Staged Metaprogramming for Shader System Development

KERRY A. SEITZ, JR., University of California, Davis, USA
TIM FOLEY, NVIDIA, USA
SERBAN D. PORUMBESCU, University of California, Davis, USA
JOHN D. OWENS, University of California, Davis, USA

The shader system for a modern game engine comprises much more than just compilation of source code to executable kernels. Shaders must also be exposed to art tools, interfaced with engine code, and specialized for performance. Engines typically address each of these tasks in an ad hoc fashion, without a unifying abstraction. The alternative of developing a more powerful compiler framework is prohibitive for most engines.

In this paper, we identify *staged metaprogramming* as a unifying abstraction and implementation strategy to develop a powerful shader system with modest effort. By using a multi-stage language to perform metaprogramming at compile time, engine-specific code can consume, analyze, transform, and generate shader code that will execute at runtime. Staged metaprogramming reduces the effort required to implement a shader system that provides earlier error detection, avoids repeat declarations of shader parameters, and explores opportunities to improve performance.

To demonstrate the value of this approach, we design and implement a shader system, called Selos, built using staged metaprogramming. In our system, shader and application code are written in the same language and can share types and functions. We implement a design space exploration framework for Selos that investigates static versus dynamic composition of shader features, exploring the impact of shader specialization in a deferred renderer. Staged metaprogramming allows Selos to provide compelling features with a simple implementation.

CCS Concepts: • **Computing methodologies → Computer graphics**; • **Software and its engineering** → *Compilers*; *Domain specific languages*; Extensible languages.

Additional Key Words and Phrases: Computer Graphics, Shaders, Shading Languages, Metaprogramming, Multi-Stage Languages

ACM Reference Format:
Kerry A. Seitz, Jr., Tim Foley, Serban D. Porumbescu, and John D. Owens. 2019. Staged Metaprogramming for Shader System Development. *ACM Trans. Graph.* 38, 6, Article 202 (November 2019), 15 pages. https://doi.org/10.1145/3355089.3356554

Authors' addresses: Kerry A. Seitz, Jr., University of California, Davis, Dept. of Computer Science, One Shields Avenue, Davis, CA, 95616, USA, kaseitz@ucdavis.edu; Tim Foley, NVIDIA, 2788 San Tomas Expressway, Santa Clara, CA, 95051, USA, tfoley@nvidia.com; Serban D. Porumbescu, University of California, Davis, Dept. of Electrical and Computer Engineering, One Shields Avenue, Davis, CA, 95616, USA, sdporumbescu@ucdavis.edu; John D. Owens, University of California, Davis, Dept. of Electrical and Computer Engineering, One Shields Avenue, Davis, CA, 95616, USA, jowens@ece.ucdavis.edu.

1 INTRODUCTION

Developers of real-time 3D game engines like Unity,[1] Unreal Engine,[2] and other in-house engines[3] want to deliver high-quality, efficient, and accessible frameworks on which many types of games can be built. In order to accomplish this goal, they must implement a *shader system* that provides a variety of functionality, including shader parameter reflection and metadata extraction, effect and material systems, shader optimization frameworks, and cross compilers to support different target graphics APIs (e.g., Direct3D [Blythe 2006] and OpenGL [Segal et al. 2015]). Such a shader system is *multifaceted* – it must provide different interfaces to different clients of the system. These clients include a wide variety of users (ranging from expert graphics programmers to non-technical artists), as well as runtime engine code that configures shaders for high-performance execution. As a result, engine developers spend a lot of effort designing shader systems that both result in highly optimized final code while simultaneously providing the appropriate interfaces for each type of person involved in development.

The shader programming interfaces in graphics APIs do not directly help with building such multifaceted shader systems, because they only focus on issues that affect loading and execution of GPU shader code. Modern graphics APIs are designed to facilitate robust, high-performance implementations on a wide range of hardware, and as such their shader programming interfaces focus on minimal, low-level abstractions (as evidenced by the shift from high-level languages as the standard interface to lower-level intermediate representations like SPIR-V [Kessenich and Ouriel 2018]). Thus, developers are left to create layered implementations of missing facets on top of these API interfaces.

Engine developers must balance the cost to implement functionality against the benefits in improved features or robustness of the system. Features are typically built in an incremental, ad hoc fashion, with increasing complexity as the system evolves. The initial version of a shader system might read and write shaders as strings of text, performing pattern matching, substitution, etc. To add further functionality, an engine might wrap an underlying shading language with a custom domain-specific language (DSL). Finally, making more invasive changes to a shading language requires building or modifying a compiler. Because engines require different design choices, it is difficult to amortize this work by developing a single shader system that can be used in multiple engines.

When examining the disparate techniques used to implement shader systems, we observe that they largely fall under the umbrella of *metaprogramming*: writing code that manipulates other code.

[1]https://unity3d.com/
[2]https://www.unrealengine.com/
[3]e.g, http://www.frostbite.com/ and https://www.cryengine.com/

Textual-based processing tools, DSL implementations, and compilers must all read, analyze, transform, and generate code. However, the metaprogramming methods currently employed by modern shader systems are on an unfavorable continuum – methods with greater capabilities require greater effort for implementors of the shader systems. Using the key insight that these techniques are all examples of metaprogramming, we present the following contributions:

- We identify *staged metaprogramming* as a unifying methodology that sidesteps the trade-off between capabilities and implementation complexity.
- We present the design of Selos,[4] a shader system built using staged metaprogramming, to demonstrate the efficacy of this technique.
- We demonstrate how staged metaprogramming can open opportunities for optimizations by creating a design space exploration framework in our system. This framework investigates static versus dynamic composition of features in order to balance between execution efficiency and the number of compiled shader variants.

We present the design of Selos in Section 5. Prior to that, we introduce staged metaprogramming, the underlying methodology on which it is built (Section 4). To motivate our decision to use staged metaprogramming, we examine other methods of creating shader systems (Section 3), which also inform our design goals (Section 2). We then use Selos to explore static versus dynamic composition of shader features in Section 6.

2 DESIGN GOALS

Motivated by issues in other modern systems, we built a shader system guided by the following goals:

- **Minimize implementation effort and maintenance costs**
  Each engine requires a unique shader system, customized to the engine's design and the needs of its users. Developers must often balance between the effort required to add features versus the benefits those features provide to users. To better enable the development of robust and feature-rich shader systems, we must minimize the resource investments required to build them.
- **Early error detection**
  Underlying graphics APIs, as well as many shader systems, expose shader parameters to CPU code through "stringly-typed" runtime interfaces, which provide poor validation. In contrast, our goal is to detect errors as early as possible.
- **Don't Repeat Yourself (DRY)**[5]
  Programmers should not need to declare the same shader parameter, uniform buffer, etc. in more than one place.
- **Performance**
  In real-time graphics applications, performance is paramount, so a shader system must not decrease game runtime performance. The system must strive to minimize overheads to GPU shader code and CPU engine code, as well as enable developers to explore opportunities to improve performance.
- **Productivity for artists and technical artists**
  While engine and graphics developers often prioritize performance over programming inconveniences, shader systems are also used by artists for whom productivity is key. Therefore, a shader system must provide artists with familiar workflows.
- **Support options for static and dynamic composition**
  Game engines generate specialized shader variants to achieve maximum performance. However, complete static specialization can lead to additional overheads that decrease performance. Thus, exploring the trade-offs between static and dynamic composition is important for future shader systems.

Given the landscape of existing solutions (Section 3), our first design goal ("Minimize implementation effort and maintenance costs") seems at odds with some of our other goals. While this observation is true in many languages, we will demonstrate that certain programming techniques alleviate this concern. Specifically, our system meets these goals using *staged metaprogramming* (Section 4).

Our set of design goals cannot be readily realized in current versions of the languages commonly used by game engines today (e.g., C++, HLSL, GLSL) because they lack more modern programming techniques. As such, we do not restrict ourselves to using these languages. Therefore, while ease of adoption is an important practical consideration, it is largely orthogonal to the core contributions of this paper. While we explore opportunities presented by other languages and programming techniques, we discuss the potential for these techniques to be used in future versions of C++ in Section 8.

3 EXISTING SOLUTIONS

In this section, we briefly discuss existing ways to implement some aspects of a shader system, ordered by increasing levels of complexity. Since we cannot survey every possible solution, we have chosen a few representative examples to illustrate the need for a better overall approach. Note that when we refer to HLSL below, we could substitute any modern shading language like GLSL or Metal Shading Language. As we will discuss, our key insight is that all of these methods heavily utilize some form of metaprogramming.

3.1 Plain C++ and HLSL

Simple graphics applications might rely on the facilities provided by C++, HLSL, and the Direct3D (D3D) API directly. Consider this (abridged) example shader written in HLSL:

```
cbuffer LightData : register(b0) {
  float3 lightDirection;
};
...
float4 surfaceShader(...) {
  ...
#if defined(STANDARD)
  color = evalStandardMaterial(shadingData);
#elif defined(SUBSURFACE)
  color = evalSubsurfaceMaterial(shadingData);
#elif defined(CLOTH)
  color = evalClothMaterial(shadingData);
#endif
  return color * max(0, dot(shadingData.normal, lightDirection));
}
```

[4]https://github.com/kseitz/selos
[5]https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

This shader has one parameter (`lightDirection`) and expresses three specialization options (`STANDARD`, `SUBSURFACE`, and `CLOTH`) that each use a different bidirectional reflectance distribution function (BRDF). The only way we know about these specialization options is by examining the HLSL code directly; therefore, to generate specialized shader variants, a shader author must manually specify the appropriate `#defines` to the shader compiler.

