Title

Supporting Unified Shader Specialization by Co-opting C++ Features

Authors

Seitz, Kerry A
Foley, Theresa
Porumbescu, Serban D
et al.

Publication Date

2022-07-25

DOI

10.1145/3543866

Supporting Unified Shader Specialization by Co-opting C++ Features

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

Modern unified programming models (such as CUDA and SYCL) that combine host (CPU) code and GPU code into the same programming language, same file, and same lexical scope lack adequate support for GPU code specialization, which is a key optimization in real-time graphics. Furthermore, current methods used to implement specialization do not translate to a unified environment. In this paper, we create a unified shader programming environment in C++ that provides first-class support for specialization by co-opting C++'s attribute and virtual function features and reimplementing them with alternate semantics to express the services required. 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

ACM Reference Format:
Kerry A. Seitz, Jr., Theresa Foley, Serban D. Porumbescu, and John D. Owens. 2022. Supporting Unified Shader Specialization by Co-opting C++ Features. *Proc. ACM Comput. Graph. Interact. Tech.* 5, 3, Article 25 (July 2022), 17 pages. https://doi.org/10.1145/3543866

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 [Segal et al. 2017; The Khronos® Vulkan Working Group 2016], 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

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.

2577-6193/2022/7-ART25
https://doi.org/10.1145/3543866

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 SYCL [The Khronos® SYCL™ Working Group 2021] author GPU code as C++ lambda functions or as named function objects 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 construction—avoid an entire class of compatibility issues that must be handled manually in graphics programming.[1] Because of the associated code reuse, compatibility, and ease-of-use benefits, our overarching goal is to enable development of unified shader programming systems that are practically useful for large-scale real-time graphics applications.

While CUDA and SYCL provide powerful unified programming models for General-Purpose GPU (GPGPU) computing, neither they nor other popular unified systems provide adequate support for GPU code *specialization*. 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 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. While this dichotomy can be worked around in systems that use separate environments and separate parameter declarations for host code and GPU code, it represents a fundamental tension in unified systems where the same parameter must serve both a compile-time and runtime role.

Because of its importance in real-time graphics, we aim to demonstrate how to provide support for specialization in a unified system. Furthermore, to be practically useful today, such a unified specialization implementation must be feasible in an existing, widely used language in real-time graphics, and the effort required to build and maintain such a system must be sufficiently modest in order to be tractable for most design teams. To this end, we present the following contributions:

[1] Similar to how compilers help keep .h and .cpp files in sync, a unified system extends this benefit to GPU code. Current shader programming models do not validate code across the host (.cpp) and GPU (.hlsl) boundary, whereas unified systems do.

- The design of a unified programming environment in C++ that provides first-class support for specialization by co-opting existing language features (C++ attributes and virtual functions) and implementing them with alternate semantics; and
- A Clang-based tool[2] that translates code using our modified C++ semantics to standard C++ and HLSL code, compatible with Unreal Engine 4.

An explicit goal of this work is to show that a unified environment can be integrated into an existing, large-scale engine and coexist with its current shader programming system. As mentioned above, our implementation targets Unreal Engine 4 for this purpose. Also, our current implementation focuses on compute shaders, but we believe the design of our specialization system is compatible with other shader types as well. We discuss other shader types, along with other limitations and future work, in Section 6.

2 RELATED WORK

2.1 Implementing Specialization

Popular game engines like Unreal Engine 4 (UE4) and Unity implement GPU shader code specialization using preprocessor-based methods.[3] Programmers express specializations in GPU code using C-style preprocessor facilities (e.g., `#ifs`), and then this GPU code is compiled multiple times with different `#define` options to generate multiple *shader variants* of the original code. At runtime, host code selects which variant to invoke based on runtime variables that correspond to these `#defines`. Effectively, code in these systems has two representations of these *specialization parameters*: a runtime variable for host code and a compile-time `#define` for GPU code.

However, this technique is not viable in a unified environment, where we desire a unified representation for specialization parameters. C-preprocessor directives are evaluated in the first step of compilation. 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. Vulkan's "specialization constants" allow host code to modify the values of constants in GPU code at application runtime. While this feature eliminates the need to specify all possible values for a specialization parameter upfront, it still uses distinct, non-unified definitions of these parameters in host code and GPU code.

