Evolving a Direct3D 12 Rendering Engine to use Shader Components for Modularity and Performance

Anonymous Author(s)

ABSTRACT

Shader components were recently proposed as an abstraction for authoring modular libraries of shader code that realize the high rendering performance typical of globally optimized, less modular solutions. This paper conducts a detailed case study that explores the challenge of incrementally modifying an existing Direct3D 12 engine (the open source Falcor engine) to use shaders authored in terms of shader components. We discuss the set of changes to Falcor needed to integrate shader components, written using the Spire shading language, into the engine. These changes included modifying the engine's CPU host code to use Spire's shader components-related compiler services and rearchitecting Falcor's material/light shader library in terms of components. We find that the modified Falcor engine and its accompanying shader library exhibit a notably simpler design, allowing new features, such as tessellation, to be quickly added to the engine. Extensibilty does not come at the cost of performance. The Direct3D version of Falcor build upon Spire's shader component services achieves higher rendering performance due to more-efficient shader parameter binding and use of static shader specialization. We discuss the case study in detail, and provide guidance to future engine implementors that wish to adopt the Spire shader component idea in their rendering architectures.

CCS CONCEPTS

• **Computing methodologies** → *Graphics systems and interfaces*;

KEYWORDS

shading languages, real-time rendering

ACM Reference format:
Anonymous Author(s). 2017. Evolving a Direct3D 12 Rendering Engine to use Shader Components for Modularity and Performance. In *Proceedings of High Performance Graphics, Los Angeles, USA, July 2017 (Submitted to HPG17)*, 10 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Designers of real-time rendering engines must balance the conflicting goals of architecting a modular, extensible shading system, and maximizing rendering performance. Modularity is desirable for software maintenance and extensibility, particularly since popular commercial engines such as Unreal [Epic Games 2015] or Unity [Unity Technologies 2017] feature large shader libraries that are used for many titles, and with different shading feature requirements. Unfortunately, modular shading architectures can suffer from lower performance. For example, modularity may complicate the ability to statically specialize shaders (causing the application to "pay" for features it does not use), and it prevents modern engines from making efficient use of the parameter binding model presented by modern Direct3D12/Vulkan graphics APIs [McDonald 2016; Pranckevičius 2015].

Recently the idea of shader components [He et al. 2017] was proposed as an abstraction for authoring modular libraries of shader code that realize high rendering performance (static specialization to shading features used, efficient binding of shader parameters) that is typical of globally optimized, less modular solutions. However, prior evaluation of shader components was performed in the context of an academic renderer, which was written from the ground up with the component idea in mind. Since real rendering engines can rarely be redesigned from scratch, this paper describes the results of a detailed case study that explores the challenge of incrementally modifying an existing Direct3D 12 engine (the open source Falcor engine [Benty 2016]) to use shaders authored in terms of shader components. The case study was performed by integrating the Spire shading language and compiler framework [He et al. 2016, 2017], which provides a full implementation of shader components, into Falcor. Specifically we contribute:

- A detailed description of the challenges faced when porting Falcor's modular shading system, originally designed for OpenGL with bindless texture support, to the Direct3D 12 API.
- A rearchitecture of the Falcor engine around the shader component abstraction. This involved modifying host CPU code to use shader components-related services provided by the Spire compiler, and rearchitecting Falcor's material/-light shader library in terms of Spire shader components.
- An assessment of the benefits of the modified Falcor engine. We demonstrate that the design based on shader components features improved modularity and extensibility, while also delivering higher rendering performance than Falcor's original Direct3D 12 implementation.

All efforts described in this paper (the Falcor engine, the componentized Falcor shader library, and the Spire compiler) will be released as open source to the public. For the purposes of submission all code is included as supplemental material.

2 BACKGROUND

Falcor [Benty 2016] is a real-time rendering framework that aims to accelerate and support prototyping of new rendering effects and algorithms. As a vehicle for research and prototyping, Falcor must be modular and extensible to support a wide variety of use cases. At the same time, Falcor must deliver sufficient rendering performance to support state-of-the-art real-time graphics effects. For example, new algorithms for virtual reality (VR) rendering must be deployable at high frame rate to be properly evaluated. The combination of these goals–modularity and performance–makes

Submitted to HPG17, Los Angeles, USA
2017. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

Falcor an ideal testbed for the shader component idea, which offers the promise of helping rendering engines achieve both of those goals.

The Falcor framework includes a layer of convenient abstractions over an underlying graphics API, so that a users need not interact with the low level API except when desired. Falcor also provides a library of shader code, including a flexible implementation of multi-layer physically-based materials. The framework includes many other services, but for this paper we only consider those related to authoring and using shaders.

Falcor was originally developed for OpenGL [Segal and Akeley 2016], where its design relied heavily on the "bindless" texture extension [Bolz and Brown 2014]. The framework is currently undergoing a transition to use Direct3D 12 (D3D12) [Microsoft 2017], with support for Vulkan [Khronos Group, Inc. 2016] planned. These modern graphics APIs impose a different model for working with shader parameters, so that the transition brings new challenges when trying to maintain the goals of modularity and performance.

In this section, we first examine how the original OpenGL-based shading system and engine architecture achieves a mixture of modularity and performance, and make note of key guarantees provided by a shader compiler that enable this architecture. We then discuss the challenges that arise in the transition to D3D12, when these same guarantees are not provided. In Section 3 we discuss how shader components were applied to both improve on the original design, and better address the challenges imposed by modern APIs.

2.1 Shader features are opt-in modules

In many game engines, users might write small amounts of shader code that "plug in" to an existing rendering approach. Researchers typically require greater flexibility and control, so instead a Falcor application is responsible for implementing the main rendering loops and shader entry points. In order to make use of the services of the Falcor shading framework, the application must *opt in* to using the specific modules that it needs.

