Toward Unified Shader Programming

By

KERRY A. SEITZ, JR.

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

John D. Owens, Chair

Nina Amenta

Theresa Foley

Committee in Charge

2021

To my parents, Sue and Kerry; and to my sister, Andrea.

CONTENTS

List of Figures

LIST OF TABLES

List of Code Listings

ABSTRACT

Toward Unified Shader Programming

Real-time graphics programming is more complex due to the strict separation of programming languages and environments between host (CPU) code and GPU code. In contrast, popular general-purpose GPU (GPGPU) programming models provide *unified* programming environments, in which both host and GPU code are written in the same language, can be in the same file, and share lexical scopes. Such unified systems avoid code duplication, subtle compatibility bugs, and additional development and maintenance costs that arise from manually coordinating between host code and GPU code. While real-time graphics predates—and, in many ways, inspired—GPGPU programming, it has yet to incorporate the advantages of unified programming that helped to propel the popular GPGPU systems to success.

In this dissertation, we propose two overarching implementation methodologies for creating unified programming environments for real-time graphics. These methods have complementary trade-offs, with one motivated by practical applicability in existing systems today and the other providing a principled approach for new systems utilizing future programming language evolutions. The first method is to co-opt existing features of a programming language and implement them with alternate semantics to express the host-GPU interface requirements and code optimization techniques ubiquitous in real-time rendering. Using this strategy, we integrate GPU shader code into C++ by co-opting annotations, inheritance, and virtual functions to express the shader specialization optimization. Our compiler-based tool transforms host and GPU shader code into efficient implementations by generating specialized shader variants in place of virtual function calls. The second method is a general-purpose language feature called *staged metaprogramming*. Using this technique, we create a unified shader system entirely in user-space code, without the need for a compiler-based implementation. Our use of staged metaprogramming enables exploration of the shader specialization design space, resulting in improved performance relative to the complete, static specialization baseline. Along with presenting these two implementation strategies, we also compare their strengths and shortcomings to better inform graphics programmers seeking to create unified environments today and in the future.

Acknowledgments

Something I learned very early in my graduate student career—and that was continually reinforced throughout—is that my advisor, John D. Owens, genuinely cares about his students. He goes above and beyond to act in their best interests and to help them succeed, and I am sure that all of his students agree. I am so very grateful to him for pushing me to grow, both as a student and as a person, and to have more confidence in myself. I have benefitted immensely from his technical insights, research acumen, admirable patience, and constant support, and I hope to one day pay it forward. I have a hard time imagining my success as a graduate student without him. John, thank you for all that you have done for me, and thank you for being the awesome person that you are.

For practically my entire time in graduate school, Tess Foley has been a collaborator and mentor that I have valued beyond measure. She has contributed to and guided my dissertation research from the start, and her expertise in the field has shaped my understanding of real-time graphics and shader programming. Tess's mentorship has not only strengthened my technical knowledge but also has helped me to be more confident about my understanding of complex topics. Among other things, I look up to her ability to clearly communicate subtle and challenging points both to domain experts and to those less experienced in the field, and I appreciate that she does not shy away from having difficult conversations when they are warranted. I want to thank Tess for all of the time, experience, and advice she has shared with me over the years.

Thank you to Nina Amenta for serving on my dissertation committee, as well as for being a fantastic teacher. I appreciate her enthusiasm and insights that have helped to strengthen my work. Her Parallel Algorithms course is my favorite of all the courses I took during my time as a graduate student! Nina finds a way to explain the complex course material in an approachable manner, while simultaneously fostering deep understanding for her students.

Serban D. Porumbescu has been another wonderful source of guidance over the past several years, both on technical questions as well as on topics related to personal and professional growth. I'm grateful that he always managed to find time to have impromptu conversation with me, and I valued having a go-to person in the lab with whom to discuss day-to-day research challenges. When he joined the staged metaprogramming project, he brought a new perspective that not only influenced and improved the research itself, but also enabled us to communicate our ideas more effectively. It was Serban's idea to use the term "research substrate" to describe our use of Lua-Terra in the staged metaprogramming work, and I credit this suggestion with providing much-needed clarity not just for readers, but for my own views on the work as well.

Throughout my time at UC Davis, I have been fortunate to connect with many industry experts that have influenced both my research work and my career aspirations. I would like to thank Chuck Lingle for facilitating conversations at and financial support from Intel, Brian Karis for project and technical advice and for connections within Epic Games, and Dave Shreiner for feedback on my work and for professional advice. I also wish to thank the many people who provided guidance and feedback for my work, including Andrew Lauritzen, Angelo Pesce, Aras Pranckevičius, Brett Lajzer, Charlie Birtwistle, Hugues Labbe, Joe Forte, Michael Vance, Ola Olsson, Padraic Hennessy, Paul Lalonde, Wade Brainerd, Yong He, Yuriy O'Donnell, Zachary DeVito, and the anonymous reviewers. Thank you to Francois Demoullin for early code contributions on the staged metaprogramming work.

I am grateful for the financial support I have received throughout my graduate student career. Thank you to Intel Corporation and the National Science Foundation Graduate Student Fellowship Program (NSF GRFP) for financial support. Also, thanks to Intel Corporation and NVIDIA Corporation for hardware donations. 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 author(s) and do not necessarily reflect the views of the National Science Foundation.

I also want to thank the fantastic people in the Department of Computer Science and the Department of Electrical and Computer Engineering at UC Davis. The staff has provided excellent support throughout my time as a graduate student, and I have enjoyed learning from the great faculty members in these departments. I especially want to thank Jessica Stoller and Alyssa Bates. Whenever I had a question about anything related to the graduate program, they always responded quickly and with exactly the information I needed! Their hard work and extensive knowledge meant that I never had to worry about questions and issues related to the program, which relieved a great deal of stress that I might have otherwise had.

Over the years, I have had the immense pleasure of watching John's research group grow, and I am glad to have had the opportunity to work and interact with so many wonderful people! I was fortunate to begin my time in the group by working with Anjul Patney and Stanley Tzeng, who were—and continue to be—great sources of knowledge and support. Thank you to everyone who has made my grad school experience fulfilling, memorable, and fun, including Pınar Muyan-Özçelik, Andrew Davidson, Yangzihao Wang, Saman Ashkiani, Afton Geil, Jason Mak, Calina Copos, Edmund Yan, Leyuan Wang, Yuechao Pan, Carl Yang, Vehbi Eşref Bayraktar, Collin McCarthy, Andy Riffel, Muhammad Osama, Chenshan Shari Yuan, Weitang Liu, Yuduo Wu, Ahmed H. Mahmoud, Muhammad Awad, Yuxin Chen, Zhongyi Lin, Matthew Yih, Jonathan Wapman, Agnieszka Łupińska, Radoyeh Shojaei, Chuck Rozhon, Toluwanimi Odemuyiwa, and Teja Aluru. You all are the best!

I could not have had a successful graduate student career without my amazing and caring friends. I met Alex Kennedy during computer science graduate student orientation when he overhead me talking about StarCraft, and I'm so thankful that he joined in on that conversation. From the very start, Alex has been a great friend, providing interesting technical discussions, constant support, and a myriad of great times (and so much ARAM)! It was through Alex that I met another great friend, Brennan Beam. Brennan's open and caring nature had led to him becoming a much-needed confidant for me. My friendships with Alex and Brennan have helped me to relish the good times and to stay afloat during the difficult ones. I also want to thank Mike and Emily Bedwell and Jake and Katy Feldman. Though there are many miles between us, they continue to be the best of friends. Also, thank you to the many people I met through swing dancing and Lindy Hop, both for their friendships and for challenging me to constantly improve my skills. Joining the Davis Swing Dancers and the larger Sacramento swing dance community was one of the best decisions I made in Davis!

Finally, thank you to my family. Throughout my life, my parents, Sue and Kerry, have always loved and supported me, and my success would not be possible otherwise. My sister, Andrea, has also been an immense source of love and support. Despite having known her practically all my life, she still manages to find new and creative ways to make me laugh. I am so fortunate to have wonderful and loving grandparents, aunts, uncles, cousins, and other family members who always make me feel at home, no matter where we are or the distances between us.

Thank you to everyone who has had a positive impact on my life for leading me to a rewarding graduate school experience and beyond!

Chapter 1

Introduction

From video games large and small, to 3D visualization applications, to generating training data for self-driving cars, the field of real-time computer graphics impacts many aspects of our modern technological experience. In the pursuit of ever-increasing visual fidelity, graphics programmers develop massive systems to manage the complexities inherent in real-time rendering. A major aspect of this complexity arises from the need to utilize multiple types of processing resources with different architectural and performance characteristics. Exploring such heterogeneous systems in the context of real-time rendering can help to improve the field of computer graphics, while also teaching us lessons transferrable to other domains wishing to take advantage of the performance and energy-efficiency benefits of heterogeneous computing.

The high-level task of a modern real-time graphics application is to produce images to display on-screen based on underlying data that represents the scene being drawn. This data consists of a set of objects and their properties (such their shapes/geometries, materials, and positions within the scene), as well as lighting information. A renderer produces an image by calculating how illumination from each light interacts with the materials and shapes of the objects in the scene, which ultimately produces a color to display for each pixel on the screen. In order to maintain a fluid, real-time experience, the renderer must produce each image quickly. For example, a typical target is to generate 60 images per second (which translates to a time budget of ~16 milliseconds per image), but some applications, such as those using head-mounted virtual reality displays, require even faster image generation. Modern scenes often contain thousands of objects and hundreds of lights which are used to calculate colors for millions of pixels, meaning that runtime performance of these applications is of utmost importance in order to generate a high-quality image within the limited time budget necessary for a real-time experience.

Because of this high-performance requirement, real-time graphics applications utilize specialized hardware, namely Graphics Processing Units (GPUs), to perform the vast majority of rendering calculations. GPU hardware is highly parallel, and programming models for this hardware are designed to make efficient use of this parallelism. Because GPUs are coprocessors, a host processor (i.e., a CPU) is ultimately responsible for orchestrating the code that runs on the GPU. Therefore, graphics programming consists of both GPU code that performs the highly parallel rendering calculations, as well as host (CPU) code that coordinates and invokes rendering work that uses this GPU code.

Unfortunately, real-time graphics programming is made more complicated by the use of distinct languages and programming environments for host code and GPU code. While host code is typically written in a general-purpose systems language (such as C++), GPU code is authored in a special-purpose shading language (e.g., HLSL [52], GLSL [37], or Metal Shading Language [4]). When using a shading language and its corresponding graphics API (e.g., Direct3D [56], Vulkan/OpenGL [39, 67], or Metal [3]), programmers issue API calls to migrate data between host and GPU memory and to set up and invoke GPU code that uses this data. They must ensure not only that data is transferred efficiently, but also that data availability and layout in GPU memory match what the GPU code expects. Because host and GPU code exist in two separate programming environments, programmers are ultimately responsible for ensuring compatibility between host and GPU code, with little help from the graphics APIs.

In contrast, heterogeneous programming is simpler in a *unified* environment, where both host and GPU code are written in the same language, can be in the same file, and share lexical scopes. For example, in CUDA [57], developers write both host and GPU code in C++, and passing parameters and invoking GPU code looks essentially like a regular function call. Similarly, programmers using C++ AMP [29] author GPU code as C++ lambda expressions and invoke them using API functions, allowing both host and GPU code to coexist within a single C++ function. In these unified systems, host and GPU code can use the same types and functions and reference the same declarations. Thus, these unified systems—by definition—avoid an entire class of compatibility issues that must be handled manually in graphics programming. Enabling unified programming for real-time graphics would allow graphics programers to utilize the inherent code reuse, compatibility, and ease-of-use benefits that such environments provide.

While CUDA and C++ AMP provide powerful unified programming models for General-Purpose GPU (GPGPU) computing, graphics programming has additional challenges that complicate development of a unified model. At its core, rendering attempts to solve the problem of simulating light. Thus, the complexity of graphics programming comes from reality—light interacts with the environment in a variety of ways that result in myriad visual phenomena. These interactions are complex and costly to simulate, so graphics programmers use many different approximations to render visual effects. A given scene may use any number of these effects in various combinations, so graphics programming systems need to enable composition of these approximations. Additionally, such systems should also be extensible so that programmers can implement and integrate new techniques as our understanding of rendering continues to evolve.

Moreover, because performance is crucial in real-time graphics applications, programmers strive to utilize the underlying hardware as efficiently as possible. This task entails not only using specialized hardware (like GPUs) but also tailoring code to the specific performance characteristics of that hardware. One important performance characteristic of modern GPUs is that unused code paths in GPU code can lead to reduced performance by increasing register pressure and precluding potential compiler optimizations. Thus, compositions of various rendering effects should result in code that is *specialized* to the exact set of features used at runtime, thereby removing any code not needed in a particular invocation. Specialization is a pervasive and critically important optimization in real-time graphics—it can have a significant impact on runtime performance [15, 32, 68], major game engines create mechanisms specifically to support it [24, 76], and game developers go to great lengths to enable it even in scenarios where it may not initially seem feasible [17]—but it is not nearly as common in GPGPU applications. Therefore, typical GPGPU programming environments like CUDA do not provide the support for specialization that graphics applications need. Furthermore, programmers using CUDA generally target code to only a limited set of hardware (namely, NVIDIA GPUs), whereas developers using real-time graphics for 3D games often target a wide range of hardware platforms, from low-end mobile devices, to various gaming consoles, to high-end PCs.

Rather than using any single graphics API directly, 3D games are typically built on top of abstractions that handle differences in hardware platforms under the hood. We call these abstractions *shader systems*, and they constitute a major component of real-time 3D game engines like Unity,[1] Unreal Engine,[2] and other in-house engines.[3] To deal with the aforementioned challenges in real-time graphics programming, shader systems provide mechanisms for modularizing code, composing these modules together, generating specialized GPU code for these compositions, and invoking the correct specialization at run time. In addition, because these game engines must support a wide range of users (from expert graphics programmers to non-technical artists), a shader system must provide different interfaces to different clients of the system. 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 programming interfaces in graphics APIs do not directly help with building these shader systems because they only focus on issues that affect loading and execution of GPU code. Modern graphics APIs are designed to facilitate robust, high-performance implementations on a wide range of hardware, and therefore, their 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 DXIL [55] and SPIR-V [38]). Thus, developers are left to create layered shader system implementations on top of these graphics APIs.

Engine developers must balance the cost to implement functionality against the benefits in improved features or robustness of the shader 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 code as strings of text, performing pattern matching, substitution, etc. To add further functionality, an engine might wrap an underlying shading lan-

[1]`https://unity3d.com/`

[2]`https://www.unrealengine.com/`

[3]e.g., `https://www.frostbite.com/` and `https://www.cryengine.com/`

guage with a custom domain-specific language (DSL). Finally, making more invasive changes to a shading language requires building or modifying a compiler. While modern shader systems help to navigate the boundary between host code and GPU code, they currently do not present a unified environment for writing these two disparate portions of graphics code. 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. Therefore, at present, it is unclear how to create and maintain a unified shader programming system without prohibitively expensive time and resource costs. Given the size and complexity of graphics code in modern games, a unified shader system has the potential to significantly improve application robustness and engine user productivity.

1.1 Contributions

Thesis Statement To better enable developers of real-time graphics applications to create powerful shader systems, we propose the following thesis:

> A unified shader programming system that integrates GPU code into the same environment as the host code is advantageous for real-time graphics programming and can be achieved with modest effort.

In support of this thesis, this dissertation makes the following contributions:

- **Chapter 3**: We show that a unified programming environment for real-time graphics can be achieve in an existing, widely used programming language (C++) by co-opting existing language features and implementing them with alternate semantics. Using this technique, we implement a unified environment that provides first-class support for specialization by co-opting C++'s attribute and virtual function features. Our implementation uses a Clang-based tool that translates code using our modified semantics to standard C++ and HLSL, compatible with Unreal Engine 4.

- **Chapter 4**: Exploring opportunities in newer programming languages, we present *staged metaprogramming* as an underlying implementation methodology for building unified shader systems. Staged metaprogramming is a family of language features that enables us to build a unified shader system entirely in user-space code, without modifying the language or its compiler. Additionally, we demonstrate how staged metaprogramming can open opportunities for optimizations by creating a design-space exploration framework within our system that results in performance improvements.

While the two implementation techniques described above achieve the same high-level goal of enabling unified shader system development, they have significantly different benefits and trade-offs. In Chapter 5, we contrast these two methods in terms of what they provide to users, as well as what they require of the underlying programming language, to better inform developers seeking to create their own unified systems. By exploring multiple methods, we hope that this work will facilitate the creation of such unified systems within real-time graphics engines today, while also providing guidance for the future as programming languages continue to evolve.

Chapter 2

Background

Having described the high-level task of graphics applications in Chapter 1, we now provide additional details on real-time graphics programming. We focus specifically on the shader system interfaces presented by modern game engines to highlight the necessary components of these interfaces, the techniques used to create them today, and the shortcomings of these current systems and techniques.

2.1 Terminology and Key Concepts

Often, graphics programmers use the term "shader" to refer to the code they write. The meaning of this term differs based on context, so we begin by defining how we use it and other related terminology in this work.

We define a *shader* as consisting of both *GPU shader code* that performs highly parallel rendering calculations and *host shader code* that provides an interface between GPU shader code and the rest of the application.[4] Since GPUs are coprocessors, invocation of GPU code must be initiated from host code running on a CPU. A *shader program* is an invocable unit of GPU code consisting of parameters, functions, and one or more entry points (further described in Section 2.2.1). While compute shader programs are manually invoked by user-written host code, most types of shader programs (e.g., vertex shaders, geometry shaders, fragment/pixel shaders, etc.) are invoked implicitly by the underlying graphics API at the appropriate times

[4]Many graphics programmers think of a "shader" as just the GPU code, but we believe it is useful to include the corresponding host code in the definition as well.

during execution of the hardware graphics pipeline.

A crucial optimization in real-time graphics is *shader specialization*. Shader specialization involves generating multiple compiled versions of a particular GPU shader program, with different compile-time configuration options controlling code generation. We call these configuration options *specialization parameters*, and compiling the shader program with different values for these options generates multiple *shader variants* of the original code. As an example (further explored in Section 2.3), consider a shader used to control the appearance of an object's material. This shader might have large portions of code that are common to all types of materials, as well as portions of code specific to each bidirectional reflectance distribution function (BRDF) implemented in the rendering system. In order to achieve the best performance, a different shader variant will be generated for each of the BRDFs. A BRDF's corresponding shader variant will contain only the code necessary for that BRDF implementation—stripping away code specific to other BRDFs—so that more computationally expensive BRDFs do not impact the performance of cheaper ones. Shaders often have multiple specialization parameters that drive specialization, resulting in many compiled shader variants for each original GPU shader program.

A GPU shader program's corresponding host shader code is responsible for selecting which shader variant to invoke based on dynamic data available at application runtime, such as information about the scene, the underlying hardware platform, and user settings. The data necessary to decide which shader variant to invoke is typically not available until runtime, but graphics developers usually need to generate the specialized variants ahead of time because just-in-time compilation can increase game load times, can hurt performance during gameplay, and is disallowed on some platforms. We make the key observation that expressing and implementing specialization requires coordination between specialization parameters that are *compile-time parameters* for GPU code but *runtime parameters* for host code. Along with variant selection, host shader code also coordinates data transfer between host and GPU memory to provide a shader program with its runtime parameters. A unified environment for real-time graphics programming needs to support both the host- and GPU-related aspects of shader programming.

