Implementing the Render Cache and the Edge-and-Point Image On Graphics Hardware

Edgar Velázquez-Armendáriz
Department of Computer Science
Cornell University

Eugene Lee
Department of Computer Science
Cornell University

Bruce Walter
Program of Computer Graphics
Cornell University

Kavita Bala
Department of Computer Science
Cornell University

Figure 1: Sample scenes rendered using the GPU version of the Edge-and-Point Image. From left to right: Cornell Box, Chains, Mackintosh Room, David Head and Dragon with Grid.

Abstract

The render cache and the edge-and-point image (EPI) are alternative point-based rendering techniques that combine interactive performance with expensive, high quality shading for complex scenes. They use sparse sampling and intelligent reconstruction to enable fast framerates and to decouple shading from the display update.

We present a hybrid CPU/GPU multi-pass system that accelerates these techniques by utilizing programmable graphics processing units (GPUs) to achieve better framerates while freeing the CPU for other uses such as high-quality shading (including global illumination). Because the render cache and EPI differ from the traditional graphics pipeline in interesting ways, we encountered several challenges in using the GPU effectively. We discuss our optimizations to achieve good performance, limitations with the current generation hardware, as well as possibilities for future improvements.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms; I.3.7 [Computer Grpahics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: graphics hardware, interactive rendering, sparse sampling, silhouette and shadow edges

1 Introduction

Rendering, including expensive shading effects such as global illumination, is still too expensive for interactive applications. Interesting non-local effects like soft shadows, glossy reflections, and indirect illumination can only be included by using highly simplified approximations of varying quality and cost. One potential solution is to adapt expensive rendering to run directly on the graphics hardware as is done in hardware ray-tracing [4] and hardware photon mapping [5]. While promising, these systems are still limited in the quality and scene complexity they can handle at interactive rates.

An alternate approach is to introduce an intelligent display process that decouples expensive shading from the image framerate. We implement two such intelligent display algorithms, the render cache [12, 13] and the edge-and-point image (EPI) [1]. These point-based approaches scale to complex scenes by exploiting image coherence to achieve interactive performance with expensive, slow shading. By decoupling shading from display they bridge the gap between expensive shaders and fast display allowing high quality shaders to run asynchronously producing results at whatever rate they can.

The render cache stores a fixed size cache of shading results as colored 3D points and reprojects and filters them to approximate each frame. It also prioritizes where future shading points (samples) should be computed to maximize image quality. The edge-and-point system builds on the render cache by adding explicit tracking of important discontinuities like silhouettes and shadow boundaries. These edges are then used for edge-respecting interpolation and inexpensive anti-aliasing to greatly improve image quality.

In this paper we present a multi-pass rendering system that accelerates the render cache and EPI by adapting them to take advantage of modern programmable GPUs. Because these algorithms consist of many components each of which differs somewhat from standard hardware rendering, we encountered many challenges in adapting to the various programming limitations imposed by GPUs. We will discuss the speedups we achieved, and GPU related optimizations and limitations.

In the next section we briefly review related work. Section 2 gives a brief overview of the render cache and the EPI. Section 3 presents the GPU implementation strategies and how we handled hardware limitations. Our results are in Section 4 and lastly in Section 5 we conclude and propose future improvements.

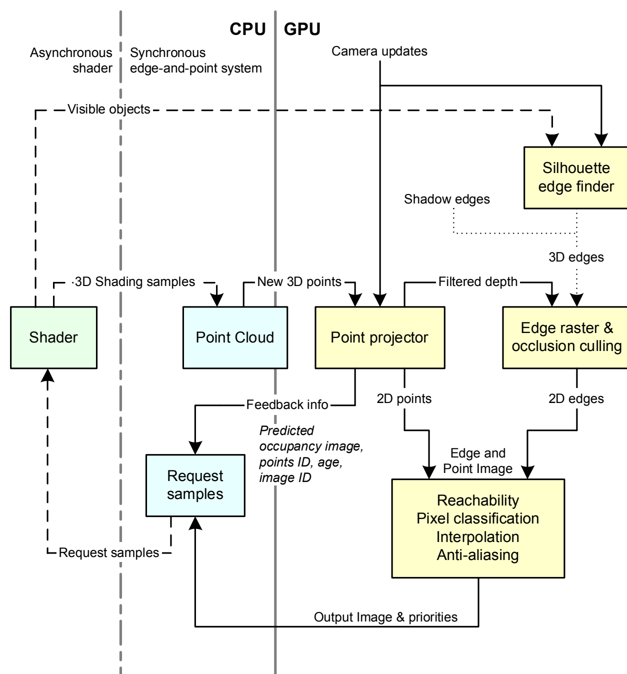

Figure 2: Overview of the GPU system architecture.

