

Unified Shader Programming in C++

KERRY A. SEITZ, JR., University of California, Davis, USA

THERESA FOLEY, NVIDIA, USA

SERBAN D. PORUMBESCU, University of California, Davis, USA

JOHN D. OWENS, University of California, Davis, USA

In real-time graphics, the strict separation of programming languages and environments for host (CPU) code and GPU code results in code duplication, subtle compatibility bugs, and additional development and maintenance costs. In contrast, popular general-purpose GPU (GPGPU) programming models like CUDA and C++ AMP avoid many of these issues by presenting unified programming environments where both host and GPU code are written in the same language, can be in the same file, and share lexical scopes. To bring the benefits of unified programming to real-time graphics, this paper examines graphics-specific challenges that complicate the development of such a unified model and explores how to overcome them in a widely used programming language.

We observe that GPU code specialization, a key optimization in real-time graphics, requires coordination between parameters that are compile-time-constant in GPU code but are assigned values at runtime in host code based on dynamic data. Current methods used to implement specialization do not translate to a unified environment where host and GPU code share declarations of these parameters. Furthermore, this compile-time vs. runtime coordination is not innately expressible in the popular languages used in this domain.

In this paper, we create a unified environment for real-time graphics programming in C++ by co-opting existing features of the language and implementing them with alternate semantics to express the services required. Specifically, we co-opt C++ attributes and virtual functions, which enables us to provide first-class support for specialization in our unified system. By co-opting existing features, we enable programmers to use familiar C++ programming techniques to write host and GPU code together, while still achieving efficient generated C++ and HLSL code via our source-to-source translator.

CCS Concepts: • **Computing methodologies** → **Computer graphics**; • **Software and its engineering** → *General programming languages*; *Compilers*.

Additional Key Words and Phrases: Shaders, Shading Languages, Real-Time Rendering, Heterogeneous Programming, Unified Programming

1 INTRODUCTION

Real-time graphics programming is made more complicated by the use of distinct languages and programming environments for host (CPU) code and GPU code. GPU code performs highly-parallel rendering calculations and is typically authored in a special-purpose shading language (e.g., HLSL [Microsoft 2014], GLSL [Kessenich et al. 2017], or Metal Shading Language [Apple Inc. 2021]), while host code, which coordinates and invokes rendering work that uses this GPU code, is written in a general-purpose systems language (e.g.,

C++). When using a shading language and its corresponding graphics API (e.g., Direct3D [Microsoft 2020], Vulkan/OpenGL [Khronos Group 2016; Segal et al. 2017], or Metal [Apple Inc. 2014]), 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 [NVIDIA Corporation 2007], 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 [Gregory and Miller 2012] 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. We are interested in exploring how to create a unified programming environment for real-time graphics because of the code reuse, compatibility, and ease-of-use benefits that it provides.

While CUDA and C++ AMP provide powerful unified programming models for General-Purpose GPU (GPGPU) computing, developing a unified system for real-time graphics is complicated by the need to *specialize* GPU code based on dynamic data coming from host code. *Specialization* is a pervasive and critically important optimization in real-time graphics—it can have a significant impact on runtime performance [Crawford and O’Boyle 2019; He et al. 2018; Seitz et al. 2019], major game engines create mechanisms specifically to support it [Epic Games, Inc. 2019; Unity Technologies 2019], and game developers go to great lengths to enable it even in scenarios where it may not initially seem feasible [El Garawany 2016]. For this optimization, GPU code is compiled multiple times with different options to generate multiple compiled *variants* of the original code, each specialized to a particular combination of features and configurations. Then, at game runtime, host code selects which variants to invoke based on dynamic data such as information about the scene, the underlying hardware platform, and user settings. The data necessary to decide which GPU variants to invoke is not available until runtime, but developers need to generate the specialized

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

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.

Because of its importance, a unified environment for real-time graphics programming must provide support for specialization, but unfortunately, the popular programming languages used in graphics cannot express parameters that are part compile-time and part runtime. 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 paper, 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 tool¹ 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 4 and 5, respectively. In Section 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.

2 BACKGROUND

In this section, we define the services that a unified system needs to provide through the lens of modern real-time graphics programming. 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 paper.

