Enhancing and Optimizing the Render Cache

Bruce Walter† and George Drettakis‡ and Donald P. Greenberg†

† Program of Computer Graphics, Cornell University, USA

Abstract
Interactive rendering often requires the use of simplified shading algorithms with reduced illumination fidelity. Higher quality rendering algorithms are usually too slow for interactive use. The render cache is a technique to bridge this performance gap and allow ray-based renderers to be used in interactive contexts by providing automatic sample interpolation, frame-to-frame sample reuse, and prioritized sampling.

In this paper we present several extensions to the original render cache including predictive sampling, reorganized computation for better memory coherence, an additional interpolation filter to handle sparser data, and SIMD acceleration. These optimizations allow the render cache to scale to larger resolutions, reduce its visual artifacts, and provide better handling of low sample rates. We also provide a downloadable binary to allow researchers to evaluate and use the render cache.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Display algorithms

1. Introduction

There has been a divergence between the rendering algorithms and illumination models used for interactive use and those used for high quality realistic image generation. For users this has often required them to switch between two different rendering modes. Lower quality renderers are used for interactive tasks such as modelling, viewpoint selection, and walkthroughs. Computationally expensive illumination effects such as shadows, reflections, refraction, and global illumination are provided at low fidelity or omitted entirely. To see their work under the full illumination model, the user must switch to a higher quality non-interactive renderer that often takes minutes or longer to produce a single image. Such mode switches disrupt the user’s concentration, and can make fine-tuning their work a very tedious process.

We proposed the render cache to bridge this gap and allow slower renderers to be used in interactive contexts. It relies on an underlying renderer to perform all shading computations, but communicates with the renderer asynchronously allowing the frame rate to be independent of the speed of the underlying renderer. The renderer’s speed does still affect on image quality and visual convergence rate.

The render cache stores recent shading results from the underlying renderer as colored 3D points in a fixed size cache. For each frame, these points are projected onto the current image plane and filtered to reduce visibility errors and fill small gaps in the data. This allows us to quickly approximate the current image even if only a small fraction of the pixels are being rendered each frame. In addition, the render cache prioritizes where new rendering results are most needed and guides the image plane sampling of the underlying renderer.

In this paper we discuss a number of optimizations and extensions beyond the original render cache algorithm to increase performance at larger image resolutions, and improve image quality during rapid camera motions and when using lower sampling rates. Among the enhancements that we introduce are: a split projection and tiled z-buffer approach for better memory coherence, predictive sampling to request data for new regions before they become visible, a prefilter stage with a larger kernel footprint to fill in larger gaps between the point data when necessary, and a highly optimized implementation including use of SIMD instructions.

With these improvements, we believe that the render cache is now ready to become a widely used tool in software-based interactive rendering. Although the basic techniques are not difficult, creating a highly optimized implementation takes a considerable amount of work. Thus we have decided to release a binary version of our implementation at
Walter et. al. / Enhancing the Render Cache

http://www.graphics.cornell.edu/research/interactive/rendercache
to help other researchers evaluate and use the render cache.
The provided library and application are free for educational
non-commercial use, and we encourage other researchers to
integrate the render cache into their own rendering systems.

1.1. Related Work
We will only include a quick survey of recent related tech-
niques here. See the references for more comprehensive sur-
vey of older related work. In this paper we are specifically
building on the render cache approach\textsuperscript{10}, but researchers
have also proposed a number of related techniques. Indeed
the idea of layering interactive display processes over high
quality but slow renderers is becoming increasingly popular.

Much of the recent work has concentrated on display rep-
resentations that can be directly displayed using standard
graphics hardware. This allows for very high frame rates
and image resolutions by taking advantage of the consid-
erable amount of specialized graphics hardware that is eas-
ily and cheaply available. For example, both the Holodeck\textsuperscript{11}
and Tapestry\textsuperscript{4} systems construct a display mesh by project-
ing rendering results onto the sphere of directions surround-
ing the current viewpoint. These points are then triangulated
to create a Gouraud-shaded mesh for hardware display.

