Shader Pipeline and Effect Encapsulation using CLOS

Nicolas Hafner

Shirakumo.org

Zürich, Switzerland

shinmera@tymoon.eu

ABSTRACT

KEYWORDS

Common Lisp, OpenGL, GPU, CLOS, Object Orientation

ACM Reference Format:

Nicolas Hafner. 2019. Shader Pipeline and Effect Encapsulation using CLOS. In *Proceedings of the 12th European Lisp Symposium (ELS'19).* ACM, New York, NY, USA, 3 pages.

ELS'19, April 1–2 2019, Genova, Italy

1 INTRODUCTION

Modern graphics systems such as OpenGL Core and DirectX offer a lot of customisation to the programmer. Particularly, in order to render an image, they allow the programmer to supply code fragments (shaders) that are run directly on the GPU. These code fragments fill in steps of a fixed rendering pipeline that is executed on the GPU in order to transform vertex data into the pixels of an image. The pipeline for OpenGL is illustrated in figure 1.

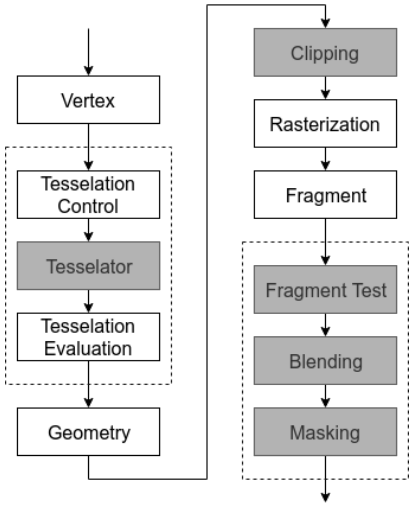

Figure 1: The stages of the OpenGL rendering pipeline. White boxes represent stages that can be customised with shader code.

For consistency, we refer to a step within the hardware rendering pipeline as a "stage", an invocation of the hardware pipeline as a "pass", and all invocations of the hardware pipeline to produce an image as a "frame".

Each of the customisable stages only accepts a single shader for each pass, making it difficult to separate, encapsulate, and ultimately combine behaviour. Furthermore, the steps required in order to change the shaders and shader inputs can be non-trivial and expensive to execute.

The complexity of managing the graphics state and the order of rendering can be massively complicated for modern requirements. Rendering a frame often requires a multitude of passes, each with their own parameters and shared data. The rendering of each object

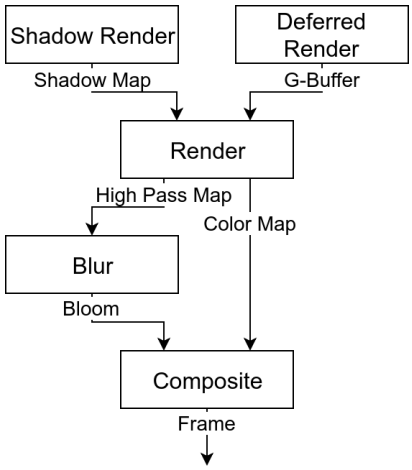

Figure 2: A sample render pipeline with shadow mapping, deferred lighting, and bloom. Each box represents a pass.

within a pass can also differ, leading to even more state that needs to be correctly managed.

This results in a difficult challenge for modularity. We attempt to solve this challenge through several systems:

- A protocol for communicating information between a pass and the objects rendered within.
- A protocol to connect the inputs, outputs, and parameters of different passes.
- An algorithm to automatically allocate shared textures.

2 RELATED WORK

Courreges[1] presents an in-depth analysis of the rendering procedure employed by the modern, high-production game GTA V. It illustrates the many passes to produce a final image, as well as their data dependencies.

Harada et al.'s work on Forward+[2][3] also clearly illustrates the need for systems that support multi-pass rendering pipelines with complex data interaction schemes.

Gyrling[4] presents an overview of the techniques used to perform parallel rendering in Naughty Dog's commercial engine. Individual stages within a pass, render passes of a frame, and multiple frame renderings are divided up into many small jobs that can run in parallel and are synchronised using counters on a shared structure.

The case study of the Unity game engine by Messaoudi et al[5] shows the availability of a set of fixed rendering pipelines that can be customised in a very limited extent with custom shaders. However, these shaders must fit into Unity's existing lighting and overall rendering model. While Unity does allow building a custom pipeline via their Scriptable Rendering Pipeline[6], they do not seem to offer any specific encapsulation or modularity features.

The work by He et al.[7] introduces a framework for general encapsulation of shaders and their parameters into structures that minimise the overhead of changing GPU state while retaining the ability to dynamically compose shader parts. They do however not create a distinction between properties for the rendering of an object, and those for the rendering of an overall pass.

Foley et al.'s work on Spark[8] presents a high-level graph-based system for defining reusable and composable shader components. Their system shows a much more distanced view of the underlying graphics hardware than we attempt. Similar to He's work, they do not present a separation between object logic and pass logic.

In our previous work[9] we introduce a system to tie shader code to classes and compose behaviour through inheritance. We make use of this system and extend it to allow further control over rendering behaviour in individual passes.

3 OVERVIEW

4 PASSES

5 PIPELINES

6 ALLOCATION

7 CONCLUSION

8 FURTHER WORK

9 ACKNOWLEDGEMENTS

10 IMPLEMENTATION

An implementation of the proposed system can be found at
https://github.com/Shirakumo/trial/blob/f34a79f0a6df21d1ed9259e85fbb3c7eed39352b/shader-pass.lisp
https://github.com/Shirakumo/trial/blob/f34a79f0a6df21d1ed9259e85fbb3c7eed39352b/pipeline.lisp
https://github.com/Shinmera/flow

A more in-depth discussion of the system can be found at
https://reader.tymoon.eu/article/363
https://reader.tymoon.eu/article/364

REFERENCES

[1] Adrian Courreges. Gta v-graphics study. http://www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study/, 2015. [Online; accessed 2019.01.24].
[2] Takahiro Harada, Jay McKee, and Jason C Yang. Forward+: Bringing deferred lighting to the next level. 2012.

[3] McKee, Jay Harada, Takahiro. Forward rendering pipeline for modern gpus. https://www.gdcvault.com/play/1016435/Forward-Rendering-Pipeline-for-Modern, 2012. [Online; accessed 2019.01.24].
[4] Christian Gyrling. Parallelizing the naughty dog engine using fibers. https://www.gdcvault.com/play/1022186/Parallelizing-the-Naughty-Dog-Engine, 2015. [Online; accessed 2019.01.24].
[5] Farouk Messaoudi, Gwendal Simon, and Adlen Ksentini. Dissecting games engines: The case of unity3d. In *Proceedings of the 2015 International Workshop on Network and Systems Support for Games*, page 4. IEEE Press, 2015.
[6] Scriptable render pipeline. https://docs.unity3d.com/Manual/ScriptableRenderPipeline.html, 2018. [Online; accessed 2019.01.24].
[7] Yong He, Tim Foley, Teguh Hofstee, Haomin Long, and Kayvon Fatahalian. Shader components: modular and high performance shader development. *ACM Transactions on Graphics (TOG)*, 36(4):100, 2017.
[8] Tim Foley and Pat Hanrahan. *Spark: modular, composable shaders for graphics hardware*, volume 30. ACM, 2011.
[9] Nicolas Hafner. Object oriented shader composition using clos. In *11 th European Lisp Symposium*, page 80, 2018.