Slang – A Shader Compilation System for Extensible, Real-Time Shading

Yong He

CMU-CS-18-115

September 2018

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Kayvon Fatahalian, Chair
Jonathan Aldrich
Jim McCann
Tim Foley

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was sponsored by NVIDIA, Intel, Qualcomm, and the National Science Foundation under grant number IIS-1253530.

Keywords: shading languages, real-time rendering

For my parents.

Abstract

A modern production renderer may contain tens of thousands of code that define a diverse palette of shading effects. These effects model visual phenomena such as the light-scattering properties of materials, the animation of surfaces, or complex lighting environments. To support productive maintenance of the renderer codebase and the frequent addition of new shading features, it is desirable to implement these shading effects in a flexible and extensible framework that intuitively models the key rendering concepts. Unfortunately, shading system designs that offer high development productivity have historically not met the extreme performance requirements of real-time graphics systems. This thesis consists of a series of contributions that together form a blueprint for architecting modular, extensible real-time shading systems that also achieve state-of-the-art performance on modern CPU/GPU platforms. First, we established a set of shading system design principles, called *shader components*, which serve as a design pattern for realizing both intuitive decompositions of the rendering concepts and performance-critical global optimizations such as static GPU code specialization and efficient CPU–GPU communication. Next, we designed the Slang shading language and compilation system to facilitate implementation of shader components without the need for engine-specific or heavily preprocessor-based code generation techniques. Slang extends HLSL with general-purpose programming language mechanisms: generics with interface constraints, associated types, and interface/structure extensions, that we identify as necessary and sufficient language features for implementing modern shading systems. Last, we demonstrated how to rearchitect a large open source renderer using Slang's compiler services to adopt the shader components pattern. In this case study, we highlight the benefits of Slang's design by observing that the resulting shading system is substantially easier to extend with new features and achieves higher rendering performance than the original HLSL-based implementation.

Acknowledgments

During my school career, I have come across with many people who offered me a great amount of help and guided me to this important point of my life. I owe everyone a big thank-you.

At CMU, I received the most professional training and an incredible level of support from my advisor, Kayvon Fatahalian, for which I cannot be more grateful. Over the past years, he taught me how to think more profoundly and communicate more effectively as a researcher. His support has gone way beyond academic guidance and extends into creating countless opportunities for me to practice my skills and to connect with experts in the domain.

One of the most fortunate things in my career has been meeting Tim Foley. Kayvon introduced me to Tim during a research meeting in 2015, and he has since become my mentor. He is always excited to share his experience and expertise, and it is the most enjoyable thing in the world to discuss either technical or research problems with Tim. A lot of good ideas, including the key ideas in this thesis, sparked from our conversations.

I would like to thank Jim McCann and Jonathan Aldrich for serving on my thesis committee. Jim's questions led me to conduct a more in-depth study of expressing shader code for deferred and ray-tracing renderers, which enhanced my understanding of the problem. Jonathan's suggestions for a quantitative evaluation of the extensibility benefits helped me created a more convincing case study.

Thanks to Michael Coblenz, who helped me design a survey to collect developers' feedback on Slang. Thanks to the Falcor development team (in particular, Nir Bentley), Petrik Clarberg and Anjul Patney for sharing their experiences with Slang.

A great of number of people supported me during my PhD career. It is my honor to get to know Aaron Lefohn. I had the luxury of working twice as an intern in his research group at NVIDIA to learn the state of the art on real-time graphics. Discussions with Natalya Tartachuk, Peter-Pike Sloan, Wade Brainerd, Michael Vance and Angel Pesce enhanced my understanding of the shader compilation problems in the game industry and leads to successful research projects. Thanks to my officemates Ravi Mullapudi and Alex Poms for their help in making my presentations more accessible to general audiences. My PhD study is supported by NVIDIA Corporation, via both research funding and a graduate scholarship; Intel Corporation; and the

National Science Foundation.

It would not have been possible for me to start my PhD study at CMU without the help of Zhengping Qian, my mentor while I was an intern at Microsoft Research Asia. He offered me an opportunity to work on a system research project while I was an undergrad student. It is this experience that ignited my excitement in research and in pursuing a PhD degree. Zhengping also gave me massive support in my graduate school application.

Special thanks to Yanzhe Yang, who is always willing to listen to my thoughts and encourages me to overcome difficulties. Thanks to Yan Gu and Yihan Sun for all the happy and eventful memories in Pittsburgh.

Finally, I would like to thank my parents for their love and support. Your encouragement is the key to everything I have achieved today. Thank you!

Contents

List of Figures

List of Tables

List of Code Listings

Chapter 1

Introduction

Modern real-time graphics applications, such as video games and 3D visualization tools, are designed to produce realistic and high-resolution images of virtual scenes at high refresh rates. Fig. 1.1 shows two images rendered in Unreal Engine 4 [28], one of the most popular game engines. In order to produce images with this level of visual fidelity at 60 Hz (a typical refresh rate targeted by many games), the application must simulate a wide range of visual effects—including light reflection from various types of surfaces, shadowing, atmosphere scattering, human animation, high-resolution geometry details—and do so within a time budget of 16 ms for a smooth user experience. A sub-system of the renderer, called a shading system, is responsible for performing these operations. The role of a shading system is to compute the shape and visual appearance of each scene object in various environment conditions as viewed from a given camera angle. A shading system must model many real-world concepts, such as light, materials, surfaces, camera lens, etc., and must be capable of simulating an object's visual appearance as a result of a specific composition of these concepts.

Modern shading systems are complex software systems that support an extensive library of shading features that model a variety of real-world concepts. For example, Unreal Engine 4 contains over 70,000 lines of shader code that runs on the GPU [29]. Even a smaller scale research shading system, such as Falcor [14], contains over 15,000 lines of shader code.

Similar to any complex system, a shading system must be carefully architected to enable productive maintenance and extension of its features. To make the code base easy to understand, developers should architect a shading system with a modular decomposition following the mental model of the rendering problem (e.g. geometry transform, materials, lighting, etc.). To produce a wide variety of visual effects, the shading system framework should allow mixing and

Figure 1.1: Images rendered by Unreal Engine 4, one of the most popular game engines. Thanks to its large library of shading features simulating a wide range of physics phenomena, Unreal Engine 4 is capable of achieving high visual fidelity. Images are copyrighted to Epic Games.

matching different shading features (that are implemented as separate modules) when drawing an object. For productive maintenance, the system should be designed to allow graphics programmers to quickly add more shading features that simulate new types of natural phenomena without modifying or even understanding existing code.

To enable productive development, shading systems for many off-line film renderers [4, 6, 76] have been authored using object-oriented abstractions to model the graphics concepts, such as geometry, material and lights. These off-line renderers typically have a modular system architecture that is flexible and extensible. However, flexibility and extensibility are often at odds with the extreme performance demands in a real-time shading system. Many game developers go into great lengths to optimize both CPU and GPU code of a game engine's shading system. These optimizations come at the cost of software modularity, as they often involve eliminating the use of software design patterns or programming language mechanisms that incur runtime overhead, such as additional levels of indirection or dynamic dispatch.

Specifically, high-performance real-time shading systems feature aspects of the following key performance optimizations:

Accelerate drawing tasks using the GPU graphics pipeline. To draw objects with complex visual appearances in such a small time budget, graphics applications must offload all the drawing task to the GPU graphics pipeline. Using the graphics pipeline means that shading system developers must implement shading features' core computation logic as GPU *shader kernels* (referred to as *shaders* in this thesis), which are executed at different stages of the graphics pipeline, using C-like *shading languages* such as GLSL [46] and HLSL [63]. This poses greater challenges in modularizing and maintaining a heterogeneous code-base. Developers also are constrained by the GPU's performance characteristics, which requires the following commitments from a developer.

Specialize GPU code to avoid dynamic dispatching when possible. Typically, a shading system implements a large library of shading features, but only a small set of features is enabled when drawing any particular object. Existing object-oriented languages like C++ implement polymorphism via dynamic dispatching, which is not efficient enough on the GPU to meet shading systems' performance requirements. Since GPU kernel code runs millions of times for each vertex or pixel when drawing each object, repetitively running the same dispatching logic for millions of times is a significant overhead. Besides, modern GPU architectures exhibit reduced efficiency when executing large shader kernels with dynamic control flows to dispatch execution. For this reason, languages for writing GPU shader kernels (referred to as *shading languages*) do

not support dynamic object-oriented mechanisms (e.g. virtual function calls) for dispatching code, and developers are encouraged to generate and use *specialized* variants of shader kernels that contains only the logic of needed shading features when drawing each object. Due to the lack of object-oriented mechanisms in shading languages, authoring extensible and flexible shader code becomes a challenge. As we will describe in Chapter 2, system developers have been using various ad-hoc meta-programing or C-preprocessor based techniques to modularize and specialize shader code. These techniques often lead to degraded code maintainability, compromised performance, and increased system complexity.

Make CPU-GPU interoperation efficient. Using the GPU graphics pipeline offloads the main part of drawing logic to the GPU. However, the CPU still needs to compose drawing commands and transfer parameter data to the GPU. Although appearing simple, these tasks can become a performance bottleneck if not optimized properly. Consider a video game that draws approximately 10,000 objects per frame. Rendering at 60 Hz means that the game needs to draw 600,000 objects per second. That means the system has only 5,000 CPU cycles to dispatch commands and transfer data for each object when running on a main-stream 3 GHz CPU core, which is a tight time budget for these seemingly simple tasks. To prevent the CPU from being a performance bottleneck, the shading system should cache and reuse the drawing commands and parameter data required to draw each object, and minimize data transfer to the GPU.

As we will discuss in detail in Chapter 2, designing a modular and extensible system architecture targeting the real-time graphics pipeline is challenging. This is largely due to the lack of high-level language features in existing shading languages for authoring modular and efficient shader code that does not incur runtime overhead when executed on the GPU. Existing shading languages such as HLSL and GLSL are designed with a focus on exposing the hardware capabilities with a minimal amount of programming language constructs, instead of on providing necessary support for shading systems' high-level tasks such as dispatching GPU execution to different shading features and efficient CPU-GPU communication.

The limitations of existing shading languages present a barrier that prevent developers from adopting more efficient and productive shading system designs. We believe that the shading language should be redesigned with goals derived from a shading system's needs. Unfortunately, most shading system development teams, such as the rendering teams at Unity and Epic Games, are small groups with fewer than fifteen people and thus do not have the developer resources to develop a principled shading language. As a result, developers have come up with various ad-hoc code generation tools including preprocessor hacking, meta-programming, and proprietary domain-specific languages (DSLs) [39, 91], to fill the compiler gap for building a modular shad-

ing system. These ad-hoc solutions typically introduce trade-offs to balance the conflicting goals of facilitating developer productivity and achieving high rendering performance.

1.1 Contributions

This thesis consists of a set of contributions that together enable a shading system to meet its modularity, extensibility and performance goals. These contributions include a shading system design pattern; shading language and compiler support that makes efficient implementations of these system architectures possible; demonstration of adopting the new shading language features into an existing shading system; and an evaluation of the performance and modularity benefits that results.

Specifically, this thesis contributes:

1. A series of shading system design principles, called the *shader components design pattern*, that serve as a guideline to implement shading systems in a modular fashion that is consistent with a common mental organization of rendering concepts (e.g. geometry, material, lights and etc.), while retaining key performance optimizations such as shader code specialization and efficient shader parameter communication.
2. Identification of a minimal set of general language features, missing in existing shading languages, that allows the shader components design pattern to be implemented without relying on ad-hoc code generation tools or domain specific language constructs. Although shader components can be implemented with ad-hoc code generation techniques such as preprocessor hacking, these implementations often involve performance overhead and are prone to bugs. The need for ad-hoc compilers or special language constructs also presents a barrier towards wide adoption of the shader components idea. This thesis demonstrates that the same benefits of an ad-hoc code generation tool can be achieved via a set of general language features commonly available in modern programming languages such as Rust [82], Swift [9] and C# [25].
3. Design and implementation of the Slang shading language, a variant of the de facto standard shading language, HLSL [63], extended with the following general-purpose language features: generics with interface bounds, associated types and interface/structure extensions. These features represent a minimal set of extensions to meet our performance and productivity goals while providing an incremental path of adoption for current HLSL developers.
4. Definition of the Slang runtime API, which provides services for introspecting modules

of shading features and assembling collections of shading features into efficient, statically optimized GPU shaders. Instead of being a proprietary shader compiler designed for a specific shading system, Slang is designed to be used by different shading systems that make different decisions about shader optimization and execution (e.g., what effects to use, if and when to specialize shaders, when to communicate shader parameters). As such, Slang provides general mechanisms for defining shading effects and introspecting and compiling shader code, without presuming or imposing any specific shading system run-time policies.

5. Evaluation of the value of shader components idea and Slang's mechanisms through the creation of a large, extensible shader library implemented using Slang's features, and rearchitecting a large open source research renderer [14] to use this library and Slang's run-time API. We show via quantitative and qualitative evaluations that the resulting shading system implementation (both the shading library itself and the CPU-side renderer "host" code to compile and execute shaders) is more aligned with developer's mental model of graphics concepts and easier to extend with new features, while achieving better performance over the original HLSL-based implementation.

We hope that the findings in this thesis will serve as a useful guideline for developers to architect future shading systems and that the shader compilation system we have designed will liberate developers from implementing ad-hoc shader compilation tools.

1.2 Thesis Roadmap

Chapter 2 first provides a brief introduction to the GPU graphics pipeline and the shading system for readers who are not familiar with real-time graphics rendering. It then demonstrates the challenges, workarounds, and compromises in designing and implementing a simple shading system using existing shading languages.

Chapter 3 introduces the idea of shader components, which is a shading system design pattern for achieving system extensibility and flexibility, and enabling performance optimizations.

Chapter 4 presents the Slang shading language. This chapter discusses the key language mechanisms in Slang and its associated runtime API that enable implementation of shader components.

Chapter 5 provides a case study of adopting Slang and the shader components design pattern in a research rendering framework. This chapter demonstrates how Slang's language features

can be used to implement shader components, and presents an evaluation of the extensibility and performance benefits in the rearchitected code base.

Chapter 6 discusses the relationship of this thesis to previous works on shading language and shader compiler design.

Chapter 7 summarizes the key decisions made while designing Slang, discusses the limitations of Slang's current implementation, and proposes future directions for improvement.

Chapter 2

Background

The demand for extreme performance in real-time graphics applications has led to the development of highly specialized hardware architecture: the real-time graphics pipeline. Most of the unique challenges in designing and implementing a real-time shading system stem from the need to use the GPU implementations of the graphics pipeline efficiently and without sacrificing the flexibility and extensibility of the shading system. This chapter provides the necessary background to illuminate how these challenges arise. The chapter begins with an introduction to the graphics concepts and shading features that need to be computed by a shading system, followed by an introduction to the graphics pipeline architecture and the software constraints for achieving high performance on the GPU. With these basics explained, we work through an example shading system design to explain why it is challenging to achieve the modularity, extensibility and performance goals using existing APIs and programming languages for the GPU graphics pipeline.

2.1 Shading System Concepts

This section briefly introduces the basic graphics concepts in shading systems. For clarity, we omit many complex features in a production renderer that are not essential to the discussion of the challenges in designing a shading system.

The goal of a renderer is to draw a picture of many different objects, such as the image shown in Fig. 2.1. The input to a renderer is a list of objects to render. For each object, the renderer's shading system must compute a set of *shading features* to determine its shape and color. By computing different combinations of shading features, the shading system can produce a variety

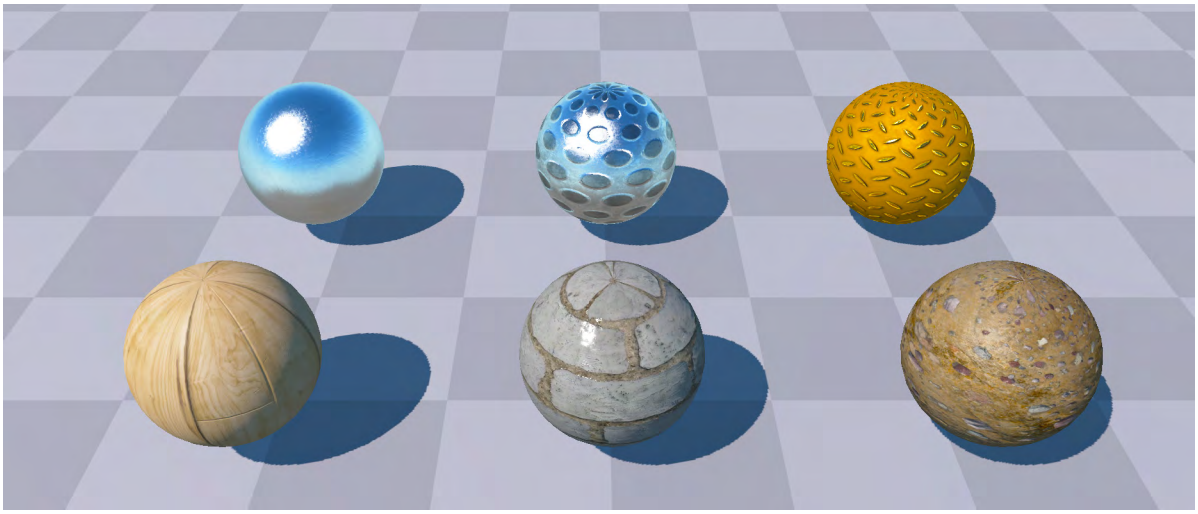


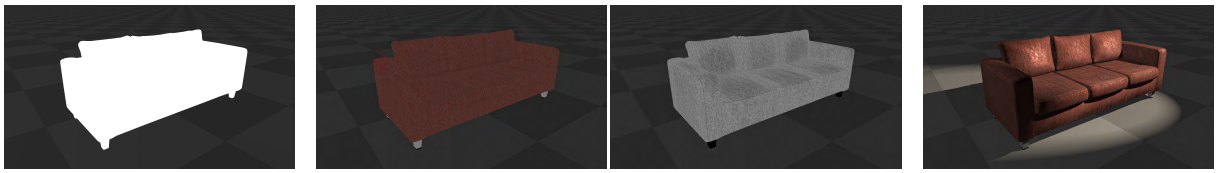
Figure 2.1: The input to a renderer is a list of objects to draw. For each object, a renderer's shading system evaluates a different combination of shading features to create its unique appearance.

(a) (b)

Figure 2.2: The GUI tools in Unreal Engine 4 reflects a typical graphics engineer's mental decomposition for the shading problem. (a) Partial screenshots of Unreal Engine 4's level editor show different types of geometry and lights. (b) Unreal Engine 4's material editor allows a user to define object's visual appearances.

of unique visual appearances for different objects. This thesis focuses on three of the most common types of shading features computed at different stages: shape and geometry, materials, and lighting. These are key concepts that reflect a typical graphics engineer's mental model for the shading problem, and similar decompositions can be seen in offline renderer implementations such as PBRT [76], Pixar's RenderMan [6], and Arnold [4], as well as in GUI tools for games,

such as Unreal Engine's level editor (Fig. 2.2).



