GPU Production Rendering

Larry Gritz
NVIDIA

April 22, 2005

Chapter 9

GPU Production Rendering

Larry Gritz
NVIDIA
lgritz@nvidia.com

9.1 Introduction

This chapter is not about realtime rendering. It's about a world in which rendering takes seconds, minutes, or hours. It's about users who require features that are extremely difficult to implement on modern graphics hardware. But we'd still like to take advantage of the tremendous horsepower that GPUs afford us.

First, we'll discuss the arcane world of production rendering for film (and other non-game high-end uses) — what people need from a film renderer, and how this is different from a game engine.

Next, we'll dive into the gory details about how GPUs were used for certain features in NVIDIA's Gelato™ renderer. There are some clever techniques here, but it's also an object lesson about how insanely tricky it is to get GPUs to render in ways they weren't designed for, though occasionally if you do it right, you can get fantastic performance benefits.

NVIDIA's Gelato renderer is an offline (batch-oriented, non-realtime) renderer designed to render absurdly complex scenes at high enough quality to be used in feature film visual effects and animation. Unique among similarly capable renderers (as of this writing), it *requires* a GPU, which it uses to accelerate various of its algorithms. But it is not using graphics hardware for previews or reduced-quality rendering — it uses the GPU to help generate the highest quality final pixels. Indeed, part of the requirement of the product is that we use GPU where we can, but never allow hardware usage to sacrifice image quality, flexibility, or feature set.

The bottom line is that we've spent the past few years learning how to use GPUs to make very high-quality imagery, often in ways quite different from how a game or any other realtime application would use the graphics hardware.

For these notes, we'll concentrate on just a small subset of Gelato's functionality: hidden

surface removal. That is, the process of taking diced, displaced, shaded geometry and creating a final antialiased, motion-blurred image. This may seem straightforward, but a good film renderer requires many features not supported by current commodity graphics hardware: wide smooth filters, high sampling rates, order-independent transparency, spectral opacity (that is, separate alphas for R, G, and B), motion blur, and depth of field. We present a GPU-based hidden-surface algorithm that implements all these features.

These notes won't go into any detail about other aspects of Gelato's architecture, including handling the high-order primitives (such as NURBS or subdivision surfaces), or how the actual shading occurs (including texturing, displacement, and global illumination), or memory management, or about scene or image input and output. There's just too much material. It's more instructive to cover just one major subsystem, but go into a good deal of depth about how we use the GPU. This is not meant to imply that other subsystems don't, or couldn't, or shouldn't, use the GPU. In many cases, they do, could, or should. In many other cases, they do not.

9.2 Production Rendering

Gelato is a "no compromises, production-quality" renderer. We wanted our renderer to have the properties that make it ideally suited, and indeed actually used, for rendering final frames for motion picture production. Visual effects and feature-length animation have historically been the most demanding users of such products, so success in that domain is very important to us.

Just what is a "no compromises" renderer? What does it take to render for film? What does "production quality" really mean? How does this differ from a game engine, realtime graphics, or other types of rendering?

9.2.1 How Film Rendering is Different

It's important to note critical differences (and similarities) between film and games (and other realtime graphics applications).

Both games and films are displayed as a series of *frames*, or individual images. Motion picture film displays 24 frames per second, broadcast is typically 25 fps (PAL) or 29.97 (NTSC). Games may have variable frame rates, but typically aim for 30-60 fps. For both film and games (these days), a final frame that a user/viewer sees is probably a composite of many passes, layers, or subimages. Very rarely is a final frame the result of a single rendered image pass.

Both games and films are organized into groups of temporally contiguous frames, rendered consecutively, with continuous (in the mathematical sense) camera positions, and with a common set of art assets used in the group of frames. For games, these sets of similar frames are called *levels*, and may last for many minutes of game play. For films, such sets are called *shots*, and typically last for only a few seconds of screen time.

Games, therefore, amortize overhead such as loading textures over tens of thousands of frames, and usually perform such tasks before the first frame is drawn (thus, in most people's minds, "not counting" in the computation of per-frame rendering speed). In contrast, most film rendering is done a single frame at a time. But even if entire shots were rendered all at once to maximize coherence (and there are reasons to believe this would be prohibitively difficult),

overhead could only be amortized over a few tens or possibly hundreds of frames comprising a shot.

On the basis of texture requirements alone, we can see why film rendering is unlikely to ever happen in real time. Film frames require many gigabytes of texture (versus perhaps tens of MBs for games). Just the time to read the required texture for a frame from disk or network could take up to several minutes. Similar costs apply to reading, culling, sorting, and dicing the several GB of scene geometry.

No matter how fast the graphics, no matter how much is performed with the GPU instead of the CPU, the data manipulation just to *prepare* to render a film frame will still preclude realtime rendering. This shouldn't be surprising – games wouldn't be realtime, either, if we had to load new game levels every 2-10 seconds of play (let alone every frame). The lower limit of film final frame rendering speed is dictated by disk transfer and other factors, not, ultimately, by transformation, fill, or even shading speed.

In addition to the apparent impossibility of rendering film frames in realtime, we need to contend with *Blinn's Law*. The modern rephrasing of his observation is that at any studio, frame rendering times are constant despite hardware or algorithmic improvements. Frames take 30 minutes, or an hour, or 6 hours (depending on the studio) — exactly as long, or possibly longer than they took at the same studio 5, 10, or 15 years ago. Studios always respond to increased rendering resources by constructing more complex scenes with more geometry, more lights, more complex shaders, more expensive realistic lighting, but almost NEVER by simply taking advantage of the speedup. This is the obvious choice for them, since the final images are precomputed and delivered on film. Thus, more time spent creating better images only improves the audience's experience (whereas spending more time rendering detracts from a game player's experience). Therefore, it would behoove us to design primarily to gracefully handle massive scene complexity at the highest quality level, rather than architecting for realtime performance.

The exception to these trends is that there is one special case where production needs interactive speeds for many successive similar renderings. That case is interactive lighting, where the geometry, camera, and shaders are fixed, and only the parameters to light sources change from frame to frame, and we would like to see the results of such changes interactively (possibly acceptable at a somewhat reduced quality level). Indeed, the ability to do such interactive rerendering should be considered an essential design goal of a film renderer.

Another difference between games and film is related to their development and product cycles. A game will be played by millions of people, and for every invocation tens or hundreds of thousands of frames may be rendered. Image quality, code reusability and clarity, and artist/TD time will always be happily traded for guaranteed frame rates. Images for film, on the other hand, are only rendered once at full resolution. The amount of machine time spent rendering an object is often dwarfed by the amount of TD time it takes to create the model, write the shader, or light the shot, and the human time is the more expensive resource anyway. This observation has many implications for API development, and explains why API's and languages appropriate for realtime (OpenGL, Cg, etc.) may not be appropriate for offline rendering, and vice versa.

9.2.2 Essential Properties of Film Production Rendering

Let's take an inventory of essential features and charasteristics of a renderer that is intended to create high-end imagery for film. It probably will not escape your notice that nearly every one of these categories contains a requirement that is at best inconvenient — and frequently impossible — on graphics hardware (at least in its "native" mode).

Geometric Primitives

Games, realtime graphics, graphics hardware, and even "low-end professional" packages rely on polygons, but the heavy lifting for film work is almost always based on bicubic patches, NURBS (often trimmed), and, increasingly, subdivision surfaces. Points (for particles) and curves (for hair) are also important, and must be rendered efficiently by the million. Furthermore, all curved primitives must dice adaptively based on screen area and curvature, so as to NEVER show tessellation artifacts.

Geometric Complexity

Geometric input can easily be several gigabytes. As a primary design goal, a film renderer must be able to handle more geometry than could fit into RAM at once. Strategies for doing this often include bucketing, sorting, aggressive culling, occlusion testing, procedural geometry, and caching to disk.

Texture complexity and quality

It's not unusual for a film frame to require hundreds to thousands of textures totalling tens of Gigabytes of storage. Successful film renderers use a caching scheme where individual coherent tiles are paged from disk as needed. Not only is it impractical to read the textures into memory before rendering, but in many cases even the filenames of the textures are not known until in the middle of the shader requesting it.[1] All texture map lookups must have the highest fidelity. Texture, environment, and shadow maps must blur (as requested by the shader) without artifacts, seams, or noise. In many cases, lookups must have a better filter than trilinear mipmap.

