SEMINAR

ANALYSE, PLANUNG UND KONSTRUKTION

COMPUTERGRAFISCHER SYSTEME

The OpenGL Shading Language

Stephan Brumme

HASSO-PLATTNER-INSTITUT
UNIVERSITÄT POTSDAM

FACHGEBIET COMPUTERGRAFISCHE SYSTEME
PROF. DR. J. DÖLLNER

WINTERSEMESTER 2003/2004

The OpenGL Shading Language

Stephan Brumme
stephan.brumme@hpi.uni-potsdam.de
Hasso-Plattner-Institute at the University of Potsdam

Abstract The introduction of programmable graphics hardware opened a door to a new era of real-time rendering. The manifold uses of vertex and fragment shaders enable developers and artists to achieve many visuals effects at high frame rates on consumer level hardware, which has been formerly rendered offline. The increasing effort of writing efficient and effective shaders leads to the creation and establishment of new higher-level languages which can be compared to already existing general-purpose languages such as C or Java. One of these shading languages, the OpenGL Shading Language (known as GLSL), will be discussed and examined in depth. I describe its conceptual design, explain how key decisions were resolved and present a working implementation. Moreover, I assess the current situation and what particular extensions or improvements can be expected in the near future. These conclusions will be matched up to concurrently evolving languages like NVIDIA's Cg or Microsoft's High Level Shading Language (HLSL) in order to identify cross-pollinations. A 3D chess game written using GLSL demonstrates a real-life application of modern vertex and fragment shader programming.

CR Categories and Subject Descriptors D.3.2 [Language Classifications] Specialized Application Languages; I.3.1 [Computer Graphics] Hardware Architecture; I.3.3 [Computer Graphics] Picture/Image Generation; I.3.6 [Computer Graphics] Methodologies and Techniques; I.3.7 [Computer Graphics] Three-dimensional Graphics and Realism.

Keywords shading language, procedural shading, real-time image generation.

1 INTRODUCTION

Recent customer level graphics hardware undergoes a major shift from hardly supporting basic drawing operations to rendering complex scenes in real-time. The astonishing demonstration of selected scenes from Square Studios' movie Final Fantasy at SIGGRAPH 2001 [NVIDIA01a] brilliantly underlined the amazing power of modern graphics accelerators.

The fixed functionality enforced by graphic library standards like OpenGL [OpenGL03a] or DirectX [Microsoft02a] has been dominating the conceptual chip design for many years. Meanwhile, commercially successful software renderers such as Pixar's Renderman [Hanrahan90, Pixar] introduced fundamentally new concepts like procedural shading by allowing to program and configure almost each step of the rendering process. These programs are called shaders.

Many applications, such as CAD software or the steadily growing market of computer games, would clearly benefit from the use of shaders if they could run at real-time frame rates thus allowing interactive user interfaces. Since software based solutions turned out to be too slow, graphics hardware vendors provide a limited set of shader processing units in hardware. These shaders are written in a dedicated shader assembler languages mapping directly to the hardware. Only recently a standardized assembler language evolved which does not exploit the actual hardware features to their full extent since it is restricted to the common subset of the major vendors' assembler languages.

Inventing a completely new language does not always lead to a handy and intuitive syntax. Almost all high level shading languages borrow their basic concepts and structures from C [Kernighan88] which is well-known for its high performance and portability while being easily mapped to hardware instructions. The widespread use of C (incl. C++) in the software in-

dustry allows many programmers to get into touch with these shading languages without too much effort.

A high level language pushes the programmer's productivity by reducing hardware dependencies, supporting abstraction layers and enhancing code readability. Even interactive shader development appears to be feasible right now [Maya, ATI].

Moreover, just-in-time compilation allows to map the language to the actual chip using code optimizers. These optimizers will improve over time and thus speed up application even after they were shipped already. New graphics chips and/or innovative architectures can be supported as well by providing adequate drivers that come with specialized shading language compilers. Vice versa, features not available in nowadays' hardware can be emulated: yet, branching is replaced by equivalent operations and loops still get unrolled.

In this paper, I present an overview of the OpenGL Shading Language (abbreviated *GLSL*, sometimes also called *glslang*), which is an essential part of the soon-to-be-released OpenGL 2.0 [OpenGL03b, 3Dlabs02]. I describe the evolution of the basic concepts of shading languages and their influence on GLSL. An emphasis will be put on the design of the language, how it is integrated in the OpenGL framework and what improvements can be expected in the future.

Furthermore, the OpenGL Shading Language's ideas will be compared to similar languages recently developed by NVIDIA and Microsoft, too.

A brief introduction to the language's syntax accompanied by an example should assist the reader in writing shader based applications.

2 RELATED WORK

Cook's shade trees [Cook84] laid the foundations of hierarchically subdividing a shading process into simple routines called *shaders*. An exemplary shade tree is shown in figure 1. A shader's task is to modify or create values associated to a surface, such as its position or color [Akenine-Möller, pp. 213].

