing the rendering. Noise is used to bring variation to the volume data. While it is possible to fully define a volumetric effect such as fog as a series of voxels imported into the program from a data file, this is suboptimal as it is possible to randomly generate the data at runtime. Visually interesting and appealing fog effects, for instance, can be generated by combining 2D noise, such as that introduced by Perlin [Perlin 1985], with a simple falloff function for height. Random generation of the data has other advantages, such as the potential for making the visual experience different each time the program is run, which raises its appeal as well.

This paper has two primary contributions. The first of these is an improvement upon existing pseudorandom number generation techniques for noise data on a GPU. This paper explores a PRNG based on the Tiny Encryption Algorithm

(TEA) and how to improve it across multiple-GPU systems so that each GPU generates a unique stream of pseudorandom numbers, regardless of the input stream. In addition, this paper explores the use of noise based on these pseudorandom numbers to generate simple volumetric effects in a GPU-based raytracer using OptiX.

The remainder of this paper is organized as follows. Related work on GPGPU computation, GPU-based raytracing, and GPU PRNGs is presented in section 2. The implementation details of the improved TEA-based PRNG, as well as the OptiX-based raytracer, is presented in section 3. Section 4 presents results and performance data. Future work is presented in section 5, and conclusions are presented in section 6.

2. Related Work

2.1. *General Purpose GPU Computation*

Many frameworks have been proposed for using the computational power of the GPU for general purpose computations. The first widespread framework for general purpose computation on GPUs was the OpenGL Shading Language [Kessenich et al. 2004]. The OpenGL Shading Language (GLSL) was primarily designed for real-time shading of Z-buffer rendered scenes but found acceptance as a general purpose computational language as well, through its support of a SIMD programming paradigm. More recently, frameworks have begun appearing that focus entirely on the general purpose aspect and do not constrain the programmer to a graphics pipeline. The most widespread of these is NVIDIA's Compute Unified Device Environment (CUDA), introduced in 2006 [NVIDIA 2010]. CUDA allows programmers to think in terms more natural than graphics shaders for computation. Specifically, CUDA allows one to write so-called kernel functions that perform some computation. These kernel functions are to be executed on many threads at a time sectioned off in a grid. However, one must pay careful attention to how memory is accessed, as the memory access pattern can negatively affect the performance of the application.

CUDA is a very good model for general purpose programming on the GPU; however, it is limited in scope. CUDA is only available for GPUs from NVIDIA. To rectify this, Apple and the Khronos group proposed the Open Compute Language (OpenCL) as a standard compute framework for GPUs [Khronos Group 2010]. The design of OpenCL closely resembles that of CUDA, as NVIDIA also was involved in the development of OpenCL. Unlike CUDA, OpenCL is available for use without an NVIDIA GPU, with implementations provided by Apple, Intel, AMD, and IBM.

2.2. *Real-time GPU-based Raytracing*

There has been a recent interest in work on GPU-based real-time raytracing algorithms. Among the early work on this subject, a modified KD-tree algorithm found use [Foley and Sugerman 2005]. Specifically, the algorithms kd-restart and kd-backtrack were found to be quite helpful in developing fast GPU-based raytracers. These algorithms are noteworthy as they do not require the recursive structure that most raytracing algorithms require. GPU compute languages all disallow recursion due to a lack of a hardware supported stack on the architectures. Various other work has also been done on KD-Tree based acceleration of GPU raytracing [Reiter Horn et al. 2007; Zhou et al. 2008].

In addition to KD-Tree based acceleration, other acceleration structures have been investigated for GPU-based raytracing. Bounding Volume Hierarchies have found use and have been found to be quite suitable for acceleration of a GPU-based raytracer [Carr et al. 2006; Günther et al. 2007]. In addition, several more exotic approaches have been tried, including using the GPU's Z-buffer rasterization to accelerate raytracing intersection operations [Chen and Liu 2007].

Recently, NVIDIA has proposed a system called OptiX, which is developed in CUDA to provide a framework for easy GPU-based raytracing [Parker et al. 2010]. The OptiX framework was designed to help bring interactive raytracing to programmers in a simple-to-use manner.