Motion blur

Motion blur is essential and must be of extremely high quality (no "strobing" and little noise). Production renderers are required to have both transformation blur (changing position, orientation, and scale) as well as geometric deformation blur. To be really competitive, they must support multisegment motion blur (arbitrary number of knots, rather than merely linear blur). Depth of field effects are also important, though not as much so as motion blur. Ideally, a moderate amount of motion blur should add almost no expense to the rendering of a scene.

[1] In advanced renderers such as Gelato or RenderMan, texture file names can be, and often are, generated dynamically by the shader and looked up immediately.

Antialiasing

Simply put, NO amount of visible aliasing artifacts are acceptable. This is usually achieved by taking many point samples per pixel (64-100 samples per pixel is common for scenes with hair or fur). and the subpixel data must be reconstructed into pixels with a high-quality, user-specified filter having a width larger than one pixel.

Image size, depth, format, data

Film requires arbitrary image resolutions (2k - 4k for final frames, 8k or higher are not uncommon for shadow maps), flexible bit depths (8, 16, half, float), useful image formats (at least TIFF, increasingly OpenEXR, and preferably with an easy API for users to supply their own image writing routines). It's also critical that a renderer be able to output not just color, alpha, and depth, but any data computed by the shaders (in film, this is called "arbitrary output variables" (AOVs), though in the realtime graphics world it is often called "multiple render targets," or MRTs). Frames must be saved to disk; rendering directly to the frame buffer is not important for film.

No limits

Arbitrary limits are not tolerable. This includes number of textures, total size of textures, size of geometric database, resolution of images or textures, number of output channels, number of lights, number of shaders, number of parameters to shaders.

Global Illumination

The days of strictly local illumination are over. Modern film renderers simply must support ray-traced reflections and shadows, indirect light transport, caustics, environment lighting, "ambient occlusion," and subsurface scattering.

Displacement

Adaptively diced, subpixel-frequency displacements are a required feature, and should ideally be no more expensive than undisplaced geometry.

Flexible Programmable Shading

It used to be enough to simply say that production-quality rendering needed programmable shading (which was impossible in hardware and rarely done in software). Now that it's more common – even in hardware – it is helpful to enumerate some of the specific features that distinguish programmable shading for film work from that suitable (or at least accepted) for realtime games:

- Rich data types, such as vector, color, point, normal, matrix, strings, and arbitrary-length arrays of any these.

- String manipulation, and the ability to refer to textures, coordinate systems, and external data by name, rather than by handles or explicit passing of matrices as parameters. It's also really handy to be able to printf/fprintf from a shader.
- Hiding of hardware details and limits. Shader writers should not need to know about reduced precision data types, memory or instruction limits, names or details of hardware registers, etc. There should never be artifacts or loss of precision related to any such limits that actually exist in the hardware.
- The ability to call out to user-supplied code on the host (often known as "DSO shadeops").
- Data-dependent looping, especially for applications such as ray marching for volumetric effects, or numerical integration inside shaders.
- Separate compilation of different types of shaders (especially light shaders).
- High-quality calculation of derivatives and antialiasing. This means that derivatives must use central differencing and extrapolation at grid edges. Simple forward differencing (especially if it misses edges, as ddx/ddy on NV3x do) leads to unacceptable artifacts and poor estimates of filter sizes for texture lookups.
- Arbitrary name, data type, and number of parameter arguments into shaders and as outputs of shaders.

Appropriate API's and formats

A "film renderer" needs to fit into a film production pipeline. That means, at a minimum:

- A clean procedural (C or C++) API. OpenGL and DirectX both expose too many hardware details, are oriented to polygons with no real support for higher-order surfaces, have poor or awkward support for shader assignment and binding, and are cluttered with huge sections of the API dedicated to interactive functionality that we don't want or need.
- A scene archive format for "baking geometry." For RenderMan, this was RIB; there is no equivalent metafile for OpenGL or DirectX. Gelato unifies a scene archive format with the means of writing procedural primitives, by using the Python scripting language. It is also possible to write Gelato plugins that directly read any scene format, without needing to translate into an arbitrary intermediate format.
- Procedural geometry – archive files read, geometry-producing programs run, or API-calling DSO's executed lazily as demanded by the renderer – as a first-class concept.
- A shading language. Cg or GLSL are not the right ones, largely for the reasons stated above. Gelato has its own shading language (GSL) to satisfy that need.
- Minor API's: DSO shadeops, image file format readers and writers, querying shader parameters, etc.

- Appropriate Platform: Most of the big studios use Linux. Many smaller houses use Windows. We are also carefully watching Mac and OS X and will consider it for future support if the film industry demands it, but so far the demand in major studios is small or nonexistant. But the bottom line is that Windows-only drivers, support, APIs, or tools will simply not cut it in the studios.

9.3 Hidden Surface Removal in Gelato

9.3.1 Goals

To allow Gelato to render for film and other high-end applications, we require hidden surface removal features that have not traditionally been available in graphics hardware: highly supersampled antialiasing, filtered with a high-quality kernel with support wider than one pixel; motion blur and depth of field; order-independent transparency with spectral opacities (separate values for red, green, and blue); and floating-point output of color, opacity, and any other arbitrary values computed by shaders.

Figure 9.1: A motion blurred image rendered with Gelato (courtesy of Tweak Films).

In this section, we present a high-quality hidden surface removal algorithm that is accelerated by modern commodity graphics hardware. Specifically:

- The algorithm incorporates supersampling, user-selected filters with arbitrarily wide support, depth of field, and multiple output channels.
- The algorithm supports transparency. It uses the depth peeling technique, enhanced to allow multi-channel opacity and opacity thresholding. We also present optimizations that, for typical scenes, allow transparency performance to scale linearly rather than as $O(N^2)$.

- The algorithm uses grid occlusion culling to avoid additional shading.
- The algorithm produces comparable quality and superior performance to the hidden surface removal algorithms used in CPU-only software renderers, for many typical scenes (see Figure 9.1).
- We explore a variety of engineering trade-offs in implementing this algorithm on modern GPUs, and discuss its performance on a variety of scenes.

This work does not attempt to create real-time rendering capabilities (e.g., 30 fps). Rather, our goal is to accelerate the rendering of images at the highest quality possible, suitable for film or broadcast, that now take minutes or hours. We have already achieved significant speed improvements by using graphics hardware, and we expect those improvements to become even greater over time as graphics hardware becomes more capable and its speed improvements continue to outpace those of CPUs.

Previous attempts at high-quality rendering with graphics hardware have generally involved costly custom hardware whose performance is often soon overtaken by improvements in general purpose CPUs. In contrast, this work is concerned strictly with commodity graphics hardware. We find this approach promising because the economy of scale present with modern commodity hardware makes it very inexpensive, and because graphics hardware increases in capability at a much faster rate than CPUs are improving. For several years graphics hardware has doubled in speed every six to twelve months versus every eighteen months for CPUs. In addition to traditional geometric transformation and rasterization, modern GPUs feature (1) significant programmability via vertex and fragment shaders, rapidly converging on the capabilities of stream processors; and (2) floating-point precision through most or all of the pipeline. These two facilities are the key enablers of the work described in this paper.

Related Work

Prior work can be categorized according to the hardware it requires. Software-only systems for high-quality rendering, particularly the Reyes architecture (Cook et al., 1987; Apodaca and Gritz, 1999) and its commercial implementations, form both the jumping-off point for our approach and the baseline against which we measure our results. Systems using specialized hardware for high-quality rendering have been proposed or built, but are outside the scope of this paper. This leaves GPU and GPU-assisted methods.

Current GPUs support *multisampling*, which computes visibility for multiple samples per pixel but reuses a single color from the center of the pixel (Möller and Haines, 2002). This improves real-time graphics but is not high-quality: current hardware limits the filter shape to a box with no overlap between adjacent pixels, full floating-point formats are not yet supported, and only low, fixed numbers of samples per pixel are supported (currently 4 to 8, versus the dozens typically used for film rendering). The proposed Talisman architecture (Torborg and Kajiya, 1996) mitigates the memory expense of multiple samples by rendering one screen region at a time, and also supports multisampling, though at a fixed 4×4 samples per pixel.

