Mitsubishi Electric Research Laboratory
MERL - Cambridge Research

Technical Report 2000-TR2000-25 June, 2000

Shadermaps: A Method For Accelerating Procedural Shading

Thouis R. Jones
Ronald N. Perry
Michael Callahan

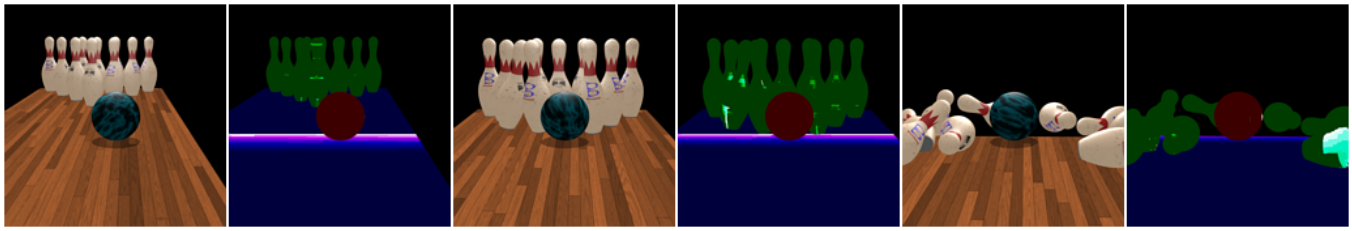
Abstract

Procedural shading has proven to be an indispensable tool for providing photorealistic, photosurrealistic, and artistic effects in computer generated animations. However, due to its computational cost, the time to produce a single frame is measured in hours or days. In this paper we introduce shadermaps, a new method for accelerating procedural shading which exploits inter-frame coherence to significantly reduce rendering times of animations.

Presented at SIGGRAPH 2001 Conference Abstracts and Applications.

Shadermaps: A Method For Accelerating Procedural Shading

Thouis R. Jones, Ronald N. Perry, Michael Callahan
MERL*



Abstract

Procedural shading has proven to be an indispensable tool for providing photorealistic, photosurrealistic, and artistic effects in computer generated animations. However, due to its computational cost, the time to produce a single frame is measured in hours or days. In this paper we introduce *shadermaps*, a new method for accelerating procedural shading which exploits inter-frame coherence to significantly reduce rendering times of animations.

1 Introduction

Procedural shading has become an indispensable tool for producing expressive and compelling computer generated animations. Procedural shading's primary benefit is its flexiblity, since shaders, procedures that calculate the appearance of objects in a scene, can be arbitrarily complex in their actions. This flexibility has its price, however. It can take hours or days to render a single frame for a studio animation.

Shadermaps are a new method for accelerating procedural shading, driven by two observations. First, objects tend to have *intrinsic appearances* that are consistent from frame to frame, or *static*. Second, the intrinsic appearance of an object is usually responsible for most of its visual complexity and its rendering cost. Shadermaps accelerate procedural shaders by taking advantage of this static complexity. Shader computations are separated into static and dynamic phases, and the output of the static phase is stored and reused from frame to frame. In typical animations, shadermaps can significantly reduce the cost of procedural shading.

2 Related Work

An early form of procedural shading appears in [15], where the shading code for the renderer can be rewritten to extend the built-in functionality. [27] introduces a more formal approach by defining a shader dispatch table with built-in shaders; their testbed permits custom shaders to be written, added to the dispatch table, and invoked by the renderer. [27] also develops the notion of deferred shading, where shading parameters are scan converted, stored, and used by a later shading pass. [4] converts simple expressions representing shading computations into a parse tree which is then interpreted by the renderer. [4] was the first to organize the shading computations into light, surface, and atmospheric shaders; he also introduced the term appearance parameters, referring to those parameters that can affect a shading computation. [21] extends [4] with his Pixel Stream Editor, permitting a full procedural language to be used to express a shading computation. [10] combines the best concepts from [4] and [21] with additional features to define the RenderMan Shading Language [26], the premier shading language in use today. [5, 7, 11, 22] provide hardware support for simplified forms of procedural shading. PixelFlow [19] implements a full procedural shading language in hardware, but limited per-pixel memory restricts shader complexity.

[8] describes a limited method for accelerating procedural shading. Their technique caches non-varying computations of a shader in screen space and permits a single appearance parameter to change, reflecting this change interactively. Because the scene must be rendered prior to the interactive stage, many image properties such as the eyepoint and the positions and orientations of objects cannot be altered.

Numerous techniques exist that manipulate data stored in texture maps to enable more flexible shading models. Examples of such techniques range from environment maps [2] and bump maps [3] to the more recent work on a parameter-space shading method that operates at interactive rates [18], an abstract programmable model for multi-texturing [16], and a sample based BRDF representation permitting physically accurate reflection models for local illumination [12]. Similar to shadermaps, [18] includes on-the-fly generation of data in a mipmap, but for a fixed, much more limited shading calculation.