Renderman™ supports various kinds of shaders to increase the versatility according to the Reyes (Render Everything You Ever Saw) architecture of Cook et al. [Cook87]. These shaders include displacement, surface, light,

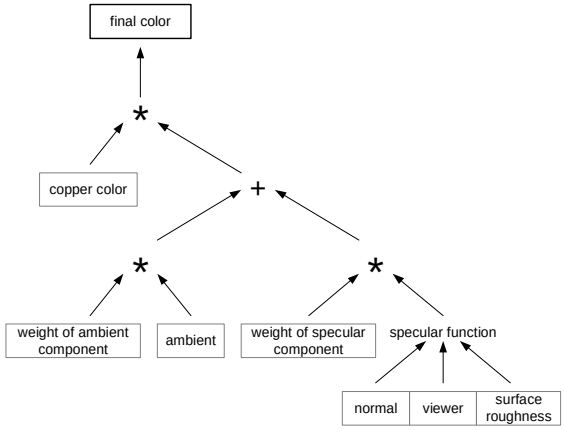

Figure 1. Shade tree for a copper shader [Cook84]

volume, and image shaders which proved to be sufficient for rendering almost any visual effect of movies, games etc.

The PixelFlow system [Olano95] attempted to implement a Renderman-like language in a real-time system. It was a SIMD (single instruction, multiple data) multiprocessor system and utilized the concept of deferred shading in order to shade only visible fragments.

SGI's OpenGL shader system removed the major fragment limitations of graphics hardware by mapping mathematical operations to multipass techniques, mostly texture based. It has been shown that it is possible to generically map almost any shading computation to multipass techniques [Peercy00]. Unfortunately, some of them require an enormous amount of memory bandwidth and a high floating point number precision due to their large number of rendering passes.

The Quake III engine, one of the best selling computer game engines ever, provides its own shading language that targets at fragment level effects, too. It restricts the shaders versatility very much in order to achieve very high frame rates.

The Stanford Shading Group decided to carefully change today's hardware accelerators' design to achieve a more efficient resource usage. They offload lots of fragment operations to the vertex level and rely on the hardware's interpolation abilities [Proudfoot01]. The paper identified four basic classes of scene data by classifying their computation frequency:

- constant for the scene (e.g. lighting)
- constant for a group of primitives (material of a mesh)

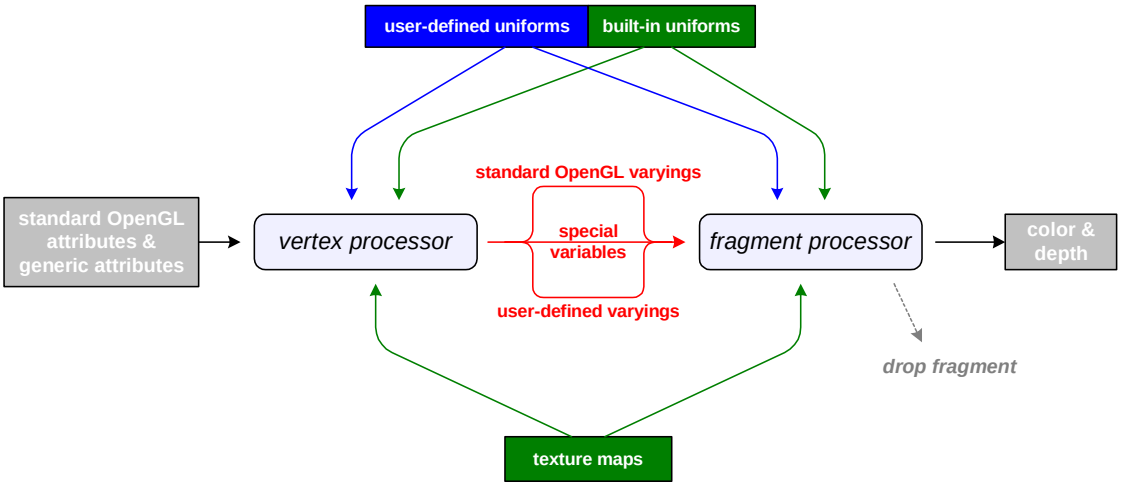

Figure 2. Input, intermediate data flow and output of vertex and fragment shaders

- constant per vertex
- constant per fragment

The first two classes are input to each vertex and/or fragment shader. The third class is both output of a vertex shader and input of a fragment shader while the last class represents data generated by a fragment shader and usually written to the frame buffer. In consequence, graphics hardware's pipelines tend to be subdivided into vertex and fragment stages each being nearly fully programmable. Sometimes the term pixel shader is used when actually referring to a fragment shader.

NVIDIA developed a C-based language called Cg ("C for graphics") [NVIDIA03b] to bring high level shading languages to the mass market. Cg was one of the first real-time languages actually available for consumer level hardware. Its architecture shares many aspects with Microsoft's DirectX9 HLSL because both companies collaborated closely in the process of creating these two languages. The flexible design of Cg is considerably open towards future extensions and fortunately not limited to NVIDIA graphics processors.

3 ARCHITECTUR

3.1 Hardware Considerations

GLSL is designed with the intention in mind to cover at least every functionality of the replaced OpenGL *fixed functionality* and to additionally provide advanced features. It is possible to write shaders that exactly match the operations being replaced without a loss of performance [NVIDIA01b].

Some vendors [ATI] do not even implement the fixed functionality in the chip design, instead they fully emulate it by transparently executing dedicated shaders. McCool et al. reported that an optimized vertex shader outperformed the standard path of a NVIDIA GeForce3 by about 25% [McCool01], too.

3.2 Shader Concept in GLSL

Although strongly related to each other, the OpenGL Shading Language is actually divided into two parts: one language for the *vertex processor* and one for the *fragment processor*.

