Ashli – Advanced Shading Language Interface

Avi Bleiweiss, Arcot Preetham
ATI Research, Inc.

Overview

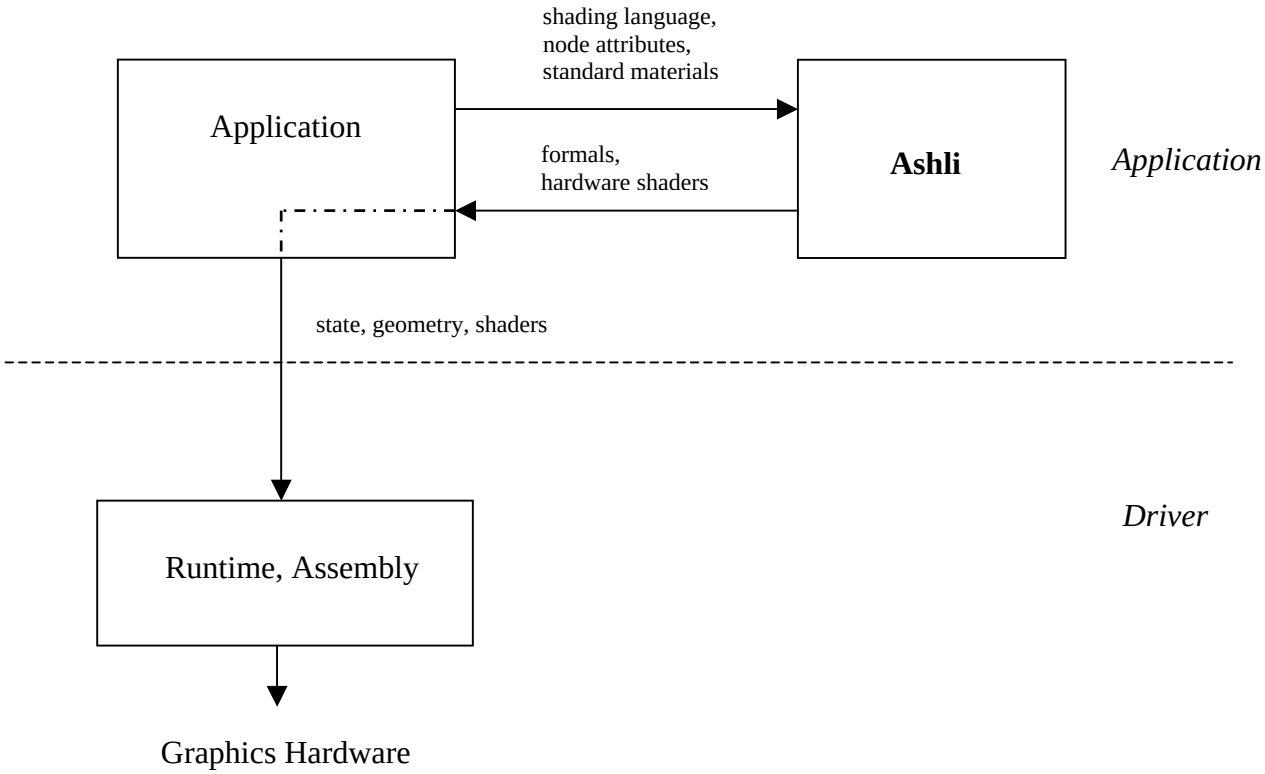
Digital content creators (DCC) have relied on software rendering systems to generate imagery that is a familiar part of our movie and television experience. Packages such as Maya®, 3D Studio Max® produce the images that set the standard for cinematic rendering. These packages use descriptive scripts, shading languages and visual network builders to best serve an artist in the most productive form. Graphics hardware renderers however, until now have made little inroad if at all into a movie production studio. This is primarily due to the lack of fine computation precision artists use to produce their high dynamic range images. The attitude towards embedding hardware rendering in a film or video production flow appears to have changed in the past year or so.

Graphics hardware accelerators have evolved recently to fully support floating point computation and storage inside their pipe. In addition, hardware shading functionality is exposed through a low level programmable layer by each Microsoft's Direct3D and OpenGL graphics interface standards. Both the virtues of fine arithmetic precision and a standardized appearance interface make graphics hardware a viable candidate for serving the artist in a subset of studio production workflow tasks. Essentially, retaining appearance quality at the level produced by the software renderer, but at a significant higher rate of interaction speed. Nevertheless, there still remains the interface gap between the shading description an artist is used to and the shading constructs the hardware expects. The primary motivation for *Ashli*, an *advanced shading language interface* tool, is to fill in the interface void described, between the artist, who drives the content creation application, and the hardware. Bridging the artist to hardware shading gap interface, while preserving the existing content creation culture, is Ashli's primary goal.

Ashli is a case study for a tool, which takes in high level shading languages and descriptions and at the end emits standard graphics hardware shading API streams. Much of Ashli's horsepower resides in its processing stages for simplifying incoming shading constructs down to multiple intermediate representations. All for the sake of producing efficient hardware shading code. Ashli's main contribution is in its seamless cooperation with the DCC graphics application. Well-established shading descriptions such as the RenderMan®[1] Shading Language, Maya® Shading Network and 3DS Max® Standard Materials are part of Ashli's input abstraction. Shading path computation complexity and conformance to hardware resource constraints are owned by Ashli and essentially made transparent to the artist. Ashli returns to the application a well-defined shader *formals* description for runtime control of appearance parameters and final hardware shader code.