1.1 Related work

There has been considerable recent interest in intelligent display processes that automatically exploit spatial and temporal coherence to allow interactive use of shading algorithms that would otherwise be too slow for interactive use. Some examples also use graphics hardware, including the shading cache [11], corrective texturing [9], tapestry [7], and adaptive frameless rendering [2]. In a non-interactive context, irradiance caching [8] stores irradiance samples sparsely on objects to speed up the shading of dynamic scenes that use global illumination.

Pighin [3] explored using edges to improve image reconstruction from sparse points for progressive update of a still image. The edge-and-point renderer and the silhouette shadow maps [6] each explore image representations that combine sharp discontinuities with sparse samples for different goals. Silhouette shadow maps embed approximate edges to reduce artifacts from limited resolution shadow maps.

Another promising approach is to accelerate traditionally slow shading algorithms by mapping them to graphics hardware, for example, implementing ray tracing [4] or photon mapping [5] on programmable GPUs. Szirmay-Kalos et al. presented approximate ray tracing using distance impostors [10] as a way to generate estimated reflections, refractions and caustics using specially addressed environment maps. Since these can be used to produce shading samples for the display process, we view these approaches as orthogonal and complementary to the work presented here.

2 Algorithm Overview

In this section we will briefly review the main components of the render cache and edge-and-point image (see Figure 2). For more detail refer to their original papers [12, 13, 1]. Both algorithms use a fixed size cache of colored 3D points to store recent shading results. In each frame these points are projected on the image plane and the resulting image is processed by several filters to reduce potential visual artifacts such as gaps and occlusion errors. Feedback data from this process is used to prioritize the location of new shading results to improve future images. The edge-and-point image includes additional steps to compute discontinuity edges such as silhouettes and shadow boundaries, to rasterize these edges, and it also includes additional filters to improve image quality using these edges.

2.1 Point cloud and projection

Each point caches the result of a shading computation and consists of a 3D location, a color, an age, and the pixel to which it most recently projected (its image id). These points are stored in a fixed size cache called the point cloud that is slightly larger than the number of pixels. For each frame, these points are projected onto the current image plane using z-buffering. The projection also stores 4 bits of sub-pixel location information with each projected point for later EPI use.

Because of dynamic changes such as camera motion, there will not usually be a one-to-one mapping between points and pixels. A depth cull filter removes points that likely should have been occluded except that no foreground point mapped to that pixel. It also creates an interpolated depth image used during edge rasterization.

2.2 Edge finding and rasterization

The EPI method tracks two kinds of edges: silhouette edges and shadow edges. Silhouette edges are view-dependent and need to be recomputed each frame. Shadow edges are view-independent and are only recomputed when their associated light, blocker, or receiver moves. For each frame, the relevant edges are found and then rasterized onto the image. Each time an edge crosses a pixel boundary, that intersection is stored with 1/8 of a pixel precision into a bit mask. Pixels are treated as squares for this operation and since they share boundaries only two 8 bit masks have to be stored per pixel.

2.3 Image filters

Once the points have been projected and the edges rasterized, a series of image filters are applied: pixel classification, reachability, interpolation and anti-aliasing.

The system classifies each pixel either as empty, simple or complex based on the edge crossings recorded on its boundaries. Empty pixels have zero or one crossings, simple pixels have exactly two intersections and all others are classified as complex.

Simple pixels are approximated by a single edge. This edge divides the pixel into a primary and a secondary region. The empty and complex pixels contain only a primary region. Primary region colors are computed during interpolation whereas secondary colors are estimated during anti-aliasing. If a simple pixel contains a point, the side where it is will be the primary region or the point is eliminated if we don't have enough sub-pixel precision to determine its sidedness.

Pixels and points are considered reachable if there is a reasonably direct path between them that does not cross any edges (for a detailed explanation please refer to Figure 6 of the EPI paper [1]). The reachability filter determines this from every pixel to each of its neighbors in a 5x5 pixel neighborhood and stores this result in a per-pixel bit mask for use by interpolation. Reachability computation includes determining reachability between immediate neighbors, chaining reachability through intermediate pixels, and combining different paths. Reachability is symmetric.

