Universidade Federal Fluminense

Christian Frantz Ruff

Dynamic per Object Ray Caching Textures for Ray Tracing

NITERÓI

2013

Universidade Federal Fluminense

Christian Frantz Ruff

Dynamic per Object Ray Caching Textures for Ray Tracing

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic Area: Visual Computing.

Advisor:
Prof. D.Sc. Esteban Walter Gonzalez Clua

Co-advisor:
Prof. D.Sc. Leandro Augusto Frata Fernandes

NITERÓI

2013

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

R922 Ruff, Christian Frantz

Dynamic per object ray caching textures for ray tracing / Christian Frantz Ruff. – Niterói, RJ : [s.n.], 2013.

69 f.

Dissertação (Mestrado em Computação) - Universidade Federal Fluminense, 2013.

Orientadores: Esteban Walter Gonzalez Clua, Leandro Augusto Frata Fernandes.

1. Computação gráfica. 2. Traçado de raios. 3. Tempo real. I. Título.

CDD 006.6

Dynamic per Object Ray Caching Textures for Ray Tracing

Christian Frantz Ruff

Thesis presented to the Computing Graduate Program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic Area: Visual Computing.

Niterói, December 09th, 2013.

“Anyone who has never made a mistake
has never tried anything new.”
Albert Einstein

Acknowledgments

I would like to thank my parents, André Fernando Ruff and Patrícia Frantz Ruff, and my sister, Priscila Frantz Ruff, for all the support and love given. I would never get this far without your help. I love you very much!

To Esteban Walter Gonzalez Clua and Leandro Augusto Frata Fernandes, for all your patience and support. Thank you for believing in me and helping me finish my masters degree.

To my roomates, André Luiz Brandão and Giancarlo Taveira, for your friendship and support throughout these three years.

To all my colleagues at MediaLab, for helping me in many problems that I had along the way. Thank you all!

To Thales Luis Sabino, for lending me his ray tracing hybrid solution that was used in this work. Thank you for all the tips and advice given about your work.

To my colleagues and superiors of the navy simulator, for the acquired experience and for the opportunity to be an employee in such an amazing project.

To Universidade Federal Fluminense, for providing me a work environment, and CAPES, for the scholarship.

To everyone else that supported me, thank you!

Resumo

A técnica de traçado de raios torna possível a renderização de cenas com intensas interações de luz. A abordagem se baseia na ideia de que a reflexão, refração e sombra podem ser computadas ao seguirmos recursivamente o caminho que a luz percorre dentro de um ambiente. No entanto, apesar do conceito parecer simples, o traçado de raios é uma tarefa de alto custo computacional. Além disso, otimizar o gerenciamento de memória para aumentar a eficiência desta técnica é difícil, pois o acesso coerente no espaço tridimensional não garante o acesso coerente à memória.

Este trabalho apresenta uma nova estratégia que utiliza *cache* de texturas para o traçado de raios, capaz de armazenar dados gerados em quadros anteriores, tornando possível o acesso coerente à memória uma vez que os dados são utilizados em quadros subsequentes. Ao armazenarmos os resultados dos raios traçados em um *cubemap* que está atrelado a cada objeto na cena, nós mostramos que é possível explorar o mecanismo de amostragem eficiente da memória fornecido pelo hardware gráfico com o intuito de aumentar a taxa de quadros por segundo da renderização. A técnica de *cache* de raios é utilizada com duas estratégias diferentes. A primeira armazena somente a informação de cor do traçado de raios, com procedimentos simples para armazenar e recuperar a informação. A segunda, armazena a cor e a profundidade do reflexo e usa a técnica de mapeamento com relevo para armazenar e recuperar a informação.

A técnica proposta pode ser utilizada em cenas estáticas e pode prevenir a profunda interação dos raios com a cena, bem como permitir a computação síncrona dos raios em arquiteturas paralelas. Além disso, ambas as estratégias podem ser facilmente integradas a qualquer solução de traçado de raios já existente.

Palavras-chave: traçado de raios, memória reserva, mapeamento cúbico, mapeamento com relevo, tempo real.

Abstract

Ray tracing allows scenes with very complex light interactions to be rendered. It is based on the idea that reflection, refraction and shadows can be modeled by recursively following the path that light takes as it bounces through an environment. However, despite its conceptual simplicity, tracing individual rays is a computationally intensive task. Additionally, optimizing memory management to increase efficiency is difficult because coherent access in 3-dimensions does not guarantee coherent memory access.

This work proposes a ray caching technique suitable for interactive and real-time ray tracing that is capable of storing the data generated in previous frames in such a way that coherent memory access is achieved while the data are reused by subsequent frames. By storing the light bounce results of previously traced rays in a cubemap attached to each scene object, we show that it is possible to explore the efficient memory sampling mechanism provided by the graphics hardware to increase the frame rate. The ray caching technique is used in two different strategies. The first one only stores the color information of the ray tracing, with simple storage and retrieval procedures. The second one stores the color and the depth of the reflection information and uses relief mapping to store and retrieve the information, which improves parallax effects.

Our technique is suitable for static scenes and may prevent deep interactions of rays with the scene as well as enable the synchronous computation of rays in parallelized architectures. Additionally, both strategies can be easily integrated into any existing ray tracing solutions, especially those that require real time or interactive frame rates.

Keywords: ray tracing, cache memory, cube mapping, relief mapping, real time.

Contents

Glossary

BVH	:	Bounding Volume Hierarchy
CCT	:	Color Caching Technique
CDCT	:	Color and Depth Caching Technique
FPS	:	Frames Per Second
GLSL	:	OpenGL Shading Language
GPU	:	Graphics Processing Unit
LBVH	:	Linear Bounding Volume Hierarchy
RGB	:	Red, Green and Blue Channels
RGBA	:	Red, Green, Blue and Alpha Channels
SBVH	:	Split Bounding Volume Hierarchy

List of Figures

List of Tables

Chapter 1

Introduction

Computer-generated images of outstanding quality and unsurpassed realism have been created through the use of ray tracing [Whitted 1980]. This technique is based on the idea of tracing the path of individual light rays from the eye into the scene and calculating the effects of their interactions with the environment. The recursive task of tracing a ray consists of traversing the 3-dimensional space until the ray hits an object and generates up to three new types of rays leaving the intersection point: a reflected ray continues on in the mirror-reflection direction from a shiny surface, a transmitted ray travels through transparent materials, and a shadow ray is used to test if a surface is visible to a light source. For each ray-surface interaction, an intensity is computed, added to the color of the pixel related to the ray, and some energy is lost. The creation of the reflected and transmitted rays stops when the computed intensity becomes less than a certain threshold. While the technique is capable of producing a high degree of visual realism, the large number of rays, intersections and recursive calls has a large computational cost. For this reason, ray tracing has been best suited for applications that do not require real-time or interactive frame rates, such as still image creation and cinematography visual effects.

Because light beams can be treated independently from each other, ray tracing can be trivially parallelized using threads in a naïve way. However, the incoherence of the direction of neighboring rays makes it difficult to coalesce memory access and data arrangement. Additionally, the independence of the rays' computation naturally leads to different efforts when calculating paths and usually results in idle threads. Because the rays related to neighbor pixels travel through different levels of the data structure used to arrange the objects (e.g., octree), the algorithm is prone to high a degree of thread disparity, making the optimization of the code in the Graphics Process Unit (GPU) architecture a difficult task. The existence of efficient ray-caching mechanisms to ensure memory coherence and to avoid inter-frame redundant ray computation could significantly increase the performance of ray tracing algorithms by making the different rays shot towards the same object return synchronously, reducing the number of threads and the time they are kept in an idle state and maximizing the efficiency of the GPU.

In this work, we introduce a novel and efficient technique that manages caching textures of rays. This technique explores the idea that, in a specific frame, many reflexive and transparent recursive rays are recomputed, although they were already calculated in previous frames. Our approach, entitled Per Object Ray Caching, is capable of storing the reflection information computed by tracing the rays in previous frames and reusing it while producing the next frames of the sequence. Although our technique can be used in any ray tracing approach, it is especially suitable for real time and GPU based renderers. The main idea is to assign to each reflective object a cubemap, i.e., six textures that are mapped onto a cube to create a 360 degree panorama, to store the reflection information calculated by the rays leaving the object in a given direction. The cached information is dynamically completed in the render time and then used in subsequent frames to avoid retracing redundant reflected and refracted rays. Thus, before casting new rays into the environment, the technique tries to fetch the reflection information in the ray's direction from the caching cubemap textures. If the information is available, the algorithm returns the color, and the recursive tracing of the respective ray stops. When the reflection information is not available, the ray tracing proceeds normally, and the resulting color is used to update the caching textures.

In our work, we chose to create two different strategies for the Per Object Ray Caching technique. In the first one, entitled the Color Caching Technique (CCT), the only information stored is the color of the reflection. In the second strategy, called the Color and Depth Caching Technique (CDCT), we also store the depth information in addition to the color information. Because the information in these caching textures is not the same, the way that we store and retrieve data from them is also different.

Ray tracing is a very well-known technique that can render scenes with complex light interactions. Because of that, many techniques have been created to make the ray tracing algorithm less computationally intensive. Some of them focus on the geometry, creating new data structures so that the rays can travel more efficiently through the scene and detect intersections more quickly [Glassner 1984, Foley and Sugerman 2005, van Reeth et al. 1996, Stich et al. 2009, Lauterbach et al. 2009]. Some are more interested in guaranteeing coherent memory access through methods such as improving the cache performance [Pharr et al. 1997, Aila and Karras 2010, Yang et al. 2013]. Other researchers, such as Sabino et al. [Sabino et al. 2012], are interested in optimizing the ray tracing algorithm itself, using the GPU's high level of parallelism to create a hybrid rasterized and ray tracing algorithm. Chapter 2 presents these related works. It discusses the techniques for optimizing ray tracing, previous attempts to achieve coherent access to 3-dimensional space and memory, and the local storage of rendered data for computing lighting effects.

1.1 Main Idea

Ray tracing techniques have a high computational cost because of the large number of rays, intersections and recursive calls computed. This work proposes the use of an efficient caching technique to store the information computed by a ray tracing solution and to use this information in later frames to avoid the computing of redundant rays. The key observation is that "*less rays to be computed will make the ray tracing technique a less intensive task.*" Thus, it is intuitive that the frame rate will increase. However, the quality of the reflection can be affected. Therefore, this work intends to present two different strategies (CCT and CDCT) for the cached information, to show that the Per Object Ray Caching technique can be used to eliminate the computation of redundant rays in a ray tracing solution. We also intend to analyze the quality of the reflections produced by each of proposed strategies.

Some of the primary challenges include the following:

- Storing and retrieving the color information of the ray tracing using the Color Caching Technique (CCT).
- Storing and retrieving the color and depth information of the ray tracing using the Color and Depth Caching Technique (CDCT).
- Combining these two approaches with an interactive ray tracing solution, completely based in the GPU.

The structure used to store and retrieve the information is a cubic box attached to each one of the reflective objects in the scene. The cubic boxes have six faces, and a caching texture is assigned to each one of these faces. These textures store the reflection information from the ray tracing. The first proposed approach, the CCT, has a very simple caching strategy. It retrieves and stores the color information of a given ray direction in the respective texel of the caching texture. The proposed structure and the Color Caching Technique are described in Chapter 3.

Our second strategy presented, the Color and Depth Caching Technique, uses the same cubic structure described earlier. The difference is that the depth information is also stored with the color. The information is stored in the same way as in the CCT strategy. In the CDCT, the technique known as relief mapping, proposed by Policarpo et al. [Policarpo et al. 2005], is used to retrieve the information from the caching structure. The relief mapping creates an augmented texture that generates a tridimensional detailed surface for the current view, which, in our case, will be the center of the object. The relief mapping had to be adapted because it was designed to work with texture with depth information captured from an orthogonal projection, rather than from a perspective projection, as utilized in our technique. These subjects are discussed in Chapter 4.

The two proposed strategies, the CCT and the CDCT, are inserted and executed in a ray tracing solution. When executing the ray tracing technique, after we calculate the direction of the reflection/refraction ray, instead of tracing it, we check if the information is available in our structure. If it is, we return that information without tracing the ray. Otherwise, the ray tracing executes normally and returns the color of the reflection, which is later stored in our structure. The idea is to completely fill the textures, avoiding the retracing of redundant rays by the ray tracing.

1.2 Contributions

The main contribution of this work is the ray caching technique, entitled Per Object Ray Caching [Ruff et al. 2013]. It provides a dynamic construction and usage of cubemaps attached to scene objects to store rendered generated data for use in subsequent frames of ray tracing frame sequences. This simple but effective strategy prevents the creation of redundant rays and deep interactions with the environment within subsequent frames. Our approach allows the synchronous computation of rays cast to the same object, maximizing the efficiency of the GPU. The approach also improves memory coalescence by converting the task of traversing the scene into the simpler problem of fetching coalescent regions of textures as in rasterization-based reflection mapping rendering. We have implemented and tested the solution to demonstrate that our approach works well in static scenes with convex objects and that it is suitable for rendering in ray tracing systems, especially when real time is required.

Ray tracing in static scenes can be useful in many situations, such as walkthrough in indoor scenes, rendering and inspection of large polygon datasets, such as oil platforms. Video games traditionally separate and mark objects that are static for a specific treatment, being able to include at them specific optimizations, such as the proposed in this work.

In this context, the contributions of this work include the following:

- A caching technique that avoids the computation of redundant rays of a ray tracing solution.
- A simple caching technique that stores the color information using the proposed approach.
- A caching technique that stores the color and depth information using the proposed approach based on the relief mapping technique.

In addition to these original contributions, assertions about the proposed approach include the following:

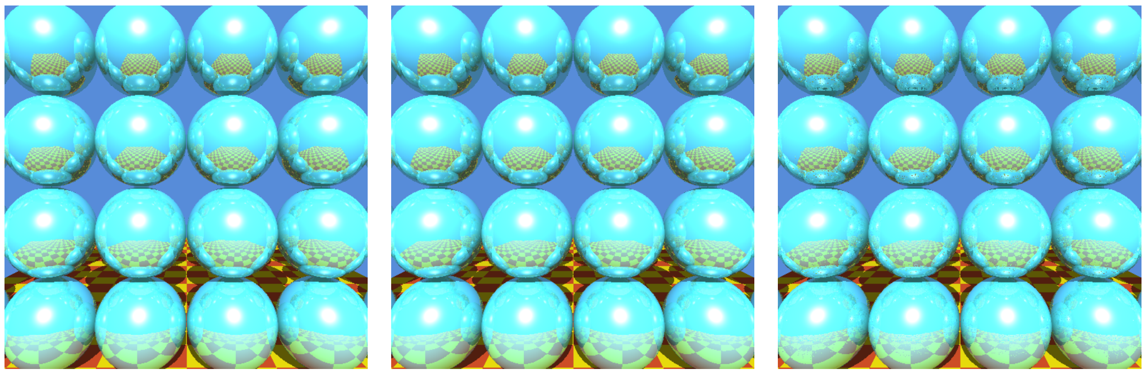
- Although we focus on real time ray tracing systems, the two proposed caching techniques can be easily combined with any ray tracing solution, optimized or not.
- The relief mapping technique can be used with depth textures captured with perspective projection.

It is important to emphasize that our caching approach is not dependent on any of the previous works and that it can be considered an optimization of the ray tracing algorithm. The concept of using textures or data structures to store information of any type and use it later is widespread in the community.

Finally, we believe that this work provides enough information to show that the proposed ray caching technique can be used to achieve the same quality as a state-of-the-art ray tracing solution with a better frame rate, allowing complex scenes to be rendered faster.

1.3 Demonstration and Validation

The CCT and the CDCT were tested with a variety of scenes containing diffuse materials, reflective materials and objects with many shapes and sizes. The techniques were implemented in two different ray tracing solutions. The first one is a state-of-the-art ray tracing solution implemented over a ray tracer engine [Parker et al. 2010]. The second is a ray tracing optimization exclusively executed in a GPU [Sabino et al. 2012]. The intention is to show that our technique can be easily combined with other ray tracing solutions. Figure 1.1 shows the comparison between a standard ray tracing solution, the CCT and the CDCT.



(a) Conventional (b) CCT (c) CDCT

Figure 1.1: A comparison between ray tracing and the two proposed strategies. In (a), we have the results of the conventional ray tracing technique. In (b) and (c), we have the results of the ray tracing combined with the CCT and with the CDCT, respectively.

To evaluate the two proposed strategies, the Per Object Ray Caching technique described in this work was compared against the solutions used as base for our implementation. The aspects compared between them were the frame rate and the reflection quality. For the frame rate comparison, several scenes were created and tested with each one of these techniques. The quality of the reflection was compared by subtracting the output generated from the base ray tracing solutions with the output of our techniques. The results generated an image that represents the differences between the outputs, based on the color of the texels. These subjects are presented in Chapter 5.

Chapter 2

Related Work

The ray casting technique was proposed by Appel [Appel 1968]. The idea behind this technique is to shoot rays from the camera, one for each pixel of the image, and to find the closest object that intercepts its path. Using the material property and lighting properties in the scene, this technique can determine whether that specific pixel is in a shadow or not. Thus, if a ray cast from the surface intercepts the light source, the light will reach that surface and not be blocked or in shadow.

