**From laggard to leader, game graphics
are taking us in new directions.**

NICK PORCINO, LUCASARTS

It has been a long journey from the days of multicolored sprites on tiled block backgrounds to the immersive 3D environments of modern games. What used to be a job for a single game creator is now a multifaceted production involving staff from every creative discipline. The next generation of console and home computer hardware is going to bring a revolutionary leap in available computing power; a teraflop (trillion floating-point operations per second) or more will be on tap from commodity hardware. This leap in power will bring with it a leap in expectations, both on the part of the consumer and the creative professional.

The seeds of the present revolution were sown a long time ago. Everything we do in games started with the telling of a simple story: man against man, man against nature, man against himself. A kid sits down in front of a television and fends off waves of identical invaders marching down the screen toward his inadequately defended base; this story was told a million times in the 1980s, as was the kid's story about himself: "I got the high score of 8,000 points, and there's my initials!"

Gaming Graphics:

Road to Revolution

Gaming Graphics:

Road to Revolution

There's more to telling a story than those simple scenarios, and technology has always had a role to play. Every time our story got better, we needed better hardware. We needed more colors, more memory, better sound, better controllers—and we needed faster processors.

In the early days, characters were represented as simple bitmap images known as *sprites* (see figure 1). Today, a game character is no longer a simple pixilated image. A main character is typically composed of 1,500 to 10,000 geometric primitives and uses somewhere between 100 kilobytes and a few megabytes of textures. Characters have complex animation rigs with driving bones, constraints, attach points, physics setups, and hundreds of unique animations (see figure 2).

In the early days, we looked at the work presented at a fledgling conference called SIGGRAPH (1973), and we thought, "We're going to do that one day." Things really started to change when consoles made the leap from 16-bit processors to 32. Up until then, machines had gotten better and better at moving sprites and backgrounds around; the planar view into flat worlds sported more and more elaborate characters and interactions. A few high-profile consoles pushed this paradigm to the ultimate limit, moving, rotating, scaling, and compositing huge numbers of enormous sprites.

In the end, these consoles fell flat on their faces because of a single innovation present in the competition: hardware-accelerated 3D. Inexpensive graphics cards for the PC and the dedicated polygon hardware in the first Sony PlayStation sounded the death knell of the old paradigm. At last we cut the camera loose, showing our characters and scenes in the round. We looked at SIGGRAPH proceedings from 10 years before and thought, "You know what? We can do that." We used to gauge our progress in terms of how many years we lagged SIGGRAPH; today that gap has closed: We now write papers for SIGGRAPH.

The next generation of consumer hardware is nearly upon us. It will be characterized by supercomputer performance. Multiple CPUs will together provide a teraflop;

there will be dedicated multiprocessing GPUs (graphics processors)—also running at about a teraflop; floating-point pixels will enable superior image compositing. These systems will be 100 times more powerful than the current generation of gaming consoles. Today's game systems expend around 10 GPU cycles per vertex on average; the next generation will be able to expend 100 or more. This amount of processing per pixel will allow shading effects as complex as those we see in special-effects-rich feature films or animated films such as those produced by Pixar Studios.

Current-generation game systems average total character *and* scene loads measured in tens of thousands of triangles; the next generation will dedicate that much complexity to single characters and much more to the environments. We will see multiprocessed rendering pipelines, sophisticated materials, and nearly seamless intercutting between live action or prerendered sequences and interactive content.

Our stories will have the potential for the same depth and sophistication as is expected today in a film or television show. In games, we bring something new to the party with our ability to influence the unfolding of the story, modify the point of view, and selectively reexperience in new ways sequences that particularly intrigue us—possibly in ways not even foreseen by the original crafters of the experience.

This article explores the graphics revolution and some

Typical Sprite

A sprite is a 2D artwork, of characteristically low pixel and color resolution. Encoding a series of frames in individual sprites creates an animation.

FIG 1

Typical Modern Game Character

This character exhibits a fairly conservative level of detail.

FIG 2

of the concepts that will take us into the next generation of game systems. I look at the influence of cinematic realtime rendering, the promise of advanced lighting techniques and high-dynamic range images, the uses of the rendering pipeline, and the future of multiprocessor-based rendering and advanced geometry.

