A Comparison of Real Time Graphical Shading Languages

CS4983 Senior Technical Report

Anthony Lovesey
3068528

Faculty of Computer Science
University of New Brunswick
Canada

March 26, 2005

Executive Summary

As the technologies present in modern real-time 3D graphics hardware continue to expand, programmers and artists are increasingly in need of tools and APIs with which to utilize the power and flexibility that programmable graphic processing units (GPUs) now provide. Such capabilities permit increasingly realistic and accurate visuals to be rendered on consumer level hardware far beyond what traditional fixed-function rendering pipelines can provide. With the advent of high-level shading languages, programmers can directly manipulate the functionality of the 3D rendering pipeline by creating custom "shaders" that supplement the fixed function vertex and fragment processing stages. These shading languages are Microsoft's DirectX High Level Shading Language (HLSL), NVIDIA's C for Graphics (Cg) and OpenGL's Shading Language (GLSL). Programmers deciding whether or not to utilize a high-level shading language need to be aware of the changes necessary to the rendering process overall as well as the inherent differences, besides just those of syntax and semantics, between each language.

This technical report explores graphical shading languages and provides a comparison of the three high-level graphical shading languages by contrasting language features including available language functions, program flexibility, pipeline control and state access, data type support, ease of maintenance features, compile and execution times, vendor support, implementation maturity and driver stability. The results of these comparisons show that all three languages provide comparable language features and that performance and usability largely depends on drivers and compiler implementations as well as the choice of graphical API.

Results of a light-weight C++-based test framework used to implement configurations of all three languages examines execution time differences in one unique vertex processing bound shader test case and one unique fragment processing bound shader test case. The results from these simple tests indicate that the actual execution time of shaders is dependent on the maturity of the graphical hardware's drivers and the language's compilers. The vertex-bound shader test results display a minimal difference in execution time, with all three languages providing similar performance. Under the pixel-bound shader test, results show that alternative lighting algorithms that produce a greater visual quality can be used at real-time frame rates, with HLSL executing the test pixel shader up to 10% faster then Cg and GLSL.

Recommendations are also provided to help developers wishing to create programmable shaders. HLSL is the recommended shading language choice if DirectX is the API of choice, with Cg recommended over GLSL for use with OpenGL until 2006 when GLSL implementations are expected to become widely available. These recommendations are a starting point to aid in deciding what shading language would prove the most beneficial to developer requirements.

Table of Contents

Table of Figures

1. Introduction

Over the last decade, the field of computer graphics and imaging has experienced a significant evolution in regards to both available hardware and software. Increasingly affordable consumer-level commodity graphics accelerators coupled with ever-evolving API and operating system support, has permitted advanced real-time, graphically intense 3d applications to become accessible to an increasing number of developers and end-users. Following a similar evolutionary track as contemporary general purpose CPUs, today's graphical processing units (GPUs) offer raw performance features that permit the majority of 3d imaging and composition operations to be fully offloaded from the CPU. With this continuing evolution in rendering technology, there is a strong desire to give developers greater control over GPU processing capabilities. As GPUs are primarily responsible for transforming and rasterizing 3D primitives, a great deal of research has been invested by vendors to create languages and APIs that allow for the programmable control over what has traditionally been a fixed function 3D pipeline. This originally crude flexibility has given way to elegant high-level graphical *shading languages* that allow for the construction of programs that utilize graphics hardware to execute a wide variety of rendering algorithms.

Currently there exist three primary high-level graphical shading languages that are vendor supported: Microsoft's DirectX High Level Shading Language (HLSL) [10], NVIDIA's C for Graphics (Cg) [11] and the OpenGL Shading Language (GLSL) [13]. These languages are meant to expose a high level of programmability on current and future consumer level graphics hardware. As a developer or researcher wishing to utilize one of these languages in an application or graphics algorithm test environment, what are

some of the considerations that should be made? What unique features does each of these shading languages offer to developers? Are stable and mature compilers and drivers available?

This paper presents an in-depth comparison between these three current high-level graphical shading languages by examining unique features, implementation philosophies and language flexibility. An emphasis is placed on aspects of these languages that are crucial for a greater overall understanding of the technology that is producing a significant shift in how real-time rendering algorithms are being implemented. This paper begins by briefly describing the evolution that the 3d rendering pipeline has gone through, leading to the creation of current shading languages. Each high level shading language is then discussed with details about their design philosophies. Section 3 details common data types and functions available for all three languages, followed by features of each language that aid or hinder maintainability of shaders within large projects. Crucial to integrating shading technology into standard graphic APIs is the concept of having access to fixed function states that is explored in Section 6. Section 7 describes the driver and implementation maturity of all three languages on 3 common operating systems, Microsoft Windows, Linux, and Apple OS X. To investigate any potential performance penalties of using graphical shaders, results from vertex and pixel shader test cases are given with execution speed statistics. Finally, recommendations are made about the usability and important aspects of each language that developers should be mindful of.

2. Evolution of the Programmable Graphics Pipeline

With the introduction of affordable 3d graphics hardware nearly a decade ago, developers were free to create applications that could display interactive real time three-dimensional images by utilizing many of the accelerated 3d pipeline features present in these new graphic processors [1]. When creating 3d spaces, all source geometry and shading information must be submitted to the GPU and utilized to construct the final output image [7]. This is accomplished by transmitting the input geometry through an application API such as OpenGL or DirectX over the system bus to the actual graphics hardware. Once in the accelerator hardware's memory, all information is processed through a standard collection of pipeline stages [1]. Per-vertex operations are performed first as each incoming vertex is transformed from the local space in which it was defined into world space by modeling and viewing transformations specified by the user. If per-vertex lighting is requested lighting calculations may also be performed. After a number of vertices is received that equals the number of vertices of the primitive (e.g. 3 for a triangle) currently being constructed, the primitive is assembled and projection and clipping is performed on the primitive [14]. The primitive then reaches the rasterization stage of the pipeline where each pixel that represents the primitive's surface is processed. The goal of this stage is to calculate a final output color for the pixel by taking into account user defined lightening, fog, and surface material properties [1]. The pixel is tested against per-fragment testing parameters such as depth and stencil properties to decide if the fragment should proceed any farther or be discarded. Finally, the rasterized primitive must have any frame buffer operations, such as blending applied, before being written to the system's frame-buffer [14]. This entire process is shown in Figure 1.

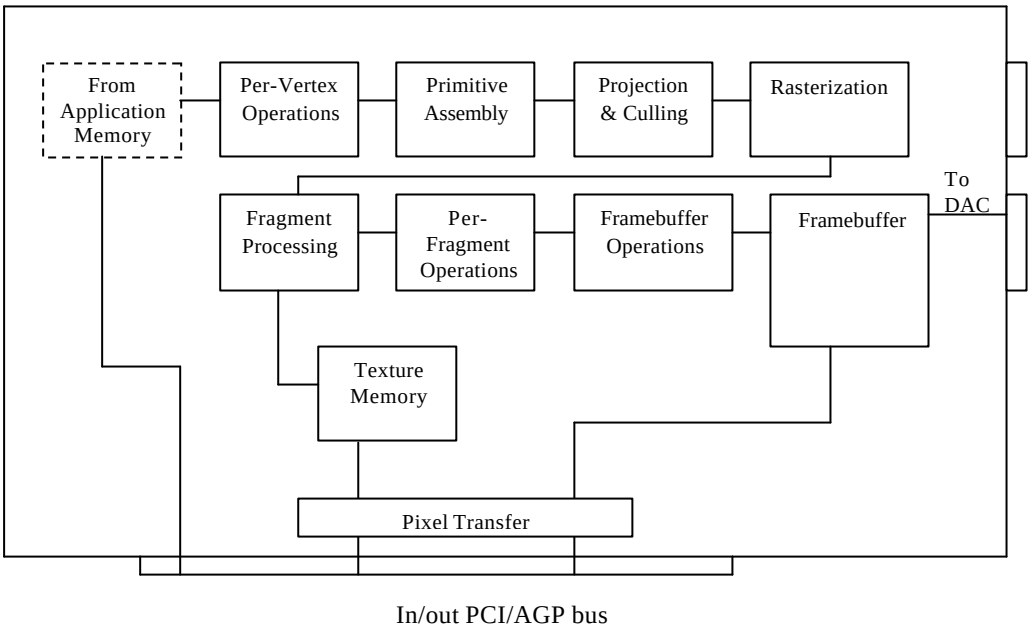

Figure 1: Fixed Function 3D Graphical Pipeline

As early hardware was designed around being able to provide the most efficient rendering results for what accelerated power was available, graphics API architects and hardware vendors provided a fixed function pipeline that was statically written into the drivers or designed into the physical hardware. Under a fixed function pipeline data can be sent to the pipeline and pipeline states set, but no direct altering of the vertex and fragment-processing stages can be specified [4]. A common example is the lighting model used by most 3d pipelines. Both DirectX and OpenGL use a Phong based lighting formula as it can be easily computed on a per-vertex basis. Even though many other lighting models exist such as Global Illumination and BRFD [3], developers where restricted to the one available model. Similar examples of restrictions existed throughout the pipeline process.

As GPU architectures were refined and increased in speed, vendors slowly began to expose some programmability into their hardware by allowing for very simple operations to be performed on the rasterization stage of the pipeline through the use of register combiners. Implemented as additions to the graphic APIs, register combiners allowed for primitive mathematical operations to be performed on the texture register contents at this stage [1]. While somewhat crude, this permitted an initial glimpse of what increased control of the pipeline could provide as multiple texels applied to a surface could be added, subtracted, multiplied, and even have their dot product computed. The most significant change then came with the introduction of assembly-like shading languages. With enough power and flexibility in hardware to perform calculations outside of those of the fixed function pipeline, these languages allowed for the first true implementation of a programmable pipeline [9].

A programmable pipeline allows for complete control over specific stages of the pipeline through the creation of *shaders*; small snippets of code that when compiled, dynamically replace a particular stage's standard vendor supplied execution implementation. The two most computationally intensive stages of the pipeline, per-vertex processing and fragment processing, are the stages that provide support for programmable shading technology [3]. Similar in appearance to traditional CPU assembly languages, these early shading languages allow a developer to write a separate string based vertex and pixel program that can be run through the graphics API and replace the fixed function calculations of both pipeline stages. Figure 2 shows a typical design of a programmable pipeline. The stages marked *Custom Vertex Processor* and

Custom Fragment Processor represent the stages where programmable shaders replace the functionality of the fixed-function pipeline.

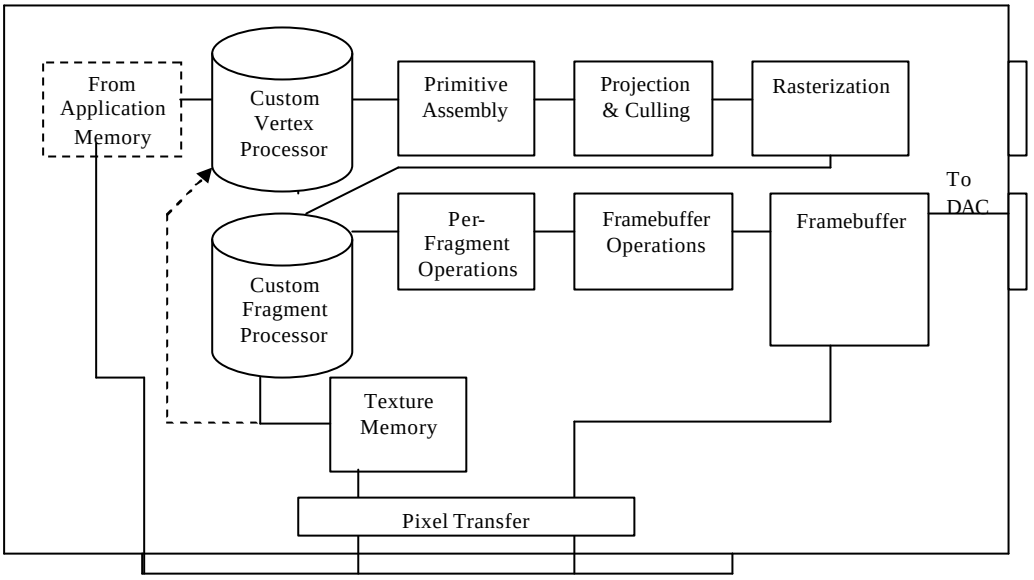

Figure 2: Programmable 3D Graphical Pipeline

Hardware registers, texture units and primitive and state information can be easily accessed and modified with a variety of standard mathematical and geometric functions. This flexibility immediately allowed developers to start implementing graphical algorithms that up until that point could only be tested in software on the CPU, usually rendering at a less then real time speeds [12].

