Graphics Hardware (2003)
M. Doggett, W. Heidrich, W. Mark, A. Schilling (Editors)

Automatic Shader Level of Detail

Marc Olano,* Bob Kuehne† and Maryann Simmons†

* University of Maryland, Baltimore County
† SGI

Abstract

Current graphics hardware can render procedurally shaded objects in real-time. However, due to resource and performance limitations, interactive shaders can not yet approach the complexity of shaders written for film production and software rendering, which may stretch to thousands of lines. These constraints limit not only the complexity of a single shader, but also the number of shaded objects that can be rendered at interactive rates. This problem has many similarities to the rendering of large models, the source of extensive research in geometric simplification and level of detail. We introduce an analogous process for shading : shader simplification. Starting from an initial detailed shader, shader simplification automatically produces a set of simplified shaders or a single new shader with extra level-of-detail parameters that control the shader execution. The resulting level-of-detail shader can automatically adjust its rendered appearance based on measures of distance, size, or importance, as well as physical limits such as rendering time budget or texture usage. We demonstrate shader simplification with a system that automatically creates shader levels of detail to reduce the number of texture accesses, one common limiting factor for current hardware.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image generation I.3.6 [Computer Graphics]: Methodology and Techniques I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism

Keywords: *Interactive Rendering, Rendering Systems, Hardware Systems, Procedural Shading, Languages, Multi-Pass Rendering, Level of Detail, Simplification, Computer Games, Reflectance & Shading Models*

1. Introduction

Procedural shading is a powerful technique, first explored for software rendering in work by Cook[11] and Perlin[35], and popularized by the RenderMan Shading Language[19]. A shader is a procedure written in a special purpose high-level language that controls some aspect of the appearance of an object to which it is applied. The term *shader* is used generically to refer to any such procedures, whether they compute surface color, attenuation of light through a volume (as with fog), light color and direction, fine changes to the surface position, transformation of control points or vertices, or any combination of these factors.

Recent graphics hardware can render procedurally shaded objects in real-time, through shaders defined in a low-level assembly language[4, 30, 40] or a high-level shading language[27, 33, 34, 36]. Even though the hardware is capable of rendering these shaders interactively, the number of texture units, total texture memory used, number of instructions, or other factors can affect overall performance or prevent a shader from running at all. Even on programmable PC graphics hardware, it is easy to exceed the hardware's abilities for rendering of a single object. Such shaders may be rendered using multiple passes through the graphics pipeline, but choosing the partitioning into passes is a dif-

Figure 1: *Shader simplification applied to a leather shader.*

ficult compilation task and the final number of passes has a direct impact on performance[5, 8, 34].

Consider the leather shader in Figure 1. With a bump map requiring three texture accesses per light and homomorphic BRDF factorization[28] requiring two texture accesses per light plus one additional texture access, it is complex enough to benefit from the several automatically generated simplification steps shown. An even more realistic leather shader might include multiple measured BRDFs for worn and unworn areas, bumps for the stitching, dust collected in the crevices, scuff marks, and changes in color due to variations in the leather. The options are limited only by the imagination and skill of the shader writer. But even though such a complex shader might look good applied to a single closely examined chair, it is overkill as you move away to see the rest of the room, all the other furniture in the room, other buildings, trees, cars and pedestrians — all using shaders of similar or greater complexity.

In this paper, we introduce the automatic generation of level-of-detail shaders (LOD shaders) from arbitrarily complex shaders. Our examples use SGI OpenGL Shader running on an SGI Octane[41]. From each input shader, our system automatically creates a single parameterized level-of-detail shader that can adjust the shading complexity (and thus number of rendering passes produced) based on a level-of-detail input parameter. The application sets the level parameter to control detail for the current viewing conditions and resource limits, thus allowing both interactive performance and high-quality shading of many objects in the same scene. The methods described in this paper could also be applied to produce a series of shaders for an application to select, or could be adapted for simplification of shaders on commodity graphics hardware as suggested by Vlachos[45].

1.1. Background

Automatic transformation of non-procedural surface appearance has been explored by a number of researchers. Kajiya was the first to pose the problem of converting large-scale surface characteristics to a bump map then lighting model[22]. Fournier used nonlinear optimization to fit a bump map to a sum of several standard Phong *peaks*[13]. Cabral, et al. addressed the conversion from bump map to BRDF through a numerical integration pre-process[7], and Becker and Max solved it for conversion from RenderMan-based displacement maps to bump maps and then to a BRDF representation[6]. Kautz approached the problem in reverse, creating bump maps to statistically match a chosen fractal micro-facet BRDF[24].