2.3. *GPU Random Numbers*

Generation of random numbers on a GPU has been researched by many. Random numbers have many uses in graphics related applications. Among these uses are generation of noise for texture synthesis and various shading effects. While it is possible to generate noise into a texture on the CPU and upload that to the GPU, this is often suboptimal. Doing so stresses the texture fetch unit of the GPU and can cause slowdowns if a large amount of noise is needed. Thus, it is helpful to have a completely computational solution that does not stress the GPU's memory or texture fetch units.

Many techniques for random number generation have been ported from CPU-based systems to GPUs, and there is large amounts of information about the techniques that have been tried with success. One area that has seen research is in the use of cryptographic and hashing algorithms to generate random numbers. Often, the purpose of cryptography is to make real data indistinguishable from random noise (especially in the area of disk encryption), thus encryption algorithms seem well suited to the task of generating random numbers.

Among the work that has followed the path of using cryptography to generate random numbers, many algorithms have been tried. Olano's mNoise algorithm, for instance, uses the Blum Blum Shub pseudorandom number generator to generate the random numbers needed for quality noise in very few instructions in a GPU fragment shader [Olano 2005]. Following this, Tzeng and Wei implemented a pseudorandom number generator on GPUs using the MD5 hashing algorithm as its base [Tzeng and Wei 2008]. This approach vastly improved the randomness of the numbers generated over the BBS-based approach of Olano. However,

this increase in quality was accompanied by a decrease in the performance. Even the most highly optimized MD5-based PRNG presented by Tzeng and Wei took over 28 times the amount of time in their tests to generate a 4096x4096 patch of random data when compared to BBS.

Zafar et al. proposed using the Tiny Encryption Algorithm to produce high-quality random numbers on the GPU with higher speed than MD5 [Zafar et al. 2010]. TEA is generally used in encryption with 32 rounds; however, adequate random numbers for noise generation can be generated with as few as two rounds of the algorithm. In comparison, the MD5-based approach was found to require at least 6 rounds before adequate randomness for noise was found. Zafar et al. demonstrated that a 2-round TEA PRNG compares quite well in performance to the BBS approach of Olano; however, the results are of greater quality. Higher quality numbers can be obtained by performing additional rounds of TEA, which makes the algorithm quite flexible and able to be tailored to the application.

3. Implementation

To implement the raytracing application and the random number generation driving the volumetric rendering, NVIDIA's CUDA was used. CUDA is perhaps the most widespread GPGPU programming environment at the current time, even though it is limited to supporting only NVIDIA's GPUs. OpenCL was a possibility; however, there is not as full of a support infrastructure available with OpenCL as there is for CUDA. Perhaps the most important infrastructure piece to this project is NVIDIA's OptiX framework, which is built on top of CUDA. Indeed, the raytracing application is a modification of one of the OptiX SDK example programs to add the volumetric fog effect that demonstrates the techniques described herein.

All benchmarking for the raytracing application was performed on an Apple MacBook Pro using Mac OS X 10.6.7. The particular machine used is a mid 2010 model, featuring a 2.66 Intel Core i7 (Arrandale) processor and 4GB of RAM. The machine has two GPUs that the OS switches between as applications demand it. This application only makes use of the NVIDIA GeForce GT 330M GPU, as it is based on CUDA. This particular GPU has 512 MB of dedicated RAM and is run on a PCI Express x16 bus. The GeForce GT 330M has a total of 48 unified shader cores which are used for CUDA computation.

The version of the CUDA toolkit used for this research was 3.2, with CUDA driver version 3.2.17. The GPU driver was version 1.6.26.31 (256.00.35f05). In addition, version 2.1.0 Release Candidate 2 of the OptiX SDK was used. During the course of this research, NVIDIA released Version 4.0 RC2 of the CUDA toolkit and the corresponding CUDA driver; however, for consistency of results, these architectural pieces were not updated.