As the number of assembly commands and features available increased, it was easily seen that the assembly-like languages would prove inadequate as shaders continued to grow in size and complexity. To rectify this problem, designers began work on the next level of shading languages, producing high level shading languages that provided a C like syntax and increased readability that could be compiled down to vendor

hardware specific machine instructions and run similar to the earlier assembly shading languages [4]. The increased ease of writing custom shading code that these languages provide introduces an even greater opportunity for graphics developers to implement custom, unique rendering algorithms and techniques.

3. Modern High Level Graphical Shading Languages

Presently, there are two primary graphical APIs used in real-time 3d graphics; Microsoft's proprietary DirectX and OpenGL maintained by the Architecture Review Board (ARB). Each API currently provides a high level shading language, HLSL and GLSL respectively, with the graphic hardware vendor NVIDIA providing a third, Cg, which can use either API. All three high level shading languages are designed around a familiar framework and syntax that closely resembles a subset of the C programming language [8]. User-defined functions, mathematical calculations, loops and conditional statements[1] are available. The two categories of shaders, vertex and pixel (fragment[2]), are written as separate string based source files that can be loaded through the corresponding API. Each distinctive shader is written as a collection of one or more functions, where one must be a function with a main() declaration to indicate a point of entry where execution of the corresponding pipeline stage is to begin [13]. All three shading languages also currently abstract or limit access to the actual hardware of each programmable stage such as direct video or texture memory manipulation [4]. There are a number of unique features of each language that prove important, most notably what each language defines as compliant hardware.

Additionally, for each of the discussed high level shading languages we provide a simple vertex and pixel shader example for comparison that execute the identical operations in each language. The vertex shader transforms the incoming vertex (from a sphere mesh) into clip space and feeds it back out to the next pipeline stage. Likewise, the

[1] Dynamic flow control is supported only on the most recent hardware [11].

[2] DirectX uses the term “pixel” shader whereas OpenGL utilizes “fragment” shader.

pixel shader calculates the color of the current pixel of the current primitive by fetching a texel from an active texture unit containing a rock-like diffuse texture and adding the current primitive color, green, to the result. The resulting identical image produced by all three shading languages is shown in Figure 3.

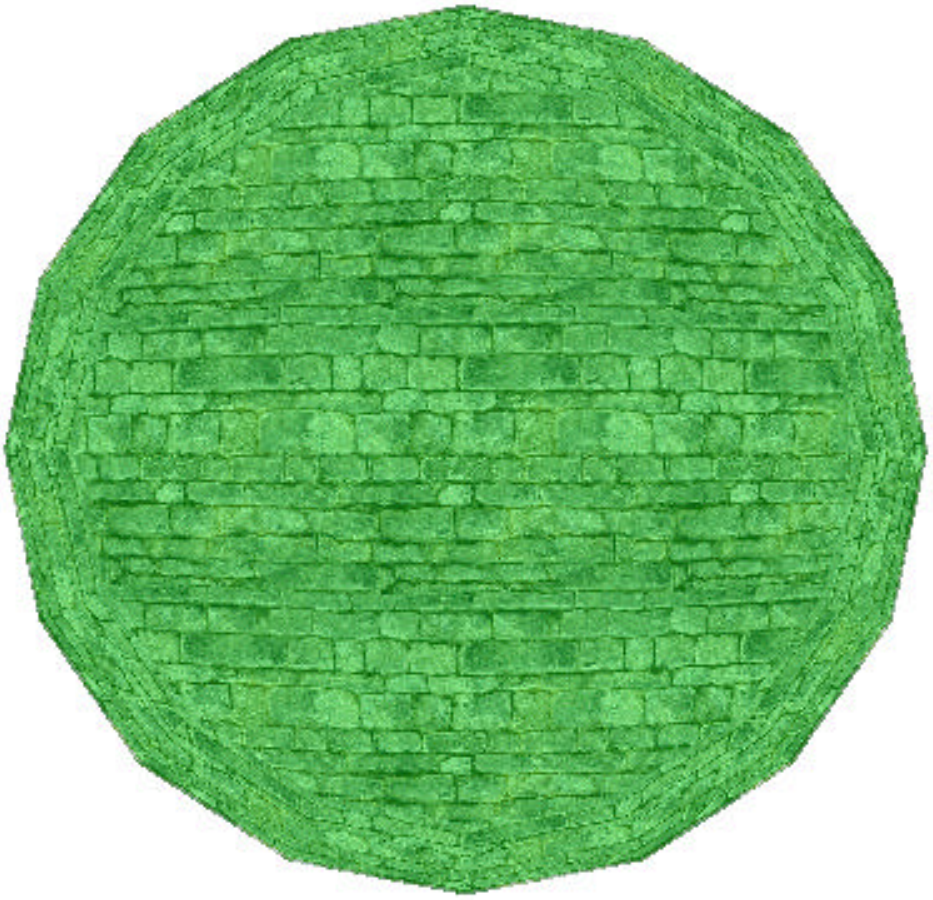

Figure 3: Result image for shading languages example

It is important to note that current high level shading languages provide the ability to replace computations at the vertex and pixel processing stages of the pipeline but must

still accept and output the same data as under a fixed function pipeline. Therefore, while shaders can be used to calculate other lighting models and effects by utilizing the speed of modern GPUs, they have no knowledge of the entire scene composition or other surrounding geometric objects. The use of shaders also helps reduce the amount of graphical API specific calls that are needed in the user application since rendering algorithms that required the modification of numerous pipeline states and multiple rendering passes can be removed.

3.1. HLSL

A younger API then OpenGL, Microsoft's DirectX framework, whose Direct3D component is used for 3d graphics, has provided developers with access to shading capabilities for approximately 4 years [10]. HLSL is a recent attempt to create a high level shading language with support for legacy as well as future hardware. Referred to as "Vertex Shaders" and "Pixel Shaders", DirectX denotes a graphic hardware's level of shader support by the respective shader version supported [10]. Each version of the vertex and pixel shader specification denotes a minimum set of hardware requirements that must be met for a graphics adaptor to be compliant. This can include mandatory support for a specific number of underlying temporary registers, constant registers, number of instructions per active shader, number of dependent texture reads, and ability to pass shared values between shaders. Current specifications include vertex shader versions 1.1, 2.0, 2.x, 3.0 and pixel shader versions 1.1, 1.2, 1.3, 1.4, 2.0, 2.x, 3.0, commonly referred to as Shader 1.x, Shader 2.0, Shader 2.x and Shader 3.0 [8]. DirectX's high level shading language, HLSL, provides compilation target support for all shader

versions, but version 2.x of both shader types is recommended as lower levels often provide inadequate support for implementing complex algorithms [10].

When an application is required to utilize a shader, the complete source for the shader is loaded and transmitted to the HLSL translator that is provided by the DirectX runtime. This translator produces an intermediary binary stream representation of the shader that is then passed to the hardware's driver where the stream is assembled into vendor specific instructions and cached on the actual hardware [2]. When the shader is to be run, an API command informs the driver that the compiled code is to be used in place of the fixed function operations and until deactivation the shader becomes an active part of the pipeline. This process is illustrated in Figure 4.

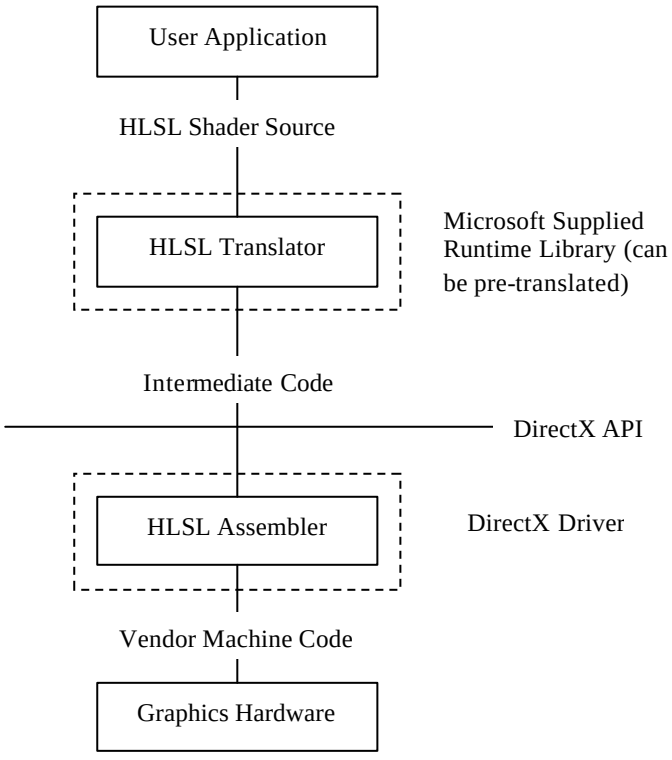

Figure 4: HLSL Compilation and Execution Process

As HLSL is tightly integrated into the DirectX runtime, shaders written in this language can only be used under the Windows operating system [10]. Figures 5 & 6 below provide an example of our simple HLSL vertex and pixel shader example described earlier. The resulting image can be seen in Figure 3.

```
/*
Sample HLSL Vertex Shader
Transforms an incoming vertex into clip-space and output to next
pipeline stage. Also passes along vertex color and texture coordinates.
*/

float4x4 worldViewProj;   //Uniform variable – set by application

/*Structure containing per-vertex input values*/
struct VS_INPUT
{
      float3 position  : POSITION;        //Object space vertex coords
      float4 color0    : COLOR0;          //Current vertex color
      float2 texcoord0 : TEXCOORD0;       //Current vertex texture
                                          //coords
};

/*Structure containing per-vertex output values*/
struct VS_OUTPUT
{
      float4 position  : POSITION;        //Homogenous clip space
                                          //vertex coords
      float4 color0    : COLOR0;          //Current vertex color
      float2 texcoord0 : TEXCOORD0;       //Current vertex texture
                                          //coords
};

/*Vertex shader entry point*/
VS_OUTPUT main( VS_INPUT IN )
{
      VS_OUTPUT OUT;

      //Store incoming vertex's coordinates
      float4 v = float4( IN.position.x, IN.position.y,
                         IN.position.z,1.0f );

      //Transform current vertex into clip space using model-view-
      //projection matrix
      OUT.position  = mul( v, worldViewProj );
      OUT.color0    = IN.color0;
      OUT.texcoord0 = IN.texcoord0;

      //Output current vertex values
      return OUT;
}
```

Figure 5. HLSL Simple Vertex Shader Sample

```
/*
Sample HLSL Pixel Shader
Calculates the final color of the current pixel by looking up a diffuse
color value from a bound texture and adding the interpolated current
color.
*/

//Sampler that indicates the loaded texture to be sampled from
sampler testTexture;

/*Structure containing per-pixel input values*/
struct VS_OUTPUT
{
      float4 position  : POSITION;     //Screen space x,y coordinates
      float4 color0    : COLOR0;       //Interpolated color
      float2 texcoord0 : TEXCOORD0;    //Interpolated texture coords.
};

/*Structure containing per-pixel output values*/
struct PS_OUTPUT
{
      float4 color : COLOR;   //Final pixel color
};

/*Pixel shader entry point*/
PS_OUTPUT main( VS_OUTPUT IN )
{
      PS_OUTPUT OUT;

      //Using the interpolated texture coordinates for this pixel
      //look up a diffuse RGB color value from the bound texture
      //and add the current pixel color. The result is the final color
      //value for the current pixel.
      OUT.color = tex2D( testTexture, IN.texcoord0 ) + IN.color0;

      //Return current pixel values
      return OUT;
}
```

Figure 6. HLSL Simple Pixel Shader Sample

3.2. Cg

Developed in cooperation with Microsoft during the design of HLSL, NVIDIA's Cg high level shading language is nearly identical in syntax and semantics to HLSL [13]. Originally designed as a cross API shading language and an early OpenGL high level shading language candidate, Cg combines an interesting number of features from both of

the other two shading languages. Through the use of an external runtime translator, Cg's string based vertex and fragment shaders are translated to either DirectX or OpenGL assembly language programs that are then run through the respective API's shader assemblers [11]. Figure 7 details this approach.

