Ashli – Advanced Shading Language Interface

Arcot J. Preetham*, Avi Bleiweiss†.

ATI Research.

Abstract

The support for IEEE floating point computation and the exposure of shading functionality in a standardize API form have made graphics hardware a viable workflow solution to an artist involved in digital content creation (DCC). Nevertheless, there still remains a significant interface void between the abstract shading description an artist is used to and the low level shading constructs the hardware expec ts. We have developed Ashli, an advanced shading language interface tool, with the primary motivation to bridge the interface gap fore mentioned. Ashli is developed in the form of a case study. It takes in high level shading languages and descriptions and at the end emits standard graphics hardware shading API (e.g. Microsoft DirectX and OpenGL). Ashli's main contribution is in its seamless cooperation with the DCC graphics application. The RenderMan®‡ Shading Language, Maya® Shading Network and 3ds Max® Standard Materials are the subset of input abstraction we have experimented with to validate our approach. Shading path computation complexity and conformance to hardware resource constraints are owned by Ashli and for the most part made transparent to the artist. A tool like Ashli essentially retains rendered image appearance quality comparable to the level produced by a software renderer, but at a significant higher rate of interaction efficiency.

We present Ashli in the context of a stand-alone viewing application framework, depicting shading path computation bound to scene rendering. We demonstrate the application design and results for using Ashli as a shading coprocessor assist.

Keywords:
Graphics hardware, scene, rendering, shading, texture, shadowin g, vertex, color, normal

1. Introduction

Digital content creation artists have relied primarily on software renderers to produce photo realistic images. Applications such as Maya® and 3ds Max® provide the standard for an artist development environment. These packages use shading descriptive scripts and visual network builders to best serve an artist in producing digital content. Software renderers are the ultimate choice for flexibility and appearance quality to the user, but at a cost of significant reduced interaction and overall productivity. Graphics hardware renderers however, have until now made very little integration inroads to actively participate in a movie production workflow. The major acceptance roadblock was the lack of fine computation precision artists are used to.

Recently, graphics hardware has evolved to support IEEE floating point in both arithmetic calculations and storage. In addition, hardware shading functionality is exposed through a low level programmable layer by each Microsoft's DirectX and OpenGL graphics interface standards. These enhancements potentially restate graphics hardware as a viable candidate for deployment in a small subset of production workflow tasks. There still remains though the interface void between the shading development cultures the artist is used to and the low level shading constructs the graphics hardware expects. The motivation behind *Ashli*, an *advanced shading*

* Email: preetham@ati.com

† Email: avi@ati.com

‡ RenderMan is a registered trademark of Pixar.

language interface tool is to bridge the interface gap fore mentioned. Ashli's goal is to expose an almost identical level of appearance quality artists enjoyed using software renderers, but at a much higher interaction speeds.

2. Overview

The part of Ashli's case study we present involves shaders written in the RenderMan® Shading Language and in the OpenGL Shading Language. The languages have many references out there in the form of specifications [12, 15] and books [1, 13]. It is recommended that as you go over this document you have one of these references handy. Ashli supports a modest subset of any of the languages. Nevertheless, the functionality realized deems appropriate to demonstrate relatively complex shaders displayed in real time, when ran on recent graphics hardware.

The rest of the paper is organized in the following manner. We first give a brief overview of Ashli core compiler in Section 3. In Section 4 we present the Ashli Viewer application design and methodology for interfacing with both Ashli and the rendering engine. This follows by an operational description of the viewer application, in Section 5. Finally, in Section 6 we depict results of rendered geometry bound with an assortment of material and light shaders in the form of image snapshots.

3. Ashli

An application provides Ashli with a collection of RenderMan® shaders e.g. *displacement*, *surface*, *light*, *volume* and *imager* types or OpenGL shaders e.g. *vertex* and *fragment* shaders. The collection of input shaders forms an Ashli *program*, destined for compilation. Any one of the shader types in the collection could optionally be instantiated multiple times. Shader instancing applies mainly to a scene of multiple lights, each of the same basic type. Ashli is fairly agnostics with respect to the hardware shading language API it emits. Incoming shaders are translated to any of DirectX 9.0 Pixel Shader and Vertex Shader Version 2.0 [8] or OpenGL ARB_vertex_program [11] and ARB_fragment_program [10]. In addition, Ashli generates a *formals* text data structure that specifies the mapping of runtime appearance parameters onto hardware shader resources such as *input*, *constant* and *sampler* registers.