A ray tracing algorithm was first presented by Whitted [Whitted 1980]. The research breakthrough of this work is that after a ray hit an object, the light ray may continue in a recursive process. After a ray strikes a surface, it can generate three types of rays: reflection, refraction, and shadow. The reflection continues in the mirror-reflection direction from a shiny surface. Refraction rays traveling through transparent material work similarly, although the refractive ray could be entering or exiting a material. The shadow ray is used to test if the surface is visible to light. If the surface is in a shadow, the tracing of recursive rays will be avoided. If the surface faces a light, a ray is traced between this intersection point and the light. If any opaque object is found in between the surface and the light source, the surface is in shadow, so the light does not contribute directly to its shade.

In the ray tracing technique, the resulting color of each pixel in the visualization area is determined by the adopted illumination model, which is flexible enough to add or remove components as needed. In [Phong 1975], the author presents a very basic model that has been improved over the years with the addition of new components.

Figure 2.1 illustrates the ray tracing algorithm. C is the camera from which the ray originates, and P represents the sampled pixel in the visualization area. L is the light source, and S indicates shadow rays that are traced to check if the surface is facing the light source. R represents the reflection rays that continue in the mirror-reflected direction from a reflective object.

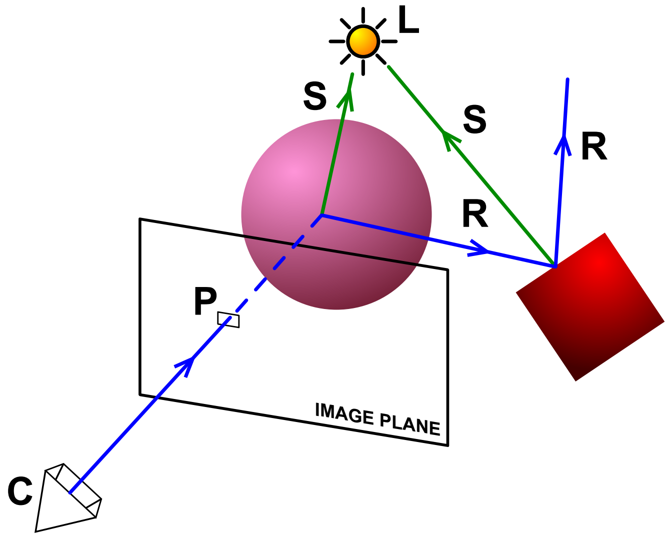


Figure 2.1: The ray tracing algorithm. A ray originating at the camera C through pixel P hits an object in the scene. R represents the reflection rays that continues on in the mirror-reflected direction from a reflective object. A ray S is also shot to check if the pixel is facing the light source L.

The ray tracing technique is known for the complexity of light interactions that it can render in scenes. Unfortunately, it has a very high computation cost. Because of the cost, many studies have tried to make the ray tracing algorithm less computationally intensive. Some have focused on the geometry, creating data structures to make rays travel more efficiently throughout the scene and detect intersections more efficiently [Glassner 1984, Foley and Sugerman 2005, van Reeth et al. 1996, Lauterbach et al. 2009]. These types of optimization will be described in Section 2.1. Some other studies, were more interested in guaranteeing coherent memory access, such as improving the cache performance [Pharr et al. 1997, Aila and Karras 2010, Yang et al. 2013]. Works related to memory optimization techniques will be described in Section 2.2. Another work, performed by Sabino et al. [Sabino et al. 2012], was more interested in the high level of parallelism of the GPU by creating a hybrid rasterized and ray tracing optimization. This particular work was one of the projects used in combination with our implementation and developed by the same group. The hybrid solution created by Sabino et al. will be described in Section 2.3. The concept of using textures or data structures to store information of any type and be consulted them later is seen in many works [Catmull 1974, Ward 1994, Jensen 2001], which will be described in Section 2.4.

2.1 Structure and Geometry-Based Optimizations

Spatial data structures have been widely used to optimize the interaction between traced rays and scene objects. In [Glassner 1984], the author proposed the traversal of an octree while computing the intersection of rays and the environment. Foley and Sugerman [Foley and Sugerman 2005] were the first to demonstrate k-d tree traversal algorithms suitable for the programmable rendering pipeline of GPUs and to integrate them into a streaming ray tracer. Complete GPU implementations of ray tracers that use an octree as an acceleration structure were also developed in the CUDA architecture by Barboza and Clua [Barboza and Clua 2011]. Interactive ray tracing of moderate-sized animated scenes was achieved by Wald et al. [Wald et al. 2006] by traversing frustum-bounded packets of coherent rays through uniform grids. Regular grids and quadtree structures were also investigated by van Reeth et al. [van Reeth et al. 1996].

The ability to efficiently rebuild the grid on every frame enabled the treatment of fully dynamic scenes that are typically challenging for k-d tree or octree-based architectures. However, state-of-the-art ray tracing engines such as OptiX [Parker et al. 2010] use more sophisticated acceleration algorithms, such as the Bounding Volume Hierarchy (BVH). Of these algorithms, the Split Bounding Volume Hierarchy (SBVH) algorithm [Stich et al. 2009] has been used because its data structure is simple to construct, it has low memory footprint, it allows refitting in animations, and it works well with packet tracing techniques. The idea behind this structure is to split a given node using either object list partitioning or spatial partitioning by selecting the more cost-effective scheme. In the same way, the Linear Bounding Volume Hierarchy (LBVH) algorithm [Lauterbach et al. 2009] has recently been attracting increased attention because it is focused on minimizing the cost of construction, while still producing BVHs of good quality. It uses spatial Morton codes to reduce the BVH construction problem to a sorting problem. The splits of the hierarchy are executed in parallel, removing many of the existent bottlenecks in the construction algorithm.

2.2 Memory-Based Optimizations

Unfortunately, efficient spatial data structures may not guarantee coherent memory access, because the rays may bounce in random directions. In the literature, it is possible to find several ray tracers developed with this goal. Pharr et al. [Pharr et al. 1997] described algorithms that use caching and lazy creation of texture and geometry to manage scene complexity. To improve the cache performance, they increased locality of references by dynamically reordering the rendering computation based on the contents of the conventional cache. Aila and Karras [Aila and Karras 2010] proposed a massively parallel hardware architecture based on a hierarchical treelet subdivision of the acceleration structure and the repeated queuing and postponing of rays to reduce cache pressure (i.e., the shortage of cache memory). As a result, the authors reduced the total memory bandwidth. In a recent work, Yang et al. [Yang et al. 2013] presented efficient data and task management schemes designed for GPU-based ray tracing. Their approach uses fuzzy spatial analysis, a two-level ray sorting method, and a ray bucket structure to reorganize ray data. By doing so, the threads can be scheduled to achieve coherent access to the geometry and to reduce the memory bandwidth. Our approach uses the spatial data structures provided by OptiX to handle geometry, and no data reordering is necessary to achieve coherent access to the rays cached in previous frames. We explore the existing texture mapping functionalities of the GPU to store and sample the cached rays using textures in a rasterization-based rendering.

The techniques presented in Section 2.1 and 2.2 intend to optimize the spatial data structure or the geometry of the scene using trees and others structures to do so. Alternatively, our approach is focused on optimizing the ray tracing by eliminating redundant rays. These two types of optimization have no dependencies on each other, meaning that all the referred techniques can be combined with our approach to achieve even greater results. In fact, the LBVH and the SBVH techniques were used in several of our tests because they are easy to use and already built in to the OptiX engine.

2.3 Hybrid Optimization

While many works are interested in optimizing ray tracing by using new data structures and geometry arrangement, others are interested in different approaches, such as using the GPU high parallelism of the rasterization pipeline as part of the process. [Sabino et al. 2012] created a hybrid rasterized and ray tracing algorithm completely executed in a GPU. The basic idea is to use the deferred rasterization-based technique to compute the shading of the primary rays, followed by a ray tracing phase that computes the recursive ray effects such as transparency and specular reflection. Each stage produces an individual image as output, requiring one more stage for combining the two generated images to achieve the final result. The resulting image contains ray traced effects that are not trivially obtained with rasterization, such as reflections, refractions and shadows.

The hybrid rasterized and ray tracing pipeline proposed by Sabino et al. has a sequence of phases. Figure 2.2 illustrates the partial results of each one of these phases described below:

- **Deferred rendering and primary ray computation:** The first phase consists of the rendering of the scene using the deferred shading, filling the G-Buffer with information. This G-Buffer is a collection of the scene's information that is being rendered in the pipeline, including position, normals, depth and albedo. After the G-Buffer has been filled, it contains information about the visible fragments and can be considered the resulting image of the primary rays in a conventional ray tracing (see Figure 2.2a).
- **Shadows:** With the information from the G-Buffer, shadow rays are generated only for valid points. A valid point is one that represents surface information; invalid points are those that represent the background. In Figure 2.2a, invalid points are those that must be illuminated with the ambient map, represented by the blue points. The result of this phase can be seen Figure 2.2b.
- **Reflections and refractions:** In this phase, ray tracing is used to trace the reflections and refractions through the scene. The resulting image is represented in Figure 2.2c. Note that the floor has a reflective material, used to illustrate the application of the reflection effects.
- **Final composition:** The final phase of the algorithm is the composition of the stages represented by Figures 2.2a and 2.2c, i.e., the deferred rendering calculation and the reflection/refraction computation. The two images are composed using a shader program, taking advantage of the GPU hardware specialized in this task. The final image can be seen in Figure 2.2d.

The results achieved by this technique are impressive. The average speed up is approximately two times the frame rate achieved by a conventional GPU ray tracing. There are no differences when comparing the quality of the final images produced with the conventional and hybrid approaches. Because there is no simplification in the ray tracing technique. All the existing features in the conventional ray tracing are also present in the hybrid implementation.

In the Chapter 5, many of the tests of our approach were performed by combining the hybrid technique with our strategy and then comparing it with the standard hybrid technique. The implementation was done using the OptiX ray tracing engine.

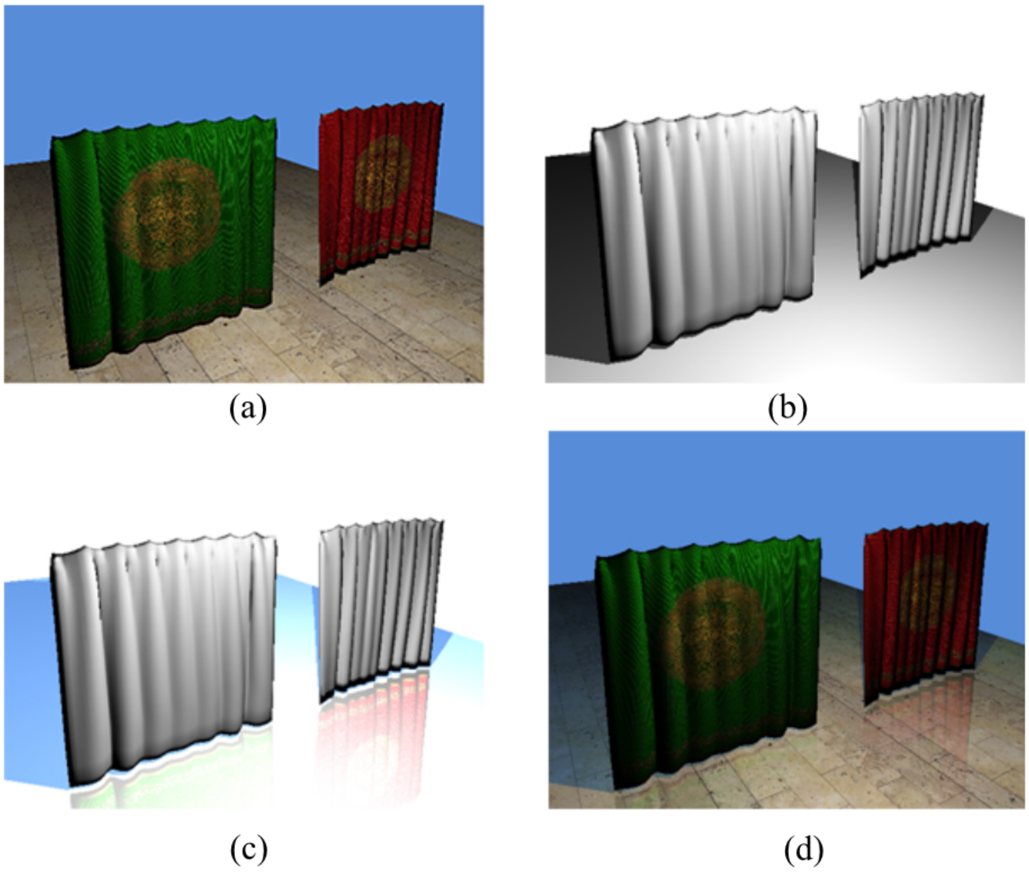


Figure 2.2: The hybrid pipeline overview [Sabino et al. 2012]. In (a), we have the trace of the primary rays, done by the rasterization-based deferred rendering technique. In (b), the shadow rays are computed and in (c), the ray tracing algorithm is used to compute the rest of the rays. In (d), we have the final result, created by a composition of (a) and (c).

2.4 Texture-Based Techniques

The idea of storing information in textures and using it for computing lighting effects is widespread. In 1974, Catmull [Catmull 1974] developed a new algorithm for rendering images of bivariate patches. Years later, Blinn and Newell [Blinn and Newell 1976] presented an extension of the algorithm in the areas of texture simulation and lighting models, developing the environment mapping. It can even now be considered an efficient image-based lighting technique for approximating the look of a reflective surface by means of a pre-computed texture image. This texture is used to store the image of the distant environment surrounding the rendered object. There are several ways to handle the environment information as textures. The first technique was the sphere mapping proposed by Blinn and Newell, in which a single texture contains the image of the surroundings as reflected on a mirror ball. It has been almost entirely surpassed by cube mapping [Greene 1986], in which the environment is projected onto the six faces of a cube and stored as six square textures or unfolded into six square regions of a single texture. Figure 2.3 illustrates the cube mapping process. A ray launched from the camera hits the object at a certain point. A reflected ray continues on in the mirror-reflection direction. This direction is then used to sample the information of the cube mapping. The sampled texel will be the pixel seen by the camera.

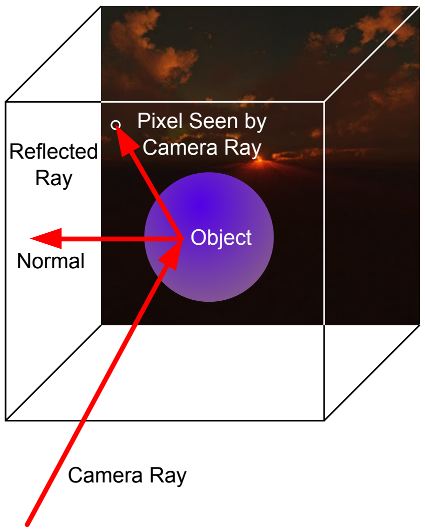


Figure 2.3: The cube mapping technique. A ray that comes from the camera hits the target. The reflected ray is created by rotating the camera ray around the normal. The pixel seen by the camera ray will be mapped onto the surface of the object.

The environment mapping is an efficient image-based lighting technique used to approximate the appearance of a reflective surface. The reflection color used in the shading computation at a pixel is determined by calculating the reflection vector at the point on the object and mapping it to the texel in the environment map. This technique often produces results that are superficially similar to those generated by ray tracing, but it is less computationally expensive because a texture lookup is much simpler than following the recursive rays traced through the scene geometry. The main drawback of this technique is that the environment around the object is the only aspect mapped. Additionally, there is no self-reflection or reflection of a nearby object. Miller and Hoffman [Miller and Hoffman 1984] extended Blinn and Newell's work to cover a wider class of reflectance models. Panoramic images of environments were used as illumination maps that were blurred and transformed to create reflection maps. Miller and Hoffman were the first to use perspective projection onto cube faces. A couple years later, Greene [Greene 1986] created the well-known cube mapping. The main difference between the conventional cube mapping and our approach is that in the cube mapping, the textures are filled with information in a preprocessing stage, while in ours, the textures are dynamically filled with the ray tracing information that is being computed on-the-fly.

The storage of the rendered data on the object's surface was explored in [Ward 1994], that describes a physically based rendering system to create the radiance effect on objects. In that case, the information of the radiance is stored in structures to be later combined with the objects in the scene. Jensen [Jensen 2001] developed the photon mapping technique that enable the efficient simulation of global illumination in complex scenes. It is divided into two stages, the construction of the photon map and the rendering. In the first stage, the light sources send light packets called *photons* into the scene. When a photon intersects an object, this intersection point and the incoming direction are stored in a cache called the *photonmap*. In the second step, the photon map created is used to calculate the radiance of every pixel in the final image. The ray tracing technique is used to find the closest surface of intersection. The photon mapping can simulate caustics, diffuse inter-reflections, and participating media such as clouds and smoke. The information of these effects is stored in temporary data structures to be used later to generate the final image. In contrast to these techniques, our approach writes the traced rays in the proposed cached memory. Additionally, the data are generated and updated in render time.

Chapter 3

Per Object Ray Caching

The approach uses 2-dimensional caching textures (Section 3.1) to store the color of recursive rays calculated by a ray tracer. Each reflective object of the scene contains a particular set of six caching textures, which we call a *caching cube* (Section 3.2). For each frame, before tracing a ray leaving the object, the algorithm verifies whether the color information for that ray is already available or not using the caching cube of the object. If the color is not available, the ray tracer calculates the ray's color using the conventional ray tracing technique and updates the caching cube with the new color information (Section 3.3). However, when updated color information is available in the caching cube, the algorithm returns the stored color, and the ray tracing does not need to be evaluated for that ray (Section 3.4). The algorithm for the technique can be seen in Algorithm 1. Because we are storing information for later use, it is clear that the more the camera moves around the scene, the more stored information we will have, allowing us to increasingly avoid the ray tracing calculation after some scene navigation.

