Realistic Rendering in 3D Walkthroughs with High Quality Fast Reflections

Daniel Valente de Macedo Maria Andretia Formico Rodrigues

Universidade de Fortaleza (UNIFOR), PPGIA, Brasil

Abstract

Graphics applications have generated visual effects increasingly realistic and for such purpose, reflection effects are essential. To render realistic images, the reflectance of surfaces must be simulated accurately. In this context, it is common to use ray tracing, since it represents with great fidelity the behavior of light. However, ray tracing involves a very costly algorithm, so far indicated only in offline rendering scenarios. This paper presents a new hybrid solution for generating realistic rendering of objects in 3D walkthroughs. Moreover, the results show that our algorithm has a competitive performance, especially in generating high quality fast reflections, when compared with those generated with a fully implemented ray tracing algorithm.

Keywords: realistic rendering, high quality fast reflections, ray-tracing

Author’s Contact:

{danielvalentem Macedo, andreia.formico}@gmail.com

1 Introduction

The real world has reflections. In 3D scenes, the aim of realistic rendering is to calculate the amount of light reflected from the visible object surfaces that arrives to the synthetic camera through image pixels. Thus, reflection models are fundamental to that, since they are used to describe the appearance of surfaces, being generally part of a rendering engine [Gregory 2009].

Basically, every realistic scene contains a robust description of the geometry, viewpoint, texture, reflection, 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.

More specifically, rendering engines are responsible for several key tasks of the graphics pipeline, which include: loading of objects, rendering of visual effects (which consumes the largest computational resources), creation of visual effects (shading, shadows, reflections, indirect light, etc.), and display of this information as an image, on the output device’s screen.

Ideally, for real-time applications it is necessary to maintain at least 30 FPS to display the moving images with creating the illusion of smooth movement. In digital games, the images are generally generated by rasterization [Popescu and Rosen 2006], a method used to convert geometries formed by vertices, edges and faces, in pixels. The fact is that rasterization is an extremely fast process which takes advantage of APIs such as OpenGL [OpenGL 2015], benefiting directly from hardware (GPU).

However, another traditional method well-known in Computer Graphics to generate visually realistic images is ray tracing [Parker et al. 2013]. Basically, ray tracing simulates the opposite direction of the light, by tracing rays from the observer’s eye position, one or more per pixel, to find the nearest object blocking the path of the ray of light. That is, until it finds the object that should be represented at that pixel. The use of ray tracing facilitates the simulation of reflection and refraction effects, since they are already built into this recursive algorithm.

Even facilitating the process of representing with high quality important visual effects such as reflections, refractions and shadows, the ray tracing is, actually, a very costly algorithm. However, it is also a highly parallelizable algorithm on the GPU. Recently, due to such features, libraries dedicated to developing applications using GPU ray tracing were developed. One of these libraries is the OptiX [NVidia 2015.a] which is used in this work. OptiX ray tracing offers solutions in a transparent and efficient manner. In addition, it runs in parallel on the video card, through CUDA [NVidia 2015.b].

Since visual effects such as reflections are extremely important in computer graphics applications at run time, some game engines, such as the Unreal Engine [ Epic Games 2015], [Macedo et al. 2015], include those effects.

However, they typically use a combination of the Screen Space Reflection (SSR) technique and cubemaps (created from Sphere Reflection Capture) for generating the objects. This approach is not very accurate, since the SSR works only on screen space, therefore, failing to calculate the reflections of objects that are located outside the camera field of view or those that are being, for example, occluded by other objects.

Actually, the reflections generated by using cubemaps are not very realistic, since the synthesized reflections on every scene object comes from a same point of origin. Another important problem related with cubemap is the occurrence of lighting seams between adjacent objects when using different cubemaps.

This paper presents a simple and novel hybrid solution for generating high quality fast reflections, developed with a real-time algorithm for realistic rendering of scenes in 3D walkthroughs, with frame rates greater than 30 FPS. The algorithm combines rasterization (SSR) and a pure ray tracing through the OptiX, a library that focuses on the implementation of parallel ray tracing algorithms.

More specifically, our algorithm uses g-buffer information to identify which pixels of the scene surface are producing reflections. This information is then used by the SSR technique to generate these reflections.

