

Level-of-Detail Shaders

Marc Olano, Bob Kuehne *
SGI

Abstract

Current graphics hardware can render objects using simple procedural shaders in real-time. However, detailed, high-quality shaders will continue to stress the resources of hardware for some time to come. Shaders written for film production and software renderers may stretch to thousands of lines. The difficulty of rendering efficiently is compounded when there is not just one, but a scene full of shaded objects, surpassing the capability of any hardware to render. This problem has many similarities to the rendering of large models, a problem that has inspired extensive research in geometric level-of-detail and geometric simplification. We introduce an analogous process for shading, *shader simplification*. Starting from an initial detailed shader, shader simplification produces a 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.

CR categories and subject descriptors: I.3.3 [Computer Graphics]: Picture/Image generation — Display algorithms; I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism — Color, shading, shadowing and texture.

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 and Perlin [10, 35], and popularized by the RenderMan Shading language [20]. A shader is a simple procedure written in a special purpose

*email:{olano, rpk}@sgi.com

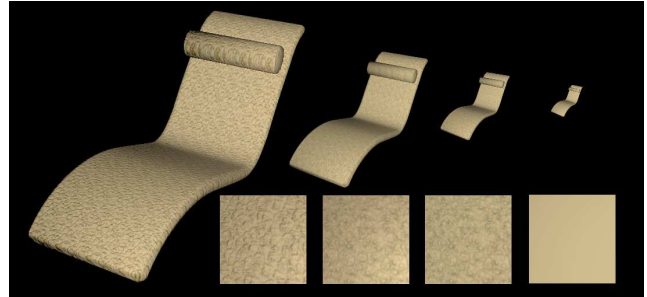


Figure 1: LOD shader upholstery a Le Corbusier chair.

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 procedures that compute surface color, attenuation of light through a volume (as with fog), light color and direction, fine changes to the surface position, or transformation of control points or vertices.

Recent graphics hardware can render simple procedural shaders in real-time [4, 5, 31, 33, 34, 36]. Shaders that exceed the hardware's abilities for rendering of a single object must be rendered using multiple passes through the graphics pipeline. The resulting multi-pass shaders can achieve real-time performance, but many complex shaders in a single scene can easily overwhelm any graphics hardware. Even for shaders that execute in a single rendering pass, the number of textures or combiner stages used can affect overall performance [31].

Consider a realistic shader for a leather chair. Features of this shader may include an overall leather texture or bump map, a couple of measured BRDFs (bidirectional reflectance distribution functions) for worn and unworn areas on the seat, bumps for the stitching, with dust collected in the crevices, scuff marks, changes in color due to variations in the leather, and potentially even more. Such a shader can provide a satisfying interactive rendering of the seat for detailed examination, but is overkill as you move away to see the rest of the room and all the other, buildings, trees and pedestrians using shaders of similar complexity. Figure 1 does not have all the features described, but with a bump map and measured leather BRDF it still exceeds current single pass rendering capabilities.

In this paper, we introduce level-of-detail shaders (LOD shaders) to solve the problem of providing both interactive performance and convincing detailed shading of many objects in a scene. A level-of-detail shader automatically adjusts the shading complexity based on one or more input pa-

rameters, providing only the detail appropriate for the current viewing conditions and resource limits. We present a general framework for creating a level-of-detail shader from a detailed source shader which could be used for automatic LOD shader generation. Finally, we provide details and results from our building-block based level-of-detail shader tools, where the general framework for shader simplification has been manually applied to building-block functions used for writing complex shaders.

1.1 Background

This work is directly inspired by the body of research on geometric simplification. Specifically, many of our shader simplification operations are 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, 41]. 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 [24]. These ideas have had a large influence on more recent polygonal simplification work ([16, 22, 25] and many others).

Many shader simplifications involve generating textures to stand in for one or more other shading operations. Guenter, Knoblock and Ruf replaced static sequences of shading operations with pre-generated textures [19]. Heidrich has analyzed texture sizes and sampling rates necessary for accurate evaluation of shaders into texture [32]. 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. [40].

We also draw on the body of BRDF approximation methods. 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 [13]. Kautz and McCool presented a similar method for real-time BRDF rendering, computing functions of view, light, or other bases as textures using either SVD or a simpler normalized integration method [27]. McCool, Ang and Ahmad's homomorphic factorization uses only products of 2D texture lookups, fit using least-squares [29]. In a related area, Ramamoorthi and Hanrahan used a common set of spherical harmonic basis textures for reconstructing irradiance environment maps [37].