[1] RenderMan® is a registered trademark of Pixar

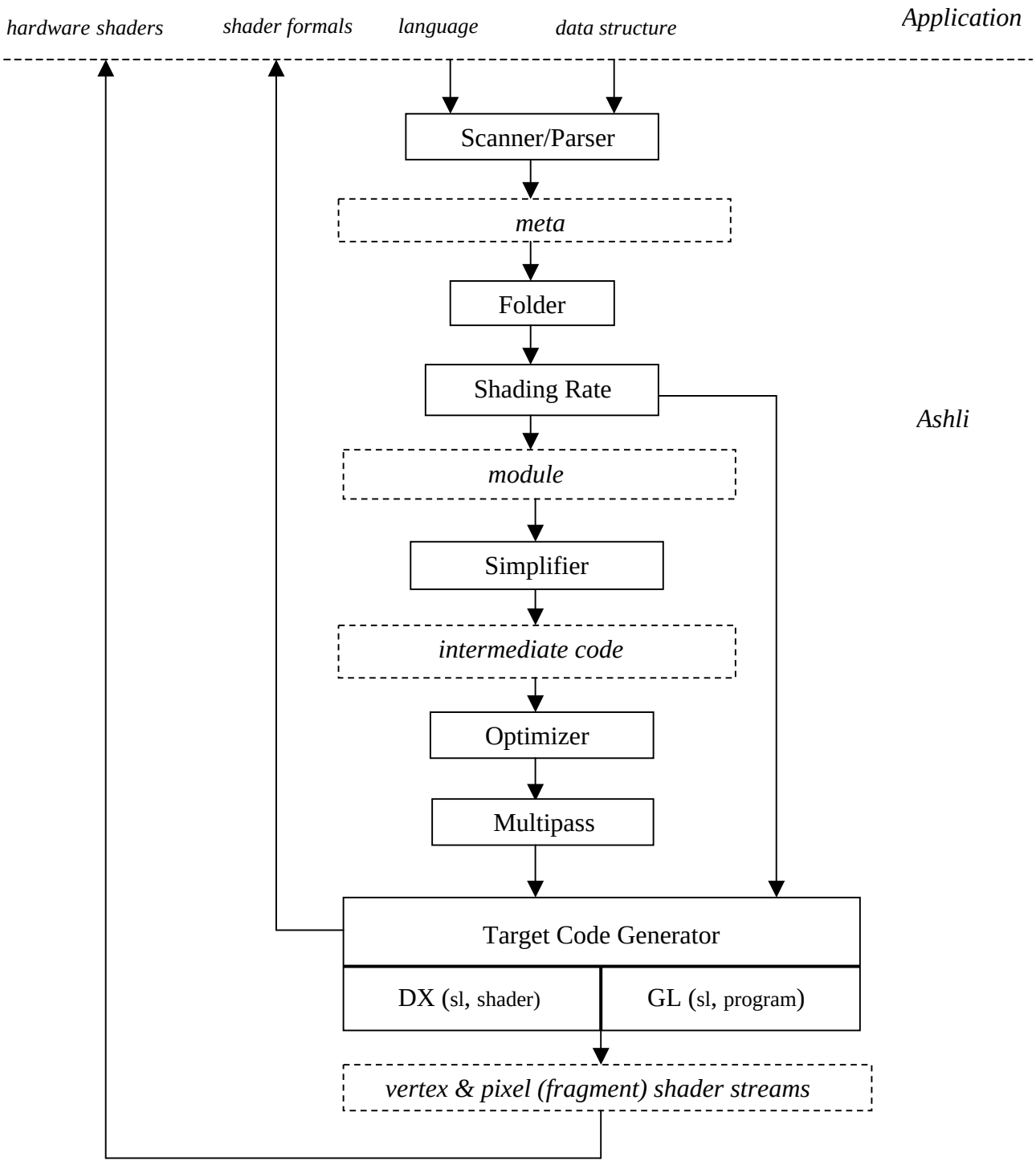
The application then uses the shading path output and binds it with the designated geometry in the course of rendering. Ashli is hardware shader format agnostics and supports both Microsoft's Direct3D and OpenGL targets. The following figure depicts Ashli's position in the overall application to hardware shading path:



Application to Hardware Shading Path

Shading Pipe

Ashli embeds a compiler technology in performing shading path computations. It has a front end and a back end component. The front end is a directed acyclic graph, whose nodes are functions commonly found in the interfaces used by artists. The back end is a code generator that produces code described in a hardware interface of choice. Currently, the back end supports Microsoft DirectX 9.0 Vertex and Pixel Shader version 2.0 and OpenGL ARB_{vertex, fragment}_program. It is being extended to support DirectX 9.0 HLSL and the OpenGL Shading Language. The following drawing illustrates Ashli's shading pipe stages (intermediate data representations are shown in dashed boxes; they are positioned right after the producing stage and are consumed by stages there after):



Ashli Pipeline

The discussion hereafter focuses on the RenderMan® Shading Language as the primary input to Ashli. Ashli supports a relatively modest subset of the language. An application

provides Ashli with a collection of shaders e.g. *displacement*, *surface*, *light*, *volume* and *imager* types. The collection of input shaders forms an Ashli *program*, destined for compilation. The program is an assembly of unique shader types and every one could optionally be instantiated numerous times. Shader instancing applies mainly to a scene of multiple lights, each of the same basic type. A shader instance can be assigned its own set of default formals for differentiation.

Initially, each shader of the program is lexically scanned, parsed and then converted into an abstract tree data representation – *meta*. The meta tree data structure is composed of conventional compiler objects representing language statements, expressions and operators. The language built in function components are shared across shaders and are considered at the program level. Texture binding info is captured from the texture access statements in the shader and is retained in a unique meta object. Next, the entire program shaders represented in meta are being collapsed and folded into one final execution shader. In the folding stage shader instances are expanded, emission and integration lighting constructs are unrolled and evaluated for color contribution and derivative operators are triple coded for performing parametric differencing. The folding stage output is a single shader and a set of non-inline function components, all in the meta representation format. The shading rate phase load balances final shader execution across graphics hardware processors e.g. *vertex* and *pixel* (*fragment*). Ashli provides means for both implicit and explicit processor execution division of labor. The shading rate stage emits a *module* data representation for each of the hardware processors. For all practical purposes a module is a meta superset data structure. Simplification is the following shading pipe stage. Here, modules are in lined for both the shader and the functions code. In addition, all statements are expanded into a series of simplified assignments using a dyadic typed operator expression. The in lined shader is formatted as a tree data structure, where its nodes are the expression operators and the node children are the expression operands. This representation form is termed *intermediate code*. The intermediate code format lends itself nicely onto a hardware register based shading language. Next, code optimization takes place. The shader tree is traversed to perform live range analysis of register usage, which facilitates the reuse of temporary registers. In addition, a series of tasks aimed at hardware resource constraint relief are conducted. Amongst them are dead code elimination, restating variables as constants, data type promotion, branch evaluation and a generic form of operator minimization. The multi-pass stage consults hardware shading resource constraints e.g. registers and code space and maps the optimized intermediate code onto multiple code segments, if necessary. The presence of multiple code segments implies progressive rendering of the scene. Finally, program formals description is produced for setting hardware shading state. And intermediate code segments are formatted to one of the standardized graphics hardware shading interfaces. DirectX 9 Vertex or Pixel Shader 2.0 and OpenGL ARB_{vertex, fragment}_program streams are emitted at the bottom of the shading pipe. Hardware high level shading languages e.g. DirectX 9 HLSL and the OpenGL Shading Language are generated post the shading rate stage.