2.2 Shader Programming

In this section, we discuss GPU shader code and host shader code in more detail. We describe GPU shader code using HLSL and its programming model, but other shading languages like GLSL and Metal Shading Language are similar. We describe host shader code in the context of Unreal Engine 4 (UE4). While the graphics APIs provide underlying functionality necessary to interface between host and GPU code, most major game engines implement systems layered on top of these APIs to provide additional features aimed at making this task easier for their users. UE4's shader programming system puts significant emphasis on imposing structure on host shader code, which allows users to benefit from additional type checking and other static tools (e.g., the shader variant mechanism discussed in Section 2.2.2). We choose to focus on UE4 for this discussion because this structure helps to clearly illustrate the host-related aspects of shader programming and, in many ways, represents the limits of what these systems can accomplish in a non-unified environment. Nevertheless, the tasks that UE4 host shader code must accomplish, as well as the issues it faces, are also applicable to other game engines and to the underlying graphics APIs.

2.2.1 GPU Shader Code

Listing 2.1 shows an example of a typical shader program written in HLSL. Common practice is to modularize each GPU shader program as its own HLSL source file.[5] When invoked from host code, multiple *instances* of this program are executed in parallel, with each instance operating mostly independently.[6]

The shader program in Listing 2.1 has an *entry point* function (`MainCS()` on line 19), which is where GPU code execution begins for each instance. Because this particular shader program is a compute shader, the entry point is annotated with information about how many threads to invoke per thread group (line 18).[7] This example also includes other GPU functions (e.g., the `doFiltering()` functions) that can be called from GPU code. In HLSL, an en-

[5]However, other files can be `#included` to allow for reuse of HLSL code.

[6]The HLSL programming model provides some inter-instance synchronization and communication capabilities, but we will not discuss them here.

[7]For information about the HLSL compute shader programming model, as well as other types of shaders, we refer interested readers to the HLSL documentation [52].

```
 1 #define LOW 0
 2 #define MEDIUM 1
 3 #define HIGH 2
 4
 5 Texture2D ColorTexture;
 6 SamplerState ColorSampler;
 7 RWTexture2D<float4> Output;
 8
 9 #if QUALITY == LOW
10   float4 doFiltering(float2 pos) { /* Low Quality Method */ }
11 #elif QUALITY == MEDIUM
12   float4 doFiltering(float2 pos) { /* Medium Quality Method */ }
13 #elif QUALITY == HIGH
14   int ExtraParameter;
15   float4 doFiltering(float2 pos) { /* High Quality Method */ }
16 #endif
17
18 [numthreads(8, 8, 1)]
19 void MainCS(uint2 DispatchThreadID : SV_DispatchThreadID) {
20   float2 pixelPos = /* ... */;
21   float4 outColor = ColorTexture.Sample(ColorSampler, pixelPos);
22
23   for(int i = 0; i < ITERATION_COUNT; ++i) {
24     outColor *= doFiltering(pixelPos);
25   }
26
27   Output[DispatchThreadID] = outColor;
28 }
```

Listing 2.1: An example GPU shader program written in HLSL.

try point may declare *varying parameters*, whose values can differ for each instance within an invocation. For example, each instance in a given invocation has a unique ID, and the `DispatchThreadID` varying parameter (line 19) provides an instance with its ID value. The value for this parameter is provided implicitly by the HLSL programming model; the code attaches the user-defined function parameter to the system-defined value using the HLSL "semantic" named `SV_DispatchThreadID`.

This shader program also has several *uniform parameters* (lines 5–7). Unlike varying parameters, the value of a uniform parameter is the same for all instances in a given invocation. For example, `ColorTexture` is a uniform parameter that represents a 2D image containing color information, and all instances within an invocation access the same 2D image when using this parameter.

```cpp
enum class QualityEnumType : int {
  Low,
  Medium,
  High
};

class FilterShaderCS : public FGlobalShader {
public:
  DECLARE_SHADER_TYPE(FilterShaderCS, Global);

  BEGIN_SHADER_PARAMETER_STRUCT(FParameters, )
    SHADER_PARAMETER_RDG_TEXTURE(Texture2D, ColorTexture)
    SHADER_PARAMETER_SAMPLER(SamplerState, ColorSampler)
    SHADER_PARAMETER_RDG_TEXTURE_UAV(RWTexture2D<float4>, Output)
    SHADER_PARAMETER(int, ExtraParameter)
  END_SHADER_PARAMETER_STRUCT()

  class QualityDimension :
    SHADER_PERMUTATION_ENUM_CLASS("QUALITY", QualityEnumType);

  class IterationCountDimension :
    SHADER_PERMUTATION_SPARSE_INT("ITERATION_COUNT", 2, 4, 8, 16);

  using FPermutationDomain = TShaderPermutationDomain<
    QualityDimension, IterationCountDimension>;
};

IMPLEMENT_GLOBAL_SHADER(FilterShaderCS,
  "/path/to/HLSL/file.usf", "MainCS", SF_Compute);
```

Listing 2.2: Host shader code corresponding to the GPU shader program in Listing 2.1. This code is written in C++ and uses features provided by UE4.

The example in Listing 2.1 also uses two *specialization parameters*: `QUALITY` and `ITERATION_COUNT`. Specialization parameters express the different compile-time options that are used to generate multiple shader variants of this shader program (as mentioned above). Notice that these parameters are not declared explicitly in the GPU code, but their values are implicitly required for the shader program to compile properly. When compiling this program, the value for each parameter is passed in as a macro (i.e., a `#define`) for the C-style preprocessor that HLSL supports. As a result, the value for each specialization parameter is constant at compile-time in GPU code, which allows the compiler to better optimize the code (e.g., by unrolling the loop on line 23). Specialization parameters can also be used to define additional GPU functions and uniform parameters (e.g., the `ExtraParameter` uniform on line 14 is only defined when `QUALITY == HIGH`). Note that the possible values for `QUALITY` are also defined using the preprocessor (lines 1–3).

2.2.2 Host Shader Code (in Unreal Engine 4)

Listing 2.2 shows the UE4 host shader code corresponding to the example shader program in Listing 2.1. In UE4, each shader program is accompanied by a C++ class (line 7) that provides the host-side interface to the GPU code. The host code class is associated with a GPU shader program using a UE4 macro to indicate the filename of the HLSL code and name of the entry point function (lines 28–29).

The host code class includes declarations of the shader's uniform parameters using UE4's `SHADER_PARAMETER` macros (lines 11–16). These macros define a struct, which implements a strongly typed interface for passing parameters from host code to GPU code:

```
FilterShaderCS::FParameters* Parameters =
  /* allocate parameter struct */;

Parameters->ColorTexture = colorTexture;
Parameters->ColorSampler = colorSampler;
Parameters->Output = outputTexture;
```

Unlike in GPU shader code, host shader code in UE4 includes explicit declarations for specialization parameters (lines 18–22). Using the `SHADER_PERMUTATION` macros, programmers provide the system with a static set of options for each parameter (e.g., all values of an enum type or individual integer values). Then, these parameter declarations are used to create an `FPermutationDomain` for this shader (lines 24–25). This construct serves two purposes. First, at shader program compile time, it enables the UE4 system to statically generate all shader variants of the GPU shader program by using the statically specified set of options for each specialization parameter. Second, at game runtime, it enables host shader code to easily select which variant to invoke, based on runtime information:

```
FilterShaderCS::FPermutationDomain PermutationVector;
PermutationVector.Set<FilterShaderCS::QualityDimension>(quality);
PermutationVector.Set<FilterShaderCS::IterationCountDimension>(
  iterCount);
```

2.2.3 Issues in a Non-Unified Environment

At this point, we can identify several issues that arise as a result of a non-unified shader programming environment. These issues apply to both UE4 and other game engines, as well as to applications using the graphics APIs directly.

As shown in the example above, GPU and host code must both declare the uniform parameters that the shader needs (e.g., Listing 2.1 lines 5–7 and Listing 2.2 lines 11–16, respectively). Programmers must ensure that they use the same types and variable names in both declarations, and they must also keep these duplicate declarations consistent as the code changes. Similarly, host and GPU code cannot share types and functions in non-unified environments, leading to additional code duplication (e.g., the enum values in Listing 2.2 lines 1–5 are redeclared in Listing 2.1 lines 1–3). Failing to properly maintain consistency between host and GPU code can lead to runtime errors and bugs that are potentially difficult to track down and fix.

Additionally, note that in GPU code, specialization parameters are not explicitly defined. Instead, GPU code references these parameters and expects that they will be available at compile time. As a result, there is little verification—at either compile-time or runtime—that these parameters are referenced correctly (e.g., a typo in GPU code may result in a difficult-to-debug logic error). A programmer could easily `#include` GPU code that uses an implicit specialization parameter but omit the corresponding declaration in the host code class, leaving future readers to wonder whether the omission is a mistake or if the default value is always correct for that particular shader.[8]

In contrast, in a unified system where host and GPU code share parameters, types, and functions, these kinds of issues do not exist. Unified systems can therefore reduce programmer burden and increase code robustness. However, the best way to support specialization in a unified shader programming environment is unclear. The difficulty arises from the need to compile-time specialize GPU shader code but then select which specializations to invoke based on information only available at runtime. After presenting our solution to this problem in a C++-based environment (Chapter 3), we explore design alternatives in Section 3.2.4.3 that

[8]We found ourselves in this exact scenario when looking through the UE4 codebase. A later commit added the parameter to the host code, confirming that the omission was a mistake: `https://github.com/EpicGames/UnrealEngine/commit/2a2cebffe6a5a7164dbe2401ba2d5dd1901b649e`
(Note: access to this page requires permission to access the UE4 source code on GitHub)

demonstrate why the typical compile-time metaprogramming methods familiar to graphics programmers today (preprocessor-based methods and template metaprogramming) are insufficient for implementing specialization in a unified environment.

2.3 Modern Shader System Implementations

As mentioned above, modern game engines provide *shader systems* that aid in writing, configuring, and executing shader code. Along with managing compilation of GPU shader source code to executable kernels, a major task of a shader system is to enable a wide variety of users to control different aspects of the rendering process. There are *expert graphics programmers* who typically write GPU shader code using a shading language like HLSL or GLSL, as well as host shader code in a systems language like C++. *Technical artists* also write some shader code; however, unlike graphics programmers, technical artists are typically not experts in shader optimization techniques. Finally, (non-technical) *artists* do not write shader code directly. Instead, they create different appearances and effects by using pre-written shaders and assigning different values to the parameters of the underlying code.

The services provided by a shader system are designed to present the appropriate interfaces for these different categories of users while also ensuring that the final runtime code is highly efficient in order to achieve a smooth gameplay experience. For example, rather than requiring artists to write host code to set the parameters of GPU shader programs, shader systems instead provide graphical user interfaces (GUIs) that artists can use to configure shaders. Since the productivity of artists and technical artists is highly important, these systems usually have mechanisms for exposing shader parameters to these GUIs with minimal boilerplate. Similarly, setting shader program parameters and invoking GPU shader code is tedious and error-prone when using the underlying graphics APIs directly. Thus, shader systems often have some facilities to aid in navigating the boundary between host code and GPU code, with varying levels of robustness (as discussed further below). Lastly, like in many complex applications across a variety of domains, rendering effects often consist of multiple composable and interchangeable components. However, more unique to graphics is the need for the final, compiled compositions of these components to result in highly optimized code, where fractions of a millisecond might determine whether or not an effect is usable in a shipping application. Shader specialization is therefore a crucial service of a shader system.

❧

In the remainder of this section, we briefly discuss existing ways to implement these aspects of a shader system, ordered by increasing levels of complexity.[9] 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. While some of these systems provide better features than others, none of them present a unified environment for authoring both host and GPU code.

2.3.1 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 program 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));
}
```

This shader program 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 op-

[9]The below discussion of existing shader systems is adapted from Section 3 of our paper "Staged Metaprogramming for Shader System Development" [68].

tions 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 host and GPU code is left to the programmer. Hence, setting the value of parameters from C++ host code 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 program (`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 host-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 host-side struct matches what the GPU shader program 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 program has a corresponding C++ class written by the shader writer (as shown in Section 2.2) [24]. 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.

2.3.2 A Layered DSL with Embedded HLSL

To provide further functionality, some engines implement a custom layered DSL on top of an underlying shading language. Shader programs in Unity are written in ShaderLab [76]:

```
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 code 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.

2.3.3 A DSL That Manipulates and Generates HLSL

Bungie's TFX language [74] features better integration with HLSL at the cost of greater implementation effort for the engine developers. The surface shader program 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 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 from host code to shader programs. “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, which could be significantly time-consuming operations given the large scale of modern game engines.

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.

2.3.4 Modifying HLSL

Rather than creating a DSL that embeds HLSL, one could instead extend the shading language itself. The Slang shading language [32] extends HLSL by adding some general-purpose language features from other popular programming languages. Here is the surface shader program 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 set up 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 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 host-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 [55]) 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.

2.3.5 Summary

The shader code in modern, real-time graphics applications represents a significant time and resource investment. Some games ship with tens of thousands of programmer- and artist-authored shader code components, compiled to hundreds of thousands of GPU kernels [33]. Thus, deficiencies in a shader programming system significantly impact user productivity, code robustness, and application performance.

While each subsequent example presented above improves upon some deficiencies of the previous examples, this improvement comes at the cost of greater implementation effort. Moreover, none of these examples provide a unified system for authoring both host and GPU code.

They each use one language for host code (C++ or C#) and another for GPU code (HLSL or Slang), require host and GPU code to be in separate files, and do not allow host and GPU code to reference the same parameters. Using the above implementation methods, the costs of developing a unified system outweigh the potential benefits provided to users. Ideally, we believe developers should be able to create unified systems that improve upon the results of the more complex solutions discussed above, while only requiring effort similar to the simpler implementation methods.

Chapter 3

Unified Shader Programming in C++

To be practically useful today, the goal of creating a unified shader programming system must be achievable in the languages commonly used in real-time graphics programming. The effort required to create such a system must be reasonable for a game/engine developer to use in practice, because the industry cannot depend on popular, general-purpose languages to evolve around its needs, especially in the near term. Any unified environment for shader programming must provide support for specialization because of the importance of this optimization (as discussed above), but unfortunately, the popular programming languages used in graphics cannot express parameters that are part compile-time and part runtime, which is a necessary requirement for specialization parameters. Moreover, existing unified GPU programming environments like CUDA are insufficient as well because they do not provide a mechanism to drive GPU code specialization from host data/logic.

In this chapter, we show that a unified programming environment for real-time graphics can be achieved in an existing, widely used programming language (C++) by co-opting existing language features and implementing them with alternate semantics to provide the services required. Using this key insight, we present the following contributions:

- The design of a unified programming environment for real-time graphics in C++ that provides first-class support for specialization by co-opting C++ attributes and virtual functions

- A Clang-based tool that translates code using our modified C++ semantics to standard

C++ and HLSL code, compatible with Unreal Engine 4

We present the design of our unified environment and the implementation of our translation tool in Sections 3.2 and 3.3, respectively. In Section 2.2, we briefly introduce modern real-time graphics programming and identify some issues that result from using a non-unified environment. This discussion helps to motivate our goals, constraints, and non-goals, which we present in Section 3.1.

3.1 Goals, Constraints, and Non-Goals

Our overarching goal is to enable development of unified shader programming systems that are practically useful for large-scale real-time graphics applications. Motivated by the benefits that such unified systems can provide along with the barriers to creating them, we establish the following high-level design goals:

- **Write the host and GPU portions of shader code in the same language, file, and lexical scope**

  This goal comes directly from our definition of a *unified* environment and, thus, is a necessary condition that unified shader programming systems must achieve. However, it alone is not sufficient to define a practically useful system.

- **First-class support for GPU shader code specialization**

  GPU code specialization is a ubiquitous optimization in modern real-time graphics, but the popular methods currently used to express and implement specialization do not translate to a unified system. We would like to bring this optimization to the forefront by providing it first-class support.

- **Declare each shader parameter only once**

  One benefit of a unified system is that host and GPU code can share various declarations, like types and functions, so that programmers do not need to manually maintain disparate definitions across the host-GPU boundary. We wish to extend this benefit to shader parameters, allowing both host and GPU code to reference the same parameter declarations.

- **Ease of integration into current real-time graphics applications**

  To promote adoption of unified shader programming, we would like to provide a path for ease of integration of our ideas into existing systems.

- **Encourage better software engineering practices in shader development**

  Because typical shading languages are feature-poor compared to modern systems languages, GPU shader code often relies on features that can lead to additional development and maintenance effort (e.g., preprocessor `#if` and `#define`). Instead, we wish to leverage other language features in shader development to enable better software engineering practices.

Along with these design goals, we also aim to satisfy some design constraints:

- **Use a programming language that is widely used in real-time computer graphics**

  Related to our ease-of-integration goal, we want to explore unified shader programming in a language that is commonly used for real-time graphics today. We want the code in our system to look and feel familiar to programmers using this language, and so we strive to modify this language a little as possible to achieve our goals.

- **Minimize internal developer costs**

  Also related to ease-of-integration, we would like to limit the developmental costs of our implementation so that engine developers could conceivably build and maintain such a unified system themselves. This precludes building a compiler, for example, since the effort required is not tractable for most design teams.

- **Equivalent performance compared to current implementations**

  In order for unified shader systems to be viable, they should introduce little to no performance overheads compared to current systems.

Finally, we want to be explicit about our non-goals:

- **Compiling arbitrary host language code to GPU-compatible code is out of scope for this work**

While this task is important and necessary for unified shader systems, other efforts are already attempting to accomplish this task—both for C++ and for Rust—which we discuss in Section 6.5. Instead, we focus on the next layer: once a language can generate both host- and GPU-compatible code, what else is required to achieve useful unified shader programming?

- **Executing GPU-side shader code on the host (and vice versa) is a non-goal of this work**
  Modern shader programming has a clear distinction between the host- and GPU-related aspects of shader programming from which we do not attempt to deviate. However, individual functions may be callable from both host and GPU code, provided these functions are compatible with both the host and GPU processors.

- **Our work specifically targets only CPU hosts and modern GPUs**
  Targeting other types of processors might be an interesting area of future work.

3.2 Design Decisions

The key insight of this work is that we can develop a unified model for shader programming by co-opting existing features of a programming language and implementing them with alternate semantics to provide the services required by real-time graphics. This insight represents the overarching design philosophy for our system and influences the other design decisions that allow us to achieve our high-level goals. In contrast to this approach, we could instead either develop an entirely new language or add new features to an existing language to provide the missing services. However, neither of these alternatives align with our objectives, especially given our constraint of minimizing internal developer costs.

Creating a new programming language complicates integrating unified shader programming into existing large-scale graphics applications. Such a new language would need to interface with the language currently used in an existing system, not only to allow programmers to rewrite shader code incrementally but also to enable other subsystems (such as the physics and animation subsystems) to communicate with the rewritten code. That latter aspect would increase development costs, since programmers would need to write and maintain additional code to usher data between the two languages (whereas today, most graphics applications use the same host language for all subsystems). The alternative of rewriting the entire application using the new language is also not ideal because a new language designed specifically for unified shader programming might not be a good fit for the other subsystems. Therefore, we have chosen to base our work on an existing language that is widely used in real-time graphics today.