Finally, shadermaps require high quality anisotropic filtering as described in [13, 17, 24].

3 Shadermaps

3.1 Background

Shaders[1] model the interaction of light with an infinitesimal area on a surface. The shader is evaluated for a point on the surface and generates the visible color of that point viewed from a particular direction. The sources of input to a shader, known as *appearance parameters*, include local surface properties and lighting and viewing conditions. The local surface properties might include the color, normal, and (u, v) parameterization of the point where the shader is being evaluated. Lighting conditions include the directions from which the surface is lit, and with what intensity. The viewing conditions provide other information about the scene, such as the current eyepoint. Shaders also have *instance variables*, constants assigned when the shader is created and bound to a surface, which allow multiple behaviors from a single source description. For example, an instance variable can be used to adjust the spacing of features on a surface.

*MERL - Mitsubishi Electric Research Laboratory, jones@merl.com, perry@merl.com, callahan@xmission.com

[1]There are three primary types of shaders in most procedural shading systems: surface, light, and atmospheric. We use "shader" to mean surface shader, though shadermaps can be applied to other types of shaders as well.

3.2 Observations

Most objects in the real world have intrinsic appearances that are consistent over time. For example, a wooden figurine has the same grain pattern even when viewed from different angles or under different lighting conditions. Similarly, procedurally shaded objects in animations tend to have intrinsic appearances that do not change. This is achieved by keeping some subset of the appearance parameters that control the shader *static* between frames.

The concepts of static and (conversely) dynamic appearance parameters should not be confused with the concepts of *uniform* and *varying* appearance parameters present in many shading languages [10]. The former indicate whether an appearance parameter varies between frames, the latter whether it changes across a surface. In the case of the wooden figurine, the grain is static, but obviously not uniform. It is also important to note that even if an object's geometry changes in some way, appearance parameters are still static as long as they remain the same relative to the (u, v) parameterization.

Another observation is that, usually, for an object that looks complex, most of the surface's (non-geometric) complexity is due to the object's intrinsic appearance rather than other causes, such as lighting. In shaders, this complexity results in more computation devoted to determining the intrinsic appearance of an object, rather than to lighting or viewing calculations.

There are counterexamples to the above. Rippling water derives most of its complexity from the visual effect on objects seen through it, and the complexity of a sheet of paper under disco lights is almost entirely due to lighting. We do not address such extremes here, but strive to accelerate the more common cases.

3.3 Algorithm

It is possible to accelerate procedural shading in animations by taking advantage of static complexity. We define the *static phase* of a shader as the part that depends only on static appearance parameters, and the *dynamic phase* as the remainder of the calculation. Since the computations in the static phase depend only on static appearance parameters, it is possible to reuse those computations from one frame to the next. In our algorithm, the output of the static phase is generated at multiple resolutions and stored in a *shadermap*, a mipmap of intermediate computations indexed by the surface parameterization. These intermediate computations are a "snapshot" of the interface between the static and dynamic phases at particular locations on the surface. For each frame, it is possible to reconstruct the intermediate computation at any point on the surface from the shadermap data. This reconstruction is a warp of the shadermap data to screen space according to the surface parameterization and geometry (analogous to warping a texture map), and is performed by a high quality anisotropic filter to minimize aliasing. The dynamic phase uses the result of this reconstruction to complete the shader calculation and produce the final color. Since the anisotropic warp is usually much less expensive to compute than an evaluation of the static phase, the reuse of static computations results in a significant acceleration of the shading calculation.

Most objects are rendered as polygons or patches, both of which have inherent two-dimensional piecewise parameterizations. If a single parameterization over the entire object does not exist, these local parameterizations enable per-patch shadermaps, at a minimum. There are also techniques for creating (u, v) parameterizations over entire objects [14].

Some shaders generate the color for a point based on the three-dimensional position of that point, so that surfaces effectively "carve" their appearance, similar to three-dimensional texture mapping. Shadermaps can still be used with these shaders, since (u, v) determines three-dimensional position, and by extension, the output of the static phase.

One of the benefits of procedural shading is resolution independence; a shader can have detail at a wide range of scales. Generating shadermap data for all of these scales would be wasteful, since in any single frame the dynamic phase requires only part of the data from the shadermap. To avoid excess computation, shadermap data is generated on demand at the resolutions and locations necessary to render the current frame, and stored in a sparsely populated mipmap. The method of storage and generation is discussed in the next section.

In some cases, shadermaps can be applied to computations that are not part of the intrinsic appearance of an object. For example, if an object is viewed under static lighting conditions, then much of the lighting calculation will be static and can be accelerated via shadermaps. The delineation between the static and dynamic phases of a shader depends on the context of the animation that the shader is used in. In an extreme case, the lighting on an object could be static while the appearance parameters affecting surface color were changed each frame. In this case only the lighting calculation could be accelerated by shadermaps, opposite the more common acceleration of the surface color computation.

