Introduction to GPU Programming with GLSL

Ricardo Marroquim
Istituto di Scienza e Tecnologie dell'Informazione
CNR
Pisa, Italy
ricardo.marroquim@isti.cnr.it

André Maximo
Laboratório de Computação Gráfica
COPPE – UFRJ
Rio de Janeiro, Brazil
andmax@cos.ufrj.br

Figure 1. Teapot textured with the SIBGRAPI 2009 logo using the fixed functionality (left), and with a combination of shaders (right).

***Abstract*—One of the challenging advents in Computer Science in recent years was the fast evolution of parallel processors, specially the *GPU* – graphics processing unit. GPUs today play a major role in many computational environments, most notably those regarding real-time graphics applications, such as games.**

The digital game industry is one of the main driving forces behind GPUs, it persistently elevates the state-of-art in Computer Graphics, pushing outstanding realistic scenes to interactive levels. The evolution of photo realistic scenes consequently demands better graphics cards from the hardware industry. Over the last decade, the hardware has not only become a hundred times more powerful, but has also become increasingly customizable allowing programmers to alter some of previously fixed functionalities.

This tutorial is an introduction to GPU programming using the OpenGL Shading Language – *GLSL*. It comprises an overview of graphics concepts and a walk-through the graphics card rendering pipeline. A thorough understanding of the graphics pipeline is extremely important when designing a program in GPU, known as a *shader*. Throughout this tutorial, the exposition of the GLSL language and GPU programming details are followed closely by examples ranging from very simple to practical applications. It is aimed at an audience with no or little knowledge on the subject.

***Keywords*-GPU Programming; Graphics Hardware; GLSL.**

I. Introduction

The realism of interactive graphics has been leverage by the growth of the GPU's computational power throughout the years, turning a simple graphics card into a highly parallel machine inside a regular PC. The main motivation of this growth is given by the game community which increasingly demands realism and consequently graphics processing power. To add even more realism, graphics units have also shifted from being simple rendering black boxes to powerful programming units, allowing a variety of special effects within the graphics pipeline, such as the one shown in Figure 1.

One of the challenges of GPU programming is to learn how streaming architectures work. The true power of graphics processing comes from the fact that its primitives, e.g. vertices or pixels, are computationally independent, that is, the graphics data can be processed in parallel. Hence, a large number of primitives can be streamed through the graphics pipeline to achieve high performance.

Streaming programming is heavily based on the single instruction multiple data (SIMD) paradigm, where the same instruction is used to process different data. Within the GPU context, the data flows through the graphics pipeline while being processed by different programmable stages, called

shaders. Differently from traditional sequential programming, the GPU streaming model forces a fixed data flow through pipeline stages, i.e. a shader is restricted to deal with a specific input and output data type in a specific stage of the pipeline. From the many different rendering stages, only three of them is currently available in hardware to be programmable: the vertex shader, responsible to transform vertex primitives; the geometry shader, responsible to build the geometry from vertices; and the fragment shader, responsible to color fragments generated by geometric primitives. The graphics pipeline will be further explained in more details in Section II.

Despite the limitations, GPU programming is essential to implement special graphics effects not supported by the fixed pipeline. The power to replace a pipeline stage by a shader represents a major shift of implementation control and freedom to a computer graphics programmer. For real-time applications, such as games, the topic has become so important that it is no longer an extra feature but a necessity. Not surprisingly, the gaming industry represents the main driving force to constantly elevate the graphics hardware power and flexibility.

GPU programming goes beyond graphics and it is also exploited by completely different purposes, from genome sequence alignment to astrophysical simulations. This usage of the graphics card is known as General Purpose GPU (GPGPU) [1] and is playing such an important role that specific computational languages are being designed, such as *nVidia*'s [2] Compute Unified Device Architecture (CUDA) [3], which allows the non-graphical community to fully exploits the outstanding computational power of GPUs. Although this survey will not cover this area, some aspects of GPGPU are provided in Section VII.

The goal of this survey is to describe the creation of computer generated images from the graphics card side, that is, the graphics pipeline, the GPU streaming programming model and the three different shaders. The standard graphics library *OpenGL* [4] and its shading language *GLSL* [5] are used in order to reach a wider audience, since they are cross-platform and supported by different operating systems.

The remainder of this survey is organized as follows. In Section II, the graphics pipeline is summarized and its basic stages are explained. After that, Section III draws major aspects of the GPU's history, aiming at important features about its architecture. Section IV briefly describes how the GLSL language integrates with OpenGL and GLUT. The main part of this tutorial is in Sections V and VI, where GLSL is explained by analyzing six examples step-by-step. The GPGPU argument raised in the introduction is further analysed in Section VII. Finally, conclusions and insights on the future of shader programming wrap up this survey in Section VIII.

II. Graphics Pipeline

Before venturing into the topic of GPU programming itself, it is important to be familiar with the graphics pipeline. Its main function is rendering, that is, generating the next frame to be displayed. To accomplish this, geometric primitives are sent to the graphics pipeline, where they are processed and filled with pixels (process called *rasterization*) to compose the final image. Each stage of the pipeline is responsible for a part of this process, and programmable stages are nothing more than customizable parts that can perform different methods than those offered by the graphics API. Understanding the different parallel paradigms of the rendering pipeline is also crucial for the development of efficient GPU applications.

The rendering pipeline is responsible for transforming geometric primitives into a raster image. The primitives are usually polygons describing a model, such as a triangle mesh, while the raster image is a matrix of pixels. Figure 2 illustrates the rendering pipeline. First, the model's primitives, usually described as a set of triangles with connected points, are sent to be rendered. The vertices enter the pipeline and undergo a transformation which maps their coordinates from the world's reference to the camera's. When there are light sources in the scene, the color of each vertex is affected in this first stage by a simple illumination model. To accelerate the rest of the rendering process, clipping is performed to discard primitives that are outside the viewing frustum, i.e. primitives not visible from the current view point. The vertex processing stage is responsible for these transformation, clipping and lighting operations.

The transformed vertices are then assembled according to their connectivity. This basically means putting the primitives back together. Each primitive is then rasterized in order to compute which pixels of the screen it covers, where for each one a fragment is generated. Each fragment has interpolated attributes from its primitive's vertices, such as color and texture coordinates. A fragment does not yet define the final image pixel color, only in a last stage of the pipeline it is computed by composing the fragments which fall in its location. This composition may consist of picking the frontmost fragment and discarding others by means of a *depth buffer*, or blending the fragments using some criteria.

Note that the graphics pipeline, as any pipeline, is independent across stages, that is, vertices can be processed at the same time pixels are being processed. In this way, rendering performance (measured in frames per second) can be significantly improved, since while final pixels of the current frame are being handled, the vertex processors can already start processing the input vertices for the next frame. Besides being independent across stages, the graphics pipeline is also independent inside stages, different vertices can be transformed at the same time by different processors as they are computationally independent among themselves.