Similarly, modifying an existing language to add new language features for unified shader programming also violates our ease-of-integration goal but for different reasons. Adding a new feature to a language requires understanding how that feature interacts with every other feature in the language, including future features as a language continues to evolve. This approach could be viable if a language chooses to formally adopt these new features; however, there is no guarantee that a general-purpose language will incorporate graphics-specific features. We instead wish to maintain compatibility both with existing code and with future language versions, so we have decided to repurpose existing language features to express the requirements of unified shader programming.

In the remainder of this section, we discuss the other major design decisions that enable our system to provide a unified shader programming environment that achieves our high-level goals. While some aspects of these decisions might be specific to our language of choice (Section 3.2.1), we believe that many of the ideas presented below are transferable to other languages as well. Different languages provide different features, so the choice of which features to co-opt for shader programming might depend on the specifics of the language. Nevertheless, we believe that our key insight will enable integrating unified shader programming into other languages beyond what our implementation demonstrates.

3.2.1 C++ for Both Host and GPU Code

C++ is a natural choice for exploring unified shader programming. It is one of the most widely used languages in real-time graphics, as evidenced by its use in many in-house and 3rd party game engines (e.g., UE4 [24], Godot Engine [45], Frostbite [18], and Lumberyard [1]). While it is mostly used for host code today, there are indications that it may become more prevalent for GPU code in the future. For example, the Metal Shading Language is based on C++ [4], a presentation from SIGGRAPH 2016 suggests that the game development industry could move toward C++ for GPU shader code [26], and efforts are already underway to generate SPIR-V [38] and DXIL [55] from C++ (see Section 6.5). Thus, using C++ for our work helps us to meet our goals and constraints related to ease-of-integration.

For similar reasons, we could have instead chosen HLSL as our unified shader programming language. However, C++ provides many additional features compared to HLSL. Rather than converting host code to HLSL, we feel that the long-term goal for shader programming should be to support more language features in GPU code. Therefore, we believe that our choice to unify host and GPU code into C++ is more representative of the future of shader programming. As mentioned above, while some of the results of our investigation are likely specific to C++, we hope that the broader ideas are useful to programmers using other languages as well.

❧

Listing 3.1 shows the example shader from Section 2.2 rewritten using our unified C++-based shader programming environment. We explain the various parts of it in the next three sections.

3.2.2 Use C++ Attributes to Express Declarations Specific to Shader Programming

In our system, programmers use C++ attributes to annotate declarations related to shader-programming-specific constructs. The attributes feature was introduced in C++11 to provide a standardized syntax for implementation-defined language extensions, rather than different compilers continuing to use custom syntaxes (e.g., GNU's `__attribute__((...))` or Microsoft's `__declspec()`). Our implementation supports the following shader-specific attributes:

- Uniform parameters are annotated using the `[[uniform]]` attribute (lines 3–5).
- Specialization parameters are indicated using the `[[specialization]]` set of attributes (lines 7–11). We defer discussion of specialization to Section 3.2.4.
- The `[[entry]]` set of attributes declares a function as the *entry point* to use when invoking GPU code execution. For compute shaders, this attribute requires arguments for

```
 1 class [[ShaderClass]] FilterShader {
 2 public:
 3   [[uniform]] Texture2D           ColorTexture;
 4   [[uniform]] SamplerState        ColorSampler;
 5   [[uniform]] RWTexture2D<float4>  Output;
 6 
 7   [[specialization-ShaderClass]]
 8   FilterMethod* filterMethod;
 9 
10   [[specialization-SparseInt(2, 4, 8, 16)]]
11   int IterationCount;
12 
13   [[entry-ComputeShader(8, 8, 1)]]
14   void MainCS(
15     [[SV_DispatchThreadID]] uint2 DispatchThreadID) const
16   {
17     float2 pixelPos = /* ... */;
18     float4 outColor = ColorTexture.Sample(ColorSampler, pixelPos);
19 
20     for (int i = 0; i < IterationCount; ++i) {
21       outColor *= filterMethod->doFiltering(pixelPos);
22     }
23 
24     Output[DispatchThreadID] = outColor;
25   }
26 };
```

Listing 3.1: An example shader using our unified C++ shader system. A ShaderClass can contain both host and GPU code, written using standard C++11 syntax. Special C++ attributes are used to express various shader-specific constructs (e.g., uniform parameters, specialization parameters, and entry point functions).

the thread group size (line 13), similar to the `numthreads` attribute in HLSL.

- System-defined varying parameters are attached to entry point function parameters using corresponding attributes, which are named following HLSL's convention (e.g., `[[SV_DispatchThreadID]]` on line 15).

- Because our system unifies host and GPU code into the same file, all non-entry-point GPU functions must be annotated with the `[[gpu]]` attribute.[10] By manually annotating GPU functions, we can disallow or reinterpret certain language features in GPU code when appropriate, while continuing to allow host functions to freely use any language feature

[10]CUDA uses a similar approach, where GPU-only functions are annotated with `__device__` and functions that are callable from both host and GPU code with `__host__` `__device__`.

(see Section 3.2.4 for further discussion).

Our use of C++ attributes to express elements specific to shader programming represents a departure from the intent of this language feature. In general, non-standard attributes can be ignored by the compiler and, thus, should not change the semantics of a program. However, our attributes are integral to correctly defining the semantics of shader code; ignoring these attributes will result in an incorrect program. Nevertheless, attributes provide a clean and concise method for expressing the above concepts, so our system co-opts this language feature for unified shader programming.

3.2.3 Modularize Host and GPU Shader Code Using Classes

To promote more maintainable coding practices, our design uses C++ classes to modularize shader code. Programmers declare that a class contains shader code using the `[[ShaderClass]]` attribute (line 1). Our *ShaderClass* design has similarities with UE4's use of C++ classes in that both declare uniform and specialization parameters. However, a major difference is that our ShaderClasses can contain both host and GPU code.

Because of this unified design, host and GPU code reference the same shader parameter declaration. Thus, these declarations are—by construction—always kept consistent in both host and GPU code, avoiding the need to maintain separate definitions. Host code provides data to GPU code by assigning values to these parameters, for example:

```
FilterShader shader;

shader.ColorTexture = colorTexture;
shader.ColorSampler = colorSampler;
shader.Output = outputTexture;
```

Host code can also set shader parameters using methods defined within a ShaderClass (e.g., the class's constructor).

GPU methods within a ShaderClass must be declared `const` (line 15). In general, GPU shader code cannot modify uniform and specialization parameters, so requiring that these methods be `const` imposes this restriction. However, some uniform parameter types (e.g., `RWTexture2D`) allow modification from GPU code using specific operations, and our system does provide support for these operations accordingly (e.g., writing to the `Output` texture on line 24).

A ShaderClass may or may not be a complete, invocable shader program. If a ShaderClass contains an entry point method, then it can be used as an invocable shader program. However, programmers can also write a ShaderClass without an entry point method, allowing for encapsulation of functionality that can then be reused across different shader programs by using the ShaderClass as a member variable (as shown on line 8). Member variables of a ShaderClass type must be declared as specialization parameters, for reasons we discuss next.

3.2.4 Implement Specialization by Co-opting Virtual Function Calls

3.2.4.1 Basic Specialization Parameters

Like uniform parameters, ShaderClasses also express specialization parameters as member variables that both host and GPU code can reference, providing explicit declarations of these parameters for both halves of shader code. Therefore, our system can catch more errors at compile time than other systems where specialization parameters are implicit in GPU code.

Host code can set these parameters based on runtime information using the same mechanisms that apply to uniform parameters, e.g.:

```
FilterShader shader;
shader.IterationCount = settings.getIterationCount();
```

While these parameters are runtime-assignable in host code, they must instead be compile-time-constant in GPU code to allow the underlying GPU code compiler to perform the optimizations that programmers expect when they use specialization. Thus, to support specialization, the set of possible values for all specialization parameters must be statically available at compile time. For some types (e.g., enums and bools), our system can determine these values automatically; for other types (e.g., ints), we follow UE4's approach by requiring that programmers manually enumerate the possible values (line 10). Using these options, our translator (Section 3.3) can then statically generate all GPU shader variants of a ShaderClass at compile time, while still allowing host code to easily select which variant to invoke at runtime by assigning

```
class [[ShaderClass]] FilterMethod {
public:
  [[gpu]] virtual float4 doFiltering(float2 pos) const = 0;
};

class [[ShaderClass]] LowQualityFilter : public FilterMethod {
public:
  [[gpu]] virtual float4 doFiltering(float2 pos) const override
  {
    /* Low Quality Method */
  }
};

class [[ShaderClass]] MedQualityFilter : public FilterMethod {
public:
  [[gpu]] virtual float4 doFiltering(float2 pos) const override
  {
    /* Medium Quality Method */
  }
};

class [[ShaderClass]] HighQualityFilter : public FilterMethod {
public:
  [[uniform]] int ExtraParameter;
  [[gpu]] virtual float4 doFiltering(float2 pos) const override
  {
    /* High Quality Method */
  }
};
```

Listing 3.2: ShaderClasses can contain virtual `[[gpu]]` methods. In GPU code, virtual function calls are converted from dynamic dispatch to static dispatch, generating multiple shader variants accordingly.

values to the specialization parameters based on runtime information.[11]

This approach provides a simple mechanism that cleanly handles the runtime-for-host-code vs. compile-time-for-GPU-code requirements of specialization parameters. However, by using class member variables for specialization parameters, it is not obvious how to conditionally declare uniforms and functions based on these parameters (e.g., the `ExtraParameter` uniform and the `doFiltering()` functions in Listing 2.1). We solve this issue by allowing a ShaderClass to use another ShaderClass as a specialization parameter.

[11]To provide better error checking during development, our translator generates asserts to ensure that a specialization parameter's runtime value is one of the statically enumerated options. UE4 has similar error checking, but some other systems do not.

3.2.4.2 ShaderClass Specialization Parameters

As shown in Listing 3.1 on line 21, the `doFiltering()` function is provided by a member variable of type `FilterMethod` (line 8). `FilterMethod` is itself a ShaderClass, and it also has ShaderClass subtypes. Listing 3.2 shows the implementations of these types.

The `doFiltering()` method is declared as a virtual method in the base `FilterMethod` class (line 3). Then, each subclass overrides this method to provide their own implementations (lines 8, 16, and 25). Based on runtime information, the host shader code can select which implementation to use in the `FilterShader`:

```
FilterShader shader;
QualityEnumType quality = settings.getQuality()
if (quality == QualityEnumType::Low)
  shader.filterMethod = new LowQualityFilter();
else if (quality == QualityEnumType::Medium)
  shader.filterMethod = new MedQualityFilter();
else if (quality == QualityEnumType::High)
  shader.filterMethod = new HighQualityFilter();
```

In C++, virtual methods normally use *dynamic dispatch*—at runtime, the method implementation that gets invoked depends on the runtime type of the variable. However, in GPU shader code, *static dispatch*—where the method that gets invokes is known statically at compile time—results in significant performance benefits. This difference creates a conflict between host code and GPU code: host code needs to select which type to use based on runtime information, but GPU code should use static dispatch (which requires this type information at compile time) for optimal performance.

Therefore, when a ShaderClass uses another ShaderClass as a member variable, our system requires that variable to be a specialization parameter, which allows us to avoid dynamic dispatch in the generated GPU code. Our translator generates different shader variants for each possible subclass of a ShaderClass-type specialization parameter in order to convert the virtual method calls into static function calls, thereby replacing dynamic dispatches with static dispatches. At runtime, the correct shader variant is selected by using the runtime type of the specialization parameter.[12] By co-opting virtual functions and implementing them with alternate semantics for shader code, we are able to provide first-class support for GPU code specialization in our unified shader programming environment.

As an added benefit, this design also encourages more robust software engineering practices. In Listing 2.1, the `ExtraParameter` uniform is only declared when `QUALITY == HIGH`. If other parts of the HLSL code need to access that parameter, programmers can (and often do) write additional `#if` checks before using the parameter. This practice leads to difficult-to-maintain code, since these various dependencies can be scattered throughout a large HLSL file. In contrast, our design promotes encapsulation of these dependencies by allow programmers to organize uniform parameters, specialization parameters, and (host and GPU) functions into C++ classes. In Listing 3.2, the `ExtraParameter` uniform is only declared in the `HighQualityFilter` class (line 24), ensuring that programmers cannot use it elsewhere by mistake.

3.2.4.3 Design Alternatives

Before arriving at the decision to co-opt C++ virtual functions, we considered two other methods for implementing specialization: preprocessor techniques and C++ templates. However, neither meets our needs in a unified environment.

As noted in Section 2.2.3, many systems implement GPU shader code specialization using preprocessor-based methods. These methods include C-style preprocessor facilities (e.g., macros, `#defines`, `#ifs`), as well as composing together small strings of GPU code to form a complete shader program. Both of these techniques fail to translate into a unified shader programming environment. C-preprocessor directives are evaluated in the first step of the compilation process. In non-unified systems, this compile-time-only technique can be used for GPU code because the host and GPU code exist in separate files and separate programming environments. However, it does not work in host code because of the need to dynamically control shader variant selection based on runtime information. Therefore, in a unified environment, where we desire a unified representation for specialization parameters, we cannot use C-preprocessor-based methods to implement specialization for either portion of shader code.

[12]Rather than using the built-in C++ runtime type information feature, we use our own, simplified mechanism to minimize performance overheads.

```
template<QUALITY, ITERATION_COUNT>
class FilterShader {
public:
  /* Uniform parameter declarations */

  /* GPU function */
  void MainCS(uint2 DispatchThreadId) {
    float2 pixelPos = /* ... */;
    float4 outColor = ColorTexture.Sample(ColorSampler, pixelPos);

    for (int i = 0; i < ITERATION_COUNT; ++i) {
      if (QUALITY == QualityEnumType::Low)
        /* Low Quality Method */
      else if (QUALITY == QualityEnumType::Medium)
        /* Medium Quality Method */
      else if (QUALITY == QualityEnumType::High)
        /* High Quality Method */
    }

    Output[DispatchThreadID] = outColor;
  }
};
```

Listing 3.3: A mockup of a unified shader design that uses C++ templates for specialization. Templates are insufficient for implementing and expressing specialization in a unified environment, as demonstrated in Section 3.2.4.3.

String-based methods are also inadequate in a unified environment because host and GPU code are necessarily not unified—host code is written in a programming language, while GPU code is represented as just strings.

Along with the preprocessor, C++ has an additional mechanism for static specialization of code: templates. However, using templates has the same basic issue mentioned above—they are insufficient for expressing runtime decisions in host code. Consider the mockup in Listing 3.3, which attempts to use templates to express specialization parameters. This design does adequately enable specialization of GPU shader code in the `MainCS()` function, since both `ITERATION_COUNT` and `QUALITY` are compile-time-constant parameters. However, the complications with this design are readily apparent when considering how to select the correct specialization in host code based on runtime parameter values, as shown in Listing 3.4.

Template parameter values must be statically available at compile time, so the only way to set specialization parameters based on runtime values is to manually enumerate all possible

```
if (quality == QualityEnumType::Low) {
  if (iterCount == 2)
    FilterShader<QualityEnumType::Low, 2> shader;
  else if (iterCount == 4)
    FilterShader<QualityEnumType::Low, 4> shader;
  else if (iterCount == 8)
    FilterShader<QualityEnumType::Low, 8> shader;
  else if (iterCount == 16)
    FilterShader<QualityEnumType::Low, 16> shader;
}
else if (quality == QualityEnumType::Medium) {
  /* repeat ifs for iterCount */
}
else if (quality == QualityEnumType::High) {
  /* repeat ifs for iterCount */
}
```

Listing 3.4: Using dynamic runtime data to control template-based specialization leads to greater development effort and maintenance costs, even for this simple example corresponding to Listing 3.3.

combinations with corresponding `if` statements to select the right combination at runtime.[13] This design requires significantly greater programmer effort even for this simple example with only two specialization parameters, and modifying the set of options for these parameters is also extremely cumbersome. Therefore, using C++ templates as-is for specialization is not a viable option. We also considered co-opting templates for specialization, but this design would require changing template semantics for host code. By co-opting virtual functions instead, we could leave host code semantics intact and only change semantics for GPU code, ensuring backward compatibility with existing C++ code.

Other programming languages might have other features that are suitable for expressing and implementing specialization (e.g., generics). However, preprocessor- and template-based methods are the main ones available and familiar to graphics programmers using the popular HLSL and C++ languages. An interesting area of future work is to explore unified shader development in other languages with different sets of features.

[13]In fact, the UE4 codebase has code very similar to this example in various places. Prior to introducing the `FPermutationDomain` feature, UE4 used templates on host-code shader classes to express integer and boolean specialization parameters. The code to invoke these shaders uses multiple nested `if` and `switch` statements to select which template specialization to use based on runtime values of these parameters, resulting in verbose, complex, and unwieldy code. Eventually, this code may be rewritten to use the `FPermutationDomain` system, but this feature does not extend to a unified environment.

3.2.5 Limitations

Graphics programmers sometimes use specialization parameters to modify struct definitions in HLSL code by using `#ifs` to include or exclude certain data member declarations. They then write corresponding `#ifs` throughout the HLSL file whenever they need to access those conditionally defined members.[14] While our current system does not support conditional struct definitions, we believe that our idea to co-opt virtual functions for specialization of ShaderClass types can also be applied to specialization of GPU-only struct types. The key difference is that the data members in a ShaderClass (i.e., uniform and specialization parameters) have the same values for all invocations of a shader program, whereas a GPU-only struct might contain different values per invocation (e.g., if the struct is used as a local variable within a GPU function). However, as long as all invocations use the same runtime type for the struct (which is equivalent to the HLSL case described above), then the same basic principles can be applied.

In our implementation, we have chosen to focus on shaders that align with UE4's *Global Shaders* concept, which are shaders that do not need to interface with the material or mesh systems. These Global Shaders are sufficient to demonstrate the issues that arise in a non-unified environment and the challenges to developing unified shader programming, as well as how our solutions address these issues and challenges. Therefore, we leave exploration of UE4's *Material* and *MeshMaterial* shaders as future work (see Section 5.3.1 for a discussion on supporting additional shader types). While we think that the basic ShaderClass design can extend to support them, these other shader types do pose additional challenges. Many modern game engines provide a graphical user interface (GUI) for creating materials models. Material shaders need to access the parameters of these materials (e.g., diffuse color, specular color, roughness), but different materials might have different sets of parameters. Determining the best way to interface these GUI-defined materials with a unified shader requires balancing complexity trade-offs between the GUI tools and the programming system. Supporting shaders that interact with meshes has the added challenge of coordinating varying parameter declarations between different shader types. For example, a vertex shader outputs varying parameters that a pixel shader then consumes. Ideally, a unified system would provide a robust mechanism for coordinating

[14]This technique is similar to how they conditionally declare uniform parameters based on specialization parameters and, thus, has similar code maintainability downsides.

this information between different shader types. Nevertheless, shaders that fall into the Global Shader category make up an increasingly large portion of a modern game's shader code, so we feel that our choice to focus on them for this work is justifiable.

3.3 Translation Tool Implementation

To implement our unified shader programming environment design, we built a source-to-source translator based on Clang. The translator uses Clang's LibTooling API,[15] which provides a high degree of flexibility and power without requiring modifications to Clang. Because our implementation is external from the Clang codebase, we can more easily update to newer Clang versions in the future to remain compatible with future C++ features. In addition, we use HLSL++[16] to provide definitions of HLSL-specific types and intrinsics in C++.