The OpenGL committee defines a shader as a *unit of compilation* where a set of shaders linked together is called a *program*.

Each shader always operates on exactly a single entity, i.e. a vertex or a fragment, at a time. There is no explicit knowledge about any of the preceding or following entities, too. It is impossible to create new entities within a shader, and only fragment shaders are allowed to drop or discard an entity (i.e. a fragment).

The data flow of a shader is shown in figure 2. The vertex shader is granted read-only access at runtime to so-called *attributes* representing the standard OpenGL vertex attributes (`gl_Color`, `gl_Normal`, etc.). In contrast, *constants* are known at compile-time and thus offer a great potential of performance optimizations.

A *uniform* variable does not change across the primitive being processed, e.g. the position of a light source. All uniforms are read-only, too, and initialized either directly by an application via API commands or indirectly by OpenGL. The available storage for uniforms may be limited, exceeding that thresholds

throws a either a compile-time or a link-time error.

Many effects rely on the availability of *texture maps* to implement lookup tables or to render images to object surfaces. Hence, GLSL permits read-only access via *samplers* that may perform filtering if desired.

A very important stage is located between the vertex and the fragment processor: the rasterizer. It splits primitives modeled by vertices into discrete portions called fragments. The rasterizer has to interpolate many values generated by the vertex processor in order to pass them the fragment processor. For example, the texture coordinates varies for each fragment but can be derived from the vertices via a suitable interpolation. These values – slightly varying for each fragment – are called *varying*s and guaranteed to be perspective correct.

The final result of the graphics pipeline are two values which are usually written to the frame buffer: the fragments' color and their z-depths. Figure 3 visualizes the described relationships.

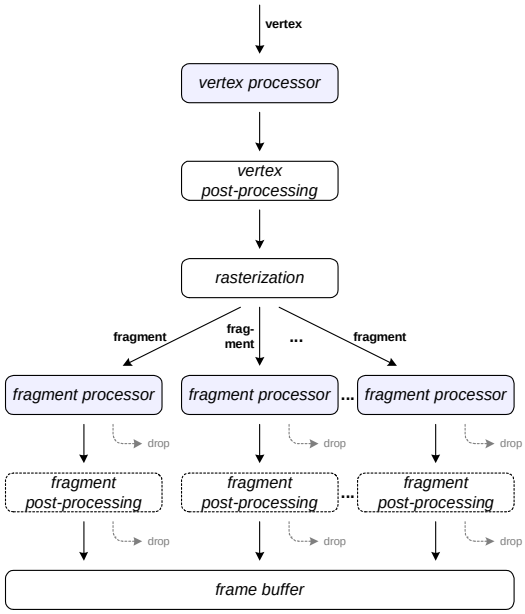

Figure 3. Data processing

3.3 Vertex Processor

Whenever an applications invokes a `glVertex` call (or one of its `glDrawArray` derivates), the vertex and its associated *attributes* – such as its color, its normal, its texture coordinates, user-defined attributes and so on – are forwarded to the vertex processor which is in charge of [OpenGL03a]:

- vertex transformation
- normal transformation and normalization
- texture coordinate generation
- texture coordinate transformation
- lighting
- color material application
- clamping of colors

It is important to be aware of the fact that a vertex shader is fully responsible for all of the above functions. When bypassing the fixed functionality of OpenGL, the programmer has to perform all tasks on his own. This applies especially to vertex transformations which are usually done in the first few lines of a vertex shader.

A great variety of vertex shaders employ texture lookup techniques in order to achieve interesting visual effects. Indeed, that capability of the vertex processor seems to be the most used one and hence requires a huge amount of careful optimizations to overcome cache stall issues as done in the recent NVIDIA ForceWare driver series [NVIDIA]. Vertex shaders often serve to provide the opportunity of handling user-defined data formats fitting the application's specialized needs. Vertex compression/decompression [Deering95, Engel02] pushes the memory bandwidth efficiency by magnitudes which increases the overall system performance. Especially normals seem to offer a great potential of optimizations since in the most cases the viewer does not notice even a dramatically reduced precision.

Some operations cannot be done in vertex shader since they require information about more than a single vertex:

- perspective projection
- primitive assembly
- frustum and user clipping
- backface culling
- two-sided lighting selection
- polymode processing
- polygon offset
- depth range

These operations are performed subsequently to the vertex shader and still conform to the OpenGL 1.4 definitions.

3.4 Fragment Processor

The rasterizer splits primitives into fragments and interpolates all varyings across the primitives.

The fragment processor operates on fragment values and is executed after all vertex stages, including vertex shaders, were successfully completed. It generates just two output values, namely color and depth. They may be written to the frame buffer or an off-screen texture, too.

Typical tasks of a fragment shader are:

- operations on interpolated values
- texture access
- texture application
- fog
- color sum

Like the vertex processor, some parts of the fixed functionality are executed past finishing the shader:

- shading model
- coverage
- pixel ownership test
- scissor
- stipple
- alpha test
- depth test
- stencil test
- alpha blending
- logical operations
- dithering
- plane masking

These operations are implemented in hardware to improve the overall performance and reduce the shader's overhead. Although they can be completely done in a fragment shader, vendors claim that they are cheap to add to the chip design and offer great optimization deals such as preventing pipeline stalls. If desired it is possible to disable all these operations.

