Texture Shaders*

Michael D. McCool† University of Waterloo

Wolfgang Heidrich‡ Max Planck Institute

Abstract

Extensions to the texture-mapping support of the abstract graphics hardware pipeline and the OpenGL API are proposed to better support programmable shading, with a unified interface, on a variety of future graphics accelerator architectures. Our main proposals include better support for texture map coordinate generation and an abstract, programmable model for multitexturing.

As motivation, we survey several interactive rendering algorithms that target important visual phenomena. With hardware implementation of programmable multitexturing support, implementations of these effects that currently take multiple passes can be rendered in one pass. The generality of our proposed extensions enable efficient implementation of a wide range of other interactive rendering algorithms.

The intermediate level of abstraction of our API proposal enables high-level shader metaprogramming toolkits and relatively straightforward implementations, while hiding the details of multitexturing support that are currently fragmenting OpenGL into incompatible dialects.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics Processors; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture.

Keywords: Hardware acceleration and interactive rendering, BRDFs, shadows, shading languages, illumination, OpenGL.

1 Introduction

Several multipass algorithms have been proposed recently to implement photorealistic and/or physically-based rendering algorithms with hardware acceleration. Algorithms exist for global illumination, local lighting models with arbitrary anisotropic reflectance distributions and normal/tangent interpolation, and simulation of arbitrary lens systems. Unfortunately, the multipass nature of these algorithms can result in relatively disappointing performance.

Multipass algorithms suffer a severe performance penalty if the geometry must be sent through the pipeline more than once. Effectively, the utilization of the geometry stage of the pipeline is divided by the number of passes, since for many multipass algorithms, the geometry of each pass is identical. This is a serious problem as for many applications existing acceleration architectures are geometry transformation or geometry bandwidth limited.

In addition, multipass algorithms can consume a great deal of memory storing frames of intermediate results, a problem which will only be compounded when signed high-precision frame buffer and texture map color formats, needed for more advanced rendering effects [20, 21], are available. Tiling the framebuffer can help to address these problems, but only at a cost in software complexity, and possibly in overall performance.

In order to improve the performance of high-quality hardware-accelerated rendering and reduce the geometry transformation and bandwidth bottleneck, more should be done on each pass. The tradeoffs are similar to the RISC *vs.* CISC tradeoffs in CPU design. However, in graphics accelerator design there are multiple dimensions in which complexity could be added or reduced, and in some cases the conclusions may be different than those in CPU design. For instance, RISC programs are longer than CISC programs since they do less with each instruction, but because (non-self-modifying) programs are static, instruction caches can address the potential bandwidth problem. The equivalent issue in graphics accelerators is geometry bandwidth. Unfortunately, in animations and highly interactive virtual environments (for instance, games) the transformed geometry can change in every frame, so geometry caching may be of little use.

One way to reduce the number of passes would be to provide specialized hardware support for each desired rendering effect: Phong shading, shadows, bump mapping, *etc*.

However, implementing specialized hardware for specific rendering algorithms and effects is not cost-effective, and does not permit the resources used for these effects to be retargeted to other rendering problems. Such specific rendering effects are the CPU equivalent of an "evaluate polynomial" instruction.

On the other hand, adding too much generality and programmability to graphics hardware, for instance by using a parallel MIMD array of general-purpose processors or a SIMD processor-enhanced frame buffer memory, may be overkill. In particular, we usually only have to evaluate relatively simple expression trees [4] to compute the color to apply to a fragment, not Turing-complete programs.

As a middle ground, we propose a small number of conceptual changes to the standard texture-mapping and compositing pipeline and the OpenGL API to enable practical hardware-accelerated procedural shaders. These "texture shaders" would add a measure of programmability to graphics hardware but are not so general that analysis and optimization are impossible.

While the abstract model of texture shaders is at the level of multitexturing, the API is in fact at a high enough level that a wide range of hardware implementations are possible, including pure multipass, SIMD processor-enhanced memory, hybrid multipass combined with fixed-architecture multitexturing, and finally single-pass programmable multitexturing. On the other hand, texture shaders are simple enough that mapping them to a given architecture will not be a monumental task. In fact, simulating texture shaders using multipass techniques on top of the existing compositing, multitexturing, and pixel transfer operators available on current machines is feasible, with the single addition of signed compositing arithmetic

---

*To appear in SIGGRAPH/Eurographics Workshop on Graphics Hardware 1999.

†mmccool@cgl.uwaterloo.ca; Computer Graphics Lab, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1

‡heidrich@mpi-sb.mpg.de; Computer Graphics Group, Max Planck Institute for Computer Science, Im Stadtwald, 66123 Saarbrücken, Germany

and color representations. *Vice versa*, for backward compatibility existing multitexturing operations can be implemented using any hardware capabilities added to implement texture shaders.

Texture shaders are general enough that they can be used as an intermediate target language for higher-level shader compilers or object-oriented shader metaprogramming toolkits (for instance, using a "shader node graph" akin to the scene graph in OpenInventor). As with standard compilers, the use of an intermediate language decouples the high-level languages from the implementation, which permits a variety of high-level shader translators to map to a wide range of implementations. This permits the cost of developing a high-level shader language to be amortized over a wide product line. At the same time, compliance verification with the simple API proposed is more straightforward than with a high-level language.

Our proposal [26] has several parts. These parts are somewhat independent but are designed to work together in order to express the implementations of certain photorealistic rendering algorithms cleanly and efficiently.

First, we extend and generalize existing texture coordinate generation modes so internally generated or maintained geometric information (such as the normalized vector to the light source from a vertex) can be converted into texture coordinate information or shader parameters in useful ways. We also add tangent vectors to complete the specification of a surface coordinate system to complement the existing object and view space coordinate systems.

The texture coordinate generation extensions are very powerful in their own right and could significantly improve performance when rendering sophisticated lighting effects, yet they will be generally easier to implement than full texture shader support. In fact, on implementations that use the host CPU for texture coordinate generation and lighting, the new texture coordinate generation modes proposed can be added to the drivers of existing graphics accelerators.

Secondly, we extend and generalize multitexturing to support a directed acyclic graph (DAG) of blending operators; currently multitexturing supports only a fixed-length chain of such operators. At the root (output) of the DAG is the color assigned to a fragment destined for the frame buffer; at the leaves (inputs) are filtered texture lookups, the colors interpolated from vertex colors and/or hardware lighting, and possibly parameters passed in or generated from local geometric information using the texture coordinate generation modes and a texture coordinate pass-through mode. Signed arithmetic and scaling by powers of two is permitted within the shader DAG, with internal values clamped to $[-1, 1]$ and the output clamped to $[0, 1]$.

The shader definition includes a mechanism for dynamically binding texture coordinates and texture lookups. This mechanism enables reuse of texture coordinate generation and interpolation computations, which is especially useful when a software implementation of the geometry stage of the pipeline is used.

The shader DAG is specified at the API level as a postfix (reverse Polish notation) stack program operating on a stack of 4-vectors, with some support for infix notation.

It is unlikely that the texture shading DAG will actually *be* implemented as a stack machine, but rather as an optimized parallel VLIW or SIMD machine, or even using hidden multipass rendering. We sketch a few possibilities in Section 6. However, the abstract stack machine permits the removal of variable names in the shader and permits modular metaprogramming. These features can be exploited by higher-level shader metaprogramming software toolkits.