C++ templates, another common code specialization technique familiar to many programmers, are also inadequate for supporting GPU shader code specialization. In C++, template parameters are evaluated at application compile time, resulting in multiple compiled versions of templated source code (analogous to compiling multiple shader variants). However, templates are insufficient for expressing runtime decisions in host code, since all template parameter values must be available at application compile time. In other words, templates provide compile-time polymorphism, which is desirable for GPU code, but host code needs runtime polymorphism instead. This issue is illustrated by the CUDA sample discussed in Listing 1, which shows how our design (introduced in Section 3) results in significantly shorter and more maintainable code compared to the template-based CUDA sample.

In the Slang shading language [He et al. 2018], specialization options in GPU code are expressed primarily through interface-constrained generics, and host code interfaces with GPU code to generate specialized variants using a runtime reflection API. Because host and GPU code exist in different environments, Slang sidesteps the compile-time vs. runtime dichotomy of specialization parameters. Furthermore, popular host-code languages like C++ do not support Slang-style generics/interfaces,

[2] https://github.com/owensgroup/UnifiedShaderSpecialization
[3] For additional background, please see the supplementary material.

```
1  // Boilerplate
2  template<class T>
3  class [[ShaderClass]] ReduceBase {
4      [[specialization_SparseInt(1024, 512, ...)]] int numThreads;
5      [[specialization_Bool]] bool nIsPow2;
6
7      [[gpu]] virtual void reduceKernel(...) = 0;
8  };
9
10 // Original CUDA code, replaced by ShaderReduce6
11 template <class T, unsigned int blockSize, bool nIsPow2>
12 __global__ void reduce6(...) { ... }
13
14 // Our version (2 extra lines vs. reduce6)
15 template<class T>
16 class [[ShaderClass]] ShaderReduce6 : public ReduceBase {
17     [[gpu]] virtual void reduceKernel(...) { ... }
18 };
19
20 // Boilerplate
21 template<class T>
22 class [[ShaderClass]] ReduceRunner {
23     [[specialization_ShaderClass]] ReduceBase* reducer;
24
25     [[entry_ComputeShader(reducer->numThreads, 1, 1)]]
26     void runReducer(...) {
27         reducer->reduceKernel();
28     }
29 };
30
31 // The following replaces the ~400 lines of switch-/if-statements
32 template<class T>
33 void reduce(...) {
34     ...
35     ReduceBase* reduceShader;
36
37     switch (whichKernel) {
38         case 0: reduceShader = new ShaderReduce0<T>(); break;
39         ...
40         case 9: reduceShader = new ShaderReduce9<T>(); break;
41     }
42
43     reduceShader->numThreads = threads;
44     reduceShader->nIsPow2 = isPowTwo(size);
45
46     ReduceRunner runner;
47     runner.reducer = reduceShader;
48     runner.addComputePass(...);
49 }
```

Listing 1. Because our system provides first-class support for shader specialization, invoking GPU code from host code is significantly more maintainable compared to using C++ templates for this task. The code in this Listing accomplishes the same task as lines 633–1025 of the reduction_kernel.cu CUDA sample [NVIDIA Corporation 2022]. Invoking the templated GPU code in the CUDA sample requires ~400 lines of code, with nested switch- and if-statements and significant repetition (variadic templates, mp_list, and typelists do not fix this issue either). Our system (Section 3) can accomplish this same tasks with ~30 lines of code (plus minor additional boilerplate), as shown in this Listing.

While C++ templates support compile-time specialization of code, they are inadequate for implementing the shader specialization optimization because of the compile-time vs. runtime needs in GPU vs. host code, respectively. The problem stems from the need to map runtime values to compile-time template parameters, which results in a major maintenance burden. For example, adding support for 2048 threads in the CUDA sample requires adding a new case to each of the seven `switch (threads) {...}` blocks, which leads to further code duplication. In contrast, in our version shown above, adding a new option requires simply adding another integer to the specialization parameter (discussed in Section 3.3) on line 4.

which limits the ability to apply Slang's specialization methods in existing engines seeking to create unified systems. Similarly, Rodent [Pérard-Gayot et al. 2019] utilizes partial evaluation [Futamura 1983] to generate specialized renderers for CPUs and GPUs, but widely used languages in computer graphics do not have such partial evaluation features.