(a) shape and geometry (b) material shading (base color and roughness properties) (c) light integration

Figure 2.3: Rendering results of a couch after computing different shading features.

For example, the spheres in Fig. 2.1 are rendered using the same lighting feature, but different material features that model the appearances of metal, wood, brick and dirt. From left to right, Fig. 2.3 illustrates results after three major steps of shading: (a) geometry transform, (b) material shading, and (c) light integration, which we describe in detail below.

2.1.1 Evaluating Surface Position

The first step of rendering an image is to compute the shape of an object and figure out where on the final image the object should appear. For objects that have a static shape (e.g. terrain, buildings, and furniture), the shape is usually defined directly by a triangle mesh. For deformable objects, such as water surface or human characters, the shading system needs to update the shape of the object every frame by simulating physics models [13, 41] or running shape deformation algorithms such as shape interpolation and skeletal animation [55]. Once the shape is computed, it is projected to the screen using a geometry transformation (which takes into account the camera position, view direction, and a field-of-view angle), producing an image as shown in Fig. 2.3 (a). The projected shape of the couch appears correctly in the rendered image, but the image shows no details of the couch surface. All the pixels within the couch's outline are painted white because we have not yet included any shading features to determine the color of each pixel.

2.1.2 The Rendering Equation

After determining the shape of an object, the renderer computes the object's visual appearance, or the "feel" of its material (e.g. wood, plastic or metal) by evaluating the amount of light reflected by each point of the object's surface into a virtual camera. This is done by evaluating the rendering equation [45]:

$$L_o(\boldsymbol{x}, \omega_o) = \int_\Omega f(\boldsymbol{x}, \omega_i, \omega_o) L_i(\boldsymbol{x}, \omega_i)(\omega_i \cdot \boldsymbol{n}) \, \mathrm{d}\omega_i \tag{2.1}$$

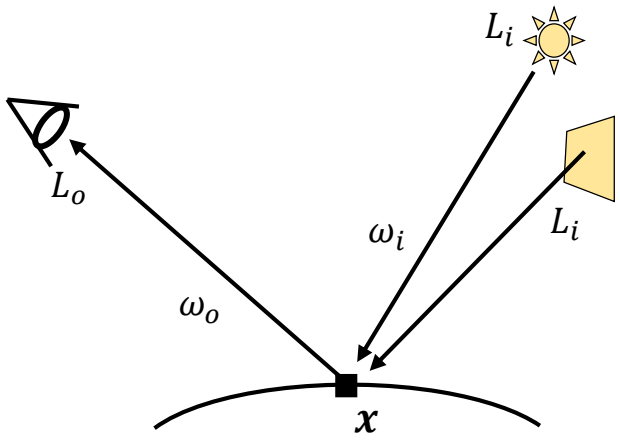


Figure 2.4: Explanation of rendering equation. L_o is the amount of light (radiance) reflected by the surface point into the camera. ω_o is the view direction, $\boldsymbol{x}$ is the location of the surface point where appearance is being computed. ω_i represents an incoming light direction, $L_i(\boldsymbol{x}, \omega_i)$ is the incoming radiance along direction ω_i, and $\boldsymbol{n}$ is the normal vector of the surface. f is a BxDF.

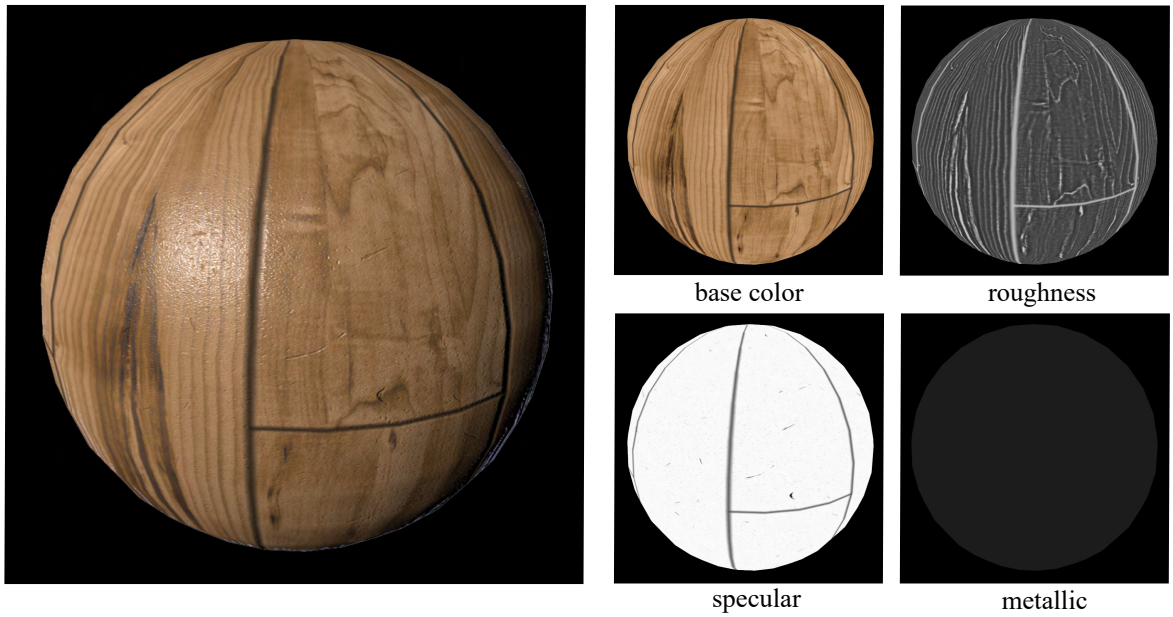
Fig. 2.4 illustrates different terms of the rendering equation. L_o is the amount of light (radiance) reflected along direction ω_o into the eye. ω_o is the view direction, i.e. the direction between the surface point and the camera) and $\boldsymbol{x}$ is the location of the surface point where appearance (color) is being computed. ω_i represents an incoming light direction, $L_i(\boldsymbol{x}, \omega_i)$ is the incoming light(radiance) along direction ω_i, and $\boldsymbol{n}$ is the normal vector of the surface.

This equation has three parts. First, the shading system must determine L_i, the amount of light energy (radiance) traveling onto surface position $\boldsymbol{x}$ from direction ω_i.

The reason for different types of surfaces having distinct appearances is that they reflect different amounts of incident light. The second part of the equation, the f term, models the reflection ratio of a surface. In graphics, f is called a *Bidirectional Scattering/Reflectance/-Transmittance Distribution Function*, or a BxDF. A BxDF defines the ratio of light hitting the surface from direction ω_i that is being scattered/reflected/transmitted onto direction ω_o.

The last part of the equation is the integral called *light integration*. The final color of surface point $\boldsymbol{x}$ is computed by integrating the light energy coming from all incident directions that are reflected onto view direction ω_o.

For a more detailed description of the rendering equation, we refer the readers to PBRT [76] and the text book "Fundamentals of Computer Graphics" [88].



(a) rendering result (b) visualization of BxDF parameters

Figure 2.5: Close-up view of a sphere rendered with a wood appearance. (a) The final rendering result of the sphere using material and lighting features. (b) A visualization of the BxDF parameters computed in the material shading step (base color, roughness, specular, metallic).

A typical shading system evaluates the rendering equation in three steps.

Material Shading

Graphics researchers and developers have proposed many BxDFs that model different surface types with various accuracy and performance trade-offs. In practice, Lambertian, Phong [77], Blinn [17], GGX [96] and Disney [21] BxDFs are most frequently used in real-time shading systems. All of these BxDFs can be configured with additional parameters to model different physical materials. These parameters characterize the properties of a surface (e.g. base color, roughness and opacity) and are invariant to the lighting environment (ω_i and L_i). As such, most shading systems compute these lighting-environment-invariant parameters as the first step of shading, which is called material shading, or surface pattern generation. The result of this step is the BxDF term f. Note that this step does not evaluate the BxDF by plugging actual values into f. Instead, it computes surface-specific parameters that will be used in the BxDF. In other words, this step returns a higher order function, or a closure that represents f, which will be used later when computing the light integration. This step can be expressed as the following pseudo-code:

```
func f = evalMaterial(x)
```

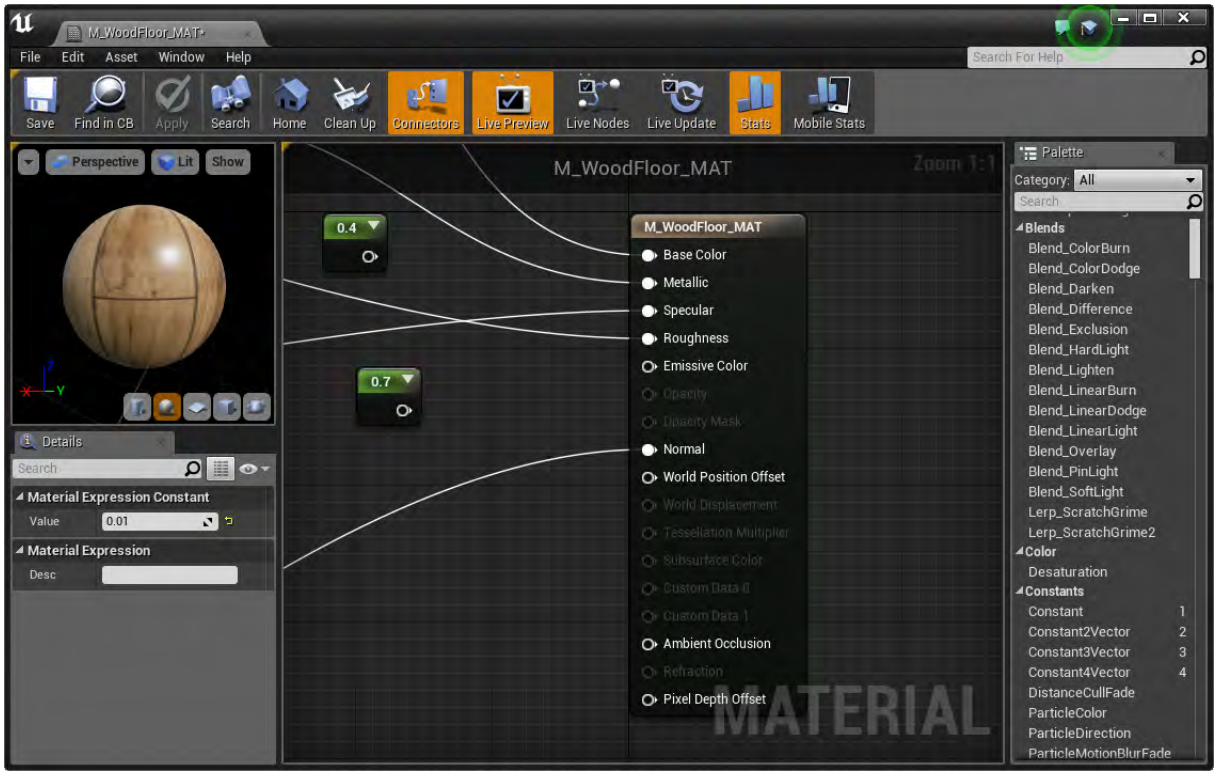


Figure 2.6: The material editor in Unreal Engine 4 allows an artist to select a BxDF and specify how its parameters are computed in material shading step.

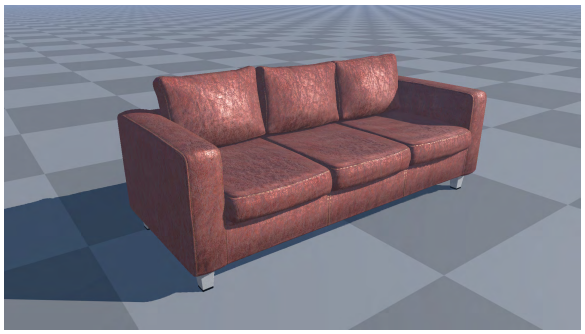
Fig. 2.5 (a) shows an example of a sphere rendered with a wood appearance, using a simplified implementation of the Disney BxDF [21] featuring four parameters: base color, roughness, specular and metallic. The parameter values used at each pixel are visualized in Fig. 2.5 (b). By providing using BxDF parameters at different surface locations, the artist creates the details of imperfection such as dents and scratches. As another example, Fig. 2.3 (b) visualizes two BxDF parameters in the couch's material: base color and roughness. These parameters reveal the leather pattern, the seams between patches of leather, and wear on the cushion.

In many shading systems, the combination of a BxDF and its parameters is called a *material*. Modern AAA video games include tens of thousands of unique materials, which defines the detailed appearance for each type of scene object. For example, Bungie's Destiny [20] features approximately 18,000 unique materials. To aid the development of this many materials, many shading systems (such as Bungie's shader system and Unreal Engine 4) feature a GUI material editor. See, for example, the Unreal Engine 4 material editor (Fig. 2.6) that enables artists to select a BxDF and create surface pattern for the BxDF by mixing and matching reusable building blocks from a predefined library.

(a) Spotlight

(b) Directional light

(c) Sky light



(d) Directional light + Sky light

Figure 2.7: Renderings of the couch object illuminated by different types of light sources. (a) A spot light. (b) A directional light representing the sun. (c) A sky light. (d) A combination of the sky light and directional light, a common outdoor lighting setup.

Light Shading

The second step is to compute the amount of light energy traveling towards the surface point $\boldsymbol{x}$ from all directions. For faster performance, it is common to consider only the light traveling directly from an infinitesimal light source (e.g., point light and directional light) onto the surface point (called *direct lighting*). In this case, the rendering equation is simplified into a discrete sum over directions from all light sources in the scene:

$$L_o(\boldsymbol{x}, \omega_o) = \sum_i f(\boldsymbol{x}, \omega_i, \omega_o) L_i(\boldsymbol{x}, \omega_i)(\omega_i \cdot \boldsymbol{n}) \tag{2.2}$$

The light shading step computes radiance emitted from each light source. This step can be expressed as the following pseudo-code:

```
for k = 0 : N
    Li[k], Wi[k] = lights[k].illum(x)
```

Modern shading systems support many types of light sources and can compute lighting from

an arbitrary collection of light sources. By setting up light sources, artists can create a diversity of scene atmospheres. Fig. 2.7 shows the same couch object illuminated by different types of lights.

The most commonly used light sources in real-time shading systems are *point lights*, which emit light into all directions from a single point in the scene, *spot lights*, which emit light along a small cone from a single point, *directional lights* which emit light in the same direction to everywhere in the scene (e.g., the sun), *area lights*, which emit light from a surface (e.g., a computer screen), and *sky lights*, which model the entire sky (excluding the sun) as a light source.

Light Integration

The final step in computing an object's appearance is light integration. In a simplified shading system that considers only direct lighting, this step repetitively calls the BxDF and sums up the reflected light energy from each light source. In code, this is represented as:

```
for k = 0 : N
    Lo += integrate(Li[k], f, Wi[k], Wo);
```

2.2 The Graphics Pipeline

As previously mentioned, a real-time shading system needs to use the graphics pipeline to draw objects efficiently. In this section, we provide a brief introduction to the graphics pipeline and how the shading computations are executed in an end-to-end system. Readers may refer to the OpenGL [87] and Direct3D [64] documentation for a full description of the graphics pipeline and how it can be programmed.

At a high level, the GPU graphics pipeline can be viewed as a virtual machine that executes two types of instructions: modifying the state of the machine (referred to as a *state-change command* in this thesis), and drawing an object on an image (a *draw command*). A state-change command, as the name suggests, changes the states kept by the virtual machine that configures how an object will be drawn, such as which shader kernels to use. A draw command takes as input the geometry data defining the shape of a scene object (usually a set of triangles), and modifies the content of an image as a result of drawing the object.

Similar as in a pipelined CPU architecture, execution of a draw command is divided into multiple pipeline stages on the GPU to enable pipelining parallelism over multiple draw commands. Fig. 2.8 illustrates the stages of a modern GPU graphics pipeline architecture (as defined

The GPU Graphics Pipeline

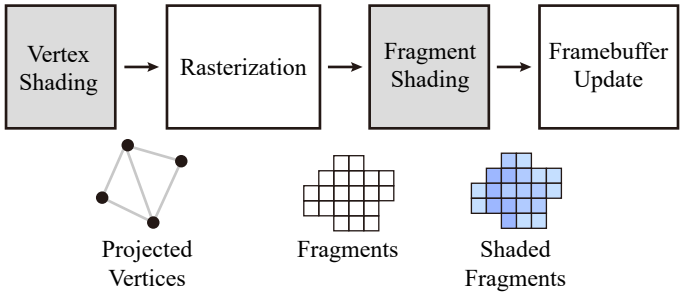


Figure 2.8: The GPU graphics pipeline architecture. Pipeline stages are shown in square boxes. Boxes in gray background represent programmable pipeline stages, and boxes in white background represent fixed-function stages.

by Vulkan [49], Direct3D 12 [64] and OpenGL 4.5 [87]). This figure omits the Input Assembly, Geometry Shader, and Tessellation stages for simplicity, because they do not affect the discussion of challenges in a real-time shading system. The stage names established in this figure will be used consistently throughout this thesis.

The graphics pipeline can be configured to include different sets of stages. In its simplest form (as shown in Fig. 2.8), the pipeline has four stages: Vertex Shading, Rasterization, Fragment Shading and Framebuffer Update. A draw command provides a stream of triangle vertices as input. The vertex shading stage runs an application provided kernel function (called *vertex shader*) to map each input vertex (often in 3D space) to a 2D position on the final image. The Rasterization stage then generates a set of image pixel locations (called *fragments*) that are covered by each triangle, using the projected vertex positions computed in the vertex shading stage. Next, the Fragment Shading stage uses another application provided kernel function (called *fragment shader*) to compute the color for each fragment. Finally, the shaded fragments are used to update the image in the Framebuffer Update stage. Note that both the Vertex Shading and Fragment Shading stages execute application-provided shader kernels, and they are called *programmable stages*, while the Rasterization and Framebuffer Update stages do not execute application code, and they are referred to as *fixed-function stages*.

In a real-time shading system, the shading features described in Section 2.1 are evaluated in programmable stages of the graphics pipeline. For example, animation and geometry transform is done in the Vertex Shading stage and executed once for each vertex; this provides the graphics pipeline the projected screen-space coordinates of the object's shape. Surface pattern and lighting is typically computed in the Fragment Shading stage and executed once for each pixel occupied

by the object, because the surface property values stored in a BxDF closure vary across different locations of the surface. Since vertex and fragment shader kernels are being executed many times for each vertex and fragment, it is critical to ensure they are efficient code.