The main task of the translator tool is to convert unified C++ shader code that uses our co-opted features into standard C++ and HLSL code that implements the alternate semantics for these features. This transformation lets our system use existing C++ and HLSL compilers and toolchains for final executable code generations, rather than requiring a full compiler implementation. By using this translation strategy, we better facilitate ease of integration into existing applications, since these applications do not need to replace their existing toolchains to use our designs. Our translator tool is separated into three major components: the frontend, the host backend, and the GPU backend.

The translator's frontend traverses the Clang Abstract Syntax Tree (AST) to retrieve relevant information from user-written source code. Rather than operating on arbitrary regions of the AST, the frontend only inspects C++ declarations that are annotated with the `[[ShaderClass]]` or `[[gpu]]` attributes. An internal representation is created for each ShaderClass that contains information about its shader-specific elements (Section 3.2.2), including its uniform parameters, specialization parameters, entry point method, and GPU shader code methods. Our translator operates on each C++ translation unit individually, creating internal representations for all ShaderClasses and GPU functions within. Then, our host and GPU backends use these internal representations to generate UE4-compatible C++ and HLSL code,

[15]`https://clang.llvm.org/docs/LibTooling.html`
[16]`https://github.com/redorav/hlslpp`

respectively.

The host backend generates one or more UE4 Global Shader class implementations (hereafter referred to as an *ImplClass*) for each ShaderClass. These generated ImplClasses use UE4's macro system to implement the host-side representation of a ShaderClass's uniform parameters, as well as its boolean-, integer-, and enum-type specialization parameters. If a ShaderClass has no ShaderClass-type specialization parameters, then only one ImplClass is generated. To support ShaderClass-type specialization parameters, the translator generates multiple ImplClasses based on all possible combinations of runtime types for each such parameter. For example, the shader in Listing 3.1 would result in three ImplClasses, one for each `FilterMethod` subtype. In addition, the translator also generates code to interface user-written ShaderClasses with their underlying ImplClass implementations. This task includes selecting which ImplClass to use based on the runtime types for each ShaderClass-type specialization parameter (if applicable), as well as communicating uniform and basic-type specialization parameters to their underlying UE4-based implementations. Thus, while our system uses UE4's under the hood, programmers do not need to interact with this underlying implementation directly. Instead, they can simply use the features provided by our unified system.

Our translator's GPU backend outputs an HLSL file for each ShaderClass with an entry point function.[17] A ShaderClass's generated HLSL file contains all of the GPU shader code needed for every ImplClass of that ShaderClass. This includes all uniform parameters and GPU functions from both the main ShaderClass as well as all ShaderClasses that it uses as specialization parameters (and their subtypes). Any code that is specific to an ImplClass (e.g., the code specifically for each `FilterMethod` mentioned above) is output under a distinct `#if` for that ImplClass. When generating executable kernel code from these HLSL files, each ImplClass supplies the proper `#define` option to the underlying HLSL compiler, ensuring that the generated shader variant is specialized to only the code it needs.

Our implementation also supports writing hardcoded HLSL directly within ShaderClasses and GPU functions. This code is copied to the output HLSL files as-is. This feature serves two practical purposes. Primarily, it lowers the barrier to porting shader code to use this system

[17]ShaderClasses without entry point functions are not invocable shader programs, so outputting HLSL files for them is unnecessary.

by allowing programmers to rewrite existing HLSL code incrementally, which better enables existing systems to adopt a unified shader design. Secondarily, as mentioned in Section 3.1, full C++-to-HLSL translation is a non-goal of our work. While our backend does convert some C++ code to HLSL, not all HLSL features are supported, nor do all C++ features translate to HLSL code properly. By supporting hardcoded HLSL in our current implementation, we are able to explore unified shader programming without first implementing every HLSL feature in C++, and vice versa, as a prerequisite.

Currently, our implementation only supports compute shaders, but we believe it can easily be extended to support other types of Global Shaders. We expect the biggest challenge will be coordinating inputs and outputs between different shader types (e.g., vertex shader outputs are used as pixel shader inputs). We can address this challenge by using the *pipeline shader* design [64]. Programmers would write ShaderClasses with both a vertex shader entry point and a pixel shader entry point. Then, within the ShaderClass, they would provide a singular definition for the data that is passed between these two shader types. We present an expanded discussion on supporting additional shader types in Section 5.3.1.

3.4 Evaluation

To evaluate whether our unified design enables better software engineering practices in shader development, while still maintaining the high performance necessary for real-time graphics, we ported shaders from UE4 to use our system. Because feature-complete C++-to-HLSL translation is out of scope for this work, we use hardcoded HLSL code (Section 3.3) in some parts of our ported code. All results were obtained using UE4 version 4.25.4 built from source.[18] Since the unified shaders contain both host and GPU code, we rebuilt the modified files accordingly prior to benchmarking the ported code. We review our findings in the sections below.

3.4.1 ShaderClass Modularity

Listing 3.5 shows a simplified segment of GPU code from UE4's temporal anti-aliasing (AA) shader, and Listing 3.6 shows the same segment rewritten in our system. This code implements three different methods for caching texture reads, and the decision about which method to use

[18]We used the release branch at commit b1e746725e8e540afe7ac586496b4ee4c081a10e

```
Texture2D SceneDepth;
SamplerState SceneDepthSampler;
Texture2D SceneColor;
SamplerState SceneColorSampler;

#if CACHE_METHOD == REGISTER_CACHING
  #define PRECACHE_COLOR

  void PrecacheColor(inout InputParams Input)
    { /* Sample from SceneColor; cache in Input */ }

  float4 SampleCachedColor(InputParams Input)
    { /* Return color from cache in Input*/ }

#elif CACHE_METHOD == GROUPSHARED_CACHING
  #define PRECACHE_DEPTH
  #define PRECACHE_COLOR

  void PrecacheDepth(InputParams Input)
    { /* Sample from SceneDepth; cache in groupshared memory */ }

  float SampleCachedDepth(InputParams Input)
    { /* Return depth from groupshared cache */ }

  void PrecacheColor(InputParams Input)
    { /* Sample from SceneColor; cache in groupshared memory */ }

  float4 SampleCachedColor(InputParams Input)
    { /* Return color from groupshared cache */ }
#endif

#if !defined(PRECACHE_DEPTH)
  void PrecacheDepth(InputParams Input)
    { }

  float SampleCachedDepth(InputParams Input)
    { /* Return depth sampled from SceneDepth */ }
#endif

#if !defined(PRECACHE_COLOR)
  void PrecacheColor(InputParams Input)
    { }

  float4 SampleCachedColor(InputParams Input)
    { /* Return color sampled from SceneColor */ }
#endif
```

Listing 3.5: A simplified selection of HLSL GPU shader code taken from UE4's temporal anti-aliasing shader. The corresponding host code for this GPU code is not shown.

```
class [[ShaderClass]] NoCaching {
public:
  [[uniform]] Texture2D SceneDepth;
  [[uniform]] SamplerState SceneDepthSampler;
  [[uniform]] Texture2D SceneColor;
  [[uniform]] SamplerState SceneColorSampler;

  [[gpu]] virtual void PrecacheDepth(InputParams Input) const {}

  [[gpu]] virtual float SampleCachedDepth(InputParams Input) const
    {/* Return depth sampled from SceneDepth */}

  [[gpu]] virtual void PrecacheColor(InputParams Input) const {}

  [[gpu]] virtual float4 SampleCachedColor(InputParams Input) const
    {/* Return color sampled from SceneColor */}
};

class [[ShaderClass]] RegisterCaching : public NoCaching {
public:
  [[gpu]] virtual void
  PrecacheColor([[inout]] InputParams Input) const override
    {/* Sample from SceneColor; cache in Input */ }

  [[gpu]] virtual float4
  SampleCachedColor(InputParams Input) const override
    {/* Return color from cache in Input*/}
};

class [[ShaderClass]] GroupsharedCaching : public NoCaching {
public:
  [[gpu]] virtual void
  PrecacheDepth(InputParams Input) const override
    {/* Sample from SceneDepth; cache in groupshared memory */}

  [[gpu]] virtual float
  SampleCachedDepth(InputParams Input) const override
    {/* Return depth from groupshared cache */}

  [[gpu]] virtual void
  PrecacheColor(InputParams Input) const override
    {/* Sample from SceneColor; cache in groupshared memory */}

  [[gpu]] virtual float4
  SampleCachedColor(InputParams Input) const override
    {/* Return color from groupshared cache */}
};
```

Listing 3.6: The unified C++ shader code ported from the code in Listing 3.5. Because the uniform declarations are shared between host and GPU code, this selection shows both halves of shader code.

is controlled by a specialization parameter. While Listing 3.5 only presents the original GPU code, our rewritten version in Listing 3.6 necessarily shows both host and GPU code because of the unified design.

From this simplified example, we can observe several ways in which our design leads to clearer, more maintainable code. First, the uniform definitions in the original GPU code (Listing 3.5 lines 1–4) are expressed as global variables, and each must have a corresponding definition in the original UE4 host code (not shown here). In contrast, in our implementation, uniform parameters are declared once for both host and GPU code (Listing 3.6 lines 3–6), and the uniforms are encapsulated within a ShaderClass. This encapsulation makes clear which uniform parameters are required when using this segment code. In the original UE4 HLSL file, these global uniforms parameters are declared alongside many others, even though they are only used within the segment shown here.

Similarly, our ShaderClass design clearly shows the code reuse relationship between the different caching implementations. `NoCaching` declares and provides implementations for four virtual member functions, and `GroupsharedCaching` overrides all four of them to provide its own implementations. `RegisterCaching`, however, only overrides two of these functions and uses the default implementations from `NoCaching` for the other two. With careful examination, one can observe the same pattern in the HLSL code in Listing 3.5. However, when looking at the original UE4 HLSL file, programmers must track down the `PRECACHE_DEPTH` and `PRECACHE_COLOR` dependencies across ~500 lines of code in order to discover the overall code structure that our design instead makes readily apparent. In total, our unified design provides clear modularity that spans both the host and GPU portions of shader code, whereas in non-unified systems such as UE4's, programmers must carefully manage component dependencies across the boundary between host and GPU code.

3.4.2 Lines of Code

Since our system design utilities various abstractions for shader programming, we want to verify that these abstractions do not lead to excess code bloat. Table 3.1 compares the lines of code (LOC) for our rewritten shaders against the corresponding original UE4 code. In UE4, an HLSL

[19] `https://github.com/AlDanial/cloc`

Table 3.1: Lines of code (LOC) comparisons for original UE4 shader code vs. the versions ported to our unified system. We report only non-commented, non-empty lines, as reported by CLOC.[19] The UE4 LOC number for each shader includes both the C++ file (host code) and the corresponding HLSL file (GPU code), while the unified code uses a single file for both host and GPU code.
*The unified C++ file includes some hardcoded HLSL code, since full C++-to-HLSL translation is out of scope for this work. This embedded HLSL code is included in the LOC counts.

Shader	**Original UE4 Code**	**Unified Code**
	C++ file & HLSL file	C++ file*
	Lines of Code	Lines of Code
Motion Blur Filter	902	920
Temporal AA	2,138	2,251

file can contain code for multiple shader programs; however, we have not necessarily ported all shader programs within an HLSL file to use our system. To present a fair comparison, we only count lines of HLSL code related to the shader programs we have ported.

As shown, the LOC counts for the unified shader code are comparable to the original code. The additional lines in the unified code come primarily from stylistic choices (e.g., putting the `[[gpu]]` function attribute on its own line). However, some additional lines come from temporary code duplication. Because we have not ported all UE4 HLSL files to our system, some code in our unified files is duplicated from HLSL header files that were `#included` in the original shader code (and, thus, this code is not counted in the UE4 LOC numbers). While this duplication is ideally temporary, programmers still need to manage this code as a necessary overhead when incrementally porting large systems. We believe the benefits of a unified system outweigh this extra temporary overhead, especially given that unified programming can reduce code duplication by allowing host and GPU code to share types, functions, and parameters.

3.4.3 Performance

Lastly, we evaluate the impact of our unified design on the runtime performance of GPU code generated by our translator. We run the Infiltrator Demo [22] (Figure 3.1) using both the original UE4 shader code and our rewritten versions and compare the GPU performance in Table 3.2. These results were produced using a resolution of 2560×1440 on a machine with an Intel Core

Table 3.2: GPU performance comparisons for original UE4 shader code vs. the versions ported to our unified system. The table shows the minimum, average, and maximum per-frame execution time in milliseconds for these shaders when running the Infiltrator Demo [22]. These numbers were obtained using benchmarking tools provided by UE4.

Shader	**Original UE4 Code**			**Unified Code**		
	(time in ms)			(time in ms)		
	Min	Avg	Max	Min	Avg	Max
Motion Blur Filter	0.06	0.18	0.70	0.06	0.18	0.70
Temporal AA	0.23	0.28	0.74	0.24	0.28	0.75

i7-6700K CPU and an NVIDIA Titan RTX GPU. As shown in the table, the performance of the shaders ported to our unified environment is comparable to the performance of the original code.

Figure 3.1: A screenshot from the Infiltrator Demo [22]. We use this demo for our performance evaluation.

3.5 Chapter Conclusion

In this chapter, we have presented the design of a unified programming environment for real-time graphics in C++. By co-opting existing features of the language and implementing them with alternate semantics, we are able to express the necessary shader-programming-specific features, including first-class support for GPU code specialization. Our system allows programmers to write host and GPU shader code using familiar modularity constructs in C++, and our source-to-source translator transforms this code into efficient standard C++ and HLSL.

A major focus of this work is specialization—a crucial optimization in real-time graphics—because the current techniques used to express and implement specialization do not work when GPU and host code share a unified, C++-based environment. The underlying issue is that specialization parameters should be runtime in host code but compile-time in GPU code. This observation is easy to overlook when using two separate environments with distinct parameter definitions and compiler toolchains, but it creates a fundamental tension in a unified system. Given specialization's importance in real-time graphics, any future unified system will need a solution to support specialization. By keeping our system as close to standard C++ as possible, we hope that our ideas can provide a foundation for supporting unified shader programming, thereby enabling future work to focus on other challenges in graphics programming.

Chapter 4

Staged Metaprogramming for Shader System Development

By removing the constraint of using a popular real-time graphics programming language, we can expand the scope of potential implementation strategies for building a unified shader programming environment. Like any modern discipline, the field of programming languages is continually evolving, with new languages exploring novel programming techniques and older languages incorporating the lessons from these newer methods over time. Investigating opportunities outside of the popular languages of today can provide insights that help developers of current and future languages decide which features to adopt, as well as provide guidance for using these features if and when they are adopted.

When examining the disparate techniques used to implement shader systems (Section 2.3), we observe that they largely fall under the umbrella of *metaprogramming*. We broadly define metaprogramming as writing code that manipulates other code, which includes reading, analyzing, transforming, or generating code. Textual-based processing tools, custom DSL implementations, and shading language compiler modifications all fit this definition. 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. Therefore, we hypothesize that the effort required to implement a robust shader system can be reduced by making metaprogramming a fundamental design principle and

This chapter is largely taken from our paper "Staged Metaprogramming for Shader System Development" [68].

utilizing a metaprogramming technique that sidesteps this apparent trade-off between capability and complexity.

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,[20] 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 4.3. Prior to that, we introduce staged metaprogramming, the underlying methodology on which it is built (Section 4.2). To motivate our decision to use staged metaprogramming, we examine other methods of creating shader systems (Section 2.3), which also inform our design goals (Section 4.1). We then use Selos to explore static versus dynamic composition of shader features in Section 4.4.

4.1 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.

[20] `https://github.com/kseitz/selos`

- **Early error detection**

  Underlying graphics APIs, as well as many shader systems, expose shader parameters to host 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)**[21]

  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 conveniences, 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 2.3), 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 4.2).

[21]`https://en.wikipedia.org/wiki/Don%27t_repeat_yourself`

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 work. 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 4.5.

4.2 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 2.3), 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.

4.2.1 Definition

Our definition of staged metaprogramming aligns with the description of a *multi-level* language in Taha's dissertation [72]. 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 [48]).

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 [5]. 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-quoted code using an *eval* mechanism.

As we will discuss in Section 6.4, 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 staged metaprogramming 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.

4.2.2 Example Shader

Returning to the surface shader example from Section 2.3, 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 variant.

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 pre-processor `#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 4.4 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 host-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 4.3.2, we show how staged metaprogramming enables our shader DSL implementation, and we also present a description of this syntax with a more complex example.

4.2.3 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 4.2.1), we must use a different language to demonstrate why these features are useful for shader systems. We want to model the programming environments of typical game engines as closely 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 [16].

Lua-Terra is a multi-stage language that uses Lua [35] 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 [65], as well as future versions of C++, are trending toward supporting staged metaprogramming facilities, as we discuss in Section 4.5. 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.

4.2.4 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 4.3.2). 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.

4.3 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. Figure 4.1 shows an overview.

In Selos, programmers write shader code in Terra (Section 4.3.3), with custom syntax extensions for shader-specific features (e.g., Listing 4.1). Our shader DSL (Section 4.3.2) implementation parses the shader-specific features and generates a Shader Intermediate Representation, or SIR, object (Section 4.3.1). 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 4.3.5) 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 4.3.4) from the SIR to allow engine and application code to control shader

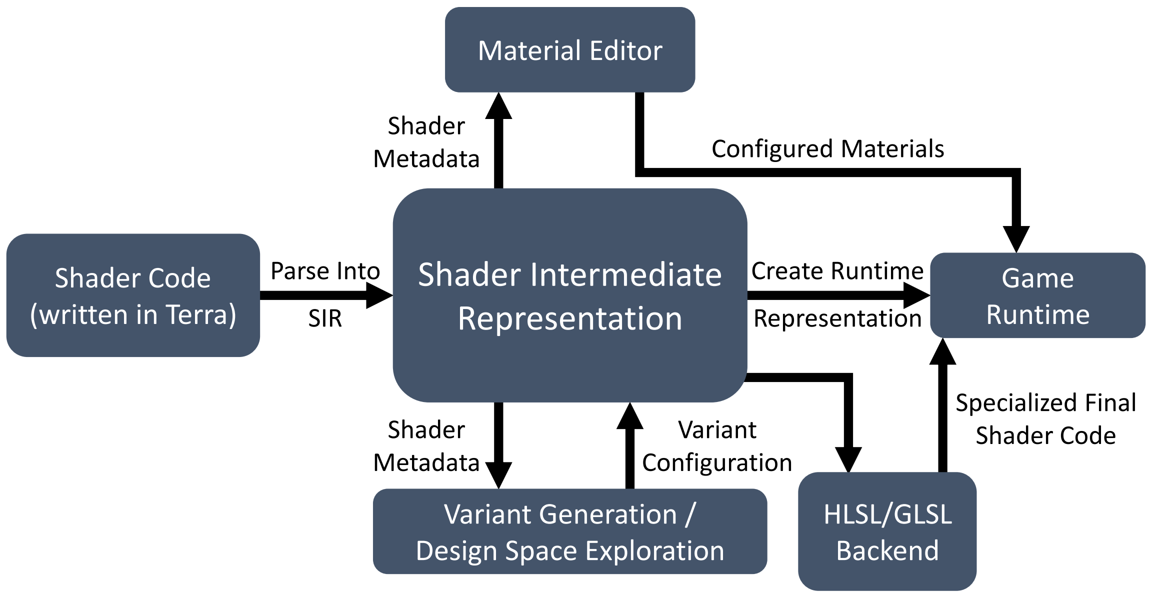

Material Editor
Shader Metadata
Configured Materials
Shader Code (written in Terra)
Parse Into SIR
Shader Intermediate Representation
Create Runtime Representation
Game Runtime
Specialized Final Shader Code
Shader Metadata
Variant Configuration
Variant Generation / Design Space Exploration
HLSL/GLSL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

Figure 4.1: An overview of the Selos Shader System, as discussed in Section 4.3.

parameters and to set up 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.