This work is also directly derived from efforts to antialias shaders. The primary form of antialiasing provided in the RenderMan shading language is a 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 sam-

ple width) [11]. Perlin describes automatic use of blending where `if` is used in the shading code [11]. Heidrich and his collaborators also did automatic antialiasing, using affine arithmetic to compute the shading results and estimate the frequency and error in the results [23].

Finally, there have been several researchers who have done more ambitious shader transformations. Goldman described multiple versions of a fur shader used in several movies, though switches between *realfur* and *fakefur* were only done between shots [18]. Kajiya was the first to pose the problem of converting large-scale surface characteristics to a bump map or BRDF representation [26]. Along this line, Fournier used nonlinear optimization to fit a bump map to a sum of several standard Phong *peaks* [12]. Cabral, Max and Springmeyer 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]. More recently, Apodaca and Gritz manually created a hierarchy of filtered level-of-detail textures [3], while Kautz approached the problem in reverse, creating bump maps to statistically match a chosen fractal micro-facet BRDF [28].

This work is set within the context of recent advances in interactive shading languages, motivating the need for shaders that can transition smoothly from high quality to fast rendering. The first such system by Rhoades et al. was a relatively low-level language for the Pixel-Planes 5 machine at UNC [38]. This was followed by Olano and collaborators with 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 work presented here uses their OpenGL Shader ISL language as the format for both input shaders and LOD shader results.

There are many emerging options for assembler-level interfaces to hardware accelerated shading, including offerings by NVIDIA and ATI as well as a shading interface within DirectX [4, 5, 30, 31]. The shading group at Stanford, led by Kekoa Proudfoot and Bill Mark, created another high-level real-time shading language that can be compiled into either multiple rendering passes or a single pass using NVIDIA or ATI hardware extensions [36]. A group at 3DLabs, led by Randi Rost, is also spearheading an effort to create a high-level shading language for OpenGL version 2.0.

2 USING LOD SHADERS

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. The following directly echoes the points from Hoppe's original progressive mesh paper [24].

- *Shader simplification*: The LOD shader can be generated automatically from an initial complex shader using automatic tools (though as in the early days of mesh

simplification, these tools are not yet as automatic as we would like).

- *LOD approximation*: Like a progressive mesh, an LOD shader contains all levels of detail. Thus it can include the shader equivalent of Hoppe's *geomorphs* to smoothly transition from one level to the next.
- *Progressive transmission and compression*: The representation of a shader is much smaller than that of a mesh. Even relatively complex RenderMan shaders are typically only a few thousand lines of code. Shaders for real-time are seldom more complex than several tens of lines of code. Yet a scene with thousands of LOD shaders may still benefit by first storing and sending the simplest levels followed by transmission of the more complex 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. For an LOD shader these aspects are treated independently. Current 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 may also apply certain simplifications and not others based on pressure from hardware resource limits. 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.

Many of these points depend on the storage of an LOD shader. Starting from a complex shader we create a series of simplification operations to produce the most simplified shader, represented as another shader in the source shading language. This combined shader includes all of the levels within a single shading function with additional level control parameters. This provides several practical advantages as the LOD shader is indistinguishable, beyond its additional parameters, from a non-LOD shader. Since OpenGL Shader (and most other shading systems) set shader parameters by name, with default values for unset parameters, LOD shaders are easily interchanged with other shaders. For example, this can allow easy drop in replacement of the covering on a car seat, from a simple stand-in to a non-LOD vinyl shader, an LOD leather shader, or an LOD fabric shader.

The set of level-control parameters are the one aspect that distinguishes the interface to an LOD shader from other shaders. For interchangeable use the parameter set should be agreed upon by both the application and shader simplifier. These parameters are used within the LOD shader to switch

```
FB=diffuse();           FB=diffuse();
                        if (time<10)
FB*=texture("tex");    FB*=texture("tex");
```

a) basic block b) split blocks

Figure 2: Candidate blocks. a) a single basic block that could be simplified. b) blocks split by a conditional — will not be merged together

and blend between different levels as well as to define the ranges where each level is valid. As with geometric level-of-detail, parameter choices may include distance to the object, approximate screen size of the rendered object, importance of the object, or available time budget. For shading, we may also add budgets for hardware resource limits such as texture memory availability. Many of these parameters could instead be collected into a single aggregate parameter, or controlled through an optimization function as done by Funkhouser and Séquin [15]. All examples in this paper use a single parameter set using a distance metric.

3 SIMPLIFICATION FRAMEWORK

Shader simplification creates an LOD shader from an arbitrary source shader. We describe the simplification process in terms of four stages. First, identify candidate blocks of shader code. Second, produce a set of simplified versions of the candidate blocks. Third, associate level parameters with the simplified blocks, and finally assemble the result into an LOD shader. These stages can be repeated to achieve further simplification, where two or more simplified blocks can be combined into a single larger candidate block for another simplification run.