2.2.1 Programming Shader Kernels

Shader kernels executed in the Vertex Shading and Fragment Shading stages are expressed as C-like functions in shading languages such as GLSL [46] and HLSL [63]. The signature of a shader kernel is defined by the pipeline stage. For example, a vertex shader kernel is a function that takes as input one vertex from the input geometry data, and outputs one transformed vertex. A fragment shader kernel function takes interpolated vertex shader output at a fragment location and computes the values that should be written to the destination image at that fragment location.

Shader kernels for different pipeline stages are linked together to define the *shader program* that can be set as a part of the pipeline state. This thesis uses the term *bind shader program* to mean configuring the programmable pipeline state to use shader kernels defined by the shader program.

2.2.2 Managing Pipeline States

The pipeline must be properly set up before it can correctly draw an object. The programmer must tell the GPU not only which shader program to use, but also where to fetch the input geometry data, and what parameter values to use when running the shader kernels. In this thesis, we use the term *pipeline states* to refer to all the parameters and configurations required by the graphics pipeline to draw an object.

2.2.3 Using the Graphics Pipeline

Applications control the graphics pipeline via standardized graphics APIs, such as OpenGL [87], Direct3D [64] and Vulkan [49]. These APIs provide interfaces to send state-changing and drawing commands to the graphics pipeline. In this thesis, we use the term *bind shader program* to refer to setting the shader program to use by the graphics pipeline, and the term *bind shader parameters* to mean configuring the pipeline state to use the specified shader parameter values when executing a shader kernel.

To draw an object using the graphics pipeline, the application need to perform the following

operations:

1. Bind shader program, which sets the shader kernels for all the programmable stages.
2. Specify the memory location of the input geometry data.
3. Bind shader parameters, which provides concrete parameter values to the shader kernels.
4. Send a draw command.

2.3 GPU Architecture Characteristics

So far, we have discussed the types of shading computations and where they run within the GPU graphics pipeline. To achieve high performance, the shading computation must map to code that runs efficiently on the GPU. Writing high-performance GPU code requires understanding the underlying GPU architecture that implements the real-time graphics pipeline. In this section, we discuss three key performance characteristics of GPU architectures that have a significant impact on shading system design.

2.3.1 Prefer Static Specialization Over Dynamic Dispatch

Drawing a single object typically requires running tens of thousands of vertex shader instances(to process each input vertex) and up to millions of fragment shader instances. Since the vertices and fragments are processed by the same program, a GPU accelerates the execution of shader programs by using a wide SIMD architecture, with each SIMD lane processing one vertex or fragment. For example, a single hardware thread on NVIDIA's Pascal GPU architecture features 32 SIMD lanes [69].

GPU's wide SIMD architecture is less efficient when executing code with a *divergent control flow*. For example, consider the following code in a fragment shader that is running concurrently on 32 SIMD lanes of a GPU core:

```
if (c) x(); else y();
```

If the value of `c` is not the same across all SIMD lanes, both branches need to be executed for the SIMD group. To ensure correctness, GPUs must keep track of the execution state (program counter and active mask) for each SIMD lane; this incurs overhead on some platforms [30, 68].

In addition to the wide SIMD architecture, the GPU relies on hardware multi-threading to hide memory latency [30]. Each GPU core maintains the execution context (registers) for multiple threads on the chip. When the currently executing thread is blocked by memory access,

the core switches context to execute the arithmetic instructions in another thread. To effectively hide memory latency, the context switching must be a lightweight operation and should not incur additional main memory accesses. Therefore, modern GPUs keep the execution contexts of a small number of threads (e.g., sixteen 32-wide SIMD threads) in the limited on-chip memory. If a shader kernel requires a large execution context (i.e., has many live local variables), the GPU's on-chip memory will not be able to hold as many execution contexts, thus reducing its capability to hide memory latency effectively and resulting in degraded performance.

For these reasons, a common implementation practice called *uber shader*, where all computations are written in a single big GPU kernel dispatched via a big `if` statement, is considered harmful for performance [53]. To avoid the overhead of dynamic branching, developers always strive to avoid using dynamic control flows in the shader code. This means that shading systems should use *specialized* shader code for each object, containing only the code branches required in drawing the object. For similar reasons, modularity constructs that involve dynamic dispatching, such as virtual functions, should also be avoided.

2.3.2 Deep Pipelining

GPU architectures are highly pipelined, executing draw commands in different hardware stages similar to the logical stages in the graphics pipeline abstraction described in Section 2.2. Pipelining enables the GPU to execute multiple draw commands at a time. However, this also means that changing some pipeline states, such as the shader kernel or certain types of shader parameter values, may cause stalls in the pipeline and reduces GPU efficiency.

In addition to incurring costs in GPU time, changing GPU states has costs in CPU time. These CPU costs include time spent by the application and driver to prepare and send hardware commands [86]. Both kinds of costs can negatively impact frame rates, depending on whether an application is CPU- or GPU-bound.

As such, shading systems must reduce state changes, by reordering draw commands to exploit the state coherence between draws.

2.4 An Example Shading System Design

Due to the GPU performance characteristics we have discussed in the previous section, real-time shading systems must must perform key optimizations of shader specialization and state

change (parameter communication). In this section, we describe an example shading system that implements the mental model of image synthesis using shapes, materials, and lights, as described in Section 2.1, and show the challenges in achieving performance goals. The example shading system is architected with design patterns observed in current AAA game engines. This section provides detailed background on the tasks a modern shading system must perform. It also illustrates 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 Chapter 5 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 this section.

The purpose of a shading system is to draw a scene containing many objects that use different combinations of shading features. Recall the scene shown in Fig. 2.1, which consists of six spheres using different material features. A typical scene in AAA video games of 2018 contains anywhere from tens of thousands to hundreds of thousands of objects.

Since a shading system uses the graphics pipeline for drawing objects, it comprises both GPU code for a *shader library* (defining shading features such as geometry transform, material shading, and light integration, etc.) and the CPU *host code* responsible for preparing and invoking GPU work (compiling and executing shaders, communicating shader parameters to the GPU). To achieve the desired performance level, both the GPU code and the CPU code must run fast. Due to reasons discussed in Section 2.3, the shading system should use specialized GPU shader code that implements only the required shading features when drawing each object in order to ensure efficient GPU execution. The shading system should also seek to communicate parameters with minimum CPU overhead and CPU-GPU bandwidth usage.

Extensibility is another important requirement of shading systems. We consider a real-time shading system (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 such as a new type of material or light without requiring modification of the host code or shader library of the renderer.

Fig. 2.9 and Fig. 2.10 provide an overview of key pieces of the example shading system, which uses C++ for host code and HLSL for the shader library. The example is kept simple for clarity. Real-world shading systems are much more complicated in terms of numbers and types of shading features, but the system architecture and the implementation challenges presented in this example are representative of state-of-the-art systems.

Figure 2.9: An example shading system architecture using HLSL and C++. The framework provides modules that comprise both HLSL `struct` types and corresponding C++ `classes`. The marshaling of data from CPU to GPU is managed explicitly by the `bindParams` methods on the C++ objects. A light type's `evalLight` function is called by the `illuminate` function, which is defined in Listing 2.3.

Figure 2.10: An application using the example shading system, which extends the framework with new features written as HLSL/C++ modules, and specifies the features to use in a rendering operation by instantiating and filling in C++ objects.

Fig. 2.9 lists code provided by the shading system framework, while Fig. 2.10 lists code specific to a particular application (specific materials, light sources, etc.). Note that the code base is modularized to match the same mental model of the shading process discussed in Section 2.1, with separate modules for cameras, materials and lights. This example shading system is architected to prioritize extensibility and performance. As will be discussed in detail, the example shading system is efficient since its architecture allows the renderer to statically specialize GPU shaders to exactly the features in use.

2.4.1 Authoring a Modular Shader Library

To aid developer productivity, shading system features should 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 [28].

While the camera feature has only a single implementation in Fig. 2.9, other features of the shader system support multiple implementations — notably, materials and light sources. In C++, such choices can be 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; For clarity Fig. 2.9 and Fig. 2.10 use color to group types by feature: camera, materials, and lights.

GPU shader execution begins with an *entry point* function, such as the `entryPoint` for the fragment shading pipeline stage. We omit the entry point for the vertex shading stage since most interesting computations are done in the fragment shading stage, and all issues are similar between these two stages. `entryPoint` 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, `entryPoint` invokes the material shading step (Section 2.1.2) for a surface material feature and then invokes `illuminate` to integrate lighting using the lighting environment defined by `gLights`.

```
#include "MyMaterial.hlsl"
typedef MyMaterial Material;
typedef MyMaterialBxDF BxDF;
#define PointLightCount 1
#define QuadLightCount 1
#include "LightEnv.hlsl"
#include "EntryPoint.hlsl"
```

Listing 2.1: HLSL code that specializes the `entryPoint` function of the shader system in Fig. 2.9 and Fig. 2.10 to use the material defined in `MyMaterial.hlsl`. It also uses `#defines` and `typedefs` to specialize the entry point's lighting code (provided in Listing 2.3) to use a single point light and an area light. Colors correspond to interactions with features in the example shading system.

2.4.2 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 when drawing each object. 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 `entryPoint` is written abstractly in terms of types (`Material`, `LightEnv` and `BxDF`) and functions (`evalMaterial`, `illuminate`) that it does not define. Listing 2.1 shows an example of how a variant of `entryPoint` 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 the text of the forward pass entry point. A similar approach to specialization appears in the shader library for the Lumberyard engine [3].

The specialization in Listing 2.1 could be generated by a renderer's host code by pasting together strings of HLSL according to the shading features a scene object requires (a simple form of metaprogramming). Listing 2.2 shows the pseudocode for this metaprogramming process. First, it queries the name of the HLSL type corresponding to a material in use via the virtual `getTypeName` operation (line 6) on a C++ `Material` instance, then it uses this string to generate the appropriate `typedef` lines (lines 7-9). Similarly, the number of each type of light source in-use is pulled from a `LightEnv` instance, which is then used to generate the `#defines` for lighting code specialization (lines 12-13).

```
 1: string getSpecializedShaderCode(Material* material, LightEnv* lightEnv)
 2: {
 3:     std::stringstream sb;
 4:
 5:     // material
 6:     string materialType = material->getTypeName();
 7:     sb << "#include␣\"" << materialType << ".hlsl\"\n";
 8:     sb << "typedef␣" << materialType << "␣Material;\n"
 9:     sb << "typedef␣" << materialType << "BxDF␣BxDF;\n";
10:
11:    // lighting environment
12:    sb << "#define␣PointLightCount␣" << countPointLights(lightEnv) << "\n";
13:    sb << "#define␣QuadLightCount␣" << countQuadLights(lightEnv) << "\n";
14:
15:    sb << "#include␣\"LightEnv.hlsl\"\n";
16:    sb << "#include␣\"EntryPoint.hlsl\"\n";
17:
18:    return sb.str();
19: }
```

Listing 2.2: Pseudocode for generating a specialized shader variant as in Listing 2.1. The code collects material type names from the input `Material` instance as well as lighting environment compositions, and uses this information to paste together an HLSL code that represents a specialized shader variant for the given material and lighting environment.

The approach to specialization illustrated in Listing 2.1 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 `evalMaterial` with a unique signature so that, e.g., when `entryPoint` calls `evalMaterial`, type-based overload resolution by HLSL statically dispatches to the appropriate code (based on the type of the `gMaterial` argument). The HLSL file defining a material must be named `<MaterialType>.hlsl`, and must include a struct named `<MaterialType>` that encapsulates the shader parameters for the material shading logic, and a `<MaterialType>BxDF` type encapsulating the BxDF parameters returned by `evalMaterial`. The connection between the `evalMaterial` call site in `entryPoint` and the material shading code for `MyMaterial` is not explicit, nor is the material functionality expected by `entryPoint` explicitly defined or enforced in the code. *Instead, realizing a valid HLSL shader is a matter of adhering to a shading system's policy*. If an entry point requires an operation that some, but not all, materials support, no error will be raised until the shading system 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 such as Lumberyard [3]. 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.

2.4.3 Separating Phases of Material Shading and Light Integration

As mentioned in Section 2.1.2, physically-based shading systems typically separates the evaluation of surface materials into distinct phases for material shading and lighting integration. In general, the result of material shading is a function closure consisting of a BxDF and surface property values used as parameters to the BxDF, which together define how the surface interacts with light. This function closure is called in the lighting evaluation phase along with additional parameters from light sources to compute the final appearance. This separation can be seen in OpenSL [42], where a surface shader expresses material shading (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. Separating material shading from light integration is good for performance, since the material shading phase computes the intrinsic surface properties that are independent of the lighting environment, which can then be reused when accumulating lighting contributions from multiple light sources.

Because each BxDF requires a unique set of input parameters, material surface shaders that use different BxDFs will use different types to store the parameter values provided by material shading. In the `entryPoint` function shown in Fig. 2.9, the `BxDF` type is used to represent the BxDF closure produced by material shading.

Notice that the choice of `BxDF` type in Listing 2.1 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 example system. The `BxDF` type is defined via a chain of two `typedef`s representing two choices: the shader code for `MyMaterial` in Fig. 2.10 selects the concrete `Lambertian` BRDF as the result of its `evalMaterial`, and the specialization logic in Listing 2.1 selects `MyMaterial` for use by `entryPoint`.

As in Section 2.4.2, the example system's design achieves extensibility with new features (material shading and light sources), but relies on implicit system policies for how features must be authored.

2.4.4 Specialization to Lighting Environment

In addition to specializing to a material shading feature, it is also desirable to statically specialize GPU shaders to the structure of a lighting environment — for example, per-object light lists in a forward renderer, or per-tile light lights in a deferred renderer. Although material specialization typically must consider only 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 2.3 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 high performance 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 show only 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. 2.9 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 2.1).

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. 2.10, 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 2.1. 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 2.3 specialized for a specific composite lighting environment. The use of code generation enables extensibility, but further increases the complexity of the shading system relative to the preprocessor-based solution.

2.4.5 Adding BxDF-dependent Light Types

The light loops in Listing 2.3 may include different code for each light type. For example, rather than sampling incident illumination along a single ray (as done for point or directional light

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

#include "PointLight.hlsl"
// ...
struct LightEnv
{
#if PointLightCount
  PointLight pointLights[PointLightCount];
#endif
  // ...
}

float3 integrate(BxDF bxdf,
                 LightSample s,
                 float3 wo)
{
  return s.Li * evalBxDF(bxdf, s.wi, wo)
              * max(0, dot(s.wi, wo));
}

float3 illuminate(LightEnv env, BxDF bxdf,
                  SurfaceGeometry geom, float3 wo)
{
  float3 sum = 0;
#if PointLightCount
  for(int i = 0; i < PointLightCount; i++)
  {
    float3 L = evalLight(env.pointLights[i], geom);
    sum += integrate(bxdf, L, wo);
  }
#endif
  // ...
}
```

Listing 2.3: 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. 2.9.

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) [40]. 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 Fig. 2.10 calls out to `integrateQuadLight`, which is expected to be implemented by the `BxDF` 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 2.4.4, 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 help a user identify the changes that must be made to implement the required engine policy.

2.4.6 Efficiently Drawing Many Objects

So far we have covered many concepts related to drawing *one* object. Note that a real shading system must draw tens of thousands of objects for each frame. Therefore, it is important for a shading system to optimize the total CPU and GPU time required to draw not only one, but *all* objects in the scene.

Consider rendering the scene of spheres in Fig. 2.1. The top three spheres use a metal material (same shader code, but with different parameters values), and the bottom three spheres use different materials such as wood, brick and dirt (all using different material shader code and different parameter values). All spheres are rendered with the same camera settings and lighting environment. In this case the inputs to the shading system are shown in Fig. 2.11. They include a list of six objects, each associated with a material and geometry transform feature to use, as a set of parameters to the camera transform, plus directional light features that will be used globally to draw all objects.

Listing 2.4 illustrates a simple implementation that loops through the list of objects and draws each object. In real-time rendering terms, this draw loop is called a *render pass*. For each object, the shading system sets the GPU pipeline state to use a specialized variant of a given shader entry point (`entryPoint`) that implements the combinations of the shading features (material and lighting environment) for drawing the object, and communicates the parameters for all required

Objects:

ID	Shape & Transform	Material
0	Static Shape location [-10 0 -40]	Metal roughness 0.1 reflectance [Texture 0]
1	Static Shape location [-10 0 -20]	Wood patternMask [Texture 3] baseColor [Texture 4]
2	Static Shape location [10 0 -40]	Metal roughness 0.4 reflectance [Texture 2]
3	Static Shape location [0 0 -20]	Brick roughness 0.2 baseColor [Texture 5]
4	Static Shape location [0 0 -40]	Metal roughness 0.2 reflectance [Texture 1]
5	Static Shape location [10 0 -20]	Dirt baseColor [Texture 6] colorVariation 0.34

Global Features

Camera Transform

```
pos        [0    20   40]
viewDir    [0  -0.4 -0.8]
viewAngle  75.0
```

Directional Light

```
dir        [-0.5 -0.7 -0.3]
intensity  [5.0   5.0  5.0]
```

Figure 2.11: The inputs to a shading system to draw the scene of spheres in Fig. 2.1. These inputs include a list of objects to draw, with each object using different material and geometry transform parameters, and a set of shading features to enable globally for all objects.