Often times, a complex high level shader may not fit into a single low level shader program, which conforms to hardware resource constraints. E.g. the number of any of input, constant, temporary or sampler registers has been exceeded, or instruction space reached its limits. Under such conditions, Ashli breaks the complex shader into many smaller *segments* (passes), where each segment uses up resources within the prescribed hardware limits. This process of breaking up complex shaders into smaller units is known as *multipassing*. Ashli's multipass implementation is based on the technique demonstrated by Chan et al. [5]. Ashli provides the artist a framework for creating arbitrarily complex programs, avoiding hardware resource limitation concerns. The reader is referred to [4] for a more detailed description of Ashli core.

4. Ashli Viewer

Ashli Viewer is an application that demonstrates Ashli as a shading coprocessor assist. It facilitates the user means for previewing a modest subset of RenderMan® shaders or OpenGL Shading language programs in real time. A scene to be rendered is commonly composed of lights and materials and is defined in the form of a *scene file*. Ashli Viewer provides the framework for editing any of the scene description and the shaders, compile the shaders program and finally render a geometry model with the hardware generated shaders, in real time. The user previews the scene by moving lights and/or changing properties of the lights and materials.

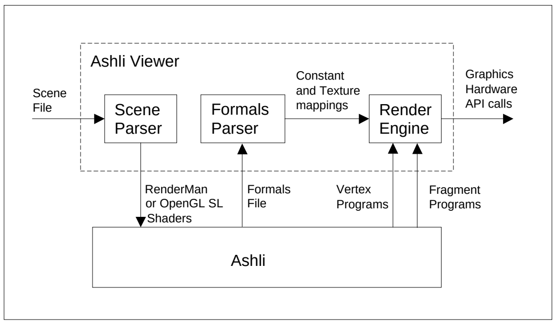


Figure 1: Block diagram of Ashli Viewer

Ashli Viewer is composed of a *scene parser*, a *formals parser* and a *rendering engine* (shown in Figure 1). The scene parser parses the scene file to identify RenderMan® shaders or OpenGL shaders, which are sent as inputs to Ashli. Ashli generates formals, vertex and pixel (or fragment) programs from these high level shaders. The formals parser takes the formals text data structure as input and generates the following outputs for each pass: the mapping of shader runtime parameters onto constant registers and the binding of textures onto sampler registers. The vertex program operates on geometry attributes such as position, normal and texture coordinates destined to the hardware pixel (fragment) processor input registers. Similarly, appearance runtime parameters and textures are set by the application in the pixel (fragment) processor. The rendering engine renders the scene using programmable shaders through a standard graphics API calls (DirectX or OpenGL). In summary, the scene file is translated with the help of Ashli and rendered on graphics hardware.

4.1. Scene file

The rendering process requires a definition for the scene of concern. The scene file is a simple text interface to describe material properties of the geometric object, the lights in the scene, atmospheric effects in the environment and post rendering image processing filters. Every scene file contains one object with at least one material. The material properties are described through RenderMan ® or OpenGL shaders. The following discussion focuses on scene files in the context of RenderMan® shaders. Scene file formatting using OpenGL shaders is for the most part identical with the addition of language specific keywords.

A list of lights used in the scene is specified along with the light definition. The definition of a light includes the light shader and its runtime parameters. The absence of runtime parameters falls back into the use of default values specified by the source shader. The modification of the shader runtime parameters is realized by the use of user interface *sliders*. Every runtime parameter is associated with a slider. The user can optionally specify the range of values for a slider. Alternatively, the user can make a shader runtime parameter a constant using the keyword *const.* Const'ing runtime parameters relieves hardware shading resource usage and is highly encouraged by the user, when possible. As a result, compute operations are further simplified by Ashli to yield a more optimized low-level hardware shader. For convenience, all runtime shader parameters can be made constant by using the keywords *const all*. Following is an example of a light definition using a spot light shader with the runtime parameters 'coneangle' and 'intensity'. The parameter 'intensity' takes values in the range 0 to 1:

```
light spotlight.sl

float coneangle 0.4 range 0 1.0

const intensity 1.0
```

A volume shader for creating atmospheric effects, such as fog or smoke, can be optionally specified. Following is the definition of a fog volume shader, which takes a 'fogcolor' runtime parameter:

```
atmosphere fog.sl

color fogcolor 0.3 0.3 0.3
```

A scene should have at least one material definition specified. A material is described by a material name and is followed by an optional displacement shader and a mandatory surface shader with each of their runtime parameters. Following is a definition of a material named 'table', which uses a wood surface shader and its 'grainy' runtime parameter:

```
material table

shader wood.sl

float grainy 0.5
```

