Shader Components: Modular and High Performance Shader Development

YONG HE, Carnegie Mellon University
TIM FOLEY, NVIDIA
TEGUH HOFSTEE, Carnegie Mellon University
HAOMIN LONG, Tsinghua University
KAYVON FATAHALIAN, Carnegie Mellon University

Modern game engines seek to balance the conflicting goals of high rendering performance and productive software development. To improve CPU performance, the most recent generation of real-time graphics APIs provide new primitives for performing efficient batch updates to shader parameters. However, modern game engines featuring large shader codebases have struggled to take advantage of these benefits. The problem is that even though shader parameters can be organized into efficient modules bound to the pipeline at various frequencies, modern shading languages lack corresponding primitives to organize shader logic (requiring these parameters) into modules as well. The result is that complex shaders are typically compiled to use a monolithic block of parameters, defeating the design, and performance benefits, of the new parameter binding API. In this paper we propose to resolve this mismatch by introducing *shader components*, a first-class unit of modularity in a shader program that encapsulates a unit of shader logic and the parameters that must be bound when that logic is in use. We show that by building sophisticated shaders out of components, we can retain essential aspects of performance (static specialization of the shader logic in use and efficient update of parameters at component granularity) while maintaining the modular shader code structure that is desirable in today's high-end game engines.

CCS Concepts: • **Computing methodologies** → *Graphics systems and interfaces*;

Additional Key Words and Phrases: shading languages, real-time rendering

ACM Reference format:
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.
DOI: http://dx.doi.org/10.1145/3072959.3073648

1 INTRODUCTION

A modern game engine must achieve high performance to render detailed scenes with complex materials and lighting. At the same time, high productivity is required when authoring and maintaining the large libraries of shader code that define materials and lighting in these scenes. Today, a major challenge limiting the performance of real-time rendering systems is the inability for the CPU (even a multi-core CPU) to supply the GPU with rendering commands at a sufficient rate. Many of these commands pertain to binding shader parameters (textures, buffers, etc.) to the graphics pipeline. This problem is sufficiently acute that a new generation of real-time graphics APIs [Khronos Group, Inc. 2016; Microsoft 2017] has been designed to provide new primitives for performing efficient batch updates to shader parameters. However, modern game engines have been unable to fully take advantage of this functionality.

The problem is that even though shader parameters can now be organized into efficient modules and bound to the pipeline only at the necessary frequencies, modern shading languages lack corresponding primitives to organize shader logic (that requires these parameters) into modules as well. As a result, complex shaders are compiled into monolithic programs that access a monolithic block of parameters. These parameters are typically updated en masse by engines, defeating the design, and performance benefits, of the new parameter binding API.

In this paper we propose a solution to this mismatch by introducing *shader components*, a first-class unit of modularity in a shader program that encapsulates both the shader logic of a feature and the parameters that must be bound when that feature is in use. From the perspective of a shader writer, shader components provide a mechanism that can be used to organize large shader codebases. From the perspective of a game engine, shader components provide both the logical units (e.g., a specific material, an animation effect, and a lighting model) used to assemble shading features into complete shaders, and also the granularity at which blocks of shader parameters (needed as inputs to these features) are bound to the graphics pipeline. By supporting shader components directly in a shading language, our system is able to provide services such as static checking of shader components against interfaces, specialization of complete shaders to a specific set of components (features) in use, and generation of shader parameter block interfaces that facilitate efficient binding. These services allow a well-written engine to retain the benefits of modular shader software development, while also realizing the low CPU overhead and high rendering performance promised by modern graphics APIs.

Our specific contributions include:

- The design of the shader component concept and its implementation in a shader compiler library. This implementation includes a shading language front-end and host side APIs that engines use to assemble components into complete shaders and to allocate and bind blocks of shader parameters.

0730-0301/2017/7-ART1 $15.00
DOI: http://dx.doi.org/10.1145/3072959.3073648

- A reference implementation of a large library of shader components and a Vulkan-based rendering engine that achieves high rendering performance when using shaders assembled from these components. Specifically, we demonstrate that our design simultaneously achieves modular shader authoring and efficient shader parameter binding, yielding over 2× lower CPU cost than a system implemented using current AAA engine best practices.

2 BACKGROUND

2.1 Organizing a Renderer for Performance

In order to achieve high performance, a renderer must minimize changes to GPU *state*. This state comprises the configuration of fixed-function pipeline stages, compiled shader *kernels* to be used by programmable stages, and a set of *shader parameter bindings* that provide argument values to kernel parameters.

Changing parts of the GPU state has costs, in both CPU and GPU time. CPU costs include time spent by the application and driver to prepare and send hardware commands. GPU costs include possible hardware pipeline stalls waiting for work using an old state to complete. Both kinds of costs can negatively impact frame rates, depending on whether an application is CPU- or GPU-bound. Reducing CPU costs during rendering frees CPU resources for other key engine tasks such as AI, animation, audio, or input processing. Every millisecond counts when, e.g., only 11ms are available per frame in VR.

In order to reduce state changes, engines often organize rendering work around different rates of change. For example, a rendering pass can be thought of in terms of a set of nested loops, e.g.:

```
for(v in view)
    for(m in materials)
        for(o in objects)
            draw(v, m, o);
```

The key observation is that some state (e.g., per-object transformations) changes at a higher rate than others (e.g., view parameters).

In order to maximize CPU efficiency when changing GPU state, graphics APIs provide mechanisms to group state into objects that can be updated en masse. For example, Direct3D 10 [Blythe 2006] organized fixed-function state into coarse-grained objects. Modern APIs such as Vulkan and Direct3D 12 allow shader parameter binding state to be grouped into modules, referred to as descriptor sets and descriptor tables respectively. For clarity, we refer to such API objects as *shader parameter blocks*. Multiple parameter blocks can be bound to the GPU pipeline, and accessed by shaders, at one time.

Grouping parameter binding state into blocks has two main benefits. First, the bulk nature of parameter blocks means that a single API call can change a large portion of the GPU state; e.g., all of the per-material parameter bindings. Second, because multiple parameter blocks can be used at once, these changes can be organized by expected frequency of update. For example, by grouping parameter blocks according to the loop nesting illustrated above, an engine can save work by ensuring that only a small amount of (per-object) state needs to be changed in the inner rendering loop.