Fewer researchers have attempted to tackle automatic antialiasing of arbitrary shading language code. The primary form of antialiasing provided in the RenderMan shading language is manual transformation of the shader, relying on the shader-writer's knowledge to effectively remove high-frequency components of the shader or smooth the sharp transitions from an `if`, by instead using a `smoothstep` (cubic spline interpolation between two values) or `filterstep` (smoothstep across the current sample width)[12]. Perlin describes automatic use of blending wherever `if` is used in the shading code[12]. Heidrich et al. also did automatic antialiasing, using affine arithmetic to estimate the frequency and error while computing shading results[20].

Thus far, creation of shaders at multiple levels of detail for rendering speed or computational efficiency has been primarily a manual process. Goldman manually created multiple independently written level-of-detail versions of a fur shader for movie production[17]. Apodaca and Gritz describe several general options for manually creating shaders with multiple complexity levels[3]. Olano and Kuehne provided a set of building block functions with manually created levels of detail, so shaders using these building blocks inherit those levels of detail[32]. Guenter et al. automatically created *specialized shaders*, when only some shader parameters were expected to vary[18]. Expressions using other parameters were evaluated into textures.

While most of this prior work is in the context of off-line rendering systems like RenderMan, our work is set specifically within the context of recent advances in *interactive* shading languages. The first interactive shading system was a low-level assembly-like language for the Pixel-Planes 5 machine at UNC[38]. Later work at UNC developed a full interactive shading language on UNC's PixelFlow system[33]. Peercy and coworkers at SGI created a shading language that runs using multiple OpenGL Rendering passes[34]. The Real-Time Shading group at Stanford has created another high-level shading language, RTSL, that can be compiled into one or more rendering passes on SGI, NVIDIA, or ATI hardware[8, 36]. Many aspects of these research efforts appear in the several recent commercial shading languages and compilers, NVIDIA's Cg language, Microsoft's HLSL, ATI's Ashli, and the OpenGL shading language[27, 29, 5, 25].

Several aspects of interactive shading languages motivate the need for shader simplification and level-of-detail shaders. The languages they use share some features with traditional shading languages like RenderMan, but tend to be simpler, with operations that graphics hardware can and cannot do a major factor in their design. Hardware limits bound shader complexity and encourage the use of results precomputed into textures. Both of these factors make the simplification problem more tractable. Additionally, the desire to have the appearance of high-quality shaders on every object creates the need for shaders that can transition smoothly from high quality to fast rendering while maintaining interactive frame rates.

2. Automatic Simplification

Shader simplification automatically creates multiple levels of detail from an arbitrary source shader. Our simplification

process draws on two major areas of prior work — geometric simplification and compiler optimization.

Specifically, our shader simplification strategy is modeled after operations from the topology-preserving geometric level-of-detail literature. Schroeder and Turk both performed early work in automatic mesh simplification using a series of local operations, each resulting in a smaller total polygon count for the entire model[39, 44]. Hoppe used the collapse of an edge to a single vertex as the basic local simplification operation. He also introduced progressive meshes, where all simplified versions of a model are stored in a form that can be reconstructed to any level at run-time[21]. These ideas have had a large influence on more recent polygonal simplification work[26].

From this work we take several desired properties for our shader simplification algorithm. It should perform only local simplification operations for computational efficiency. Each operation should move monotonically toward the goal. Each simplification operation has an associated cost and the simplification of lowest remaining cost should be chosen at each step. The outline of our algorithm becomes:

```
for each candidate simplification site
  find simplification cost
while (simplifications remain)
  choose site with lowest remaining cost
  perform simplification
  re-compute costs for area local to site
```

The second area we draw on in developing our simplification algorithm is classic compiler *peephole optimization*[1]. Peephole optimization occurs toward the end of the compilation process when the program has already been reduced to blocks of simple instructions. The optimizer looks at small windows of instructions for certain patterns to replace.

Peephole optimization performs only local operations. If several optimizations overlap the optimizer will choose between them based on a set of costs. The golden rule for optimizations is to never change the program output. Shader simplification is in effect an optimization process but is one that may or may not break this rule. We can classify the simplifications into three categories:

Lossless: Obeying the strict compiler definition of optimization. This can be expanded to include some specializing shader-like optimizations[18] where certain non-constant parameters are assumed to be constant for the simplification. The geometric level of detail equivalent is simplification of highly tessellated, but flat regions of a model.