3.1 Finding Candidate Blocks

The first step toward creating an LOD shader is identifying blocks of shader code that are candidates for simplification. These are like edges for edge-collapse based polygonal simplification. Finding the set of candidate blocks in a shader is slightly more complicated than finding the set of edges in a model, but can be done with a static analysis of the original shader code.

A static analysis is one done before actual execution; it only has access to what can be inferred from the source code itself. In particular, results for conditionals and loops involving compile-time constants are known (*uniform* in ISL parlance), but not ones that might change at run-time (*parameter* in ISL). As a result, choosing a static analysis restricts simplification possibilities to what can be done within a basic block, without crossing a run-time loop or conditional (Figure 2).

Each block within the shader has some variables that are input to the computations within the block and others that are results computed by the block. Expressions within the block form a dependence graph with operations represented as nodes in the graph and variables as edges linking operation to operation. This graph can be partitioned into subgraphs

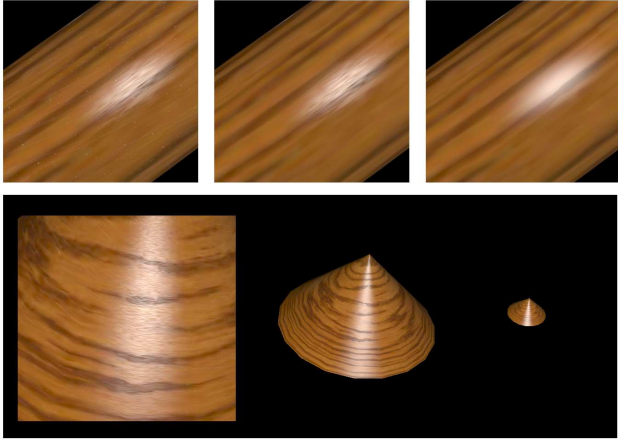


Figure 3: Removal of operations as contributions become imperceptible. Top row, left to right: Close-up of torus mapped with detail dust and scratch textures, with dust and scratches removed, with specular mask removed. Bottom row, left to right: image sequence of the wood applied to a cone with each removal displayed at its expected switching distance.

where each subgraph computes one block output or intermediate result. These subgraphs are the candidate blocks for simplification. Any basic block can be partitioned in many ways, and the choice of block partitioning is somewhat analogous to choosing edges for mesh simplification.

3.2 Simplifications

Each of the candidate blocks described above computes one result based on a set of inputs. The simplification operations on this block perform a local substitution of a simpler form in place of the original, producing equivalent output while keeping the form of the total shader the same. Simplifications that are not lossy are handled by the shading compiler optimization [19, 33, 34, 36].

Simplifications are chosen by matching a set of heuristic rules. While logically separate, the selection of simplification rules and partitioning of the basic block can be done at the same time using a tool like *iburg* [14]. *Iburg* is a compiler tools designed for use in code generation. Given a piece of code represented as an expression tree, it finds the least cost cover by a set of rules through a bottom-up dynamic programming algorithm.

Finding simplification rule costs for use by *iburg* requires analysis of input textures as well as the shader itself, and application of a rule may require generating a new derived texture as part of the LOD shader generation pre-process.

We classify these rule-based substitutions into one of four forms.

Remove: A candidate block that doesn't contribute enough anymore, or that consists of only high-frequency elements above the Nyquist frequency is replaced by a constant. This effectively removes the effect of portions of the shader that are no longer significant (Figures 3,4).

Collapse: A candidate block consisting of several opera-

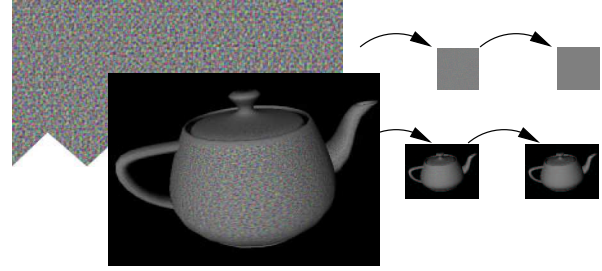


Figure 4: Band-limited Perlin noise texture, noise at a distance, and noise replaced with average value

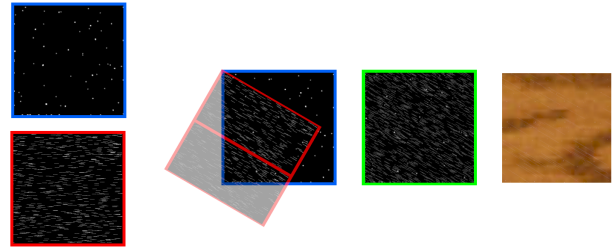


Figure 5: Collapsing two texture operations into a single texture. Left to right, the two initial textures, the two textures transformed and overlaid, the collapsed texture result, and an example of the collapsed texture in use as dust and scratch wood detail.

tions may be merged into a single new operation. For example, a coarse texture and a rotated and repeated detail texture can be combined into a single merged texture of a new size (Figure 5).