2 Prior Art

Several researchers have developed multipass techniques for generating high-quality images using the operations available in contemporary graphics hardware. The effects covered by these techniques include bump mapping [27], normal mapping [19], and reflections off planar [7] and curved reflectors [12, 17, 29]. The traditional local illumination model used by graphics hardware can also be extended to include shadows (using either shadow maps [35] or shadow volumes [7]), arbitrary reflectance functions [20, 17], and complex light sources [15, 35].

Other researchers have developed methods for solving the global illumination problem with the help of graphics hardware. Keller [22] uses multipass rendering to generate indirect illumination for diffuse environments. Stürzlinger and Bastos [37] can render indirect illumination in glossy environments by visualizing the solution of a photon map algorithm. Stamminger *et al* [36] and Walter *et al* [39] place OpenGL light sources at virtual positions to simulate indirect illumination reflected by glossy surfaces. Heidrich and Seidel [17] use multiple rendering passes and environment maps to render global illumination solutions for non-diffuse environments.

Another useful class of techniques uses light fields; these can be either rendered directly [9, 24], used to illuminate other objects [15], or used to simulate realistic cameras and lens systems [18].

While it has been clearly demonstrated that high quality images can be achieved using multipass techniques, the performance of these algorithms can be disappointing, especially when used in combination. There are two reasons for this: hardware mismatch and multiplicative pass combinations.

2.1 Hardware Mismatch

Hardware mismatch results because contemporary hardware has been derived from empirical models of rendering. In contrast, photorealistic rendering algorithms are often physically derived and require higher dynamic range and precision than is usually provided in hardware. Compositing operations often clamp color values at inconvenient places in the computation. Current frame buffer arithmetic is also usually unsigned, which poses an additional problem for some algorithms.

Surmounting these difficulties is possible but usually requires extra operations, such as scaling and biasing color values in the frame buffer. These extra operations can result in a loss of precision and a severe performance penalty.

The hardware mismatch problem can, for the most part, be overcome with modifications of color representations (such as signed values and/or floating-point color representations) and the addition of a small number of new operations (such as scale factors greater than unity, perhaps limited to powers of two, in the compositing operators).

2.2 Pass Combination

More seriously, the organization of contemporary hardware permits only a relatively small amount of work to be accomplished on each pass.

Each new photorealisitic effect typically requires multiple passes. When combining effects, sometimes with some ingenuity (and loss of modularity) passes can be combined, but in the worst case the total number of passes is the *product* of those required for each individual effect.

Consider, for example, the combination of multipass planar reflections [7] with separable lighting models/BRDFs [17, 20]. A maximum reflection depth of k requires k passes; in each pass texture maps representing reflected images must be generated using the texture maps generated in the previous pass. However, if n passes are needed to compute the local lighting, these n passes must be used for *each* of the k passes required for reflectance, resulting in nk passes in total.

Similar multiplicative effects can be observed when combining multipass algorithms for simulating lens systems [11, 18], shadows cast from multiple light sources, and so on. While the cost of individual multipass algorithms is usually tolerable, the expense of their combination explodes.

2.3 Shading Languages

Shading languages, such as Pixar's RenderMan shading language, [13, 38] can be used for more than just specifying local lighting models. Since shading programs assign the color to a surface with a relatively arbitrary computation, and can use other sources of pretabulated information, they can also be used to render shadows, generate non-photorealistic "sketch" renderings for visualization and artistic purposes, and can be potentially used to integrate the results of global illumination solutions into a rendering.

Several attempts have been made to integrate shaders into interactive rendering. Some researchers have accelerated the software evaluation of shaders by precomputing parts of them [10], or by parallelizing the computations on a number of MIMD or SIMD processors; for example, see Perlin [31].

Hardware support for procedural shading has begun to appear in graphics accelerators. The most prominent example is the PixelFlow architecture [28, 30], which is in the process of being commercialized. In this architecture, several rendering units based on general-purpose microprocessors and specialized processor-enhanced memory run in parallel a program implementing a rendering pipeline on part of the scene database. Shaders are implemented using a SIMD array, with shader programs compiled from a high-level language (pfman) almost identical to Pixar's RenderMan shading language, except for the addition of fixed-point types.

Unfortunately, due to differences in demand, production volume, and position on the learning curve, SIMD arrays are likely to remain significantly more expensive per bit than conventional memory.

2.4 Sample-Based Approaches

The traditional rendering pipeline used by most contemporary graphics architectures has been extended recently by some new features, available through the OpenGL API, that can be used to improve the realism of rendered images.

The imaging subset that has been added to OpenGL 1.2.1 consists of color transformations and color lookup tables as well as scaling, biasing, and extended compositing operations. In combination with the "pixel texture" extension, which feeds rendered images back as per-pixel texture coordinates, it is possible to combine normal maps with environment mapping and interesting per-pixel reflection models [17]. Pixel textures can also be used to implement shadows and sample light fields [19].

However, the most widely available new feature at the low end is the ability to apply multiple textures to a single object in a single rendering pass, using a chain of blending operators. This can reduce the number of required rendering passes dramatically in many multipass rendering algorithms, including some of the examples discussed in the next section.

One of the central premises of this paper is that multitexturing, whether or not it is actually implemented in hardware, is at least a useful conceptual abstraction at the API level. However, more flexibility is needed in the specification of how samples from different texture lookups are combined. We will establish this through case studies of the expressions needed to compute several important rendering effects.

3 Rendering Effects Targeted

The following examples will be used to motivate the specific extensions we propose. We do not mean to imply, by using these examples, that texture shaders can *only* be used to implement these effects. However, each of the following is a very important visual phenomenon not well served by existing graphics hardware.

3.1 Local Lighting Models

Physically-based local reflectance can be expressed in terms of incoming radiance $L_I(\underline{\mathbf{x}}, \hat{\omega}_I)$ and outgoing radiance $L_O(\underline{\mathbf{x}}, \hat{\omega}_O)$ at each surface point $\underline{\mathbf{x}}$:

$$L_O(\underline{\mathbf{x}}, \hat{\omega}_O) = \int_\Omega f_R(\hat{\omega}_O, \underline{\mathbf{x}}, \hat{\mathbf{n}}(\underline{\mathbf{x}}), \hat{\mathbf{t}}(\underline{\mathbf{x}}), \hat{\omega}_I) L_I(\underline{\mathbf{x}}, \hat{\omega}_I) [\hat{\mathbf{n}}(\underline{\mathbf{x}}) \cdot \hat{\omega}_I]_+ \, d\hat{\omega}_I. \quad (1)$$

where $\hat{\omega}_I$ is an incoming light direction, $\hat{\omega}_O$ is the view direction, $[a]_+ = \max(a, 0)$, the integral is taken over the incoming hemisphere Ω relative to the solid angle measure, the factor $L_I(\underline{\mathbf{x}}, \hat{\omega}_I)[\hat{\mathbf{n}}(\underline{\mathbf{x}}) \cdot \hat{\omega}_I]_+$ is called the *irradiance* at $\underline{\mathbf{x}}$, and f_R is called the *bidirectional reflectance distribution function*, or BRDF.

