A High-Level Abstraction for Graphics Hardware Programming

DANILO TULER, WALDEMAR CELES

Tecgraf - Computer Science Department, PUC-Rio
Rua Marquês de São Vicente 225, 22450-900 Rio de Janeiro, RJ, Brazil
{tuler, celes}@tecgraf.puc-rio.br

Abstract. Currently available graphics hardware has become programmable. However, this programmability is still offered at a very low abstraction level, usually based on a specialized assembly language, and is difficult to use. To overcome this problem, we present a simple programmable pipeline abstraction. Our proposal is built over the Lua programming language. The application programmer writes a Lua code that, when executed, generates the equivalent assembly code to be loaded by the hardware. We compare our proposal to other recently published ones or still under consideration.

Keywords: programmable pipeline, shading language, rendering.

1. Introduction

Graphics hardware has been evolving quickly in the last few years, both in processing power and in functionality. Part of the rendering pipeline, such as vertex transformation and illumination and rasterization, formerly performed by the CPU, is today performed by the graphics processor, or GPU. The range of graphics effects obtained has also increased. For instance, it is possible today to obtain shadow effects in complex scenes in real time.

Graphics libraries are used by an application program to access hardware resources. Today's mainstream graphics libraries are OpenGL [1] and DirectX [2], both vastly supported by hardware manufacturers.

For a library to be aligned with hardware evolution, it must be extensible or regularly issue a new version or update. DirectX has opted for the second model, and is today in version 8.1, with plans for version 9. OpenGL, currently in version 1.3, has an extension mechanism, managed by the OpenGL Architecture Review Board (ARB), an independent consortium with industry participation. Typically, a hardware manufacturer proposes an extension and implements it for its hardware. Other manufacturers can also implement it. After evaluations and studies by the ARB, this extension can become "official", or even part of the kernel of a new version of OpenGL.

An OpenGL application can make use of the extension, but it must always verify, at run time, whether it is supported by the hardware. Alternatives must be implemented in case the extension is not available. This makes the application's work more difficult and specialized, but still flexible. The OpenGL mechanism is interesting; however, nowadays there is an excess of extensions – almost 270 [3] –, thus degrading code portability.

To increase the range of possible effects without the need for issuing an extension (or a new version) for each new effect, graphics hardware has become programmable. This is an important step in the evolution of real-time rendering, being even considered as the beginning of a new hardware generation.

The first versions of programmable hardware expose their internal structures for the programmer by means of a very low-level mechanism. The programmer is required to write the code in a specialized assembly language and to manage sets of registers. Programming in this level is difficult and prone to errors. Different proposals have been presented to abstract the hardware and to gain productivity, portability and ease of use [4, 5, 6, 7, 8, 9]. These abstrations are usually referred to as *shading language*.

This paper proposes another higher-level abstraction for the programmer to access the features offered by the hardware. Our proposal is less ambitious than others; on the other hand, we believe it is easier to use. Its design is closer to today's hardware and it makes use of an existing extension language to support its implementation.

2. Related Work

Shading languages have evolved from Cook's work [10], which showed how illumination trees offer a flexible and powerful mechanism for illumination computations, and

from Perlin's work [11], which describes a language for pixel processing.

Research into shading languages has materialized commercially in products such as RenderMan [12], used mainly in motion picture productions. RenderMan is a scene-description and rendering interface developed with the intention of becoming a standard. The specification defines an illumination language, the RenderMan Shading Language, which is extremely generic and powerful.

Recently, several papers have shown that the new hardware generation is capable of producing in real time effects that were formerly obtained after hours or days of rendering. Olano and Lastra [4] developed *pfman*, a language similar to RenderMan, for the PixelFlow [13] system – an architecture proposal that was not successful in the hardware industry but which provided great contribution to research.

The game Quake III Arena [14], produced by Id Software, has defined a script language for configuring the several layers of texture and color applied to objects. It also has a mechanism for animating and manipulating texture coordinates. The graphics engine interprets the program and translates it in multiple rendering steps in OpenGL, possibly employing multi-texture to reduce the number of steps.

Peercy et al. [5] presented a new approach for the implementation of a language using multiple-step rendering. The central idea is to consider the rendering pipeline as a SIMD (Single Instruction, Multiple Data) machine. A rendering step is considered as an instruction applied to data, vertices and fragments. Peercy has shown that the RenderMan language can be compiled using multiple-step rendering with OpenGL.