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.² Since GPUs are coprocessors, invocation of GPU code must be initiated from host code running on a CPU. A *shader program* is a host-invokable unit of GPU code consisting of parameters, functions, and one or more entry points (further described in Section 2.1). A shader program often provides compile-time configuration options called *specialization parameters*, and compiling the shader program with different values for those options generates multiple *shader variants* of the original code. A shader program’s corresponding host shader code is responsible for selecting which

shader variant to invoke based on dynamic information available at application runtime, as well as coordinating data transfer between host memory 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.

In the next two sections, we discuss these two halves of 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). 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.

2.1 GPU Shader Code

Listing 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.³ When invoked from host code, multiple *instances* of this program are executed in parallel, with each instance operating mostly independently.⁴

The shader program in Listing 1 has an *entry point* function (`MainCS()` on line 22), 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 21).⁵ This example also includes other GPU functions (e.g., the `doFiltering()` functions) that can be called from GPU code. In HLSL, an entry 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 22) 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 6–8). Unlike varying parameters, the value of a uniform parameter is the same for all instances in a given invocation. For example,

³However, other files can be `#included` to allow for reuse of HLSL code.

⁴The HLSL programming model provides some inter-instance synchronization and communication capabilities, but we will not discuss them here.

⁵For information about the HLSL compute shader programming model, as well as other types of shaders, we refer interested readers to the HLSL documentation [Microsoft 2014].

¹Upon publication, we plan to release our implementation as open-source software.

²Many 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.

```

1  #define LOW 0
2  #define MEDIUM 1
3  #define HIGH 2
4
5
6  Texture2D ColorTexture;
7  SamplerState ColorSampler;
8  RWTexture2D<float4> Output;
9
10
11 #if QUALITY == LOW
12     float4 doFiltering(float2 pos) { /* Low Quality Method */ }
13 #elif QUALITY == MEDIUM
14     float4 doFiltering(float2 pos) { /* Medium Quality Method */ }
15 #elif QUALITY == HIGH
16     int ExtraParameter;
17     float4 doFiltering(float2 pos) { /* High Quality Method */ }
18 #endif
19
20
21 [numthreads(8, 8, 1)]
22 void MainCS(uint2 DispatchThreadID : SV_DispatchThreadID) {
23     float2 pixelPos = /* ... */;
24     float4 outColor = ColorTexture.Sample(ColorSampler, pixelPos);
25
26     for (int i = 0; i < ITERATION_COUNT; ++i) {
27         outColor *= doFiltering(pixelPos);
28     }
29
30     Output[DispatchThreadID] = outColor;
31 }

```

Listing 1. An example GPU shader program written in HLSL.

```

1  enum class QualityEnumType : int {
2      Low,
3      Medium,
4      High
5  };
6
7  class FilterShaderCS : public FGlobalShader {
8  public:
9      DECLARE_SHADER_TYPE(FilterShaderCS, Global);
10
11      BEGIN_SHADER_PARAMETER_STRUCT(FParameters, )
12          SHADER_PARAMETER_RDG_TEXTURE(Texture2D, ColorTexture)
13          SHADER_PARAMETER_SAMPLER(SamplerState, ColorSampler)
14          SHADER_PARAMETER_RDG_TEXTURE_UAV(RWTexture2D<float4>,
15              Output)
16          SHADER_PARAMETER(int, ExtraParameter)
17      END_SHADER_PARAMETER_STRUCT()
18
19      class QualityDimension :
20          SHADER_PERMUTATION_ENUM_CLASS("QUALITY", QualityEnumType);
21
22      class IterationCountDimension :
23          SHADER_PERMUTATION_SPARSE_INT("ITERATION_COUNT",
24              2, 4, 8, 16);
25
26      using FPermutationDomain = TShaderPermutationDomain<
27          QualityDimension, IterationCountDimension>;
28  };
29
30  IMPLEMENT_GLOBAL_SHADER(FilterShaderCS,
31      "/path/to/HLSL/file.usf", "MainCS", SF_Compute);

```

Listing 2. Host shader code corresponding to the GPU shader program in Listing 1. This code is written in C++ and uses features provided by UE4.

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.