The *accumulation buffer* (Haeberli and Akeley, 1990) supports antialiasing, motion blur, and depth of field effects by accumulating weighted sums of several passes, where each pass

is a complete re-rendering of the scene at a different subpixel offset, time, and lens position, respectively. Direct hardware support has not yet included full 32-bit floating point precision, but programmable GPUs can use fragment shaders to apply this technique at floating point precision, although still at the expense of sending geometry to the GPU once per spatial filter sample. In contrast, supersampling avoids this expense, and has another advantage on the GPU (discussed in Section 9.3.3): each micropolygon covers more pixels. Interleaved sampling (Keller and Heidrich, 2001) can reduce the artifacts from regularly-sampled accumulation buffers.

Scenes with partially transparent surfaces, possibly mixed with opaque surfaces, are challenging. Methods that require sorting surfaces in depth are problematic because intersecting surfaces may have to be split to resolve depth ordering. The *depth peeling* method, described in general by (Mammen, 1989) and for GPUs by Everitt (Everitt, 2001), solves *order-independent transparency* on the GPU. The method is summarized later in this paper. Its principal drawback is that a scene with maximum depth complexity D must be sent to the GPU D times. For some scenes, such as particle systems for smoke, this can result in $O(N^2)$ rendering time, where N is the number of primitives. Mammen suggests an *occlusion culling* optimization (though not by that name). In his algorithm, once an object no longer affects the current pass, it is dropped for subsequent passes. Kelley et al. (Kelley et al., 1994) designed hardware with four z-buffers, able to depth peel four layers at once. They use screen regions so that only regions with high D require high numbers of passes. The hardware *R-Buffer* (Wittenbrink, 2001) proposes to recirculate transparent pixels rather than transparent surfaces; this avoids sending and rasterizing the geometry multiple times, but can require vast memory for deep scenes.

9.3.2 Architectural Overview

Gelato has a Reyes-like architecture (Cook et al., 1987; Apodaca and Gritz, 1999). High-order surface primitives such as NURBS, subdivision surfaces, and polygons are recursively split until small enough to shade all at once. These patches are diced into *grids* of pixel-sized quadrilateral micropolygons. Grids typically contain on the order of 100 micropolygons, and are limited to a user-selected size. The grids are possibly displaced, then assigned color, opacity, and optionally other data values at each grid vertex by execution of user-programmable shaders. Finally, the shaded grids are "hidden" to form a 2D image.

It is this final stage—hidden-surface removal of the shaded grids of approximately pixel-sized quadrilaterals—that is the concern of this paper. Whether performed on the GPU or CPU, the earlier stages of handling high-order geometry and the shading to assign colors and opacities to the quadrilateral vertices are largely orthogonal to the methods used for hidden surface removal. Although beyond the scope of this paper, our shading system implements advanced techniques including global illumination and ray tracing.

Like many Reyes-style renderers, we divide image space into rectangular subimages called *buckets* (Cook et al., 1987) in order to reduce the working set (of both geometry and pixels) so that scenes of massive complexity can be handled using reasonable amounts of memory. For each bucket, the grids overlapping that bucket are rasterized using OpenGL, as described in Section 9.3.3. To handle partially-transparent grids, we use depth peeling but extend it in several important ways to allow multi-channel opacity and opacity thresholding. We also introduce a

batching scheme that reduces the computational complexity of depth peeling for typical cases. The transparency algorithms are discussed in detail in Section 9.3.4.

Motion blur and depth of field are achieved using an accumulation-buffer-like technique involving multiple rendering passes through the geometry for the bucket. This is described in Section 9.3.5.

In order to achieve sufficiently high supersampling to ameliorate visible aliasing, buckets are rendered for all of these passes at substantially oversampled resolution, then filtered and downsampled to form the final image tiles. The filtering and downsampling is performed entirely on the graphics card using fragment programs, as described in detail in Section 9.3.6.

The handling of transparency, motion blur and depth of field, and arbitrary output channels can lead to large numbers of rendering passes for each bucket (though of course much of that work must be performed only for buckets that contain transparent or moving geometry). Specifically,

$$passes = P_{output} \times P_{motion} \times P_{transparent}$$

where P_{output} is the number of output images; P_{motion} is the number of motion blur or depth of field samples; and $P_{transparent}$ is the number of passes necessary to resolve transparency.

GPUs are well optimized to rasterize huge amounts of geometry very rapidly. Large numbers of passes render quite quickly, generally much faster than using CPU-only algorithms. In short, brute force usually wins. We will return to discussion of performance characteristics and other results in Section 9.3.7.

9.3.3 Hiding Opaque Surfaces

This section describes our basic hidden-surface algorithm for opaque geometry. For simplicity, we describe the process for a single bucket in isolation; extension to multiple buckets is straightforward. The steps are repeated for each output image[2].

The algorithm has the overall splitting-and-dicing form of a Reyes-style algorithm, but with a new grid occlusion test towards the end:

```
for each output image:
  for every object from front to back:
    if object's bounding box passes the occlusion test:
      if object is too big to dice:
        split object and re-insert split sub-objects
      else:
        dice object into grid of pixel-sized quads
        if diced grid passes the occlusion test:
          shade the grid
          render the grid
```

[2]Users' shaders can optionally output other variables besides color. Each is rendered into a separate *output image*. Examples include surface normals or separate specular and diffuse images.

Grid Rendering

To render diced grids with high quality on the GPU, we use *regular supersampling* — in other words, we render large, and minify later. We render the grids using standard OpenGL primitives into an image buffer which is larger than the final pixel resolution by a user-selected factor, typically 4-16x in each dimension. For a typical bucket size of 32×32 pixels, the supersampled buffer fits easily in GPU memory. It is a happy synergy that buckets, primarily designed to limit in-core scene complexity, also help to limit buffer size. Rendering grids large also helps tune the geometry for the GPU. The projected screen size of a grid micropolygon is usually on the order of a single pixel in the final image. However, current GPUs are designed to be most efficient at rendering polygons that cover tens or hundreds of pixels; smaller polygons tend to reduce the parallel computation efficiency of the hardware. Supersampling helps to move our grid polygons into this "sweet spot" of GPU performance.

Shading data can be computed at the resolution of the grid—the *dicing rate*—or at another, higher resolution. If the shading and dicing rates match, then the shaded colors are passed down to the GPU as per-vertex data. A full-precision register interpolates the colors across polygons, resulting in smooth-shaded grid.

If the shading rate differs from the dicing rate, so that shading data is no longer per-vertex, then colors are passed to the GPU as a floating-point texture instead. Sending the texture to the GPU and then sampling it for each fragment is slower than passing a color per vertex. In our implementation, the penalty was 10–20%, and quality suffered because the hardware we used could not efficiently filter floating-point texture lookups.

Occlusion Culling

In modern Reyes-style algorithms, before an object is split, diced, or shaded, its bounding box is tested to see whether it is completely occluded by objects already rendered, and if so, it is culled. To maximize culling (that is, to reduce shading of objects that will turn out to be occluded), objects are processed in roughly front-to-back order, using a heap data structure ordered by the near z values of the objects' camera space bounding boxes. In a high-quality renderer, shading tends to be more expensive than occlusion testing, since it may include texture lookups, shadows, or even global illumination. So occlusion culling before shading can often produce great performance benefits, typically an order of magnitude or more.

CPU-based renderers typically maintain a hierarchical z-buffer for occlusion testing (Greene and Kass, 1993). By contrast, our renderer uses the GPU hardware to occlusion test against the full z-buffer, via the OpenGL *occlusion-query* operations (Segal and Akeley, 2003). Occlusion-query returns the number of fragments that passed the z-buffer test between the query start and end calls. In other words, it returns the number of visible fragments. We turn off writes to the framebuffer, begin an occlusion-query, render the bounding box, then end the query. If the query reports zero visible fragments, the object is culled.