Substitute: A candidate block identified as implementing a known shading method may be replaced by a simpler method with similar appearance. For example, a bump map can be replaced by a gloss map to modulate the highlight intensity, or a simple texture map (Figure 6). A texture indexed by the surface normal is probably part of a lighting model, and depending on the contents of the texture, may be replaced by the built-in diffuse lighting model. Similarly, a texture indexed by the half angle vector ($\text{norm}(V + L)$ for view vector V and light vector L) is a candidate for replacement by one or more applications of the built-in Phong specular model. A texture can be replaced by a smaller low-pass filtered version of the texture and a constant representing the removed high-frequency terms.

Approximate: Approximation rules treat the candidate block as a general function to be approximated. They can theoretically be applied to any block, though not always as effectively as the application-specific rules.

While a variety of function approximation methods are possible, we have focused on ones developed for BRDF approximation [27, 29]. As these methods are texture-based, they are most useful when total texture usage is not the limiting factor. Two issues prevent our approximation rules from being more generally useful, though we believe they are aspects of the approximations we chose to explore and not all applicable function approximation methods.

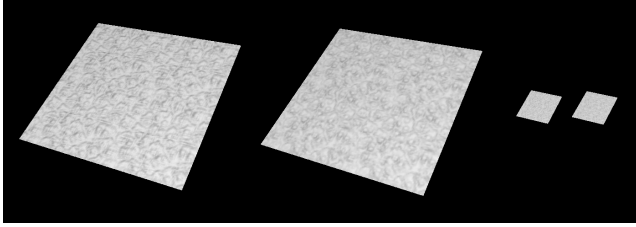


Figure 6: Replacing a bump map with a texture. Left to right, the original bump map, the bump texture at full scale, and the bump map and texture at the expected switching distance.

First, these approximations are based on a factorization into products or sums of products of functions of two variables that can be stored in a texture. In the right coordinate system, BRDFs are well suited to this factorization, usually requiring only one or two terms. Automatic simplification calls for automatic determination of a coordinate system. Arbitrary shading expressions can also be poorly suited to such a factorization in any coordinate system, allowing no acceptable approximation by the homomorphic factorization method, or needing so many SVD terms as to become more expensive than the original expression.

Second, the least squares or singular value decomposition problems are stated in terms of matrices with a number of rows and columns equal to the total number of texels in each approximating texture. Computing these textures rapidly scales to gigabytes, even for modest component texture sizes. Worse, we want to speculatively compute the approximations to evaluate their fitness. The original application to BRDFs limited the component texture sizes to 32x32 or 64x64 resulting in computations with 1024x1024 to 4096x4096 matrices.

3.3 Level Parameters

Selection of simplified versus unsimplified blocks is based on one or several level parameters. For example, switching from a band-limited noise texture to a constant value should happen when the changes in the noise texture are no longer visible (Figure 4). That point can be approximated based either on the distance or screen size of the object. The same transition can also be triggered by a lack of available rendering time, or a lack of available texture memory to store the noise texture.

To manage these different level parameters, we can associate a range for each parameter with each simplified block. Using the noise example above, a constant should be used instead of the noise texture whenever the available texture memory is less than the size of the texture, or there is not enough time to render another texture, or the expected mapping to screen pixels will blur the band-limited noise away.

3.4 Assemble

Given the simplified blocks and level parameter ranges, it is straightforward to assemble them with appropriate conditionals into an LOD shader. Rendering-metric level param-

Shader	Level 1	Level 2	Level 3
Plastic (Collapse)	36.4, 27.6	44.5, 34.4	—, —
Wood (Remove)	18.4, 11.6	18.9, 11.9	19.1, 64.3
Leather (Replace)	25.4, 14.1	43.7, 25.3	79.8, 64.3

Table 1: Result times for test LOD shaders on the 1772 triangle chair model performed on an SGI Octane MXE. Each table entry includes frames-per-second for a small window size, and a large window size with 4x the rendered pixels.