Resolution-specific lossless: Producing numerically or visually identical results but only at a specific resolution. This would include the majority of specializing shader simplifications and any others that evaluate results into textures. The equivalence is dependent on the texture resolution and minimum viewing distance. It also includes simplifications that replace textures with computed results, where the computed results fit a specific MIP-level of the texture. In geometric simplification, the equivalent is Simplification Envelopes or Appearance Preserving Simplification, with strong guarantees on geometric deviation[9, 10]

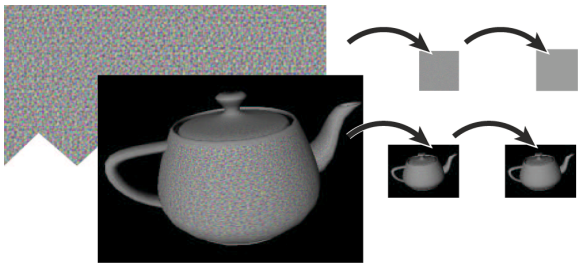


Figure 2: *Band-limited noise texture, noise **almost** blended away at a distance, and noise replaced with average value.*

Lossy: Not identical, but not noticeably or objectionably different. This includes many approximations that would never be considered for traditional optimization, but produce visually similar results at lower cost. Most geometric simplifications would fall in this category, as they minimize visual impact without making any guarantees, and assume slight changes in shape for distant objects are acceptable in exchange for interactive performance.

2.1. Simplifications

One of the most severe restrictions of current hardware is the cost of each texture access, with limits on either accesses or active textures per rendering pass in the tens at most.

Shaders that make heavy use of textures for precomputed expressions, for math and shading functions, and as actual textures can easily exceed these limits[8]. We have chosen the reduction of texture accesses as our simplification goal. Reduction by multiples of the single-pass hardware texturing limit provides an obvious speedup by reducing the number of passes required, but reductions by less than the single pass limit can also be beneficial as some hardware has higher rendering rates if fewer texture units are used, and fewer texture accesses indirectly leads to fewer operations and fewer active textures.

Many geometric simplifications use a single simplification rule, for example collapsing an edge to a point. We proceed using a choice of two simplification rules. The first is a lossy simplification that replaces a texture access with a simple non-texture-based approximation (Figure 2); the second is a lossless simplification that replaces one or more textures accesses and other operations with a single texture (Figure 3).

Texture Removal: Our first simplification rule clearly moves us toward the goal as it results in the direct removal of one texture access at each application. Our measure of

error for this simplification is the least-squares difference between the texture and non-texture approximation at each MIP-MAP level. The use of scale-based MIP filtering introduces a frequency and distance factor, while the least-squares error provides a measure of the contrast between pre- and post-simplification representations.

In the work presented here, we only approximate a texture by its average color. The most blurred level of a MIP map is just the average color, so applying this simplification switches to this constant color earlier than would be done by standard MIP filtering. For example, replacing

```
FB *= texture("marble.tx");
```

with

```
FB *= color(.612,.618,.607,1);
```

The least-squares error between texture and average color is the standard deviation of the texture, but we prefer the least-squares error interpretation since it generalizes more easily for future approximations. For example, an environment map may be well approximated by one or more light sources using the built-in Phong model, with light sources located at peaks of the environment map. The least-squares error between environment map and Phong lighting measures the error in this approximation, with lower error expected at larger MIP levels due to the closer fit of the approximation to the texture.

While the texture removal operation alone is theoretically sufficient to eventually remove all texturing operations from any complex shader, it does not always reduce texture uses as quickly as should be possible. The problem is not the local error metric, but with the global effect of the introduced error. Subsequent operations using the removed texture may amplify or attenuate the computed error. Directly computing this propagation of error can be done, as was shown in the sampling of RenderMan shaders by Heidrich et al.[20]. In our experience, global error measures were not necessary, even for shaders that did have fairly significant amplification of texture results like the watermelon in Plate 2(a). We attribute this to the common practice of writing complex shaders in layers[3].

Texture Collapse: Our second simplification rule is the collapse of one or more textures and the operations performed onto them into a single new texture. We guarantee to never increase the total texture accesses by including at least one existing texture in any set of collapsed operations.

Transformation of textures within a collapse, as illustrated in Figure 3, introduces resolution-specific error through resampling of the source textures into the collapse texture. In our current version, we support only lossless collapse (with no relative rotation or scaling of either texture).

By limiting ourselves to lossless collapse, texture collapse operations will happen immediately, at no additional cost.