While modern APIs allow shader parameters to be defined using multiple parameter blocks, other aspects of GPU state are set using a monolithic *pipeline state object* (PSO). A PSO comprises state for fixed-function stages, and a set of shader kernels to be used. We refer to each unique combination of kernels as a shader *variant*. This paper will ignore fixed-function state and focus only on generating and selecting shader variants; statements made about the handling of variants can be extended, without loss of generality, to PSOs.

2.2 Composing Shaders from Modules

Consider the nested loops from the previous section. The right shader variant to use in the inner loop might depend on contributions from many sources. The object might contribute animation, tessellation, or other geometric effects. The material might contribute logic to fetch and composite texture layers. The render pass itself might contribute logic to evaluate light sources, write to a G-buffer, etc. Figure 1 depicts one way of composing a shader variant from these different concerns.

For an engine featuring a large library of shaders and effects it is desirable that disparate concerns or features can be defined as composable *modules*. For example, any given surface shader should work with any given light, and vice versa. Different modules are illustrated as different boxes at the left in Figure 1, grouped by concern.

In other domains, it is common to compose separately-developed modules dynamically at runtime, by using a level of indirection (e.g., using function pointers). However, in the context of real-time shaders, the cost of runtime indirection is usually prohibitive, and almost all shader composition is achieved by static *specialization*.

As illustrated in Figure 1, a typical way for an engine to perform specialization is to use the C preprocessor to perform ad hoc code generation. The code for a given feature is guarded with `#ifdefs` and enabled by a `#define`. In this case, a shader compiler must be invoked for each combination of features, to generate a statically specialized variant that "links" different features together. When shipping a game, all required variants are typically compiled ahead of time, and stored in a *variant database*.

Because specialized variants are generated statically, the desired behavior of composing modules dynamically at runtime must be achieved by looking up an appropriate variant in the database. The key used for lookup will depend on everything that can impact the final shader code: e.g., the object, material, and render pass.

2.3 Modular Shaders and Efficient Binding Don't Mix

Ideally, a game engine should be able to define and compose shader features in a modular fashion, while also achieving efficient parameter binding using parameter blocks and knowledge of update rates. In practice, it is challenging to achieve these goals with shading languages in common use. To illustrate this challenge, we will discuss the Unreal Engine (UE4) [Epic Games 2015] as an example of a game engine concerned with both modularity and performance.

Shader features for UE4 are authored in HLSL using traditional `#ifdef` techniques. In order to improve modularity for dynamic composition, important shader features are exposed as C++ classes in host code. Instances of these classes encapsulate a choice of shader features (`#ifdef` flags) and the values of shader parameters needed by those features. Composition of shader features in UE4 is thus expressed by runtime composition of C++ objects.

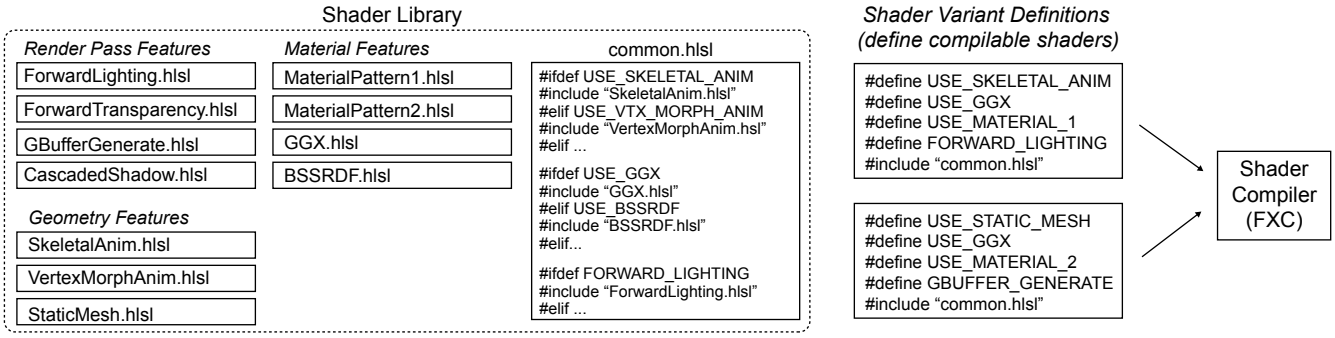

Fig. 1. A shader library organized as snippets of code which are combined together to define a compilable shader *variant*. In the example above, a measure of code modularity is reached using the HLSL preprocessor. Different variants enable and disable features via preprocessor defines, and the necessary code is brought together by selectively including it from the common shader template "common.hlsl". Since modularity is achieved via preprocessor macros, module information is lost by the time shader variant code reaches the shader compiler.

In order to bind shader parameters efficiently in modern APIs, it is desirable to associate these C++ objects, which already hold the values of shader parameters, with parameter blocks. The *layout* for such a parameter block would need to be derived from the corresponding shader code for the feature–and therein lies the problem.

When shader modules are defined in an ad hoc fashion using preprocessor techniques, there is no way for a shader compiler to utilize knowledge about modules to group parameters for efficient binding. For example, the default behavior of the GLSL compiler is to group all shader parameters into a single block. Engines which rely on this default compiler behavior must therefore fill in and bind a complete monolithic parameter block whenever switching between shader features, even if only a few parameters need to be changed.

To bind parameters more efficiently, using multiple parameter blocks, a developer must currently use explicit directives to group parameters, by manually assigning them to fixed binding locations. However, such an approach comes at a cost to modular shader development, as grouping and layout decisions must be coordinated across modules. Rather than accept this complexity the developers of many engines, including UE4, have opted for the simple alternative of monolithic parameter blocks when initially porting their engines to Vulkan or Direct3D 12 [McDonald 2016; Pranckevičius 2015].

We observe that the crux of this problem is the lack of a suitable modularity construct in the shading language, which can map efficiently to the shader parameter binding model in modern APIs.

3 SYSTEM DESIGN

In this section we describe our system design for synthesizing shader modularity with efficient parameter binding. Elements of this design will be familiar to some readers; we build upon concepts that have appeared in various guises in prior work. Section 7 will discuss connections between prior work and our design in detail.

Figure 2 illustrates the key concepts in our system, using a simplified version of the examples we will present in Section 4. The remainder of this section describes the concepts in this figure, and has been organized around key principles that underlie the design.