In order to encapsulate features of the shading system into well-defined modules, the Falcor framework uses an idiom of paired shader (GLSL) and application (C++) types. Figure 1 (left) shows examples of this approach. For example, Falcor's support for multi-layer physically-based materials is encapsulated in a C++ type, `Material`, and a corresponding GLSL type `MaterialData`. The `MaterialData` type encapsulates all the parameter data required by the material module, but the material module does not directly declare any top-level shader parameters; instead, applications use the provided type to declare their own shader parameters.

Figure 1 (right) shows how a user application opts in to using various shader modules in Falcor. In the shader file, an instance of `MaterialData` is declared inside of a `uniform` block, which has the effect of giving this shader a material parameter. Later, the `evalMaterialPattern` function from the material module is invoked, and the `material` shader parameter is passed in. Because each module is encapsulated as a simple `struct` type, it is simple for application code to declare and use exactly the features it needs.

The corresponding host C++ code in Figure 1 (right) is also simple. GLSL shader code is loaded into a `Program`, which manages the compiled program and also allocates constant buffers to store parameter values. The application creates instances of the C++ types that correspond to different modules: `Camera`, `Light`, and `Material` (here there is a material attached to each `Mesh` that was loaded in a scene). The connection between a C++ variable like `cam` (of type `Camera`) and a GLSL shader parameter like `camera` (of type `CameraData`) is then made by calling the `writeParam()` function on the C++ object. This operation copies parameter values from the data stored on the C++ object into a constant buffer associated with a `Program` so that it can be used in subsequent rendering operations.

2.2 Bindless textures enable efficient parameter binding

The simple approach to shader modularity described so far in OpenGL relies on support for *bindless* resources. "Bindless" refers to the ability to store a reference to a texture or other resource in ordinary user-accessible memory, rather than having to use specific API-managed binding slots. Support for bindless resources means that all kinds of shader parameters, including textures, can be declared together in a per-module parameter `struct`, which is then used to declare a parameter inside of a `uniform` block. Support for bindless resources thus enables encapsulation in the shading system; without it a single declaration like `Material material` would need to be replaced by multiple declarations that would depend on the internal details of how materials are represented.

Using bindless resources is also beneficial to the CPU efficiency of shader parameter binding. When all of the parameters of a module are declared in a `struct` type, which is then used inside of constant buffers, the compiler and API provide a fixed and stable layout for the parameter data. For example, the layout of `LightData` in memory is fixed, and independent of any particular shader entry point. Having a fixed layout means that the corresponding C++ class `Light` can directly store parameter data in the appropriate layout and the `writeParam` operation can be as simple as a single `memcpy`.

2.3 Specializing shader code for performance

The Falcor material system is flexible: a material can be composed of many layers, and each layer might have a different reflectance function and parameters. Compared to a one-size-fits-all BRDF, such flexibility can be valuable when researchers want to focus on a particular reflectance model, or compose a complex scenario for testing. In other cases, however, flexibility is not needed, and merely adds a performance cost. A goal of the Falcor shading system is to mitigate these costs in common cases, while still supporting full flexibility when desired; this goal must also be reconciled with the need for modularity.

In real-time rendering, a typical way to avoid the overhead of highly flexible code is to start generate *specialized* variants of an input *uber-shader* (e.g., using the C preprocessor). In order to enable generation of specialized code, while also retaining the ability to run the flexible "generalized" case, Falcor uses what amount to *tagged unions*.

Listing 1 shows a more detailed (but still simplified) view of the `MaterialData` and `MaterialLayerData` types seen in Figure 1. The `brdfType` field in the layer type is an example of a *tag*, and is used to select between Lambertian, GGX, and other reflectance functions.

Engine-defined Modules

Shader Library

Camera

```
struct CameraData {
   vec3 worldPos;
   mat4 viewProjMatrix;
   ...
}
vec4 cameraProj(
   CameraData cam,
   vec3 vertex);
```

Engine Runtime

```
class Camera
{
  void setWorldPos(vec3 p);
  void setViewProjMat(mat4 m);
  ...
  void writeParam(Buffer, str);
};
```

Material

```
struct MaterialLayerData {...};
struct MaterialData {
   MaterialLayerData layers[8];
   ...
}
Pattern evalMaterialPattern(
   MaterialData material,
   vec2 uv, vec3 pos);
```

```
class Material
{
  void setLayer(id, LayerData);
  ...
  void writeParam(Buffer, str);
  void setDefines(Program p);
};
```

Light

```
struct LightData {
   int lightType;
   vec3 worldPos, intensity;
}
vec4 evalLighting(
    Pattern p, LightData l);
```

```
class Light {
  void setLightType(int t);
  void setWorldPos(vec3 p);
  void setIntensity(vec3 i);
  void writeParam(Buffer, str);
};
```

User Application

MainShader.glsl

User-defined shader entry point

```
#include "VertexAttribs.glsl"
layout(binding = 0)
uniform PerFrame {
  CameraData camera;
  LightData light;
};
layout(binding = 1)
uniform PerInstance {
  MaterialData material;
  mat4 worldMat;
}
out vec4 fs_color;

void vs_main() {
  vec3 pos=worldMat*vertPos;
  gl_Position=cameraProj(pos);
}
void fs_main() {
  Pattern pat;
  pat = evalMaterialPattern(
      material, vertUV, pos);
  fs_color = evalLighting(
     pat, light);
 }
```

main.cpp

User-defined host entry point

```
Program prog;
Camera cam;
Light light;
vector<Mesh> meshes;

void main() {
  prog.Load("MainShader.glsl");
  LoadScene();
  while(1) renderFrame();
}
void renderFrame() {
  buf0 = prog.buffer["PerFrame"];
  buf1 = prog.buffer["PerInstance"];
  cam.setWorldPos(vec3(0, 0, 0));
  cam.writeParam(buf0, "camera");
  light.writeParam(buf0, "light");
  for (auto mesh : meshes) {
    setVertexLayout(mesh.vertLayout);
    mesh.material.writeParam(buf1,
      "material");
    mesh.material.setDefines(prog);
    draw(mesh);
  }
}
```

