ISSN 2347 - 3983
Volume 8. No. 9, September 2020
International Journal of Emerging Trends in Engineering Research
Available Online at http://www.warse.org/IJETER/static/pdf/file/ijeter165892020.pdf
https://doi.org/10.30534/ijeter/2020/165892020

A Shader Language Off-line Cross-Compiler on the Desktops

Nakhoon Baek
School of Computer Science and Engineering, Kyungpook National University,
Daegu 41566, Republic of Korea,
oceancru@gmail.com

ABSTRACT

In our modern computer graphics systems, they use GPUs (graphics processing units) to execute the low-level parallel instructions, compiled from the shader language source codes.
Recently, some graphics system specifications, including OpenGL SC 2.0, introduce the off-line compilation. With this new scheme, the independent off-line compiler will generate the final binary executable codes. Typically, the target system will execute the pre-compiled binary executable codes, at some time later. In this paper, we analyzed the detailed requirements on the shader language off-line compilation. Based on these analysis, we designed the whole compiling process and the binary file format for the off-line compiler. As the verification process of our design, we implemented the prototype off-line compiler for our existing OpenGL SC 2.0 implementation on the target embedded system. We executed a set of pre-compiled binary files on the target system, and all the results are the same to the desktop executions. We finally showed that the shader language off-line compilers are possible with our proposed schemes.

Key words : shader language, off-line compiler, cross-compiler, prototype implementation

1. INTRODUCTION

In these days, we have many remarkable changes in many computer-oriented fields, along to the development of modern computer graphics and its related parallel execution technology. The parallel computations originated from the GPU (graphics processing unit) technology is one of the most important changes. Shader language concepts [10,14] are another renovations in the field of computer graphics.

At this time, among the 3D graphics Application Program Interfaces (APIs), the most widely used one is OpenGL (Open Graphics Library). It can be used for variety of platforms, including desktops, tablets, and smartphones. Traditionally, OpenGL is based on the fixed function hardware pipelines, and it provides the fixed API functions, as shown in Figure 1 [13,15,22]. From OpenGL version 2.0 and later, they introduce the new programmable graphics pipeline, as shown in Figure 2 [3,12,16].

The *Khronos Group*[7], which is the *de facto* standard organization, manages all the standard specifications for the OpenGL family including OpenGL [5,6,15,16,18], OpenGL ES (for embedded systems) [9,12,13,19,20], and OpenGL SC (safety critical profile) [3,22]. At this time, in our commercial markets, we have plenty of graphics hardware and also application programs, providing and using OpenGL and/or OpenGL ES facilities.

In contrast, OpenGL SC is actually derived from the OpenGL family, as a safety critical version. This new standard is designed to meet the needs for military, medical, automotive, avionics, and other industrial applications in the safety critical market. Actually, OpenGL SC plays an important role in the safety-critical market. It provides the 3D graphical interfaces. We have much increased needs for this 3D graphics standard, along to the growth of the safety-critical applications [2,21]. Especially, this standard has strong focus in the medical applications and automotive applications [1].

From the year of 2015, they started to upgrade this safety-critical standard, as a brand-new one. It targets the new safety-critical graphics standard for the avionics and automotive displays. As the final result, the OpenGL SC 2.0 specification [3] defines the new safety critical version of OpenGL ES 2.0 [12], as shown in Figure 3. Now they are working to adapt the new Vulkan API [4] for high-efficiency graphics applications and computing programs. The Safety Critical working group in the Khronos Group is also developing cross-API guide-lines, which is effective for the development of the safety critical systems and its related standards.

[1] This work has supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (Grand No.NRF-2019R1I1A3A01061310).