4.3.1 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 is in contrast 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 an SIR shader. Along with storing names and type information, an 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.

4.3.2 Write Shader Definitions Using a DSL

As Section 4.2.4 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 4.3.3) and use the DSL syntax to express shader-specific features that are not inherent in Terra, such as declaring uniforms, inputs, outputs, and textures. Listing 4.1 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., Listing 4.1 line 9), avoiding the double-declaration issue in ShaderLab. By including both vertex program and fragment program code in the same shader, varying parameters and shared uniform buffers are declared only once as

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

Listing 4.1: A simple shader in our Selos DSL syntax. This shader, which a technical artist might write, computes directional lighting (line 34), modulated by a diffuse texture map (line 33). The DSL syntax allows vertex **`inputs`** and fragment **`outputs`** to be declared, along with blocks of **`uniform`** parameters. Shaders also contain explicit **`params`**, which can then be used to set uniforms from within host shader code (lines 13–14). Uniforms without initializers automatically become explicit parameters (line 9). Parameters can also contain information about how they should be exposed to artist GUI applications (line 9). Ordinary Terra statements inside **`code`** blocks are attached to the vertex or fragment kernel, with intermediate values carried by **`varying`** parameters. `Position` (line 27) is a built-in variable for specifying vertex position (equivalent to GLSL's `gl_Position`).

```
local SimpleShader  = ShaderBuilder.new("SimpleShader")
local diffuseMap = SimpleShader:declareTextureSampler(sampler2D)

local model = SimpleShader:declareParam(mat4)
local view  = SimpleShader:declareParam(mat4)
local proj  = SimpleShader:declareParam(mat4)

local PerFrame        = SimpleShader:declareUniformBlock()
local lightDirection  = PerFrame:declareUniform(vec3, nil, Slider3)

local PerObject     = SimpleShader:declareUniformBlock()
local modelViewProj = PerObject:declareUniform(mat4,
  quote proj*view*model end)
local modelViewIT   = PerObject:declareUniform(mat4,
  quote inverse(transpose(view*model)) end)

local position   = SimpleShader:declareInput(vec3)
local normal     = SimpleShader:declareInput(vec3)
local uv         = SimpleShader:declareInput(vec2)

local vNormal    = SimpleShader:declareVarying(vec3)
local vUV        = SimpleShader:declareVarying(vec2)

local outColor   = SimpleShader:declareOutput(vec4)

SimpleShader:addVertexCode(quote
  Position = modelViewProj * make_vec4(position, 1)
  vNormal  = (modelViewIT * make_vec4(normal, 0)).xyz
  vUV = uv
end)

SimpleShader:addFragmentCode(quote
  var diffuse = texture(diffuseMap, vUV)
  outColor = diffuse * max(0, dot(vNormal, lightDirection))
end)

SimpleShader:finalize()
```

Listing 4.2: By using our Lua *shader builder* API, the shader in Listing 4.1 can also be constructed programmatically, without custom syntax. Other system components, like Variant Generation and the HLSL/GLSL Backends, use this API to construct and modify shaders.

well. Furthermore, our DSL's method of expressing specialization options is akin to that of TFX and Slang (as described in Sections 2.3.3 and 2.3.4, respectively). Our method provides greater flexibility than the simple preprocessor-based methods of Unity (Section 2.3.2). We discuss this method further in Section 4.3.5.

The implementation of our DSL is driven by staged metaprogramming. Our parser constructs an SIR shader from DSL code by calling into an underlying *shader builder* API, written in (compile-time) Lua code.[22] Other Selos components can use this API to programmatically construct and modify shaders, which does require writing some metaprogramming code directly. Listing 4.2 shows how the shader from Listing 4.1 can be constructed using the builder API. Note the explicit use of key staged metaprogramming features (listed in Section 4.2.1)—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 4.1 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 4.3.3). We believe these backends are not engine-specific and can be shared between multiple shader systems as an open-source component, similar to hlsl2glslfork [62].

While lines-of-code metrics are not standalone proof of the effort required to use a programming technique, Table 4.1 suggests that staged metaprogramming is similar in complexity to using textual-based preprocessing methods like in ShaderLab (given that Terra is C-like and

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

Table 4.1: 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.[23] *The Unity count was obtained via personal communication and is estimated to include 10–15% blank lines and comments [63].

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

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.

4.3.3 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 host and GPU 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 GPU 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, meaning that they are usable in application code as well.[24] Other shader systems

[23] `http://cloc.sourceforge.net/`

[24] We heavily utilized metaprogramming when implementing the host-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`

typically provide a host-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 host and GPU code, programmers need not worry about any differences between the host and GPU types. When used in GPU 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 host and GPU code allows programmers to debug shader code by running it on the CPU. Furthermore, it is easier to migrate compute-intensive host code to GPU compute shader programs.

Our decision to write shader logic in Terra was motivated not only by the benefits of using the same language for both host and GPU code, 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 GPU shader code. Instead, Terra's language frontend parses and syntax-checks the quotes, which are then stored 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 4.1).

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.

4.3.4 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 4.2.2).

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 host and GPU code by default (Section 4.3.3). 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 host-side setup logic that is expressed with shader code (e.g., lines 13–14 in Listing 4.1). 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 [47], 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 host-side logic requires recompiling the game executable.[25] 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 mod-

[25]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.

ification to shaders that would be type-checked and recompiled into the application behind the scenes.

4.3.5 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 4.4, 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 4.2.2, so we omit such a discussion here. By using the `ConfigurationOptions`, our system allows experienced developers to use direct staged metaprogramming to implement 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 `#ifdefs`, 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 4.4).

However, using direct staged metaprogramming 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.

4.4 Exploring the Specialization Design Space

4.4.1 Background and Motivation

Through staged metaprogramming, Selos allows us to target challenging problems faced by designers and users of modern shader 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 GPU 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; shader programs 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 shader programs to the features needed on a per-tile basis (where a tile is a 16×16 group of pixels) [17]. 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 `ifs` 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 variant that omits the code for that light type when shading that tile.

However, overspecialization can lead to negative consequences. Generating the full set of shader variants for all combinations of material and light types results in a combinatorial explosion of 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 trade-offs between compile-time and runtime specialization in order to achieve the best performance; however, the variant design space is large, so automatically exploring this trade-off 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.

4.4.2 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 [23]. 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 [8] 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 [30] 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.

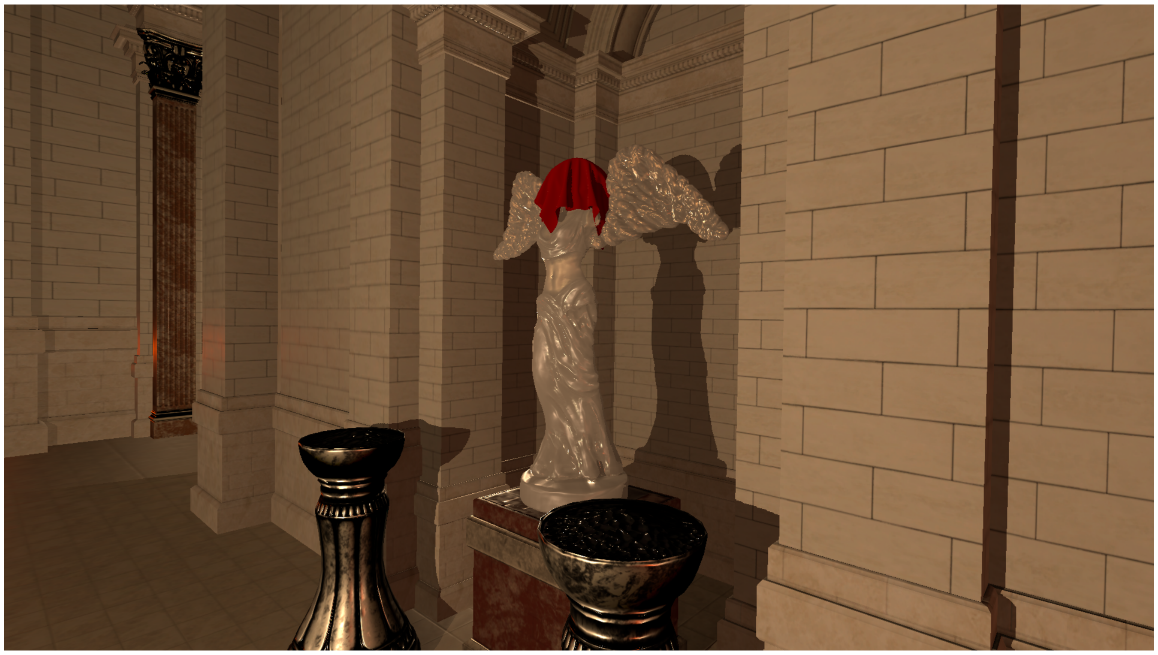
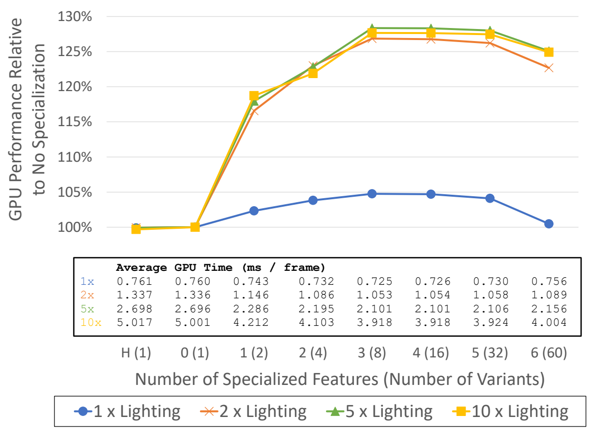


Figure 4.2: The test scene used for evaluating different sets of shader variants. This scene is a modified version of the ORCA Sun Temple, in which we added red cloth to the angel statues.

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. Figure 4.2 shows an image of our scene.

In order to find the optimal trade-off 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 $0 \leq k \leq n$ and $n = 6$ (number of material types + number of cullable light types). This generation results in $\binom{n}{k}$ variant sets for each k.

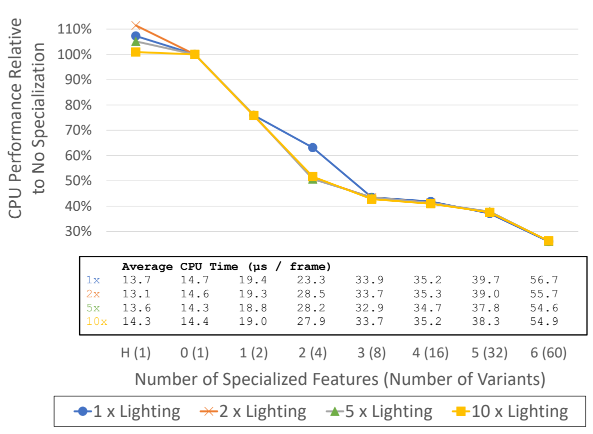
Generating these variant sets was straightforward in Selos, due to staged metaprogramming. We authored two `ConfigurationOptions` that control how specialization options compile into shader variants: `TiledDeferredMaterialType` for material types and `TiledLightListEnv` for light types. The system specifies which type(s) to include in a given variant, and these implementations modify the SIR of the deferred



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

Figure 4.3: 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 4.4 (Figure 4.2). 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.

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 program (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 Appendix A.

4.4.3 Performance Results

We run our deferred renderer on the modified Sun Temple scene for each variant set and present the results in Figure 4.3 for the best performing set for each value of k.[26] We also hand authored an HLSL shader program equivalent to the fully general case (as is the default for deferred rendering) and compare its performance in Figure 4.3 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 [12]. 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 [2], Detroit: Become Human has 124 lights [46], and Doom has ~300 light sources [69].

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 (Figure 4.3a). 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 Figure 4.3b, 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 program is comparable to that of the fully general shader variant generated by our system. Therefore, the code generation and manipulation that Selos performs does not negatively impact the performance of final GPU 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 trade-

[26] 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).

offs 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.

4.5 The Future of Metaprogramming in C++

As mentioned in Section 4.2.3, the common programming languages used in real-time graphics programming do not have the features required of staged metaprogramming, hence our decision to study this technique using Lua-Terra. However, we are optimistic that these languages will adopt more interesting metaprogramming capabilities in the future.

For example, some recent proposals to the C++ Standards Committee seek to add more robust and powerful metaprogramming facilities to the language. P0194 [11] 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 [77] 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 [71] 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 [6] 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.

4.6 Chapter Conclusion

In this chapter, we have demonstrated how staged metaprogramming provides the proper facilities with which to build an expressive, unified shader system, complete with a unifying shader intermediate representation, the ability to express both host and GPU 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 over-specialization. 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 therefore wish to encourage future programming languages, as well as future versions of today's popular languages, to include support for staged metaprogramming so that developers—both in real-time graphics and in other domains—can take advantage of this powerful, flexible, and general-purpose set of language features.

Chapter 5

Discussion

In the previous two chapters, we have presented two methods for building unified shader programming environments: co-opting existing features of a programming language and implemented them with alternate semantics (Chapter 3) and staged metaprogramming (Chapter 4). While both methods enable us to achieve our high-level goal, they have significantly different trade-offs, both in the resulting user-facing systems and in terms of what they provide to system builders. We now turn our attention to discussing these trade-offs to aid future shader system builders in choosing an implementation strategy that best suits their needs. We begin by presenting the high-level takeaways, followed by an in-depth analysis of each method in the two sections that follow.

Co-opting Existing Features Real-time graphics developers with large investments in an existing language, or those looking to create unified shader programing systems in the near term, should use the co-opting method. In principle, one can co-opt the features of any language, allowing developers to add graphics programming to their languages of choice. This strategy lowers the barrier to entry for shader programming by enabling users to utilize familiar language features when writing shader code. However, because code using co-opted features looks like ordinary code but operates with alternate semantics, users must contend with behaviors that might contradict their understanding of a programming language's rules. Moreover, system creators can only co-opt features already present in a language, which limits the extent to which a language can be customized to the graphics domain. A related downside is that implementing alternate semantics for co-opted features requires creating and maintaining a compiler-based tool, which increases shader system development costs. Nonetheless, a unified programming environment provides substantial benefits to end users, so we think that these additional costs are acceptable for existing large-scale systems.

Staged Metaprogramming While not practically useful today, we believe that staged metaprogramming is the better long-term approach for creating unified shader systems. Because developers can build unified environments entirely in user-space code, staged metaprogramming results in reduced development and maintenance costs relative to compiler-based implementations. Staged metaprogramming's key features provide system builders with the flexibility to create customized, layered implementations of graphics-specific features, which better enables them to adapt their systems to the needs of graphic programming. This flexibility comes with the cost of additional programming complexity—developers must reason about multiple stages of code execution, including understanding which portions of code run in which stage of execution, what code is generated, and how that generated code is used. Therefore, we advocate that end users should not write staged metaprogramming code directly. Rather, expert programmers should use it as an underlying implementation technique to provide appropriate higher-level interfaces to the various users of the system. The biggest downside of staged metaprogramming is that it can only be used in languages that support its required features, which excludes the popular languages used in real-time graphics today. We encourage the maintainers of current languages and future language designers to build staged metaprogramming into their languages both for graphics developers and for developers in other domains who will benefit from this general-purpose programming technique.

5.1 Analysis of Co-opting Existing Features

Perhaps the most prominent difference between the two implementation techniques is the requirements they impose on the underlying programming language. An advantage of the co-opting approach is that one could theoretically co-opt the features of any language in order to add support for different semantics and optimizations. The specific features to co-opt will depend on the specifics of the language, and some languages may be more amenable to modification than others. Nevertheless, the underlying idea is, to a first approximation, transferable to many different kinds of languages. In contrast, utilizing staged metaprogramming very clearly requires using a language that supports its key features as defined in Section 4.2.1. Unfortunately, the languages most commonly used in real-time graphics programming today do not support staged metaprogramming. These languages are unlikely to add staged metaprogramming features in the near future, and switching to a new language is typically cost-prohibitive for existing, large-scale game engines. Therefore, the benefits of this technique are difficult to employ in modern shader systems today, whereas the co-opting method provides an avenue for existing systems to integrate unified shader programming in the near term.

Because our use of the co-opting technique emphasizes minimizing changes to the underlying language by repurposing existing features, the resulting system presents users with a familiar programming environment. Programmers can utilize C++'s modularity features (classes and virtual functions) for shader programming, much like they would for other parts of the application. As a result, shader code looks and feels much like regular C++ code, but the translation tool compiles this code into implementations that are more efficient on the underlying target hardware. This familiarity and consistency of feature usage is an advantage of this approach because it limits the scope of engine-specific programming paradigms with which shader programmers must contend, while still enabling efficient final executable code. Newcomers to an engine can more easily read, understand, and begin writing shader code without requiring that they first learn new shader-specific programming aspects imposed—whether explicitly or implicitly—by the shader system. However, this benefit comes with a significant detriment: when choosing how best to express elements of shader programming, system builders are limited to using only the features available in their chosen language. The available features (either as is or co-opted) may or may not be well suited to the needs of shader programming, so engine developers may have to make compromises in expressibility, flexibility, or performance. Moreover, code using co-opted features with alternate semantics looks identical to code using those same features but with standard semantics, which could cause confusion for users when the co-opted code behaves counter to their expectations. We believe these trade-offs are the right choice for most existing systems, not only due to the reasons discussed above, but also because minimizing changes to C++ helps to ensure that user-written code remains compatible with newer versions of C++ as the language continues to evolve.

From a system builder's perspective, a significant downside of employing the co-opting method is the need to build and maintain a compiler-based tool. Creating a compiler from scratch requires a massive investment in time and effort and is likely cost-prohibitive for all but the largest development teams. Thus, building a compiler-based tool is, in practice, tractable only if there exists some existing framework that one can extend or build upon. Many languages have open-source compilers, so creating a custom fork of a compiler is one option. However, compiler codebases tend to be large and complex, so determining what code to modify and how in order to compile certain features with alternative semantics is non-trivial. Furthermore, modifying a compiler directly can lead to integration challenges when pulling in changes from the upstream source, resulting in significant additional maintenance costs.

A better approach—if available—is to build on top of a compiler framework that provides explicit mechanisms for custom extensions and tools, which can help to alleviate both of the challenges mentioned above. Our decision to use Clang for our translation tool in Section 3.3 was motivated by this reasoning. Clang has multiple APIs to enable custom extensions, providing different trade-offs for different use cases.[27] We chose the LibTooling API for two reasons: 1) our tool only needs to be run on files containing shader code, rather than on every file in the build process and 2) this API allows our tool to live external to the Clang codebase, which streamlines the process of updating to newer versions of Clang in the future. However, our implementations is a research prototype, so the evaluation of Clang's APIs may differ for system builders aiming to create a production-quality implementation. For example, a Clang Plugin might be a better fit if the system already uses Clang for compilation, as it would more easily integrate with the existing build process. No matter which API one chooses, switching to a newer version of Clang can still result in additional maintenance costs, especially if the APIs themselves change between versions. Nonetheless, Clang provides a mature and well-supported platform on which to build a compiler-based tool for C++ (and the other languages that Clang supports). Other compiler frameworks, especially newer ones, might not have as robust ex-

[27] A summary of these APIs and their pros and cons is available here: `https://clang.llvm.org/docs/Tooling.html`

tension and tooling systems, so developers seeking to integrate unified shader programming into other languages may need to invest additional effort to achieve their goals when using the co-opting technique.