```
foreach (o in objects)
{
    // Step 1: find the appropriate shader variant to use
    shaderDefines = [];
    collectMaterialDefines(shaderDefines, o.material);
    collectLightingDefines(shaderDefines, directionalLight);
    shaderVariant = findSpecializedVariant(entryPoint, shaderDefines);
    BindShaderProgram(shaderVariant);

    // Step 2: communicate all shader parameters to the GPU
    SetShaderParameter(camera.parameters);
    SetShaderParameter(directionalLight.parameters);
    SetShaderParameter(o.material.parameters);
    SetShaderParameter(o.transform.parameters);

    Draw();
}
```

Listing 2.4: A simple draw loop that draws all objects in the scene. For each object, the code collects the preprocessor `#defines` that are needed to specialize material and lighting shader code from the object's material and the scene's lighting environment. Next, the code finds the shader variant compiled with this set of `#defines` and change the GPU state to use this variant. Finally, all shader parameter values are communicated to the GPU.

shading features to the GPU.

As discussed in Section 2.3.2, in order to achieve high performance, a renderer must minimize changes to the 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.

Unfortunately, the simple implementation in Listing 2.4 is not efficient in terms of GPU state changes in two ways. First, it is repetitively communicating the parameters for camera transform and the directional light once for each object. However, these parameters stay the same for all objects, and the repetitive communication wastes the bandwidth between CPU and GPU as well as CPU cycles needed to gather and marshal those parameter values. Second, when drawing the objects in their original order in the list present in Fig. 2.11, the GPU state must be changed to use a different shader variant before drawing each object. As discussed in Section 2.3.2, this will lead to excessive GPU pipeline stalls and degenerate system performance.

In order to reduce state changes, shading systems often reorder the objects so that objects

```
// Set global shader parameters
SetShaderParameter(camera.parameters);
SetShaderParameter(directionalLight.parameters);

shaderDefines = [];
currentShaderProgram = null;
collectLightingDefines(shaderDefines, directionalLight);

// Draw objects sorted by material
for(m in materials)
{
    SetShaderParameter(m.parameters);
    collectMaterialDefines(shaderDefines, m);
    shaderVariant = findSpecializedVariant(entryPoint, shaderDefines);
    if (shaderVariant != currentShaderProgram)
    {
        BindShaderProgram(shaderVariant);
        currentShaderProgram = shaderVariant;
    }
    for(o in objects using material m)
    {
        SetShaderParameter(o.transform.parameters);
        Draw();
    }
}
```

Listing 2.5: An optimized draw loop that reduces the number of GPU state changes required to draw all objects. Parameter values that are shared among all objects are communicated only once, and objects are grouped by material to minimize switching shader programs between draws.

that require similar GPU states are submitted to the GPU next to each other. The key observation is that some parts of the GPU state (e.g., materials used by objects) change at a higher rate than others (e.g., camera parameters). Listing 2.5 shows an optimized draw loop that significantly reduces GPU state changes. First, parameters that are globally the same for all objects (e.g., directional light and camera transform) are communicated only once. Second, objects are grouped by material before submission to the GPU, eliminating shader program changes between drawings of objects with the same material. For example, the simple implementation will change the shader program six times, once before every draw, while the optimized implementation will draw all metal spheres together, reducing the number of shader program changes by two.

To maximize CPU efficiency when changing GPU states, graphics APIs provide mechanisms to group states into objects that can be updated en masse. For example, Direct3D 10 [18]

organized fixed-function state into coarse-grained objects. Modern APIs such as Vulkan and Direct3D 12 allow shader parameter binding states to be grouped into modules, referred to as descriptor sets [52] and descriptor tables [66] respectively. For clarity, we refer to such API objects as *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.

However, 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) [60, 79] 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.

2.5 Summary of Challenges

In summary, a good shading system design should meet the following requirements:

1. Shading features are implemented in a modular decomposition that has a clean structure aligned with the developers' general understanding of graphics concepts.
2. The modular decomposition of the code base can be easily extended with new shading features, such as a new type of light source or material.
3. The shading system should be able to generate and use specialized shader variants for efficient GPU execution, without incurring CPU overhead.
4. The shading system should minimize GPU state changes when drawing objects, and should communicate shader parameters efficiently using parameter blocks.

While there are simple approaches for meeting each of the individual requirements listed above, the challenge in system design is to find a unified solution that meets all the requirements simultaneously with minimal trade-offs or compromises. As we have shown through our example, shading systems are forced to implement different and dedicated mechanisms for shader specialization and parameter communication, which makes code more complex and conflicts with developers' mental models of graphics concepts. The question is, is it possible to implement a shading system design that aligns the performance optimizations with logical decomposition of shading features? In Chapter 3, we will introduce a design pattern that requires additional shader compiler support, but allows a shading system to meet both performance and extensibility requirements.

Chapter 3

Shader Components

As discussed in Chapter 2, developers wish to author a shading system in a modular fashion that is consistent with common mental organization of rendering concepts such as shapes, geometry transform, materials, lights, etc., while retaining key performance optimizations like shader code specialization and efficient shader parameter communication.

A key observation in this thesis is that a shading system can achieve its performance and extensibility goals simultaneously when the system is designed such that the boundaries of code modules are aligned with the boundaries of performance critical operations such as shader parameter communication and shader kernel specialization.

In this chapter, we present a shading system design pattern that we call *shader components* that would allows a shading system to specialize shader code and communicate parameters in a way that is efficient and aligned with the system's modular decomposition.

The core idea of shader components can be summarized as three design principles:

1. A shading feature corresponds to a component type, which encapsulates both parameter and shader code.
2. Component types are used to specialize shader code.
3. Parameters of a component (i.e., a shading feature) are communicated to the GPU using a unique parameter block.

This chapter provides an overview of the shader components design pattern. To convey the key idea of the design, this chapter presents a sketch of the system design using code snippets written in Slang, a shading language that we will describe in detail in Chapter 4. The rest of the chapter explains the reasoning and impact of shader components' design principles, and

motivates the requirements for the shading language and run-time compilation API. In Chapter 5, we will show a concrete implementation of the shader components design.

3.1 Shader Components Design Principles

The shader components design pattern is interesting in that it is centered around a single abstraction that governs the organization of both CPU and GPU code. On the GPU shader side, it encapsulates the core computation logic and the shader parameters of a shading feature. On the CPU side, it is responsible for holding and communicating the parameter values to the GPU, and ensuring that the correct shader program implementing the required code path is used by the graphics pipeline. We call this abstraction a *shader component*.

Fig. 3.1 illustrates the key concepts of the shader components abstraction, using the same examples we have shown in Chapter 2. The remainder of this chapter describes the concepts in this figure, and has been organized around key principles that underlie the design.

Shader components serve as a bridge between GPU shader and host-side CPU system code, and so we discuss both shader- and host-side 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.

3.1.1 Encapsulating Shader Code and Parameters

To implement shading features in a modular code base, developers need to define the boundaries of a shading feature. Different implementations of the same type of feature (e.g., different materials or lights) 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 and their associated computation logic in a named entity.

In our design, a shader *component type* represents a shading language modular unit that encapsulates both the GPU shader code and parameters of a particular shading feature. A `struct` type in many programming languages such as Swift and C++ serves well for the purpose of representing a component type. For example, `Couch` in Fig. 3.1 is a component type with parameters for a base color texture map, a normal texture map, and a scaling factor to apply to texture coordinates. Another material component type, like `Wood`, will in general have different parameters.

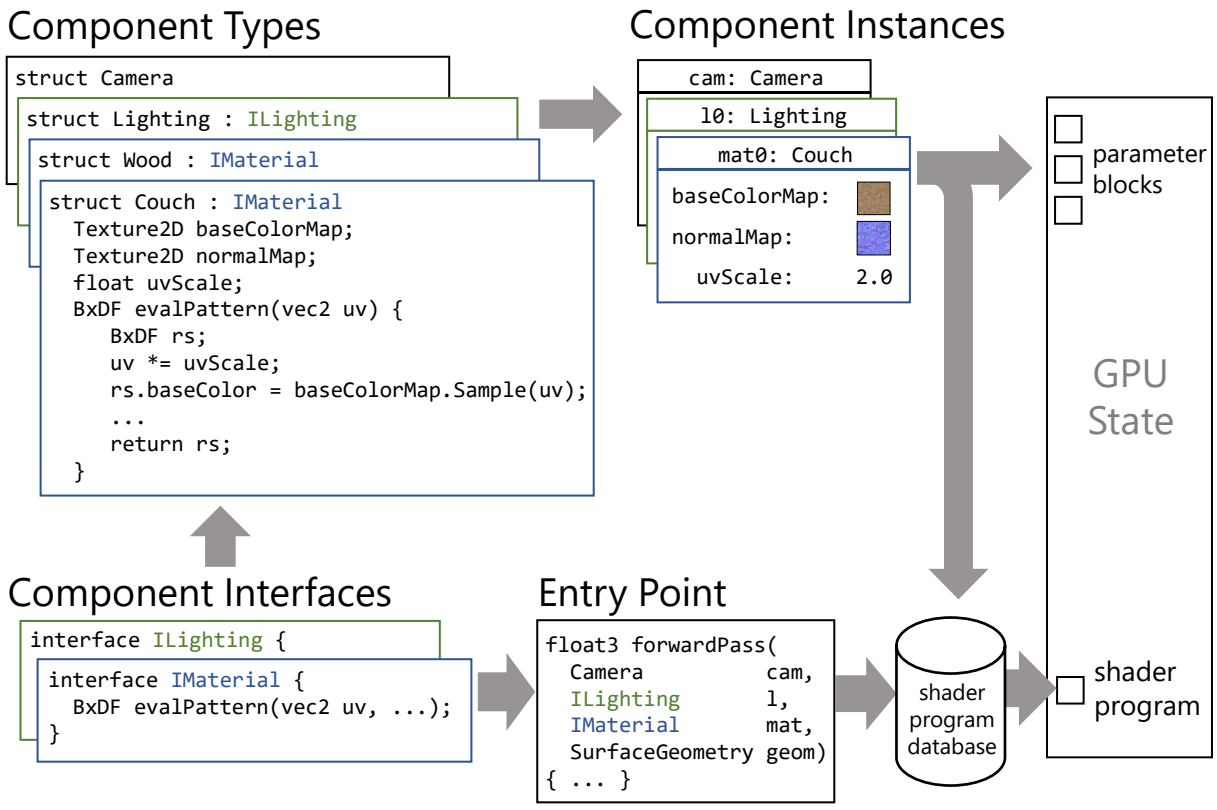


Figure 3.1: Conceptual model for shader components. GPU state is driven from a shader *entry point* function and *component instances* that serve as its arguments. Shading features and their parameters are defined as *component types*. Instances of these types encapsulate parameter values, and map to parameter blocks in modern APIs. An entry point and component arguments together determine a shader variant. Component types and instances have been color-coded to match the interfaces they implement.

3.1.2 Specialize Shader Code Using Component Types

To use shading features implemented as independent component types, a shading system developer must write a shader entry point that coordinates the overall execution and data flow of shader code when rendering an object.

Listing 3.1 shows an evolved version of the `entryPoint` function given in Fig. 2.9. The main shader logic in Listing 3.1 is almost the same, except it wraps all shader parameters in the parameter list of the `entryPoint` function.

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. The shape of a hole can be given by concrete component types (e.g., `Camera` in Listing 3.1), or by *component interfaces* (e.g., `material`

```
float3 entryPoint(
    Camera cam,
    ILighting lightEnv,
    IMaterial material,
    SurfaceGeometry geom)
{
    float3 V = cam.worldPos - geom.P;
    var BxDF = material.evalPattern(geom);
    return lightEnv.illuminate(BxDF, V);
}
```

Listing 3.1: An example shader entry point written in an imaginary shading language that coordinates execution of surface pattern and lighting features.

and `lightEnv`).

To represent a kind of feature (such as material or lighting environment) in a shader library, a system architect defines the component interface, which declares the methods that all implementations of that feature must provide. For example, the `IMaterial` interface in Fig. 3.1 declares a method named `evalPattern` that returns a BxDF closure encapsulating surface properties at a given surface location.

The terminology "interface" is frequently used in object-oriented programming languages, and is typically associated with dynamic dispatch (e.g., virtual function tables). However, the default 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:

```
float3 entryPoint(
    Camera cam,
    ILighting lightEnv,
    IMaterial material,
    SurfaceGeometry geom);
```

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

```
float3 entryPoint<TLighting : ILighting, TMaterial : IMaterial> (
    Camera cam,
    TLighting lightEnv,
    TMaterial material,
    SurfaceGeometry geom);
```

3.1.3 Holding and Communicating Shader Parameters

The modularity benefits of shader components should extend to the CPU (host) side. 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.

In shader components design, the host application code communicates the parameters required by each shader component in that component's own parameter block, by allocating objects called *component instances* from component types.

For example, in Fig. 3.1, `mat0` is an instance of the `Couch` type. It binds `baseColorMap` parameter to a particular texture.

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 a shading system allocates is backed by a parameter block in the target graphics API. In Fig. 3.1, the component instance `mat0` can be used to set a parameter block in the GPU state.

It should be noted that in our design, shader component *types* are implemented in shader code (which requires the shading language to provide necessary mechanisms to encapsulate code and parameters), while shader component *instances* are created by the shading system at runtime, using a runtime API that will be described in the next section. Specifically, we do not argue for a one-size-fits-all runtime library that implements component instances for all shading systems. Instead, the shader compiler provides services that allow shading systems to implement component instances efficiently using parameter blocks; we discuss these services in Section 3.2.

A component instance encapsulates two pieces of information: a parameter block that contains the parameter values of the component, and the component type that this instance is created from. By "plugging in" a component instance to a shader entry point, the shading system gains the knowledge of both the parameters that needs to communicated to the GPU and the component types that should be used to specialize shader code. A component instance is therefore the key to align the boundaries of shader specialization and parameter communication with the boundaries of modular shading feature decomposition.

3.2 Implementing Shader Components

In the previous section, we described how to define component types and entry points in shader code. Given the definitions, a shading system runtime must create component instances and specialize full shaders (top-level entry point code) to the types of these component instances.

A key insight of this thesis is to separate the implementation of shading system design patterns (e.g., shader components) from the shading language and compiler runtime. With this separation, the shader compiler is responsible for generating specialized code and providing reflection information on shader code, and a shading system developer is responsible for using these compilation services to implement any design patterns that suit the system's needs. This separation of responsibilities ensures that the shading system design pattern can be customized and evolved independently of the shading language and the shader compiler, allowing the shading language itself to be generally applicable in many different shading systems.

This section briefly describes the runtime operations of shader components and highlight the key shader compilation services required to facilitate the implementation. Section 4.3 provides detailed documentation of Slang's runtime API for implementing these runtime tasks.

3.2.1 Compiling and Loading Shader Code

The shading system must first load a library of shader code that defines all the shader component types and shader entry points:

```
Module* lib = loadModule("shaderlib.shader");
```

After this call, the shading system can look up shader component types and entry points by name:

```
Type* couchType = findType(lib, "Couch");
Entrypoint* entryPoint = findEntryPoint(lib, "entryPoint");
```

The handles to component types and entry points can be used to query the shader parameters of a component, or to compile a final executable shader program.

3.2.2 Creating Component Instances

At runtime, the shading system needs to create component instances, including the underlying API objects that represent parameter blocks. It is important to note that a component instance

is not a part of shader compilation API and must be defined and implemented by the shading system.

Component instances are used to hold shader parameters of a shading feature, and to specify what GPU code to execute. Shading systems may choose to embody the concept of a component instance in many different ways, but the key is to have a runtime entity that encapsulates both a shader component type (for specializing shader code) and a parameter block created from the type (for communicating the parameters). One possible implementation is the C++ class like the following:

```
class ComponentInstance
{
    Type* componentType;
    ParameterBlock* parameterBlock;
};
```

When creating a component instance, the shading system must determine the required layout and size for its parameter block. To this end, the shader compiler must provide an API for inspecting the parameter layout of a component type. This allows the shading system to allocate a parameter block and fill in concrete parameter values in the format anticipated by the shader.

One benefit of component instances is that they can be created once when the shading system loads and initializes a scene, and then reused each frame when drawing objects that uses the corresponding shading feature. For best performance, a shading system should create a component instance for every material used in the scene during initialization, which effectively creates a parameter block and communicates the parameter values defined by the material to the GPU up-front. When drawing an object that uses a specific material, the component instance created for the material can be used directly without further modification, eliminating the need to repetitively communicate the parameters for the material to the GPU.

For example, the `Material` class previously shown in Fig. 2.9, which is responsible for loading and holding shader parameters for material shading, can be modified to create and maintain a component instance at the material's initialization time. As shown in Listing 3.2, the `Material` class's `loadFromFile` method calls `init` when all data are loaded from disk to create the material's component instance. The `init` method uses the shader compiler's reflection API (`findType` and `getTypeLayout`, as will be discussed in Section 4.3) to query the symbol of the material's component type in the shader library as well as the layout information for the component type. The layout information is then used to fill the parameter values into a parameter block. By con-

```
class Material
{
private:
    // the type name of the correponding shader component for this material
    string typeName;

    // a dictionary holding all the parameter values
    std::vector<std::string, ParamValue> parameters;

    // a component instance for shader specialization and parameter communication
    ComponentInstance instance;
    ...
    void init()
    {
        // We create component instance at intialization time,
        // by finding the type symbol from the shader library and
        // creating a parameter block using the type's layout info.
        instance.type = findType(shaderLib, typeName);
        TypeLayout layout = getTypeLayout(materialType);
        instance.parameterBlock = graphicsAPI::createParameterBlock(layout);
        fillParameterBlock(instance.parameterBlock, layout, parameters);
    }
public:
    void loadFromFile(string fileName)
    {
       typeName = readMaterialTypeName(...);
       parameter = readParameters(...);
       init(); // run initialization after reading all necessary info from disk
    }
    ComponentInstance* getComponentInstance()
    {
        return &instance;
    }
    ...
};

Material couchMaterial;
couchMaterial.loadFromFile("couch.material");
ComponentInstance* materialInstance0 = couchMaterial.getComponentInstance();
```

Listing 3.2: Host-side logic for creating material component instances. This `Material` class implemented in C++ represents material objects in a shading system. Following shader components design, the shading system creates a component instance for the material (and effectively communicates all the parameters to the GPU via creation of a parameter block) once at its initialization time (when parameters are loaded from disk).

structing an instance of the `Material` class from a material file, the developer is also creating a component instance for the material.

A key assumption that allows component instances to be reused in this manner is that a parameter block created from a component type stores the parameter values in the same layout that is expected by all specialized shader variants using the same component type. In most programming languages, this is an obvious guarantee provided by the compiler (e.g., a variable created from a `struct` type can be used to call any function that expects an argument of the same type). However, as we have mentioned in Section 2.4.6, many existing shading languages such as HLSL do not provide this guarantee, making it challenging to use parameter blocks efficiently in a modular code base.

A downside of this component instance implementation is that the host language compiler cannot provide any type checking service for a component instance. For example, the C++ compiler will not have enough knowledge to report an erroneous host code that uses a component instance of a `Camera` type as an argument to `Material` typed shader parameter. As we will discuss in Section 7.3.1, this is a limitation that is difficult to address without modifying the host language or authoring both shader code and host code in the same language.

3.2.3 Drawing Objects

When submitting draw commands to the GPU pipeline, the shading system is responsible for selecting an entry point and a set of component instances to use for its parameters. Listing 3.3 illustrates the sequence of operations when drawing objects. The functions `BindEntryPoint`, `BindComponentInstance` and `Draw` are all shading system defined functions, not actual GPU commands. First, the shading system selects a shader entry-point by calling `BindEntryPoint` in line 1. The `BindEntryPoint` function only changes a shading-system-maintained state tracking the currently selected shader entry point (`entryPoint` in Fig. 3.2).

When the shader entry point is selected, the shading system must fill in its "holes" with component types, and communicate the required shader parameters of the used component type to the GPU. In order to accomplish these tasks, the shading system executes a `BindComponentInstance` operation (line 3-5) to select the component instances to use. This function modifies another shading-system-maintained state tracking the currently selected component instances at various slots (`currentInstances` in Fig. 3.2).

When it is time to issue a draw call, the chosen combination of entry point and components kept by the shading system state as shown in Fig. 3.2 is used to prepare the GPU pipeline for

```
BindEntryPoint(entryPoint);