Shading languages and their corresponding graphics APIs are only concerned with providing an interface to programmers. Thus, a programmer would need to separately prepare a list of parameters (e.g., using XML) to expose them to a GUI-based tool for artists. Similarly, coordinating the interaction between CPU and GPU code is left to the programmer. Hence, setting the value of parameters from C++ is a manual process as well:

```
dxContext->PSSetConstantBuffers(0, 1, &lightDataBuf)
```

where the first argument refers to the `register` binding slot written in the shader (**`register`**`(b0)`). Neither HLSL nor the D3D API perform any checks to ensure that the correct register was used or that the layout of the CPU-side `lightDataBuf` data structure matches the layout of the GPU-side `LightData` constant buffer.

To work around these issues, one could write a shared header that declares a common data structure for the constant buffer, using C preprocessor `#defines` to handle the differences between HLSL and C++. Each time a programmer authors such a shared struct, they need to manually account for the packing rules of the underlying API so that the layout of the CPU-side struct matches what the shader expects. Furthermore, a developer can write additional infrastructure for each shader to better interface with C++ code. Unreal Engine uses this approach, where each HLSL shader has a corresponding C++ class written by the shader writer [Epic Games, Inc. 2019]. Though these classes provide a clean interface for other parts of the engine to use the shaders, the programmer is responsible for ensuring that, e.g., the parameter names and types match those specified in the separately-written HLSL code.

The user-written class implementations make heavy use of preprocessor macros defined by Unreal Engine. By using the macro mechanisms built into C++ and HLSL, the Unreal Engine developers do not need to invest resources in developing their own mechanisms. However, C preprocessor facilities are limited in what they can express, resulting in extra effort for users of the engine.

3.2 A Layered DSL with Embedded HLSL

To provide further functionality, some engines implement a custom layered DSL on top of an underlying shading language. Shaders in Unity are written in ShaderLab [Unity Technologies 2019]:

```
Shader "SurfaceShader" {
  Properties {
    lightDirection {"Light Direction", Vector} = (0,0,0)
  }
  ...
  CGPROGRAM
  #pragma multi_compile STANDARD SUBSURFACE CLOTH

  float3 lightDirection;

  float4 surfaceShader(...) {
    ...
  #if defined(STANDARD)
    color = evalStandardMaterial(shadingData);
  #elif defined(SUBSURFACE)
    color = evalSubsurfaceMaterial(shadingData);
  #elif defined(CLOTH)
    color = evalClothMaterial(shadingData);
  #endif
    return color * max(0, dot(shadingData.normal, lightDirection));
  }
  ENDCG
}
```

The code between `CGPROGRAM` and `ENDCG` is HLSL. Shader variants are again expressed using preprocessor `#if` commands. However, ShaderLab uses a custom preprocessor to implement the `#pragma multi_compile` syntax, which exposes the variant options to the system and allows the engine to generate the set of compiled variants automatically. After textual preprocessing, ShaderLab treats HLSL code as a black box.

Because the ShaderLab compiler has no understanding of the embedded HLSL code, shader authors must repeat themselves by declaring each artist-configurable parameter twice – once in the HLSL and again in the “Properties” listing – which is more error prone and can lead to issues with refactoring tools. Analogous to the D3D API, programmers use a “stringly-typed” interface to set shader parameters, which is also error prone:

```
shader.SetVector("lightDirection", Vector4(1.0, 1.0, 1.0, 1.0));
// bug: lightDirection is a float3 in the shader, not a float4
```

If a programmer specifies the wrong parameter name, the system may generate a runtime error. However, if the wrong type is used (as above), no error is reported and instead they are left with a bug.

By using a mix of preprocessor features and a simple DSL compiler, the effort required to implement Unity’s ShaderLab is relatively modest. However, the system precludes early error detection and results in shader authors repeating themselves, thereby hindering user productivity.

3.3 A DSL That Manipulates and Generates HLSL

Bungie’s TFX language [Tatarchuk and Tchou 2017] features better integration with HLSL at the cost of greater implementation effort for the engine developers. The surface shader example written in TFX might look (roughly) like:

```
import "MaterialComponents.tfx"

c_materialType:* material @default(none);
float3 lightDirection @default(float3(0,0,0)) @UI(Slider);
...
#hlsl
  float4 surfaceShader(...) {
    ...
    color = material.apply(shadingData);
    return color * max(0, dot(shadingData.normal, lightDirection));
  }
#end
```

In this example, the different material BRDFs are written as “components” (imported from a separate file). The shader parameter named `material` will be exposed to a GUI, where an artist can select a specific implementation of the `c_materialType` interface (e.g., `c_materialType:standard`, `c_materialType:cloth`) in order to generate a specialized shader variant for the required BRDF.

While the TFX compiler does not understand all of HLSL, it does know enough to manipulate it. For example, it can translate DSL features like components (`material.apply(shadingData)`) into plain HLSL. TFX also has a custom metadata system that allows one to express information (e.g., default values, GUI controls) directly alongside the parameter declaration, thus avoiding the double-declaration problem in ShaderLab. This feature is possible because TFX generates HLSL from these parameters, rather than just treating HLSL code as a black box.

TFX provides multiple mechanisms for communicating runtime data to shaders. "Object channels" and "global channels" allow scripts and artist-authored content to bind data to shaders. Because this data comes from content (not code), it is loaded dynamically (and, thus, cannot be validated at compile time). In contrast, "externs" communicate engine-provided data to shaders. This data is tightly bound to the C++ engine code, so any changes require recompilation of the engine and rebaking of the affected shaders.

The TFX system provides a better way to encapsulate shader variants and does not require shader authors to repeat themselves. However, such features require tighter integration with HLSL, resulting in higher implementation effort.

3.4 Modifying HLSL

Rather than creating a DSL that embeds HLSL, one could instead extend the shading language itself. The Slang shading language [He et al. 2018] extends HLSL by adding some general-purpose language features from other popular programming languages. Here is the surface shader example in Slang:

```
include "MaterialComponents.slang"

float3 lightDirection;
...
float4 surfaceShader<M : IMaterial>(ParameterBlock<M> material) {
  ...
  color = material.eval(shadingData);
  return color * max(0, dot(shadingData.normal, lightDirection));
}
```

Similar to TFX, the material BRDFs are again written as components that implement a common interface (`IMaterial`). Slang uses generics, constrained by these interfaces, to express specialization options (`<M : IMaterial>`). Programmers use the Slang runtime API to generate specialized shader variants:

```
Module* module      = loadModule(/* path */);
EntryPoint* entry   = findEntryPoint(module, "surfaceShader");
Type* clothType     = findType(module, "ClothMaterial");
Kernel* clothShader = specializeEntryPoint(entry, &clothType, 1);
```

This introspection API provides runtime validation to ensure that the final types are compatible with the interface constraints. The API also includes the ability to query type layout information (not shown here), which the renderer can use to properly setup and populate parameter blocks by accounting for GPU data packing rules. However, since the API uses strings to identify entry points, types, etc., it cannot perform validation at application compile time.

Slang is a shader compiler, not a multifaceted shader system. Therefore, it does not directly provide the various interfaces needed by such a system, instead requiring that users (e.g.) implement artist tools and facilitate setting parameters across the CPU-GPU boundary.

Creating a new language and compiler to implement missing shader system features is cost prohibitive for the vast majority of development teams. Similarly, forking an existing compiler (such as Slang or Microsoft's DirectX Shader Compiler [Microsoft 2019]) brings along maintenance costs as both the fork and the main project continue to evolve. In contrast, the previous approaches discussed in this section could reuse an existing compiler without modification, thereby limiting the resource investments needed to use them.

3.5 Summary

While each subsequent example presented above improves upon some deficiencies of the previous examples, this improvement comes at the cost of greater implementation effort. Ideally, we believe developers should be able to achieve the results of the more complex solutions, while requiring effort similar to the simpler methods.

When examining these solutions, we observe that the techniques they employ are all examples of metaprogramming. We broadly define metaprogramming as writing code that manipulates other code, which includes reading, analyzing, transforming, or generating code. C preprocessor facilities, custom DSL implementations, and shading language compiler modifications all fit this definition. Therefore, we hypothesize that the effort required to implement a robust shading system can be reduced by making metaprogramming a fundamental design principle and utilizing a metaprogramming technique that sidesteps this apparent trade-off between capability and complexity.

4 STAGED METAPROGRAMMING

The principal design decision for our system, underlying the core of its implementation, is to use a technique called *staged metaprogramming*. Unlike the metaprogramming techniques commonly used by shader systems today (discussed in Section 3), staged metaprogramming provides a more favorable balance between the effort required to use it and the capabilities it provides, which better enables us to achieve our design goals. In this section, we will present staged metaprogramming and motivate our decision to use it as a basis for our system.

4.1 Definition

Our definition of staged metaprogramming aligns with the description of a *multi-level* language in Taha's dissertation [1999]. In staged metaprogramming, code running in an earlier stage of execution can construct and manipulate code that will run in a later stage using explicit *staging annotations* (e.g., *quasi-quote* and *unquote*). Staged metaprogramming also includes *multi-stage* languages, which extend multi-level languages by allowing explicit invocation of next-stage code (e.g., by `eval` in Lisp [McCarthy 1960]).

The key features of staged metaprogramming are:

- Code is a first-class citizen, meaning programs can operate on code in the same ways that they can operate on other entities (including passing code as arguments, returning code from functions, and storing code in data structures).