The need for an additional compiler-based tool can also lead to increased compilation times for shaders (which is a downside shared with staged metaprogramming, further discussed below). Our implementation parses each unified C++ file into an Abstract Syntax Tree (AST), performs analysis using information from the AST, and then generates two files: one containing GPU code in HLSL and the other containing host code in standard C++. These two files must then be compiled by downstream HLSL and C++ compilers, respectively. While the unified C++ source files for our examples are reasonably sized (Table 3.1), these files `#include` multiple core UE4 header files, which in turn include other header files (which might include more headers, and so on). As a result, a large amount of code must be parsed into an AST, which takes a significant amount of time.[28] Once the AST is created, our implementation's analysis and code generation steps run very quickly in comparison, but since our tool outputs source HLSL and C++ files, the same code must effectively be parsed again by the downstream compilers. Compiling the same code twice is the largest contributing factor to the increase in compilation times in our system. Using precompiled headers or C++20's *modules* feature could help to mitigate this issue. Additionally, integrating the analysis and code transformation passes directly into the main compiler (e.g., by creating a Clang plugin instead of a standalone tool) would improve compilation times. To be effective, this type of implementation would need to transform the internal representation of the host code directly during the standard compilation process, rather than perform source-to-source translation. The GPU code would still need to be separated out and recompiled by a downstream GPU compiler, but overall, this approach should lead to better compilation times compared to our current implementation.

A related issue, both in our systems specifically and in unified programming in general, is

[28]This downside disproportionally affects our ability to debug our compiler tool implementation, since Clang runs substantially slower when using the debug mode build compared to the release mode build. To work around the slow run times of the debug Clang build, we instead compiled Clang using the "RelWithDebInfo" build option, which performs more compiler optimizations on the Clang codebase than the "Debug" build option, while still generating debug symbols. The debug build of our translation tool links with this RelWithDebInfo Clang build. As a result, our tool's debug build runs much faster than if it were linked with a Debug build of Clang, but the tool's release build (which links with a "Release" build of Clang) runs faster still.

the need to recompile host code and GPU code more often compared to non-unified systems. This issue affects both our co-opting-based system and our staged-metaprogramming-based system. Since both host and GPU code share definitions of various programming constructs in a unified environment, modifying these constructs requires recompilation of both portions of code. If a code change only affects either the host code or the GPU code, then recompiling both is unnecessary. A simple diffing tool can greatly reduce the impact of this downside—if the generated HLSL or C++ code is equivalent to the version previously generated by the translation tool, then recompilation can be skipped. However, some changes to the unified code might lead to semantically equivalent but syntactically different generated code, resulting in additional unnecessary recompilations. In contrast, in a non-unified system, programmers always knows whether they are modifying only host code or only GPU code since these two portions of code are in different files. More important, however, is the fact that recompilation of both host code and GPU code oftentimes is necessary. For example, if a programmer changes a shared struct or a shader parameter, this change must be propagated to both host and GPU code. In non-unified systems, the programmer needs to manually change both the host-side representation and the GPU-side representation of this struct or parameter. Omitting the change on one side or the other—or otherwise failing to make the change consistently on both portions of code—could result in a bug. In contrast, this type of issue cannot occur in a unified system, by definition. Because of this benefit and the other advantages associated with programming in a unified environment, we believe that the additional compilation time costs of using the co-opting method are acceptable.

Overall, the approach of co-opting existing features and implementing them with alternate semantics is a good fit for developers of existing systems seeking to integrate shader programming into the host-side programming languages they are using today. The languages commonly used in real-time graphics do not inherently support the necessary features for unified shader programming as is, nor do they provide a sufficient set of features for adding shader programming without language modifications. Therefore, adding unified shader programming to these languages requires compiler modifications or a compiler-based tool, and the co-opting method is a way to help manage some of the costs associated with these types of implementations. We believe that the additional development costs are worthwhile for engine developers, because the resulting unified system would have the ability to greatly improve the engine user experience.

5.2 Analysis of Staged Metaprogramming

The greatest strength of staged metaprogramming is that it allowed us to create a unified shader system entirely in user-space code. In Section 5.1, we presented the biggest downside of staged metaprogramming today—that popular languages used in real-time rendering do not support its required features—but we now wish to reframe this idea. Based on our exploration, the features of staged metaprogramming are sufficient to add unified shader programming to a systems language. Thus, if a language adds this general-purpose mechanism in the future, then this language also automatically enables construction of unified shader programming environments, without needing to add any graphics-specific features to the language. We did not need to build a compiler-based tool or otherwise modify a compiler for the Selos implementation, thus avoiding a significant drawback of the co-opting approach. This property is an advantage not only for real-time rendering but also for other domains that could benefit from improved metaprogramming functionality.

From user-space code, staged metaprogramming provides a substantial amount of flexibility in the kinds of features programmers can build and express using it. System builders can create interfaces similar to those in existing systems but with improved properties (such as those discussed in Section 4.3), which has the advantage of presenting familiar constructs to users of the system (similar to our use of the co-opting method, as discussed above). Alternatively, they could design and implement drastically different systems to potentially provide users with significantly improved experiences (e.g., the design-space exploration framework in Section 4.4). Either way, staged metaprogramming enables engine developers to experiment with different shader programming features and interfaces in order to create expressive, powerful, and robust systems best suited to the needs of their users. Programmers can be confident that their unified shader system features will not conflict with future versions of the language because their implementations can be built entirely in user-space code when using staged metaprogramming.

As discussed in Section 4.2.4, debugging metaprogrammed code is a challenging problem in general, and staged metaprogramming suffers from this limitation. Nested code generation steps across multiple stages of metaprogramming can sometimes lead to cryptic error messages, requiring programmers (and possibly technical artists) to trace errors through various code locations and files. While the staged metaprogramming environment in Lua-Terra provides better error messages than those arising from typical C-preprocessor-based metaprogramming and template metaprogramming (but, of course, different languages will vary in this regard), confusing error messages can still permeate to end users, who might not be well versed in the metaprogramming constructs on which a shader system feature is built. In contrast, because the co-opting method requires building a compiler-based tool, developers can provide better error messages when parsing user-written code that uses the features of a unified shader system. In the future, we expect the debugging situation to improve through a combination of research and better tooling implementations. For example, in Lua-Terra, programmers can manually insert function calls to print quasi-quotes and functions from Lua code, which allows them to see the results of any code generation that happened prior. A future debugging tool could insert these printing calls automatically at various points in the program and display the results hierarchically, allowing programmers to visualize the results of metaprogrammed code across multiple stages of code execution.

With any metaprogramming code comes a potential for increased compilation times (and, as mentioned above, the co-opting method has this downside as well). Each time a file is compiled, all of the metaprogramming code must be evaluated. Since programmers can execute arbitrary code at compile time in a staged metaprogramming environment, this compile-time evaluation can become arbitrarily complex and time-consuming. An intelligent build system can help to mitigate this issue during incremental builds, but the impacts of increased compilation times might still be evident compared to systems with less metaprogramming.[29] Our use of staged metaprogramming emphasizes compile-time metaprogrammed so that code generation cannot impact runtime game performance; however, using runtime code generation can help to reduce compile times during the development process. Rather than statically executing all code generation at compile time, a system could evaluate only the metaprogrammed code that is used

[29]Readers might be familiar with a similar issue in C++: applications and libraries that make heavy use of template metaprogramming result in significantly longer compile times compared to those that do not.

in a particular invocation by using runtime code generation via a Just-In-Time (JIT) compiler. This setup would improve iteration time, which is often more important during development than in-game run times. Also, as discussed at the end of the previous section, our shader system based on staged metaprogramming results in additional host and GPU code recompilations compared to non-unified systems (an issue that will affect any unified system, regardless of implementation methodology). However, the same JIT-based solution could also help to lessen the impact of these recompilations. Runtime GPU code compilation is already common during the development process (e.g., to support hot reload of GPU kernels), so extending this idea to host code would be useful as well.

Extensive use of metaprogramming leads to increasingly complex code. Programmers must understand the code performing metaprogramming, the resulting generated code, and the interactions between multiple stages of code execution. This extra layer in the programming mental model can become arbitrarily complicated, which motivated our decision to hide most of the metaprogramming code behind the Selos shader DSL (Section 4.3.2). End users should not be exposed to the complexities of staged metaprogramming. Instead, it should be used as an underlying implementation technique for shader systems, utilized by expert programmers to create familiar interfaces for artists and technical artists and to explore optimization opportunities.

Staged metaprogramming provides a solid foundation upon which unified shader systems can be built. This methodology is not available in modern popular programming languages, but we hope that current languages will add support over time and that future languages will consider incorporating staged metaprogramming functionality from the outset. While the co-opting technique is more readily useful today, we believe that staged metaprogramming is a better long-term approach because it enables the creation of unified shader programming environments entirely in user-space code. Not needing to maintain a compiler-based implementation, while still providing powerful shader systems that remain compatible with future language versions, is a major advantage of staged metaprogramming.

5.3 Future Work

Our work has lead us to several ideas for future directions, both for near-term improvements and forwarding-looking advancements.

5.3.1 Supporting Additional Shader Types

On the practical side, we can improve the implementations of both of our systems by adding support for more shader types. Neither of our current implementations support all of the shader types available in modern graphics APIs. The Selos shader system supports vertex, fragment/pixel, and compute shaders, while the C++-based system supports only compute shaders. Compute shaders are an increasingly significant portion of a modern game's shader code, and they are sufficient for demonstrating the shader specialization issues that arise in a unified system. Thus, we chose to use compute shaders to show how our C++-based design solves these issues. Nevertheless, in order to be viable for modern, large-scale game development, supporting the other shader types is important too.

As mentioned in Section 3.3, the *pipeline shader* design [64] is a possible fit for supporting other shader types. A major challenge in utilizing multiple shader types is coordinating inputs and outputs between them (e.g., a fragment shader's inputs most often come from a vertex shader's outputs). In a unified environment, we believe this coordination should be enforced by the system. By authoring both a vertex shader entry point and a fragment shader entry point within the same encapsulation construct (e.g., a ShaderClass in the C++-based system or the Selos DSL's shader type), these entry points can share definitions of the data members that flow between them (e.g., the *varying* parameters in Listing 4.1 for the Selos shader system). Selos already supports this design for vertex and fragment shaders, and extending the C++ ShaderClass design similarly is relatively straightforward. Since vertex, fragment, and compute shaders make up the majority of a game's shader code, we have not yet attempted to incorporate geometry or tesselation shaders into either of our systems. We think our use of the pipeline shader design can extend to these shader types too; however, we are uncertain of the possible challenges that might arise in practice.

An alternative to pipeline shaders is to write the code for different shader types as independent ShaderClasses (or shader types in Selos's DSL) and then use earlier-stage ShaderClasses as class members in later-stage ShaderClasses. For example, a fragment ShaderClass could include a vertex ShaderClass as a class member and reference its parameters directly (similar to how ShaderClass specialization members are used). This design would lead to better modularity and reuse compared to authoring the code for both shader types directly within the same ShaderClass, which may also be preferable when expanding support beyond the vertex/fragment case to include geometry, tesselation, and mesh shaders too. The best approach is likely a hybrid of the two designs, allowing users to utilize the pipeline shader style while also supporting composition of separately written shader types. While these kinds of designs have been explored before, seeing how a unified system changes their benefits and pitfalls will be interesting.

5.3.2 Static, Dynamic, and Hybrid Dispatch Strategies

Further exploration of static specialization, dynamic dispatch, and hybrid combinations of the two (where the system statically specializes some features while using dynamic dispatch for others) is another open area for future work. For example, given user-written shader code for Standard, Subsurface, Hair, and Cloth material types, we would like the option to generate four statically specialized variants (one for each material type), as well as one shader variant that uses dynamic dispatch within GPU shader code to select between all four material types. In addition, we would like to generate shader variants that are partially specialized to a subset of material types, which then use dynamic dispatch to select between the types in their respective subsets (e.g., one variant for Standard plus Subsurface, and a second variant for Hair plus Cloth). Selos already supports a version of each of these (described in Section 4.4), and we are interested in expanding the C++ ShaderClass design similarly. Our decision to co-opt virtual functions intentionally leaves open the possibility of compiling shader variants that use dynamic dispatch—or a hybrid of static and dynamic dispatch—in the future. Given the importance of specialization, supporting it in our system took priority, but we believe that in the future, dynamic dispatch will serve an increasingly important role in modern game rendering systems. By expanding our ShaderClass implementation to support both static and dynamic dispatch, programmers could better modularize various aspects of their shader code, and the underlying system could then generate either fully specialized, partially specialized, or fully dynamic shader variants to improve overall performance without manual changes to user-written code.

Expanding support for dynamic and hybrid dispatch strategies beyond the limited cases covered by the Selos implementation presents further interesting research challenges. The current system only handles cases where different options at a dispatch point use a same underlying representation for the data they need. For example, to dynamically dispatch between different material types, these material types must all either use the same data or be authored to map their data to a common underlying representation (e.g., a Geometry Buffer—or GBuffer—in a deferred renderer). A more useful setup would automatically determine how to map heterogeneous data from different shader code options into a homogeneous representation when using dynamic dispatch for this code, but creating this automatic mapping is non-trivial. Concatenating the data for each option into a single datatype would lead to bloated data transfers that would negatively impact performance. Thus, the system should attempt to de-duplicate and pack data into a compact representation, while also ensuring that the combined datatype can efficiently interface with its separate-datatype counterparts used elsewhere in the application. Even with a best-effort mapping, the resulting combined data representation will likely still be larger than the separated representations, adding another axis that affects overall performance.

Another major question is how to decide when to use static versus dynamic dispatch. The exploration in Section 4.4 used an exhaustive search to find the optimal performance, but the specific results will differ based on end-user machine configurations. It might not be feasible to run this kind of analysis on an end-user's machine at game runtime without interrupting the gameplay experience, so performing a simpler runtime analysis would be more viable. Another option is to use static analysis of the code to make static specialization / dynamic dispatch decisions ahead of time. Our explorations lead us to believe that differences in register pressure are a major contributing factor in determining which options are most important to specialize.

Along with the above, other design decisions will inevitably arise while investigating more complex dispatch scenarios. For example: At runtime, how should the system communicate to the GPU code which option to invoke when using dynamic dispatch? How does this added communication affect memory bandwidth usage and performance for host, GPU, and host-to-GPU memory transfers? If graphics APIs support function pointers and virtual functions in the future, can and should the system be adapted to utilize these features? All of these questions will lead to interesting design decisions, both for the underlying implementation and for the developer-facing programming model.

5.3.3 Real-time Ray Tracing

Given its growing relevance in real-time graphics applications, supporting hardware-accelerated ray tracing using both the co-opting method and staged metaprogramming is an important future direction. We think automatic generation of static, dynamic, and hybrid dispatch strategies (as described above) is crucial to flexible and efficient ray tracing support in unified systems. When a ray intersects with an object, the hardware invokes a compiled shader kernel associated with that object. Ideally, the hardware should group together invocations of the same kernel into batches for efficient execution. If there are too many different compiled kernels within a scene, then the hardware may be unable to completely fill each batch. Instead, we can group together shader code into fewer overall compiled kernels by using dynamic dispatching within each kernel to execute different portions of code appropriately. A system that compiles user-written shaders into a variable number of executable kernels would allow programmers to experiment with the trade-offs between batch size versus the cost of using dynamic dispatch within the kernels. While we believe that both of our shader system designs can be extended to support real-time ray tracing by utilizing their respective underlying implementation methodologies, we cannot be certain that the decisions we made for rasterization and compute shaders will translate cleanly to ray tracing.

5.3.4 Co-opting the Features of Rust

Beyond C++, we are interested in utilizing the co-opting approach to bring unified shader programming to other languages. Given the ongoing work on Rust GPU [20], Rust is a potential next choice, and Rust's generics seem like a viable feature to co-opt to express shader specialization decisions. The current Rust compiler uses *monomorphization*[30] to implement generics, which is analogous to static, compile-time specialization. This behavior is desirable for GPU code, but we instead want runtime dispatch in host shader code. Since generics in general do not require a compile-time specialization implementation (unlike templates in C++), we could

[30]`https://rustc-dev-guide.rust-lang.org/backend/monomorph.html`

change the implementation for the host code to achieve the desired results. However, generics are just one option. Other Rust features might prove equally or more beneficial to co-opt, so more investigation is necessary to determine the best unified shader system design for Rust.

5.3.5 Adding Staged Metaprogramming to Another Language

Rather than modifying additional languages for shader programming specifically, an alternative is to add staged metaprogramming features instead. By adding staged metaprogramming to a language, programmers would benefit from the general-purpose usefulness of its features both in their shader systems and in other parts of their applications. Along with the technical challenges of adding staged metaprogramming, designing the user-facing programming model will raise interesting questions. Metaprogramming Terra code is straightforward because it is a relatively simple language. Creating a staged metaprogramming model for more complex languages like C++ could prove to be challenging. This task would be made easier by omitting some of the features we discussed in Section 4.2.1. While these features are sufficient for implementing unified shader programming, they may not all be necessary. Attempting to create a unified shader programming system using only a subset of staged metaprogramming could provide insights that help language developers decide which features are most important.

5.3.6 Future Opportunities Enabled by Unified Programming

Programming in a unified environment not only comes with inherent advantages as previously discussed, but it also opens opportunities to tackle other challenging issues in shader programming. While allowing host and GPU code to reference the same parameters avoids mismatches between these two halves of shader code, a remaining open question is how best to express and handle memory allocation, memory transfers to and from the GPU, and sharing of resources between different shaders and shader instances. Additionally, synchronization between host and device, as well as between different rendering passes, is another area with potential for improvements. By combining host and GPU shader code together into one environment, a shader system tool (whether compiler-based or otherwise) has a more cohesive view of the entire application. A potentially impactful future direction is to take advantage of this cross-device information to investigate optimization opportunities that would otherwise not be possible in a non-unified system.

Additionally, by having all pieces of rendering code in the same environment, we can begin to explore programming models beyond writing individual, separate shaders. Rather than authoring independent shaders for each rendering pass, programmers could instead write rendering code as one integrated unit. Then, the underlying programming system could split up work into different executable shader kernels automatically, similar to the ideas in Bulk-Synchronous GPU Programming (BSGP) [34]. Some analysis and kernel generation could happen purely at compile time (like in BSGP). However, greater optimization opportunities would be available by performing runtime analysis as well. Modern games and game engines are increasingly utilizing render graph systems (e.g., Frostbite's FrameGraph [59], Unreal Engine's Rendering Dependency Graph,[31] and Unity's RenderGraph API[32]). At runtime, these systems create a graph of the rendering operations needed for each frame, which (for example) allows them to optimize memory allocations by tracking dependencies between multiple rendering passes. Hypothetically, by examining the sequence of rendering steps—and the associated host and GPU code—for a given frame at runtime, a system could generate a set of shader kernels optimized for that frame. During development, the system could JIT-compile these kernels per frame and cache the results for subsequent frame that generate the same render graph. Then, when shipping a finished application, the system could bake out all of the required shaders to avoid the costs of runtime code generation.