The bounding box occlusion cull before objects are diced is similar to CPU-based algorithms. But later, after the object is diced (but before it is shaded), we occlusion cull again using the actual grid geometry, an exact test for whether rendering the grid would change any pixels. This test would be quite expensive on the CPU, but GPUs rasterize quickly. Grid culling

is especially effective near silhouette edges, where grids behind the silhouette very often have a bounding box that spills out past the silhouette. Grid culling can also help when an object's bounding box pokes out from behind other geometry, but the actual object does not (e.g., two concentric spheres with slightly different radii). Grid culling reduces shading by 10-20% in our test scenes and provides a significant performance boost. In scenes dominated by shading time such as ambient occlusion, ray tracing, and complex user-defined shaders, the performance boost is even more pronounced.

The PC bus architectures in current use allow much greater data bandwidth to the GPU than back from it. Occlusion-query fits this mold well, since it returns only a single integer. However, the GPU must finish rendering the primitives before the occlusion-query result is available. Thus the frequent queries in our hider algorithm tend to cause the GPU to run at reduced efficiency, although some latency can be hidden by careful ordering of operations.

9.3.4 Transparency

As mentioned earlier, we use depth peeling to render grids that may be transparent and may overlap in depth and screen space. Standard depth peeling is a multipass method that renders the nearest transparent surface at each pixel in the first pass, the second nearest surface in the second pass, etc. Each pass is composited into an RGBAZ "layers so far" buffer that accumulates the transparent layers already rendered. Each pass renders the whole scene using ordinary z-buffering, with an additional test that only accepts fragments that are behind the corresponding z in the layers-so-far buffer. An occlusion query test on each pass indicates whether that layer was empty; if it was, the algorithm halts.

We considered an alternative to depth peeling, an algorithm to "bust and sort" overlapping grids into individual micropolygons, then render them all in front-to-back order. This approach was abandoned because current GPUs do not support floating-point alpha blending or allow framebuffer reads from within fragment programs. This might be worth revisiting if future GPUs add these capabilities.

Our depth peeling (Listing 1) extends that of (Everitt, 2001) in several ways. First, like (Mammen, 1989), we process all opaque surfaces beforehand into a separate RGBAZ buffer, leaving only the transparent grids for depth peeling. Second, we perform opacity thresholding between batches of transparent surfaces sorted in depth, resulting in greatly improved performance for typical cases. Third, we handle spectral opacity, using three passes when needed.

Opaque Preprocessing

Because most scenes are mostly opaque, rendering opaque surfaces first drastically cuts the number of required depth peeling passes. The remaining transparent grids are occlusion culled against the opaque z-buffer, further reducing depth peeling effort. We use the opaque z-buffer texture as an additional depth comparison during depth peeling, so transparent micropolygons occluded by opaque surfaces do not add additional depth peeling passes.

Figure 9.2 illustrates the algorithm. The opaque preprocess renders all opaque surfaces into the opaque z-buffer. From here on, we render only transparent surfaces. Grids occluded by the opaque z-buffer are culled. Pass 1 computes RGBAZ of the nearest transparent surfaces

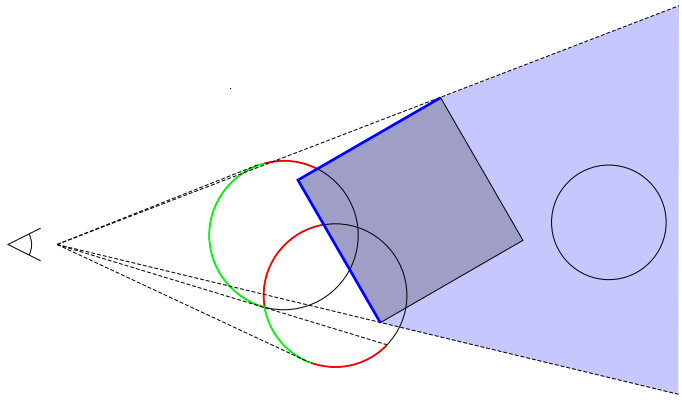

Figure 9.2: Depth peeling of three transparent spheres and an opaque cube, after one layer has been peeled. Green indicates the layers-so-far buffer (nearest layer) while red indicates the layer being computed (second-nearest layer). The blue line on the surface of the cube indicates the opaque-surfaces buffer, which culls away all surfaces behind it.

(green); this initializes the layers-so-far buffer. Pass p, $p > 1$, computes RGBAZ of the p-th nearest transparent surfaces. The fragment program for Pass p rejects fragments unless they are both behind the layer-so-far buffer's z and in front of the opaque buffer's z; the z-buffer test for Pass p selects the nearest of the accepted fragments. We halt when the occlusion query test for Pass p reports that no fragments were accepted. Otherwise, we merge the RGBA of Pass p under the layers-so-far buffer, and replace the z of the layers-so-far buffer with the z of Pass p.

Z-Batches

Depth peeling has $O(N^2)$ worst-case performance for N grids. If we could depth peel the grids in batches of B grids each, the worst case would be only $O((N/B)B^2) = O(BN) = O(N)$. The problem is that the grids from separate batches may overlap in depth, so we cannot simply depth peel the batches independently and composite the results. We solve this problem by restricting each batch to a specific range of z values, partitioning the z axis so that standard compositing yields correct results.

We do this with constant-z clipping planes. Recall that we have sorted our grids by the z value of the front plane of the camera-space bounding box. We break the list of transparent grids into *z-batches* of B consecutive grids each. The $z_{\min}$ of each z-batch is the $z_{\min}$ of its first primitive. While depth peeling a z-batch, we clip away fragments nearer than the batch's $z_{\min}$ or farther than (or equal to) the next batch's $z_{\min}$; see Figure 9.3. Grids that cross a $z_{\min}$ boundary are rendered with both batches. Each z-batch now produces the correct image for a non-overlapping range of z values, and simple compositing of these images now works.

Grids that cross multiple $z_{\min}$ planes appear in multiple z-batches. In the worst case, grids overlap so extensively in depth that z-batch size effectively approaches N, and we still do $O(N^2)$ work. More typically, batches are bigger than B but still much smaller than N, and the speed-up is enormous.

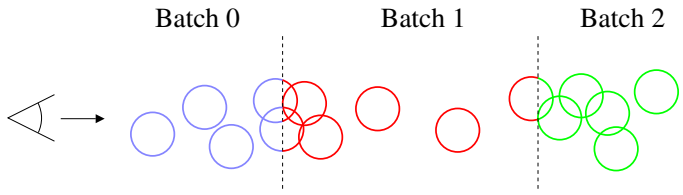

Figure 9.3: Clipping planes in z let us depth-peel each z-batch independently. Here we see three z-batches, with $B = 5$. Assume each sphere is one grid. Note that grids that "trail" into a following batch are rendered in both batches, with z-clipping at the batch boundary.

Opacity Thresholding

Due to the nature of the Porter-Duff *over* operation (Porter and Duff, 1984) used to accumulate transparent layers, the opacity value will approach but never equal full opacity. Opacity thresholding (Levoy, 1990) approximates opacity values above a user-specified threshold as fully opaque, thus reducing the effective depth complexity. This meshes nicely with z-batches. After each z-batch is rendered, we identify these pseudo-opaque pixels and merge their z values into the opaque buffer. Before grids are added to a z-batch, they are occlusion culled against this opaque buffer.

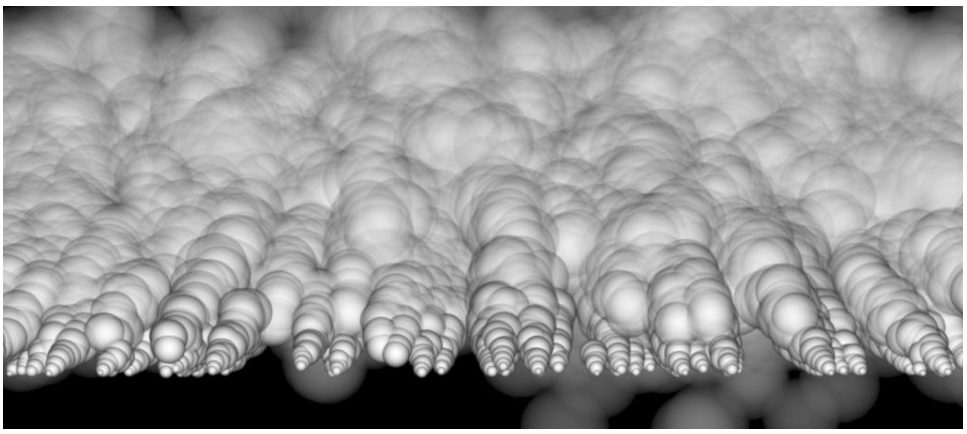

Figure 9.4: A particle system with sixteen thousand transparent overlapping spheres. Courtesy of Tweak Films.