Proudfoot et al. [6] described a procedural rendering system by formalizing a programmable pipeline abstraction and defining four computation frequencies: constant, per primitive, per vertex and per fragment. The user code, written in a language similar to RenderMan, is analyzed and compiled, generating programs for this abstraction. The system runs on OpenGL and profits from recent programmable hardware.

The proposal for version 2.0 of OpenGL is under development. The first specification documents have been produced by initiative of 3Dlabs [7] and are being discussed in meetings of the ARB. The main purposes of this new version are to support programmability, via a shading language; to improve memory management; and to provide the application more control over the rendering pipeline. The OpenGL 2.0 shading language includes features not supported by today's hardware, such as control-flow structures.

More recently, NVIDIA has proposed the Cg programming language [8], a high-level, C-like language to access their hardware features, and McCool et al. [9] have proposed, using standard C++, a high-level shading language directly in the graphics library API.

Our work is similar to the proposal presented by Proudfoot et al. [6]. We have borrowed from them the concept of different computation frequencies, but we have opted to map different functions to different frequencies. We believe this choice to be better, because it simplifies the language and its use. Our work is also close to the OpenGL 2.0 proposal, but we have deliberately designed our language to fit in today's hardware, while keeping the language as abstract and simple as possible.

3. Pipeline Abstraction

The processing performed by graphics boards is traditionally modeled by a pipeline composed of stages with specific functions. Graphics primitives described by vertices are sent to the pipeline. The vertices are usually transformed, illuminated, projected, and mapped to window coordinates. The primitives are then rasterized, producing fragments that may be promoted to pixels on the screen.

The new pipeline model allows vertices and fragments to be processed by means of a procedural code developed by the application. Two manufacturers of graphics processors have products that support the programmable pipeline: NVIDIA and ATI.

The extensions for vertex processing are similar in both NVIDIA and ATI proposals. A code in a specialized assembly language, called *vertex shader*, is responsible for transforming and lighting the vertex, apart from generating its texture coordinates. This code operates over a limited set of operators and registers.

Fragment processing consists basically of accessing texture units and combining several colors to generate a unique color to be stored in the screen buffer. The extension proposed by NVIDIA separates the tasks of accessing the texture and combining the colors into two distinct sub-stages, called *texture shader* and *register combiner*. ATI's extension provides a single programming unit, in which processing is made by a program called *fragment shader*.

A shading language is based on an abstract programmable pipeline. Different proposals adopt different abstractions. We shall now describe our abstraction by pointing out its similarities and differences with others.

3.1 Hardware Virtualization

In different degrees, both Proudfoot et al. [6] and OpenGL 2.0 [7] proposals virtualize the existing hardware to remove resource constraints (OpenGL's proposal is less ambitious). Should the shader exceed resource limits, a multi-pass rendering strategy is used. Similar to McCool et al. [9], we have opted to be pragmatic: resources are to be limited and dependent on the existing hardware. We abstract the access to resources but not their limits. The programmer is warned if the resources are insufficient. Our intention is to simplify the abstraction's implementation and to ensure real-time rendering.

3.2 Stage Interfacing

Proudfoot et al. [6] have identified four computation frequencies (constant, per primitive, per vertex, per fragment) and supported all of them into a single shader. The frequency of each computation can be either explicitly provided or inferred by the compiler. The OpenGL 2.0 proposal identifies two separate units: vertex shader and fragment shader. Constants can only be mapped via a conventional host language API.

The problem with using different units is in how to interface them. OpenGL 2.0 borrows the *varying* modifier from RenderMan, so that, in the vertex shader, the programmer explicitly indicates which variables must be interpolated to be available to the fragment shader. The binding is done by matching types and names of variables in both shaders.

Similarly to the NVIDIA extension, our proposal also distinguishes four computation frequencies, but we have chosen to adopt different processing units. We believe this is easier to the programmer, promotes cleaner codes, and facilitates shader reuse. Constants are mapped to global variables and *primitive-frequency functions* manipulate them. Such variables are mapped to hardware constant registers.

The input for a vertex shader comes from an *input* table[1]. Vertex position, for instance, is accessed by *input.pos*. The vertex shader output must be stored in an *output* table that becomes the *input* for the fragment shader. Except for a few fields (such as *pos* in vertex and *color* in fragment), the programmer is free to choose appropriate names to denote their semantics.

[1] *Table* is a basic type of the programming language Lua [16] and represents an associative array.