Utilizing both compile-time and runtime information when deciding how to divide operations into individual executable kernels has the potential to greatly improve runtime performance of both host and GPU code. Which decisions are best suited for compile time versus run time is an interesting question that these future systems will need to investigate. A prerequisite for these systems is having all code together in one programming environment, including both host code and GPU code. Thus, unified shader programming sets the stage to explore these types of programming and optimization improvements in the future.

[31]`https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Rendering/RenderDependencyGraph/`

[32]`https://docs.unity3d.com/Packages/com.unity.render-pipelines.core@13.1/manual/render-graph-system.html`

Chapter 6

Related Work

6.1 Shader Programming in GPU-based Graphics APIs

While the earliest programmable shader environments for GPUs exposed assembly-level languages [44], most current graphics APIs that target modern GPUs, such as Direct3D, OpenGL, and Metal, provide only high-level language interfaces (for HLSL, GLSL, and an extended subset of C++, respectively). While Direct3D and Metal allow shader programs to be compiled offline, the resulting binary formats are not officially documented. Despite the detailed differences, all of these platforms provide a broadly similar C-like programming model.

The recently introduced Vulkan API [39] consumes a documented binary GPU shader code format, SPIR-V [38], instead of text. The SPIR-V instruction set is similar to the LLVM IR [42] and is intended to allow alternative front-end languages to be developed. However, neither SPIR-V nor LLVM provides assistance with the higher levels of a shader compilation system: parsing, type checking, etc. The primary benefit of LLVM is for lower-level optimizations and code generation, which are still performed by GPU drivers that consume SPIR-V. While the introduction of a documented intermediate language is an important architectural decision, it does not greatly simplify the task of developing engine-specific shader systems.

In contrast to earlier APIs, newer APIs include features aimed at providing direct support for some aspects of shader specialization. Vulkan's "specialization constants" [75] allow host code to modify the values of constants in GPU code at application runtime, and Metal has a similar

Sections 6.2, 6.3, and 6.4 have largely been taken from our paper "Staged Metaprogramming for Shader System Development" [68].

feature. These features could be used to implement basic-type specialization parameters without requiring programmers to statically enumerate all possible value options. However, they are insufficient for expressing and generating specializations that include different uniform parameters and GPU functions, which is the purpose of ShaderClass-type specialization parameters in Chapter 3 and of the ConfigurationOptions feature in Chapter 4.

These graphics APIs form the foundation on which modern graphics applications are built. Our work focuses on the next layer up the software stack: what engine-specific systems should be built on top of these APIs?

6.2 Extended Shader Programming Models

A wide range of alternative programming models have been developed on top of the baseline interfaces 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 [64] shows that a high-level shading language, inspired by the RenderMan Shading Language (RSL) [31], can be compiled with good efficiency for early programmable hardware. Abstract shade trees [50] build upon the idea of Shade Trees [13] to enable composition of real-time shader programs from separately developed pieces. Spark [27] extends the approach of RTSL to modern rasterization pipelines, improving support for modular software development.

Spire [33] demonstrates that a suitable high-level shader IR can allow complex rate-placement optimizations to be applied automatically. Our exploration of specialization decisions (Section 4.4) 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.

The Slang shading language [32] is a variant of HLSL that adds a number of features that are common in other modern programming languages. In GPU shader code, Slang enables users to express specialization options primarily through interface-constrained generics. Specialized shader variants are generated by providing concrete types for the generic parameters. Host code interfaces with GPU code using a runtime API, which provides reflection information and allows host code to generate specialized variants. Because host and GPU code exist in different environments, Slang sidesteps the compile-time vs. runtime dichotomy of specialization parameters.[33] While it does tackle some of the same problems as our work, Slang focuses on the GPU shading language specifically, whereas we explore unifying both the host and GPU portions of shader code into the same language. Furthermore, rather than providing a single solution for all engines, our work aims to enable developers to implement custom solutions that best fit the needs of their engines.

Effect systems expand the scope of shader code beyond programmable pipeline stages to include configuration of fixed-function state [51, 58] and even abstraction over multiple rendering passes [43]. The EAGL system [41] further expands the role of shaders to include offline processing of art assets to condition them for efficient rendering on particular platforms.

Several GPU shading languages for real-time graphics support encapsulation of shader code and parameters via object-orientation, including Cg interfaces [61], HLSL classes [54], Spark [27], and Slang [32]. The idea dates back to the RenderMan Shading Language (RSL) [31]. Furthermore, aspects of our ShaderClass design take inspiration from Kuck and Wesche [40]. Their work implements an object model for GLSL that is managed by corresponding proxy objects in C++. Whereas their system uses dynamic dispatch in GPU code (with optimizations to remove dispatch code when possible), ours in Chapter 3 guarantees static dispatch in generated GPU code. More fundamentally, our work differs from these previous works by extending shader objects to include both GPU and host code, with unified representations of types, functions, and parameters.

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.

[33]Hypothetically, if Slang were to introduce a unified shader programming model, we imagine that generics/interfaces would serve a significant role in the host-GPU specialization interface. Consequently, applying lessons learned from such a system to a C++-based environment would be difficult, since C++ does not support Slang-style generics/interfaces. By limiting ourselves to the features available in C++, we hope that the lessons from our work in Chapter 3 are more directly applicable in the field today.

6.3 Multi-Stage Programming and Syntax Extension

As described in Section 4.2.1, we take our definition of multi-stage programming from Taha [72]. Our Selos implementation was built using Terra [16], which adds multi-stage constructs and low-level programming support to the Lua language [35], along with user-defined syntax extensions.

The BraidGL language [66] uses the syntax of staging both for metaprogramming and for mapping 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 [65] is a systems programming language that supports limited syntax extension using a macro system in the tradition of Scheme [25, 70].

Some projects have attempted to add multi-stage programming features to existing systems programming languages like C/C++. 'C ("tick C") [21] adds a quasiquotation construct to C, with a focus on code generation at runtime (similar to MetaML [73] and MetaOCAML [9]). OpenC++ adds compile-time code generation and limited syntactic extension to C++ using a metaobject protocol [10]. The extension-oriented compiler Xoc [14] 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. We therefore could not use these languages to implement the staged metaprogramming work in Chapter 4 and, conversely, could not use such features when focusing on C++ in Chapter 3. Fortunately, ongoing C++ projects and proposals continue to explore the types of facilities needed for staged metaprogramming (Section 4.5).

6.4 Shader Metaprogramming

Several previous systems have applied metaprogramming techniques to shaders. For example, the PyFX system [43] uses Python code to compose multi-pass effects from GPU shader code written in Cg. Shader programs are authored as strings, and the system extracts parameter data to expose named parameters to Python code.

Sh [49] and Vertigo [19] expose shaders as an embedded DSL (eDSL) in C++ and Haskell, respectively. Special types are used to express GPU 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 GPU shader code. In Sh, the host and shader languages use distinct syntax for control flow constructs (GPU 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 GPU 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 in Chapter 4, 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 runtime 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 code and host/GPU shader code in Selos (as well as in our C++-based system in Chapter 3), are expressed in both the same language and same stage of execution; both are written in Terra (or C++), and they can share types and subroutines. In contrast, prior shader systems using staged metaprogramming separate host and GPU code into distinct stages with distinct languages, libraries, etc. Our approach is thus more similar to CUDA [57], where host and GPU compute code are deeply integrated using the same language and execution stage.

Partial evaluation [28] is a concept related to metaprogramming. Rodent [60] 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 at compile time.

6.5 Single-Language Shader Programming

While most real-time graphics applications use separate languages for host code and GPU code, some recent projects explore using the same language for both.

Rust GPU [20] is an early-stage project with the goal of compiling Rust code to SPIR-V (and possibly DXIL in the future). Similarly, the Circle compiler has recently added support to compile C++ code to SPIR-V (with DXIL support in progress) [7]. Both of these projects are working to satisfy a necessary condition for unified shader programming—the ability to author both host code and GPU code in the same language. Circle also allows both host and GPU code in the same file. We view Rust GPU and Circle's C++ shaders as important first steps toward unified shader programming in these languages. The task of compiling arbitrary Rust and C++ code to a GPU-compatible language is a massive undertaking that benefits any engine using these languages. However, neither of these systems include language design provisions to allow dynamic logic in host code to influence compile-time specialization and selection of GPU code, which is central to supporting unified shader specialization.

While these two projects do help to unify some aspects of shader programming, developers using them will still need to develop customized shader systems that handle the missing aspects (e.g., shader modularity, composition, and specialization). If, for example, developers choose to utilize Circle's metaprogramming features to implement these aspects, then our work on staged metaprogramming can provide guidance for their implementations. However, if these metaprogramming features are undesirable or unsupported in another compiler that facilitates host-language-to-GPU-language compilation, then our method of co-opting existing and implementing them with alternate semantics may prove more useful.

Along with GPU shader code support, Circle also adds many other language features to C++, including new general-purpose metaprogramming features.[34] Using these new features, it may be possible to build a unified shader programming system within the Circle language. The philosophy of our work in Chapter 3 differs from that of Circle's in two key ways. First, creating and maintaining a compiler to add arbitrary features to a language requires significantly more effort than our approach of using a source-to-source translator to co-opt existing features. The resources necessary to achieve the former are prohibitive for most real-time graphics teams. Secondarily, and more fundamentally, our goal is to create a system in which programmers write code that looks and feels like normal C++, both to themselves and to others who may be less familiar with the system. Therefore, we focus on introducing as few syntactic and semantic changes to C++ as possible while still achieving our other goals.

The aforementioned BraidGL [66] is a prototype language in which both host and GPU code are written in the base Braid language, with extensions for shader programming. Its compiler generates Javascript for host code, GLSL for GPU code, and WebGL [36] API function calls to set up and invoke rendering. Programmers write a top-level render loop stage (host code) that contains a nested vertex shader stage (GPU code), which in turn contains a nested fragment shader stage (GPU code). Code in nested stages can refer to variables in parent stages, and the BraidGL compiler generates the appropriate WebGL functions to communicate data between the stages. Thus, BraidGL fits our definition of a unified system for shader programming.

Our goals differs from those of BraidGL in that we aim to enable real-time graphics developers to create their own unified systems, customized to their specific needs. BraidGL presents a specific abstraction for shader programming and maps that abstraction to the underlying graphics API in a particular manner. However, engine developers may wish to provide a different abstraction to users and/or utilize alternate graphics API functionality for invoking rendering (e.g., direct vs. indirect dispatch) and communicating data between host and GPU code (e.g., uniform/constant buffers vs. individual uniform parameters). Therefore, our work explores techniques for implementing unified shader programming systems, rather than advocating for any

[34]However, Circle does not support full staged metaprogramming as defined in Section 4.2.1, since it lacks some of the necessary features, such as quasiquote.

one specific system.

Chapter 7

Conclusion

In this dissertation, we have proposed two methods for implementing unified shader systems for real-time graphics programming. By co-opting existing language features and implementing them with alternate semantics, developers can integrate unified shader programming into the languages used in their existing engines today. On the other hand, staged metaprogramming provides a powerful and general-purpose set of language features with which a unified shader system can be built, without the need to modify the underlying programming language. By exploring multiple methods for unifying the host and GPU aspects of shader programming, we provide graphics developers with options that trade off different advantages and meet different constraints.

Our work in Chapter 3 focuses on real-time graphics programming in C++, but we hope that the broader lessons can be applied to other programming languages as well. Bringing unified shader programming to other languages may involve co-opting different features depending on the specifics of the language, but we think that the principles that guided our design are largely transferrable to other, similar languages. This strategy enables programmers to incrementally integrate unified designs while still maintaining compatibility with existing code, which helps to encourage adoption of new ideas and features in existing large-scale systems.

While less useful for current engines today, staged metaprogramming has the significant advantage of being a more flexible technique without requiring language or compiler modifications. 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 [53] and Vulkan [39]) 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. Our experience leads us to believe that staged metaprogramming will be an important feature of such future heterogeneous systems.

Beyond real-time graphics applications, GPU-based graphics provides a complex and well-explored domain in which to investigate the broader concept of unified, 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 domains to efficiently utilize a wide range of heterogeneous processing resources. The lessons learned while studying graphics programming models can help inform such future designs. While co-opting existing language features provides the ability to partially modify a language for other applications, we believe that staged metaprogramming is a more powerful and more generally useful technique applicable to a wide variety of domains. We hope that this work helps to inform future language designs by advocating for the inclusion of more powerful metaprogramming features, both in new languages and in future versions of the popular languages today.

Appendix A

The Implementation of Selos's Specialization Framework

A.1 Introduction

In this appendix, we briefly present parts of the specialization implementation in our Selos shader system. Specifically, we show the deferred pass shader (Section A.2) and the two `ConfigurationOptions` used to express specialization options and generate shader variants: `TiledLightListEnv` (Section A.3) and `TiledDeferredMaterialType` (Section A.4). The full source code for Selos is available at `https://github.com/kseitz/selos`.

A.2 Deferred Pass Shader

Listing A.1 shows the shader used to compute deferred shading in our tiled deferred renderer. This shader would be written by a graphics developer that is implementing the deferred renderer. The two `ConfigurationOptions` are declared on lines 5–8, a few texture images (including the GBuffers and the output image) are declared on lines 12–14, and a uniform buffer for the `Camera` data is declared on lines 16–20.

In the compute shader GPU code (lines 22–46), the shader first determines the tile ID and pixel coordinates for the given invocation (lines 23–33) in order to fetch the shading data (including material type ID and BRDF parameters) from the GBuffers (lines 35–36).

Then, lighting is computed by calling the `illuminate` function pro-

```
local MaterialSystem = require("MaterialSystem")
local LightSystem = require("LightSystem")

local pipeline DeferredMaterialShader {
  ConfigurationOptions {
    MaterialType = MaterialSystem.TiledDeferredMaterialType.new()
    LightEnv = LightSystem.TiledLightListEnv.new()
  }

  numthreads(TILE_SIZE, TILE_SIZE, 1)

  textureImage(Gfx.TextureFormat.RGBA32F, Gfx.MapMode.ReadOnly) gbuffers : image2D[5]
  textureImage(Gfx.TextureFormat.RGBA16U, Gfx.MapMode.ReadOnly) tileList : uimage2D
  textureImage(Gfx.TextureFormat.RGBA16F, Gfx.MapMode.WriteOnly) outImage : image2D

  uniform Camera {
    cameraPos       : vec3
    numTilesX       : int
    cameraVP        : mat4
  }

  compute code
    var tileIDxyzw : uvec4 =
      imageLoad(tileList, make_ivec2(WorkGroupID.x / 4, [getConfigurationID()]))
    var tileID = tileIDxyzw(WorkGroupID.x % 4)

    var tileCoords = make_ivec2(tileID % numTilesX, tileID / numTilesX)

    var locID : ivec2
    locID.x = LocalInvocationID.x
    locID.y = LocalInvocationID.y

    var pixelCoords = (tileCoords * TILE_SIZE) + locID

    var sd : MaterialType:ShadingDataType()
    [GBufferShadingData.prepare(gbuffers, cameraPos, pixelCoords, sd)]

    var clipZ : float = (cameraVP * make_vec4(sd.wPos, 1f)).z

    var colorAcc = [LightEnv:illuminate(MaterialType)](sd, clipZ, tileID)

    colorAcc = colorAcc + sd.emissive

    var color = make_vec4(colorAcc, 1)
    imageStore(outImage, pixelCoords, color)
  end
}
```

Listing A.1: The Selos deferred shader used in our design space exploration framework. A graphics programmer might author such a shader.

vided by the `TiledLightListEnv` (line 40). Notice the brackets around `[LightEnv:illuminate(...)]`. They indicate a Lua-Terra *escape* (i.e., staged metaprogramming's *unquote*). `LightEnv:illuminate()` is a Lua function that returns a Terra function, which is then spliced into the shader on line 40. This staged metaprogramming functionality allows us to generate different versions of the deferred shader, specialized to a particular lighting environment (Section A.3). Similarly, this code passes the `MaterialType` to the `illuminate` function, so that the generated code can be specialized based on a particular set of material BRDFs (Section A.4).

A.2.1 Light Types

Separate from this shader, a graphics developer or technical artist would author the code to implement various light and material types. Listing A.2 shows an example implementation of a light type.

This code declares a struct for the light's data (lines 1–4), which can be used in both host and GPU code. It includes a function for determining if the object being rendered is in shadow (lines 6–15). Lines 17–43 express the illumination function for this light type. Because we wish to specialize based on material types as well, the function declared on line 17 is a Lua function that returns a Terra function. The Terra function, specialized for a given set of material types, will be spliced into the deferred shader during shader compilation.

The code to compute material shading will be spliced into this Terra function on line 39, in place of `[MaterialType:eval(...)]`. The `TiledLightListEnv` implementation will call the Lua `illuminate()` functions for each active light type in order to generate a specialized deferred shader variant, as we will show in Section A.3. The code registers the light type with the system on line 45, setting the last parameter to `true` to indicate that the light type is cullable. The `registerLight()` function is provided by the shader system (and, thus, written by an engine developer), and it adds the light type to the list of all light types so that the rest of the system can operate on it.

```
local struct ShadowedPointLight {
  intensity  : vec3
  position   : vec3
}

terra ShadowedPointLight.isInShadow(light : ShadowedPointLight,
    shadowMap : samplerCube, wPos : vec3, normal : vec3,
    farPlane : float, idx : int)
  var fragToLight = wPos - light.position
  var closestDepth = textureLod(shadowMap, fragToLight, 0).r
  closestDepth = closestDepth * farPlane
  var currentDepth = length(fragToLight)
  var bias = 0.05f
  return currentDepth - bias > closestDepth
end

function ShadowedPointLight:illuminate(MaterialType)
  local terra illuminate(l : ShadowedPointLight,
                         sd : MaterialType:ShadingDataType())
    var dir = l.position - sd.wPos
    var dist = length(dir)

    if dist >= POINT_LIGHT_RADIUS then
      return make_vec3(0,0,0)
    end

    var distSquared = dot(dir, dir)

    if distSquared > 0.00001f then
      dir = normalize(dir)
    else
      dir = make_vec3(0,0,0)
    end

    var falloff = getDistanceFalloff(distSquared)
    var intensity = l.intensity * falloff
    var ls = make_LightSample(intensity, dir, sd.eyeDir, sd.normal)

    return [MaterialType:eval(LightSample)](sd, ls)
  end

  return illuminate
end

registerLight(ShadowedPointLight, MAX_SHADOWED_POINT_LIGHTS,  true)
```

Listing A.2: An example light type implementation in Selos. A graphics programmer or technical artist might author such an implementation.

A.2.2 Material Types

Similar to light types, a graphics programmer or technical artist would also create various material types to express different BRDF options. Listing A.3 shows an example.