Figure 1: Simplified architecture of the Falcor shading system. Left: the framework provides a number of modules that each comprise both an GLSL `struct` type and a corresponding C++ `class`. Right: an application makes use of those modules by using instances of the corresponding types. Data is explicitly marshaled from C++ to GLSL by calling `writeParam` to fill in a constant buffer.

```
struct MaterialLayerData {
    int brdfType;
    bool isParamConstant;
    vec4 constantParam;
    sampler2D textureParam;
};
struct MaterialData {
    int numLayers;
    MaterialLayerData layers[8];
};
```

Listing 1: The Falcor material system uses a *tagged union* to represent layers. The `brdfType` field selects between different reflectance functions. By substituting in compile-time constant values for `brdfType`, `isParamConstant`, and/or `numLayers`, specialized code can be generated from general-purpose subroutines.

The `isParamConstant` field serves a similar role, selecting which of the following to fields should be used to provide a color parameter (GLSL and HLSL do not support `union` declarations, so storage is declared for both alternatives).

By default, the evaluation of material patterns and light/surface interaction in Falcor is done with a dynamic uber-shader approach. For example, the inner lighting loop performs a `switch` on the value of `brdfType` for each layer to pick the right reflectance function.

When greater performance is required, Falcor provides the ability to specialize a shader using the `setDefines` operation on `Material`, as shown in Figure 1 (right). This operation allows the material in use to specify `#define` values that should be used for compiling a specialized version of the user's shader code. This `#define` values are used to overwrite the tag fields like `brdfType` with compile-time constant values (in the interests of space we will not describe the details of this mechanism). The GLSL compiler in the GPU driver is then relied on to optimize operations such as a `switch` or `if` on a compile-time constant.

Specialization to different materials might result in different program *variants*, comprising different compiled GPU kernels. To deal with this complexity, the Falcor `Program` class does not wrap a single API-level program, but instead maintains a cache that maps a combination of active `#define` flags to a corresponding variant compiled using those flags. In order to make this mapping convenient and transparent, the Falcor framework queries this cache at `draw` time, if any `#defines` have been changed.

The same mechanism that is used to handle specialization of shader code to materials is also used to specialize shader code for meshes with different vertex attributes. The Falcor framework automatically injects `#defines` like `HAS_NORMAL` based on the currently bound vertex attributes, and an applications shader code can use this definition to adapt itself. Most Falcor applications need not be aware of this detail, because they opt in to a default set of vertex attributes by including a Falcor-provided GLSL header.

2.4 Guarantees provided by the shader compiler

The approach described so far achieves good modularity and performance, but relies on some very specific guarantees provided by the underlying shader compiler:

- *The parameters for a module can be encapsulated in a single named type.* This allows shaders to opt in to using a module by declaring a shader parameter of the associated type.

- *That type has a simple and contiguous layout in memory.* This allows parameter data for a module to be stored and bound efficiently.
- *That layout is independent of how a module is used in a particular program variant.* This allows parameter data to be prepared in a GPU-ready format and then re-used across multiple programs and variants.
- *No global (or external) knowledge is required to secure the above guarantees.* The user just writes ordinary shader code, and the framework extracts the information it needs via reflection.

These are not especially strong guarantees; a C compiler can provide the first three with a `struct` declaration.

2.5 Challenges in porting to modern graphics APIs

So far, our description of the Falcor shading system has focused on how it was embodied in OpenGL and GLSL. The current version of Falcor represents an effort to port the same architecture to modern graphics APIs, starting with D3D12. The primary goal in moving to new APIs is to better align rendering researchers with future efforts in production game engines. A secondary goal is to exploit the increased efficiencies promised by these new APIs, including support for more efficient binding of shader parameters through *parameter blocks* (called "descriptor tables" in D3D12).

The D3D12 porting effort has met challenges that threaten the simplicity of Falcor's existing approach to modularity, efficient parameter binding, and specialization. Based on our investigation of the Falcor code base and its evolution, we believe that these challenges stem from the fact that the standard shader compilers for modern APIs do not provide the guarantees that the Falcor design had relied upon in OpenGL.

Module parameters can be encapsulated in a single type. Encapsulating a module as a shader-side `struct` type is supported in the HLSL language for D3D11 and later (but does not work in current versions of GLSL for Vulkan). Because D3D does not support bindless resources, the HLSL compiler automatically splits a `struct` type that mixes resource types (e.g., textures) and simple data types (e.g., vectors) into distinct API-visible parameters.

Layout of a module type in memory is not simple. Because the HLSL compiler may split a `struct` into different API-level parameters based on field types, such a structure cannot be usefully shared between shader and host code via a common `#include` file, and parameter data cannot be transferred with a simple block copy. For example, a `Material` can no longer hold a value of type `MaterialData` that is properly formatted for GPU consumption, since the latter type contains a mixture of resources and ordinary data fields.

To work around this issue, modules in the Falcor shader system declare their parameters as multiple types that are already split in the way the API requires. Manual splitting ensures that the ordinary data parameters can be bound with a block transfer. Maintaining split types is inconvenient, although most of the complexity affects reusable library code, and not shaders written by end users.

The layout of a module type is specific to a program variant. The HLSL compiler provides no particular guarantees about how resources are assigned to logical binding slots (HLSL calls these "registers"). The compiler is free to omit resources it determines are unused by the shader, and the modern HLSL compiler does this aggressively. Resources that are omitted do not get assigned slots, and do not even appear in the reflection data output by the compiler. The code paths that are enabled or disabled in a given program variant may result in different numbers of resources being used, and so the assignment of resources to slots is, in general, specific to a variant.

