

Chapter 3. Shading Compilers

Avi Bleiweiss

Shading Compilers

Avi Bleiweiss
ATI Research, Inc.

Introduction

Graphics hardware evolution in the past year has been highlighted by improved shading resources, the introduction of dynamic flow control and the emergence of multi-GPU system configurations. Executed program length increased substantially and register availability became more affordable. As a result, compiler support for segmenting a shading program to fit hardware constraints has diminished and is mostly required on low cost hardware platforms. The presence of dynamic flow control functionality has made true loops and conditionals, nested or un-nested, first class citizens in a shader description. Dynamic branching brings speed-ups to many algorithms that contain early out opportunities, while also simplifying shader programmability. Data dependent flow control might first be looked at odds with the SIMD architecture nature of the GPU. Non-linear flow control usually results in poor efficiency manifested by idling internal processors and other resources. Shading compilers however have been alleviating these shortfalls by transforming data dependent structures into linear flow control nodes operating on large data sets. Finally, multi-GPU load partition schemes are now in place to better address performance scalability in several application domains. Both image and time distribution methods have been deployed and leverage of the significant high bandwidth of the relatively newly introduced PCI-Express® [PCIExpress 2004] bus architecture.

The remainder of the notes describes selected topics of the rather broad shading compilers gamut. Shading technology has been evolved in recent years where the set of challenges identified for compilation became evidently devisable into *high* and *low* level spans. Shading representation with improved interfaces to the rendering sub system, toning down shading language differences, shader partition across GPUs, and the remapping of GPU processors are some of the matters considered high level. Traditional and the more conventional compiler technology aspects e.g. optimization and instruction scheduling, require more intimate hardware knowledge. The discussion hereafter is mostly concerned with high level shading compiler subjects yet it is result oriented and demonstrates compiler code emission. The topics are organized as follows: *Shader Representation*, *Multi-GPU Shader Partition*, *Remapping GPU Processors*, and finally a *Source Level Debugger*. In most of the sections code snippets of source representation (or shading language), assembly level target and rendered image snapshots are attached to improve clarity. Throughout the notes references are made to *Ashli* [ASHLI 2003], a shading technology toolkit developed at ATI for the past several years.

Shader Representation

Graphics hardware has evolved from a fixed function pipeline into a programmable *vertex* and *pixel* processors. Initially, programmability was exposed in low level assembly standards, now commonly referred to as Shader Model 2.0. Accessibility was fairly limited to users, who traditionally were accustomed to higher level shading abstractions and increased complexity. The industry has been properly reacting to this interface void by introducing graphics hardware shading languages. Most influential are Microsoft DirectX's *HLSL* [HLSL 2004], OpenGL's *GLSL* [GLSL 2004] and NVidia's *Cg* [Cg 2004]. The languages inherited some core functionality from the well established Pixar's RenderMan® [RenderMan® 1986] shading language, and thereby resemble many similarities across them. Nevertheless, differences and ties to either a specific graphics API or a hardware platform have made them take their own development path and hence supporting tools. Ashli was one of the first shading tools to take in multi lingual shader descriptions at arbitrary level of complexity and produce platform independent assembly code.

The reality of multiple shading languages has led the graphics community strive at defining higher level abstraction standards, for which differences are as much seamless. Hiding language details is desirable to users, who are less intimated with hardware knowledge and need not be concerned with fast paced GPU architectural evolution. These standards have the added benefit of addressing broader shading to overall rendering and appearance control interfaces. One such standard is Microsoft's *Effect* format. Effect had practically become the de-facto portable shading representation in both the gaming and digital content creation markets. Effects can be written with a high level language or with shader assembly semantics. The Effect format views a high level shader as a collection of *parameters* and *functions*. The original Effect internals call for types and expressions to be a valid HLSL shader reference. However, the format is generic enough to embed shaders of any other language. The Effect encapsulates rendering state and any combination of vertex and pixel shaders to define a rendering style. It is easily extensible to map shaders on newly evolved graphics hardware processors as they become realizable e.g. *tessellation* and *geometry*. Shading language abstraction is only one part of a solution to an inevitable programmability concern. It provides harnessing to the growing GPU power without the need of in depth processor level coding knowledge. There still remains the constraint of having a shading language bound to a particular graphics API. Cross language translation support by compilers is one way to circumvent the API barrier.

Ashli toolkit has been recently augmented to support multi-lingual Effect format representation. *AshliFX* is the relatively new software component introduced for this purpose. In addition, Ashli provides the generation of GLSL shaders from any of HLSL and RenderMan® descriptions, primarily for smoother inter language transition. The remainder of this section introduces AshliFX, its API and main software components. Then, a sample GPU based image processing library – *AshliDI* - illustrates facilitating AshliFX in hiding any notion of shading language and hardware specifics. The discussion hereafter assumes some familiarity with the Effect format basics.

AshliFX

The following diagram in Figure 1 depicts AshliFX interfaces and its interaction with an application attached to Ashli:

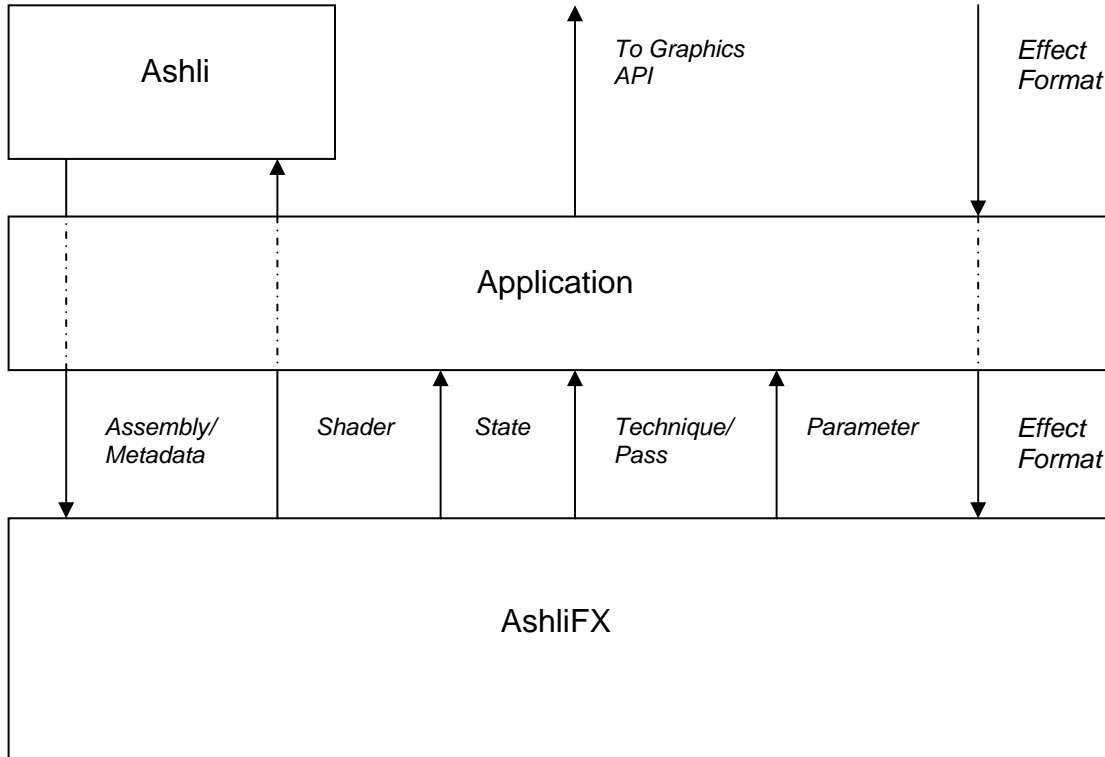


Figure 1: Application in an AshliFX/Ashli Framework

An application takes in an Effect format and passes it to AshliFX. The Effect abstraction may be bound to any of HLSL or GLSL. Binding is either explicit or implicit, where for the latter the representation optionally embeds a language pragma. The Effect discovery mechanism provided is hence fully automatic and the user just needs to query the bound language any time post parsing. The Effect format is further rendered into its logical components: *parameters*, *functions*, *techniques*, *passes* and *annotations*. Components are exposed to the user via API query methods. They are accessed being qualified by either a handle or by name. The handle is merely a zero based running index, tied to component order of appearance inside the Effect format. A technique is made up of one or multiple passes, each containing rendering state assignments. At a minimum, pipe state must have one of vertex or pixel (or both) shaders assigned. Shaders are identified with a compile target and an entry point function name, or they could be assigned an assembly block. In addition, entry point default values may optionally be assigned to uniform arguments, once present. AshliFX provides an *Effect State Observer*, a user implemented interface

that furnishes callbacks into an application for setting graphics API state. The user attaches a state observer and queries to either parameter or pipe state invokes one of the callback state methods. Callback methods are provided for *light*, *material*, *render*, *sampler*, *shader* and *transform* categories and all have the same signature:

```
void set{Category}State({Category}State state, int index, const char* value);
```

An Effect observer callback returns an enumerated type that identifies an API neutral pipe state item, an optional index indicating a particular state within an array of Effect states and an assigned value. For example the states *ZEnable*, *LightAmbient[1]* and *VertexShaderConstant[3]* return indices of -1, 1 and 3, respectively. Enumerated Effect pipe state is graphics API independent and an application is free to map them onto any of Microsoft DirectX or OpenGL state semantics. AshliFX validates Effect state keys for each pass to match Microsoft Effect States definitions.