BindComponentInstance(0, cameraInstance);
BindComponentInstance(1, lightingInstance0);
BindComponentInstance(2, materialInstance0);
Draw(object0);

BindComponentInstance(2, materialInstance1);
Draw(object1);

BindComponentInstance(1, lightingInstance1);
BindComponentInstance(2, materialInstance2);
Draw(object2);
```

Listing 3.3: Sequence of operations to draw objects using shader components.

rendering.

The currently bound shader entry point (`entryPoint`), together with the component types of all currently selected component instances (`currentInstances`) provide complete information to set up the GPU for rendering. The component types of the selected component instances are used to specialize shader entry point, and the handles of the parameter blocks associated with each component instance are sent to the GPU to complete parameter communication.

To avoid repetitively compiling specialized shader variants, the shading system can look up a shader program cache to see if a shader variant resulting from this combination of entry point and component types has already been compiled. If not, the shading system calls the shader compiler to generate the required shader program and put it into the cache. The shading system then sets the GPU pipeline state to use the resulting shader program.

Note that it is a shading system's policy on when and how this program cache is generated. A shading system may choose to generate all potentially used variants in an offline process, or generate it on-demand when rendering a scene. The shader components design pattern does not dictate a specific implementation.

3.3 Benefits of the Shader Components Design

Compared to a shading system implementation using existing shading language, such as the one discussed in Chapter 2, shader components provide the following benefits to a shading system:

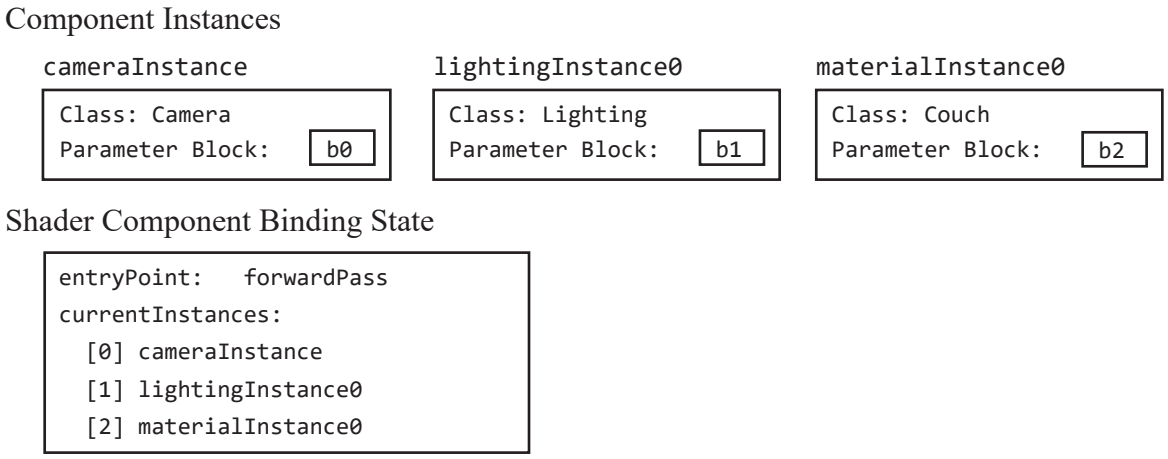


Figure 3.2: Shading-system-maintained states used to configure graphics pipeline for a draw command. `entryPoint` tracks the currently selected shader entry point. `currentInstances` tracks the currently selected component instances.

Modular, extensible shader code base. With shader components, shading features are encapsulated in individual component types, which provides a clear boundary that localizes the parameter definition and computation logic of a shading feature. Component interfaces serve as a mechanism to define the "kind" of a shading feature and enable different implementations of the same kind to be used interchangeably. These mechanisms enable adding a new shading feature to the shading system as a new component type that implements an existing component interface, rather than having to make modifications to various places in the shader library.

Efficient shader parameter communication. Because a shader component instance encapsulates a parameter block, the shading system can pre-allocate parameter blocks and fill them with parameter values during initialization (when the scene is loaded). Parameter blocks reside on the GPU memory, so no additional CPU-GPU communication is required to transfer parameter values to GPU when drawing objects. The shader compiler is required to guarantee that a parameter block created from a component type can be used with any compiled shader programs that use the component type. This allows a parameter block (such as the parameter block for the camera or lighting feature) to be reused in drawing many different objects.

Streamlined shader kernel compilation and selection. Instead of using preprocessor macros to control shader code generation, in the shader components design the same mechanism used to set shader parameters (bind component instances) is also the mechanism to select shader code. This streamlines the host-side logic of a shading system. Furthermore, as we will show in detail

in Section 4.3, it is possible to design very efficient mechanisms for looking-up shader programs from a cache, since a shader program can be uniquely identified by the entry point and the component types used to compile the program. In contrast, a shading system that employs a preprocessor to generate specialized shader programs often uses a concatenated string of preprocessor macros passed to the compiler as the identifier of a shader program. Generating and comparing complex strings incurs large CPU overhead, which is eliminated in the shader components design.

Chapter 4

The Slang Shading Language

In Section 2.4, we demonstrated an example shading system implementation that embodies many aspects of the shader components idea. However, the implementation involved complex code assumptions and compromised extensibility due to a lack of necessary shading language features. This chapter describes how extensions to modern shading languages can help address this challenge. Specifically, we will present an extension of HLSL called Slang.

While it is possible to implement shader components by compiling a fully featured modern language such as C++ to GPU, the design of Slang answers a more fundamental research question: what are the *necessary* and *preferred* language mechanisms for building a modular and efficient shading system?

The goal of Slang is to modernize shading languages for better software modularity, extensibility and runtime performance, not to invent new programming language concepts. In fact, to aid ease of adoption, we wish to avoid making shading system developers to learn and use a fundamentally new shading language (e.g., Spire [38]) or developing their own proprietary shader compilers such as in Bungie's TFX [91]. Instead, we are interested in extending HLSL with standard, general purpose language mechanisms (ones that are already in use in other popular languages today) to facilitate a more streamlined shading system designs.

This chapter first summarizes Slang's design constraints and principles, then describes the set of necessary language mechanisms supported by Slang, and finally documents Slang's runtime compilation API, along with how it can be used to implement shader components.

4.1 Design Constraints and Principles

Slang's design is motivated by the goal of creating a system for use in production real-time rendering environments. This implies that Slang should not only provide software development productivity and performance benefits to graphics developers. It should also be easy to learn and it should improve productivity in most use scenarios. To ensure these properties, we impose the following principles when making design decisions for Slang:

Support implementing shader components. We believe that shader components are a good shading system design. Slang should provide sufficient language mechanisms to make it easy for developers to implement shader components.

Language features should be familiar to graphics developers. We expect Slang to be used by graphics developers who write shaders in HLSL, but are familiar with popular object-oriented languages such as C++, C# [25], and Swift [9]. For this reason, we prefer using well-known language features from these languages, and avoid introducing exotic language constructs that would require significant learning effort. This ensures that any shader code written is Slang can be easily understood by developers even if they have not used Slang before.

Compatible with HLSL. Adopting a new shading language in an existing shading system can be a very challenging task. We would like to make this adoption process easy by allowing developers to adopt the new language features for small parts of the code base as needed and gradually migrate to a new shading system design. This means that the Slang compiler should be able to accept both refactored shader code that uses the new language features, as well as any existing HLSL shader code. For this reason, we made Slang fully compatible with HLSL. New language features are implemented as extensions to HLSL that do not change the semantics of existing language constructs.

No presumed shading system policies. Shading systems often impose policies on organizing shading features, using a certain mechanism to communicate parameter data, and when or whether to use specialized shader code for certain shading features. Different shading systems make different policy decisions. To make Slang a generally applicable shading language in many different shading systems, we must ensure that shader components is not the only design pattern supported by Slang.

```
float3 integrateSingleRay<B:IBxDF>(B bxdf, float3 wi, float3 wo, float3 Li)
{
    return bxdf.eval(wo, wi) * Li;
}
```

Listing 4.1: An example generic function in Slang. The `integrateSingleRay` function has a generic parameter `B` that can be specialized with any type that implements the `IBxDF` interface.

Report errors early and accurately. We would like the Slang compiler to be able to independently check code correctness for each module and report error messages that pin-point the true source of the problem. To improve development productivity, Slang guarantees that any valid composition of individually checked modules will form a valid shader kernel. We seek a design that enables error messages to point directly to the code location that is to be blamed, instead of leaving the programmer to infer the true cause from error messages. This is in contrast to C++ templates or ad-hoc preprocessor based code composition, where the compiler only sees the specialized code and thus cannot provide effective feedback on pre-specialization module code.

4.2 Language Mechanisms

This section covers the key language features we add to Slang. Slang inherits most of the syntax and language features from HLSL, so that most existing HLSL code can be compiled directly by the Slang compiler. We do not discuss the features that are already in HLSL unless they interact with the new features in an interesting way.

4.2.1 Generics

Slang supports parametric polymorphism using the syntax of generics as in Rust [82], Swift [9], C# [25] and Java [34].

For example, Listing 4.1 defines a function that evaluates the rendering equation for any BxDF, given incident illumination along a single ray. In this example the generic type parameter `B` stands in for the unknown BxDF.

One important use of generics is to represent a shader entry point that can be specialized to use different implementations of material, lighting, and other shading features (following the shader components pattern). For example, Listing 4.2 shows a fragment shader entry point simi-

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

Listing 4.2: 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.

lar to `entryPoint` in Fig. 2.9, 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 to type-check `entryPoint` and determine that it is compatible with any material and light types that conforms to the corresponding 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.

4.2.2 Interfaces as Generic Constraints

As in most languages with generics, but unlike C++ templates, a generic like `integrateSingleRay` (Listing 4.1) 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 [93].

The declaration of the `IBxDF` interface used in `integrateSingleRay` is given in Listing 4.3. The `IBxDF` interface defines two *requirements*: `eval` and `getPosition`. Any type that wants to *conform* to this interface will need to provide a concrete method to satisfy this requirement. For

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

Listing 4.3: Definition of the `IBxDF` interface.

clarity, this dissertation 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 is guaranteed to conform to the required interface).

We decide to stick with the same design as Java and C# that requires the user to explicitly specify type conformance when defining a concrete type. This differs from the Go language [33], where interface conformance is derived by the compiler from the use site of the concrete type. While automatic interface conformance deduction saves the programmer's time in writing the declarations, it delays the checking of interface conformance until a use of the concrete type is seen. In fact, the use of a concrete type may not be seen until it is used to specialize a shader variant, which typically happens are run time of the shading system. In contrast, one of our design principles is to allow the compiler to check as early as possible. Explicit interface conformance declarations allow the Slang compiler to check for incomplete concrete type implementations at shading system compile time, when the compiler sees only individual types.

4.2.3 Associated Types as Interface Requirement

As discussed in Section 2.4.3, 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.

Consider a generic shader entry point that must work with any surface material shader of type `M`:

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

As explained in Section 2.4.3, the entry point needs to perform two steps: material shading (evaluating the surface pattern), and compute light integration using the surface pattern. The

```
interface IMaterial
{
    associatedtype BxDF : IBxDF;
    BxDF evalMaterial(SurfaceGeometry geom);
}
```

Listing 4.4: Definition for the `IMaterial` interface. An associated type (`BxDF`) is used to model different return types of `evalMaterial` in different `IMaterial` implementations.

material shading step yields a BxDF closure, where the type of BxDF depends on the type of material. *Associated types* are a language mechanism that allows the `shade` function to name the BxDF type associated with `M`, without knowing it exactly.

Listing 4.4 shows Slang declarations of the `IMaterial` interface that express the concepts of a reflectance function and material surface shader respectively. `IMaterial` makes use of an associated type to capture the dependence of the reflectance function type on the choice of surface shader.

Associated types [22] are a feature we borrowed from Swift [10] and Rust [83]. This feature is also similar to abstract type members in Scala [85]. In Slang, an associated type is an interface requirement that is a type, rather than a method. A concrete type that conforms to the `IMaterial` interface must define a suitable type named `BxDF`, either as a nested `struct` or `typedef`. An associated type may come with bounds, just like a generic parameter; in this case, the concrete `BxDF` type must conform to the `IBxDF` interface.

A concrete implementation of `IMaterial` will define the specific type to use for the associated type `BxDF`. For example, the `SimpleMaterial` type in Listing 4.5 defines `BxDF` to be of `Lambertian` type, which implements a trivial diffuse BxDF. As another example, `CouchMaterial` is another implementation to the `IMaterial` interface, with its `eval` method returning a `DisneyBxDF` closure instead of a `Lambertian` BxDF.

Shader code that takes a material type parameter, such as the parameter `M` of the `shade` function, can refer to the associated reflectance function type as `M.BxDF`:

```
M.BxDF bxdf = material.evalMaterial(...);
```

Because the associated type `BxDF` is bounded using the `IBxDF` interface, only the operations provided by that interface can be used in `entryPoint`. 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.

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

struct SimpleMaterial : IMaterial
{
    typedef Lambertian BxDF;

    Texture2D albedoMap;
    SamplerState sampler;
    Lambertian evalMaterial(SurfaceGeometry geom)
    {
        Lambertian bxdf;
        bxdf.albedo = albedoMap.Sample(sampler, geom.uv);
        return bxdf;
    }
}

struct DisneyBRDF : IBxDF { ... }

struct CouchMaterial : IMaterial
{
    typedef DisneyBRDF BxDF;
    DisneyBRDF evalMaterial(SurfaceGeometry) { ... }
    ...
    Texture2D baseColorTexture;
    float3 tint;
}
```

Listing 4.5: Reflectance and material shading logic defined as types implementing the `IBxDF` and `IMaterial` interfaces (Listing 4.4), respectively. Note that `SimpleMaterial` and `CouchMaterial` are different implementations to the `IMaterial` interface and they return different BxDF types by specifying the `BxDF` associated type to different concrete BxDF types.

4.2.4 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 [11], but equivalents exist in Rust (through use of "traits" [83]) and C# (extension methods [62]). This `extension` declaration makes the type `Lambertian` conform to `IBxDF`; it is the responsibility of the extension author to fulfill the requirements of the interface.

4.2.5 Explicit Parameter Blocks

While HLSL and GLSL do not have a first-class language construct that corresponds to a parameter block, Slang 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 programmer using the Metal API simply declares an entry point parameter with the type `ArgumentBuffer<PerFrameData>`. The `ArgumentBuffer<T>` type [8] in Metal shading language has similar semantics as to a C++ pointer or reference, where the memory layout of the parameter block is given by the `struct`'s definition. However, the concept of an argument buffer is Metal-specific, as no other graphics APIs (e.g., Vulkan and Direct3D 12) provide an equivalent feature in their shading languages.

To generalize the argument buffer concept over a variety of graphics APIs, Slang exposes a `ParameterBlock<T>` type in its standard library. Because different graphics APIs implement the memory layout of a parameter block differently (they are standards created by different groups), Slang leaves the data layout of a parameter block abstract and implements it differently on each target platform.

On Direct3D 12, Slang maps a `ParameterBlock<T>` in two different ways depending on the definition of `T`. If `T` contains only ordinary data fields, a `ParameterBlock<T>` simply maps to a `ConstantBuffer<T>`, where all fields in `T` are communicated via a Direct3D constant buffer. If `T`

```
type_param M : IMaterial;
type_param L : ILightingEnv;

ParameterBlock<M> gMaterial;
ParameterBlock<L> gLights;
ParameterBlock<Camera> gCamera;

float4 main(SurfaceGeometry geom)
{
  float3 viewDir = normalize(gCamera.P - geometry.P);
  M.BxDF bxdf = gMaterial.evalPattern(geometry);
  return gLights.illuminate(surface, viewDir);
}
```

Listing 4.6: The entry point from listing 4.2, changed to declare its parameters (both type parameters and shader parameter blocks) at global scope, as is conventional in current HLSL/GLSL code bases.

contains both ordinary data fields and resource-typed fields (such as `Texture2D`), only the ordinary data fields can be communicated via a constant buffer. In this case, Slang will generate HLSL source code that takes as input all the ordinary data fields via a constant buffer, with all resource-typed fields separated out as global parameters. Slang will also generate layout annotations to these global parameters (as discussed in Section 2.4.6) to enable communicating the handles to both the constant buffer and global resource-typed parameters en mass via descriptor tables.

On all platforms, Slang will not attempt to eliminate unused parameters before generating layout annotations (assigning binding indices) for resource-typed parameters, or removing them from reflection information. For efficient shader parameter communication, the Slang language definition requires all compiler implementations to guarantee that the layout of a parameter block is the same across all shader variants that use the parameter block. This guarantee enables a shading system to safely reuse a parameter block created after a given type for all shader variants that requires a parameter block of the same type. The shader components design pattern relies on this guarantee, since a component instance created from a component type must be compatible with any shader variants specialized with the same component type.

4.2.6 Global Generic Type Parameters

It is common practice in current HLSL and GLSL codebases to declare some of the parameters of a shader entry point at global scope, rather than as explicit parameters of an entry-point function.

While the use of explicit generics as shown in Listing 4.2 has benefits, the need to refactor a global shader parameter into an explicit function parameter on every entry point that uses it presents a barrier to incremental adoption of generics. In our experience this barrier is significant for users with large HLSL shader libraries, and therefore Slang includes an alternative syntax for generics that simplifies adoption.

Listing 4.6 shows an entry point equivalent to Listing 4.2, with the key change that the material, lighting, and camera parameter blocks are now declared at global scope. (In practice they might even be in a different file than the `main` function.)

In order to make use of generics for specialization, `M` and `L` are declared as *global scope type parameters* in Listing 4.6. Global-scope type parameters are semantically equivalent to parameters of an explicit generic type. Conceptually they behave as if the whole shader library is nested in a generic declaration, with the given parameters.

We do not claim that global-scope type parameters are an elegant feature, nor one that other languages should emulate. However, we strongly believe that lowering barriers to adoption is critical when bringing new ideas to established languages. Allowing users to write the code in Listing 4.6 has proved valuable, even if it is ideally only a stepping stone to code like Listing 4.2. As we will discuss in Section 7.2.3, global generic type parameters introduce practical issues, and should not be used when compatibility with existing HLSL code is not required.

4.3 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 shading system host code in composing and executing the shader effects in the library and preparing parameter blocks.

In this section, we document the detailed services provided by Slang's runtime API. Previously in Section 3.2, we laid out the overall flow of a shading system runtime implementing the Shader Components design pattern. This section is organized in the same logical order and provides more details on how Slang's runtime API can be used to implement key tasks in the Shader Components design pattern.