A more detailed coverage of some of the shading pipe stages is provided elsewhere in the notes.

Interface

Ashli's Application Programming Interface (API) is designed for simplicity and portability. The interface methods provide the programmer means to set up a compile of a shading program, to invoke a compile process and finally to query the results. By convention, an input program must include a surface type shader (could potentially be a null shader). An input program shader is either a single *item* (e.g. a file or a stream) or is optionally composed of an ordered collection of *items*. Custom functions are expected first and the shader code body appears last. Ashli's generated outputs are made available back to the application by using the API query calls. Compiled results are grouped into those who feed directly onto hardware shading resources and instructions and those considered compile analysis assist data.

Ashli's hardware generated outputs are the reconstructed shader *formals,* which map onto hardware shading registers, and a *vertex* and a *pixel* shader instruction streams. Formals data is text based and is formatted for ease of parsing. The formals description specifies how to bind textures, set program constants, and specify intermediate shader results in the case of multiple shader segments generation. The shader formals description provides the application means for trading off runtime vs. compile time appearance control. Each code segment emitted by Ashli is associated with its own pair set of formals and shader instruction streams. Essentially, a code segment maps onto a rendering pass in the application space.

In the course of compilation Ashli records program statistics of input program complexity characteristics, hardware resource usage and instruction count. Program statistics is exposed in an API query method and is useful for conducting iterative program efficiency tuning.

Ashli's API is logically divided into four method groups: *initialization, setup, compile and query*. One-time, static compile parameters are set by the init methods. Amongst them are hardware target interface type, file vs. stream based input program and outputs, and hardware interface resource caps settings. Hardware resources include registers e.g. *input*, *temporary*, *constant* and *sampler* and overall instruction code space. Setup methods point to an optional program include path, provide means to compose a shader out of shader source items and to instantiate shaders, as necessary. There is one compile invocation method and numerous compile result query methods. The following sample excerpt demonstrates how Ashli's API might be used in an application shading context:

```
#include "IAshli.h"

IAshli ashli;

ashli.init();

int shader, instance;

ashli.setIncludePath("Shaders/Include");
```

```
ashli.addShaderItem("wood.sl");
shader = ashli.addShader();
instance = ashli.addShaderInstance(shader);

ashli.addShaderItem("distantlight.sl");
shader = ashli.addShader();
instance = ashli.addShaderInstance(shader);

if(ashli.invoke("wood")){
    for(int i = 0; i < ashli.getNumSegments(); i++) {
        printf("Formals:\n%s\n", ashli.getFormals(i));
        printf("Vertex shader:\n%s\n", ashli.getVertexShader(i));
        printf("Fragment shader:\n%s\n", ashli.getPixelShader(i));

    }
} else {
    printf("Error:\n%s\n", ashli.getError());
}

printf("Stats:\n%s\n", ashli.getStats(""));
```

Several built in functions of the language (e.g. noise variants, random and atan2) are mapped onto hardware procedural textures. Ashli employs on the fly built-in procedurals generation via the use of an API procedural method. The method fills in a user provided memory space by the produced texture data. The API procedural method takes as arguments the built in function name, texture resolution and optionally sampling rate, for each of the texture dimensions. The procedural API section is of infinite life span and is outside the scope of compilation.

Folding

The folding phase of the shading pipe collapses the collection of input program shaders into a single shader. Shader formals merging, lighting constructs unrolling, shadow samples resolution and derivative operator expansion are the primary folder tasks that are highlighted in the following discussion.

Formal parameters from each of the shaders in the group are renamed to avoid potential conflict. The handles of both the shader and the instance append originating shader formal names, and each is assigned a user settable default value. Ashli's API provides means for const'ing any of the shader collection formals. The folding stage evaluates programmer const setting and tags the parameters on a per shader instance basis. Formals const'ing is significant in relieving pressure off hardware shading register usage and is highly encouraged by the user. Shader formals merging results in a new data structure, which retargets original formals direction property e.g. *input* or *output* to its final folded orientation. All varying class formals are destined for mapping onto hardware input registers.

Surface and lights interact via *integrate* and *emission* lighting constructs, respectively. A light shader is assigned as either being *directional* or *non-directional*. The unrolling of

lighting constructs in the folder essentially matches any light integrate construct with one or multiple light emission constructs. The pair of integrate and emission constructs evaluated must be either directional or non-directional (the latter is for the case of ambient light) to yield any surface color contribution. Both shader and lighting function statement bodies are being searched for lighting constructs to perform code unrolling. Light emission constructs may well repeatedly interact with multiple surface lighting integrate constructs. Ashli's folder stage records internally the first instance of any light emission construct unrolled. It does so on a per light shader instance. Any subsequent interaction with a marked light emission construct is considered a reuse of the already expanded code. Consider the case of a surface lighting model, which uses both a diffuse and a specular component. Also, assume both are only affected by directional lights and that there are three spot light instances in the scene. The unrolling of the emission constructs of the spotlights is performed when evaluating the diffuse lighting component. The expanded code is fully reused though for the specular component. The reuse of expanded lighting constructs contributes significantly to resultant code efficiency, especially in the case of tens of light sources present in the scene.

