Slang: language mechanisms for extensible real-time shading systems

YONG HE, Carnegie Mellon University
KAYVON FATAHALIAN, Stanford University
THERESA FOLEY, NVIDIA

Designers of real-time rendering engines must balance the conflicting goals of maintaining clear, extensible shading systems and achieving high rendering performance. In response, engine architects have established effective design patterns for authoring shading systems, and developed engine-specific code synthesis tools, ranging from preprocessor hacking to domain-specific shading languages, to productively implement these patterns. The problem is that proprietary tools add significant complexity to modern engines, lack advanced language features, and create additional challenges for learning and adoption. We argue that the advantages of engine-specific code generation tools can be achieved using the underlying GPU shading language directly, provided the shading language is extended with a small number of best-practice principles from modern, well-established programming languages. We identify that adding generics with interface constraints, associated types, and interface/structure extensions to existing C-like GPU shading languages enables real-time renderer developers to build shading systems that are extensible, maintainable, and execute efficiently on modern GPUs without the need for additional domain-specific tools. We embody these ideas in an extension of HLSL called Slang, and provide a reference design for a large, extensible shader library implemented using Slang's features. We rearchitect an open source renderer to use this library and Slang's compiler services, and demonstrate the resulting shading system is substantially simpler, easier to extend with new features, and achieves higher rendering performance than the original HLSL-based implementation.

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

ACM Reference Format:
Yong He, Kayvon Fatahalian, and Theresa Foley. 2018. Slang: language mechanisms for extensible real-time shading systems. *ACM Trans. Graph.* 37, 4, Article 1 (August 2018), 13 pages. https://doi.org/10.1145/3197517.3201380

1 INTRODUCTION

Designers of real-time rendering engines must balance the conflicting goals of facilitating developer productivity and achieving high rendering performance. Code maintainability and extensibility are key aspects of productivity, particularly since popular commercial engines such as Unreal [Epic Games 2015] or Unity [2017] feature large shader libraries used across many titles, each requiring different shading features. At the same time, to achieve high GPU rendering performance, an engine must perform key optimizations such as statically specializing shader code to the rendering features in use, communicating shader parameter data between the CPU and GPU efficiently, and minimizing CPU overhead using the new parameter binding model offered by the modern Direct3D 12 and Vulkan graphics APIs.

To help navigate the tension between performance and maintainable/extensible code, engine architects have established effective design patterns for authoring shading systems, and developed code synthesis tools, ranging from preprocessor hacking, to metaprogramming, to engine-proprietary domain-specific languages (DSLs) [Tatarchuk and Tchou 2017], for implementing these patterns. For example, the idea of *shader components* [He et al. 2017] was recently presented as a pattern for achieving both high rendering performance and maintainable code structure when specializing shader code to coarse-grained features such as a surface material pattern or a tessellation effect. The idea of shader components is to drive both code specialization and CPU-GPU communication using the same granularity of decomposition, but the implementation of this idea was coupled to a custom DSL, which presents a barrier to adoption.

Beyond the coarse-grained specialization addressed by shader components, a modern shading system must be extensible to include new features and to assemble collections of shading effects into statically optimized GPU shaders. (Examples include: composing different types of lights, decoupling material patterns from reflectance models, adopting closed-form solutions and approximations for specific surface-light interactions.) AAA graphics engines implement proprietary code generation tools to aid with these tasks. Unfortunately, metaprogramming tools add complexity on top of the underlying shading language, and thus create additional challenges for learning and adoption. Engine-specific DSLs often lack advanced language features, and create the problem that shader code and learned skills do not transfer between engines.

In this paper we argue that the advantages of specialized DSLs (both the Spire language used to implement shader components [He et al. 2017] and other proprietary, engine-specific tools) can be achieved using the underlying GPU shading language directly, provided the shading language is extended with a small number of best-practice principles from modern, well-established programming languages. Our key contribution is to identify the necessary set of general-purpose modern programming language features that, when added to existing C-like GPU shading languages (GLSL/HLSL/Metal), can be used by real-time rendering engines to build shading systems that are extensible, maintainable, and execute efficiently on modern GPUs *without the need for additional layered DSLs*. Specifically, we:

- Propose the design of the *Slang shading language*, a variant of HLSL extended with the following general-purpose language features: generics with interface bounds, associated types, and interface/structure extensions. The choice of features is intended as a minimal set of extensions to meet our performance and productivity goals, while providing an incremental path of adoption for current HLSL developers.

Authors' addresses: Yong He, Carnegie Mellon University; Kayvon Fatahalian, Stanford University; Theresa Foley, NVIDIA.

0730-0301/2018/8-ART1 $15.00
https://doi.org/10.1145/3197517.3201380

Engine Framework

Shader Library (HLSL)

```
struct Camera {                         Camera.hlsl
  float3 worldPos;
  float4x4 viewProj;
  ...
};
```

```
struct Lambertian {                     Lambertian.hlsl
  float3 albedo;
};
float3 evalBxDF(
  Lambertian bxdf,
  float3 wi, float3 wo)
{ return albedo / PI; }
```

```
...                                     DisneyBRDF.hlsl
```

```
...                                     SkinBRDF.hlsl
```

```
struct PointLight                       PointLight.hlsl
{
  float3 pos;
  float3 intensity;
};
float3 evalLight(
  PointLight light,
  ...)
{ ... }
```

```
...                                     DirectionalLight.hlsl
```

```
...                                     CascadedShadowMap.hlsl
```

```
Camera   gCamera;                       ForwardPass.hlsl
Material gMaterial;
LightEnv gLights;
float3 forwardPass(
  SurfaceGeometry geom) {
  float3 V = gCamera.worldPos - geom.P;
  MaterialPattern pat = evalPattern(
    gMaterial, geometry);
  return illuminate(gLights, pat, V);
}
```

Host Code (C++)

```
class Camera {                          Camera.h
  void setWorldPos(float3 p);
  ...
  void bindParams(Program p);
};
```

```
class Material {                        Material.h
  virtual String getTypeName();
  virtual void bindParams(Program p);
};
```

```
class Lambertian : Material             Lambertian.h
{ float3 albedo;
  String getTypeName() {"Lambertian"}
  void bindParams(p) {
    p->setParam("albedo", albedo); }
};
```

```
class Light {                           Light.h
  virtual String getTypeName();
  virtual void bindParams(Program p);
};
class LightEnv {
  Array<Light*> lights;
};
```

```
class PointLight : Light {              PointLight.h
  float3 pos; float3 intensity;
  String getTypeName() {
    return "PointLight"; }
};
```

```
class Program {                         Program.h
  String getFileName();
};
```

```
class ForwardPass                       ForwardPass.h
  : Program
{
  String getFileName() {
    return "ForwardPass.hlsl" }
};
```

User Application

Shader Code (HLSL)

```
struct MyMaterial {                     MyMaterial.hlsl
  Texture2D albedoTex;
  SamplerState sampler;
};

typedef Lambertian MyMatPattern;

MyMatPattern evalPattern(
  MyMaterial mat,
  SurfaceGeometry geom)
{
  MyMatPattern pat;
  pat.albedo = mat.albedoTex
    .Sample(mat.sampler, geom.uv);
  return pat;
}
```

```
struct QuadLight                        QuadLight.hlsl
{ float3 vertices[4]; };
float3 evalLight(
  QuadLight light,
  MaterialPattern bxdf,
  float3 wo)
{ // integrateQuadLight() provided
  // by BxDF implementation
  return integarteQuadLight(
    bxdf, light, wo);
}
```

Host Code (C++)

```
class MyMaterial: Material              MyMaterial.h
{
  Texture* albedoTex;
  Sampler* sampler;

  String getTypeName()
  {
    return "MyMaterial";
  }
  void bindParams(Program p)
  {
    p->setParam("albedoTex",
                albedoTex);
    ...
  }
};
```

```
class QuadLight : Light                 QuadLight.h
{
  float3 vertices[4];
  String getTypename()
  {
    return "QuadLight";
  }
  void bindParams(Program p)
  { ... }
};
```

```
int main(...) {                         main.cpp
  Camera* cam = new Camera();
  Material* mat = new MyMaterial();
  LightEnv* lights = new LightEnv();
  lights->add(new QuadLight(...));
  ...
  ForwardPass* fwd = new ForwardPass();
  context->bindProgram(fwd);
  camera->bindParams(fwd, "gCamera");
  mat->bindParams(fwd, “gMaterial”);
  lights->bindParams(fwd, “gLights”);
  context->draw(...);
}
```

Fig. 1. An example shading system architecture using HLSL and C++. Left: the framework provides modules that comprise both HLSL `struct` types and corresponding C++ `classes`. Right: an application can extend the features of the framework with new HLSL/C++ modules, and specifies the features to use in a rendering operation by instantiating and filling in C++ objects. The marshaling of data from CPU to GPU is managed explicitly by the `bindParams` methods on the C++ objects.