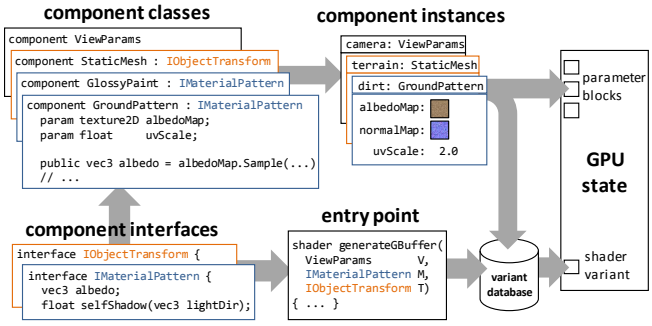

Fig. 2. Conceptual model for shader components. GPU state is driven from a shader *entry point* function and *component instances* that serve as its arguments. Shader features and their parameters are defined as *component classes*. Instances of these classes encapsulate parameter values, and map to parameter blocks in modern APIs. An entry point and component arguments together determine a shader variant. Component classes and instances have been color-coded to match the interfaces they implement.

Shader components serve as a bridge between shader and host application code, and so we discuss both shader- and application-facing design decisions together. An overriding goal of the design is that a component should feel like a single coherent *thing* even as it is accessed from both CPU and GPU code.

3.1 A shader component encapsulates both the code and parameters of a feature

A typical shader feature, such as a surface material, requires a number of parameters, such as texture maps, to perform its computations. Different implementations of the same type of feature (e.g., different materials) will in general need different parameters. In order to allow various implementations of a feature to be easily swapped in and out, it must be possible to *encapsulate* their parameters.

In our design, a shader *component class* is a modular definition of both the code and parameters of a particular shader feature. For example, `GroundPattern` in Figure 2 (and later in Listing 1) is a component class with parameters for an albedo texture map and a scale factor to apply to per-vertex texture coordinates. Another material component class, like `GlossyPaint`, will in general have different parameters.

3.2 A component is exposed to host code as an object

The modularity benefits of shader components should extend to host code. In particular, the encapsulation of code and parameters should be preserved, so that switching between different feature implementations and/or combinations of parameter values can be accomplished with a single operation.

Building on the idea of a shader component class, our design allows host application code to allocate objects called *shader component instances* from these classes. An instance stores concrete values for the parameters declared in the class. For example, `dirt` in Figure 2 is an instance of the `GroundPattern` class that binds `albedoMap` to a particular texture.

3.3 A component instance maps to a parameter block

By using multiple shader component instances, an application can conveniently switch between sets of shader parameters. To make this operation efficient, each component instance that an engine allocates is backed by a parameter block in the target graphics API. In Figure 2, the component instance `dirt` can be used to set a parameter block in the GPU state.

It should be noted that in our design, shader component *classes* are implemented by the shader compiler, while shader component *instances* are implemented by a particular game engine; Specifically, we do not argue for a one-size-fits-all runtime library that implements component instances for all engines. Instead, the compiler framework provides services that allow engines to implement component instances efficiently using parameter blocks; we discuss these services in Section 5.2.

3.4 Entry points are parameterized on interfaces

In our design, each render pass in a game engine corresponds to a shader *entry point*, which coordinates the overall execution and dataflow of shader code for the pass. For example, the `generateGBuffer` entry point in Figure 2 (and Listing 2) corresponds to a G-buffer generation render pass.

A key point of our design is that an entry point should be thought of as incomplete, with "holes" where specific components will be plugged in (these holes are parameters in a technical sense, but should not be confused with *shader* parameters, so we avoid the term). The shape of a hole can be given by concrete component classes (e.g., `ViewParams` in Figure 2), or by component *interfaces*. For example, the `generateGBuffer()` entry point depends on a surface material component that must implement the `IMaterialPattern` interface.

A component interface represents a kind of feature (e.g., materials, lights) in a shader library, and declares the methods and fields that all implementations of that feature must provide. For example, the `IMaterialPattern` interface in Figure 2 (and Listing 3) declares a field named `albedo`; the `GroundPattern` class must define a corresponding field in order to implement the interface.

3.5 Components use static polymorphism

The terminology we use for shader components uses object-oriented concepts like classes and interfaces, which are typically associated with dynamic dispatch (e.g., virtual function tables). However, the semantics of our model are those of *static polymorphism*, where different code is generated for each combination of component types. In essence, one can think of a shader entry point like the following:

```
shader Simple( ILight light, IMaterial material ) {...}
```

as being syntactic sugar for the following "templated" definition:

```
shader Simple<L, M>( L light, M material ) {...}
```

An alternative approach would be to implement dispatch for components dynamically. However, as discussed in Section 2.2, real-time shader code typically benefits from aggressive specialization; this is the default for preprocessor-based approaches, and static polymorphism achieves a similar result for components.

3.6 The engine controls variant lookup and caching

At runtime, an engine will fill in all the holes in an entry point with compatible components, and thereby select both the shader parameters and variant to use. In our design, a variant depends only on the classes of components used as arguments, and not on dynamic parameter values, so it is possible to populate a variant database by enumerating the space of possible component classes ahead of time.

The performance of lookup in a variant database is critical, because variants may be selected in the inner-most rendering loop. Rather than try to implement a one-size-fits-all variant database in a language runtime library, our design leaves the responsibility for caching and lookup up variants to the engine, with the compiler providing specific services to enable efficient implementation. We discuss one implementation strategy, used in our engine, in Section 5.5.

3.7 Components support multi-rate programming

Our design is implemented on top of a multi-rate shading language, in order to better support shader modularity. In particular, multi-rate shading can be used to support modular definition of geometry effects that must work with (abstract over) any number of per-vertex attributes. Most effects that use the programmable tessellation and geometry shader stages of current GPUs fit this description. We also find that it is natural to express shader entry points as multi-rate programs, coordinating work across multiple pipeline stages.

Our multi-rate approach builds on RTSL [Proudfoot et al. 2001], Spark [Foley and Hanrahan 2011], and Spire [He et al. 2016]. As in Spire, we allow rate qualifiers to be elided in single-rate code, so that most programmers need not be aware of multi-rate constructs.

4 EXAMPLES

4.1 Material Pattern Component

Listing 1 shows a shader component class, `GroundPattern`, which evaluates the material pattern for a terrain. The syntax of our system is largely based on that of GLSL.