Light, which cast shadows using a depth map access, is optionally assigned by the number of samples for performing *percentage closer filtering*. The folding stage expands the shadow map access expression of the light shader into a series of depth compare operations. The expansion takes the form of a sequence of relational operations. The depth component of the transformed position in light space is evaluated against the depth accessed from the depth map. The boolean result of each operation is accumulated across samples and finally normalized. The normalized value serves as the light color attenuation factor. The shadow samples resolution described demonstrates the need of converting a rather compacted nature of the source code and piecing it onto a meaningful execution form.

The common form of a derivative operator is to compute the difference of an expression across the parametric surface directions, *u* and *v*. Ashli supports derivative operators for input shader parameters. The application is expected to provide both the current and neighboring parameters (e.g. position and normal) along the parametric space axes, per vertex. Optionally, parametric deltas *du* and *dv* can be provided and used to derive the neighbors from the current texture coordinate set. Derivative operators are commonly used in displacement shaders for re-calculating the normal, once the current position has been perturbed. Essentially, the cross product of the position derivatives across both parametric directions is computed. The folding stage searches a displacement shader for a derivative operator in any of its expressions. Once a derivative operator has been located it is marked as a sink point. Previous statements relative to the sink point are triply expanded. The main branch of execution is the original code. The two additional branches are identical to the main one, but with the input parameters of concern substituted from current to the one of the parametric space neighbors. The derivative operator performs the difference between any of the parametric neighbors and the current input parameter. The derivative operator code expansion discussed above is destined to run serially on a graphics hardware processor, which is SIMD by its nature.

Multipass

A generic program, which runs on a general purpose CPU does not enjoy infinite resources, but is still able to gracefully handle very large pieces of code. It does so via spilling computational resources into virtual memory, which for all practical purposes are considered unlimited. Spilling in the CPU domain is completely transparent to the user and carried on by system software. In a similar spirit, one would have liked to throw at the graphics hardware a considerable large program and see it runs smoothly. Currently, shader processors, especially the pixel (fragment) one, of the graphics hardware have a finite and a fairly limited set of resources. e.g. input, constant, temporary and sampler registers as well as the total available execution code space. In addition, shading resources must be processor resident for the program to run at all and the onus of insuring resource limits conformance is on the programmer. It is fairly difficult to demand the artist to form or write a high level shader and attempt to predict hardware resource usage. It should also be noted that exceeding *any* of the resource caps is considered a non-recoverable compile error. For example, when program input registers required is larger than that supported natively by the hardware. Hence, to execute arbitrary computations, it is necessary to break them into multiple passes through the hardware. Ashli provides a seamless multipass solution to the application by breaking down a very large high level shader into multiple hardware shading segments. Each code segment produced fits the prescribed hardware resource paradigm.

The shader program in the intermediate code format is represented in an expression tree data structure. A tree node denotes an expression operator and node children are the expression operands. In Ashli's shading pipe the expression tree is constructed post the optimization phase, explained elsewhere. Each operator node of the tree maps directly onto an instruction in the hardware target language. The shader program expression tree is traversed for evaluating its *cost*. The cost is a direct representation of the hardware shading resources used. When the cost of the program tree exceeds the hardware resource limits, the tree is broken into sub trees, each fits into a pre-defined hardware resource cost metrics. Hardware resource constraints are set by the programmer using Ashli's API. The multipass solution for a given program is thereby adaptive based on the underlying hardware interface target. The algorithm Ashli uses to create multiple sub trees from a single tree is referenced in [Chan et al. 2002] paper. The sub trees generated from the multipass process are the source for generating hardware interface code segments.

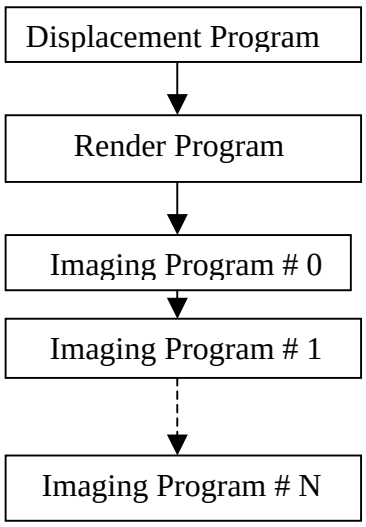
The output of multi shader segments from Ashli implies progressive rendering on the application side. Ashli returns to the user a pair of shader control and instruction data, per code segment. The control info, in the format of the fore mentioned formals, is parsed by the application for extracting shader runtime parameters and the assignment of intermediate shader results to temporary rendering buffers. Runtime shader parameters map onto hardware resources. Resources are set by the application using the shading state methods of the hardware interface. Scene geometry is rendered multiple times, each bound with the correlated segment of instruction data. Intermediate segment results, scalar or vector data type, are stored into a single or multiple IEEE 32 bit floating-point

component buffers, at the end of each rendering pass. As a result, multipass rendering on the graphics processor is performed with no loss of precision between passes. Subsequent rendering passes reuse the temporary buffer data, allocated from any of the previous passes, as a texture for restoring intermediate shader segment results. The final pass resolves appearance data into the visible buffer.

Finally, rendering that requires multiple passes tends to be limited by the shading computation, instead of either geometry processing or memory bandwidth. As a result, the cost of saving and restoring data between passes is relatively small. The save of a floating point vector variable into a temporary buffer will normally be pipelined and in most cases incurs little overhead. The restore of an intermediate value from a texture has its latency overhead for reaching the cache, but the implicit hardware scheduling of non-dependent instructions can be made to overlap with the fetch and alleviate much of the cost.