AshliFX shader interface furnishes the query of Effect embedded high level shader(s) or assembly block(s), per pass. API methods are provided for retrieving compilation target and extracted shader(s) for each of the GPU processors. AshliFX composes vertex and pixel high level shaders out of global parameters and a collection of functions, amongst them the entry point as specified by the shader assigned compile expression. In constructing the shaders AshliFX filters out non-compile related Effect data. Some of the info excluded from the shaders coalesced includes string and texture parameters, annotation(s) and semantic, the latter in the case of a GLSL binding. Effect extracted shaders are passed onto Ashli for compilation and the generated metadata and hardware assembly shaders are deposited back into AshliFX. Ashli metadata is parsed internally by AshliFX and constant and sampler registers and their assigned parameters are mapped onto Effect shader states. The user then need not be concerned with parsing Ashli metadata - pass state provides all the necessary pipe settings for rendering. Optional defaults assigned to entry point uniform parameters (HLSL) can be further queried by either a handle or by name. Returned parameter name, default value and number of elements directly match Ashli's set default methods signature. Each default parameter is further expected to be const tagged for code generation efficiency.

The Effect format also defines annotation framework. Annotations are user specific data that can be attached to any of technique, pass, or a parameter. One or multiple annotations can be grouped together to add information to individual components, in a flexible way. The information can be read back and used any way the application desires. An annotation can also be added or edited dynamically in AshliFX. Annotations are primarily useful for interface customization. Typically, they will incorporate a template for automatically depicting user interface elements, such as sliders with delimiter range values. Similarly, they can hold binding information to geometry and texture objects. AshliFX API methods allow for querying annotation type, name and assigned data value by either a handle or by name.

Figures 2 and 3 illustrate an AshliFX/Ashli workflow for an HLSL bound Effect format, which compiles per pass onto OpenGL ARB fragment code. It demonstrates platform API independence regardless of the native language binding. Both code snippets and a

snapshot of an image, rendered inside a Maya® viewport on ATI X 800 series hardware, are shown:

```
// texturedphong.fx (only shown pixel parameters and entry point)

float4 fvAmbient <
  string UIName = "fvAmbient";
  string UIWidget = "Color";
  bool UIVisible = true;
  > = float4( 0.37,0.37,0.37, 1.00 );

float4 fvSpecular <
  string UIName = "fvSpecular";
  string UIWidget = "Color";
  bool UIVisible = true;
  > = float4( 0.49,0.49,0.49, 1.00 );

float4 fvDiffuse <
  string UIName = "fvDiffuse";
  string UIWidget = "Color";
  bool UIVisible = true;
  > = float4( 0.89,0.89,0.89, 1.00 );

float fSpecularPower <
  string UIName = "fSpecularPower";
  string UIWidget = "Numeric";
  bool UIVisible = true;
  float UIMin = 1.00;
  float UIMax = 100.00;
  > = float( 25.00 );

texture base_Tex <
  string ResourceName = "Textures\\Fieldstone.tga";
  >;

sampler2D baseMap = sampler_state {
  Texture = (base_Tex);
  AddressU = WRAP;
  AddressV = WRAP;
  MinFilter = LINEAR;
  MagFilter = LINEAR;
  MipFilter = LINEAR;
};

struct PS_INPUT {
  float2 Texcoord : TEXCOORD0;
  float3 ViewDirection : TEXCOORD1;
  float3 LightDirection : TEXCOORD2;
  float3 Normal : TEXCOORD3;
};

float4
TexturedPhongPS( PS_INPUT Input ) : COLOR0
{
  float3 fvLightDirection = normalize( Input.LightDirection );
  float3 fvNormal = normalize( Input.Normal );
  float fNDotL = dot( fvNormal, fvLightDirection );
  float3 fvReflection = normalize( ( ( 2.0f * fvNormal ) * ( fNDotL ) ) - fvLightDirection );
  float3 fvViewDirection = normalize( Input.ViewDirection );
  float fRDotV = max( 0.0f, dot( fvReflection, fvViewDirection ) );
  float4 fvBaseColor = tex2D( baseMap, Input.Texcoord );
  float4 fvTotalAmbient = fvAmbient * fvBaseColor;
  float4 fvTotalDiffuse = fvDiffuse * fNDotL * fvBaseColor;
  float4 fvTotalSpecular = fvSpecular * pow( fRDotV, fSpecularPower );
  return( saturate( fvTotalAmbient + fvTotalDiffuse + fvTotalSpecular ) );
}

technique TexturedPhong {
  pass Pass0 {
    VertexShader = compile vs_2_0 TexturedPhongVS();
    PixelShader = compile ps_2_0 TexturedPhongPS();
  }
}

// metadata

begin fragment
t 0 -1 _T0_
t 1 -1 _T1_
t 2 -1 _T2_
t 3 -1 _T3_
c 4 -1 fvAmbient_1_0 0.37 0.37 0.37 1
c 1 -1 fvSpecular_1_0 0.49 0.49 0.49 1
c 3 -1 fvDiffuse_1_0 0.89 0.89 0.89 1
c 2 0 fSpecularPower_1_0 25
s 0 2D File baseMap
o 0 -1 _CO_
end

// ARB fragment program

!!ARBfp1.0

# Instructions (Alu): 28
# Instructions (Tex): 1
# Registers (Temp): 4
# Registers (Constant): 5
# Registers (TexCoord): 4
# Registers (Texture): 1
# Registers (Output): 1

ATTRIB Tex0 = fragment.texcoord[0];
ATTRIB Tex1 = fragment.texcoord[1];
ATTRIB Tex2 = fragment.texcoord[2];
ATTRIB Tex3 = fragment.texcoord[3];

PARAM Const0 = { 1, 0, 2, 0 };

PARAM Const1 = program.local[1];
PARAM Const2 = program.local[2];
PARAM Const3 = program.local[3];
PARAM Const4 = program.local[4];

TEMP Temp0;
TEMP Temp1;
TEMP Temp2;
TEMP Temp3;

OUTPUT Output0 = result.color;

TEX Temp0, Tex0, texture[ 0 ], 2D;
DP3 Temp1.r, Tex3, Tex3;
RSQ Temp1.r, Temp1.r;
MUL Temp1, Temp1.r, Tex3;
DP3 Temp2.r, Tex2, Tex2;
RSQ Temp2.r, Temp2.r;
MUL Temp2, Temp2.r, Tex2;
DP3 Temp3.r, Temp1, Temp2;
MUL Temp1, Const0.b, Temp1;
MAD Temp1, Temp1, Temp3.r, -Temp2;
DP3 Temp2.r, Temp1, Temp1;
RSQ Temp2.r, Temp2.r;
MUL Temp1, Temp1, Temp2.r, Temp1;
DP3 Temp2.r, Tex1, Tex1;
RSQ Temp2.r, Temp2.r;
MUL Temp2, Temp2.r, Tex1;
DP3 Temp1.r, Temp1, Temp2;
MAX Temp1.r, Const0.g, Temp1.r;
POW Temp1.r, Temp1.r, Const2.r;
MUL Temp1, Const1, Temp1.r;
MUL Temp2, Temp3.r, Temp0;
MAD Temp1, Const3, Temp2, Temp1;
MAD Temp0, Const4, Temp0, Temp1;
ADD Temp1, Temp0, -Const0.r;
CMP Temp1, Temp1, Temp0, Const0.r;
CMP Temp0, -Temp0, Temp1, Const0.g;
MOV Output0, Temp0;

END
```