Fragment shaders are allowed to read multiple texture multiple times. The corresponding filtering is done according to the set OpenGL state but can be computed in the shader as well in order to obtain specialized results.

4 LANGUAGE

4.1 Goals

The OpenGL Shading Language aims at real-time applications running at interactive frame rates. A shader written in GLSL has to run at a speed comparable to that of a shader written in assembler language. That does not mean that shaders need to be real-time, they can be used for slow and complex computations as well.

OpenGL has been known for many years for its tremendous portability. Unlike the market of operating systems, a single vendor does not dominate the graphics hardware segment, which means that portability should play a still more important role when developing a shading language. On the other hand, almost anything available at the assembly language level should be supported by GLSL.

It is desirable to add GLSL support to currently existing applications with as less as possible effort. The integration into OpenGL should be very tight, seamless, and transparent in order to achieve a high acceptance among programmers.

The language should support means to structure the code and to allow the creation as well as the usage of shader libraries. Common mathematical operations, such as square root or even the noise function, have to be integral part of the system.

4.2 Key Decisions

Some of these goals are in conflict or contradict indeed. Even though the speed of graphics hardware is increasing at an astonishing rate, the emphasis of the OpenGL Shading Language is undoubtedly put on performance.

Achieving an optimal performance requires deep knowledge of the system the shaders will run on. Due to the diversity of available hardware and software configurations, it seems to be impossible to reach always the highest speed possible on each machine. When compiling the shaders just-in-time, i.e. while the application that uses shaders runs, the optimizer has the opportunity to adapt the generated assembler code to the host system. Since the graphics hardware vendors provide these optimizers and put much effort into it, the shaders will perform nearly optimal. Taking advantage of parallel vertex and/or fragment processors available on the chip and streaming extensions of the CPU [Intel], shaders written in a high-level language such as GLSL may even outperform hand-tuned assembler code.

Today's most often used general-purpose languages include C, C++, and Java. Their imperative language proved to be reasonably simple to learn but rather powerful in their application. There is a deep knowledge available how to write fast but optimizing compilers and link-

ers for these languages which simplifies the graphics driver development. On the other hand, the Renderman™ shading language tends to be more declarative or functional because the framework implicitly decides at runtime which kind of shaders (like surface, light, etc.) get invoked. Their order and calling frequencies is not fixed and usually cannot be determined in advance.

Current shaders tend to be quite short in length, even when used for non-real-time digital imaging they seldom exceed 1,000 lines of code. Therefore, object oriented concepts were considered to be not much of help in these cases and therefore omitted. In consequence, GLSL is closer to C than to any of the aforementioned languages. It attempts to be a general-purpose language while still focusing on graphics functionality and a clean, straightforward syntax. Domain-specific languages may perform better by means of productivity. On the other hand, they restrict the application of shaders to graphics while some new innovative uses in non-graphics areas like the computation of massively parallelized algorithms were shown [Krüger03].

The language is strictly case sensitive and supports implicit scoping the way C does. It relies on the same pairing of *compiler* and *linker* to separate the process of code translation from the process of creating reference binding. Therefore, shader libraries only have to be compiled once in advance and then are independent from the shaders invoking them. This approach saves valuable resources, e.g. compile time.

Reducing type checking to compile-time detects most of the common errors but maintains a high performance at runtime. Usually, only wrong texture formats cannot be notified.

4.3 Variables and Types

Experiences gathered in using various languages to develop shaders, such as done in Renderman, revealed the need for just three elementary data types:: `bool`, `int`, and `float`. All integers are always signed and limited to 16 bits. `float` should comply to the IEEE single precision floating-point definition for precision and dynamic range. Internal processing may be less accurate but has to match the OpenGL 1.5 specification.

Pointers are forbidden in GLSL, there is no need for strings, too. An undefined return value of a function is `void`.

New dedicated data types allow for an easy and simple access to vectors and matrices: `vec2`, `vec3`, and `vec4` as well as `mat2`, `mat3`, and `mat4`. Vectors support `bool`eans and `int`s if preceded by the letter `b` or `i` (e.g. `ivec3`). Each vector or color is treated as an implicit union and its components can be read or written via its `x`, `y`, `z`, `w` or `r`, `g`, `b`, `a` members.

Matrices are always constructed using `float`s. A matrix' number of rows equals its number of columns, i.e. the size is restricted to 2x2, 3x3, and 4x4.

Texture access needs the invocation of samplers. They are available for 1D, 2D, 3D, and cube mapped textures. Shadow mapping is supported by 1D and 2D depth textures with automatic comparison.

Variables sharing a common semantics may be composed to a `struct`. It must not be empty, all members types have to be defined in advance.

Another way to organize values are one-dimensional arrays. They can be of any (positive) size, the indices always start at zero.

4.4 Built-in Variables

In the OpenGL Shading Language, there is a data flow from the fixed functionality to the programmable processors and back. These two main parts of the pipeline communicate their shared state through the use of built-in variables. All built-in variables have a global scope and start with the reserved prefix "`gl_`", e.g. `gl_Position`. or `gl_FragColor`.

Not all variables are available to each shader since some are restricted to vertex shaders while others are restricted to fragment shaders.

In addition to the aforementioned special built-in variables, GLSL provides some built-in vertex attributes to access a vertex' color, texture coordinates etc.