Imaging

Ashli supports image-processing operations on a post-rendered image in the context of an imager shader. Essentially, the resolved rendering image is retargeted as a texture, and is bound to a full window sized, quadrilateral geometry object. The imager shader performs filtering or color space operation in raster (or device) coordinate space. The overall modality for performing post-rendering image processing is to provide Ashli multiple shader programs in succession. The first one is an optional displacement program, which is composed of a displacement and a null surface shader. Next is a mandatory render program to include any combination of surface, light and volume shaders. Finally, a single or multiple optional imaging programs, each involves an imager and a null surface shader, are provided to perform a cascade of image processing operations. The following diagram illustrates the program modality flow into Ashli for post rendering image processing:



Ashli Program Modality

To conclude the imaging discussion we demonstrate a couple of typical imager shaders we have experimented with. They perform a water ripples and a waves type filter, respectively. The rendered image is provided to the shaders in the form of a texture formal. The examples illustrate point image processing, where each pixel of the image is altered based only on the original value of the pixel. Imager shaders could easily be extended to perform area filters, where neighboring pixels are weighted contributors to the final center pixel value. The screen shot of the imaging program, which uses the ripples shader, appears somewhere, in the following *results* section:

```
/*
 * ripples.sl -
 *
 * a simple minded implementation of a water ripples filter
 *
 * distorts the linear transfer function for radius with a cosine curve.
 * f(r) = r + cos(theta);
 *
 * Author: Arcot Preetham, ATI Research.
 */

imager
ripples(float waveFreq = 1;
        float waveAmp = 0.1;
        float time = 0;
        string imagename = "";)
{
    point PP = P - point(0.5, 0.5, 0);
    float P0 = xcomp(PP);
    float P1 = ycomp(PP);
    float radius = sqrt(P0 * P0 + P1 * P1);
    float cosangle = P0 / radius;
    float sinangle = P1 / radius;

    float waveangle = (radius - time) * waveFreq;

    waveangle = mod(waveangle, 2 * PI);
    float offset = 1 - cos(waveangle - PI);
    offset *= waveAmp;

    float newradius = radius + offset;

    Ci = color texture(imagename,
                    newradius * cosangle + 0.5, newradius * sinangle + 0.5);
}


/*
 * waves.sl -
 *
 * a simple minded implementation of a waves filter
 *
 * distorts the position mesh by a 2D noise vector field.
 * f(P) = P + noise();
 *
 * Author: Arcot Preetham, ATI Research
 */
```

```
imager
waves(float waveFreq = 1;
      float waveAmp = 0.1;
       float time = 0;
       string imagename = "";)
{
       point n = point noise(P * waveFreq - time);
       point newP = P + waveAmp * (2 * n - 1);
       Ci = color texture(imagename, xcomp(newP), ycomp(newP));
}
```

Results

We present Ashli's results in two forms. First, the source code of a sample program of a shader collection and the compiled segmented code are depicted. This is followed by a screen shot of a rendered geometry bound with the generated hardware shader code. Finally, additional rendering screen shots illustrate a broader scope of shader functionality, the use of shader instancing for multiple lights and in general addressing higher level of code complexity. All screen shots were generated on ATI's Radeon® 9700 graphics hardware.

Input Shader Program:

The input shader program consists of a surface shader and a signle distant light shader. The source of the shaders follows:

```
/*
 * wood.sl -
 *
 * RenderMan® wood shader.
 * Source: The Renderman Companion by Steve Upstill, Page 351.
 *
 */

surface
wood (
       float ringscale = 6;
       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;
       color ambientcolor = color(1);)
{
       point NN, V;
       point PP;
       float y, z, r;

       NN = normalize(N);
       V = -normalize(I);

       PP = transform("object", P);
       PP += noise(PP) * ampscale;
```

```
        /* compute radial distance r from PP to axis of "tree" */
        y = ycomp(PP);
        z = zcomp(PP);
        r = sqrt(y * y + z * z);

        /* map radial distance r into ring position [0, 1] */
        r *= ringscale;
        r += abs(noise(r));
        r = mod(r, 1);

        /* use ring position r to select wood color */
        r = smoothstep(0, 0.8, r) - smoothstep(0.83, 1, r);
        color Cw = mix(woodcolor1, woodcolor2, r);

        /* shade using r to vary shininess */
        Ci = Cw * (Ka * ambientcolor + Kd * diffuse(NN)) +
               (0.3 * r + 0.7)* Ks * specular(NN, V, roughness);
        Oi = Os;
}

/*
 * distantlight.sl –
 *
 * Standard distant light source for RenderMan® Interface.
 * (c) Copyright 1988, Pixar.
 *
 * The RenderMan® Interface Procedures and RIB Protocol are:
 * Copyright 1988, 1989, Pixar.  All rights reserved.
 * RenderMan(R) is a registered trademark of Pixar.
 */

light
distantlight ( float intensity = 1;
               color lightcolor = 1;
               point from = point "shader" (0, 0, -2);
               point to = point "shader" (0, 0, 0); )
{
    solar (to-from, 0) {
        Cl = intensity * lightcolor;
    }
}
```

Output Formals and Hardware Shader Code Segments:

Hardware Resource constraints per shader segment:

- *ALU instructions*: 63
- *Texture instructions*: 31
- *Input registers*: 10
- *Constant registers*: 32
- *Temporary registers*: 16
- *Sampler registers*: 16

Segment # 0

Formals:

```
s 0 3D Procedural noise
t 0 -1 Pobj
c 1 0 ampscale_0_0 0.038
c 1 1 ringscale_0_0 86.6
o 0 -1 r
```