texturedphong Effect
HLSL bound

Ashli metadata and
OpenGL ARB fragment program

Figure 2: HLSL bound Effect compiled to OpenGL ARB Fragment Program

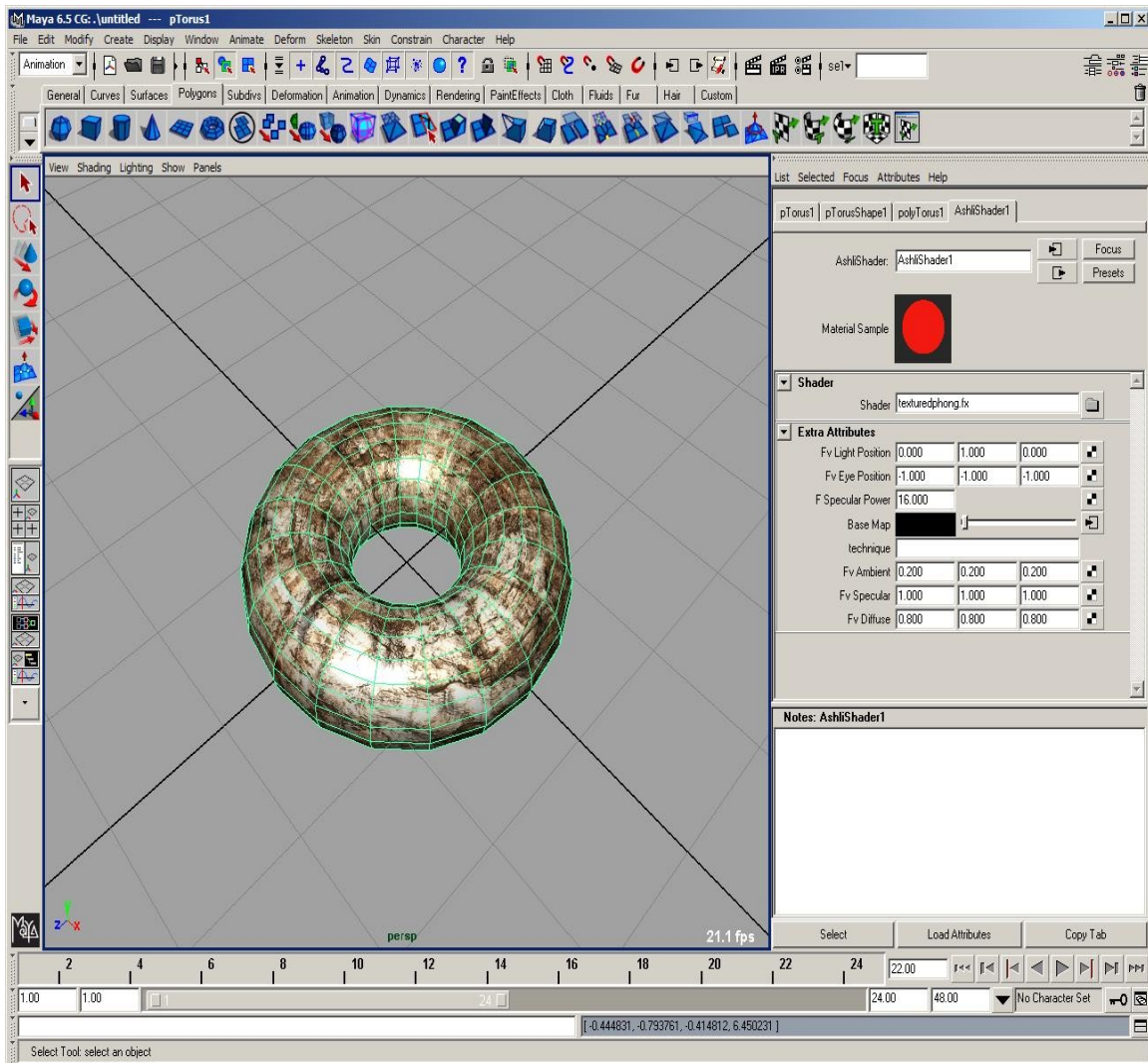


Figure 3: AshliFX/Ashli Workflow in a Maya Plugin Context

AshliDI

Creating high performance and accurate image processing solutions generally requires a deep knowledge of graphics hardware API. A development environment, for which users can take full advantage of the ever growing GPU power seamlessly, is highly desirable. Apple's Core Image [CoreImage 2004] is one platform that leverages programmable graphics hardware by relieving users from any pixel level coding burden. Image processing programs are expressed at a simple high level compact code and the underlying library takes care of GPU optimization and precision considerations. The end result produces detail, quality and range comparable to a CPU, but at a much higher interactive responsiveness. This section introduces *AshliDI* - Ashli for Digital Imaging - an image processing staging library that encapsulates graphics shading language and

hides it from the programmer. Figure 4 illustrates AshliDI interfaces and connectivity in an AshliFX/Ashli framework:

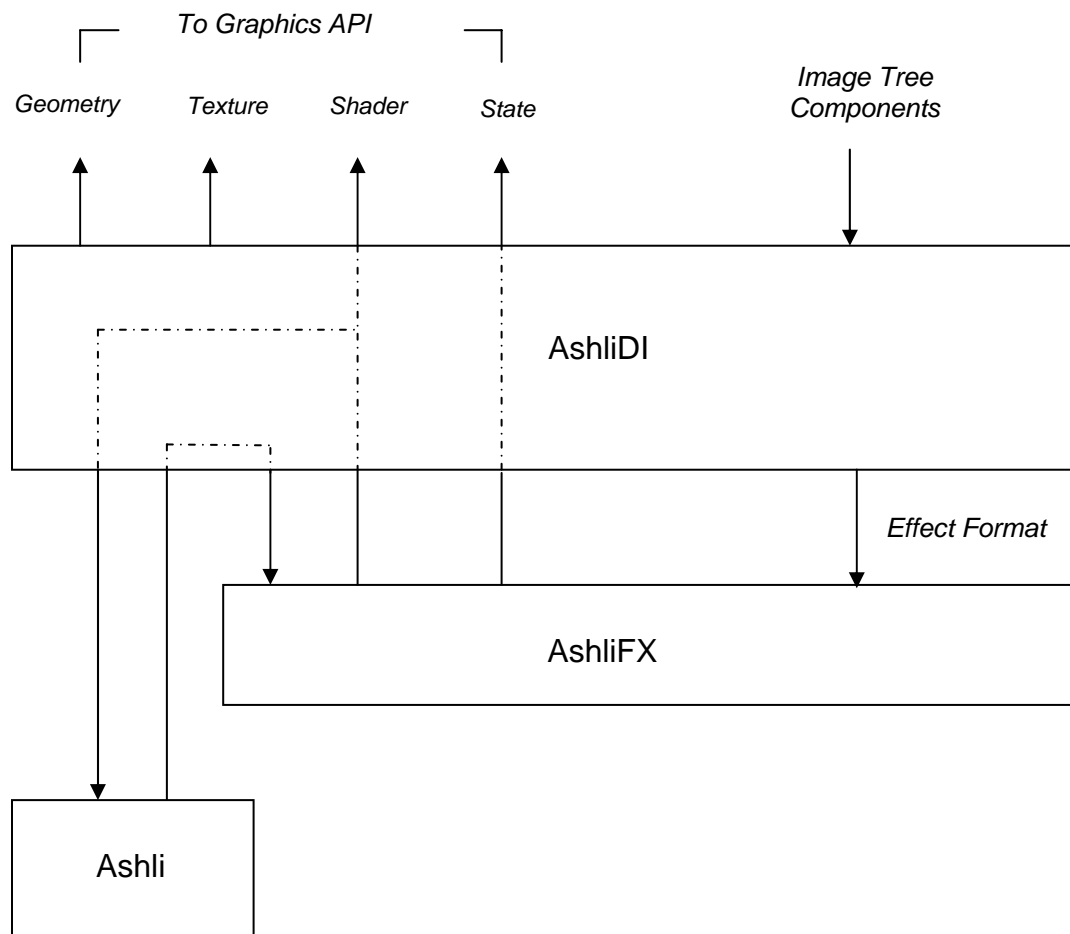


Figure 4: AshliDI High Level Overview

AshliDI is intended for imaging markets that are intensively seeking the use of a GPU, and include digital photography, film post production, scientific visualization and desktop publishing. AshliDI is graphics hardware API agnostic and caters to both Microsoft DirectX and OpenGL. It takes in conventional image processing abstractions in the form of *image tree* components. AshliDI provides an API to construct the image tree procedurally without requiring any shading language knowledge. At the tree nodes reside image *operators* that include geometry and color space transforms, point and area filters, compositing, and halftones. Nodes can optionally hold processing attributes for defining type conversion and region of interest. Leaf nodes are source image(s) provided by the user. An operator node essentially maps onto a render-to-texture pass in the graphics context. The traversal of the image tree results in a sequence of rendering passes for

which the previously rendered texture is a leaf node to a top level node. The final image is produced at the root of the tree. AshliDI performs its processing using 32-bit IEEE floating point math. This means that a considerable deep image tree with high bit accurate image leaves can perform its processing steps with no loss of precision. The library is designed to cope with a single common image file format, anticipating all file format conversion to occur outside its scope. ILM's OpenEXR [OpenEXR 2004] is a high dynamic range image file format that supports both 16 and 32 bit floating point. OpenEXR fits well AshliDI image processing modality and overall design for being platform and system independent.

