Semantic-Based Shader Generation Using Shader Shaker

Michael Delva, Julien Hamaide, and Ramses Ladlani

2.1 Introduction

Maintaining shaders in a production environment is hard, as programmers have to manage an always increasing number of rendering techniques and features, making the amount of shader permutations grow exponentially. As an example, allowing six basic features, such as vertex skinning, normal mapping, multitexturing, lighting, and color multiplying, already requires 64 shader permutations.

Supporting multiple platforms (e.g., HLSL, GLSL) does not help either. Keeping track of the changes made for a platform and manually applying them to the others is tedious and error prone.

This chapter describes our solution for developing and efficiently maintaining shader permutations across multiple target platforms. The proposed technique produces shaders automatically from a set of handwritten code fragments, each responsible for a single feature. This divide-and-conquer methodology was already proposed and used with success in the past, but our approach differs from the existing ones in the way the fragments are being *linked* together. From a list of fragments to use and thanks to user-defined semantics that are used to tag their inputs and outputs, we are using a pathfinding algorithm to compute the complete data flow from the initial vertex attributes to the final pixel shader output.

Our implementation of this algorithm is called Shader Shaker. It is used in production at Fishing Cactus on titles such as *Creatures Online* and is open source for you to enjoy.

2.2 Previous Work

As mentioned earlier, there are two main categories of issues graphic programmers may have to deal with at some point when it comes to shader maintenance: the (possibly high) number of feature permutations and the multiple backends to support (e.g., HLSL, GLSL).

2.2.1 The Permutation Hell Problem

The *permutation hell* problem is almost as old as the introduction of programmable shaders in the early 2000s. [Kime 08] categorizes the solutions to this problem into three main families (code reuse through includes, subtractive approaches, and additive approaches). To these categories, we added a fourth one that we will call *template-based approaches.*

Code reuse. This should be the solution that is the most familiar to programmers. It consists of implementing a library of utility functions that will be made available to the shaders thanks to an inclusion mechanism (e.g., include preprocessor directive) allowing code to be reused easily. The main function of the shader can then be written using calls to these functions and manually feeding the arguments. This is a natural way of editing shaders for programmers, but it gets difficult for the less tech savvy to author new permutations and still requires maintaining all permutations by hand.

A related solution is the one described in [Väänänen 13], where the Python-based Mako templating engine is used to generate GLSL shaders.

Subtractive solutions. Über-shader solutions rely on one (or a few) mammoth shader(s) containing all the code for all features. The different permutations are generated using a preprocessor to select the relevant portions of code. This technique has proved to be a valid solution for a long time and has been used in countless productions. Nevertheless, its major drawback is that über-shaders are usually hard to maintain (because of their length and the lack of readability caused by the preprocessor directives), especially in a multilanguage environment. Another problem with this approach is that shader semantics can also be tricky to work with (their number is limited and they sometimes need to be sequentially numbered, making it hard to use them with a simple preprocessor).

Additive solutions. These work the other way around by defining a series of elementary *nodes* (or functions) to be aggregated later (either online or offline) to produce the shader. The aggregation is performed by wiring nodes' inputs and outputs together, either visually using a node-based graph editor or programmatically. This approach has seen lots of implementations [Epic Games Inc. 15, Holmér 15] largely because of its user friendliness, allowing artists to produce visually pleasing effects without touching a single line of code. Its main drawback remains the difficulty to control the efficiency of the generated shaders [Ericson 08, Engel 08b].

A complete system for generating shaders from HLSL fragments is described in [Hargreaves 04] in which each shader fragment is a text file containing shader code and an interface block describing its usage context. In this framework, fragments are combined without actually parsing the HLSL code itself. The system was flexible enough to support adaptive fragments, which could change their behavior depending on the context in which they were used, but lacked the support of a graph structure (i.e., the system was restricted to linear chain of operations). Tim Jones implemented this algorithm for XNA 4.0 in [Jones 10].