ARB_fragment_program :

```
!!ARBfp1.0

# Instructions (Alu):      30
# Instructions (Tex):      2
# Registers    (Temp):     2
# Registers    (Constant): 4
# Registers    (Color):    0
# Registers    (TexCoord): 1
# Registers    (Texture):  1
# Registers    (Output):   1


ATTRIB Tex0 = fragment.texcoord[0];

PARAM Const0 = { 0, 1, 2, 0.8 };
PARAM Const2 = { 0, 0, 0, 1 };
PARAM Const3 = { 1.25, 0.83, 5.88235, 0 };

PARAM Const1 = program.local[1];

TEMP Temp0;
TEMP Temp1;

OUTPUT Output0 = result.color;

TEX Temp0, Tex0, texture[0], 3D;
MAD Temp0, Temp0.r, Const1.r, Tex0;
MOV Temp1.r, Temp0.g;
MOV Temp0.r, Temp0.b;
MUL Temp0.r, Temp0.r, Temp0.r;
MAD Temp0.r, Temp1.r, Temp1.r, Temp0.r;
RSQ Temp0.g, Temp0.r;
MUL Temp0.r, Temp0.r, Temp0.g;
MUL Temp0.r, Temp0.r, Const1.g;
MOV Temp1, Const2;
MOV Temp1.r, Temp0.r;
TEX Temp1, Temp1, texture[0], 3D;
MOV Temp0.g, Temp1.r;
ABS Temp0.g, Temp0.g;
ADD Temp0.r, Temp0.r, Temp0.g;
FRC Temp0.r, Temp0.r;
CMP Temp0.g, -Temp0.r, -Const0.g, Const0.g;
ADD Temp0.b, Temp0.r, -Const0.a;
CMP Temp0.b, Temp0.b, -Const0.g, Const0.g;
MUL Temp0.a, Temp0.r, Const3.r;
CMP Temp0.b, Temp0.b, Temp0.a, Const0.g;
CMP Temp0.g, Temp0.g, Temp0.b, Const0.r;
ADD Temp0.b, Const3.g, -Temp0.r;
CMP Temp0.b, Temp0.b, -Const0.g, Const0.g;
ADD Temp0.a, Temp0.r, -Const0.g;
CMP Temp0.a, Temp0.a, -Const0.g, Const0.g;
ADD Temp0.r, Temp0.r, -Const3.g;
MUL Temp0.r, Temp0.r, Const3.b;
CMP Temp0.r, Temp0.a, Temp0.r, Const0.g;
CMP Temp0.r, Temp0.b, Temp0.r, Const0.r;
ADD Temp0.r, Temp0.g, -Temp0.r;
MOV Output0, Temp0.r;

END
```

Formals:

```
t 2 -1 I
c 3 0 Ka_0_0 0.1
c 3 1 Kd_0_0 0.8
c 3 3 Ks_0_0 0.4
t 1 -1 N
v 1 -1 Os
s 0 2D Target Pass0_0
t 0 -1 Pproj
c 4 -1 ambientcolor_0_0 1 1 1 1
c 7 -1 from_1_0 0.1 0.5 1 1
c 3 2 intensity_1_0 1
c 5 -1 lightcolor_1_0 1 1 1 1
c 8 0 roughness_0_0 0.05
c 6 -1 to_1_0 0 0 0 1
c 2 -1 woodcolor1_0_0 0.96 0.77 0.47 1
c 1 -1 woodcolor2_0_0 0.68 0.49 0.15 1
o 0 -1 Ci
```

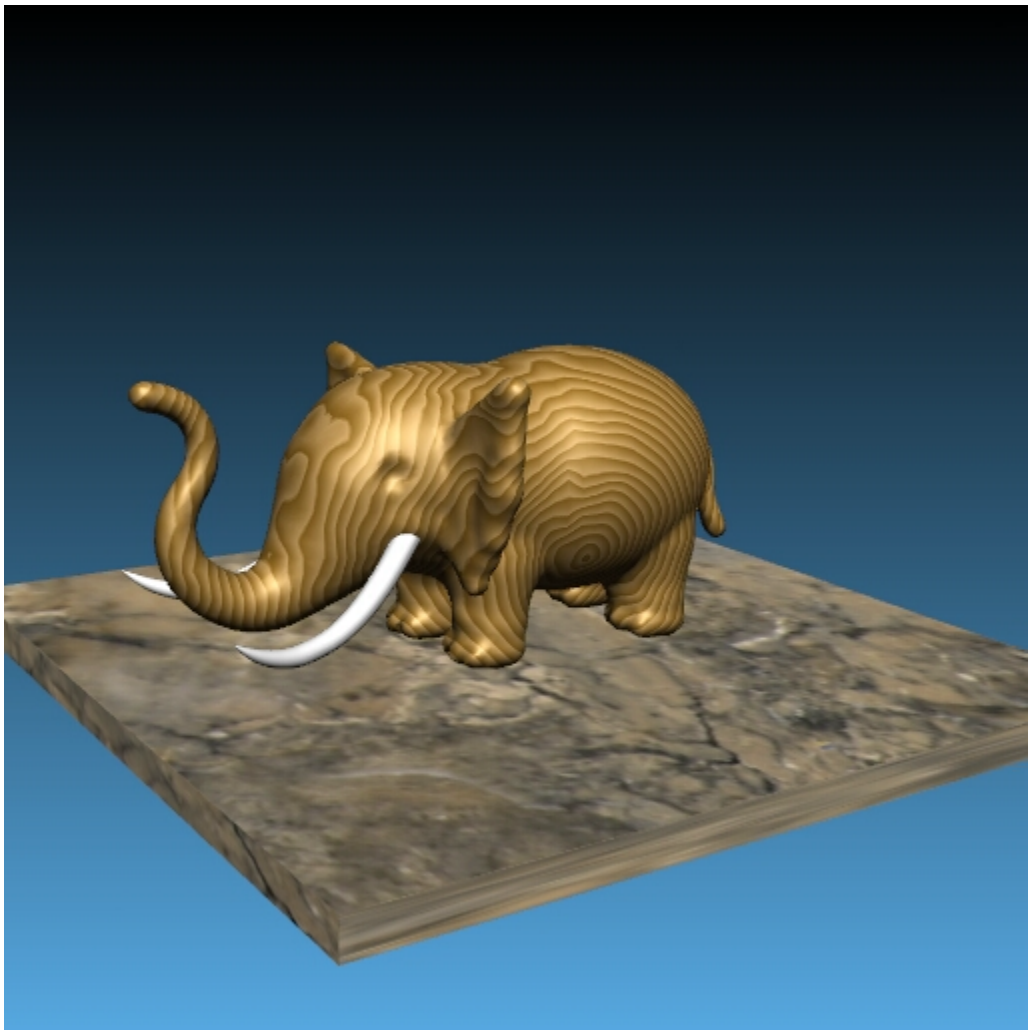
ARB_Fragment_Program :

```
!!ARBfp1.0

# Instructions (Alu):      38
# Instructions (Tex):      1
# Registers    (Temp):     8
# Registers    (Constant): 9
# Registers    (Color):    1
# Registers    (TexCoord): 3
# Registers    (Texture):  1
# Registers    (Output):   1

ATTRIB Color1 = fragment.color.secondary;

ATTRIB Tex0 = fragment.texcoord[0];
ATTRIB Tex1 = fragment.texcoord[1];
ATTRIB Tex2 = fragment.texcoord[2];

PARAM Const0 = { 3, 0, 0.3, 0.7 };

PARAM Const1 = program.local[1];
PARAM Const2 = program.local[2];
PARAM Const3 = program.local[3];
PARAM Const4 = program.local[4];
PARAM Const5 = program.local[5];
PARAM Const6 = program.local[6];
PARAM Const7 = program.local[7];
PARAM Const8 = program.local[8];

TEMP Temp0;
TEMP Temp1;
TEMP Temp2;
TEMP Temp3;
TEMP Temp4;
TEMP Temp5;
TEMP Temp6;
TEMP Temp7;

OUTPUT Output0 = result.color;

TXP Temp0, Tex0, texture[0], 2D;
MOV Temp1.r, Color1.r;
```

```
LRP Temp2, Temp0.r, Const1, Const2;
MUL Temp3, Const3.b, Const5;
DP3 Temp0.g, Tex1, Tex1;
RSQ Temp0.g, Temp0.g;
MUL Temp4, Temp0.g, Tex1;
ADD Temp5, Const6, -Const7;
DP3 Temp0.g, -Temp5, -Temp5;
RSQ Temp0.g, Temp0.g;
MUL Temp6, Temp0.g, -Temp5;
DP3 Temp0.g, Temp4, Temp6;
MAX Temp0.g, Const0.g, Temp0.g;
MUL Temp6, Temp3, Temp0.g;
MUL Temp6, Const3.g, Temp6;
MAD Temp6, Const3.r, Const4, Temp6;
MAD Temp0.r, Const0.b, Temp0.r, Const0.a;
DP3 Temp0.g, -Temp5, -Temp5;
RSQ Temp0.g, Temp0.g;
DP3 Temp0.b, Tex2, Tex2;
RSQ Temp0.b, Temp0.b;
MUL Temp7, Temp0.b, Tex2;
MAD Temp5, Temp0.g, -Temp5, -Temp7;
DP3 Temp0.g, Temp5, Temp5;
RSQ Temp0.g, Temp0.g;
MUL Temp5, Temp0.g, Temp5;
DP3 Temp0.g, Temp4, Temp5;
MAX Temp0.g, Const0.g, Temp0.g;
LG2 Temp0.g, Temp0.g;
RCP Temp0.b, Const8.r;
MUL Temp0.g, Temp0.g, Temp0.b;
EX2 Temp0.g, Temp0.g;
MUL Temp3, Temp3, Temp0.g;
MUL Temp3, Const3.a, Temp3;
MUL Temp0, Temp0.r, Temp3;
MAD Temp0, Temp2, Temp6, Temp0;
MOV Temp0.a, Temp1.r;
MOV Output0, Temp0;

END
```

