# Shader Pipeline and Effect Encapsulation using CLOS

Nicolas Hafner
Shirakumo.org
Zürich, Switzerland
shinmera@tymoon.eu

## ABSTRACT

Modern real-time graphics make use of a lot of tricks in order to produce stunning visuals. Many of these tricks require separate rendering passes, as well as separate rendering logic for each pass. These passes are then combined in a variety of ways in order to produce a final image. The interaction between such a rendering pass and the objects it draws, as well as the interaction between multiple passes within a pipeline can become quite complex. Often times, in order to handle this complexity, the passes and objects are generalised, and the render logic is left to be executed almost entirely with either the object or the pass. We present a new method of representing objects, passes, and pipelines, which allows a modular encapsulation of effects and rendering behaviour, as well as object-oriented composition through inheritance. We make use of CLOS' multiple inheritance, multimethods, and standard method combination to form extensible protocols that allow this new method. We also make use of the MOP, in order to introduce additional metadata to classes.

## CCS CONCEPTS

• **Software and its engineering** → **Object oriented architectures**; *Abstraction, modeling and modularity*; *Object oriented frameworks*; Compilers; • **Computing methodologies** → *Computer graphics*;

## 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 **??**.

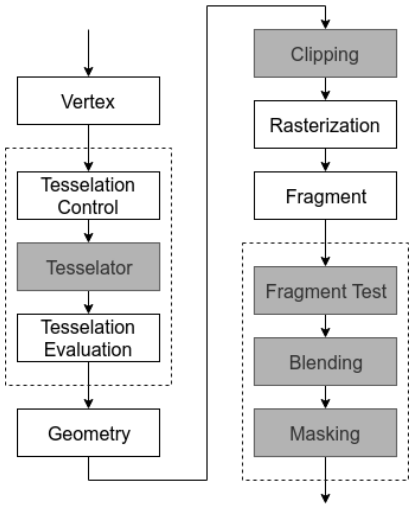


**Figure 1:** The stages of the OpenGL rendering pipeline. White boxes represent stages that can be customised with shader code.

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 accepts only 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.

Managing this graphics state and the order of rendering can be very 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.

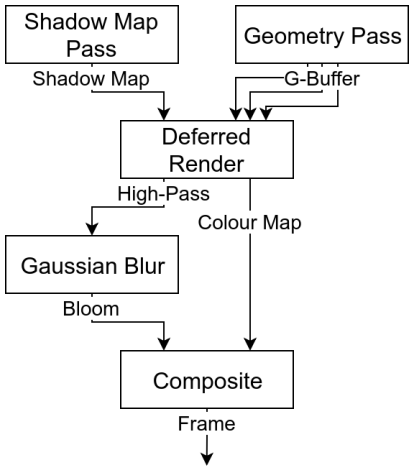


**Figure 2:** A sample frame pipeline with shadow mapping, deferred rendering, and bloom. Each box represents a pass and each edge a texture buffer.

This complexity 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 private game 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. How rendering logic requirements are communicated is however not explained.

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 to a very limited extent with custom shaders. 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 rendering an object and for those for rendering 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 to 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

The system is composed of three distinct entities: shader objects, shader passes, and pipelines. Both shader objects and shader passes encapsulate rendering logic. The pipeline, on the other hand, represents the assembly of a complete frame, and only influences rendering in the sense that it tracks which passes to run in what order.

Due to the restrictions of the rendering pipeline of OpenGL, the combination of the render logic of objects and of passes is not trivial. We present a protocol to solve this difficulty in section 4.

In section 5 we present a protocol for representing input and output information of a pass and for connecting these inputs and outputs between passes together.

Due to the complex interactions between passes in a pipeline, allocation of intermediary buffer textures is error-prone and tedious. We present algorithms to automate this in section 6.

Finally, in section 7 we show the results of implementing a medium-sized pipeline using these techniques.

## 4 PASSES

A shader pass should encapsulate the logic for drawing objects in a certain way. In order to accomplish its task, the pass needs to be able to partially or fully control the shaders of objects. For instance, an object should still be able to have control over how its vertices are constructed and what materials and textures are applied to it. The shader pass should be able to make use of this information if it needs to, or discard it completely if unneeded.

In order to permit this amount of control, we make use of the shader composition capabilities that we presented in our previous work[9]. This technique allows us to combine pieces of GLSL code. On top of this shader combination sits a new protocol to control the interaction between shader passes and objects.

```
(defclass shader-pass ()
  ()
  (:metaclass shader-pass-class))

(defgeneric register-object-for-pass (shader-pass object))
(defgeneric shader-program-for-pass (shader-pass object))
(defgeneric make-pass-shader-program (shader-pass object))
(defgeneric coerce-pass-shader (shader-pass object stage))
(defgeneric render-with (shader-pass object))
(defgeneric render (object target))
```

**Listing 1:** The protocol for shader passes.

For any object that should be rendered using a given pass, first `register-object-for-pass` must be called. This call allows the pass to prepare the shader program that will be used during rendering for this object. A shader program is an OpenGL resource that compiles the shaders for the stages of a pass together. Using `shader-program-for-pass` this program can then later be retrieved. The shader program is important for objects to access, as it allows them to set values for "uniform" variables that are used in the shader code.