Interpolation then combines all the nearby reachable samples using a center-weighted kernel to estimate the primary color. Missing or unreachable samples are given a weight of zero and the weights renormalized for each pixel. Complex pixels ignore reachability since their edge configuration is too complicated to reconstruct, but

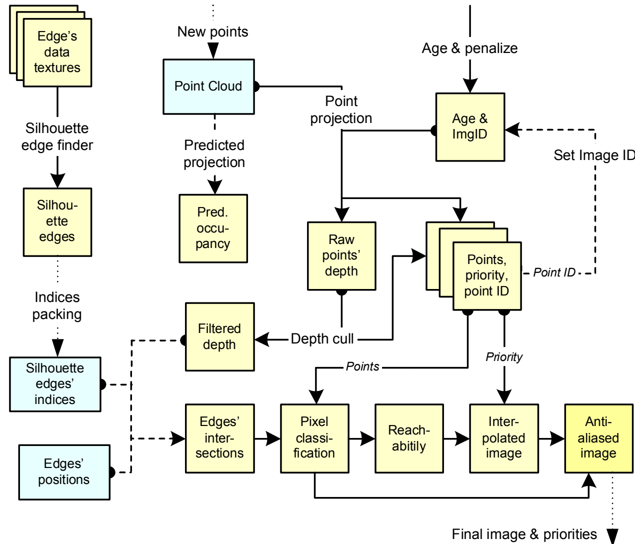

Figure 3: Data flow on the GPU. The yellow squares on the figure represent textures, the blue rectangles VBOs. Dashed lines are vertex/pixel shaders, solid lines pixel programs and dotted lines processes not handled directly by the GPU.

they tend to be rare and in portions of the image where error is less noticeable anyway. EPI uses a single 5x5 interpolation while the render cache uses a combination of 3x3 and 7x7 kernels without reachability.

After interpolation, simple pixels are anti-aliased using an area weighted combination of their primary color and an estimated secondary color which is copied from a neighboring pixel chosen based on edge orientation.

2.4 Sample prioritization and request

The system expects that only a small number of new samples (points) can be shaded per frame, so it is important to prioritize their locations for maximum benefit. During image reconstruction, we also create a priority image that estimates the value of generating a new shading result at a pixel. The priority is based on the point's age for pixels with points and on the total interpolation weight for pixels without points to reflect the local point density. An error-diffusion dither then converts this priority image to a sparse set of locations where new shading results will be requested [13].

New results are integrated into the point cloud as they become available, overwriting older points. Aging penalties are applied to points that do not project inside the image and in image regions where shading changes are detected. Points exceeding a maximum age are not drawn.

Because there is a lag between when a new sample is requested and when the shaded result is returned, we also include some predictive sampling. The camera position is predicted several frames ahead and a simplified projection is used to detect large gaps in point data and samples requested for these regions. This allows the system to request points for regions before they become visible, thus improving visual quality.

3 Mapping to the hardware

In this section we will present how we modified the combined render cache and EPI system to take advantage of the GPU. Based on computational similarity, we have divided the system components into three parts: point processing, edge finding and rasterization, and image filters. As each part required different solutions, we will discuss the issues and limitations we encountered.

Most of the processing was successfully moved to the GPU but some components like the point cloud management and dither for prioritized sampling had to be left on the CPU.

3.1 Data Flow on the GPU

We will briefly explain the data flow on the GPU, illustrated on Figure 3. More detailed explanations of each stage are on the corresponding sections. The system uses different types of data structured in various ways depending on the origin and intended use of the information. The system builds four textures with the edges and normals of the current model and a Vertex Buffer Object (VBO) with the position of the edges. This process is executed only once during the scene's initialization.

As new points arrive the system updates the GPU point cloud, organized as a VBO which stores the samples' position and color, and as a texture which stores the age and last Image id of each point. Every frame a pixel shader updates the points' considering possible penalizations and then the system projects the point cloud using vertex and pixel shaders. This projection step writes to a raw depth texture the z-value of the projected points and draws simultaneously to textures storing the color and subpixel location data for each point and priority values based on their age. It also tracks which point was projected to each pixel by writing the point's index (Point id) into the pixel. The depth cull filter deletes points which are likely to be occluded from theses textures. A second cloud projection using simplistic shaders generates the predicted occupancy image which is drawn to a texture.

A pixel shader uses the edges textures to detect the silhouette using an intermediate texture to mark which edges belong to the silhouette. The CPU reads back this texture and writes the index of the proper edges to a tightly packed index VBO. Using these fast VBOs and the filtered depth texture as a read-only z-buffer wwe rasterize the silhouette edges encoding the position of their intersections with the pixels in a texture. Using the edges' intersections, the projected color and priority textures, successive pixel shaders apply the different filters of the EPI algorithm to intermediate textures. The final output is a texture with the anti-aliased image and the priorities for requesting new samples.