CINEMATIC REALTIME RENDERING

The creation of photorealistic images has a rich history, starting with the earliest renderings of Jim Blinn and other pioneers, leading in a steady progression of refinement to the pixel-perfect renders of *Star Wars* (George Lucas, 1977, 1980, 1983, 1999, 2002), and the sweeping scope of *Lord of the Rings* (Peter Jackson, 2001, 2002, 2003). Progress in realtime rendering has kept pace. Not so long ago, cinematic-quality renderings took hours to complete; with the advent of next-generation game systems, images approaching that quality will be generated at 60 frames per second.

The appearance of an object depends both on shape and shading. *Shape* refers to the geometric or procedural description of an object. *Shading* refers to the combination of light, shade (as in shadows), texture, and color that determines the appearance of an object.¹ A well-designed rendering program provides clean interfaces between geometric processing involving the shape and transformation of objects and optical processing involving the transport and filtering of light.

The final appearance of a scene is achieved by the composition, or *layering*, of elements. Elements may be prerendered as photographed pictures (called plates), shaded shapes, special-effects passes such as atmospheric and pyrotechnic effects, and title components. It has long

been recognized in cinema that some images cannot be captured all at once. Since the dawn of cinematic special effects, background paintings, holdout mattes, multiple exposures, and painted effects over exposed images have been used to create imaginative scenes. To a limited degree, these effects have been exploited in realtime computer graphics; most computer games almost universally render in three layers: a sky dome, the scene geometry, and a head-up display.

The modern realtime architecture was proposed as early as 1982.² Turner Whitted and David Weimer proposed the use of a set of shaders selected according to required rendering features. The next major step was a model separating lights, surface properties, and atmospheric effects, described in 1987 by Robert L. Cook, Loren Carpenter, and Edwin Catmull.³ Since then, there has been much incremental progress, but no major revision to the basic concepts.

The earliest shaders provided only a fixed shader operating on single-surface types. That shader typically used a Lambertian lighting model, affecting simple fall-off of illumination as a surface turned away from a light source. The visibility of an object was determined using simple algorithms such as binary spatial partitioning,

Our stories will have the potential for the same **depth and sophistication** as is expected today in a film or television show.

where objects are recursively grouped in halfspaces, and visibility determined by testing whether the halfspace is visible to the camera. The next evolution was to provide families of shaders that could create different surface appearances by varying parameters. Tables were the next big innovation. Image maps encode surface parameters; texture maps, environment maps, displacement maps, bump maps, and many others are common. Image maps can vary any parameter in a shader equation.

Prominently, Id Software's Quake III put shading under script control. The Quake III shader system allowed control of texture bindings, blending modes, texture coordinate generation, and other aspects of the OpenGL state machine, but it wasn't extensive enough to be considered a full shading language.

Gaming Graphics:

Road to Revolution

Shading systems of the period shoehorned as much as they could into parameters for a single shader, leading to a one-size-fits-all approach. That approach limited the effects that can be achieved and put a cap on performance since even the simplest shaders needed to carry some overhead to support more complicated cases. The introduction of shade trees and shading languages introduced flexible and convenient conceptual models for shaders. Shade trees introduced the concept of a dataflow network for shading computations; shading languages provided a programming model. Customized shaders can be optimized to encapsulate only the data and code necessary for a particular element.⁴

The current leader in shading languages is Pixar's RenderMan. James Kajiya formulated rendering as a light transport problem.⁵ RenderMan defines three categories of shader, based on Kajiya's formula:

1. *Light source shaders.* Calculate the lighting term, which is the color and intensity of light emitted from a source in a particular direction.
2. *Surface shaders.* Calculate the integral of the bidirectional reflectance function with the incoming light distribution.
3. *Volume shaders.* Calculate scattering effects through participating media such as dusty air, salty water, fog, and smoke. Other participating media include translucent materials such as marble, skin, and plants.⁶

Image-Based Lighting

This lighting technique applies sampled "radiant" energy to an object.