The image tree is first converted into an Effect format representation. The Effect generated is bound to either HLSL or GLSL, based on the graphics platform. The tree is initially searched for dependencies and is broken into separate rendering entities for which connectivity is loose. Trees are in essence directed graphs without cycles. Leaf sub trees take as inputs source images provided by the user. Figure 5 depicts an image tree with three source images as inputs. Images 0 and 1 take a unique path until blended with the transformed image 2. Sub trees for intermediate image generation are highlighted:

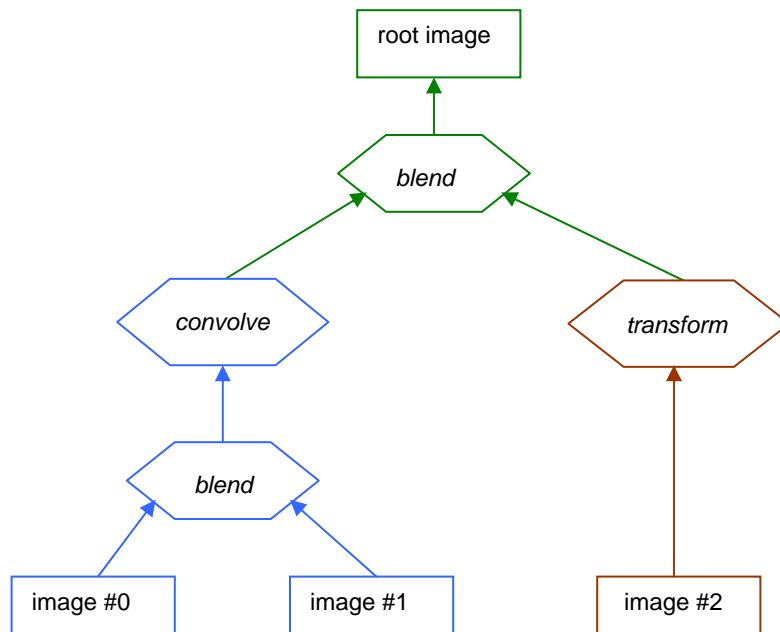


Figure 5: AshliDI Sample Image Tree

Sub trees are then mapped onto Effect techniques. The relation between the sub trees in terms of render order and the passage of intermediate image results from one technique to another are specified in the Effect format by means of annotations. Operator nodes of the sub tree are designated each to an Effect pass. Node type of operation and its attached attributes map onto pass derived pipe state. The Effect data structure constitutes the rendering format of the transformed image tree. AshliDI uses then the AshliFX/Ashli

framework described in the previous section for compiling the Effect representation and making its components available for rendering. AshliDI query interfaces include *geometry*, *texture*, *state* and *shader* sections. A rectangle geometry entity delimits the 2D region of interest specified by the user. Source leaf node images, in a precision prescribed by the user, are returned to the user for the first pass of a leaf sub tree. Intermediate image results, inputs to non leaf sub trees, are the render-to-texture products of the previous pass. Any of high level HLSL or GLSL shaders, or post compilation assembly entities are provided in the shader interface. Finally, pipe state per pass derived from the image operator node is available in the state interface segment.

AshliDI supports a subset of image operators based on market importance. The design is scalable to accommodate additional operators as they deem necessary. AshliDI is a staging library independent of the graphics API and provides a plug-in style framework to fit into an imaging application like Adobe's Photo-Shop®.

Multi-GPU Shader Partition

The increased bandwidth of the PCI Express® bus architecture, reaching throughputs of up to 4GBytes/sec, has made multi GPU configurations more affordable for scalability and increased performance purpose. Expedient peer-to-peer data transfers of partially rendered scenes and synchronization state were the prime motivation for exploring concurrency across multiple GPUs. *Image* and *time* based subdivision methods have been most popular in devising load balance criteria of multiple GPU architectures. Image or screen based tiling can be made fairly adaptive and achieve a relatively good distribution of the load. Similarly, as long as frames are consistent, altering them across yields a reasonable division of labor amongst GPUs. For the sake of the following discussion scene geometry and textures are assumed to be replicated on all GPUs. This simplifies the software model for concurrent GPUs and avoids unnecessary copy overhead. The current GPU architecture exploits the rather straight forward *vertex-pixel* performance modality. The vertex shader is a one-to-one vertex map and holds then an *amplification* factor of one. Performance wise, the vertex shader execution rate is fairly predictable, barring flow control and cache behavior. The *rasterizer* however, is a one vertex (or a few vertices) to many pixels type of generator. Hence, its multiplication ratio is any number based on the primitive screen space area. In the present GPU framework the vertex and pixel shaders are separate entities, each with their own set of resources, running multiple execution threads. In general, there is a higher concurrency degree in the pixel shader compared to the vertex shader. Vertex limited scenes would have then very little performance gain when run on multiple GPUs - they both have to render the same geometry. Pixel shader bound rendering context, for which multiple GPU architectures are intended for in the first place, are more likely to exhibit speedups in ranges up to as close to the optimal linear scale. The vertex-pixel performance modality discussed so far is rapidly turning though into a more involved *geometry-pixel* one. Figure 6 illustrates the graphics pipeline in both the vertex-pixel and geometry-pixel forms. The geometry-pixel paradigm is of more pipe stages each with varying dynamics and might require the reassessing of load distribution considerations in parallel GPUs framework.

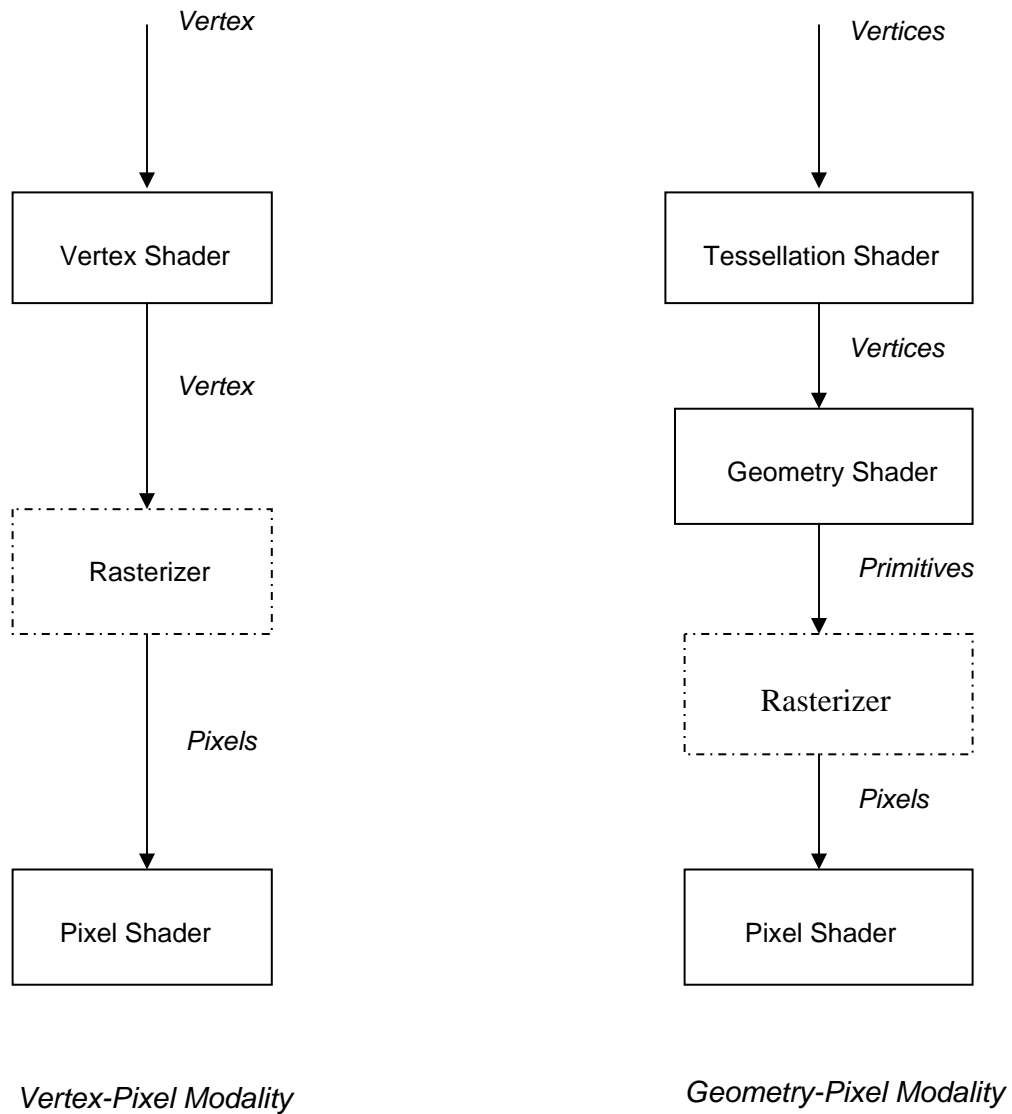


Figure 6: GPU Vertex-Pixel vs. Geometry-Pixel Modality

The input to the geometry-pixel pipe is a collection of vertices. The *tessellation* shader takes in control points of an implicit surface representation and performs a refinement process that yields smoother geometry. The input surface format is in *parametric* space and embeds both positional and topological information. The finer level of detailed geometry produces vertices of count larger compared to the input collection size. The tessellation engine hence departs from the traditional one-to-one predecessor vertex shader and forms a geometry amplification stage at the top of the graphics pipeline. A single vertex input implies a vertex shader behavior of the tessellation unit, identical to the one in the vertex-pixel modality. The tessellation shader passes its results to the

geometry shader, the next pipeline stage. The geometry shader takes in vertices grouped into primitives e.g. one vertex for a point, two vertices for a line, and three vertices for a triangle. The geometry unit is capable of outputting multiple vertices forming a single selected primitive topology. Topology options are one of a *triangle-strip*, a *line-strip* and a *point list*. Any invocation of the geometry shader could potentially vary the number of primitives emitted. As such the geometry stage introduces a manifold multiplicity form mid pipe. The rasterizer model and onwards remains for the most part identical to the one exists in the vertex-pixel modality. Geometry-pixel shader stages all share the same processing engines and resources. As a result connectivity is much more transparent amongst the shader units and outputs from one can be redirected as input to another almost seamlessly. The introduction of both the tessellation and the geometry shaders with intrinsic expansion properties clearly changes pipe dynamics. This potentially opens up alternative schemes for partitioning load across GPUs. Next, a displaced and motion blurred geometry will be used as a walkthrough example to further illustrate internal pipe interaction.

