A GPU-Accelerated Render Cache

Tenghui Zhu Rui Wang David Luebke

Department of Computer Science, University of Virginia

Email: {tz3d, rw2p, luebke}@cs.virginia.edu

Abstract

The Render Cache can assist a high quality renderer to achieve interactive frame rate by reusing cached 3D samples. These samples can be reprojected to the new frames when viewpoint changes. However, the original Render Cache imposes high computation overheads on the CPU. With highly parallelized vector computation units, the GPU is a good candidate to address the computational requirements of the Render Cache. In this paper, we present a system that implements the Render Cache on the GPU. Utilizing new features available on high end GPUs, our accelerated Render Cache can run more than four times faster than a highly optimized CPU implementation.

Keyword: Render Cache, Real-time Rendering, Image Reconstruction, Programmable Graphics Hardware

1. Introduction

In rendering, it is difficult to achieve high quality and efficiency at the same time. High quality renderers usually take minutes to hours to compute a frame, real time rendering only allows tens of milliseconds. Therefore, a gap remains between the goals of high quality rendering and real-time performance.

To bridge this gap, Walter et al. [8] propose the *Render Cache*. The key idea is to reuse the samples generated from the high quality renderer by caching old samples from previous frames and reprojecting them into the new frame for reuse. In details, the Render Cache stores recent generated samples into a fixed size cache, which contains the geometry, color and priority information. Whenever viewpoint changes, the samples in the cache are reprojected to the current frame to approximate the result. Since the sparse samples always result in some gaps, which are called *holes* here, they use filtering and depth culling to decrease such artifacts. Moreover, the ray tracer is not fast enough to update all the samples for each frame for real time rendering. Therefore, they introduce the priority map for sample updating. The priority map depends on the age of the samples and the distribution of the holes. It helps to guide the renderer to generate the samples to the places most needed. In general, Render Cache can effectively improve interactivity with various high quality renderers.

However, even after Walter et al. [9] improve the Render Cache by adding certain enhancements and optimizations, the algorithm still imposes high computation overheads on the CPU. Now with new features available on the GPU such as render to vertex array (RTVA) and render to color array (RTCA)[3], we show a successful GPU implementation of the Render Cache which is four times faster than the original algorithm.

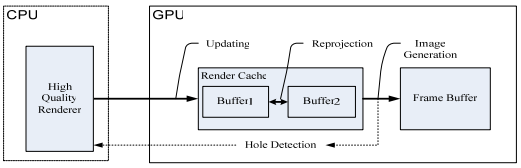

Figure 1: System overview. Note that most of the computations are implemented on the GPU.

2. Related Work

Recently there have been many advances in high-speed ray tracing to improve the algorithm toward interactive use. These include advances on the CPUs by highly optimized and parallelized software implementations [4,7], on the GPUs by taking advantage of the massively parallel streaming graphics processors [5], and on special-purpose hardware architecture [6]. However, even the fastest ray tracer fails to provide real-time performance given complex scene and surface shader. Therefore, many researchers have explored ways to exploit spatial-temporal coherence in ray tracing image sequences and reduce costs by reusing samples computed in previous frames[1,2,8,9].

3. Implementation

The major components of the GPU Render Cache are reprojection, sample updating, hole detection and image generation. Figure 1 shows an overview of our system and how it incorporates each of these computations. In the following sections we explore each of these processes and their implementation in detail.

3.1 Representation of Render Cache on GPU

Since the Render Cache consists of both of the location and color information for the primary ray hit point, we store it as one vertex array and one color array on the GPU. Since the age information for each sample is also important, we use the alpha channel of the color array to store it. With RTVA and RTCA techniques, we can upload the samples generated from

high quality renderer into the GPU Render Cache, and then reuse such data as vertices and colors. In our system, we store the vertex array and color array in two surfaces of one pixel buffer.

For both the vertex array and color array, we use 16-bit floating point precision. This is required by the reprojection because lower precision would introduce reprojection errors. Although 16-bit precision is not strictly necessary for the color array, on current hardware the MRT extension dictates that all render targets be the same precision.