The example in Listing 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 26). Specialization parameters can also be used to define additional GPU functions and uniform parameters (e.g., the ExtraParameter uniform on line 16 is only defined when QUALITY == HIGH). Note that the possible values for QUALITY are also defined using the preprocessor (lines 1–3).

2.2 Host Shader Code (in Unreal Engine 4)

Listing 2 shows the UE4 host shader code corresponding to the example shader program in Listing 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 30–31).

The host code class includes declarations of the shader's uniform parameters using UE4's SHADER_PARAMETER macros (lines 11–17).

These macros define a struct, which implements a strongly typed interface for passing parameter 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 19–24). 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 26–27). 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);

```

2.3 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 1 lines 6–8 and Listing 2 lines 11–17, 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 lines 1–5 are redeclared in Listing 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.⁶

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, we explore design alternatives in Section 4.4.3 that 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.

3 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

⁶We 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/2a2cebfe6a5a7164dbe2401ba2d5dd1901b649e>

(Note: access to this page requires permission to access the UE4 source code on GitHub)

this task—both for C++ and for Rust—which we discuss in Section 7. 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.

4 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 4.1), 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.

4.1 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 [Epic Games, Inc. 2019], Godot Engine [Linietzky et al. 2021], Frostbite [Electronic Arts Inc. 2021], and Lumberyard [Amazon Web Services, Inc. 2021]). 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++ [Apple Inc. 2021], a presentation from SIGGRAPH 2016 suggests that the game development industry could move towards C++ for GPU shader code [Foley 2016], and efforts are already underway to generate SPIR-V [Kessenich and Ouriel 2018] and DXIL [Microsoft 2019] from C++ (see Section 7). 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 shows the example shader from Section 2 rewritten using our unified C++-based shader programming environment. We explain the various parts of it in the next three sections.

4.2 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:

```

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,
19                                             pixelPos);
20
21         for (int i = 0; i < IterationCount; ++i) {
22             outColor *= filterMethod->doFiltering(pixelPos);
23         }
24
25         Output[DispatchThreadID] = outColor;
26     }
27 };

```

Listing 3. An example shader using our unified 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).

- 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 4.4.
- 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 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.⁷ 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 (see Section 4.4 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

⁷CUDA 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__`.

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.

4.3 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 25).

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.

4.4 Implement Specialization by Co-opting Virtual Function Calls

4.4.1 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.:


```

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 4. 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.

```

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 5) 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 values to the specialization parameters based on runtime information.⁸

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 1). We solve this issue by allowing a ShaderClass to use another ShaderClass as a specialization parameter.

4.4.2 ShaderClass Specialization Parameters. As shown in Listing 3 on line 22, 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 4 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 invoked 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.⁹ 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 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 4, the `ExtraParameter` uniform is only declared in the `HighQualityFilter` class (line 24), ensuring that programmers cannot use it elsewhere by mistake.

⁸To 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.

⁹Rather than using the built-in C++ runtime type information feature, we use our own, simplified mechanism to minimize performance overheads.

```

1  template<QUALITY, ITERATION_COUNT>
2  class FilterShader {
3  public:
4      /* Uniform parameter declarations */
5
6      /* GPU function */
7      void MainCS(uint2 DispatchThreadId) {
8          float2 pixelPos = /* ... */;
9          float4 outColor = ColorTexture.Sample(ColorSampler,
10                                             pixelPos);
11
12          for (int i = 0; i < ITERATION_COUNT; ++i) {
13              if (QUALITY == QualityEnumType::Low)
14                  /* Low Quality Method */
15              else if (QUALITY == QualityEnumType::Medium)
16                  /* Medium Quality Method */
17              else if (QUALITY == QualityEnumType::High)
18                  /* High Quality Method */
19          }
20
21          Output[DispatchThreadId] = outColor;
22      }
23  };

```

Listing 5. 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 4.4.3.

4.4.3 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, 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. 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 5, 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