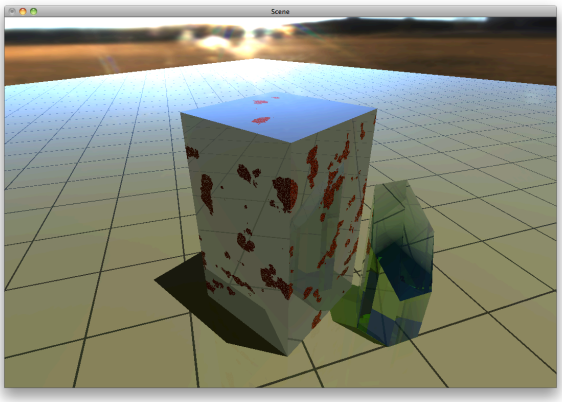

Figure 1: *Initial version of the tutorial application used to develop the raytracing application for this research. This is the tenth iteration of the "tutorial" example from the OptiX SDK.*

3.1. *Implementation of the Raytracing Application*

The raytracing application that was enhanced with volumetric effects as a part of this work is based on an example program provided with the OptiX SDK. Specifically, the "tutorial" application was used. This application is used by the OptiX SDK documentation to teach application developers how to add features to a raytracing application using OptiX one step at a time. For this research, the tenth iteration of the tutorial was used, and all nonessential pieces to that version were removed. The tenth tutorial draws a reflective box and a translucent crystalline object on a floor, with an environment map in the background. Both the reflective box and the floor have procedurally generated textures applied to them, as is shown in Figure 1.

Once the tutorial program was adequately stripped down to remove anything other than the tutorial that was modified, the framework for the volumetric effects were added. To this end, OptiX requires a set of material programs to be defined. A set of material programs consist of two CUDA device functions. The first, and simplest of these two functions is called the "any hit" program. The any hit program is written to be called for calculations involving shadows. It should be fast and easy to compute, as it can be called quite often in the execution of a scene. The any hit program for the volumetric fog that is demonstrated simply ignores the intersection with the object containing the fog. While this is not completely physically accurate, it does reflect what the author generally expects from fog. The second material program is called the "closest hit" program. The closest hit program is responsible for calculating the full model of the material in use on the primitive that has been hit by a ray. Closest hit programs can cast additional rays (for reflection or refraction, for instance), and must calculate the full contribution of the ray that is currently being cast to the visual output. For the volumetric fog effect, a very simple "refraction" is performed through the fog layer to determine what the ray would hit underneath

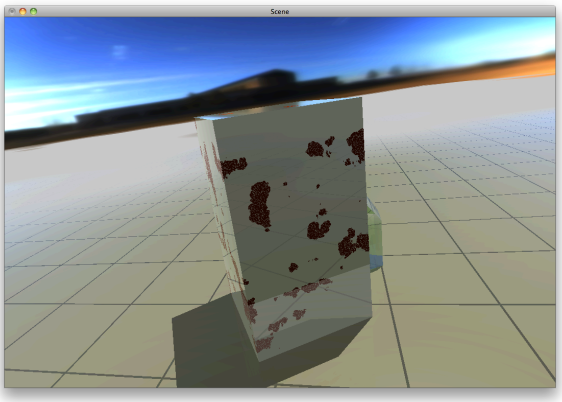

Figure 2: *Raytracing application with added volumetric fog effect. This image was rendered with 2 rounds of the Tiny Encryption Algorithm used to generate noise for the fog.*

the fog and calculate its contribution to the scene. After performing this step, the actual fog rendering is performed by stepping the incoming ray through the volume until it exits the volume. The step size can be controlled by setting a variable in the material program. The contribution of each step is determined by a TEA-based noise function in the XZ plane and a linear falloff in the Y direction. The TEA-based noise is detailed further later in this section. The resulting program is shown in Figure 2.

The fog is implemented inside of a box primitive, which was provided with the tutorial example. In this application, the box is the same dimensions in the XZ plane as the floor underneath, however it, unlike the floor, has height in the Y direction. In the example shown in Figure 2, the fog primitive has a height of 2 units (to give context, the reflective box has a height of 7 units).

3.2. *Implementation of Pseudorandom Numbers*

To generate gradient noise, a pseudorandom number generator is used to generate some data. While many different PRNGs are available, one that has been studied on GPUs for efficient implementation is based on the Tiny Encryption Algorithm. Zafar et al. demonstrated that TEA is capable of generating relatively high quality pseudorandom numbers on a GPU with relatively little code and thus with high performance. A simple implementation of TEA with CUDA is shown in Figure 3.