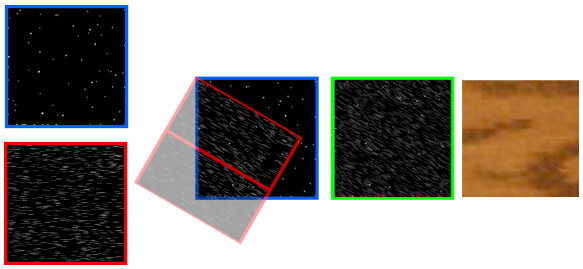


Figure 3: *An illustration of the collapse of two textures and the associated computations into a single texture. Left to right: the original dust (top) and scratch (bottom) textures; the textures as transformed and overlaid by the shader (the scratch texture is compressed, rotated and repeated, only two copies of the repeated texture are shown); the collapsed single-texture result; and an example of the collapsed texture in use as dust and scratch wood detail.*

However, texture removals at one scale may enable further collapses. For example:

```
FB =  texture("silk.tx");
FB *=  texture("cone.tx");
FB += color(0.1,0,0,0);
FB *= environment("flowers.env");
```

is reduced first by texture collapse creating a new temporary texture `loctx_0_silk.tx` (naming of generated textures is explained in Section 3.1), producing

```
FB = texture("loctx_0_silk.tx");
FB *= environment("flowers.env");
```

then by texture removal to

```
FB = texture("loctx_0_silk.tx");
FB *= color(.264,.238,.279,1);
```

then by a second collapse creating new temporary texture `loctx_1_loctx_0_silk.tx` producing

```
FB = texture("loctx_1_loctx_0_silk.tx");
```

and finally by texture removal to

```
FB = color(.111,.076,.090,1);
```

2.2. LOD Shader Representation

While each simplified block could be provided as a single stand-alone unit, we assemble all simplified blocks for a shader into a single unit, the LOD shader. We replace the full shader with an `if`. The true branch is the original shader while the false branch is the shader after one step of simplification. We iterate the simplification process on this false branch producing an LOD shader of the following form:

```
if(autoLOD < threshold0)
  original_shader
else
```

```
if(autoLOD < threshold1)
  simplified_once
else
  simplified_twice
```

Within OpenGL Shader, such parameter-based conditionals control which portions of the shader are executed by the hardware. The threshold levels monotonically increase with each level of simplification and provide a simple means to choose between levels of detail within the shader itself. The resulting LOD shader can be directly substituted as a replacement for the original shader. If `autoLOD` is not set, the original shader will be executed every time, but if `autoLOD` is set the appropriate level of detail will be used instead.

The existence of level-control parameters are the one aspect that distinguishes the interface to an LOD shader from other shaders. We control our LOD levels through the single parameter, `autoLOD`. This parameter represents the degree of texture scaling and is a function of object size, object parameterization, and object distance. As with geometric level of detail, other parameter choices are possible, including object importance, distance, size, time budget, or any of the hardware resources mentioned above. Several of these parameters could be combined into more complex conditionals selecting simplified blocks, collected into a single aggregate parameter, or controlled through an optimization function as done by Funkhouser and Séquin[15].

3. System Design

The bulk of this paper has focused on shader simplification and creation of levels of detail for arbitrary shaders. These capabilities must fit into the larger context of a shading system. In this section of the paper we will explore how our shading system architecture has been modified to allow both generation and usage of automatic level of detail.

3.1. Compile and Simplify

The first step for using any interactive shader is to compile it into a form executable by the graphics hardware. During the compilation we also perform any simplifications. Simplifications are only performed on shaders that define an *autoLOD* parameter. The existence of this parameter triggers use of the simplifier.

The simplification process also needs to know the size and contents of each texture, information not normally needed during shader compilation. Our system leaves the application in control of all aspects of texture loading and paging, so we require an application-provided image data *callback* function to get this data. The simplifier may call the image data callback during compilation to get a copy of texture data to analyze. Textures are identified to this callback by their string name. The callback can return the texture data or an error code indicating that the texture is unknown or dynamic and cannot be removed or simplified.

Texture collapse operations may require new textures to hold the combined textures and operations. Since the application is in charge of texture allocation and paging, we ask for image data for a *local texture* with a name beginning `loctx_%d_`. For example `loctx_1_stone` would be a copy of a texture named `stone` that the simplifier is free to write and replace. Later requests for `loctx_1_stone` should return the modified data (for analysis for texture removal or further collapse). Since a texture collapse may build on a previous collapse, these names may also build, so `loctx_5_loctx_1_stone` is a writable copy of `loctx_1_stone`.

3.2. Between Frames

The LOD shader may use different active textures on different frames depending on which level of detail is in use. This is not inconsistent with our goal of reducing texture accesses rather than global texture memory use, but many applications already use enough textures to require some form of texture paging. Adding an additional set of generated textures to that burden may be a problem for these applications.