Corrective texturing\textsuperscript{6} starts with a conventional hardware
rendering of the scene and then constructs view-dependent
projective textures to "correct" the appearance of objects
when the hardware does not match that produced by the un-
derlying renderer (e.g., on reflective or refractive objects).

Another approach\textsuperscript{7} constructs a Gouraud-shaded display
mesh by refining the input geometry mesh in a prioritized,
view-dependent, and lazy manner as rendering results be-
come available. It also provides automatic de-refinement of
the mesh when shading changes are detected.

Each of these approaches has its strengths. Using a
Gouraud-shaded display mesh allows the output image to be
generated at any resolution and provides better interpolation
when the samples are very sparse. However inserting new re-
sults into such meshes is expensive compared to the render
cache, thus they work best at very low sampling rates (e.g.,
when the underlying renderer would take several minutes or
longer to produce an image on its own). Corrective textur-
ing works best when the hardware shading matches the true
shading for most surfaces.

Another approach to achieve interactivity is to create a
highly optimized ray tracing engine\textsuperscript{9}, but while this cer-
tainly helps, it is not currently sufficient for interactive per-
formance on complex models with complex shading models.
The optimized ray engine can accelerate the underlying ren-
derer while still using a separate interactive display process
such as the render cache.

2. RenderCache Overview
We will provide only a brief overview of the render cache,
so that we can concentrate on the new enhancements that
we are introducing. A more detailed description of the basic
render cache algorithms can be found in\textsuperscript{10}.

The render cache works by caching rendering results pro-
duced over many frames and using them to estimate the cur-
rent image. The results are cached as 3D points with an as-
associated color. For each frame, any new rendering results are
integrated into the fixed size point cache, and then projected
onto the current image plane. Because there is generally not
a one-to-one mapping between points and pixels, we next
apply some filters to correct for gaps in the point data. A
depth cull heuristic is used to remove points that should not
be visible and an interpolation/smoothing filter is used to fill
small gaps in the point data. The result is an estimate of the
current image.

During image reconstruction, a priority image is also gen-
erated to encode where new rendering samples are the most
needed to improve the quality of future frames. Since we ex-
pect that only a small number of pixels can be rendered per
frame, it is very important to guide the location of those
samples for maximum benefit. An error-diffusion dither is
used to select the locations where the renderer should spend
its effort. The render cache can then immediately begin com-
puting the next frame without waiting for the renderer. The
new samples will be integrated into the point cache once they
become available.

In this paper we have added several additional stages to
3. Enhancements and Optimizations

3.1. Predictive Sampling

The basic sampling algorithm in the render cache is purely reactive. Sample locations are chosen based on where in the current image, more data was needed. While this helps to concentrate scarce rendering resources where they are most needed, it does not work well when large regions are becoming newly visible each frame (e.g., see Figure 2).

There is always at least one frame of latency between when a new rendering request is generated and when the result can be computed and integrated into the point cache. This latency may be even longer when running the underlying renderer in a parallel distributed configuration, due to network latencies.

The solution is to predict several frames ahead of time when regions without data are likely to become visible. We project the points onto a predicted image plane using predicted camera parameters and then look for large regions without data. This projection can be done much more cheaply than the non-predicted projection for several reasons. Because we do not need to resolve the depth ordering of the points, there is no need to use a z-buffer with this projection. Also since we are only interested in larger regions without data, we can project the points onto a lower resolution image.

We use an image with one quarter resolution in each dimension (or 1/16 as many pixels) and store each pixel in one byte (1 if at least one point maps to it, 0 otherwise). This allows the entire predicted occupancy image to fit in the processor’s cache. This avoids the need for a two pass projection (as discussed below).