Shader	Level 1	Level 2	Level 3
Plastic (Collapse)	52.9, 33.8	68.2, 42.1	—, —
Wood (Remove)	20.7, 9.2	23.0, 10.0	25.2, 10.7
Leather (Replace)	30.7, 12.3	55.2, 22.8	140.9, 80.3

Table 2: Result times for test LOD shaders on a 3280 triangle draped cloth model consisting of 40 length-82 triangle strips, performed on an SGI Octane MXE. Each table entry includes frames-per-second for a small window size, and a large window size with 4x the rendered pixels.

ters, like distance or screen coverage, are shared by all blocks in the shader, each emitting a statement of the form

```
if(distance < low_threshold)
    do_simplified_block
else if(distance < high_threshold)
    do_transition_block
else
    do_original_block
```

For resource-accounting level parameters (e.g. available time or texture memory) the blocks are prioritized, and comparisons are emitted for the total consumed by this block and all higher priority blocks.

4 RESULTS

We have described a general theory of shader simplification. Our current results are a modest start within this framework. Specifically, we have produced a set of LOD-aware building block functions for shader construction. This style of shader writing is similar to Abram and Whitted’s graphical building-block shader system [1]. Example building-blocks include bump map, a BRDF model, Fresnel reflectance, or noise or turbulence textures with a lookup as used by Hart [21].

Our LOD blocks were created by manually following the steps described in our simplification framework: identify candidate blocks within a building block function, apply one of the simplification rules described in Section 3.2, associate it with a range of an aggregate level parameter, and create conditional blocks for the original code, transition code and simplified code. Despite the manual simplification, we call this semi-automatic because any shaders written using the building blocks, either knowing about level-of-detail or not, become LOD shaders by switching to the LOD building blocks.

Shader	Level 1	Level 2	Level 3
Plastic (Collapse)	9.2, 11.2	11.8, 14.0	—, —
Wood (Remove)	3.6, 5.3	4.1, 5.8	4.5, 6.5
Leather (Replace)	6.4, 8.8	14.7, 18.7	27.7, 35.7

Table 3: Result times for test LOD shaders on the 1772 triangle chair model performed on an SGI O2. Each table entry includes frames-per-second for a small window size, and a large window size with 4x the rendered pixels.

Shader	Level 1	Level 2	Level 3
Plastic (Collapse)	13.6, 15.9	18.2, 20.4	—, —
Wood (Remove)	4.9, 6.9	5.4, 7.6	6.0, 8.5
Leather (Replace)	8.1, 10.3	19.8, 23.9	40.3, 52.3

Table 4: Result times for test LOD shaders on the 3280 triangle draped cloth model performed on an SGI O2. Each table entry includes frames-per-second for a small window size, and a large window size with 4x the rendered pixels.

Tables 1–4 show LOD shader timing in frames per second for several sample LOD shaders. Each shader demonstrates several transitions of specific LOD simplification operations. The Wood shader used in these tests first removes an overlay scratch texture, then removes a specular masking operation, creating three levels-of-detail. Figure 3 shows the removal LOD sequence. The Plastic shader demonstrates the collapse simplification by taking two textures, each applied with its own transformation, and merging these two separate texture passes in a third texture. This resultant texture is then used to shade the object in a single texture for lower levels-of-detail as shown in Figures 5 and 7. The Leather shader demonstrates the replace simplification in the first level-of-detail by replacing a true bump map with a simple texture. The second level in the Leather removes the texture with a simple constant color. Results of this operation sequence are seen in Figure 9.

An overview of the performance results shows much what we would expect — that less detailed shaders result in faster overall rendering. However, as the different results indicate, the shading operations are not purely fill-limited, and rendering nearly 4x fewer pixels in certain cases results in only a modest performance improvement. As certain passes occur, the object’s geometry is also re-rendered, yielding a coupling between type of rendering passes constructed for a particular

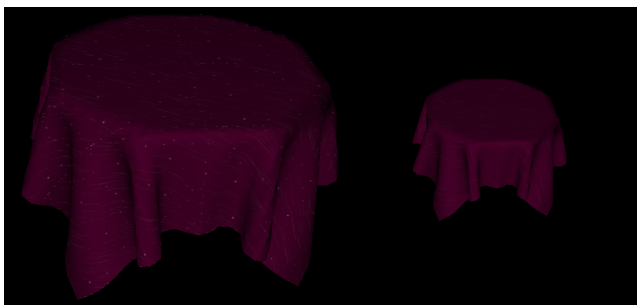


Figure 7: Plastic shader and cloth model.

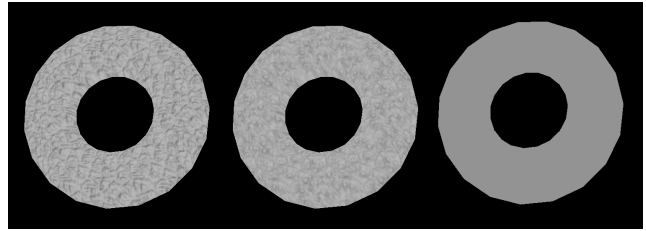


Figure 8: Two replace simplifications in a bumpy leather shader.

shader and that shader’s. This implies that LOD shaders can accomplish only part of the task, and should also be accompanied by geometric simplification.