We provide an optional snapshot function that an application can call between frames. The snapshot evaluates all run-time parameters and conditionals in the shader to provide a frozen version the application can store and use. In the process of building the snapshot, the application can find out exactly which textures will be used for a given set of run-time parameters, including the `autoLOD` setting. The application does not need to use the frozen shader that results if it only wants to know future texture usage. It can take a snapshot just one frame in advance or compute several speculatively to page textures for possible future views.

3.3. Draw and Shade

The final shaded object is drawn by the same mechanisms as any unsimplified object. If the application does not set the `autoLOD` parameter, it assumes the default value which triggers the full unsimplified shader. If the application does set an `autoLOD` value, the appropriate level of detail will be selected and executed. Applications using frozen snapshots must set their `autoLOD` values before taking the snapshot.

During the drawing of the shaded object, different textures may need to be loaded and bound to texture units for rendering. The draw action indicates which textures to load by calling an application provided *texture bind* callback function. Like the image data function, this function identifies textures by their string name. The texture bind callback also indicates the texture unit to bind to the texture (if the hardware supports multiple textures in each rendering pass). It is then the application's responsibility to load or page in the texture if necessary and prepare it for use. The texture names may be one of the names from the original shader source code or one of the generated `loctx` textures.

LOD	*Active*	*Accesses*	*Reduction*	*Speedup*
0	14	45	0.00	1.0
1	11	23	0.49	1.8
2	5	9	0.80	1.9
3	0	0	1.00	2.3

Table 1: *Results for test scene:* ***LOD****: A selection of simplification levels for this scene, from most detailed (0) to all constant colors (3).* ***Active****: number of active, unique textures.* ***Accesses****: number of texture accesses.* ***Reduction****: percentage of texture accesses removed.* ***Speedup****: framerate speedup factor.*

4. Results

We ran the automatic simplification on a number of shaders, all of which were written independently from our work on shader LOD. Once the user enables simplification by including the `autoLOD` parameter, the process is entirely automatic.

Results are shown in Plates 1 and 2, with performance results for Plate 1(a) shown in Table 1. As these results show, the automatically generated levels of detail are visually comparable to the fully detailed version at the appropriate viewing distances, at a significant reduction in texture accesses. Even further reductions could be achieved within the current framework by allowing more aggressive texture collapse.

5. Discussion

Using a single LOD shader that encapsulates the progression of levels of detail provides many of the advantages for simplified shaders that progressive meshes provide for geometry. In this section, we directly echo the points from Hoppe's original progressive mesh paper[21]. Not only does this place our current system in context, but it also suggests some logical extensions and more ambitious future work.

- *Shader simplification*: The LOD shader can be generated automatically from an initial complex shader using automatic tools. Our shader simplifier operates with the sole goal of reducing the number of texture accesses. Other valid simplification goals may include texture memory used, instruction count, balance between direct textures and dependent textures, or a weighted combination of these. Reducing texture accesses also indirectly reduces the number of active textures and instruction count, and so is relevant across a wide range of hardware.
- *LOD approximation*: Like a progressive mesh, an LOD shader contains all levels of detail. Thus it could include the shader equivalent of Hoppe's *geomorphs* to smoothly transition from one level to the next. Within OpenGL Shader, we have implemented continuous, per-pixel LOD at the cost of an additional pass that renders the object texture-mapped with a special MIP LOD texture that approximates the sampling rate of the shader[43]. The result is read back and used to set a per-pixel LOD level, that can also be used to smoothly blend between levels.
- *Selective Refinement*: Selective refinement for meshes refers to simplifying some portions of the mesh more than others based on current viewing conditions, encompassing both variation across the object and a guided decision on which of the stored simplifications to apply. Within OpenGL Shader, we can treat per-pixel LOD as noted above[43]. Programmable PC hardware does not realize any benefit from shading variations across a single object, but a single LOD shader will present a high quality appearance on some surfaces while using a lower quality for others, based on distance, viewing angle or other factors. The LOD shader could also apply certain simplifications and not others based on pressure from hardware resource limits, though our current implementation does not. For example, if available texture memory is low, texture-reducing simplification steps may be applied in one part of the shader while leaving more computation-heavy portions of the shader to be rendered at full detail.
- *Retargetability*: Retargetability is not found in mesh simplification. Since shading simplification can be built into a shading compiler, it gains the advantages of the compiler framework. Compilers consist of a sequence of modules that perform a simple operation on an intermediate representation of the shader. Since simplification can be dropped in as one or more modules in the chain, it is easy to add to existing shading compilers and easy to add new simplification modules. Further, since the shading compiler can be retargeted through multiple compiler *backends* to different shading hardware, it is easy to create simplifications for one hardware platform and use them on another.