`register-object-for-pass` calls `make-pass-shader-program` to compute this program, and registers it internally in the shader pass so that it can be retrieved later. If the shader pass would like to retain complete control over how each object is drawn, it would forego this call and instead generate its own shader program. Doing so is especially useful for post-processing effects that don't render any objects at all, and instead simply operate on textures that are output by previous passes.

`make-pass-shader-program` gathers all the shader sources for a program using `coerce-pass-shader` and then generates a representation for an OpenGL shader program. This representation also includes additional information such as the data buffers used.

`coerce-pass-shader` computes the effective shader source for a particular shader stage or type. Typically this computation simply involves the combination of the shader sources of both the shader pass and of the object, for the given shader type. We perform this combination using the same parsing and code walking strategy as we described in our previous work.[9]

Objects are typically rendered using the `render` function. Users are encouraged to add methods that specialise the behaviour and perform necessary setup, as well as to perform the final draw call for their custom object classes. For instance, a very primitive class could look as shown in **??**.

This function is fine for allowing the object control over the behaviour. However, the pass cannot exert the same amount of control, due to the generic function's argument precedence. For example, if an `:around` method specialised on the object exists, a pass would not have any way of preventing it from firing, as its own `:around` method would be executed afterwards. We thus introduce another function with inverted argument order that is called first.

```
(defclass simple-object ()
  ((vertex-array :accessor vertex-array)
  (:metaclass shader-class))

(defmethod render ((oject simple-object) target)
  (let ((vao (vertex-array object)))
    (gl:bind-vertex-array (gl-name vao))
    (%gl:draw-elements :triangles (size vao) :unsigned-int 0)))

(defmethod render :before ((object simple-object)
                           (pass shader-pass))
  (let ((program (shader-program-for-pass pass object)))
    (setf (uniform program "projection_matrix")
          (projection-matrix))))
```

**Listing 2:** A simple object class and its render methods. The first method tells OpenGL to render a list of vertices. The second method sends the projection matrix to the GPU via a uniform variable.

`render-with` thus exists mostly as an entry point to allow shader passes greater control over the rendering. Typically it will simply defer to `render`. This inversion is very important for shader passes that take all control away from the objects.

This protocol thus allows a great amount of control both over the effective shader code used to render an object, as well as the behaviour leading up towards the actual draw call for an object.

## 5 PIPELINES

Shader passes not only retain information about how objects are drawn, but also about the input and output textures that a pass interacts with. For this reason, each shader pass is also a node in a graph, with distinct input and output ports.

```
(defclass deferred-render (shader-pass)
  ((position-map :port-type input)
   (normal-map :port-type input)
   (albedo-map :port-type input)
   (color :port-type output))
  (:metaclass shader-pass-class)
```

**Listing 3:** An outline of a deferred rendering pass, taking position-, normal-, and albedo-map textures as input, producing a single color texture as output.

An example of such a pass is illustrated in **??**. The ports are modelled as slots of the class that carry additional metadata, requiring a new metaclass. A slot with a port-type will not only hold a texture for its value, but also retain information about how it is connected to other passes. Using slots in this manner also allows us to inherit ports from other classes, and thus combine behaviour alongside the shader source code.

For the sake of brevity and ease of explanation, we have left out the details of texture constraints in these code samples. Nevertheless, constraining the features of textures tied to the ports is important, and we discuss the technique for dealing with that in section 6.

```
(defclass shadow-render (shader-pass)
  ((shadow-map :port-type input)
   (color :port-type output))
  (:metaclass shader-pass-class))
```

**Listing 4:** An outline of a shadow rendering pass, taking into account the information from a shadow-map to render shadows onto the output color texture.

```
(defclass high-pass-render (shader-pass)
  ((high-pass :port-type output))
  (:metaclass shader-pass-class))
```

**Listing 5:** An outline of a high-pass renderer, which splices off colours of a high intensity into a high-pass output texture.

For instance, the "Deferred Render" pass shown in **??** is created by combining the passes illustrated in **??**, **??**, and **??** through inheritance.