Here we have explicitly shown the dependence on surface position $\underline{\mathbf{x}}$ and surface orientation, as specified by the normal vector $\hat{\mathbf{n}}(\underline{\mathbf{x}})$ and a tangent vector $\hat{\mathbf{t}}(\underline{\mathbf{x}})$, which in turn also depend on surface position $\underline{\mathbf{x}}$. The BRDF is actually a function of the coordinates of $\hat{\omega}_I$ and $\hat{\omega}_O$ *relative* to the position-dependent coordinate basis $\hat{\mathbf{n}}(\underline{\mathbf{x}})$, $\hat{\mathbf{t}}(\underline{\mathbf{x}})$, and $\hat{\mathbf{b}}(\underline{\mathbf{x}}) = \hat{\mathbf{t}}(\underline{\mathbf{x}}) \times \hat{\mathbf{n}}(\underline{\mathbf{x}})$. In the following we will take these dependencies as implied and write the BRDF as $f_R(\hat{\omega}_O, \underline{\mathbf{x}}, \hat{\omega}_I)$. There is also, of course, an implied dependence on wavelength/color in L_I, L_O, and f_R, which is not necessarily separable from the geometric dependency.

If L point sources are used (a gross approximation of realistic lighting situations), the incoming radiance is a sum of delta functions scaled by I_ℓ / r_ℓ^2, where I_ℓ is the intensity of the ℓth light source and r_ℓ is the world-space distance to the ℓth light source. Let $\hat{\omega}_{I,\ell}$ be a normalized direction vector pointing towards the ℓth light source. In this situation the reflectance integral reduces to

$$L_O(\underline{\mathbf{x}}, \hat{\omega}_O) = \sum_{\ell=1}^{L} f_R(\hat{\omega}_O, \underline{\mathbf{x}}, \hat{\omega}_{I,\ell}) \frac{I_\ell}{r_\ell^2} [\hat{\mathbf{n}}(\underline{\mathbf{x}}) \cdot \hat{\omega}_{I,\ell}]_+. \quad (2)$$

Usually, we generalize and use an (empirical) quadratic attenuation function $a_\ell(r_\ell)$ in place of just r_ℓ^2. This permits point sources to better approximate the attenuation properties of area sources. For convenience denote the irradiance from light source ℓ as E_ℓ, to give the following:

$$E_\ell = \frac{I_\ell}{a_\ell(r_\ell)} [\hat{\mathbf{n}}(\underline{\mathbf{x}}) \cdot \hat{\omega}_{I,\ell}]_+ \quad (3)$$

$$L_O(\underline{\mathbf{x}}, \hat{\omega}_O) = \sum_{\ell=1}^{L} f_R(\hat{\omega}_O, \underline{\mathbf{x}}, \hat{\omega}_{I,\ell}) \, E_\ell. \quad (4)$$

Unfortunately, even evaluating this (gross) approximation requires L evaluations of the 6-dimensional function f_R, assuming we encode each of $\hat{\omega}_O$, $\underline{\mathbf{x}}$, and $\hat{\omega}_I$ with two degrees of freedom.

If we store f_R in a tabulated form, we cannot use a naive approach; it simply takes too much space. If we remove some of the degrees of freedom, such as the dependence on surface position $\underline{\mathbf{x}}$ or on the local orientation of the surface (*i.e.* assume isotropic BRDFs), we potentially throw away interesting visual phenomena. While we might want to do this some of the time to save space when a BRDF *is* isotropic, we don't want to be forced to do it *all* the time.

A shift-invariant BRDF does not depend on surface position $\underline{\mathbf{x}}$. Elsewhere [20, 17] we have found that visually useful representations of shift-invariant BRDFs can be obtained using a reparameterized separable expansion:

$$(s,t,u,v) = \mathbf{P}(\hat{\omega}_O, \hat{\omega}_I) \quad (5)$$

$$f_R(\hat{\omega}_O, \hat{\omega}_I) \approx \sum_{i=0}^{N} g_i(s,t)\, h_i(u,v) \quad (6)$$

where $\mathbf{P}$ is a relatively simple-to-compute reparameterization, for example using the halfvector $\hat{\mathbf{h}} = \text{norm}\,(\hat{\omega}_I + \hat{\omega}_O)$ as one of the axes of a new coordinate system [20, 21, 34]. With a good parameterization, typically good approximations can be found with $N \leq 5$, and often even $N = 1$ gives good visual results. The separable decomposition is in effect a compressed representation, but an asymmetric one; it's relatively hard to find the right factors $g_i(s,t)$ and $h_i(u,v)$, but evaluation of the BRDF at a point from the compressed representation is trivial.

With the right reparameterization and texture-map representations of $g_i(s,t)$ and $h_i(u,v)$ (*e.g.* parabolic maps [16, 17]) interpolation of (s,t) and (u,v) gives Phong-shading like results, so the parameters (s,t,u,v) need only be computed at the vertices of polygons, as texture map coordinates; see Figure 6. Unfortunately, for $N > 1$, we need to store signed values in $g_i(s,t)$ and $h_i(u,v)$.

Dependence on $\underline{\mathbf{x}}$ can be reintroduced in various ways. For instance, we can assume that the glossy part of the BRDF is shift-invariant, but an additive diffuse term $d(\underline{\mathbf{x}})$ depends on $\underline{\mathbf{x}}$. This requires a reconstruction of the following form:

$$f_R(\hat{\omega}_O, \underline{\mathbf{x}}, \hat{\omega}_I) \approx d(\underline{\mathbf{x}}) + \sum_{i=0}^{N} g_i(s,t)\, h_i(u,v) \quad (7)$$

Alternatively, we could blend between two BRDFs using compositing. This might be useful to represent, for example, a chess board with the white squares using the BRDF of ivory and the black squares using the BRDF of slate, or to represent localized corrosion or scuffing of metal:

$$f_R(\hat{\omega}_O, \underline{\mathbf{x}}, \hat{\omega}_I) \approx (1 - \alpha(\underline{\mathbf{x}})) \sum_{i=0}^{N_A} g_{i,A}(s,t)\, h_{i,A}(u,v) + \alpha(\underline{\mathbf{x}}) \sum_{i=0}^{N_B} g_{i,B}(s,t)\, h_{i,B}(u,v) \quad (8)$$

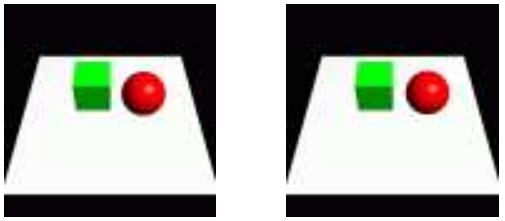
Straightforward extensions of this approach could be used for any other mixed-reflectance model with a small number of base BRDFs.

What is interesting about *all* the above expansions is that only multiplication and addition of color values are needed to reconstruct the reflectance; these operations can be implemented with existing compositing operations. If $N > 1$ is used (giving an asymptotic error of 0 as $N \to 0$) signed arithmetic is needed; however, biasing can be used at some loss in efficiency and precision.

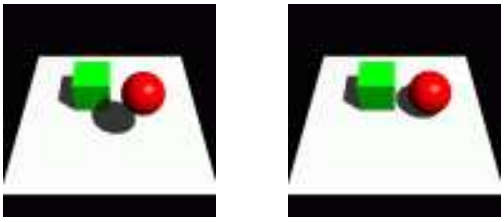
Compared to implementing Phong shading in hardware, the texture-map based shader approach provides much more flexibility (*any* BRDF) while exploiting existing capabilities in texture mapping.

3.2 Specular and Glossy Reflection

So far we have only considered local lighting models responding to point sources. But this is of course a gross oversimplification, and what we *really* want is a glossy reflection of the incoming radiance from the entire environment.