A subdivision patch of a triangular control net with a topological valence of six vertices is assumed the input to the tessellation shader. Subdivision surfaces are evaluated recursively and the regeneration factor of vertices is fairly predictable for every step. Figure 7 demonstrated a triangular net following one refinement step:

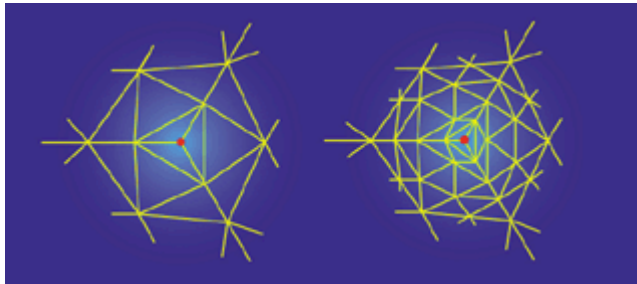


Figure 7: Refinement of a Triangular Control Net [Sharp 2000]

The number of refinement steps applied to the surface varies from one patch to another. It is typically to require a finer level of detail once the geometry is displaced. Displacement map is performed in the tessellation shader after the final evaluation step. Vertex displacement computation is conducted in parametric space, where neighbor tangential derivatives are available for accurate normal recalculation. The tessellation shader provides its produced vertices on to the geometry shader. The geometry shader performs motion blur operation on the previously displaced geometry in an orthogonal manner. Triangle primitives are the input anticipated at the geometry unit. Let's define time 0 at which a hypothetical camera aperture opens on a scene, and time *aperture* is the time at which the hypothetical camera aperture closes. Input triangles to the geometry shader are considered at time 0 . The motion blur method extrudes the input triangle by first linearly

transforming it from time 0 to time aperture. The triangle pair of the delimited aperture time space forms a convex hull as illustrated in Figure 8. Triangle samples are then interposed inside the hull, each depicting a snap shot in time. Extruded triangles are being emitted and passed onto the rasterizer. The motion blur method described is hence an additional mid pipe geometry growth source that further perturbs load distribution variations.

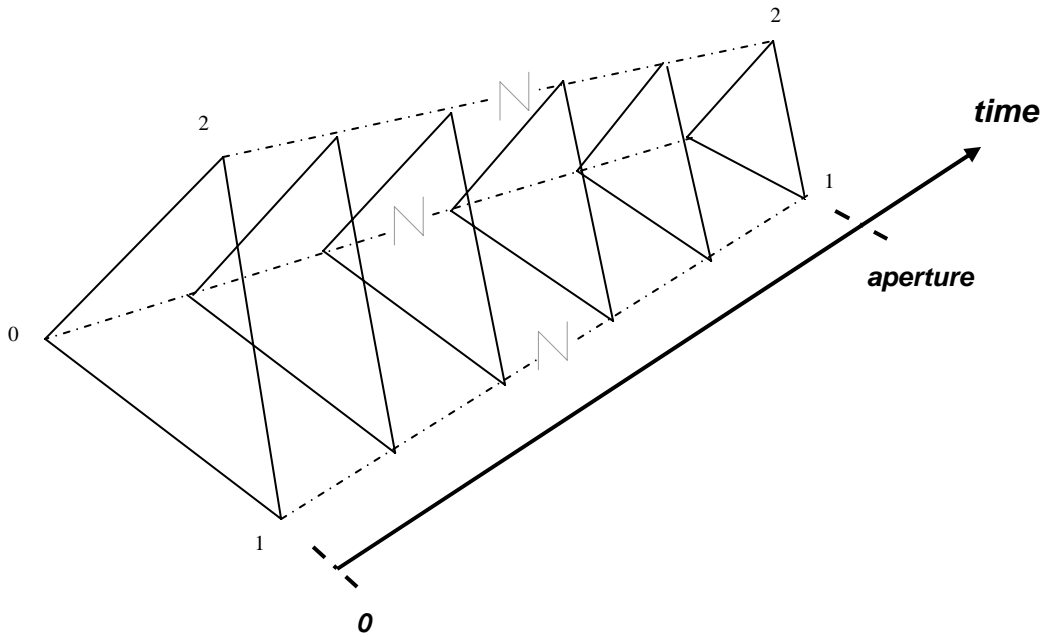


Figure 8: Geometry Shader - Motion Blur Primitive Expansion

The displaced, motion blurred geometry walkthrough presented inside the geometry-pixel pipeline demonstrated performance dynamics alters substantially compared to the vertex-pixel modality. In fact, the more subdivision refinement steps and increased motion aperture triangle samples might easily shift the load towards the geometry part of the pipe. In addition, tessellation, geometry and pixel shaders are macro threads, all running and sharing the same resources. Micro computational threads are scheduled and launched for each of the shader units to explore parallelism as much as possible. Outputs of each shader type are stored in a memory resource that is exposed both internally and externally. The availability of mid pipe shader results to the outside world is a leverage point in a multi GPU configuration. A graphics system for which one GPU feeds the other in a pipeline manner will be considered for shading load distribution. Figure 9 depicts potential shader partitions across two GPUs. In all three configurations the tessellation shader is executed in the GPU located at the top of the chain. GPU #0 is the only one who has access to the single copy scene description. Configuration options then vary by the pairing of shader units in one or the other GPUs. Pairing choice is arrived at

by optimally dividing overall shading computational labor and amortizing PCI-Express copy overhead for passing results from one processor to the other.

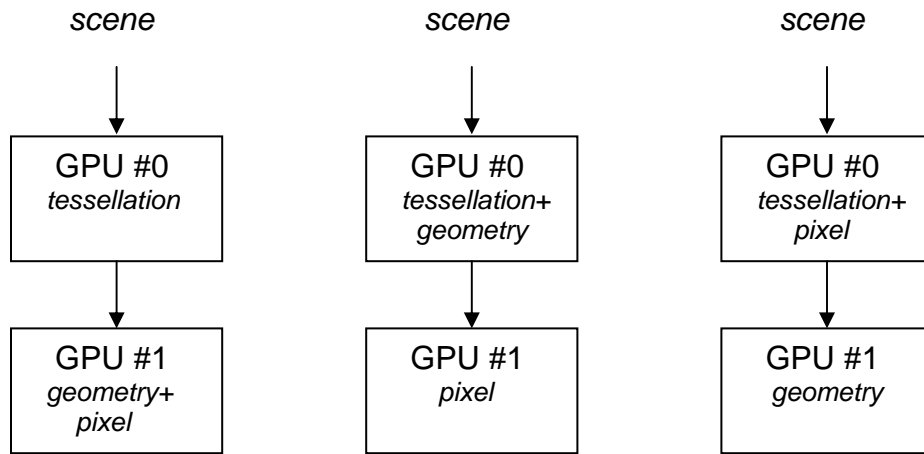


Figure 9: Multi-GPU Shader Partition Options

In the left column of Figure 9 tessellation vertices are copied from GPU #0 to GPU #1, who runs both the geometry and pixel shaders in sequence. In the middle column the top GPU performs tessellation and geometry shaders and the vertices of the latter are passed on to the rasterizer of GPU #1. The third option on the right has both the tessellation and pixel shader executed in the top GPU and the geometry shader runs on the second GPU. Here, the tessellation vertices are passed from GPU #0 to GPU #1 and then the vertices emitted by the geometry shaders are passed back to the first GPU. Hence, there are two copy operations of results from one GPU to the other.

Multi-GPU schemes provided are mostly for future reference and intended to rethink alternate load partition techniques.