The caching technique proposed in this work can be considered a simplification of the plenoptic function, defined by Adelson and Bergen [Adelson and Bergen 1991] for the subspace of the reflective points. The plenoptic function is a parametric function that describes everything that is possible to see, from anywhere in the scene, at any time. In McMillan and Bishop [McMillan and Bishop 1995] this function is defined by:

$$p = P(\theta, \phi, \lambda, V_x, V_y, V_z, t) \tag{3.1}$$

where (V_x, V_y, V_z) is the position from where the observer is at, θ and ϕ are the azimuth and elevation angles, and λ is the light frequency to be observed. For dynamic scenes, t describes the time.

It can be said that this function is able to describe all possible views of a given scene. Implement such function for a real-time ray tracing would be impractical. In our work, since we use only static scenes, t is constant. Also, the values V_x, V_y and V_z are constants, because we do not store the position from where the information was taken. The values of θ and ϕ describe the direction where we store and retrieve the information, and λ is the information stored. For this reason, we assume that the information stored in the textures are an approximation of the values calculated by a ray tracing solution, but for the purpose of real-time, the obverser can hardly notice the difference because the results achieved are very coherent.

The process of consulting a texel on the caching cube is much faster than computing the color information through a ray tracing procedure. Because the ray tracing algorithm will be executed more often in the first frames than the simpler texture lookup, it is supposed that when a new reflective object appears before the camera, there will be a decrease in the frame rate. However, as ray tracing fills the textures of the object's caching cube, the fetching procedure returns valid results so that the number of ray tracing calls reduces. Stochastically, new reflective objects will appear at the camera frustum in a well distributed manner, so that while new objects will require ray tracing calls, older ones will primarily use the cached rays. The technique presented in this chapter, as well as some of the results of Chapter 5, was published in [Ruff et al. 2013].

Algorithm 1: The algorithm of the Per Object Ray Caching technique.

```
input       : $rayDirection$
output      : $reflexColor$
rayDirection: The direction of the reflected ray
reflexColor : The reflection color stored in caching cube

1 $info \leftarrow$ RetrieveInformation($rayDirection$)
2 if $info$ is valid then
3     $reflexColor \leftarrow info$
4 else
5     $reflexColor \leftarrow$ RayTracing($rayDirection$)
6     StoreInformation($reflexColor$, $rayDirection$)
7 end
```

3.1 The Caching Textures

In this work, we use 2-dimensional $RGBA$ textures to store the color information of the ray tracing. The color information that is stored in these textures is the pure color of the reflection in addition to the shadow information, which means that the specular highlight will not affect the stored information. Since static scenes do not have dynamic lights, the shadow information will also be stored in the same way light maps deal with. The specular highlight however, will be ignored because it depends on the position of the observer and the light so that, if the camera moves around the scene, the position of the specular highlight will change, leading to inconsistency in the results. For the caching technique described in this chapter (the CCT), the alpha channel of the textures will not be used for transparency, because the reflection colors have no transparency at all. The alpha channel will store a flag to identify whether the stored information is valid. For alpha equal to zero, the information is invalid and if it is equal to one, the information is valid. This is important because we are filling the textures dynamically, and thus, all the textures need to be initially marked as invalid. After storing a valid color in the texture, we change the flag identification to valid, so when retrieving the information, we only do so for a valid color. The size of these textures will greatly affect the quality of the reflection. The chosen values are between 128 by 128 and 512 by 512. Below 128 by 128, the quality of the reflection returns very poor results, and above 512 by 512 there is little observable difference, but the memory used for allocation increases greatly.

3.2 Caching Cube Setup

We define a cubic box for each reflective object in the scene. This box will not be affected by the rotation of the object, so if the object inside moves around the scene and rotates around itself or around other objects, the box will follow the object's position but not the orientation. Additionally, both object and box are centered on the origin of the object coordinate system. Figure 3.1 shows a representation of the cubic box around a reflective sphere. The faces of the box are oriented according to the cubemap texture convention, i.e., the faces' normals pointing inside the cube. In our implementation, the cubic box is not created as a piece of geometry. It is only a convention about the geometrical relation between the object and its respective caching cube.

The caching cube setup consists of creating six caching textures, one for each face of the box. It is easy to see that our technique uses more memory than a common ray tracing. The greater the number of reflective objects in the scene, the greater the memory used to store these textures. The use of the caching cube is a trade-off between speed and memory in favor of speed. However, we remark that these textures remain in the GPU memory, which is abundant in modern architectures. Actually, for this reason, many

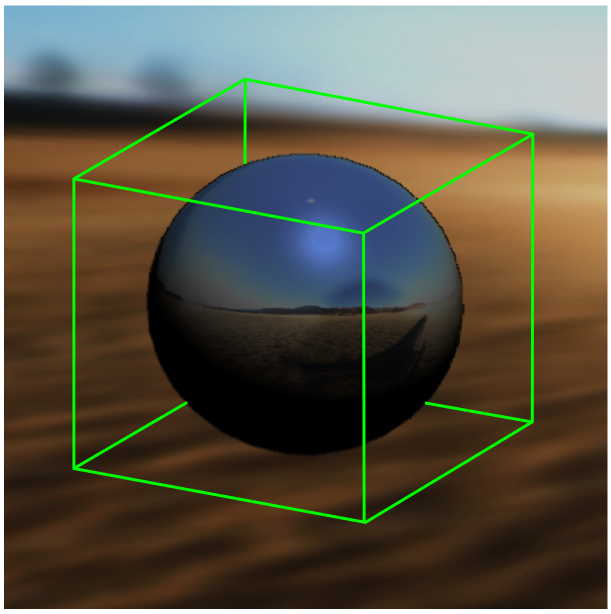


Figure 3.1: An axis-aligned cubic box assigned to a reflective sphere in object space during the loading of the scene. The faces of the box coincide with the faces of the caching cube.

other optimization techniques currently use extra textures. In Chapter 5, we discuss the frame rates and the quality of images produced by conventional ray tracing and by the use of caching cubes having different resolutions.

3.3 Caching the Reflection Information

Once the caching cube is created and initialized, it is ready to store the reflection information. The caching procedure is performed with the ray tracing. For each ray leaving the object, we compute the direction of the reflected (3.2) or refracted (3.3) ray using standard formulas [Pharr and Humphreys 2004]:

$$\vec{r} = \vec{i} + 2\,\vec{n}\,\cos\theta_i \tag{3.2}$$

$$\vec{t} = \eta\,\vec{i} + \left(\eta\,\cos\theta_i - \sqrt{\cos^2\theta_t}\right)\,\vec{n} \tag{3.3}$$

where $\vec{i}$ is the direction of the unitary incident ray pointing to the surface, and $\vec{n}$ is the unitary surface normal. θ_i and θ_t are, respectively, the angles of incidence (3.4) and transmission (3.5), leading to:

$$\cos\theta_i = -\left(\vec{i}\cdot\vec{n}\right) \tag{3.4}$$

$$\cos^2\theta_t = 1 - \eta^2\,\left(1 - \cos^2\theta_i\right) \tag{3.5}$$

In (3.3), $\eta = \eta_1/\eta_2$, where η_1 and η_2 are the indices of refraction of each medium. Notice that we assume that the rays are leaving the object while updating the caching cube. So, for refracted rays, the first refraction and internal refractions must be calculated by the standard ray tracing technique. Eventually, when some of these reflection rays leave the object, the caching cube will store only the information from that point on. That is why we assume that the first refraction and internal reflections are already solved.

We update the caching cube only for rays whose stored information is invalid (i.e., A-channel equals *zero*). The current state of the cached information is retrieved for a given ray $\vec{r}$ by fetching the texture position related to the cubic box and the coordinates of a ray r_s traced from the center of the box in the reflection direction $\vec{s}$ ($\vec{s}$ is equal to $\vec{r}$ (3.2) or $\vec{t}$ (3.3), depending of the type of ray leaving the object). Based on the largest magnitude coordinate of the ray, we calculate which of the six textures of the caching cube has to be used to store the reflection information. The details are described later in this section. In the case of invalid data, the standard ray tracing procedure is evaluated for $\vec{r}$ leaving the surface of the object to compute its color.

Figure 3.2 shows the process of storing the reflection color for the CCT (Color Caching Technique). It shows a simplified 2-dimensional view of a complete ray tracing procedure. First, the ray r_0 is traced from the camera to the scene and hits a spherical object at point P_0. In turn, the reflected ray r_1 is created and shot into the environment, hitting a square object at point P_1. The recursive ray tracing continues until the ray hits a non-reflective object or another stop condition is achieved. In this example, the reflection color to be stored is computed as the intensities accumulated along r_k, for $k > 1$. The location in which the reflection color is stored in the caching square is computed from the intersection of the ray r_s leaving the center C of the square with the same direction as r_1. In Figure 3.2, r_s hits the square box at P_2. In our implementation, all rays traveling from the camera (red arrows in Figure 3.2) only intercept renderable objects in the scene because the caching cubes do not have geometry.

The face texture that stores the current ray's color (RGB) and state (A) is selected based on the largest magnitude coordinate direction of the reflected ray (see r_1 in Figure 3.2). The texture coordinates to be used in the selected texture are computed as follows:

$$t_u = \frac{1}{2}\left(\frac{t'_u}{|m|} + 1\right), \text{ and } \; t_v = \frac{1}{2}\left(\frac{t'_v}{|m|} + 1\right) \tag{3.6}$$

where m is the coordinate value related to the major axis direction of $\vec{s}$ in object space, and t'_u and t'_v are defined according to Table 3.1. It is important to comment that this is

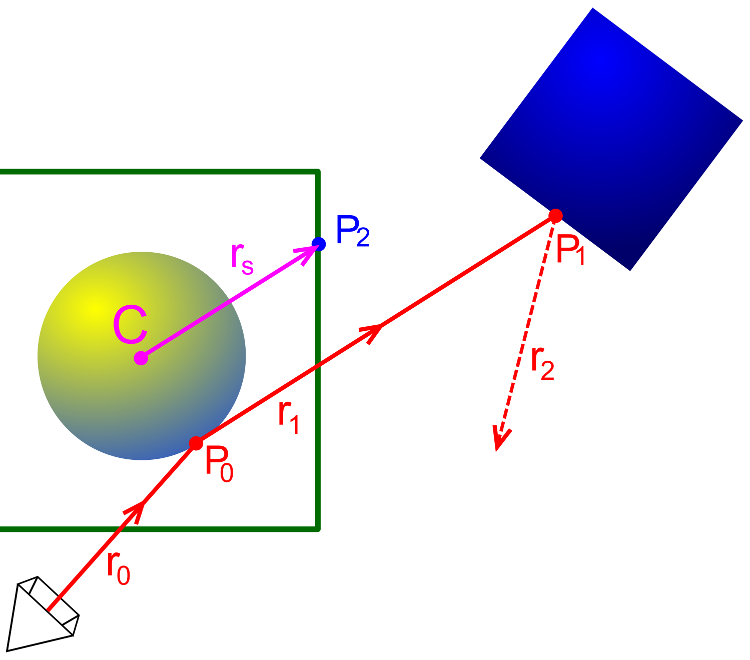


Figure 3.2: The path of a primary ray traced from the camera to the scene is defined by its interactions with the scene objects. In this 2-dimensional example, the primary ray r_0 intersects the yellow spherical object at point P_0, producing the reflected ray r_1, which, in turn, hits a squared object at point P_1. The resulting reflected ray r_2 keeps interacting with the scene until some stop condition is achieved. With our approach, the creation of r_1 is prevented if the caching square (the green box) has the updated color information at point P_2, denoted by the intersection between one of the faces of the box. The ray r_s is cast from the center C in the direction of r_1.

the procedure adopted for selecting the face of a cubemap in OpenGL. More information about this process can be found in [Shreiner and Group 2009]. We assume the nearest-neighbor interpolation while sampling the rays from the caching textures because the writing operation is also performed on the texel's exact location.

Table 3.1: The OpenGL's cubemap arrangement table. The values will be used in (3.6) while computing the caching coordinates.

Major Axis	m	t'_u	t'_v
$+x$	$\vec{s}_x$	$-\vec{s}_z$	$-\vec{s}_y$
$-x$	$\vec{s}_x$	$+\vec{s}_z$	$-\vec{s}_y$
$+y$	$\vec{s}_y$	$+\vec{s}_x$	$+\vec{s}_z$
$-y$	$\vec{s}_y$	$+\vec{s}_x$	$-\vec{s}_z$
$+z$	$\vec{s}_z$	$+\vec{s}_x$	$-\vec{s}_y$
$-z$	$\vec{s}_z$	$-\vec{s}_x$	$-\vec{s}_y$

The texture selection and the computation of the texture coordinates have been defined to be consistent with DirectX's and OpenGL's cubemap arrangements. However, it is important to comment that we had to implement those operations as part of our programs because the ray tracing engine adopted in this work (OptiX 2.6) does not support native environment mapping techniques.

We choose the cube mapping approach instead of the sphere mapping simply because our technique may resemble the environment mapping, which currently uses the cube mapping approach to store the surroundings. Another reason for not using the sphere mapping is that the degree of deformation near the poles of the sphere is very high, causing a significant loss in the quality of the reflection. Despite that, there are no constraints that invalidate the use of a caching sphere with our technique.

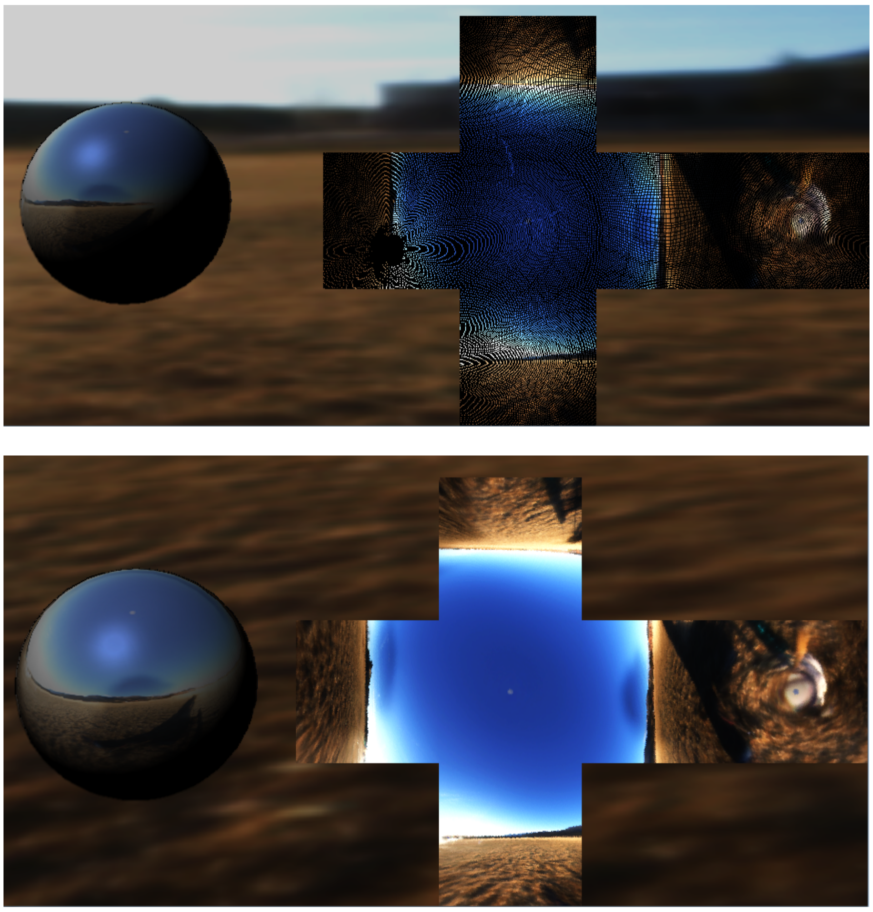


Figure 3.3: The color information stored in the caching cube from a reflective sphere after executing the first frame of the application (top) and after moving the camera around the object (bottom). The right side of both images shows the caching cube opened to show the cached information from the spheres on the left side. In this particular example, we chose to store only the pure color of the reflex and not shadow information.

Figure 3.3 (top) shows the data from the caching cube proposed by our approach when using the CCT strategy. Black texels represent outdated information. After moving the camera and completing a rotation of approximately 30 degrees around the object, all the texels of the caching cube were filled with updated information (see Figure 3.3 (bottom)).

As previously explained, the reflection, shadowing and specular highlight for the current object are computed separately and then combined together, so as we can see in this example, the caching textures are only storing the pure color of the reflex, and not the shadow or specular highlight. However, in our technique the pure color is stored along with the shadow information.

A simplified version of the algorithm used to store the color information can be seen in Algorithm 2. In this pseudo-code we only present the calculation of a single face of the cube. Note that the algorithm barely changes for the other faces. In fact, the only difference is that, depending on the face, the values of m, t'_u and t'_v will change, according to Table 3.1.