- Code is constructed (metaprogrammed) using regular language syntax by enclosing the code to generate in a *quasi-quote* construct. Code within quasi-quotes is syntax- and type-checked at application compile time.
- Generated code created with quasi-quote is inserted into the runtime application using *unquote.*
- To prevent variable capture issues, quasi-quoted code is hygienic and lexically scoped by default [Bawden and Rees 1988]. However, there are mechanisms to intentionally violate lexical scoping when needed.
- Quasi-quotes can be specialized to generate different versions of the code as needed.
- Current-stage code can execute quasi-quote code using an *eval* mechanism.

As we will discuss in Section 7.3, some previous shader systems have employed a staged metaprogramming approach, albeit not by name. These works are examples of *runtime* staged metaprogramming, focusing on generating code at application runtime. While runtime staged metaprogramming is indeed useful for shader development, real-time graphics applications must be high performance, and excess code generation at runtime will degrade performance. Therefore, our work focuses on *compile-time* staged metaprogramming (while supporting runtime as well) in order to prevent code-generation overhead from affecting runtime performance.

Staged metaprogramming allows programmers to run arbitrary code at compile time, written in a fully-featured language. Engine developers can create libraries that get invoked during the compilation processes to analyze and generate both application and shader code. This functionality gives them a level of control over compilation that would otherwise only be possible by creating a custom language and compiler. By providing a feature-rich environment in which to create, modify, and transform code, staged metaprogramming allows developers to express powerful code generation and manipulation implementations using semantic information, with effort only slightly greater than ad hoc approaches based on textual preprocessing.

4.2 Example Shader

Returning to the surface shader example from Section 3, here is how this shader looks in our staged metaprogramming-based system:

```
local MaterialSystem = require("MaterialSystem")
...
shader SurfaceShader {
  ConfigurationOptions {
    MaterialType = MaterialSystem.MaterialTypeOption.new()
  }
  ...
  uniform LightData {
    @UIType(Slider3) lightDirection : vec3
  }
  ...
  fragment code
    ...
    color = [MaterialType:eval()](shadingData)
    return color * max(0, dot(shadingData.normal, lightDirection))
  end
}
```

In our system, specialization is expressed and controlled through `ConfigurationOptions`. Different shader variants are generated by changing the configuration:

```
local Cloth  = require("MaterialTypes").Cloth
local config = SurfaceShader:getDefaultConfiguration()
config.MaterialType:setMaterial(Cloth)
SurfaceShader:setConfiguration(config)
local src = SurfaceShader:generateShaderSourceCode()
```

In this case, the code to evaluate the Cloth BRDF (imported from the "MaterialTypes" file) will replace the call to `[MaterialType:eval()]`, and `src` will contain the HLSL/GLSL of the specialized shader.

Beyond manually specifying a single specialization, our system can generate all specialized variants automatically because the `ConfigurationOptions` contain information about all specialization options. ShaderLab's `#pragma multi_compile` feature enables Unity's shader system to generate all variants as well. However, because ShaderLab relies on preprocessor `#if` directives to express the specialization options, they are limited to generating variants that either statically include or statically exclude each option. In contrast, we will show in Section 6 that staged metaprogramming provides greater flexibility when generating variants, allowing our system to explore additional specialization decisions.

Because our system is better able to understand and manipulate the code of a shader, shader writers can express metadata for artist GUIs directly alongside the parameter declaration. Therefore, shader writers do not have to repeat themselves when declaring such parameters, in contrast to ShaderLab's separate "Properties" listing. Furthermore, our system can readily generate a statically-checked interface for CPU-side code to set shader parameters, in order to detect errors at compile time:

```
var myShader = SurfaceShader.new()
var lightData = myShader.LightData:map(...)
lightData.lightDirection = vec4(1.0f, 1.0f, 1.0f, 1.0f)
-- compile-time error: lightDirection is of type vec3
```

Notice that the example shader above looks similar to a shader written in GLSL or HLSL, and it does not exhibit aspects of staged metaprogramming directly. This design is intentional. While staged metaprogramming underlies our shader system, technical artists should not be confronted with foreign metaprogramming constructs, as these constructs may interfere with their productivity. Therefore, we present a DSL to these artists so that they can work with a familiar interface. The example shader above is written in this DSL. In Section 5.2, we show how staged metaprogramming enables our shader DSL implementation, and we also present a description of this syntax with a more complex example.

4.3 Lua-Terra: A Research Substrate for Staged Metaprogramming

Because C++, HLSL, and GLSL do not have the features required of a staged metaprogramming environment (as listed in Section 4.1), we must use a different language to demonstrate why these features are useful for shader systems. We want to model the programming environment of typical game engines as much as possible, meaning that our runtime engine should be implemented in a low-level systems programming language similar to C++. Therefore, we built our shader system using the Lua-Terra programming language [DeVito et al. 2013].

Lua-Terra is a multi-stage language that uses Lua [Ierusalimschy et al. 1996] code (a commonly-used scripting language in games today) in the first stage to manipulate next-stage code in Terra (a low-level statically-typed C-like language). Lua-Terra extends the syntax of Lua to allow Terra expressions and statements to be quasi-quoted (`` `(expr) `` or `quote stmts end`). Lua expressions that evaluate to Terra code can be *spliced* into a quasi-quote using the unquote operator (`[expr]`). Lua-Terra also provides a mechanism for writing syntax extensions to Lua, allowing for rapid DSL implementation.

Lua-Terra is primarily designed for runtime multi-stage metaprogramming: a running Lua program generates and executes Terra code on demand. Our focus here is instead on *compile-time* metaprogramming, in which the Lua code runs entirely ahead of time, yielding a final Terra program for deployment, free of metaprogramming or dynamic features. During development, however, there are many cases where more flexible multi-stage programming is valuable. For example, development builds of an engine may implement *hot reload* (reloading shaders while the application is running) by invoking the compiler (Lua code) from runtime (Terra) code.

Lua and Terra are significantly different languages, since Lua is a dynamic scripting language whereas Terra is a static systems language. In our implementation, runtime application code, runtime engine code, and shader code are all authored in Terra (and, thus, can share types and subroutines), while Lua code performs all metaprogramming tasks. We conjecture that an ideal metaprogramming system for graphics would use the same language for both metaprogramming and for final application code; however, to our knowledge, a staged metaprogramming C++-like systems language does not currently exist.

Newer languages like Rust [Rust Project Developers 2015], as well as future versions of C++, are trending towards supporting staged metaprogramming facilities, as we discuss in Section 8. However, from our investigations, they do not yet have all of the features we need. While Lua-Terra is less practical for building a production game engine, it does have the features necessary for us to investigate our design ideas today, which will hopefully guide future designs as more popular systems languages continue to evolve.

4.4 Limitations of Staged Metaprogramming

Debugging programs with significant metaprogramming can be challenging, and staging can compound the issue. Programs might have nested metaprogramming components, requiring developers to track down issues through multiple levels of code generation. However, programmers already cope with debugging metaprogrammed code (e.g., C++ template metaprogramming issues, which traditionally have convoluted error messages), and the additional code manipulation facilities of staged metaprogramming allow developers to generate more descriptive error messages. Furthermore, developers can perform most of the metaprogramming in library code, thereby hiding metaprogramming concerns from artists and technical artists (see Section 5.2). Nevertheless, both engine developers and application/shader code authors can encounter difficulty debugging metaprogramming issues, so exploring how to more easily debug such issues is an interesting area for future research.

Excessive and undisciplined use of metaprogramming may transform user-written shader code in ways that obfuscate final shader code generation. Such obfuscation can negatively impact a developer's ability to predict how changes in shader code will affect final shader performance. Often, game developers will forgo using certain programming techniques if they reduce the ability to understand how authored code is compiled to final executed code because performance tuning is critical to the application. Still, engines already use metaprogramming techniques successfully, so employing a new, more structured metaprogramming method like staged metaprogramming can provide significant value.

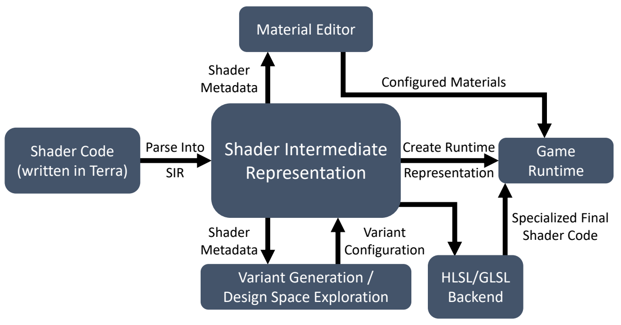

Material Editor
Shader Metadata
Configured Materials
Shader Code (written in Terra)
Parse Into SIR
Shader Intermediate Representation
Create Runtime Representation
Game Runtime
Shader Metadata
Variant Configuration
Specialized Final Shader Code
Variant Generation / Design Space Exploration
HLSL/GLSL Backend

Fig. 1. An overview of the Selos Shader System, as discussed in Section 5.

5 OTHER KEY DESIGN DECISIONS

Having presented our decision to use staged metaprogramming as the underlying implementation technique, we now discuss the design of our shader system, called Selos. Fig. 1 shows an overview.

In Selos, programmers write shader code in Terra (Section 5.3), with custom syntax extensions for shader-specific features (e.g., Fig. 2 Listing 1). Our shader DSL (Section 5.2) implementation parses the shader-specific features and generates a Shader Intermediate Representation, or SIR, object (Section 5.1). The rest of the system interfaces with the SIR to extract information about the shaders, as well as to manipulate the shaders prior to final code generation. The Material Editor pulls data from the SIR to display artist-editable parameters in a GUI. Shader variants are generated by manipulating the SIR to express each required variant (Section 5.5) and then sending the SIR to our backend code generators to emit HLSL or GLSL code. Finally, the Selos game runtime, also written in Terra, creates a statically-checked runtime shader representation (Section 5.4) from the SIR to allow engine and application code to control shader parameters and to setup graphics state appropriately prior to shader execution.