Because the layout for resource parameters is specific to a program variant, it is difficult for a framework like Falcor to exploit efficient parameter binding APIs. A parameter block that is allocated based on reflection information from one program variant might not be applicable to another variant of the same program, even if the two variants arose from changing only a single `#define`. Even if suitable parameter blocks could be allocated, each program variant would potentially require a different shader parameter binding signature (called a "root signature" in D3D12), and performance guidelines discourage frequent switching of root signatures.

These challenges have resulted in significant compromises in Falcor. First, most uses of shader specialization (including specialization to materials) have been disabled to avoid parameter signature changes. Second, the host-side classes for modules like materials do not store their parameter data in GPU-ready parameter blocks.

The conventional fix requires global knowledge. The conventional approach for resolving the above issues in modern APIs is to use manual annotations that assign resources to binding slots (e.g., `register` qualifiers in HLSL). Indeed, the Vulkan API *requires* manual annotations for a shader to exploit multiple parameter blocks.

Unfortunately, explicit binding annotations are not compatible with modular shader software development, nor rapid exploration and experimentation. In order to manually assign binding slots to shader parameters, a user must have complete knowledge of how many slots (of each kind) every shader parameter requires, defeating the purpose of encapsulation into modules. Furthermore, adding or removing resource parameters, whether in user or framework code, can require cascading edits to shift other declarations around.

The benefits of the Falcor shading system–simplicity, modularity, and performance–have been challenged by the transition to modern APIs. We have argued that one source of these challenges is a shader compiler that does not provide suitable guarantees about the layout of parameter data.

3 ADOPTING SHADER COMPONENTS

Recently, He et al. [2017] introduced the idea of shader components as a mechanism for organizing libraries of shader code, and efficiently managing the graphics pipeline state associated with these shaders. Shader components build on approaches to shader code modularity in many prior systems [Foley and Hanrahan 2011; Mark et al. 2003; Microsoft 2011; Pharr 2004; Proudfoot et al. 2001], to provide a first-class modularity construct in shader code. In the interests of brevity we direct readers to [He et al. 2017] for information on the design and implementation of shader components

```
class Camera {
public:
  ComponentInstance* compInst;
  void setPos(vec3 point) {
    compInst->setVariable("camPos", point);
  }
  void setViewProjMat(mat4 m) {
    compInst->setVariable("viewProjMat", m);
  }
};
```

Listing 2: Definition of the revised `Camera` class. The revised host class has the same interface as the original definition (Fig. 1), but now holds a *component instance* of the corresponding shader component class.

```
template shader MainPass <
  vertexFormat : IVertexFormat,
  camera : Camera,
  transform : ModelTransform,
  material : IMaterial,
  lighting: ILighting>
{
  using vertexFormat;
  using camera;
  using transform;
  using material;
  using lighting;
  vec3 worldPos = (worldMat * vertPos).xyz;
  @Vertex vec4 projPos = cameraProj(worldPos);
  out @Fragment fs_out = lighting.lit(worldPos);
}
```

Listing 3: Spire definition of the shader entry point in Fig. 1. The user explicitly opts in to the shading logic of a component with a `using` statement, and then uses values and functions exported by those components to compute its outputs.

and a discussion of related work. (A copy of this recent prior work has been included as supplemental material.)

In this section, we describe how we utilize shader components, as implemented in the Spire shading language [He et al. 2016, 2017], to streamline the Falcor engine's design and address the challenges of efficiently mapping the engine to D3D12. Our redesign of Falcor uses the creation of and use of shader components *as a single mechanism* to achieve a variety of benefits, including: efficient shader parameter binding, performant shader variant selection, specialization of shaders, and ensuring code modularity that facilitates simple extension of a shader library with new features, such as tessellation. We first provide an overview of the modified Falcor rendering system, and then discuss the benefits the new design in each of the following subsections.

3.1 System Overview

We revised the D3D12 branch of Falcor framework following the shader component concept and improved the engine's performance and modularity.

Fig. 2 summarizes the revised architecture of the renderer. We found that many existing modules in the shading system could map one-to-one to shader components, and we created three shader components to represent these features: `Camera`, `Material` and `Lighting`. In addition, we identified two concepts in the engine code which had not previously been thought of as shading features, but which either affect shader code generation or encapsulate shader parameters, and turned them into components: vertex format definitions for meshes (`VertexFormat`) and object- to world-space transformation logic (`ModelTransform`).

In order to turn a concept like `Camera` into a shader component, we first refactor the existing shader code for the feature into a Spire *component class* definition. For example, in Fig. 2 (top left), the `Camera` component class encapsulates the shader library code of the camera module in Fig. 1. Next, we modify the framework's host code to create an *instance* of that class. A component instance is a host-side object, that stores concrete values for the parameters declared in its class, using a parameter block in the underlying graphics API. Fig. 2 (top right) shows 6 instances; each corresponds to a parameter block holding the component's input parameters.

As described in section 2, Falcor defines a C++ class for each shading module. For example, the `Camera` class is provided for user to pass parameters of a camera object to the shader. In the revised engine, component instance creation is handled by these C++ classes. For example, Listing 2 shows the revised `Camera` class in host code. It provides the same interface as the old class (as in Fig. 1) that sets camera parameters. In addition, it hold a component instance created from the `Camera` component class; setter functions like `setPos` are implemented to write parameter data directly into the component instance's parameter block. A user of the framework can think of the shader-side component class `Camera` and the host-side C++ class `Camera` as representing the same logical type.

Similar to architecture in Fig. 1, the user writes a shader entry point for each render pass, and opts in to features from the shader library. In the revised architecture, this is expressed very directly by writing a Spire shader entry point that is parameterized on components. Such an entry point can be thought of as a shader "template" with holes where specific component instances can be plugged in. For example, the `MainPass` entry point in Fig. 2 has five *template parameters*. Each template parameter drives a different a different aspect of shader logic: the vertex format, camera projection, surface material, world space transformation, and lighting.