In the presence of more than one material in the scene, the lights affect each of the materials independently. To have a light runtime parameter affect all the materials in the scene, the parameter has to be made common by the use of the keyword *common*. Similarly, volume shader parameters can be made common to affect all the materials in the scene. For example, in the following definition, the 'intensity' runtime parameter of the spot light is common to all materials:

```
light spotlight.sl

float intensity 1.0

common intensity
```

String type shader parameters are generally associated with textures where a bitmap file should be specified. Keywords *SCENE* or *texShadowMap_n* for string arguments signify special meanings discussed later. Following is an example showing a string parameter bound to a texture name 'design.tga':

```
string texturename design.tga
```

A background image can be added to the scene by specifying the image file as shown in the following example:

```
background skyline.tga
```

The scene file might contain initial geometry transformation for an object. It is specified by any combination of translation, scale and rotation offsets as shown below:

```
translate z 5

scale y 1.2

rotate x 30
```

Post rendering image processing (e.g. blur filter, noise reduction filter) is performed on the rendered scene through imager shader execution. A list of imager shaders can be specified in the scene file. The imager shaders are applied in order to the rendered scene in a cascade manner. All imager shaders take a string parameter, which is the input texture for image processing. This parameter is bound to the keyword *SCENE* in the scene file. It refers to the most recent rendered scene. A hierarchy of scene files is probably the most intuitive way to specify a series of image processing operations, post rendering. Following is an example that shows the use of a gaussianfilter imager shader with runtime parameters 'imagename' and 'kernelsize'.

```
include room.scn

imager gaussianfilter.sl

string imagename SCENE

float kernelsize 1.5
```

Shadows provide a very important cue to artists in placing lights. The shaders for shadow casting lights should contain the formal parameters to define *from* and *to* positions. In addition, the string parameter representing the shadowmap from light instance #*n*, should be bound to the keyword *texShadowMap_n* in the scene file. This provides a hint to the application to render shadowmaps from the shadow casting light position. Light shaders casting shadows can use multiple samples to avoid aliasing of shadow edges. The number of samples have to be specified at compile time through the use of the *const* keyword. The following example shows a shadow casting spot light with runtime parameters ‘from’, ‘to’ and ’shadowmap’:

```
light spotlight.sl

point from 0 0 0

point to 0 0 -1

string shadowmap texShadowMap_0
```

4.2. Rendering

The rendering engine is built on top of both DirectX 9 and OpenGL graphics API. The target graphics API is runtime selectable in the application. The rendering unit sets up hardware shading state into both the vertex and pixel (fragment) processors and it binds hardware shaders with the geometry destined for rendering. In the case of multipassing, materials are each rendered progressively for all the passes before moving on to the next material. Intermediate passes are rendered into an IEEE floating point component temporary buffer. A temporary buffer is used as a texture in all subsequent passes. The availability of floating point target buffers in the current generation of graphics hardware (e.g. ATI Radeon 9700/9800, and nVidia GeForceFX) provides arbitrary complex shader decomposition without a loss of precision.

Floating point temporary buffers are also used in the case of shadow map generation and post rendering image processing. Shadow maps are generated, by rendering the scene from the light position. Light position, object position and object size are used to determine an approximate viewing frustum for rendering from the light. Percentage closer filtering technique is used to reduce aliasing of shadow boundaries [14]. In this course of rendering the floating point depth value of visible pixels are stored in a temporary buffer [16]. Similarly, for scenes containing imager shaders, the scene is rendered into a temporary invisible floating point buffer, which is later used as a texture in the imager pass.

Currently, the number of temporary buffers created in the application is equal to the number of passes. Often, an intermediate temporary buffer may not be needed through the rendering of all the passes and can be reused as a target buffer more than once. An optimal solution would be to create only as many temporary buffers as required after doing a live range analysis on the temporary buffers from all passes and reusing the temporary buffers. This has been implemented in other Ashli case studies where, native 3ds Max® standard materials and Maya® shading networks are rendered in real time [3, 6].

5. Demo

The Ashli Viewer demo runs on any graphics hardware that supports supports DirectX 9.0 Vertex and Pixel Shader Version 2.0 or OpenGL ARB_vertex_program and ARB_fragment_program. Ashli Viewer framework window (shown in Figure 2) has a program list in the left column. Each program item contains a scene file and one or more shaders. A scene file or a shader file is opened in the center column by clicking on the item of interest. These files can be edited and saved for subsequent compilation and rendering. Hardware shading API target (one of DirectX 9.0, OpenGL) is selected from the pull down list below the left column. The user hits the "Compile" button to generate hardware shading target programs of the desired API. These programs are depicted in the right hand column. Information on the number of instructions and passes for each material is shown in the status panel at the bottom. Target programs (vertex or pixel) for any pass and any material can be viewed in the column on the right panel by choosing a material and pass from the pull down lists on the top. Alternatively, the user can hit "Compile And Run" button to compile and render the scene in one of two modes – *Window* or *Full Screen*. In windowed mode, an additional window containing the sliders representing shader formal parameters appear and are used for runtime appearance control. Figure 3 shows a scene rendered in window mode with slider controls for manipulating the material properties.