The `GroundPattern` class declares several shader parameters using the `param` keyword. For example, the `albedoTex` parameter is a texture map that will be used to fetch surface albedo. `GroundPattern` also implements the `IMaterialPattern` interface, shown in Listing 3. To meet the requirements of the interface, `GroundPattern` exports fields like `albedo` and methods like `selfShadow()` using the `public` keyword.

```
component GroundPattern : IMaterialPattern
{
  param texture2D albedoMap;
  param texture2D normalMap;
  param texture2D displacementMap;
  param float uvScale;
  // ...

  require vec2 vertUV;
  require vec4 viewVec;

  float getGroundHeight(vec2 uvCoord) {
    return displacementMap.Sample(samp, uvCoord).x;
  }
  using pom = ParallaxOcclusionMapping(
    GetHeight: getGroundHeight,
    viewDirTangentSpace: viewVec,
    uv: vertUV * uvScale,
    parallaxScale: 0.02
  );
  vec2 uv = pom.uvOut;

  public vec3 albedo = albedoMap.Sample(samp, uv).xyz*0.7;
  public vec3 normal =
    normalize(normalMap.Sample(samp, uv).xyz*2.0-1.0);
  public float roughness = 0.5;
  public float metallic = 0.3;
  public float specular = 1.0;
  public float selfShadow(vec3 lightDir) {
    return pom.selfShadow(lightDir);
  }
}
// definition of ParallaxOcclusionMapping omitted
```

Listing 1. A shader component class for a surface material pattern, simplified from a material used in our evaluation (Section 6).

```
shader generateGBuffer(ViewParams       V,
                       IMaterialPattern M,
                       IObjectTransform T)
{
  using viewParams = V();
  using va = VertexAttributes();

  using xform = T();
  vec3 P_world = xform.TransformPoint(va.P);
  vec3 TanU_world = xform.TransformVector(va.TanU);
  // ...
  vec3 V_world = normalize(viewParams.eyePos - P_world);
  vec3 V_tangent = WorldToTangentSpace(V_world);
  using mat = M(vertUV: va.uv, viewVec: V_tangent);
  // ...

  public out vec4 outputAlbedo =
    vec4(mat.albedo, mat.opacity);
  public out vec4 outputPbr =
    vec4(mat.roughness, mat.metallic,
         mat.specular, mat.ao);
  public out vec4 outputNormal =
    vec4(mat.normal*0.5+0.5,
         mat.isDoubleSided ? 1.0 : 0.0);
}
// omitted: ViewParams, IObjectTransform, and VertexAttributes
```

Listing 2. Shader entry point for a G-buffer generation pass, parameterized on interfaces for the material pattern and transformation/animation features of objects being rendered.

```
interface IMaterialPattern
{
  vec3 albedo = vec3(1.0);
  vec3 normal = vec3(0.0, 0.0, 1.0);
  float roughness;
  float metallic;
  float specular;
  float opacity = 1.0;
  float ao = 1.0;
  bool isDoubleSided = false;
  float selfShadow(vec3 lightDir) { return 1.0; }
}
```

Listing 3. Shader component interface for material patterns, defining the outputs expected by the entry point in Listing 2.

As a convenience, our compiler allows an interface to provide default implementations of fields and methods; for example, `GroundPattern` does not export a value for `isDoubleSided` and so uses the default value of `false`.

This paper focuses on shader parameters provided by a host application, but a shader component may also depend on values that will be provided by other shader code, using the `require` keyword. For example, `GroundPattern` depends on a texture coordinate, `vertUV`, that will be provided by another component.

Our system allows one component to directly invoke another by name; this is conceptually similar to a constructor call. Our example surface pattern invokes the `ParallaxOcclusionMapping` component with the `using` keyword. A component may export multiple values with `public`, so the `using` construct collects these values under a user-selected name (in this case, `pom`), so that they can be used in subsequent computations.

4.2 Shader Entry Point

Listing 2 shows an example shader entry point, `generateGBuffer()`, which coordinates the execution of a G-buffer generation render pass. This entry point sequences the overall shader execution by `using` different components. These include components passed as arguments, like `T`, but can also include components referenced directly by name, such as `VertexAttributes`. When a component has `require`d inputs, their values must be provided at the `using` site. In this example, a texture coordinate (`vertUV`) and view vector (`viewVec`) are explicitly provided to the material pattern `M`.

While much shader logic is expressed in components, a shader entry point can also perform computation. The entry point in Listing 2 invokes methods on the transformation component to transform values into world space, then transforms quantities to tangent space before invoking the material pattern. Finally, the entry point packs computed values into a small number of framebuffer outputs, marked `out`.

5 IMPLEMENTING SHADER COMPONENTS

We have developed an open-source[1] implementation of shader components. Because a shader component bridges shader and host application code, our implementation comprises both a compiler and a high-performance reference rendering engine. The compiler is responsible for translating the shader component language into GLSL

[1] https://github.com/spire-lang/spire

```
// (a) Loading and inspecting shader code
void loadLibraryFromFile(const char* path);
void loadLibraryFromSource(const char* source);
ComponentClass* findComponent(const char* name);
Shader* findEntryPoint(const char* name);

// (b) Inspecting parameter block layouts
enum ResourceType { TEXTURE, BUFFER, ... };
int getResourceGroupCount(ComponentClass* cls)
ResourceType getResourceGroupType(ComponentClass* cls,
                                  int index);
int getResourceGroupSlotCount(ComponentClass* cls,
                              int index);
int getUniformGroupSize(ComponentClass* cls, int index);

// (c) Generating shader variants
enum Stage { VERTEX, FRAGMENT, ... };
Variant* compileVariant(Shader*          entryPoint,
                        ComponentClass** argClasses,
                        int              argCount);
const char* getStageKernel(Variant*, Stage);

// (d) Specialization to parameter values
ComponentClass* specializeClass(ComponentClass* cls,
        int const* paramValues,
        int        paramCount);
```

Fig. 3. Compiler API services provided to support engines using shader components.

kernels. The rendering engine is responsible for the implementation of shader component instances, and for the lookup and caching of variants.

Because we have already covered the details of our language design in Sections 3 and 4, the remainder of this section focuses on the engine implementation, and in particular on the API services that the compiler library must provide to enable that engine to achieve high-performance binding of shader parameters. Figure 3 summarizes these services.

Our engine supports both Vulkan and OpenGL rendering; this section focuses primarily on the Vulkan path.

5.1 Compiling and Loading Shader Code

The engine must first load a library of shader code before it can use the components and entry points it defines:

```
loadLibraryFromFile("shaderlib.spire");
```

After this call, engine code can look up shader component classes and entry points by name:

```
ComponentClass* ground = findComponent("GroundPattern");
Shader* gbufferGen = findEntryPoint("generateGBuffer");
```

In our current compiler API, libraries are always loaded from source code. A more production-ready implementation might introduce a serialized binary format to amortize the cost of front-end parsing and checking.