3.2 Point projection

The point cloud is replicated into two copies, one on the CPU side for management and one on the GPU side for fast projection. The point data on the GPU is split into both a Vertex Buffer Object (VBO) containing the 3D positions and colors and a texture containing their ages and last pixel coordinates (image id). Positions and colors only change when new points overwrite old ones, while the ages and image id need to be updated every frame. The age texture is read by a shader program during each projection.

The point projection writes the point image's color, its subpixel information, a priority value (based on the point's age) and the point's id (index in point cloud) into the appropriate pixel. Therefore we chose to use Multi Render Target (MRT) to project the point cloud only once, writing simultaneously to four textures (three color textures and one depth texture).

A combination of vertex and shader programs project the points, and additional passes are used to update the point's image ids and ages including aging penalties. We use Framebuffer Objects (FBOs) to be able to write to the age and image id texture, but we need to keep two copies (swapped each frame) as it is used as both a source and destination in these operations. In general FBOs allow us to pass data from one pass to the next with minimal overhead. To avoid excessive state changes, we perform the projection

for predictive sampling immediately after the normal point projection. The predictive projection is done without depth testing and simply writes a one into each occupied pixel.

The depth cull deletes samples from the projection that are likely to be occluded by other surfaces. A pixel shader computes a filtered depth, which is the average depth of the points that were projected in a 3x3 neighborhood, plus an offset. This stage then erases the pixels that are behind the calculated depth using alpha blending and MRT. The depth cull also writes the 3x3 average depth to a z-buffer for later use by the edge rasterization stage.

Converting the priority map into sample requests and integrating new samples into the point cloud are still handled by the CPU. Error-diffusion dither for sampling just does not map well onto the restricted dataflow allowed on GPUs. Creating or updating geometry on the GPU is not yet well supported, so updating the point cloud has been left as a CPU task. Unfortunately this requires the CPU to access and update the VBO each frame causing extra CPU to GPU memory transfers. To help minimize this the CPU keeps a copy of the point cloud in main memory in addition to the GPU copy.

3.2.1 Limitations

We found that the point projection stage is mainly bandwidth limited. Due to the hybrid nature of our approach, we have to write data to the point cloud VBO and read back data from the GPU each frame. This traffic has a very significant effect on our overall performance. The data that is read back is the predicted occupancy image, point ids, the point ages and image ID, and the priority image.

When updating the point cloud, we map the VBO as write-only and only overwrite the points that have actually changed. The corresponding ages are reset by drawing a point into the age texture. This technique avoids copying the complete point cloud every frame, but there is still significant overhead. Our experiments show that projecting a completely static point cloud would be three times faster than our results where the VBO is modified each frame. Thus, implementing point cloud update completely on the GPU would result in a very large performance gain and hopefully this will be possible on the next generation of cards.

To reduce the dynamic VBO penalty, we modified the predictive sampling projection to project only a quarter of the points per frame but drawn as larger splats. This works well because the predictive sampling is only looking for large gaps in the point data and the points are distributed randomly through the point cloud.

3.3 Silhouette detection

The EPI [1] uses hierarchical structures to find silhouettes. This is efficient for high complexity scenes, but does not map well to the GPU due to its high branching factor and data dependencies. Since transferring the edges to the GPU each frame has a significant cost, we decided to implement a more brute-force edge finding method directly on the GPU. The shadow edges are still detected on the CPU, because they require a lot of dynamic information from the scene and change much less frequently as they are view-independent. They only need to be uploaded to the GPU when they change.

The system stores all the model's edges in a static VBO, so they can be drawn quickly, and as textures, to detect the silhouette edges with a pixel shader. A pixel shader reads each edge along with the normals of the neighboring faces and decides if it is a silhouette edge or not. Silhouette edges border one front-facing and one back-facing polygon. Then, the system writes either a zero or a one to a texture to indicate if the corresponding edge is on the silhouette. Next, we compress this sparse texture into a packed array of edge indices using the CPU and a readback. Then, the silhouette edges can be efficiently drawn using these indices and the static VBO already stored on the GPU. This process has turned out to be surprisingly fast, and is faster than the CPU-based hierarchical silhouette finding even for complex scenes. We continue to use the CPU hierarchy for efficiently finding shadow edges.