Remapping GPU Processors

Vertex and pixel shaders in Microsoft DirectX Shader Model 3.0 have been simplified considerably compared to earlier versions. Instruction set on both processors has been for the most part identical with fairly few exceptions. One particular new feature introduced in Shader Model 3.0 is *vertex texture fetch*, which lets the vertex processor read data from textures. The ability to access textures on both the vertex and pixel processors opened an opportunity for retargeting pixel shader code onto the seemingly underutilized vertex shader. Programmers have exercised vertex texture fetch in their applications, touting the potential of improving overall GPU load balance. On the other hand the pixel processor explores a much higher degree of parallelism compared to the vertex processor and is natively designed to perform efficient texture fetches. Remapping vertex code, which includes texture fetches, onto the pixel processor might prove after all better GPU performance in certain circumstances. In addition, making GPU processor mapping as

transparent as possible to the programmer is highly desirable. Ashli implements an automatic *vertex-to-pixel* processor code conversion, providing a vehicle for analyzing code execution trade offs between the vertex and pixel processor domains. The technique also enables exploring geometry tessellation functionality, expanding the limited one-to-one vertex processor model into perhaps a few-to-many type of amplification.

In shader model 3.0 we can almost safely exclude instruction set adversity between the vertex and pixel shader. Similarly, internal shading resources on both processors are fairly close matching. Sampler registers is where most of the disparity lies - the pixel shader has sixteen over the four vertex ones. This makes a truly bi-directional processor mapping difficult to realize in a compiler, but works though in favor towards the vertex-to-pixel path. The items remained to implement for the vertex-to-pixel conversion process are then: the prolog fetch of vertex attributes from textures and the mapping of vertex outputs to pixel multiple render targets.

The vertex-to-pixel conversion path implies a two pass rendering process. It is assumed the vertex stream(s) has been redirected into a *vertex buffer* type render target(s). For the first rendering pass then, one or multiple IEEE floating point component 2D textures are available to the pixel processor for fetching vertex attributes. The input vertex format can have attributes in any of one, two, three or four component combinations as depicted in the following data structure:

```
struct VertexInput {  
    float4 pos      : POSITION;  
    float3 normal   : NORMAL;  
    float  color    : COLOR0;  
    float2 tex0     : TEXCOORD0;  
};
```

Two primary vertex data storage models were considered for vertex-to-pixel conversion: *packed* and *unpacked*. In the packed mode vertex data is stored contiguously in video memory, all in one texture. This lends itself into better cache locality when accessing vertex attributes. The compacted vertex data storage implies an addressing mode composed of one base pointer and a set of fixed offsets to each of the attributes. The packed storage version results in a per-component fetch from texture memory. When all vertex attributes are of four components (e.g. padded as necessary), fetch could be commenced on a per attribute basis, thus yielding a much efficient vertex data transfer. In the unpacked mode each vertex attribute is stored in a unique texture. An attribute texture is associated with a base pointer and offsets are format implicit. The two vertex storage choices represent traditional memory bandwidth to compute design trade offs. In both storage models input geometry mesh dimensions dictates texture width and height properties.

There are twelve vertex shader outputs and four pixel shader color outputs defined in Shader Model 3.0. To address this constraint for the vertex-to-pixel conversion, Ashli incorporated the notion of *virtual color outputs* for the pixel shader. Pixel shader outputs could be conceivably unbounded, but are currently set to top sixteen, primarily for practical purposes. For simplicity, Ashli keeps internally a fixed implicit vertex output to pixel output table as illustrated below (C# stands from pixel color output):

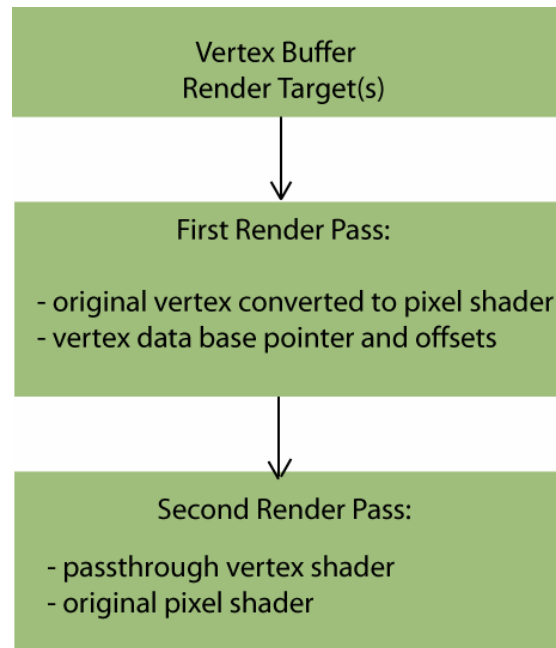
Vertex Output	Pixel Output
Position	C0
Color	C1
SecondaryColor	C2
TexCoord0	C3
TexCoord1	C4
TexCoord2	C5
TexCoord3	C6
TexCoord4	C7
TexCoord5	C8
TexCoord6	C9
TexCoord7	C10
Fog	C11

Vertex outputs exceeding render target cap for Shader Model 3.0 (e.g. four) results in segmenting the converted pixel shader to fit hardware constraints. Shader resource virtualization is an embedded technique in Ashli and plays handily in the vertex-to-pixel conversion process.

The converted vertex-to-pixel shader would be executing then three steps. First it performs *texture vertex fetch* for all of the present attributes. Ashli provides an optional *ContiguousVertex* compile time flag to produce pixel code tightly coupled with the vertex data storage format expected. Following the vertex fetch prolog the converted shader commences with the original vertex shader processing part. Unified (for the most part) vertex and pixel shader instruction set makes this part relatively seamless. Finally, vertex shader outputs are mapped onto pixel shader color outputs, based on the fore mentioned translation table.

As part of the vertex-to-pixel conversion process Ashli also generates a *passthrough* vertex shader, intended to be used in the second rendering pass. The converted pixel shader and the passthrough vertex shader are detached from each other and assume no parameter linkage across them. The passthrough vertex shader takes in the produced color outputs by the pixel shader as vertex attributes and remaps them back onto vertex outputs. The inverse mapping of pixel color outputs back onto vertex outputs is available to the application in Ashli's pixel section of the metadata produced.

The workflow steps for an application to embed the vertex-to-pixel shader conversion process are illustrated as they are related to the shading portion:



To conclude this section an Ashli code generation example of the vertex-to-pixel conversion process is depicted in Figure 10. The input is an HLSL vertex shader, which includes a vertex texture fetch statement. The pixel shader converted code highlights the prolog vertex attribute fetch from texture, assuming packed vertex storage format. Output mapping onto two physical pixel color outputs terminates the code. The pixel section metadata illustrates the pixel color physical to virtual output correspondence. This info further assists the application in performing inverse mapping back to vertex outputs e.g. C0 maps back to vertex position in clip space and C4 onto the vertex second texture coordinate set. Finally, the simple minded vertex passthrough shader is outlined – vertex attribute inputs are the ones coalesced from the render targets of the previous rendering pass.

Being multi rendering pass, the vertex-to-pixel conversion process incurs a copy overhead of vertex attributes stored in video memory render targets back onto the vertex buffer. This overhead is particularly noticeable for a small sized geometry mesh. Here, vertex-to-pixel conversion is performance inferior to running the shader on the vertex shader. However, the overhead is diminished as either the mesh size of the geometry increases or the computation vertex shader core is long enough. The pixel shader is able to launch more execution threads compared to the vertex shader and hence the speedup edge. The vertex-to-pixel shader mapping when used sparingly deems beneficial to certain applications. Its added value is primarily to analyze graphics pipe dynamic behavior and experiment with algorithms that constitutes geometry multiplicity.

