Cluster Comput
DOI 10.1007/s10586-017-1113-z

# Prototype implementation of the OpenGL ES 2.0 shading language offline compiler

**Nakhoon Baek**[1,2,3] · **Kuinam J. Kim**[4]

Received: 3 May 2017 / Revised: 20 July 2017 / Accepted: 10 August 2017

**Abstract** Latest 3D graphics libraries, including OpenGL and OpenGL ES, provide programmable rendering pipelines. These programmable pipelines naturally requires special-purpose programming languages. It is so called *OpenGL Shading Language* in the case of OpenGL. In this paper, we show a new compact design scheme for the implementation of OpenGL SL offline compilers. This scheme is used for our OpenGL SL offline compiler prototype implementation. Our implementation shows feasibility and correctness with respect to the standard specification.

**Keywords** OpenGL ES · Shading language · Offline compiler · Design · Prototype implementation

## 1 Introduction

*OpenGL* is one of the most widely used 3D graphics library [10,14]. It has a variation for embedded systems, named *OpenGL ES* [8,12,13]. Originally, OpenGL and OpenGL ES was designed to be implemented into fixed function VLSI chips.

Since OpenGL version 2.0 [16] and OpenGL ES version 2.0 [12], they introduced the new concept of *programmable* VLSI chips for the 3D graphics library. Thus, it was realized as *OpenGL Shading Language* [5,6], or shortly, OpenGL SL. This special-purpose high-level programming language is similar to the C programming language [4] with some pre-defined data types.

Since they introduce a new programming language, they also designed the compiling process for this programming language. OpenGL and OpenGL ES is basically a set of API functions. Thus, OpenGL Shading Language compilers are also provided as several number of API functions. They call them as the *online* shading language compilers.

In this paper, we present an implementation scheme for the *offline* compiler of the OpenGL ES 2.0 Shading Language [17,18]. This offline compiler will accept the full specification of the OpenGL ES 2.0 Shading Language and generates the GPU execution codes for some GPU's. We will show the details of requirements and implementation schemes in the following sections.

## 2 Shader language compiler details

A bit differently from the typical programming language compilation process [1,9], the OpenGL Shading Language source codes are usually compiled during the execution of OpenGL application program, with specially designed API functions [5]. In this section, we will represent the details and characteristics of the OpenGL Shader Language compilation process, including the API functions, offline compilation, and binary format supports.