In the rest of this section, we discuss the major design decisions of our shader system. Many parts of our system are similar to other modern shader systems (e.g., similar syntax, artist tool chain). Therefore, we focus on the differences and how staged metaprogramming, specifically, enables our design and leads to inherent benefits.

```
 1  shader SimpleShader {
 2    textureSampler diffuseMap : sampler2D
 3
 4
 5    param model : mat4
 6    param view  : mat4
 7    param proj  : mat4
 8
 9    uniform PerFrame {
10      @UIType(Slider3) lightDirection : vec3
11    }
12
13    uniform PerObject {
14      modelViewProj : mat4 =
15        proj*view*model
16      modelViewIT   : mat4 =
17        inverse(transpose(view*model))
18    }
19
20    input position  : vec3
21    input normal    : vec3
22    input uv        : vec2
23
24    varying vNormal   : vec3
25    varying vUV       : vec2
26
27    output outColor   : vec4
28
29    vertex code
30      Position = modelViewProj * make_vec4(position, 1)
31      vNormal  = (modelViewIT * make_vec4(normal, 0)).xyz
32      vUV = uv
33    end
34
35    fragment code
36      var diffuse = texture(diffuseMap, vUV)
37      outColor =
38          diffuse * max(0, dot(vNormal, lightDirection))
39    end
40  }
41
```

Listing 1. A simple shader in our DSL syntax.

```
 1  local SimpleShader  = ShaderBuilder.new("SimpleShader")
 2  local diffuseMap =
 3    SimpleShader:declareTextureSampler(sampler2D)
 4
 5  local model = SimpleShader:declareParam(mat4)
 6  local view  = SimpleShader:declareParam(mat4)
 7  local proj  = SimpleShader:declareParam(mat4)
 8
 9  local PerFrame        = SimpleShader:declareUniformBlock()
10  local lightDirection  = PerFrame:declareUniform(vec3, nil,
11                             Slider3)
12
13  local PerObject     = SimpleShader:declareUniformBlock()
14  local modelViewProj = PerObject:declareUniform(mat4,
15    quote proj*view*model end)
16  local modelViewIT   = PerObject:declareUniform(mat4,
17    quote inverse(transpose(view*model)) end)
18
19
20  local position  = SimpleShader:declareInput(vec3)
21  local normal    = SimpleShader:declareInput(vec3)
22  local uv        = SimpleShader:declareInput(vec2)
23
24  local vNormal   = SimpleShader:declareVarying(vec3)
25  local vUV       = SimpleShader:declareVarying(vec2)
26
27  local outColor = SimpleShader:declareOutput(vec4)
28
29  SimpleShader:addVertexCode(quote
30    Position = modelViewProj * make_vec4(position, 1)
31    vNormal  = (modelViewIT * make_vec4(normal, 0)).xyz
32    vUV = uv
33  end)
34
35  SimpleShader:addFragmentCode(quote
36    var diffuse = texture(diffuseMap, vUV)
37    outColor =
38      diffuse * max(0, dot(vNormal, lightDirection))
39  end)
40
41  SimpleShader:finalize()
```

Listing 2. Lua code to construct the same shader as Listing 1.

Fig. 2. (1) This shader, which a technical artist might write, computes directional lighting (lines 37–38), modulated by a diffuse texture map (line 36). The DSL syntax allows vertex **input**s and fragment **output**s to be declared, along with blocks of **uniform** parameters. Shaders also contain explicit **param**s, which can then be used to set uniforms from within shader code (lines 14–17). Uniforms without initializers automatically become explicit parameters (line 10). Parameters can also contain information about how they should be exposed to artist GUI applications (line 10). Ordinary Terra statements inside **code** blocks are attached to the vertex or fragment kernel, with intermediate values carried by **varying** parameters. `Position` (line 30) is a built-in variable for specifying vertex position (equivalent to GLSL's `gl_Position`). (2) The shader in Listing 1 can also be constructed programmatically, without custom syntax, using our Lua *shader builder* API. Other system components, like Variant Generation and the HLSL/GLSL Backends, use this API to construct and modify shaders.

5.1 Represent Shaders as Compile-time Lua Objects

The biggest implementation difference between Selos and other modern shader systems is our use of a unified shader intermediate representation (SIR) throughout the system. The components of Selos all interface with the same SIR, which contrasts to, e.g., Unity, where the representation of a shader is different for different system components. This key structural difference in our design is a direct consequence of staged metaprogramming because we are able to store code directly in a data structure.

The SIR encodes shaders in terms of Lua objects that exist at compile time only. SIR is a high-level typed representation with detailed semantic information, similar to an Abstract Syntax Tree (AST) representation (as opposed to a low-level assembly-like format).

Since code is first-class in staged metaprogramming, we can store type- and syntax-checked shader code directly in the SIR data structure using quasi-quotes. The SIR Lua object contains a set of members that represent each construct in the shader. Shader inputs, outputs, uniform blocks, etc. are all stored as members in a SIR shader. Along with storing names and type information, a SIR shader also stores metadata about each member, such as bindings/locations for inputs, outputs, uniforms, and textures, as well as which graphical element to display for each artist-editable parameter. All components of Selos operate on the same SIR, as described above.

Because the SIR exists only at compile time, the overhead of operating on it does not affect the performance of the final game executable. This property is guaranteed in our system because Lua code can only be executed at compile time. The ability to act on a unified representation of a shader at compile time differentiates our system from previous work on shader metaprogramming.

5.2 Write Shader Definitions Using a DSL

As Section 4.4 notes, unconstrained use of metaprogramming could lead to code that is more difficult to write, read, maintain, and debug. Therefore, we minimize direct metaprogramming where possible. While staged metaprogramming is the underlying technology of our shader system, the actual metaprogramming code is primarily written by engine developers, not shader writers.

To preserve technical artist and shader writer productivity, Selos provides a custom shader DSL that allows them to author shaders in a familiar style, similar to HLSL and GLSL code. Shader authors write the core logic of a shader in plain Terra code (discussed in Section 5.3) and use the DSL syntax to express shader-specific features that are not inherent in Terra, such as declaring uniforms, inputs, outputs, and textures. Fig. 2 Listing 1 shows a simple shader in our DSL syntax.

Along with hiding metaprogramming concerns from shader writers, other elements of this DSL are also designed in the interest of productivity. In our shaders, artist GUI information is expressed directly alongside uniform parameters (e.g., Fig. 2 Listing 1 line 10), avoiding the double-declaration issue in ShaderLab. By including both vertex and fragment code in the same shader, varying parameters and shared uniform buffers are declared only once as well. Furthermore, our DSL's method of expressing specialization options is akin to that of TFX and Slang (as described in Sections 3.3 and 3.4, respectively). Our method provides greater flexibility than the simple preprocessor-based methods of Unity (Section 3.2). We discuss this method further in Section 5.5.

The implementation of our DSL is driven by staged metaprogramming. Our parser constructs a SIR shader from DSL code by calling into an underlying *shader builder* API, written in (compile-time) Lua code.[6] Other Selos components can use this API to programmatically construct and modify shaders, which does require writing some metaprogramming code directly. Fig. 2 Listing 2 shows how the shader from Fig. 2 Listing 1 can be constructed using the builder API. Note the explicit use of key staged metaprogramming features (listed in Section 4.1) – the **`quote`** keyword specifies the creation of a Terra quasi-quote, the code inside the quote is written as "just plain code," and the quoted code is added directly to the builder data structure.

❧

Because staged metaprogramming provides a favorable balance between code manipulation capabilities and the effort required to use them, we implemented the features of our DSL with only a modest development effort. Table 1 compares lines of code for the Selos implementation against the ShaderLab and Slang implementations. The ShaderLab and Slang compilers most closely relate to our SIR, DSL, and Builder API implementations.

ShaderLab and Selos require a comparable amount of code (but Selos provides additional benefits as discussed above), while Slang consists of a significantly larger codebase because it required building/modifying an HLSL compiler. For Selos, we also have to implement HLSL and GLSL backends to support writing shader code in Terra (Section 5.3). We believe these backends are not engine-specific and can be shared between multiple shader systems as an open-source component, similar to hlsl2glslfork [Pranckevičius 2013].

While lines-of-code metrics are not standalone proof of the effort required to use a programming technique, Table 1 suggests that staged metaprogramming is similar in complexity to using textual-based preprocessing methods like in ShaderLab (given that Terra is C-like and Lua is an imperative language commonly used in games). Furthermore, modifying Slang to implement additional features requires understanding how those changes interact with every existing language feature in a complex compiler with a large codebase, whereas adding features to ShaderLab or Selos requires understanding significantly fewer interactions in a much smaller body of code.

Table 1. Lines of code for various Selos components, as well as for Unity's ShaderLab DSL implementation and the Slang compiler (v0.12.6). We report only non-commented, non-empty lines for Selos and Slang, as reported by CLOC.[7] *The Unity count was obtained via personal communication and is estimated to include 10–15% blank lines and comments [Pranckevičius 2016].

System Component	Language(s)	Lines of Code
Unity ShaderLab DSL	Flex/Bison/other	~2000*
Slang Compiler	C++	~67,000
Selos		
SIR/DSL/Builder	Lua-Terra	~2300
HLSL/GLSL Backend	Lua-Terra	~2200

5.3 Write Shader Logic and Application Code in the Same Language

As GPU shader cores continue to evolve to support more general-purpose code, the distinction between general purpose systems languages and special purpose shading languages becomes less relevant. Therefore, in Selos, we use the same language for both the game runtime application and for shader code. Both are written in Terra and can use the same types and functions, both system- and user-defined, which increases programmer productivity. The only exception is that shader code cannot use Terra constructs that are unsupported in the target shading languages (e.g., pointers).