Implementation-specific limitations like the number of available vertex texture units are exposed through built-constants.

Unlike the previously mentioned built-in variables, varying variables do not map strictly one-to-one between vertex shaders and fragment shaders. The cause may be an interpolation algorithm as it is applied for colors across a

primitive. Therefore, a fragment shader's input front color does not need to exactly match the color generated by the vertex shader.

4.5 Functions

A function computes a result (output) by applying an algorithm to the function's arguments (input) and the current state. Each argument is either `in`, `out` or `inout`. If none of these qualifier is specified the argument is always assumed to be `in`. All input and output values are addressed by-copy, i.e. the functions works with local copies and thus aliasing problem are avoided. A shader may write to `in`-parameters since that will modify only the local copy. The `const` keyword prevents the shader writer from such a modification of `in` arguments but the qualifier cannot be used for `inout` or `out` for obvious reasons. All of the following functions are essentially the same, only the last two differ slightly by their calling conventions:

```
vec4 diffuse(vec4 N, vec4 L, vec4 C)
{
    C = C*max(0,dot(N,normalize(L)));
    return C;
}

vec4 diffuse(in vec4 N, in vec4 L,
             in vec4 C)
{ ... } // same code as above

vec4 diffuse(const in vec4 N,
             const in vec4 L,
             in vec4 C)
{ ... } // same code as above

void diffuse(in vec4 N, in vec4 L,
             in vec4 C,
             out vec4 result)
{
    result = C*max(0,
               dot(N,normalize(L)));
}

void diffuse(in vec4 N, in vec4 L,
             inout vec4 C)
{
    C = C*max(0, dot(N,normalize(L)));
}
```

All predefined functions can be subdivided into three groups:

I. The first group cannot be emulated by a shader since they map to some hardware functionality such as texture mapping.

II. Another group represents functions achieving a high performance gain when implemented in hardware such as trigonometric operations.

III. The third and last group are supported for convenience and are likely to perform trivial tasks such as clamping. The programmer should call predefined functions as often as possible since they can be translated to optimal code or even map one-to-one to a hardware instruction.

Overloading of functions is not supported, hence two functions known under the same name must differ by at least one argument. Overwriting a built-in function is possible but not advisable.

The language disallows direct or indirect recursion yet. This feature may be added in later revisions of GLSL if proved to be necessary.

4.6 Control Structures

Five fundamental building blocks are available in GLSL:

- statements and declarations
- function definitions
- selection (`if-else`)
- iteration (`for`, `while`, `do-while`)
- jumps (`return`, `break`, `continue`, `discard`)

The grammar defines a shader as a sequence of declarations and functions bodies. Each function in turn consists of statements which may be selections, iterations or jumps.

Each statements is delimited by a semicolon. In addition, statements can be grouped by curled braces into compound statements.

All these language feature comply to the well-known syntax of C. The only addition is the `discard` keyword that allows a fragment shader to drop a single fragment.

Later releases of GLSL may add the `switch` statement which is missed in the initial version since it is desirable to support floating-point variables but yet still unclear how implement that feature efficiently. By now, the `switch` statement has to be emulated by repeated `if` statements.

4.7 Preprocessor

Many of the core features of C can be found in GLSL, too. A preprocessor handles almost all of the commonly used C pragmas such as #ifdef and #error. A few predefined macros, like __LINE__, are also expanded.

```
/* a very long comment
that covers several lines */

// a short annotation
```

Pragmas are used to give the compiler hints about debug and optimization issues, too. Yet there is no standardized set of these compiler pragmas so their use may lead to non-portable shaders.

Comments are defined according to the C++ standard, which means that they cannot be nested, too. The example below give a demonstration of both comment styles.

5 USING SHADERS

5.1 Integration into the OpenGL API

Objects represent an generalized OpenGL structure containing a state and some associated data. A handle that is shareable across context boundaries references them. Prior to the GLSL, two kinds of objects existed: texture objects and display lists.

A special kind of objects are *shader objects* that encapsulate the source code of shaders. A single shader may consist of many shader objects and is either a vertex shader or a fragment shader. For example, some shader objects contain handy functions repeated called from another shader object and thus serve as a library. No more than one vertex shader object and one fragment shader object has to provide the `main` function which serves as an entry point like in C.

Program objects aggregate all shader objects necessary to form a shader. Technically spoken, all required shader objects have to be attached to a program object. These program objects may be added to the current OpenGL context in order to be activated. Only one program object is active at a time.

Compared to the C development model, one can think of shader objects as source code being compileable while program objects refer to the step of linking and generating an actually executable component.