Listing 4.7 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. 4.1, and are motivated by the requirements of shader components design pattern, as discussed in

```
/*** Front End ***/
// load shader code from file and compile it into IR:
Module* loadModule(const char* pathToSource, char** errorMessage);

/*** Reflection ***/

// lookup type and entry point symbols in a module
EntryPoint* findEntryPoint(Module* module, const char* name);
Type* findType(Module* module, const char* name);
// query parameter layout from a type symbol
TypeLayout* getTypeLayout(Type* type);

// access detailed layout information from a TypeLayout object:
struct ResourceUsage
{
    int texture, sampler, buffer, ...;
};
ResourceUsage getResourceUsage(TypeLayout* layout);
int getConstantBufferSize(TypeLayout* layout);

// alternate reflection APIs: query layout directly from field names
Type* getFieldType(TypeLayout* layout, const char* fieldName);
ResourceUsage getFieldResourceOffset(TypeLayout* layout, const char* fieldName);
int getFieldConstantBufferOffset(TypeLayout* layout, const char* fieldName);

// specify default layout rule for parameter blocks
enum LayoutRule { Native, Standard }
void setDefaultLayoutRule(LayoutRule rule);
Type* specializeType(Type* type, Type* types[]);

// query dynamic unique ID for each symbol
int getTypeUID(Type* type);
int getEntryPointUID(EntryPoint* entryPoint);

/*** Back End ***/
enum Platform { HLSL, SPIRV }
// generate specialized kernel code for a target platform
Kernel* specializeEntryPoint(EntryPoint* entryPoint, Type* types[], Platform pfm);
```

Listing 4.7: An overview of API services provided by the Slang implementation, grouped according to the system diagram in Fig. 4.1. These API services are used by the shading system to perform key tasks needed to implement shader components, such as populate the contents of parameter blocks to create component instances, and compose components instances into top-level shaders.

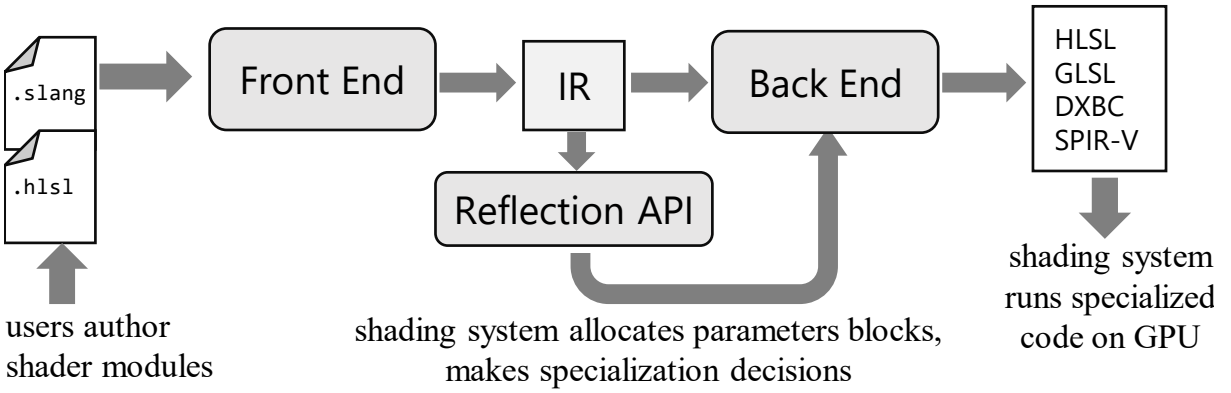


Figure 4.1: The Slang system implementation comprises a compiler front-end (for checking and determining parameter layouts for unspecialized modules) and back-end (for generating specialized GPU kernels), as well as supporting APIs. Applications use the Slang reflection API to obtain shader parameter layout information used to allocate GPU parameter blocks and to guide the specialization choices needed for low-level code generation.

Section 3.2. A shading system developer defines component types and entry points as separate modules. The modules are processed by the compiler front end, which performs type-checking and emits intermediate representations (IR) of the checked modules. To support the runtime tasks of shading systems (such as creating parameter blocks and specializing shader variants), the shader compiler runtime provides a reflection API for querying the layouts of component types, and a compilation API for generating specialized shader kernel code that can be executed on the GPU.

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 shading system 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).

4.3.1 Loading Shader Modules

To get started, a shading system loads a shader library comprising one or more modules of Slang (or HLSL) code into the compiler's front end by calling the `loadModule` function:

```
char *errMsg;
Module* shaderModule = loadModule("shader_library.slang", &errMsg);
```

This triggers type-checking and intermediate code generation steps of the compiler. If the compiler finds any error in the source file, it will report them via `errMsg`, otherwise, the function returns a compiled module containing the intermediate representation of types and functions. The type and function definitions stored in a loaded module are used for further compilation tasks, such as querying parameter layouts and generating specialized shader kernels, which we discuss in the following subsections.

4.3.2 Reflection API for Parameter Communication

A shading system following shader components design needs 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.

The following C++ class illustrates a possible implementation of component instances:

```
class ComponentInstance {
    Type* componentType;
    API::ParameterBlock parameterBlock;
    // ...
};
```

In this definition, `componentType` is a Slang type symbol representing the shader component type, and `parameterBlock` is a platform-specific handle to a parameter block. The type symbol can be acquired with the `findType` function. For example, the following code obtains the `CouchMaterial` type symbol from loaded module:

```
Type* couchMatType = findType(shaderModule, "CouchMaterial");
```

To ensure the parameter values are written into a parameter block in the layout expected by shader code, the shading system uses Slang's `getTypeLayout` function for inspecting the layout of a type:

```
TypeLayout* layout = getTypeLayout(couchMatType);
```

The `TypeLayout` object provides the kinds of resource (buffer, texture, etc.) and the number of "slots" of that kind that are used in a type (queried by `getResourceUsage` function). Ordinary data fields (`int` and `float` fields) are communicated via a separate constant buffer, and the required constant buffer size can be queried with the `getConstantBufferSize` function. Additionally, Slang's runtime API provides a traditional refection interface for looking up the index/offset of an individual field by name.

```
ComponentInstance createCouchMaterialInstance(Texture2D baseColor, float3 tint)
{
    ComponentInstance inst;

    // Step 1: find type symbol
    inst.componentType = findType(shaderModule, "CouchMaterial");

    // Step 2: create parameter block andd fill in parameter values
    // Query layout information
    TypeLayout* layout = getTypeLayout(inst.componentType);
    // Query parameter block size
    ResourceUsage usage = getResourceUsage(layout);
    int cbufferSize = getConstantBufferSize(layout);
    // Allocate parameter block
    // In Direct3D, this is a descriptor table and an associated constant buffer
    inst.parameterBlock.descTable = api::AllocateD3D12DescriptorTable(usage);
    inst.parameterBlock.cbuffer = api::AllocateConstantBuffer(cbufferSize);
    // Fill in texture handles into descriptor table
    int offset = getFieldResourceOffset(layout, "baseColorTexture");
    api::WriteDescriptor(inst.parameterBlock.descTable, offset, baseColor);
    // Write the value of tint into the associated constant buffer
    offset = getFieldConstantBufferOffset(layout, "tint");
    api::WriteConstantBuffer(inst.parameterBlock.cbuffer, offset, tint);

    return inst;
}
```

Listing 4.8: C++ pseudocode for creating component instances of `CouchMaterial` (defined in Listing 4.5). The function takes the shader parameters for `CouchMaterial` component type as input. It finds the type symbol for `CouchMaterial` type using the `findType` API, and allocate a parameter block (represented by a Direct3D descriptor table for resource typed parameters and a constant buffer for ordinary parameters) using layout information provided by the reflection API.

As an example, Listing 4.8 shows a C++ pseudo code for creating component instances of `CouchMaterial` (defined in Listing 4.5) using Slang's reflection API. The function takes as input the shader parameters of `CouchMaterial`.

The first step is to obtain the symbol for `CouchMaterial` component type using `findType`.

The next step is to allocate a parameter block for the component instance. In this example, we are targeting Direct3D 12, so a parameter block includes a descriptor table (`descTable`) to store resource-typed parameters and a constant buffer (`cbuffer`) to store ordinary parameters. In order to allocate the parameter block, we first call `getTypeLayout` to obtain an object encapsulating

all the layout information, and use the `getResourceUsage` API function to find out the number of resources used by the type, and `getConstantBufferSize` to query the required constant buffer size to store all ordinary parameters of this type.

The final step is to fill in the parameter values into the parameter block. Since `baseColorTexture` is a resource-typed parameter, we call `getFieldResourceOffset` to find the location of the parameter in the descriptor table, and write the `baseColor` texture handle to the queried location. In contrast, `tint` is an ordinary parameter, so we call `getFieldConstantBufferOffset` to find out its location, and write the parameter value to that location in the constant buffer.

An alternative method to ensure parameter layout consistency between the CPU and GPU code is to enforce a global standard on how parameter values should be laid out in parameter blocks in both the shader compiler and the shading system host code. This will eliminate the need for using the reflection information to fill out parameter blocks. A shading system can opt in to this method by calling `setDefaultLayoutRule` and requiring the shader compiler to follow a platform-independent rule to lay out the parameters. However, doing so will require the shading system to follow exactly the same rules when filling parameter values, which is bug-prone and could lead to undefined behaviors when there is a mismatch between the implementation of the shader compiler and the shading system. We believe that using the reflection data provided by the compiler is a preferred option in many cases. First, this results in a more robust system, eliminating the need for the shading system to implement the layout rule along with the possibility of a mismatch in the implementations that can be disastrous. Second, in an extensible shading system, developers are constantly adding new features. A general reflection-based implementation eliminates the developer attention required to maintain layout consistency for new features and improves development productivity. A dynamic reflection API is also essential for supporting data-driven shading systems and tools. Finally, in some case there are performance benefits to storing the parameters in the optimal layout specific to different target hardware platforms.

The added support for querying parameter layout directly from individual types is the most significant deviation from HLSL's reflection API. Recall that the shader components design pattern relies on the ability to reuse a parameter block in different specialized shader variants to achieve efficient parameter communication. This means that the parameter block layout information must be made available to the shading system before any specialized shader variants are generated. In contrast, HLSL's runtime only allows querying parameter layout on a complete shader variant and does not guarantee that the parameter block layout queried from one shader variant is consistent with the layout of the same type queried from another shader variant. This limitation prohibits efficient reuse of parameter blocks across different shader variants.

4.3.3 Specializing Shader Code

A shading system must also specialize shader code for the set of shading features used by each object. In the example shading system in Chapter 2, this was performed by pasting HLSL strings of `#defines` and `typedefs` to create Listing 2.1. The Slang compiler back-end provides the API to specialize an entry point for a particular set of type arguments, yielding platform-specific GPU kernels.

To generate a specialized shader variant the shading system first calls `findEntryPoint` to locate the symbol for the entry point function that needs to be specialized:

```
EntryPoint* entryPoint = findEntryPoint(module, "entryPoint");
```

Next, the shading system calls `specializeEntryPoint` to specialize the entry point with a set of type symbols and produce target code:

```
Type* types[] = {couchMatType, ...};
Kernel* kernel = specializeEntryPoint(entryPoint, types, HLSL);
```

Slang currently supports generating HLSL output for Direct3D and SPIR-V [47] output for Vulkan.

In a shading system that follows the shader components pattern, the type arguments can be obtained from the list of component instances that are used for each object. Note that the shader components design does not dictate when the shading system generates specialized shader code. A shading system may choose to compile specialized shader code on the fly immediately before drawing each object, or generate all possible specialized shader variants in an off-line process and store the generated kernels in a cache. In both implementations, the shading system uses the same Slang API for the compilation task.

In addition, Slang's runtime API allows creating specializations of generic types (not just entry points) via the `specializeType` function, which enables shading systems to perform both fine- and coarse-grained composition (i.e., at both shader entry point level and sub-component level). The shading system may also query layout information for specialized types, and use this information when allocating and populating parameter blocks.

Note that when the compiler loads a shader module, it is already performing type-checking and generating intermediate code from the source file. At the time of generating specialized shader kernels, the only work that is left is to link together all the involved intermediate code and emit final output code for the target platform. The front-end work, including parsing and type-

checking, is done only once on the unspecialized code, and reused when generating different specialized shader variants.

4.3.4 Looking Up Specialized Shader Variants

To avoid compiling shader code on the fly, most shading systems will generate all shader variants in an off-line process and store the shader variant kernel in a cache. When drawing an object, the shading system forms a lookup key into the cache to locate the shader variant specialized with the required component types.

A naive implementation might construct a unique ID for each shader variant by concatenating the entry point and component type names used to generate the variant. In our experience, looking up shader variants is critical to performance, so a shading system should avoid using inefficient key representations such as variable-length strings. A strategy to optimize variant lookup is to assign sequential small integers, called *dynamic IDs* to all entry points and component types at load time of a shader library. For most code bases, these dynamic IDs are within the range of a 16-bit integer. Therefore, a 128-bit key is sufficient for entry points using up to 7 generic type parameters. This key is then used to index a hash table of shader variants. Hashing and comparison of this 128-bit key representation is more efficient than using variable-length strings.

To support shading system implementations using dynamic IDs, Slang's compilation API provides the utility functions (`getTypeUID` and `getEntryPointUID`) to query the IDs for entry points and types.

4.3.5 Summary

Slang is a shader compilation system that includes both a language and a corresponding runtime API to control compilation and inspect shader code. With the runtime API, the shader components design pattern can be implemented efficiently without relying on ad-hoc shader code generation tools or pre-processor hacks. Unlike some prior work such as Bungie's TFX system [91], or the classes and interfaces constructs in Cg [75] and HLSL [61], neither the Slang language nor the runtime API subsumes any specific shading system policies, such as when to generate specialized shader code, or how to communicate shader parameters. This differentiation makes Slang a general shader compilation service that can be used by many different shading systems that make different runtime policy decisions.

Chapter 5

A Case Study of Adopting 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 [14]. 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. This version of Falcor features 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 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 incrementally refactor the Falcor shading system to adopt Slang's language features. In this chapter, we provide an overview of the changes we made to the Falcor code base in the process of adopt-

ing the shader components design, and discuss our experiences working with the refactored code base in terms of system extensibility and performance. Although the discussion in this chapter will make use of many example code snippets, the full source of the refactored shader library is available at `https://github.com/shader-slang/Falcor/tree/slang-refactor`. The repository is at tag `thesis_ref` at the time of this writing, so code mentions correspond to the state of the code in this version of the branch.

5.1 Refactoring the Falcor Code Base

In this section, we summarize the major changes made to the existing Falcor code base in order to adopt the shader components design pattern to improve both performance and extensibility. These changes include host-side infrastructure changes to communicate parameters and specialize shaders following the shader components pattern, and shader library changes to encapsulate material shading and lighting logic into shader components. The improved infrastructure enables the refactored system to achieve things that were difficult to implement in the original branch without any additional effort, such as supporting two different implementations of material shading logic simultaneously, and specializing shader code to different lighting environments.

5.1.1 Using Parameter Blocks to Communicate Parameters

Similar to our example shading system in Section 2.4, the existing Falcor shader library uses `struct` types to encapsulate related shader parameters. For example, Falcor defines material parameters (colors, textures, etc.) using a single global shader parameter (placed in an HLSL constant buffer, denoted by `cbuffer` in the HLSL code below):

```
cbuffer PerFrameData { MaterialData gMaterial; }
```

The `MaterialData` type is defined in Listing 5.1. For the reasons described in Section 2.4.6, 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. This approach results in extra CPU cost (to fill in the parameter block) and additional GPU state change overhead (for updating all the parameter values between draws). Using Slang's explicit parameter block construct (Section 4.2.5) we easily were able to 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, similar to the modification to the `Material` class discussed in Section 3.2.2, using Slang's runtime API (discussed in Section 4.3). By re-using 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.

5.1.2 Use Generics for Shader Specialization

We followed the examples in Section 4.2.6 to use global generic type parameters to specialize a shader entry point with different material pattern generation and lighting implementations. As will be discussed in the following sections, we defined the `IMaterial` and `ILightingEnv` interfaces to bound the generic types, so that the Slang compiler can statically type-check the entry point code.

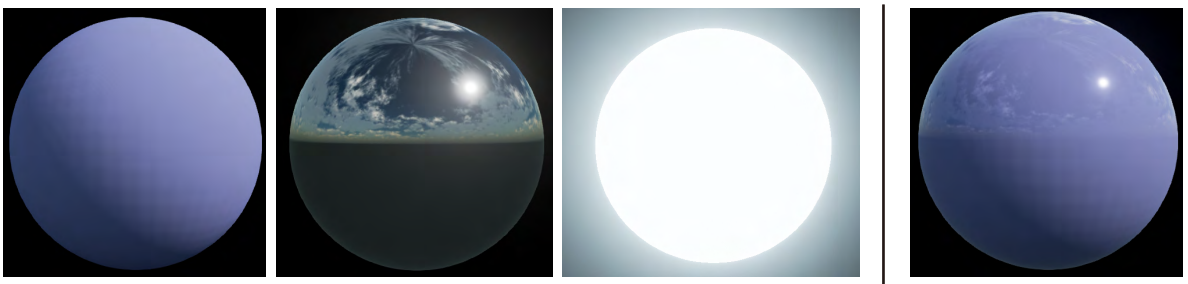
5.1.3 Refactoring Material Computation

At the same time frame that we were refactoring Falcor's material shading code, the Falcor's development team was replacing its legacy material model that uses a layered BxDF with a simpler material model with a consolidated BxDF. The new material system is more constrained than the original system, but its implementation is simpler and runs faster on the GPU. In this section, we will introduce both the legacy and the new material model, and show that the refactored architecture allows the implementations to cleanly co-exist.

Refactoring the Legacy Material Model

In the legacy model, a material consists of a combination of *layers*. Each layer models a specific component of the surface's light reflectance, and can be represented by a BxDF. Fig. 5.1(a) illustrates three different types of material layers that models diffuse, specular and emissive (light emitted from the object itself) light reflectance. By combing different layers, an artist is effectively composing a complex BxDF from a series of simpler BxDFs. For example, Fig. 5.1(b) shows an object that has a combination of diffuse and specular material layers.

Listing 5.1 shows the definitions for the legacy material model. A material is defined by the `MaterialData` type, which contains an array of layers, as defined in MaterialLayerData. The `brdfType` field of `MaterialLayerData` defines which BRDF to use for the layer, and the `blendFunc` field defines how the result of this layer is blended with the next layer.



Diffuse Layer | Specular Layer | Emissive Layer | Diffuse + Specular Layers

(a) | (b)

Figure 5.1: Examples of different types of material layers. (a) Three common types of material layers modeling the diffuse, specular and emissive light components. (b) An object rendered with a combination of diffuse and specular material layers.

```
struct MaterialLayerData
{
    // the bxdf represented by this layer
    int bxdfType;
    // how the lighting result of this layer is blended with the next layer
    int blendFunc;
    // the texture containing parameter values to the BxDF
    // (e.g., base color for diffuse layer, or roughness for specular layer)
    Texture2D colorTex;
};
struct MaterialData
{
    int numLayers; // the actual number of layers in this material
    MaterialLayerData layers[MAX_LAYERS];
};
```

Listing 5.1: Definitions for the legacy material model. A material is defined by `MaterialData`, which contains an array of layers defined by MaterialLayerData. Each layer is defined by a `bxdfType` flag, which states the type of the BxDF represented by this layer (such as a diffuse or a specular layer); a `blendFunc` flag, which controls how the lighting result of this material layer is blended with the next layer; and a `colorTex` field, which is a handle to a texture containing a parameter value to the BxDF. This parameter value is interpreted differently depending on the concrete type of the BxDF in use. For example, a diffuse layer will treat the parameter value as base color, and a specular layer will treat the parameter value as a roughness value.