```

1  if (quality == QualityEnumType::Low) {
2      if (iterCount == 2)
3          FilterShader<QualityEnumType::Low, 2> shader;
4      else if (iterCount == 4)
5          FilterShader<QualityEnumType::Low, 4> shader;
6      else if (iterCount == 8)
7          FilterShader<QualityEnumType::Low, 8> shader;
8      else if (iterCount == 16)
9          FilterShader<QualityEnumType::Low, 16> shader;
10 }
11 else if (quality == QualityEnumType::Medium) {
12     /* repeat ifs for iterCount */
13 }
14 else if (quality == QualityEnumType::High) {
15     /* repeat ifs for iterCount */
16 }

```

Listing 6. 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 5.

apparent when considering how to select the correct specialization in host code based on runtime parameter values, as shown in Listing 6.

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 combinations with corresponding `if` statements to select the right combination at runtime.¹⁰ 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.

4.5 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

¹⁰In 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.

access those conditionally defined members.¹¹ 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 this paper, 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. 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 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.

5 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,¹² 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++¹³ 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 4.2), 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, 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 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.¹⁴ 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,

¹¹This technique is similar to how they conditionally declare uniform parameters based on specialization parameters and, thus, has similar code maintainability downsides.

¹²<https://clang.llvm.org/docs/LibTooling.html>

¹³<https://github.com/redorav/hlslpp>

¹⁴ShaderClasses without entry point functions are not invocable shader programs, so outputting HLSL files for them is unnecessary.

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 by allowing programmers to rewrite existing HLSL code incrementally, which better enables existing systems to adopt a unified shader design. Secondly, as mentioned in Section 3, 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 [Proudfoot et al. 2001]. 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.

6 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 5) in some parts of our ported code. All results were obtained using UE4 version 4.25.4 built from source.¹⁵ 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.

6.1 ShaderClass Modularity

Listing 7 shows a simplified segment of GPU code from UE4’s temporal anti-aliasing (AA) shader, and Listing 8 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 is controlled by a specialization parameter. While Listing 7 only presents the original GPU code, our rewritten version in Listing 8 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 7 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

Table 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.¹⁶ 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 Lines of Code	C++ file* Lines of Code
Motion Blur Filter	902	920
Temporal AA	2,138	2,251

both host and GPU code (Listing 8 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 7. 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.

6.2 Lines of Code

Since our system design utilizes various abstractions for shader programming, we want to verify that these abstractions do not lead to excess code bloat. Table 1 compares the lines of code (LOC) for our rewritten shaders against the corresponding original UE4 code. In UE4, an HLSL 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]]`

¹⁵We used the release branch at commit b1e746725e8e540afe7ac586496b4ee4c081a10e

¹⁶<https://github.com/AlDanial/cloc>

```

1 Texture2D SceneDepth;
2 SamplerState SceneDepthSampler;
3 Texture2D SceneColor;
4 SamplerState SceneColorSampler;
5
6 #if CACHE_METHOD == REGISTER_CACHING
7     #define PRECACHE_COLOR
8
9     void
10    PrecacheColor(inout InputParams Input)
11        /* Sample from SceneColor; cache in Input */
12    {
13        float4
14        SampleCachedColor(InputParams Input)
15            /* Return color from cache in Input */
16        {
17            #elif CACHE_METHOD == GROUPSHARED_CACHING
18                #define PRECACHE_DEPTH
19                #define PRECACHE_COLOR
20
21                void
22                PrecacheDepth(InputParams Input)
23                    /* Sample from SceneDepth; cache in groupshared memory */
24                {
25                    float
26                    SampleCachedDepth(InputParams Input)
27                        /* Return depth from groupshared cache */
28                {
29                    void
30                    PrecacheColor(InputParams Input)
31                        /* Sample from SceneColor; cache in groupshared memory */
32                {
33                    float4
34                    SampleCachedColor(InputParams Input)
35                        /* Return color from groupshared cache */
36                }
37            #endif
38            #if !defined(PRECACHE_DEPTH)
39                void PrecacheDepth(InputParams Input) {}
40            #endif
41            float
42            SampleCachedDepth(InputParams Input)
43                /* Return depth sampled from SceneDepth */
44            {
45                #if !defined(PRECACHE_COLOR)
46                    void PrecacheColor(InputParams Input) {}
47                #endif
48                float4
49                SampleCachedColor(InputParams Input)
50                    /* Return color sampled from SceneColor */
51                {
52                    #endif

```

Listing 7. 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.

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.

6.3 Performance

Lastly, we evaluate the impact of our unified design on the runtime performance of GPU code generated by our translator. We run