In addition, Selos provides implementations of special types (and functions) commonly found in shading languages, such as vector, matrix, and texture types. We implement these types as Terra structs,

[6]We implement the actual parsing functionality using Terra's syntax extension mechanisms: http://terralang.org/api.html#the-language-and-lexer-api

[7]http://cloc.sourceforge.net/

meaning that they are usable in application code as well.[8] Other shader systems typically provide a CPU-side vector and matrix library that is distinct from (but compatible with) the shader's equivalent types. In contrast, because Selos's vector and matrix types are used exactly the same in both CPU and GPU code, programmers need not worry about any differences between the CPU and GPU types. When used in shader code, however, our backend code generators replace these structs with the built-in equivalents in the target language to ensure no overhead is added by our abstraction. Sharing types and functions between CPU and GPU code allows programmers to debug shader code by running it on the CPU. Furthermore, it is easier to migrate compute-intensive CPU code to GPU compute shaders.

Our decision to write shader logic in Terra was motivated not only by the benefits of heterogeneity but also by the ease of implementing cross-compilation to HLSL and GLSL using staged metaprogramming. Since we could encapsulate shader logic in quasi-quotes, we did not need to implement a frontend to parse and separate out shader code. Instead, Terra's language frontend parses and syntax-checks the quotes, which are then store in the SIR. Our backend code generators convert these quotes into human-readable HLSL and GLSL (which helps facilitate debugging). Staged metaprogramming allows us to directly reuse Terra's frontend and AST for the statements, expressions, and types used within shader code, minimizing the development effort needed to add shader support to Terra. Thus, our backends required only a modest amount of code (Table 1).

Most importantly, we were able to implement all of this functionality in user-space code, without modifying the Lua-Terra compiler. In contrast, attempting to create something similar in C++ today would require a custom compiler implementation.

5.4 Generate Runtime Data Structures for Shaders

Given that shader and application code are both written in Terra, we can easily extract parameter information from the SIR to generate a Terra struct for each shader. Our system generates these structs at application compile time to provide a static, type-checked interface for the runtime application to set shader parameters. This interface accounts for shader packing rules, so that users do not have to manually navigate data across the CPU-GPU boundary. Furthermore, it allows us to catch more errors at application compile time (like the example in Section 4.2).

The effort to implement this functionality was minimized because we could utilize semantic information provided directly by the staged metaprogramming features and because all types are the same in both CPU and GPU code by default (Section 5.3). In contrast, other systems may similarly generate C++ structs from constant buffers, but doing so requires parsing underlying HLSL/GLSL code and accounting for type differences between the host and shading languages, which requires greater implementation effort.

In addition, these generated structs contain CPU-side setup logic that is expressed with shader code (e.g., lines 14–17 in Fig. 2 Listing 1). These code expressions are stored as quasi-quotes in the SIR and are later inserted into the game application code using unquote. This feature allows shader writers to expose one set of parameters to artists, and then use the artist-configured values to precompute data on the CPU prior to sending the data to the GPU. Both TFX, as well as the renderer used in Far Cry 5 [McAuley 2018], provide similar functionality.

The TFX compiler implements this functionality using an HLSL interpreter. The CPU logic is extracted from the TFX shader file and interpreted at application runtime to set the shader parameters appropriately. In the Far Cry 5 system, a programmer writes a Lua script to calculate shader parameters from artist inputs. This script is loaded and executed at game runtime. Both of these implementations required extra infrastructure to provide this additional functionality. In contrast, our system utilizes staged metaprogramming's quasi-quote and unquote, thus requiring minimal effort to implement.

However, one downside to our approach is that changes to a shader's interface or the CPU-side logic requires recompiling the game executable.[9] TFX and Far Cry 5 do not have this downside, since they support loading shaders dynamically at runtime. Lua-Terra supports just-in-time (JIT) compilation of Terra code, so we could use this functionality to support dynamically loading shaders if desired. Also, if JIT compilation is disallowed (e.g., on consoles), then a system like ours could fall back to an interpreter, as is used by TFX and Far Cry 5.

Dynamic loading allows for more rapid iteration; however, the added validation of a static, type-checked interface to shaders reduces the likelihood of errors caused by out-of-sync shader and application code. An interesting area of future work is to combine these approaches using a system that dynamically recompiles changes to source files. Users could make runtime modification to shaders that would be type-checked and recompiled into the application behind the scenes.

5.5 Implement Complex Specialization Options Using Staged Metaprogramming Constructs Directly

While our previous design decisions emphasize hiding much of the metaprogramming from shader writers, direct use of staged metaprogramming constructs like quasi-quotation enables greater flexibility when expressing specialization options. Therefore, we encourage engine developers and shader writers with a more technical background to use these constructs directly when creating parts of the shader library that have interesting specialization decisions. As we will show in Section 6, this decision enables us to explore both static and dynamic composition of features, which has performance implications.

However, *using* components with complex specialization options should still be straightforward for end users. Therefore, Selos exposes these components to shaders and controls their specializations through `ConfigurationOptions`. We showed an example of this functionality in Section 4.2, so we omit such a discussion here. By using the `ConfigurationOptions`, our system allows experienced developers to use direct staged metaprogramming to implement

[8]We heavily utilized metaprogramming when implementing the CPU-side versions of the HLSL/GLSL built-in types and functions, which greatly reduced the effort required. For examples, see https://github.com/kseitz/selos/blob/master/src/builtin.t

[9]If only the core GPU logic of a shader changes, then we need not recompile the application. Selos can also hot reload shaders when only the GPU logic changes.

complex specialization options, while hiding the intricacies from end users.

To explore a design alternative, we also implemented functionality similar to Unity's `#pragma` system. Shader writers express shader features using syntax similar to Unity's `#pragma multi_compile` and preprocessor `#ifdef`s, and compilation is controlled programmatically through the SIR. Unlike in Unity, our version has the ability to syntax- and type-check each feature in isolation (rather than having to compile all possible variants). In addition, because our version does not treat shader code as a black box (whereas Unity does), it is able to make more interesting choices when generating specializations (such as those presented in Section 6).

However, using direct staged metaprogramming coupled with `ConfigurationOptions` provides greater flexibility and results in simpler shaders. In the `#pragma`-like design, all shader features must be written within the same shader, resulting in complicated and bloated shaders (as they would be in Unity as well). Nevertheless, C preprocessor-like mechanisms can sometimes be useful for expressing straightforward specialization decisions, and we can use such mechanisms alongside our recommended design.

6 EXPLORING THE SPECIALIZATION DESIGN SPACE

6.1 Background and Motivation

Through staged metaprogramming, Selos allows us to target challenging problems faced by designers and users of modern shading systems. One major technique for increasing performance is shader specialization, which takes an input shader (or shaders) that may express rendering code for many different options (e.g., various material types, light types, and platform-specific optimizations) and generates final kernel code by outputting a subset of those options, based on some compile-time parameters. We refer to a specialized kernel as a *variant* of the original input shader. The goal of specialization is to increase the performance of final kernel code by optimizing away unused code paths, which eliminates unnecessary computation, reduces register pressure, and allows for more backend compiler optimization opportunities.

Sometimes, however, complete static specialization is not feasible. For example, when performing shading in a deferred renderer, different pixels might require different material or lighting features; shaders must use dynamic branches to enable or disable features per-pixel.

When complete specialization may be unfeasible, some specialization can still be beneficial. The renderer used in Naughty Dog's Uncharted 4 specializes shaders to the features needed on a per-tile basis (where a tile is a 16×16 group of pixels) [El Garawany 2016]. For example, if no pixels in a given tile contain fabric, then the renderer uses a shader variant that removes fabric-related code when rendering that tile. Additionally, if all pixels in a tile use the exact same set of features, then a "branchless" variant is used, which removes the runtime `if`s around each feature.

This approach can be extended to include specialization based on light types. Some games implement light culling by generating a per-tile list of lights that are known to affect that tile. When shading a tile in the deferred pass, the shader will use the tile's light list, rather than computing lighting for all lights in the scene. Similar to Uncharted 4's material specialization, if a given tile's list has no lights of a given type, then we can use a shader that omits the code for that light type when shading that tile.

However, overspecialization can lead to negative consequences. Generating the full set of shaders for all combinations of material and light types results in a combinatorial explosion of shader variants. Instead, we may wish to statically specialize only a subset of the features in order to decrease the number of shader variants (which would decrease game load time, shader switching overhead, dispatch overhead, etc.). We, thus, would like to explore the tradeoffs between compile-time and runtime specialization in order to achieve the best performance; however, the variant design space is large, so automatically exploring this tradeoff is essential.

While implementing such an exploration using the C preprocessor is challenging and requires shader writers to explicitly plan for it, we can implement this technique in Selos completely in engine library code without manual changes to shaders.

6.2 Experimental Setup

To demonstrate the benefits of this approach, we implemented a tiled deferred renderer and used it to render the ORCA Sun Temple scene [Epic Games 2017]. However, this scene does not specify what type of BRDF to use for each material in the scene (nor do other widely available test scenes). In order to be representative of modern games, which use a variety of material and light types throughout, we render the scene as follows:

- Most objects use our *StandardMaterial*, based on Falcor's [Benty et al. 2018] diffuse and specular BRDFs (using the Frostbite diffuse term).
- Since many objects have a clear coat layer on them, the pedestals use a *StandardMaterialWithClearCoat* type, which adds Filament's [Guy and Agopian 2019] clear coat model on top of our StandardMaterial.
- We render the angel statues as if they were made of marble by using a *SubsurfaceScattering* type, based on Filament's subsurface model but using our diffuse and specular terms.
- We drape instances of a cloth model (based on the model provided with Filament) on top of the angel statues and render them with a *ClothMaterial* type, also based on Filament's.