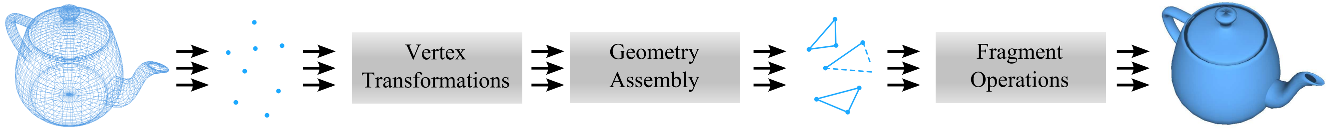

Figure 2. The basic stages of the rendering pipeline.

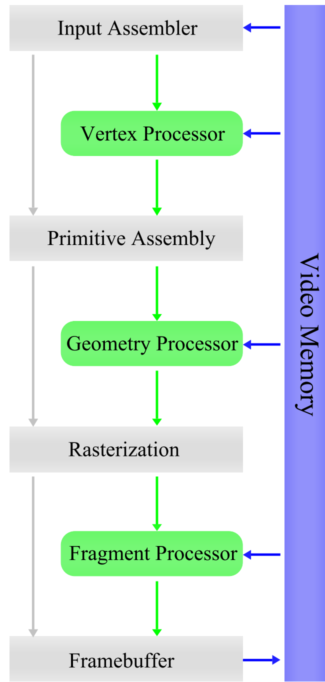

Figure 3. The data flow inside the graphics pipeline.

Figure 3 illustrates the three shader stages of the graphics pipeline, where they are represented by the *programmable stream* (arrows in green), in contrast to the *fixed built-in stream* (arrows in gray). The three *programmable stages* (green boxes) have read access to the video memory, while the *fixed stages* (gray boxes) access it for input and output.

The vertex shader is further detailed in Figure 4. This first stage treats only vertex primitives in a very strict input/output scheme: exactly one vertex enters and exactly one vertex exits. The processing is done in a completely independent way, no vertex has information about the others flowing through the pipeline, which allows for many of them to be processed in parallel.

In the next stage, depicted in Figure 5, the geometry shader processes primitives formed by the connected vertices, such as lines or triangles. At this stage all the vertices information belonging to the primitive may be made available. The geometry shader may receive as input different primitives than those it outputs, however, information such as the type of input and output primitives and maximum number of exiting vertices must be pre-defined by the user. Each input primitive may generate from zero to the defined maximum number of output vertices.

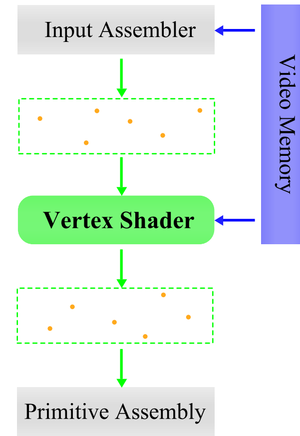

Figure 4. Pipeline Vertex Shader.

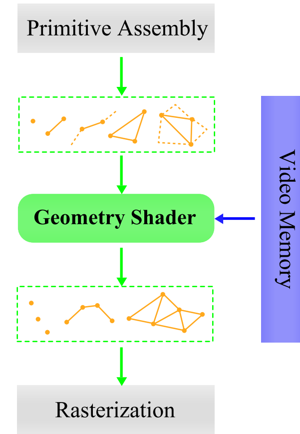

Figure 5. Pipeline Geometry Shader.

Finally, the fragment shader pipeline, illustrated in Figure 6, runs once for each fragment generated by each primitive. It has no information about which primitives it belongs to neither about other fragments. However it receives all the interpolated attributes from the primitive's vertices. The shader's main task is to output a color for each fragment, or discard it.

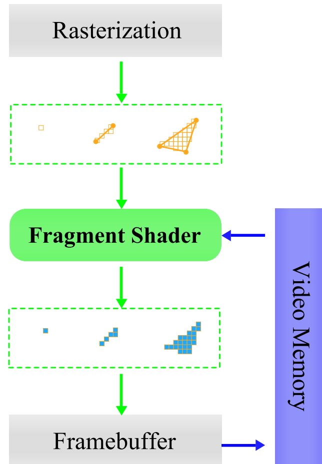

Figure 6. Pipeline Fragment Shader.

III. The Evolution of GPUs

One of the biggest evolution steps on GPU's history occurred in 1999, when nVidia designed a graphics hardware capable of performing vertex transformation and lighting operations, advertising their *GeForce 256* as "the world's first GPU". At first, many were sceptical about its usefulness, but it soon became the new paradigm in graphics hardware. Heavy graphics processing operations was previously done in software except for some expensive high-end graphics cards that could performed them in a dedicated chip; with the GeForce everything was integrated in a single-chip processor which allowed for a major price reduction. Followed by the *GeForce 2* and *ATI's Radeon 7500*, these cards framed the so called second generation GPUs allowing more powerful effects, such as multi-texturing.

Nonetheless, hardware had only been designed to improve the graphics performance, and the rendering process was still based on built-in fixed functionalities, i.e. hard-coded functions on graphics chips. The major drawback was that moving from software to hardware traded implementation freedom for speed, restricting the possible achievable effects. In 2001 this drawback started to be overcome with the first programmable graphics card: the *GeForce 3* introduced a programmable processor for any per-vertex computation. Small assembly codes could be uploaded to the graphics card replacing the fixed functionality. These were known as vertex shaders and framed the third GPU generation.

In the next two years, a programmable processor for any per-fragment computation was introduced. The new shader replaces fragment operations allowing better illumination effects and texture techniques. Note that even though per-fragment operations influences the pixel colors, the final pixel composition is still up to now a fixed functionality of GPUs. The fragment shader is also named pixel shader, but during the survey will use the former nomenclature.

The fourth generation of GPUs had not only introduced the fragment shader, but the maximum number of instructions per shader was also raised, and conditional branches were now allowed. The fast pace of GPU evolution is evidenced even more with the introduction of the *GeForce 6* and *Shader Model 3.0* in 2004. With this shader model also came along different high-level shading languages, such as: OpenGL's GLSL [5], Cg from nVidia [6] and Microsoft's HLSL [7]. Shader Model 3.0 allows for longer shaders, better flow control, and texture access in vertex shader, among other improvements.

In 2006 the *Shader Model 4.0*, also called *Unified Shader Model*, introduced a powerful GPU programming and architecture concept. First, a new programmable stage in the graphics pipeline was introduced: the geometry shader. This is, up until now, the last available programmable shader in hardware, which allows manipulation and creation of new graphics primitives, such as points, lines and triangles, directly inside the graphics card. Second, the three shader types were bounded to a single instruction set, enabling a new concept of a *Unified Shader Architecture*. The new GPU architecture, starting with the *GeForce 8* and *ATI's Radeon R600*, allows a more flexible usage of the graphics pipeline, where more processors can be dedicated to demanding stages of the pipeline. While the old architectures had a fixed number of processors and resources throughout the graphics pipeline, with the unified model the system can allocate resources to the computationally demanding shaders.