Many of these points depend on the storage of an LOD shader. Our choice to combine all levels into a single LOD shader would work well for most of the points mentioned, with the added advantage that LOD shaders can easily be dropped in as replacements for their non-LOD counterparts.

6. Conclusions and Future Work

We have presented a method for automatic simplification of complex procedural shaders designed for use on graphics hardware. The resulting LOD shaders automatically adjust their level of shading detail for interactive rendering. We presented a general strategy for shader simplification, a specific example for reducing texture accesses, and a system that provides a shader compiler and shader simplification to an application.

6.1. Other Simplification Goals

The two simplifications discussed are specific to our goal to reduce the number of texture accesses. Future work may

optimize other simplification goals, including the previously suggested options of reducing total number of instructions or texture memory used. We have not fully explored simplification operations appropriate for these other simplification goals, but some directions inspired by prior research appear particularly promising.

Texture-based simplification for both shaders and geometry provides examples of ways to move computations into an increased number of textures. Guenter, Knoblock and Ruf[18] replaced static sequences of shading operations with pre-generated textures[18]. Heidrich has analyzed texture sizes and sampling rates necessary for accurate evaluation of shaders into texture[31]. In a related vein, texture-impostor based simplification techniques replace geometry with pre-rendered textures, either for indoor scenes as has been done by Aliaga[2] or outdoor scenes as by Shade et al.[42].

The body of BRDF approximation methods also suggests approaches to reduce computation at the cost of increased numbers of textures. Like shading functions, BRDFs are positive everywhere. Fournier used singular value decomposition (SVD) to fit a BRDF to sums of products of functions of light direction and view direction for use in radiosity[14]. Kautz and McCool presented a similar method for real-time BRDF rendering, computing functions of view direction, light direction, or other basis as textures using either SVD or a simpler normalized integration method[23]. McCool, Ang and Ahmad's homomorphic factorization uses only products of 2D texture lookups, fit using least-squares[28]. In a related area, Ramamoorthi and Hanrahan used a common set of spherical harmonic basis textures for reconstructing irradiance environment maps[37]. Many of these could be generalized to approximate blocks of shading code, which can be seen as a black-box producing a result from an arbitrary number of input variables.

6.2. Going Further

There are other promising overall research directions for shader simplification. Following the lead of texture-based simplification researchers like Aliaga and Shade et al., we could generate new textures for run-time parameter-dependent texture collapse or other simplification on the fly, warping them for use over several frames or updating when they become too different[2, 42].

Since rendering with LOD shaders will usually be accompanied by geometric level of detail, the two should be more closely linked. Cohen et al.[9], Garland and Heckbert[16] and others have shown that geometric simplification can be driven by appearance. Shader simplification should also be affected by geometric level of detail, with a trade-off between performing the same operation per-vertex or per-pixel depending on object tessellation.

Finally, our error metric measures the actual error in each replacement but provides no hard guarantees on the perceptual fidelity of our simplifications. Many geometric simplification algorithms have been successful without providing exact error metrics or bounds. However, algorithms such as simplification envelopes by Cohen et al.[10] provide hard bounds on the amount of error introduced by a simplification — guarantees that are important for some users. Further investigation is necessary to bound the error introduced by shader simplification.

