A Hybrid Rendering Engine Prototype for Generating Real-Time Dynamic Shadows in Computer Games

Daniel V. Macedo Maria Andréia F. Rodrigues

Universidade de Fortaleza(UNIFOR)
Programa de Pós-Graduação em Informática Aplicada (PPGIA)
Av. Washington Soares 1321, J(30)
60811-905 Fortaleza-CE Brasil

Abstract

Shadows are an extremely important part of accurate 3D rendering and key components of visual realism in computer games. In this work, we present a rendering engine prototype we have developed for 3D digital games, with a hybrid algorithm for the generation of dynamic shadows in real-time. We have implemented three solutions and in all of them, we generate an image that represents the shadows of the scene. Several tests were conducted and show that the ray tracing algorithms are quite competitive in complex scenes starting at 1M polygons, although they still depend on the image resolution, since the higher it is, the more rays will be generated towards the scene.

Keywords: rendering engines, games, ray tracing, dynamic shadows, real-time

Authors' contact:
{danielvalentemacedo,andreia.formico}gmail.com

1. Introduction

Rendering engines are responsible for generating realistic scenes in digital games [Gregory 2009]. They correspond to one of the most important and complex components of a game engine. Every realistic 3D scene contains a robust description of the geometry, viewpoint, texture, lighting and shading of its objects. The data describing the scene are sent to a rendering program, which processes and displays them as an image. Thus, rendering engines are responsible for several key tasks of the graphics pipeline and they are developed with the aid of interface libraries to graphics hardware, such as OpenGL [OpenGL 2014] and DirectX [DirectX 2014].

Rendering engine's excellence in robustness and performance can be determined by analyzing its ability to render 3D scenes containing photorealistic effects, as well as the processing time taken for completion of this task. Recently, several optimized solutions have been proposed in an attempt to minimize the processing time and labor expended in this process, as well as to generate realistic 3D scenes at interactive rates, around 30 FPS [Eberly 2000; Ericson 2005].

A step before generating the output (an image) in the 3D rendering pipeline is the rasterization or scan conversion. Through mathematical calculations, the vertices are mapped to pixels [Williams 1978], generating an image that represents the synthetic scene. However, this is a complex task, since the synthesized 3D model itself does not exactly match a scene with physical characteristics of the real world. In general, the rasterization process is done a million times and currently runs on the hardware level, almost entirely. It also takes into account ordering of objects and converting coordinates from object to camera space.

On the other hand, a powerful method for rendering 3D graphics with very complex light interactions is ray tracing. The algorithms used in a ray tracer simulate the reverse path of the light in the scene. In the real world, rays are emitted by a light source, spreading through all the objects in the scene. When they collide with any of these objects, rays undergo refraction or reflection, depending on the material composing their surfaces. Some of these rays reach the viewer's eye and generate the image being viewed. The simulation of this whole process is still a very complex task, since the light source in the real world generates a large number of rays, which makes the process extremely expensive and, most often, unfeasible to be performed.

In particular, visual effects such as shadows are extremely important to be modeled in digital games, both for the player (helping to identify the distances between objects), and for the proper rendering of the scene, making it much more realistic. Some existing rendering engines available in the current game engines, such as *Unreal Engine* [Unreal Engine 2014], *Unity* [Unity 2014] and *Cry Engine* [Cry Engine 2014], include this shadow feature (both static and dynamic).

2. Related Work

Several important visual effects, such as shadows, are quite important in generating realistic 3D scenes. Unlike other visual effects, shadows are not rendered objects. Instead, they are areas of the screen that are darker than others because they receive less light during illumination calculations. Thus, the hard part of adding shadows to a rendering engine is finding those areas in real time [Mcguire 2004]. There are many

algorithms to create shadows, for example, shadow volume and mapping [Kolivand and Sunar 2012]. Many of them still remain the main shadow algorithms to date [Heidmann 1991]. One of the first algorithms was the shadow volumes [Franklin 1997], which was implemented lately in graphics cards [Heidmann 1991]. It creates sharp, per-pixel accurate shadows from point, spot, and directional lights [Mcguire 2004].