<pre> struct VertexInput { float4 pos :POSITION; float2 tex :TEXCOORD0; }; struct VertexOutput { float4 pos :POSITION; float energy :TEXCOORD1; }; float4x4 mWorldViewProjection; float3x3 mWorldInverseTranspose; sampler shadow; VertexOutput main(VertexInput vi) { VertexOutput vo; vo.energy = 1.0f - tex2Dlod(shadow, float4(vi.tex, 0, 1)).x; vo.pos = mul(vi.pos, mWorldViewProjection); return vo; } </pre>	<pre> ps_3_0 dcl_texcoord0 v2.rgba dcl_2d s0 dcl_2d s1 dcl_2d s2 def c0, 0, 1, 0, 0 def c11, 0, 0, 0, 1 dp3 r0, t0, c10 mov r1, c0.r frc r1.r, r0.r add r1.g, r0.r, -r1.r mul r1.g, r1.g, c10.a texld r0, r1, s2 texld r1, r2, s2 dp3 r2, t0, c8 mov r3, c0.r frc r3.r, r2.r add r3.g, r2.r, -r3.r mul r3.g, r3.g, c8.a texld r1, r3, s1 texld r2, r4, s1 texld r3, r5, s1 texld r4, r6, s1 mov r2, c11 mov r2.r, r0.r mov r2.g, r1.r mov r0, c0.r mov r0.rg, r2 mov r0.b, c0.r mov r0.a, c0.g texldl r0, r0, s0 mov r5, c0.r mov r5.r, r1.r mov r5.g, r2.r mov r5.b, r3.r mov r5.a, r4.r mov r1, c0.r dp4 r1.r, c4, r5 dp4 r1.g, c3, r5 dp4 r1.b, c2, r5 dp4 r1.a, c1, r5 add r0, c0.g, -r0.r mov oC0, r1 mov oC1, r0 </pre>	<pre> vs_3_0 dcl_position0 v0 dcl_texcoord1 v9 dcl_position o0 dcl_texcoord1 o4 mov o0, v0 mov o4, v9 </pre>
<p>HLSL vertex shader</p>	<p>vertex-to-pixel</p>	<p>passthrough vertex</p>
<pre> begin fragment t 0 -1 _T0_ s 0 2D File shadow s 1 2D File attr0 s 2 2D File attr1 c 3 -1 mWorldViewProjection_0_0[0] 0 0 0 0 c 2 -1 mWorldViewProjection_0_0[1] 0 0 0 0 c 1 -1 mWorldViewProjection_0_0[2] 0 0 0 0 c 0 -1 mWorldViewProjection_0_0[3] 0 0 0 0 o 0 -1 _C0_ o 1 -1 _C4_ end begin vertex a 0 -1 _Vertex_ a 9 -1 _TexCoord1_ o 0 -1 _Position_ t 1 -1 _T1_ end </pre>		
<p>metadata</p>		

Figure 10: Vertex-to-Pixel Conversion Code Sample

Source Level Debugger

As GPU becomes more powerful to handle long and complex shaders, debugging tools emerged an increasingly important item for graphics hardware users. Microsoft® Visual Studio® .NET [Microsoft 2004] and the Shadesmith [Purcell & Pradeep 2003] amongst others are examples of shader debugging offerings. Microsoft .NET tool supports debugging both assembly-level and high-level language, vertex and pixel shaders. The .NET debugger tool is very powerful and provides both file/line and pixel area type of break points, and allows inspecting hardware register content. Nevertheless, no debugging support is available for direct hardware vertex and pixel processing. Shadesmith on the other side of the spectrum provides an automatic way to debug fragment programs and runs on graphics hardware. The user can set watch registers and examine them for all pixels and potentially could edit the program inline, without the need of exit and rerun. Shadesmith is an assembly level debugger and is platform dependent.

In Ashli we wanted the shader debugger to be source high-level language, orthogonal to all the languages supported e.g. HLSL, GLSL and RenderMan®, and most importantly - run on hardware. The intended user for Ashli debugger is more at the level of the content creator for inspecting and editing shader source and less so for the hardware savvy. The debugger aids programmers to write lengthy shaders and debug for correctness directly on graphics hardware, with minimal or no source intrusion. Counter to debugging a program on the CPU, shader debugging process rather employs visualization for code validation. No hardware shading state is available for review in Ashli's debugger and it is only applicable for the pixel shader.

Ashli implements a basic source level debugger that is exposed in the API via a handful of methods. Debugging functionality includes adding and removing a file/line type break point, continue to the next break point, single stepping source statements and querying the current break point. Ashli debugger automatically replaces an lhs (left-hand-side) of a break point assignment statement with a pixel shader color output. The code generated by Ashli for a given break point constitutes the section preceding the break point, inclusive. The user adds or removes source break points by making the calls:

```
void addBreakPoint(const char* line);  
void removeBreakPoint(const char* line);
```

The add/remove break point call argument is of the form: *<file name>|<line #>* e.g. *matte.sl|76*. Once break points have been established the user will invoke a compile session using the *invoke* method. The *invoke* method will get code generated for the source delimited by the first break point. The user will then optionally call either the *next* or the *step* methods:

```
bool next();  
bool step();
```

These methods are for all practical purpose identical to the invoke one, aside from maintaining all the internal compiler state established by the first invoke call, for the purpose of reuse. The next method advances the program to the next break point and generates code for the new source extent, set forth. The step function progresses one statement at a time, for finer grain program advancements. Finally, the user can query the current state (file/line) of the source break point, including the single step offset, relative to a break point. This call is useful in highlighting the source code designated break point line on the programmer's editor end:

```
const char* getBreakPoint() const;
```

Debugger break point validation inside Ashli is a two part process. First, original pixel shader output(s) are searched and substituted with a dummy local variable. Then, the lhs of the current break point assignment is replaced with a color output. The expression tree constructed internally in Ashli results in an intentional degenerated nodes e.g. from the newly designated output node all the way to the root of the tree. The valid expression subtree that represents the break point delimited program segment is extracted for code generation. The debugging of dynamic flow control statements requires more attention and has not been worked out to its full extent. For a break point inside an if or an else block, the lhs of both the marked assignment statement inside the block and the one right before the block will be replaced by a color output. This is to insure the case for which branches are not taken for some of the pixels and a color output assignment is always present in the program. Ashli maintains the ability to unroll loops conditionally in certain scenarios when dependencies are only partially resolved. This is despite the dynamic flow control support in Shader Model 3.0. In the case of a break point inside a loop, the loop could possibly be unrolled internally and for every debugger continue call the count is incremented by one till it reaches a runtime cap set by the user.

Figures 11 and 12 depict the use of Ashli shader debugger and illustrate both code generation and visual code validation. In the left column of Figure 11 the RenderMan® spiderweb source shader is highlighted with two breakpoints set for debugging. The generated pixel shader assembly code for each of the breakpoints is shown in the middle and right column of the figure, respectively. Total number of instructions executed for the first break point is 53, and 60 instructions for the second one. The left image of Figure 12 demonstrates a partial spiderweb image delimited by the first break point set, for which only the radial webs are rendered. The right image is the result of the complete shader and both the radial and tangential webs are included. The Ashli viewer application was used to render the spiderweb images on an ATI X800 hardware platform.

```

float
sqr(float x)
{
    return x*x;
}

float dist (float x,y)
{
    return (sqrt(sqr(x)+sqr(y)));
}

float Gaussian(float x,m,y)
{
    return (exp(-sqr(x)-(m)/sqr(y)));
}

surface
spiderweb(
    float   cRadialWebs   = 7;
    float   cCrossWebs    = 10;
    float   rWebDiam       = .01;
    float   rWebCenterT   = 0;
    float   rWebCenterS   = 0)
{
    float TT=(t-rWebCenterT);
    float SS=(s-rWebCenterS);
    float rRadius = dist(SS,TT);
    float rAngle=atan(SS,TT);
    float rAngleStep=2*Pi/cRadialWebs;
    float rDist;
    float rAccum=0;
    float cRadSeg;
    float cCrossSeg;

    /* Calculate which radial section its in */
    cRadSeg=floor(rAngle/rAngleStep);

    /* Color Radial Webs */
    rAccum+=Gaussian(rAngle,
        cRadSeg*rAngleStep,
        rWebDiam / rRadius);
    rAccum+=Gaussian(rAngle,
        mod((cRadSeg+1),cRadialWebs)*rAngleStep,
        rWebDiam / rRadius);
    rDist=(SS * cos((cRadSeg+.5)*rAngleStep) +
        TT * sin((cRadSeg+.5)*rAngleStep));

    /* Calculate which cross section its in */
    cCrossSeg=floor(rDist * cCrossWebs / 2);

    /* Color Tangential Webs */
    rAccum+=Gaussian(rDist,
        cCrossSeg/cCrossWebs * 2,
        rWebDiam / rRadius);
    rAccum+=Gaussian(rDist,
        (cCrossSeg+1)/cCrossWebs*2,
        rWebDiam / rRadius);

    /* clamp to [0,1] */
    rAccum=clamp(rAccum,0,1);

    Ci=Os * mix( color(0,0,0), Cs, rAccum);
    Oi=Os * rAccum;
}