The graphics card is now considered a massively parallel co-processor of CPUs not only for graphics programs but for any general purpose application. The non-graphical community has been using GPUs since 2002 under the concept of GPGPU [1], however the new architecture allows the design of non-graphical languages, such as CUDA [3] and OpenCL [8], to fully exploit the highly parallel and flexible computation power of present GPUs. This concept will be further discussed in Section VII.

The history of GPUs points to a future where the graphics cards will be fully programmable, without restriction on shader programmability and resources usage. Last year two new types of shader were announced with the *Shader Model 5.0* to be implemented in graphics hardware this year (2009): the hull and domain shaders to better control the tessellation procedure. This survey will not stray to these new shader, focusing only on the current three available shaders in hardware. Beyond the evolution of GPUs, it seems that the future holds the end of sequential single-processor programming computation, in favor for massively parallel programming models. Another indication of this future path is the Cell Broadband Engine Architecture (CBEA) released by IBM, Sony and Toshiba, and the recent announcements of Intel's Larrabee architecture, where a major convergence of CPU and GPU pipelines and design strategies is promised.

IV. OpenGL Basics

This survey uses GLUT – OpenGL Utility Toolkit [9] – a simple platform-independent toolkit for OpenGL [4]. GLUT is used to create the OpenGL context, that is, it opens a graphics window for rendering, and handles the user's inputs. We have chosen GLUT for its popularity and simplicity, but among other options are QT [10] and wxWidgets [11].

Listing 1 shows a simple example of GLUT usage: after several basic initializations, the OpenGL window is created and the following callbacks are registered: `reshape` and `display` functions, called when the window is resized or its display contents needs to be updated; `keyboard` and `mouse` functions, called when one of such user's input occurs. At the end, the program enters the main event loop where it waits for events, such as window draw calls and keyboard hits.

```
#include <GL/glut.h>
// C Main function
int main( int argc, char** argv ) {
  // GLUT Initialization
  glutInit( &argc, argv );
  glutInitDisplayMode( GLUT_DOUBLE | GLUT_RGBA );
  glutInitWindowSize( 512, 512 );
  // Create OpenGL Window
  glutCreateWindow( "Simple Window" );
  init(); // non-GLUT initializations
  // Register callbacks
  glutReshapeFunc( reshape );
  glutDisplayFunc( display );
  glutKeyboardFunc( keyboard );
  glutMouseFunc( mouse );
  // Event Loop
  glutMainLoop();
  return 0;
}
/// The result is a window with 512x512 pixels
```

Listing 1. Simple GLUT Example.

GLUT is used to create a layer between graphics functions and the window management system, namely the OpenGL context. Once the context is created, OpenGL functions can be used for rendering. Moreover, OpenGL has a number of functions to manage shader codes in its native library. In despite of newer versions, this survey uses OpenGL version 2.1, as well as GLSL 1.2 and GLUT 3.7, which are sufficient to implement the introductory concepts.

The OpenGL functions to manage shader codes are illustrated in Listing 2. The `initShader` function in this listing is called among OpenGL initialization calls, e.g. inside the `init` function showed in Listing 1, where the goal is to initialize the shaders. In this example, only a vertex shader will be set in order to illustrate the shader management. Nevertheless, the functions shown here can be used for a geometry or fragment shader as well. The source code of a shader to be uploaded is simply a string and can be declared together with the application, as done in this example, or saved in a file to be read at run-time. The vertex shader code inside the `vsSource` string is the corresponding *Hello World* shader, and it will be explained in more details in Section V.

```
// OpenGL initialization calls for shaders
void initShader() {
  // Vertex Shader code source
  const GLchar* vsSource = {
  "#version 120\n"
  "void main(void) {\n"
  "     gl_FrontColor = gl_Color;\n"
  "     gl_Position = gl_ModelViewProjectionMatrix
                  * gl_Vertex;\n"
  "}"
  }
  // Create program and vertex shader objects
  programObject = glCreateProgram();
  vtxShader = glCreateShader( GL_VERTEX_SHADER );
  // Assign the vertex shader source code
  glShaderSource( vtxShader, 1, &vsSource, NULL );
  // Compile the vertex shader
  glCompileShader( vtxShader );
  // Attach vertex shader to the GPU program
  glAttachShader( programObject, vtxShader );
  // Create an executable to run on the GPU
  glLinkProgram( programObject );
  // Install vertex shader as part of the pipeline
  glUseProgram( programObject );
}
/// The result is a vertex shader acting as a
/// simplified version of the fixed functionality
```

Listing 2. OpenGL setup example for GLSL.

The `glCreateProgram` function creates a GPU program object to hold the shader objects. The `glCreateShader` function creates a shader object to maintain the source code string, while the `glShaderSource` function is responsible to assign a given source code to the created shader. The `glCompileShader` function compiles the shader object, while the `glAttachShader` function attaches it to a target program object. Note that, in the example shown in Listing 2, only a vertex shader is attached to the program object and, consequently, the rest of the pipeline remains running the fixed functionality.

Finally, the `glLinkProgram` function links the program creating the final executable to be processed on the GPU. To switch among program objects and the fixed functionality, the `glUseProgram` function is used. This function must be called before rendering, in order to change the pipeline execution accordingly.

The action of compiling and linking shaders do not raise errors as normally happens in applications. The error/success states must be queried using OpenGL functions such as `glGetProgramInfoLog`.

V. GPU Programming with GLSL

In this section, GPU programming and the OpenGL Shading Language – *GLSL* – are introduced. GLSL is a high-level language based on C/C++, and can be used to write shader codes. The three types of shaders discussed in this survey

are the vertex, geometry and fragment shaders. However, it will cover only the overall aspect of the language, for more detailed information refer to Bailey and Cunningham's newly released book on *Graphics Shaders* [12], as well as the OpenGL *Orange Book* [13] for GLSL references.

The strategy chosen here is to explain the language by showing GLSL code examples and, for each piece of code, detail types and functions among other aspects. The examples grow in difficulty ranging from very simple "hello world" examples to more sophisticated ones, such as phong shading and environment mapping.

A. Hello World

The first example is a GPU *Hello World* that illustrate the three available shaders, showing a simple loop and presenting different GLSL types. Listing 3 shows the vertex shader. The full set of operations performed by the fixed functionality includes a set of transformations, clipping and lighting; however, the goal here is to accomplish a simplified result where only the position transformation is realized. More specifically, each vertex position is transformed from the world's coordinate system to the camera's system. This transformation uses the model-view and projection matrices (refer to the OpenGL *Red Book* [14] for more information about transformation matrices).

```
#version 120
// Vertex Shader Main
void main(void) {
  // Pass vertex color to next stage
  gl_FrontColor = gl_Color;
  // Transform vertex position before passing it
  gl_Position = gl_ModelViewProjectionMatrix
    * gl_Vertex;
}
```

Listing 3. *Hello World* Vertex Shader.

The first line indicates the minimum GLSL version required for this shader, in this case version 1.2. From now on, this version is always used and the directive is omitted. Ignoring comments, the next line is a function declaration. In any shader, the main function is obligatory and serves as entry point. In the next two lines the vertex's color is passed to the next step and the position transformed to screen space. Note that the right side of the last line produces exactly the same result as calling the build-in function `ftransform()`, which shall be used during the rest of the text. Names starting with `gl_` represent GLSL standard names and have different meanings, explained later.