The following code excerpt demonstrates how to setup a plain program object consisting of a vertex and a fragment shader:

```
GLhandleARB vs, fs, program;

// create vertex shader object
vs = glCreateShaderObjectARB
       (GL_VERTEX_SHADER_ARB);
glShaderSourceARB(vs,1,&vsSource,NULL);
glCompileShaderARB(vs);

// create fragment shader object
fs = glCreateShaderObjectARB
       (GL_FRAGMENT_SHADER_ARB);
glShaderSourceARB(fs,1,&fsSource,NULL);
glCompileShaderARB(fs);

// create program object
program = glCreateProgramObjectARB();

/* attach both shader object
   to the program object */
glAttachObjectARB(program, vs);
glAttachObjectARB(program, fs);

// link program object
glLinkProgramARB(program);
glUseProgramObjectARB(program);
```

Most shaders can be extensively configured by various parameters. Passing these *uniforms* to the program objects works via the new API extensions, too. After receiving a handle by calling `glGetUniformLocationARB`, you can set the corresponding uniform by invoking `glUniform{1…4}{f,i,b}ARB`. There are specialized versions of this instruction available for pointers and matrices as well.

```
Glint hParamFloat,
      hParamVector,
      hParamMatrix;

// get locations of parameters
hParamFloat =
 glGetUniformLocationARB(program, “f”);
hParamVector =
 glGetUniformLocationARB(program, “v”);
hParamMatrix =
 glGetUniformLocationARB(program, “m”);

// set parameters
glUniform1fARB(hParamFloat, 0.42);
glUniform3fARB(hParamVector,
               0.5, 1.0, 0.0);
glUniformMatrix3fvARB(hParamMatrix,
               9, false, pMatrix);
```

Calling `glDeleteObjectARB` frees all resources associated to an object, no matter whether it is shader of program object.

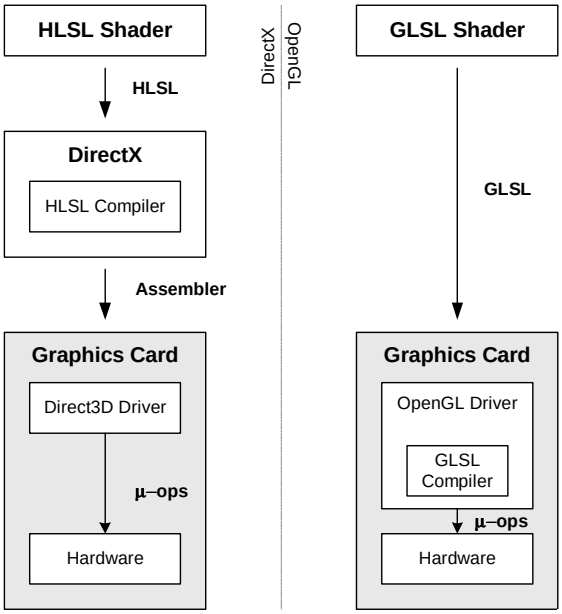

Figure 4. Shader compilation of HLSL and GLSL

Since setting up and enabling a shader modifies the current OpenGL context it cannot be done between `glBegin` and `glEnd`.

6 CG AND HLSL

NVIDIA designed Cg with the intention in mind to provide a very abstract and extensible shading language based on C (therefore Cg is an abbreviation of "C for Graphics"). Cg is available for both major 3D graphics APIs OpenGL and DirectX hugely increasing the possible application scenarios of shaders following the philosophy "write once, run anywhere". It introduced the profile concepts that gives the compiler some hints how to optimize the generated code depending on hardware capabilities and system features.

Cg is built around the same vertex shader / fragment shader separation as the OpenGL Shading Language but gives the developer the opportunity to mix assembler and high-level shaders arbitrarily. In consequence, Cg requires an explicit binding of attributes, uniforms and varyings allowing more flexibility at the cost of additional API calls.

HLSL shares lots of concepts and features with Cg since both were developed in close cooperation. It does compile the shaders outside the graphic card's driver, too. Therefore the application is aware of the actually executed assembler code which is not the case and even impossible for the OpenGL Shading Language.

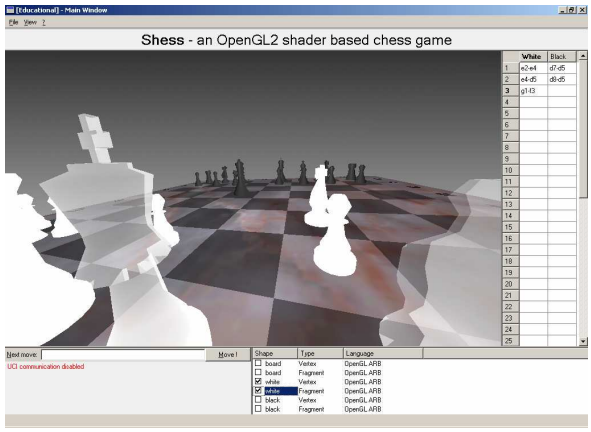

Figure 5. Screenshot of Shess

Because HLSL and Cg come with a compiler already written it is easier for a hardware vendor to deliver drivers for both languages. The long delay for stable GLSL drivers is mainly caused by insufficiently working GLSL compilers and not by hardware limitations (figure 4).

7 EXPERIENCES

Shess, a shader based chess game supports three kinds of shaders available for OpenGL: ARB assembler, NVIDIA's Cg and GLSL (see figure 5). All three languages follow similar strategies of integrating themselves in the OpenGL environment. They offer extensions that may be invoked at any time (except within `glBegin` and `glEnd` blocks) but not all graphics accelerators fully support them.

I often missed a proper debug mechanism. Although GLSL by default provides access to its error log via the `glGetInfoLogARB` and `glValidateProgramObjectARB` interfaces, the returned messages were not always clear and sometimes even misleading.

To bypass driver insufficiencies, I had to write short shaders consisting of basic operations. The drivers had no problems to optimize them properly and there was no noticeable loss of performance compared to ARB assembler. Unfortunately, it is not possible to retrieve the assembler code generated by the GLSL compiler which would be interesting to examine. However, the code may differ depending on the actual hardware and the installed graphics driver.