To take advantage of the main benefits that both techniques offer and bypass their limitations, our algorithm initially generates both the reflections using the SSR and a mask map of the areas with the scene points in which the SSR failed to calculate the reflections (a well-known problem of the SSR, since it only processes what is visible in the image). After that, the ray tracing uses the SSR mask map, the g-buffer (worldspace, normal, z-buffer and diffuse color) from the deferred shading pipeline, the lighting information and the vertices of the polygons to calculate the secondary rays for the reflections (data on the primary ones are already available in the g-buffer) and to merge the SSR reflection with the additional ray traced reflections. Finally, the results obtained are combined with shadows [Macedo and Rodrigues 2014] (particularly a cube-map made of shadowmaps for point lights) and light calculations for the final composition of realistic scenes and the generation of reflections in real-time.

2 Related Work

Realistic Real-time graphics in 3D walkthroughs have been promised for many years.

Recently, many game engines (which some are open-source) have been proposed and are available to the gaming community. Some examples are BGE [Blender 2015], Unity [Unity Technologies 2015], Unreal [Epic Games 2015], etc. Those game engines have been offering new opportunities to anyone interested in building visually realistic graphical applications and, consequently, to advance the state-of-the-art in the field.
The simulation of reflections and refractions in the context of graphical applications in real time has been possible using environment maps and GPU, from the technique introduced by Blinn and Newell [Blinn and Newell 1976].

In digital games and interactive 3D applications, it is very common to use the cubemap technique to simulate the effects of reflections. In this technique, a cubemap is generated at the center of the reflective object and in accordance with the normal vectors of the reflection point (the pixel that is being rendered at a given time). A sampling is made in the cubemap [Fernando and Kilgard 2003]. The cubemap consists of six textures generated from six cameras. Each camera points to a different direction, having the same origin. The visual results obtained using cubemaps are usually convincing. However, since it is a very costly process where it is necessary to render the scene six times from the point of reflection, the cubemap is not generated by each object, nor by each frame at runtime.

An alternative solution is to spread out reflection points in the scene to be rendered and force reflective objects (which are closer) to use the closest reflections for the calculations. This approach also generates a reasonable result, however, physically incorrect, since it is only an approximation. Physically correct way to represent reflection, for example, would be to generate it from the surface of the point (or pixel) that is being rendered at a given time.

As regards refractions, Wyman [Wyman 2005] makes real-time refraction look more realistic by introducing a simple, image-space approach that easily runs on modern graphics cards. His method requires two passes on a GPU, and allows refraction of a distant environment through two surfaces, compared to current interactive techniques that are restricted to a single surface.

Sun et al. [Sun et al. 2008] presents also a technique for simulating light through refractive surfaces using the GPU at interative rates. In a more recent work, Mara et al. [Mara et al. 2013] use a two-layer deep g-buffer to simulate lighting effects such as mirror reflections. Although their two-layer deep g-buffer captures occluded objects, the reflected object still has to reside within the camera’s view frustum.

In order to accelerate the process of simulating the reflection, new techniques have been proposed, such as Real-Time Local Reflections or SSR [Mcguire 2014], which simulates the light behavior in a manner equivalent to the ray tracing technique by ray marching in the zbuffer. However, it only works on screen space. The algorithm performs similarly to a post-processing image effect, that is, it is independent of the scene complexity since it uses the g-buffer to do the calculations.

Recently, with the advance of GPUs and parallel processing, researchers again turned their attention to ray tracing [Whitted 1980] for interactive simulation of visual effects, at this time, in the context of digital games and graphics applications in real time [Bikker 2007] [Pohl 2014].

Carr et al. proposed the first ray tracing algorithm on GPU. However, it actually runs only on GPU the calculation of the intersection between rays and triangles [Carr et al. 2002]. Subsequently, Purcell 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 [Purcell et al. 2002].

Some techniques also use pre-computed maps to assist and accelerate the processing of rendering scenes with ray tracing, or cache data between frames [Blinn and Newell 1976], [Miller and Hoff-man 1984]. A graphical application in real time generated with the ray tracing and path tracing algorithms, with rates around 50-60 FPS, has also been reported [Lafortune and Willems 1993].