Unlike the shadow volume, which is an object space algorithm, the algorithms for shadow maps [Woo *et al.* 1990; Hasenfratz 2003] are fast and image based. These algorithms are simple to be implemented, applicable to any kind of geometric primitive, easy to be optimized at the hardware level, and efficient enough to be used in complex scenes, essentially because they work in screen space. However, they have two main drawbacks: they generate aliased shadows and require the use of a bias to compensate the precision loss during the conversion of values between the *z-buffers*, preventing the generation of undesirable artifacts in the final shadow.

Currently, the emergence of GPUs has strongly motivated researches to explore new ideas to implement ray tracing in the context of digital games [Bikker 2007; Pohl 2014b]. However, the major challenge is still to perform the entire processing with dynamic scenes in real time. Despite all these technology advancements in graphics cards, the development of algorithms that guarantee interactive frame rates per second in games with more realistic visual effects, is still a complex and challenging task.

Carr *et al.* proposed the first ray tracing algorithm on GPU, but it runs only on GPU the calculation of intersection between rays and triangles [Carr *et al.* 2002]. Subsequently, Hanrahan *et al.* presented a solution for the generation of rays, traversal, intersection between rays and triangles, shading and creation of secondary rays, all running on separate GPU kernels [Hanrahan *et al.* 2002]. Some techniques also uses cache data between frames, as shown in [Ruff *et al.* 2013]. However, most of the techniques proposed so far do not focus effectively in dynamic scenes.

Additionally, there are already some engines, for example, *Arauna* [Bikker 2007] and *Brigade* [Brigade 2014], capable of generating images of dynamic scenes, at interactive rates (around 30 FPS). OpenRL [OpenRL 2014] and OptiX [OptiX 2014] libraries have emerged to assist programmers in developing more visually realistic graphical applications with ray tracing in real-time. Some hybrid solutions have also been presented by combining rasterization techniques and ray tracing to create a scene [Andrade *et al.* 2014].

In this work, we focus on the implementation of a more modular and hybrid rendering engine prototype, in which practically every effect works independently. Moreover, these effects can be implemented with more than one algorithm, either with a rasterization technique or ray tracing algorithm, and combined together at the end of the rendering process through the deferred shading pipeline. This approach can also be useful in performing comparative tests between existing modules, in a more standardized and independent way. To analyze the performance of our implementation, we have conducted several testes to compare three different solutions.

3. Rendering Pipelines

Rendering pipeline complexity has increased significantly since the advent of programmable shaders. Three-dimensional geometric shapes are composed of a set of vertices that form triangles. Mathematical calculations are performed to map these vertices to pixels [Akenine-Möller *et al.* 2008]. Two popular rendering pipelines that most engines use for scene rasterization in the area of games are *forward rendering* [Engel 2013] and *deferred shading* [Akenine-Möller *et al.* 2008]. In the former, the algorithm chooses an object from the scene to be rendered and calculates the surface shading, as well as all the lights that are influencing the rendering of each point displayed, in accordance with the material set to the objects. In the latter, the rendering is deferred until all geometries have passed down the pipeline [Deering 1988]. That is, the scene is first rendered into discrete buffers without any information about lighting or shadows. It splits the shader task into smaller sub-tasks that are written into an intermediate buffer (*g-buffer*), a series of textures/buffers that contain, per pixel, all the screen space information needed to compute the final color of each pixel to form the final image.

4. Rendering Engine with Dynamic Shadows in Real-time

While the current version of this work is in prototype status and not yet feature complete (other also realistic effects are planned to be incorporated soon into it as engine modules), it provides evidence to validate its feasibility and compare the performance differences between the three different solutions we have implemented for real-time shadows in dynamic scenes. The rendering engine we have designed and implemented uses a *deferred rendering* pipeline, making possible to develop different algorithms to calculate the dynamic shadows in real-time, making easier the integration of them into the engine.