3.4 Shader and Renderer Changes

From a shader writer's point of view, it is not difficult to adapt a shader to use shadermaps. First, the shader is factored into static and dynamic phases by hand. The static phase becomes a *shadermap shader*, returning its output in some number of channels to be stored in a shadermap by the renderer, and the dynamic phase is modified to access this shadermap as if it were simply a texture map containing arbitrary data. An example shader is shown in Appendix A, both in its original form and factored into static and dynamic phases.

Few changes to the renderer are necessary, again because of the similarity between shadermaps and textures, and also because the dynamic phase simply replaces the unfactored shader. The renderer rasterizes (or otherwise samples) geometry and appearance parameters in screen space, calling the dynamic phase at each sample point. The dynamic phase accesses the shadermap data as if it were a texture, using an anisotropic filter to recreate the output of the static phase at that point, from which it completes the shading calculation to produce a color. Figure 1 shows a comparison of the dataflow for a traditional shader and for a shader using shadermaps. The dynamic phase takes the place of the unfactored shader, and the shadermap is similar to a cache between the dynamic and static phases.

To avoid unnecessary computation, the data in the shadermap is generated on demand. If the anisotropic filter operation requires shadermap data that has not yet been created, the shadermap shader (i.e., the static phase) is invoked to produce the needed data, and the result stored in the shadermap for future computations. To reduce the overhead of on-demand generation, shadermaps are stored in a sparse mipmap made up of tiles, and data for an entire tile is generated when that tile is first accessed. Since the data in a shadermap is produced at several resolutions, it is similar to a mipmap. Unlike mipmaps, however, the data is generated directly by a procedure, rather than lower-resolution levels being filtered versions of a higher-resolution image.

The optimal size for a shadermap tile depends on several factors, not the least being the access pattern of the shadermap which depends on the animation in non-obvious ways. Good results have been achieved with tile sizes between 8×8 and 64×64, but an in-depth analysis of the effect of tile size on speed has not been performed.

Note that when using an unfactored shader, a procedural shading system must rasterize all of the appearance parameters that contribute to the shading computation. When using shadermaps, only the dynamic appearance parameters and (u, v) need to be rasterized. This alone can be a significant reduction in computation.

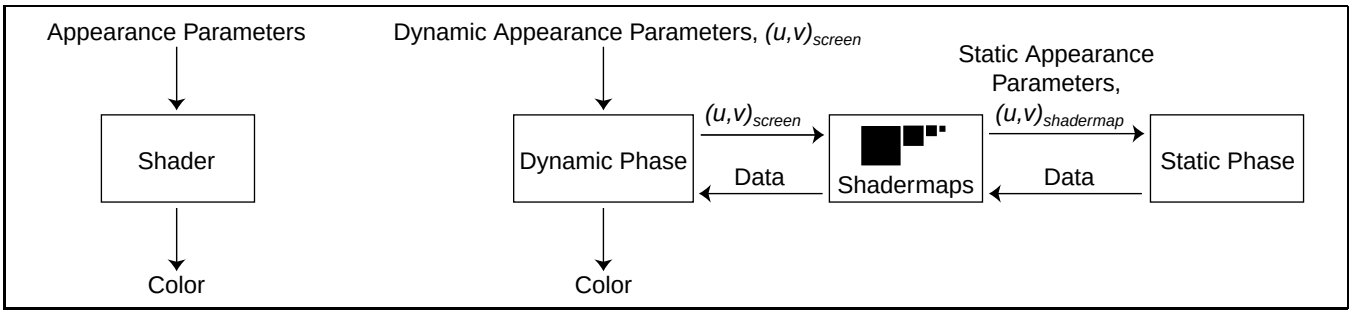


Figure 1: Dataflow diagrams for a traditional shader and a shader using shadermaps. On the right, the shader has been separated into its dynamic and static phases. The shadermap is similar to a cache between the two phases. Though not shown, the $(u, v)_{screen}$ and dynamic appearance parameters include screen-space derivatives, necessary for the anisotropic filter. The $(u, v)_{shadermap}$ and static appearance parameters include (u, v) space derivatives, used for antialiasing.

3.5 $(u, v)_{screen}$ vs $(u, v)_{shadermap}$

The dynamic phase's only access to the shadermap data is via the anisotropic filter. A request for the data at a given (u, v) results in the filter being applied to a set of samples, each at a different (u, v) location in the shadermap. The distinction is denoted by $(u, v)_{screen}$ and $(u, v)_{shadermap}$ (see Figure 1). Note that because the shadermap is evaluated at grid locations (similar to a mipmap), it is unlikely that any one $(u, v)_{shadermap}$ corresponds exactly to the original $(u, v)_{screen}$.