We use the lights as specified in the Falcor scene file: one *DirectionalLight* and thirteen *PointLights*. We replace two of the point lights with *ShadowedPointLights*, since games typically render shadows for only a subset of point lights. Our implementation of these light types are based on Falcor's. Fig. 3 shows an image of our scene.

In order to find the optimal tradeoff between reduced register pressure from specialization versus decreased shader switching overhead from using fewer, more general shaders, we must determine which material and light types are the most important to specialize. Therefore, we generate all combinations of specializations where only k features are specialized, for all values of k where $0 \le k \le n$ and $n = 6$ (number of material types + number of cullable light types). This generation results in $\binom{n}{k}$ variant sets for each k.

Generating these variant sets was straightforward in Selos, due to staged metaprogramming. We authored two

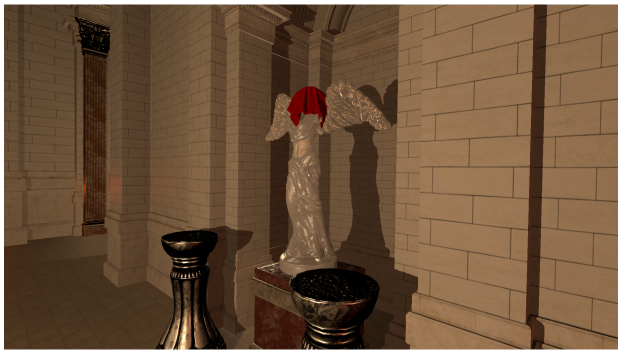


Fig. 3. The test scene used for evaluating different sets of shader variants. This scene is a modified version of the ORCA Sun Temple, in which we added red cloth to the angel statues.

`ConfigurationOptions` that control how specialization options compile into shader variants: `TiledDeferredMaterialType` for material types and `TiledLightListEnv` for light types. The system specifies which type(s) to include in a given variant, and these implementations modify the SIR of the deferred shader accordingly to express that specialization. This functionality is possible because the `ConfigurationOptions` know what material and light types are available, have access to the code for these types via quasi-quote, and can splice together the correct combination of quotes into the shader (with runtime branches inserted to select which code to run on a per-pixel basis). We present and explain further details of our specialization implementation in the Supplementary Materials.

6.3 Performance Results

We run our deferred renderer on the modified Sun Temple scene for each variant set and present the results in Fig. 4 for the best performing set for each value of k.[10] We also hand authored an HLSL shader equivalent to the fully-general case (as is the default for deferred rendering) and compare its performance in Fig. 4 as well. Because the complexity of shaders used in games can vary widely, we emulate increasing shader complexity by (redundantly) computing lighting within a shader 1, 2, 5, and 10 times [Clarberg and Munkberg 2014]. Furthermore, games often use many more lights than the 14 in our test scene. In some scenes, Battlefield 4 has up to 40 lights per tile [Andersson 2011], Detroit: Become Human has 124 lights [Marchalot 2018], and Doom has ~300 light sources [Sousa and Geffroy 2016].

For this particular combination of scene, material types, light types, hardware, etc. the best GPU performance was achieved by specializing either three or four features (Fig. 4a). In addition, most of the benefits of specialization can be achieved by specializing only one or two features (resulting in 2 or 4 total variants). The ClothMaterial type was in the best performing variant set in all cases, but the second feature differed based on shader complexity. Furthermore, the impact of specialization increases with shader complexity.

Beyond improving GPU performance, using fewer variants has additional benefits. Whenever shader code changes, all affected variants must be recompiled, so using fewer variants saves build time. Game load times are improved too, because fewer variants need to be loaded. In addition, as shown in Fig. 4b, runtime CPU overhead increases as the number of variants increase. Thus, developers may wish to trade off GPU performance to save CPU cycles, or vice versa.

Finally, the performance of the fully-general handwritten HLSL shader is comparable to that of the fully-general shader generated by our system. Therefore, the code generation and manipulation that Selos performs does not negatively impact the performance of final shader code.

Staged metaprogramming allowed us to easily build a tool to explore compile-time and runtime specialization in a principled and straightforward way. Because the performance tradeoffs in the specialization design space depend on the game, shader features, scene, platform (including D3D11 vs. OpenGL, operating system, drivers, CPU, GPU, etc.), and other variables, exploiting automation is essential to achieve the best performance across various configurations. Using staged metaprogramming, we are able to rapidly explore the specialization design space, without requiring shader writers to explicitly include code for each case in the shaders.

While we have demonstrated one potential method for investigating the shader permutation problem, exploring this issue more fully is an interesting area of future work. We believe staged metaprogramming provides the proper abstraction for solving this and other types of issues faced by game engine developers.

7 RELATED WORK

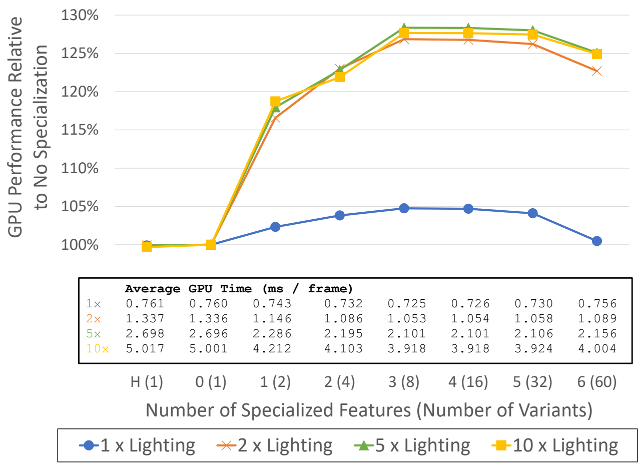
7.1 Extended Shader Programming Models

A wide range of alternative programming models have been developed on top of the baseline interface provided by graphics APIs. Often, the primary goal of these systems is to improve the software-development productivity of shader programmers while simultaneously maintaining high performance.

The Real-Time Shading Language (RTSL) system [Proudfoot et al. 2001] shows that a high-level shading language, inspired by the RenderMan Shading Language (RSL) [Hanrahan and Lawson 1990], can be compiled, with good efficiency, for early programmable hardware. Abstract shade trees [McGuire et al. 2006] build upon the idea of Shade Trees [Cook 1984], to enable composition of real-time shaders from separately-developed pieces. Spark [Foley and Hanrahan 2011] extends the approach of RTSL to modern rasterization pipelines, improving support for modular software development.

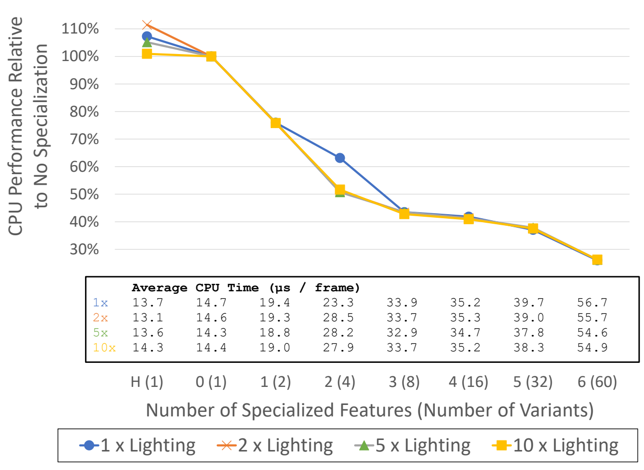
Spire [He et al. 2016] demonstrates that a suitable high-level shader IR can allow complex rate-placement optimizations to be applied automatically. Our exploration of specialization decisions (Section 6) is conceptually similar, where one of the rates involved is "constant." We believe that this kind of exploration is important, and is just one example of the kinds of shader optimizations tools developers can build with staged metaprogramming.

[10]These results were produced using a resolution of 1920×1080 pixels with a tile size of 16×16, on a computer running Windows 10 with an Intel Core i7-6700K CPU and an NVIDIA GeForce GTX 1080 GPU. We benchmarked every 100th frame (30 frames total over the 50 second camera path).



Average GPU Time (ms / frame)	H (1)	0 (1)	1 (2)	2 (4)	3 (8)	4 (16)	5 (32)	6 (60)
1x	0.761	0.760	0.743	0.732	0.725	0.726	0.730	0.756
2x	1.337	1.336	1.146	1.086	1.053	1.054	1.058	1.089
5x	2.698	2.696	2.286	2.195	2.101	2.101	2.106	2.156
10x	5.017	5.001	4.212	4.103	3.918	3.918	3.924	4.004

(a) GPU Performance



Average CPU Time (μs / frame)	H (1)	0 (1)	1 (2)	2 (4)	3 (8)	4 (16)	5 (32)	6 (60)
1x	13.7	14.7	19.4	23.3	33.9	35.2	39.7	56.7
2x	13.1	14.6	19.3	28.5	33.7	35.3	39.0	55.7
5x	13.6	14.3	18.8	28.2	32.9	34.7	37.8	54.6
10x	14.3	14.4	19.0	27.9	33.7	35.2	38.3	54.9

(b) CPU Performance

Fig. 4. GPU and CPU performance for the deferred pass, relative to using no specialization (higher is better). Also shown in the tables are the absolute GPU and CPU times (lower is better). This data was gathered using our Direct3D 11 implementation and the test scene described in Section 6 (Fig. 3). While each number of specialized features has multiple possible combinations of shader variants, we display results for the best performing variant set, based on GPU time. We also compare against a handwritten HLSL shader with no specialization (presented as H in the graphs), which is representative of a typical deferred shader implementation. We repeat lighting calculations within the shaders 1, 2, 5, or 10 times to emulate increasing shader complexity and because games often have many more lights than in our scene. As complexity increases, specialization has a greater positive impact on GPU performance. For this particular scene and set of specialization options, the best performance is achieved using a partially specialized variant set, and most of the benefits of specialization can be achieved by specializing only one or two features. More specialization results in worse CPU performance, because the average number of compute shader dispatches per frame increases (thus causing more CPU overhead). The design of our shader system, and specifically our use of staged metaprogramming, made this exploration possible.