5 CONCLUSIONS AND FUTURE WORK

We have presented LOD shaders: procedural shaders that automatically adjust their level of shading detail for interactive rendering. We also presented a general framework for shader simplification — the process of creating LOD shaders from an ordinary shader. This framework is sufficiently general to serve as a guide for manual shader simplification or as a basis for automatic simplification. Finally, we presented our results for semi-automatic shader simplification using manually generated shading function building blocks for SGI’s OpenGL Shader. These LOD shader building blocks implement the same functions as building blocks already provided with OpenGL Shader, but with added level-of-detail parameters to control aspects of their shading complexity.

In the future, we would like to create tools for fully automatic shader simplification. Our current simplification framework also only considers a static analysis of the shader for simplification. Following the lead of texture-based simplification researchers like Aliaga and Shade et al., we could generate new textures on the fly warping them for use over several frames or updating when they become too different [2, 40].

Logically, it should be possible to generalize our remove, collapse and substitution rules into a more widely applicable approximation rule form. Other function fitting methods should be tried to make the approximation rules more useful.

Since rendering with LOD shaders will usually be accompanied by geometric level-of-detail, they should be more closely linked. Cohen et al. Garland and Heckbert and others have shown that geometric simplification can be affected by appearance [8, 17]. Shader simplification should also be affected by geometric level-of-detail (e.g. whether per-vertex Phong shading is a good substitute for a texture-based illumination depends on how the object is tessellated).

Finally, we provide no guarantees on the 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. provide hard bounds on the amount of error introduced by a simplification [9], guarantees that are important for some users. Further investigation is necessary to bound the error introduced by shader simplification.

6 ACKNOWLEDGMENTS

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

We'd also like to thank Dave Shreiner for his helpful comments on the drafts paper.