2.2 Encapsulation of Shader Code

Several shading languages support encapsulation of shader code and parameters via object-orientation, including Cg interfaces [Pharr 2004], HLSL classes [Microsoft 2018], Spark [Foley and Hanrahan 2011], Slang [He et al. 2018], and the RenderMan Shading Language [Hanrahan and Lawson 1990]. Our design in Section 3.2 utilizes this approach and takes further 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.

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, presenting a unified environment where host and GPU code use the same types and functions. Additionally, Sh uses runtime metaprogramming to generate GPU code, whereas our system performs all code generation at compile time.

2.3 Unified Shader Programming

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, such as C++. C++ does not support the *static staging* feature that BraidGL uses to express shader code and specializations. Similarly, Selos relies on *staged metaprogramming*, defined as a key set of language features that are available in Lua-Terra, but C++ lacks these key language features. 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 specialization.

New projects like Rust GPU [Embark Studios 2021] and Circle [Baxter 2021] allow programmers to author GPU shader code in general-purpose systems languages (Rust and C++, respectively). Both of these projects aim to satisfy a necessary condition for unified shader programming—the ability to author both host and GPU code in the same language. 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 Circle is based on C++, it has departed significantly from the standard language by adding other language features, including new general-purpose metaprogramming features. While it may be possible to build a cohesive specialization system using these new Circle features, our goal is instead to introduce as few syntactic and semantic changes to C++ as possible, which both lowers development and maintenance costs and better enables programmers to write code that looks and feels like standard C++.

3 UNIFIED SPECIALIZATION SYSTEM DESIGN

The key insight of this work is that we can add support for specialization in a unified programming environment by co-opting existing features of a programming language and implementing them

```
class [[ShaderClass]] FilterShader {
public:
  [[uniform]] Texture2D           ColorTexture;
  [[uniform]] SamplerState        ColorSampler;
  [[uniform]] RWTexture2D<float4> Output;

  [[specialization_ShaderClass]]
  FilterMethod* filterMethod;

  [[specialization_SparseInt(2, 4, 8, 16)]]
  int IterationCount;

  [[entry_ComputeShader(8, 8, 1)]]
  void MainCS([[SV_DispatchThreadID]] uint2 DispatchThreadID) const
  {
    float2 pixelPos = /* ... */;
    float4 outColor = ColorTexture.Sample(ColorSampler, pixelPos);

    for (int i = 0; i < IterationCount; ++i) {
      outColor *= filterMethod->doFiltering(pixelPos);
    }

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

Listing 2. 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).

with alternate semantics to provide the services required. Because our high-level motivation is to bring unified programming to existing, large-scale graphics applications, we must ensure that our methodologies can integrate with existing code and toolchains. Thus, we have chosen to demonstrate our approach by building a unified system in C++, since it is the most widely used language in real-time graphics (as evidenced by its use in many game engines [Amazon Web Services, Inc. 2021; Electronic Arts Inc. 2021; Epic Games, Inc. 2019; Linietsky et al. 2021]). In contrast, creating a new programming language or using an uncommon one would lead to increased development costs, because graphics developers would need to rewrite large portions of their codebases or write additional code to interface between languages. Similarly, rather than adding new language features to C++, we have chosen to stay as close to standard C++ as possible in order to avoid the potential maintenance costs of integrating arbitrary new features with future versions of C++. Instead, our approach is to co-opt features already present in C++, thereby maintaining compatibility as the language continues to evolve. While our current implementation utilizes C++, we believe that our key insight, as well as many of the ideas presented below, are transferable to other languages as well.

❧

In the remainder of this section, we discuss the major design elements of our unified specialization system. Listing 2 shows an example shader written using our unified C++-based programming environment. We explain the various parts of it in the next three sections.

3.1 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 3.3.
- 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 14).
- 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.[4] 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 3.3 for further discussion).

Using 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 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). As mentioned in Section 2.2, other systems have used object orientation to modularize shader code. However, our ShaderClasses can contain both host and GPU code, which is a major departure from prior work.

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:

```
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 14). 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

[4]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__`.