4. Shading Language

A major difference between our proposal and others is that we have not created a new specific language to support programming, while still using a small, high-level language. Our shading language was implemented over the Lua programming language [15]. Lua is a simple, lightweight extension language with extensible semantics. New types of data can be defined and operations can be intercepted by means of an event mechanism. Lua has been widely used in different graphics applications, especially in computer games [16].

In our proposal, each shader is implemented by a function. All types of constructions are allowed in a shader code, except for general conditionals and loops involving graphics objects as control variables. However, the programmer can, for instance, use loop constructions to integrate each light-source contribution, and use conditionals to parameterize the code. Auxiliary functions can be defined to perform a traditional illumination computation, or libraries of texture-coordinate generators can be built, thus allowing code reuse. Conceptually, function calls are inlined and loops are linearized before the code is compiled to the hardware.

There are only three types of graphics objects: *scalar*, *vector* and *matrix*. Scalars are real numbers, vectors are 4-dimension arrays, and matrices are 4x4 two-dimensional arrays. Our first evaluation has indicated that these types are enough to code shaders. Similarly to the OpenGL 2.0 proposal, the vector components may be accessed by means of a single letter or a numeric value (*v.x*, *v.r*, *v.s*, and *v[1]* all refer to the first component of *v*). Swizzling, replication and masking are also supported, reflecting hardware facilities. Matrices represent an array of row vectors, so individual components can be accessed similarly (e.g., *m[1].x*).

The operators available to be used on these graphics types are limited to the set of operators provided by currently programmable hardware.

The shaders are written in Lua. Each functional stage, such as vertex processing, is represented by a Lua function. The global environment can define objects to be used by the functions, such as constants or tracked transformation matrices. An API in C was defined to create and manage shaders, as well as to access the values of the defined constants.

The execution of the Lua code, as a whole or in each stage's function, triggers a code generator that translates the Lua code into an equivalent assembly code to be loaded to the hardware, or into a set of function calls used to properly configure the graphics library. The application

is responsible for selecting the correct shaders to render the scene and for loading the appropriate textures.

Because Lua provides both procedural constructions and descriptive facilities, the programmers are able to combine shading language code into scene description, in a way similar to RenderMan.

4.1 Code Generator

Our code generator architecture is illustrated in Figure 1. We currently generate code to run over the OpenGL library, but it should be simple to implement a DirectX interface.

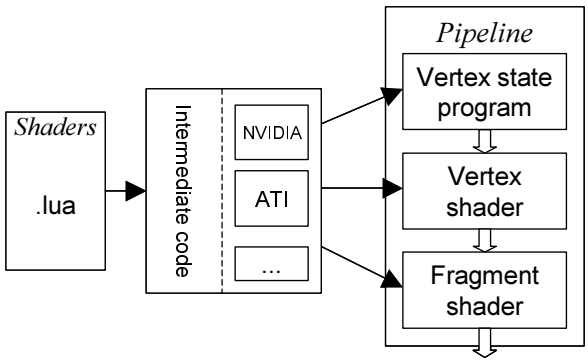

Figure 1 Code generator architecture.

The hardware programming extensions are exported to Lua, so that Lua codes can invoke functions to set the hardware up. The *pipeline* is an object composed by three short Lua functions (for primitive, vertex, and fragment shaders) referred here as *binding functions*. These functions are automatically generated and, when executed, load the corresponding assembly program and set the appropriate hardware configuration. For instance, for the vertex shader, using NVIDIA extension, the binding function consists uniquely in a call to bind the generated vertex program to the hardware by means of the graphics library.

The application programmer writes the shaders in Lua. Each shader must be implemented as a different function. From the execution of a Lua shader function, the code generator creates two distinct codes: the equivalent assembly shader code and the corresponding Lua binding function. The programmer then chooses the shaders to compose the pipeline object and binds it to the hardware.

The code generator is implemented in a high abstraction level once it is also written in Lua. Therefore, to keep the code up-to-date to evolving hardware should not be a problem. The code generator extends Lua semantics to deal with graphics object. Any operation on these objects is intercepted, allowing the conversion from Lua code to assembly code. An intermediate representation of the shader code is created, to be analyzed and optimized before the assembly code is generated. The intermediate representation of the code is used to perform efficient resource allocation and code optimization, and to cope with the non-orthogonal aspect of the hardware. Simple optimizations include the elimination of copy instructions and computation standards, which can be replaced by specialized instructions.