3.2 Reprojection

Reprojection, consisting of many vector operations, is the heart of the Render Cache and a substantial portion of the computation. In previous papers [8,9], we can see that the reprojection step occupies 40% of the computation.

Using RTVA and RTCA, the reprojection in our implementation is straightforward. Similar to the original Render Cache, we store the original 3D position v_0 in object space and compute the reprojected position v' as $v' = \mathbf{P}'\mathbf{M}' v_0$, where $\mathbf{P}'$ is the new projection matrix and $\mathbf{M}'$ is the new modelview matrix. Therefore, we can utilize the vertex shader to compute v', and pass the color C and original coordinates v_0 as texture coordinates to the pixel shader, which stores each in the appropriate surface of the pixel buffer. In order to implement the aging of samples during the reprojection, we can add a step value d_A to the alpha value of the color C. The larger the d_A is, the sooner the samples get evicted.

Because the GPU cannot simultaneously read from and write to the same memory object, reprojection requires double buffers to serve as source and destination. As shown in figure 1, each buffer serves as a Render Cache.

3.3 Hole Detection

Reprojection unavoidably introduces holes where there are no samples to cover pixels. Holes can occur in several ways such as disocclusion or optical flow. Holes can also appear at the edges of the screen during camera rotation or panning. Note that to compensate for this effect, the enhanced Render Cache introduced a prediction component to guide sampling. Similarly, the samples we store in GPU Render Cache extend slightly beyond the screen size. Finally, our system sometimes creates holes due to *sample attrition*. Since the samples in the GPU Render Cache have a one-to-one correspondence with the pixels they occupy, sometimes two samples will reproject to the same pixel. In this situation depth buffer ensures that the nearer sample will overwrite the farther sample, so that later reprojections will contain holes where the now-overwritten samples would have gone. However, in practice we find that sample attrition accounts for only a small percentage of the holes caused by disocclusion and optical flow.

In order to implement hole detection, a straightforward solution would read back the per-pixel occupancy of the Render Cache from GPU. However, image readback is such an expensive process on modern graphics cards. Instead, we use a tiled occlusion query strategy: we split the image plane into small tiles, then we issue an occlusion query for each tile during reprojection. The occlusion query counts the number of samples which are holes. The results from all these occlusion queries provide an overview of the hole distribution in screen space.

3.4 Sample Update

After the tiled hole detection, the CPU renderer generates new samples and update the Render Cache on the GPU using the RTVA and RTCA. Two important intermediate data structures for this process are the priority map and the request map [8].

The priority map is based on the result of the hole detection. More holes means higher priority for sample updating. In addition to considering priority, the request map also needs a good spatial distribution to improve the visual results. We use a weighted random approach that first determines the number of samples in each tile according to the weight, and then chooses the samples randomly inside each tile. The weight w_{ij}' for each tile is computed as:

$$w'_{ij} = \alpha * w_{ij} + (1-\alpha) * u_{ij}, \quad w_{ij} = Q_{ij} / (\sum_{1}^{N}\sum_{1}^{N} Q_{ij}), \quad u_{ij} = \frac{1}{N * N}$$

Here Q_{ij} is the number of holes for the tile at ith row and jth column and u_{ij} is a constant representing the uniformly distribution. Setting $\alpha = 0$ achieves a uniform distribution without any priority, while setting $\alpha = 1$ creates a distribution that is completely prioritized. In our experiments we have found $\alpha = 0.25$ to be a good choice. Figure 2 shows an example of the priority and request maps. Note that the request map reflects the priority map but also provides some global spatial distribution.

Figure 2: A frame when viewpoint changing (left), priority map (center) and request map (right).

3.5 Image Generation

Conceptually, the image generation process simply renders the vertex and color arrays of the Render Cache to the framebuffer.