Without Shadows



With Shadows

Figure 1: *Are the spheres sitting on the ground planes, or not? Are the cubes? What are the relative distances of the cube and sphere in these two images? Are these images of the same scene or of different scenes? With shadows, all of these questions are easy to answer.*

Reflection of the environment can be approximated with environment maps. However, when the incoming radiance is filtered through the integral in Equation 2, what results is a blurry, or *glossy* reflection of the environment. If the BRDF is a constant shape, as in the Phong lighting model, then glossy reflection can be simulated simply by blurring the environment map.

In general, however, the shape of the BRDF depends on the relationship of $\hat{\omega}_I$ and $\hat{\omega}_O$ to the local surface frame. In particular, due to the Fresnel term and foreshortening of microgeometry, glancing reflectances are generally more specular than normal reflectances.

Better approximations can be obtained by:

1. Generating a representation of the desired BRDF in terms of a weighted sum of a set of parameterized basis functions.

2. Generating a set of filtered environment maps over the parameter space of the basis functions.

3. Selecting and blending the prefiltered environment maps using the coefficients and parameter settings found during the fit of the basis functions to the BRDF.

An example is shown in Figure 7, using only two environment maps: one very glossy, one a near mirror reflection. Glancing-angle reflectance is typically closer to being specular. This effect can be simulated by blending in more of the mirror environment map when $[\hat{\omega}_I \cdot \hat{\mathbf{n}}]_+ + [\hat{\omega}_O \cdot \hat{\mathbf{n}}]_+$ is small.

This is a *very* simple, *ad hoc* example. More than two environment maps would lead to more convincing results and better approximations of arbitrary BRDFs. Also, while in this case a 3D texture (or a MIP/pyramidal map with 3D texture map indexing) would be adequate, in general the basis fitting may require different texture coordinates for each term, and texture-mapped weighting functions that are non-zero for more than two texture maps.

For example, if a multiple lobe model is used [23] to fit the BRDFs, the direction, width, and radial profile of each lobe could potentially vary independently, and several environment maps could potentially contribute to a pixel's color, not just two as in 3D texture mapping with linear interpolation.

3.3 Shadows

Shadows are a very important depth cue, and require no special display hardware to present (unlike, for instance, stereo). Consider the demonstration in Figure 1. Despite this fact, almost no commercially available midrange hardware supports shadows directly.

There are (at least) two practical ways to implement general shadows with hardware acceleration: shadow maps and shadow volumes [1, 2, 3, 5, 8, 33, 35, 42]. There are several possible variants of these two algorithms and algorithms that are hybrids of them [25].

Any implementation of shadows should not consider merely the cost of casting a shadow from a single light source, but also the cost of integrating shadows into the lighting model and of casting shadows from multiple light sources in a single pass. This latter capability is especially important for implementing global illumination algorithms based on distributing a number of light sources throughout the scene [22].

Ultimately, the problem boils down to the integration of the information that a given point is in shadow with respect to a given light source with the shading model. For every surface point $\underline{\mathbf{x}}$ being rendered, let $s_\ell(\underline{\mathbf{x}})$ be 1 if the point is not in shadow relative to light source ℓ, and 0 if it is in shadow. To antialias shadow edges or to represent penumbra, $s_\ell(\underline{\mathbf{x}})$ can take on intermediate values. Now the local lighting expression should be

$$L_O(\underline{\mathbf{x}}, \hat{\omega}_O) \quad = \quad \sum_{\ell=1}^{L} f_R(\hat{\omega}_O, \underline{\mathbf{x}}, \hat{\omega}_{I,\ell})\, s_\ell(\underline{\mathbf{x}})\, E_\ell. \qquad (9)$$

Obtaining $s_\ell(\underline{\mathbf{x}})$ can be done using a shadow map, in which case $\underline{\mathbf{x}}$ must be backtransformed into a light-centric coordinate system. Alternatively, $\underline{\mathbf{x}}$ can be projected forward into screen space (something that must be done anyways) and an appropriate stencil bit or group of bits can be referenced, where the stencil values can be set with either the shadow volume algorithm or a forward-projection shadow map algorithm.

If the stencil planes can be treated as textures and appropriate bits can be extracted during evaluation of the above expression, then the second approach does not require a special purpose backtransformation.

What is needed to implement this is a texture-map lookup, possibly with a projective transformation of the position $\underline{\mathbf{x}}$, followed by generation of a scale factor $s_\ell(\underline{\mathbf{x}})$ either by using a comparison or by simply extracting bits from a (stencil) texture map value.

Note the most important aspect of Equation 9: it is a summation that occurs *after* we have evaluated L samples of the local lighting model, each of which may require its own summations and multiplications for reconstruction from a compressed form, and an illumination-modulating multiplication.

The current OpenGL multitexturing specification does not permit single-pass shading computations with the right form to handle even two light sources; we must use multiple passes and framebuffer compositing or the accumulation buffer.

4 Extensions Proposed

Our extensions have been developed in the context and using the conventions of the OpenGL 1.2.1 API and abstract machine. This is for concreteness and to directly address a very important graphics hardware API. Of course, these ideas are also applicable to other hardware APIs that use a similar graphics pipeline. Due to space limitations the extensions are presented only briefly and by example. A full specification is available elsewhere [26].

The goal of these extensions is to significantly reduce the number of rendering passes required for our target applications with relatively small changes to the existing pipeline and API. We are intentionally keeping the changes as simple as possible to make them easier to implement and verify.

4.1 Tangent Vectors

In order to support anisotropic reflection models, the concept of a surface orientation needs to be added to OpenGL. This should take the form of a new function `Tangent*()` analogous to `Normal*()`. The `Tangent*()` function would specify a current tangent vector that should become attached to vertex data in exactly the same way as normal information[1].

Tangent vectors should be transformed into eye coordinates by the model/view matrix. Normalization and scaling "hints" for handling normals should be cloned for the tangent. It is most practical to assume that it is the programmer's responsibility to ensure that the tangent and normal are initially perpendicular to each other in object space, so they remain perpendicular in eye space.

4.2 Texture Coordinate Generation

One of the problems with the current OpenGL specification is that useful information is computed at various places in the pipeline, but cannot be accessed. For instance, we would like to have access to the normalized light and view vectors generated at the vertices for lighting, and be able to convert this information directly into texture coordinates.

New texture coordinate generation modes should be provided that take the form of dot products between any of $\hat{\mathbf{x}}_V$, $\hat{\mathbf{y}}_V$, $\hat{\mathbf{z}}_V$, $\hat{\mathbf{x}}_O$, $\hat{\mathbf{y}}_O$, $\hat{\mathbf{z}}_O$, $\hat{\omega}_{I,\ell}$, $\hat{\omega}_O$, $\hat{\mathbf{t}}$, $\hat{\mathbf{n}}$, $\hat{\mathbf{r}}$, $\hat{\mathbf{b}}$, and $\hat{\mathbf{h}}_\ell$. A different texture generation mode should be specifiable for each of the four available texture coordinates.

The vectors $\hat{\mathbf{r}}$, $\hat{\mathbf{b}}$, and $\hat{\mathbf{h}}_\ell$ are generated internally:

$$\hat{\mathbf{r}} \quad = \quad 2\hat{\mathbf{n}}(\hat{\mathbf{n}} \cdot \hat{\omega}_O) - \hat{\omega}_O \qquad (10)$$

$$\hat{\mathbf{b}} \quad = \quad \hat{\mathbf{t}} \times \hat{\mathbf{n}} \qquad (11)$$

$$\hat{\mathbf{h}}_\ell \quad = \quad \mathrm{norm}\,(\hat{\omega}_{I,\ell} + \hat{\omega}_O) \qquad (12)$$

The vectors $\hat{\mathbf{x}}_V$, $\hat{\mathbf{y}}_V$ and $\hat{\mathbf{z}}_V$ are the axes of the viewing coordinate system; the vectors $\hat{\mathbf{x}}_O$, $\hat{\mathbf{y}}_O$ and $\hat{\mathbf{z}}_O$ are the axes of the object coordinate system.