Using z-batches and opacity thresholding can improve performance by orders of magnitude. Figure 9.4 rendered in just under 4 hours without using either technique; in 13 minutes with z-batching ($B = 26$) but no opacity thresholding; in 3 min. 38 sec. with $B = 26$ and an opacity threshold of 0.95. The stochastic CPU algorithm (as implemented in PRMan) renders the scene in 5 min. 15 sec. with the same opacity threshold.

Thresholding results are inexact because the threshold test is performed after an *entire* z-batch is processed and not as individual fragments are processed. The first grid to push a pixel over the threshold may be in the middle of a batch, so the pixel may still accumulate a few more

layers. This artifact is easy to see when we lower the opacity threshold to an unreasonably low value such as 0.2, especially since batching can change at bucket boundaries (see Figure 9.5). However, it is nearly invisible at ordinary opacity thresholds (e.g., 0.95).

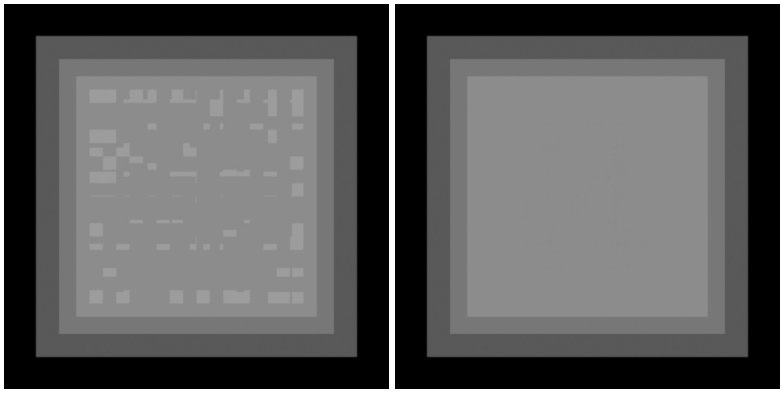

Figure 9.5: Exaggerated artifacts of opacity thresholding by batches. A stack of camera-facing planes, each with opacity 0.1, with the opacity threshold is set to the unreasonably low value of 0.2. Image on right shows the ideal solution.

If future GPUs were to extend the occlusion query operation to return the pixel bounding box and the z range of the rendered fragments that passed the occlusion test, we could optimize the thresholding pass by only running it on the bounding rectangle from the previous depth peeling pass. This would allow us to run the test after each primitive instead of per z-batch, which would reduce error and improve culling efficiency. Occlusion query is the only reduction operator currently available in GPU hardware; enhancements could be useful for many general computations.

Spectral Opacity

During geometry processing, we detect whether any primitive contains spectral opacity, as opposed to monochromatic opacity. Hardware supporting *multiple draw buffers* (ATI, 2003) can render spectral opacity in a single pass. Otherwise, three passes of the whole transparency algorithm are required, one each for red, green, and blue.

9.3.5 Motion Blur and Depth of Field

Motion blur and depth of field are computed by rendering multiple passes into a supersampled accumulation buffer stored on the GPU, with the number of time and/or lens samples a user-specified parameter. Each pass computes a single instant in time and lens position for all pixels. This contrasts with the approach of using a different time and lens position for each pixel or sample (Cook et al., 1984; Cook, 1986).

The passes can be accumulated either by rendering into a single buffer multiple times, or by rendering into multiple buffers simultaneously (Listing 2). The former reduces memory usage at the expense of repeated traversals of the geometry, while the latter increases memory usage for the buffers, but minimizes geometry traversals and any per-primitive computations and state

Listing 1 Pseudo-code for the transparency rendering loop including depth peeling, z-batch management and opacity thresholding.

```
for every object from front to back:
  if object's bounding box passes the occlusion test:
    if object is splittable:
      split object and re-insert split sub-objects
    else:
      dice object into grid
      if diced grid passes the occlusion test:
        shade grid
        if grid is transparent:
          append grid to current z-batch
          if current z-batch size exceeds threshold:
            save opaque RGBAZ
            while not finished rendering all transparent layers:
              for every transparent grid in z-batch:
                render transparent grid
              composite transparent layer under layers so far
            store accumulated transparent RGBA
            restore opaque RGBAZ
        else:
          render opaque grid

if transparent grids were rendered:
  composite accumulated transparent layers over opaque layer
```

changes. With either method, the same geometry will be rendered the same number of times by OpenGL. Accumulation is always on the GPU, to avoid readback.

Listing 2 Pseudo-code for the accumulation algorithms.

Multibuffer accumulation:

```
for every primitive P:
  for every time sample t from 0 to T-1:
    render P(t) into buffer[t]
for every time sample t from 0 to T-1:
  accumulate buffer[t] into final image
```

Multipass accumulation:

```
for every time sample t from 0 to T-1:
  for every primitive P:
    render P(t) into buffer
  accumulate buffer into final image
```

If the time-sample buffers exceed the available video memory, performance drops radically, as buffers are swapped back and forth to CPU memory. Therefore we prefer to use multiple passes.

Rather than transfering grid data to the video memory for each pass, we cache them in *vertex buffer objects* (VBOs) (SGI, 2003). This can double motion blur performance. We have not seen swapping issues with VBOs, perhaps because we use them only for grids above a certain size, and one bucket will generally have a finite number of these visible.

Occlusion-culled grids provide another opportunity for performance improvement over CPU culling. Motion-blurred bounding boxes are particularly inefficient for occlusion culling, whereas grid culling at each time sample is easily implemented during standard motion blur rendering.

Vertex Motion

Each pass renders the geometry for a specific time in the shutter interval, and in the case of depth of field, a specific position on the lens. We use a GPU vertex program that performs the motion blur interpolation, the model-to-view transformation, and the lens position offset (based on depth; see formula in (Potmesil and Chakravarty, 1981)).

Sampling shutter times and lens positions simultaneously for the whole bucket, as opposed to having the time and lens correspondence differ for every pixel, leads to correlated artifacts at low sampling rates. However, with a sufficiently high number of passes (relatively inexpensive with graphics hardware), artifacts become negligible.

We automatically clamp the number of passes for a bucket based on maximum vertex motion, for motion blur, and maximum area of the circle of confusion, for depth of field. This is a big speed-up for buckets with little or no blur.

Per-Pixel Time Sampling

GPUs can also simulate the traditional stochastic sampling technique of associating a specific time value with each subpixel sample (Cook et al., 1984; Cook, 1986). For each triangle, we rasterize a quadrilateral covering the screen space bounds of the triangle's entire motion, applying a fragment program that interpolates the triangle's vertex coordinates to the time values associated with that pixel and tests the sample point to see if it intersects the interpolated triangle. Points that fail the intersection test `KILL` the fragment, otherwise the depth value of each intersected fragment is computed and output along with the fragment color. This results in an image as shown in Figure 9.6.

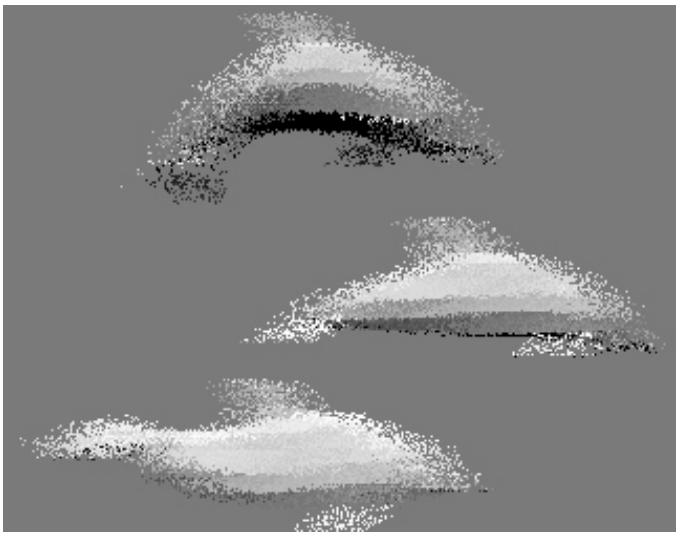

Figure 9.6: Using a fragment program to sample a moving geometry at a different time value per pixel. For clarity, we have used just one time sample per pixel.