Next in the pipeline is the geometry shader, where each geometric primitive is processed and one or more primitives is outputted, which may be of a different type than the one entered. For example, a geometry shader to decompress primitives could be defined over points and produce triangles, where for each point several triangles can be generated. As stated previously, the geometry type of input and output, as well as the maximum number of vertices on the output, have to be defined by the OpenGL application before linking the shader program and after compiling it.

Listing 4 shows the geometry shader for our *Hello World* example. The goal here is to simply pass the relevant information forward, that is, read the input values from the vertices and write them to the output primitives.

```
#extension GL_EXT_geometry_shader4: enable
// Geometry Shader Main
void main(void) {
  // Iterates over all vertices in the input
     primitive
  for (int i = 0; i < gl_VerticesIn; ++i) {
    // Pass color and position to next stage
    gl_FrontColor = gl_FrontColorIn[i];
    gl_Position = gl_PositionIn[i];
    // Done with this vertex
    EmitVertex();
  }
  // Done with the input primitive
  EndPrimitive();
}
```

Listing 4. *Hello World* Geometry Shader.

The first line enables the geometry shader extension as defined by the Shader Model 4.0. This directive is always required when using a geometry shader and it is omitted for the next examples. The main function body contains a loop iterating over the number of vertices in the input geometry. This is a pre-defined constant by GLSL and it has the same value for all the geometric primitives, depending on the input geometry type. Each loop iteration receives the color and position of the corresponding input vertex and assigns them to the color and position of the output vertex. The `EmitVertex` function finishes the output vertex, while the `EmitPrimitive` function finishes the primitive. In the geometry shader, only one primitive type can be received as input with a constant number of vertices.

Listing 5 shows the fragment shader, the last programmable step. The only action here is the assignment of a color to the fragment. The input color of each fragment is computed by the rasterization stage, where the primitives produced by the geometry shader are filled with fragments. For each fragment all vertices' attributes are interpolated, such as the color.

```
// Fragment Shader Main
void main(void) {
  // Pass fragment color
  gl_FragColor = gl_Color;
}
```

Listing 5. *Hello World* Fragment Shader.

In this last shader, the `gl_Color` name relate to a different built-in type than the one showed on the vertex shader (see Listing 3). A built-in type is a graphics functionality, made available by OpenGL or by the graphics hardware, to provide access to pre-defined values inside or outside the rendering pipeline. In the vertex shader case, `gl_Color` was assigned outside the pipeline as a vertex attribute,

while in the fragment shader case it was defined inside the pipeline as an interpolated value. The GLSL language has five different built-in types:

- *built-in constants* – variables storing hardware-specific constant values, such as maximum number of lights. These values can be accessed by any shader and may change depending on the graphics card.
- *built-in uniforms* – values passed from the OpenGL application to one or more shaders. These values are OpenGL states, such as the projection matrix, and must be set before rendering. Inside a draw call the uniforms remain fixed.
- *built-in attributes* – values representing an attribute of a vertex, such as color. Like uniforms, the attributes are passed from the OpenGL application to (and only to) the vertex shader. However, unlike uniforms, the attributes may change for each vertex.
- *built-in varying variables* – variables used to convey information throughout the rendering pipeline. For instance, the vertex color can be passed from the vertex shader to the geometry shader and then to the fragment shader. Note that, even though the vertex color enters the pipeline as a vertex attribute, it flows through the pipeline as output and input varying variables.
- *built-in functions* – basic and advanced functions for different types of operations. The basic functions, such as *sin* and *cos*, can be used in any shader, while the advanced functions, such as the geometry shaders's function to emit a primitive, may be only used in a specific shader.

Listing 6 illustrates one example of each built-in type described. The data types are similar to the C language, e.g. `int` and `float`, with some additions explained later.

```
const int        gl_MaxLights;
uniform mat4     gl_ProjectionMatrix;
attribute vec4   gl_Color;
varying vec4     gl_FrontColor;
genType          sin( genType );
```

Listing 6. Built-in types examples.

The *genType* above may represent one of the following types: float, vec2, vec3 or vec4. The *vec* data type is a special GLSL feature that allows easy manipulation of vectors. It also allows the processor to optimize vector operations by performing them in a parallel element-wise fashion.

All the built-in types are represented on the previous three shaders of this first example. For instance, the `gl_Color` name from the vertex shader is a built-in attribute, while the `gl_Color` from the fragment shader is a built-in input varying variable. Other examples include a built-in constant in the geometry shader, `gl_VerticesIn`, a built-in uniform in the vertex shader, `gl_ModelViewProjectionMatrix`, and two built-in functions within the geometry shader, `EmitVertex` and `EmitPrimitive`.

Figure 7 shows the difference between the fixed functionality and the *Hello World* shader program. The normal pipeline does lighting computation, which enhances the details of the teapot model and yields a 3D appearance. On the other hand, the three presented shaders of this first example aim to simplify the fixed functionality and simply carries on the blue color of the vertices, losing, for instance, the lighting effect and the sensation of depth.

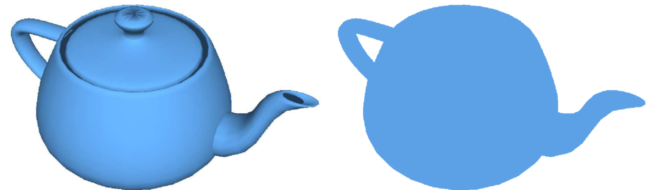

Figure 7. Difference between the fixed functionality and the *Hello World* shaders. The 3D appearance is lost due to lack of lighting computation in the vertex shader.

B. Cartoon Effect

The second example is a GPU *Cartoon Effect* that illustrate the use of normals and lighting, and mimics a cartoonist drawing effect. This effect is achieved by rendering the model with an extremely reduced color palette (4 shades of blue in this example). The color is chosen depending on the intensity of direct lighting arriving at the surface position. Listing 7 shows the vertex shader, the first step towards achieving this *Cartoon Effect*.

```
// Output vertex normal to fragment shader
varying out vec3 normal;
void main(void) {
  // Compute normal per-vertex
  normal = normalize(gl_NormalMatrix * gl_Normal);
  gl_FrontColor = gl_Color;
  // Transform position using built-in function
  gl_Position = ftransform();
}
```

Listing 7. *Cartoon Effect* Vertex Shader.

In the vertex shader, two new lines are added from the previous example. The first is an user-defined output varying variable called *normal*, responsible to carry on the per-vertex normal vector to the fragment shader. The other new line assigns a value to this varying by multiplying the vertex normal by its corresponding matrix. The normal matrix stores transformations to normal vectors as the model-view matrix stores about vertex positions. The last line computes the vertex position using the built-in function `ftransform()`, which does the same computation as the last line of the Hello World vertex shader (see Listing 3). More information about the normal matrix and built-in functions can be found on the RedBook[14].