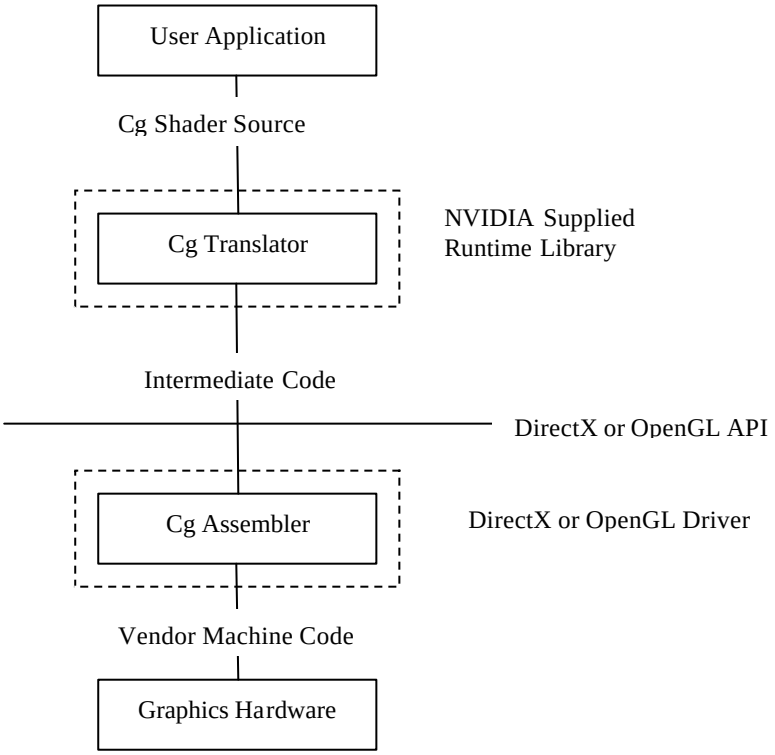

Figure 7: NVIDIA Cg Compilation and Execution Process

This cross API support is accomplished through the use of translation "profiles", similar to HLSL's shader levels, that allow for the shaders to target a wide variety of hardware, including the standard DirectX shader levels, OpenGL's shading extensions, and specific NVIDIA hardware features. When a shader is sent to the translator the user specifies a profile flag and the translator will attempt to generate code that runs on

hardware with the corresponding level of physical support [11]. For example, the VS_2_0 and PS_2_0 profiles will try and compile vertex and fragment programs to fit in the hardware restrictions of the Shader 2.0 specification whereas the VP40 and FP40 profiles will translate Cg shaders to utilize many of the new features available on only most recent NVIDIA hardware. This can be quite confusing as depending on the API and targeted hardware a developer can chose between 7 separate vertex and pixel shading profiles. Attempting to run translated Cg code on hardware that does not support the profile specified when compiling the shader source code will result in an error or possible rendering anomalies [11].

The runtime translator supplied by NVIDIA supports any hardware whose drivers includes DirectX or OpenGL shading capabilities but generally compiles Cg shaders most efficiently for NVIDIA hardware [9]. Even though the Cg compiler source is available from NVIDIA to other vendors, none have attempted to implement custom compilers that optimize for their own hardware [8]. It is not surprising then that while Cg shaders will run on other hardware, they often run slower.

As stated, a Cg shader's syntax is nearly identical to one written in HLSL as shown in Figures 8 & 9 below. The resulting image from this example can be seen in Figure 3.

```
/*
Sample Cg Vertex Shader
Transforms an incoming vertex into clip-space and output to next
pipeline stage. Also passes along vertex color and texture coordinates.
*/

/*Structure containing per-vertex input values*/
struct appin
{
      float3 position  : POSITION;      //Object space vertex coords
      float4 color0    : COLOR0;        //Current vertex color
      float2 texcoord0 : TEXCOORD0;     //Current vertex texture
                                        //coords


};

/*Structure containing per-vertex output values*/
struct outfragment
{
      float4 position  : POSITION;    //Homogenous clip space
                                      //vertex coords
      float4 color0    : COLOR0;      //Current vertex color
      float2 texcoord0 : TEXCOORD0;   //Current vertex texture
                                      //coords

};

/*Vertex shader entry point*/
outfragment main( appin IN, uniform float4x4 worldViewProj )
{
      outfragment OUT;

      //Store incoming vertex's coordinates
      float4 v = float4( IN.position.x, IN.position.y, IN.position.z,
                         1.0f );
      //Transform current vertex into clip space using model-view-
      //projection matrix
      OUT.position  = mul( worldViewProj, v );
      OUT.color0    = IN.color0;
      OUT.texcoord0 = IN.texcoord0;

      //Output current vertex values
      return OUT;
}
```

Figure 8. Cg Simple Vertex Shader Sample

```
/*
Sample Cg Pixel Shader
Calculates the final color of the current pixel by looking up a diffuse
color value from a bound texture and adding the interpolated current
color.
*/

/*Structure containing per-pixel input values*/
struct fragmentin
{
      float4 position  : POSITION;     //Screen space x,y coordinates
      float4 color0    : COLOR0;       //Interpolated color
      float2 texcoord0 : TEXCOORD0;    //Interpolated texture coords.
};

/*Structure containing per-pixel output values*/
struct pixelout
{
      float4 color : COLOR;            //Final pixel color
};

/*Pixel shader entry point,
“uniform sampler2D testTexture” defines the texture to use in the
shader. This is set by the calling application.
*/
pixelout main( fragmentin IN, uniform sampler2D testTexture )
{
      pixelout OUT;

      //Using the interpolated texture coordinates for this pixel
      //look up a diffuse RGB color value from the bound texture
      //and add the current pixel color. The result is the final color
      //value for the current pixel.
      OUT.color = tex2D( testTexture, IN.texcoord0 ) + IN.color0;

      //Return current pixel values
      return OUT;
}
```

Figure 9. Cg Simple Pixel Shader Sample

3.3. GLSL

OpenGL is an open standard whose API is controlled by a central body. A majority of the members must agree upon any core additions to the language specification [13]. Due to competing interests of the voting members, OpenGL’s has lacked a high level shading until recently with the inclusion of GLSL into the official OpenGL 2.0 specification.

Even a standard assembly level shading language extension has only recently been added[3]. Unlike DirectX's and Cg's support for legacy hardware, GLSL is designed around high GPU hardware requirements. Targeted towards future hardware and for the sake of compiler efficiency, support for older hardware is extremely limited or non-existent [3]. Any fully compliant GLSL implementation must have hardware that equates roughly to the Shader 3.0 specification. Since few high-end cards currently support all the language's specified features, many vendors currently support a slightly reduced implementation of the language that corresponds to the Shader 2.x level of hardware requirements [6].

Contrary to the design of HLSL and Cg, GLSL moves all shader compilation and linking functions directly into the hardware's driver. Text based shader source is sent directly to the OpenGL driver where vertex or pixel shader objects are constructed and linked to form a GLSL program that will run in the programmable pipeline [13]. The developer has complete control over when these steps can occur through commands in the OpenGL API. With no intermediary translation layer and to keep with OpenGL design philosophies, vendors are provided with only a specification and are free to implement and optimize the GLSL compiler as best suits their respective hardware. As new hardware features become available vendors can easily modify the driver's GLSL compiler to make use of them without the need to redistribute separate updated run time libraries [3]. Figure 10 details the process involved in compiling GLSL shaders.

[3] ARB_vertex_program and ARB_fragment_program extensions [1]

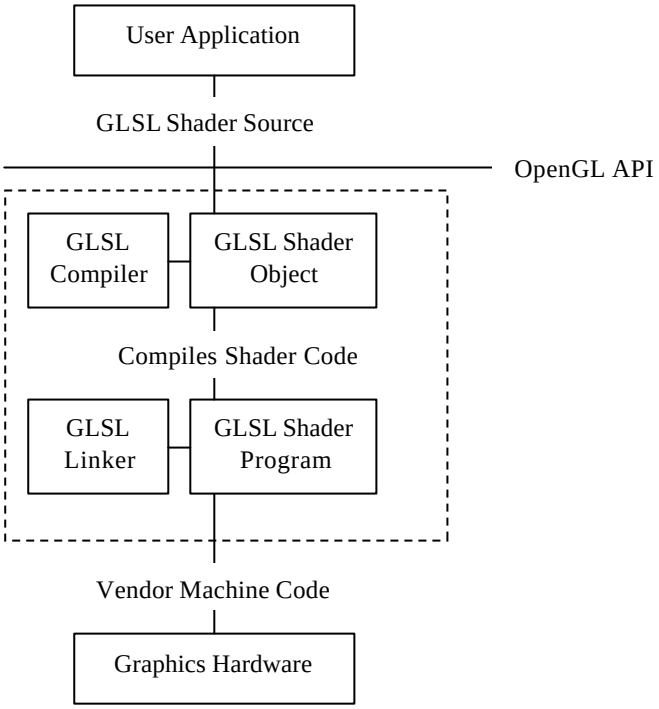

Figure 10: GLSL Compilation and Execution Process

As the burden for providing GLSL support rests completely in the hands of the vendors, only 3 vendors, ATI, NVIDIA, and 3DLabs, currently have fully functioning GLSL implementations.

The sample source below in Figures 11 & 12 illustrates GLSL's more compact syntax compared to HLSL and Cg. This compactness is a result of providing very tight integration with the OpenGL pipeline that helps to eliminate lengthy *in* and *out* pipeline data structures [6]. Once again, the resulting image can be seen in Figure 3.

```
/*
Sample GLSL Vertex Shader
Transforms an incoming vertex into clip-space and output to next
pipeline stage. Also passes along vertex color and texture coordinates.
*/

/*Vertex shader entry point*/
void main( void )
{
      //Transform current vertex into clip space using model-view-
      //projection matrix and output to next pipeline stage.
      // →gl_ModelViewProjectionMatrix (GLSL pre-defined, contains
      //   current ModelView-Projection Matrix from OpenGL pipeline)
      // →gl_Vertex (GLSL pre-defined, contains incoming vertex's
      //   x,y,z,w cords.)
      // →gl_Position (GLSL pre-defined, mandatory output variable for
      //   transform vertex)
      // →gl_TexCoord[0] (GLSL pre-defined, output variable for current
      //   vertex's texture coordinates)
      // →gl_FrontColor (GLSL pre-defined, output variable for current
      //   vertex's color)
      gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
      gl_TexCoord[0] = gl_MultiTexCoord0;
      gl_FrontColor = gl_Color;
}
```

Figure 11. GLSL Simple Vertex Shader Sample

```
/*
Sample GLSL Pixel Shader
Calculates the final color of the current pixel by looking up a diffuse
color value from a bound texture and adding the interpolated current
color.
*/

//Sampler that indicates the loaded texture to be sampled from
uniform sampler2D testTexture;

/*Pixel shader entry point*/
void main( void )
{
    //Using the interpolated texture coordinates for this pixel
    //look up a diffuse RGB color value from the bound texture
    //and add the current pixel color. The result is the final color
    //value for the current pixel and is assigned to the mandatory GLSL
    //pre-defined pixel color output variable, gl_FragColor.
    gl_FragColor = texture2D(testTexture, gl_TexCoord[0].xy)
    gl_FragColor += gl_Color;

}
```

Figure 12. GLSL Simple Pixel Shader Sample

4. Data Types and Language Functions

For any graphical shading language to be useful it must provide support for many of the basic data structures used for representing geometry, transformations and shading values. Geometrical structures can include vectors, points, matrices representing transformations and shading values described using arrays or scalar values [14]. These basic structures are needed within both the vertex and pixel shaders to describe incoming and outgoing data as well to represent intermediary calculations. For example, an incoming vertex needs to be represented by a 4 component vector and will be multiplied by a 4x4 model view projection matrix. However, as GPUs deal with a finite set of possible calculations, the same diversity seen in traditional programming languages such as C is unneeded in shading languages [12]. Basic data types supported by all three shading languages can be categorized into five groups: scalars, vectors, matrices, texture samplers, and user defined [5]. Even with a restricted set of available data types, those writing shaders should be aware that variable type declaration syntax differences exist between the languages.