FIG 3

The current generation of hardware shaders differs from RenderMan fundamentally; RenderMan is concerned with the description of shaders and lighting, whereas hardware shaders expose the pixel and vertex processors explicitly. Nonetheless, Mark Peercy and colleagues have demonstrated hardware acceleration of Pixar's RenderMan shading system.⁷ The approach they describe is innovative; the OpenGL architecture is treated as a general-purpose SIMD (single instruction stream, multiple data stream) computer, and a compiler translates RenderMan into commands to OpenGL—in essence, treating OpenGL as assembly language. Computation of the shader is performed in the framebuffer directly, using framebuffer-blended copy operations as superscalar computations. Eric Chan and colleagues have another approach, using a method to translate directly from shading languages such as RenderMan to multipass hardware shaders.⁸

A variety of pixel and vertex shading technologies are available in commodity graphics hardware. The OpenGL 1.4 shader language GLSLang, nVidia's Cg (C for Graphics), and Microsoft's DirectX 9 High Level Shading Language are all in this category. Advanced lighting and shading systems will define next-generation game-graphics technology and are absolutely vital for achieving the goal of realtime cinematic rendering.

ADVANCED LIGHTING AND HIGH-DEFINITION RANGE IMAGING

Until recently, compositing, anti-aliasing, motion blur, depth of field, soft shadows, and area lights have been accomplished in hardware via specialized buffers. Limited buffer precision has greatly complicated the problem. The standard representation of an image has been eight bits each for red, green, blue, and alpha—the color components making up each pixel. Every pixel composed into the output buffer reduces the precision of the result. For example, adding a red value of 140 to a red value of 180 yields a final result of 320, which requires nine bits for its representation. Since the framebuffer has only eight bits

for red, this value will be truncated to 255. The result is that colors saturate to muddy values. This saturation limits the number of layers that can be accommodated and the amount of post-processing that can be applied, and thus the complexity and quality of the final image.

The eight-fixed-point-bits-per-channel image representation is now known as LDRI (low dynamic range image). HDRI (high dynamic range image) is a crucial new development. Recent hardware accommodates 16- or 32-bit floating-point values; these HDRI representations are expected to rapidly become the norm. In retrospect, it is obvious that the range and precision of illumination in the real world far exceed that possible in a fixed-point framebuffer. HDRI was pioneered by researchers such as Paul Debevec and promoted through initiatives such as OpenEXR from Industrial Light and Magic.⁹ The next generation of video cards and game consoles will be fully capable of using HDRI in realtime, and many video cards will natively support the OpenEXR format.

An HDRI scene is rendered through a kernel or other transfer function known as a tone map. A *tone map* moves an HDRI into a format suitable for a framebuffer and digital/analog converter, after adjusting the image for perceptual effects.¹⁰ The exposure and other optical properties of the image can be controlled in a way similar to the control possible from a film source. HDRI also opens the door to perceptual effects such as glows and blooming, where saturated or bright regions of an image spill into neighboring regions.

To take advantage of HDRI, next-generation games will need to discard old lighting models in favor of global illumination techniques and light source reflection. One advanced lighting technique that will become increasingly common is image-based lighting. The radiant energy in the environment and reflected components are sampled at a location then applied to an object. The object appears to be perfectly integrated into the scene, as demonstrated by the teapot in figure 3.

Many other techniques will be possible with the new hardware—for example, traditional ray tracing, photon mapping where photons are discretely simulated through a scene,¹¹ and spherical harmonic encoding of radiance transfer. Shadows have not yet received realistic treatment; they are typically represented in games as either hard-edged silhouettes or simple fuzzy blobs.

THE RENDERING PIPELINE

The rendering pipeline manages the conversion of shaders, images, lighting, and geometry into a final composed image (see figure 4). The pipeline has two major categories of API (application program interface):

ries of API (application program interface):

- *Immediate-mode APIs*, such as OpenGL, work on demand, placing geometry fragments and shading instructions in a pipeline for execution in an order determined by the application.
- *Retained-mode APIs* implement a scenegraph or other organizing paradigm. A scenegraph encodes spatial or logical relationships and hierarchies of a scene in a graph structure.