Once we have computed the occupancy image, we generate a rendering sample request for each pixel which did not have a point map to it. If there are more empty pixels than allowed requests, we use a simple decimation scheme which takes roughly every nth sample in scanline order. For each frame, the render cache is given a target number of rendering sample requests to generate. By default, we allocate up to half these requests to the predicted sampling, with the remainder generated by the normal sampling stage.

This prediction scheme fills predicted empty regions with point data that is just dense enough to allow the prefilter and interpolation stages to fill in the gaps, but sparse enough to avoid wasting too much effort on regions that might never become visible. Prediction significantly improves image quality during camera motions at an acceptably small cost. It consumes roughly 13% of the total render cache execution time for one frame. We rely on the application to provide the predicted camera since it has the most up-to-date information about what the user is currently doing.

3.2. Tiled Z-Buffer for Memory Coherence

Computational speeds continue to advance at a much faster rate than memory speeds, making memory latency an increasingly important bottleneck. Thus making sure that algorithms have good memory coherence and predictable memory access patterns is important. While most of the render cache exhibits nice linear memory access, the combined projection/z-buffer as done in the original render cache does not. Because the points in the cache are unordered, directly projecting them onto the image plane results in a nearly random access pattern to the image plane data structures.

This was not a major issue in the original render cache implementation because it used smaller images, and ran on a processor with relatively large caches. However when we compared an earlier implementation on a 1GHz Pentium III and a 1.7GHz Pentium 4, we found that all the stages were accelerated on the Pentium 4 except for the combined projection/z-buffer. The memory latency on the Pentium 4 was slightly worse due to its use of RDRAM memory, and that stage was almost entirely memory latency-bound because at 512x512 the image plane data structures occupy 3 megabytes and are too large to fit in cache.

One way to make the algorithm more cache friendly is to divide the image into regions, or tiles, that are small enough
Figure 3: The 7x7 uniform filter and 3x3 weighted filters used in the prefilter and interpolation/smoothing stages respectively.

...filter kernel. Uniform kernels have the advantage that...
Figure 4: These images were generated using only the 3x3 filter (left), only the 7x7 prefilter (center) and both filters (right). We artificially lowered the point density on the left side of each image and cleared the image to black between frames to show where data is missing. The 3x3 filter alone has trouble filling in the larger gaps in the points, while the 7x7 filter produces objectionable blurring in the dense regions. Using both filters allows us to fill in larger gaps while preserving sharpness in the dense regions.

The new instructions also provide techniques for turning control dependencies into data dependencies. For example, there are instructions that set a variable to a mask of zeros or ones based on the comparison of two values. Boolean operations can then be used to select between two values based on the mask. Viewpoint clipping can be implemented by this technique without using a branch instruction. Because unpredictable branches are relatively expensive on modern processors, removing them can increase performance.

As mentioned earlier, memory latency is becoming more and more of a bottleneck. We have tried to carefully organize our data structures to minimize the amount of memory that must be accessed by any single computation stage and to ensure that the memory is accessed in a predictable linear manner. Adding prefetch instructions can also help to hide memory latency, though this is somewhat less expensive on modern processors, removing them can increase performance.