By default, Falcor evaluates a material's overall reflectance by dynamically looping over an array of layers and conditionally executing code based on each layer's BxDF type (`bxdfType`) and blending function (`blendFunc`). 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`.

In our refactored shader library, we introduced an `IMaterial` interface, as shown in Listing 4.4. Note that we have modeled the return type of material shading phase (`evalMaterial` method) as an associated type, so that different material implementations can use different BRDFs for lighting. The shader entry points use a generic type parameter `TMaterial : IMaterial` to specialize to a selected material type (as shown in Listing 4.2). The definition of the material parameter block changed again, to:

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

We then created a `LegacyMaterial` type from the existing Falcor `MaterialData` type, 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 only a standard set of layers:

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

The `StandardLegacyMaterial` 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 `StandardLegacyMaterial` type (when the material features only the standard layers). Following the shader components design pattern discussed in Chapter 3, Falcor was modified to look up a specialized variant based on the types of component instances bound to the pipeline. By giving each specialized StandardLegacyMaterial type a small integer ID, this lookup step can be implemented more efficiently than Falcor's previous string-based lookup using `#defines`.

```
struct MaterialData
{
    // flags controls whether a texture or a constant value
    // should be used for each channel
    uint32_t flags;

    // base color channel
    float4 baseColorValue;
    Texture2D baseColorTex;

    // specular channel
    float4 specularValue;
    Texture2D specularTex;

    // emissive channel
    float3 emissiveValue;
    Texture2D emissiveTex;
};
```

Listing 5.2: The new `MaterialData` type in the orignal Falcor shader library. The type encapsulates all parameters for a single built-in BxDF.

Refactoring the New Material Model

In Falcor's new material implementation, a single BxDF is used to evaluate lighting for all types of materials. A material only defines the surface properties that serve as parameters to the BxDF. The value of a BxDF parameter is either provided directly as a constant value or via a texture image that must be sampled during material shading stage.

As shown in Listing 5.2, the new `MaterialData` type encapsulates all surface properties that will be used as parameters to a built-in BxDF. A material has three properties: a base color, a specular term and an emissive term. Depending on the `flags` field, the base color property may come from `baseColorValue` or from a sampled result of `baseColorTex` texture; the same is true for the specular and emissive term.

The material shading logic for the new material model is simple: it either copies the BxDF parameter value directly from the input `MaterialData` to the resulting BxDF closure (when the parameter value is provided as a constant), or samples the texture at the shading point's parametric coordinate and writes the resulting value to the returning BxDF closure.

Listing 5.3 shows Falcor's implementation of material shading. Falcor use dynamic branching on material `flags` field to determine whether the associated texture should be sampled.

```
cbuffer PerFrameData
{
    MaterialData gMaterial;
}

struct BxDF
{
    float3 baseColor;
    float3 specular;
    float3 emissive;
};

BxDF evalMaterialPattern(SurfaceGeometry geom)
{
    BxDF rs;

#ifdef MATERIAL_FLAGS
    int flags = MATERIAL_FLAGS;
#else
    int flags = gMaterial.flags;
#endif

    if (flags & BASE_COLOR_TEXTURE_BIT)
        rs.baseColor = gMaterial.baseColorTex.Sample(sampler, geom.uv);
    else
        rs.baseColor = gMaterial.baseColorValue;

    if (flags & SPECULAR_TEXTURE_BIT)
        rs.specular = gMaterial.specularTex.Sample(sampler, geom.uv);
    else
        rs.specular = gMaterial.specularValue;

    if (flags & EMISSIVE_TEXTURE_BIT)
        rs.emissive = gMaterial.emissiveTex.Sample(sampler, geom.uv);
    else
        rs.emissive = gMaterial.emissiveValue;

    return rs;
}
```

Listing 5.3: Material shading logic of the new material model in the orignal Falcor shader library. The code implements a single BxDF and uses a flag to control whether a BxDF parameter (e.g., base color, spcular or emissive) comes from a texture or a constant value. To support static specialization, the shading system provides a `MATERIAL_FLAGS` macro that defines the actual flag value when compiling this code.

Similar to the legacy material model, the existing Falcor code includes support for specializing material code by passing the material's `flags` field as a preprocessor `#define`. As shown in Listing 5.3, the code will use material flags defined by the `MATERIAL_FLAGS` proprocessor macro if it exists, and relies on the compiler's constant propagation optimization to eliminate the dynamic branching.

```
interface IChannel
{
    float3 getValue(SurfaceGeometry geom);
}

struct ConstantValueChannel
{
    float3 value;
    float3 getValue(SurfaceGeometry geom)
    {
        return value;
    }
}

struct TextureChannel
{
    Texture2D texture;
    float3 getValue(SurfaceGeometry geom)
    {
        return texture.Sample(sampler, geom.uv);
    }
}
```

Listing 5.4: The `IChannel` interface, which models the data source of a BxDF parameter, and two of its implementations in refactored Falcor shader library.

We created a `NewMaterial` type in the refactored branch, which also implements the same `IMaterial` interface. As shown in Listing 5.4, we introduced the concept of a *channel* to model the source of a surface property, which can come from two ways — `ConstantValueChannel` when its value comes from a constant, and `TextureChannel` when its value comes from sampling a texture. Listing 5.5 shows the `NewMaterial` type, which is now a generic type parameterized on the types of its channels. This design allows materials to be specialized on channel types to eliminate dynamic branching during material shading.

After these changes, both `NewMaterial` and `LegacyMaterial` exist in the refactored branch at the same time, allowing Falcor's users to pick the material model that best suits their needs.

```
struct NewMaterial
    <TBaseColor : IChannel,
     TSpecular : IChannel,
     TEmissive : IChannel> : IMaterial
{
    TBaseColor baseColor;
    TSpecular specular;
    TEmissive emissive;

    struct BxDF
    {
        float3 baseColor;
        float3 specular;
        float3 emissive;
    };
    BxDF evalMaterial(SurfaceGeometry geom)
    {
        BxDF rs;
        rs.baseColor = baseColor.getValue(geom);
        rs.specular = specular.getValue(geom);
        rs.emissive = emissive.getValue(geom);
        return rs;
    }
}
```

Listing 5.5: The refactored material type. `NewMaterial` is a generic type parameterized on channel types for specialization of material shading logic.

5.1.4 Refactoring Lighting Computation

In our refactored branch, we wanted also to make it easy to extend lighting logic with new types of lights and to improve the efficiency of lighting computation by specializing shader code to the actual lighting environment in-use.

To avoid the complexity of preprocessor solutions, Falcor's developers chose not to employ the static specialization approaches for lighting computation as discussed in Section 2.4.4. Instead, the original Falcor implementation used dynamic branching to deal with different types of lights in a scene.

As shown in Listing 5.6, the original shader library uses a single HLSL type, `LightData`, to represent all supported light types. Similar to `MaterialData` type, each light has a `lightType` field that indicates its specific type (point, spot, directional), followed by a union of the fields required

```
/*** lighting computation ***/
struct LightData
{
    int lightType; // either POINT_LIGHT or DIRECTIONAL_LIGHT
    float3 position; // used when the light is a POINT_LIGHT
    float3 intensity; // used for both POINT_LIGHT and DIRECTIONAL_LIGHT
    float3 direction; // used for DIRECTIONAL_LIGHT only
    ...
};
float3 evalLight(BxDF bxdf, LightData l)
{
    if (l.lightType == POINT_LIGHT)
       return evalPointLight(bxdf, l.position, l.intensity);
    else if (l.lightType == DIRECTIONAL_LIGHT)
       return evalDirLight(bxdf, l.direction, l.intensity);
    else if (l.lightType == ...)
       ...
}

/*** entry point ***/
cbuffer PerFrameData
{
    MaterialData  gMaterial;
    int           gLightCount;
    LightData     gLights[MAX_LIGHTS];
    ...
};

float3 entryPoint(SurfaceGeometry geom)
{
    BxDF bxdf = evalMaterialPattern(geom);
    float3 result = 0;
    for (int i = 0; i < gLightCount; i++)
        result += evalLight(bxdf, gLights[i]);
    return result;
}
```

Listing 5.6: Lighting logic in the original Falcor shader library. A single `LightData` type is used to encapsulate the parameters of all different types of light sources. The `evalLight` function uses dynamic branching to dispatch to different lighting computation logic based on the type of light source. This entry point coordinates execution of material shading and lighting computation by calling `evalMaterialPattern` to obtain a BxDF closure and using it to integrate lighting from all lights defined in the `gLights` array.

by the different cases.

The `evalLight` function implements the logic for computing lighting from one light source. It branches on the `lightType` field of the input light source and dispatches to different lighting code at runtime.

The light environment is communicated to the shader via an array (`gLights`). The shader entry point performs light computation by looping over the array of lights and repetitively calling `evalLight` for each light source.

In contrast to the monolithic light type in shader code, Falcor's C++ code has a `Light` class with distinct subclasses for `PointLight`, etc. This is because C++'s object hierarchy allows the host code to be easily extensible, and unlike GPU code, the dynamic dispatch introduced in this representation does not lead to critical performance issues on the CPU.

Using Slang's generics feature, our refactored shader code can be specialized to different lighting environments. The key idea to implement lighting specialization is to specialize the shader entry point in Listing 4.6 with a specialized light collection type as type argument to the `L` parameter. The rest of this section presents the design of this specializable lighting computation architecture. Section 5.2 discusses the benefits of this design. Section 5.4 presents an extension to this architecture to support ray tracing and deferred rendering techniques.

First, we defined an `ILight` interface to abstract different types of lights. Listing 5.7 shows the definition for `ILight` and two example implementations - `PointLight` and `DirectionalLight`.

Since a lighting environment typically consists of more than one light, it is necessary to develop a mechanism to communicate a collection of lights (each may have a different type) to the shader entry point, and allow the shader entry point to iterate through the content of the heterogeneous collection to compute the sum of lighting contributions from each light. Recall that the original branch's implementation (as shown in Listing 5.6) chose to encapsulate all lights in a single type to avoid heterogeneity and use dynamic branching to select lighting code. This decision sacrifices extensibility and performance.

The high level idea of our refactored branch is to abstract light collection as an interface and specialize shader entry point with the different light collection types that represent different compositions of light sources. Based on the types of light sources used in a scene, the system constructs a concrete light collection type using composite operators.

Listing 5.8 defines an `ILightEnv` interface to represent a light collection. This interface declares that every lighting environment must provide an operation to compute an aggregated value

```
interface ILight
{
    float3 illuminate<B:IBxDF>(B bxdf, float3 wo);
}

struct DirectionalLight : ILight
{
    float3 direction;
    float3 intensity;
    float3 illuminate<B:IBxDF>(B bxdf, float3 wo)
    {
        return bxdf.eval(direction, wo) * intensity;
    }
}

struct PointLight : ILight
{
    float3 position;
    float3 intensity;
    float3 illuminate<B:IBxDF>(B bxdf, float3 wo)
    {
        float3 direction = normalize(position - bxdf.getPosition());
        return bxdf.eval(direction, wo) * intensity;
    }
}
```

Listing 5.7: Definition of the `ILight` interface and two example implementations of `ILight` implementing a directional light and a point light.

using the lights contained by the collection. The `aggregate` operation can be used as a way to iterate through the collection: its `integrator` parameter represents a callback function that will be invoked on each light, which will compute a value from each light and return an combined value aggregated with the values computed from previous lights.

Listing 5.9 shows an example function that sums up lighting contributions using lights from a `ILightEnv`. It defines a `StandardIntegrator` type that computes the lighting contribution from a single light and returns an accumulated light contribution. The `evalLighting` function creates an instance of `StandardIntegrator` and uses it to call `gLights.aggregate`, which returns the sum of lighting contributions from all lights.

With the `ILightEnv` interface abstracting the light integration logic over different representations of light collections, the next problem is how to construct an implementation to the `ILightEnv`

```
interface IIntegrator
{
    associatedtype TResult;
    TResult integrate<TLight : ILight>(TResult previous, TLight light);
}
interface ILightEnv
{
    TIntegrator.TResult aggregate<TIntegrator:IIntegrator>(TIntegrator integrator,
         TIntegrator.TResult initVal);
}
```

Listing 5.8: The `ILightEnv` interface in the refactored Falcor shader library representing a collection of lights. The interface declares that every light collection must provide an `aggregate` method, which can be used for iterating through the light collection for lighting computation.

```
struct StandardIntegrator<TBxDF : IBxDF> : IIntegrator
{
    typedef float3 TResult;
    TBxDF bxdf;
    float3 viewDir;
    float3 integrate<TLight : ILight>(float3 previous, TLight light)
    {
        float3 lighting = light.illuminate(bxdf, viewDir);
        return previous + lighting;
    }
};

float3 evalLighting<TBxDF : IBxDF>(TBxDF bxdf, float3 viewDir)
{
    StandardIntegrator i;
    i.viewDir = viewDir;
    i.bxdf = bxdf;
    return gLights.aggregate(i, 0.0);
}
```

Listing 5.9: An example shader code listing that computes light contributions from all lights defined in `gLights`. The listing first defines `StandardIntegrator`, which acts as a callback that will be executed on every light in the lighting environment. The callback accumulates lighting contribution from each given light. The `evalLighting` function simply calls the light collection's `aggregate` method with a `StandardIntegrator` instance to get the final lighting result from all lights in the `gLights` collection.

interface that represents a specific heterogeneous collection of light sources.

Listing 5.10 defines three implementations to the `ILightEnv` interface — `LightSingleton`, `LightArray` and `LightPair` — that can be composed to represent arbitrary collection of lights. `LightSingleton` wraps a single light into an `ILightEnv`. 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` is used to support a dynamically-sized, but homogeneous composition, the `LightPair` type can be used to compose a heterogeneous lighting environment. A light pair like

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

represents a light collection consisting of a directional light and a point light.

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 plus up to 16 point lights, it can construct the type:

```
LightPair<LightSingleton<DirectionalLight>, LightArray<PointLight, 16>>
```

We modified Falcor's host code to generate Slang types from the actual scene lighting environments. First, the lights are sorted and grouped by type. Next, from the collection of light types used in the lighting environment, the host code uses Slang compiler's runtime API (Section 4.3) to construct the composite lighting environment type. To communicate the light parameters, the host code queries the layout information of the constructed light environment type and stores the parameter values in a parameter block according to this layout.

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 in an intricate way (as discussed in Section 2.4.4), the shading system can specialize shader code to different lighting environments by composing shader types and creating instances of those types.

5.1.5 Generating and Looking Up Shader Variants

In the refactored branch, the host-side logic for implementing shader specialization is simpler and more efficient. Before drawing each object, Falcor needs to generate or find a specialized

```
// Represents a light environment consists of a single light
struct LightSingleton<L:ILight> : ILightEnv {
    L light; // the light object
    TIntegrator.TResult aggregate<TIntegrator:IIntegrator>(
        TIntegrator integrator, // the callback function to run on each light
        TIntegrator.TResult initVal // initial aggregate value to add to
        )
    {
        // since we have only a single light, run the callback
        // directly on the light.
        return integrator.integrate(initVal, light);
    }
}
// Represents a light environment consists of a homogeneous array of lights
struct LightArray<L:ILight, const N:int> : ILightEnv {
    L   lights[N]; // the homogeneous light array
    TIntegrator.TResult aggregate<TIntegrator:IIntegrator>(TIntegrator integrator,
                TIntegrator.TResult initVal)
    {
        TIntegrator.TResult result = initVal;
        // we have an array of lights, so loop over the array
        // and run callback on each light
        for (int i = 0; i < N; i++)
            result = integrator.integrate(result, light);
        return result;
    }
}
// Represents a light environment consists of two sub light environments
struct LightPair<H:ILightEnv, T:ILightEnv> : ILightEnv {
    H head; // the first child light environment
    T tail; // the second child light environment
    TIntegrator.TResult aggregate<TIntegrator:IIntegrator>(TIntegrator integrator,
                    TIntegrator.TResult initVal)
    {
        // we have two child light environments,
        // recursively call aggregate on them to cover all lights
        TIntegrator.TResult r1 = head.aggregate(integrator, initVal);
        return tail.aggregate(integrator, r1);
    }
}
```

Listing 5.10: Composite operators for defining heterogeneous lighting environments. A `LightSingleton` constructs a `ILightEnv` from a single light. A `LightArray` can be used to encapsulate a homogeneous array of lights, while `LightPair` can be used to "unroll" a heterogeneous list of lights.

shader variant based on the shading features used by the objects.

In the original branch, Falcor looks up shader variants in a cache based on the set of active `#define` strings, including any material or lighting environment specialization `#define`, such as the `#defines` and `typedefs` used to generate the specialized shader in Listing 2.1. However, since shader variant lookup is invoked for every object and for every frame, this string-based lookup is a significant source of CPU overhead.

In the refactored branch, the appropriate variant to use depends on the shader entry point in use and the classes of all its argument component instances. To look up a shader variant, Falcor concatenates the dynamic IDs of the entry point and component types in use to form a *variant key*, as discussed in Section 4.3.4. Note that dynamic IDs are assigned to component *types*, not instances, so even a 8-bit ID can be sufficient for Falcor, which has less than 256 component types in its shader library.

Our implementation uses a 64-bit integer as the variant key, which is sufficient for entry points using up to 7 parameter blocks (assuming 8 -bit dynamic IDs for entry points and types). The variant key is then used to index a hash table of previously compiled shader variants. If the requested variant is not found, Falcor invokes Slang to generate the shader kernels on-demand.

5.2 Experience with the Refactored Code Base

By rearchitecting Falcor using Slang, we believe the resulting code base is more modular and extensible. Being more modular means that the structure of the code base is more aligned with the developer's mental model. Being more extensible means that developers working with Falcor should find it easier to integrate a new feature. In this section, we report experiences with the refactored code base in both aspects.

5.2.1 Matching the Programmer's Mental Model More Accurately

In this section, we examine two cases where the refactored shader library represents a better match to the developer's mental model of the rendering concepts.