Listing 3 shows the `MainPass` entry point in detail. The entry point explicitly invokes the shading logic of various components with `using` statements. Each component may export named values and functions, and the entry point uses these exported definitions to compute both a per-vertex projected position for the rasterizer, `projPos`, and a per-fragment value that will be output to a framebuffer target, `fs_out`.

Because a Spire entry point acts like a template, the user need not specify the concrete class of every component to be used. For example, the entry point shader shown in Listing 3 can be used with different vertex format components (e.g., `vf1` or `vf2` in Fig. 2), so long as they satisfy the `IVertexFormat` interface. Different combinations of component instances can in general result in different code being generated for an entry point.

The existing Falcor `Program` class maps one-to-one to the notion of an entry point in Spire. Each represents shading logic that spans all of the programmable GPU pipeline stages, that has user-declared

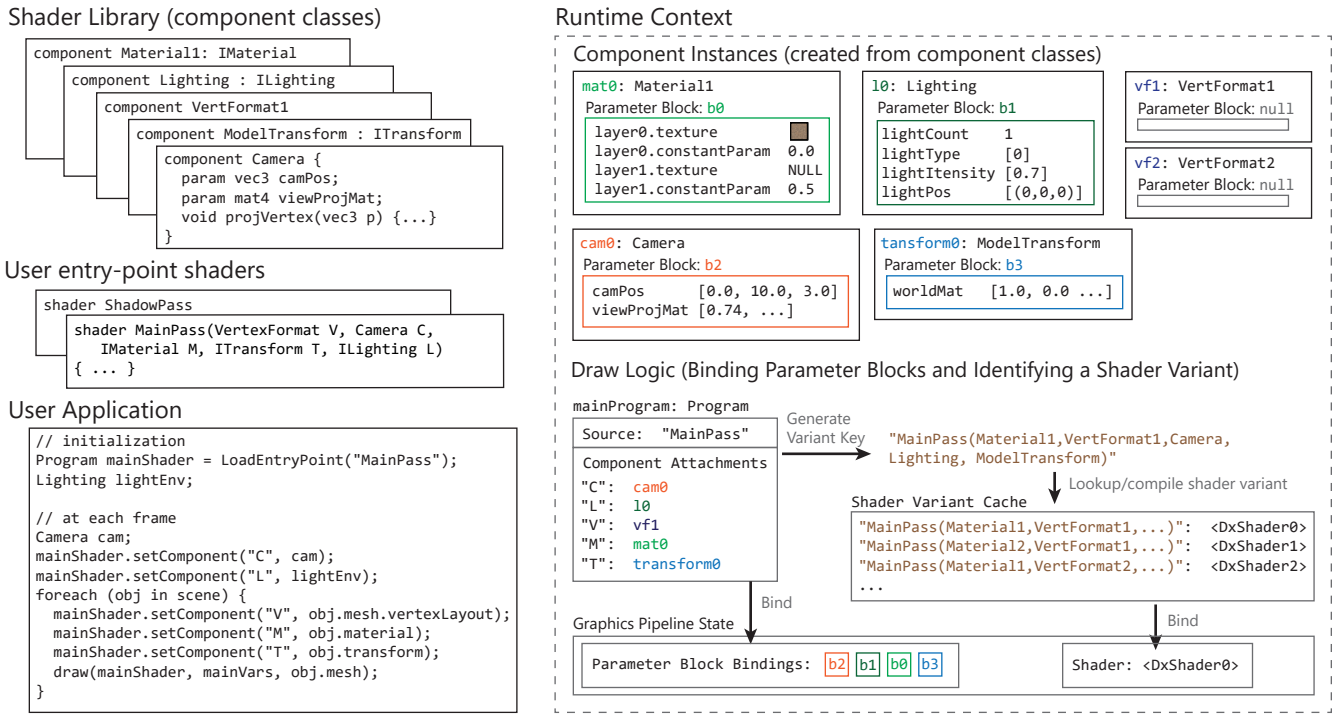

Figure 2: The overall architecture of the revised engine. A shader library is organized as a set of component classes. The user defines shader entry points, parameterized on components. The host application creates component instances, from component classes, to store shader parameter data. An entry point is loaded as a `Program` object, and component instances are attached to configure its logic and parameters. The entry point and instances together form a key used to determine the GPU shader kernels to execute. Keys are depicted as strings for simplicity; the implementation uses a more efficient representation.

parameters corresponding to modules in the shader library, and that can be specialized to produce different executable kernels based on how those modules are configured.

When submitting rendering work, the user application is responsible for selecting an entry point and set of component instances to use for its parameters. Fig. 2, shows how a user application can load a shader entry point by name to yield a Falcor `Program` object, which associates the entry point with a set of component instances attached to its template parameters. The user attaches component instances to template parameters by name `setComponent`, serving much the same role as the `writeParam` calls in Fig. 1. When it is time to issue a draw call, the chosen combination of the entry point and components, managed by a `Program`, are provided to the Falcor framework.

In order to configure the GPU for rendering, the framework must both determine the final compiled shader kernels to use, and bind suitable parameter blocks that will provide parameter data to those kernels. With shader components, these two tasks are unified. A shader component instance contains two pieces of information: the component class of the instance (which represents shader code), and a parameter block that holds the values for the input parameters of that class. Once the application has selected component for use with an entry point, the revised architecture uses the classes of the components to select, or generate, an appropriate generated variant of the entry point. New variants are generated by invoking the Spire compiler API, instructing it to replace template parameters with particular concrete component classes. The parameter blocks of the instances can then be bound to provide parameter data in the format expected by the generated code.

Because Falcor is a prototyping framework, generating new compiled variants on demand at runtime is acceptable, but compilation is expensive enough that the results need to be cached and reused. The revised architecture in Fig. 2 uses a shader variant cache that is indexed by a key that includes the entry point and the classes of the components selected for use. This cache is queried before issuing a draw call, if any changes have been made to the classes of components bound to the `Program` in use.

The following subsections describe the benefits of this system design in more detail.