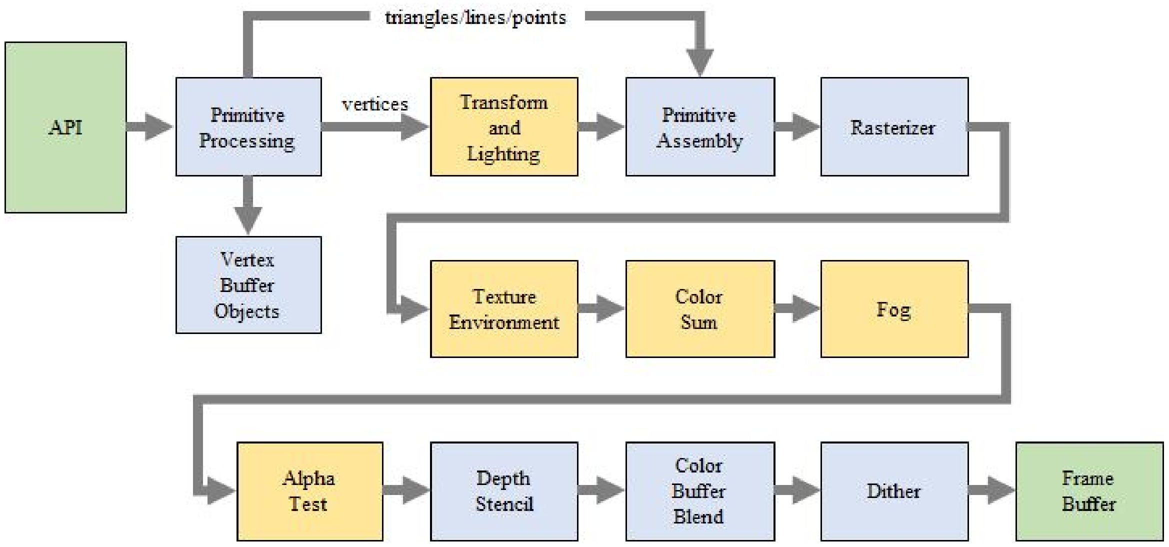

Figure 1: An example fixed function graphics pipeline of the OpenGL family.

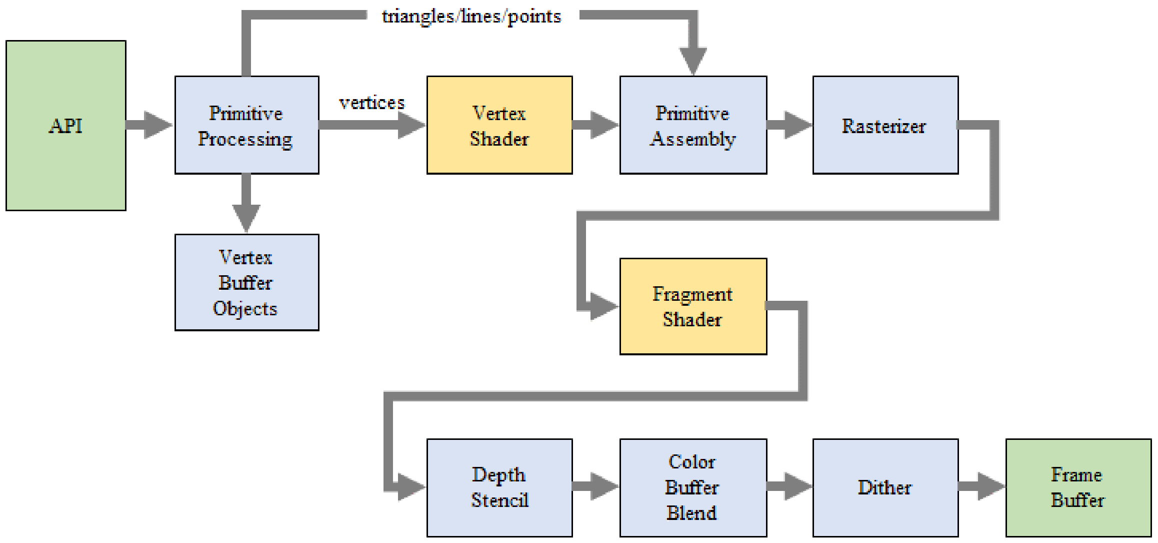

Figure 2: An example of the programmable graphics pipeline of the OpenGL family.

One of the most remarkable changes in the OpenGL SC 2.0 [3] is the introduction of off-line compilers in its shader language compilation process. Previously, some graphics vendors provide the off-line compiling features as the OpenGL extensions. In contrast, OpenGL SC 2.0 requires the off-line compiler as one of its core features. The importance of the offline compiler is more emphasized in these days.

In this paper, we show the design schemes for the OpenGL family shader language offline compiler. We assume that the system will be operated on the desk-tops, to generate a specific target GPU instructions, and those instructions will be executed on the independent embedded systems. The design details and implementation results are followed in the following sections.