References

- [1] ABRAM, G. D., AND WHITTED, T. Building block shaders. In *Computer Graphics (Proceedings of SIGGRAPH 90)* (Dallas, Texas, August 1990), vol. 24, pp. 283–288. ISBN 0-201-50933-4.
- [2] ALIAGA, D. G. Visualization of complex models using dynamic texture-based simplification. In *IEEE Visualization '96* (October 1996), IEEE, pp. 101–106. ISBN 0-89791-864-9.
- [3] APODACA, A. A., AND GRITZ, L. *Advanced RenderMan*, first ed. Morgan Kaufmann, 2000.
- [4] ATI. *Pixel Shader Extension*, 2000. Specification document, available from <http://www.ati.com/online/sdk>.
- [5] ATI. *Vertex Shader Extension*, 2001. Specification document, available from <http://www.ati.com/online/sdk>.
- [6] BECKER, B. G., AND MAX, N. L. Smooth transitions between bump rendering algorithms. In *Proceedings of SIGGRAPH 93* (Anaheim, California, August 1993), Computer Graphics Proceedings, Annual Conference Series, pp. 183–190. ISBN 0-201-58889-7.
- [7] CABRAL, B., MAX, N., AND SPRINGMEYER, R. Bidirectional reflection functions from surface bump maps. In *Computer Graphics (Proceedings of SIGGRAPH 87)* (Anaheim, California, July 1987), vol. 21, pp. 273–281.
- [8] COHEN, J., OLANO, M., AND MANOCHA, D. Appearance-preserving simplification. In *Proceedings of SIGGRAPH 98* (Orlando, Florida, July 1998), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 115–122. ISBN 0-89791-999-8.
- [9] COHEN, J., VARSHNEY, A., MANOCHA, D., TURK, G., WEBER, H., AGARWAL, P., JR., F. P. B., AND WRIGHT, W. Simplification envelopes. In *Proceedings of SIGGRAPH 96* (New Orleans, Louisiana, August 1996), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 119–128. ISBN 0-201-94800-1.
- [10] COOK, R. L. Shade trees. In *Computer Graphics (Proceedings of SIGGRAPH 84)* (Minneapolis, Minnesota, July 1984), vol. 18, pp. 223–231.
- [11] EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. *Texturing and Modeling*, second ed. Academic Press, 1998.
- [12] FOURNIER, A. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination* (May 1992), Canadian Information Processing Society, pp. 45–52.
- [13] FOURNIER, A. Separating reflection functions for linear radiosity. In *Proceedings of Eurographics Workshop on Rendering* (Dublin, Ireland, June 1995), pp. 296–305.
- [14] FRASER, C. W., HANSON, D. R., AND PROEBSTING, T. A. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems* 1, 3 (September 1992), 213–226.
- [15] FUNKHOUSER, T. A., AND SÉQUIN, C. H. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *Proceedings of SIGGRAPH 93* (Anaheim, California, August 1993), Computer Graphics Proceedings, Annual Conference Series, pp. 247–254. ISBN 0-201-58889-7.
- [16] GARLAND, M., AND HECKBERT, P. S. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH 97* (Los Angeles, California, August 1997), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 209–216. ISBN 0-89791-896-7.
- [17] GARLAND, M., AND HECKBERT, P. S. Simplifying surfaces with color and texture using quadric error metrics. In *IEEE Visualization '98* (October 1998), IEEE, pp. 263–270. ISBN 0-8186-9176-X.
- [18] GOLDMAN, D. B. Fake fur rendering. In *Proceedings of SIGGRAPH 97* (Los Angeles, California, August 1997), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 127–134. ISBN 0-89791-896-7.
- [19] GUENTER, B., KNOBLOCK, T. B., AND RUF, E. Specializing shaders. In *Proceedings of SIGGRAPH 95* (Los Angeles, California, August 1995), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 343–350. ISBN 0-201-84776-0.
- [20] HANRAHAN, P., AND LAWSON, J. A language for shading and lighting calculations. In *Computer Graphics (Proceedings of SIGGRAPH 90)* (Dallas, Texas, August 1990), vol. 24, pp. 289–298. ISBN 0-201-50933-4.
- [21] HART, J. C., CARR, N., KAMEYA, M., TIBBITTS, S. A., AND COLEMAN, T. J. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. In *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (Los Angeles, California, August 1999), ACM SIGGRAPH / Eurographics / ACM Press, pp. 45–53.
- [22] HECKBERT, P., ROSSIGNAC, J., HOPPE, H., SCHROEDER, W., SOUCY, M., AND VARSHNEY, A. Multiresolution surface modeling. In *SIGGRAPH 1997 Course Notes* (August 1997), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley.
- [23] HEIDRICH, W., SLUSALLEK, P., AND SEIDEL, H.-P. Sampling procedural shaders using affine arithmetic. 158–176. ISSN 0730-0301.
- [24] HOPPE, H. Progressive meshes. In *Proceedings of SIGGRAPH 96* (New Orleans, Louisiana, August 1996), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 99–108. ISBN 0-201-94800-1.
- [25] HOPPE, H. View-dependent refinement of progressive meshes. In *Proceedings of SIGGRAPH 97* (Los Angeles, California, August 1997), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 189–198. ISBN 0-89791-896-7.
- [26] KAJIYA, J. T. Anisotropic reflection models. In *Computer Graphics (Proceedings of SIGGRAPH 85)* (San Francisco, California, July 1985), vol. 19, pp. 15–21.
- [27] KAUTZ, J., AND MCCOOL, M. D. Interactive rendering with arbitrary brdfs using separable approximations. In *Eurographics Rendering Workshop 1999* (Granada, Spain, June 1999), Springer Wein / Eurographics.
- [28] KAUTZ, J., AND SEIDEL, H.-P. Towards interactive bump mapping with anisotropic shift-variant brdfs. *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (August 2000), 51–58.
- [29] MCCOOL, M. D., ANG, J., AND AHMAD, A. Homomorphic factorization of brdfs for high-performance rendering. In *Proceedings of SIGGRAPH 2001* (August 2001), Computer Graphics Proceedings, Annual Conference Series, ACM Press / ACM SIGGRAPH, pp. 171–178. ISBN 1-58113-292-1.
- [30] MICROSOFT. *DirectX Graphics Programmers Guide*, directx 8.1 ed. Microsoft Developers Network Library, 2001.
- [31] NVIDIA. *NVIDIA OpenGL Extensions Specifications*, March 2001.
- [32] OLANO, M., HART, J. C., HEIDRICH, W., LINDHOLM, E., MCCOOL, M., MARK, B., AND PERLIN, K. Real-time shading. In *SIGGRAPH 2001 Course Notes* (August 2001).
- [33] OLANO, M., AND LASTRA, A. A shading language on graphics hardware: The pixelflow shading system. In *Proceedings of SIGGRAPH 98* (Orlando, Florida, July 1998), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 159–168. ISBN 0-89791-999-8.
- [34] PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. Interactive multi-pass programmable shading. In *Proceedings of SIGGRAPH 2000* (July 2000), Computer Graphics Proceedings, Annual Conference Series, ACM Press / ACM SIGGRAPH / Addison Wesley Longman, pp. 425–432. ISBN 1-58113-208-5.
- [35] PERLIN, K. An image synthesizer. In *Computer Graphics (Proceedings of SIGGRAPH 85)* (San Francisco, California, July 1985), vol. 19, pp. 287–296.