Algorithm 2: The algorithm for storing the information. A simplified version only for the positive X axis. Note that this code uses the information of the first row of Table 3.1.

```
input         : $rayDirection$, $reflexColor$
texelPos      : The non-normalized position of the texel
m             : The coordinate value of the major axis direction
textureSize   : The size of the cubemap's texture
cubemap_pos_x: The bidimensional array that represents the texture

if $x$ is the major axis direction then
    $m \leftarrow$ Absolute $(rayDirection.x)$
    if $x$ is positive then
        $float2\ texelPos \leftarrow (-s.z, -s.y)$
        $uint\ t \leftarrow 0.5 * ((texelPos/m) + 1.0) * textureSize$
        $cubemap\_pos\_x[t] \leftarrow reflexColor$
    end
end
```

3.4 Using the Cached Values

The process of retrieving information from the caching cube using the CCT strategy is performed every time a ray hits the object and produces a reflected or transmitted ray whose direction ($\vec{s}$) is defined by Equation (3.2) or Equation (3.3), respectively. The 2-dimensional texture of the caching cube and the texture coordinates in which the ray information resides are computed using Equation (3.6) and are implemented with a minor change to Algorithm 2. The only difference is that instead of storing the information, we retrieve it and assign the retrieved color information to the $reflexColor$ variable. Once changed, this block of pseudo-code now represents the *retrieveInformation()* procedure from Algorithm 1. When the stored information is valid (i.e., A-channel equals *one*), the recursive ray tracing of the ray leaving the object is avoided, and the cached information is combined with the shadow ray to compose the final intensity of the incoming ray.

When we retrieve valid information, the creation and tracing of all secondary rays is avoided and replaced by a simpler texture fetching procedure. Thus, because it is hard to achieve coalescent memory access in the ray tracing technique, as two neighboring rays can travel to different places in the scene, it is more likely that threads related only to primary rays will not be idle for long periods. In GPU-based ray tracing architectures, this tends to maximize the efficiency of the GPU.

The technique is implemented inside an object's material process, which can be assigned to any objects in the scene. It contains all the information needed for the ray tracing to calculate the reflection and refraction of the object, such as ambient, diffuse, specular and reflectivity coefficients. After assigning this material to one or more objects in the scene, we only need to run the application to see the results. A comparison between a result produced by a traditional ray tracing and our technique can be seen in Figure 3.4.

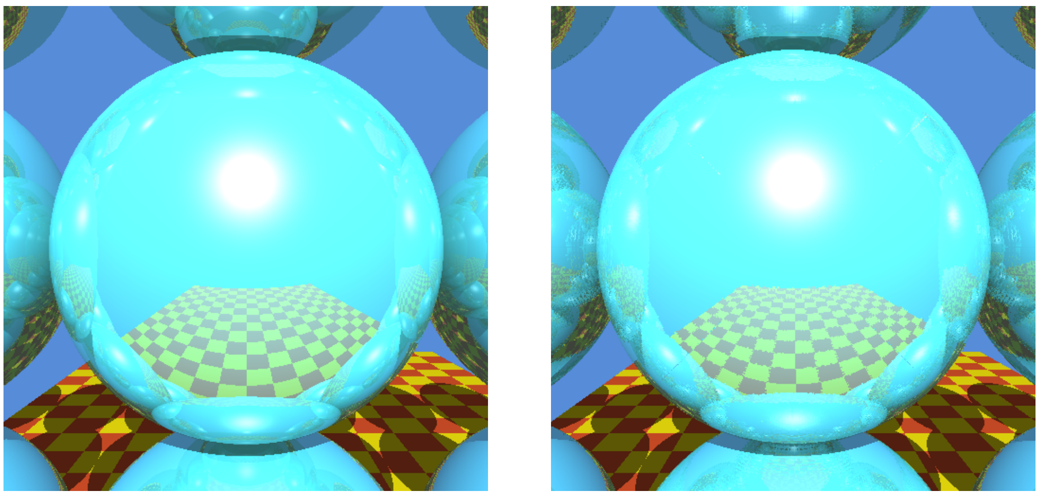

(a) Ray Tracing (b) Color Caching

Figure 3.4: A comparison between the result of a conventional ray tracing (a) and the results of the ray tracing with the CCT (b). The scene is composed of a grid of 32 spheres. Note that despite some minor differences in the recursive reflection, the CCT strategy can generate similar results.

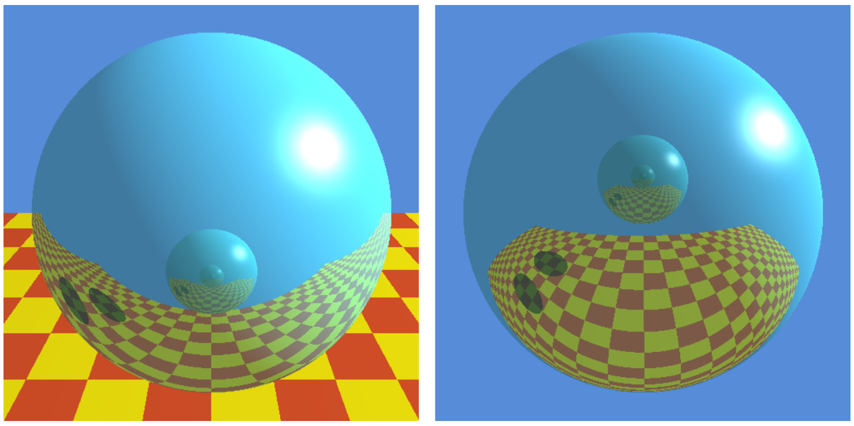
The results achieved by our technique look very similar to those of ray tracing. There are some minor differences that can be noticed when closely analysing Figure 3.4a and Figure 3.4b. In the first figure, we can see that the qualities of the reflection of the objects that are very close to the main sphere look a little more realistic and accurate than in the second figure. The main reason for this phenomenon is that, the caching textures have a fixed size and they use nearest-neighbor interpolation, so if we zoom in close enough, we will start to see a pixelated reflection in our technique but not in the ray tracing.

3.5 Discussion

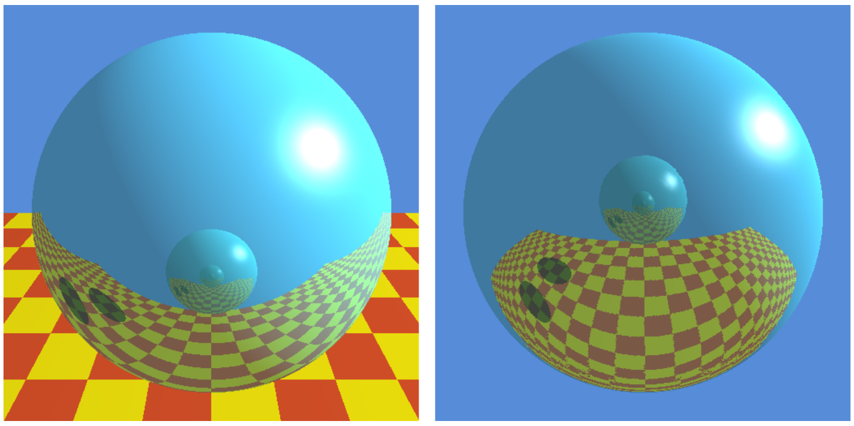
One of the main limitations of the CCT strategy is that when we store the color information of the reflection, the information is taken from a certain point of view. In a later frame, if we try to retrieve the information from another point of view, the color information stored will look planar, without tridimensionality. As a consequence, the parallax effect (i.e., the difference in the apparent position of an object viewed along two different lines of sight) will not occur. Thus, distinct objects that are close to or far from our object will look like they are at the same distance, giving the impression that they are glued together. Figure 3.5 shows that the parallax effect does not occur when using our technique.

Note that in Figure 3.5a, because the sphere is closer to the object in question and the plane is farther, when the camera moves around the scene, the effect known as parallax will occur. Note that from Figure 3.5a (left) to Figure 3.5a (right), a gap appeared between the sphere and the plane. That happens because in the parallax effect, closer objects seem to move faster than farther objects. However, in Figure 3.5b, the camera moves in the same path as before, but no gap appears between the plane and the sphere between Figure 3.5b (left) and Figure 3.5b (right). The reason why the parallax effect did not occur is because we are assuming that the depth of the information stored in the textures is all the same, as if every object is infinitely distant from the center of the cube.

A possible solution for that problem would be to store the depth information of the reflection and then use a technique that can, based on this information, warp the texels in such a way that the parallax effect can occur. To fix this issue, we included a technique called relief mapping [Policarpo et al. 2005] in our approach. In the next chapter, the relief mapping technique will be described in detail, and afterwards, we present our second ray caching strategy, the CDCT (Color and Depth Caching Technique).



(a) Ray Tracing



(b) Color Caching

Figure 3.5: The parallax effect comparison between the ray tracing and CCT. The scene is composed of two spheres facing each other. In (a), we have the ray tracing technique and when the camera moves around, the parallax occurs normally. In (b), we have the CCT, and because there is no depth information on the textures, when the camera moves around, the parallax effect does not occur.

Chapter 4

Color and Depth Caching Technique

As we have seen in the previous chapter, despite the similar results achieved by the CCT, some problems appeared. The loss of the parallax effect highlights the bidimensional nature of the cache. To solve this problem, we will use the relief mapping (Section 4.1) developed by [Policarpo et al. 2005], a texture mapping technique used to render surface details of three dimensional objects. This technique generates the correct views of 3-dimensional meshes by augmenting textures with per texel depth information. To use the relief mapping, we need a different caching strategy, the Color and Depth Caching Technique (CDCT). This caching strategy has some differences from the CCT presented in the previous chapter. The first difference is the information that we store in the caching textures. The relief mapping technique requires the depth information of each one of the texels in the texture, so this information must be added to our caching cube textures. Because the three color channels cannot be changed, the simpler solution is to store the depth information in the alpha channel, where we previously stored the outdated and updated flag information. Another possible solution would be to create an extra texture to store only the depth information, but that would only increase the amount of memory used by our technique.

The relief caching cube will be the same as the caching cube. We will have an axis-aligned cubic box, centered at the origin of the object coordinate system, following the object's position but not the rotation. The faces of the box will also be oriented according to the cubemap texture convention, i.e., with the faces' normals pointing inside the cube. The only difference between the relief caching cube and the caching cube is that the cubic box needs to have coordinates in the object coordinate system. To achieve that, we will always create a box with coordinate values that contains the object, but again, these are just values for the intersection calculation between the box and the rays. The geometry of the box is not really on the scene. The reason why the box needs coordinates in the local coordinate system is because the relief mapping needs a starting point in the space to move along the face of the cube. To calculate this starting point, we will compute the intersection point between the ray and the relief caching cube.

With the new caching strategy created and the relief caching cube setup, we need to adapt the way that we store (Section 4.4) and retrieve (Section 4.5) the information of the textures in the methods $StoreInformation()$ and $RetrieveInformation()$ of Algorithm 1. The first will require a simple modification, while in the second, we will implement the relief mapping technique. Unfortunately, the relief mapping technique cannot be used in its standard version, so we must adapt it. The problem is in the textures that the relief mapping samples for the information. The standard relief mapping samples the color and depth in a texture produced through orthogonal projection, while in our work, we use perspective projection to fill the texture in rendering time. Later in this chapter, we will explain the adaptation of the relief mapping technique in detail.

4.1 Relief Mapping

The relief mapping technique has the ability to simulate a detailed tridimensional mesh in a scene using only a 2-dimensional texture with depth information and coarse geometry. The technique starts when a ray intercepts the coarse geometric object and, based on a linear and binary search, computes the intersection point between the ray and the depth information of the corresponding texel. At the end, it returns the correct color and depth values to be sampled, giving the user a feeling that there are 3-dimensional details where there is only an augmented texture mapped into a few geometric primitives.

The process is illustrated in Figure 4.1. At point A, the depth is 0 (zero), and in B, the depth is 1 (one). The first stage of the technique consists of a linear search until the depth value of the texture is smaller than the ray's depth. It starts at point 1, which represents an offset of δ in relation to A. After that, it compares the depth of point 1 with the depth of the texture at that particular point. Because the depth value in point 1 is smaller than the texture depth, it repeats the process of offsetting the ray and calculating the depth until it arrives at point 3. The ray's depth at point 3 is bigger than the texture's depth, so the algorithm moves to the next stage, the binary search. In this stage, it calculates the midpoint between the last two ray positions (2 and 3), resulting in point 4. After that, it continues to compare the depth between the ray and the texture

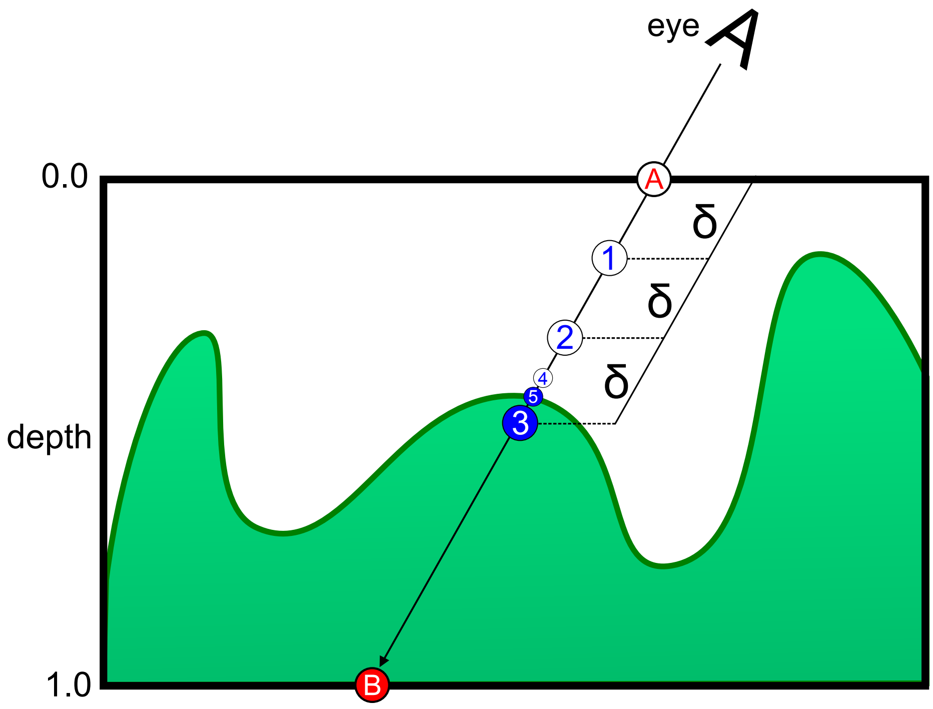


Figure 4.1: The relief mapping technique. The process starts with a linear search followed by a binary search. The linear search stops when the depth of the ray gets bigger than the depth of the texture (3), and the binary search stops when the threshold gets smaller than a certain value (5). The points 1, 2 and 3 are done by the linear search, and the points 4 and 5 are done by the binary search.

in the current position. The binary search stops when the length between the two last points gets smaller than a certain threshold, called ϵ. The values of δ and ϵ are previously established by the user. For small values of δ and ϵ, more steps of the linear and binary search will be executed. This will lead to more precise results, because the error and the offset are smaller. Unfortunately, they will also greatly increase the computational cost.

In Figure 4.2, Policarpo et al. [Policarpo et al. 2005] compare the results produced by the relief mapping and other methods. In Figure 4.2a, we have the bump mapping, in Figure 4.2b, the parallax mapping and in Figure 4.2c, the relief mapping with self-shadowing. Notice that the technique generates impressive results if we consider that there is only one quad (two triangles) being rendered in the screen.

4.2 Relief Mapping in Orthogonal and Perspective Depth Images

The relief mapping described in Section 4.1 uses a 2-dimensional texture with depth information captured with orthogonal projection. Because our objective is to capture all

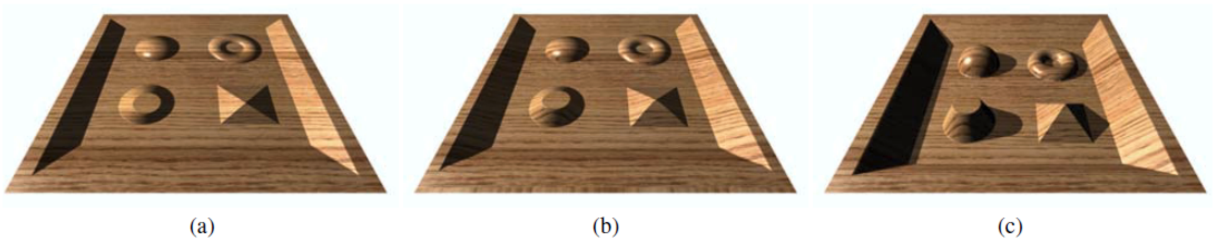


Figure 4.2: A quad rendered from the same point-of-view using three different techniques: (a) bump mapping, (b) parallax mapping and (c) relief mapping with self-shadowing. Source: [Policarpo et al. 2005].

the reflection information around a certain object, we face a problem: The orthogonal capture is not capable of capturing the entire environment around an object. We have only six textures, one on each face of the relief caching cube that contains the object. Each one of these faces will be used to store the information outside the cube. By using orthogonal projections, some parts of the environment will not be captured, illustrated by the red regions in Figure 4.3a. One possible solution is to use perspective projections to capture all the surrounding information. Unfortunately, the perspective projection does not preserve the size and form of the objects when storing them in the textures. However, it can capture the whole environment around an object, as shown in Figure 4.3b.