Though straightforward, this approach is inefficient for a variety of reasons. The GPU contains dedicated hardware for rasterization, clipping, and culling that is not used by this technique. The moving triangle's bounding box must be computed, either on the CPU or in the GPU vertex program. The resulting rectangular bounds contain many points outside the triangle which must be run through the complex fragment program. The lengthy fragment program that interpolates and tests intersections becomes a bottleneck for GPU throughput. Current GPUs do not efficiently implement *early fragment kill*, which would prevent the computation of the interpolated depth for pixels that fail the hit test. Finally, the fragment program must compute and output z; this is called *depth replacement* and diminishes GPU performance. We have found the multipass techniques we describe to have significantly better performance.

9.3.6 Filtering and Downsampling

Once a bucket has been completely rendered at the supersampled resolution (w_q, h_q), it is downsampled to the final pixel resolution (w_p, h_p) by a two-pass separable filtering algorithm that

runs on the GPU and that supports user-selected high quality filters with large support regions. The downsampled image is then read back to the CPU for output. Care is taken to avoid any readback of the higher resolution images, since readback is slow.

The first pass convolves the x filter kernel with the rows of the supersampled image, resulting in an intermediate image of size (w_p, h_q). The second pass convolves the y filter kernel with the columns of the intermediate image.

The fragment program for each pass is generated on the fly by unrolling the user-specified filter into straight-line code that includes all filter weights and pixel offsets as numerical constants. This avoids per-pixel evaluation, or even per-pixel texture lookup, of the filter kernel, and it avoids loop overhead. Details are available in (Wexler and Enderton, 2005).

9.3.7 Results

We have implemented these algorithms in the context of a production-quality renderer. It has been tested on a wide range of scenes, both real-world and contrived, by ourselves and by others. Compared to software-based stochastic sampling renderers, we have found that our algorithm has comparable image quality and superior performance.

Image Quality

Figure 9.7 shows a portion of a radial test pattern rendered with our algorithm (top) versus stochastic sampling (bottom) for a variety of sampling rates (from left to right, 1, 4, 16, and 32 samples per pixel). At low sampling densities, the regular sampling of the hardware rasterization shows egregious aliasing, but the superiority of stochastic sampling becomes negligible surprisingly quickly. In real-world examples, noticeable artifacts are even less visible than in pathological examples such as this test pattern.

Figure 9.8 compares the motion blur of our algorithm's regular sampling (top) with that of stochastic sampling (bottom), for a variety of temporal sampling rates. Below a certain threshold (dependent on the amount of motion in the scene), regular sampling suffers from significant strobing artifacts, while stochastic sampling degrades more gracefully. However, above that threshold, regular sampling gives a smooth appearance, without the grain of stochastic sampling. Regular sampling (both spatial and temporal) will always have visible artifacts or strobing when the sampling rates are not adequate for the scene geometry or motion. Somewhat more care may be necessary for users to choose adequate sampling rates, compared to stochastic sampling. But modern hardware can rasterize at high resolution and with many passes very rapidly, making the necessary sampling rates quite practical, even in a production setting.

Performance

All of the trials discussed below were timed on a 2.1 GHz Athlon 3000 running Linux, with an NVIDIA Quadro FX 3000 (NV35). We report the sum of user and system time, since time spent in the graphics driver can sometimes register as system time. Rendering times are compared against a highly optimized commercial renderer that uses CPU-based stochastic point sampling (PhotoRealistic RenderMan 12.0). The test scene used is either 1 or 8 copies (not instances) of

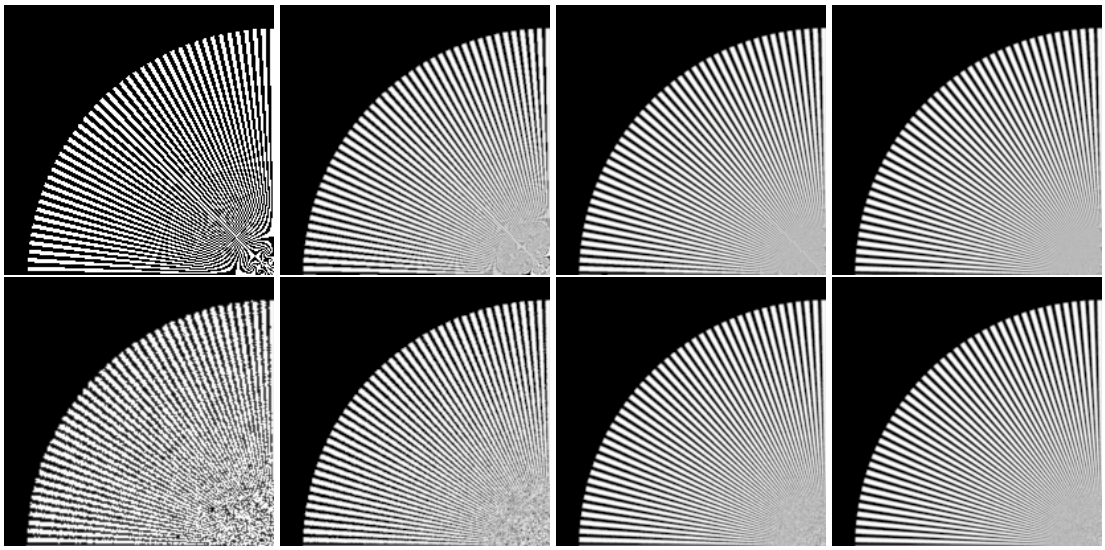

Figure 9.7: Antialiasing quality with 1, 4, 16, and 32 samples per pixel. Top: regular sampling; bottom: stochastic sampling.

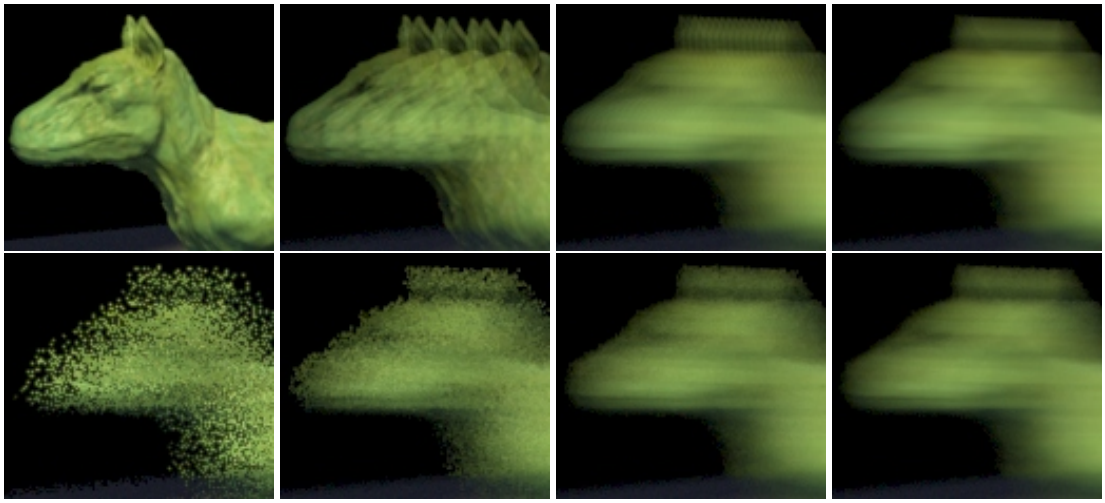

Figure 9.8: Motion blur quality with (from left to right) 1, 4, 16, and 32 temporal samples. Top: regular sampling; bottom: stochastic sampling. Model courtesy of Headus, Inc.

a NURBS character (courtesy of Headus) of about 17,000 control points, rendered with mapped displacement (not just bump mapping), procedural color, simple plastic shading, and four light sources, one of which is shadow-mapped.