2. DESIGN ASPECTS

In this section, we will show all the design aspects of our shader language off-line compiler. In each subsection, we will handle one of the technical issues in the whole design process.

2.1 Functional Interface changes

Along to the development of modern computer graphics hardware, the role of GPU becomes much important. From

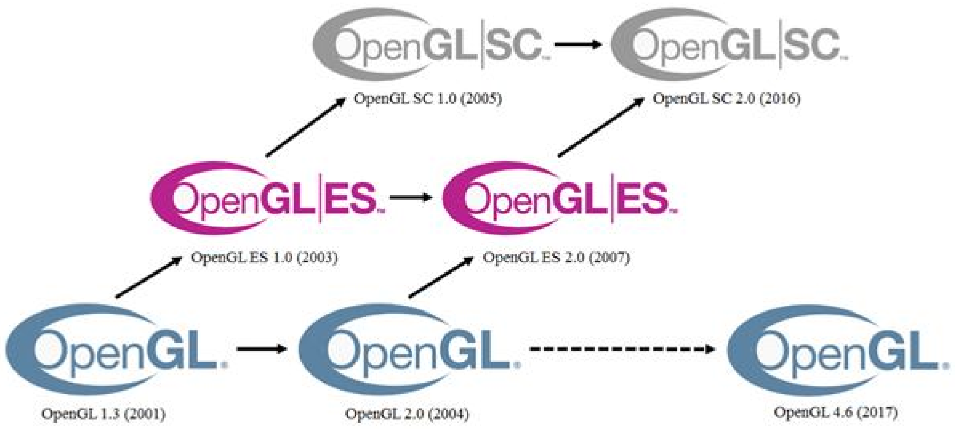

Figure 3: Historical chart for the OpenGL family.

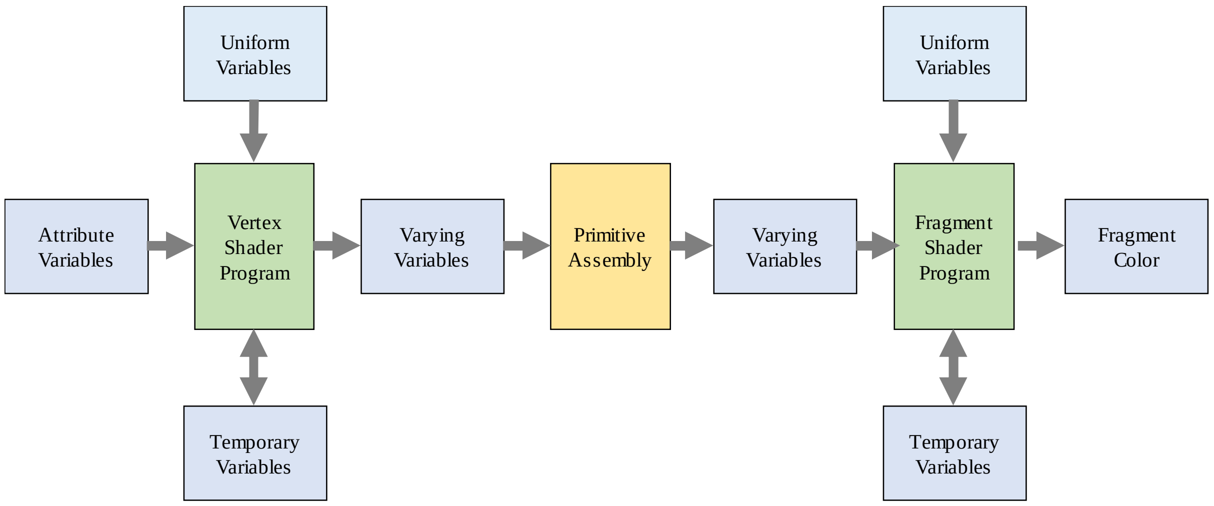

Figure 4: Architectural view of the OpenGL vertex shader and fragment shader.

the software developer's point of view, it is better to provide the direct access to the underlying GPUs. Thus, the newer versions of OpenGL, more precisely, OpenGL 2.0 [16] and later, provide the OpenGL shader language, its compiler, and the shader language handling functions. All these features are already provided by OpenGL [17], OpenGL ES [9], and also OpenGL SC [3].