4.2 Constant Frequency

The constant computation frequency is used to set global variables to be used by the shader functions. It is also possible to request the tracking of matrix values – a handy feature available in current extensions. We have identified that these tracking mechanisms should be extended, for instance to lights and materials, and plan to do so in the future. Figure 2 illustrates a code in the global environment.

```
-- light position in eye space
light_pos = Vector{0.0,0.0,1.0,0.0}
material_color = Vector{0.3,0.5,1.0,1.0}

-- tracking of matrices
mvp = Matrix("MODELVIEW_PROJECTION")
mvi = Matrix("MODELVIEW","INVERSE")
mv  = Matrix("MODELVIEW")
```

Figure 2 Global environment example.

The pieces of code presented here may all be part of the same Lua module. How to sever the codes into modules is a programmer decision. Each shader can be loaded separately by executing the corresponding Lua function.

4.3 Primitive Frequency

The primitive frequency is used to compute values shared by one or more primitives. It is directly mapped to the *vertex state program* provided by the NVIDIA extension. Due to its low frequency, it is not a problem to simulate this stage for hardware that does not externalize one. It may even be feasible to implement the simulation in Lua. As an example, the code in Figure 3 transforms the light and view vector to object space e computes the half vector in object space.

The allocation of constant registers is made by analyzing the usage, in vertex and fragment shaders, of variables declared in the global section of the code and the ones assigned in the primitive functions. Each register is composed by four real numbers. Matrices are allocated to

blocks of four registers in a sequence, vectors are allocated to a register, and scalars are allocated to a component of a register. We use the following allocation policy: first the matrices are allocated, then the vectors, and finally the scalars, using a greedy algorithm. Register allocation is done assuming that the whole set of registers is available. It might be interesting to prevent the use of pre-defined sets, so that we could have different shaders using different sets.

```
l = Vector {} -- object space light vector
v = Vector {} -- object space view vector
h = Vector {} -- object space half-vector

function prim_shader ()
  -- transform light to object space
  l = -(mvi * light_pos)

  -- Fetch camera space view vector
  -- Transform into object space
  v = -(mvi * eye)

  -- Compute H vector in object space
  h = normalize(l + v)
end
```

Figure 3 Primitive function example.

4.4 Vertex Shader

A vertex shader, called *vertex program* by NVIDIA, is a function to replace the stages of transformation, illumination, projection, and texture-coordinate generation of the conventional pipeline.

In the hardware, a vertex shader is a sequence of low-level instructions that operate on registers. It is executed for each vertex sent to the pipeline. Vertices cannot be created or destroyed, and there is no adjacency information. Figure 4 illustrates the schematic model of vertex shader.

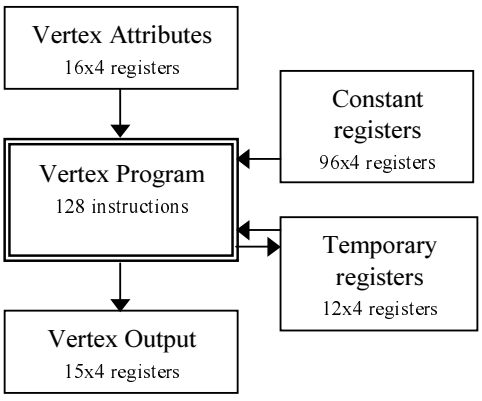

Figure 4 NVIDIA's GeForce 3 vertex program.

There are four sets of registers that can be used and must be managed by the code generator: vertex attributes, output attributes, temporary registers and constant registers. Vertex attributes are read-only registers that contain information on the vertex such as position, normal, texture coordinates, color, or other attributes with a semantics defined by the application. Output registers are write-only and contain the values computed by the program. The vertex shader must at least compute the transformed vertex position, but it may also compute values to be interpolated during rasterization. Temporary registers can be used to store values during computation, and can be freely written and read. Constants are read-only variables for the vertex shader (the global environment and primitive functions set them). Any data needed by the program to carry out the computation, such as transformation matrices, information on light sources and materials, must be loaded into these registers.

All instanced objects reference a given register in the vertex shader machine – either input, output, temporary or constant registers. The input and output registers are mapped to *input* and *output* tables in the Lua vertex shader code. Temporary register allocation is performed when the Lua shader function is executed, that is, during the generation of the assembly code. New, local objects are created dynamically, being either scalars, vectors or matrices. This creation can be explicit or implicit, resulting from an operation. The allocation policy is similar to that of constant registers. We take advantage of Lua's garbage collection mechanism to reuse registers. Should we run out of resources, we can trigger garbage collection, which automatically identifies temporary registers no longer in use.