```

1 class [[ShaderClass]] NoCaching {
2 public:
3     [[uniform]] Texture2D SceneDepth;
4     [[uniform]] SamplerState SceneDepthSampler;
5     [[uniform]] Texture2D SceneColor;
6     [[uniform]] SamplerState SceneColorSampler;
7
8     [[gpu]] virtual void PrecacheDepth(InputParams Input) const {}
9
10    [[gpu]] virtual float
11    SampleCachedDepth(InputParams Input) const
12        /* Return depth sampled from SceneDepth */
13    {
14        [[gpu]] virtual void PrecacheColor(InputParams Input) const {}
15
16        [[gpu]] virtual float4
17        SampleCachedColor(InputParams Input) const
18            /* Return color sampled from SceneColor */
19        {
20        };
21
22    class [[ShaderClass]] RegisterCaching : public NoCaching {
23    public:
24        [[gpu]] virtual void
25        PrecacheColor([[inout]] InputParams Input) const override
26            /* Sample from SceneColor; cache in Input */
27        {
28        [[gpu]] virtual float4
29        SampleCachedColor(InputParams Input) const override
30            /* Return color from cache in Input */
31        {
32        };
33
34    class [[ShaderClass]] GroupsharedCaching : public NoCaching {
35    public:
36        [[gpu]] virtual void
37        PrecacheDepth(InputParams Input) const override
38            /* Sample from SceneDepth; cache in groupshared memory */
39        {
40        [[gpu]] virtual float
41        SampleCachedDepth(InputParams Input) const override
42            /* Return depth from groupshared cache */
43        {
44        [[gpu]] virtual void
45        PrecacheColor(InputParams Input) const override
46            /* Sample from SceneColor; cache in groupshared memory */
47        {
48        [[gpu]] virtual float4
49        SampleCachedColor(InputParams Input) const override
50            /* Return color from groupshared cache */
51        {
52        };

```

Listing 8. The unified shader code ported from the code in Listing 7. Because the uniform declarations are shared between host and GPU code, this selection shows both halves of shader code.

the Infiltrator Demo [Epic Games 2015] (Figure 1) using both the original UE4 shader code and our rewritten versions and compare the GPU performance in Table 2. These results were produced using a resolution of 2560×1440 on a machine with an Intel Core 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.

7 RELATED WORK

Several GPU shading languages for real-time graphics support encapsulation of shader code and parameters via object-orientation, including Cg interfaces [Pharr 2004], HLSL classes [Microsoft 2018], Spark [Foley and Hanrahan 2011], and Slang [He et al. 2018]. The idea dates back to the RenderMan Shading Language (RSL) [Hanrahan and Lawson 1990]. Furthermore, aspects of our ShaderClass

Table 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 [Epic Games 2015]. These numbers were obtained using benchmarking tools provided by UE4.

Shader	Original UE4 Code (time in ms)			Unified Code (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

design take inspiration from Kuck and Wesche [2009]. 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 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.

Vulkan’s “specialization constants” [The Khronos Vulkan Working Group 2021] allow host code to modify the values of constants in GPU code at application runtime (Metal has a similar feature). This feature could be used to implement basic-type specialization parameters without requiring programmers to statically enumerate all possible value options. However, it is insufficient for expressing and generating specializations that include different uniform parameters and GPU functions, which is the purpose of ShaderClass-type specialization parameters.

Sh [McCool et al. 2002] implements shader programming as an embedded domain-specific language (DSL) in C++. GPU shader code is expressed using special types and operators, meaning that host and GPU code use distinct syntax for things like control flow. In contrast, our work uses regular C++ for both host and GPU code (with attributes to annotate elements specific to shader programming), presenting a unified environment where host and GPU code use the



Fig. 1. A screenshot from the Infiltrator Demo [Epic Games 2015]. We use this demo for our performance evaluation.

same types and functions. Additionally, Sh uses runtime metaprogramming to generate GPU code, whereas our system performs all code generation at compile time.