6. Results

The Ashli demo package is available for public download on ATI’s developer main website [2]. Screenshots of several scenes rendered on an ATI Radeon 9700 are shown.

In Figure 4, the object comprises of 3 materials – the base is a simple specular surface, tusk is a matte surface, and the elephant’s body is the classic RenderMan® wood [13] shader. The scene has 5 distant lights, 2 of which cast shadows. In Figure 5, the bunny is rendered with a cartoon shader. In Figure 6, a full screen plane is rendered with the flame procedural shader [9]. In Figure 7, the apple is rendered using a procedural shader [7], and the scene has two lights.

7. Acknowledgements

We would like to thank Dominik Behr and Seth Sowerby for helping us with the rendering engine, Mike Huber for the models and Mark Peercy and Raja Koduri for their invaluable advice.

References:

1. Anthony A. Apodaca and Larry Gritz: *Advanced RenderMan®, Creating CGI for Motion Pictures, Morgan Kaufmann Publishers 2000*.
2. ATI Developer site: http://www.ati.com/developer/.
3. Dominik Behr: Rendering 3ds Max Standard Materials on Hardware. http://www.ati.com/developer/gdc/AshliMaxGDC.pdf.

4. Avi Bleiweiss, Arcot Preetham: Ashli – Advanced Shading Language Interface, To appear in *Real Time Shading Course Notes, Siggraph 2003*.

5. Eric Chan, Ren Ng, Pradeep Sen, Kekoa Proudfoot, Pat Hanrahan: Efficient Partitioning of Fragment Shaders for Multipass Rendering on Programmable Graphics Hardware. *Proceedings of the SIGGRAPH/Eurographics Workshop on Graphics Hardware 2002.*

6. Matt Komsthoeft: Rendering Native Materials in Real Time. http://www.ati.com/developer/gdc/AshliMayaGDC.pdf.

7. Jonathan Meritt: http://www.renderman.org/RMR/Shaders/JMShaders/JMredapple.sl.

8. Microsoft DirectX: http://www.microsoft.com/windows/directx/.

9. Ken Musgrave: http://www.renderman.org/RMR/Shaders/KMShaders/KMFlame.sl.

10. OpenGL Arb_Fragment_Program: http://oss.sgi.com/projects/ogl-sample/registry/ARB/fragment_program.txt.

11. OpenGL Arb_Vertex_Program: http://oss.sgi.com/projects/ogl-sample/registry/ARB/vertex_program.txt.

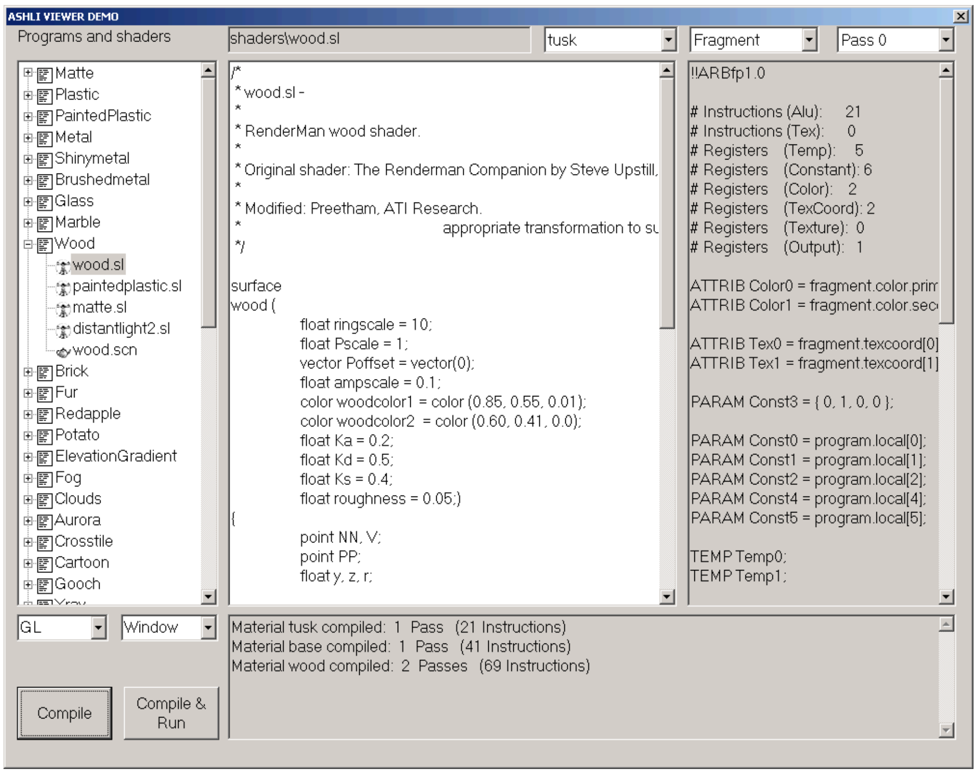
12. OpenGL Shading Language: http://www.opengl.org/developers/documentation/gl2_workgroup/.