Retained-mode APIs present significant advantages over immediate APIs for interactive rendering as they offer numerous opportunities for parallelization and pipeline optimization. Pixar's hardware-accelerated RenderMan applies compiler optimization technologies to a scenegraph to achieve major performance gains. The Electronic Arts graphics pipeline is another excellent example: The scenegraph is analyzed to sequence render instructions to maximize throughput, and composition interoperates with rendering to produce the final scene.¹²

The rendering pipeline converts the description of a scene into pixels in a framebuffer. A database of geometry is traversed—and objects not visible to the camera are culled, as there is no need to use resources for invisible things. The individual objects to be rendered are submitted

Gaming Graphics:

Road to Revolution

ted as primitives to be transformed into the homogeneous space. The vertices of the geometry are transformed and lit, then turned into primitives such as triangles and line segments. The hardware rasterizes and interpolates the primitives into runs of pixels called fragments. Further operations such as tinting and blending occur on the pixels in the fragments (this is when the pixel shaders run), and the resulting pixels are written to the framebuffer. Intermediate framebuffers are composed into the visible framebuffer for display.

Latency and framerate are two of the most important factors in a satisfactory realtime rendering experience. *Framerate* is the presentation rate of completed images. *Latency* is the delay between the submission of rendered data and the final display of the resulting image. To achieve a solid-looking display, rendering cannot occur to the visible framebuffer except under difficult-to-achieve conditions that involve precise timing to beat the display's refresh. Accordingly, rendering must occur to a nonvisible buffer, and either that buffer must be copied to the visible buffer, or the display refresh must be alternated between two draw buffers as rendering completes. Such a system has a latency of one frame; in other words, the visible frame lags behind the current frame by a single frame.

To run without interlocking processors, modern consoles multiprocess rendering. To accomplish this, not only must the display buffer be alternated with the nonvisible buffer, but the rendering list being constructed by the game's application thread must also alternate with the rendering list being executed by the GPU. The result is that the visible frame lags two frames behind the frame currently being computed. Some modern rendering pipelines have an even greater latency of three frames when a third processor is being used

to construct data (such as cloth or flesh simulation) for rendering.

This latency highlights the importance of framerate, and hence the need for powerful hardware. Flicker fusion studies in the early days of cinema deemed a 24-Hertz framerate satisfactory. In fact, 24 Hz is not sufficient to represent rapidly moving objects, such as might be observed in a flight simulation or fast-moving game. If display lags gameplay by three frames, and the framerate is 30 Hz, the user will see the results of inputs nearly one-tenth of a second after the input is made. One-tenth of a second is perceptible by most people, and that perceptual lag is one of the sources of nausea sometimes experienced by gamers in scenarios with rapidly moving cameras. It follows logically that the higher the framerate and the lower the latency, the better the user experience.

PARALLEL RENDERING

Until recently, rendering was handled by serial processing managed by the CPU. Rendering primitives were handed off to a GPU, the CPU went off and did a little work, then

fed the GPU a bit more. Things have advanced considerably. PC programmers today have to contend with the parallelism of the CPU and the graphics card, and console programmers have to contend with the parallelism of multiple CPUs and a graphics processor. The programmers of the very near future are going to have to deal with multiple homogeneous CPUs and a great many parallel graphics units. The current pipeline paradigms do not hold up well against that kind of architecture.

Most modern realtime rendering pipelines are based on the serial processing of objects into a framebuffer. Objects can be submitted in an arbitrary order, but they must be sorted as a last step before rasterization. This

Until now, geometry
has had fixed
complexity
and topology.

sorting has two purposes: layering of transparent objects and rendering speed. Since objects are drawn into a framebuffer cumulatively, transparent objects need to be drawn back to the front so that things behind the transparent object show through; if the transparent objects were drawn first, the scene would have apparent holes. Changing states on the GPU—uploading textures, turning lighting on and off, and so on—can be very expensive. By sorting all similar states together (everything with the same texture, things that are lit, etc.), you can minimize state changes, and the GPU takes the least amount of time to render the scene.

In general, a multiprocessor-based parallel pipeline distributes geometry among several processors, whose results must ultimately be gathered together into the framebuffer, as shown in figure 5.

There are three kinds of multiprocessor-based architectures: Sort First, Sort Middle, and Sort Last.¹³ They differ according to when primitives are sorted, either before submission to the geometry processors, before submission to the raster processors, or during composition into the framebuffer—as shown in figure 6.