3.6 Antialiasing in the Static Phase

Most shaders support antialiasing by automatically bandlimiting the signal they generate based on the screen-space derivatives of appearance parameters [6]. Such techniques translate seamlessly to shadermaps, since the derivatives of an appearance parameter in (u, v) space, where the static phase operates, can be calculated trivially during that appearance parameter's interpolation. It is important to note that the static phase of the shader is responsible for antialiasing the data that appears in the shadermap. This is analagous to an unfactored shader being responsible for antialiasing its output in screen space. With shadermaps, the anisotropic filter minimizes aliasing during the warp of the shadermap data to screen space. In effect, the combination of antialiasing in the static phase and in the anisotropic filter antialiases data in screen space.

4 Results

4.1 Acceleration

There are three factors that determine the effectiveness of shadermaps at accelerating procedural shading calculations: reuse of shadermap data, the cost of the static phase, and the cost of reconstructing the intermediate computation.

The amount of reuse of shadermap data for an animation is the ratio of the total number of evaluations of the dynamic phase to total number of evaluations of the static phase. A ratio of 10:1, for example, means that for every ten evaluations of the shader (i.e., the dynamic phase), the static phase is evaluated only once. This ratio does not depend on the particular shader applied to an object, but rather on the directions from which the object is rendered along with the object's surface parameterization and geometry. Table 1 shows evaluation counts and ratios for several animations.

A frame from the Bowling animation (inspired by "Textbook Strike" from the cover of [26]) is shown in Figure 2, the Spinning animation is of rotating teapot, and Walking is a first person walk down a hallway. As can be seen, reuse of shadermap data is high, with most ratios around 15:1, or about 95% reuse. This has been the case in all animations that we have generated.

The cost of reconstruction of the intermediate computation is the next most significant factor governing shadermap's effectiveness.

Animation	Object	Shader Evaluations	Static Phase Evaluations	Ratio
Bowling	Floor	3.2×10^7	1.7×10^6	$19:1$
	Pins	6.8×10^7	5.4×10^6	$13:1$
	Ball	1.2×10^7	5.0×10^5	$24:1$
Spinning	Teapot	2.1×10^7	2.6×10^6	$8:1$
	Floor	9.7×10^6	2.7×10^5	$36:1$
Walking	Walls	1.0×10^8	1.2×10^7	$8:1$

Table 1: Number of shader and static phase evaluations

Shader	Static Phase	Overhead	Number of Channels	Reconstruction Cost (Isotropic)
Floor	280	0	3	60
Pins	20	50	3	60
Ball	130	40	1	20

Table 2: Approximate FLOP counts for shaders

We use the FELINE algorithm, which constructs an anisotropic filter as a weighted sum of a set of trilinear "probes." The probes are taken along the major axis of the ellipse formed by projecting a circular pixel window to the surface. Further details can be found in [17].

The cost of the FELINE reconstruction is dominated by the evaluation of trilinear probes. The number of probes used in a single evaluation of FELINE depends on the anisotropy of the surface in screen space. We estimate the cost of a single probe to be ≈ 20 FLOPS, per shadermap channel. Note, however, that the computation of trilinear probes is highly regular and amenable to parallelization; significant speedups can be achieved by taking advantage of modern processor enhancements such as Intel's Streaming SIMD Extensions or the Alpha processor's deep floating-point pipelines.

The savings due to shadermaps also depend on the cost of the static phase, including any overhead induced by factoring. Table 2 gives the results of a hand analysis of the shaders used to generate the Bowling animation.

The Floor shader is similar to the **oakplank** shader given in [1]. It involves several calls to expensive shading language functions, such as `noise`. The boundary between the static and dynamic phases is three variables which control the grain of the wood, its shininess, and the lines between planks. The Ball shader is a classic turbulence function modulated by a color spline, with a single variable linking the static and dynamic phases. Finally, the Pin shader (based on [26]) is fairly simple, involving classification of the pin surface into regions (base, body, crown) by three-space position, and the use of texture maps for the labels and scratches on the pins. The intermediate computation is simply the pin's intrinsic surface color.

The values in the *Overhead* column in Table 2 are from the extra effort necessary to convert (u, v) into three-space position for

Shader	Static Phase (Unfactored)	Shadermaps (Factored)
Floor	280	165
Ball	130	57

Table 3: Comparison (FLOP count) of unfactored versus factored static phases

the two latter shaders, which "carve" appearance from a three-dimensional function. This overhead, part of the static phase, is not necessary in unfactored shaders because three-space position is already calculated in the normal course of rendering. Since the static phase is not evaluated at exactly the same locations as the dynamic phase (§3.5), it must calculate three-space position independently. This calculation can be expensive, requiring calls to trigonometric functions, but its effect on the overall computation is significantly reduced because of the high reuse of shadermap data.

The *Reconstruction Cost* column shows the cost of reconstructing the intermediate computation, ignoring anisotropy. From Table 2, it is obvious that some shaders, such as Pins, cost so little to evaluate that they do not benefit from shadermaps.

Finally, Table 3 compares the cost of the static phase in an unfactored shader to the cost when this phase is accelerated using shadermaps. The formula for the latter is $C = R \times A + (G + O)/U$, where R is the cost of an isotropic reconstruction of the intermediate computation, A the average anisotropy (2.5 for the Bowling animation), G the cost of an evaluation of the static phase, O the overhead from factoring, and U the ratio from Table 1. As noted above, the pin shader does not benefit from shadermaps.