```

RenderMan spiderweb shader

```

ps_3_0
// Instructions (Alu):    53

dcl_texcoord0 v2.rgb

def c0, c2, series factors
def c3, -0.11573, 0.0519506, 1, 0

add r0.r, v2.g, -c1.a
abs r0.g, r0.r
add r0.b, v2.r, -c1.b
abs r0.a, r0.b
cmp r1.r, -r0.b, r0.b, c3.b
cmp r1.r, r0.b, r1.r, -c3.b
mul r1.g, r1.r, c0.a
cmp r0.a, -r0.a, c3.a, r1.g
rcp r1.g, r0.r
mul r1.g, r0.b, r1.g
abs r1.b, r1.g
cmp r1.g, -r0.r, r1.b, r1.g
abs r1.b, r1.g
add r1.b, c3.b, -r1.b
rcp r1.a, r1.g
cmp r1.a, r1.b, r1.g, r1.a
mul r2.r, r1.a, r1.a
mov r3, c0
mad r2.g, r2.r, r3.g, c3.g
mad r2.g, r2.r, r2.g, c3.r
mad r2.g, r2.r, r2.g, c2.a
mad r2.g, r2.r, r2.g, c2.b
mad r2.r, r2.r, r2.g, c2.g
mul r1.a, r2.r, r1.a
cmp r2.r, -r1.g, r1.g, c3.b
cmp r1.g, r1.g, r2.r, -c3.b
mad r1.g, r1.g, c0.a, -r1.a
cmp r1.g, r1.b, r1.a, r1.g
add r1.b, c2.r, -r1.g
mul r1.r, r1.r, r1.b
cmp r1.r, -r0.r, r1.r, r1.g
rcp r0.a, c1.g
mul r0.a, r0.a, c0.r
rcp r1.r, r0.a
mul r0.b, r0.b, r1.r
frf r1.r, r0.b
add r0.b, r0.b, -r1.r
add r0.b, r0.b, c3.a
mul r0.b, r0.b, r0.a
mul r0.a, r0.b, r0.b
mad r1.r, c3.b, r0.a, c3.g
mad r1.r, r1.r, r0.a, c3.r
mad r1.r, r1.r, r0.a, c2.a
mad r1.r, r1.r, r0.a, c2.b
mad r1.r, r1.r, r0.a, c5.a
mad r1.g, c2.g, r0.a, c2.r
mad r1.g, r1.g, r0.a, c0.a
mad r1.g, r1.g, r0.a, c0.b
mad r0.a, r1.g, r0.a, c5.a
mul r0.b, r0.a, r0.b
mul r0.g, r0.g, r0.b
mad r0.r, r0.r, r1.r, r0.g
mov r1, c1
mul r0.g, r1.r, c3.a
mul r0.r, r0.r, r0.g
frf r0.g, r0.r
add r0.r, r0.r, -r0.g
mov r1, r0.r
mov oC0, r1

```

code @ first break point

```

ps_3_0
// Instructions (Alu):    60

dcl_texcoord0 v2.rgb

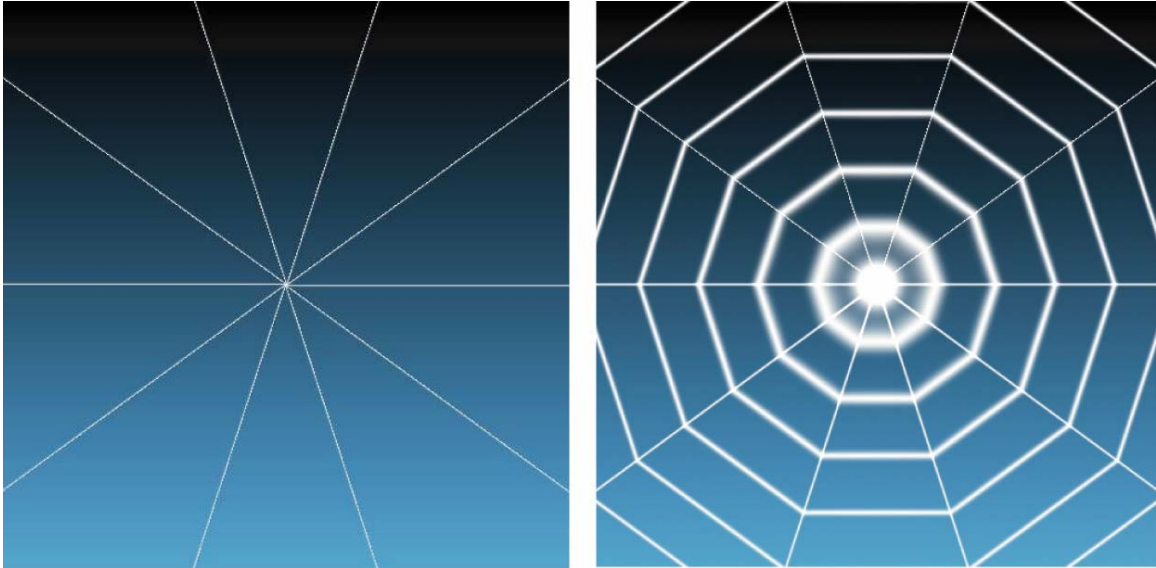
def c0-c5, series factors
def c6, 0, 0, 0, 0

add r0.r, v2.r, -c1.b
add r0.g, v2.g, -c1.a
abs r0.b, r0.g
abs r0.a, r0.r
cmp r1.r, -r0.r, r0.r, c5.a
cmp r1.r, r0.r, r1.r, -c5.a
mul r1.g, r1.r, c4.r
cmp r0.a, -r0.a, c6.r, r1.g
rcp r1.g, r0.g
mul r1.g, r0.r, r1.g
abs r1.b, r1.g
cmp r1.g, -r0.g, r1.b, r1.g
abs r1.b, r1.g
add r1.b, c5.a, -r1.b
rcp r1.a, r1.g
cmp r1.a, r1.b, r1.g, r1.a
mul r2.r, r1.a, r1.a
mov r3, c0
mad r2.g, r2.r, r3.g, c5.b
mad r2.g, r2.r, r2.g, c5.g
mad r2.g, r2.r, r2.g, c5.r
mad r2.g, r2.r, r2.g, c4.a
mad r2.r, r2.r, r2.g, c4.b
mul r1.a, r2.r, r1.a
cmp r2.r, -r1.g, r1.g, c5.a
cmp r1.g, r1.g, r2.r, -c5.a
mad r1.g, r1.g, c4.r, -r1.a
cmp r1.g, r1.b, r1.a, r1.g
add r1.b, c4.g, -r1.g
mul r1.r, r1.r, r1.b
cmp r1.r, -r0.g, r1.r, r1.g
cmp r0.b, -r0.b, r0.a, r1.r
rcp r0.a, c1.g
mul r0.a, r0.a, c0.r
rcp r1.r, r0.a
mul r0.b, r0.b, r1.r
frf r1.r, r0.b
add r0.b, r0.b, -r1.r
add r0.b, r0.b, c3.a
mul r0.b, r0.b, r0.a
mul r0.a, r0.b, r0.b
mad r1.r, c3.b, r0.a, c3.g
mad r1.r, r1.r, r0.a, c3.r
mad r1.r, r1.r, r0.a, c2.a
mad r1.r, r1.r, r0.a, c2.b
mad r1.r, r1.r, r0.a, c5.a
mad r1.g, c2.g, r0.a, c2.r
mad r1.g, r1.g, r0.a, c0.a
mad r1.g, r1.g, r0.a, c0.b
mad r0.a, r1.g, r0.a, c5.a
mul r0.b, r0.a, r0.b
mul r0.g, r0.g, r0.b
mad r0.r, r0.r, r1.r, r0.g
mov r1, c1
mul r0.g, r1.r, c3.a
mul r0.r, r0.r, r0.g
frf r0.g, r0.r
add r0.r, r0.r, -r0.g
mov r1, r0.r
mov oC0, r1

```

code @ second break point

Figure 11: Ashli Source Debugger, Code Generation Sample



rendered image @ first break point

rendered image of complete shader

Figure 12: Ashli Source Debugger, Visual Results Sample

References

[ASHLI 2003] ATI: *ASHLI - advanced shading language interface*,
<http://www.ati.com/developer/ashli.html>.

[Bleiweiss & Preetham 2003] Avi Bleiweiss, Arcot J. Preetham: *ASHLI – Advanced Shading Language Interface*, Siggraph 2003 Real-Time Shading Course Notes

[Cg 2004] Nvidia: *Cg Shading Language*,
http://developer.nvidia.com/page/cg_main.html

[CoreImage 2004] Apple: *Core Image, Ultra Fast Pixel Accurate*,
<http://www.apple.com/macosx/tiger/coreimage.html>

[HLSL 2004] Microsoft: *DirectX Shading Language*,
<http://www.microsoft.com/windows/directx>

[Microsoft 2004] Microsoft: *DirectX 9 Shader Debugger Tool*,
<http://www.microsoft.com/windows/directx/graphics/tools>

[Olano et al. 2003] Olano Marc, Bob Kuehne and Maryann Simmons: *Automatic Shader Level of Detail*, Proceedings of Graphics Hardware 2003, Eurographics/ACM SIGGRAPH

[GLSL 2004] *OpenGL Shading Language*,
<http://opengl.org/documentation/oglsl.html>

[OpenEXR 2004] ILM: *OpenEXR High Dynamic Range Image Format*,
<http://www.openexr.com/>

[PCIExpress 2004] PCI Express: *Performance Scalability for the Next Decade*,
<http://www.pcisig.com/specifications/pciexpress/>

[Purcell & Pradeep 2003] *Shadesmith: A Fragment Program Debugger*,
<http://graphics.stanford.edu/projects/shadesmith/>

[RenderMan® 1986] Pixar: *RenderMan® Shading Language*,
<https://renderman.pixar.com/>

[Sharp 2000] Brian Sharp: *Subdivision Surface Theory*,
http://www.gamasutra.com/features/20000411/sharp_01.htm