Effect systems expand the scope of shaders beyond programmable pipeline stages to include configuration of fixed-function state [Microsoft 2010; NVIDIA Corporation 2010], and even abstraction over multiple rendering passes [Lejdfors and Ohlsson 2004].

Each of these projects, representing different visions of what shader programming should be, is implemented as a stand-alone system, often with considerable effort. We believe that staged metaprogramming provides an approach that reduces the cost of implementing novel programming models like these, whether in research or production.

7.2 Multi-Stage Programming and Syntax Extension

As described in Section 4.1, we take our definition of multi-stage programming from Taha [1999]. Our Selos implementation was built using Terra [DeVito et al. 2013], which adds multi-stage constructs and low-level programming support to the Lua language [Ierusalimschy et al. 1996], along with user-defined syntax extensions.

The BraidGL language [Sampson et al. 2017] uses the syntax of staging for both metaprogramming and to map communication to the vertex and fragment stages of a rasterization pipeline, in a manner similar to *rates* (as used in RTSL, Spark, and Spire). The design of BraidGL promotes staging as a language mechanism to be used by most shader writers. In contrast, Selos promotes staging as a mechanism to be used in implementing shader systems, but most shader writers need not use or understand it directly.

Rust [Rust Project Developers 2015] is a systems programming language that supports limited syntax extension using a macro system in the tradition of Scheme [Flatt 2002; Sussman and Steele 1998].

Some projects have attempted to add multi-stage programming features to existing systems programming languages like C/C++. ‘C (“tick C”) [Engler et al. 1996] adds a quasiquotation construct to C, with a focus on code generation at run-time (similar to MetaML [Taha and Sheard 2000] and MetaOCAML [Calcagno et al. 2003]). OpenC++ adds compile-time code generation and limited syntactic extension to C++ using a metaobject protocol [Chiba 1995]. The extension-oriented compiler Xoc [Cox et al. 2008] allows extensions to the syntax and semantics of C to be loaded dynamically by the compiler. However, the current (and upcoming) C and C++ standards do not have such features, and we therefore could not use these languages to implement this work. Fortunately, ongoing C++ projects and proposals continue to explore the types of facilities needed for staged metaprogramming (Section 8).

7.3 Shader Metaprogramming

Several previous systems have applied metaprogramming techniques to shaders. For example, the PyFX system [Lejdfors and Ohlsson 2004] uses Python code to compose multi-pass effects from Cg shaders. Shaders are authored as strings, and the system extracts parameter data to expose named parameters to Python code.

Sh [McCool et al. 2002] and Vertigo [Elliott 2004] expose shaders as an embedded DSL (eDSL) in C++ and Haskell, respectively. Special types are used to express shader code, and operators on those types are overloaded to construct an intermediate representation (IR).

Arbitrary code running in the host language can be used to generate or specialize the shader code. In Sh, the host and shader languages use distinct syntax for control flow constructs (shader control flow is expressed with macros). In both systems, the type checking rules of the host language are used to guarantee type safety of generated shader code, and they can also provide for statically-checked setting of shader parameters.

These previous systems may be viewed as examples of *runtime* staged metaprogramming. In each case, runtime application code (in Python, C++, or Haskell) in the first stage is used to synthesize shader code for subsequent execution. While shader code is *embedded* in the application language (whether as strings of Cg source code, or via macros and overloaded operators), they belong to different stages, and so cannot easily share types or subroutines.

Our approach, based on *compile-time* staged metaprogramming, differs from prior work in two key ways. First is the simple fact that we perform code generation and manipulation tasks at compile time, which reduces run-time costs and enables deployment on platforms where runtime code generation is either disallowed or not advised. Furthermore, metaprogramming code running in the compile-time stage has access to more complete information about source locations and symbol names than is available at runtime, allowing engine-specific services to emit higher-quality diagnostic messages (errors and warnings).

Second, and more fundamentally, runtime application and shader code in Selos are expressed in both the same language and same stage of execution; both are written in Terra, and they can share types and subroutines. In contrast, prior shader systems using staged metaprogramming separate CPU and GPU code into distinct stages with distinct languages, libraries, etc. Our approach is thus more similar to CUDA [NVIDIA Corporation 2007], where CPU and GPU compute code are deeply integrated using the same language and execution stage.

Partial evaluation [Futamura 1982] is a concept related to metaprogramming. Rodent [Pérard-Gayot et al. 2019] utilizes partial evaluation to generate a specialized renderer for a given scene description. Both Rodent and Selos are motivated by reducing the effort required to implement rendering frameworks. However, we focus on real-time rendering, whereas their emphasis is on offline path tracing. Our work provides a clear distinction between compile-time code and runtime code, so real-time graphics developers can be confident that expensive operations happen at compile time. In contrast, a partial evaluator will evaluate as much as it can based on data available to it at compile time, but it does not provide strong guarantees about what it will or will not evaluate.

8 THE FUTURE OF METAPROGRAMMING IN C++

Some recent proposals to the C++ Standards Committee seek to add more robust and powerful metaprogramming facilities to the language. P0194 [Chochlik et al. 2018] proposes adding support for compile-time reflection to C++ by having the compiler generate meta-object types that represent certain program declarations. These meta-object types can be used at compile time to obtain information about the program being compiled. This functionality is akin to the introspection abilities of staged metaprogramming.

The authors of P0633 [Vandevoorde and Dionne 2017] explore the design space for metaprogramming in C++, looking at aspects of reflection, code synthesis, and control flow constructs. For example, they discuss supporting raw string injection, where arbitrary strings could be consumed by the compiler to generate code. They presume that the compiler would provide local scoping when translating these string to avoid variable capture issues. Since strings are first-class citizens in C++, this functionality could mimic a quasi-quote construct (albeit without the syntax-checking guarantees, since the underlying representation would still be just strings).

Metaclasses [Sutter 2018] would allow programmers to write new class features as “just code,” without requiring compiler modifications for these features. A programmer could write compiler-enforced patterns, requiring that all instances of the metaclass adhere to certain constraints. We are interested to see if we could create a metaclass for shaders using this functionality.

The Circle compiler [Baxter 2019] extends C++17 by including new introspection, reflection, and compile-time execution features. For example, one can introspect a struct, extract the parameters from it, and then generate new code based on these parameters, all using regular C++ syntax. We believe that some parts of Selos could be implemented using Circle, given that it meets some of the criteria for staged metaprogramming. However, it lacks a quasi-quote construct at present and, thus, is not a full staged metaprogramming environment.

These projects represent an increasing interest in evolving C++ toward better metaprogramming features. While they do not yet enable staged metaprogramming in C++, they are a step in the right direction. In the future, we hope that staged metaprogramming becomes a staple in modern systems programming languages.

9 CONCLUSION

In this paper, we have demonstrated how staged metaprogramming provides the proper facilities with which to build an expressive shader system, complete with a unifying shader intermediate representation, the ability to express heterogeneous code within a shader, and cross compilers for HLSL and GLSL. We also showed an example of using staged metaprogramming to explore the shader variant design space, which increased performance for our test scene by determining which features were most important to specialize, thus preventing overspecialization. Implementing these system components required only a modest effort, thanks to staged metaprogramming.

Beyond the components presented here, staged metaprogramming provides the flexibility to implement many more types of designs, such as graphical node-based material editors (e.g., Unreal Engine’s Material Editor). Furthermore, the shader permutation problem is far from solved, so using staged metaprogramming to implement new solution ideas is an interesting area for future work.

We hope that in the future, GPU shader code will be a first-class construct in mainstream systems programming languages, just as CUDA gives GPU compute code first-class treatment in C++. However, even if shaders are better supported in modern systems languages, graphics programming is far from achieving heterogeneity similar to CUDA. In fact, the newer graphics APIs (Direct3D 12 [Microsoft 2017] and Vulkan [Khronos Group 2016]) push graphics further away from this heterogeneity by requiring lower-level management of GPU states and resources. Exploring ways to integrate the performance benefits of these APIs into a heterogeneous environment is a challenging endeavor.

GPU-based graphics provides a complex and well-explored domain in which to investigate the broader concept of heterogeneous programming. In a future with potentially many different processor types (e.g., accelerators for high-performance machine learning), we will need programming models that enable many domains to efficiently utilize a wide range of heterogeneous processing resources. The lessons learned while studying heterogeneous graphics programming will help inform such future designs. Our experience leads us to believe that staged metaprogramming will be an important feature of such future heterogeneous systems.

ACKNOWLEDGMENTS

The authors would like to thank the many people who gave us valuable feedback on this work throughout its development, including Anjul Patney, Ahmed Mahmoud, Alex Kennedy, Angelo Pesce, Aras Pranckevičius, Brett Lajzer, Brian Karis, Dave Shreiner, Hugues Labbe, Joe Forte, Michael Vance, Padraic Hennessy, Paul Lalonde, Wade Brainerd, Zachary DeVito, and the anonymous reviewers. Thanks also to Chuck Lingle for facilitating feedback and support from Intel and to Francois Demoullin for contributions during earlier stages of this project.