Some reasonable approximations are available for $\hat{\mathbf{h}}_\ell$. The binormal $\hat{\mathbf{b}}$ in combination with $\hat{\mathbf{n}}$ and $\hat{\mathbf{t}}$ gives a complete orthonormal "surface" coordinate frame at each vertex[2]. The reflection vector, $\hat{\mathbf{r}}$, is already required for sphere maps.

Practically speaking, the API for this extension consists simply of a number of new enumerated types for the `TexGen` and `MultiTexGen` function calls.

Given these modes, some other useful texture coordinates can be synthesized. In particular, using an appropriate projective texture transformation matrix we can evaluate parabolic map coordinates, which give a low-distortion parameterization of the hemisphere.

These extensions offload texture coordinate generation and transfer from the host CPU, and are very powerful in their own right. In combination with even basic multitexturing and/or compositing, these extensions can be used to implement single-term separable BRDFs, microcylinder shading models for line segments, view-independent environment maps [16, 17], and attenuation from linear and triangular light sources—at least.

[1] The `EXT_coordinate_frame` extension actually supports both a tangent vector and a binormal, and is close to what is required.

[2] This coordinate frame is useful for bump mapping as well as anisotropic reflectances.

4.3 Multiple Primary Colors

In order to handle shadows correctly while still using hardware lighting for computing irradiance, the "primary" colors computed for each active light should be transferred *independently* to the texture shading stage.

Unfortunately, the primary colors cannot be summed before they have been properly masked by shadows, and this information will not be available until a shadow map (or the equivalent) has been accessed.

Since texture shaders can be used to implement lighting models, subsuming the function of hardware lighting, it would be reasonable to implement the transfer of primary colors in a way that does not require a great deal of extra bandwidth. For instance, transfer of a primary color to the texture shader could disable one of the texture access channels.

4.4 Extended Texture Types

In order to compute shadows and otherwise use depth and stencil conditional expressions in shaders, it is proposed that the internal texture types be extended to include depth and stencil types.

As stencil planes typically have only 8 bits, a stencil value s (converted to fixed point format) can be treated as an RGBA matte value $(1, 1, 1, s)$ for the purposes of blending in the texture shader.

The depth type needs specialized support since it should have at least 16 bits of precision, and preferably 32. Depth types may use an internal representation that can only be compared with other depths, not used in arithmetic. Fortunately, depth textures need not be filtered.

4.5 Multitexturing Stack Machine

The core of our proposal is a more flexible and "programmable" way to combine the results from multiple texture lookups.

The current specification of multitexturing only allows for blending together the results of a fixed number of textures and the fragment color in a fixed order (a chain). The color resulting from each blend operation is used as an input color for further blending with the color from the next texture. This fixed architecture is very limiting for many of our target applications. For example, it is not possible to compute the sum of two products of values looked up from a texture, and so we cannot do two-term separable expansions of BRDFs in one pass.

We propose to replace this fixed order of blending operations by a simple stack-based "programming language". Each entry of the stack contains a "color" with signed components in the range $[-1, 1]$. Each color on the stack actually consists of a set of 48 or more bits that can be used either as a high-precision RGBA value, 4D texture coordinate, or a single depth value.

The postfix notation of a stack machine enables a simple metaprogramming API and in particular permits the semantics of the host language to be used for modularity. Since there are no variable names, there is no potential for naming conflicts; stack machine "subroutines" can simply be wrapped in host language functions that emit them inline into an open shader definition.

Conceptually, the stack is manipulated by a sequence of instructions that place texture lookups and primary colors on the stack and then combine them with various operators. At the end of the computation, the color value at the front of the stack is clamped to $[0, 1]$ before being used as the fragment color.

A mechanism for the loose binding of texture coordinate vectors to texture lookups is also defined that would permit reuse of texture coordinates, as is common in multiple-term approximations.

It should be noted that despite the conceptualization of the stack machine as a sequential execution unit, it need not be implemented this way, as we outline in Section 6.

Definitions of shader programs are surrounded by `BeginShader(uint` i`)` and `EndShader()` statements. The parameter i passed to the `BeginShader(`i`)` statement defines the "name" of a shader object for later reference. The shader identifier 0 is reserved for "default OpenGL behaviour" but other identifiers are available for the programmer's use.

Shaders are made active using the function `Shader(uint` i`)`. Activating a shader also activates and binds associated texture objects and texture coordinate generation modes. This "shader object" interface is used in case any non-trivial compilation is needed to map a shader program onto a given implementation, and also to permit fast switching between shaders when drawing a scene.

Stack operations are divided into categories: stack manipulation (for instance, pulling items out of the middle of the stack to support shared subexpressions), arithmetic and blending, comparison of colors and depth values (comparisons return black or white for blending with further computations), logical operations, and component shuffling.

Access operations permit placing of texture lookups, texture coordinates, and interpolated primary colors on the stack to act as parameters for shaders. When the shader is defined a list of all the necessary parameters is formed and a schedule created for delivering them to the shader.

Finally, for cases when postfix notation is too verbose or inconvenient, a `ShaderExpr` function accepts a string specifying an infix expression that can initiate a number of blend and access operations.

4.6 Texture Coordinate Generation

In order to support shaders that require multiple texture lookups, support for programmable texture coordinate generation is also useful. For example, use of environment maps combined with bump mapping would require two stages of texture lookup, one to evaluate the bump function and one to sample the appropriate place in the environment map.

Shader programs with multiple texture accesses can be specified using the following function to bind together several primitive shader programs:

`ShaderTexGen(uint` s`, enum` k`, uint` n`)`

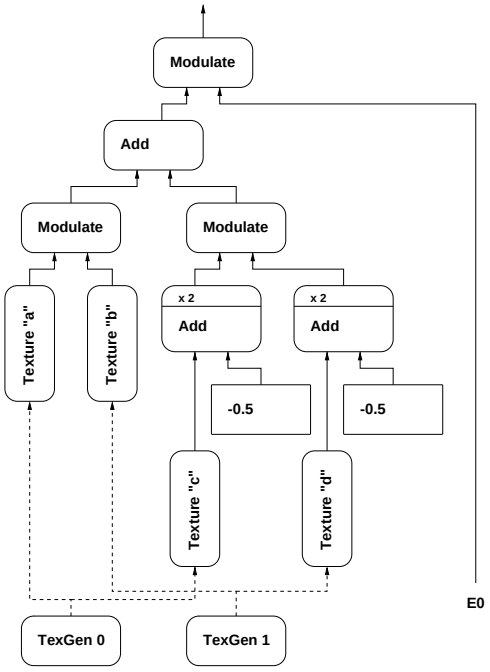
This function takes the n entries on the top of the stack after execution of shader s and sends them back to the texture generation unit, binding these results to the texture coordinate identifiers k through $k + n - 1$.

Evaluating and binding all texture coordinates at once ensures that in packetized implementations, all the information for the next shader will be available simultaneously, so no buffering will be required at the input to the texture lookup units.

5 Example

In this example we show how the proposed extensions can be used to implement one of the targeted rendering effects: multiple-term separable BRDF approximations.