This simple implementation has one major drawback that this research hopes to address. Namely, how to choose and appropriate key to perform the encryption. This key must be chosen such that it generates data that appears to be random. In the research of Zafar et al., the key used was chosen somewhat arbitrarily, and no further explanation is given for why that key was used over any other key. While many methods are available for generating a cryptographically random key, none seem well suited to generate such a key on a GPU.

```
__device__ uint2 tea(uint2 v, uint4 k) {
  unsigned int v0 = v.x;
  unsigned int v1 = v.y;
  unsigned int sum = 0;
  unsigned int delta = 0x9E3779B9;
  int i;
  uint2 rv;

  for(i = 0; i < NROUNDS; ++i) {
    v0 += ((v1 << 4) + k.x) ^
          (v1 + sum) ^ ((v1 >> 5) + k.y);
    v1 += ((v0 << 4) + k.z) ^
          (v0 + sum) ^ ((v0 >> 5) + k.w);
  }

  rv.x = v0;
  rv.y = v1;
  return rv;
}
```

Figure 3: *CUDA implementation of TEA. Input is two 32-bit words containing the data to be encrypted (v) and four 32-bit words containing the key to use for the encryption (k). The result is the encrypted data. Note that for performance, the loop should be unrolled.*

In addition to the key being random, it would be desirable for a system containing multiple GPUs to be able to have separate random streams for each GPU. The initial work of Zafar et al. would generate the same random number given the same input on all GPUs if run in parallel. This is quite undesirable for some types of applications, even if the objective is only to generate random data for a noise function. To this end, a relatively simple bootstrapping procedure to be performed on the PRNG is proposed, using the same algorithm to generate the key that is used to generate the random numbers later on. The general idea is relatively simplistic, however it is found to create decent results that are comparable to the fixed key used by Zafar et al.

The approach taken is to essentially encrypt some device-specific information along with a seed value to become the key. The initial key chosen to encrypt the data can be arbitrary. While 2 rounds of TEA are appropriate for generating random data used for noise, this is not quite as appropriate for generating the key to be used. At least 6 rounds of TEA should be used, but more can be used for additional assurance of the randomness of the key. As the key generation phase only occurs once for the duration of the program, it does not negatively affect the performance of the random number generation to use many rounds for the key generation. The only difference is in the startup time for the algorithm, which is not terribly significant. Pseudocode for the method used to combine the device-specific data and the random seed value is shown in Figure 4.

```
uint4 generate_key(uint4 ik, uint2 d) {
  uint4 rv;

  rv.yw = tea(d, ik);
  rv.xz = ik.yw;
  rv.xz = tea(d, rv);

  return rv;
}
```

Figure 4: *Function to be used to combine the unique device information and a random seed (stored as the elements of d) to form a device-unique key. ik is an initial key, chosen arbitrarily. The tea() function is as defined in Figure 3, with NROUNDS equal to 16.*

	FPS
No Fog	8.63031
Fog, step size = 0.1	3.32626
Fog, step size = 0.05	2.0666
Fog, step size = 0.01	0.556651

Table 1: *Average Frames Per Second (FPS) rendered with and without added volumetric fog. All tests were averaged over 30 seconds, with a moving camera.*

4. Results

4.1. *Effects of Volumetric Fog on Performance*

Volumetric fog that is rendered by marching along rays can reasonably be expected to be quite expensive to render, especially when many rays will need to be traced. As the primitive containing the fog covered the entirety of the floor of the scene, fog calculations were involved in almost every pixel that did not fall completely through to the background environment map.

The OptiX SDK examples have built-in support for performing benchmarks, and this support was used to generate the results shown in Table 1. For the first of these experiments, fog was disabled completely and the program was tested to get a baseline. This is equivalent to the original Tutorial 10 SDK example. For the tests with fog enabled, the effect of the step size of rays within the volume was tested. A smaller step size produces more visually pleasing fog, but it also can have a drastic influence on the rendering speed. As the step size is the amount along each ray that each sample is taken within the volume, the effect on rendering speed is easily understandable. As would be expected, for more realistic volume rendering, there is quite a bit of a performance degradation.