Trapp and Döllner have developed a system based on code fragments, typed by predefined semantics that can be combined at runtime to produce an über-shader [Trapp and Döllner 07].

In [Engel 08a], Wolfgang Engel proposes a shader workflow based on maintaining a library of files, each responsible for a single functionality (e.g., lighting.fxh, utility.fxh, normals.fxh, skinning.fxh), and a separate list of files responsible for stitching functions calls together (e.g., metal.fx, skin.fx, stone.fx, eyelashes.fx, eyes.fx). This is similar to the node-based approach, but it is targeted more at programmers. As will be shown later, our approach is based on the same idea but differs from it (and the other node-based solutions) by the fact that the wiring is done automatically based on user-defined semantics.

Template-based solutions. The last category finds its roots in the famous template method pattern [Wikipedia 15b], where the general structure of an algorithm (the program skeleton) is well defined but one is still allowed to redefine certain steps.

This is one of the higher-level techniques adopted by Unity (alongside the regular vertex and fragment shaders), which is itself borrowed from Renderman: the surface shader [Pranckevičius 14b]. By defining a clear interface (predefined function names, input and output structures), the surface shader approach allows the end user to concentrate on the surface properties alone, while all the more complex lighting computations (which are much more constant across a game title) remain the responsibility of the über-shader into which it will be injected. It should be noted that it would be possible to combine this with any of the previous three methods for handling permutations at the surface level only.

Taking the idea a bit further, [Yeung 12] describes his solution where he extends the system with interfaces to edit also the vertex data and the lighting formula. Unnecessary code is stripped by generating an abstract syntax tree and traversing it to obtain the variables' dependencies.

2.2.2 The Language Problem

Extensive reviews about the different techniques and tools available to maintain shaders across different languages are available in [Pranckevičius 10a, Pranckevičius 12, Pranckevičius 14a]. We refer the reader to these articles for more information, but we summarize the approaches to handling this problem into the following four families.

The manual way. This could eventually be performed with the help of macros where the languages do differ, but it does not scale well. It is still tricky because of subtle language differences and is hard to maintain.

Use another language. Use a language (eventually a graphical one) that will compile into the target shader language as output.

Cross-compile from one language to another. Lots of tools are available to translate from one language to the other at source code level. The problem can be considered as solved for DirectX 9–level shaders, but there is still work to do for supporting the new features that have appeared since then (e.g., compute, geometry, etc.).

Compile HLSL to bytecode and convert it to GLSL. This is easier to do than the previous technique but suffers from a partly closed tool chain that will run on Windows only.

2.3 Definitions

Our technique is based around the concepts of *fragments* and *user-defined semantics* (not to be confused with the computer graphics fragment used to generate a single pixel data).

- **Fragment:** In this context, a fragment is a single file written in HLSL that is responsible of implementing a single feature and that contains all the information required for its execution, including uniforms and samplers declarations, as well as code logic. A fragment example is provided in Listing 2.1.
- **User-defined semantic:** A user-defined semantic is a string literal used to tag a fragment input or output (e.g., `MeshSpacePosition`, `ProjectedPosition`). This tag will be used during shader generation to match a fragment's output to another one's input. User-defined semantics use the existing HLSL semantic feature, used for mapping input and output of shaders.

2.4 Overview

Shader Shaker, our shader generator, uses a new idea to generate the shader. User-defined semantics are added to intermediate variables, as shown in Listing 2.1. The generation algorithm uses those intermediate semantics to generate the list of call functions. The algorithm starts from expected output, e.g.,

```
float4x4 WvpXf;

void GetPosition(
    in float3 position : VertexPosition,
    out float4 projected_position : ProjectedPosition
    )
{
    projected_position = mul( float4( position, 1 ), WvpXf );
}
```

Listing 2.1. `GetPosition` fragment.

`LitColor`, and creates a graph of the function required to generate the semantic up to the vertex attributes.