Figure 5 illustrates a vertex shader code and Figure 6 shows the generated NVIDIA assembly code. The gain in readability and ease of programming is clear.

Naturally, the programmer is able to write and reuse auxiliary functions such as *sin*, *cos*, *exp*, *sqrt*, etc [17]. The shader function can also receive conventional input parameters, being possible to write parameterized codes.

```
function vertex_shader ()
   output.pos = mvp * input.pos
   output.tex0 = input.tex0
   local p = mv * input.pos
   local l = normalize(light_pos - p)
   local n = mvit * input.normal
   local NdotL = max(dot(n, l), zero)
   output.color = mat_color * NdotL
end
```

Figure 5 Vertex shader example.

```
!!VP1.0
DP4   o[HPOS].x, c[0], v[OPOS];
DP4   o[HPOS].y, c[1], v[OPOS];
DP4   o[HPOS].z, c[2], v[OPOS];
DP4   o[HPOS].w, c[3], v[OPOS];
MOV   o[TEX0], v[TEX0];
DP4   R0.x, c[4], v[OPOS];
DP4   R0.y, c[5], v[OPOS];
DP4   R0.z, c[6], v[OPOS];
DP4   R0.w, c[7], v[OPOS];
ADD   R1, c[12], -R0;
DP3   R1.w, R1, R1;
RSQ   R1.w, R1.w;
MUL   R1.xyz, R1, R1.w;
DP4   R2.x, c[8], v[NRML];
DP4   R2.y, c[9], v[NRML];
DP4   R2.z, c[10], v[NRML];
DP4   R2.w, c[11], v[NRML];
DP4   R3, R1, R2;
MAX   R4, R3, c[14].x;
MUL   o[COL0], c[13], R4;
END
```

Figure 6 NVIDIA's vertex program example.

4.5 Fragment Shader

Fragment processing is approached differently by the NVIDIA and ATI extensions. NVIDIA proposes a separation of the processing in two parts: texture access and combination. ATI proposes a model similar, though less versatile, to that of vertex shader: a set of input, output, constant and temporary registers and a set of instructions for mathematical computations and texture access.

ATI's fragment-processing model is more orthogonal and flexible, making it easier to develop a higher-level abstraction. NVIDIA's proposal, though having similar expression power, makes it more difficult to implement an optimization of resource allocation. The architecture of ATI's model for fragment processing is illustrated in Figure 7.

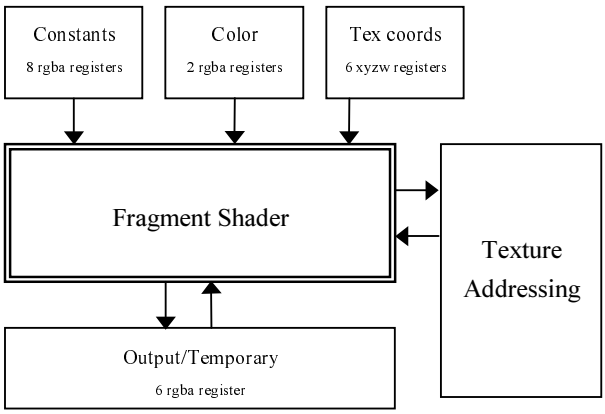

Figure 7 ATI's fragment shader architecture.

The fragment shader is a register-based machine. There are four sets of registers: constant, temporary, texture and color. Constant registers hold four fixed-point values loaded before shader execution. Temporary registers are used to store intermediate results. The first temporary register is special because it holds the resulting output color. Texture registers contain read-only texture-coordinate data, and color registers contain per-fragment color values.

There are two kinds of instructions available: texture instructions and arithmetic instructions. Prior to performing arithmetic instructions in the fragment shader, maps may be sampled and texture coordinates may be routed into registers using texture instructions.

Arithmetic instructions include common mathematical operations such as *add*, *subtract*, *multiply*, and *dot product*. The extension provides multiple shading passes separated by texture sampling, thus it is possible to make use of dependent texturing.