We are grateful to Intel Corporation and the National Science Foundation Graduate Research Fellowship Program for financial support of this project, and for hardware donations from Intel and NVIDIA.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-1148897. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Johan Andersson. 2011. DirectX 11 Rendering in Battlefield 3. Game Developers Conference 2011. http://www.dice.se/news/directx-11-rendering-battlefield-3/

Alan Bawden and Jonathan Rees. 1988. Syntactic Closures. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming (LFP '88)*. 86–95. https://doi.org/10.1145/62678.62687

Sean Baxter. 2019. Circle. https://github.com/seanbaxter/circle

Nir Benty, Kai-Hwa Yao, Tim Foley, Matthew Oakes, Conor Lavelle, and Chris Wyman. 2018. The Falcor Rendering Framework. https://github.com/NVIDIAGameWorks/Falcor https://github.com/NVIDIAGameWorks/Falcor.

David Blythe. 2006. The Direct3D 10 System. *ACM Transactions on Graphics* 25, 3 (July 2006), 724–734. https://doi.org/10.1145/1141911.1141947

Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE '03)*. 57–76. https://doi.org/10.1007/978-3-540-39815-8_4

Shigeru Chiba. 1995. A Metaobject Protocol for C++. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '95)*. 285–299. https://doi.org/10.1145/217838.217868

Matus Chochlik, Axel Naumann, and David Sankel. 2018. Static reflection. C++ Standards Committee Papers. http://wg21.link/p0194

Petrik Clarberg and Jacob Munkberg. 2014. Deep Shading Buffers on Commodity GPUs. *ACM Transactions on Graphics* 33, 6, Article 227 (Nov. 2014), 12 pages. https://doi.org/10.1145/2661229.2661245

Robert L. Cook. 1984. Shade Trees. In *Computer Graphics (Proceedings of SIGGRAPH 84)*. 223–231.

Russ Cox, Tom Bergan, Austin T. Clements, Frans Kaashoek, and Eddie Kohler. 2008. Xoc, an Extension-oriented Compiler for Systems Programming. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*. 244–254. https://doi.org/10.1145/1346281.1346312

Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A Multi-Stage Language for High-Performance Computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. 105–116. https://doi.org/10.1145/2491956.2462166

Ramy El Garawany. 2016. Advances in Real-time Rendering, Part I: Deferred Lighting in Uncharted 4. In *ACM SIGGRAPH 2016 Courses (SIGGRAPH '16)*. http://advances.realtimerendering.com/s2016/index.html

Conal Elliott. 2004. Programming Graphics Processors Functionally. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell '04)*. 45–56. https://doi.org/10.1145/1017472.1017482

Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. 1996. 'C: A Language for High-level, Efficient, and Machine-independent Dynamic Code Generation. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. 131–144. https://doi.org/10.1145/237721.237765

Epic Games. 2017. Unreal Engine Sun Temple, Open Research Content Archive (ORCA). https://developer.nvidia.com/ue4-sun-temple

Epic Games, Inc. 2019. Unreal Engine 4 Documentation. https://docs.unrealengine.com/en-us/

Matthew Flatt. 2002. Composable and Compilable Macros: You Want It When?. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. 72–83. https://doi.org/10.1145/581478.581486

Tim Foley and Pat Hanrahan. 2011. Spark: Modular, Composable Shaders for Graphics Hardware. *ACM Transactions on Graphics* 30, 4 (July 2011), 107:1–107:12. https://doi.org/10.1145/2010324.1965002

Yoshihiko Futamura. 1982. Partial Computation of Programs. In *RIMS Symposium on Software Science and Engineering, Kyoto, Japan, 1982, Proceedings*. 1–35. https://doi.org/10.1007/3-540-11980-9_13

Romain Guy and Mathias Agopian. 2019. Filament. https://github.com/google/filament https://github.com/google/filament.

Pat Hanrahan and Jim Lawson. 1990. A Language for Shading and Lighting Calculations. In *Computer Graphics (Proceedings of SIGGRAPH 90)*. 289–298.

Yong He, Kayvon Fatahalian, and Tim Foley. 2018. Slang: Language Mechanisms for Extensible Real-time Shading Systems. *ACM Transactions on Graphics* 37, 4, Article 141 (July 2018), 13 pages. https://doi.org/10.1145/3197517.3201380

Yong He, Tim Foley, and Kayvon Fatahalian. 2016. A System for Rapid Exploration of Shader Optimization Choices. *ACM Transactions on Graphics* 35, 4, Article 112 (July 2016), 12 pages. https://doi.org/10.1145/2897824.2925923

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. 1996. Lua—An Extensible Extension Language. *Software: Practice and Experience* 26, 6 (June 1996), 635–652. https://doi.org/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P

John Kessenich and Boaz Ouriel. 2018. *SPIR-V Specification (Version 1.00)*. https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf.

Khronos Group. 2016. *Vulkan 1.0.12 - A Specification*. https://www.khronos.org/registry/vulkan/specs/1.0/pdf/vkspec.pdf.

Calle Lejdfors and Lennart Ohlsson. 2004. PyFX – An active effect framework. In *Proceedings of SIGRAD 2004*. Linköping University Electronic Press, 17–24. http://www.ep.liu.se/ecp/article.asp?issue=013&article=006

Ronan Marchalot. 2018. Cluster Forward Rendering and Anti-Aliasing in 'Detroit: Become Human'. Game Developers Conference 2018. https://www.gdcvault.com/play/1025420/Cluster-Forward-Rendering-and-Anti

Stephen McAuley. 2018. Advances in Real-time Rendering in Games, Part I: The Challenges of Rendering an Open World in Far Cry 5. In *ACM SIGGRAPH 2018 Courses (SIGGRAPH '18)*. ACM. http://advances.realtimerendering.com/s2018/index.htm

John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (April 1960), 184–195. https://doi.org/10.1145/367177.367199

Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. 2002. Shader Metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '02)*. 57–68. http://dl.acm.org/citation.cfm?id=569046.569055

Morgan McGuire, George Stathis, Hanspeter Pfister, and Shriram Krishnamurthi. 2006. Abstract Shade Trees. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games (I3D '06)*. 79–86. https://doi.org/10.1145/1111411.1111425

Microsoft. 2010. Effect Format (Direct3D 11). (2010). https://msdn.microsoft.com/en-us/library/windows/desktop/ff476118(v=vs.85).aspx.

Microsoft. 2017. Direct3D 12 Programming Guide. https://docs.microsoft.com/en-us/windows/desktop/direct3d12/directx-12-programming-guide

Microsoft. 2019. DirectX Shader Compiler. https://github.com/Microsoft/DirectXShaderCompiler https://github.com/Microsoft/DirectXShaderCompiler.

NVIDIA Corporation. 2007. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. (Jan. 2007). http://developer.nvidia.com/cuda.

NVIDIA Corporation. 2010. Introduction to CgFX. (2010). http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html.

Arsène Pérard-Gayot, Richard Membarth, Roland Leißa, Sebastian Hack, and Philipp Slusallek. 2019. Rodent: Generating Renderers without Writing a Generator. In *ACM Transactions on Graphics (TOG)*, Vol. 38. 40:1–40:12. https://doi.org/10.1145/3306346.3322955

Aras Pranckevičius. 2013. HLSL to GLSL shader language translator. https://github.com/aras-p/hlsl2glslfork

Aras Pranckevičius. 2016. Personal Communication.

Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. 2001. A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *Proceedings of SIGGRAPH 2001*. 159–170. https://doi.org/10.1145/383259.383275

Rust Project Developers. 2015. *The Rust Programming Language*. https://doc.rust-lang.org/book/.

Adrian Sampson, Kathryn S. McKinley, and Todd Mytkowicz. 2017. Static Stages for Heterogeneous Programming. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 71 (Oct. 2017), 27 pages. https://doi.org/10.1145/3133895

Mark Segal, Kurt Akeley, Chris Frazier, Jon Leech, and Pat Brown. 2015. *The OpenGL© Graphics System: A Specification (Version 4.5 (Core Profile) - May 28, 2015)*. https://www.opengl.org/registry/doc/glspec45.core.pdf.

Tiago Sousa and Jean Geffroy. 2016. Advances in Real-time Rendering, Part II: The Devil is in the Details: idTech 666. In *ACM SIGGRAPH 2016 Courses (SIGGRAPH '16)*. http://advances.realtimerendering.com/s2016/index.html

Gerald Jay Sussman and Guy L. Steele, Jr. 1998. Scheme: A Interpreter for Extended Lambda Calculus. *Higher-Order and Symbolic Computation* 11, 4 (Dec. 1998), 405–439. https://doi.org/10.1023/A:1010035624696

Herb Sutter. 2018. Metaclasses: Generative C++. C++ Standards Committee Papers. https://wg21.link/P0707

Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science* 248, 1–2 (Oct. 2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

Walid Mohamed Taha. 1999. *Multistage Programming: Its Theory and Applications*. Ph.D. Dissertation. Oregon Graduate Institute of Science and Technology.

Natalya Tatarchuk and Chris Tchou. 2017. Destiny Shader Pipeline. Game Developers Conference 2017. http://advances.realtimerendering.com/destiny/gdc_2017/

Unity Technologies. 2019. Unity User Manual (2019.1). https://docs.unity3d.com/Manual/index.html

Daveed Vandevoorde and Louis Dionne. 2017. Exploring the design space of metaprogramming and reflection. C++ Standards Committee Papers. https://wg21.link/P0633