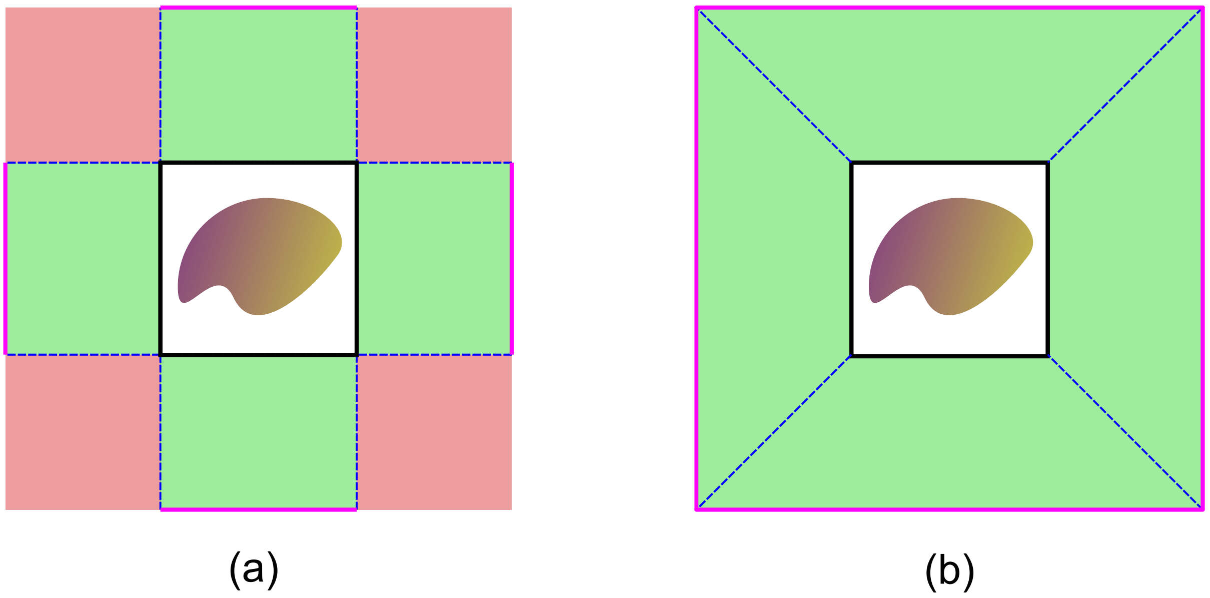


Figure 4.3: A simplified 2-dimensional view of the difference between capturing information using orthogonal and perspective projections. In these images, the center of projection is the center of the white square, and the textures are over the edges. In (a), the environment is being captured using the orthogonal projection, so objects in the red regions will not be captured. In (b), we have the perspective projection. Notice that there are no red regions.

Many of the features that an orthogonal projection has cannot be seen in a perspective projection, including the following:

- **The size of the objects:** In the orthogonal projection, objects of the same size will always have the same size, regardless of the distance between them and the point of view. In the perspective projection however, objects close to the center of projection will look bigger, and objects far from it will look smaller.
- **The form of the objects:** Because closer objects look bigger and distant objects look smaller, a piece of the object that is closer to the camera will look bigger, and the other pieces that are not so close will look smaller. Additionally, parallel lines that go to one of the vanishing points of the projection will not remain parallel; hence, their projections intercept themselves in the infinite.
- **The field-of-view deformation:** Because we have a center of projection, depending on the aperture angle from the center of projection to the projection plane, the information stored can suffer a large deformation. Additionally, information near the center of the projection plane suffers minor deformation, but information near the edges of the projection plane may have major deformation.

The perspective projection has some limitations, but what allow us to use the perspective projection to capture the depth information is that, the straight lines in the scene will keep being straight lines in the perspective projection. In fact, this is the only property maintained by the perspective projection. Thus, a ray traced in the scene that intersects the cube's face and penetrates the depth of the image will itself remain a line segment, and the relief mapping algorithm will work properly.

The main difference between the relief mapping in orthogonal depth images and perspective depth images is the procedure that retrieves the information. In an orthogonal image, each coordinate pair s and t of the texture stores the depth information of that same point, where the r is the exact depth, as we can see in Figure 4.4a. In contrast, perspective depth images suffer from deformation in the storage process, so each coordinate pair s and t stores the depth information of another point in space, as we can see in Figure 4.4b. As previously explained, the deformation caused by the perspective capture is not constant. Given that:

$$d_0 = (s_1 - s_3, t_1 - t_3) \tag{4.1}$$

$$d_1 = (s_2 - s_4, t_2 - t_4), \tag{4.2}$$

where d_0 and d_1 are the distances between the coordinate values and the depth values, and the texture coordinates (s_1, t_1), (s_2, t_2), (s_3, t_3) and (s_4, t_4) are the points shown in Figure 4.4b, it is easy to notice that:

$$|d_1| > |d_0| \tag{4.3}$$

where the point (s_2, t_2) is closer to the edge of the face and it suffers a greater deformation, so the distance between the texture point and the point in space increases when comparing texels with the same depth information.

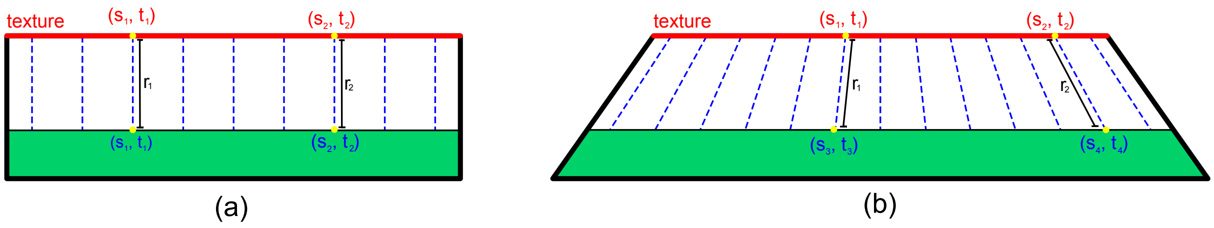


Figure 4.4: Relief mapping difference between the orthogonal projection (a) and the perspective projection (b). Note that in the orthogonal projection, if we want to sample the depth of point (s_1, t_1), the depth information r_1 is in the same point (s_1, t_1). However, in the perspective projection, when sampling the depth of point (s_1, t_1), the depth information r_1 is actually stored in the point (s_3, t_3).

4.3 Relief Mapping Adaptation

The standard relief mapping technique cannot be used to retrieve the information from the CDCT and must be adapted. The main problem is that in the orthogonal textures, when a step is taken into the texture because there is no deformation, the s and t coordinates from where we must sample are the exact same s and t coordinates where the ray is, as shown in Figure 4.5a. However, when perspective depth images are used, because of the deformation suffered, if we sample the texture from the s and t coordinates from where the ray is, we will not get correct depth information, leading to inaccurate results. For the relief mapping to work, we must find a way to calculate the texel that is storing the correct information, as shown in Figure 4.5b. Moreover, because the degree of deformation varies depending on the distance to the center, there is no simple way we can input the coordinate where the ray is located to get the correct coordinate from which we must sample.

To find the correct texel to sample the texture, we first need the center of the projection that, in our case, will always be the center of the object (the origin of the object

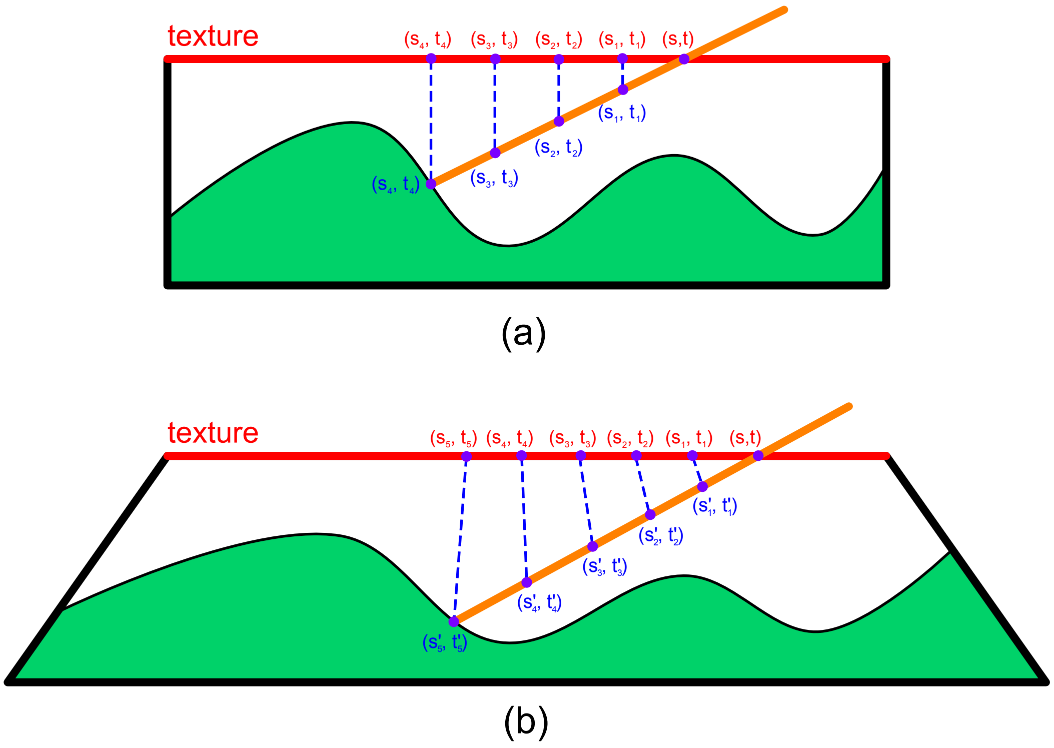


Figure 4.5: The difference between walking on the ray in an orthogonal depth image and a perspective depth image. Notice that in (a), the coordinates where the relief must sample are the same. However, in (b), the coordinates are not the same because of the distortion of the perspective images.

coordinate system). This point will be converted into the world coordinates system. For each step of the relief mapping, when the ray penetrates a little further into the texture, we will create a line segment that starts in the center of the object and ends at a certain step along the ray. This line segment must be normalized, and then, we will use Equation (3.6) and Algorithm 3 to retrieve the information from the cubemap information and assigned it to the $reflexColor$ variable.

The detailed process can be seen in Figure 4.6. The relief mapping process starts at point P_{t_0}, and by walking a small distance along the ray, we arrive at P_{r_1}. This is the point where we will compare the depth of the ray with the depth P_{d_1} of the texture at point P_{t_1}. To find the correct depth, we need to create the vector $\overrightarrow{CP_{r_1}}$ and use its direction to sample the cubemap. The largest magnitude coordinate of this vector will define what face we are sampling, and the other two values will be used to find where in the face to sample using Equation (3.6). With the point P_{t_1} in hand, we only need to compare the values between the alpha channel (which is represented by the point P_{d_1}) and the depth of the ray. If the depth of the ray is smaller than the depth of the texture point P_{t_1}, we continue to walk along the ray; otherwise, we stop. This process will continue until the depth of the ray is bigger than the depth of the texture. Once this condition is achieved, we stop and return the correct point to sample the texture.

Algorithm 3: The algorithm for retrieving the information. A simplified version only for the positive X axis. Note that this code uses the information of the first row of Table 3.1.

```
input        : $rayDirection$
output       : $reflexColor$
texelPos     : The non-normalized position of the texel
m            : The coordinate value of the major axis direction
textureSize  : The size of the cubemap's texture
cubemap_pos_x: The bidimensional array that represents the texture

1 if $x$ is the major axis direction then
2     $m \leftarrow$ Absolute $(rayDirection.x)$
3     if $x$ is positive then
4         $float2\ texelPos \leftarrow (-s.z, -s.y)$
5         $uint\ t \leftarrow 0.5 * ((texelPos/m) + 1.0) * textureSize$
6         $reflexColor \leftarrow cubemap\_pos\_x[t]$
7     end
8 end
```

The relief mapping technique is based on a linear search followed by a binary search. Figure 4.6 only shows the concept of walking along the ray into the texture and comparing the depth values of the ray and the texture. The linear search will also come first and will be followed by a binary search in the adapted relief mapping. The overall process is not changed, so it will be the same as previously explained and shown in Figure 4.1. The only aspect needing to be changed is the way in which we walk along the ray and consult the depth of point in the texture.

4.4 Caching the Reflection Information

With the adaptation to the relief mapping, the technique used to store the information in the textures also needs to be changed. The procedure used in the previous chapter (Figure 3.2) to store the information will no longer work because if we stored the information in the direction from where the ray leaves the object, the relief mapping will return inaccurate results, as detailed in Section 4.6. The new procedure used to store the information is shown in Figure 4.7. The path of a ray traced from the camera into the scene executes in the same way as before. After a certain ray is traced, the ray tracing technique will return the color of the reflection. Now, instead of creating the ray r_s as

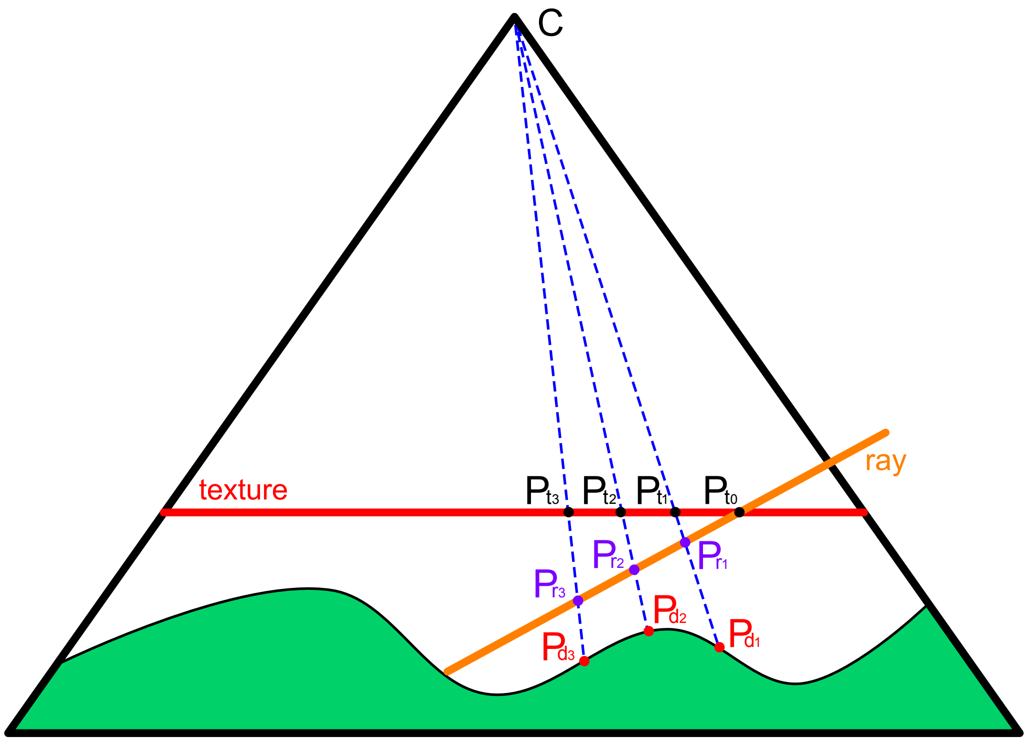


Figure 4.6: The relief mapping technique in perspective depth images. The points P_{t_0}, P_{t_1}, P_{t_2} and P_{t_3} represent the texture points where the correct depth is stored. The points P_{r_1}, P_{r_2} and P_{r_3} represent the points along the ray that we are walking on. The points P_{d_1}, P_{d_2} and P_{d_3} represent the depth value of the corresponding point P_t.

shown in Figure 3.2 (in the same direction as r_1), r_s will now represent the direction from the center C to the point P_1. This direction is calculated by subtracting the point P_1 from the center of the object C. The resulting vector is then normalized, creating r_s. The ray r_s hits the caching cube at P_2. To calculate P_2 we use again the largest magnitude coordinate of the vector r_s to detect which face we need to store and use in Equation (3.6) to calculate the coordinates for the respective texture.

Because we wish to use the reflection information with the relief mapping technique, we also need to store the depth information. The alpha channel of the texture stores the distance from point C to point P_1. The value stored will not be normalized, i.e., between 0 and 1, as in the standard relief mapping shown in Figure 4.1. Because the alpha channel is used to store the valid/invalid information, the change is simple. Positive values indicate valid information, and negative values indicate invalid information (i.e., all the textures will start with alpha equal to minus one). The texture used in our implementation will store the non-normalized distance. The reason for not using the normalized depths is because they are stored dynamically, so we do not know the minimum and maximum depth values for each one of the objects, making it time consuming to normalize them.