When assembling a pipeline, we can then connect these ports together, in order to dictate how the various output textures generated by a pass are used as inputs for other passes. As an example, the pipeline from **??** is created in **??**.

```
(let ((pipeline (make-pipeline))
      (shadow (make-instance 'shadow-map-pass))
      (geometry (make-instance 'geometry-pass))
      (deferred (make-instance 'deferred+shadow-pass))
      (blur (make-instance 'gaussian-blur-pass))
      (composite (make-instance 'composite-pass)))
  (connect (port shadow 'shadow-map)
           (port deferred 'shadow-map) pipeline)
  (connect (port geometry 'position-map)
           (port deferred 'position-map) pipeline)
  (connect (port geometry 'normal-map)
           (port deferred 'normal-map) pipeline)
  (connect (port geometry 'albedo-map)
           (port deferred 'albedo-map) pipeline)
  (connect (port deferred 'high-pass)
           (port blur 'previous-pass) pipeline)
  (connect (port deferred 'color)
           (port composite 'color) pipeline)
  (connect (port blur 'color)
           (port composite 'bloom) pipeline)
  (prepare pipeline))
```

**Listing 6:** An assembly of the pipeline shown in **??**.

The pipeline object itself retains information on the order in which the passes should be executed, and keeps track of the textures that are allocated in order to run the passes. How this information is computed is described in section 6.

The procedure to actually render objects with this pipeline technique is then as follows:

1. Create a pipeline object and instances of the desired shader passes.
2. Connect the ports of the shader passes.
3. Prepare the pipeline to allocate the needed resources.
4. Create instances of the desired objects.
5. Register each object with each pass.
6. Call `render` on the pipeline with a collection of all objects to draw.
7. The output texture of the output port of the last shader pass in the pipeline will contain the finished frame.

The completed frame can then be blitted onto the screen, or be used as the input for another computation.

## 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 allocated as well. As each input and output from a pass can specify constraints on the features of the texture, 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[**?** ][**?** ]. When two ports are connected that specify different constraints on the texture properties, a join must be performed. A wide range of the texture property values are fundamentally incompatible, meaning that a lot of the logic can simply error. However, other options require more complicated joining logic. For simplicity and brevity, we will focus on the join operator for a single texture property here: the internal format. The internal format is arguably the most important property. This property 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[**?** ]. 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 whose value has no relation to the features that format includes. This representation means that we first need to destructure each format name into a list of 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 channels, 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 this constant in the real texture specification. Note that this join could produce texture format specifications that are not legal according to the OpenGL specification. However, this can only occur if one of the specifications 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 texture specification 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 texture specification 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 texture specification 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 texture specification 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 texture specification 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 output ports of predecessors with unique colours, then marking unconnected ports with unique colours, and finally marking all colours at the node's 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. We also have not performed any analysis as to whether the algorithm produces optimal allocation results in every case.

Unfortunately printed 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 **??** is available online:

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

# 7 PROOF OF CONCEPT

As a proof of concept we have implemented these protocols and mechanisms, as well as the passes shown in **??**. As the code required to do so is quite lengthy and involved, and the graphics techniques used are beyond the scope of this paper, we will omit the code. The relevant outline of the passes and the pipeline involved has already been shown in section 5.

Nevertheless, you can find the complete code online at the following pages:
https://github.com/Shirakumo/trial/blob/b889d50/deferred.lisp
https://github.com/Shirakumo/trial/blob/b889d50/hdr.lisp
https://github.com/Shirakumo/trial/blob/b889d50/shadow-map.lisp
https://github.com/Shirakumo/trial/blob/b889d50/workbench.lisp

??, ??, ??, ??, and ?? show the output textures produced by the various stages for a simple scene.

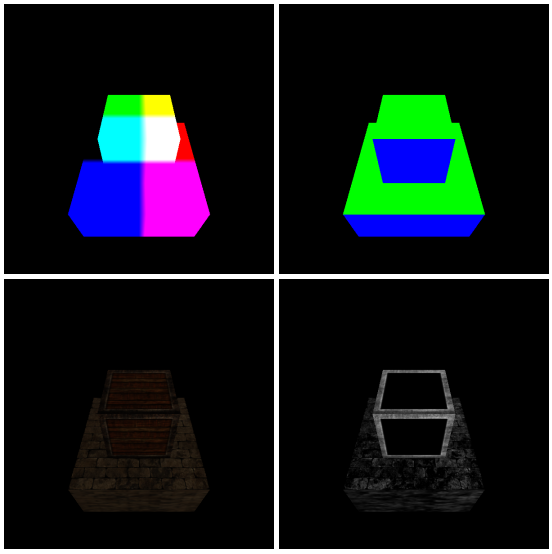

**Figure 3:** The output of the geometry pass, producing a "G-buffer" composed out of a position- (top left), normal- (top right), albedo- (bottom left), and specular-map (bottom right).

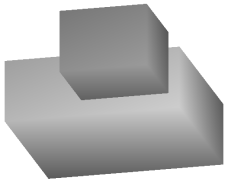

**Figure 4:** The depth map produced by the shadow map pass. It shows the distance of surfaces from the point of view of the primary light source.