8 CONCLUSION

The OpenGL Shading Language proved to be quite effective and comfortable in its application. The effort to integrate it in modern graphics engine is low and does not require substantial changes of existing designs.

The insufficient support of GLSL by the major graphics hardware vendors defers a satisfactory usage in current applications. It is to expect that NVIDIA and ATI will officially release adequate driver in early 2004 opening the consumer market for GLSL. A beta version of GLSL for ATI R300 chips is yet available but still very instable. Even ATI's offline GLSL test suite called Ashli cannot handle all well-formed GLSL shaders.

The latest computer games tend to heavily utilize shaders to achieve various effects and improve the overall rendering performance. Up to now, none of the major game engines, like Unreal Warfare or Doom 3, directly supports GLSL. On the other side, the importance of DirectX 9 compliance in the field of marketing led to a market where nearly all hardware vendors support HLSL. Cg does not play an important role as of today and there are rumors that Cg may be obsolete in the near future.

There are severe compiler and hardware improvements required to bring shaders' performance closer to the goal of providing an interactive real-time experience. Especially expensive fragment operations known from high quality images rendered by programs like Maya are still unrealistic today.

9 ACKNOWLEDGMENTS

Marc Nienhaus, PhD candidate at the Hasso-Plattner-Institute, supervised and reviewed this paper as well as the accompanied presentation. Florian Kirsch, a PhD candidate at the same institute, shared his experiences gathered in writing basic GLSL samples and integrating them in VRS.

VRS, the Virtual Rendering System, has been developed at the Chair of Computer Graphics Systems headed by Prof Jürgen Döllner at the Hasso-Plattner-Institute.

10 REFERENCES

[3Dlabs02] *OpenGL Shading Language White Paper*, version 1.2, 2002.

[Akenine-Möller02] Tomas Akenine-Möller and Eric Haines, *Real-time Rendering, 2nd edition*, A. K. Peters, 2002.

[Alias] Maya 5 by Alias Systems Inc., http://www.alias.com

[ATI] ATI, http://www.ati.com

[Cook84] Robert L. Cook, *Shade Trees*, SIGGRAPH 1984, Minneapolis.

[Cook87] Robert L. Cook, Loren Carpenter and Edwin Catmull, *The Reyes Image Rendering Architecture*, SIGGRAPH 1987, Anaheim.

[Deering95] Michael Deering, *Geometry Compression*, SIGGRAPH 1995, Los Angeles.

[Engel02] Wolfgang F. Engel, ed., *Direct3d ShaderX*, Wordware, 2002.

[Fernando03] R. Fernando and Mark J. Kilgard, *The Cg Tutorial: The definitive guide to programmable real-time graphics*, Addison-Wesley, 2003.

[Gray03] Kris Gray, *Microsoft DirectX 9 Programmable Graphics Pipeline*, Microsoft Press, 2003.

[Hanrahan90] Pat Hanrahan and Jim Lawson, *A language for shading and lighting calculations*, SIGGRAPH 1990, Dallas.

[Intel] Intel Corporation, http://www.intel.com

[Jaquays99] Paul Jaquays and Brian Hook, *Quake 3: Arena Shader Manual*, Revision 12, 1999.

[Kernighan88] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice Hall, 1988.

[Krüger03] Jens Krüger and Rüdiger Westermann, *Linear Algebra Operators for GPU Implementation of Numerical Algorithms*, SIGGRAPH 2003, San Diego.

[McCool01] Michael D. McCool, Jason Ang and Anis Amad, *Homomorphic Factorizations of BRDFs for High-Performance Computing*, SIGGRAPH 2001, Los Angeles.

[Microsoft02a] Microsoft Corporation, *DirectX 9.0 graphics*, http://msdn.microsoft.com/directx

[Microsoft02b] Microsoft Corporation, *High-level shader language*. In: [Microsoft02a].

[NVIDIA] NVIDIA Corporation, http://www.nvidia.com

[NVIDIA01a] NVIDIA Corporation and Square, Final Fantasy Technology Demo, SIGGRAPH 2001, Los Angeles, http://www.nvidia.com/object/final_fantasy.htm

[NVIDIA01b] Chris Maughan and Matthias Wloka, *Vertex Shader Introduction*, NVIDIA White Paper, 2001.

[NVIDIA03a] NVIDIA Corporation. *Cg toolkit*, Release 1.1, http://developer.nvidia.com/Cg

[NVIDIA03b] William R. Mark, R. Steven Glanville, Kurt Akeley, Mark J. Kilgard, Cg: *A system for programming graphics hardware in a C-like language*, SIGGRAPH 2003, San Diego.

[Olano95] Marc Olano, Anselmo Lastra, Steven Molnar and Yulan Wang, *Real-time Programmable Shading*, Symposium on Interactive 3D Graphics, 1995.

[Olano98] Marc Olano, Anselmo Lastra, *A Shading Language on Graphics Hardware: The PixelFlow Shading System*, SIGGRAPH 98, Orlando.

[OpenGL02] Mark Segal and Kurt Akeley, *The OpenGL Graphics System: A Specification*, version 1.4, OpenGL Architecture Review Board, 2002, http://www.opengl.org

[OpenGL03a] Mark Segal and Kurt Akeley, *The OpenGL Graphics System: A Specification*, version 1.5, OpenGL Architecture Review Board, 2003.