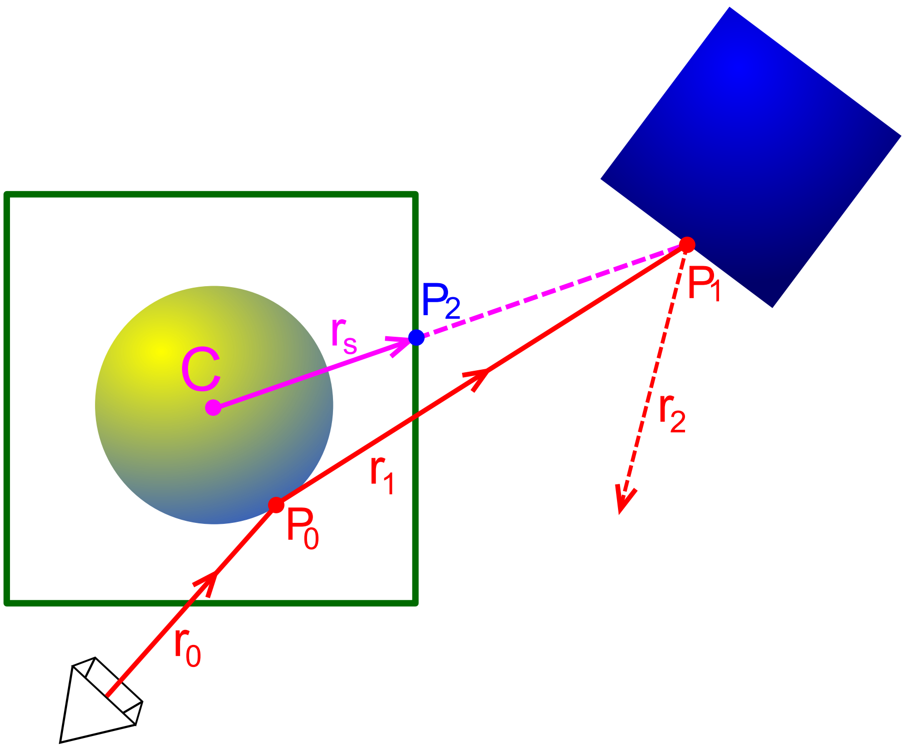


Figure 4.7: The path of a primary ray as traced from the camera to the scene is defined by its interactions with scene objects. In this 2-dimensional simplification, the ray tracing is executed normally until the end. The ray r_s is cast from the center C in the direction of P_1, hitting P_2, the point where the reflection information will be stored.

The algorithm used to store the information with the relief mapping is the same one utilized for storage without it (Algorithm 2). The only difference is that the direction represented by $rayDirection$ will not be the same, as explained earlier.

4.5 Using the Cached Values

The algorithm for retrieving the information of the textures is represented in Algorithm 4. The first step in the algorithm of the adapted relief mapping is to calculate the initial point where the relief will start. For that, we need to obtain the intersection between the line that starts where the ray hit the object in question and ends where the reflection/refraction ray hit any other object in the scene. Next, we need to store our current point for later use. We also need to walk along the ray a short distance and sample the cubemap. With the depth value of the texture in hand, we can compare it to the ray's depth. This step will continue until the depth of the ray is larger than the depth of the texture. Once this step is over, the binary search will start to subdivide the last offset taken, using the last and current coordinates of the point. This process will last until the last step taken is

Algorithm 4: The storing technique with the adapted relief mapping. The process starts with a linear search followed by a binary search. By decreasing the value of the $offset$ and *threshold* variables the algorithm can return more precise results, but the computational cost will significantly increase.

```
input        : rayDirection
output       : correctPoint
currentPoint : The current point along the ray to be checked
objectCenter : The coordinates of object's center
sample       : The RGBA value retrieved from the caching cube
offset       : The value added to the ray at each step
threshold    : The error allowed in the binary search
rayDirection : The direction of the ray in the current step
correctPoint : The correct point to sample from the texture

currentPoint ← GetIntersection (line, box)
for i ← 1 to steps do
    lastPoint ← currentPoint
    currentPoint ← currentPoint + offset
    rayDirection ← currentPoint − objectCenter
    sample ← RetrieveInformation (rayDirection)
    if sample.w ≤ Length (rayDirection) then
        currentError ← Absolute (sample.w− Length (rayDirection)
        offset ← offset/2
        currentPoint ← lastPoint + offset
        while currentError ≥ threshold do
            rayDirection ← currentPoint − objectCenter
            sample ← RetrieveInformation (rayDirection)
            offset ← offset/2
            if sample.w ≤ Length (rayDirection then
                currentPoint ← currentPoint − offset
            else
                currentPoint ← currentPoint + offset
            end
            currentError ← Length (currentPoint − lastPoint)
        end
        correctPoint ← currentPoint
        break
    end
end
```

smaller than a certain threshold. The for statement in Algorithm 4 represents the linear search. Notice that every time we compare the depths in the following if statement. The *while* statement represents the binary search, which will continue until a certain threshold is achieved. Notice that the binary search is only executed if the linear search found a point where the depth of the ray is greater than the depth of the texture.

4.6 Discussion

The CCT proposed in Chapter 3 has undergone several adaptations, leading to the creation of the CDCT presented in this chapter. We have adapted the textures, the way that we store and retrieve the information from the caching cube and the relief mapping technique. The quality and frame rate comparison between our strategies and the other state-of-the-art techniques are presented in Chapter 5.

The intersection between the line segment that represents the reflection ray and the caching cube was not in the first version of the technique (i.e., the CCT). Because we only use the coordinates of the reflection vector to determine where we must store the information, there is no need to calculate the intersection point between this ray and the caching cube. In Chapter 3, the technique without relief mapping do not use it. In fact, the only reason why this intersection is calculated is because we need to find the starting point for the relief mapping technique. As we can see in Algorithm 4, the intersection calculation is only done in the first step of the relief mapping, and it is never used again. The caching cube was not supposed to have coordinates in the object space, but because of this intersection calculation, some coordinates had to be assigned to it.

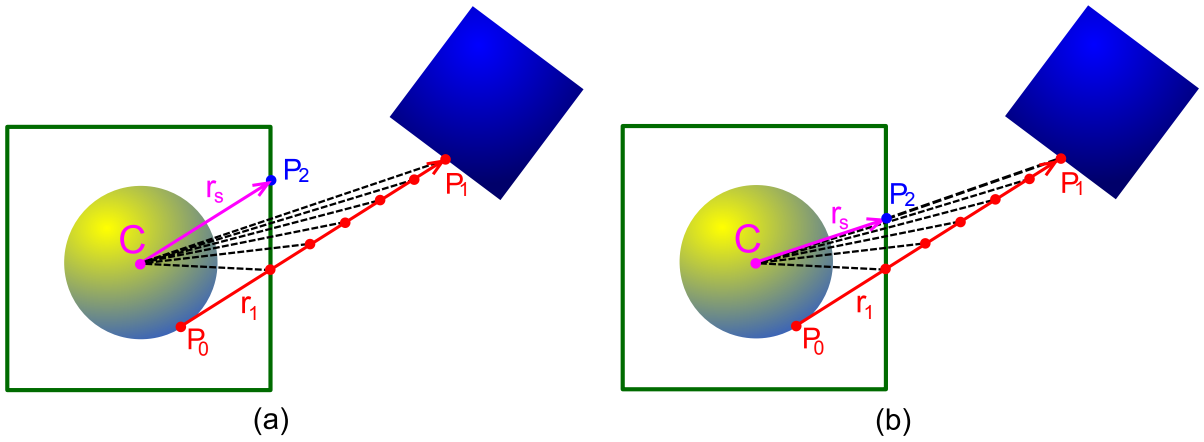


Figure 4.8: The comparison between the two ways used to store the reflection information. In (a), r_s is in the same direction as r_1, and using the adapted relief mapping will lead to inaccurate results, because the point P_2 will never be found. In (b), the r_s is created in the direction of P_1, and it is possible for the relief mapping to find the correct depth of P_1 that is stored in P_2. The black dashed lines represent each step of the adapted relief mapping.

In this work, we used an incremental development. In Chapter 3, the technique to store and retrieve information was very simple. We finished implementing it, and when analyzing the initial results, they showed us exactly what we were expecting, accurate results with no parallax effect. The relief mapping was introduced to solve the lack of parallax effect and to enhance the results. When implementing it, we had to adapt it to work properly. Using the technique presented in Chapter 3 to store the information using the same direction of the reflection/refraction ray leaving the object presented us with inaccurate results. As we can see in Figure 4.8a, if we store the depth information of P_1 in P_2, when executing the relief mapping (where the black dashed lines each represent one of the steps), it will stop before hitting P_2. This means that the correct depth information will never be found. However, if we use the new storage technique, it will be possible to find the point P_2, as we can see in Figure 4.8b, which is the main reason for a change to the technique used to store the information when using the adapted relief mapping. After some tests, we realized that in the strategy without relief mapping, both ways to retrieve the information, i.e., the one represented in Figure 3.2 and the one in Figure 4.7, will generate similar visual and performance results.

Chapter 5

Results

We have implemented our technique using C++ with OpenGL 4.3, CUDA 4.0, and OptiX 2.6. We have used the Assimp [Assimp 2011] library to load the 3-dimensional scenes and the DevIL [Woods et al. 2001] library to load the environment maps. The experiments were performed on a 3.40 GHz Intel Core i7-2600K machine with 8 GB of RAM and a NVIDIA GTX680 graphics card with 2048 MB of memory. Microsoft Windows 7 (64-bit) was used as the operating system.

To the best of our knowledge, the proposed ray caching strategy can be integrated into any existing ray tracing solution. In our experiments, we have integrated it with two different ray tracing solutions. The first one, a hybrid raster and ray tracing framework developed by Sabino et al. [Sabino et al. 2012], totally executed at the GPU. It was implemented using the same programming languages adopted by our solution, except for the use of GLSL as a shading language, which it is not required by our approach. The framework of Sabino et al. uses the raster deferred shading technique [Deering et al. 1988, Saito and Takahashi 1990] to prevent the computation of primary rays hitting diffuse objects in the scene. In a subsequent step, the hybrid solution applies conventional ray tracing to compute and add visual effects such as reflection and transmission for pixels neglected in the previous step. The last stage of Sabino's technique consists of the composition of the images created by the raster and by the ray tracing stages. The second ray tracing solution is a conventional ray tracing implemented with OptiX.

Experiments were performed using the two caching strategies developed in our work. The first one, the CCT, was introduced in Chapter 3. This technique was combined with the hybrid ray tracing and with the standard ray tracing. The idea was to prove that the Per Object Ray Caching technique can be easily combined with both an optimized and a non-optimized ray tracing solution. For the second strategy, the CDCT, presented in

Chapter 4, the experiments were made only with the standard ray tracing implementation, and not with the hybrid solution, because we believe that the type of ray tracing solution used would not influence the speed up of the process.

5.1 Visual Quality and Performance

The visual quality of reflections and refractions are dependent on many factors. The first is the size of the texture used to stored the reflection information. The influence of the texture size in the reflection quality will be described in Section 5.1.1. Another factor that influences the quality of the results is the strategy used to stored and retrieve this information. Because the two proposed approaches use different procedures to store and retrieve the information, the results produced by the CCT (Section 5.1.2) and the CDCT (Section 5.1.3) are not the same.

5.1.1 Texture Size

The size of the textures used as the faces of the caching cube had a large influence on the quality of the results in both strategies created in our work, i.e., the CCT and the CDCT. Figure 5.1 presents the quality comparison between the rendering of two scenes using the CCT on a hybrid ray tracing solution. The first one (Figure 5.1, top) is composed of 10 objects that have diffuse materials and a reflective sphere facing the camera. The second one is composed of 33 reflective spheres. From left to right, the first three pairs of images were produced using the proposed CCT, while the last pair was generated by the hybrid ray tracing. The resolutions of the 2-dimensional caching textures used in these examples are, 128×128, 256×256, and 512×512 pixels. The resolution of the output images is 1024×1024 pixels. Closer inspections of the resulting renders are presented in the detailed image insets. Notice that in Figure 5.1a, the edges of the objects are clearly bumpy. In Figure 5.1b, the texture size was increased, leading to slightly better results. By comparing the detailed views of Figures 5.1c and 5.1d, it can be seen that the use of 512×512 caching textures produced results that are visually equivalent to those produced by ray tracing.

In this Chapter, all the performance results were achieved by using caching textures completely filled and the resolution of 512×512 because after the caching cube has been filled, the frame rates achieved with different resolutions of the caching textures for both the CCT and CDCT are the same.

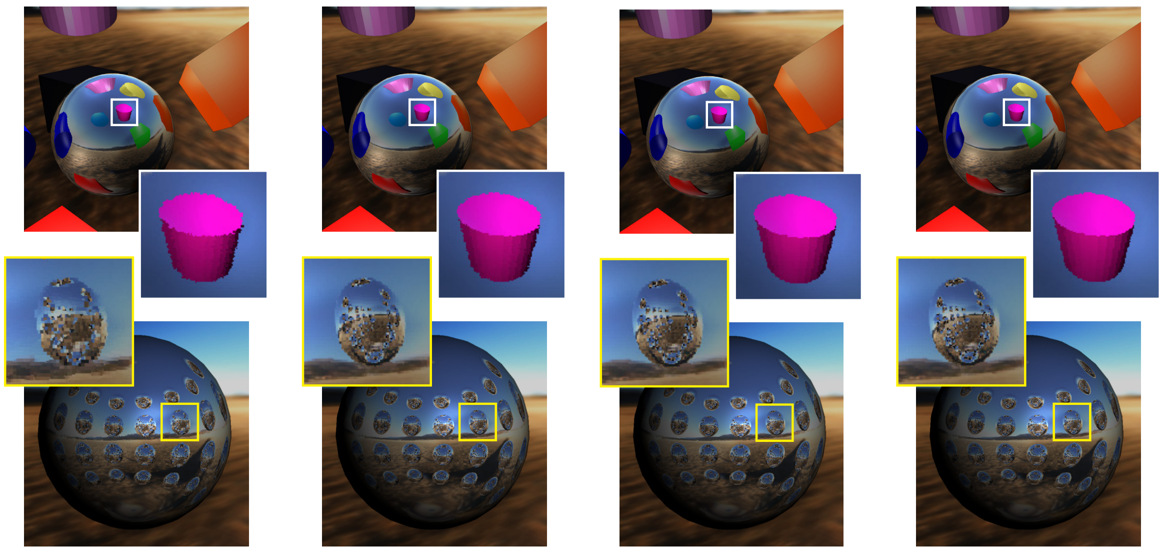

(a) 128 × 128 (b) 256 × 256 (c) 512 × 512 (d) Hybrid Ray Tracing

Figure 5.1: Images produced by ray tracing two scenes while using the hybrid ray tracing with the CCT (a)-(c) and without it (d). The quality of the resulting images increases with the resolution of the proposed caching structure. Notice that caching textures having 512 × 512 texels (c) generated images equivalent to the standard ray tracing (d). As seen in Table 5.1, for the images on the bottom, the CCT has the advantage of producing results such as (a)-(c) at 42 fps, while (d) is rendered at 12 fps.

5.1.2 Color Caching Technique

The performance of the CCT was evaluated by comparing the frame rates of the standard implementation of the framework developed by Sabino et al. with the same framework enhanced with the proposed technique. The measurements were made using the scenes presented in Figure 5.2. All these scenes are composed of a main reflective sphere facing the camera and of a set of reflective spheres placed behind the observer. This setup makes the extra spheres visible only from their reflection in the main sphere.

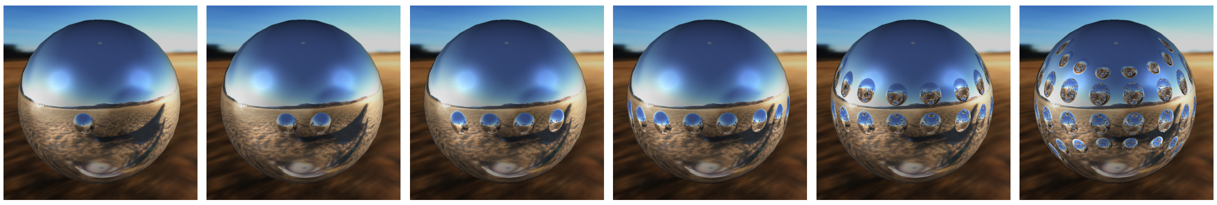


Figure 5.2: Scenes used in the performance comparison between the CCT implemented as part of the hybrid ray tracing solution and the standard implementation of the same framework. The complexity of the scenes varies from the left to the right. The scenes include a sphere with our technique and, respectively, 1, 2, 4, 8, 16, and 32 additional reflective spheres placed behind the camera. They can be observed from their reflections in the main object. The resulting frames rates are presented in Table 5.1 and Figure 5.3.

From the left to the right, the sets of additional objects are composed of, respectively, 1, 2, 4, 8, 16 and 32 spheres. The material of the main sphere implements the proposed CCT. The material of the additional objects uses conventional ray tracing. The resolution of the output images is 1024×1024 pixels. The frame rates were taken after the caching textures had been filled. The update rate at different caching resolutions is discussed in Section 5.2. Table 5.1 presents the performance of the compared implementations (columns 2 and 3) and the increasing speed achieved by the use of the caching textures having resolutions of 512×512 pixels (column 4).

Table 5.1: The performance comparison between the hybrid ray tracing solution running with and without the CCT.

Additional Reflective Objects	Hybrid Ray Tracing		Speed Up
	Standard	Color Caching Technique	
1	63 fps	120 fps	$\sim$ 1.90x
2	59 fps	117 fps	$\sim$ 1.98x
4	49 fps	98 fps	$\sim$ 2.00x
8	38 fps	85 fps	$\sim$ 2.24x
16	23 fps	63 fps	$\sim$ 2.74x
32	12 fps	42 fps	$\sim$ 3.50x

Each scene in Figure 5.2 is related to a row of Table 5.1. Notice that the greater the number of reflective objects in the scene, the greater the increase in speed with the CCT. This happens because, in contrast to the proposed approach, the computational cost of the ray tracing technique is proportional to the number of bounces performed by the rays, which is dependent on the number of objects. Figure 5.3 complements Table 5.1 by showing that the relative performance between the CCT and the hybrid solution has an approximately linear relationship to the number of additional objects in the scene.