A direct comparison of an unfactored shader and that of a shader using shadermaps is very context dependent. The overall cost of either method depends on the complexity of the dynamic phase. Under simple lighting conditions, shadermaps can reduce total shading computations to a fraction of their original number. However, the relative improvement is diminished as the dynamic phase becomes more expensive.

From inspection of the formula above and the results in Table 3, it can be seen that the computational cost of shadermaps is dominated by anisotropic reconstruction. The other costs, from evaluation of the static phase and the overhead from factoring, are attenuated by the high reuse of shadermap data. Thus, shadermaps allow the use of significantly more complicated static phase computations without a large increase in shading cost.

4.2 Quality Comparison

A side-by-side comparison of images generated with and without shadermaps is given in Figure 2. Though subtle variations exist, there is no visible difference in quality. (The source of these variations is discussed in the next section.)

Figure 3 shows frames from the Bowling animation shaded according to the age of the shadermap data accessed during rendering, with brighter areas younger.

4.3 Approximation and Error

Shadermaps do not exactly reproduce the output of the shader they are derived from, since the warp to screen space implies a loss of accuracy. In most cases, the approximation is visually indistinguishable, but there are situations where it causes incorrect results. If the dynamic phase is highly nonlinear, minor errors could result in large changes in appearance, precluding the use of shadermaps. This problem is not unique to shadermaps; bump maps [3], for example, do not render correctly when filtered directly because of their nonlinear interaction with lighting [23, 20].

5 Discussion and Future Work

It might be possible to automatically factor shaders using dataflow analysis, as in [8], or simplify hand factoring by adding a syntactic device, such as a *static* keyword, to the shading language. However, a benefit of explicit factoring is that it gives more control to the shader writer, as well as allowing static and dynamic phase procedures to be reused separately and shared between objects simultaneously.

It is interesting to consider the amount of memory required to support shadermaps. An upper bound of memory accessed while rendering a single frame (the minimum requirement) is given by multiplying the number of pixels ($\approx$ 2 Mpixels), the average size of the intermediate computations stored in a shadermap ($\approx$ 10 bytes/shadermap sample), and the average per-pixel anisotropy after the warp to screen space ($\approx$ 2-3 shadermap samples/pixel). If the renderer supports transparency, the depth complexity ($\approx$ 10) also contributes, but optimizations for opaque objects apply. These quantities all vary considerably, and in some cases the memory cost might be so large as to require cache replacement strategies or preclude the use of shadermaps altogether.

Shadermaps have potential applications beyond acceleration of software procedural shading. So far, hardware systems implementing procedural shading [19] have been limited in functionality, largely because of the per-pixel memory required to store appearance parameters and intermediate values throughout shader calculations. Shadermaps reduce the number of appearance parameters that need to be rasterized and can also act to simplify the shader computation, both of which reduce memory requirements. On such systems, shadermaps would allow the use of more complex shaders without a per-pixel memory increase.

Shadermaps might also apply to more common hardware. One view of shadermaps is as dynamically generated texture maps containing arbitrary data. The dynamic phase can likewise be seen as a complex multitexturing engine. Interestingly, consumer-level graphics hardware is already moving in directions compatible with these interpretations. PC graphics cards with support for on-demand loading of partial textures from system memory have recently become available [28]. Adapting this technology to dynamically generated data is conceivable, though methods for dealing with "cache misses" would be necessary, as in [25]. Concurrently, increasingly complex multitexturing graphics engines have been proposed [12, 16] and the trend in consumer hardware has also been towards more powerful multitexturing [9]. The advances from these two areas combined with shadermaps point to a possible shortcut to powerful procedural shading on consumer-level graphics hardware. However, some problems remain to be solved. Texture memories are usually smaller than those of general purpose CPUs, so careful cache management and replacement strategies would have to be devised. Shadermaps also increase the bandwidth requirements of texture memory, an already strained resource. However, the advent of embedded DRAM might remove this limitation. Finally, the static evaluations increase the load on the host CPU, or require a second general purpose processor dedicated to this task.

Another possibility is to use hardware to accelerate the most expensive part of shadermaps, the reconstruction calculation, while continuing to perform the other rendering and shading calculations in software. Such a "reconstruction accelerator" would be much simpler than most graphics accelerators, consisting of not much more than an anisotropic filtering engine and memory for storing the shadermaps.

In conclusion, shadermaps accelerate procedural shading by exploiting inter-frame coherence to reduce computation. Shadermaps enable the use of much more complex shaders without a corresponding increase in shading time since the cost of shading is amortized over several frames. Furthermore, shadermaps recast the shading computation into a form which is amenable to hardware acceleration, potentially providing a leap in quality in consumer-level graphics.