One more important change to the OpenGL SC 2.0 [3] is the addition of off-line compilation features. In the case of desktop OpenGL, OpenGL version 4.1 has its corresponding off-line compilation features. The OpenGL SC 2.0 provides this feature based on the desktop version of OpenGL 4.1 or later.

To provide this off-line compilation feature, we started from the real-time interception of the compiled binary images, from the shader compiler. During the on-line compilation of a specific shader language file, the result of compiled binary images are intercepted to be saved to the external storage. This captured binary image is later reused by the API function of glProgramBinary. We have tested the implementation results with those of the previous MESA implementation [11], and confirmed that it works well.

In the original design of the OpenGL shading language, they have online compilation features. Thus, from the OpenGL ES 2.0 and later, the shader compilers provide the immediate compilation of the given shader source code. In the case of OpenGL SC 2.0, they provide an alternative way of off-line compilation. They provide a binary image interface, for the following function syntax:

void glProgramBinary(
   GLuint *program*,
   GLenum *format*,
   const void* *image*,
   GLsizei *length*);

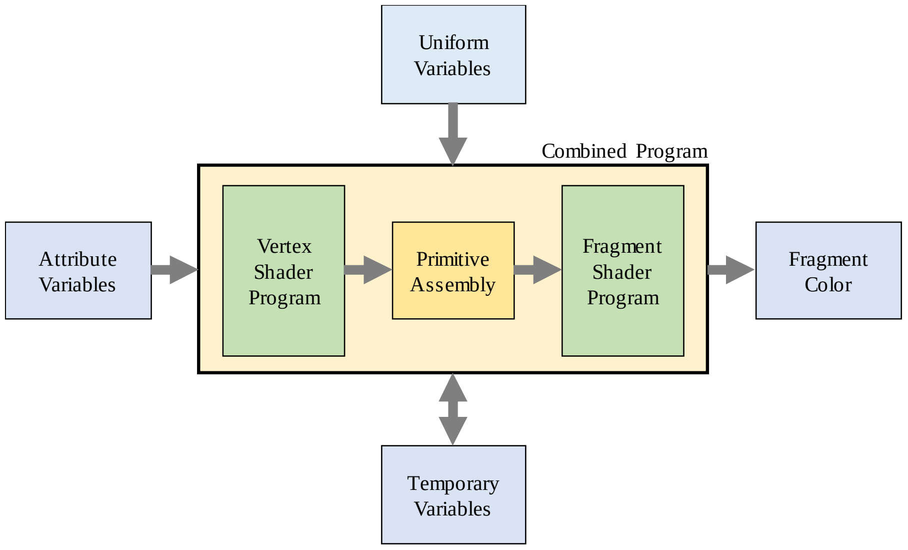

Figure 5: Conceptual view of the combined OpenGL shaders.

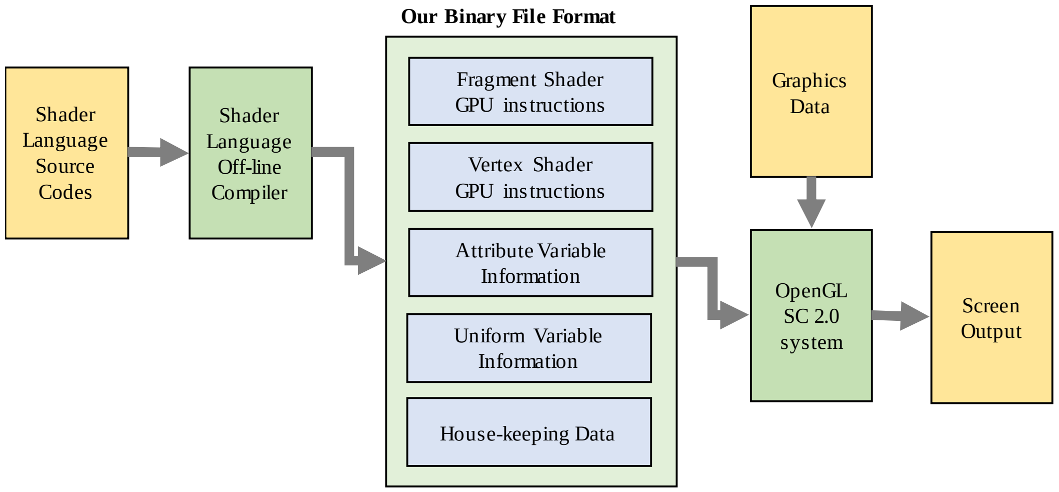
Our Binary File Format
Shader Language Source Codes
Shader Language Off-line Compiler
Fragment Shader GPU instructions
Vertex Shader GPU instructions
Attribute Variable Information
Uniform Variable Information
House-keeping Data
Graphics Data
OpenGL SC 2.0 system
Screen Output