3.2 Efficient Use of Parameter Blocks

We achieved high performance shader parameter binding in our revised Falcor D3D12 implementation. Many shader components are used for rendering more frequently than their parameter data is changed; for example, the texture and color parameters of a surface material usually do not change at runtime. Using shader components, we leverage observation to improve parameter-binding performance in two ways. First, because each component instance stores parameter data in a GPU-friendly parameter block, we reduce driver overhead when switching components; we only need

to invoke the graphics API to bind a pre-generated parameter block. Second, we only allocate and populate parameter blocks as needed, when parameter values change. In contrast, some engine designs allocate and populate parameter blocks on the fly every frame, even when parameter data does not change. Our design choice means that once a `Material` component instance is created, it can be used across multiple frames, by multiple entry points, with minimal overhead.

The ability to associate a material, or other component, with a reusable parameter block depends on guarantees provided by the Spire compiler. The compiler guarantees that the layout for parameter data depends only on the component class, and not how it is used in any particular entry point. Furthermore, this layout can be queried using the Spire API, even before any entry points have been loaded or compiled.

We observe that Falcor's use of OpenGL uniform buffers and the bindless texture extension closely resembled the concept of parameter blocks in modern APIs. They are both used to represent a collection of shader parameters, including both simple values and resources, prepared in a GPU-consumable form and bound to the pipeline with a single operation. The original effort to adapt Falcor to D3D12 followed more conventional thinking, which instead identifies OpenGL uniform buffers with D3D constant buffers.

Given the understanding we have developed, we might hope that Falcor's HLSL shader library could be organized to group shader parameters into parameter blocks. However, the concept of parameter blocks in D3D exists only in the host-side API and does not have a corresponding construct in the shading language. All HLSL shader parameters are declared at global scope, and grouping can only be achieved by means of manual `register` annotations. As discussed in Section 2, such manual annotation conflicts with modular software development. The lack of first-class support for parameter blocks in HLSL hinders the ability of modular engines to efficiently exploit parameter blocks.

3.3 Streamlined and Efficient Shader Variant Selection

In the redesigned engine, shader variant selection is not only significantly simpler, but also much faster.

In the original Falcor design (Fig. 1), a client of a module must call either `writeParam` to bind shader parameter data, `setDefines` to drive variant selection, or both, depending on the subsystem. In some cases, variant selection is influenced behind the user's back: e.g., an operation like `setVertexLayout` appears to only affect an isolated piece of GPU state, but also silently introduces some `#defines` that may select a different shader variant.

In contrast, shader components allow the framework to present a single mechanism instead of two, and make explicit some operations that were formerly obscured. Because a shader component defines both shader code and its parameters, attaching a component instance to a codeProgram with `setComponent` achieves both parameter binding and shader code selection. This subsumes the behavior of both `writeParam` and `setDefines`.

In addition, the revised design represents mesh vertex formats as shader components, so that selecting a vertex format (which may require a suitably specialized variant of an entry point) corresponds to the same explicit mechanism used for binding, e.g., a material.

Because the selection of shader variant kernels is driven by a single mechanism, we can devise an efficient strategy for lookup and caching. In our revised Falcor implementation, each `Program` maintains its own cache of shader variants. The keys for this cache are flat arrays, where each element identifies the component class used for a particular template parameter; currently this is an array of pointers to component class metadata. More efficient indexing schemes are possible, but even this simple choice ensures that that variant lookup is not a bottleneck.

In contrast, the original Falcor approach must perform a similar lookup step, but because variant generation is driven by preprocessor macros, the lookup key encodes a list names and values of active `#defines`. Generating and using this more complex key representation in tight draw loops adds overhead, which we greatly reduce in the transition to shader components.

3.4 Material Specialization Becomes Simple

Because shader components unify variant selection to be driven entirely by component classes, we can easily support performance-oriented specialization by simply generating specialized component classes. For example, the `material` template parameter of the `MainPass` entry point in Listing 3 can be used with various material component classes, so long as they support the `IMaterial` interface. Therefore, to implement specialization of material shader code, we only need to generate a new material component class and attach an instance of that class to the `material` parameter.

To simplify the process of generating specialized material component classes, we extracted the shader logic for a single material layer into a separate `MaterialLayer` component. Listing 4 shows the definition of the `MaterialLayer` component, and an example two-layer material generated by composing two `MaterialLayer` sub-components. A `MaterialLayer` component defines the shader parameters for the layer (BRDF type and texture or constant parameter, similar to `MatData` in Fig. 1), and a function to evaluate the layer's BRDF. A material component can then be composed from one or many `MaterialLayer` sub-components. In Listing 4, the material component instantiates two `MaterialLayer` sub-components and calls each layers' `evalBRDF` method. In this case, the result is a two-layer material, where the first layer uses a Lambertian BRDF for diffuse response, and the second layer uses phong for specular response.

In our modified Falcor implementation, the C++ `Material` class has support for generating specialized Spire source code akin to `TwoLayerMaterial` in Listing 4; it is straightforward to generate code for a material with n layers. Because the `brdfType` parameters of the layers are specified using compile-time constants, standard optimizations are able to specialize the body of the per-layer `evalBRDF` methods.

When shader components are used as sub-components (e.g., the materials in this example), their parameters are grouped into the same parameter block as the top-level component that uses them. In this way, each material can still use a single efficient parameter block, although different `Material` instances may use differently-sized blocks, based on their number of layers.

```
component MaterialLayer {
  param Texture2D textureParam;
  param vec4 constantParam;
  require int brdfType;
  require bool useTextureParam;
  require vec2 vertUV;
  vec4 paramValue = useTextureParam ?
    textureParam.Sample(vertUV) : constantParam;
  vec4 evalBRDF(vec3 P, vec3 L, vec3 N, vec3 V)
    { ... }
}

// engine generated component
component TwoLayerMaterial{
    using level0 = MaterialLayer(
        brdfType:Lambert, useTextureParam: true);
    using level1 = MaterialLayer(
        brdfType:Phong, useTextureParam: false);
    vec4 evalMaterial(vec3 P, vec3 L,
                      vec3 N, vec3 V) {
        return level0.evalBRDF(P,L,N,V)+
               level1.evalBRDF(P,L,N,V);
    }
}
```

Listing 4: Definition of the `MaterialLayer` component, and a two-layer material component generated by composing two instances of `MaterialLayer` sub-component.