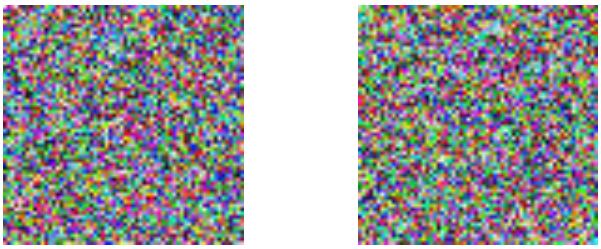

Figure 5: 64 x 64 pixel *white noise produced by the original GPU-based TEA implementation of Zafar et al. on the left, modified TEA implementation on the right (both using 4 rounds of TEA). Both produce similarly random results.*

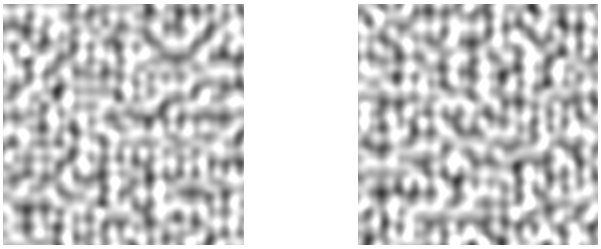

Figure 6: *256 x 256 pixel gradient noise produced by the original GPU-based TEA implementation of Zafar et al. on the left, modified TEA implementation on the right. Both render a single octave of noise, using 4 rounds of TEA.*

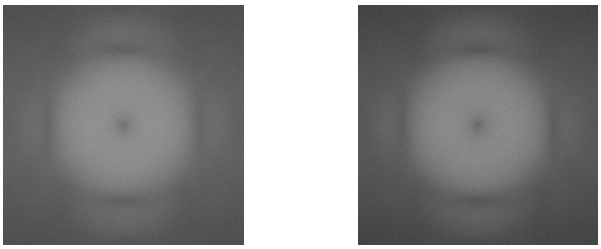

Figure 7: *Fourier Transform of a larger patch of noise (1024x1024 pixels). As with earlier images, the technique of Zafar et al. is on the left. Both were created with The GIMP and the G'MIC plugin, and are shown with 4 rounds of TEA.*

4.2. *Analysis of Modifications to the TEA PRNG*

The version of the TEA-based pseudorandom number generator presented by Zafar et al. uses one key that is never modified and produces decent results. With the modifications to TEA proposed in this paper, similar visual results are seen. In addition, the results produced by the modified TEA are different for each run and potentially each device available in the system.

	Time to Generate (µs)	Megabytes Per Second
Original	444,180	4,610.74
Modified	457,443	4,477.06

Table 2: *Performance of modified TEA versus the technique of Zafar et al. Numbers given are for generating 2048 MB of data. Tests were performed by running 2048 calculations of 1MB of data each using each of the two algorithms.*

As can be seen from Figures 5, 6, and 7, the quality of random data and noise between the implementation of Zafar et al. and that of this paper is quite comparable.

One side effect of the way the key generation is performed in the modified TEA algorithm is that the key is stored in the device memory on the GPU. In the original algorithm, the constant key would be rolled into the code directly. The fact that the key is stored in device memory does decrease performance a slight amount, as the data must be loaded into the registers of the GPU to be used. However, the flexibility afforded by the modified algorithm more than makes up for the small difference in time to run. Table 2 summarizes the timing information for generating 2048MB of random data using the original and modified TEA.

5. Future Work

There are several areas in which this work can be extended in the future. The most obvious of these is to address some of the limitations of the design that was chosen. As the fog volume was contained within a single large box primitive, there is no way for rays originating inside of the volume to be properly shaded. This could be alleviated somewhat by dividing the volume into smaller boxes, but this would not solve the problem in its entirety. A deeper understanding of the way that OptiX handles some aspects would certainly help solve this problem as well.