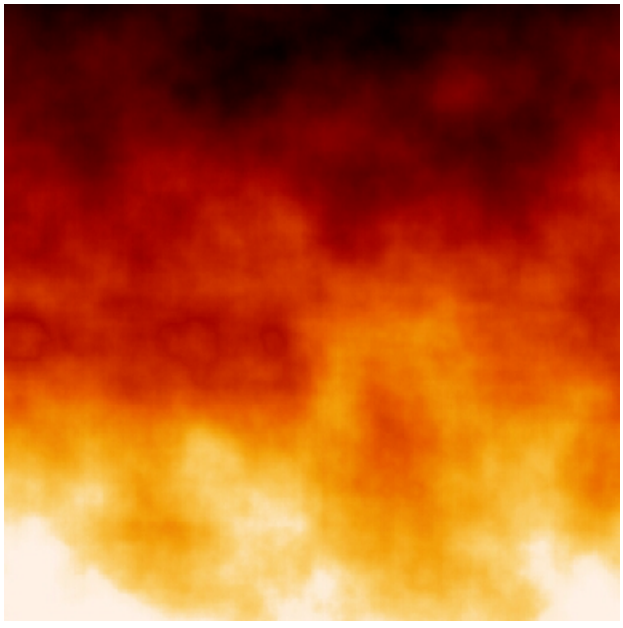
Screen Shots:

The elephant in the following screen shot was rendered in two passes using the above shader segments with a total of 71 instructions.

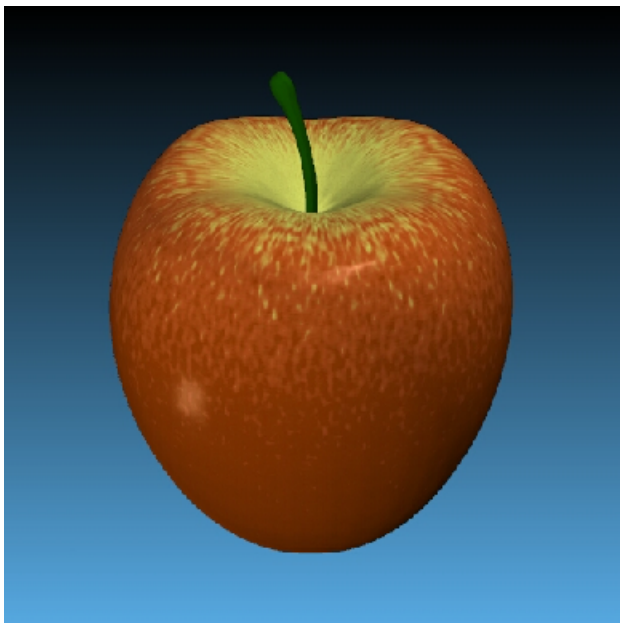


- *Shader*: *brick* surface with one distant light
- *Source*: [Olano et al. 2002]
- *Output*: 5 segments with a total of 171 instructions