Developed in conjunction with one another, HLSL and Cg share the greatest similarity in their supported basic data types. Supported data types for scalars and 2-4 component vectors and NxM matrices include boolean, int, half, float, double [10]. The actual syntax declaration is identical for both languages. When a shader wishes to sample a texel from a texture map a “texture sampler” must be declared. Texture samplers take one, two, or three component vectors representing texture coordinates and return the sampled three-component color value from that position in the texture map. Samplers are available for 1D, 2D, 3D and cube map textures. HLSL defines a single *sampler* data type where the type is defined as a parameter when sampling whereas Cg provides a separate

sampler type for each texture format (i.e. sampler2D) [9]. Additionally, both languages support *Typedef* and *Struct* types that allow for user defined grouping of the above basic types. Figure 13 below is an example of some simple variable declarations for both languages.

```
float3 myColor = float3(0.1,0.4,0.5);     //3 component float vector
float4x4 modelView;                       //4x4 float matrix
sampler2D myTexture;                      //Cg 2D texture sampler
struct appin  {                           //User defined struct
      float4 Position    : POSITION;       //with binding semantics
      float4 Normal      : NORMAL;
};
```

Figure 13. Example declaration of HLSL and Cg data types

Both languages also support the use of *binding semantics* as can be observed in Figure 13 by the inclusion of the ": POSITION" postfix of the variable declaration. Binding semantics allow for declared input and output variables to be specifically bound to hardware defined registers. Some, such as POSITION, must always be present as an input of a vertex shader since without it the current incoming vertex position from the API will not be available unless it is explicitly assigned to a variable. Vertex and pixel shaders for both HLSL and Cg share a large number of possible binding semantics, with both types of shaders requiring certain semantics that must be bound for correct functionality [10].

In comparison to HLSL and Cg, the designers of GLSL chose to include a more restrictive set of syntax rules for declaring data types. The available scalar data types are bool, int and float; similar to the other shading languages. In GLSL, a *vec* data type, instead of a normal C-style array, represents vectors of any of these types. Two, three and

four component vec types are available for booleans, integers and floats, declared as bvec(2,3,4), ivec(2,3,4) or fvec(2,3,4) [13]. A *mat* data type is also available for use, however only matrices of NxN floating point values are allowed unlike the NxM dimension matrices of HLSL and Cg [3]. It is these differences in data types that represent one of the greatest differences in syntax that GLSL contains. Texture samplers are declared as in Cg with additional support for shadow map texture samplers [6]. The use of structs is also available. Figure 14 shows an example containing a number of variable declarations under GLSL.

```
vec3 myPosition = vec3(2.0,3.4,-5.1);     //3 component vec
mat4 modelViewMtx;                        //4x4 float matrix
sampler2D mysampler;                      //A 2d texture
                                          //sampler
sampler1DShadow shadowMap;                //A 1D shadow
                                          //texture sampler
```

Figure 14. Example declaration of GLSL data types

Essential to all shading languages is the ability to support a wide selection of common mathematical and geometric calculations. Coordinate system transforms, lighting calculations, and coloring mixing are common techniques used by the fixed function pipeline and a programmable pipeline requires even greater support for functions that may be needed by a wide variety of possible rendering algorithms [14]. All three shading languages other excellent support for many useful built-in functions such as vector dot-product, vector cross-product, matrix multiplication, vector normalization, range clamping, trigonometric cosine, trigonometric sine, vector reflection, linear interpolation of values, maximum and minimum calculation, logarithms and many more [11]. Many of

the functions act as they do in C and the majority have direct vertex and fragment machine instruction implementations, providing fast execution times. While all three languages specifications include a diverse selection of available functions, developers should ensure that a particular function has actually been implemented before using it. Some functions such as GLSL's noise() function for generating random noise values is not actually implemented in any current vendor compilers [3].

5. Available Maintenance Features

When making use of shaders the situation can quickly arise where a developer begins writing separate vertex and pixel shaders to calculate shading properties for every type of surface within a given 3d space. For a complex scene containing dozens of unique surfaces, each described by a different shader, shader developers can find themselves trying to manage a large number of separate source files. Therefore, there exist some features inherent to each of the shading languages' designs that attempt to help manage the maintenance of shaders. One concept is the ability to not restrict all source code for a single shader to exist in one exclusive file or to force shader developers to manually parse and combine separate code fragment files. HLSL uses the concept of *shader fragments* that allow multiple shaders to be combined together before they are compiled. A central shader source fragment that defines the vertex or pixel shader's main() entry point can call separate functions that are defined in other shader fragments [10]. Using the DirectX API, a complete shader can be constructed and passed to the driver.

Similarly, GLSL provides an elegant solution to combining shader source modules by allowing multiple *shaders objects* to be attached to a *shader program* before being linked [3]. This greatly increases the ability to easily maintain functions shared by multiple independent shaders.

Cg currently provides no mechanism to combine user-defined functions in a single shader. A single shader program accepts only the complete and final source string for a shader. Should a similar maintenance feature be required when using Cg, a user could implement their own set of functions to take in multiple shader source strings and

combine them into a complete program before passing it to the Cg runtime to be compiled [12].

Within a commercial environment it may also be desirable to protect rendering algorithms used in applications or prevent users from manually editing shaders to change their specific functionality. Currently only HLSL supports at least a partial solution by allowing HLSL shaders to be pre-translated into the custom DirectX assembly shader stream format [13]. For Cg and GLSL, solutions such as embedding shader source strings directly into the compiled application executable or packing shaders into a custom archive format are the only alternatives.

6. Pipeline State Access

Another aspect of any real time shading language that is crucial to being able to easily integrate a shader framework into a graphical application is the ability to access certain elements of the fixed function pipeline from within the shaders that replace the normal vertex and fragment processing stages. A program that utilizes the DirectX or OpenGL APIs will usually set and modify a large number of pipeline states in between rendering passes such as transformation matrices, lighting values, and surface material and fog settings. Allowing an application to specify these values and giving the shaders access to them through some standard mechanism provides a convenient method to easily integrate shaders into the pipeline.

As OpenGL's architecture uses a state machine methodology, state values specified through the API become part of the current context until they are once again modified by the user [1]. This design is carried over into GLSL through the inclusion of a wide variety of predefined uniform variables that permit access to state information. For vertex shaders this includes the current incoming vertex (*gl_Position*), normal (*gl_Normal*), modelview matrix (*gl_ModelViewMatrix*), projection matrix (*gl_ProjectionMatrix*), fog settings (*gl_FogParameters* struct), and any enabled light properties (*gl_LightSourceParameters* struct) [6]. The advantage to this approach is that it greatly reduces the number of *varying* (per primitive), *uniform* (per pass), and *attribute* (per vertex) values that must be passed explicitly to either shader and reduces the possibility of introducing errors into the application. This is contradictory to the approach used by HLSL and Cg shaders that target DirectX. Shaders designed to run under the DirectX programmable pipeline must pass all values that are needed by a shader before it

is set as active [10]. This may require an application to continually query current values from the fixed function pipeline and resubmit the obtained values as shader parameters. Semantic bindings help to alleviate this restriction somewhat but only for per vertex or per pixel information such as position or color, not for those such as the current model view matrix.

When using Cg targeted towards the OpenGL programmable pipeline, the DirectX style restrictions are dropped and complete state access is granted similar to GLSL [11]. A single predefined structure *glstate*, contains complete accessibility to all states useful in the vertex and fragment processing stages of the pipeline. However this feature also illustrates how Cg shaders can easily lose their cross API appeal. For Cg shaders to remain compatible with both the DirectX and OpenGL rendering pipeline, state variables have to be explicitly sent by the user [3].

7. Implementation and Driver Maturity

Any developer wishing to incorporate high level shader features into a project must be aware of the current stability issues that may be present in any of the three shading languages' compilers and drivers.

Having gone through a number of iterations and design changes, HLSL can be considered the most stable of all the current shading languages. The introduction of DirectX 9.0c completed a long series of driver and API refinements, producing a robust framework for utilizing vertex and pixel shaders including a shader effects framework that allows for multiple shaders to be collected into rendering passes, easing the amount of code necessary to implement complex rendering [4]. Graphics hardware vendors wishing to advertise there products as DirectX 9 compliant must include drivers that fully support the HLSL specification either directly in hardware or through a software fallback code path [10]. With its tight integration into the general DirectX libraries, the HLSL runtime and drivers are available exclusively on the Microsoft Window's operating system.

Targeted as a cross platform shading language, Cg's runtime libraries are surprisingly stable and error free. The Cg runtime exists for Microsoft Windows, Apple OSX and Linux operating systems. Since Cg's runtime translator converts shader source code to either DirectX shader assembly streams or OpenGL assembly extension instructions, a system that properly supports either of these standards should theoretically support Cg [11]. Being controlled by a single vendor, the Cg translator is updated frequently and at version 1.3 supports many of the most recent NVIDIA hardware features. As described previously, while the translator is able to target a wide variety of

hardware, optimization is rarely performed for non-NVIDIA hardware, instead producing the most straightforward interruption [7]. A developer wishing to implement an application that utilizes Cg should be aware of this fact if they wish to support a wide user base. One operating system where Cg is the only option is Apple's OS X. OS X's visual sub system relies heavily upon OpenGL. As Apple restricts vendors from releasing drivers outside of their development network, GLSL support has still not been exposed. Until Apple activates support for GLSL, Cg is currently the only available high level shading language available for their platform.

As the youngest of all three shading languages, many drivers for GLSL are still considered experimental by their vendors [13]. With modifications and refinements still being made to the specification, coupled with its high hardware requirements for full compliance, many features of the language are still unavailable in certain implementations [1]. Graphic hardware vendors ATI, NVIDIA and 3DLabs currently provide a satisfactory level of support for GLSL, with 3DLabs offering the most mature compiler [13]. As each vendor must provide a shader source code-to-machine instruction GLSL compiler entirely within their respective drivers, numerous incompatibilities are still present. Some have even forgone writing a full compiler instead utilizing alternative implementations. NVIDIA, whose current line of graphics hardware is the only one to incorporate hardware that allows a complete GLSL implementation, chooses to provide an internal GLSL to Cg translator [3]. The advantage of this approach is that by moving the Cg translator directly into the drivers a stable and reliable compiler is available to produce hardware specific code. Unfortunately, this has produced the side effect of

making the interpretation GLSL source compilation errors difficult, as currently, vague Cg error strings may be returned.

8. Execution Speed Comparisons

To illustrate the difference in driver and compiler maturity, the results of two simple shaders emphasizing vertex bound and fragment bound shading operations are given. A simple custom C++ framework was used to test corresponding implementations of both shaders for each of the three shading languages. Under both tests a timer function recorded the average number of completed frames that could be rendered in one second (FPS). The tests were intended to stress their respective stages of the pipeline through a combination of large batches of incoming geometry from the API and a non-trivial number of shader instructions being performed.