Sort First subdivides the framebuffer into tiles that are mapped to the available processors. Geometry processors are coupled one to one with rasterizers to form complete rendering units. First, geometry is transformed into screen space, and then the transformed primitives are dispatched to the appropriate rendering units. The Sort First architecture can exploit frame-to-frame coherence, redistributing primitives to processors only when they move between screen regions. Sort First is susceptible to load imbalances since some portions of the screen may have many more things to render than other portions.

Sort Middle also subdivides the framebuffer into tiles. Geometry is sorted and distributed among the geometry processors. This is a global operation, because any object could potentially occupy any region of the screen. This is also a high-bandwidth operation since a description of the entire dataset must be transferred among the processors in

every frame. Primitives are transformed, sorted by screen region, and routed from geometry processors to rasterizers. The rasterizers render their region of the screen, then the fragments are collected and assembled into the framebuffer.

The advantage of Sort Middle is that geometry can be distributed among processors without regard to the subdivision of the screen. It has a number of disadvantages, however. It has poor load distribution since some areas of the screen may be relatively unpopulated with geometry.

Gaming Graphics:

Road to Revolution

There is a latency issue since all processors must finish before the final image can be composed. Order-dependent primitives (such as transparent objects) are difficult to accommodate since fragments arrive for processing in nondeterministic order. Bandwidth is the ultimate limiter of performance.

Sort Last renderers pair geometry and raster processing, as they do in *Sort First*; however, *Sort Last* renderers are responsible for rendering a full-screen image using their share of the primitives. The partial images are composited together, taking into account the distance of each pixel in each layer from the camera, which guarantees that the results of the individual renderers are layered correctly. *Sort Last* has the primary advantage of requiring no sorting or redistribution of primitives; each renderer computes its image as if it were the only renderer in the system.

Efficient compositing is key to the *Sort Last* renderer, using the pixel algebra described by Thomas Porter and Tom Duff.¹⁴ Each composited element has an associated matte containing coverage information designating the shapes in the element. Porter and Duff derived a full set

of compositing operations, defining both binary and unary operations. The pixel algebra is associative, making optimization for parallelism possible. The composition of a number of elements can be described as a tree, where terminal nodes are images to be composited, and interior nodes are pixel operators. Recognizing that such a tree is similar to a parse tree, compiler optimization techniques can be used to optimally match the composite operations to the hardware.¹⁵ After the tree has been optimized to minimize operations, it can be partitioned spatially in such a way as to load-balance multiple processors. A final composition pass gathers the outputs of the partitioned subtrees and layers them into the framebuffer.

As processors are added to a system, *Sort Last* architectures scale better than *Sort First* or *Sort Middle*.¹⁶ *Sort Last* architectures will most likely suit next-generation hardware best. The primary disadvantage of *Sort Last* is that it requires a high bandwidth image compositor.

ADVANCED GEOMETRY

Until now, geometry has had fixed complexity and topology. This has been necessary because the preparation of geometry for hardware rendering has involved slow processing to optimally order primitives, maximize texture reuse, and so on. Recent games have seen terrain generated on the fly, creating complexity only where it can be seen. Next-generation platforms will take this technique to new levels since processing power will be available to apply procedural geometry to many more objects in a scene.

Particle systems are a type of procedural geometry used to render time-varying phenomena such as clouds, smoke, and fire, which are difficult to model as surfaces. William Reeves at Lucasfilm first formalized them in 1983 as a follow-up to Jim Blinn's early work on dust and cloud rendering.¹⁷ Up until now, particle systems have been fairly limited, consisting of tens to hundreds of particles. On upcoming hardware, that number should increase a hundredfold, making possible dense volumetric and environmental effects that we've shied away from until now.

FIG 7

Subdivision Surface Model

Subdivision smooths a polygonal surface by repeating a sequence of steps that introduce finer and finer resolution geometry until a user-defined limit is achieved. In this example, some polygon edges have been tagged as being creased, allowing for hard edges and corners where appropriate.