13. Steve Upstill: *The Renderman® Companion: A Programmer's Guide to Realistic Computer Graphics, Addison Wesley 1990*.

14. William Reeves, David Salesin, Robert Cook: Rendering Antialiased Shadows with Depth Maps. *Computer Graphics Forum 21(4): 1987.*

15. RenderMan® Repository: http://www.renderman.org.

16. L. Williams: Casting Curved Shadows on Curved Surfaces. *In Proceedings of Siggraph, 270-274, 1978.*

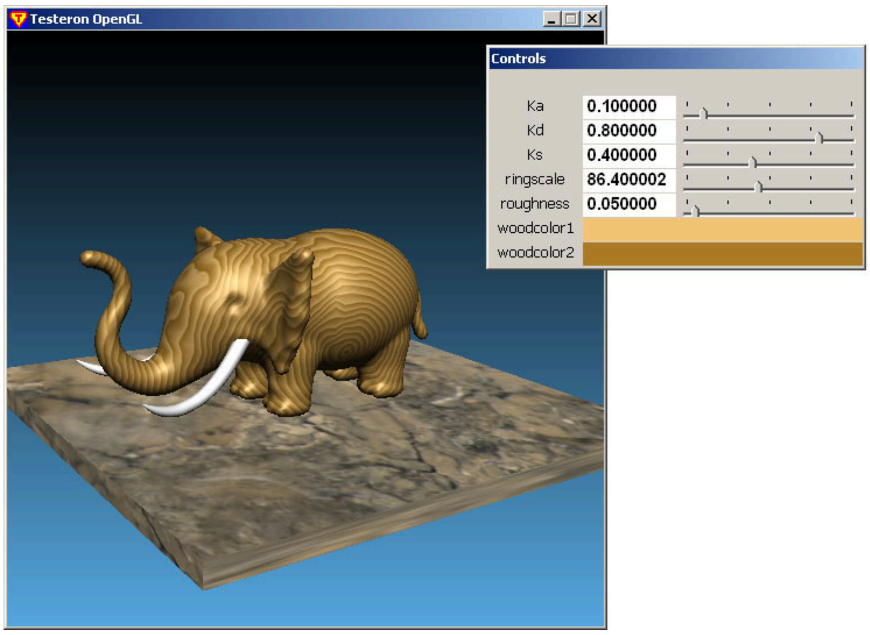


```
/*
 * wood.sl -
 *
 * RenderMan wood shader.
 *
 * Original shader: The Renderman Companion by Steve Upstill,
 *
 * Modified: Preetham, ATI Research.
 *                                  appropriate transformation to su
 */

surface
wood (
        float ringscale = 10;
        float Pscale = 1;
        vector Poffset = vector(0);
        float ampscale = 0.1;
        color woodcolor1 = color (0.85, 0.55, 0.01);
        color woodcolor2  = color (0.60, 0.41, 0.0);
        float Ka = 0.2;
        float Kd = 0.5;
        float Ks = 0.4;
        float roughness = 0.05;)
{
        point NN, V;
        point PP;
        float y, z, r;
```

```
!!ARBfp1.0

# Instructions (Alu):     21
# Instructions (Tex):      0
# Registers   (Temp):      5
# Registers   (Constant): 6
# Registers   (Color):     2
# Registers   (TexCoord): 2
# Registers   (Texture):  0
# Registers   (Output):    1

ATTRIB Color0 = fragment.color.prim
ATTRIB Color1 = fragment.color.seco

ATTRIB Tex0 = fragment.texcoord[0]
ATTRIB Tex1 = fragment.texcoord[1]

PARAM Const3 = { 0, 1, 0, 0 };

PARAM Const0 = program.local[0];
PARAM Const1 = program.local[1];
PARAM Const2 = program.local[2];
PARAM Const4 = program.local[4];
PARAM Const5 = program.local[5];

TEMP Temp0;
TEMP Temp1;
```

```
Material tusk compiled:  1  Pass   (21 Instructions)
Material base compiled:  1  Pass   (41 Instructions)
Material wood compiled:  2  Passes  (69 Instructions)
```

Figure 2: Snapshot of Ashli Viewer showing the list of programs, RenderMan® shader and target ARB_fragment_program.