Our implementation of material specialization is internal to the `Material` class: we simply change the initialization logic to create a component instance of a specialized component class instead of the general material component class defined in the shader library. Generation of specialized code occurs only when a new material is created, which is usually at scene load time. No other engine subsystems were changed in this process; we view this as an indication of good modularity.

3.5 Adding Tessellation without Changing Existing Shader Code

The modular design of the revised Falcor shader library makes it easy to add new features. To demonstrate this, we added support for PN-Triangles tessellation [Vlachos et al. 2001] to Falcor. Tessellation is normally considered a difficult feature to integrate into an engine because it cross-cuts multiple graphics pipeline stages, and can thus require modifying many different pieces of engine and shader code. While the Falcor framework includes support for compiling and loading shaders that target the tessellation pipeline stages, there was no existing code in the shader library designed to exploit or interoperate with tessellation.

Spire builds on the multi-rate shading language design of Spark [Foley and Hanrahan 2011], by enabling a shader component to define shader logic that spans all shading stages. Introducing logic that execute on tessellation stages require no modifications to logic that runs in vertex or fragment shading stages. Additional inter-stage dataflow or interpolation logic is inserted by the compiler.

We observed that tessellation has the same logical role as our `VertexLayout` component: it defines vertex positions used to produce the shader's geometry output. Following this observation, we added a new `PNTessellation` component that can be used as an alternative to existing `VertexLayout` components when a mesh is intended to be used with tessellation. The only code we changed during this process is adding the `PNTessellation` component in the shader library, and change the C++ `Mesh` implementation to create a `PNTessellation` component when tessellation is enabled on the mesh. None of existing shader library code or user code need to change with this extension.

4 EVALUATION

We have integrated shader components into Falcor as described in previous section. We also ported all major features of the Falcor shader library to Spire shader components, including:

- Diffuse, specular and emissive material layers
- Material layer blending
- Full suite of BRDFs: GGX and Beckmann normal distribution, conductor/dielectric Fresnel
- Normal mapping
- Normal filtering for shading anti-aliasing [Kaplanyan et al. 2016]
- Dynamic lighting from point, spot and directional lights
- Alpha testing

In addition, our shader library includes an implementation of PN triangles tessellation.

Our revised Falcor engine has a simpler and more modular design, achieves better CPU and GPU performance, and produces the same visual results as the original Falcor engine.

4.1 Modularity and Clarity

Besides the modularity benefits discussed in Section 3, our revised shader library code becomes simpler and shorter. The original shader library contains over 2,500 lines of HLSL code, while our revised shader library has 1,700 lines of code (including the PN-triangles tessellation algorithm).

The reduction in code size is partly due to elimination of parameter-passing boilerplate where modules interact, and partly due to the fact that Falcor's original shader library used preprocessor conditionals to abstract over differences between HLSL and GLSL (to allow a single library to be used with multiple APIs). Our ported shader library instead relies on the Spire compiler to perform cross-compilation to HLSL or GLSL as needed.

The Spire-based shader library does not represent a complete rewrite of the original code, and in fact much of the shader library code was not modified at all (Spire accepts a large subset of existing HLSL and GLSL code). Most of our changes involved refactoring library code for each module into corresponding components, while leaf code (e.g., implementations of different BRDFs, evaluating material layers or lighting computations) is typically only slightly modified.

We find that the refactored shader code becomes more readable because shader components impose an explicit structure to the shader library: each high level shading feature now maps to a self-contained shader component that fully defines both the shader parameter and the computation logic for the feature. This structure helps establishing a logical hierarchy of the library code.

Scene	ORIGINAL		COMP-GEN		COMP-SPEC	
	CPU	GPU	CPU	GPU	CPU	GPU
Restaurant	1.09	8.28	0.78	5.21	0.81	2.97
Classroom	0.64	0.83	0.44	0.79	0.47	0.53
Sponza	0.24	0.64	0.16	0.55	0.18	0.27
San-Miguel	2.53	2.58	1.70	2.18	1.86	1.83

Table 1: CPU and GPU frame times (in ms) before and after re-architecting Falcor engine with shader components. COMP-SPEC: performance of revised Falcor engine using specialized material components. COMP-GEN: performance of revised Falcor engine using general material component. ORIGINAL: performance of the original Falcor engine.

Figure 3: Scenes used in performance benchmark.

4.2 Performance

To evaluate the performance improvement of our revised engine, we created a Falcor-based application that renders scenes containing meshes with different materials under dynamic lights. Because shader components makes it easy to use specialized shaders for different materials, we also implemented material specialization in the engine. We evaluate the performance of the revised engine that uses shader components, both with material specialization (COMP-SPEC) and without material specialization (COMP-GEN), to the performance of the original Falcor branch (ORIGINAL) that does not use shader components, when rendering the same set of test scenes.

Table 1 shows the average CPU and GPU time required per frame for all five test scenes. Measurements are performed on a machine with an Intel i7-5820K CPU and a NVIDIA GeForce GTX 980Ti GPU. All scenes are rendered at 1920×1080 resolution. Renderings of the benchmark views of each scene are shown in Fig. 3.

As expected, Falcor using shader components incurs lower CPU cost due to efficient use of parameter blocks and high-performance shader variant lookup. The CPU performance benefit is proportional to scene complexity. San-Miguel has the largest number of draw calls and exhibits 1.4× speed-up in CPU time. Sponza generates fewer draw calls and the CPU performance benefit of the shader-component-based Falcor implementations is less significant.