To generate a shader, one has to provide the system with a list of fragments to use (`vertex_skinning` + `projected_world_space_position` + `diffuse_texturing` + `normal_mapping` + `blinn_lighting`, for example). Thanks to the semantics, it is possible to link the desired fragments together to produce the final output semantic required by the system (e.g., `LitColor`) and generate the corresponding complete shader.

Fragments are completely uncoupled; code can be written without consideration of where the data comes from. For example, for a fragment that declares a function that needs an `input` argument with a semantic of type `ViewSpaceNormal`, the tool will search another fragment with a function that has an `output` argument of the very same semantic to link to this one. In deferred rendering, the fragment that provides this `output` argument with the semantic `ViewSpaceNormal` would read the geometry buffer to fetch that value, whereas in forward rendering, a function could, for example, just return the value of the view-space normal coming from the vertex shader. In any case, the fragment in the pixel shader that uses this `ViewSpaceNormal` is agnostic to where the data it needs comes from.

To achieve this, the code generator adopts a compiler architecture, going through separate phases:

- HLSL fragments are processed by Shader Shaker to generate for each of them an abstract syntax tree (AST).
- The ASTs are processed to create a final AST, which contains all the needed code (functions/uniforms/samplers). The algorithm (explained in detail in the following section) generates this final AST from the required output semantics (the output of the pixel shader), then goes upward to the input semantics, calling successively all functions whose output semantic match the input semantic of the previous function.
- Eventually, this final AST is converted to the expected output language (e.g., HLSL, GLSL, etc.).

As the concept has been introduced, let's dig into the algorithm.

```
struct FunctionDefinition
{
    set<string> InSemantic;
    set<string> OutSemantic;
    set<string> InOutSemantic;
};
```

Listing 2.2. `FunctionDefinition` structure.

2.5 Algorithm Explanation

The algorithm used to generate the shader is inspired by the A* path-finding algorithm [Wikipedia 15a]. The idea is to find a path from the required output semantic to the existing input semantics, i.e., the vertex attributes. The path is searched using open and closed semantic sets, in the same way as the open and closed node lists of the original algorithm. To successfully generate the code, the compiler must be provided the following information:

- the list of fragments to use, i.e., the feature list;
- the list of required output semantics (each of them will be mapped to a system semantic such as `COLOR0`; multiple render target code can be generated by defining multiple final output semantics);
- the list of available input semantics (this can change from mesh to mesh, creating tailored shaders for a given vertex format).

After the parsing of all fragments, the AST is inspected to extract the signature of functions. Each function that declares one or more semantics for its arguments is processed, others being considered as helper functions. A `FunctionDefinition` structure describing the function is filled up with these semantics information (see Listing 2.2). A fragment is then defined by a map of definitions addressed by function names. It's important to notice that `inout` function arguments are supported. It's useful when a fragment wants to contribute to a variable, like summing different lighting into a final lit color or when transforming a vertex position through several fragments. When processing an `inout` semantic, the semantic is kept in the open set. As each function can only be used once, another function outputting the semantic is required.

The code generation is done in two steps. The first step consists of the creation of the call graph. The algorithm is described in Listing 2.3. This algorithm generates a directed acyclic graph of all function calls from the output to the input. The second step consists of code generation from the call graph. As the graph represent the calls from the output, it must be traveled depth first. To

```
open = {required semantic}
closed = {}

repeat until open is empty
    for each fragment from last to first
        for each semantic in open
            if unused function with semantic
                        in OutSemantic exists
                add_function( function )
                restart
            end
        end
    end

    report error, semantics in open set do not resolve
end

add_function( f )
    node = { f, f.InSemantic, f.InOutSemantic}
    open -= f.InSemantic
    open += f.OutSemantic
    //Add inout semantic back in the open set
    open += f.InOutSemantic
    closed += f.InSemantic
    //Link node that required the semantic
    for each semantic in { f.OutSemantic, f.InOutSemantic }
        node[ semantic ].children.add( node )
    end

    //Report as requiring those semantics
    for each semantic in { f.InSemantic, f.InOutSemantic }
        node[ semantic ] = node
    end
    //Remove semantic provided by vertex
    open -= Vertex.AttributeSemantics;
end
```

Listing 2.3. Code generation.

simplify code generation and debugging, the semantic is used as the variable name. The code generation algorithm is described in Listing 2.4. Finally, a map of user semantics to system semantics is generated, information to be used in the engine to bind vertex attributes accordingly.