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 23).

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 (line 8). Member variables of a ShaderClass type must be declared as specialization parameters, for reasons we discuss next.

3.3 Implement Specialization by Co-opting Virtual Function Calls

3.3.1 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.[5]

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 (Listing 2 line 10). Using these options, our translator (Section 4) 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.[6]

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 standard practice of using preprocessor `#ifs` does not work in a unified system, as discussed in Section 2.1). We solve this issue by allowing a ShaderClass to use another ShaderClass as a specialization parameter.

3.3.2 ShaderClass Specialization Parameters. As shown in Listing 2 on line 20, the `doFiltering()` function is provided by a member variable of type `FilterMethod` (Listing 2 line 8). `FilterMethod` is itself a ShaderClass, and it also has ShaderClass subtypes. Listing 3 shows the implementations of these types.

[5] E.g., UE4's system, as discussed in the supplementary material.

[6] 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.

```
1  class [[ShaderClass]] FilterMethod {
2  public:
3    [[gpu]] virtual float4 doFiltering(float2 pos) const = 0;
4  };
5
6  class [[ShaderClass]] LowQualityFilter : public FilterMethod {
7  public:
8    [[gpu]] virtual float4 doFiltering(float2 pos) const override
9    {
10     /* Low Quality Method */
11   }
12 };
13
14 class [[ShaderClass]] MedQualityFilter : public FilterMethod {
15 public:
16   [[gpu]] virtual float4 doFiltering(float2 pos) const override
17   {
18     /* Medium Quality Method */
19   }
20 };
21
22 class [[ShaderClass]] HighQualityFilter : public FilterMethod {
23 public:
24   [[uniform]] int ExtraParameter; // specific to this class
25   [[gpu]] virtual float4 doFiltering(float2 pos) const override
26   {
27     /* High Quality Method */
28   }
29 };
```

Listing 3. 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.

The doFiltering() method is declared as a virtual method in the base FilterMethod class (Listing 3 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 this 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.[7] 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. For example, in Listing 3, the `ExtraParameter` uniform parameter (line 24) only applies when using the high-quality filter method. Because this parameter is encapsulated within the `HighQualityFilter` class, it cannot be accessed elsewhere by mistake. In contrast, standard practice in other systems would be to declare this parameter under a preprocessor `#if`. If other parts of the HLSL code need to access this 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. Our design not only brings the specialization optimization to a unified environment but also enables shader programmers to utilize more features of C++ to organize their code, rather than relying solely on the limited feature sets provided in standard shading languages.

While our current implementation always converts virtual functions to use static dispatch in GPU code, our design intentionally leaves open the possibility of compiling this same code to use dynamic dispatch instead. For example, our implementation could generate conditional statements to dynamically select which function to invoke, or if HLSL adds support for virtual functions in the future, our system could compile directly to that feature. Dynamic dispatch can reduce the number of compiled shader variants and is also important for real-time ray tracing. In addition, generating partially specialized variants that utilize both static and dynamic dispatch can improve performance in deferred rendering [El Garawany 2016; Seitz et al. 2019]. Alternatively, relying on C++ template metaprogramming for shader specialization would necessarily always generate fully specialized, static shader variants, thereby limiting future adaptability of the system (in addition to the issues discussed in Listing 1). Thus, co-opting virtual functions for specialization provides the flexibility to support future scenarios that will be important in real-time rendering.

4 TRANSLATION TOOL IMPLEMENTATION

To implement our design, we built a source-to-source translator based on Clang. The translator uses Clang's LibTooling API [The Clang Team 2022], 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++ [López 2022] 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. Our translator is separated into three major components: the frontend, the host backend, and the GPU backend.

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

4.1 Frontend

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 3.1). 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.

4.2 Host Backend

Our current implementation outputs code that utilizes UE4's macro and render graph systems under the hood, in order to easily integrate with existing UE4 code. Other game engines and renderers could also be supported by writing additional backends for them using a similar approach.

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 2 would result in three ImplClasses, one for each `FilterMethod` subtype.

In addition, the translator 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.