Figure 6: Our binary file format for the off-line shader compiler.

We have the specific parameters, as the followings:

- *program* is the name of an OpenGL SC shader program object. The saved program binary will be loaded into this program object.
- *format* is the format specification of the OpenGL SC shader language binary data.
- *image* is the memory address to the byte-unit array containing the binary image to be loaded into the OpenGL SC program object.
- *length* is the number of bytes contained in the one-dimensional array of *image*.

Therefore, our goal is to provide all the features to support this new function interface to the OpenGL SC 2.0 or similar OpenGL-family libraries. Actually, our OpenGL SC 2.0 library implementation on the target embedded system supports this API function, with the off-line compiled binary files.

2.2 Architectural view of the OpenGL Shaders

Although some different views on the OpenGL shaders are possible, we focused on the architectural view of the simplest vertex shader and fragment shader pairs, as shown in Figure 4. In this architectural view, the attribute variables are the input of the vertex shader, and the varying variables are the output of the vertex shader. In the case of fragment shader, it accepts varying variables as its input, and generates fragment colors as its output. Between the vertex shader and fragment shader, the fixed function unit of primitive assembly will be located. The uniform variables are used to provide some global variables and texture samplers.
The architectural view of the OpenGL shaders can be simplified as shown in Figure 5. The shaders will be

combined, and the attribute variables will be provided as the input data, and the fragment colors will be set as the final result. Uniform variables are regarded as another input data of the globally defined constant values.

Based on this conceptual view of the combined OpenGL shaders, we designed the underlying binary file format as the output of the off-line shader compiler. Since the finally combined shaders will be executed on an independent system, we should design the file format as the complete source of the all required information on the target embedded system. The details are explained in the next subsection.

2.3 Our file format design

The overall binary file format for our off-line shader compiler consists of the following elements, as shown in Figure 6:

- **fragment shader binary code:** It contains the GPU instructions for the fragment shader. User will provide the fragment shader source code to the off-line compiler, and then, the compiler will generate this GPU instructions, which may be specific to the fragment shaders.
- **vertex shader binary code:** It contains the GPU instructions for the vertex shader. It is somewhat similar to the fragment shader binary code, while it is specific to the vertex shader, rather than the fragment shader.
- **attribute variables:** The binary file should provide all the information on the attribute variables. The data type, the number of elements, the array size, and others should be indicated for each attribute variables. In some cases, the redundant attribute variables can be removed by the compiler optimization process.
- **uniform variables:** The binary file should also provide all the information on the uniform variables. The data type, the number of elements, the array size, and others should be indicated for each uniform variables. In some cases, the redundant uniform variables can be removed by the compiler optimization process.
- **house-keeping information:** Some extra data are also provided for the house-keeping purpose. For example, the binary file format version should be provided as the key of the binary file version controls. Also, the OpenGL shader language compiler version should be included into the binary file. Other data values are dependent to the GPU's and target embedded systems.

We also have some security issues. The off-line compiler and its corresponding OpenGL functional interfaces should be protected from any kind of hacks and intrusions. As the first step, the binary file format should be encrypted, and later decrypted at the execution time. We already added these features to the off-line compiler output files, and the binary file loading routines.

```
precision mediump float;
varying vec2 texCoord; /* texture coordinate */
uniform sampler2D texMap; /* texture mapping */
void main( void ) {
   gl_FragColor = texture2D(texMap, texCoord);
}
```

Figure 7: An example fragment shader source code.

```
precision mediump float;
varying vec2 texCoord; /* texture coordinate */
attribute vec2 a_texCoord;
attribute vec4 pos; /* vertex position */
uniform mat4 mvp; /* transformation matrix */
void main(void) {
   gl_Position = mvp * pos;
   texCoord = a_texCoord;
}
```

Figure 8: An example vertex shader source code

3. PROTOTYPE IMPLEMENTATION

To show the feasibility of our conceptual design on the shader language off-line compilers, we used the existing MESA shader language compiler [11] as the test platform. Our selection of the MESA version 10 has only the on-line compiling feature. We have succeeded the compilation of the isolated part of the MESA compiler implementation.