Initially, we load an *obj* file with the 3D scene information to be rendered. After loading this data, the next step of the rendering engine is to generate the contents of the *g-buffer* (diffuse color, worldspace, normal and *z-buffer*), *i.e.*, maps containing geometric information of the scene in screen coordinates. Then, once the *g-buffer* is ready, the next step is to create another map that represents the pixels located in shadow areas of the scene. Additionally, for the

generation of these maps, we have implemented three different solutions. The first solution uses a cubemap of shadow maps to represent point light's shadows. In the second and third solutions the same algorithm is used, however, together with two different ray tracing libraries (OpenRL and OptiX, respectively) for generating the map containing the shadow pixels. In particular, on these both algorithm implementations we provide the *g-buffer* from the *deferred shading* pipeline, lighting information and the vertices of polygons to perform only the calculation of secondary rays (data on the primary ones are already available in the *g-buffer*). In all three solutions, we generate an image that represents shadow areas of the scene (middle right of Figure 1). This image is sent to the *deferred shading* pipeline, it is shaded, and the resulting image is then multiplied by the shadow image to generate the final scene. During the ray tracer initialization we pass these information: 3D scene geometry, lights and *g-buffer*. Then, the ray tracer extracts each pixel position from the *g-buffer* and fires a ray from each position to the light. If there is a collision of this ray with any object located between this position and the light, the ray tracing algorithm returns '0' for this pixel in the image, otherwise '1'. In the end, an image equivalent to that of Figure 1 (bottom right) is generated, however, using a ray tracing algorithm. After generating the shadow image, the rendering engine applies the light calculation for each pixel and multiplies the result by the shadow image to generate the final result. The execution flow diagram of the rendering engine is detailed in Figure 1.

5. Tests and Performance Results

Performance testings were conducted using the same 3D scene (bottom right of Figure 1), but with different settings. In each test we used 3 different mesh versions of the Happy Buddha (the first and the second meshes with 50k and 250k polygons, respectively, and the third one, a 1087k polygon model), in a 3D living room. Each of the scenes were rendered at 3 different resolutions: 480p, 720p and 1080p. Each test was repeated 3 times and the average frame rates (measured in FPS) were calculated and used to plot the results graphically. The goals of our tests were two-fold: to verify the general rendering engine performance and to evaluate its feasibility; and (2) to compare the behavior of the 3 different solutions (using shadow maps, OpenRL and OptiX) we have implemented. All tests were executed on an Intel Core i7-4770 Haswell Quad-Core 3.4GHz machine with 16GB RAM 1600MHz and NVidia GeForce GTX 780 Ti 3GB GDDR5 384-bit graphics card.

The results in Figures 2(a), 2(b) and 3(a) show that the ray tracing algorithms are quite competitive in complex scenes starting at 1M polygons, although they still depend on the image resolution, since the higher it is, the more rays will be generated towards the scene. It is interesting to note that the performance of the shadow maps algorithm (Figure 2(a)) depends strongly on the number of rendered polygons, influencing the FPS drops in the scene with 1087k. On the other hand, as the number of rendered polygons increases, both ray tracing algorithms (Figures 2(b) and 3(a)) do not degrade very much their performances. However, their performance decays with increased image resolution, since more rays in the scene need to be generated. In Figure 3(b) one can observe that the OptiX ray tracing algorithm outperforms the shadow maps in the scene with 1087k, showing that it can become competitive in scenes with a massive number of polygons.