Ka	0.100000
Kd	0.800000
Ks	0.400000
ringscale	86.400002
roughness	0.050000
woodcolor1	🟧
woodcolor2	🟫

Figure 3: Snapshot of a rendered scene in window mode with the controls for adjusting the properties of the wood material.

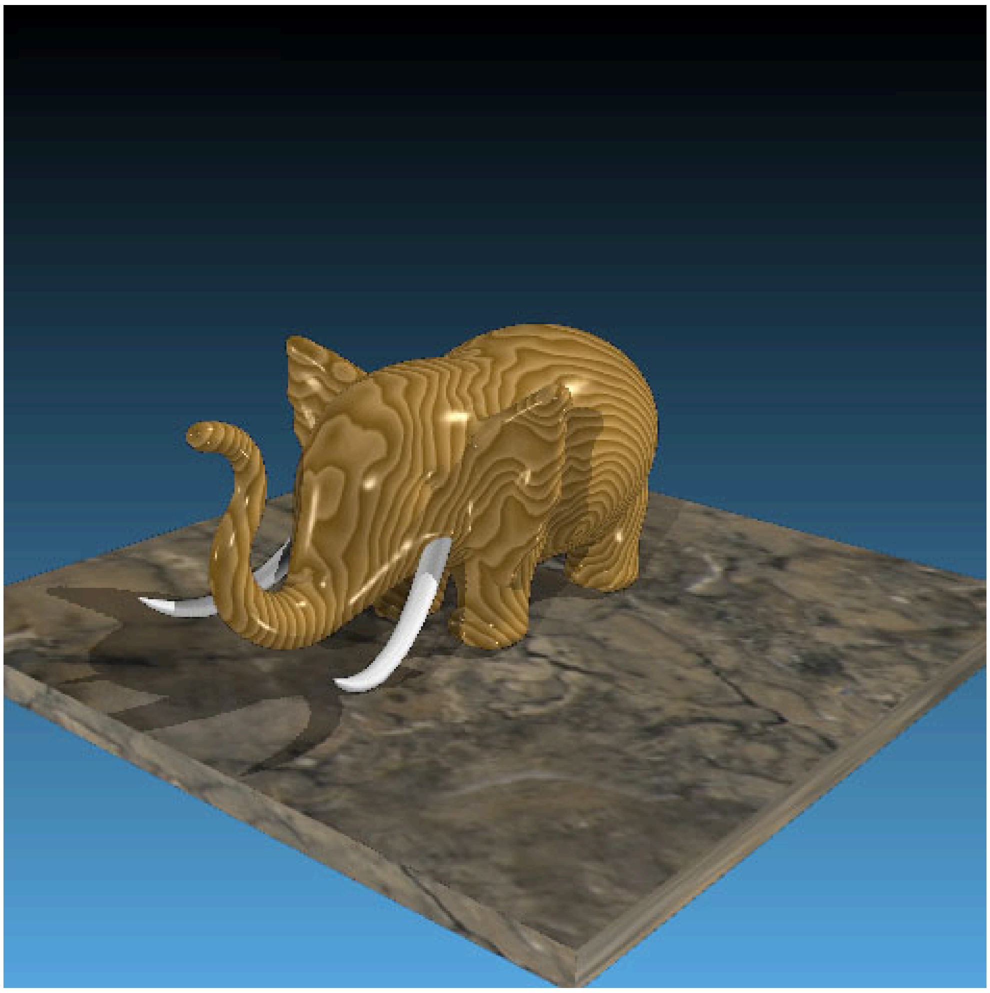


Figure 4: Elephant rendered using RenderMan® wood shader and 5 distant lights with 2 of them casting shadows.