Subdivision surfaces (see figure 7) have emerged as the dominant alternative to the smooth surface patches, or NURBS (nonuniform rational B-splines), used in CAD applications, and have mostly replaced NURBS in many applications. Many games have shipped with simple fixed-subdivision schemes. It is a powerful methodology, and with faster hardware on the way, more complex subdivision schemes are possible. Subdivision models do not suffer from many of the problems inherent in patches. These problems include creases, seams, tears, cracking, inability to localize detail, and topology problems such as valence and continuity limitations. The topology of subdivision models is not limited to a rectangular or triangular topology. Subdivision models are hierarchically editable; the surface can be edited at different resolutions. Recent advances in subdivision formulations allow for hardware acceleration.¹⁸

The subdivision modeling approach involves a simple, low polygon-modeling workflow, involving beveling, extruding, collapse, and a few other elementary operations. Subdivision models achieve a much finer surface resolution with less work by the artist, and the complexity of the final model can be adapted to match scene and performance conditions.

OPPORTUNITIES

The graphics revolution that is upon us will be a creative one; present work methods are too labor intensive to scale to the volume of data that we will need to create to support the medium. We will need new production methodologies blending techniques from games, film, and television. These new techniques will feed back synergistically to their sources and open new creative windows to take storytelling to levels beyond anything we've yet seen. Q

REFERENCES

1. Hanrahan, P., and Lawson, J. A language for shading and lighting calculations. *Computer Graphics* 24, 4 (Aug. 1990), 289–298.
2. Whitted, T., and Weimer, D. M. A software testbed for the development of 3D raster graphics systems. *ACM Transactions on Graphics*, 1, 1 (Jan. 1982), 44–58.
3. Cook, R. L., Carpenter, L., and Catmull, E. The Reyes image-rendering architecture. *Computer Graphics*, 21, 4 (July 1987), 95–102.
4. Abram, G. D., and Whitted, T. Building block shaders. *Computer Graphics* 24, 4 (Aug. 1990), 283–288.
5. Kajiya, J. T. The rendering equation. *Computer Graphics* 20, 3 (Aug. 1986), 143–149.
6. Henrik W. *Realistic Image Synthesis Using Photon Mapping*. AK Peters, Natick: MA, 2001.
7. Peercy, M. S., Olano, M., Airey, J., and Ungar, J. Interactive multi-pass programmable shading. *Proceedings of ACM SIGGRAPH* (2000), 425–432.
8. Chan, E., Ng, R., Sen, P., Proudfoot, K., and Hanrahan, P. Efficient partitioning of fragment shaders for multi-pass rendering on programmable graphics hardware. *Proceedings of SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (2002), 69–78.
9. OpenEXR (from Industrial Light and Magic): <http://www.openexr.org>.
10. Debevec, P. HDRI and image-based lighting. Course 19, SIGGRAPH 2003.
11. See reference 6.
12. Lalonde, P., and Schenk, E. Shader-driven compilation of rendering assets. *ACM Transactions on Graphics* 21, 3 (July 2002), 713–720; see: <http://www.cs.brown.edu/~tor/sig2002/ea-shader.pdf>.
13. Molnar, S. Image-Composition Architectures for Real-Time Image Generation. Ph.D. thesis, University of North Carolina, Chapel Hill, 1991.
14. Porter, T., and Duff, T. Compositing digital images. *Computer Graphics* 18, 3 (July 1984), 253–259.
15. Ramakrishnan, C.R., Silva, C. T. Optimal processor allocation for Sort-Last compositing under BSP-tree ordering. *SPIE Electronic Imaging, Visual Data Exploration and Analysis IV*, 1999.
16. See reference 13.
17. Reeves, W. T. Particle systems—A technique for modeling a class of fuzzy objects. *ACM Proceedings of SIGGRAPH* 17, 3 (July 1983), 359–375.
18. Schaefer, S., and Warren, J. On C2 Triangle/Quad Subdivision. Rice University, preprint (2003).

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

NICK PORCINO has been working in games, computer graphics, artificial intelligence, and robotics since 1981. His adventures have seen him working on autonomous submersibles, robotic toys in Tokyo, game consoles since the industry's earliest days, and games for Disney Interactive and LucasArts. Most recently, he led the team that created R2, LucasArts' high-performance realtime rendering engine. He is a member of ACM SIGGRAPH and the steering committee of the AI Interface Standards Workgroup of the IGDA (International Game Developers Association).

© 2004 ACM 1542-7730/04/0400 \$5.00