- *Shader*: *flame* surface
- *Source*: http://www.renderman.org/RMR/Shaders/KMShaders/KMFlame.sl
- *Output*: 3 segments with a total of 201 instructions



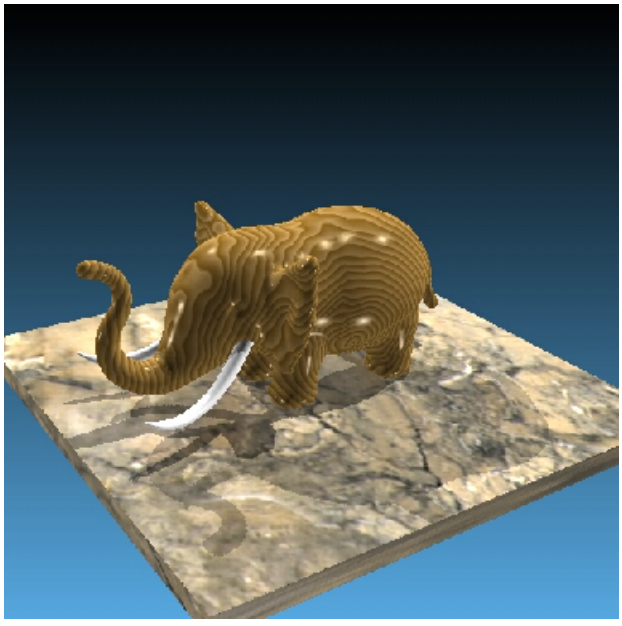
- *Shader*: *redapple* surface with two distant lights
- *Source*: http://www.renderman.org/RMR/Shaders/JMShaders/JMredapple.sl
- *Output*: 4 segments with a total of 193 instructions



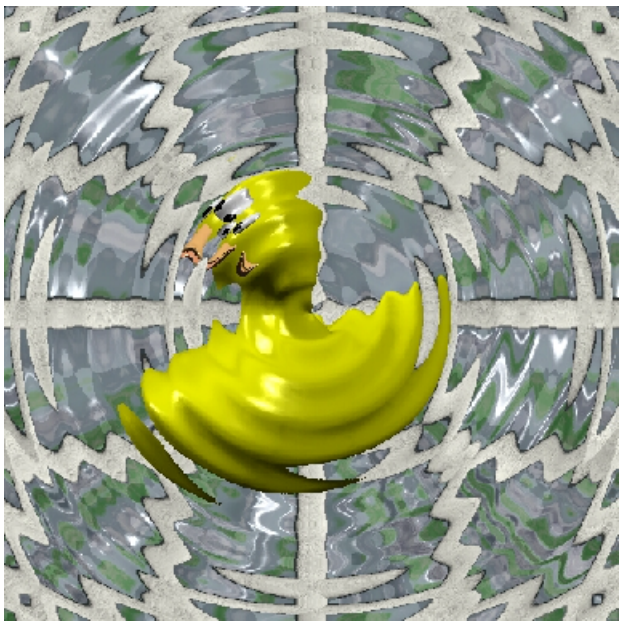
- *Shader*: matte surface using *windowlight* light
- *Source*: http://www.renderman.org/RMR/Shaders/BMRTShaders/windowlight.sl
- *Output*: 3 segments with a total of 118 instructions

- *Shader*: *potato* surface with two distant lights
- *Source*: http://dspace.dial.pipex.com/adrian.skilling/shaders/code/potato.sl
- *Output*: 5 segments with a total of 187 instructions

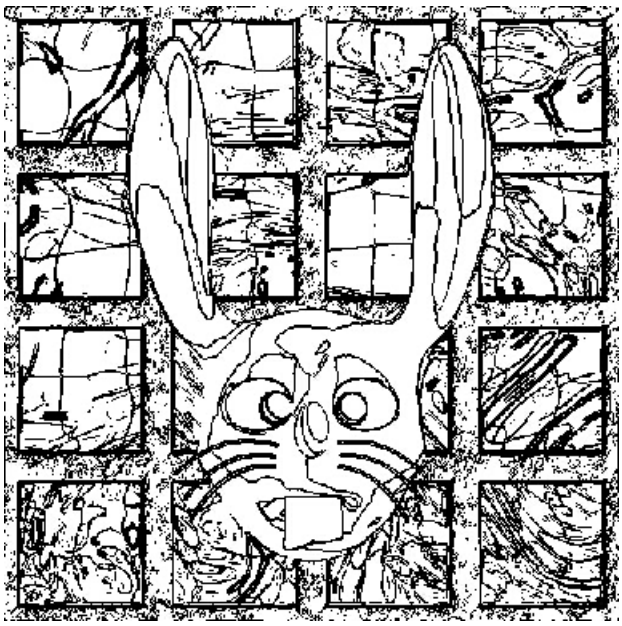
- *Shader*: *wood* surface with five distant lights, two of them casting shadows
- *Source*: [Upstill 1990]
- *Output*: 4 segments with a total of 204 instructions



- *Shader*: *ripples* imager shader
- *Source*: water ripples filter (internal)
- *Output*: 1 segment with a total of 31 instructions



- *Shader*: *edge detection* imager shader
- *Source*: Sobel edge detection filter (internal)
- *Output*: 1 segment with a total of 45 instructions



References

[Olano et al. 2002] Marc Olano, John C. Hart, Wolfgang Heidrich, Michael McCool: *Real-Time Shading*, 2002 (A K Peters Ltd)

[Chan et al. 2002] Eric Chan, Ren Ng, Pradeep Sen, Kekoa Proudfoot, Pat Hanrahan: *Efficient Partitioning of Fragment Shaders for Multipass Rendering on Programmable Graphics Hardware*, 2002 Graphics Hardware (pp. 1-11)

[Apodaca & Gritz 2000] Anthony A. Apodaca, Larry Gritz: *Advanced RenderMan: Creating CGI for Moton Pictures*, 2000 (Morgan Kaufmann)

[Upstill 1990] Steve Upstill: *The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*, 1990 (Addison Wesley)