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, 4 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.

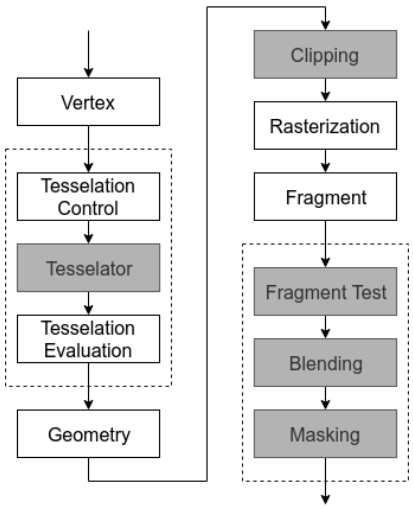

Figure 1: The stages of the OpenGL rendering pipeline. White boxes represent stages that can be customised with shader code.

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

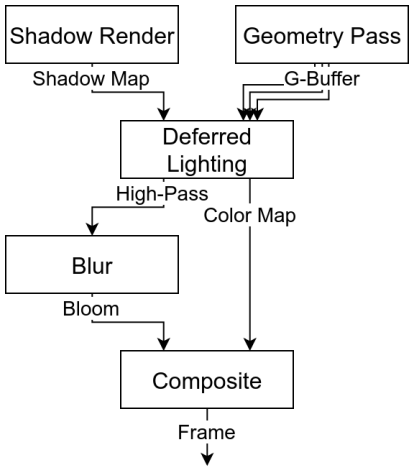

Figure 2: A sample frame pipeline with shadow mapping, deferred rendering, and bloom. Each box represents a pass and each edge a texture buffer.

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 used as buffers between passes.

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

In order to prepare the shader pipeline, several resources need to be allocated. Each shader stage needs a "framebuffer," an OpenGL resource that allows one to render to off-screen textures. These framebuffers then need to have the required textures to render onto allocated as well. As each input and output from a pass can specify constraints on the texture's features, these constraints must be matched up for any connecting edges as well. Finally, in order to minimise memory usage we would like to re-use textures where possible.

Thus the allocation proceeds in three phases: reconciling texture constraints on edges between passes, computing how textures are shared between passes, and finally constructing all the necessary resources with the previously gathered information.

6.1 Constraint Merging

OpenGL textures include a massive amount of information.[10][11] When two ports are connected that specify different constraints on the texture properties, a join must be performed. Fortunately, a wide range of the texture property values are fundamentally incompatible, meaning that a lot of the logic can simply error. For simplicity and brevity, we will focus on the join operator for a single texture property here, the internal format. The internal format specifies how many colour channels the texture has, how many bits of precision each channel has, which format each channel has, as well as whether the texture is compressed or has sRGB gamma normalisation applied.

The list of specified texture formats is quite large.[12] Unfortunately OpenGL does not give us an interface to handle these formats in a way that lets us pick the individual features easily. Instead, each format is represented by a constant number, with no relation to the features that format includes. This means that we need to first destructure each format into a list of its features:

- `R`, `G`, `B`, `A` How many bits to use for each channel, and the format of the channel (normalised, float, integer, unsigned integer).
- `depth` How many bits to use for the depth channel.
- `stencil` How many bits to use for the stencil channel.
- `shared` Whether bits are shared across the channel, and if so how many.
- `features` Whether the format has compression, sRGB, RGTC, BPTC, SNORM, or UNORM features.

Once the texture format specifications are destructured, we can perform a join as follows.

1. If the `features` and sharing are not the same, a join is impossible.
2. The `features` list of the output spec is set to the same as either of the specs.
3. If both include a `depth` component:
   3.1. The `depth` feature is set to the join of both.
   3.2. If either include a `stencil` feature, the `stencil` feature is set to the join of both.
4. If both include a `stencil` component:
   4.1. The `stencil` feature is set to the join of both.
5. If neither include a `depth` component:
   5.1. The `R` feature is set to the join of both.
   5.2. The `G` feature is set to the join of both.
   5.3. The `B` feature is set to the join of both.
   5.4. The `A` feature is set to the join of both.
6. Otherwise a join is impossible.

Wherein the "join of both" is computed as the join of two channel formats as follows.

1. If both include the format:
   1.1. If the channel format is not the same, a join is impossible.
   1.2. The channel bit depth is set to the maximum of both.
2. If one includes the format, that format is returned.
3. Otherwise, the absence of the channel is indicated.

If the join is successful, we then re-encode the texture format specification into OpenGL's constant and use it in the real texture specification. Note that this join could texture format specifications that are not legal according to the OpenGL specification. However, this can only occur if one of the specs to join is already illegal. We thus deem it unnecessary to handle such cases.

6.2 Port Allocation

In compilers, allocation of a graph of variables with use-relations is typically handled with a graph colouring algorithm. However, since we represent our graph in a different fashion than usual, with nodes having multiple distinct ports on which edges are connected, we devised a different kind of colouring algorithm to maximise texture sharing.

Given a set of nodes the allocation algorithm proceeds as follows.

1. The set of unique texture specifications is computed by joining each port's texspec with every other and comparing for equality.
2. For each unique texture specification T:
   2.1. The nodes are sorted topologically.
   2.2. The number of colours is set to 0.
   2.3. For each node N:
      2.3.1. For each output port P of N:
         2.3.1.1. If P's texspec is joinable with T...
         2.3.1.2. The number of colours is increased
   2.4. An array is allocated to fit the number of colours. Each index of the array represents a colour and each value at the index whether the colour is currently available or unavailable.
   2.5. For each node N in *reverse order*:
      2.5.1. For each input port P of N:
         2.5.1.1. For each neighbour port O of P:
            2.5.1.1.1. If O's texspec is joinable with T...
            2.5.1.1.2. and O does not yet have a colour...
            2.5.1.1.3. The first available colour is assigned to O.
            2.5.1.1.4. This colour is marked as unavailable.
      2.5.2. For each non-input port P of N:
         2.5.2.1. If P's texspec is joinable with T...
         2.5.2.2. and P does not yet have a colour...
         2.5.2.3. The first available colour is assigned to P.
         2.5.2.4. This colour is marked as unavailable.
      2.5.3. For each port P of N:
         2.5.3.1. If P's texspec is joinable with T...
         2.5.3.2. and P has a colour...
         2.5.3.3. P's colour is marked as available.

In other words, the algorithm proceeds backwards from the last node in the graph, marking predecessors' output ports with unique colours, then marking unconnected ports with unique colours, and finally marking all colours at its own output ports as available again. We repeat this process for each unique texture specification, each time ensuring we only consider ports that share that texture specification.

This algorithm is by no means efficient, but since pipeline allocation only has to happen during loading phases, we currently do not consider this to be a big problem.

Unfortunately print documents cannot yet display animations, so illustrating the algorithm in motion is not possible directly in the document. However, a brief animation of the pipeline illustrated in figure 2 is available online:

https://raw.githubusercontent.com/Shinmera/talks/master/els2019-shader-pipeline/pipeline-allocation.gif

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.

[10] Khronos Group. gltexparameter. https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glTexParameter.xhtml.

[11] Khronos Group. gleximage2d. https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glTexImage2D.xhtml.

[12] Khronos Group. Texture internal formats. https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glTexImage2D.xhtml#id-1.6.13.1.