References

[1] Anthony A. Apodaca and Larry Gritz. *Advanced Renderman*. Morgan Kaufmann, 2000. ISBN 1558606181.

[2] J. F. Blinn and M. E. Newell. Texture and Reflection in Computer Generated Images. *Communications of the ACM*, 19:542—546, 1976.

[3] James F. Blinn. Simulation of Wrinkled Surfaces. *Computer Graphics (Proceedings of SIGGRAPH 78)*, 12(3):286–292, August 1978.

[4] Robert L. Cook. Shade trees. *Computer Graphics (Proceedings of SIGGRAPH 84)*, 18(3):223–231, July 1984.

[5] Michael F. Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics. *Computer Graphics (Proceedings of SIGGRAPH 88)*, 22(4):21–30, August 1988.

[6] David Ebert, Kent Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, second edition, 1998. ISBN 0122287304.

[7] David Ellsworth. Parallel Architectures and Algorithms for Real-time Synthesis of High-quality Images using Deferred Shading. In *Workshop on Algorithms and Parallel VLSI Architectures*. 1991.

[8] Brian Guenter, Todd B. Knoblock, and Erik Ruf. Specializing Shaders. *Proceedings of SIGGRAPH 95*, pages 343–350, August 1995.

[9] Pat Hanrahan. Real Time Shading Languages: 10 Years Back to the Future. Keynote Speech, 1999 Eurographics/SIGGRAPH Workshop on Graphics Hardware, 1999.

[10] Pat Hanrahan and Jim Lawson. A Language for Shading and Lighting Calculations. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4):289–298, August 1990.

[11] John C. Hart, Nate Carr, Masaki Kameya, Stephen A. Tibbitts, and Terrance J. Coleman. Antialiased Parameterized Solid Texturing Simplified for Consumer-Level Hardware Implementation. In *Proceedings of the 1999 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 45–53. 1999.

[12] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, Hardware-Accelerated Shading and Lighting. *Proceedings of SIGGRAPH 99*, pages 171–178, August 1999.

[13] Tobias Hüttner and Wolfgang Straßer. Fast Footprint MIPmapping. In *Proceedings of the 1999 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 35–43. 1999.

[14] Bruno Lévy and Jean-Laurent Mallet. Non-Distorted Texture Mapping for Sheared Triangulated Meshes. *Proceedings of SIGGRAPH 98*, pages 343–352, July 1998.

[15] Nelson L. Max. Vectorized Procedural Models for Natural Terrain: Waves and Islands in the Sunset. *Computer Graphics (Proceedings of SIGGRAPH 81)*, 15(3):317–324, August 1981.

[16] Michael D. McCool and Wolfgang Heidrich. Texture Shaders. In *Proceedings of the 1999 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 117–126. 1999.

[17] Joel McCormack, Ronald Perry, Keith I. Farkas, and Norman P. Jouppi. Feline: Fast Elliptical Lines for Anisotropic Texture Mapping. *Proceedings of SIGGRAPH 99*, pages 243–250, August 1999.

[18] Gavin Miller, Mark Halstead, and Michael Clifton. On-the-Fly Texture Computation for Real-Time Surface Shading. *IEEE Computer Graphics & Applications*, 18(2):44–58, March 1998.

[19] Marc Olano and Anselmo Lastra. A Shading Language on Graphics Hardware: The PixelFlow Shading System. *Proceedings of SIGGRAPH 98*, pages 159–168, July 1998.

[20] Marc Olano and Michael North. Normal Distribution Mapping. Technical Report 97-041, University of North Carolina, 1997.

[21] Ken Perlin. An Image Synthesizer. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):287–296, July 1985.

[22] John Rhoades, Greg Turk, Andrew Bell, Andrei State, Ulrich Neumann, and Amitabh Varshney. Real-time procedural textures. *1992 Symposium on Interactive 3D Graphics*, 25(2):95–100, March 1992.

[23] Andreas Schilling. Toward Real-Time Photorealistic Rendering: Challenges and Solutions. *1997 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 7–16, August 1997.

[24] Andreas Schilling, Günter Knittel, and Wolfgang Straßer. Texram: A Smart Memory for Texturing. *IEEE Computer Graphics & Applications*, 16(3):32–41, May 1996.

[25] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The Clipmap: A Virtual Mipmap. *Proceedings of SIGGRAPH 98*, pages 151–158, July 1998.

[26] Steve Upstill. *The Renderman Companion*. Addison Wesley, Reading, MA, 1989. ISBN 0201508680.

[27] J. Turner Whitted and D. M. Weimer. A software test-bed for the development of 3-D raster graphics systems. *Computer Graphics (Proceedings of SIGGRAPH 81)*, 15(3):271–277, August 1981.

[28] 3Dlabs, Inc. *The Virtual Texture Engine*, 1999. Available at www.3dlabs.com/-products/vtpaper.pdf.