Consider the illumination of an object with a two-term separable expansion of a BRDF with the irradiance evaluated using the existing hardware lighting model. The required shader expression is shown in Figure 2 along with a shader program which implements it. Note that expression requires the sum of two products with negative values in the second product; neither operation can be implemented in a single pass with the existing multitexturing facility.




```
BeginShader(1);
  //--- Establish base of light and coord indices
  ShaderBaseLight(LIGHT0);   // (actually, is default)
  ShaderBaseParam(TEXTURE0); // (actually, is default)

  //--- Bind texture object a to coords 0, b to coords 1
  ShaderTexture(a,0); // a(0)
  ShaderTexture(b,1); // b(1), a(0)
  //--- Multiply a and b
  ShaderModulate();   // b(1)*a(0)

  //--- Bind texture object c to coords 0
  ShaderTexture(c,0); // c(0), b(1)*a(0)
  //--- Convert c over [0,1] to C over [-1,1]
  ShaderConstant1(-0.5);
  ShaderShift(1);
  ShaderAdd();        // C(0), b(1)*a(0)
  ShaderShift(0);

  //--- Bind texture object d to coords 1
  ShaderTexture(d,1); // d(1), C(0), b(1)*a(0)
  //--- Convert d over [0,1] to D over [-1,1]
  ShaderConstant1(-0.5);
  ShaderShift(1);
  ShaderAdd();        // D(1), C(0), b(1)*a(0)
  ShaderShift(0);

  //--- Multiply C and D
  ShaderModulate();   // D(1)*C(0), b(1)*a(0)
  //--- Add the two products
  ShaderAdd();        // D(1)*C(0)+b(1)*a(0)

  //--- Multiply irradiance and reflectance; scale
  ShaderLighting(0);  // E0, D(1)*C(0)+b(1)*a(0)
  ShaderShift(3);
  ShaderModulate();   // (E0*(D(1)*C(0)+b(1)*a(0)))<<3
EndShader();
```

Figure 2: *Example expression tree and shader.*

6 Implementations

In this section we briefly consider the implementation of shaders, including a simple preliminary "plausibility" design for single-pass shading and some global architectural consequences.

6.1 Global Architecture

A typical high-level architecture for a graphics accelerator is shown in Figure 3. In this architecture we have shown a single memory, as opposed to separate framebuffer and texture memory. The units which this paper is primarily concerned with, the texture coordinate generation unit and the texture shader, are shown with bold outlines.

6.2 Texture Coordinate Generation

Despite their relative simplicity, the new texture coordinate generation modes proposed are crucial for making local geometric information available to shader programs, and would be a useful addition even without programmable multitexturing. As they require only dot products between normalized and unnormalized vectors that already must be computed at the vertices of primitives, a high-performance pipelined implementation is straightforward.

6.3 Managing Texture Access Latency

Typically, the texture lookup units have long latencies, since texture memory accesses are pipelined and filtering and interpolation operations must be performed. They may also have *variable* latencies, as a texture cache may be used, and if a multibank memory is used bank conflicts may occur.

Shader programs do not *request* texture samples; due to the long texture access latency, texture samples must be *scheduled* to arrive at the shader at the right time. It is also necessary for all texture samples from multiple textures to arrive at the shader together. For instance, if a cache miss occurs on one texture channel, stalling all channels to keep them synchronized may be required. Of course, these are concerns in existing multitexturing implementations as well.

The additional concern that texture shaders introduce is that complex shaders may require the use of more texture samples than there are parallel units available. In this case sequential texture accesses must be scheduled, and if one of the accesses in a texture sample "cluster" is stalled, all must be delayed to synchronize arrival at the shader. Alternatively, the shader compiler can decompose shaders that have too many inputs into multiple passes.

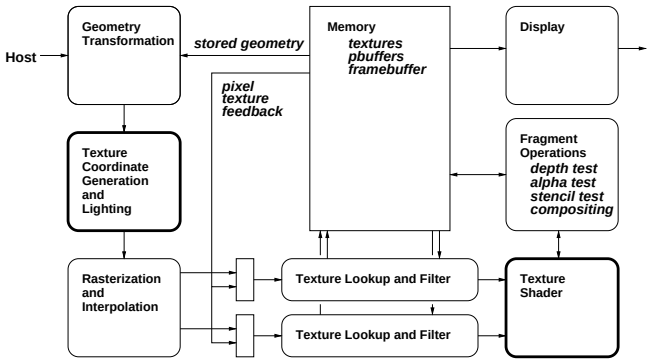


Figure 3: *Global accelerator architecture.*

6.4 Multipass Implementation

Texture shaders can be simulated, if necessary, using multipass algorithms. Basically, pbuffers can be used to hold intermediate results of the computation, with compositing and texturing operations used to implement each texture shader operation.

Both color storage formats and the compositing and texturing operations must be extended to support high-precision signed arithmetic, but this is a relatively small change that does not require rearchitecting existing designs. This implementation possibility permits a single API to cover machines both with and without multitexturing capabilities.

The chief disadvantage of the multipass approach is that even after optimization of the shader program, a great deal of memory may be required, particularly since high precision may be required in the pbuffers. This limits the complexity of the shaders than can be implemented. Furthermore, if a scene has multiple shaders that each cover a small part of the display area, then memory utilization will be low unless tight bounding boxes around the pixels affected by the shader can be built.

If a processor-enhanced memory is used, as in PixelFlow, similar problems arise, except the memory costs more and so the memory allocation problems are more acute.

Therefore, the multipass approach will be most useful in domains such as industrial design, where the number of shaders in the scene is low and shader coherence is high. The multipass approach is also useful as a fall-back in case shaders get too complex for a single pass. However, the addition of multitexturing and texture shader support should greatly reduce the number of passes required when only moderately complex shaders are required.

6.5 Single-Pass Texture Shaders

A literal sequential implementation of the conceptual stack machine would be too slow to be useful. We could replicate stack machines or pipeline a single stack machine by replication. However, implementing a sequential stack machine efficiently is difficult, and the pipelined approach results in large busses between pipeline stages and low utilization of the functional units.

An alternative is to reconstruct the shader expression and then map it onto a different implementation architecture, such as an array processor, or a specialized SIMD or MIMD processor. These need not be general-purpose units; in particular, branches conditioned on data values are not necessary, and this can simplify implementation.

An example is shown in Figures 4 and 5. Here a shared bus is used to deliver texture samples to a small number of parallel processing units. The register units are organized in banks so multiple texture samples can be loaded simultaneously.

The processing units are all executing the same instruction stream. However, because the instruction stream is pipelined, they are out of step. Texture samples (or interpolated primary colors, or texture coordinates passed through from the rasterizer) are delivered to processing units in a round-robin order using a shared bus as each processing unit reaches the beginning of another execution of the current shader program. When the results are ready they are sent to a shared output bus for delivery to the next stage.

If this architecture results in too many wires, the result and input buses can be combined, and perhaps the inputs can be serialized onto a single bus. To initialize a shader, constants can be preloaded into the PEs using the texture input busses. Since the instruction memory is shared, it can be large enough to hold several shader programs, to minimize the time required to switch between them.