✉ Kuinam J. Kim
kuinamj@gmail.com

Nakhoon Baek
oceancru@gmail.com

1 School of Computer Science and Engineering, Kyungpook National University, Daegu 41566, Republic of Korea

2 Software Technology Research Center, Kyungpook National University, Daegu 41566, Republic of Korea

3 dassomey.com Inc., Daegu 41566, Republic of Korea

4 Department of Convergence Security, Kyunggi University, Suwon-si 16227, Republic of Korea

Published online: 30 August 2017

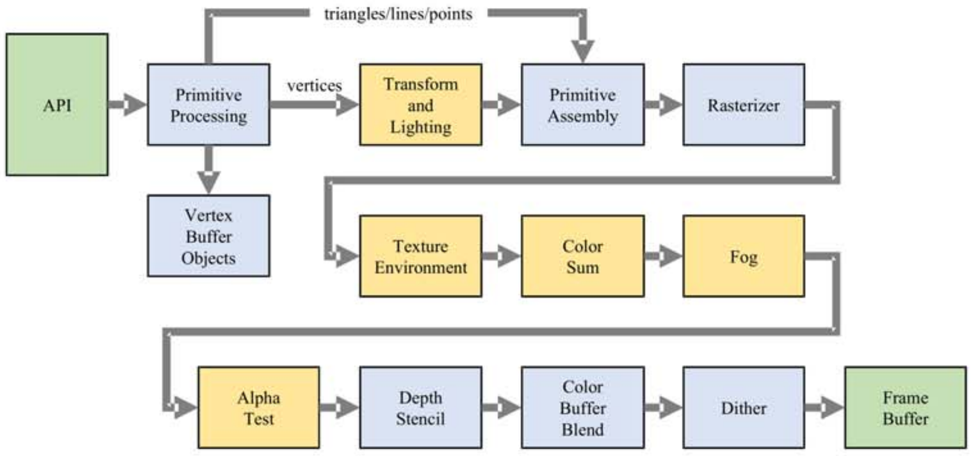

**Fig. 1** The fixed function graphics pipeline. *Source* Courtesy of Khronos

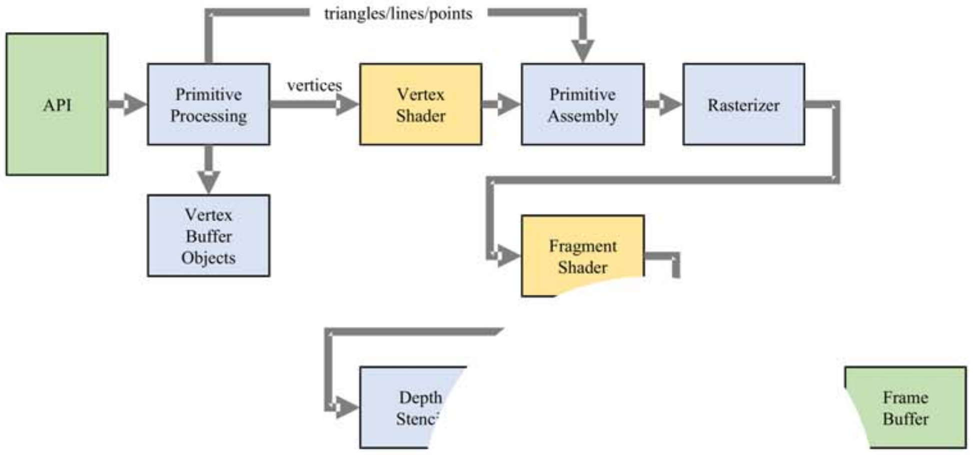

**Fig. 2** The programmable graphics pipeline. *Source* Courtesy of Khronos

## 2.1 Shader language API functions

Before OpenGL 2.0, the OpenGL graphics pipeline was typically realized with VLSI chips, and the internal processing can be presented as the fixed function modules [13,15,19], as shown in Fig. 1.

With OpenGL 2.0 [16] and OpenGL ES 2.0 [12], they introduced programmable shaders into the OpenGL graphics pipeline, as shown in Fig. 2. Those shaders are small-size special-purpose executable programs to be executed with GPU's [2,12,16]. The shader programs are written in a specially-designed programming language, the OpenGL Shading Language [2,5,17], for ease of programming with GPU's.

According to the OpenGL specifications, OpenGL ES 2.0 introduced the concept of program objects. A typical program object consists of two shader programs: *vertex shader* and *fragment shader*. Thus, a pair of shader programs are compiled, linked, and finally executed as a single program object, as shown in Fig. 3.

Thus, typical OpenGL application programs contain the source codes for the shader programs, and perform the whole compilation process of compiling, linking, and execution with the standard OpenGL API functions. We will explain these API functions and its details in the followings.

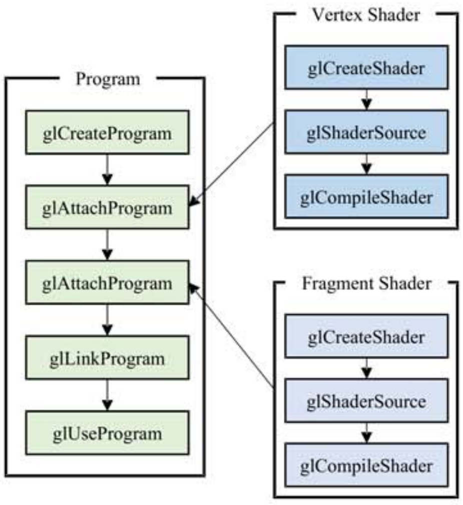

**Fig. 3** Shader program compilation process. *Source* Courtesy of Khronos Group

A typical program object consists of two shader objects: *vertex shader* and *fragment shader*. These two shader objects shares many common features. A few minor differences are handled in the shader language, rather than the API functions. Thus, they use the same set of API functions, to manage two shader objects.

As the first step, application programmers will create a shader object:

**GLuint** glCreateShader( **GLenum** *type* );

where the parameter *type* will select the shader types of GL_VERTEX_SHADER and GL_FRAGMENT_SHADER. This function returns the unique identifier for the created shader object.