A Example Shader

The **rings** shader below is a simplified shader for a wooden object prior to factoring, written in the RenderMan shading language. It simulates wood as nested cylinders with a periodically varying color, from which surfaces take their appearance. The calculation takes place in a local shader space, so that when the associated object moves, the rings stay in the same position on the object. The position within a ring determines the color of a point and its shininess; darker parts of the wood appear shinier.

The boundary between the static and dynamic phases of the shader is marked in the code, under the assumption that this shader is used in the typical way, i.e. associated with an object with fixed geometry.

This shader is much simpler than most. For instance, there is no code to handle antialiasing, opacity, or ambient light. The static phase would typically be much more complicated, with calls to expensive functions such as `noise`, and a more elaborate color mapping performed with `spline`.

rings requires that two built-in appearance parameters be interpolated, the normal N and the position P. P is used in the ring color computation, and both N and P are used in the lighting calculation (P is an implicit parameter in the calls to `diffuse` and `specular`.)

```
surface
rings(float Ks = .6, Kd = .6,
            roughness = .1;
      color lightwood = color (.69, .44, .25),
            darkwood  = color (.35, .22, .08),
            specularcolor = 1;)
{
   point PP;
   float dist;
   normal Nf;

   /* Transform position P to local shader space.      */
   PP = transform("shader", P);

   /* Find the distance from PP to the line z = 0.     */
   dist = sqrt(xcomp(PP)*xcomp(PP) +
               ycomp(PP)*ycomp(PP));

   /* Make rings from fractional part of dist...       */
   dist -= floor(dist);
   /* ...and use it to linearly blend wood colors.     */
   Ci = mix(lightwood, darkwood, dist);

   /* ----------- STATIC/DYNAMIC BOUNDARY ------------ */

   /* Multiply this color by the diffuse lighting.     */
   Nf = faceforward (normalize(N),I);
   Ci *= Kd*diffuse(Nf);

   /* Add the specular highlight, attenuated by ring   */
   /* position.  Darker areas have brighter specular   */
   /* highlights.                                      */
   Ci += dist * specularcolor * Ks *
         specular(Nf,-normalize(I),roughness);
}
```

rings can be replaced with the two shaders below, **rings_static** and **rings_dynamic**, joined by a shadermap storing the intermediate computations, represented by Ci and dist. The shadermap storage will have been defined in the scene description before these procedures are invoked. The definition includes a name to reference the shadermap and the number and type of each element of the shadermap (a color and a float, in this example). The code below borrows the syntax of the RenderMan shading language, introducing the new functions **shadermap_store** and **shadermap** along with a new shader type, `shadermap_shader`.

The shadermap shader **rings_static** performs the static phase of the **rings** shader. Its output is stored in a shadermap, referenced by name. **rings_static** uses the built-in appearance parameter P, which must be interpolated based on the current (u,v). The **shadermap_store** function takes (u,v) as an implicit argument, specifying the location at which to store its data.

```
shadermap_shader
rings_static(string shadermap_name;
             color  lightwood = color (.69, .44, .25),
                    darkwood  = color (.35, .22, .08);)
{
   point PP;
   float dist;

   PP = transform("shader", P);
   dist = sqrt(xcomp(PP)*xcomp(PP) +
               ycomp(PP)*ycomp(PP));
   dist -= floor(dist);
```

```
    Ci = mix(lightwood, darkwood, dist);

    /* Store intermediate results in the shadermap.     */
    /* The (u,v) location to store the data at is an    */
    /* implicit argument.                               */
    shadermap_store(shadermap_name, 0, dist);
    shadermap_store(shadermap_name, 1, Ci);
}
```

The surface shader **rings_dynamic** performs the dynamic phase of the **rings** shader. It first reconstructs the output of the static phase through calls to **shadermap**. It then performs the lighting calculation, which completes the shader computation.

```
surface
rings_dynamic(string shadermap_name;
              float  Ks = .6, Kd = .6,
                     roughness = .1;
              color  specularcolor = 1;)
{
    normal Nf;
    float dist;

    /* Reconstruct the intermediate results from the    */
    /* shadermap. As in the texture function, (u,v) for */
    /* the current point is an implicit argument to the */
    /* shadermap function.  shadermap is responsible    */
    /* for performing the anisotropic filter of the     */
    /* shadermap data as well as handling the on-demand */
    /* generation of that data.                         */
    dist = float shadermap(shadermap_name, 0);
    Ci = color shadermap(shadermap_name, 1);

    /* Complete the shader computation                  */
    Nf = faceforward (normalize(N),I);
    Ci *= Kd*diffuse(Nf);
    Ci += dist * specularcolor * Ks *
          specular(Nf,-normalize(I),roughness);
}
```