To illustrate this algorithm, a toy example will be executed step by step. The fragments are defined as shown in Listing 2.5, Listing 2.6, and Listing 2.7. The function definitions are created as shown in Listing 2.8. The required semantic is `LitColor`. The algorithm generates a graph as shown in Figure 2.1. One can see the open and closed set populated as the algorithm creates the graph. Finally, the graph is processed to create the code shown in Listing 2.9. It is important to notice that the code just uses functions declared in fragments. The final code aggregates all the fragments codes, only with semantic information removed. It's not the purpose of this module to prune unused code. This step can be left to further modules.

```
write function definition with required attributes/varyings
for each node in depth first order
    for each output variable
        if variable has not been encountered yet
            write variable declaration
        end
    end

    write function call

end
```

Listing 2.4. Code generation.

```
Texture DiffuseTexture;

sampler2D DiffuseTextureSampler
{
    Texture = <DiffuseTexture>;
};

void GetDiffuseColor( out float4 color : DiffuseColor,
    in float2 texcoord : DiffuseTexCoord
    )
{
    color = tex2D( DiffuseTextureSampler, texcoord );
}
```

Listing 2.5. Diffuse color from texture fragment.

```
void ComputeNormal( in float3 vertex_normal : VertexNormal,
    out float3 pixel_normal : PixelNormal )
{
    pixel_normal = normalize( vertex_normal );
}
```

Listing 2.6. Simple normal fragment.

```
float4 SomeLighting( in float4 color : DiffuseColor,
    in float3 normal : PixelNormal ) : LitColor
{
    return ( AmbientLight
        + ComputeLight( normal ) )* color;
}
```

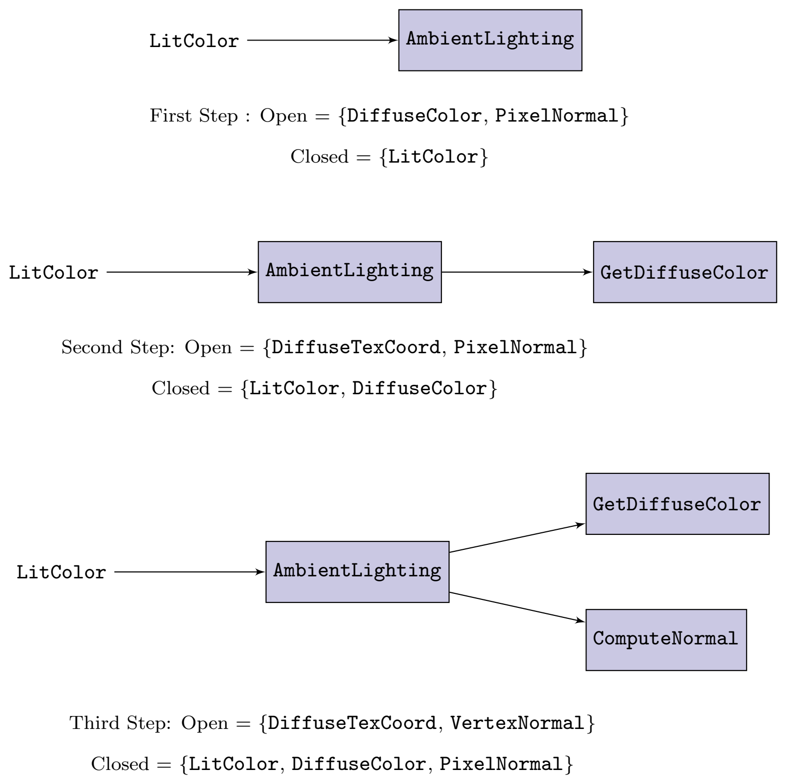
Listing 2.7. Some lighting fragment.

```
GetDiffuseColor :
{
    InSemantic : { "DiffuseTexCoord" }
    OutSemantic : { "DiffuseColor" }
}

ComputeNormal :
{
    InSemantic : { "VertexNormal" }
    OutSemantic : { "PixelNormal" }
}

SomeLighting :
{
    InSemantic : { "DiffuseColor", "PixelNormal" }
    OutSemantic : { "LitColor" }
}
```