As the next step, we modified the routines to newly generate our own binary file formats. Using those output of the binary files from the shading language compilers, our OpenGL SC 2.0 library implementation has the contribution to the final integration of the binary file execution. Therefore, our OpenGL SC 2.0 library can execute the binary format API function of **glProgramBinary**.

To test our implementation, we used variety of OpenGL shader language source codes. Those shader language source codes are compiled on the Linux-based systems, and we saved the compiled result as the binary files. Those files are then transferred to the target embedded system. The system then use our OpenGL SC 2.0 library implementation, and execute those shader programs, without any compilation on the target embedded system.

Figures 7 and 8 shows one of the example shader language source codes, for the fragment and vertex shader, respectively. Figure 9 shows the final result of the off-line compiler. We have the GPU-specific instructions for the fragment and vertex shader, variable information on the attribute variables and uniform variables, and some extra house-keeping information.

With the example shader language binary file, we have executed it, and finally get the successful results, as shown in Figure 10. We executed the same OpenGL program on the desktops, and compared it with the embedded system results, we concluded that the system works successfully with various OpenGL shader programs.

4. CONCLUSION

In these days, the 3D graphics systems typically executes the GPU machine instructions, with the programmable graphics pipeline and the compilation results from the shader language supporting features [23,24,25]. The original OpenGL specification provided only the on-line compilation, which means the on-the-y compilation of the source codes at the execution time, on the target system. In contrast, they recently introduced the off-line compilation features, especially for the OpenGL SC 2.0 specification. In this case, the independent off-line compiler generates all the executable information for the given shader language source codes. Later, the target execution system should execute those compiled results, as the separate system.

In this paper, we analyzed the detailed requirements on the shader language off-line compilation. Based on these analysis, we designed the whole compiling process and the binary file format for the off-line compiler. As the verification process of our design, we implemented the prototype off-line compiler with MESA implementation. Our implementation of OpenGL SC 2.0 library on the target embedded system is enhanced with the binary file format support. We tested variety of shader programs, to be compiled and transferred to the target embedded system. We executed the pre-compiled binary files on the target system, and all the results are the same to the desktop executions.

Conclusively, we show that the shader language off-line compilers are possible with our schemes. We also presented the feasible verification results with various shader programs. Our result is promising to the extensions. In addition to the fragment and vertex shader units, we can add the geometry shader unit to the possible set of pre-compilation. The new language specifications since OpenGL 3.0 are also target to be added to our system.

```
112 28 64
0x20024b5a 0x02047ee0 0x20224b5a 0x02047fe0 0x20010d01 0x00007d07 0x02600032
0x20003a40
0x0e8d0fa0 0x860a0001 0x2000007e 0000000000 0x0040007e 0x20000208 0x06690000
0000000000
0x2002565a 0x02067ce0 0x2022565a 0x02067ee0 0x20010d01 0x00007b07 0x02800032
0x20003a40
0x0e8d0f60 0x8a0c0001 0x2000007e 0000000000
768 192
0x20224b41 0x09060ee0 0x202a4b41 0x090612e0 0x00600041 0x22c03ae8 0x3a0000d8
0x008d0120
0x00600041 0x23403ae8 0x3a0000dc 0x008d0120 0x20004d01 0x00017707 0x20010b01
0x000c7c07
0x20010b01 0x000d7d07 0x0060015b 0x0f1e0000 0x3900e1c8 0x01800410 0x0060015b
0x131e0000
... (omitted) ...
0x3903b1c8 0x00d80452 0x0060015b 0x781e0000 0x390301c8 0x00e0045a 0x0060015b
0x791e0000
0x390341c8 0x00e8045a 0x0060015b 0x7a1e0000 0x390381c8 0x00f0045a 0x0060015b
0x7b1e0000
0x3903c1c8 0x00f8045a 0x06600031 0x20003ae0 0x0e8d0ee0 0x92080017 0x2000007e
0000000000
0x92e3 0x1 4 17 a_position
0x92e3 0x1 4 18 a_texcoord
0x92e4 0x10 5 2 gl_FragColor
0x92e1 0x1 0 0 0x8b5c 4 4 0 mvp
0x92e1 0x10 0 3 0x8b5e 0 0 0 s_map
0x60000 0x4000000
```

Figure 9: The binary codes generated by our off-line compiler.