To isolate the time for hidden surface removal, we would like to subtract the cost of the other rendering phases such as geometry management (reading, splitting, dicing) and shading (which includes reading texture and shadow maps from disk). We expect those costs to be fixed with respect to spatial and temporal sampling rates. However, we cannot separate these

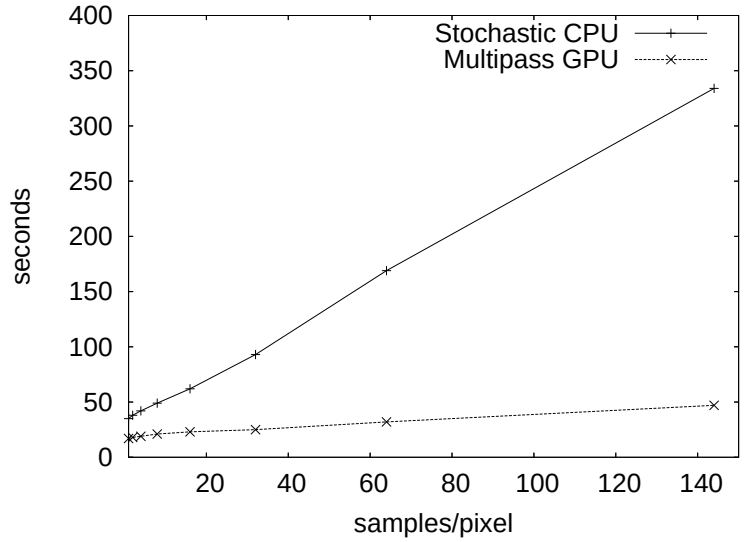

Figure 9.9: Performance comparison of our GPU-assisted multipass algorithm versus a CPU-only stochastic point sampling renderer (PRMan), rendering the displayed frame at 1800×1080 pixels with 1, 2, 4, 8, 16, 32, 64, and 144 samples per pixel. Total render time minus shading delta.

phases precisely, since hiding informs shading. A low estimate of these fixed costs is the time difference between a full render and one with trivial surface shaders, which we call the shading delta.

Figure 9.9 shows rendering time versus spatial sampling for both renderers. Here we have

subtracted a measured shading delta of 32s seconds for our GPU-accelerated renderer and 44s seconds for the CPU renderer. For both renderers, hiding time appears linear in the number of samples, but the marginal cost per sample is 10 times lower for the GPU. At high sample rates, the GPU hider is much faster.

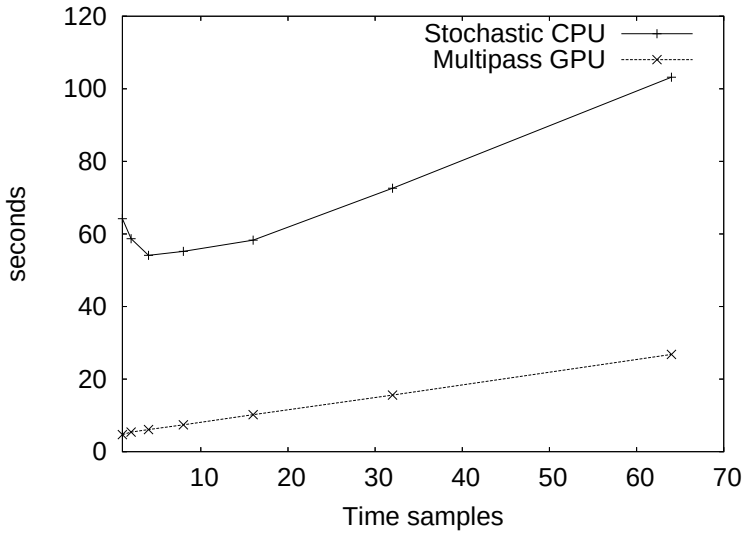

Figure 9.10: Motion blur performance, rendering the full creature shown cropped in Figure 9.8 at 1024×786 pixels. The CPU renderer uses the same samples for time and space; the GPU renderer was run with 16 pixels per sample.

Figure 9.10 shows rendering time versus temporal sampling rate. The stochastic CPU renderer renders 16 time samples in about the same time as 1, for a noisy but usable image. At such low sample rates, the GPU renderer is very fast, but is likely to strobe. For an image that requires 32 regular time samples but only 16 stochastic samples, the GPU renderer is still 3 times faster. Each added sample costs about 0.35 seconds on the GPU or 0.94 seconds on the CPU. The stochastic hider has the constraint that adding time samples requires adding spatial samples, because it uses the same samples for both. This is a disadvantage, particularly since, visually, blurrier scenes normally require fewer spatial samples, not more.

Figure 9.11 shows a *geometry-heavy* scene, an army of 800 NURBS characters. Rendering it with trivial shaders is over 5 times faster with our GPU renderer than with the CPU renderer.[3] While it is hard to estimate how much of that time is hidden-surface removal versus geometry management, it demonstrates that even though large amounts of grid data are being transferred to the GPU, our algorithm still behaves very well.

A scene using global illumination or other slow shading methods will be *shading heavy*. Here the hider's speed is less important than how agressively it culls occluded points before they are shaded. This is more difficult to compare between renderers with differing shading

[3]Stochastic CPU: 721 sec full shaders, 554 trivial shaders. Multipass GPU: 163 sec full shaders, 96 sec trivial shaders.

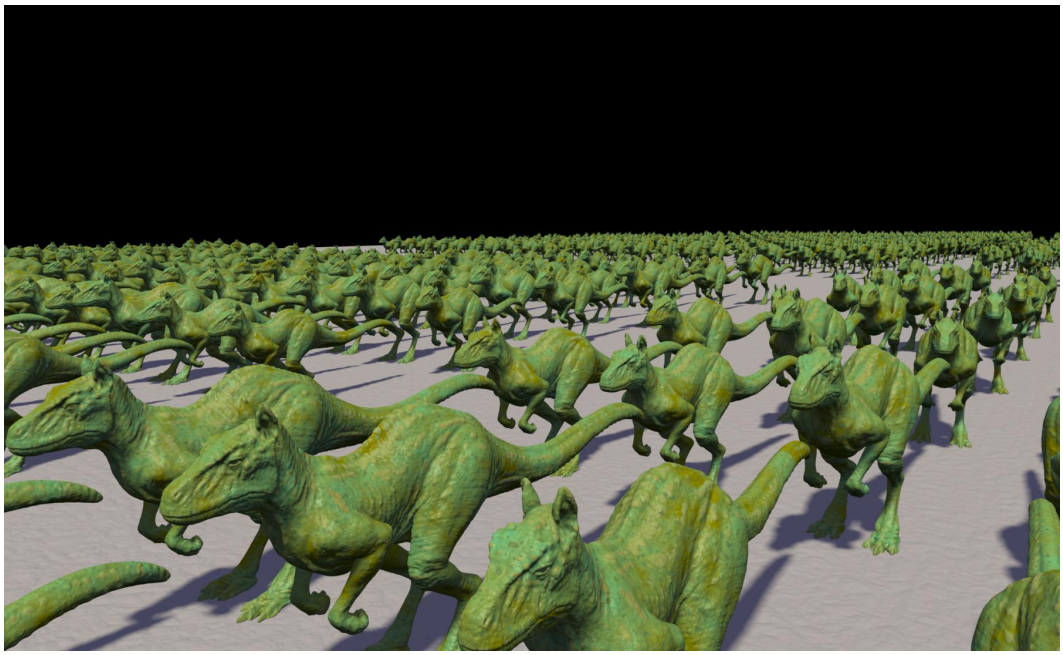

Figure 9.11: Army of 800 displaced NURBS creatures, 1800×1100 pixels, 36 samples per pixel.

and geometry systems. But as mentioned earlier, the GPUs ability to do relatively fast grid occlusion allows us to cull an extra 10-20% of points, versus a typical Reyes CPU algorithm.

Our GPU renderer slows drastically for scenes with *small grids*, such as a million pixel-sized triangles that each form a one-micropolygon grid. Small batch sizes are the bane of GPU performance. A future project is to reduce this problem by combining nearby grids. We would not expect a CPU hider to have this issue.