Listing 2.8. Function definition examples.



First Step : Open = {`DiffuseColor`, `PixelNormal`}

Closed = {`LitColor`}

Second Step: Open = {`DiffuseTexCoord`, `PixelNormal`}

Closed = {`LitColor`, `DiffuseColor`}

Third Step: Open = {`DiffuseTexCoord`, `VertexNormal`}

Closed = {`LitColor`, `DiffuseColor`, `PixelNormal`}

Figure 2.1. Graph generation process.

```
float4 main( in float3 VertexNormal : NORMAL,
    in float2 DiffuseTexCoord : TEXCOORD0 )
{
    float4 DiffuseColor;
    GetDiffuseColor(DiffuseColor, DiffuseTexCoord);
    float3 PixelNormal;
    ComputeNormal(VertexNormal,PixelNormal);
    float4 LitColor
        = SomeLighting(DiffuseColor, PixelNormal);
    return LitColor;
}
```

Listing 2.9. Generated code.

2.6 Error Reporting

2.6.1 Syntax Errors

Syntactic errors existing in fragments are reported as a shader compiler would. Each fragment should be a valid compilation-wise shader. This is detected when parsing the fragments.

2.6.2 Fragment Compliance

A fragment must comply to a list of rules:

- It must only output a given semantic once.
- It must define all constants and sampler it uses.

If either of these rules is broken, the generator reports the error and how to fix it.

2.6.3 Missing Semantics

If any semantic is not found while generating the call graph, the user is informed and the generation is stopped.

2.6.4 Graph Cycles

When a node is inserted in the graph, the graph is checked for cycles. If any are found, the semantics found in the cycle are output.

2.6.5 Mismatching Types for a Semantic

After graph generation, a sanity check is run to ensure all occurrences of a semantic are of the same type. No automatic casting is allowed (e.g., from `float3` to `float2`).

2.6.6 Target Code Restrictions

When targeting a specific platform, additional conditions are checked (e.g., sampler count, vertex attribute count, unsupported texture fetch in vertex shader, etc.)

2.7 Usage Discussions

On top of solving the permutation and the multiplatform problems mentioned earlier, this technique offers the ability to support some noteworthy tricks and features. This discussion lists how they can be leveraged to improve both programmers' and artists' experiences.

2.7.1 Fragments Variants

By using a system similar to [Frykholm 11], it becomes easy to allow your engine file system to choose among multiple variants of a given fragment (e.g., `lighting.fx`, `lighting.low.fx`). We exploit this feature for various purposes:

Platform-specific code. When dealing with multiple graphic platforms, it may happen that the default implementation of a fragment cannot be used natively because the resulting shader is not compatible with the rendering API, or the hardware (e.g., vertex texture fetch). This mechanism allows us to provide a platform-specific version of a given fragment.

Graphic quality. The same principle can be used to manage graphic quality settings. Depending on user settings or based on device capabilities, appropriate fragments can be selected to balance quality against performance.