The next and last step of the *Cartoon Effect* is the more involving fragment shader, shown in Listing 8. It uses

the normal from the vertex shader and a built-in uniform defining light position.

```
// Input vertex normal from vertex shader
varying in vec3 normal;
void main(void) {
  // Compute light direction
  vec3 ld = normalize( vec3(
    gl_LightSource[0].position) );
  // Compute light intensity to the surface
  float ity = dot( ld, normal );
  // Weight the final color in four cases,
  // depending on the light intensity
  vec4 fc;
  if (ity > 0.95) fc = 1.00 * gl_Color;
  else if (ity > 0.50) fc = 0.50 * gl_Color;
  else if (ity > 0.25) fc = 0.25 * gl_Color;
  else fc = 0.10 * gl_Color;
  // Output the final color
  gl_FragColor = fc;
}
```

Listing 8. *Cartoon Effect* Fragment Shader.

The first line indicates that the normal is received as an user-defined input varying variable, corresponding to the ouput varying from the vertex shader. Remember that varying variables arriving in fragment shaders have been previously interpolated from the vertices. In contrast, the light source position is directly fetched from the built-in uniform `gl_LightSource[0].position`, and used to compute the light direction. An intensity value is defined by the dot product between the light direction and the normal vector, i.e. by the angle between these two vectors. The intensity value is distributed in four possible ranges, where for each one, a different shade of the original color is used as the fragment final color. Figure 8 illustrates the final result.

Figure 8. Teapot rendered using the *Cartoon Effect*.

C. Simple Texture Mapping

The three shaders provide access to texture memory, even though earlier versions of programmable graphics card did not allow texture fetching in the vertex shader, and did not possess geometry shaders at all. The texture coordinates can be passed per vertex and interpolated to be accessed per fragment, however, any shader is free to access any position in any of the available textures. A texture is made available by setting it as an uniform variable and then calling one of the straightforward *sampler2D* functions to access it (see the GLSL reference [5] for further details on these functions).

Nevertheless, there are a few considerations that should be taken into account when designing a shader. First, the GPU is optimized for accessing texture in a more or less sequential manner, thus accessing cached memory is extremely fast compared to random texture fetches, but usually the GPU cache memory is substantially small. In this manner, linear algebra operations, such as adding two large vectors, run extremely fast.

Another important point is that, since the GPU is optimized for arithmetic intense applications, many times it is worthwhile recomputing than storing a value in a texture for latter access. The texture fetch latency is to some extend taken care of by the GPU: it switches between fragment operations when a fetch command is issue, i.e. it may start working on the next fragment while fetching the information for the current fragment. Therefore, as long as there is enough arithmetic operations to hide the fetch latency, the application will not be limited by the memory latency.

When texture coordinates are passed per vertex during the API draw call, they can be transformed within the vertex shader and retrieved after interpolation by the fragment shader. Texture coordinates are automatically made available without the need to define new varying variables. The following shaders exemplify this operation.

```
void main(void) {
  // Pass texture coordinate to next stage
  gl_TexCoord[0] = gl_TextureMatrix[0]
    * gl_MultiTexCoord0;
  // Pass color and transformed position
  gl_FrontColor = gl_Color;
  gl_Position = ftransform();
}
```

Listing 9. *Simple Texture* Vertex Shader.

The only difference to the vertex shader on the previous example is the replacement of the normal by a texture coordinate using a built-in varying variable `gl_TexCoord`. The computation is similar, the input texture coordinates are transformed by the texture matrix and stored in a varying variable. The limit on the number of input texture coordinates `gl_MultiTexCoordN` per vertex is imposed by the graphics hardware.

```
// User-defined uniform to access texture
uniform sampler2D texture;
void main(void) {
  // Read a texture element from a texture
  vec4 texel = texture2D( texture,
    gl_TexCoord[0].st );
  // Output the texture element as color
  gl_FragColor = texel;
}
```

Listing 10. *Simple Texture* Fragment Shader.

Within the fragment shader the texture defined as an uniform variable within the OpenGL API is accessed, and

Figure 9. Applying texture to a model within the fragment shader.

the fetched value used as the final color. The built-in function `texture2D` is used to access the texture, while the `gl_TexCoord[0].st` is a 2D vector containing the input texture coordinates. The `.st` field returns the first two vector components of the 4D vector gl_TexCoord[0], whereas `.stuv` returns the full vector. Any vector can be accessed by using the following components: `.xywz`, `.rgba` or `.stuv`. These components depend only on the semantic intended to the vector, for example, a 3D point in space can use `.xyz` coordinates while the same `vec3` can be used as a three-channel color `.rgb`. Additionally, these components can be swizzled when accessed by changing its order, such as the *blue-green-red* channels of a color `.bgr`.

The result of applying a texture inside the fragment shader is illustrated in Figure 9.

D. Phong Shading

One nice example that helps to put together the basics of GPU programming is *Phong Shading*. OpenGL performs built-in Gouraud shading when the `GL_SMOOTH` flag is set, where illumination is calculated per vertex, and the vertex colors are interpolated to the fragments. In this model, normal variations inside the triangle are not really well accounted for, and sometimes the border between triangles are very evident giving an unpleasant rendering result. A better way to do this is to interpolate the normals inside the triangle, and then compute the illumination per fragment. Surely it is more computationally involving, but the trade off from speed to quality is usually well rewarding.

From the last examples, it was shown that it is possible to customize other attributes to be passed from the vertex to the fragment shader as *varying variables*. To get our Phong Shading working we need to pass the vertex normals as varying variables to the fragment shader, as well as transfer the illumination computation from vertex to fragment shader.

The vertex shader below only transforms the normal and vertex to camera coordinates. Note that the `vert` varying variable is only transformed by the modelview as we will need it to compute the direction from the light source:

```
// Output vertex normal and position
varying out vec3 normal, vert;
void main(void) {
  // Store normal per-vertex to fragment shader
  normal = normalize( gl_NormalMatrix
    * gl_Normal );
  // Compute vertex position in model-view space
  // to be used in the fragment shader
  vert = vec3( gl_ModelViewMatrix * gl_Vertex );
  // Pass color
  gl_FrontColor = gl_Color;
  // Pass transformed position
  gl_Position = ftransform();
}
```

Listing 11. Phong Vertex Shader

Since the normals were passed as varying variables, they are also interpolated and accessible in the fragment shader. The code below (Listing 12) performs illumination within the fragment shader:

```
// Additional input from vertex shader:
// vertex normal and position
varying in vec3 normal, vert;
void main(void) {
  // Compute light and eye direction
  vec3 lp = gl_LightSource[0].position.xyz;
  vec3 ld = normalize( lp - vert );
  vec3 ed = normalize( - vert );
  // Compute reflection vector based on
  // light direction and normal
  vec3 r = normalize( -reflect( ld, normal ) );
  // Compute light parameters per fragment
  vec4 la = gl_FrontLightProduct[0].ambient;
  vec4 lf = gl_FrontLightProduct[0].diffuse
    * max( dot(normal, ld), 0.0 );
  vec4 ls = gl_FrontLightProduct[0].specular
    * pow( max( dot(r, ed), 0.0 ),
           gl_FrontMaterial.shininess );
  // Use light parameters to compute final color
  gl_FragColor = gl_FrontLightModelProduct.
    sceneColor + la + lf + ls;
}
```

Listing 12. Phong Fragment Shader

This follows closely how OpenGL computes illumination per vertex (for more details refer to the Red Book [14]), but here we are performing per pixel illumination. An example of the quality improvement can be observed in Figure 10. Figure 11 illustrates how the data flow for this example is integrated within the rendering pipeline.