If the shader programs are small, then output can be generated on every clock. If every operation takes one clock cycle, then with four shaders three operations can be executed before the next input is ready. A tree with four leaves has three internal nodes, each

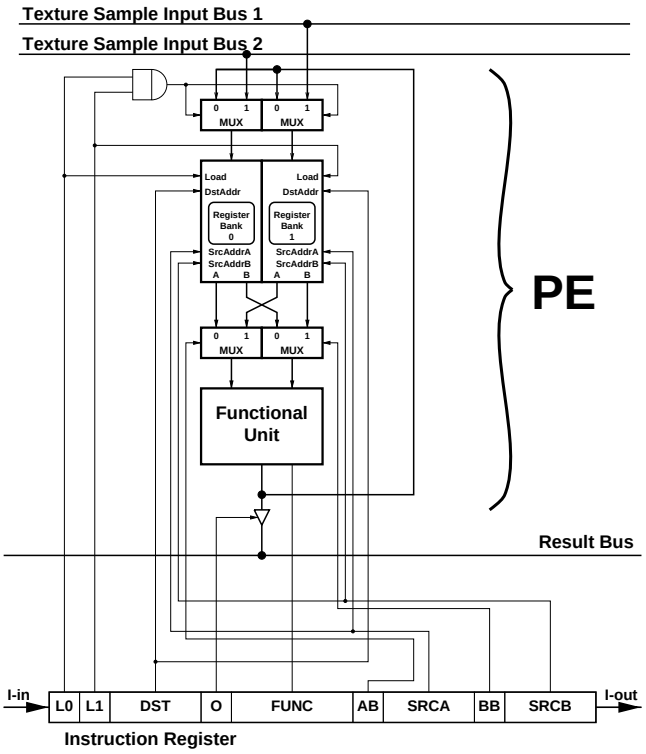


Figure 4: *One processor element for the staggered-SIMD approach to the implementation of texture shaders.*

representing an operation, so this permits shader trees with two constants and two space-variant operands to execute at full speed.

More complicated shaders with the same number of inputs are possible, for example if analytic evaluation of a tone operator is used, or if a function is synthesized as a polynomial. In this case the output rate should degrade gracefully. On the other hand, if a shader requires more inputs, they can be delivered in additional round-robin passes, although the rasterizer now has to schedule and reorder texture accesses over fixed-size groups of pixel fragments and the texture access units need to synchronize the arrive of each group. For this and other reasons, it may be simplest to implement multitexturing using sequential texture access using only a single texture lookup unit; in this case, the texture shader would require only a single input bus, and high performance would be achieved by having several parallel texture lookup and shader pipelines.

If some operations take multiple cycles to complete, variations on this architecture may be necessary. If we assume the functional units can still issue one operation per cycle, then we can attempt to schedule the operations specified by the shader tree to maximize throughput. As shader programs are small, this can be difficult. Two observations can be made:

1. Shader coherence can be assumed, so shader execution cycles can be unrolled and overlapped. Fewer parallel operations are typically available at the end of a shader cycle and more at the beginning. Therefore while the previous shader completes and scheduling becomes more difficult, the arguments for the next shader can be loading and early operations for the next shader begun.

2. We have shown each PE as a vector machine, with each register holding a four-vector and the functional unit operating in parallel on all four components. We could use an 8-bank

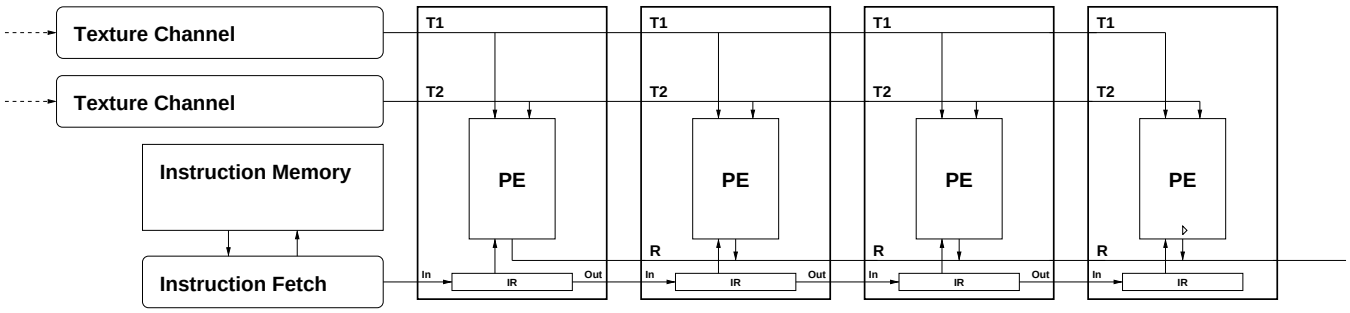


Figure 5: *A staggered-SIMD architecture for execution of texture shader programs.*

register file instead, and use separate specification of the operations on each component. This provides four times as many operations for each shader cycle, making scheduling easier. This flexibility is required anyways; for instance, computing a dot product requires multiplying two vectors and adding the components, and texture coordinate generation may also require irregular computation. Furthermore, some savings can be made for computations that involve only RGB or greyscale values and not RGBA values.

With separate components, we can use either a VLIW parallel functional unit or a simpler sequential unit and more PEs. The more PEs we add, the more we depend on shader coherence. If texture packets also carry pixel destination addresses and the downstream fragment processor is capable of routing fragments to different pixels, then we do not have to assume shader spatial coherence, and can depend merely on temporal coherence.

6.6 Texture Coordinate Feedback

In order to use shader programs to generate texture coordinates and write shaders that permit multiple texture accesses, the output of a shader needs to be fed back to the texture lookup unit(s).

There are two ways to accomplish this: an additional feedback channel that can feed back additional texture lookup requests to the start of the texture lookup units, or a multipass approach using pixel textures. The packet approach requires that a new shader be selected on a per-fragment basis, which inhibits some of the coherence-based optimizations mentioned above.

Therefore, it probably makes the most sense to use a multipass approach to implement texture coordinate feedback. It might be interesting to permit writing to several pbuffers at once, so multiple texture coordinates can be generated at once and more operations are available for scheduling. This requires generalizing the concept of "pixel address" to include a pbuffer address, and can be implemented using additional result output cycles for each shader program.

7 Conclusions

A procedural shader extension to the OpenGL API has been presented based on a generalization of multitexturing. This extension was motivated by a number of rendering problems that currently require several composition passes, but whose exact analytical form does not quite match that of multitexturing. We have sketched how a hardware implementation of these texture shaders could be implemented in a way that preserves pixel throughput for simple shaders, and degrades gracefully for more complex shaders.

Further progress will require extending a software implementation of OpenGL and then trying to implement the algorithms discussed here on top of it.

Interactive computer graphics has been very fortunate in having an API, OpenGL, which mapped onto a wide range of implementations. Recently the portability of OpenGL has been degraded because several features, such as multitexturing, have not been available universally. In addition, it is clear that fixed-architecture multitexturing is very limiting.

The texture shader extension permits a simple hardware abstraction that can map in a straightforward way to a wide range of hardware architectures, including those without hardware multitexturing support or with only fixed multitexturing, and so is one possible way to resolve these conflicts.

Acknowledgements

We would like to acknowledge the help and advice of members of the Computer Graphics Lab at the University of Waterloo and the Computer Graphics Group at Universitat Erlangen, especially Jan Kautz, whose work (on separable representations of BRDFs and variable-glossiness reflection) motivated much of what is found in this paper. One of the reviewers of the draft of this paper suggested the staggered-SIMD approach. This research was sponsored in part by a research grant from the National Science and Engineering Research Council of Canada.

References

[1] P. Bergeron. Shadow Volumes for Non-Planar Polygons. In *Proc. Graphics Interface*, pages 417–418, May 1985. Extended abstract.

[2] P. Bergeron. A General Version of Crow's Shadow Volumes. *IEEE CG&A*, 6(9):17–28, September 1986.