Filtering is important because the result image will unavoidably contain some holes. The enhanced Render Cache [9] uses a 3x3 weighted filter and a 7x7 box prefilter to improve the image quality. We can implement these filters easily on the GPU; The 3x3 filter applies to all pixels, but the 7x7 filter applies around large holes that cannot be covered even using the 3x3 filter. Therefore, we apply a stencil buffer as a mask to record the pixels which are untouched after the 3x3 filter, so that the 7x7 filter only affects the pixels belonging to larger holes.

4. Results and future work

We compare the CPU-optimized enhanced Render Cache [9] and our GPU implementation running on a machine with 2.8 GHz Pentium 4 CPU and nVidia 6800 card. The GPU Render Cache achieves 123 frames per second at 512x512 resolution (also supported by the CPU Render Cache, and thus our standard for comparison) and 38 frames per second on a 1024x1024 image. This is at least four times faster than the CPU version. Figure 3 shows the running time of both systems.

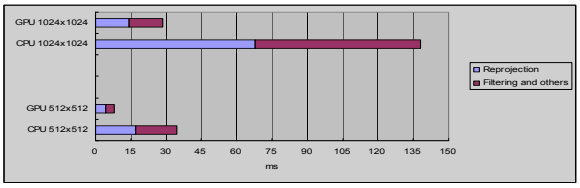

Figure 3: The performance of our system compared with an optimized CPU implementation of Walter et al., operating on an image resolution of 512x512 and 1024x1024.

Note that in this comparison, both systems run with the same functionality, including reprojection, filtering, prediction, depth culling and image generation. However, we do not include the updating and new sample generation in either system, since that will include the computation time of underlying ray tracer.

Our implementation shows that it is feasible and beneficial to accelerate the Render Cache on the GPU. However, there are certain limitations in our implementation. One is sample attrition. We are considering schemes to address this with more flexible storage allowing multiple samples per pixels, but the more complex control flow that this would require might unacceptably hinder performance on current GPUs.

Another limitation is caused by the tile based priority map, which prevents precise sample generation. With filtering, such artifacts are not easily observable, but it takes longer to converge to a perfect image. The higher bandwidth of the soon-to-be ubiquitous PCI-Express bus will certainly help.

References

[1]BALA, K., DORSEY, J., TELLER, S. 1999. Radiance interpolants for accelerated bounded-error ray tracing. ACM Trans. Graph, 18, 3, 213-256.

[2]BISHOP, G., FUCHS, H., MCMILLAN, H., SCHERZAGIER, E.J. 1994. Frameless rendering: double buffering considered harmful. Proc. ACM SIGGRAPH, 175–176.

[3]KIPFER, P., SEGAL, M., WESTERMANN, R. 2004. “UberFlow: A GPU-Based Particle Engine”,Graphics Hardware 2004

[4]PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P-P. 1999. Interactive Ray Tracing. Proceedings of Interactive 3D Graphics (I3D), 119–126.

[5]PURCELL, T-J., BUCK, I., MARK, W-R., AND HANRAHAN, P. 2002. Ray Tracing on Programmable Graphics Hardware. ACM Transactions on Graphics, 21(3):703–712. (Proceedings of SIGGRAPH 2002)

[6]SCHMITTLER, J., WALD, I., AND SLUSALLEK, P.2002. SaarCOR – A Hardware Architecture for Ray Tracing. In Proceedings of Eurographics Workshop on Graphics Hardware, pages 27–36.

[7]WALD, I., KOLLIG, T, BENTHIN, C., KELLER, A., AND SLUSALLEK P. 2002. Interactive Global Illumination using Fast Ray Tracing. Rendering Techniques 2002, pages 15–24.

[8]WALTER, B., DRETTAKIS, G., AND PARKER, S. 1999. Interactive rendering using the render cache. Eurographics Rendering Workshop 1999. Granada, Spain.

[9]WALTER, B., DRETTAKIS, G., AND GREENBERG, D. 2002. Enhancing and optimizing the Render Cache. In 13th Eurographics Workshop on Rendering, 37-42.