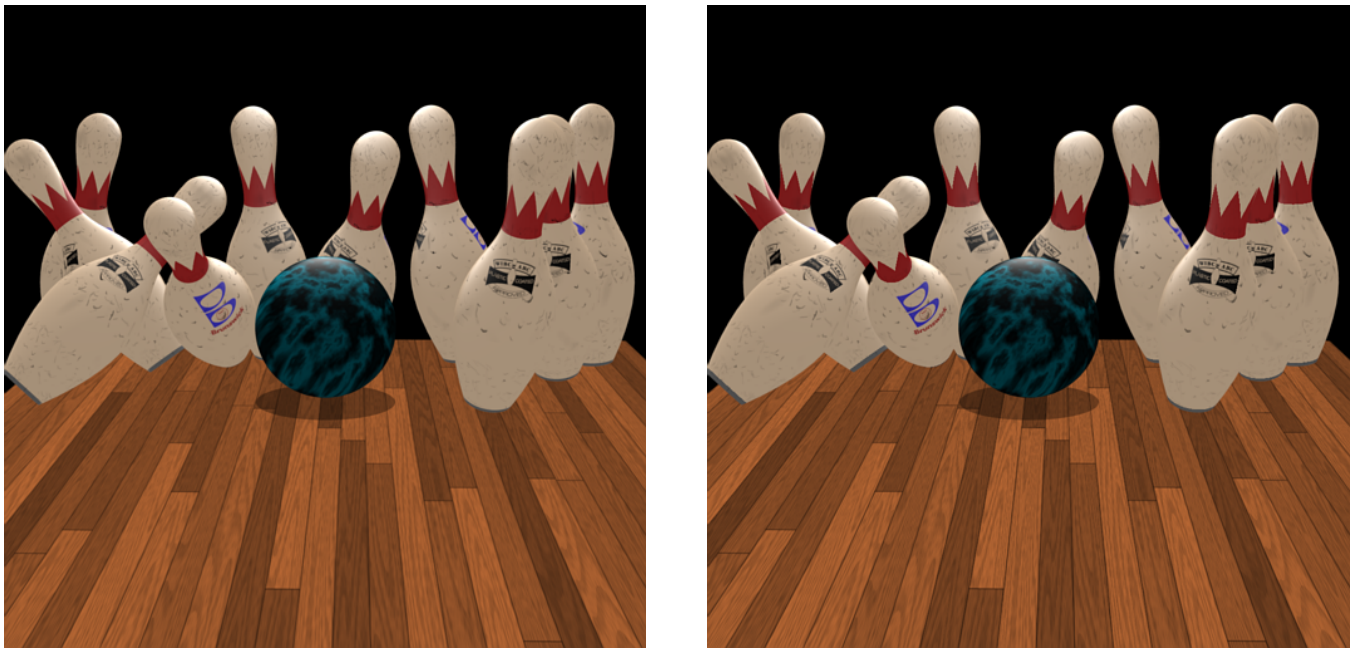


Figure 2: Images generated with shadermaps (left) and without shadermaps (right)

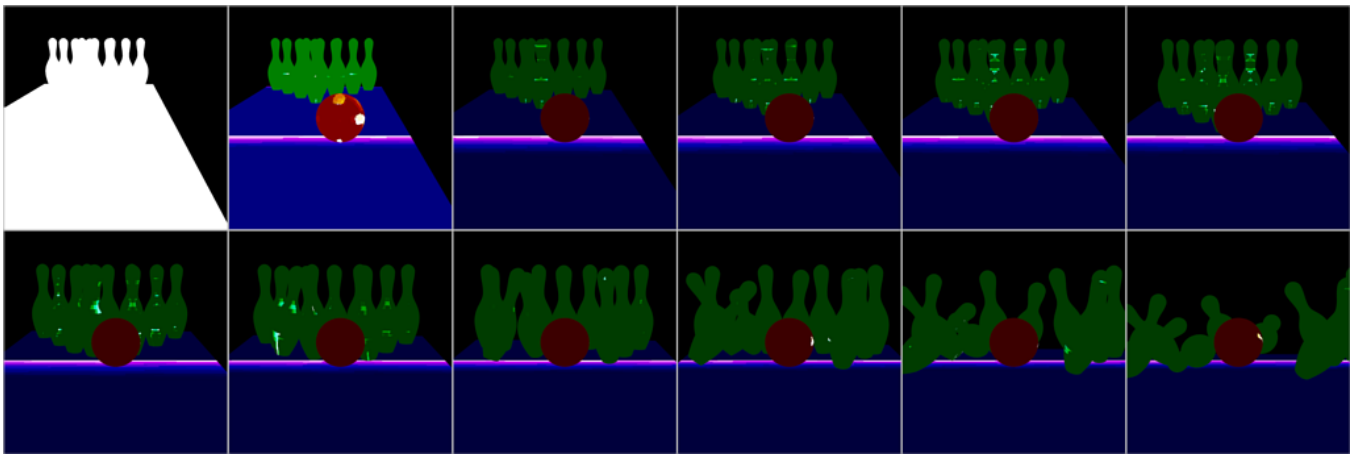


Figure 3: Frames from the Bowling animation. Pixels are shaded according to the age of the shadermap data accessed during rendering, with brighter areas younger. The bright line on the floor is due to the camera's forward movement.