[3] L. Brotman and N. Badler. Generating Soft Shadows with a Depth Buffer Algorithm. *IEEE CG&A*, 4(10):71–81, October 1984.

[4] Robert L. Cook. Shade Trees. In *Proc. SIGGRAPH*, pages 223–231, July 1984.

[5] F. Crow. Shadow Algorithms for Computer Graphics. In *Proc. SIGGRAPH*, volume 11, pages 242–248, July 1977.

[6] Kristin J. Dana, Bram Van Ginneken, Shree K. Nayar, and Jan J. Koenderink. *Columbia-Utrecht Reflectance and Texture Database.* `http://www.cs.columbia.edu/CAVE/curet/`.

[7] P. Diefenbach and N. Balder. Multi-Pass Pipeline Rendering: Realism for Dynamic Environments. In *SIGGRAPH Symp. on Interactive 3D Graphics*, pages 59–70, April 1997.

[8] H. Fuchs, J. Goldfeather, J. Hultquist, S. Spach, J. Austin, F. Brooks, Jr., J. Eyles, and J. Poulton. Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes. In *Proc. SIGGRAPH*, volume 19, pages 111–120, July 1985.

[9] S. Gortler, R. Grzeszczuk, R. Szelinski, and M. Cohen. The Lumigraph. In *Proc. SIGGRAPH*, pages 43–54, August 1996.

[10] B. Guenter, T. Knoblock, and E. Ruf. Specializing shaders. In *Proc. SIGGRAPH*, pages 343–350, August 1995.

[11] P. Haeberli and K. Akeley. The accumulation buffer: Hardware support for high-quality rendering. In *Proc. SIGGRAPH*, pages 309–318, August 1990.

[12] P. Haeberli and M. Segal. Texture mapping as A fundamental drawing primitive. In *Fourth Eurographics Workshop on Rendering*, pages 259–266, June 1993.

[13] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *Proc. SIGGRAPH*, pages 289–298, August 1990.

[14] X. He, K. Torrance, F. Sillion, and D. Greenberg. A Comprehensive Physical Model for Light Reflection. In *Proc. SIGGRAPH*, pages 175–186, July 1991.

[15] W. Heidrich, J. Kautz, Ph. Slusallek, and H.-P. Seidel. Canned lightsources. In *Rendering Techniques '98 (Proceedings of Eurographics Rendering Workshop)*, 1998.

[16] W. Heidrich and H.-P. Seidel. View-independent environment maps. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 39–45, 1998.

[17] W. Heidrich and H.-P. Seidel. Realistic, hardware-accelerated shading and lighting. In *Proc. SIGGRAPH*, August 1999. Accepted for publication. Preprint soon available from http://www9.informatik.uni-erlangen.de/Persons/Heidrich.

[18] W. Heidrich, Ph. Slusallek, and H.-P. Seidel. An image-based model for realistic lens systems in interactive computer graphics. In *Graphics Interface '97*, pages 68–75, 1997.

[19] W. Heidrich, R. Westermann, H.-P. Seidel, and Th. Ertl. Applications of pixel textures in visualization and realistic image synthesis. In *ACM Symposium on Interactive 3D Graphics*, 1999. Accepted for publication.

[20] J. Kautz. Hardware Rendering with Bidirectional Reflectances. Technical Report TR-99-02, Dept. Comp. Sci., U. of Waterloo, 1999.

[21] J. Kautz and M. McCool. Interactive Rendering with Arbitrary BRDFs using Separable Approximations. In *Eurographics Rendering Workshop*, June 1999.

[22] A. Keller. Instant radiosity. In *Proc. SIGGRAPH*, pages 49–56, August 1997.

[23] E. LaFortune, S. Foo, K. Torrance, and D. Greenberg. Non-linear approximation of reflectance functions. In *Proc. SIGGRAPH*, pages 117–126, August 1997.

[24] M. Levoy and P. Hanrahan. Light field rendering. In *Proc. SIGGRAPH*, pages 31–42, August 1996.

[25] M. McCool. Shadow Volume Reconstruction. Technical Report CS-98-06, University of Waterloo Department of Computer Science, 1998.

[26] M. McCool and W. Heidrich. Texture Shaders: OpenGL Extension Specifications. Technical Report CS-99-11, University of Waterloo Department of Computer Science, 1999.

[27] T. McReynolds, D. Blythe, B. Grantham, and S. Nelson. Advanced graphics programming techniques using OpenGL. In *SIGGRAPH 1998 Course Notes*, July 1998.

[28] S. Molnar, J. Eyles, and J. Poulton. PixelFlow: High-speed rendering using image composition. In *Proc. SIGGRAPH*, pages 231–240, July 1992.

[29] E. Ofek and A. Rappoport. Interactive reflections on curved objects. In *Proc. SIGGRAPH*, pages 333–342, July 1998.

[30] M. Olano and A. Lastra. A shading language on graphics hardware: The PixelFlow shading system. In *Proc. SIGGRAPH*, pages 159–168, July 1998.

[31] K. Perlin. An image synthesizer. In *Proc. SIGGRAPH*, pages 287–296, July 1985.

[32] P. Poulin and A. Fournier. A Model for Anisotropic Reflection. In *Proc. SIGGRAPH*, pages 273–282, August 1990.

[33] W. Reeves, D. Salesin, and R. Cook. Rendering Antialiased Shadows with Depth Maps. In *Proc. SIGGRAPH*, volume 21, pages 283–291, July 1987.

[34] S. Rusinkiewicz. A new change of variables for efficient BRDF representation. In *Eurographics Workshop on Rendering*, pages 11–23, June 1998.

[35] M. Segal, C. Korobkin, R. van Widenfelt, J. Foran, and P. Haeberli. Fast Shadows and Lighting Effects using Texture Mapping. In *Proc. SIGGRAPH*, volume 26, pages 249–252, July 1992.

[36] M. Stamminger, Ph. Slusallek, and H.-P. Seidel. Interactive walkthroughs and higher order global illumination. In *Modeling, Virtual Worlds, Distributed Graphics*, pages 121–128, November 1995.

[37] W. Stürzlinger and R. Bastos. Interactive rendering of globally illuminated glossy scenes. In *Rendering Techniques '97*, pages 93–102, 1997.

[38] Steve Upstill. *The RenderMan Companion.* 1990.

[39] B. Walter, G. Alppay, E. LaFortune, S. Fernandez, and D. Greenberg. Fitting virtual lights for non-diffuse walkthroughs. In *Proc. SIGGRAPH*, pages 45–48, August 1997.

[40] G. Ward. Measuring and modeling anisotropic reflection. In *Proc. SIGGRAPH*, pages 265–272, July 1992.

[41] G. Ward. Towards More Practical Reflectance Measurements and Models. In *Graphics Interface '92 Workshop on Local Illumination*, pages 15–21, May 1992.

[42] L. Williams. Casting curved shadows on curved surfaces. In *Proc. SIGGRAPH*, volume 12, pages 270–274, August 1978.

Figure 6: *Separable approximations of reflectances for a single light source: anisotropic brushed metal [32], HTSG copper [14], velvet [6], vinyl [41], Ward's anisotropic model [40], and varnished wood [6]. The last two also use a texture mapped diffuse component.*

Figure 7: *Variable glossiness reflections simulated with superposition of filtered environment maps. Left: sharp reflection. Middle: uniformly blurred reflection (Gaussian lobe). Right: blend between blurry normal reflections and sharp glancing angle reflection.*