4.3 GPU Backend

Similar to the host backend, our implementation currently targets HLSL for GPU code, but it could support other shading languages like GLSL via additional backends or by cross-compiling HLSL to another language (UE4 uses the latter approach). Our translator's GPU backend outputs an HLSL file for each ShaderClass with an entry point function.[8] 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 and 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 verbatim. This feature serves two practical purposes. Primarily, it lowers the barrier to porting shader code to use this system by

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

allowing programmers to rewrite existing HLSL code incrementally, which better enables existing systems to adopt a unified shader design. Secondarily, 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 specialization without first implementing every HLSL feature in C++, and vice versa, as a prerequisite. Furthermore, the task of C++-to-GPU-shader-code translation is already being explored by Circle [Baxter 2021].

5 EVALUATION

To evaluate our design, we ported shaders from UE4 to our system. These shaders are complex, multi-platform, highly optimized, production shaders, and our ported versions are fully featured, drop-in replacements for their UE4 counterparts. Because feature-complete C++-to-HLSL translation is out of scope for this work, we use hardcoded HLSL code (Section 4.3) in some parts of our ported code. All results were obtained using UE4 version 4.25.4 built from source.[9] 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 below.

5.1 ShaderClass Modularity

Qualitatively, we have observed that using ShaderClasses to modularize shader code and specialization options leads to code that is more maintainable and easier to understand. For example, UE4's temporal anti-aliasing shader[10] has many different specialization parameters usages scattered throughout the GPU code. One such parameter chooses between different methods to use for caching texture reads. These caching methods all rely on the same set of uniform parameters, and these particular uniforms are not used elsewhere in this shader. However, all uniform parameters are declared as global variables, so this structure is not evident from looking at the code. In our unified version of this code, we create a base ShaderClass to encapsulate these caching-specific uniform variables, and then each caching method inherits from this base class. Thus, the code dependencies are readily apparent.

Furthermore, code reuse is made simpler and more apparent in our version of this shader as well. Our base ShaderClass declares four virtual functions and provides implementations for them. One subtype overrides all four functions, while another only overrides two of them (and reuses the base class implementations for the other two). Again, this code structure is plainly evident in our version of this shader, but uncovering this underlying structure from the original UE4 source code required spending significant time tracking dependencies across ~500 lines of HLSL code (within a ~2,000 line file). This shader (and many others) is riddled with #ifs throughout, making it very difficult to understand and modify. This practice is standard in the industry. In contrast, our system enables object-oriented organization of shader code—while still supporting shader specialization—reducing the need for the ~225 #if/#elif/#else lines in the original shader. We believe organizing code using modern programming language features is vastly preferable to hundreds of scattered #ifs. Our design enables users to cleanly modularize their shader code using C++ classes and virtual functions, while still resulting in specialized shader variants that graphics programmers expect.

[9] We used the release branch at commit b1e746725e8e540afe7ac586496b4ee4c081a10e.
[10] Interested readers can register for free access to UE4 source code and view this shader in the UE4 repository at Engine/Shaders/Private/PostProcessTemporalAA.usf.

Table 1. Lines of code (LOC) 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 [Danial 2022]. 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.

Shader (lines of code)	**Original UE4 Code** C++ file & HLSL file	**Our Unified Code** C++ file[11]
Motion Blur Filter	902	920
Temporal AA	2,138	2,251

5.2 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. Some of the additional lines in the unified code come from stylistic choices (e.g., putting the `[[gpu]]` function attribute on its own line). Furthermore, some 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. We manually copied and ported the necessary code segments from the header files into our unified shader files, so these code segments are counted in the LOC numbers for the unified versions of the shaders. We copied 7 LOC for the Motion Blur Filter shader and 5 LOC for the Temporal AA shader. The original UE4 shaders `#included` this code, so we count the `#include` lines for UE4 but not the code segments in the headers. 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.

5.3 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[12] [Epic Games, Inc. 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.

6 LIMITATIONS AND FUTURE WORK

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

[11]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.

[12]Epic Games's Infiltrator Demo video: https://youtu.be/dO2rM-l-vdQ

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, Inc. 2015]. These numbers were obtained using benchmarking tools provided by UE4.