A graphical application in real time generated with the ray tracing and path tracing algorithms, with rates around 50-60 FPS, has also been reported [Lafortune and Willems 1993]. The library called OptiX [NVidia 2015.a] has emerged to assist programmers in developing more visually realistic graphical applications with ray tracing in real-time in GPU. This library does not necessarily focus on 3D scenes rendering. Actually, it corresponds to a graphics pipeline for a shader-centric ray tracing in which shaders or programs can be written and inserted in different points of the pipeline and can be run almost entirely on GPU.

A similar library was also released, called OpenRL [Imagination 2015], which focus on ray tracing programming using a OpenGL inspired API, making it easier to learn for experienced graphics programmers. Besides that, it is already integrated into Unity 5 [Unity Technologies 2015].

Another library dedicated to high-performance ray tracing has also recently been launched for Intel CPUs, called Embree [Wald et al. 2014].

Also very recently, the use of hybrid solutions combining ray tracing and rasterization have also been explored. For example, in [Johnson 2012], the reflection simulation is done by approximating the SSR technique by using a bounding cube with, which contains associated buffers to their faces representing the scene, i.e., such information is used as a backup way to correct the flaws generated by the SSR algorithm.

Hakura and Snyder [Hakura and Snyder 2001] introduce a hybrid rendering. Actually, a scheme that dynamically raytraces the local geometry of reflective and refractive objects. However, it approximates more distant geometries by using hardware supported environment maps in a preprocessing stage.

Finally, the authors of [Ahtihd et al. 2014] have a hybrid solution that uses a simple heuristic to ignore irrelevant objects in the effects calculation phase using ray tracing. Also, in a hybrid form, Cabeleira [Cabeleira 2010] presents a solution in which all global illumination effects are generated by ray tracing. Part of the process is done on the CPU and another on GPU level. In the end, all this information is combined to generate the resulting image. Ganestam and Doggett [Ganestam and Doggett 2014] also describe a hybrid solution in which the visual effects of objects (that are located close to the camera), stored in a customized Bounding Volume Hierarchy (BHV), are processed by ray tracing and other remaining objects that are distant from the camera’s field of view are calculated by rasterization.

Unlike existing work, this article presents a generic and simple but effective solution to simulate accurately and quickly realistic reflections in 3D walkthrough scenes, without differentiating objects by distance or size on the scene, that is, considering all objects that are part of the reflection.

3 The OptiX Library

The OptiX library can implement many types of renderers. It is a programmable ray tracing framework for software developers. It is used to rapidly build ray tracing applications that yield extremely fast results across NVIDIA GPUs, with CUDA C/C++ programming. A call graph showing the control flow through the NVIDIA OptiX ray tracing pipeline is shown in Figure 1.

This library presents mechanisms for expressing ray-geometry interactions and does not have built-in concepts of lights, shadows, reflectance, ambient occlusion or any other feature for scene rendering.

Its main interests are ray generation, acceleration data structures and how to represent the scene and the rays path in the scene. OptiX follows the same idea of OpenGL, which means that it is possible to insert, at any time, customized programs or shaders to inform what information should be processed through the graphics pipeline. In the work of [Macedo and Rodrigues 2014], different test cases are presented and compared to verify the robustness of the implementations of existing structures and libraries for the process of generating shadows at run time with the ray tracing.

OptiX also offers some methods that can be used to construct the ray tracing acceleration structure, that is only valid with a correct pair of builder and traverser (more details about this can be found in [NVidia 2015.a]).
4 The Generation of Real-Time Reflections

In this section, we describe in detail the simple although effective hybrid solution we have proposed and implemented to generate reflections (Figure 2) in real-time.

Initially, the algorithm loads an obj file with the 3D scene information to be rendered. Then, we create the g-buffer with the diffuse color, worldspace, normal, z-buffer and reflection information.

The reflection calculation consists of two steps: (1) the classic SSR algorithm [Mcguire 2014]; and (2) a pure ray tracer. The complete execution flow diagram of the rendering engine is detailed in Figure 3.

During the first step, we generate two images, one that contains the reflection result based on the reflection information sampled from the g-buffer, as shown in (a) of Figure 3, and a mask that represents in which parts of the image the screen-space method for adding reflections failed, highlighted in (b) of Figure 3.