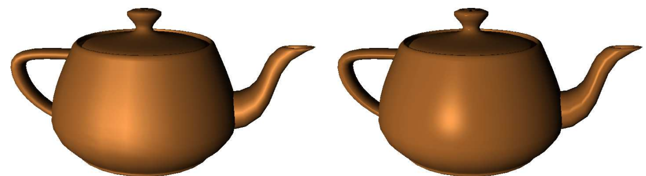

Figure 10. Gouraud shading rendered with *GL_SMOOTH* (left) and Phong shading rendered with the vertex and fragment shaders (right).

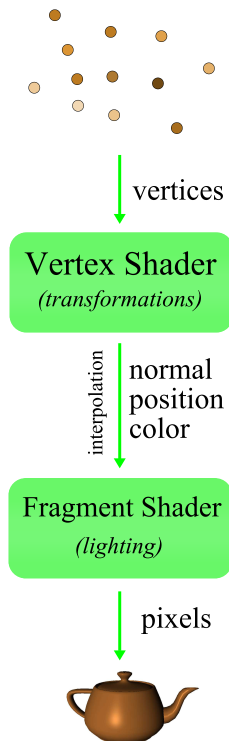

Figure 11. The data flux of the Phong shaders. Note how the lighting operation was postponed from the vertex shader to the fragment shader.

E. Environment Map

An useful texture application that is extremely simple to implement with shaders is the *environment mapping*. The goal is to simulate the reflection of an environment onto the object's surface. The idea behind this technique is to reverse apply a texture, that is, instead of directly place a texture to a surface, the reflection of the light is used to map it. One way to do this is to map a common image into a sphere, giving the illusion it was taken with a fish eye lens. Fortunately, most image editors have simple filters to perform this operation.

In this example, for each vertex we will compute the texture coordinates for the environment map, much like OpenGL would perform using sphere textures. Even though there are better ways to achieve this mapping, such as cube maps, we will follow this model for the sake of simplicity. The original and sphere mapped images used are shown in Figure 12.

The environment map vertex and fragment shaders are shown in Listing 13 and 14. The vertex shader is responsible to compute the reflection vector used by the fragment shader to access the environment texture.

```
// Output reflection vector per-vertex
varying out vec3 r;
void main(void) {
  // Pass texture coordinate
  gl_TexCoord[0] = gl_MultiTexCoord0;
```

Figure 12. The original image (left) and the image mapped to a sphere (right) to be used as the environment mapping texture, a point light was also added to the final image. Note that even though the environment texture has completely black regions they will never be accessed by the shaders.

```
  // Compute vertex position in model-view space
  vec3 v = normalize( vec3( gl_ModelViewMatrix
    * gl_Vertex ) );
  // Compute vertex normal
  vec3 n = normalize(gl_NormalMatrix*gl_Normal);
  // Compute reflection vector
  r = reflect( u, n );
  // Pass transformed position
  gl_Position = ftransform();
}
```

Listing 13. Environment Map Vertex Shader

In the vertex shader a reflected vector of the view direction over the normal is computed and passed to the fragment shader. This vector gives the direction of the simulated incoming light from the environment, i.e. is the point in space we are seeing through the reflection on the specular surface of the object.

```
// Input reflection vector from vertex shader
varying in vec3 r;
// Texture id to access environment map
uniform sampler2D envMapTex;
void main(void) {
  // Compute texture coordinate using the
  // interpolated reflection vector
  float m = 2.0 * sqrt( r.x*r.x + r.y*r.y
    + (r.z+1.0)*(r.z+1.0) );
  vec2 coord = vec2(r.x/m + 0.5, r.y/m + 0.5);
  // Read corresponding texture element
  vec4 texel = texture2D(envMapTex, coord.st);
  // Output texture element as fragment color
  gl_FragColor = texel;
}
```

Listing 14. Environment Map Fragment Shader

The interpolated reflected vector per fragment is used to fetch the texture. This is done by parametrizing the vector over a circle that will match our fish eye texture. The result is shown in Figure 13, applying the environment map of Figure 12 to different models. Figure 14 illustrates a different texture for the environment map.

F. Spike Effect

The last example is a GPU *Spike Effect* that illustrate a special effect using the geometry shader. Listing 15 shows

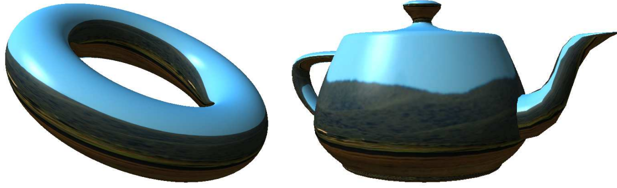

Figure 13. Torus and teapot models rendered with the environment map shaders. Note that the blue comes from the sky in the original image, and not from the teapot color from the other examples.

Figure 14. The teapot rendered with a constant color (left) and a different environment map (right).

the vertex shader. It is the simplest possible vertex shader containing only one line to receive a vertex and output it without modification. The goal is to keep the vertex untransformed in order to send it in the world's coordinate system to the geometry shader.

```
void main(void) {
  gl_Position = gl_Vertex; // Pass-thru vertex
}
```

Listing 15. Spike Vertex Shader

The next step is where the special effect takes place. The geometry shader, shown in Listing 16, receives triangle primitives with untransformed vertices, and creates new primitives rebuilding the surface to create a spike effect. Each input triangle is broken into three triangles using the centroid, where each new triangle has one-third of the original size. The centroid is displaced by a small offset along the normal direction to create the spike effect.

```
varying out vec3 normal, vert; // Output to FS
void main() {
  // Store original triangle's vertices
  vec4 v[3];
  for (int i=0; i<3; ++i)
    v[i] = gl_PositionIn[i];
  // Compute triangle's centroid
  vec3 c = ( v[0] + v[1] + v[2] ).xyz / 3.0;
  // Compute original triangle's normal
  vec3 v01 = ( v[1] - v[0] ).xyz;
  vec3 v02 = ( v[2] - v[0] ).xyz;
  vec3 tn = -cross( v01, v02 );
  // Compute middle vertex position
  vec3 mp = c + 0.5 * tn;
  // Generate 3 triangles using middle vertex
  for (int i = 0; i < gl_VerticesIn; ++i) {
    // Compute triangle's normal
    v01 = ( v[(i+1)%3] - v[i] ).xyz;
    v02 = mp - v[i].xyz;
    tn = -cross( v01, v02 );
    // Compute and send first vertex
    gl_Position = gl_ModelViewProjectionMatrix
      * v[i];
    normal = normalize(tn);
    vert = vec3( gl_ModelViewMatrix * v[i] );
    EmitVertex();
    // Compute and send second vertex
    gl_Position = gl_ModelViewProjectionMatrix
      * v[(i+1)%3];
    normal = normalize(tn);
    vert = vec3(gl_ModelViewMatrix * v[(i+1)%3]);
    EmitVertex();
    // Compute and send third vertex
    gl_Position = gl_ModelViewProjectionMatrix
      * vec4( mp, 1.0 );
    normal = normalize(tn);
    vert = vec3(gl_ModelViewMatrix*vec4(mp,1.0));
    EmitVertex();
    // Finish this triangle
    EndPrimitive();
  }
}
```

Listing 16. Spike Effect Geometry Shader

In the geometry shader code, the triangle's centroid and normal vector are computed using the original vertices positions. The centroid is then displaced by moving it half way along the normal direction. The displaced centroid is then used to build three new triangles. The model-view and projection matrices are applied to the original vertices and the displaced centroid to convert them to the camera's coordinate system. Each of the three output triangles are built using a combination of two original vertices plus the centroid.