3.3.1 Limitations

The GPU silhouette detection is limited by the fill rate of the GPU. It is constrained by the size of the textures that record all the scene edges and neighboring normals and the need to draw a texture with a pixel for each scene edge. For our scene with the most edges (dragon) the edges did not fit in the static VBO in the GPU memory and we had to fall back to main memory to store them, leading to slightly diminished performance on the edge raster stage. To reduce space we store the normals using a 16 bit floating point format, but even higher compression could be used.

3.4 Edge raster

After we found the silhouette and shadow edges, the next stage is to rasterize them onto the image plane. Unlike normal edge rasterization, we want to record their intersections with pixel boundaries to sub-pixel precision. We encode the boundary intersections as a 8 bit mask per pixel boundary and use a pixel shader to test the edge against the pixel boundaries and compute the appropriate bit mask. Each pixel stores only its left and top boundaries as the others can be retrieved from neighboring pixels. We use a boolean-or frame buffer operation to composite the results from multiple edges so that we can reliably detect when multiple edges cross a pixel. We draw the edges with thick lines extending their length by one pixel at each end, using an approach similar to the silhouette shadow maps [6] to ensure that we conservatively activate all pixels with potential intersections.

3.5 Image filters

The rest of the stages, pixel classification, reachability, interpolation and anti-aliasing, are implemented as a consecutive set of image filters using textures for passing intermediate results and share a similar set of implementation strategies and limitations. These stages use lookup tables encoded as fp16 textures to avoid control code in the fragment shaders, and some of the original operations in the EPI are pre-computed and stored in those textures.

At the classification stage, the pixels are classified as empty, simple and complex depending on the total number of edges' intersections that lay within. Only the simple pixels need comprehensive processing, so dynamic branching is used to reduce the number of pixels totally analyzed.

Due to the limited edge intersection and subpixel information precision some point samples are ambiguous, i.e., it cannot be established on which side of the edge that sample is. Consequently those samples are invalidated from the projected point image, using MRT to make both the classification and the invalidation at the same time, writing the classification to one texture and the valid point samples to another one.

The reachability operation uses three passes for computing immediate neighbor reachability, chaining these for reachability to farther neighbors and finally using the fact that reachability is symmetric to copy some reachability results from neighboring pixels. Then interpolation uses the reachability for edge-respecting interpolation by only using reachable samples and simultaneously calculating pixel sampling priorities. The anti-aliasing uses the interpolation and edge information from pixel classification to perform an inexpensive but effective anti-aliasing of all edge pixels.

As an optional component we also implemented the render cache interpolation, which does not use edges nor reachability and consists of a prefilter stage with an uniform 7x7 kernel to fill sam-

ple regions without points and a 3x3 weighted interpolation kernel for denser zones. We combined both the prefilter and interpolation stages into a single pass, using Shader Model 3.0 dynamic branching on the pixel shaders. The prefilter is used only where it is needed, thus improving the performance.

3.5.1 Branching granularity

The current GPUs have limited performance gains with the dynamic branching on the pixel shaders due to their SIMD nature. The current thread goes through the fast branch only if all the pixels being evaluated take the same path, otherwise all of them follow the long path. Only the simpler render cache interpolation showed substantial improvements with the branching because almost all of the pixels take the shortest path. However, other stages such as the pixel classification would require better granularity so that only a small amount of pixels follow the slowest path. This kind of complex scenes should benefit from newer hardware with a granularity of 4x4 pixels, instead of the 64x64 blocks used with most current generation cards.

3.5.2 Bit operations on the GPU

GPUs do not provide bit level operations, but we can still verify if a flag is set by comparing the numerical values. If the k bit of a number n is set, then $n \geq 2^k$.

An effective way to test this is using the step function: step(a, x) = x $\geq$ a ? 1 : 0. This function is better employed in its vector form, comparing four independent values in a single instruction. Using this step as a modulator, the weight and current total of the convolution filters is implemented with neither branching nor predication as:

```
modulator = step(2^k, n) * currWeight
total     += modulator  * currValue
weight    += modulator
```

To test if only the k bit is set, the floating point operation executed is step($2k^{-1}$, mod(n, 2^k)).

The convolution filters and other operations require to execute multiple sums and products in the style $a = c_1x_1 + c_2x_2 + c_3x_3 + c_4x_4$. An effective way to optimize this operation is performing a dot product between the elements of this linear combination: $a = \langle c_1, c_2, c_3, c_4 \rangle \cdot \langle x_1, x_2, x_3, x_4 \rangle$. The dot product of up to four elements vectors is a single hardware instruction.