5.2 Allocating Parameter Blocks

At runtime, the engine needs to allocate component instances, including the underlying API objects that represent a parameter block. In the context of our Vulkan engine, a parameter block comprises a descriptor set, which holds references to resource parameters of various kinds (textures, buffers, samplers, etc.), and (when required) a *uniform buffer* to hold values of simple scalar, vector, and matrix parameters. A component instance is represented with a C++ class like the following:

```
class ComponentInstance {
  ComponentClass* componentClass;
  VkDescriptorSet descriptorSet;
  VkBuffer        uniformBuffer;
  // ...
};
```

When creating a component instance, the engine must determine the required layout and size for its descriptor set and uniform buffer. To this end, the compiler provides an API for inspecting the parameters of a component class, shown in Figure 3(b). The parameters of a component are packed into "resource groups", each of which can be queried for the type of resource (buffer, texture, etc.) and the number of "slots" of that type that are used. For a group that represents uniform parameters, the required buffer size can be queried. Our API also provides a traditional reflection interface for looking up the index/offset of an individual shader parameter by name, within a component class.

5.3 Generating Shader Variants

Our engine populates a variant database as a pre-process, eliminating any need to generate variants on the fly at runtime. In order to generate a complete varaint database, the engine enumerates all of its entry points, and for each entry point, the combinations of shader component classes that could be used as its arguments. For each possible combination of entry point and arguments, the engine invokes compiler services in Figure 3(c) to generate per-stage kernel code for a variant:

```
ComponentClass* argClases[] = {
  viewParams, ground, simpleTransform
};
Variant* v = compileVariant(gbufferGen, argClasses, 3);
const char* fragmentGLSL = getStageKernel(v, FRAGMENT);
// ...
```

After extracting per-stage kernel code from a variant, our engine uses the Vulkan API to generate a suitable pipeline state object. Our engine assigns each variant a *unique ID*, simply derived from the name of the entry point and component classes used; the unique ID can be used to identify a variant across process invocations.

5.4 Binding Parameter Blocks

Recall the simplified loop structure for a render pass from section 2.1. Equivalent pseudo-code for our engine would be:

```
for(v in view)
    bindParameterBlock(0, v);
    for(m in materials)
        bindParameterBlock(1, m);
        for(o in objects)
            bindParameterBlock(2, o);
            bindVariant(...);
            draw();
```

In our engine implementation, each of these loops corresponds to a loop over component instances, and the nesting of loops corresponds to the argument list of a shader entry point (see Listing 2). Each iteration of one of these loops binds a single parameter block (a Vulkan descriptor set). Parameter block binding is performed by engine code, without any interaction with our compiler API.

5.5 Looking up Shader Variants

For each iteration of the inner-most loop in a render pass, the engine might need to bind a new shader variant. The appropriate variant to use depends on the entry point in use and the classes of all its argument components. A naive implementation strategy would construct a unique ID, as described in Section 5.3, and use it as a lookup key for, e.g., a hash table.

In our experience, looking up variants is performance critical, so an engine should avoid an inefficient choice of key (such as a variable-length unique ID string). To optimize variant lookup, our engine assigns sequential small integers, called *dynamic IDs* to all entry points and component classes, at load time. Note that dynamic IDs are assigned to component *classes*–not instances, or composed variants–so even a 16-bit ID can be sufficient.

When our engine needs to look up a shader variant, it concatenates the dynamic IDs of the entry point and components in use to form a *variant key*. Our implementation uses a 128-bit variant key, which is sufficient for entry points using up to 7 parameter blocks. The variant key is then used to index a hash table. We evaluate the performance impact of our variant lookup strategy in Section 6.2.

5.6 Specializing to Parameter Values

So far we have described how our system supports specialization of a shader entry point to a particular choice of components. There are also cases where it may be useful to specialize shader code to particular *values*. For example, we may want to generate lighting code specialized for different numbers of lights in a forward renderer.

We have extended our language to support *specializable* shader parameters. Specializable parameters build upon the `param` syntax. For example, a component class for lighting might declare:

```
component Lighting {
  specializable bool enableShadowFiltering;
  specializable int pointLightCount;
  ...
}
```

The `specializable` keyword may be added to shader parameter of type `bool` or `int`, and indicates that the given parameter is usable for specialization.

By default, our compiler does not perform specialization, and `specializable` parameters are compiled as ordinary uniform parameters. Instead, a specialized version of a component class can be explicitly requested during variant generation:

```
int paramValues[] = { false, 3 };
ComponentClass* threeLights = specializeClass(
  lighting, paramValues, 2);
```

The result of value specialization is a new component class, which will be given its own dynamic ID. As a result, all of the systems described above, such as variant lookup, transparently work with both the original class and any generated specializations.

5.7 Limitations

The design of shader components assumes a one-to-one correspondence between a shader code module, a host-side object (such as a material), and an API parameter block. This assumption works well in cases such as materials, meshes, cameras, etc. but is strained when a shader is logically parameterized on a *collection* of things: e.g., a list of lights. This issue can be mitigated by identifying more coarse-grained components (e.g., a lighting *environment*), but it is possible that future extensions or modifications to the shader component design could address collections of input more directly.

Our compiler implementation statically checks component classes against their declared interfaces, but we have not yet implemented similar checking for entry points; like a C++ template, the body of an entry point is only fully checked when a concrete shader variant is compiled. In the presence of dynamic composition, it is not immediately clear what validation can be performed statically, and which can only be performed at runtime.

6 EVALUATION

The goal of shader components is to provide both the software engineering benefits of shader code modularity and the performance benefit of efficient shader parameter binding for modern graphics APIs. To evaluate the extent to which this goal was achieved, we implemented a large, modular library of shader components and entry points, and a high-performance renderer that uses these components via the interfaces described in Section 5. We first report on the experience of creating this shader library, then provide a detailed analysis of rendering performance.

6.1 Shader Component Library

As described in Section 2.2, a modular shader code library allows many different shading effects to be composed from reusable pieces. To gain experience authoring shaders in terms of shader components, we implemented a large library of shading effects as components.