To provide the shading language source codes, we use the following API function:

**void** glShaderSource( **GLuint** *shader*,
**GLsizei** *count*, **const GLchar**** *string*,
**const GLint*** *length* );

where *shader* is the unique identifier of the existing shader object, and *count* is the number of character strings specified by the string pointer of *string* and the string length of *length*. Now, the source codes are provided to the shader objects. With the given source code, shader object performs the compilation process with:

**void** glCompileShader( **GLuint** *shader* );

where *shader* is the identifier of the shader object.

In the case of program objects, they need another set of API functions to handle them. The successfully compiled shader objects can be linked into a program. We can start the program object handling by creating it:

**GLuint** glCreateProgram( **void** );

This function returns the unique identifier for the created program object.

After creating a program object, the shader objects should be attached to the program. It can be achieved with the following function:

**void** glAttachShader( **GLuint** *program*,
**GLuint** *shader* );

where *program* and *shader* are the unique identifiers for the program object and the shader object, respectively.

After attaching the two shader objects for the vertex shader and the fragment shader, they will be linked to construct a single program object with the function:

**void** glLinkProgram( **GLuint** *program* );

where program is the unique identifier for the program object.

The linked program object is installed as part of the current rendering state with the function:

**void** glUseProgram( **GLuint** *program* );

where program is the unique identifier for the program object.

### 2.2 Binary format support

In the case of OpenGL ES 2.0 and later, they provide *online* compilers for OpenGL Shading Languages. As an alternative option, they also provide a binary image interface, with the following API function:

**void** glProgramBinary( **GLuint** *program*,
**GLenum** *binaryFormat*, **const void*** *binary*,
**GLsizei** *length* );

where they use the following parameters:

- *program* specifies the name of a program object into which to load a program binary.
- *binaryFormat* specifies the format of the binary data in binary.
- *binary* specifies the address of an array containing the binary image to be loaded into the program.
- *length* specifies the number of bytes contained in binary.

The binary images read with the above glProgramBinary function are converted into the executable programs on the GPU. Internally, the binary image will be split into vertex shader and fragment shader programs. They are then downloaded on the GPU and executed.

## 3 Design and implementation of shading language compilers

Our focus is providing the shading language compilation features. The overall design and prototype implementations are represented in the following sections.

### 3.1 Overall design

To implement OpenGL SC 2.0 Shading Language features, we need an offline shading language compiler. Fortunately, there are some open-source shading language compilers [3,11]. Based on an existing OpenGL Shading Language compiler, we will generate the compiled binary images, all the uniform variable locations, and other data information, as shown in Fig. 4.

Using our own OpenGL SC 2.0 specific binary image file format for the compiled result, we will transfer all these information to the OpenGL SC 2.0 execution core. Specifically, glProgramBinary function will decode all the information and spread them to the GPU's and OpenGL state variables.

Based on these implementation, we can provided the OpenGL SC 2.0 Shading Language features, as specified in the standard specification. Additionally, through these design scheme, we can construct the compiling system incremen-

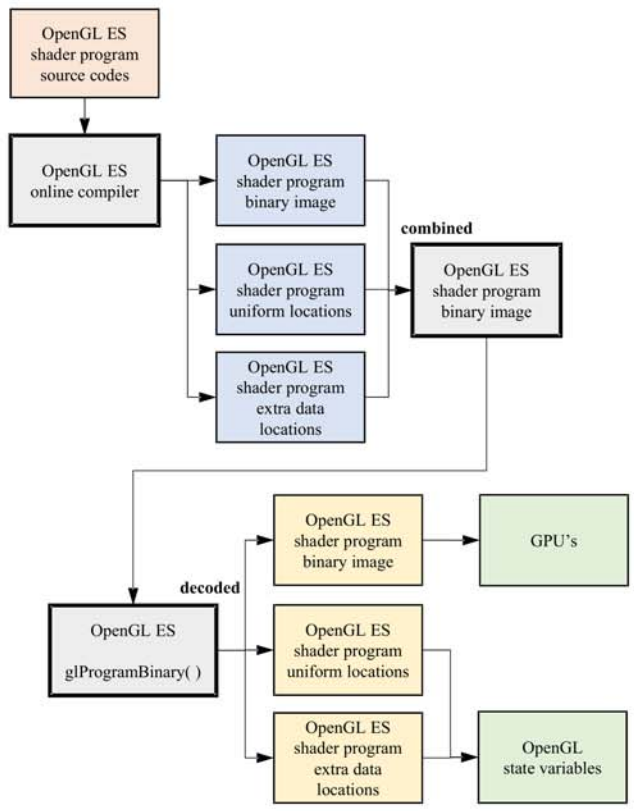
OpenGL ES shader program source codes
OpenGL ES online compiler
OpenGL ES shader program binary image
OpenGL ES shader program uniform locations
OpenGL ES shader program extra data locations
combined
OpenGL ES shader program binary image
decoded
OpenGL ES glProgramBinary( )
OpenGL ES shader program binary image
GPU's
OpenGL ES shader program uniform locations
OpenGL ES shader program extra data locations
OpenGL state variables