After that, we start the second step, in which the following information is passed: the 3D scene geometry and the g-buffer.

The ray tracer verifies each pixel of the mask generated during the first step, searching for pixels that were not correctly calculated. If any pixel satisfying this situation is found, a reflected ray from the viewer’s eye is calculated and ray traced, resulting in a new complementary reflection image, as shown in (c) of Figure 3. As a result, the reflection of objects that lie outside the screen-space or that are occluded by other objects can be processed.

With the complementary image completion, our engine combines this image with its screen-space version and applies a Gaussian blur filter [Shapiro and Stockman 2001] to blur the reflection image and minimize small artifacts that may appear due to the merging process of the SSR with the raytraced reflections.

We also apply, in both steps, an attenuation to the reflection based on the length of the reflection ray from the reflected surface (making it fade out along the reflection) for generating a polished visual result, shown in (d) of Figure 3.

The process to generate shadows is done just before the simulation of reflection. Our algorithm uses a zbuffer cubemap to calculate the shadows emitted from a point light. A mask of pixels that are suffering occlusion of light is created, i.e., an image with value 0 for pixels that are in shadow and 1, for otherwise. This mask can be seen in the item (e) of Figure 3.

Finally, all the lighting is calculated and merged with the reflection and shadow masks, resulting in the final composition of the scene, shown in (a) of Figure 4. For comparison, in (b) of Figure 4 we show the same image, rendered without reflections.

5 Tests and Performance Results

Performance testings were conducted as a walkthrough in the Sponza 3D scene from Crytek [Crytek 2010] with 6 additional mesh objects positioned in the middle of the central corridor (4 armadillos and 2 T-rex dinosaurs, with 35k and 20k vertices each, respectively), which totalizes 330k vertices in the whole scene.

For our benchmark, we have used images with a resolution of 1280 × 720 pixels and a camera in motion simulating a FPS gameplay. The 3D scene contains 618,000 triangles and 21 different...
Figure 3: Execution flow diagram of the rendering engine.
frame rates per second higher than the rates obtained purely with the SSR technique. However, even then it still reaches a more competitive FPS, regardless of the animation moment. We can see that our hybrid solution to render reflections shows a better performance than the solution using ray tracing, and that at consecutive frames of the walkthrough it competes fairly with the SSR technique.

In the FPS diagram, there are three ranges of key frames that show important characteristics and, thus, deserve to be discussed: (1) [207, 1443]; (2) [1649, 2576]; and (3) [3503, 4224].

In the first and third intervals, during the animation, the camera is pointing to the side aisles of the scene. We can see that the reflection generated on the corridor floor is made up of simple elements that do not suffer occlusion, which helps the SSR to simulate a large part of the reflections quickly, leaving little scene portions to be rendered by the ray tracing algorithm. That is, that makes our solution more competitive, since it reaches FPS results close to those obtained using a pure SSR technique.

Most importantly, in the second interval, the camera is pointed at the central corridor while it moves along the animation path where the armadillos and dinosaurs are positioned. This is a very interesting take of the animation since those characters attract great visual attention of the beholder.

The floor reflections represent several objects, including those that have complex geometries. This situation is very common in animations and it is inevitable the occurrence of occlusions of parts of objects. In such cases, the SSR algorithm fails to represent the reflections efficiently, generating far too many failures.

Our solution detects these SSR failures at runtime and activates the ray tracing algorithm to solve them. Because of this, our solution decays a bit in terms of FPS. However, even then it still reaches a FPS far superior to the solution purely implemented in ray tracing.

Based on these findings, we conclude that our solution achieves frame rates per second higher than the rates obtained purely with ray tracing. Moreover, we can also clearly note in Figure 6 that the visual quality of the animation frames we generate with our solution is very close to the quality of those generated with ray tracing.

To thoroughly and realistically analyze the behavior of our solution, we separated the main stages of the hybrid rendering engine. Then, we calculated how long each step took to execute their respective tasks, taking our walkthrough as input of our study.

To perform these calculations, we chose an animation moment as reference (the frame 2100), in which there is a very intensive processing load during the walkthrough.