The vertex bound test comprised of a single vertex shader that was responsible for transforming the incoming vertex into clip space after applying a series of trigonometric cosine and sine values to the vertex. A highly tessellated sphere comprised of approximately 50,000 vertices was sent through each API to the shader. Figure 15 shows a screenshot of the resulting sphere with the vertex shader applied, producing an animated wobble effect. The vertex offsets that are applied to each vertex are computed entirely on the GPU. The sphere is rendered in wireframe mode so that the vertex dense surface of the sphere can be observed. The source code for this vertex shader is given in Appendix 1.

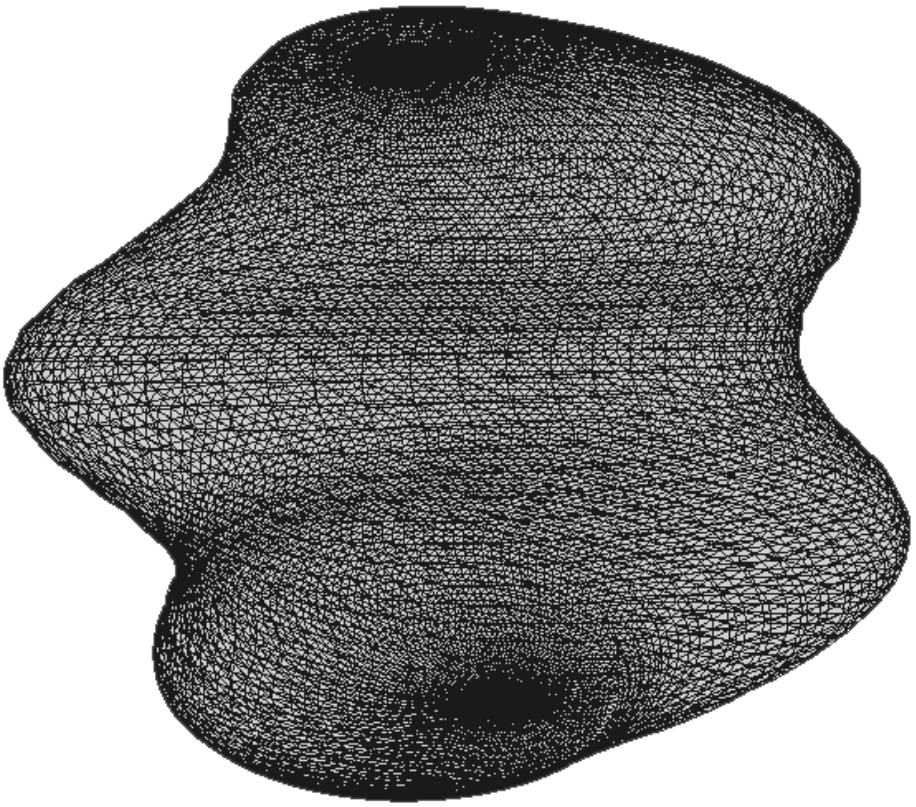

Figure 15: Tessellated sphere with Wobble vertex shader applied

For the fragment bound test, a per-pixel lighting technique was applied called *normal-mapping*. A simple sphere comprised of 50 primitives was sent into each API along with a pre-computed matrix for each vertex that provided an object space-to-tangent space transformation [13]. In the vertex shader this matrix was used to transform the passed light position into tangent (or texture) space. The pixel shader then calculated the lit value of each surface pixel by looking up the surface normal value from a normal texture and performing the dot product of the acquired normal vector with the transformed light vector. This intensity was then combined with a diffuse surface texture to produce a final color value. This test produced a surface image that appears to have

greater detail and depth over simple diffuse texture mapping. This technique is only possible using a programmable pipeline since the normal texture lookup and lighting calculation must be performed per-pixel using a pixel shader. Figure 16 shows the diffuse texture and Figure 17 shows the pre-computed normal texture from which normal values are retrieved. Figure 18 gives the final results of the pixel shader applied to the sphere. The source code for this pixel shader is available in Appendix 2.

Figure 16: Diffuse texture used in test pixel shader

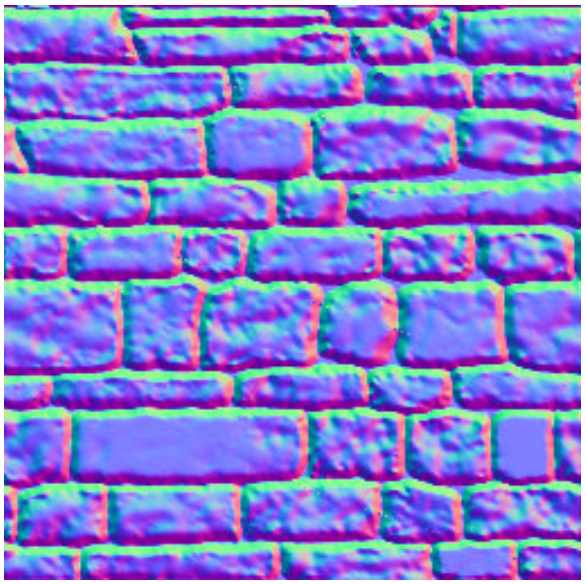

Figure 17: Normal texture used in test pixel shader

Figure 18: Sphere with applied pixel shader

Both tests were performed on an AMD-64 3000+ desktop with 1Gig of RAM running Microsoft Windows XP with a 128MB NVIDIA 6800 video adaptor. All drivers and language compilers consisted of the most recent versions available (Video adapter

driver: Forceware 71.84, Cg runtime: 1.3, DirectX runtime: 9c). The described tests produced the following results:

Table 1: Frames per second test results

Test/Language	HLSL	Cg 1.3 (OpenGL)	GLSL
Vertex bound	123	120	115
Fragment bound	102	91	94

Both tests performed under all three shading languages produced similar results with only very slight differences most likely attributable to the video adaptor's drivers. The similarities in the vertex bound test may also suggest that a bus limit from user application memory to the video adaptor was present. The differences present in the pixel bound test results are most likely due to the maturity of the adapter's drivers and the language's compilers. As these tests where performed under only one set of conditions, they are only intended to give an estimate of what performance results may be expected. What can be observed is that the programmable pipeline offers significant processing power for a variety of shading applications using any of the three shading languages.

For the vertex bound test to be performed under the fixed function pipeline each of the 50,000 vertices of the tessellated sphere would have to have its wobble offset recalculated each frame on the CPU. Performing such a high number of floating point calculations each frame may reduce performance of other running processes that require high CPU usage.

For the fragment bound test, the results in Table 1 show that a complex lighting algorithm other then the fixed function pipeline's Phong model can be utilized and produce real-time frame rates (FPS > 30). Under a fixed function pipeline, the normal

mapping lighting technique's results could not be reproduced in real time, as it requires the ability to perform specific per-pixel calculations in the 3d pipeline. The algorithm can currently only be performed on the CPU through the use of non-real time, ray-tracing software.

9. Future Roadmaps

With the pace at which programmable pipeline technology is being adopted for a wide variety of applications, the general consensus is that eventually a significant amount of time spent on developing graphical applications will be spend implementing shaders [3]. Also, future advances in graphical hardware will continue to force these shading languages to be modified to expose even greater programmability within the graphics pipeline.

The most likely change to happen in the near future is the merging of available functions and constructs between vertex and pixel shaders. Some of the current hardware restrictions that prevent both types of shaders from sharing certain functions, such as vertex shaders accessing texture memory or pixel shaders rendering to vertex arrays, are already disappearing [6]. Once the majority of standard hardware supports such features, the distinction between vertex and pixel shaders will exist only at the application development level. GLSL is currently the only language with a specification that has anticipated such changes.

Additionally, future roadmaps may permit other stages of the remaining fixed function pipeline to be transformed into a programmable framework, most notable the frame-buffer operation stage [8]. Currently, for practical reasons, shaders cannot read from the frame-buffer and all frame-buffer operations can only be specified through corresponding API calls [1]. A third type of shader, a frame-buffer shader, could allow for user defined image space operations to be performed.

10. Conclusion and Recommendations

With powerful real-time commodity 3d hardware becoming increasingly standard, high-level graphical shading languages now present flexible means to incorporate real-time rendering algorithms into graphical applications by utilizing the programmability of GPUs. HLSL, Cg, and GLSL all offer various advantages and disadvantages that should be considered before custom shading technology is used. While many of the core syntax, available data types and built-in functions are largely similar and offer similar levels of performance, issues such as driver implementation and operating system support will usually be the determining factor in what shading language is most useful.

As all three languages offer comparable features regarding syntax and library functions, the decision of what language to use currently rests on two aspects: platform and hardware. Due to the strong ties between all three shading languages and the underlying graphics hardware, any project targeted towards a wide variety of users should incorporate a suitable fallback code path as the diversity in present consumer hardware severely limits the use of advanced shaders, especially those written in GLSL. Additionally, shaders that simply recreate the standard fixed function pipeline should be avoided as they are already heavily optimized within the vendor's drivers.

If an application is written using DirectX under the Windows operating system then HLSL is the optimal choice due to is stable translator and drivers. HLSL's ability to generate vertex and pixel shaders for many levels of graphics hardware and features that protect shader source makes it appealing for those who produce commercial interactive products. Conversely, those utilizing OpenGL currently have a choice of whether to use Cg and support a wider variety of hardware or use GLSL that while requiring demanding

hardware, is designed to be a core component of the OpenGL 2.0 specification [6]. As a recommendation, any OpenGL application that is to be released to a wide audience within the next year and wishes to target a variety of operating systems should utilize Cg. Starting in 2006, GLSL would be ideal as by that time wide support for GLSL should be available.

Despite what language is ultimately chosen, the language specification documents available for each language should be thoroughly reviewed before shader development is begun. As this report has detailed, there are many small aspects of each of the three shading languages that should be carefully considered.

As programmable pipeline and shading technologies continue to be refined and standardized with support provided from additional vendors, shading languages will prove essential for opening up new techniques within the field of computer graphics and imaging.

References

[1] Bylthe, D., and McReynolds, T., *Advanced Graphics Programming Using OpenGL*, Morgan Kaufman, 2005.

[2] Elliot, C., *Programming Graphics Processors Functionally*, In Proceedings of the ACM SIGPLAN workshop on Haskell, 2004, pp. 45-56.

[3] Engel, W., ed., *ShaderX3: Advanced Rendering with DirectX and OpenGL*, Charles River Media, 2005.

[4] Fosner, R. *Real-Time Shader Programming*, Morgan Kaufmann Publishers, 2003.

[5] Heidrich, W., and Seidel, H., *Realistic, Hardware-accelerated Shading and Lighting*, In Proceedings of the 26th annual conference on Computer graphics and interactive techniques, Los Angeles, California, USA, August 8-13, 1999, pp. 171-178.

[6] Kessenich, J., Baldwin D, and Rost, R., *OpenGL Shading Language v 1.10*, 2004. [Online]. Available at http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf

[7] LaMothe, A., *Tricks of the 3D programming gurus: advanced 3D graphics and rasterization*, Sams, 2003.

[8] Lastra, A., Molnar, S., Olano, M., and Wang, Y., *Real-Time Programmable Shading*, In Proceedings of the 1995 symposium on Interactive 3D graphics, Monterey, California, USA, April 9-12, 1995, pp. 55-ff.

[9] Mark, W., Glanville, .S, Akeley, K., and Kilgard, M., *Cg: A system for programming graphics hardware in a C-like language*, In Proceedings of ACM SIGGRAPH 2003, Los Angeles, California, USA, August 8-12, 2003, pp. 896-907.

[10] Microsoft Corporation, *HLSL Shader Reference*, Microsoft MSDN, 2005. [Online]. [Updated 6 January 2005]. Available at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/reference/hlslreference/hlslreference.asp

[11] NVIDIA Corporation, *Cg Toolkit: User's Manual (Release 1.3)*, 2005. [Online]. Available at ftp://download.nvidia.com/developer/cg/Cg_1.3/Docs/Cg_1.3_Docs.zip

[12] Peercy, M., Olano, M., Airey, J., and Ungarm, P., *Interactive Multi-Pass Programmable Shading*, In Proceedings of the 27th annual conference on Computer graphics and interactive techniques, New Orleans, Louisiana, USA, July 23-28, 2000, pp. 425-432.