When using unspecialized materials, the GPU time in incurred by COMP-GEN is on par with that of the original Falcor branch. (Spire-generated shaders perform similarly to their original HLSL counterparts.) However, material specialization in COMP-SPEC yields 1.5×-2× lower GPU cost because specialized shaders are simpler and feature no dynamic branching.

We also observed that the shader-component-based Falcor implementation provides at least a 5× speedup in loading time for all scenes. (This time involves compiling all shader variants.) Profiling reveals that the original Falcor implementation used a custom preprocessor to generate shader variants. This preprocessor is not well-optimized and takes significant amount of time to emit preprocessed code. In our revised engine, all shader parsing and compilation work is done by the Spire compiler. While it is certainly possible to improve the original Falcor implementation, we view this speedup as additional evidence that a properly implemented shader compilation service that meets an engine's modularity needs is valuable, especially when rendering engine teams do not have dedicated engineering resources to develop high-quality compiler tools.

5 DISCUSSION AND CONCLUSION

An important motivation for this work was to validate the shader components concept: could the benefits of shader components be applied to an existing renderer like Falcor, that was not built with components in mind? Our evaluation demonstrates that we have achieved success, by integrating Spire shader components into Falcor and bringing improvements to performance, modularity, and extensibility.

One question that we have not directly evaluated in this work is how much of the benefit of the shader component mental model can be achieved without a component-aware shader compiler like Spire. While our focus was on the Direct3D 12 branch of Falcor, our retrospective look at the OpenGL implementation of the framework revealed a design that had many commonalities with shader components as implemented in Spire. That implementation relied on a particular set of guarantees (Section 2.4), which are not difficult for a shader compiler to provide.

If the default shader compilers in modern APIs were improved to provide those guarantees, they could enable more efficient resource binding for shaders written in a modular fashion. However, other benefits we discuss, such as unifying shader parameter and variant switching, and allowing tessellation to be a drop-in module, are specific to more advanced languages like Spire.

It may seem surprising that the existing modularity approach in Falcor turns out to have such a direct mapping to shader components. However, we note that shader components were designed based on observations of how production game engines structure their rendering code for modularity. The Falcor framework shares many goals with these production engines, albeit with a greater focus on flexibility for rapid prototyping, and follows similar structures.

Our experience integrating Spire shader components into the Falcor framework gives us hope that similar benefits could be achieved by production rendering engines. We hope that the case study we have presented here can provide guidance to game engine developers who consider adopting Spire's approach to shader components in their architectures.

REFERENCES

Nir Benty. 2016. The Falcor Rendering Framework. (08 2016). https://github.com/NVIDIA/Falcor https://github.com/NVIDIA/Falcor.

Jeff Bolz and Pat Brown. 2014. *NV_bindless_texture OpenGL Extension.* https://www.khronos.org/registry/OpenGL/extensions/NV/NV_bindless_texture.txt.

Epic Games. 2015. Unreal Engine 4 documentation. Available at http://docs.unrealengine.com. (2015).

Tim Foley and Pat Hanrahan. 2011. Spark: modular, composable shaders for graphics hardware. *ACM Trans. Graph.* 30, 4, Article 107 (July 2011), 12 pages.

Yong He, Tim Foley, and Kayvon Fatahalian. 2016. A System for Rapid Exploration of Shader Optimization Choices. *ACM Trans. Graph.* 35, 4, Article 112 (July 2016), 12 pages.

Yong He, Tim Foley, Teguh Hofstee, Haomin Long, and Kayvon Fatahalian. 2017. Shader Components: Modular and High Performance Shader Development. *ACM Trans. Graph.* 36, 4, Article 1 (July 2017), 11 pages.

A. S. Kaplanyan, S. Hill, A. Patney, and A. Lefohn. 2016. Filtering Distributions of Normals for Shading Antialiasing. In *Proceedings of High Performance Graphics (HPG '16)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 151–162. DOI: https://doi.org/10.2312/hpg.20161201

Khronos Group, Inc. 2016. *Vulkan 1.0.38 Specification.* Khronos Group, Inc.

William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. 2003. Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Trans. Graph.* 22, 3 (July 2003), 896–907.

John McDonald. 2016. High Performance Vulkan: Lessons Learned from Source 2. In *GPU Technology Conference 2016 (GTC)*. Available at http://on-demand.gputechconf.com/gtc/2016/events/day/High_Performance_Vulkan.pdf.

Microsoft. 2011. Interfaces and Classes. Available at https://msdn.microsoft.com/en-us/library/windows/desktop/ff471421.aspx. (2011).

Microsoft. 2017. Direct3D 12 Programming Guide. Available at https://msdn.microsoft.com/en-us/library/windows/desktop/dn899121(v=vs.85).aspx. (2017).

Matt Pharr. 2004. An Introduction to Shader Interfaces. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, Randima Fernando (Ed.). Pearson Higher Education.

Aras Pranckevičius. 2015. Porting Unity to new APIs. In *SIGGRAPH 2015 Course Notes: An Overview of Next-generation Graphics APIs.* DOI: https://doi.org/10.1145/2776880.2787704 Available at http://nextgenapis.realtimerendering.com/presentations/7_Pranckevicius_Unity.pptx.

Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of SIGGRAPH 01, Annual Conference Series.* ACM, New York, NY, USA, 159–170.

Mark Segal and Kurt Akeley. 2016. *The OpenGL® Graphics System: A Specification.* https://www.khronos.org/registry/OpenGL/specs/gl/glspec45.core.pdf.

Unity Technologies. 2017. Unity 5.6 Users Manual. Available at https://docs.unity3d.com/. (2017).

Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L. Mitchell. 2001. Curved PN Triangles. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics (I3D '01)*. ACM, New York, NY, USA, 159–166. DOI: https://doi.org/10.1145/364338.364387