2.7.2 Fragment Metadata

Each fragment can be associated with a metadata file to ease its integration into the tools. In our case, we chose to export this metadata automatically from the fragments themselves and in JSON format. The available information includes the list of uniforms, the list of textures, a description of what the fragment does, etc.

Thanks to this information, it is easy to populate a combo box from which the artists can select the fragment they want to add to the current material and then tweak the settings the newly added fragment offers.

Furthermore, this metadata also allows us to match the required attributes against the mesh vertex format. A missing component in the vertex format triggers an error, whereas unused data can be stripped safely.

2.7.3 Data-Driven Features

Adding a new rendering feature to the engine and the editor is as easy as adding a new fragment file to the fragment library. As the editor is data-driven, no intervention of a programmer is needed: reloading it is enough. Still, creating a new fragment requires an understanding of the underlying concept. It also requires knowledge of the set-defined semantic, as it could be project specific.

2.7.4 Programming

Accessing the metadata of generated shaders can be leveraged as a data-driven feature, e.g., binding the vertex attributes and the uniforms without using the rendering API to enumerate them. This is even more useful when the graphics API doesn't allow us to do so at all.

2.7.5 Debugging

Programmers can easily debug shaders that are generated by Shader Shaker. Indeed, the output semantics are provided as arguments to the generation process. If an issue is suspected at any level, the shader can be regenerated with an intermediate semantic as the output semantic. For example, if we want to display the view-space normal, the `ViewSpaceNormal` semantic is provided as the output semantic. If the semantic variable type is too small to output (e.g., `float2` while ouputs should be `float4`), a conversion code is inserted.

2.7.6 Choice of Semantics

Semantics are just considered links by the algorithm. Nevertheless, the choice of semantics is really important. If the set is not chosen correctly, new features might require redesigning it, which would require existing fragments' refactoring. The set should be documented precisely to remove any ambiguity on the usage.

2.8 What's Next

While Shader Shaker in its current form is already used with success in our games and tools, there is still room for improvements.

- Use custom semantics for uniforms and samplers. For now, the semantic resolution is only applied to functions and input/output arguments. Applying it to uniforms can be convenient, allowing some values to be passed either at the vertex level or as uniforms.

- The concept of semantic could be augmented. Semantics could have additional properties, such as default values, ranges, normalization, etc. On top of function calls, extra code would be emitted to answer extra specifications.

- Some improvements can be made to the error reporting. In case of an error when generating a shader, the exact position of the error in fragments could be provided with the line number. Also, currently the tool is not yet able to detect cycles in dependency between fragments. It will be of a great help to be able to detect those. Another improvement related to error reporting is a finer detection of grammar errors in the fragments.
- As said before, Shader Shaker does not do any optimizations over the generated shader. Converting Shader Shaker as a frontend to already existing modules, which could take care of those optimizations, would be an interesting improvement. In our toolchain at Fishing Cactus, we already execute the GLSL optimizer [Pranckevičius 10b, Pranckevičius 15] over the generated GLSL files produced by Shader Shaker. We could, for example, integrate LLVM [LLVM 15] at different steps of the generation to optimize the AST and/or the IR.
- We have designed Shader Shaker so that it's really easy to support new output shader languages. Currently, we only support output to HLSL and GLSL, but new languages could be easily supported.

2.9 Conclusion

This technique and its user-semantic linking algorithm brings a new ways of creating shaders. It provides a new way to manage the complexity and combinatory complexity. Each feature can be developped independently, depending only on the choice of semantics. Shader Shaker, our implementation, is distributed as open source software [Fishing Cactus 15].

Bibliography

[Engel 08a] Wolfgang Engel. "Shader Workflow." *Diary of a Graphics Programmer*, http://diaryofagraphicsprogrammer.blogspot.pt/2008/09/shader-workflow.html, September 10, 2008.

[Engel 08b] Wolfgang Engel. "Shader Workflow—Why Shader Generators are Bad." *Diary of a Graphics Programmer*, http://diaryofagraphicsprogrammer.blogspot.pt/2008/09/shader-workflow-why-shader-generators.html, September 21, 2008.