3.5.3 Fill rate and texture access

The EPI filters require abundant texture reads and most of their calculations depend on those values. Therefore faster texture access speeds would benefit those stages. The EPI filters are also limited by the fill rate: the number of pixels the GPU can process. Increasing the pixel pipelines number and clock frequency would decrease the time needed for these filters.

A technique for reducing the fill rate requirements is to diminish the number of operations in the pixel shader. In all shaders the operations were vectorized whenever possible as a means to reduce the instructions count. Moreover, the number of registers used by a shader is still important for achieving maximum performance. A 180 instruction shader which uses 25 registers performs *50%* slower than other version of the same shader of 215 instructions that employs 24 registers on our NVidia 7800 GPU.

4 Results

In this section we present detailed results for our hybrid rendering system. All images are 512 x 512. Both the shaders and the image reconstruction process ran in the same system, a Pentium 4 3.2 GHz dual core with 2 GB of memory. The GPU used is a NVidia GeForce 7800 GTX with 256 MB of memory, using Forceware drivers version 81.85. The GPU code is written in C++ using OpenGL and Cg 1.4rc. The top level rendering system runs with Java 1.5 and both components are interfaced through Java Native Interface (JNI). We experimented with SLI dual GPU rendering, but its performance was not significantly faster than a single card most probably due to synchronization overheads.

We present results for five scenes. The Cornell box scene is the simplest, with one area light and 36 polygons. The Chains scene, with two lights, includes tessellated non-convex objects casting complex shadows on each other. The Mack room has three lights with 101k triangles, and includes full global illumination computed using irradiance caching. The David head from Stanford's Digital Michelangelo project, has 250k polygons, is lit by 1 light and includes ray traced glossy reflections. The Dragon scene, from Georgia Tech's Large Geometry Models Archive, is a 871k polygons model that includes a light casting a shadow through a grid on the dragon.

Table 1 presents detailed timings for these models and Table 2 compares our GPU implementation and the original render cache and EPI system (all running on the same system). Our GPU implementation is $60-110\%$ faster than the original, pure multithreaded CPU system, which uses one processor or core for generating new shading results and other one for the EPI process. More importantly, the speed up increases along with the scene complexity; on the David head and dragon scenes we achieved twice the previous frame rate. The silhouette detection is faster using the GPU and depends of the complexity of the scene, as does the edge raster stage. All the other processes are independent of scene complexity and are related only to image size; the number of elements in the point cloud depends of the image size as well.

Model	Original	Full GPU	Speed up
Cornell Box	14.46	23.53	61.46%
Chains	13.71	24.20	76.49%
Mack Room	13.08	21.87	67.49%
David Head	9.25	18.61	101.14%
Dragon	7.42	15.59	110.01%

Table 1: Comparison of the frames per second between the original system and our GPU implementation.

Discussion of point projection performance

Originally we thought that point projection, being an operation that maps straightforwardly to graphics hardware, would be very fast, and we expected that the EPI filters, with their texture accesses and multiple passes, would be the performance bottleneck. Unexpectedly the reverse is true. The graphics cards are designed to draw thousands of pixels from a few vertices and our approach of one pixel per vertex is against that premise. This along with the bandwidth limitations of VBOs makes point projection slower than expected, occupying 33% of GPU time.

Using a point cloud of roughly 300,000 points, all points could be projected in 5 ms using a static VBO. Once we updated the point cloud every frame, the time to stream all the points increased to 11 ms, regardless of the complexity of the vertex and pixel shaders used. A plausible explanation for this puzzling behavior is that because the points are updated frequently, the video driver maps the buffer to low performance memory. During our tests, we found that if the points were updated less frequently than once every four frames, they would be read faster, achieving the original rate of 5 ms. A possible reason is that the driver's memory manager moves