- Define the Slang runtime API, which provides renderers services for introspecting modules of shading effects and assembling collections of shading effects into efficient, statically optimized GPU shaders. Slang was designed to provide renderer-agnostic mechanisms for defining shading effects and introspecting and compiling shaders. This allows engines to retain control of performance-critical, application-specific policy decisions about shader optimization and execution (e.g., what effects to use, if and when to specialize shaders, when to communicate shader parameters).
- Contribute a reference design for a large, extensible shader library implemented using Slang's features, and rearchitect a large open source research renderer [Benty et al. 2017] to use this library and Slang's compiler services. We show the resulting shading system (both the shading library itself and the CPU-side renderer "host" code to compile and execute shaders) is substantially simpler, easier to extend with new features, and improves performance over the original HLSL-based implementation.

2 BACKGROUND: BEST PRACTICES IN SHADING SYSTEM DESIGN

In this section we describe an example shading system that is architected with design patterns observed in current AAA game engines. Our goal is to provide detailed background on the tasks a modern shading system must perform, and to illustrate how shading system developers currently use a combination of coding conventions, metaprogramming via string concatenation, and the HLSL preprocessor to make trade-offs in performance, code clarity, modularity, and extensibility. In Section 4 we demonstrate that similar goals can be achieved more elegantly and productively, with fewer trade offs, given first-class language support. Readers familiar with the development of large real-time shading systems may elect to skip to Section 3.

A real-time *shading system* comprises both GPU code for a *shader library* (defining shading features such as material models, lighting, geometry effects, etc.) and the CPU *host code* responsible for preparing and invoking GPU work (compiling and executing shaders, communicating parameters to the GPU). Fig. 1 provides an overview of key pieces of the example shading system, which uses C++ for host code and HLSL for the shader library.

The shading system in Fig. 1 is architected to prioritize extensibility and performance. We consider a real-time *renderer* (e.g., a game engine like Unreal or Unity) to be a framework that is used by many different *applications* (e.g., game titles). An extensible shading system must enable an application to add features without needing to modify the host code or shader library of the renderer. Code provided by the renderer is on the left in Fig. 1, while code specific to a particular application (specific materials, light sources, etc.) is on the right. As will be discussed in detail, the example shading system is performant since its architecture allows the renderer to statically specialize GPU shaders to exactly the features in use.

2.1 Authoring a Modular Shader Library

To aid developer productivity, it is desirable for shading system features to be expressed in a clear and modular fashion. In the example system, the implementation of each feature spans code in both HLSL and C++. The decomposition of features is similar in both languages: e.g., there is both an HLSL type `Camera`, and a corresponding C++ class. The HLSL type encapsulates the parameters required by a feature (e.g., per-view camera parameters), while the C++ class holds and communicates parameter data to a shader. This pairing of shader code for a feature with a C++ class echoes the use of `FShader` and `FMaterialShader` subclasses in Unreal Engine [Epic Games 2015].

While there is only a single implementation of cameras in Fig. 1, other features of the shader system support multiple implementations: notably, materials and light sources. In C++, such choices can expressed with an abstract base class, such as `Light`, with concrete implementations in derived classes such as `PointLight`. In HLSL, each concrete implementation has a corresponding HLSL `struct`, but there is no direct encoding of a space of choices; Fig. 1 instead uses color to group types by feature: camera, materials, and lights.

GPU shader execution begins with an *entry point* function, such as the `forwardPass` entry point for the fragment shading pipeline stage. The entry point is responsible for declaring shader parameters for all features in use, such as the variable `gCamera`, which represents camera parameters, and for coordinating the execution of code across different features and subsystems. For example, `forwardPass` invokes the pattern generation step (Section 2.3) for a surface material feature and then integrates reflectance over a lighting environment.

2.2 Generating Specialized Shader Kernels

In order to optimize for throughput-oriented GPU processors, shading code is usually aggressively specialized to exactly the features that are in use. The example shader system uses a combination of metaprogramming and clever use of the HLSL preprocessor to statically specialize the code of an entry point for different combinations of material and lighting features. Specialization yields a *variant* of the entry point that can be compiled to an executable GPU *kernel* optimized for the chosen features.

Notice how the `forwardPass` entry point is written abstractly in terms of types (`Material`) and functions (`evalPattern`) which it does not define. Listing 1 shows an example of how a variant of `forwardPass` can be specified by including features that define the required types and methods. For example, the concrete `MyMaterial` type (from `MyMaterial.hlsl`) is "plugged in" for the (correspondingly colored) abstract `Material` type using a `typedef` prior to including

```
#include "MyMaterial.hlsl"
typedef MyMaterial Material;
typedef MyMaterialPattern MaterialPattern;
#define PointLightCount 1
#define QuadLightCount 1
#include "LightEnv.hlsl"
#include "ForwardPass.hlsl"
```

Listing 1. HLSL code that specializes the `forwardPass` entry point of the shader system in Fig. 1 to use the material defined in `MyMaterial.hlsl`. It also specializes the entry point's lighting code to use a single point light and an area light. Colors correspond to interactions with features in Fig. 1

the text of the forward pass entry point. A similar approach to specialization appears in the shader library for the Lumberyard engine [Amazon 2016].

The specialization in Listing 1 could be generated by a renderer's host code by pasting together strings of HLSL according to the effects a scene object requires (a simple form of metaprogramming). For example, the renderer can query for the name of the HLSL type corresponding to a material in use via the virtual `getTypeName` operation on a C++ `Material` instance, then use this string to generate the appropriate `typedef` lines.

The approach to specialization illustrated in Listing 1 is concise, allowing different material implementations to be specified simply by modifying includes and `typedef`s, but it relies on assumptions that are never explicitly declared in code. For example, each material must provide a definition of `evalPattern` with a unique signature so that, e.g., when the `forwardPass` entry point calls `evalPattern`, type-based overload resolution by HLSL statically dispatches to the appropriate code (based on the type of the `gMaterial` argument). The connection between the `evalPattern` call site in `forwardPass` and the pattern evaluation code for `MyMaterial` is not explicit. Nor is the material functionality expected by `forwardPass` explicitly defined or enforced in the code. *Instead, realizing a valid HLSL shader is a matter of adhering to engine policy.* If an entry point requires an operation that some, but not all, materials support, no error will be raised until the engine tries to generate a variant that combines the entry point with an offending material.

Alternative uses of the preprocessor, such as littering the entry point definition with a series of `#if`'s to statically specialize to each specific material in the shader library, are also common in commercial shading systems. These designs (arguably) make data and control flow more explicit, but they fail to provide clear separation between the renderer-provided entry point and set of material types, forcing applications that wish to add new shading features to modify renderer framework code.