The voxel traversal code used for the benchmarking is somewhat simplistic and is not particularly well optimized. The code used essentially blindly steps along the ray cast into the volume, not using any sort of more optimized approach to volume rendering.

A second area where this work could be extended is in automatically determining the device specific information that is needed to initialize the key generation phase of the algorithm. For the data in this paper, the device specific identifier was simply an index. The random seed spoken of was simply the UNIX timestamp when the initialization function was called. It would be interesting to research whether there are registers available to CUDA device code that could be used to determine some more unique identifier than just an index into the devices available on the system. Unfortunately, no such information seems to be available in documentation.

The key generation function used was chosen for its simplicity, and only a few alternatives were tried before deciding on the function used. A more exhaustive review of potential key generation functions, including those that do not use TEA themselves for generating the key could very likely produce a more useful key generation function than what is provided here.

6. Conclusions

This paper demonstrates that it is not only possible, but fairly straightforward to create a real-time raytracing application using OptiX that has volumetric fog effects. Also, this paper demonstrates that it is possible to modify the TEA algorithm for random number generation on a GPU to handle multiple separate random streams without a major performance impediment and without degrading the quality of the random number generation. While the volumetric fog effects demonstrated herein have degraded performance significantly, future optimization work and future directions in GPU technology should alleviate these concerns.

Acknowledgements

I would like to thank Fahad Zafar for providing the GLSL implementation of TEA-based noise that was used as a basis for this project, and aiding my understanding of how it worked. In addition, I would like to thank Marc Olano for his assistance in directing this research.

References

[Kessenich et al. 2004] KESSENICH J., BALDWIN D., ROST R.: The OpenGL® Shading Language. <http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.10.59.pdf>.

[NVIDIA 2010] NVIDIA: NVIDIA CUDA C Programming Guide. <http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf>.

[Khronos Group 2010] KHRONOS OPENCL WORKING GROUP: The OpenCL Specification. Munshi A. (Ed). <http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf>.

[Foley and Sugerman 2005] FOLEY T. AND SUGERMAN J.: KD-Tree Acceleration Structures for a GPU Raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (HWWS '05)*. pp. 15-22.

[Reiter Horn et al. 2007] REITER HORN D., SUGERMAN J., HOUSTON M., AND HANRAHAN P.: Interactive k-d tree GPU raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games (I3D '07)*. pp 167-174.

[Zhou et al. 2008] ZHOU K., HOU Q., WANG R., GUO B.: Real-time KD-tree construction on graphics hardware. In *ACM Transactions on Graphics 27*, Article 126.

[Carr et al. 2006] CARR N., HOBEROCK J., CRANE K., AND HART J.: Fast GPU ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface 2006 (GI '06)*. pp. 203-209.

[Günther et al. 2007] GÜNTHER J., POPOV S., SEIDEL H., SLUSALLEK P.: Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *IEEE Symposium on Interactive Raytracing 2007*. pp. 113-118.

[Chen and Liu 2007] CHEN C., LIU D.: Use of hardware Z-buffered rasterization to accelerate ray tracing. In *Proceedings of the 2007 ACM Symposium on Applied Computing*. pp. 1046-1050.

[Parker et al. 2010] PARKER S., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: a general purpose ray tracing engine. In *ACM SIGGRAPH 2010 papers*, Hugues Hoppe (Ed.). Article 66, 13 pages.

[Perlin 1985] PERLIN K.: An image synthesizer. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques (SIGGRAPH '85)*. pp. 287-296.

[Olano 2005] OLANO M.: Modified noise for evaluation on graphics hardware. In *Proceedings of the ACM SIGGRAPH/ EUROGRAPHICS conference on Graphics hardware (HWWS '05)*. pp. 105-110.

[Tzeng and Wei 2008] TZENG S., WEI L.: Parallel white noise generation on a GPU via cryptographic hash. *In Proceedings of the 2008 symposium on Interactive 3D graphics and games (I3D '08)*. pp. 79-87.

[Zafar et al. 2010] ZAFAR F., OLANO M., CURTIS A.: GPU random numbers via the tiny encryption algorithm. In *Proceedings of the Conference on High Performance Graphics (HPG '10)*. pp. 133-141.