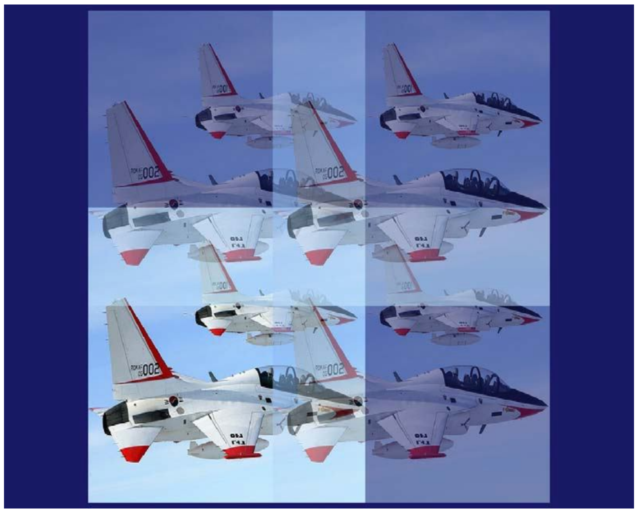

Figure 10: Screen output from the execution of the binary codes with appropriate graphics data.

REFERENCES

1. Baek, N., Baeck, G.: **Design of OpenGL SC emulation library over the desktop OpenGL 1.3.** *29th Digital Avionics Systems Conference*, 2010.
2. Cole, P.: **OpenGL ES SC: Open standard embedded graphics API for safety critical applications.** *24th Digital Avionics Systems Conference*, 2005.
3. Fabius, A., Viggers, S.: ***OpenGL SC Version 2.0.0.*** Khronos Group, 2016.
4. Group, K.: ***Vulkan 1.0.35 - A Specification.*** Khronos Group, 2016.
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: http://www.khronos.org/
8. Lipchak, B.: ***OpenGL ES version 3.0.*** Khronos Group, 2012.
9. Lipchak, B.: ***OpenGL ES version 3.2.*** Khronos Group, 2016.
10. Malizia, A.: ***Mobile 3D Graphics.*** Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
11. Mesa3D.org: ***The Mesa 3D Graphics Library***. Mesa3D.org, 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, USA, 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. Segal, M., Akeley, K.: ***The OpenGL Graphics System: A Specification***, Version 4.4 (Core Profile). Khronos Group, 2013.
18. Segal, M., Akeley, K.: ***The OpenGL Graphics System: A Specification***, Version 4.5 (Core Profile). Khronos Group, 2016.
19. Simpson, R.J.: ***The OpenGL ES Shading Language***, Language Version: 1.00. Khronos Group, 2008.
20. Simpson, R.J.: ***The OpenGL ES Shading Language***, Language Version: 3.20. Khronos Group, 2016.
21. Snyder, M.: **Solving the embedded OpenGL puzzle - making standards, tools, and APIs work together in highly embedded and safety critical environments**. *24th Digital Avionics Systems Conference,* 2005.
22. Stockwell, B.: ***OpenGL SC: Safety-Critical Profile Specification***, version 1.0.1 (difference specification). Khronos Group, 2009.
23. S. Alon, E. D. Festijo and C. D. Casuat, **Tree Extraction of Airborne LiDAR Data Based on Coordinates of Deep Learning Object Detection from Orthophoto over Complex Mangrove Forest**, *IJETER*, 8(5):2107-2111, 2020.
24. M. Kumar and R. H. Sree, **Home Computerization Monitoring System with Google Supporter**, *IJETER*, 8(6):2240-2244, 2020.
25. N. Baek, **A Simplified Implementation of the Fixed-Function Graphics Pipeline: DRM Approach**, *IJATCSE*, 9(2):1551-1555, 2020.