2.3 Separating Phases of Material Shading

It is common for physically-based shading systems to separate evaluation of surface materials into distinct phases for *pattern generation* and reflectance function evaluation. For example, an OpenSL [Imageworks 2017] surface shader expresses pattern generation (e.g., sampling and combining texture layers to compute albedo) and returns a "radiance closure" representing the reflectance function and its parameters, which is then evaluated as needed by the renderer. We will use the term *BxDF* for any reflectance function: a

```
// Assumptions:
//  - PointLight.hlsl defines PointLight type and
//     evalLight(PointLight, ...)
//  - MaterialPattern type provided by pattern definition
//  - evalBxDF() is provided by material definition

#include "PointLight.hlsl"
// ...
struct LightEnv {
#if PointLightCount
  PointLight pointLights[PointLightCount];
#endif
  // ...
}
float3 integrate(MaterialPattern pat, LightSample s, float3 wo) {
  return s.Li * evalBxDF(pat, s.wi, wo) * max(0, dot(s.wi, wo));
}
float3 illuminate(LightEnv env, MaterialPattern pat,
                  SurfaceGeometry geom, float3 wo) {
  float3 sum = 0;
#if PointLightCount
  for(int i = 0; i < PointLightCount; i++)
    sum += integrate(pat, evalLight(env.pointLights[i], geom), wo);
#endif
  // ...
}
```

Listing 2. Sketch of a lighting environment implemented using preprocessor conditionals. Adding a new light type to the renderer involves editing this code in three places. Colors highlight assumptions about features in Fig. 1

BRDF [Nicodemus et al. 1992], BSDF [Bartell et al. 1981], etc. Developers adding new BxDFs should have to consider physical correctness, while artists authoring new material patterns need not worry about breaking physical invariants.

Each BxDF requires a unique set of input parameters, so material surface shaders which use different BxDFs will use different types to store the parameter values provided by pattern generation. In the `forwardPass` entry point in Fig. 1, the `MaterialPattern` type is used to represent the material parameters produced by pattern generation.

Notice that the choice of `MaterialPattern` type in Listing 1 is tied to the choice of `Material`, but the code that generates this specialization must define both consistently; this is another implicit policy in our toy system. The `MaterialPattern` type is defined via a chain of two `typedefs` representing two choices: the shader code for `MyMaterial` in Fig. 1 selects the concrete `Lambertian` BRDF as the result of its `evalPattern`, and the specialization logic in Listing 1 selects `MyMaterial` for use by `forwardPass`.

As in Section 2.2, the example system's design achieves extensibility with new features (materials and BxDFs), but relies on implicit engine policies around how features must be authored.

2.4 Specialization to Lighting Environment

It is also desirable to statically specialize GPU shaders to the structure of a lighting environment: e.g., per-object light lists in a forward renderer, or per-tile light lights in a deferred renderer. Although material specialization typically must only consider a single material in use for an object, lighting specialization is more challenging because it must account for multiple active lights and different light types.

Listing 2 demonstrates one approach to lighting environment specialization for a forward renderer, which differs from the material specialization of the previous sections by making heavy use of preprocessor conditionals; this additional complexity is required to achieve performant specialized code. To avoid per-light conditional execution that would result from a heterogeneous array of lights, the composite lighting environment (`LightEnv`) contains distinct arrays for each type of light in use (for brevity we only show handling of point lights). The renderer also implements an `illuminate` operator that integrates reflectance over the lighting environment (in this case by looping over all lights). Fig. 1 expects the lighting environment to provide a definition of `illuminate`). This design assumes that the renderer will introduce preprocessor definitions like `PointLightCount` for each light type in use (as in Listing 1).

Using preprocessor conditionals to specialize shaders to light types creates the problem that extending the lighting subsystem with a new light type, such as the `QuadLight` added by the application in Fig. 1, requires editing the renderer's shader library implementation (modifying the `LightEnv` type and the implementation of `illuminate`). Achieving the extensibility benefits of renderer/application separation *and also* static specialization to lighting environment would require a more advanced form of metaprogramming by the engine than the specialization scripts shown in Listing 1. Specifically, each C++ light class could implement a virtual function `getIlluminateCode` that returns a string of HLSL code to insert into `illuminate` for the given light type, and the renderer could assemble these strings into an implementation of Listing 2 specialized for a specific composite lighting environment. The use of code generation enables extensibility, but further increases the complexity of the shading system over the preprocessor-based solution.

2.5 Adding BxDF-Dependent Light Types

The light loops in Listing 2 may include different code for each light type. For example, rather than sample incident illumination along a single ray (as done for point or directional light types), a real-time renderer may make use of closed-form solutions or approximations to integrate the reflectance of a surface due to more complex light sources (e.g., a polygonal area light) [Heitz et al. 2016]. Closed-form approximations may involve code that is *algorithmically specialized* to both the choice of BxDF and light type. For example, the `evalLight` operation for `QuadLight` in Figure 1 calls out to `integrateQuadLight`, which is expected to be implemented by the `MaterialPattern` type (the selected BxDF). We will refer to lights that are evaluated using such algorithmic specialization as *BxDF-dependent*.

The addition of polygonal area lights in the example helps to illustrate the steps required to add a new BxDF-dependent light source type. Beyond the steps discussed in Section 2.4, a developer must ensure that a function akin to `integrateQuadLight` is defined, with an overload provided for every BxDF implementation. Similarly, a developer adding a new BxDF must also be aware of any BxDF-dependent light source types, and ensure that their new BxDF implements the required callbacks. The shader compiler does not provide a user with assistance in identifying the changes that must be made to implement the required engine policy.

2.6 Efficiently Communicating Shader Parameters

In addition to composing shading features into highly specialized GPU kernels, a high-performance shader system must also perform efficient management and communication of shader parameters to the GPU. High-performance CPU-GPU parameter communication can be achieved by using coarse-grained *parameter blocks* that are be populated ahead of time and bound to the graphics pipeline at the necessary frequency when rendering scene objects. Modern graphics APIs like Direct3D 12 and Vulkan introduce API mechanisms (descriptor tables and sets, respectively) that can be used to implement parameter blocks efficiently in GPU memory. (See He et al. [2017], Section 5 for a tutorial on efficient parameter block use.)

The problem is that extensible shader system design conflicts with efficient use of parameter blocks. By declaring all shader parameters, such as `gMaterial` or `gLightEnv`, at global scope, the example shader system's design leaves the decision of how to lay out these parameters in GPU memory to the HLSL compiler, which is permitted by the HLSL language definition to eliminate or reorder parameters based on their usage in a fully specialized entry point. Since the layout of input parameters for each shading feature is determined by what other shading features are in use (and even the internal implementation of those features), objects rendered using the same lighting environment, but different materials, *may require different layouts for their lighting parameters.* This prevents the shader system from populating a lighting environment parameter block in advance, and efficiently re-using it as a shader input for many objects.

Due to this problem, modern shader systems either sacrifice performance by allocating, populating, and transferring a new parameter block to the GPU for each scene object drawn (incurring CPU cost and CPU-GPU communication) [McDonald 2016; Pranckevičius 2015] or employ explicit HLSL parameter layout annotations to manually specify where each parameter should be placed in the global layout for an entry point. Use of annotations enables efficient parameter communication (including use of parameter blocks), but limits shader system extensibility. Manual parameter layout is a global process requiring each shading feature (including features added by applications) to receive parameters in a designated location that does not conflict with other features.

2.7 Summary

The problems described in this section confront all modern AAA renderers, except they are amplified in the context of code bases with hundreds of shading effects and hundreds of thousands of lines of shader code. While metaprogramming and preprocessor-based solutions can address aspects of the challenge, as seen here the result is code that is difficult to understand and debug or that sacrifices key performance properties. More advanced DSLs [He et al. 2017; Tatarchuk and Tchou 2017] have elegantly addressed a subset of these challenges, but these solutions are either engine-specific or lack features of more established programming languages. As a result, we believe there is an acute need for better general-purpose language support for addressing these performance, code maintenance, and extensibility issues.