In Table 5.1, an interesting fact is that the frame rate values of the column 3 are not the same for every scene tested. That would be expected, since the observer is looking only at one object, while the other objects are placed behind him. That did not happened in our tests because we used LBVH and SBVH structures to optimize the intersection computation between rays and objects in the scene. These structures are updated on-the-fly, because they support dynamic scenes. Unfortunatelly, in our case, they hindered our results because with more objects in the scene, these structures consume more time to be updated.

The conventional OptiX ray tracing implementation was also used to compare the frame rate between the standard ray tracing algorithm and the algorithm with the CCT.

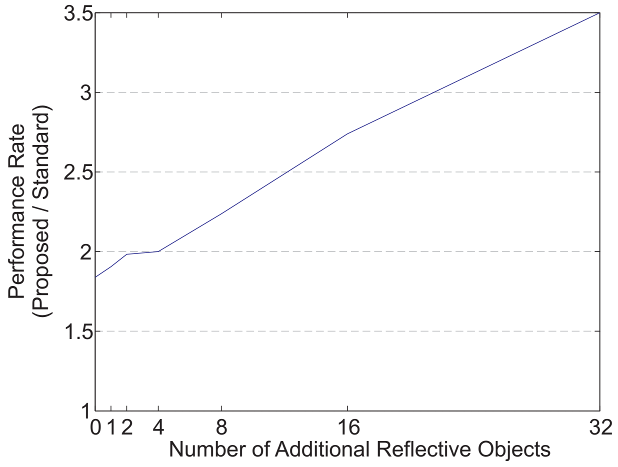


Figure 5.3: The relative performance comparison between the CCT and the hybrid ray tracing. The ray caching approach increases linearly in relation to the number of objects in the scene. See Table 5.1.

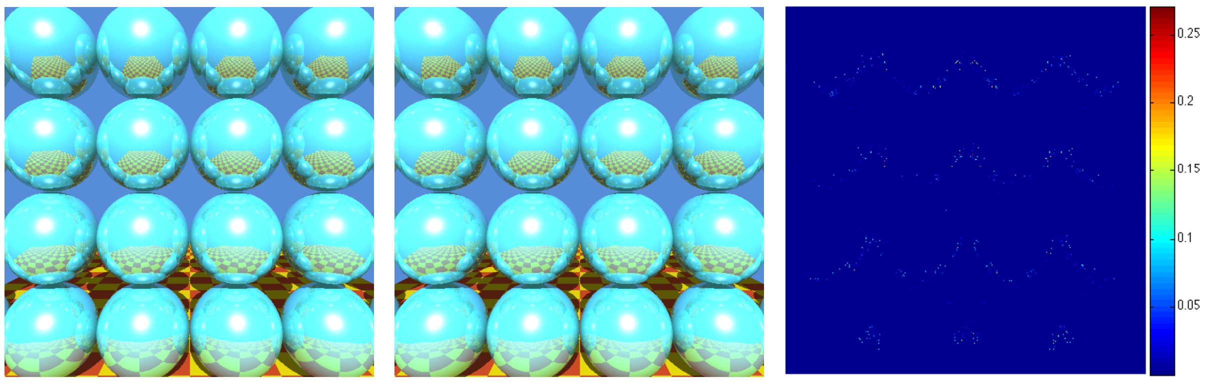
Table 5.2 presents this comparison. The scene used for this test contains a group of spheres with our technique, arranged like a grid in front of the camera (see Figure 1.1). As expected, the speed up was smaller than with the hybrid technique. In the hybrid approach, the primary rays are shot by the raterization-based graphics pipeline. The actual ray tracing is performed only for secondary and higher order rays. Thus, every ray in the hybrid approach will be replaced by our caching technique. This is in contrast to the conventional ray tracing, in which all the primary rays are traced, and many of them may hit a diffuse object or do not hit an object at all. These rays will not be replaced by our Color Caching Technique, providing no increased performance speed.

Table 5.2: The performance comparison between a conventional ray tracing system running with and without the CCT.

Additional Reflective Objects	Conventional Ray Tracing		Speed Up
	Standard	Color Caching Technique	
1	56.7 fps	57.8 fps	∼ 1.02x
2	52.0 fps	55.6 fps	∼ 1.07x
4	44.7 fps	51.8 fps	∼ 1.15x
8	32.8 fps	43.5 fps	∼ 1.32x
16	17.1 fps	33.4 fps	∼ 1.95x
32	8.0 fps	21.5 fps	∼ 2.68x

To evaluate the quality of the reflection, we used a color subtraction strategy. This consists of calculating the absolute value of the subtraction between the color information (the RGB channels) of the output image of the ray tracing and the output image of the CCT. This absolute subtraction is computed pixel by pixel, and the result is stored in another image, called the *subtraction image*. In this image, a blue pixel means that the color information from both images is the same. A red pixel indicates that the colors are totally different from each other. Intermediate values are displayed by different colors that are represented according to a color bar attached on the right side of the subtraction image. A value is also assign, where zero means same colors and one totally different colors.

A subtraction image between the standard ray tracing solution and the CCT can be seen in Figure 5.4c and in Figure 5.5c. In the first one, the subtraction image is practically blue. There are some minor differences between the output of the ray tracing and our output. Note that the camera is a considerable distance from the objects of the scene. However, if we get close enough to the objects, as in Figure 5.5, there are more noticeable differences. Note that in Figure 5.5c the number of different pixels is greater than in Figure 5.4c. For example, the checkerboard pattern presents some inaccurate information when changing from one color to another. Additionally, the reflections of the objects around the main sphere have many distinct pixel colors. This happens primarily because of the nearest-neighbor interpolation and also because of the lack of precision from some near-the-edge rays.

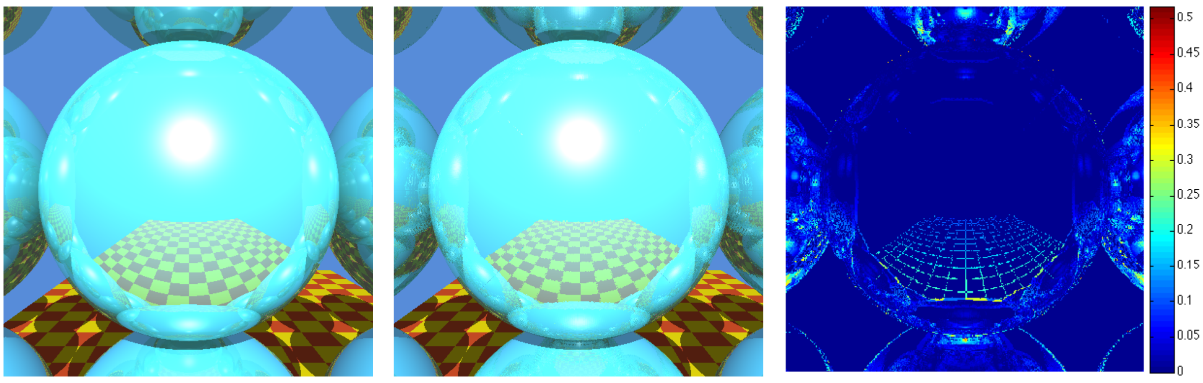


(a) Ray Tracing (b) Color Caching Technique (c) Subtraction

Figure 5.4: The image subtraction between the standard ray tracing and the CCT. In (a), we have the results of the ray tracing technique, in (b), we have the results of the Color Caching Technique, and, in (c), we have the subtraction image.

5.1.3 Color and Depth Caching Technique

The performance of the CDCT was evaluated only against the conventional ray tracing solution. The reason is that because the only difference between the two techniques is the addition of the relief mapping, the ray tracing solution used, whether standard or hybrid, will not influence the complexity of the relief mapping process, the quality of the results,



(a) Ray Tracing (b) Color Caching Technique (c) Subtraction

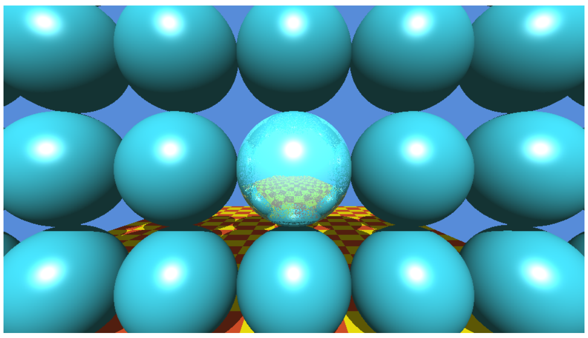
Figure 5.5: Another image subtraction between the standard ray tracing and the CCT. In (a), we have the results of the ray tracing technique, in (b), we have the results of the Color Caching Technique, and, in (c), we have the subtraction image.

or even any speed up of the technique. Note that there is no reason for speed up to vary because the relief mapping is calculated only when retrieving the information. This procedure would be executed the same number of times for any given ray tracing solution with the same configuration, such as texture size, output image and scene. Table 5.3 shows the comparison between the frame rates produced by the conventional ray tracing implementation and the CDCT for several values of δ and ϵ, where δ is the step taken into the relief texture, and ϵ the threshold. The depth of the information in the textures is represented between 0 and 100 units. Any ray that has a depth outside this range was stored with a depth of 100 units. The values of each step (δ), varied between 2 and 0.02 units, and the error threshold (ϵ) varied between 0.2 and 0.002 units. Table 5.3 also shows that for smaller values of δ and ϵ, the computational cost increases because more linear and binary steps will be executed to achieve the required goal.

Table 5.3: The performance comparison between the CCT technique, a conventional ray tracing solution and the CDCT.

Scenes	Color Caching Technique	Standard Ray Tracing	Color and Depth Caching Technique			
			$\delta = 2$ $\epsilon = 0.2$	$\delta = 0.4$ $\epsilon = 0.05$	$\delta = 0.2$ $\epsilon = 0.02$	$\delta = 0.02$ $\epsilon = 0.002$
Diffuse	76.0 fps	69.6 fps	49.5 fps	35.3 fps	30.7 fps	19.0 fps
Reflective	68.2 fps	30.1 fps	13.5 fps	12.7 fps	12.1 fps	9.5 fps

Figure 5.6 shows the two scenes used to test the CDCT. In the first scene, the "Diffuse" in Table 5.3, we have one reflective sphere with our technique and 31 other spheres with a diffuse material (see Figure 5.6a). The second scene, the "Reflective" in Table 5.3, is composed of 32 reflective spheres, all of them with the proposed technique (see Figure 5.6b). As we can see in Table 5.3, the best frame rate was achieved by the CCT,


(a) Diffuse Scene

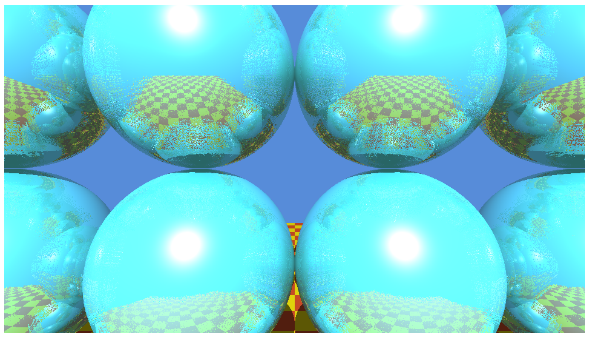

(b) Reflective Scene

Figure 5.6: The scenes used to test the CDCT. In (a), we have a diffuse scene with one reflective sphere and 31 diffuse spheres. In (b), the reflective scene has 32 reflective spheres with the CDCT.

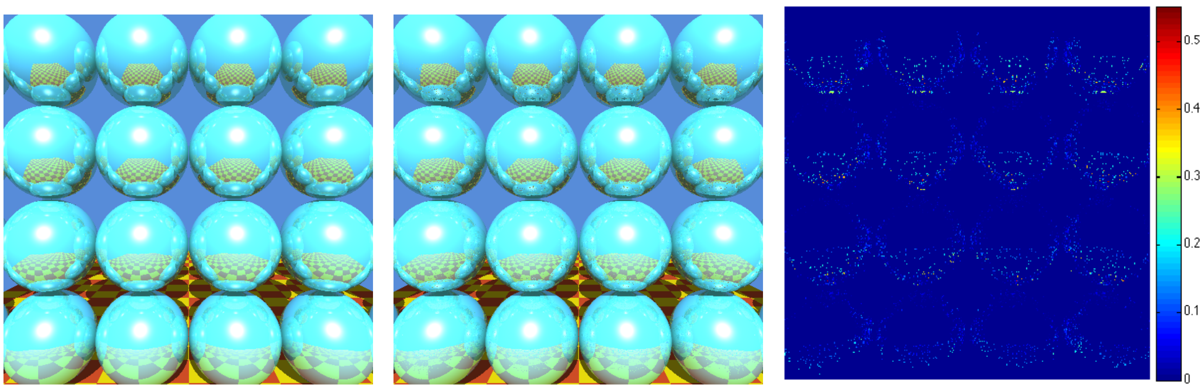
followed by the standard ray tracing solution. The worst results came from the CDCT. Each one of the textures has 512×512 pixels, and the output images have 1920×1080 pixels. The resulting image generated from the CDCT can be seen in Figure 5.6.

Unfortunately, the CDCT did not produce good frame rate results. The visual quality of the reflexes was lowered. We believe that are many factors involved. These factors include the following:

- **No native cubemap:** As mentioned before, the OptiX ray tracer engine does not support native cubemaps. That affected the CDCT more than the CCT because the relief mapping performs much more texel sampling when walking along the ray.
- **Nearest-neighbor interpolation:** The lower quality of the results may have happened due to the nearest neighbor interpolation. The proposed technique does not perform any type of filtering when sampling the caching textures. As an example, Figure 5.6a and Figure 5.6b show that the reflection of the surroundings on a specific object is suffering some type of noise, caused by an inaccurate calculation when retrieving the information from the relief mapping. The result of nearest neighbor interpolation was also aggravated by the deformation caused by the perspective projection used in the relief mapping.
- **Idle threads:** The relief mapping uses linear and binary searches to find the correct texel to sample. Some of these searches can be finished very quickly, and others require more time. This leads to idle threads in the GPU pipeline and can significantly increase the time spent to render a frame.
- **Sampling not calculated in blocks:** In a conventional rasterization process, close fragments are calculated in blocks. That increases the efficiency of the rasterization

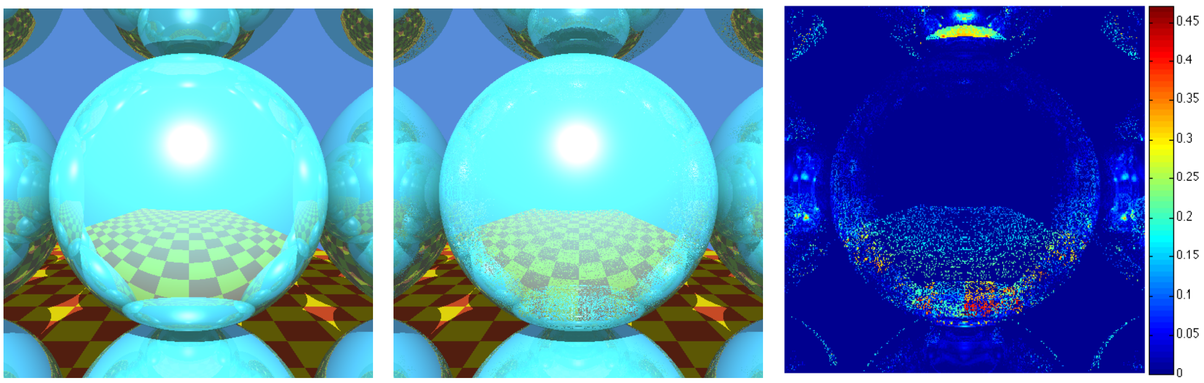
process. In our case, because we are sampling with our own procedure, we cannot guarantee that the rays are casted in blocks. That could significantly speed up the CDCT because we sample the texture several times before finding the correct texel.

To evaluate the quality of the reflection of the CDCT, we also used the color subtraction strategy. The results are presented in Figure 5.7 and Figure 5.8. Note that the subtraction image from the CDCT shows that there are more distinct pixels between the CDCT and the conventional ray tracing than with the Color Caching Technique. The noise generated by the CDCT is clearly seen when the camera is close to the object, as in Figure 5.8b.



(a) Ray Tracing (b) CDCT (c) Subtraction

Figure 5.7: The image subtraction between the standard ray tracing and the CDCT. In (a), we have the results of the ray tracing technique, in (b), we have the results of the CDCT, and, in (c), we have the subtraction image.

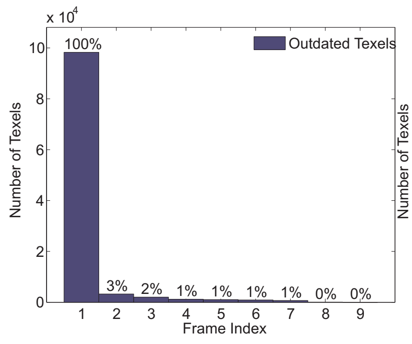


(a) Ray Tracing (b) CDCT (c) Subtraction

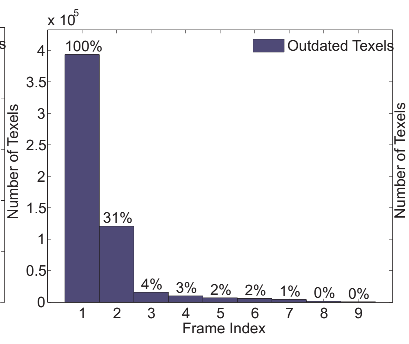
Figure 5.8: Another image subtraction between the standard ray tracing and the CDCT. In (a), we have the results of the ray tracing technique, in (b), we have the results of the CDCT, and, in (c), we have the subtraction image.