The *Spike Effect* example does not have a specific fragment shader. In order to illustrate different shaders combinations one of the two previous defined fragment shaders are used without further modifications: *Phong Shading* or *Environment Mapping*. This is a powerful feature of shader programming, the developer is free to combine different effects and shaders obtaining interesting new results.

It is also interesting to note that both shaders discard the vertex color attribute since they are not used to evaluate the final color of the fragments, *Phong shading* uses material properties while *Environment Map* fetches the color from a texture. Figure 15 shows the final result of applying the *Spike Effect* with *Phong Shading* and *Environment Mapping*.

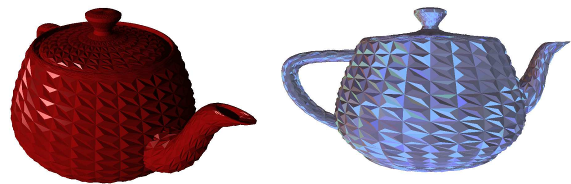

Figure 15. *Spike Effect* shader applied in combination with *Phong Shading* (left) and *Environment Map* (right).

VI. Shaders Summary

During the last sections we have introduced the GLSL language through a few examples. Since it has a very similar structure as other programming languages such as C, the real challenge resides in learning how to design the shaders within the rendering pipeline. More specifically, the data flow is one of the most important points to have in mind, that is, knowing what flows in and out of each stage.

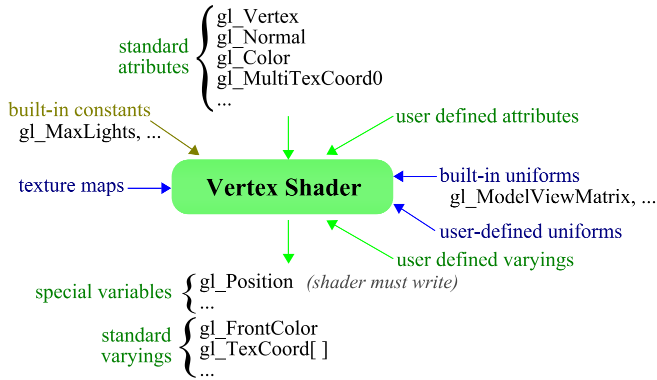

Figure 16. Input/Ouput summary of the vertex shader.

Figures 16, 17, and 18 illustrates the input/ouput variables of the each shader. Note how attributes coming in the vertex shader may be passed forward as special or varying variables.

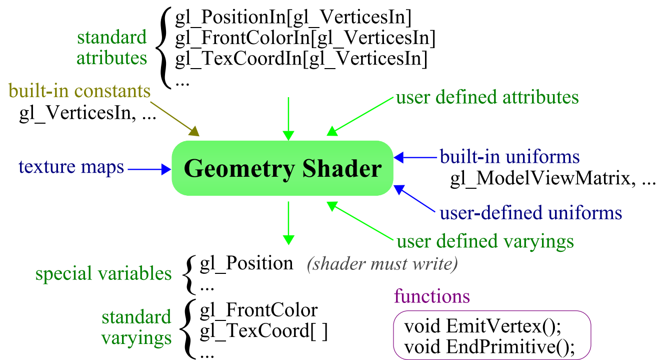

Figure 17. Input/Ouput summary of the geometry shader.

As an example, lets analyze how the color value flows through the pipeline. It first enters the vertex shader as `gl_Color` and leaves as `gl_FrontColor`; the geometry shader receives each vertex color as `gl_FrontColorIn[vertexId]` and outputs again as `gl_FrontColor` for each emitted vertex; finally the fragment shader receives the interpolated color as the `gl_Color` variable and writes the result to `gl_FragColor`. Figure 19 illustrates this particular flow of the color values.

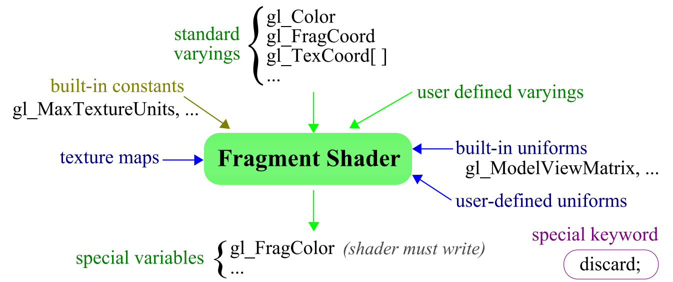

Figure 18. Input/Ouput summary of the fragment shader.

A. Upcoming Features

Recently OpenGL 3.0 was released where many fixed functions are marked as deprecated, and, in fact, have already began to be removed from the API with the newer 3.1 and 3.2 versions. GLSL is also being reformulated and versions 1.3, 1.4 and 1.5 have been released in a very narrow time frame. The newer versions point to a future where shader programming will not be an option, but a requirement for working with the graphics API, since most of the pipeline will have to be written by the programmer himself. At first, this might difficult the learning curve for newbies in graphics programming, but it will force them to gain a better grasp of how graphics computation is handled, and consequently design better and innovative graphics applications.

Another clear evidence of this tendency is the introduction of new programmable stages: the hull and domain shaders plus the tessellator, which is a configurable stage. However, up until this point, they have not yet being integrated with the hardware and are implemented only via software.

VII. General Purpose GPU

Since the GPU comprises such a powerful parallel architecture, many programmers have been using it for other purposes other than graphics, a trend known as GPGPU, or General Purpose GPU. The essence of stream programming is the data, which is a set of elements of the same type. A set of kernels can process the data by operating on the whole stream.

We recall that the graphics hardware cannot handle all type of parallel paradigms since it is not good at solving flow control issues; on the other hand, it performs extremely efficiently within the streaming paradigm. Multicore processors for example, many times perform different tasks in parallel, which is different from parallelizing a single task. The lack of complex control structures also partially explains why GPU's performance has increased in a rate much higher than CPUs: it is easier to assemble more processor together if they can act more independently and the global memory access requires little control.