3 THE SLANG LANGUAGE

The Slang language is based on the widely used HLSL shading language, extended with general-purpose language features that improve support for modularity and extensibility. In this section we briefly introduce the features Slang adds to HLSL. Section 4 will show how these features can be applied to address the challenges presented in Section 2.

Slang's design is governed by two key principles. First, we sought to maintain compatibility with existing HLSL whenever possible. New features should provide a path for incremental adoption from existing HLSL code, rather than require all-or-nothing porting. Second, we sought features that have precedent in a mainstream application or systems programming language, which can be relied upon for familiarity to developers, and to provide intuition. We emphasize that individually each of Slang's general-purpose extensions to HLSL are not novel programming language features. For example, they have equivalents in both the Rust [2015] and Swift [Apple Inc. 2014b] programming languages, and most appear in C# [ECMA International 2017]. Indeed, that is the point. Our contribution is to identify the features from modern languages that are necessary to achieve the goals of real-time shading, while eliding those that would interfere with generation of high-performance code.

3.1 Generics

Slang supports parametric polymorphism using the syntax of generics as in Rust, Swift, C#, Java, etc. For example, we can define a function that evaluates the rendering equation for any BxDF, given incident illumination along a single ray:

```
float3 integrateSingleRay<B:IBxDF>(B bxdf,
    SurfaceGeometry geom, float3 wi, float3 wo, float3 Li)
{ return bxdf.eval(wo, wi) * Li * max(0, dot(wi, geom.n)); }
```

In this example the parameter `B` stands in for the unknown BxDF.

3.2 Interfaces

As in most languages with generics, but unlike C++ templates, a generic like `integrateSingleRay` is semantically checked once in Slang, rather than once for each specialization. In order to check the body of the function, it is necessary to describe what operations are available on values of type `B`. In Slang this is done using `interface` declarations, which correspond to `traits` in Rust, `protocols` in Swift, and type classes in Haskell [Wadler and Blott 1989].

The declaration of the `IBxDF` interface used in `integrateSingleRay` looks like:

```
interface IBxDF { float3 eval(float3 wo, float3 wi); }
```

The `IBxDF` interface defines one *requirement*: `eval`. Any type that wants to *conform* to this interface will need to provide a concrete method to satisfy this requirement. For clarity, this paper will use a convention where all interface names are prefixed with `I`.

The `integrateSingleRay` function uses the `IBxDF` interface to provide a *bound* for the type parameter `B`. Because of this bound, the function can safely call the `eval` method on its `bxdf` parameter (since it must conform to the required interface).

3.3 Associated Types

A generic function that must to work with any surface material shader of type `M`:

```
float3 shade<M : Material>(M material, ...) { ... }
```

As discussed in Section 2.3, evaluating the pattern of a surface material yields a BxDF, where the type of BxDF depends on the type of material. *Associated types* are a language mechanism that allows code to name the BxDF type associated with `M`, without knowing it exactly.

An associated type is an interface requirement that is a type, rather than a method:

```
interface IMaterial { associatedtype Pattern : IBxDF; ... }
```

A concrete type that conforms to the `IMaterial` interface must define a suitable type named `Pattern`, either as a nested `struct` or `typedef`. An associated type may come with bounds, just like a generic parameter; in this case, the concrete `Pattern` type must conform to the `IBxDF` interface.

3.4 Retroactive Extensions

In some cases, applications using a framework may wish to extend features of the framework. For example, suppose an engine framework defines a `Lambertian` type that could be used as a BxDF, but doesn't specify conformance to the `IBxDF` interface. Slang supports *extension* declarations, that allow an application to "inject" new behavior into existing framework types:

```
extension Lambertian : IBxDF { float3 eval(...) { ... } }
```

Slang borrows its syntax for this feature from Swift, but equivalents exist in Rust and Haskell. This `extension` declaration makes the type `Lambertian` conform to `IBxDF`; it is the responsibility of the extension author to provide the requirements of the interface.

3.5 Explicit Parameter Blocks

A final aspect of Slang's design is not present in modern general-purpose languages, but is instead taken from the C++-based shading language for Metal [Apple Inc. 2014a]. While HLSL and GLSL do not have a first-class language construct that corresponds to a parameter block, Slang and Metal allow the user to define the contents of a parameter block as an ordinary `struct` type, where fields of the `struct` constitute shader parameters:

```
struct PerFrameData { float3 viewPos; TextureCube envMap; ... }
```

To use a type like `PerFrameData` in a parameter block, a Metal programmer simply declares an entry point parameter using a C++ pointer or reference to the type. (The memory layout of the parameter block is given by the `struct`'s definition.)

In order to support a variety of graphics APIs, which may implement the memory layout of a parameter block differently, Slang leaves the data layout of a parameter block abstract by exposing a generic `ParameterBlock<T>` type in its standard library. This type may be implemented differently on each target platform. For example, on targets that support "bindless" resource handles, a parameter block can be implemented as a simple GPU memory buffer.

```
float3 forwardPass<M : IMaterial, L : ILightingEnv>(
  ParameterBlock<Camera> camera,
  ParameterBlock<M>      material,
  ParameterBlock<L>      lights,
  SurfaceGeometry        geometry)
{
  float3 viewDir = normalize(camera.P - geometry.P);
  M.Pattern bxdf = material.evalPattern(geometry);
  return lights.illuminate(bxdf, geometry, viewDir);
}
```

Listing 3. A fragment shader entry point written as a function with generic type arguments. Plugging in different types for these arguments yields shader variants with different behavior, and different parameter data.

4 USING SLANG TO DESIGN A SHADING SYSTEM

In this Section we describe how Slang's features facilitate implementation of a version of the shading system from Section 2 that is modular (with statically checked interfaces), easily extensible, and performant (yields statically specialized shaders that efficiently receive inputs through parameter blocks).

4.1 Generating and Using Shader Variants

Listing 3 shows a fragment shader entry point similar to `forwardPass` in Fig. 1, which uses generics and interface bounds, rather than preprocessor-enabled metaprogramming, to achieve specialization. The choice of material and lighting effects is expressed with the generic type parameters `M` and `L` respectively; The shader body evaluates the surface pattern of the material, and then requests integration of incident illumination from the lighting environment. Plugging in concrete types for `M` and `L` yields a specialized variant of this entry point, with different behavior for these steps.

Using Slang interface bounds on type parameters allows the compiler can type-check `fragmentMain` and determine that it is compatible with any material and light types that implement the specified interfaces. Compatibility is guaranteed even for a type implemented in a separately compiled file, so that our static checking guarantees also benefit extensibility; a user can confidently extend an existing entry point to support new effects.

Listing 3 also demonstrates the use of parameter blocks to encapsulate shader parameters for efficient communication using modern graphics APIs. In this example, the `material` parameter block uses the generic type parameter `M`, so that the choice of a concrete material type influences not only the behavior of a specialized variant, but also the parameters it accepts.