Table 1: This table shows the time required by each stage of the render cache for a 512x512 image on a 1.7GHz Pentium 4 machine where roughly 8000 new samples are being added each frame. Times are shown in both milliseconds and as a percentage of the total time. These timings do not include any computation by the underlying renderer or the time to display the image after it has been computed. Thus in practice, actual frame time may be longer depending on the system configuration. We also show the average number of micro-operations being executed per cycle for each stage to indicate if it is computation or latency bound.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Time (ms)</th>
<th>Time (%)</th>
<th>μ-ops / cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update Points</td>
<td>7.7</td>
<td>12%</td>
<td>0.7</td>
</tr>
<tr>
<td>Predicted Projection</td>
<td>7.9</td>
<td>13%</td>
<td>0.9</td>
</tr>
<tr>
<td>Predicted Sampling</td>
<td>0.2</td>
<td>3%</td>
<td>2.2</td>
</tr>
<tr>
<td>Project and Tile Sort</td>
<td>15.8</td>
<td>25%</td>
<td>1.2</td>
</tr>
<tr>
<td>Z-Buffer Tiles</td>
<td>8.9</td>
<td>14%</td>
<td>0.5</td>
</tr>
<tr>
<td>Depth Cull</td>
<td>3.2</td>
<td>5%</td>
<td>1.2</td>
</tr>
<tr>
<td>Prefilter</td>
<td>5.9</td>
<td>10%</td>
<td>1.5</td>
</tr>
<tr>
<td>Interpolate / Smooth</td>
<td>6.5</td>
<td>11%</td>
<td>1.9</td>
</tr>
<tr>
<td>Sampling</td>
<td>6.0</td>
<td>10%</td>
<td>1.2</td>
</tr>
<tr>
<td>Total</td>
<td>62.1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We have included micro-operations executed per cycle statistics in Table 1. In a Pentium 4 processor, instructions are broken down into micro-operations and, theoretically, up to three micro-operations can be issued per cycle. Competition for execution units, dependency chains between operations, branch mispredictions, and cache misses mean that the actual rate is always lower than this. In practice maintaining a sustained rate of one micro-operation per cycle or better means that you are doing well and that the execution is com-
putation bound. Because z-buffering requires only minimal computation, its speed is limited primarily by the latency of the L2 cache. The total point cache data occupies around 7 megabytes and the need to access this large amount of data slows the point update and, to a lesser extent, the predicted projection. Nevertheless, most of the execution is computation bound which is good news because it means that performance should continue to scale with increasing processor speeds.

The render cache runs entirely on one processor, but, when available, other processors can be used to offload other tasks such as rendering, handling the user interface, and displaying the computed images. A frame time of 62ms corresponds to a potential frame rate of 16 frames per second, but the actual frame rate will be somewhat lower depending on what else the processor must handle. In practice, we are seeing frame rates up to 14 fps in a dual processor configuration and 12 fps in a single processor configuration.

The addition of the prediction stages has significantly reduced the visual artifacts during rapid camera motion, although artifacts are still apparent if the underlying renderer is not producing enough new samples to fill in the new regions at least sparsely. In practice we find that the render cache works well when running at frame rate 10 to 100 times faster than the speed of the underlying renderer (i.e. 1% to 10% of the pixels are being rendered per frame).

The prefilter with its larger kernel and the point eviction mechanism further improve performance at low sampling rates, by allowing interpolation over large distance when necessary and by allowing stale data to be removed from the cache more quickly. Also the use of a tile z-buffer approach has significantly increased performance for larger images. Our experiments indicate that the frame time scales roughly linearly with the number of pixels for images up to at least 1024x1024. The original render cache showed nonlinear scaling once the image plane data structures became too large to fit in cache.

4.1. Public Availability

With the current improvements in speed, scalability, and visual quality, we believe the render cache is ready to become a widely used tool in software interactive rendering. To further this goal, along with this paper we are releasing a downloadable binary version of the render cache that is free for educational, non-commercial use. The binary can be downloaded from the address below. Because it contains SSE 2 optimizations, it requires a Pentium 4 processor or better. See the web page for more details.

http://www.graphics.cornell.edu/research/interactive/rendercache

We have found that it is almost impossible to convey interactive performance using still images and difficult to do so even in videos. The true test of any interactive system is always to operate it yourself. We strongly encourage the reader to download and try the render cache for themselves. The sample application allows the user to dynamically disable our enhancements such prediction and the prefilter to better understand how they impact and improve visual quality. Moreover we further encourage readers to try using the render cache as a front end to their own rendering systems. The render cache can be easily connected to most ray-based renderers. Again, more details can be found on the website.

Acknowledgements

Thanks to Hector Yee for providing the lotus model and to our anonymous reviewers for their helpful comments. This work was supported by Intel Corporation and the NSF Science and Technology Center for Computer Graphics and Scientific Visualization (ASC-8920219).

References