While the GPU has few levels of memory hierarchy, the CPU has different cache levels, swap, main memory among other resources. This requires a high control level and many

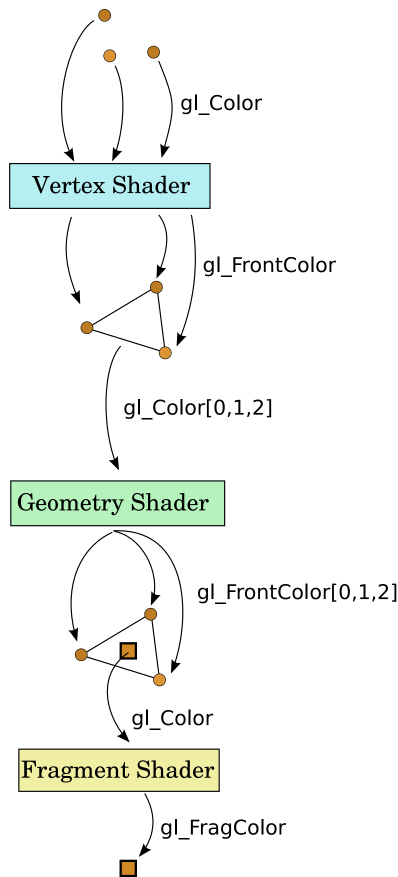

Figure 19. The flow of the color variables through the shaders inside the rendering pipeline.

transistors must be dedicated to the task, but the advantage is that it allows the CPU to significantly reduce latency issues, i.e. the time to fetch information.

The GPU has an immense parallel computational power, operates on many primitives at the same time and computes arithmetic operations extremely fast; nevertheless, it still needs data to compute. Modern graphics hardware have decreased considerably the difference from CPU to GPU memory capacity, with some achieving the mark of 1Gb of memory. Nevertheless, a common bottleneck is transferring the data to the graphics card. A large memory capacity partially solves the problem, because in most cases the data can be uploaded once to the GPU and used thereafter, avoiding the slow CPU-GPU transfer. In fact, many games rely heavy on compacting data to be able to fit everything in memory and avoid this problem. On the other hand, general purpose applications usually have to handle this deficiency with other strategies since even the compressed data may not fit in memory, or the data might be dynamic and change every frame, such as with simulation applications.

The CPU-GPU interaction works as a command buffer, and transferring data is one of such commands. There are two threads in play, one for the CPU that adds commands, and another for the GPU that reads commands. How fast the commands are written or read determines if our application is CPU or GPU bound. If the GPU is consuming commands faster than the CPU is writing, at some point the buffer will be empty, meaning that the GPU will be out of work and we are CPU bound at this point. On the other hand, if we fill up the buffer the GPU is not able to handle all the commands fast enough, and we are GPU bound in this case.

Fortunately, the CPU-GPU interaction is handled in a smart way by the API and we do not have to worry about most of the problems that one might have with bottlenecks from sequentially adding commands. The GPU can process most commands in a non sequential manner and is able to substantially reduce the latency; for example, it does a very good job of continuing with other tasks when a current job is waiting for some information to arrive.

VIII. Conclusion

In this survey, we have exposed an introductory walk-through to shader programming designing using the GLSL language, while at the same time pointing out some important aspects of GPU programming. Unfortunately, it is by no means a complete reference as this is not possible to be achieved in a few pages. Many surveys with more specific and detailed information are available on the internet [1], [2], [15], [16], [17], [18], [19]. Other source of specialized information is the GPU Gems books series, offering a variety of advanced examples on applications and effects that can only be achieved using the graphics card and programmable shaders [20], [21], [22].

GPU Programming is not anymore a small specialized niche of computer graphics, it allows the developer to achieve a wider and more efficient variety of visual effects. It is also embedded in a big turn that software development is making as a whole, the "*Think Parallel*" slogan is everyday more imminent, be it within GPUs, clusters, Cell or multicore processors.

It is no wonder that over the last decade the GPU's growing curve, on what matters GFLops/sec, is astonishingly higher than those of the CPUs. In fact, a single CPU processor has undergone little evolution during these last years, what we acknowledge is the growth in number of processors per unit.

The general purpose GPU programming languages, such as CUDA and OpenCL, are unlikely to overtake all kind of graphics hardware implementation. Programming shaders will still be a major skill for those working closely with its real objective, which is to write modifications on the graphics pipeline to achieve different, better, and faster visual effects. This is specially true for the game industry where shaders are heavily employed. As an illustrative example, a modern game may achieve the mark of a few hundreds or more shaders.

To summarize, even though GPU Programming was only a few years ago an exotic and highly technical part of computer graphics, is has evolved into a valuable skill and by now a requirement for graphics programmers. Exploiting the graphics card capability at its best is much more than learning a new language such as GLSL, it requires a deep understanding on how shaders fit into the graphics pipeline and how great efficiency can be reached by profiting from the GPUs parallel power.

Acknowledgement

This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme of the first author. We also acknowledge the grant of the second author provided by Brazilian agency CNPq (National Counsel of Technological and Scientific Development), and thank Robert Patro for his fruitful insights on the shader examples.

References

[1] "General purpose gpu." [Online]. Available: http://gpgpu.org/

[2] "nvidia corporation." [Online]. Available: http://www.nvidia.com/

[3] "nvidia's cuda." [Online]. Available: http://www.nvidia.com/object/cuda_home.html

[4] "Opengl." [Online]. Available: http://www.opengl.org/

[5] "Opengl shading language." [Online]. Available: http://www.opengl.org/documentation/glsl/

[6] "nvidia's cg." [Online]. Available: http://developer.nvidia.com/page/cg_main.html

[7] "Microsoft's hlsl." [Online]. Available: http://msdn.microsoft.com/en-us/library/bb509561\%28VS.85\%29.aspx

[8] "Opencl." [Online]. Available: http://www.khronos.org/opencl/

[9] "Glut." [Online]. Available: http://www.opengl.org/resources/libraries/glut/

[10] "Qt." [Online]. Available: http://qt.nokia.com/

[11] "wxWidgets." [Online]. Available: http://www.wxwidgets.org/

[12] M. Bailey and S. Cunningham, *Graphics Shaders Theory and Practice*. A K Peters, 2009.

[13] R. J. Rost, *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, January 2006.

[14] Opengl, D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.

[15] "Lighthouse glsl tutorial." [Online]. Available: http://www.lighthouse3d.com/opengl/glsl/

[16] "Gpu shading and rendering course." [Online]. Available: http://old.siggraph.org/publications/2006cn/course03/index.html

[17] "Textures in glsl." [Online]. Available: http://www.ozone3d.net/tutorials/glsl_texturing.php

[18] "Glsl introduction." [Online]. Available: http://nehe.gamedev.net/data/articles/article.asp?article=21

[19] "Clockwork coders glsl tutorials." [Online]. Available: http://www.clockworkcoders.com/oglsl/tutorials.html

[20] R. Fernando, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[21] M. Pharr and R. Fernando, *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005.

[22] H. Nguyen, *GPU Gems 3*. Addison-Wesley Professional, 2007.