A renderer can allocate a parameter block for a specific material type using reflection information provided by the Slang compiler's runtime API (Section 5). That API can also be used to generate specialized variants of `forwardPass` using the chosen material type. By using language and API support for the distinct mechanisms of generics and parameter blocks, this shader system implements the shader components design pattern without the need for a DSL with first-class "components" [He et al. 2017].

```
interface IBxDF {
  float3 eval(float3 wo, float3 wi);
}
interface IMaterial {
  associatedtype Pattern : IBxDF;
  Pattern evalPattern(SurfaceGeometry geom);
}
```

Listing 4. Slang interfaces for defining surface reflectance functions and surface material patterns. Each implementation of the `IMaterial` interface will be associated with the particular type of reflectance function it yields.

```
struct Lambertian : IBxDF {
  float3 albedo;
  float3 eval(float3 wo, float3 wi) {
    return albedo / PI;
  }
}
struct TexturedLambertian : IMaterial {
  typedef Lambertian Pattern;

  Texture2D albedoMap;
  SamplerState sampler;
  Lambertian evalPattern(SurfaceGeometry geom)
  {
    Lambertian bxdf;
    bxdf.albedo = albedoMap.Sample(sampler, geom.uv);
    return bxdf;
  }
}
struct DisneyBRDF : IBxDF { ... }
struct RustedMetal : IMaterial {
  typedef DisneyBRDF Pattern;
  DisneyBRDF evalPattern(SurfaceGeometry) { ... }
  ...
}
```

Listing 5. Reflectance and pattern generation functions defined as types implementing the `IBxDF` and `IMaterial` interfaces (Listing 4), respectively.

4.2 Separating Phases of Material Shading

As discussed in Section 2.3, it is desirable to enforce the separation of material shading into distinct pattern generation and BxDF evaluation steps. In the context of a preprocessor-based specialization system, this involves conditionally defining a type that stores the parameters of the chosen BxDF. When using generics for specialization, a similar function is served by associated types (Section 3.3).

Listing 4 shows Slang declarations of `IBxDF` and `IMaterial` interfaces that express the concepts of a reflectance function and material surface shader respectively. The definition of `IBxDF` is straightforward, while `IMaterial` makes use of an associated type (Section 3.3) to capture the dependence of the reflectance function type on the choice of surface shader.

A concrete implementation of `IMaterial` will define the specific type to use for the associated type `Pattern`. For example, the surface shader `TexturedLambertian` in Listing 5 defines `Pattern` to be of `Lambertian` type, which implements a trivial diffuse BRDF.

Shader code that takes a material type parameter, such as the parameter `M` of the `forwardPass` function in Listing 3, can refer to the associated reflectance function type as `M.Pattern`. Because the associated type `Pattern` is bounded using the `IBxDF` interface, only the operations provided by that interface can be used in `forwardPass`.

```
interface ILightEnv {
  float3 illuminate<B:IBxDF>(B bxdf, SurfaceGeometry geom,
                             float3 wo);
}
struct DirectionalLight : ILightEnv {
  float3 direction;
  float3 intensity;
  float3 illuminate<B:IBxDF>(B bxdf, SurfaceGeometry geom,
                             float3 wo) {
      return integrateSingleRay(
          bxdf, geom,  wo, direction, intensity);
  }
}
struct PointLight : ILightEnv { ... }
```

Listing 6. A lighting environment is represented as a type implementing the `ILightEnv` interface. A single light source (e.g., `DirectionalLight`,`PointLight`) is treated as a simple case of a lighting environment.

For example, an attempt to access the `albedo` field of the BxDF would yield a compile-time error since not every BxDF is guaranteed to have such a parameter.

Associated types achieve a similar result to the ad hoc approach in Section 2.3, with the added benefit that entry points like `forwardPass` and materials like `TexturedLambertian` can be compiled and validated independently. Thus, when extending an engine with a new material, a user can have confidence that the new material will work with all entry points that require materials to implement `IMaterial`.

4.3 Specialization to Lighting Environment

Section 2.4 showed that it was challenging for simple preprocessor-based solutions to simultaneously support specialization to a lighting environment and preserve the ability to extend the system with new light types. This motivated more general code generation techniques like string pasting. Using the language mechanisms of Slang, our shading system can support both specialization to a lighting environment and extension to new light types without resorting to string-based code generation.

Listing 6 shows pieces of a framework for defining lighting environments that we will develop here and the next section. The `ILightEnv` interface declares that every lighting environment must provide an operation to illuminate a surface sample, and that operation must be generic in the BxDF of the surface. The `illuminate` operation is expected to integrate light coming from the environment that is reflected by the surface in direction `wo`.

In the lighting system we present, a single light source is treated as a simple case of a lighting environment. For example, `DirectionalLight` in Listing 6 implements a simple directional light that conforms to the `ILightEnv` interface. For clarity, `DirectionalLight` implements its integration using the `integrateSingleRay` function, defined in Section 3.1. Additional infinitesimal light types such as `PointLight` can be defined similarly.

Given the definitions in Listing 6 it is possible to specialize the shader entry point in Listing 3 for any single light source. However, when more complex lighting environments are required, we can use generics to define types for building composite lighting environments out of simpler ones.

```
struct LightArray<L:ILight, const N:int> : ILightEnv {
  L   lights[N];
  int lightCount;
  float3 illuminate<B:IBxDF>(Surface<B> surface, float3 wo) {
    float3 result = 0;
    for(int i = 0; i < lightCount; i++)
      result += lights[i].illuminate(surface, wo);
    return result;
  }
}

struct LightPair<H:ILightEnv, T:ILightEnv> : ILightEnv {
  H head;
  T tail;
  float3 illuminate<B:IBxDF>(Surface<B> surface, float3 wo) {
    return head.illuminate(surface, wo)
         + tail.illuminate(surface, wo);
  }
}
```

Listing 7. Composite lighting environments defined using generics. A `LightArray` can be used to encapsulate a homogeneous array of lights, while `LightPair` can be used to "unroll" a heterogeneous list of lights. Each allows the simple entry point in Listing 3 to transparently work with either a single light or a list of many lights.

Listing 7 shows two types of composite lighting environments. The `LightArray` type implements a dynamically-sized (but statically bounded) array of lights. The `L` type parameter represents the type of the elements in the light array, while `N` is a generic value parameter that is an upper bound on the number of lights that may appear. For example, the type `LightArray<DirectionalLight, 16>` represents an array of (up to) 16 directional lights.

While `LightArray` used to support a dynamically-sized, but homogeneous composite, the `LightPair` type can be used to compose a heterogeneous lighting environment. A light pair like

```
LightPair<DirectionalLight, PointLight>
```

simply sums the contributions from its constituent lighting environments.

These two simple types can be used as composition operators to construct more complicated lighting environments. For example, if an application wants to specialize a shader entry point for rendering with a single directional light with cascaded shadow maps, plus up to 16 point lights, it can construct the type:

```
LightPair<CascadedShadowMap<DirectionalLight>>
  ArrayLight<PointLight, 16>>
```

In practice, we anticipate renderer designs where an application uses data-driven code (based on application or framework data structures representing scene light sources) to generate Slang types for complex scene lighting environments. The Slang compiler's runtime API (Section 5) provides applications support for constructing these composite types, and for querying the layout information required to store values of these types in parameter blocks.

By allowing light composition operators to be expressed as types, Slang raises the level of abstraction in the host code for lighting environments. Rather than pasting together strings of shader code, the engine now composes shader types, and then creates instances of those types.

```
struct QuadLight : ILightEnv {
  float3 vertices[4];
  float3 intensity;
  float3 illuminate<B:IBxDF>(B bxdf, SurfaceGeometry geom,
                             float3 wo) {
    return bxdf.acceptQuadLight(
      this,
      geometry,
      wo);
  }
}
interface IAcceptQuadLight {
  float3 acceptQuadLight(
    QuadLight      light,
    SurfaceGeometry geom,
    float3         wo);
}

extension IBxDF : IAcceptQuadLight {}

extension Lambertian : IAcceptQuadLight {
  float3 acceptQuadLight(QuadLight     light,
                         SurfaceGeometry geom,
                         float3         wo) {
    return albedo * LTC_Evaluate(light, geometry, wo,
      float3x3(1,0,0,  0,1,0,  0,0,1));
  }
}
extension DisneyBRDF : IAcceptQuadLight { ... }
```

Listing 8. Extending the shader system to support an (approximate) area light type. The new light type cannot efficiently be supported with the existing `IBxDF` interface, so a new interface for surfaces that accept area lights is introduced. `extension` declarations can be used to make pre-existing reflectance functions like `Lambertian` support the interface required by the new light type.