[13] Rost, R. A., *OpenGL Shading Language*, Addison-Wesley, 2003.

[14] Watt, A., *3D Computer Graphics (Third Edition)*, Addison-Wesley, 2000.

Appendix I: Shading Language Execution Time Test Case 1: Vertex-Bound Source Code

The following source code listing provides the information necessary to create and run the vertex-bound shader used in Test Case 1. For brevity only the initiation and per-frame update code used in the C++ test application framework on which the graphical shaders' usage depends is provided. This source can be easily integrated into any windowing system that provides the ability to set up a DirectX or OpenGL window context and handle user events. Further error checking of many of the function results should also be used. The number of frame per second (FPS) can be measured by implementing a simple timer that records the number of times that a complete frame can be drawn in one second.

Source code for the test vertex shader is supplied in its entirety for all three graphical shading languages.

User Application Global Variables:

```
/*********************************************************************/

/* Used by HLSL*/
//DirectX 9 device
LPDIRECT3D9             g_pD3D          = NULL;
LPDIRECT3DDEVICE9       g_pd3dDevice    = NULL;

//Pointer to vertex shader
LPDIRECT3DVERTEXSHADER9      g_pVertexShader = NULL;
LPDIRECT3DVERTEXDECLARATION9 g_pVertexDeclaration = NULL;

//Shader constant table – stores the locations of all user configurable
//variables that exist inside the shader
LPD3DXCONSTANTTABLE          g_pConstantTableVS = NULL;

/*********************************************************************/

/* Used by Cg*/
CGprofile   g_CGprofile;             //Cg Shader profile
CGcontext   g_CGcontext;             //Cg context
CGprogram   g_CGprogram;             //Cg shader program

CGparameter g_CGparam_ModelViewMatrix;
CGparameter g_CGparam_Timer;         //Cg Parameters used to send
CGparameter g_CGparam_Vertical;      //values to the compiled
CGparameter g_CGparam_Horizontal;    //shader each frame
CGparameter g_CGparam_TimeScale;     //

/*********************************************************************/

/* Used by GLSL*/
GLhandleARB g_programObj;            //Handle to GLSL program object
GLhandleARB g_vertexShader;          //Handle to compiled GLSL vertex
```

```
                                        //shader
GLuint g_location_Timer;                //Binding to uniform variable in
GLuint g_location_Vertical;             //the compiled vertex shader.
GLuint g_location_Horizontal;           //Allows updated values to be sent
GLuint g_location_TimeScale;            //to the active shader each frame
```

User Application Shader Initialization (Performed once upon application creation):

```
//Load/Compile the vertex shader for each shading language
switch(Language)
{
      case HLSL:
      {
          //Since our vertex shader uses explicit binding semantics we
          //have to create a vertex declaration using those semantics.

          D3DVERTEXELEMENT9 declaration[]={{0,0,D3DDECLTYPE_FLOAT3,
                          D3DDECLMETHOD_DEFAULT,D3DDECLUSAGE_POSITION,
                          0},{0,12,D3DDECLTYPE_D3DCOLOR,
                          D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR,
                          0},{0,16,D3DDECLTYPE_FLOAT2,
                          D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD,
                        0},D3DDECL_END()};

           //Set this declaration
           g_pd3dDevice->CreateVertexDeclaration(declaration,
                                              &g_pVertexDeclaration);

            HRESULT hr;
            LPD3DXBUFFER pCode;       //The buffer the shader's code is
                                      //loaded into.
            DWORD dwShaderFlags = 0;
            LPD3DXBUFFER pBufferErrors = NULL;

            //Assemble the vertex shader from the file.
            // "main" - specifies the shaders entry point
            // "vs_1_1" - specifies the vertex shader level to try
            //              and compile for
            //Any uniform values in the shader source will have
            //location entries made in the g_pConstantTableVS object
            hr = D3DXCompileShaderFromFile("wobble_vert.hlsl", NULL,
                  NULL,"main","vs_1_1", dwShaderFlags, &pCode,
                  &pBufferErrors, &g_pConstantTableVS );

            // Create the vertex shader
            g_pd3dDevice->CreateVertexShader(
                              (DWORD*)pCode->GetBufferPointer(),
                              &g_pVertexShader);
            pCode->Release();
            break;
      }
      case CG:
      {
            //Choose a vertex profile (in this case OpenGL assembly
            //profile)
            g_CGprofile = CG_PROFILE_ARBVP1;

            //Create a Cg Context
            g_CGcontext = cgCreateContext();
```

```
            //Create the vertex shader program using the created
            //context. We read in the string source from
            //“wobble_vert.cg”. The translator will try and compile
            this //to work with the specified profile.
            g_CGprogram = cgCreateProgramFromFile( g_CGcontext,
                              CG_SOURCE, " wobble_vert.cg",
                              g_CGprofile, NULL, NULL);

            //Load the program using Cg's interface
            cgGLLoadProgram(g_CGprogram);

            // Get handles to the uniform variables we will set later
            g_CGparam_ModelViewMatrix = cgGetNamedParameter(
                                g_CGprogram, "modelViewProjMatrix");

            g_CGparam_Timer = cgGetNamedParameter(g_CGprogram,
                              "Timer");
            g_CGparam_Vertical = cgGetNamedParameter(g_CGprogram,
                              "Vertical");

            g_CGparam_Horizontal = cgGetNamedParameter(g_CGprogram,
                              "Horizontal");

            g_CGparam_TimeScale = cgGetNamedParameter(g_CGprogram,
                              "TimeScale");

            break;
        }
        case GLSL:
        {
           //Create the vertex shader object
           g_vertexShader =glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);

           //User defined function that reads in shader source strings
           //from “wobble_vert.glsl” text file
           unsigned char *vertexShaderSource = readShaderSource(
                                               "wobble_vert.glsl");

           //Covert to string array parameter for next function call
           vertexShaderStrings[0] = (char*)vertexShaderAssembly;

           //Send the read in vertex shader source to vertex program
           //object.
           glShaderSourceARB(g_vertexShader,1,vertexShaderStrings,
                             NULL);

           //Now that the program has source code, compile the shader
           glCompileShaderARB(g_vertexShader);

           //Create a  GLSL program object and attach the compiled shader
           g_programObj = glCreateProgramObjectARB();
           glAttachObjectARB(g_programObj, g_vertexShader );

           //Link the GLSL program object containing the compiled GLSL
           //vertex shader.
           glLinkProgramARB(g_programObj);
```

```
         //Get handles to all the uniform variables we will set later
         g_location_Timer = glGetUniformLocationARB(g_programObj,
                                                    "Timer");
         g_location_Vertical = glGetUniformLocationARB(g_programObj,
                                                    "Vertical");
         g_location_Horizontal = glGetUniformLocationARB(g_programObj,
                                                    "Horizontal");
         g_location_TimeScale = glGetUniformLocationARB(g_programObj,
                                                     "TimeScale");
         break;
      }
}
```

Per Frame (Performed once per frame):

```
//Each redraw of the window we activate the shaders and pass in any
//dynamic variable values they may use.
switch(Language)
{
      case HLSL:
      {
                                    .
                                    .
                                    .
                                    .
                                    .
                           Update any DirectX states
                                    .
                                    .
                                    .
                                    .
                                    .

            //Get the current Model View Projection Matrix
            D3DXMATRIX worldViewProjection = g_matWorld * g_matView *
            g_matProj;

            //Pass it into the constants table
            g_pConstantTableVS->SetMatrix( g_pd3dDevice,
            "worldViewProj", &worldViewProjection );

            //Set the Horizontal, Vertical, Timer, and TimerScale
            //values in the shaders constant table
            g_pConstantTableVS->SetFloat(g_pd3dDevice,
                                         "Horizontal",0.14f);
            g_pConstantTableVS->SetFloat(g_pd3dDevice,
                                         "Vertical",7.5f);
            g_pConstantTableVS->SetFloat(g_pd3dDevice,
                                         "TimeScale",5.4f);
            g_pConstantTableVS->SetInt(g_pd3dDevice,
                                         "Timer",deltaMilliseconds);


            //Set vertex declaration and make wobble vertex shader
            //active
```

```
            g_pd3dDevice->SetVertexDeclaration( g_pVertexDeclaration );
            g_pd3dDevice->SetVertexShader( g_pVertexShader );

                                        .
                                        .
                                        .
                                        .
                                        .
                                Draw Sphere
                                        .
                                        .
                                        .
                                        .
            //Deactivate current vertex shader
            g_pd3dDevice->SetVertexShader(NULL);


      }
      case CG:
      {
                                        .
                                        .
                                        .
                                        .
                        Update any DirectX/OpenGL states
                                        .
                                        .
                                        .
                                        .
            //Enable vertex Profile used by the wobble vertex shader
            CgGL.cgGLEnableProfile(g_CGprofile);

            //Activate the wobble vertex shader
            CgGL.cgGLBindProgram(g_CGprogram);

            // Set the "modelViewProjMatrix" parameter in the vertex
            //shader to the current concatenated
            // modelview and projection matrix
            CgGL.cgGLSetStateMatrixParameter(g_CGparam_ModelViewMatrix,
                                CgGL.CG_GL_MODELVIEW_PROJECTION_MATRIX,
                                CgGL.CG_GL_MATRIX_IDENTITY);

            //Set the Horizontal, Vertical, Timer, and TimerScale
            //values used by the vertex shader
            CgGL.cgGLSetParameter1i(g_CGparam_Timer,deltaMilliseconds);
            CgGL.cgGLSetParameter1f(g_CGparam_Horizontal,0.14f);
            CgGL.cgGLSetParameter1f(g_CGparam_Vertical,7.5f);
            CgGL.cgGLSetParameter1f(g_CGparam_TimeScale,5.4f);
                                        .
                                        .
                                        .
                                        .
                                Draw Sphere
                                        .
                                        .
                                        .
                                        .
```

```
            //Disable the vertex shader profiles
            CgGL.cgGLDisableProfile(g_CGprofile);


      }
      case GLSL:
      {
                                    .
                                    .
                                    .
                                    .
                        Update any OpenGL states
                                    .
                                    .
                                    .
                                    .
                                    .
            //Enable the GLSL shader program containing the compiled
            //wobble vertex shader
            glUseProgramObjectARB(g_programObj);

            //Set the Horizontal, Vertical, Timer, and TimerScale
            //values used by the vertex shader
            glUniform1iARB(g_location_Timer,deltaMilliseconds);
            glUniform1fARB(g_location_Horizontal,0.14f);
            glUniform1fARB(g_location_Vertical,7.5f);
            glUniform1fARB(g_location_TimeScale,5.4f);
                                    .
                                    .
                                    .
                                    .
                                    .
                               Draw Sphere
                                    .
                                    .
                                    .
                                    .
            // Disable the GLSL shader objects
            glUseProgramObjectARB(NULL);
      }
}
```

HLSL Shader Source:

```
/*********************************************************************
Vertex-Bound Test: HLSL Vertex Shader
Source file: “wobble_vert.hlsl”
*********************************************************************/

/* Uniforms – changed at most once per frame*/
float4x4 matViewProjection;
float Timer;
float Horizontal;
float Vertical;
float TimeScale;

/* Vertex data from application*/
struct VS_INPUT
{
   float4 Position : POSITION0;

};

/* Data passed out from vertex shader */
struct VS_OUTPUT
{
   float4 Position : POSITION0;

};

VS_OUTPUT vs_main( VS_INPUT Input )
{
    VS_OUTPUT Output;

    //Scale down time
    float timeNow = (Timer)*TimeScale;

    //Store current incoming vertex
    float4 Po = float4(Input.Position.xyz,1);

    //Calulate offset values for “wobble” effect
    float iny = Po.y * Vertical + timeNow;
    float wiggleX = sin(iny) * Horizontal*30;
    float wiggleY = cos(iny) * Horizontal*30;

    //Apply wobble offsets to current vertex’s x and y coordinates
    Po.y = Po.y + wiggleY/5;
    Po.x = Po.x + wiggleX;

    //Transform this new vertex position into clip space and store
    //in output structures position member
    Output.Position = mul(matViewProjection,Po);
   return( Output );
}
```