Our shader component library is patterned after functionality present in AAA game engines with publicly available source [Amazon 2016; Epic Games 2015], and it consists of over 40 components. The library includes the following features:

- Skeletal animation
- PN-triangle tessellation [Vlachos et al. 2001]
- Cascaded shadow maps [Engel 2006]
- Directional lighting
- Physically based environment lighting [Karis 2013]
- Parallax occlusion mapping [Tatarchuk 2006]
- Atmosphere scattering [Bruneton and Neyret 2008]
- Per-material pattern generation (20 unique patterns)
- Material defined vertex animation
- Double-sided lighting
- Transparency and alpha masking

Our renderer also features three types of render passes implemented as shader entry points: a G-buffer generation pass for opaque objects, a forward lighting pass for transparent objects, and a pass for shadow map generation.

By authoring shaders using shader components, we were able to cleanly separate different concerns into independent modules (e.g., PN-triangle tessellation is decoupled from vertex transformation and material pattern generation), and (via component interfaces) make data dependencies between modules clear. While our shader implementations obviously do not incur the full complexity or software legacy of code in a production game engine, detailed inspection of

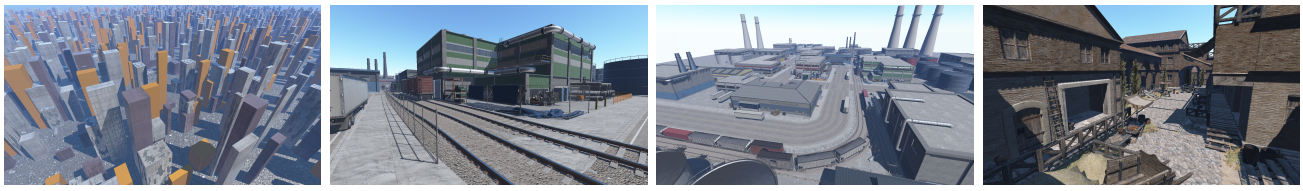

Fig. 4. Views of scenes used for performance evaluation. Statistics of these views can be found in Table 1.

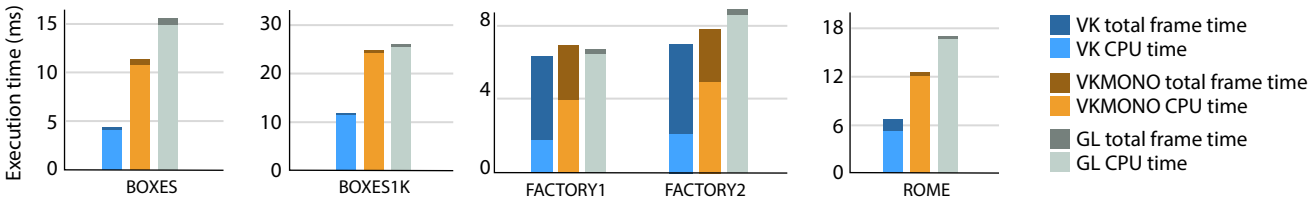

Fig. 5. Single-core CPU time (ms) and full-frame time for all three renderers. All implementations achieve high frame throughput, but VK uses over 2× less CPU time compared to VKMONO and GL. Note that total frame time includes CPU time; the bars are overlapped, not stacked. Full results are in Table 1.

the shader libraries in publicly available AAA engines suggests that our component-based shader library is significantly cleaner, shorter, and easier to read than implementations that attempt to leverage the C preprocessor to achieve similar modularity and performance goals.

6.2 Performance

A well-engineered renderer should be able to achieve high shader parameter binding performance when using modern graphics APIs with a shader component library. Specifically, it should render complex scenes featuring many materials with low CPU overhead. In the case of CPU-bound rendering, reducing CPU costs translates directly into higher frame rate. In a GPU-bound scenario, reducing CPU costs remains valuable since it enables more time to be spent on other application tasks (e.g., AI, animation, physics).

To understand the performance benefits of efficiently managing shader and shader parameter binding, we implemented a Vulkan-based renderer (VK). The renderer creates all component instances at scene load time and pre-allocates a Vulkan descriptor set for each component instance. As discussed in Section 5.4, switching component instances results in binding a single Vulkan descriptor set. When drawing consecutive objects with different material parameters, only three Vulkan commands are generated to bind descriptor sets for the material and object components and issue the draw call. To avoid unnecessary CPU work or GPU pipeline stalls, the renderer sorts scene objects by material during opaque rendering passes (sorting cannot be done when rendering transparent objects) and only binds new shader variants when required.

We compared the performance of VK against two baseline renderers. The first (VKMONO) is a Vulkan-based renderer that simulates common practice in many state-of-the-art game engines (see Section 2.3) by dynamically allocating and initializing a monolithic descriptor set for all shader parameters before each draw call (instead of pre-allocating separate descriptor sets). The second (GL) is a well-optimized OpenGL-based renderer that issues only necessary state-change API calls between draw calls. All renderers run in a single thread, so the Vulkan-based renderers do not take advantage of CPU parallelism opportunity offered by Vulkan.

We evaluated all three renderers on three variations of a microbenchmark scene (used as performance stress tests) and two full scenes purchased from the Unreal Marketplace.

- BOXES. Our microbenchmark scene consists of 10,000 randomly placed boxes. All objects share the same material (resulting in two shader variants: one for forward rendering pass and one for the shadow generation pass). As this scene requires only one parameter block binding per draw call (for the transformation component), it serves as a high-watermark for rendering performance.
- BOXES1K. The same microbenchmark scene as above, but with 1,000 unique materials requiring 100 unique shader variants per entry point (200 in total). This microbenchmark simulates a scene with high material diversity.
- BOXES1KSP. The same microbenchmark scene, but with a unique shader variant per material (2000 total). This scene simulates situations where an engine chooses to specialize shaders by "baking" material parameters into code as constant immediate values (as in UE4).
- FACTORY1 and FACTORY2 are views of a factory environment from the Unreal Marketplace. The scene has 84 unique materials (26 shader variants) and contains transparent geometry (preventing sorting by material) that increases the total number of required shader variant and material switches during rendering. FACTORY1SP and FACTORY2SP are versions of the same scenes, but with a unique specialized shader variant per material as discussed above (162 total shader variants).
- ROME (and its specialized version ROMESP) is a second complex environment from the Unreal Marketplace.