4.4 Adding BxDF-Dependent Light Types

Section 2.5 discussed the challenge of adding a BxDF-dependent light type, which relies on BxDF-specific closed-form evaluation or approximation for performance. The crux of the challenge is that the code to execute depends on both the light and the BxDF. By using the `extension` mechanism in Slang, an application can address this challenge by injecting light-type-specific operations into existing BxDF types without having to modify the simple abstractions of the framework presented so far (e.g., Listing 6).

The `QuadLight` type in Listing 8 implements a quadrilateral area light. This type may seem superficially simple, but note that the `illuminate` implementation invokes a method named `acceptQuadLight` on a BxDF, while the original definition of `IBxDF` in Listing 4 does not declare such a method.

Instead, the `acceptQuadLight` operation is defined as part of the `IAcceptQuadLight` interface in Listing 8. Three `extension` declarations are used to make existing framework types conform to the new interface. First, the `IBxDF` interface is extended with a new requirement, so that any conforming type must also support the `IAcceptQuadLight` interface. Next, the `Lambertian` and `DisneyBRDF` BxDF types are extended with concrete implementations of `acceptQuadLight` that, in our example, perform a closed-form approximation using linearly transformed cosines [Heitz et al. 2016].

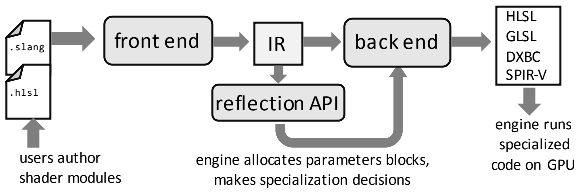

Fig. 2. The Slang system implementation comprises a compiler front-end (for checking and determining parameter layouts for unspecialized modules) and back-end (for generating specialized GPU kernels), as well as supporting APIs. Applications use the Slang reflection API to obtain shader parameter layout information used to allocate GPU parameter blocks and to guide the specialization choices needed for low-level code generation.

```
// Front End
Module* loadModule(const char* path);

// Reflection
EntryPoint* findEntryPoint(Module* module, const char* name);
Type* findType(Module* module, const char* name);
TypeLayout getTypeLayout(Type* type);
Type* specializeType(Type* type, Type* types[], int ntypes);

// Back End
Kernel* specializeEntryPoint(EntryPoint* entryPoint,
                             Type* types[], int ntypes);
```

Listing 9. An overview of API services provided by the Slang implementation, grouped according to the system diagram in Fig. 2.

Any BxDF implementation that is not extended to support the new interface will result in a compiler error. The implementation of extensions in Swift and Rust require a default implementation of `acceptQuadLight` to be provided when extending `IBxDF`; that implementation would then be used as a fallback for BxDFs that have not been explicitly extended. We are considering adding this feature to Slang.

The ability of extensions to "inject" code into existing types allows an application developer to write a module that expands the capabilities of the renderer framework's shading system without disturbing its existing implementation or adding complexity for other users. Unlike the preprocessor-based solution described in Section 2.5, the module in Listing 8 makes the policy decisions around what has been extended and what the new interface requirements are explicit (e.g., it is clear the interface contract of `IBxDF` has been extended).

5 COMPILER AND RUNTIME API

The previous section demonstrated how to use Slang language mechanisms to author an extensible shader library. However, a complete shading system also requires a runtime API to support renderer host code in composing and executing the shader effects in the library and preparing parameter blocks.

Listing 9 provides an overview of the API services that Slang provides. These services are grouped according to the main components of Slang's system implementation, shown in Fig. 2. In the first step, a renderer loads a shader library comprising one or more modules of Slang (or HLSL) code into the compiler's front end, resulting in IR code for those modules. Next the renderer uses the reflection API to access information about the types and entry points in the shader library. Importantly, the reflection API allows the renderer to create specializations of types on demand; e.g., to create the composite light environment types described in Section 4.3. In addition, a renderer may query layout information for any type, including specialized types, and use this information when allocating and populating parameter blocks. Optimized renderers might rely on hard-coded layouts (relying on the fixed algorithm used by the Slang compiler), but a dynamic reflection API is essential for supporting data-driven renderers and tools.

The compiler back-end provides an API to specialize an entry point for a particular set of type arguments, yielding platform-specific GPU kernels. In the example shading system in Section 2, this was performed by pasting HLSL strings of `#defines` and `typedefs` to create Listing 1.

The design of our runtime API was inspired by that of the Spire language used to support shader components [He et al. 2017]. In particular, Slang adopts the idea of allowing applications to load and introspect type layout of unspecialized code, so that API-specific parameter blocks can be allocated and filled in independently from the choice of entry points that may later be specialized to use those types. Slang extends this idea to also allow for the specialization of generic types (not just entry points), which enables applications to perform both fine- and coarse-grained composition.

Although we describe a "runtime" API, Slang does not enforce any policy as to *when* a renderer performs the tasks supported by its API services. The Slang runtime API can be invoked by a rendering engine to introspect and specialize shader code during offline asset processing, at load time, or on-demand during rendering (e.g., to lazily populate a shader cache or when "hot reloading" shader code).

6 REARCHITECTING A RENDERER TO USE SLANG

The overall goal of Slang is to facilitate productive development of large real-time shading systems without sacrificing renderer performance. As an initial step toward assessing whether this goal has been achieved, we have used Slang to refactor the shader library and shading system of the Falcor open source real-time renderer [Benty et al. 2017]. Falcor is a real-time rendering framework that aims to accelerate and support prototyping of new real-time rendering effects and algorithms. Although intended to be modular and extensible to support a wide variety of use cases, Falcor must also deliver high performance to support state-of-the-art real-time rendering effects. We forked Falcor version 2.0.2 for our evaluation, which featured 5,400 lines of shader code written in HLSL, implementing:

- A flexible, layered material system that can be configured to model complex reflectance functions
- Point, spot, directional, and ambient light types
- Glossy reflection using an environment map
- Cascaded, exponential and variance shadow map algorithms
- Post processing effects, such as screen space ambient occlusion and tone mapping

Falcor's material and lighting systems contribute over 2,100 lines of shader code and constitute the major fraction of CPU and GPU

TEMPLE BISTRO INTERIOR BISTRO EXTERIOR

(a) (b)

Fig. 3. (a) Scene viewpoints used for evaluating performance of the refactored Falcor renderer built using Slang (see Figure 4). (b) Support for polygonal areal lights was added to the Falcor renderer by changing a single code site.

execution time during rendering. Our refactoring focused on improving the extensibility, performance, and code clarity of these two subsystems.

Since Slang is an extension of HLSL, we were able to immediately compile the entire Falcor shader library using the Slang compiler (as a replacement for `fxc`). This allowed us to gradually refactor the Falcor shading system to incrementally adopt Slang's language features. We discuss important details of the refactoring experience below.

6.1 Using parameter blocks to communicate parameters

The existing Falcor shader library uses `struct` types to encapsulate related shader parameters, similar to our example shading system in Section 2. For example, Falcor defines material parameters (colors, textures, etc.) using a single global shader parameter:

```
MaterialData gMaterial;
```

For the reasons described in Section 2.6, similar to many engines ported to Direct3D 12, the original Falcor renderer allocates and fills a monolithic parameter block for each draw call that contains data for all shader parameters. Using Slang's explicit parameter block construct (Section 3.5) we were able to easily modify the shader library to use API-supported parameter blocks for materials:

```
ParameterBlock<MaterialData> gMaterial;
```

We also modified Falcor host code to allocate and fill in one parameter block for each material when loading a scene. By reusing the per-material parameter blocks across frames, we expect to reduce the CPU overhead of fetching and communication material parameter data for each draw call.