**Fig. 4** Implementation strategy for the OpenGL ES 2.0 Shading Language feature

tally. We can minimize any potential problems, through fully testing and debugging the corresponding modules.

## 4 Implementation strategy

Since we can use the previous OpenGL Shading Language online compilers, it can be a kind of the reference implementation. Our strategy is starting from this online compiler implementation, we substitute each compiling stage to our own implementation, in a step-by-step manner, as shown in the followings.

- **Implementation stage 0**: Use the reference online compiler. From the shader source codes, use the online compiler and use glGetProgramBinary function to get the binary images. The binary images are stored on the main memory, and used by the same program, to be executed. This is the simplest way of feasibility test for the binary images of the shader compiler.
- **Implementation stage 1**: Reuse the saved binary image. Using the glGetProgramBinary function, the compiled binary images are stored into the hard-disk files, to be retrieved later. The binary images are loaded later by another program, with the glProgramBinary function. Testing these steps, we can test the feasibility of the compiled binary images.
- **Implementation stage 2**: Implementing the front-end with Tungsten intermediate codes. We will implement a shader compiler front-end, which converts the vertex shader source codes and the fragment shader source codes to the Tungsten intermediate codes. These intermediate code images are loaded and tested with the OpenGL online compiler. Actually, these steps are not the standard way of OpenGL specification. However, some commercial implementations, including the Mesa 3D graphics library [11], allow the use of these intermediate code images. This is a step-by-step way of testing our compiler frond-end.
- **Implementation stage 3**: Full-scale offline compiler with the back-end GPU code generators. As the next step, we will implement the back-end GPU code generators. Combining the front-end implementation and these back-end code generator, we will get the full-scale offline compiler for the OpenGL Shading Languages. Since we can implement a specific code generator for each GPU model, we can have a set of GPU code generators, to construct a full set of OpenGL Shading Language offline compiling system.
- **Implementation stage 4**: Optional code optimizer. In some cases, we can also implement the code optimizer for the Tungsten intermediate code and/or GPU-specific codes.

Based on this step-by-step implementation strategy, we can construct the compiling system incrementally. We can minimize any potential problems, through fully testing and debugging the corresponding module, in each implementation stage.

## 5 Implementation results

OpenGL ES 2.0 needs an internal shader language compiler, for its OpenGL ES Shader Language (ESSL) [17]. Our preprocessor handles all the ESSL pre-processor directives, including **#define**, **#pragma**, **#version**, and others. The preprocessed shader language codes are then processed for the strict syntax and semantics checks, with respect to the OpenGL ES 2.0 standard specification [12].

We also added semantics check routines for some Ada-like flavors of ESSL. We used the OpenGL ES 2.0 Conformance test suites(CTS) [7] for its conformability to the standard specification. Our current implementation shows about 97% of the success ratio. We also tested various OpenGL ES 2.0 application programs and a set of benchmark programs, as shown in Figs. 5, 6 and 7. At this time, our implementation works with all of these real-world applications.