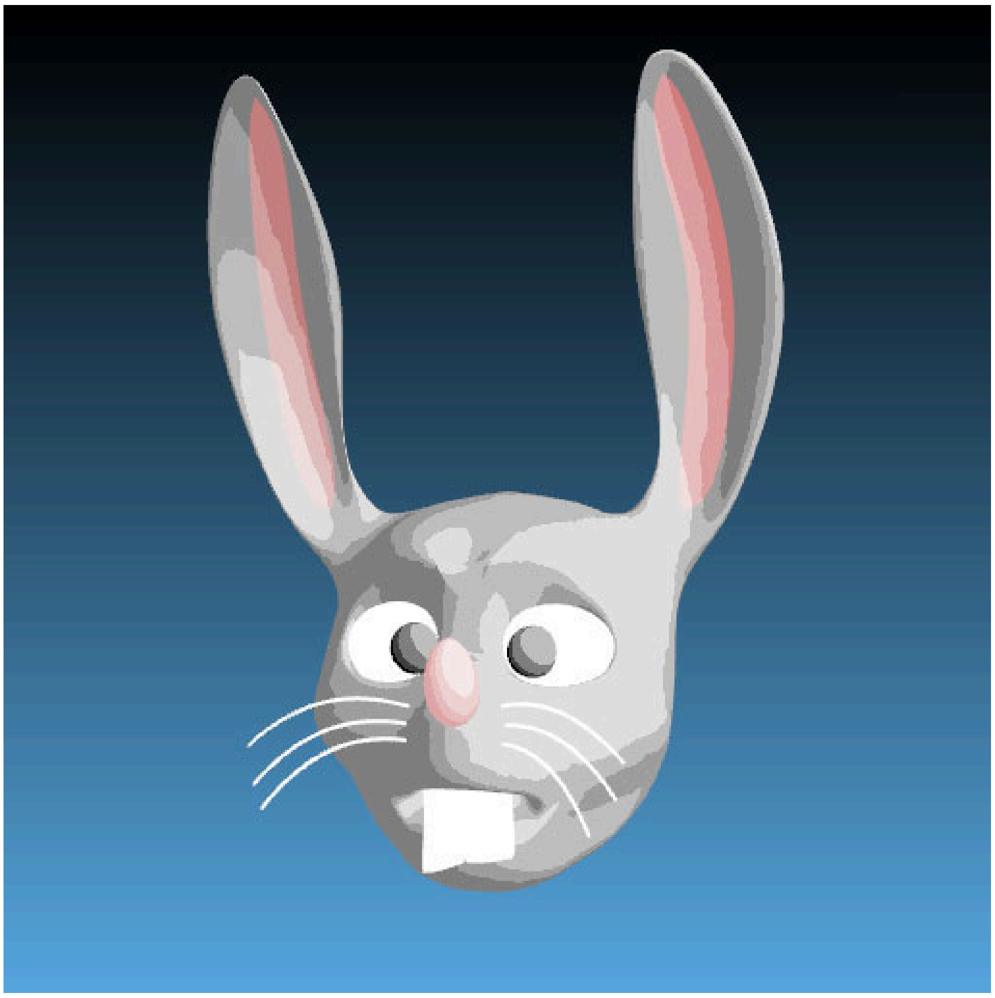


Figure 5: Bunny rendered using a cartoon shader.