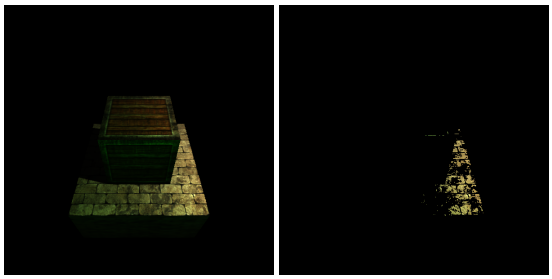

**Figure 5:** The color map (left) and the high-pass map (right) that is output by the deferred render pass. It combines the information from the G-buffer from ??, the shadow-map from ??, and information for lights to render an image. Note that due to gamma adjustments the image appears quite dark.

**Figure 6:** The blurred high-pass from ?? produced by the gaussian blur pass. This is used to produce an effect called bloom in the final composite pass.

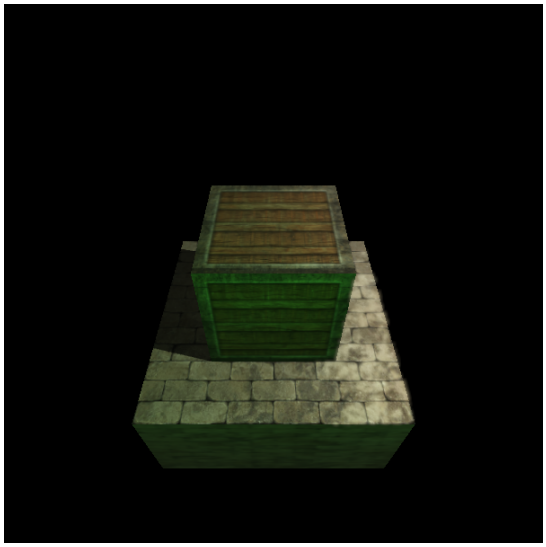

**Figure 7:** The final output frame that is produced by adding together the bloom texture from ?? and the standard colour texture from ??. Gamma adjustment and High Dynamic Range reduction are used for colour normalisation.

## 8 CONCLUSION

We have presented a system to handle the separation between objects that should be rendered, and the way in which they should be rendered. By using a protocol for the interaction between such objects and passes, we are capable of controlling and combining a variety of rendering behaviours. With an additional protocol and set of algorithms we also present a high-level view of rendering pipelines, allowing users to conveniently compose effects to produce complicated scenes.

These protocols and algorithms are largely generic enough that they could also be adapted for use in other languages and systems. However, we believe that the Common Lisp Object System gives us a set of very convenient tools to make everything feel more natural and more integrated with the rest of the system than other environments would allow us to do.

Finally, we have demonstrated the capability of this system by constructing a real-time rendering system out of individual, modular pieces of code.

## 9 FURTHER WORK

Currently there exist several restrictions in our framework.

Most severe is that there is no standardised protocol to communicate capabilities and data between objects and passes. This restriction means that, for instance, an object that does not possess texturing has no way of communicating this to a shader pass that might need to render it. Currently this problem is partially solved by introducing a mandatory superclass that objects need to be a subtype of, if they want to be renderable under a given pass. This superclass then loosely defines the interaction. However, this additional class does not solve issues of compatibility of shader code used by the object and by the shader pass. In order to properly resolve this restriction, we currently see two required features:

Shader source merging must be aware of uses-relations, meaning that the code walker must track where variables, functions, and types are used and reorder the definitions and declarations in such a way that the dependencies are fulfilled. In GLSL, variables, functions, and types cannot be referenced before they are declared, requiring this reordering step. With this problem solved, shader code could be written in such a way to be more amiable towards combination and manipulation by external code.

GLSL's limited expressiveness makes it very difficult to figure out relations between code and how to properly combine pieces. A more high-level language, as is used in other frameworks like Spark[8] or Shader Components[7], should be introduced to allow a more convenient way to declare and write shaders, objects, and passes. With a higher-level language, a compiler could more easily figure out how to combine features and reconcile differences between pieces of code.

Another restriction is that there is currently no way to encapsulate GLSL code other than associating it with a class, which is a rather heavy-weight operation. It would be far more useful to introduce a library mechanism that allows the definition of standalone shader functions, which can then be required by shader classes.

## 10 ACKNOWLEDGEMENTS

I would like to thank Michal "phoe" Herda, Robert Strandh, Michael Reis, and all the people that will review this paper. Your name could be here!

## 11 IMPLEMENTATION

An implementation of the proposed system can be found at
https://github.com/Shirakumo/trial/blob/b889d50/shader-pass.lisp
https://github.com/Shirakumo/trial/blob/b889d50/pipeline.lisp
https://github.com/Shinmera/flow

## 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.