BraidGL [Sampson et al. 2017] and Selos [Seitz et al. 2019] both present shader programming environments that meet our definition of “unified,” but neither BraidGL nor Lua-Terra [DeVito et al. 2013] (the language in which Selos is written) are widely used languages in real-time graphics. In addition, both of these systems rely on features (*static staging* and *staged metaprogramming*, respectively) that are not available in such widely used languages. Rather than requiring that new features such as these be added to the underlying language, our approach focuses on co-opting existing language features to implement unified shader programming.

While most real-time graphics applications use separate languages for host and GPU code, some recent projects explore enabling single-language shader programming. Rust GPU [Embark Studios 2021] 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) [Baxter 2021]. Both of these projects are working to satisfy a necessary condition for unified shader programming—the ability to author both host 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 towards 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.

Along with GPU shader code support, Circle also adds many other language features to C++, including new general-purpose metaprogramming features. Using these new features, it may be possible to build a unified shader programming system within the Circle language. The philosophy of our work 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. Secondly, 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.

8 CONCLUSION

In this paper, 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.

In the future, we are interested in expanding our ShaderClass design to support shader code modularity in situations where complete specialization is not feasible. For example, deferred rendering utilizes dynamic dispatch in GPU code to invoke different code per-material-type based on per-pixel data. As a result, these shader programs cannot be completely specialized for each material type. However, if a particular application or scene uses only a subset of material types, such deferred rendering shader programs can be partially specialized to that subset. By expanding our ShaderClass implementation to support both static and dynamic dispatch, we believe that programmers could better modularize material type code, and the underlying system could then generate partial specializations to improve overall performance.

While our current work focuses on real-time graphics programming in C++, we hope that the broader lessons can be applied to other programming languages, application domains, and processor types. 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. Beyond graphics programming, we believe the strategy of co-opting existing language features can be used to implement the semantics and optimizations needed for other domains and potentially other processor types besides a CPU host and a GPU coprocessor. 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.

ACKNOWLEDGMENTS

We thank Anjul Patney, Chuck Rozhon, Yong He, Brian Karis, Ola Olsson, Andrew Lauritzen, Yuriy O'Donnell, Angelo Pesce, Charlie Birtwistle, Michael Vance, and Dave Shreiner for guidance, feedback, and technical advice. Thank you to NVIDIA Corporation for hardware donations and to Intel Corporation hardware donations and financial support.

REFERENCES

- Amazon Web Services, Inc. 2021. Amazon Lumberyard. <https://aws.amazon.com/lumberyard/>.
- Apple Inc. 2014. Metal. <https://developer.apple.com/documentation/metal>.
- Apple Inc. 2021. *Metal Shading Language Specification Version 2.3*. <https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf>
- Sean Baxter. 2021. Circle C++ Shaders. <https://github.com/seanbaxter/shaders/blob/master/README.md>.
- Lewis Crawford and Michael O'Boyle. 2019. Specialization Opportunities in Graphical Workloads. In *The 28th International Conference on Parallel Architectures and Compilation Techniques* (Seattle, WA, USA) (PACT 2019). 272–283. <https://doi.org/10.1109/PACT.2019.00029>
- Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A Multi-Stage Language for High-Performance Computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) (PLDI 2013). 105–116. <https://doi.org/10.1145/2491956.2462166>
- Ramy El Garawany. 2016. Deferred Lighting in Uncharted 4. In *ACM SIGGRAPH 2016 Courses* (Anaheim, CA, USA) (SIGGRAPH 2016). <https://doi.org/10.1145/2897826.2940291> Part of the course: Advances in Real-Time Rendering, Part I.