Cg Shader Source:

```
/*********************************************************************
Vertex-Bound Test: Cg Vertex Shader
Source file: “wobble_vert.cg”
*********************************************************************/

uniform float4x4 WorldViewProj;
uniform float Timer;
uniform float Horizontal;
uniform float Vertical;
uniform float TimeScale;

/* Vertex data from application*/
struct appdata {
    float3 Position     : POSITION;
};

/* Data passed out from vertex shader */
struct vertexOutput {
    float4 HPosition    : POSITION;
};

/*Entry point of vertex shader*/
vertexOutput main(appdata IN) {

    //Our output structure
    vertexOutput OUT;

    //Scale down time
    float timeNow = Timer*TimeScale;

    //Store current incoming vertex
    float4 Po = float4(IN.Position.xyz,1);

    //Calulate offset values for “wobble” effect
    float iny = Po.y * Vertical + timeNow;
    float wiggleX = sin(iny) * Horizontal;
    float wiggleY = cos(iny) * Horizontal;

    //Apply wobble offsets to current vertex’s x and y coordinates
    Po.y = Po.y + wiggleY/5;
    Po.x = Po.x + wiggleX;

    //Transform this new vertex position into clip space and store
    //in output structures position member
    OUT.HPosition = mul(Po, WorldViewProj);
    return OUT;
}
```

GLSL Shader Source:

```
/*********************************************************************
Vertex-Bound Test: GLSL Vertex Shader
Source file: “wobble_vert.glsl”
*********************************************************************/


/* Uniforms – changed at most once per frame*/
uniform int TIME_FROM_INIT;          //Timer (allows animated effect)
uniform float TimeScale;             //Speed up/slow down wobble effect
uniform float Horizontal;            //Amplitude
uniform float Vertical;              //Wave length

/*Entry point of vertex shader*/
void main()
{
    //Scale down time
    float r = float(TIME_FROM_INIT) / 650.0;
    float timeNow = r*TimeScale;

    //Store current incoming vertex
    vec4 Po = vec4(gl_Vertex.xyz,1);

    //Calulate offset values for “wobble” effect
    float iny = Po.y * Vertical + timeNow;
    float wiggleX = sin(iny) * Horizontal;
    float wiggleY = cos(iny) * Horizontal;

    //Apply wobble offsets to current vertex’s x and y coordinates
    Po.y = Po.y + wiggleY / 5.0;
    Po.x = Po.x + wiggleX;

    //Transform this new vertex position into clip space and store in
    //GLSL vertex output variable
    gl_Position = gl_ModelViewProjectionMatrix*Po;
}
```

Appendix II: Shading Language Execution Time Test Case 2: Pixel-Bound Source Code

The following source code listing provides the information necessary to create and run the pixel-bound shader used in Test Case 2. For brevity, only the initiation and per-frame update code used in the C++ test application framework on which the graphical shaders' usage depends is provided. This source can be easily integrated into any windowing system that provides the ability to set up a DirectX or OpenGL window context and handle user events. Further error checking of many of the function results should also be used. The number of frame per second (FPS) can be measured by implementing a simple timer that records the number of times that a complete frame can be drawn in one second.

Source code for the test vertex shader and pixel shader are supplied in their entirety for all three graphical shading languages.

User Application Global Variables:

```
/*********************************************************************/

/* Used by HLSL*/
//DirectX 9 device
LPDIRECT3D9             g_pD3D          = NULL;
LPDIRECT3DDEVICE9       g_pd3dDevice    = NULL;

//Pointer to vertex shader
LPDIRECT3DVERTEXSHADER9      g_pVertexShader = NULL;
LPDIRECT3DVERTEXDECLARATION9 g_pVertexDeclaration = NULL;
//Shader constant table – stores the locations of all user configurable
//variables that exist inside the vertex shader
LPD3DXCONSTANTTABLE          g_pVertexConstantTableVS = NULL;

//Pointer to pixel shader
LPDIRECT3DPIXELSHADER9       g_pPixelShader      = NULL;
//Shader constant table – stores the locations of all user configurable
//variables that exist inside the pixel shader
LPD3DXCONSTANTTABLE          g_pPixelConstantTablePS  = NULL;

/*********************************************************************/

/* Used by Cg*/
CGcontext   g_CGcontext;             //Cg context
CGprofile   g_CGvertexprofile;       //Cg vertex shader profile
CGprofile   g_CGpixelprofile;        //Cg pixel shader profile
CGprogram   g_CGvertexprogram;       //Cg vertex shader program
CGprogram   g_CGpixelprogram;        //Cg pixel shader program


CGparameter g_CGparam_LightColor;         //Cg Parameters used to send
CGparameter g_CGparam_LightPosition;      //values to the compiled
CGparameter g_CGparam_ModelViewMatrix;    //shader each frame
```

```
/*****************************************************************************/

/* Used by GLSL*/
GLhandleARB g_programObj;            //Handle to GLSL program object
GLhandleARB g_vertexShader;          //Handle to compiled GLSL vertex
                                     //shader
GLhandleARB g_pixelShader;           //Handle to compiled GLSL pixel
                                     //shader

GLuint g_location_LightColor;        //Binding to uniform variable in
                                     //the compiled pixel shader.
```

User Application Shader Initialization (Performed once upon application creation):

```
//Load/Compile the vertex & pixel shaders for each shading language
switch(Language)
{
      case HLSL:
      {
          //Since our vertex shader uses explicit binding semantics we
          //have to create a vertex declaration using those semantics.

          D3DVERTEXELEMENT9 declaration[]={{0,0,D3DDECLTYPE_FLOAT3,
                          D3DDECLMETHOD_DEFAULT,D3DDECLUSAGE_POSITION,
                          0},{0,12,D3DDECLTYPE_D3DCOLOR,
                          D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_COLOR,
                          0},{0,16,D3DDECLTYPE_FLOAT2,
                          D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD,
                        0},D3DDECL_END()};

           //Set this declaration
           g_pd3dDevice->CreateVertexDeclaration(declaration,
                                              &g_pVertexDeclaration);

            HRESULT hr;
            LPD3DXBUFFER pCode;       //The buffer the shader's code is
                                      //loaded into.
            DWORD dwShaderFlags = 0;
            LPD3DXBUFFER pBufferErrors = NULL;

            //Assemble the vertex shader from the file.
            // "main" - specifies the shaders entry point
            // "vs_1_1" - specifies the vertex shader level to try
            //             and compile for
            //Any uniform values in the shader source will have
            //location entries made in the g_pVertexConstantTableVS
            //object
            hr = D3DXCompileShaderFromFile("normal_map_vert.hlsl",
                                 NULL,NULL,"main","vs_1_1",
                                 dwShaderFlags, &pCode,
                                 &pBufferErrors,
                                 &g_pVertexConstantTableVS );

            // Create the vertex shader
            g_pd3dDevice->CreateVertexShader(
                               (DWORD*)pCode->GetBufferPointer(),
                               &g_pVertexShader);
            pCode->Release();

            //Assemble the pixel shader from the file.
            // "main" - specifies the shaders entry point
            // "ps_2_0" - specifies the pixel shader level to try
            //             and compile for
            //Any uniform values in the shader source will have
            //location entries made in the g_pPixelConstantTableVS
            //object
            hr = D3DXCompileShaderFromFile("normal_map_pixel.hlsl",
```

```
                              NULL,NULL,"main","ps_2_0",
                              dwShaderFlags, &pCode,
                              &pBufferErrors,
                              &g_pPixelConstantTableVS );

            // Create the pixel shader
            g_pd3dDevice->CreatePixelShader(
                              (DWORD*)pCode->GetBufferPointer(),
                              &g_pPixelShader);
            pCode->Release();

            break;
      }
      case CG:
      {
            //Choose a vertex & pixel profile (in this case OpenGL
            //assembly profile)
            g_CGvertexprofile = CG_PROFILE_ARBVP1;
            g_CGpixelprofile = CG_PROFILE_ARBFP1;

            //Create a Cg Context
            g_CGcontext = cgCreateContext();

            //Create the vertex shader program using the created
            //context. We read in the string source from
            //"normal_map_vert.cg". The translator will try and
            //compile this to work with the specified profile.
            g_CGvertexprogram = cgCreateProgramFromFile(g_CGcontext,
                              CG_SOURCE, "normal_map_vert.cg",
                              g_CGvertexprofile, NULL, NULL);

            //Create the pixel shader program using the created
            //context. We read in the string source from
            //"normal_map_pixel.cg". The translator will try and
            //compile this to work with the specified profile.
            g_CGpixelprogram = cgCreateProgramFromFile(g_CGcontext,
                              CG_SOURCE, "normal_map_pixel.cg",
                              g_CGpixelprofile, NULL, NULL);

            //Load the programs using Cg's interface
            cgGLLoadProgram(g_CGvertexprogram);
            cgGLLoadProgram(g_CGpixelprogram);

            // Get handles to the uniform variables we will set later
            g_CGparam_LightColor = cgGetNamedParameter(g_CGprogram,
                              "fLightDiffuseColor");
            g_CGparam_LightPosition = cgGetNamedParameter(g_CGprogram,
                              "vLightPosition");

            g_CGparam_ModelViewMatrix = cgGetNamedParameter(
                                g_CGprogram, "modelViewProjMatrix");

            break;
      }
      case GLSL:
      {
         //Create the vertex shader object
```

```
        g_vertexShader=glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);

        //Create the pixel shader object
        g_pixelShader=glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);

        //User defined function that reads in shader source strings
        //from “normal_map_vert.glsl” text file
        unsigned char *vertexShaderSource = readShaderSource(
                                            "normal_map_vert.glsl");

        //User defined function that reads in shader source strings
        //from “normal_map_pixel.glsl” text file
        unsigned char *pixelShaderSource = readShaderSource(
                                            "normal_map_pixel.glsl");


        //Covert to string array parameter for next function call
        vertexShaderStrings[0] = (char*)vertexShaderAssembly;
        pixelShaderStrings[0] = (char*)pixelShaderAssembly;

        //Send the read in vertex shader source to vertex program
        //object.
        glShaderSourceARB(g_vertexShader,1,vertexShaderStrings,
                          NULL);

        //Send the read in pixel shader source to fragment program
        //object.
        glShaderSourceARB(g_pixelShader,1,pixelShaderStrings,
                          NULL);

        //Now that the program has source code, compile the shaders
        glCompileShaderARB(g_vertexShader);
        glCompileShaderARB(g_pixelShader);

        //Create a GLSL program object and attach the compiled shaders
        g_programObj = glCreateProgramObjectARB();
        glAttachObjectARB(g_programObj, g_vertexShader);
        glAttachObjectARB(g_programObj, g_pixelShader);

        //Link the GLSL program object containing the compiled GLSL
        //vertex & pixel shader.
        glLinkProgramARB(g_programObj);

        //Get handles to all the uniform variables we will set later
        g_location_LightColor = glGetUniformLocationARB(g_programObj,
                                                 " fLightDiffuseColor");
        break;
      }
}
```