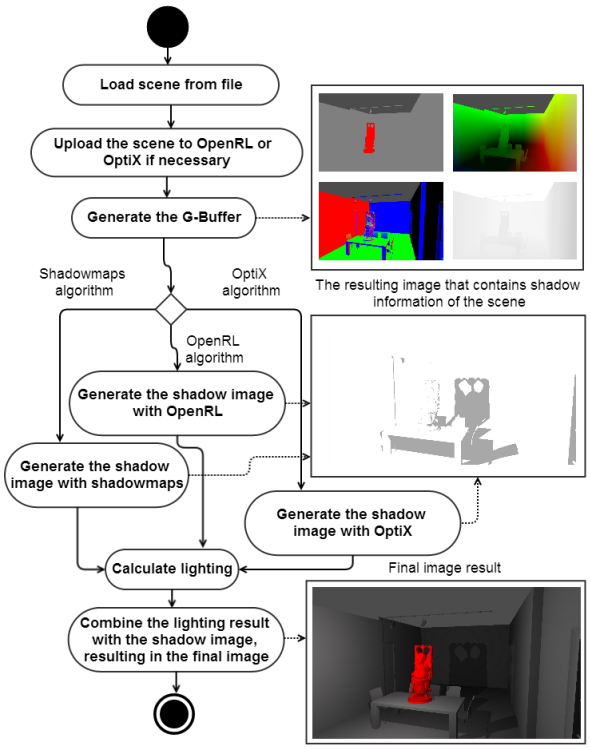

Figure 1: Execution flow diagram of the rendering engine.

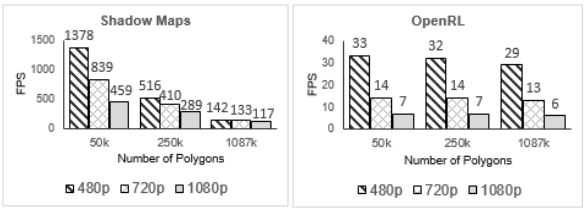

Figure 2: In (a) and (b), the FPS results using shadow maps and OpenRL, respectively

As in most any performance discussion, there are also some tradeoffs that are worth being discussed. Other scalability tests were conducted with the same 3D scene with 2M, 3M and 4M polygons (the graphical results are not shown in this work). In our tests we used the *Lbvh* builder (Table 1), which has a faster creation time and occupies less computer memory than the *MedianBvh*, *Bvh* and *Sbvh builders*, but it makes the traverser a little slower. In addition, during the tests with a 2M polygon model, only the *Lbvh builder* was able to process the scene, the other *builders* had memory overflow (even before processing

the scene). Recently, as shown in Table 1, NVIDIA OptiX released the *Trbvh builder*, which can generate an excellent data structure quickly, however, it uses about 3 times the size of the final *Bvh* for scratch space and only the commercial version is said not having memory constraints, making really difficult for the authors, under the commercial condition, to conduct additional tests using the *Trbvh* builder.

Table 1: OpenRL *vs* OptiX.

	OpenRL	OptiX
Types of shaders	3	8
Hardware dependency	It also supports the execution of programs on CPU	It runs only on NVidia graphics cards
Platforms	Windows	Windows, Mac and Linux
Syntax	Very similar to OpenGL	Own syntax
Builders	1 (unknown)	6 (*Sbvh, Bvh, MedianBvh, Lbvh, Trbvh* and *TriangleKdTree*)
Traverses	1 (unkown)	3 (*Bvh, BvhCompact and KdTree*)

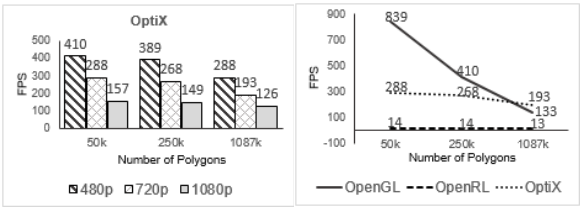

Figure 3: In (a) and (b), the FPS results using OptiX and a performance comparison of algorithms at 720p, respectively.

For the scenes composed of 3M and 4M polygons, the OptiX had memory overflow for all the available *builders* tested. However, the implementations with shadow maps and OpenRL managed to run up to 4M polygons, without manifesting any computer memory problems. Comparing the performance results of the scene containing 1M polygons with the one with 4M polygons, the shadow maps implementation had an FPS decrease of approximately 90%, whereas the OpenRL library had a decrease of only 60%.