6.2 Specialization of Material

Falcor models a material as a combination of *layers*, where a layer defines a BxDF (e.g. Lambertian, Phong, GGX) and how its results should be blended with the next layer. By default, Falcor evaluates materials using dynamic looping over layers, and dynamic branching on layer type "tags."

In the common case, a material only includes a small number of layers in a fixed ordering, so the existing Falcor code includes support for specializing material code by passing a fixed list of layer tags as a preprocessor `#define`. Falcor looks up shader variants in a cache based on the set of "active" `#define` strings, including any material specialization `#define`. This lookup mechanism supports multiple subsystems using preprocessor-based specialization, but string-based lookup is a significant source of CPU overhead.

In our refactored shader library, we introduced an `IMaterial` interface, and entry points use a generic type parameter `TMaterial : IMaterial` to specialize to a selected material type (as shown in Listing 3). The definition of the material parameter block changed again, to:

```
ParameterBlock<TMaterial> gMaterial;
```

The existing Falcor `MaterialData` type, which uses `if` statements to select code for the appropriate BxDF for each material layer, was modified to implement the new `IMaterial` interface. In addition to porting the original material implementation, we added a new generic material type to achieve efficient specialization of the common cases, where a material is using a standard set of layers only:

```
struct StandardMaterial<const bool HasDiff, const bool HasSpec,
  const bool HasDielectric, const bool HasEmissive> : IMaterial
{...}
```

The `StandardMaterial` type can be statically specialized to include or exclude diffuse, conductor/dielectric specular, and emissive terms.

We then modified the Falcor's host-side `Material` class, which encapsulates a material, so that it creates a parameter block for either the original material type, or a specialization of the new `StandardMaterial` type (when the material features only the standard layers). Following the shader components design pattern [He et al. 2017], Falcor was modified to look up a specialized variant based on the types of parameter blocks bound to the pipeline. By giving each type a small integer ID, this lookup step can be implemented more efficiently than Falcor's previous string-based lookup using `#defines`.

The `IMaterial` type that we implemented in Falcor follows the approach in Section 4.2 to separate the phases of material shading using an associated type.

6.3 Refactoring Lighting Computation

To avoid the complexity of preprocessor solutions, Falcor's developers chose to not employ the static specialization approaches discussed in Section 2.4. Instead, the original Falcor implementation used dynamic branching to deal with different types of lights in a scene. The shader library uses a single HLSL type, `LightData`, to represent all supported light types. Similar to material layers, each light has a tag field that indicates its specific type (point, spot, directional), followed by a union of the fields required by the different cases. Light integration is performed by looping over an array of lights and dynamically dispatching to the correct logic based on the tag.

In contrast to the monolithic light type in shader code, Falcor's C++ code has a `Light` class with distinct subclasses for `PointLight`, etc. Our refactored shader library closely follows the design in Section 4.3, with a `ILightEnv` interface and distinct shader types for `PointLight`, etc. We also implemented the `LightArray` and `LightPair` composites.

Similar to material specialization, we defined a generic parameter `TLightEnv` and a parameter block using this type to replace the existing global array of lights. We modified the host C++ code to construct an appropriate composite lighting environment type (and corresponding parameter block) based on the lights loaded in a scene. By eliminating the existing "tagged union" design, it becomes simpler to extend the system with new light source types, as we show in Section 7.2.

7 EVALUATION

In this section we evaluate the performance and extensibility benefits realized by refactoring Falcor's shading system to use Slang's language mechanisms and compiler services.

7.1 Performance

The refactoring described in Section 6 should reduce the CPU cost of rendering (use of parameter blocks, fast specialized shader variant lookup) and preserve the same level of GPU rendering performance as renderer using shaders specialized to materials and lighting environment. Since the original branch of Falcor did not specialize shaders to lighting environments, to facilitate fair performance comparison, we forked the original branch and extended with support for light specialization using the approach discussed in Section 2.4. We compared the performance of all three branches of the Falcor renderer: the original branch (HLSL-based, without light specialization), the modified original branch (HLSL-based, with light specialization), and the refactored (Slang-based) branch. We use three test scenes from the ORCA asset library [NVIDIA 2017]: TEMPLE, BISTRO-INTERIOR, and BISTRO-EXTERIOR (rendered views shown in Fig. 3 (a)). These scenes were created by developers of the Unreal and Lumberyard [Amazon 2016] engines to demonstrate the capabilities of their engines and are representative of modern game content (e.g., the BISTRO-EXTERIOR scene has over 2.8 million triangles). Fig. 4 compares the performance of both renderers in terms of CPU time required to generate all GPU commands per frame (top) and GPU time to execute all these commands (bottom). We conducted experiments rendering 1920×1080 images on a machine with an Intel i7-5820K CPU and a NVIDIA Titan V GPU.

As expected, the refactored renderer realizes a notable reduction in CPU cost (over 30%) across all test scenes. Note that even if a renderer is not CPU-bound, reducing CPU costs frees up CPU resources for other game engine tasks. The GPU performance of the refactored renderer is on par with the original renderer with lighting environment specialization.

Adopting Slang adds overhead to shader compilation, because the Slang compiler outputs HLSL text that must be compiled by an existing HLSL compiler. When loading the TEMPLE scene, 1s is spent in Slang, while 4.5s is spent in HLSL compilation. This is slightly slower than the original (light specialization) branch, which

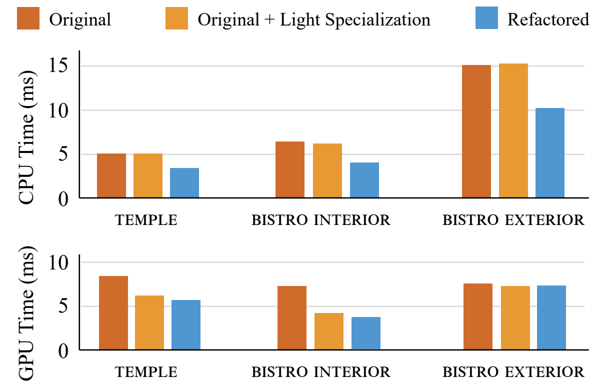

Fig. 4. Comparing average per-frame CPU time (top) and GPU time (bottom) of the original branches (with and without light specialization), and the refactored branch of the Falcor renderer, we find that the refactored renderer improves CPU performance and preserves the same level of GPU performance as the original branch with light specialization.

takes 4.1s to compile all the shader variants. Compile times could be improved in two ways. First, because Slang's language features are carefully chosen to support separate compilation, front-end work could be amortized across multiple entry points and variants. Second, and more importantly, we could eliminate the overhead of outputting HLSL and invoking a second compiler by directly translating Slang's IR to formats like SPIR-V and DXIL.

7.2 Extensibility

A major promise of Slang is that it will enable the design of a more extensible shading systems. To evaluate the extensibility of the refactored Falcor code, we added a new polygonal area light type to both the original and refactored shading system. Fig. 3 (b) shows a rendering of TEMPLE scene with a `QuadLight` type that uses linearly transformed cosines for approximate evaluation [Heitz et al. 2016].

As anticipated, the refactored shading system was significantly easier to extend. The following table summarizes the shader code changes required in each version of Falcor:

Branch	Sites Changed	Files Changed	Lines of code
Original	7	4	246
Original(w/ LS)	8	5	253
Refactored	1	1	249

Adding the area light feature to original code required changes at seven sites in the code, spanning four different files. These changes include: defining a new type tag for approximate area lights, adding new fields to the `LightData` types, inserting a new branch into the dispatch logic inside the main light integration loop, and adding logic to handle the new light type for each supported BxDF. Supporting shader specialization of lighting environments makes the original HLSL shader code even harder to extend, requiring one more change in one additional file. In contrast, adding area light support to the refactored Slang shader library was accomplished with a single block of code in a single file (a type definition plus `extensions`), and did not require modifying any existing Falcor functions or types.