All scenes are rendered at 1920×1080 with directional lighting using four shadow map cascades and image-based environment lighting. Renderings of these views are shown in Figure 4, and key

Scene	Draw Calls	Variants / Binds		Materials / Binds		CPU / Total (VK)		CPU / Total (VKMONO)		CPU / Total (GL)	
BOXES	17,431	2	5	1	5	4.1	4.4	10.8	11.4	14.9	15.6
BOXES1K	17,431	200	480	1,000	3,635	11.2	11.6	23.9	24.7	25.3	25.9
BOXES1KSP	17,431	2,000	3,635	1,000	3,635	56.2	57.1	63.0	63.9	85.8	86.9
FACTORY1	8,755	26	130	84	421	1.8	6.4	4.0	7.0	6.5	6.8
FACTORY1SP	8,755	162	418	84	421	1.9	6.5	4.1	7.1	7.6	7.9
FACTORY2	10,988	26	220	84	537	2.2	7.1	5.0	7.9	8.6	9.0
FACTORY2SP	10,988	162	533	84	537	2.3	7.0	5.1	7.8	9.9	10.5
ROME	26,834	24	129	67	829	5.1	6.6	11.9	12.4	16.5	16.9
ROMESP	26,834	158	833	67	822	5.5	6.6	12.1	12.6	19.0	19.4

Table 1. Full statistics for all scenes. Draw Calls: number of draw calls made to render the view, Variants: number of distinct shader variants (+ number of variant changes per frame), Materials: number of distinct surface pattern component instances (+ number of parameter set changes per frame). The table also provides single core CPU time (ms) and full-frame time using all three renderers.

statistics such as number of draw calls, shader variant changes, and shader parameter changes per frame, are shown in Table 1.

Figure 5 shows the performance of all renderers on all scenes. Detailed data and performance of specialized scene versions are available in Table 1. Measurements were performed on a machine with an Intel i7-5820K CPU (only one render thread) and an NVIDIA GeForce GTX 980Ti GPU. All renderers achieved high rendering throughput despite high draw-call complexity of the scenes, indicating that the baselines are well optimized.

The VKMONO baseline follows common practice in commercial engines and achieves a similar level of performance to UE4 (when configured to use D3D12). For example, VKMONO renders the entire FACTORY1SP scene with a total frame time of 4.1ms, while UE4 running on the same machine takes 4.0ms of CPU time to complete just its shadow and base passes. This comparison is not direct; we exclude passes from the UE4 timing where our engine does not have equivalent features. As such, this data should only be taken as an indication that VKMONO is a strong baseline.

Benefits of efficient parameter binding. The results in Figure 5 demonstrate that minimizing changes to shader parameter state can notably reduce CPU cost. In the FACTORY1, FACTORY2, and ROME scenes, the VK renderer incurs over 2× lower CPU cost than VKMONO. While this CPU cost savings translates into a 5-50% reduction in overall frame time for these scenes, the most important benefit of the VK renderer's efficiency would be realized by other game engine subsystems. As expected, the performance benefits of VK are most extreme on the BOXES microbenchmark, where VK incurs 2.6× and 3.6× less CPU time than VKMONO and GL. (The VK renderer issues near 4 million draw calls per second from a single CPU core on BOXES.) The CPU cost difference between VK and the VKMONO and GL baselines shrinks to 2.1× and 2.3× on BOXES1K, where the 1000 unique materials require more frequent parameter binding.

Effects of aggressive shader specialization. As expected, the performance benefit of VK compared to VKMONO narrows when an engine excessively specializes shaders to parameter values, since frequent parameter and shader variant binding changes are required in this style of renderer design. This effect is most visible for the BOXES1KSP scene, where VK provides only a 10% improvement in CPU overhead, compared to an over 2× improvement for BOXES1K (Table 1). In more realistic scenarios, the benefit of VK remains significant even with aggressive specialization: over 2× lower CPU overhead than VKMONO for FACTORY1SP, FACTORY2SP, and ROMESP .

The synthetic BOXES1K scene was created specifically to evaluate the cost of frequent material changes. BOXES1K and BOXES1KSP use over 10× more materials than the FACTORY and ROME scenes. At this scale, work done per-material and per-variant starts to dominate frame time, so the additional binding work caused by over-specialization has a larger impact: the VK rendering time for BOXES1KSP is over 5× that of BOXES1K.

Costs of dynamic shader variant lookup. Finally, we measured the CPU cost of the dynamic shader variant lookup procedure described in Section 5.5. Although our implementation of variant lookup could be further optimized, we find that the cost of variant lookup is already low: less than 5% of total CPU render thread time in the factory scenes, and 12% in the BOXES microbenchmark stress test. When objects have statically-assigned materials and the pipeline state to render an object does not change dynamically (true in our scenes), it is possible to cache the shader variant associated with each object in the object itself, avoiding the need for any dynamic shader variant lookup; we have implemented this optimization, but do not enable it for the results in this section.

7 RELATED WORK

We believe shader components are an elegant solution, embodying a carefully chosen set of design principles. Many of these same principles have been embodied in prior systems, and in this section we identify common threads across prior approaches. In some cases, similarities in prior work have been obscured by differences in terminology, syntax, or implementation, so we align them all to the conceptual model and terminology of shader components.

7.1 Components encapsulating code and parameters

Cg interfaces [Mark et al. 2003; Pharr 2004], HLSL interfaces [Microsoft 2011], and Spark [Foley and Hanrahan 2011] allow components to be expressed in a shading language using object-oriented classes which encapsulate both code and associated (parameter) data; our system follows this approach. The use of metaprogramming in Sh [McCool and Du Toit 2004; McCool et al. 2002] allows shader components to

be implemented as C++ classes in host code. RTSL [Proudfoot et al. 2001] uses function syntax for components, but takes inspiration from the RenderMan Shading Language, where shaders are semantically treated as classes [Hanrahan and Lawson 1990, Section 3.3].

GLSL shader subroutines [Kessenich et al. 2014; Khronos Group, Inc. 2009] introduce a modularity construct (a kind of restricted function pointer) that can encapsulate a choice of code, but not the corresponding parameter data (there is no support for closures).

Most production shader code is written in a procedural style using HLSL or GLSL, without their interface or subroutine features, respectively. An important, but seldom discussed feature of both HLSL and GLSL is that shader parameters are typically declared at the global scope (rather than in the parameter list of a shader entry point, which might seem more natural and explicit). Support for parameter declarations at global scope enables ad hoc modularity to be achieved by defining the code and parameters for each feature in a separate file, so that features may be included or excluded using the preprocessor (e.g., as shown in Figure 1 in Section 2.2).