This code computes the diffuse (lines 10–12) and specular (lines 15–21) components for our `StandardMaterial` type. Each material type implementation must contain an `eval` function (lines 2–25) that the `TiledDeferredMaterialType` will splice into the deferred shader for the active set of material types (Section A.4). Notice that the code for the `eval` function is stored directly in the `StandardMaterial` data structure, which is then passed to the `registerMaterial()` function (which is written by an engine developer) on line 27. The ability to treat code as a first-class construct in the programming language is a key feature of staged metaprogramming.

```
 1  local StandardMaterial = {
 2    eval = terra(sd : ShadingData, ls : LightSample)
 3      if ls.NdotL <= 0 then
 4        return make_vec3(0,0,0)
 5      end
 6
 7      var NdotV = saturate(dot(sd.normal, sd.eyeDir))
 8
 9      -- diffuse component
10      var diffuseBrdf = evalDiffuseFrostbiteBrdf(
11        sd.linearRoughness, NdotV, sd.diffuse, ls);
12      var diffuse = ls.intensity * diffuseBrdf * ls.NdotL;
13
14      -- specular component
15      var roughness = sd.linearRoughness * sd.linearRoughness;
16      var D = evalGGX(roughness, ls.NdotH);
17      var G = evalSmithGGX(ls.NdotL, NdotV, roughness);
18      var F = fresnelSchlick(sd.specular, make_vec3(1,1,1),
19        max(0, ls.LdotH));
20      var specularBrdf = D * G * F * kINV_PI;
21      var specular = ls.intensity * specularBrdf * ls.NdotL;
22
23      var color = diffuse + specular
24      return color
25    end
26  }
27  registerMaterial(StandardMaterial)
```

Listing A.3: An example material type implementation in Selos. A graphics programmer or technical artist might author such an implementation.

A.3 Specialization of Light Types

We now show the core functionality for specializing based on light types in Listing A.4. This functionality would be implemented by an expert graphics programmer. Note that we simplified this listing and omit some of the other parts of the implementation for brevity and clarity, in order to focus on the most relevant portions.

```
function TiledLightListEnv:illuminate(MaterialType)
  local sd        = symbol(MaterialType:ShadingDataType(), "sd")
  local clipZ     = symbol(float, "clipZ")
  local tileID    = symbol(int, "tileID")
  local colorAcc  = symbol(vec3, "colorAcc")

  local quoteList = terralib.newlist()

  local cullableLights = getCullableLights()
  for k, v in ipairs(cullableLights) do
    if self.statusList[k] then
      quoteList:insert(quote
        for idx=0, self.lightLists[tileID].["num" .. v.name] do
          var i = self.lightLists[tileID].["idx" .. v.name][idx]
          var l = self.uniformBlock.["all" .. v.name][i]

          var isInShadow = [generateShadowCode(v, l, i)]

          if not isInShadow then
            colorAcc =
              colorAcc + [v:illuminate(MaterialType)](l, sd)
          end
        end
      end)
    end
  end

  local terra illuminate([sd], [clipZ], [tileID])
    var [colorAcc] = make_vec3(0,0,0)
    [quoteList]
    return colorAcc
  end

  return illuminate
end
```

Listing A.4: The implementation of TiledLightListEnv in Selos. This ConfigurationOption is responsible for specializing the deferred shader based on light types. An expert graphics programmer would author such an implementation.

Lines 2–5 create *symbols* that allow these specific variables to violate lexical scoping rules. This functionality allows these variables to be used in multiple different quotes in the rest of the function (whereas variables declared within a quote cannot permeate outside of that quote).

The function then iterates over the light types that are cullable (lines 9–26). If the variant currently being generated should include code to evaluate the given light type (line 11), then a Terra quote is created (lines 12–24) to loop over the lights of this type (lines 13–23). This loop fetches the runtime data for a light (lines 14–15) and evaluates shadow mapping if applicable (line 17). Then, the light type's `illuminate` function is called to splice in the code to evaluate the light (line 21).

After iterating over all cullable light types and building up a list of Terra quotes for the active ones, a Terra function is created (lines 28–32), and these quotes are spliced into the function using an escape (line 30). Finally, the generated function is returned (line 34), where it will then be spliced into the deferred shader, as shown on Listing A.1 line 40.

Staged metaprogramming enables Listing A.4 to operate on code using a fully featured language. It iterates over the various light types, builds up a data structure containing code (using regular language if statements to decide which code to include), and then unquotes the code into a function that will be executed by the deferred shader's GPU code.

A.4 Specialization of Material Types

The implementation of the `TiledDeferredMaterialType` would also be written by an expert graphics programmer, and it follows a similar pattern as the implementation of light type specialization presented above. Listing A.5 shows relevant portions of code (again simplified for clarity and brevity).

This function first determines how many material types should be included in the variant it is currently generating (lines 3–11). Then, it creates a Terra function that will be spliced into the shader to perform material shading (lines 13–41). If only one material type is active, its `eval` function will be spliced directly into the code (lines 17–22). Otherwise, the code iterates over all material types (lines 26–37). If a given type should be included in the variant (line 28), then its `eval` function is again spliced in (line 31). However, because the material types are mutually

```
local function TiledDeferredMaterialType:eval(ShadingData,
    LightSample, statusList, componentMask)
  local numEnabled = 0
  local lastEnabled = -1

  for i, v in ipairs(statusList) do
    if v ~= FEATURE_STATUS.DISABLED then
      numEnabled = numEnabled + 1
      lastEnabled = i
    end
  end

  local terra evalMaterial(sd : ShadingData, ls : LightSample)
    [(function()

      -- Statically specialize if only one material is enabled
      if numEnabled == 1 then
        local matl = MaterialTypes.asList[lastEnabled]
        return quote
          return [matl.eval(ShadingData, LightSample)](sd, ls)
        end
      end

      local ifStatements = quote return make_vec3(0,0,0) end

      for i = #MaterialTypes.asList, 1, -1 do
        local matl = MaterialTypes.asList[i]
        if statusList[i] == FEATURE_STATUS.ENABLED then
          ifStatements = quote
            if ([matl.bitflag] and sd.materialMask) > 0 then
              return [matl.eval(ShadingData, LightSample)](sd, ls)
            else
              [ifStatements]
            end
          end
        end
      end

      return ifStatements
    end)()]
  end

  return evalMaterial
end
```

Listing A.5: The implementation of TiledDeferredMaterialType in Selos. This ConfigurationOption is responsible for specializing the deferred shader based on material types. An expert graphics programmer would author such an implementation.

exclusive (per pixel), the call to a material type's `eval` function must be nested under a runtime if statement in the GPU shader code (line 30). For example, if the system is generating a variant containing only the `StandardMaterial` and `ClothMaterial`, lines 26–37 would produce the following Terra code:

```
if (1 and sd.materialMask > 0) then
  return evalStandardMaterial(sd, ls)
else
  if (4 and sd.materialMask > 0) then
    return evalClothMaterial(sd,ls)
  else
    return make_vec3(0,0,0)
  end
end
```

This code is stored in the `ifStatements` variable (lines 24, 29, and 33), which is then spliced into the generated `evalMaterial` Terra function (line 39). This Terra function is returned and unquoted into the shader (e.g., as shown on Listing A.2 line 39).

The key features of staged metaprogramming drive our implementation of this functionality, as evidenced by our extensive use of quotes to programmatically create Terra code, which are then stored in data structures, and finally specialized and unquoted into the shader code to generate specialized variants.

A.5 Generated HLSL and GLSL code

After applying a specialization decision, the final HLSL and GLSL code generated by Selos is a fairly straightforward translation from Terra to these shading languages. Selos's backends produce human-readable code to help facilitate debugging. Instructions for generating all of the variants used by our deferred renderer are available at `https://github.com/kseitz/selos`.

REFERENCES

[1] Amazon Web Services, Inc. Amazon Lumberyard. https://aws.amazon.com/lumberyard/, 2021.

[2] Johan Andersson. DirectX 11 Rendering in Battlefield 3. In *Game Developers Conference 2011*, GDC 2011, February/March 2011. https://www.ea.com/frostbite/news/directx-11-rendering-in-battlefield-3

[3] Apple Inc. Metal. https://developer.apple.com/documentation/metal, 2014.

[4] Apple Inc. *Metal Shading Language Specification Version 2.3*, 2021. https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf

[5] Alan Bawden and Jonathan Rees. Syntactic Closures. In *Proceedings of the 1988 ACM Conference on LISP and Functional Programming*, LFP '88, pages 86–95, July 1988. https://dx.doi.org/10.1145/62678.62687

[6] Sean Baxter. Circle. https://github.com/seanbaxter/circle, 2019.

[7] Sean Baxter. Circle C++ Shaders. https://github.com/seanbaxter/shaders/blob/master/README.md, 2021.

[8] Nir Benty, Kai-Hwa Yao, Petrik Clarberg, Lucy Chen, Simon Kallweit, T. Foley, Matthew Oakes, Conor Lavelle, and Chris Wyman. The Falcor Rendering Framework. https://github.com/NVIDIAGameWorks/Falcor, August 2020.

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

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

[11] Matus Chochlik, Axel Naumann, and David Sankel. Static Reflection. C++ Standards Committee Papers, March 2018. http://wg21.link/p0194

[12] Petrik Clarberg and Jacob Munkberg. Deep Shading Buffers on Commodity GPUs. *ACM Transactions on Graphics*, 33(6):227:1–227:12, November 2014. https://dx.doi.org/10.1145/2661229.2661245

[13] Robert L. Cook. Shade Trees. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, pages 223–231, July 1984.

[14] Russ Cox, Tom Bergan, Austin T. Clements, Frans Kaashoek, and Eddie Kohler. Xoc, an Extension-Oriented Compiler for Systems Programming. In *Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 244–254, March 2008. `https://dx.doi.org/10.1145/1346281.1346312`

[15] Lewis Crawford and Michael O'Boyle. Specialization Opportunities in Graphical Workloads. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques*, PACT 2019, pages 272–283, September 2019. `https://dx.doi.org/10.1109/PACT.2019.00029`

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

[17] Ramy El Garawany. Deferred Lighting in Uncharted 4. In *ACM SIGGRAPH 2016 Courses*, SIGGRAPH 2016, July 2016. Part of the course: Advances in Real-Time Rendering, Part I, `https://dx.doi.org/10.1145/2897826.2940291`

[18] Electronic Arts Inc. Frostbite Engine. `https://www.ea.com/frostbite`, 2021.

[19] Conal Elliott. Programming Graphics Processors Functionally. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 45–56, September 2004. `https://dx.doi.org/10.1145/1017472.1017482`

[20] Embark Studios. Rust GPU. `https://github.com/EmbarkStudios/rust-gpu`, 2021.

[21] Dawson R. Engler, Wilson C. Hsieh, and M. Frans Kaashoek. '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 1996, pages 131–144, January 1996. `https://dx.doi.org/10.1145/237721.237765`

[22] Epic Games. Infiltrator Demo. `https://www.unrealengine.com/marketplace/en-US/product/infiltrator-demo`, August 2015.

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

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

[25] Matthew Flatt. Composable and Compilable Macros: You Want It When? In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP 2002, pages 72–83, October 2002. `https://dx.doi.org/10.1145/581478.581486`

[26] T. Foley. A Modern Programming Language for Real-Time Graphics: What is Needed? In *ACM SIGGRAPH 2016 Courses*, SIGGRAPH 2016, July 2016. Part of the course: Open Problems in Real-Time Rendering, `https://dx.doi.org/10.1145/2897826.2940293`

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

[28] Yoshihiko Futamura. Partial Computation of Programs. In *RIMS Symposium on Software Science and Engineering*, pages 1–35, 1983. `https://dx.doi.org/10.1007/3-540-11980-9_13`

[29] Kate Gregory and Ade Miller. *C++ AMP: Accelerated Massive Parallelism with Microsoft® Visual C++®*. Microsoft Press, October 2012.

[30] Romain Guy and Mathias Agopian. Filament. `https://github.com/google/filament`, 2019.

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

[32] Yong He, Kayvon Fatahalian, and T. Foley. Slang: Language Mechanisms for Extensible Real-time Shading Systems. *ACM Transactions on Graphics*, 37(4):141:1–141:13, July 2018. `https://dx.doi.org/10.1145/3197517.3201380`

[33] Yong He, T. Foley, and Kayvon Fatahalian. A System for Rapid Exploration of Shader Optimization Choices. *ACM Transactions on Graphics*, 35(4):112:1–112:12, July 2016. `https://dx.doi.org/10.1145/2897824.2925923`

[34] Qiming Hou, Kun Zhou, and Baining Guo. BSGP: Bulk-Synchronous GPU Programming. *ACM Transactions on Graphics*, 27(3):19:1–19:13, August 2008. `https://dx.doi.org/10.1145/1360612.1360618`

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

[36] Dean Jackson and Jeff Gilbert. *WebGL Specification.* The Khronos WebGL Working Group, 2021. `https://www.khronos.org/registry/webgl/specs/latest/1.0/`

[37] John Kessenich, Dave Baldwin, and Randi Rost. *The OpenGL$^{©}$ Shading Language (Version 4.50)*. The Khronos Group Inc., 2017. https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf

[38] John Kessenich and Boaz Ouriel. *SPIR-V Specification (Version 1.00)*. The Khronos Group Inc., 2018. https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf

[39] Khronos Group. *Vulkan 1.0.12 - A Specification*. The Khronos Group Inc., 2016. https://www.khronos.org/registry/vulkan/specs/1.0/pdf/vkspec.pdf

[40] Roland Kuck and Gerold Wesche. A Framework for Object-Oriented Shader Design. In *Advances in Visual Computing*, ISVC 2009, pages 1019—1030, 2009. https://dx.doi.org/10.1007/978-3-642-10331-5_95

[41] Paul Lalonde and Eric Schenk. Shader-Driven Compilation of Rendering Assets. *ACM Transactions on Graphics*, 21(3):713–720, July 2002. https://dx.doi.org/10.1145/566654.566641

[42] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, CGO '04, pages 75–86, March 2004. https://dx.doi.org/10.1109/CGO.2004.1281665

[43] Calle Lejdfors and Lennart Ohlsson. PyFX – An Active Effect Framework. In *Proceedings of The Annual SIGRAD Conference Special Theme — Environmental Visualization*, SIGRAD 2004, pages 17–24, November 2004. http://www.ep.liu.se/ecp/article.asp?issue=013&article=006

[44] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A User-Programmable Vertex Engine. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 2001, pages 149–158, August 2001. https://dx.doi.org/10.1145/383259.383274

[45] Juan Linietsky, Ariel Manzur, and contributors. Godot Engine. https://godotengine.org/, 2021.

[46] Ronan Marchalot. Cluster Forward Rendering and Anti-Aliasing in 'Detroit: Become Human'. In *Game Developers Conference 2018*, GDC 2018, March 2018. https://www.gdcvault.com/play/1025420/Cluster-Forward-Rendering-and-Anti

[47] Stephen McAuley. The Challenges of Rendering an Open World in Far Cry 5. In *ACM SIGGRAPH 2018 Courses*, SIGGRAPH 2018, August 2018. Part of the course: Advances in Real-time Rendering in Games, Part I, https://dx.doi.org/10.1145/3214834.3264541

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

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

[50] Morgan McGuire, George Stathis, Hanspeter Pfister, and Shriram Krishnamurthi. Abstract Shade Trees. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D 2006, pages 79–86, March 2006. https://dx.doi.org/10.1145/1111411.1111425

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

[52] Microsoft. Shader Model 5.1. https://msdn.microsoft.com/en-us/library/windows/desktop/dn933277(v=vs.85).aspx, 2014.

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

[54] Microsoft. Interfaces and Classes. https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/overviews-direct3d-11-hlsl-dynamic-linking-class, 2018.

[55] Microsoft. DirectX Shader Compiler. https://github.com/Microsoft/DirectXShaderCompiler, 2019.

[56] Microsoft. Direct3D. https://docs.microsoft.com/en-us/windows/win32/direct3d, 2020.

[57] NVIDIA Corporation. NVIDIA CUDA Compute Unified Device Architecture Programming Guide, January 2007. http://developer.nvidia.com/cuda.

[58] NVIDIA Corporation. Introduction to CgFX. http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter01.html, 2010.

[59] Yuriy O'Donnell. FrameGraph: Extensible Rendering Architecture in Frostbite. In *Game Developers Conference 2017*, GDC 2017, February/March 2017. https://www.gdcvault.com/play/1024612/FrameGraph-Extensible-Rendering-Architecture-in

[60] Arsène Pérard-Gayot, Richard Membarth, Roland Leißa, Sebastian Hack, and Philipp Slusallek. Rodent: Generating Renderers Without Writing a Generator. *ACM Transactions on Graphics*, 38(4):40:1–40:12, July 2019. https://dx.doi.org/10.1145/3306346.3322955

[61] Matt Pharr. An Introduction to Shader Interfaces. In *GPU Gems* (edited by Randima Fernando), chapter 32, pages 537–550. Addison Wesley, March 2004.

[62] Aras Pranckevičius. HLSL to GLSL Shader Language Translator. `https://github.com/aras-p/hlsl2glslfork`, 2013.

[63] Aras Pranckevičius, October 2016. Personal Communication.

[64] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A Real-Time Procedural Shading System for Programmable Graphics Hardware. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH 2001, pages 159–170, August 2001. `https://dx.doi.org/10.1145/383259.383275`

[65] Rust Project Developers. The Rust Programming Language. `https://doc.rust-lang.org/book/`, 2015.

[66] Adrian Sampson, Kathryn S. McKinley, and Todd Mytkowicz. Static Stages for Heterogeneous Programming. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):71:1–71:27, October 2017. `https://dx.doi.org/10.1145/3133895`

[67] Mark Segal, Kurt Akeley, Chris Frazier, Jon Leech, and Pat Brown. *The OpenGL© Graphics System: A Specification (Version 4.5 (Core Profile) - June 29, 2017).* The Khronos Group Inc., 2017. `https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf`

[68] Kerry A. Seitz, Jr., T. Foley, Serban D. Porumbescu, and John D. Owens. Staged Metaprogramming for Shader System Development. *ACM Transactions on Graphics*, 38(6):202:1–202:15, November 2019. `https://dx.doi.org/10.1145/3355089.3356554`

[69] Tiago Sousa and Jean Geffroy. The Devil Is in the Details: idTech 666. In *ACM SIGGRAPH 2016 Courses*, SIGGRAPH 2016, July 2016. Part of the course: Advances in Real-Time Rendering, Part II, `https://dx.doi.org/10.1145/2897826.2940292`

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

[71] Herb Sutter. Metaclasses: Generative C++. C++ Standards Committee Papers, February 2018. `http://wg21.link/P0707`

[72] Walid Taha. *Multi-Stage Programming: Its Theory and Applications*. Ph.D. thesis, Oregon Graduate Institute of Science and Technology, Beaverton, OR, USA, November 1999.

[73] Walid Taha and Tim Sheard. MetaML and Multi-Stage Programming With Explicit Annotations. *Theoretical Computer Science*, 248(1–2):211–242, October 2000. `https://dx.doi.org/10.1016/S0304-3975(00)00053-0`

[74] Natalya Tatarchuk and Chris Tchou. Destiny Shader Pipeline. In *Game Developers Conference 2017*, GDC 2017, February/March 2017. http://advances.realtimerendering.com/destiny/gdc_2017/

[75] The Khronos Vulkan Working Group. *Vulkan 1.1.178 - A Specification (with KHR extensions)*, chapter 10.8. Specialization Constants. The Khronos Group Inc., 2021. https://www.khronos.org/registry/vulkan/specs/1.1-khr-extensions/html/chap10.html#pipelines-specialization-constants

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

[77] Daveed Vandevoorde and Louis Dionne. Exploring the Design Space of Metaprogramming and Reflection. C++ Standards Committee Papers, March 2017. http://wg21.link/P0633