We use the number of changed sites as a measure of code extensibility because it reflects the programming language's intrinsic capability of localizing a concern (independent of the file organization of the code base). We also report the number of files changed, since the file organization often reflects a developer's intention of modularity decomposition. Changing fewer files is an indication of better extensibility. Even though the total number of lines of shader code is comparable in all three branches, the refactored shader library handles two BxDF implementations (the original layered material and the `StandardMaterial` that we added) while the original code only deals with one.

7.3 Summary

By refactoring Falcor's shading system to utilize the language mechanisms and compiler services offered by Slang, we improved the renderer's CPU and GPU performance, as well as made the shader library code easier to extend. Qualitatively, we found that the refactored shader code reflects the mental model of the engine developers more explicitly and clearly. Although an ultimate evaluation of Slang's ideas will involve integration into a complex production game engine, our experiences integrating with the Falcor rendering system have been promising. Because of the benefits described in this section, *the Falcor project has now adopted the Slang shader compiler*, and is increasingly adopting Slang's language mechanisms in the shader library it provides to its application developers.

8 RELATED WORK

Previous efforts have attempted to improve shader modularity by adding more flexible dispatch mechanisms to real-time shading languages, including Cg interfaces [Pharr 2004], HLSL classes and interfaces [Microsoft 2011], and GLSL shader subroutines [Khronos Group, Inc. 2009]. These approaches use the syntactic form of dynamic dispatch, but support static specialization as an optimization performed by the language runtime or GPU driver. In contrast, Slang uses explicit generics syntax and the compiler implementation *guarantees* that static specialization is performed before code is passed to a GPU driver. Prior approaches do not include detailed discussion of the shading system design problems they seek to address; in contrast, Slang was specifically motivated by inspection of real shader systems and their challenges. None of the prior systems support associated types or retroactive extensions, which we found necessary to implement our modular and extensible shading system in Section 4.

The Vulkan API allows shaders to use compile-time constant parameters called "specialization constants" (Metal supports a similar feature). A specialization constant is left as an opaque value in the SPIR-V IR, and can be used in conditional control-flow decisions, and in determining the sizes of arrays of shader parameters. These systems are similar to Slang in that front-end compilation to IR can be performed once, and amortized across specializations. However, Slang allows type parameters in addition to values and uses pre-existing language constructs rather than new syntax.

The Sh shader metaprogramming system [McCool et al. 2002] supports the construction of abstractions like our surface shader separation (Section 4.2) and composite lighting environments (Section 4.3), using C++ templates; examples similar to ours can be found in the companion book for Sh [McCool and Du Toit 2004]. Sh is an embedded DSL which relies on runtime metaprogramming to generate its IR. In contrast, Slang is an extension of an existing shading language, and its generics can be statically checked at compile time.

Fig. 5. A scene rendered using GPU ray tracing in Falcor, which uses Slang to compile shaders for new ray tracing shader stages.

9 DISCUSSION

We have demonstrated that a popular real-time shading language can be extended with carefully chosen mechanisms from modern general-purpose languages, and that these mechanisms enable the development of a high-performance and extensible shader library without the need for layered preprocessor and DSL tools. By refactoring the Falcor shading system to use these mechanisms we were able to achieve improvements in CPU and GPU performance (by exploiting the shader components pattern) and also made the framework easier to extend with a new BxDF-dependent light type.

Although our evaluation of Slang was conducted in the context of rasterization and forward rendering, we believe Slang's features for code modularity and specialization also stand to benefit programmers using deferred rendering and ray tracing. For example, systems like DirectX Raytracing (DXR) [Sandy 2018] present the user with an increasing number of shader stages that will further complicate the implementation and maintenance of complex shading systems. Falcor has recently been extended to support GPU ray tracing with DXR, and uses Slang to generate shader code for ray tracing stages. Fig. 5 shows a ray-traced image rendered by Falcor using Slang shaders. Looking forward, we are interested to see how Slang's language mechanisms can benefit a shader library that may be used with both rasterization and ray tracing.

Slang's design demonstrates that it is possible to maintain performance and compatibility with existing HLSL codebases, while moving in the direction of modern general-purpose languages with strong support for good software development practices. We view this as one step toward an ultimate goal of enabling heterogeneous CPU and GPU programming in modern general-purpose languages. Rather than evolve HLSL, there is also parallel interest in adapting C/C++ for use in shader programming, and we hope our efforts serve to highlight the language features that are essential or preferred to support extensible and high-performance shading systems. We believe that all real-time graphics programmers could benefit from a new generation of shader compilation tools informed by these ideas.

10 ACKNOWLEDGMENTS

The research was supported by the National Science Foundation (IIS-1253530) and by the NVIDIA Corporation. Special thanks to the Falcor development team for useful discussions.

REFERENCES

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

Apple Inc. 2014a. *Metal*. https://developer.apple.com/documentation/metal

Apple Inc. 2014b. *The Swift Programming Language*. https://itunes.apple.com/us/book/the-swift-programming-language-swift-4-0-3/id881256329

F. O. Bartell, E. L. Dereniak, and W. L Wolfe. 1981. The Theory And Measurement Of Bidirectional Reflectance Distribution Function (BRDF) And Bidirectional Transmittance Distribution Function (BTDF). , 0257 - 0257 - 7 pages. https://doi.org/10.1117/12.959611

Nir Benty, Kai-Hwa Yao, Tim Foley, Anton S. Kaplanyan, Conor Lavelle, Chris Wyman, and Ashwin Vijay. 2017. The Falcor Rendering Framework. https://github.com/NVIDIAGameWorks/Falcor https://github.com/NVIDIAGameWorks/Falcor.

ECMA International. 2017. C# Language Specification (ECMA-334:2017).

Epic Games. 2015. Unreal Engine 4 Documentation. http://docs.unrealengine.com.

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 100 (July 2017), 11 pages. https://doi.org/10.1145/3072959.3073648

Eric Heitz, Jonathan Dupuy, Stephen Hill, and David Neubelt. 2016. Real-time Polygonal-light Shading with Linearly Transformed Cosines. *ACM Trans. Graph.* 35, 4, Article 41 (July 2016), 8 pages. https://doi.org/10.1145/2897824.2925895

Sony Pictures Imageworks. 2017. Open Shading Language 1.9 Language Specification. https://github.com/imageworks/OpenShadingLanguage/blob/master/src/doc/osl-languagespec.pdf.

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

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.

F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. 1992. Radiometry. Jones and Bartlett Publishers, Inc., USA, Chapter Geometrical Considerations and Nomenclature for Reflectance, 94–145. http://dl.acm.org/citation.cfm?id=136913.136929

NVIDIA. 2017. ORCA: Open Research Content Archive. http://developer.nvidia.com/orca.

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*. https://doi.org/10.1145/2776880.2787704 http://nextgenapis.realtimerendering.com/presentations/7_Pranckevicius_Unity.pptx.

Rust Project Developers. 2015. *The Rust Programming Language*. https://doc.rust-lang.org/book/.

Matt Sandy. 2018. DirectX Raytracing. Game Developers Conference 2018 slides.. https://msdnshared.blob.core.windows.net/media/2018/03/GDC_DXR_deck.pdf

Natalya Tatarchuk and Chris Tchou. 2017. Destiny Shader Pipeline. http://advances.realtimerendering.com/destiny/gdc_2017/index.html.

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

P. Wadler and S. Blott. 1989. How to Make Ad-hoc Polymorphism Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. ACM, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283