6. Conclusion and Future Work

This work has demonstrated that ray tracing algorithms can become quite competitive in complex 3D scenes starting at 1M polygons. As the number of polygon increases, the curve representing the FPS variation of the shadow maps algorithm has a faster and more pronounced decay than those of the ray tracing algorithms (OpenRL and OptiX). However, in scenes containing more than 2M polygons, memory overflow occurred with the OptiX ray tracing algorithm. However, in scenes with up to 4M polygons was possible to observe the same previously reported behavior of decay between the shadow maps and the OpenRL algorithm, up to the point where the curve of the shadow maps underperforms the OpenRL one. It is noteworthy that using ray tracing algorithms, it becomes possible to represent various visual effects with physically correct models. However, for making the scene realistic it is also necessary to increase the number of samples (rays) per pixel (in this work, we used one ray per pixel), unfortunately, greatly increasing the processing time of the algorithms. As future work, we plan to extend the current rendering engine by optimizing its algorithms. This includes the implementation of new techniques for smoothing the shadow maps and also adding support for soft-shadows, reflections, and indirect illumination using ray tracing on the GPU.

Acknowledgements

Daniel V. Macedo and Maria Andréia F. Rodrigues are supported by CAPES and CNPq, under grants No. 157.257/2012-6 and 481326/2013-8, respectively, and would like to thank for their financial support.

References

AKENINE-MÖLLER, T., HAINES, E., HOFFMAN, N., 2008. *Real-time rendering*, 3rd edition, A.K. Peters Ltda.

ANDRADE, P., SABINO, T., AND CLUA, E., 2014. Towards a heuristic based real time hybrid rendering - A strategy to improve real time rendering quality using heuristics and ray tracing. In Proc. of the 9th VISAPP, p. 12-21.

BIKKER, J., 2007. Real-time ray tracing through the eyes of a game developer. In Proc. of the IEEE RT, p. 1-10.

BRIGADE, 2014. <http://icelaglace.com/projects/brigade-3-0/>. Accessed 21/02/2014.

CARR, N.A., HALL, J.D., HART, J.C., 2002. The Ray Engine. In Proc. of the 2002 ACM SIGGRAPH/ EUROGRAPHICS Conf. on Graph. Hardware, p. 37-46.

ENGEL, W., 2013. GPU Pro 4: *Advanced Rendering Techniques*. A K Peters Book. Taylor & Francis.

FRANKLIN, C., 1977. Shadow Algorithms for Computer Graphics, SIGGRAPH'77 Proc., vol. 11(2), p. 242-248.

GREGORY, J., 2009. *Game Engine Architecture. Massachusetts*. A K Peters.

HANRAHAN, P., BUCK, I., PURCELL, T.J., MARK, W.R., 2002. Ray tracing on programmable graphics hardware. ACM *TOG*, vol 21(3), p. 703-712.

HEIDMANN, T., 1991. Real Shadows, Real-time. IRIS Universe, vol. 18, p. 28-31.

KOLIVAND H., SUNAR, M.S., 2012. Real-time outdoor rendering using hybrid shadow maps. *IJICIC*, vol. 8(10B), p. 7168-7184.

McGuire, M., 2004. GPU Gems. Chap. 9: Effective Shadow Volume Rendering, Addison Wesley, p. 137-166.

POHL, D., 2014b. Quake 4: Ray Traced. <http://www.q4rt.de/>. Accessed 21/02/2014.

RUFF, C.F., CLUA, E.W.G., AND FERNANDES, L.A.F., 2013. Dynamic per Object Ray Caching Textures for Real-Time Ray tracing. In Proc. of the 2013 XXVI SIBGRAPI, p. 258-265.

WOO, A., POULIN, P. AND FOURNIER, A., 1990. A survey of shadow algorithms. *IEEE CGA*, vol. 10(6), p. 13-32.