BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2025
http://excel.fit.vutbr.cz

Language for Prototyping of Visualizations

Daniel Kříž

Abstract

Creating graphics applications is difficult because not only is the direct code targeting the GPU hard and complex, but the user must also create whole another program targeting the CPU. This work proposes a solution to this problem in the form of a new language that is embedded into GLSL in the form of high-level preprocessor directives. The language comprises ten separate directives that make it possible to define the context and structure of a whole graphics application in a single file. This source file can be further interpreted by an interpreter that initializes the wanted resource and the GPU and further executes the program thanks to the OpenGL library. Theoretically, it can be possible to extend this approach to other shading languages and graphics APIs. Thanks to this solution, it is possible for an experienced user to rapidly prototypes new techniques without the need to waste precious time on the development of context. It also provides a more user-friendly environment for total newcomers to compute graphics programming.

*xkrizd03@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Developing applications that use GPUs is hard. Not only does the user have to learn a particular language that is used to define programs that are directly targeting the hardware (shaders). However, they also have to define the whole other application targeting the CPU that is directing the whole process. This adds unnecessary friction to the whole mix, making it hard for newcomers to enter the field and for experienced users to experiment with various techniques.

Therefore, the main problem is the amount of work that has to be done before it is possible to develop the desired solution or just learn. Depending on the use case, there are several approaches that are possible to take in order to overcome, or at least mitigate this problem.

- It is possible to use generic game engines as simple rendering engines. This has several advantages: (1) the pipeline is already set up, and (2) there usually is support for importing various kinds of data. However, the solution is still bound to the said engine, and the user has to learn a whole new technology just to create something that might possibly be much simpler [1, 2].
- Another approach is to leverage the 3D modeling software. For example, it is possible to create arbitrary meshes with textures in Blender using the node-based visual language [3]. This is great for newcomers; however, they just learn how to create visualizations, not how to implement or use some low-level technique.
- In the end, there are whole web-based solutions that make it possible to develop shaders. For example, the Shadertoy [4]. In this tool, a user can define their fragment shader and experiment with it as they please. However, they are also constrained only to the fragment shader.

This gave rise to the idea of creating an entirely new language that would allow us to use any type of shader, be lightweight and require minimal additional knowledge that is needed to create some graphics applications with it.

Therefore, I have designed a language that is presented in the following sections.

2. What is VPSL?

VPSL is a language based on high-level directives that is used together with some shading language targeting GPUs. It aims to be minimal and easy to grasp, making it possible to focus on GPU program development rather than on CPU setup.

3. Why is it useful?

- Known concept – directive-based extensions of various languages are not new concepts. For example, OpenMP [5] and OpenACC [6] are also based on this concept.
- Less work on CPU – it is possible to define the host's code with directives, and then it is possible to spend more time on the development of the desired application.
- No new language – it is just an extension of already existing shader languages. Therefore, it is only necessary to learn a handful of directives on top of GLSL knowledge.
- Extensible – because it is based on preprocessor directives and internally leverages the SPIR-V. Theoretically, any language with preprocessor and SPIR-V can support VPSL.

4. The Basic Idea

The user should be able to simply create the whole application in just a single file, with the need to set up as little of the host's context as possible. This is further illustrated by Figure 1.

5. Usage

In Listing 1 an example of a source file using the VPSL can be seen. It is a program that renders a textured triangle (the output of this program can be seen in Figure 3). The long outer gray line on the left side of the figure denotes the whole program's scope. This program contains two separate shaders denoted by the two shorter gray lines: vertex and fragment shaders. The texture is then loaded from the filesystem (the path is omitted) and bound to the sampler in the fragment shader.

6. Directives

- **Begin-end** – is used to define the scope of some other directive. Otherwise, the engine would not know which lines belong to which construct. It can be replaced with brackets for higher brevity.
- **Shader** – declares a shader. It can be of various types that are known to almost any shading language. Nevertheless, it is possible to define the so-called `generic` shader, which can contain only generic functions and can be appended to other shaders as a kind of a module.
- **Program** – declares a shader program. It is possible to specify the order of programs and to compose them solely from shaders from different programs.
- **Load** – is used for loading some resource from a file from the filesystem. Three types of resources are supported: (1) meshes, (2) textures, and (3) materials.
- **Texture** – Declares that a particular texture should be bound to the particular shader.
- **Buffer** – declares an arbitrary buffer that can be used to pass data between various shaders. Its size can be either calculated automatically or manually defined (if the buffer contains an array).
- **Resource Store** – adds a particular path in the filesystem to the search path, thus reducing the number of characters needed in the `load` directive.
- **Include** – appends other VPSL source files.
- **CopyIn** – defines the attributes that are going to be passed into the graphics pipeline.
- **Option** – enables or disables particular options of the GPU runtime (e.g., face culling).

7. How it works?

First, a source file is passed to the parser. This parser then separates the shader code and directives (and context around them, if needed). The shader code is then composed into individual shaders as the directives dictate. From these directives is then assembled the context of the application (which data should be loaded and where, what is the order of programs, which shaders compose a particular program), and this context is together with shader passed to the engine, which then executes the shader programs on the GPU. This whole process can be seen in the diagram in Figure 2.

Conclusion

My work makes it possible for newcomers to simply enter the field of computer graphics. It is also useful for experienced developers because then they can focus on the implementation of some techniques rather than on the setup of context for the application.

Acknowledgements

I would like to thank my supervisor, Ing. Tomáš Milet, Ph.D., for his help with the design of the language and his willingness to work with my chaotic nature.

References

[1] Godot Foundation. Godot documentation: Shading reference.

[2] Unity Technologies. Unity - Manual: Introduction to writing shaders in code.

[3] Blender Foundation. Blender Documentation.

[4] Shadertoy BETA.

[5] The OpenMP API specification for parallel programming.

[6] OpenACC Organization. Openacc homepage: What is OpenACC?