The slowest cases for the GPU-based hider are those that require many passes. Modest motion blur performs well, as we've seen. But a bucket containing a grid that moves 100 pixels will require about 100 motion blur passes to avoid strobing. So *extreme motion* gets expensive. Extreme motion increases noise in stochastic hiders, but that is usually visually acceptable. Similar issues apply to depth of field. Scenes with *layered transparency* also require the GPU to rasterize grids multiple times, for depth peeling, whereas the CPU can maintain arbitrary lists of fragments per pixel. Combining motion blur with transparency multiplies the required number of passes, and eventually the CPU wins. Figure 9.12 shows a sphere with 550 transparency-mapped "feathers". Even without motion blur, this example runs 50% slower in our renderer than in the CPU renderer. But in motion with 64 time samples, the stochastic renderer is nearly 9 times faster (stochastic CPU: 7.5 sec static, 59 sec moving; Multipass GPU: 12.0 sec static, 525 sec moving).

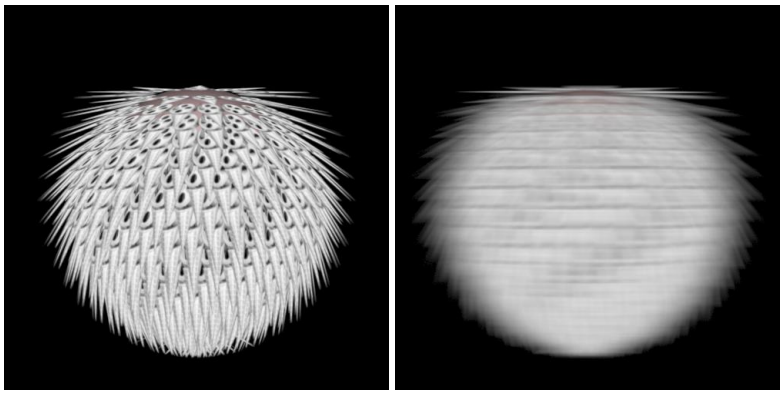

Figure 9.12: A case where our algorithm performs poorly: many stacked motion-blurred transparent objects. The "feathers" are rectangular strips that use a texture to modulate transparency.

9.3.8 Conclusions

We have described an algorithm for hidden-surface removal that leverages commodity graphics hardware to achieve superior quality and features compared to traditional hardware rendering, while outperforming traditional CPU-software-based high-quality rasterization for typical scenes.

This paper makes the following contributions:

- An algorithm that systematically incorporates high-end features into a hardware-oriented rendering framework. These include supersampling at very high rates, user-selected filters with arbitrarily wide support, motion blur and depth of field, order-independent transparency, multi-channel opacity, and multiple output channels.
- Two optimizations of the depth peeling technique, opacity thresholding and z-batches, that allow it to perform in practice as $O(N)$ rather than $O(N^2)$.
- An exploration of the performance of the algorithm in various cases, and of a variety of engineering trade-offs in using GPUs for high-quality hidden surface removal.

It is often assumed that regular spatial sampling will give inferior results to stochastic sampling. When rates of 4 or 16 samples per pixel were considered high, that may have been true. But our experience has been that at the dozens to hundreds of samples per pixel that are easily affordable when leveraging graphics hardware, the deficiencies of regular sampling are visible only with contrived pathological examples, and not in "real" scenes. If desired, artifacts from regular sampling could be further reduced by using interleaved sampling (Keller and Heidrich, 2001) or multisampling (when supported by graphics hardware).

Using advanced GPU features such as floating point precision and detailed occlusion queries can cause current GPU drivers and hardware to run at reduced speed, perhaps 4-16x slower than the optimized paths. Furthermore, despite our algorithm's minimal use of readback, we find the GPU is idle much of the time. With future work to hide more GPU latency, new

GPU designs with fewer penalties for high-precision computation, and the integration of GPU-assisted shading, we hope to recapture this lost performance.

In conclusion, high-quality rendering systems can now be built on a substrate of commodity graphics hardware, for offline rendering as well as for real-time or interactive applications. The considerable power of the GPU can be leveraged without compromising either image quality or advanced features. We expect that similar hybrid hardware/software solutions will become more common as GPUs continue to improve in speed and capability.

Acknowledgements

Some of these notes are adapted from a whitepaper I wrote a couple years ago, and contains elements of talks I've been giving over the past few years concerning film versus game rendering. Other parts of these notes are adapted from a paper I co-wrote with Dan Wexler, Eric Enderton, and Jonathan Rice (as I write these notes, it is as-yet unpublished, though it may have found a home by SIGGRAPH). Even the many parts of these notes that are new may be my words, but they are the result of many people's work and ideas, especially those named above and below.

Gelato is the result of collaboration with many talented individuals, including Dan Wexler, Jonathan Rice, Eric Enderton, John Schlag, Radomir Mech, and Philip Nemec, and with significant input from Cass Everitt and Simon Green, among others. It would be a mere research project, rather than a product that people actually use, if it were not for the talented business team in the Digital Film Group, including Laura Dohrmann, Cynthia Dueltgen, Dave Wilton, and Matt Jefferson, as well as countless other individuals and groups at NVIDIA. And, especially, for David Kirk and Beth Loughney, who provided a stable home and management for the project. (Names in the above lists, in case you are wondering, are in the order they joined the company, and are not meant to imply small relative contribution by people at the ends of the lists.) This is by no means a complete accounting of all people who have contributed in various ways to this work.

Any product names mentioned are trademarked by their respective owners.

Resources

If you're interested in Gelato, it can be downloaded from `http://film.nvidia.com`.

I strongly recommend both *GPU Gems* books. Of particular interest is Chapter 21 of *GPU Gems II*, in which Dan Wexler and Eric Enderton describe GPU-based supersampling and filtering in gory detail, including complete working code.

Bibliography

Apodaca, A. A. and Gritz, L. (1999). *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan-Kaufmann.

ATI (2003). ATI OpenGL Extension Specifications (http://www.ati.com/developer/).

Cook, R. L. (1986). Stochastic sampling in computer graphics. *ACM Transactions on Graphics*, 5(1):51–72.

Cook, R. L., Carpenter, L., and Catmull, E. (1987). The Reyes image rendering architecture. In *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 95–102.

Cook, R. L., Porter, T., and Carpenter, L. (1984). Distributed ray tracing. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 137–45.

Everitt, C. (2001). Interactive order-independent transparency. Technical report, NVIDIA Corp. (http://developer.nvidia.com/).

Greene, N. and Kass, M. (1993). Hierarchical Z-buffer visibility. In *Computer Graphics Proceedings, Annual Conference Series, 1993*, pages 231–240.

Haeberli, P. E. and Akeley, K. (1990). The accumulation buffer: Hardware support for high-quality rendering. In *Computer Graphics (SIGGRAPH '90 Proceedings)*, volume 24, pages 309–318.

Keller, A. and Heidrich, W. (2001). Interleaved sampling. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pages 269–276.

Kelley, M., Gould, K., Pease, B., Winner, S., and Yen, A. (1994). Hardware accelerated rendering of csg and transparency. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pages 177–184.

Levoy, M. (1990). Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261.

Mammen, A. (1989). Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics & Applications*, 9(4):43–55.

Möller, T. and Haines, E. (2002). *Real-Time Rendereing*. A K Peters, second edition.

Porter, T. and Duff, T. (1984). Compositing digital images. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253–259.

Potmesil, M. and Chakravarty, I. (1981). A lens and aperture camera model for synthetic image generation. In *Computer Graphics (SIGGRAPH '81 Proceedings)*, volume 15, pages 297–305.

Segal, M. and Akeley, K. (2003). *The OpenGL Graphics System: A Specification (version 1.5) http://www.opengl.org/*.

SGI (2003). OpenGL Extension Registry http://oss.sgi.com/projects/ogl-sample/registry/.

Torborg, J. and Kajiya, J. (1996). Talisman: Commodity real-time 3d graphics for the pc. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 353–364.

Wexler, D. and Enderton, E. (2005). *GPU Gems II*, chapter 21, High-Quality Antialised Rasterization, pages 331–344.

Wittenbrink, C. M. (2001). R-buffer: A pointerless a-buffer hardware architecture. In *2001 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 73–80.