Shader	**Original UE4 Code**			**Our Unified Code**		
(runtime 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

corresponding `#ifs` throughout the HLSL file whenever they need to access those conditionally defined members.[13] 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 make up an increasingly large portion of a modern game's shader code and are sufficient

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

Fig. 1. Screenshots from the Infiltrator Demo [Epic Games, Inc. 2015]. We use this demo for our performance evaluation. Screenshots used with permission of Epic Games Unreal Engine Marketplace.

to demonstrate the challenges to developing unified shader specialization, as well as how our solutions address these issues. Therefore, we leave exploration of shaders like UE4's *Material* and *MeshMaterial* shaders as future work, though we believe our method for supporting specialization can extend to these shaders as well. Similarly, our current implementation only supports compute shaders, but adding support for pixel-type Global Shaders requires only minor changes in the host and GPU backends since pixel- and compute-type Global Shaders are structured comparably.

Supporting specialization is one piece of the unified-shader-programming puzzle, and there are many other interesting questions in this space. For example, MeshMaterial shaders need to coordinate varying parameters between different shader types (e.g., a vertex shader outputs varying parameters that a pixel shader then consumes). A unified system has the potential to provide a robust mechanism for coordinating this information between different shader types, but the best way to express these cross-shader relationships is an open question (though the *pipeline shader* design [Proudfoot et al. 2001] looks promising). Another area for future work is exploring how a unified environment can better handle data movement and synchronization between host and GPU code. Our implementation uses UE4's render graph system under the hood for this purpose. However, a unified system has the opportunity to make better scheduling and memory-transfer decisions because it has a broader view of both the host and GPU code together. By keeping our system as close to standard C++ as possible, we hope that our ideas can provide a foundation for supporting the crucial shader specialization optimization in unified systems, thereby allowing future works to focus their efforts on other challenges in unified shader programming.

7 CONCLUSION

In this paper, we have presented the design of a unified shader specialization system in C++. By co-opting existing features of the language (attributes and virtual functions) and implementing them with alternate semantics, we are able to provide 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. Changing the semantics of C++, even in limited ways, does have some disadvantages. Programmers must contend with different semantics in different portions of code, which may increase cognitive load. However, we believe the benefits of unified programming and first-class shader specialization far outweigh the downsides of tweaking C++ semantics specifically in GPU shader code.

Our work demonstrates that unified shader specialization is possible in C++ with only minimally invasive, under-the-hood changes. Prior work (Section 2.3) has relied on advanced metaprogramming and partial-evaluation features in non-mainstream languages, but these features are absent from the popular languages used in real-time graphics. We aspire to bring the benefits demonstrated by these prior works to existing engines today. As such, our work is constrained by the use of—and large investment in—C++. Via the co-opting approach, our work helps to bring the benefits of unified shader programming to a widely used language, despite the lack of these advanced language features.

While our current work focuses on the shader specialization optimization in C++, we hope the broader lessons can be applied to other programming languages, application domains, and processor types. Bringing unified shader specialization to other languages may involve co-opting different features, but we think 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, Dave Shreiner, and the anonymous reviewers for guidance, feedback, and technical advice. Thank you to NVIDIA Corporation for hardware donations and to Intel Corporation for 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 *Proceedings of 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

Al Danial. 2006–2022. cloc. https://github.com/AlDanial/cloc.

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, Inc. 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 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

Yoshihiko Futamura. 1983. Partial Computation of Programs. In *RIMS Symposium on Software Science and Engineering* (Kyoto, Japan). 1–35. https://doi.org/10.1007/3-540-11980-9_13

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

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/.

Emilio López. 2017–2022. hlslpp. https://github.com/redorav/hlslpp.

Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. 2002. Shader Metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Saarbrucken, Germany) *(HWWS '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.

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

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.

NVIDIA Corporation. 2022. cuda-samples: reduction_kernel.cu. https://github.com/NVIDIA/cuda-samples/blob/b312abaa07ffdc1ba6e3d44a9bc1a8e89149c20b/Samples/2_Concepts_and_Techniques/reduction/reduction_kernel.cu#L633-L1025.

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

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 Clang Team. 2007–2022. LibTooling. https://clang.llvm.org/docs/LibTooling.html.

The Khronos® SYCL™ Working Group. 2021. *SYCL™ 2020 Specification (revision 4)*. The Khronos Group Inc. https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf

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

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