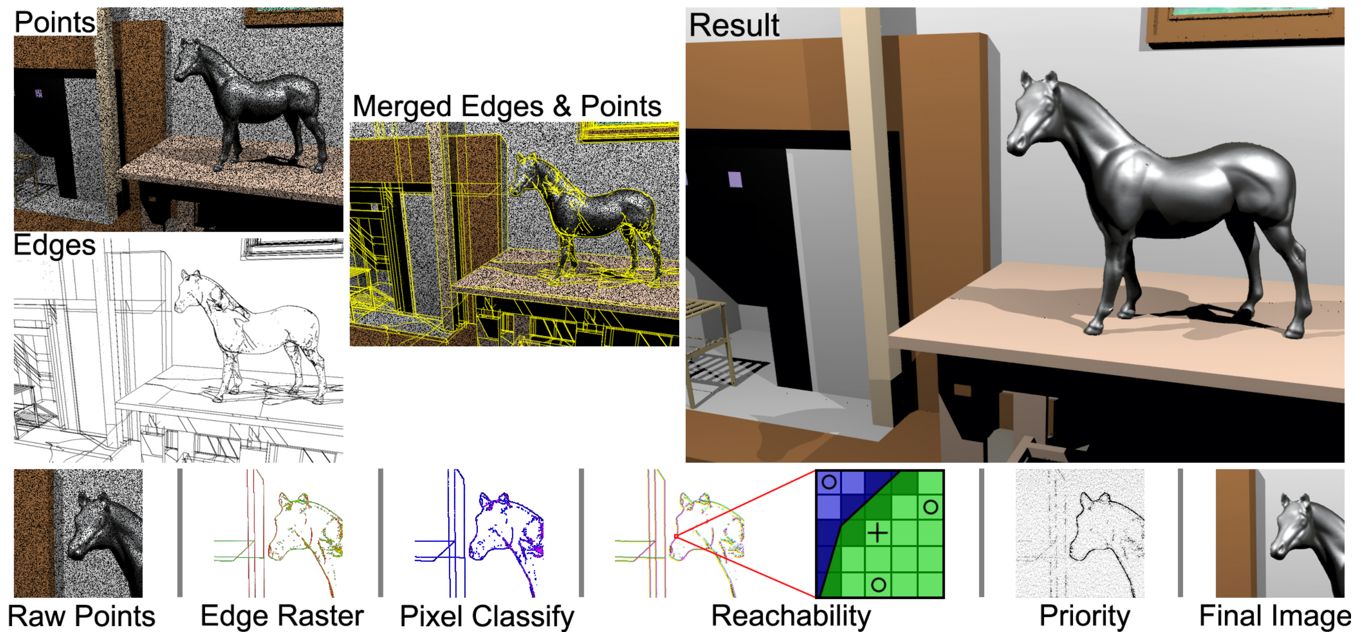

Figure 4: Illustration of the different stages of the system. The points and the edges create the edge-and-point image. Edge constrained interpolation creates a high quality approximation of the image. On the bottom there are details of data textures at different stages, from left to right: raw points as provided by the underlying renderer, edge raster stage encoding edge's crossings on the pixels, pixel classification as empty, simple or complex; reachability (showing the detail of a pixel's 5x5 neighborhood), priority depicting the places where new samples will be requested and the final result.

the buffer to the fastest memory available after it was used several times without being modified.

To implement the update of the image ID of the point cloud we evaluated the use of Vertex Texture Fetch (VTF) versus using Render To Vertex (RTV), where a read back to a pixel buffer object feeds the vertex engine. However, the VTF operation is currently only supported for some texture formats and it emerged to be $2-3$ times slower than using Render to Vertex (RTV).

Future performance

We can summarize our current performance findings:

- Demanding pixel shaders with tens of texture accesses per pixel are very fast, aside from the raw computational power of the GPUs.
- Transferring vertex data on the GPU, such as RTV, as of this writing, is disappointingly slow to be fully useful.
- Lack of scatter writes on pixel shaders prevented us for managing all data on the GPU, and thus we had to copy back some information to the CPU.

Finer granularity of the pixel pipeline branching units in the future will yield performance improvements on complex scenes for operations such as pixel classification. Scatter writes on the pixel shaders would allow us to implement the point projection and the predicted projection in a single pass, circumventing bandwidth limitations. It would also permit us to implement other stages related to point cloud management more efficiently. Better RTV implementations would allow us to keep the point cloud as a texture on the GPU. More supported texture formats for VTF, and faster performance, would make it an attractive way to update the point cloud data and project points.

Public availability

In order to provide proper implementation information of the different developed shaders, we are releasing their complete Cg source code, publicy available at:
`http://www.cs.cornell.edu/~kb/projects/epigpu`.

5 Conclusions

We have presented a hybrid GPU/CPU system for the render cache and the edge-and-point image using commodity graphics hardware. These point-based alternatives exploit the strengths of CPUs and GPUs to scale to complex scenes. Our implementation is $60-110\%$ faster than a pure CPU implementation and frees up the CPU for other computations such as expensive shading. The system's performance is likely to improve with the current trend of GPUs, which are incorporating more pixel pipelines, higher clock frequencies and more control logic for dynamic shaders. The higher frame rate coupled with additional free CPU time will achieve better interactivity and faster image convergence than the software-only system.