DirectX 9.0 will adopt ATI's model, and the OpenGL 2.0 proposal is also similar to it. Therefore, we have decided to implement our fragment shader in a way similar to what we have done for the vertex shader. Our code generator for the fragment shader is still under development. We have provided the programmers a limited set of operators and data types (only scalars and vectors). The resource allocation policy has also been implemented similarly.

5. Conclusion

It is clear that we need a higher abstraction level to access programmable hardware. Recent research has presented different abstraction proposals. Our proposal is less ambitious than others; however, we believe it is capable of expressing the shaders to be commonly used in real-time rendering.

A major difference in our proposal is the fact that we have built the abstraction over the Lua language instead of creating yet another specific language. The code generator was written entirely in Lua at a high abstraction level, thus being easy to explore new hardware features.

The facilities provided by Lua allow the codification of shaders within scene graph description, just like RenderMan. Proudfoot et al. [6] pointed out the need of a shading language intended for artists. We believe the use of Lua may be a first step toward this goal, once Lua has been widely used as a script language in the game industry.

Our pipeline abstraction conceptually separates the four computation frequencies. This makes the reuse of shaders easier and simplifies assembly code generation.

The vertex-processing architecture is far more consolidated than the fragment-processing architecture, in which what will be adopted as a standard is still uncertain.

A high-level abstraction for processing vertices was created without great difficulties, and the purposes of code transparency and reuse were reached. Portability was also obtained, since the same program in this abstraction can produce code for distinct hardware.

A future 2.0 version of ATI's fragment shader promises a more powerful and flexible architecture than the current one for processing fragments, allowing an easier creation of abstractions.

We intend to use our proposal as an educational tool for teaching sophisticated real-time rendering algorithms.

Acknowledgements

The first author was supported by CAPES. Tecgraf is a laboratory in PUC-Rio mainly funded by PETROBRAS.

References

[1] M. Woo, J. Neider, T. Davis, and D. Shreiner, *OpenGL Programming Guide*, 3rd Edition, Addison Wesley, 1999.

[2] DirectX – http://msdn.microsoft.com/directx/

[3] OpenGL Extension Registry – http://oss.sgi.com/projects/ogl-sample/registry/

[4] M. Olano and A. Lastra, "A Shading Language on Graphics Hardware: The PixelFlow Shading System", *Proceedings of SIGGRAPH 98*, pages 159-168, July 1998.

[5] M. S. Peercy, M. Olano, J. Airey, and P. J. Ungar. "Interactive Multi-Pass Programmable Shading", *Proceedings of SIGGRAPH 2000*, pages 425-432, July 2000.

[6] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan, "A Real-Time Procedural Shading System for Programmable Graphics Hardware", *Proceedings of SIGGRAPH 2001*, pages 159-170, July 2001.

[7] 3Dlabs, *3Dlabs OpenGL 2.0 Specifications*, http://www.3dlabs.com/support/developer/ogl2/

[8] NVIDIA, "The NVIDIA Cg Compiler – C for Graphics", *Technical Brief,* TB-00511-001-v01, 2002.

[9] M. D. McCool, Z. Qin, and T. S. Popa, "Shader Metaprogramming", to appear at the *SIGGRAPH Eurographics Graphics Hardware Workshop*, 2002.

[10] R. L. Cook, "Shade Trees", *Proceedings of SIGGRAPH 84*, pages 223-231, July 1984.

[11] K. Perlin, "An Image Synthesizer", *Proceedings of SIGGRAPH 85*, pages 287-296, 1985

[12] A. A. Apodaca and L. Gritz, *Advanced RenderMan: Creating CGI for Motion Pictures*, Morgan Kaufmann, 2000.

[13] S. Molnar, J. Eyles, and J. Poulton, "PixelFlow: High-Speed Rendering Using Image Composition", *Proceedings of SIGGRAPH 92*, pages 231-240, July 1992.

[14] P. Jaquays and B. Hook, *Quake 3: Arena Shader Manual*, Revision 10, September 1999.

[15] R. Ierusalimschy, L. H Figueiredo, and W. Celes, "Lua – an extensible extension language", *Software: Practice & Experience*, *26 (6)*, 1996. http://www.lua.org/

[16] R. Ierusalimschy, L. H. Figueiredo, and W. Celes, "The Evolution of an Extension Language: A History of Lua", *V Brazilian Symposium on Programming Languages*, B-14–B-28, 2001.

[17] NVIDIA Corporation, "Implementation of Missing Vertex Shader Instructions". http://developer.nvidia.com