[OpenGL03b] John Kessenich, Dave Baldwin and Randi Rost, *The OpenGL Shading Language*, version 1.05, 2003.

[Peercy00] Mark S. Peercy, Marc Olano, John Airey and P. Jeffrey Ungar, *Interactive Multi-Pass Programmable Shading*, SIGGRAPH 2000, New Orleans.

[Perlin85] Ken Perlin, *An image synthesizer*, SIGGRAPH 1985, San Francisco.

[Pixar] Pixar Inc., http://www.pixar.com

[Proudfoot01] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov and Pat Hanrahan, *A Real-Time Procedural Shading System for Programmable Graphics Hardware*, SIGGRAPH 2001, Los Angeles.

[Rost03] Randi Rost and Bill Licea-Kane, *The OpenGL Shading Language*, SIGGRAPH 2003, San Diego.

[Stroustrup00] Bjarne Stroustrup, *The C++ Programming Language*, 3rd ed, Addison-Wesley, 2000.

11 APPENDIX

11.1 Today's GLSL Availability

The OpenGL Shading Language has been designed in early 2001 by 3Dlabs [3Dlabs02] who eagerly push OpenGL 2.0, too. Until now, 3Dlabs remains the only vendor actively promoting and supporting GLSL.

Since OpenGL 2.0 has not been published, GLSL was accepted as an ARB extension known as `ARB_shading_language_100` for OpenGL 1.5 to underline the importance of GLSL. Recent drivers of ATI also expose this extension and one should expect the same for NVIDIA within a few months.

ATI published for free a neat tool called Ashli (Advanced SHading Language Interface, figure 6) translating Renderman and GLSL shaders to native OpenGL ARB or DirectX 9 assembler. Almost all modern graphics accelerators are able to execute these assembler shaders. Not all features are supported by Ashli, though, one gains remarkable insights into the general ideas and concepts behind these two languages. A basic optimizer helps to achieve real-time performance in most cases, which is interesting especially for Renderman shaders as there not designed to run in real-time by default.

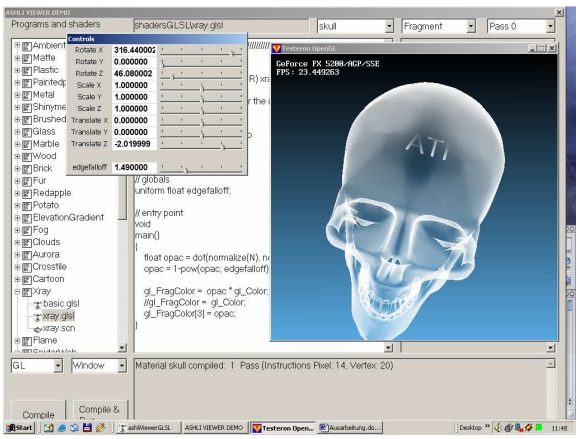

Figure 6. Ashli GLSL test suite [ATI]

11.2 Exemplary Shader

11.2.1 Vertex Shader

The following vertex shader computed the model-view projection formerly done by the fixed functionality pipeline of OpenGL. It does the same for the normals and the incident vector.

```
// modified vertex shader of Ashli

// vertex to fragment shader io
varying vec3 N;
varying vec4 I;

void main()
{
  // position in eye space
  P = gl_ModelViewMatrix * gl_Vertex;

  // position in clip space
  gl_Position =
     gl_ModelViewProjectionMatrix *
     gl_Vertex;

  // normal transform
  N = gl_NormalMatrix * gl_Normal;

  // incident vector
  I = P - gl_ModelViewMatrix[3];
}
```

11.2.2 Fragment Shader

The fragment uses the normal to generate a color and sets a random alpha value (generated by Perlin noise). The resulting image is shown in figure 7.

```
// vertex to fragment shader io
varying vec3 N;
varying vec4 I;

// entry point
void main()
{
    gl_FragColor = N;
    gl_FragColor[3] = noise1(N);
}
```

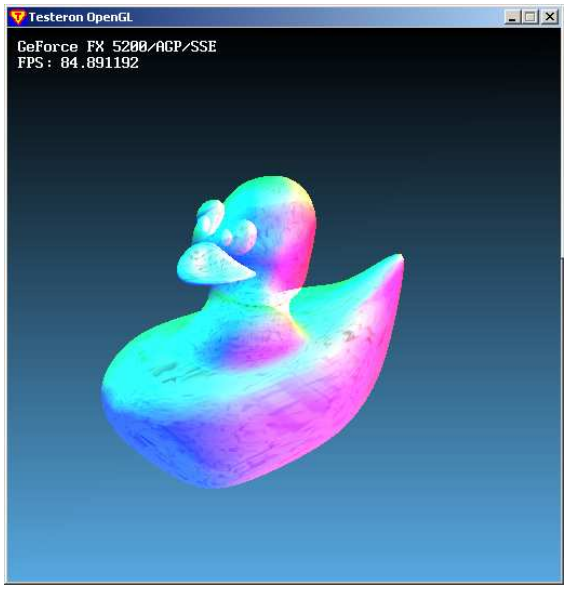

Figure 7. Running an exemplary shader