Acknowledgements

We thank Stanford's Digital Michelangelo project, the Georgia Tech Large Geometry Models Archive, and Gene Gregor for their models. This work was supported by NSF grants CPA-0539996, ACI-0205438, and Intel Corporation. The first author was funded by the Computer Science Department at Cornell University. He also wants to thank Francisco Valero-Cuevas from Cornell University and Eugenio García Gardea from Monterrey Institute of Technology and Higher Studies for their commitment with the Engineering Research Program under which this work was developed.

Scene	Edges	P_1	P_2	P_3	P_4	E_1	E_2	I_1	I_2	GPU	Total	FPS
Cornell Box	75	5.14	12.31	1.34	1.08	0.34	0.43	1.09	7.29	32.32	42.82	23.35
		6.68	13.87	4.73	2.01	0.08	5.63		28.70		69.14	14.46
Chains	110,592	4.02	10.77	1.30	1.07	1.54	0.99	1.44	7.17	31.81	41.32	24.20
		5.22	13.17	4.75	2.11	3.91	12.36		28.09		72.93	13.71
Mack Room	153,526	3.71	9.48	1.20	1.06	1.86	1.93	1.38	7.06	31.17	45.73	21.87
		4.73	13.27	4.59	1.97	2.17	14.77		27.97		76.43	13.08
David Head	374,653	4.37	11.92	1.18	1.05	4.92	5.87	1.15	7.11	40.78	53.73	18.61
		6.35	13.54	4.52	1.95	8.43	40.11		27.85		108.08	9.25
Dragon	1,305,164	4.15	12.79	1.18	1.06	13.81	7.88	1.25	7.11	52.22	64.16	15.59
		6.30	14.18	4.83	2.10	22.29	45.41		33.91		134.73	7.42

Table 2: Performance results. For each scene the first column gives the number of edges it has. The next eight columns give the timings in milliseconds for these processes: update points and request samples (P_1), point projection (P_2), predicted projection (P_3), depth cull (P_4), GPU silhouette detection (E_1), edge rasterization (E_2), pixel classification (I_1), reachability, interpolation and anti-aliasing (I_2). The next two columns give the total GPU time, including all overheads, and the total frame time (both CPU and GPU) in milliseconds. The last column presents the achieved frames per second. At each scene the upper row contains the GPU timings and the bottom one the CPU results. The original CPU version does not show isolated pixel classification times.

REFERENCES

[1] Kavita Bala, Bruce Walter, and Donald P. Greenberg. Combining edges and points for interactive high-quality rendering. *ACM Trans. Graph.*, 22(3):631–640, 2003.

[2] Abhinav Dayal, Cliff Woolley, Benjamin Watson, and David P. Luebke. Adaptive frameless rendering. In *Rendering Techniques*, pages 265–275, 2005.

[3] Frederic P. Pighin, Dani Lischinski, and David Salesin. Progressive previewing of ray-traced images using image-plane discontinuity meshing. In *Rendering Techniques '97*, pages 115–125, 1997.

[4] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. In *SIGGRAPH '02*, pages 703–712, 2002.

[5] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 41–50, 2003.

[6] Pradeep Sen, Mike Cammarano, and Pat Hanrahan. Shadow silhouette maps. *ACM Trans. Graph.*, 22(3):521–526, 2003.

[7] M. Simmons and C. Séquin. Tapestry: A dynamic mesh-based display representation for interactive rendering, 2000.

[8] Miloslaw Smky, Shin-ichi Kinuwaki, Roman Durikovic, and Karol Myszkowski. Temporally coherent irradiance caching for high quality animation rendering. In *Eurographics 2005*, volume 24, 2005.

[9] Marc Stamminger, Joerg Haber, Hartmut Schirmacher, and Hans-Peter Seidel. Walkthroughs with corrective texturing. In *Rendering Techniques 2000*, pages 377–388, 2000.

[10] Laszlo Szirmay-Kalos, Barnabas Aszodi, Istvan Lazanyi, and Matyas Premecz. Approximate ray-tracing on the gpu with distance impostors. In *Eurographics 2005*, volume 24, 2005.

[11] Parag Tole, Fabio Pellacini, Bruce Walter, and Donald P. Greenberg. Interactive global illumination in dynamic scenes. In *SIGGRAPH '02*, pages 537–546, 2002.

[12] Bruce Walter, George Drettakis, and Donald P. Greenberg. Enhancing and optimizing the render cache. In *Eurographics Workshop on Rendering*, 2002.

[13] Bruce Walter, George Drettakis, and Steven Parker. Interactive rendering using the render cache. In *Rendering techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering)*, volume 10, pages 235–246, Jun 1999.