5.2 Update Rate

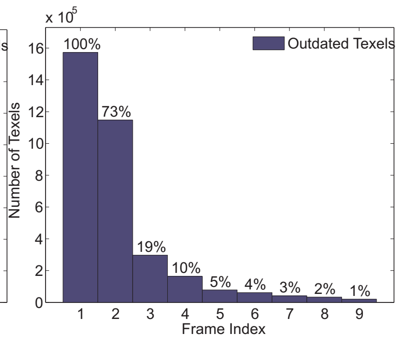
The update rates of the caching cubes with different resolutions were measured by placing a spherical reflective object in front of the camera and panning the camera around it with nine regular steps of 10/3 degrees each. The number of outdated texels were recorded every step before the rays were cast into the scene. The results are presented in Figure 5.9 for caching textures with, respectively, 128×128, 256×256, and 512×512 pixels.



(a) 128x128



(b) 256x256



(c) 512x512

Figure 5.9: The amount of outdated texels in the caching textures before each frame in a sequence of nine panning steps of 10/3 degrees each.

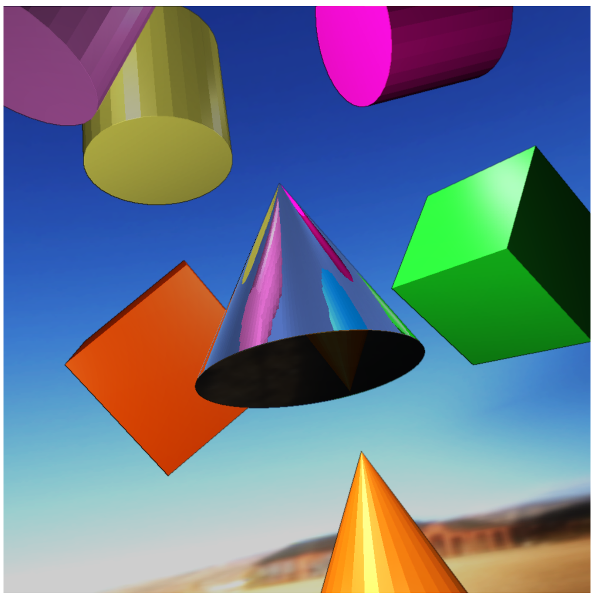
In all three cases, the caching cubes were initialized with invalid information (100% at the beginning of frame 1). According to Figure 5.9a, for 128×128, only 3% of the texels had invalid information after the first frame. After the end of the seventh frame, the caching textures achieved almost 100% valid data. For the texture with 256×256 pixels, the convergence rate is virtually the same as for the previous example. After the first frame, 31% of the texels had invalid information, but after the seventh frame, the caching textures are almost filled. For the case depicted in Figure 5.9c, as expected, the convergence rate was slightly slower than the previous cases. After the first frame, 73% of the texels had invalid information, but after 8 frames, only 1% of the texels were not yet updated. These tests show that the caching cubes can be quickly filled, even if they have high resolutions. A quick update is a desirable feature that promotes the reuse of stored information.

The update rates for both the CCT and CDCT are the same. The only difference is that because the relief mapping retrieves information from several texels for depth comparison, if one of the texels in the ray's texture path has invalid information, the relief mapping stops and the ray tracing is computed normally. This will make the relief mapping initially less efficient because even if we have the correct information for the ray, a single texel with invalid information during the relief mapping computation will prevent the technique from finding the correct information. To solve this problem, we could simply continue if invalid information is found, but after some tests, we found that invalid information can introduce some convergence problems into the binary search.

5.3 Limitations

Unfortunately, the proposed approach have some limitations. They include:

- **Dynamic scenes can generate invalid reflexes:** The proposed approach is tailored to static scenes. Because we are storing the previous ray tracing information of the scene around a certain object, it is easy to see that any movement on the scene objects would generate inconsistencies in the cached data.
- **Scenes with a massive number of reflective objects can consume the GPU memory:** The proposed approach will create a caching cube on the reflective objects in a scene. For each one of these objects, we create 6 textures with up to 512×512 pixels. Because the memory of the GPU is limited, scenes with a massive number of reflective objects may need more memory than the GPU has, making the memory allocation an impossible task.
- **Concave objects can generate conflicting rays, leading to inconsistent reflection information:** The technique is also tailored to be used with convex objects and auto-reflection features. Figure 5.10 shows that when the reflective object is convex, the resulting reflections are plausible and comparable to those produced by conventional ray tracing solutions. However, as seen in Figure 5.11, a concave object will generate ambiguous rays that hit different surfaces of the scene. As a result, some cached rays fetched from the caching cube may be inconsistent at a particular view point. See the impulsive noise in the detailed views of Figure 5.11.


(a) 512 × 512

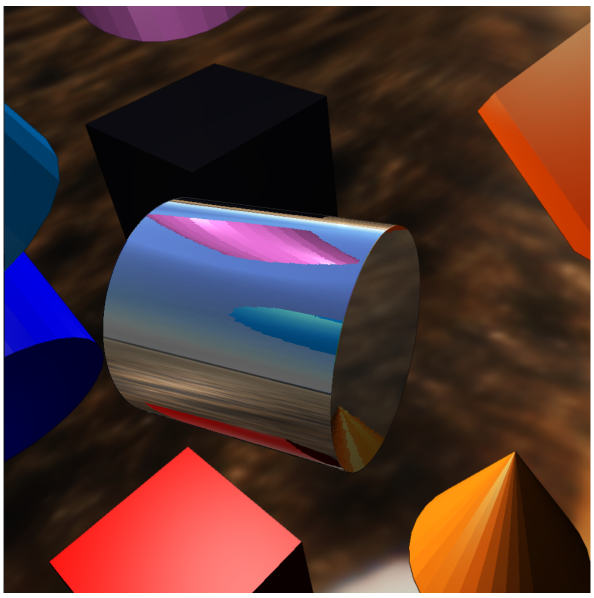
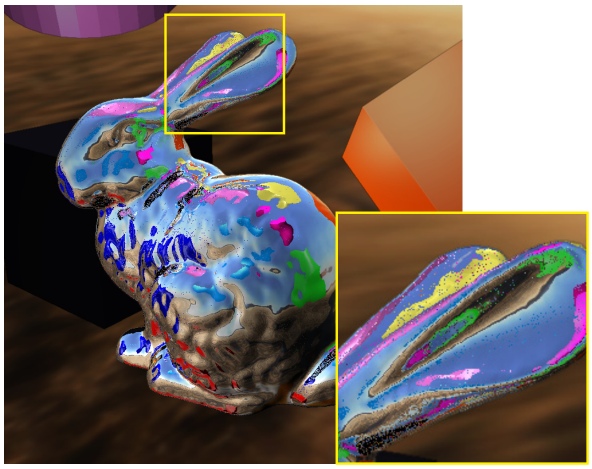

(b) 512 × 512

Figure 5.10: Images (a) and (b) show a cone and a cylinder whose material implements the CCT for the hybrid ray tracing solution. In both cases, the resolution of the caching textures was set to 512 × 512 pixels.


(a) Color Caching Technique

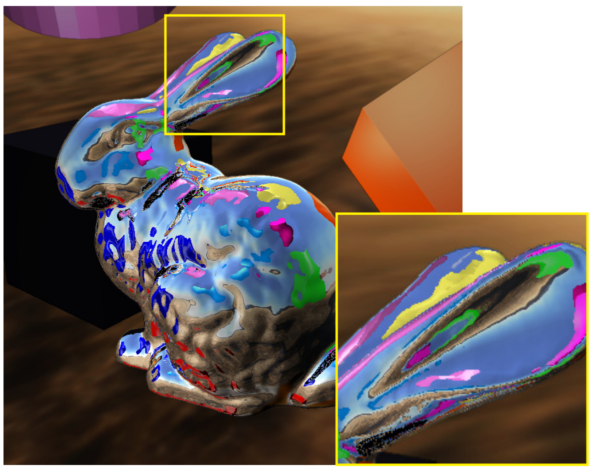

(b) Hybrid Ray Tracing

Figure 5.11: A concave object rendered using the CCT with 512 × 512 caching textures (a), and the hybrid ray tracing (b). For this type of object, conflicting rays leaving the object may have different targets.

Chapter 6

Conclusions and Future Work

Ray tracing is a very well-known technique that can produce images with a high level of realism. Although conceptually simple, tracing rays that bounce through the entire scene can be a very intensive task. That is the reason why many techniques have been created to make the ray tracing algorithm less computationally intensive.

We have presented an efficient cache strategy that improves the locality of the storage of rays computed in previous frames and the locality of cached rays' retrieval in subsequent frames as rendered by ray tracing algorithms. The approach uses 2-dimensional textures attached to the scene objects in a cubic arrangement as a cache structure. The outdated data in a given direction of the caching cube are replaced in runtime by the color of rays cast from the object's surface into the environment in that direction. When the cached data are up-to-date, the proposed approach replaces the computationally expensive bouncing of the rays leaving the object with the texture sampling mechanism of the GPU. Our local cache system has also enhanced the coherent memory access of the first interaction of those rays related to neighbor image pixels and has thus improved performance in that regard. We demonstrate the effectiveness of the proposed technique by implementing it as part of a ray tracing rendering system. Our results show increases of up to 250% in the frame rate of scenes composed of several reflecting objects. The relief mapping technique was adapted and combined with our technique to improve the visual quality of the results. Unfortunately, the visual quality was not improved, and the frame rate drops to values smaller than the standard ray tracing.

The two proposed strategies (CCT and CDCT) were tested with a variety of scenes. It was implemented in a state-of-the-art ray tracing solution and in a hybrid ray tracing pipeline. By doing so, we show that our techniques can be easily combined with other ray tracing solutions. To evaluate the two proposed strategies, they were compared against the same two ray tracing solutions that we implemented our strategies on. The aspects compared were the frame rate and reflection quality.

Improving performance by gathering related rays together in an object-driven cache memory is a powerful technique. We believe that this idea will lead to further benefits in the development of real-time ray tracing architectures for video games and virtual reality.

6.1 Future Work

Some of the limitations of our approach can be solved. For the static scene limitation, one possible solution would be to flush the information on the caching cube of objects that can be "seen" by moving elements. That could be managed by a visibility-graph-like structure that connects all of the objects that have reflection information of each other. Thus, when a given object moves, only the caching data of objects that are connected to that particular entity need to be marked as outdated.

The proposed approach has the limitation of only working well with convex objects. For concave objects, a possible solution would be to break them into convex parts.

The problem of insufficient memory of the GPU, caused by scenes with a massive number of objects, could be solved by implementing dynamic changes of the resolution, according to the distance and projected size of the reflective object. We could also have a caching cube that contains a group of objects, instead of only the individual caching cubes. Self-reflection would be a problem, but for small objects in the scene, it could generate coherent results. Another possible solution would be to use smaller texture sizes for objects that have specific reflectance properties, such as glossy or blurry materials.

To solve the problems of the invalid information when executing the relief mapping algorithm, one possible solution would be to rasterize the scene before the application starts, to store initial values of color and depth in the textures of the cube. That would make the relief mapping always return a valid color information, even if the ray tracing is not calculated for that particular point.

As a future work, we intend to analize the idle threads problem so that it could be avoided. To test if they really are the bottleneck of the relief mapping, we must have control of these threads. Unfortunately, the OptiX engine do not let us control the threads individually. When executing the ray tracing, we simply trace rays and the engine call threads, control them, and calculate possible intersection between the ray and the objects in the scene. A possible solution would be to find a tool that can control the call of these threads, or to implement our technique in a ray tracing engine that gives us the power to control the threads individually.

A promising area of future work is to use not only the rays leaving but also the rays hitting the object to populate the caching memory. The idea is to trace two new rays for each ray that hits the object, one in the reflection direction, and one in the direction of the incoming ray. This is possible due to the bidirectional nature of reflectance distribution functions [Pharr and Humphreys 2004]. By doing so, we believe that the update rate of cached rays (Figure 5.9) may increase significantly.

Bibliography

[Adelson and Bergen 1991] Adelson, E. H. and Bergen, J. R. (1991). The plenoptic function and the elements of early vision. In *Computational Models of Visual Processing*, pages 3–20. MIT Press.

[Aila and Karras 2010] Aila, T. and Karras, T. (2010). Architecture considerations for tracing incoherent rays. In *Proceedings of the Conference on High Performance Graphics*, pages 113–122.

[Appel 1968] Appel, A. (1968). Some techniques for shading machine renderings of solids. In *Proceedings of AFIPS*, pages 37–45.

[Assimp 2011] Assimp (2011). Open asset import library.

[Barboza and Clua 2011] Barboza, D. C. and Clua, E. W. G. (2011). GPU-based data structure for a parallel ray tracing illumination algorithm. In *Proceedings of Brazilian Symposium on Games and Digital Entertainment*, pages 11–16.

[Blinn and Newell 1976] Blinn, J. F. and Newell, M. E. (1976). Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–547.

[Catmull 1974] Catmull, E. E. (1974). *A subdivision algorithm for computer display of curved surfaces.* PhD thesis, University of Utah.

[Deering et al. 1988] Deering, M., Winner, S., Schediwy, B., Duffy, C., and Hunt, N. (1988). The triangle processor and normal vector shader: a VLSI system for high performance graphics. In *Proceedings of ACM SIGGRAPH*, pages 21–30.

[Foley and Sugerman 2005] Foley, T. and Sugerman, J. (2005). KD-tree acceleration structures for a GPU raytracer. In *Proceedings of ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 15–22.

[Glassner 1984] Glassner, A. S. (1984). Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–24.

[Greene 1986] Greene, N. (1986). Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29.

[Jensen 2001] Jensen, H. W. (2001). *Realistic Image Synthesis Using Photon Mapping.* A. K. Peters, Ltd.

[Lauterbach et al. 2009] Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. (2009). Fast BVH construction on GPUs. In *Proceedings of Eurographics*, pages 375–384.

[McMillan and Bishop 1995] McMillan, L. and Bishop, G. (1995). Plenoptic modeling: An image-based rendering system. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '95, pages 39–46, New York, NY, USA. ACM.

[Miller and Hoffman 1984] Miller, G. S. and Hoffman, C. R. (1984). Illumination and reflection maps: simulated objects in simulated and real environments. In *Proceedings of ACM SIGGRAPH.*

[Parker et al. 2010] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. (2010). OptiX: a general purpose ray tracing engine. In *Proceedings of ACM SIGGRAPH*, page Article No. 66.

[Pharr and Humphreys 2004] Pharr, M. and Humphreys, G. (2004). *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann.

[Pharr et al. 1997] Pharr, M., Kolb, C., Gershbein, R., and Hanrahan, P. (1997). Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of ACM SIGGRAPH*, pages 101–108.

[Phong 1975] Phong, B. T. (1975). Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317.

[Policarpo et al. 2005] Policarpo, F., Oliveira, M. M., and Comba, J. L. D. (2005). Real-time relief mapping on arbitrary polygonal surfaces. In *Proceedings of ACM I3D*, pages 155–162.

[Ruff et al. 2013] 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 *Conference on Graphics, Patterns and Images, 26 (SIBGRAPI).*

[Sabino et al. 2012] Sabino, T. L., Andrade, P., Clua, E. W. G., Montenegro, A., and Pagliosa, P. (2012). A hybrid GPU rasterized and ray traced rendering pipeline for real time rendering of per pixel effects. In *Entertainment Computing – ICEC*, volume 7522 of *LNCS*, pages 292–305. Springer.

[Saito and Takahashi 1990] Saito, T. and Takahashi, T. (1990). Comprehensible rendering of 3-D shapes. In *Proceedings of ACM SIGGRAPH*, pages 197–206.

[Shreiner and Group 2009] Shreiner, D. and Group, T. K. O. A. W. (2009). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th edition.

[Stich et al. 2009] Stich, M., Friedrich, H., and Dietrich, A. (2009). Spatial splits in bounding volume hierarchies. In *Proceedings of High-Performance Graphics*, pages 7–13.

[van Reeth et al. 1996] van Reeth, F., Monsieurs, P., Bekaert, P., and Flerackers, E. (1996). Ray tracing optimization utilizing projective methods. In *Proceedings of Computer Graphics International*, pages 47–53.

[Wald et al. 2006] Wald, I., Ize, T., Kensler, A., Knoll, A., and Parker, S. G. (2006). Ray tracing animated scenes using coherent grid traversal. In *Proceedings of ACM SIGGRAPH*, pages 485–493.

[Ward 1994] Ward, G. J. (1994). The radiance lighting simulation and rendering system. In *Proceedings of ACM SIGGRAPH*, pages 459–472.

[Whitted 1980] Whitted, T. (1980). An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349.

[Woods et al. 2001] Woods, D., Weber, N., and Dario, M. (2001). Developer's image library. Webpage.

[Yang et al. 2013] Yang, X., Xu, D., and Zhao, L. (2013). Efficient data management for incoherent ray tracing. *Applied Soft Computing*, 13(1):1–8.