Per Frame (Performed once per frame):

```
//Each redraw of the window we activate the shaders and pass in any
//dynamic variable values they may use.
switch(Language)
{
      case HLSL:
      {
                                    .
                                    .
                                    .
                                    .
                                    .
                           Update any DirectX states
                                    .
                                    .
                                    .
                                    .
                                    .

            //Get the current Model View Projection Matrix
            D3DXMATRIX worldViewProjection = g_matWorld * g_matView *
            g_matProj;

            //Pass it into the constants table
            g_pVertexConstantTableVS->SetMatrix( g_pd3dDevice,
            "worldViewProj", &worldViewProjection );

            //Set the light position in the vertex shader
            float* lightPos={vLightPos.x, vLightPos.y, vLightPos.z};
            g_pVertexConstantTableVS->SetFloat(g_pd3dDevice,
                                        "vLightPosition",lightPos,3);

            //Set the diffuse light color in the pixel shader
            float* lightColor={1.0,1.0,1.0};
            g_pPixelConstantTableVS->SetFloat(g_pd3dDevice,
                                        "fLightDiffuseColor",
                                        lightColor,3);

            //Set vertex declaration and vertex shader
            //active
            g_pd3dDevice->SetVertexDeclaration(g_pVertexDeclaration);
            g_pd3dDevice->SetVertexShader(g_pVertexShader);

            //Make pixel shader active
            g_pd3dDevice->SetPixelShader(g_pPixelShader);

                                    .
                                    .
                                    .
                                    .
                                    .
                        Set Textures & Draw Sphere
                                    .
                                    .
```

```
                                    .
                                    .
        //Deactivate current vertex & pixel shader
        g_pd3dDevice->SetVertexShader(NULL);
        g_pd3dDevice->SetPixelShader(NULL);
        break;
    }
    case CG:
    {
                                    .
                                    .
                                    .
                                    .
                    Update any DirectX/OpenGL states
                                    .
                                    .
                                    .
                                    .
        //Enable vertex Profile used by the normal mapping vertex
        //and pixel shaders
        CgGL.cgGLEnableProfile(g_CGvertexprofile);
        CgGL.cgGLEnableProfile(g_CGpixelprofile);

        //Activate the normal mapping vertex & pixel shaders
        CgGL.cgGLBindProgram(g_CGvertexprogram);
        CgGL.cgGLBindProgram(g_CGpixelprogram);

        // Set the "modelViewProjMatrix" parameter in the vertex
        //shader to the current concatenated
        // modelview and projection matrix
        CgGL.cgGLSetStateMatrixParameter(g_CGparam_ModelViewMatrix,
                            CgGL.CG_GL_MODELVIEW_PROJECTION_MATRIX,
                            CgGL.CG_GL_MATRIX_IDENTITY);

        //Set the light properties
        CgGL.cgGLSetParameter3f(g_CGparam_LightPosition,
                        vLightPos.x, vLightPos.y, vLightPos.z);
        CgGL.cgGLSetParameter3f(g_CGparam_LightColor, 1.0f, 1.0f,
                                1.0f);
                                    .
                                    .
                                    .
                                    .
                    Set Textures & Draw Sphere
                                    .
                                    .
                                    .
                                    .
                                    .
        //Disable the vertex shader profiles
        CgGL.cgGLDisableProfile(g_CGvertexprofile);
        CgGL.cgGLDisableProfile(g_CGpixelprofile);
        break;

    }
    case GLSL:
    {
```

```
                                        .
                                        .
                                        .
                                        .
                            Update any OpenGL states
                                        .
                                        .
                                        .
                                        .
                                        .
            //Enable the GLSL shader program containing the compiled
            //normal mapping vertex & pixel shader
            glUseProgramObjectARB(g_programObj);

            glUniform3fARB(g_location_LightColor,1.0f,1.0f,1.0f);
                                        .
                                        .
                                        .
                                        .
                                        .
                            Set Light, Textures & Draw Sphere
                                        .
                                        .
                                        .
                                        .
            // Disable the GLSL shader objects
            glUseProgramObjectARB(NULL);
            break;
      }
}
```

HLSL Shader Source:

```
/*********************************************************************
Pixel-Bound Test: HLSL Vertex Shader
Source file: “normal_map_vert.hlsl”
*********************************************************************/

float4x4 modelViewProjMatrix;
float3 vLightPosition;

/* Incoming vertex data*/
struct VS_INPUT {
      float4 position  : POSITION;     //The position of the current
                                       //vertex.
      float2 texCoords : TEXCOORD0;    //Diffuse texture coordinates
      float3 vNormal   : TEXCOORD1;    //Vertex normal
      float3 vTangent  : TEXCOORD2;    //Vertex tangent
      float3 vBinormal : TEXCOORD3;    //Vertex binormal
};

/* Data passed out from pixel shader */
struct VS_OUT {
      float4 positionOUT : POSITION;   //The transformed vertexfloat2
      texCoordsOUT       : TEXCOORD0;  //Send tex. coords to pixel the
                                       //shader
      float3 vLightVector: TEXCOORD1;  //Send the transformed light
                                       //vector to the pixel shader
};


VS_OUT main(appdata IN)
{
      VS_OUT OUT;
      // Calculate the light vector
      OUT.vLightVector = IN.vLightPosition – IN.position.xyz;

      //Transform the light vector from object space into tangent
      //space
      float3x3 TBNMatrix = float3x3(IN.vTangent, IN.vBinormal,
                                    IN.vNormal);

      OUT.vLightVector.xyz = mul(TBNMatrix,OUT.vLightVector);

      // Transform the current vertex from object space to clip space,
      OUT.positionOUT = mul(modelViewProjMatrix,position);

      // Send the texture map coords to the fragment shader
      OUT.texCoordsOUT = IN.texCoords;

      return OUT;
}
```

```
/*********************************************************************
Pixel-Bound Test: HLSL Pixel Shader
Source file: “normal_map_pixel.hlsl”
*********************************************************************/

float3 fLightDiffuseColor;
sampler baseTexture;
sampler normalTexture;

/* Incoming pixel data*/
struct PS_IN {
      float4 colorIN : COLOR0;
      float2 texCoords : TEXCOORD0;    //The texture map's texcoords
      float3 vLightVector : TEXCOORD1; //The transformed light vector
                                       //(in tangent space)
};

/* Data passed out from pixel shader */
struct PS_OUT {
      float4 colorOUT : COLOR0; //The final color of the current pixel
};


PS_OUT main(appdata PS_IN)
{
      PS_OUT OUT;

      //We must normalize the light vector as it's linearly
      //interpolated across the surface and its length may change
      IN.vLightVector = normalize(IN.vLightVector);

      //Since the normals in the normal map are in the (color) range
      //[0, 1] we need to uncompress them to real normal vector
      //directions in the range [-1,1].
      float3 vNormal = 2.0f * (tex2D(normalTexture, IN.texCoords).rgb
                        – 0.5f);

      //Calculate the diffuse component and store it as the final
      //color in colorOUT
      //The diffuse component is defined as: I = Dl * Dm * clamp(L•N,
      //0, 1). saturate() works like clamp().
      OUT.colorOUT.rgb = fLightDiffuseColor * tex2D(baseTexture,
           IN.texCoords).rgb * saturate(dot(IN.vLightVector, vNormal));

      return OUT;
}
```

Cg Shader Source:

```
/*********************************************************************
Pixel-Bound Test: Cg Vertex Shader
Source file: “normal_map_vert.cg”
*********************************************************************/

/* Incoming vertex data*/
struct appdata {
      float4 position  : POSITION;     //The position of the current
                                       //vertex.
      float2 texCoords : TEXCOORD0;    //Diffuse texture coordinates
      float3 vNormal   : TEXCOORD1;    //Vertex normal
      float3 vTangent  : TEXCOORD2;    //Vertex tangent
      float3 vBinormal : TEXCOORD3;    //Vertex binormal
};

/* Data passed out from pixel shader */
struct vertexOutput {
      float4 positionOUT : POSITION;   //The transformed vertexfloat2
      texCoordsOUT       : TEXCOORD0;  //Send tex. coords to pixel the
                                       //shader
      float3 vLightVector: TEXCOORD1;  //Send the transformed light
                                       //vector to the pixel shader
};


vertexOutput main(appdata IN,
                  const uniform float4x4 modelViewProjMatrix,
                  const uniform float3 vLightPosition)
{
      vertexOutput OUT;
      // Calculate the light vector
      OUT.vLightVector = IN.vLightPosition – IN.position.xyz;

      //Transform the light vector from object space into tangent
      //space
      float3x3 TBNMatrix = float3x3(IN.vTangent, IN.vBinormal,
                                    IN.vNormal);

      OUT.vLightVector.xyz = mul(TBNMatrix, OUT.vLightVector);

      // Transform the current vertex from object space to clip space,
      OUT.positionOUT = mul(modelViewProjMatrix, position);

      // Send the texture map coords to the fragment shader
      OUT.texCoordsOUT = IN.texCoords;

      return OUT;
}
```

```
/*********************************************************************
Pixel-Bound Test: Cg Pixel Shader
Source file: “normal_map_pixel.cg”
*********************************************************************/

/* Incoming pixel data*/
struct appdata {
      float4 colorIN : COLOR0;
      float2 texCoords : TEXCOORD0;    //The texture map's texcoords
      float3 vLightVector : TEXCOORD1; //The transformed light vector
                                       //(in tangent space)
};

/* Data passed out from pixel shader */
struct pixelOutput {
      float4 colorOUT : COLOR0; //The final color of the current pixel
};


pixelOutput main(appdata IN, uniform sampler2D baseTexture : TEXUNIT0,
                     uniform sampler2D normalTexture : TEXUNIT1,
                     uniform float3 fLightDiffuseColor)
{
      pixelOutput OUT;

      //We must normalize the light vector as it's linearly
      //interpolated across the surface and its length may change
      IN.vLightVector = normalize(IN.vLightVector);

      //Since the normals in the normal map are in the (color) range
      //[0, 1] we need to uncompress them to real normal vector
      //directions in the range [-1,1].
      float3 vNormal = 2.0f * (tex2D(normalTexture, IN.texCoords).rgb
                        – 0.5f);

      //Calculate the diffuse component and store it as the final
      //color in colorOUT
      //The diffuse component is defined as: I = Dl * Dm * clamp(L•N,
      //0, 1). saturate() works like clamp().
      OUT.colorOUT.rgb = fLightDiffuseColor * tex2D(baseTexture,
      IN.texCoords).rgb * saturate(dot(IN.vLightVector, vNormal));

      return OUT;
}
```

GLSL Shader Source:

```
/*********************************************************************
Pixel-Bound Test: GLSL Vertex Shader
Source file: “normal_map_vert.glsl”
*********************************************************************/

//Varying datatype allows sharing of this value between vertex and
//pixel shader in same pipeline
varying vec3 vLightVector;       //Tangent-space light position

void main()
{
      //Calculate the light vector
      vLightVector = gl_LightSource[0].position.xyz - gl_Vertex.xyz;

      //Transform the light vector from object space into tangent
      //space. Our tangent, binormal, and normal vectors are passed in
      //through the APIs multitexture texture coordinate calls
      mat3 TBNMatrix=float3x3(gl_MultiTexCoord2.xyz,
                              gl_MultiTexCoord3.xyz,
                              gl_MultiTexCoord1.xyz);

      vLightVector.xyz = vLightVector*TBNMatrix;

      Pass through our diffuse texture coordinates to the pixel shader
      gl_TexCoord[0] = gl_MultiTexCoord0;

      // Transform the current vertex from object space to clip space,
      gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}



/*********************************************************************
Pixel-Bound Test: GLSL Pixel Shader
Source file: “normal_map_pixel.glsl”
*********************************************************************/

//Uniforms set by user application
uniform vec3 fLightDiffuseColor;    //Diffuse color of light
uniform sampler2D baseTexture;      //Diffuse map texture unit handle
uniform sampler2D normalTexture;    //Normal map texture unit handle

//Set by vertex shader
varying vec3 vLightVector;          //Tangent-space light position

void main()
{
      // We must remember to normalize the light
      vLightVector = normalize(vLightVector);

      //Since the normals in the normal map are in the color range
      //[0, 1] we need to uncompress them to real normal vector
      //directions in the range [-1,1].
```

```
      vec3 vNormal = 2.0f * (texture2D(normalTexture,
                              gl_TexCoord[0].xy).rgb - 0.5f);

      //Calculate the diffuse component and store it as the final
      //fragment color.
      // The diffuse component is defined as: I = Dl * Dm * clamp(L•N,
      //0, 1)
      vec3 result = fLightDiffuseColor *
                    texture2D(baseTexture, gl_TexCoord[0].xy).rgb *
                    clamp(dot(vLightVector, Normal),0.0,1.0);

      gl_FragColor = vec4(result,1.0); //Must include alpha value
}
```