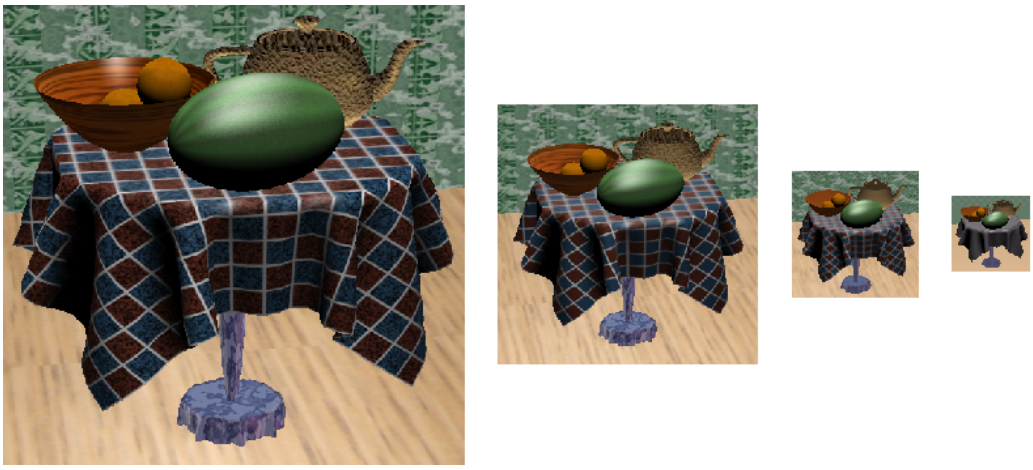
7. Acknowledgments

The leather BRDF was fit by homomorphic factorization by Michael McCool to data from the Columbia-Utrecht Reflectance and Texture Database. The car paint BRDF is also from Michael McCool, fit to data for Dupont Cayman lacquer from the Ford Motor Company and measured at Cornell University. The Porsche model was distributed by 3dcafe.com.

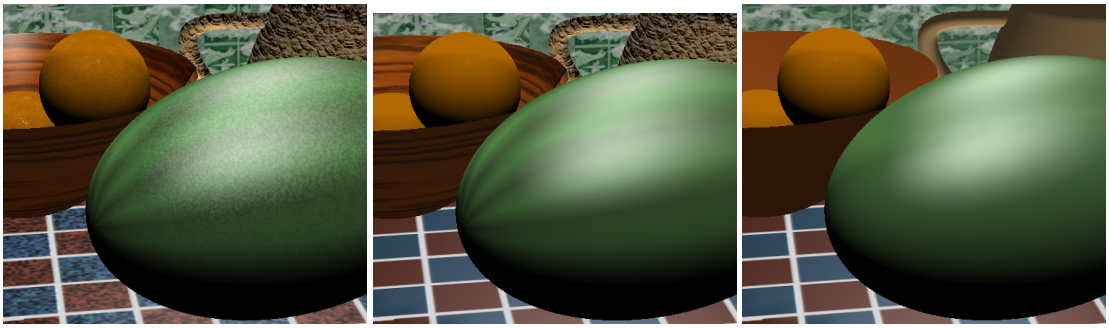
References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
2. D. G. Aliaga. Visualization of complex models using dynamic texture-based simplification. In *IEEE Visualization '96*, pages 101–106, October 1996.
3. A. A. Apodaca and L. Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann, first edition, 2000.
4. ATI. *ATI OpenGL Extensions Specifications*, 2001.
5. ATI. Ashli demo. http://www.ati.com, 2003.
6. B. G. Becker and N. L. Max. Smooth transitions between bump rendering algorithms. In *Proceedings of SIGGRAPH 93*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 183–190, August 1993.
7. B. Cabral, N. Max, and R. Springmeyer. Bidirectional reflection functions from surface bump maps. In *ACM Computer Graphics (Proceedings of SIGGRAPH 87)*, pages 273–281, July 1987.
8. E. Chan, R. Ng, P. Sen, K. Proudfoot, and P. Hanrahan. Efficient partitioning of fragment shaders for multipass rendering on programmable graphics hardware. In *Graphics Hardware 2002*. ACM SIGGRAPH / Eurographics, August 2002.
9. J. Cohen, M. Olano, and D. Manocha. Appearance-preserving simplification. In *Proceedings of SIGGRAPH 98*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 115–122, July 1998.
10. J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. Brooks Jr., and W. Wright. Simplification envelopes. In *Proceedings of SIGGRAPH 96*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 119–128, August 1996.
11. R. L. Cook. Shade trees. In *ACM Computer Graphics (Proceedings of SIGGRAPH 84)*, pages 223–231, July 1984.