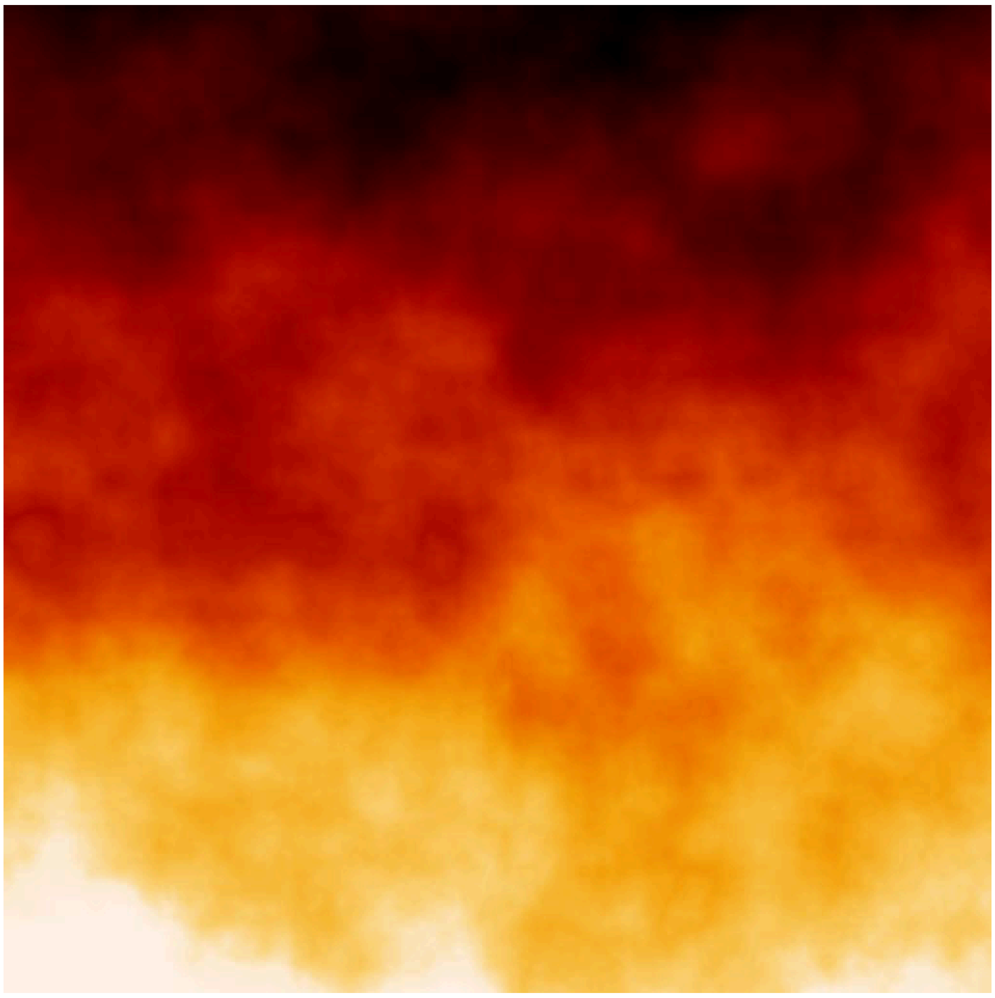


Figure 6: Full screen quad rendered using a procedural flame shader.

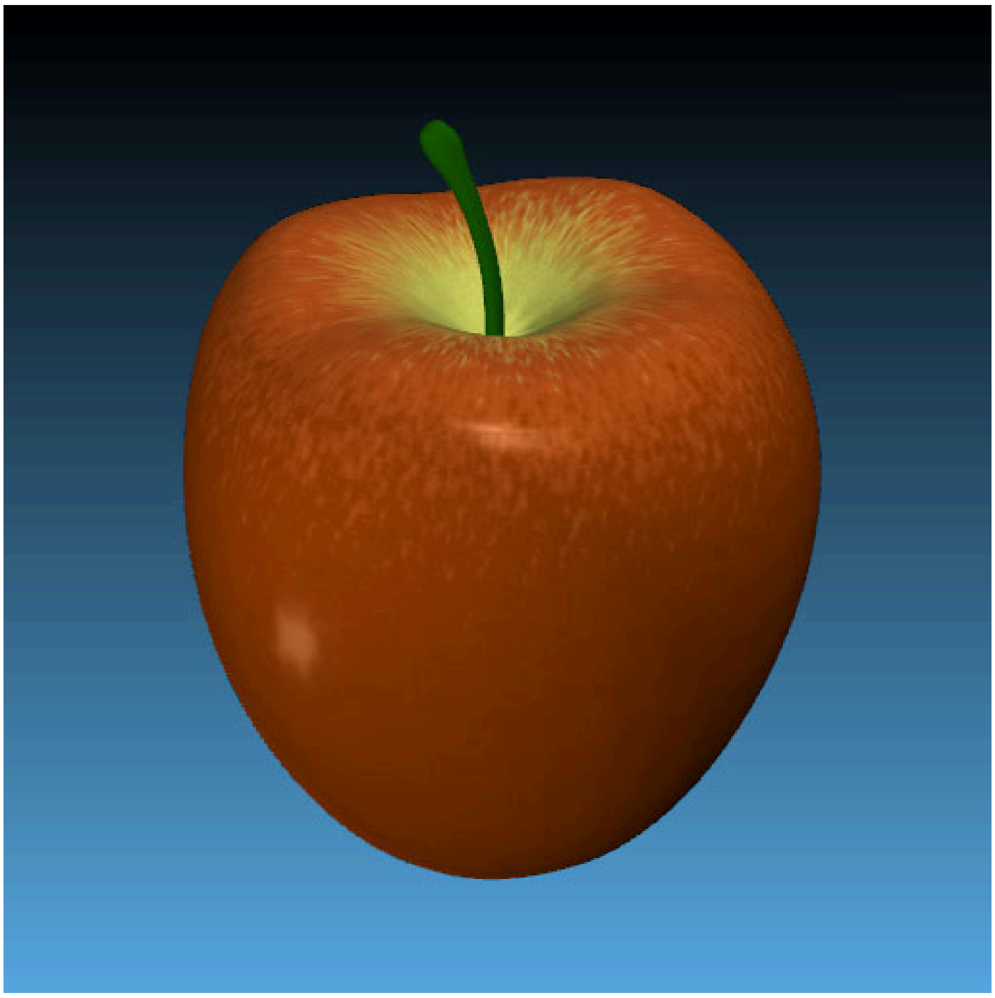


Figure 7: Apple rendered using a procedural shader and two distant lights.