- Electronic Arts Inc. 2021. Frostbite Engine. <https://www.ea.com/frostbite>.
- Embark Studios. 2021. Rust GPU. <https://github.com/EmbarkStudios/rust-gpu>.
- Epic Games. 2015. Infiltrator Demo. <https://www.unrealengine.com/marketplace/en-US/product/infiltrator-demo>.
- Epic Games, Inc. 2019. Unreal Engine 4 Documentation. <https://docs.unrealengine.com/en-us/>.
- T. Foley. 2016. A Modern Programming Language for Real-Time Graphics: What is Needed?. In *ACM SIGGRAPH 2016 Courses* (Anaheim, CA, USA) (SIGGRAPH 2016). <https://doi.org/10.1145/2897826.2940293> Part of the course: Open Problems in Real-Time Rendering.
- T. Foley and Pat Hanrahan. 2011. Spark: Modular, Composable Shaders for Graphics Hardware. *ACM Transactions on Graphics* 30, 4, Article 107 (July 2011), 12 pages. <https://doi.org/10.1145/2010324.1965002>
- Kate Gregory and Ade Miller. 2012. *C++ AMP: Accelerated Massive Parallelism with Microsoft® Visual C++®*. Microsoft Press.
- Pat Hanrahan and Jim Lawson. 1990. A Language for Shading and Lighting Calculations. In *Computer Graphics (Proceedings of SIGGRAPH 90)*. 289–298.
- Yong He, Kayvon Fatahalian, and T. Foley. 2018. Slang: Language Mechanisms for Extensible Real-time Shading Systems. *ACM Transactions on Graphics* 37, 4, Article 141 (July 2018), 13 pages. <https://doi.org/10.1145/3197517.3201380>
- John Kessenich, Dave Baldwin, and Randi Rost. 2017. *The OpenGL® Shading Language (Version 4.50)*. The Khronos Group Inc. <https://www.khronos.org/registry/OpenGL/specs/gl/GLSLangSpec.4.50.pdf>
- John Kessenich and Boaz Ouriel. 2018. *SPIR-V Specification (Version 1.00)*. The Khronos Group Inc. <https://www.khronos.org/registry/spir-v/specs/1.0/SPIRV.pdf>
- Khronos Group. 2016. *Vulkan 1.0.12 - A Specification*. The Khronos Group Inc. <https://www.khronos.org/registry/vulkan/specs/1.0/pdf/vkspec.pdf>
- Roland Kuck and Gerold Wesche. 2009. A Framework for Object-Oriented Shader Design. In *Advances in Visual Computing (ISVC 2009)*. 1019–1030. https://doi.org/10.1007/978-3-642-10331-5_95
- Juan Linietsky, Ariel Manzur, and contributors. 2021. Godot Engine. <https://godotengine.org/>.
- Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. 2002. Shader Metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Saarbrücken, Germany) (HWSW '02). 57–68. <http://dl.acm.org/citation.cfm?id=569046.569055>
- Microsoft. 2014. Shader Model 5.1. [https://msdn.microsoft.com/en-us/library/windows/desktop/dn933277\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/dn933277(v=vs.85).aspx).
- Microsoft. 2018. Interfaces and Classes. <https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/overviews-direct3d-11-hlsl-dynamic-linking-class>.
- Microsoft. 2019. DirectX Shader Compiler. <https://github.com/Microsoft/DirectXShaderCompiler>.
- Microsoft. 2020. Direct3D. <https://docs.microsoft.com/en-us/windows/win32/direct3d>.
- NVIDIA Corporation. 2007. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. (Jan. 2007). <http://developer.nvidia.com/cuda>.
- Matt Pharr. 2004. An Introduction to Shader Interfaces. In *GPU Gems*, Randima Fernando (Ed.). Addison Wesley, Chapter 32, 537–550.
- Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. 2001. 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)*. 159–170. <https://doi.org/10.1145/383259.383275>
- Adrian Sampson, Kathryn S. McKinley, and Todd Mytkowicz. 2017. Static Stages for Heterogeneous Programming. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 71 (Oct. 2017), 27 pages. <https://doi.org/10.1145/3133895>
- Mark Segal, Kurt Akeley, Chris Frazier, Jon Leech, and Pat Brown. 2017. *The OpenGL® Graphics System: A Specification (Version 4.5 (Core Profile) - June 29, 2017)*. The Khronos Group Inc. <https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf>
- Kerry A. Seitz, Jr., T. Foley, Serban D. Porumbescu, and John D. Owens. 2019. Staged Metaprogramming for Shader System Development. *ACM Transactions on Graphics* 38, 6 (Nov. 2019), 202:1–202:15. <https://doi.org/10.1145/3355089.3356554>
- The Khronos Vulkan Working Group. 2021. *Vulkan 1.1.178 - A Specification (with KHR extensions)*. The Khronos Group Inc., Chapter 10.8. Specialization Constants. <https://www.khronos.org/registry/vulkan/specs/1.1-khr-extensions/html/chap10.html#pipelines-specialization-constants>
- Unity Technologies. 2019. Unity User Manual (2019.1). <https://docs.unity3d.com/Manual/index.html>.