[Epic Games Inc. 15] Epic Games Inc. "Materials." *Unreal Engine 4 Documentation*, https://docs.unrealengine.com/latest/INT/Engine/Rendering/Materials/index.html, 2015.

[Ericson 08] Christer Ericson. "Graphical Shader Systems Are Bad." http://realtimecollisiondetection.net/blog/?p=73, August 2, 2008.

[Fishing Cactus 15] Fishing Cactus. "Shader Shaker." https://github.com/FishingCactus/ShaderShaker2, 2015.

[Frykholm 11] Niklas Frykholm. "Platform Specific Resources." http://www.altdev.co/2011/12/22/platform-specific-resources/, December 22, 2011.

[Hargreaves 04] Shawn Hargreaves. "Generating Shaders from HLSL Fragments." http://www.shawnhargreaves.com/hlsl_fragments/hlsl_fragments.html, 2004.

[Holmér 15] Joachim 'Acegikmo' Holmér. "Shader Forge." http://acegikmo.com/shaderforge/, accesssed April, 2015.

[Jones 10] Tim Jones. "Introducing StitchUp: 'Generating Shaders from HLSL Shader Fragments' Implemented in XNA 4.0." http://timjones.tw/blog/archive/2010/11/13/introducing-stitchup-generating-shaders-from-hlsl-shader-fragments, November 13, 2010.

[Kime 08] Shaun Kime. "Shader Permuations." http://shaunkime.wordpress.com/2008/06/25/shader-permutation/, June 25, 2008.

[LLVM 15] LLVM. "The LLVM Compiler Infrastructure." http://llvm.org/, 2015.

[Pranckevičius 10a] Aras Pranckevičius. "Compiling HLSL into GLSL in 2010." http://aras-p.info/blog/2010/05/21/compiling-hlsl-into-glsl-in-2010/, May 21, 2010.

[Pranckevičius 10b] Aras Pranckevičius. "GLSL Optimizer." http://aras-p.info/blog/2010/09/29/glsl-optimizer/, September 29, 2010.

[Pranckevičius 12] Aras Pranckevičius. "Cross Platform Shaders in 2012." http://aras-p.info/blog/2012/10/01/cross-platform-shaders-in-2012/, October 1, 2012.

[Pranckevičius 14a] Aras Pranckevičius. "Cross Platform Shaders in 2014." http://aras-p.info/blog/2014/03/28/cross-platform-shaders-in-2014/, March 28, 2014.

[Pranckevičius 14b] Aras Pranckevičius. "Shader Compilation in Unity 4.5." http://aras-p.info/blog/2014/05/05/shader-compilation-in-unity-4-dot-5/, May 5, 2014.

[Pranckevičius 15] Aras Pranckevičius. "GLSL Optimizer." *GitHub Repository*, https://github.com/aras-p/glsl-optimizer, 2015.

[Trapp and Döllner 07] Matthias Trapp and Jürgen Döllner. "Automated Combination of Real-Time Shader Programs." In *Proceedings of Eurographics 2007*, edited by P. Cignoni and J. Sochor, pp. 53–56. Eurographics, Aire-la-Ville, Switzerland: Eurographics Association, 2007.

[Väänänen 13] Pekka Väänänen. "Generating GLSL Shaders from Mako Templates." http://www.lofibucket.com/articles/mako_glsl_templates.html, October 28, 2013.

[Wikipedia 15a] Wikipedia. "A* Search Algorithm." http://en.wikipedia.org/wiki/A*_search_algorithm, 2015.

[Wikipedia 15b] Wikipedia. "Template Method Pattern." http://en.wikipedia.org/wiki/Template_method_pattern, 2015.

[Yeung 12] Simon Yeung. "Shader Generator." http://www.altdev.co/2012/08/01/shader-generator/, August 1, 2012.