7.2 Components exposed to host code as objects

As discussed in Section 2.3, the Unreal Engine exposes important shader features to host code as C++ objects. This kind of usage is directly supported by Sh, when components are authored as host application classes. Spark automatically generates similar C++ classes to expose shader components to host code.

The runtime API for RTSL conflates shader component classes and instances; the effect is as if only a single instance of each class is allowed. GLSL does not need a notion of component instances, because its subroutines only encapsulate code, not data.

The runtime API for Cg interfaces allows component instances to be allocated and referred to by handles. The Direct3D 11 API (which supports HLSL interfaces) includes only a minimal amount of low-level support for component instances; a more complete Cg-like API could be layered on top of this. Our work is most similar to HLSL and Direct3D 11 in this regard, as we argue that the implementation of a runtime library for component instances is best done in engine code, rather than a one-size-fits-all runtime.

Our system is the first to propose mapping components in a shading language to parameter blocks. However, prior systems predate Vulkan and Direct3D 12, so parameter blocks were not available as an implementation choice.

7.3 Entry points parameterized on interfaces

Cg and HLSL both allow a shader entry point, as well as arbitrary functions, to be parameterized on interfaces. GLSL shader subroutines and their use sites are statically checked against separately-declared subroutine types, which serve as interfaces. When using Sh, a host application function can serve as an entry point, with C++ abstract classes serving as interfaces.

Spark does not have a separate notion of an entry point, and instead expects the overall dataflow for a render pass to arise solely from a composition of components.

At first it appears that RTSL has no notion of entry points or interfaces. However, RTSL supports declarations of `surface` and `light` shaders, and these keywords can be seen as associating each shader with one of two built-in interfaces. There is effectively a single built-in entry point in RTSL, which is parameterized on a single surface shader component and zero or more light components.

7.4 Semantics of polymorphic component dispatch

Both Cg and Spark implement their object-oriented syntax with static polymorphism, as does our system (Section 3.5). Because Sh is metaprogrammed, dynamic dispatch in a host application can be used to achieve the same result.

In contrast, both HLSL interfaces and GLSL subroutines are designed to expose a limited form of dynamic dispatch. For example, the HLSL front-end translates a call through an interface into an indirect branch using a jump table. The use of dynamic dispatch effectively trades off GPU efficiency for a possible reduction in CPU overhead (because fewer state changes are made to switch shader kernels). Our experience suggests that with the CPU overhead reductions made possible by modern APIs (see Section 6.2), such trade-offs may not be worth it.

A key difference of our design from prior systems is that we argue (in Section 3.6) that the generation, specialization, and selection of shader variants should be controlled by engine-specific code, rather than an opaque language runtime or GPU driver.

8 DISCUSSION

This paper has shown that while modular shader development and high rendering performance might appear to be at odds, this is not actually the case. By introducing a first-class modularity construct, shader components, to a real-time shading language, and by implementing that construct efficiently on modern graphics APIs, we have shown that modularity and performance can be complementary, rather than conflicting, goals.

Our contribution rests on the observation that a similar decomposition arises when organizing a renderer or graphics API around rates of change as when modularizing a shader codebase around features. Despite the simplicity of this observation, it has not been made explicit in prior work.

Two of the modularity constructs we discussed in Section 7–GLSL subroutines and HLSL interfaces–are supported by mainstream graphics APIs–OpenGL and Direct3D 11, respectively. It is worth noting, that support for these constructs was *removed* as part of the shift to Vulkan and Direct3D 12. We take this as further evidence that the link between efficient rendering and shader modularity is non-obvious, even to experts in the field.

The shift to modern graphics APIs has also seen the introduction of low-level intermediate language (IL) interfaces for shaders, such as SPIR-V [Kessenich et al. 2016]. With IL interfaces comes the prospect of an ecosystem of shading languages and tools built by and for game developers, and with the freedom to experiment and innovate outside of any API standard processes, we are hopeful that our work will provide inspiration to a new generation of production shader programming models.

ACKNOWLEDGMENTS

Support for this research was provided by the National Science Foundation (IIS-1253530), a NVIDIA Faculty Partnership, and a NVIDIA Graduate Fellowship. We would like to thank Nir Benty for his valuable feedback.

REFERENCES

Amazon. 2016. Lumberyard Engine. https://aws.amazon.com/lumberyard/. (2016).

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

Eric Bruneton and Fabrice Neyret. 2008. Precomputed Atmospheric Scattering. *Computer Graphics Forum* (2008). DOI: https://doi.org/10.1111/j.1467-8659.2008.01245.x

Woflgang F. Engel. 2006. Cascaded Shadow Maps. In *ShaderX5 - Advanced Rendering Techniques*, Woflgang F. Engel (Ed.). Boston, Massachusetts, Chapter 4, 197–206.

Epic Games. 2015. Unreal Engine 4 Documentation. 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.

Pat Hanrahan and Jim Lawson. 1990. A Language for Shading and Lighting Calculations. *SIGGRAPH Comput. Graph.* 24, 4 (Sept. 1990), 289–298.

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.

Brian Karis. 2013. Real Shading in Unreal Engine 4. In *SIGGRAPH 2013 Course Notes: Physically Based Shading in Theory and Practice.* http://blog.selfshadow.com/publications/s2013-shading-course/.

John Kessenich, Dave Baldwin, and Randi Rost. 2014. *The OpenGL© Shading Language (Version 4.50).* https://www.opengl.org/registry/doc/GLSLangSpec.4.50.pdf.

John Kessenich, Boaz Ouriel, and Raun Krisch. 2016. *SPIR-V Specification Provisional (Version 1.1, Revision 4).* https://www.khronos.org/registry/spir-v/specs/1.1/SPIRV.pdf.

Khronos Group, Inc. 2009. ARB_shader_subroutine. https://www.opengl.org/registry/specs/ARB/shader_subroutine.txt. (2009).

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

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.

Michael D. McCool and Stefanus Du Toit. 2004. *Metaprogramming GPUs with Sh.* A K Peters. I–XVII, 1–290 pages.

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

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

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

Microsoft. 2017. Direct3D 12 Programming Guide. 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 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.

Natalya Tatarchuk. 2006. Dynamic Parallax Occlusion Mapping with Approximate Soft Shadows. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games (I3D '06).* ACM, New York, NY, USA, 63–69. DOI: https://doi.org/10.1145/1111411.1111423

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