12. D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling: A Procedural Approach*. Academic Press, second edition, 1998.
13. A. Fournier. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination*, pages 45–52, May 1992.
14. A. Fournier. Separating reflection functions for linear radiosity. In *Proceedings of Eurographics Workshop on Rendering*, pages 296–305, June 1995.
15. T. A. Funkhouser and C. H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of SIGGRAPH 93*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 247–254, August 1993.
16. M. Garland and P. S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In *IEEE Visualization '98*, pages 263–270, October 1998.
17. D. B. Goldman. Fake fur rendering. In *Proceedings of SIGGRAPH 97*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 127–134, August 1997.
18. B. Guenter, T. B. Knoblock, and E. Ruf. Specializing shaders. In *Proceedings of SIGGRAPH 95*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 343–350, August 1995.
19. P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *ACM Computer Graphics (Proceedings of SIGGRAPH 90)*, pages 289–298, August 1990.
20. W. Heidrich, P. Slusallek, and H. Seidel. Sampling procedural shaders using affine arithmetic. 17(3):158–176, July 1998.
21. H. Hoppe. Progressive meshes. In *Proceedings of SIGGRAPH 96*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 99–108, August 1996.
22. J. T. Kajiya. Anisotropic reflection models. In *ACM Computer Graphics (Proceedings of SIGGRAPH 85)*, pages 15–21, July 1985.
23. J. Kautz and M. D. McCool. Interactive rendering with arbitrary BRDFs using separable approximations. In *Eurographics Rendering Workshop*, June 1999.
24. J. Kautz and H. Seidel. Towards interactive bump mapping with anisotropic shift-variant BRDFs. pages 51–58. ACM SIGGRAPH / Eurographics / ACM Press, August 2000.
25. J. Kessenich, D. Baldwin, and R. Rost. *The OpenGL Shading Language*. 3Dlabs, Inc. Ltd., February 2003. Version 1.05.
26. D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan Kaufmann / Elsevier Science, 2003.
27. W. R. Mark, S. Glanville, and K. Akeley. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2003)*, 22(3), August 2003.
28. M. D. McCool, J. Ang, and A. Ahmad. Homomorphic factorization of BRDFs for high-performance rendering. In *Proceedings of SIGGRAPH 2001*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 171–178, August 2001.
29. Microsoft. *DirectX Graphics Programmers Guide*. Microsoft Developers Network Library, DirectX 9 edition, 2002.
30. NVIDIA. *NVIDIA OpenGL Extensions Specifications*, March 2001.
31. M. Olano, J. C. Hart, W. Heidrich, E. Lindholm, M. McCool, B. Mark, and K. Perlin. Real-time shading. In *ACM SIGGRAPH 2001 Course Notes*, August 2001.
32. M. Olano, J. C. Hart, W. Heidrich, and M. McCool. *Real-time Shading*. AK Peters, 2002.
33. M. Olano and A. Lastra. A shading language on graphics hardware: The PixelFlow shading system. In *Proceedings of SIGGRAPH 98*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 159–168, July 1998.
34. M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar. Interactive multi-pass programmable shading. In *Proceedings of SIGGRAPH 2000*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 425–432, July 2000.
35. K. Perlin. An image synthesizer. In *ACM Computer Graphics (Proceedings of SIGGRAPH 85)*, pages 287–296, July 1985.
36. K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of SIGGRAPH 2001*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 159–170, August 2001.
37. R. Ramamoorthi and P. Hanrahan. An efficient representation for irradiance environment maps. In *Proceedings of SIGGRAPH 2001*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 497–500, August 2001.
38. J. Rhoades, G. Turk, A. Bell, A. State, U. Neumann, and A. Varshney. Real-time procedural textures. In *1992 Symposium on Interactive 3D Graphics*, pages 95–100. ACM, March 1992.
39. W. J. Schroeder, J. A. Zarge, and W. E. Lorensen. Decimation of triangle meshes. In *ACM Computer Graphics (Proceedings of SIGGRAPH 92)*, pages 65–70, July 1992.
40. M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification*. SGI, 2002.
41. SGI. OpenGL shader. http://www.sgi.com/software/shader, 2003. Version 3.0.
42. J. Shade, D. Lischinski, D. H. Salesin, T. D. DeRose, and J. Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Proceedings of SIGGRAPH 96*, ACM Computer Graphics Proceedings, Annual Conference Series, pages 75–82, August 1996.
43. M. Simmons and D. Shreiner. Per-pixel smooth shader level of detail. In *Computer Graphics (Conference Abstracts and Applications SIGGRAPH 2003)*, 2003.
44. G. Turk. Re-tiling polygonal surfaces. In *ACM Computer Graphics (Proceedings of SIGGRAPH 92)*, pages 55–64, July 1992.
45. A. Vlachos. Designing a portable shader library for current and future APIs. In *Game Developers Conference Presentation*, March 2003. http://www.ati.com/developer/gdc/GDC2003-ShaderLib.pdf.



(a) A selection of LOD Levels for this scene (0-3) at typical viewing distances. Performance numbers in Table 1.



(b) Close-up of level 0: Highest level of detail.

(c) Close-up of level 1: Various noise and highlight details have been removed.

(d) Close-up of level 2: Extra detail on wall, bowl, watermelon and teapot removed.

Plate 1: *A simple scene showing the interaction of multiple automatically simplified level-of-detail shaders.*

(a) A slightly different watermelon shader

(b) A car shader

(c) A tile shader

Plate 2: *Individual examples of shader simplification.*