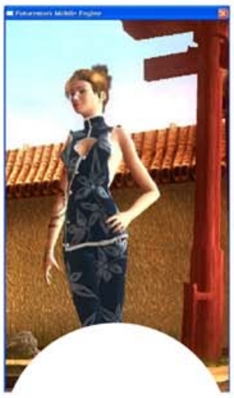
**Fig. 5** A screen shot from the "Taeji" test program. *Source* Courtesy of Rightware

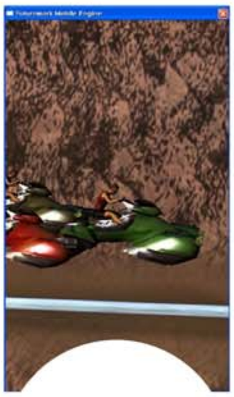
**Fig. 6** A screen shot from the "Hoverjet" test program. *Source* Courtesy of Rightware

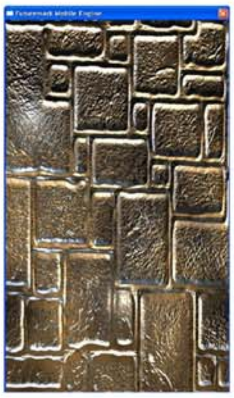
**Fig. 7** A screen shot from the "Advanced World" test program. *Source* Courtesy of Rightware

## 6 Conclusion

In this paper, we aimed to implement an offline compiler for OpenGL Shading Languages. Though we have some OpenGL Shading Language online compilers, the offline compilers are now needed for OpenGL SC 2.0 standard. We represents the related technical details for the offline compiler.

We showed the details of implementation steps for the OpenGL Shading Language offline compiler. Based on the previous reference implementations, we can build up the full scale OpenGL Shading Language compiling system.

**Acknowledgements** This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (Grant 2016R1D1A3B03935488).

## References

1. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison Wesley, Boston (2006)
2. Fabius, A., Viggers, S.: OpenGL SC Version 2.0.0. Khronos Group (2016)
3. Group, K.: OpenGL / OpenGL ES Reference Compiler (2017)
4. ISO/IEC 9899: Programming Languages – C. International Standard Organization (1998)
5. Kessenich, J.: The OpenGL Shading Language, Language Version: 1.20. Khronos Group (2006)
6. Kessenich, J.: The OpenGL Shading Language, Language Version: 4.50. Khronos Group (2016)
7. Khronos Group Conformance Test Suites: http://www.khronos.org/adopters/
8. Lipchak, B.: OpenGL ES Version 3.2. Khronos Group (2016)
9. Louden, K.C.: Compiler Construction: Principles and Practice. Course Technology (1997)
10. Malizia, A.: Mobile 3D Graphics. Springer-Verlag, New York (2006)
11. Mesa3D.org: The Mesa 3D Graphics Library (2017)
12. Munshi, A.: OpenGL ES Common Profile Specification, Version 2.0.24 (Full Specification). Khronos Group (2009)
13. Munshi, A., Leech, J.: OpenGL ES Common/Common-Lite Profile Specification, Version 1.1.12 (Full Specification). Khronos Group (2008)
14. Pulli, K., Vaarala, J., Miettinen, V., Aarnio, T., Roimela, K.: Mobile 3D Graphics: with OpenGL ES and M3G. Morgan Kaufmann Publishers Inc., San Francisco, CA (2007)
15. Segal, M., Akeley, K.: The OpenGL Graphics System: A Specification, Version 1.3. Khronos Group (2001)
16. Segal, M., Akeley, K.: The OpenGL Graphics System: A Specification, Version 2.1. Khronos Group (2006)
17. Simpson, R.J.: The OpenGL ES Shading Language, Language Version: 1.00. Khronos Group (2008)
18. Simpson, R.J.: The OpenGL ES Shading Language, Language Version: 3.2[illegible] [illegible]hronos Group (2016)
19. St[illegible] [illegible]GL SC: Safety-Critical Profile Specification, [illegible] Specification). Khronos Group (2009)

**Nakhoon Baek** is currently a professor in the School of Computer Science and Engineering at Kyungpook National University, Korea. He received his B.A., M.S., and Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 1990, 1992, and 1997, respectively. His research interests include graphics standards, graphics algorithms and real-time rendering. He is now also the Chief Engineer of Dassomey.com Inc., Korea.

**Kuinam J. Kim** received the B.S. degree from Mathematics, University of Kansas in 1989. He received the M.S. degree of Statistics and Ph.D. degree of Industrial Engineering from Colorado State University. He is currently Professor of Industrial Security Department, Kyonggi University, Korea. His research interests include industrial security.