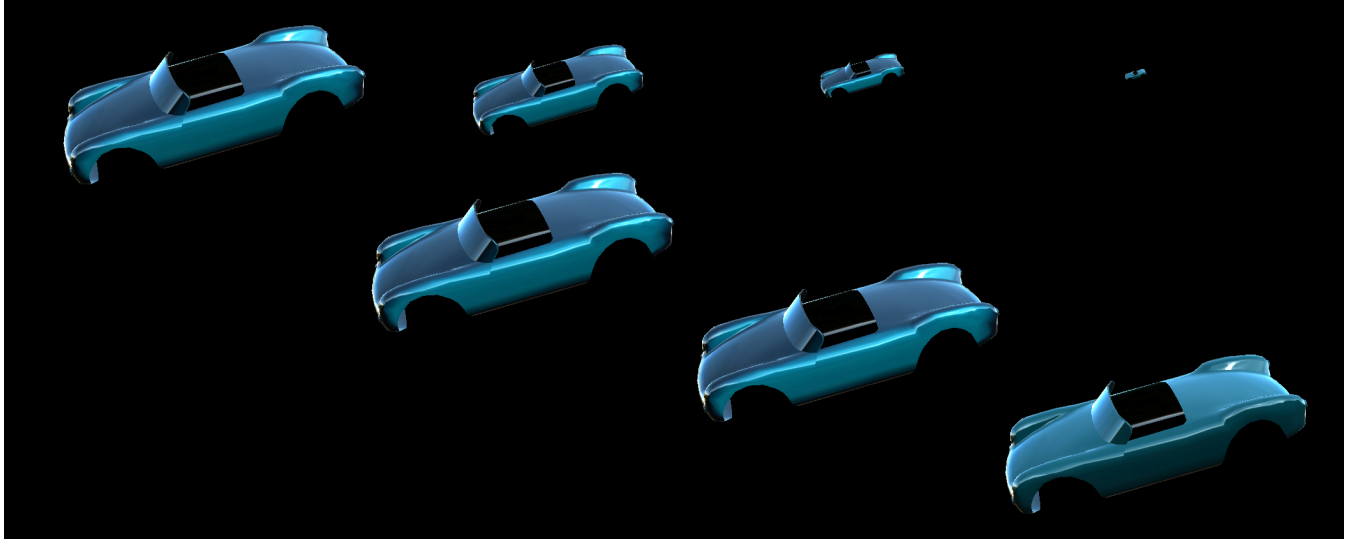


Figure 9: Car paint LOD shader using LOD versions of OpenGL Shader's `microfacetBRDF` and `hdrFresnel` building block functions.

- [36] PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of SIGGRAPH 2001* (August 2001), Computer Graphics Proceedings, Annual Conference Series, ACM Press / ACM SIGGRAPH, pp. 159–170. ISBN 1-58113-292-1.
- [37] RAMAMOORTHY, R., AND HANRAHAN, P. An efficient representation for irradiance environment maps. In *Proceedings of SIGGRAPH 2001* (August 2001), Computer Graphics Proceedings, Annual Conference Series, ACM Press / ACM SIGGRAPH, pp. 497–500. ISBN 1-58113-292-1.
- [38] RHOADES, J., TURK, G., BELL, A., STATE, A., NEUMANN, U., AND VARSHNEY, A. Real-time procedural textures. In *1992 Symposium on Interactive 3D Graphics* (March 1992), vol. 25, pp. 95–100. ISBN 0-89791-467-8.
- [39] SCHROEDER, W. J., ZARGE, J. A., AND LORENSEN, W. E. Decimation of triangle meshes. In *Computer Graphics (Proceedings of SIGGRAPH 92)* (Chicago, Illinois, July 1992), vol. 26, pp. 65–70. ISBN 0-201-51585-7.
- [40] SHADE, J., LISCHINSKI, D., SALESIN, D. H., DEROSE, T. D., AND SNYDER, J. Hierarchical image caching for accelerated walkthroughs of complex environments. In *Proceedings of SIGGRAPH 96* (New Orleans, Louisiana, August 1996), Computer Graphics Proceedings, Annual Conference Series, ACM SIGGRAPH / Addison Wesley, pp. 75–82. ISBN 0-201-94800-1.
- [41] TURK, G. Re-tiling polygonal surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 92)* (Chicago, Illinois, July 1992), vol. 26, pp. 55–64. ISBN 0-201-51585-7.