In order to generate the information stored in the g-buffer it was spent 0.25ms and in the step responsible for generating the shadow maps, 0.71ms. To calculate the pointlight illumination, it was necessary 0.03ms.

To generate the SSR reflections and produce a mask containing its identified failures, the rendering engine spend only 0.05ms. The reflection calculation using the ray tracing technique from the SSR fault mask used 12.37ms. Finally, to compose the final image from all the images previously generated, 0.13ms was consumed.

Table 1 shows a comparative study we conducted to identify how similar the walkthrough frames generated using the SSR technique and our hybrid solution are, when compared to the rendering results obtained from the purely ray tracing algorithm (which is considered in this work as a benchmark, since it generates the most accurate and realistic visual results).

To calculate the visual quality differences among the images in a quantitative manner, we used a simple technique of color buckets as follows.

For each image, we test each pixel and for each channel of this pixel, the obtained value is added to the corresponding bucket (R, G or B). After processing the entire image, the total value for each component is compared to the total image value generated by the ray tracing. Finally, we obtain a quality percentage value that represents the range of values when compared to the reference value of the image synthesized with the ray tracing.

In Table 1 we can see that our hybrid solution generates visual quality results always very close to 100%, whereas the SSR in some cases reaches around 97%. This difference of 3% can not be considered, in anyway, as unrepresentative, because the visual quality tests took into account the whole picture and not just the image parts that are suffering reflection.

In (a), (b), (c) and (d) of Figure 6 we also present the visual results of these three solutions for frames 1800, 2100, 2300 and 4400, respectively. It can also clearly be seen that the visual quality of our hybrid solution is very close to the one purely implemented in ray tracing. On the other hand, though the SSR may be quite fast, it shows several flaws in many important animation sequences, which greatly hamper its use in these situations.

### 6 Conclusions and Future Work

This paper presents a new hybrid solution for simulating realistic reflections in 3D walkthrough environments. Our comparative study shows that our hybrid solution presents quality results very close to those obtained with a purely ray tracing algorithm, with a much more competitive FPS, regardless of the animation moment. We would like to combine the strengths of both techniques to achieve a solution that is both fast and realistic.

For future work, we plan to improve our solution further by exploring more advanced ray tracing techniques and optimizing the overall rendering process. Additionally, we will investigate the use of machine learning to automatically detect and resolve SSR failures, further enhancing the performance and visual quality of our system.

### Table 1: Image quality results of four different frames from the walkthrough sequence, in terms of reflections, compared to the ideal corresponding images generated by the ray tracing.

<table>
<thead>
<tr>
<th>#Frame</th>
<th>SSR Algorithm (%)</th>
<th>Our Solution (Hybrid) (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1800</td>
<td>99.18</td>
<td>99.76</td>
</tr>
<tr>
<td>2100</td>
<td>97.68</td>
<td>99.85</td>
</tr>
<tr>
<td>2300</td>
<td>98.58</td>
<td>99.96</td>
</tr>
<tr>
<td>4400</td>
<td>99.88</td>
<td>99.97</td>
</tr>
</tbody>
</table>
Figure 6: FPS and visual quality comparisons of the animation rendered using the SSR technique, our hybrid solution and ray tracing.
believe that it is still possible to make optimizations in the rendering engine, for example, to support baking lighting and shadow for lights and static objects, besides of calculating in real time only what is really necessary to guarantee the visual quality and performance of the animation.

As future work, we intend to expand our testings for dynamic scenes with reflective objects and also to check the possibility of applying some heuristics to consider (or not) the use of the ray tracing algorithm. It may be possible with a heuristic, for example, to make the ray tracing raytraces the pixel only when needed. For example, if the camera is moving very fast in a game, it is probably not fundamental to produce extremely realistic reflections, since the player probably will not be able to identify failures and gaps on the fly easily. That is, it may be interesting to develop also an adaptive solution for our hybrid algorithm directed by the player’s behavior. In the near future, another improvement we plan is to develop a solution for our hybrid algorithm directed by the player’s behavior.

Unfortunately, as a closed-source library, the OptiX does not allow to customize the testing codes, forcing the developers to use only the data structures currently existing in the library.

Acknowledgements

Daniel Valente de Macedo and Maria Andrade Formico 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