Reflection Light Probes are Lights

Rendering reflective surfaces is a challenging task in graphics. To evaluate the reflective appearance, the shader needs to know the incoming radiance from directions reflected by the surface

```
struct LightProbeData
{
    Texture2D lightProbeTexture;
    SamplerState sampler;
    float3 pos;
    float radius;
    float3 intensity;
    ...
};

float3 evalLightProbe(ShadingData sd, LightProbeData probe)
{
    ...
}
```

Listing 5.11: Falcor's original implementation of light probes.

point being shaded. In a traditional offline ray-tracing renderer, this knowledge is obtained by tracing a ray along the reflection direction and recursively evaluating irradiance at surface locations hit by the reflection ray. This process requires global scene data which is not available to the GPU graphics pipeline (which draws a single triangle at a time). To approximate this missing scene data, real-time renderers employ a mechanism called reflection light probes [2]. A reflection light probe is a texture image that acts as a cache of incoming radiance from all directions at a world position. When rendering a reflective surface, instead of tracing a reflection ray and recursively evaluating the irradiance, the renderer simply locates a light probe that is closest to the current surface location, and use the cached radiance from the light probe texture as an approximation to the true radiance from the requested direction.

Conceptually, light probes are regarded as just another type of scene light source. When included in light integration, they add a reflection term to the lighting result. However, implementing light probe as another type of light requires extending the `LightData` type with light-probe-specific fields. In contrast to a point light or directional light, a light probe is defined by a texture rather than a set of ordinary data fields (e.g., position and direction vectors). Many GPU platforms place a small limit on the number of texture (or other resource-typed) parameters that a shader can define. Including a texture (or other resource-typed) parameter in `LightData` means that the shader entry point in Listing 5.6 will be declaring `MAX_LIGHTS` texture parameters for the worst case scenario, regardless of how many light probes are actually in use. This severely limits the number of lights (i.e., the size of the `gLights` array) that can be supported by Falcor.

To work-around this issue, Falcor implements light probes as a one-off feature independently

```
cbuffer PerFrameData
{
    MaterialData   gMaterial;
    int            gLightCount;
    LightData      gLights[MAX_LIGHTS];
    int            gLightProbeCount;
    LightProbeData gLightProbes[MAX_LIGHT_PROBES];
    ...
};

float3 entryPoint(SurfaceGeometry geom)
{
    BxDF pattern = evalMaterialPattern(geom);
    float3 result = 0;
    for (int i = 0; i < gLightCount; i++)
        result += evalLight(pattern, gLights[i]);
    for (int i = 0; i < gLightProbeCount; i++)
        result += evalLightProbe(pattern, gLightProbes[i]);
    return result;
}
```

Listing 5.12: Updated shader entry point to incorporate the contributions of light probes in Falcor's original light probe implementation. The entry point is extended with additional parameters (`gLightProbeCount`, `gLightProbes`) and additional code that calls the `evalLightProbe` function.

from ordinary lights. Listing 5.11 illustrates Falcor's original light probes implementation, which consists of a `LightProbeData` type and an `evalLightProbe` function. Instead of extending `LightData` type with more fields, the shader parameters of a light probe are encapsulated in a standalone `LightProbeData`, which includes a handle to the light probe texture (`lightProbeTexture`), the world space position of the light probe (`pos`), and an `intensity` value that scales the radiance values in the light probe. The `evalLightProbe` function takes as input the current shading point and a light probe, and returns the reflected irradiance from the shading point, which can then be added to the lighting result.

Since light probes are treated as a standalone feature, the shader entry point (`entryPoint`) must also be extended to include the reflection term from light probes. As shown in Listing 5.12, additional shader parameters for light probes (`gLightProbes`) are declared, and calls to the `evalLightProbe` function are added to the entry-point to accumulate the reflection terms.

In the refactored branch, a light probe is implemented as a new type of light, as described in Section 5.2.2 and shown in Listing 5.13. Since `LightProbe` conforms to the `ILight` interface,

```
struct LightProbe : ILight
{
    Texture2D lightProbeTexture;
    SamplerState sampler;
    float3 pos;
    float radius;
    float3 intensity;
    ...
    float3 eval(SurfaceGeometry geom)
    {
        // same code as in the original evalLightProbe function
    }
}
```

Listing 5.13: Implementation of light probes in refactored Falcor renderer. A new `LightProbe` type is added to the shader library as another implementation of the `ILight` interface. No changes to shader entry point is required.

light probes can be communicated to the shader as part of the lighting environment(`gLightEnv`), and light probe evaluation code will be invoked automatically via the `gLightEnv.eval()` call in the existing shader entry point. No additional changes to the entry point are required to enable light probe feature. Because the composite `TLightEnv` type represents a specialized lighting environment with no unused fields, a light type implementation can include texture typed parameters without limiting the shading system's ability to support many lights.

Shadows are a Part of Lighting Computation

Falcor implements many different shadow algorithms, including cascaded shadow maps [27], variance shadow maps [24] and exponential shadow maps [5]. Ideally, shadows should be considered as a part of lighting computation: when computing lighting contribution from a light source with the shadow feature enabled, the shader should run one of the shadow algorithms to determine whether the light source is visible to the surface position, and accumulate lighting contribution from the light source only when the surface position is not shadowed.

However, to incorporate shadow algorithms as part of the lighting computation, the `LightData` type must also include all parameters required by the shadow map algorithm. Similar to light probes, implementing shadow map algorithm as part of lighting computation requires adding texture-typed fields to `LightData` type, which will limit Falcor's ability to support drawing objects using many lights at the same time.

```
cbuffer PerFrameData
{
    MaterialData   gMaterial;
    int            gLightCount;
    LightData      gLights[MAX_LIGHTS];
    int            gLightProbeCount;
    LightProbeData gLightProbes[MAX_LIGHT_PROBES];
    // parameter for evaluating shadow term for the first light
    CsmData        gShadowParams;
    ...
};

float3 entryPoint(SurfaceGeometry v)
{
    BxDF pattern = evalMaterialPattern(v);
    float3 result = 0;
    for (int i = 0; i < gLightCount; i++)
    {
        // if this is the first light, compute shadow terms
        float shadowFactor = 1.0;
        if (i == 0)
            shadowFactor = getShadowFactor(gShadowParams, v);
        // multiply lighting result with shadow factor
        result += evalLight(pattern, gLights[i]) * shadowFactor;
    }
    for (int i = 0; i < gLightProbeCount; i++)
        result += evalLightProbe(pattern, gLightProbes[i]);
    return result;
}
```

Listing 5.14: The updated shader entry point that enables shadow feature in original Falcor. Only the shadow parameters for the first light source in `gLights` is defined. The entry point is computing shadow term for the first light source only.

Falcor's work-around is similar to the light probe implementation: Falcor implements shadows as a stand-alone feature independent from lighting computation. The shader library provides a `CsmData` type to encapsulate all parameters to compute the shadowing term of a single light source, and a `getShadowFactor` function that returns a *shadow factor* of a light source, which is then multiplied to the lighting result from the same light source in the lighting computation. This design means that the shader entry point must be modified accordingly to invoke the `getShadowFactor` function for each light that casts shadows, and multiply the shadow factor with the lighting result when accumulating lighting contributions. Listing 5.14 shows the actual shader entry point implemented in Falcor. For simplicity, the entry point defines only one instance of shadow parameters (`gShadowParams`) for the first light in `gLights`, and only compute shadow terms for the first light. With this implementation, it is not possible to render a scene with more than one light casting shadows.

In the refactored branch, the choice of whether or not to compute shadows as well as what shadow algorithm to use is encapsulated as a part of `Light`. The refactored shadow algorithm implementation is illustrated in Listing 5.15. First, an `IShadow` interface is added to the shader library, which defines the interface that all shadow algorithm implementations should conform to. Listing 5.15 shows two example implementations: `NoShadow`, which always return `1.0`, causing a light to never cast any shadows; and `CsmShadow`, which implements the cascaded shadow map algorithm [27]. The shadow algorithms are integrated into the shading system as a member of light types. For example, the `PointLight` type now becomes a generic type parameterized on the shadow algorithm to use with the light source. Evaluation of the shadow term becomes a part of lighting computation, instead of an explicit step performed by the shader entry point. Similar to the implementation of light probes, the addition of shadow algorithms does not require changes to the shader entry point. To enable shadows, the host simply specializes a `PointLight<CsmShadow>` type and places it in the light environment. Without any additional effort, the refactored branch naturally supports rendering scenes with more than one light casting shadows.

5.2.2 Improved Extensibility: Adding New Light Types

Section 5.1.3 provided evidence of better extensibility of the refactored material shading code: the legacy and new material shading code are allowed to co-exist. In this section, we present a quantitative evaluation of extensibility, by measuring the development effort required to add a new polygonal area light type to both the original and refactored shading system. Fig. 5.2 shows a rendering with a `QuadLight` that uses linearly transformed cosines for approximate evaluation [40].

```
interface IShadow
{
    float getShadowFactor(float3 pos);
}

struct NoShadow : IShadow
{
    float getShadowFactor(float3 pos) { return 1.0; }
};

struct CsmShadow : IShadow
{
    // parameters for cascaded shadow maps
    ...
    float getShadowFactor(float3 pos)
    {
        // cascaded shadow map implementation
        ...
    }
};

struct PointLight<TShadow : IShadow> : ILight
{
    TShadow shadow;
    ...
    float3 illuminate<B:IBxDF>(B bxdf, float3 wo)
    {
        float shadowFactor = shadow.eval(bxdf.getPosition());
        float3 direction = normalize(position - bxdf.getPosition());
        return bxdf.eval(direction, wo) * intensity * shadowFactor;
    }
};
```

Listing 5.15: The refactored shadow algorithm implementation. An `IShadow` interface is defined, and different shadow algorithms are implemented as different types conforming to this interface. Lights that can cast shadows have a generic member defining how their shadow factor should be computed. The shadow algorithm is invoked as a part of lighting computation instead of an explicit external step.

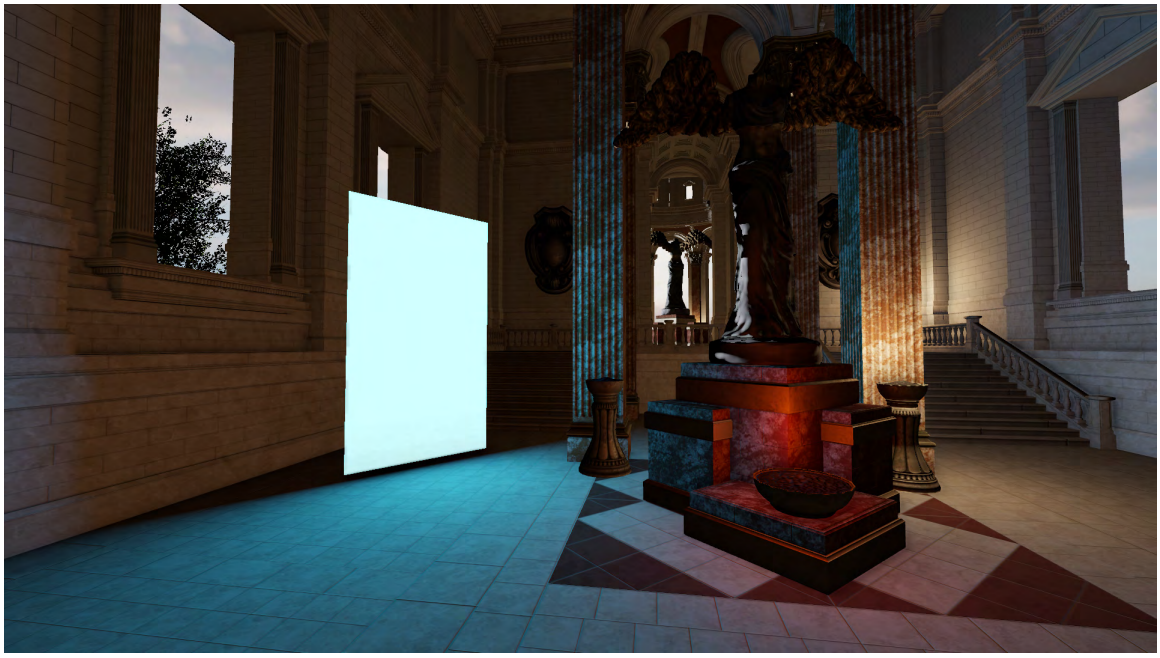


Figure 5.2: Rendering of a polygonal areal light in the TEMPLE scene. Support for polygonal areal lights was added to the Falcor renderer by changing a single code site.

Section 2.4.5 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., the interfaces defined in Listing 4.4 and Listing 5.10).

The `QuadLight` type in Listing 5.16 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 4.4 does not declare such a method.

Instead, the `acceptQuadLight` operation is defined as part of the `IAcceptQuadLight` interface in Listing 5.16. 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 [40].

The ability of extensions to "inject" code into existing types allows an application devel-

```
struct QuadLight : ILight
{
    float3 vertices[4], intensity;
    float3 illuminate<B:IBxDF>(B bxdf, float3 wo)
    {
        return bxdf.acceptQuadLight(this, wo);
    }
}
interface IAcceptQuadLight
{
    float3 acceptQuadLight(QuadLight light, float3 wo);
}
extension IBxDF : IAcceptQuadLight {}
extension Lambertian : IAcceptQuadLight
{
    float3 acceptQuadLight(QuadLight light, float3 wo)
    {
        return albedo * LTC_Evaluate(light, wo,
              float3x3(1,0,0, 0,1,0, 0,0,1));
    }
}
extension DisneyBRDF : IAcceptQuadLight { ... }
```

Listing 5.16: 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.

Branch	Sites Changed	Files Changed	Lines of code
Original	7	4	246
Original(with light specialization)	8	5	253
Refactored	1	1	249

Table 5.1: Comparison of development effort required to extend Falcor with a new type of light.

oper 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 2.4.5, the module in Listing 5.16 makes explicit the policy decisions concerning what has been extended and what the new interface requirements are (e.g., it is clear the interface contract of `IBxDF` has been extended).

Table 5.1 summarizes the shader code changes required in each version of Falcor. Adding

the area light feature to the original code required changes at seven sites in the code, spanning four different files. These changes included: 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. In all three branches, we added similar lines of code. This is because the majority of added code is the core logic of area light computation, and the refactored architecture is not introducing significant amount of boiler-plate code.

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. The total number of lines of shader code is comparable in all three branches. This is because most of the code being added is the core LTC lighting logic, and the additional boiler-plate code needed in the refactored branch is insignificant.

It is important to note that in the current implementation of `QuadLight`, any BxDF implementation that is not extended to support the new `IAcceptQuadLight` interface will result in a compiler error. The implementations of extensions in Swift and Rust require a default implementation of `acceptQuadLight` to be provided when extending `IBxDF`. Those implementations would then be used as a fallback for BxDFs that have not been explicitly extended. Allowing modules to evolve independently is an important modularity property. This property cannot be guaranteed by the compiler without supporting default implementations. We are considering adding this feature to Slang.

5.3 Performance Evaluation

5.3.1 Rendering Performance

The refactoring we have implemented 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 an optimized renderer using shaders specialized to materials and lighting envi-

TEMPLE

BISTRO INTERIOR

BISTRO EXTERIOR

Figure 5.3: Scene viewpoints used for evaluating performance of the refactored Falcor renderer built using Slang (see Fig. 5.4).

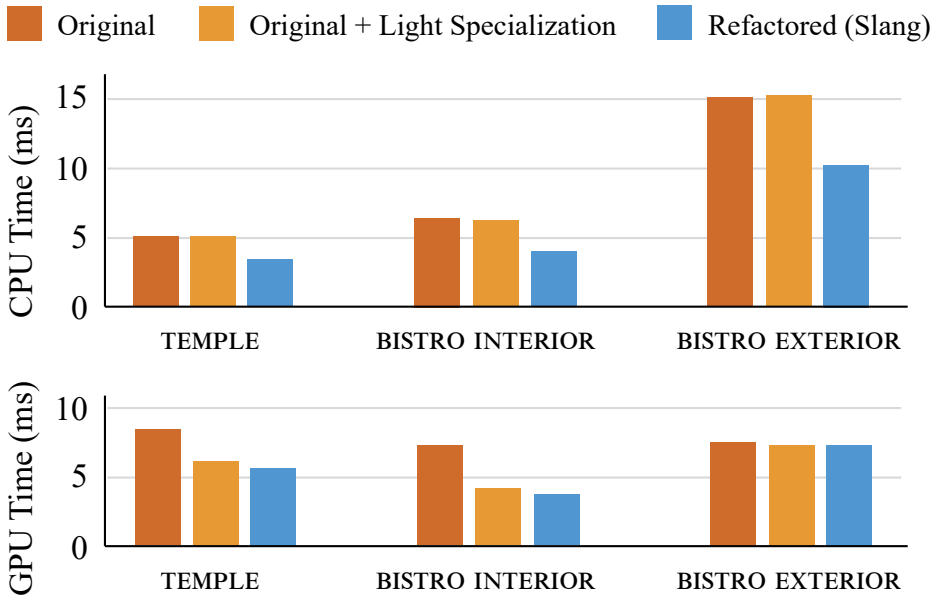


Figure 5.4: Comparing average per-frame CPU time (top) and GPU time (bottom) of the original branches (with and without light specialization), and the refactored branch of the Falcor renderer, we find that the refactored renderer improves CPU performance and preserves the same level of GPU performance as the original branch with light specialization.

ronment. Since the original branch of Falcor did not specialize shaders to lighting environments, in order to facilitate fair performance comparison, we forked the original branch and extended with support for light specialization using the approach discussed in Section 2.4.4.

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 [70]: TEMPLE, BISTRO-INTERIOR, and BISTRO-EXTERIOR (rendered views shown in Fig. 5.3). These scenes were created by developers of the Unreal and Lumberyard [3] 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. 5.4 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

Figure 5.5: A scene rendered using GPU ray tracing in Falcor, which uses Slang to compile shaders for new ray tracing shader stages.

up CPU resources for other game engine tasks. The refactored branch achieves higher GPU performance in two out of three scenes due to the added lighting specialization. While adding lighting specialization to the original branch brings its GPU performance on par with the refactored branch, the refactored branch does not involve such additional development effort for lighting specialization. These numbers confirm that Slang supports productive development of an efficient shading system that meets the performance promises of the shader components design pattern.

5.4 Random Access to Lights

In Section 5.1.4, we discussed a representation of heterogeneous light collections with generic compositional types (`LightSingleton`, `LightArray` and `LightPair`). This representation achieves two goals. First, it decouples the concept of a light collection from the types of lights supported by the system, so adding a new type of light to the shading system does not require modifying the light collection implementation. Second, it results in efficient GPU code due to the use of parametric polymorphism instead of subtype polymorphism, which enables the compiler to generate fully specialized code without using dynamic dispatch.

However, compared to a homogeneous array of a union light type (as in the original Falcor code base) or an array of an abstract light type (that uses subtype polymorphism to dynamically dispatch at runtime), the heterogeneous light collection in our refactored shader library lacks support of random access to its content. In both alternatives, a developer can use a simple array index to access any individual light in the collection in any order, which is not supported in our light collection representation.

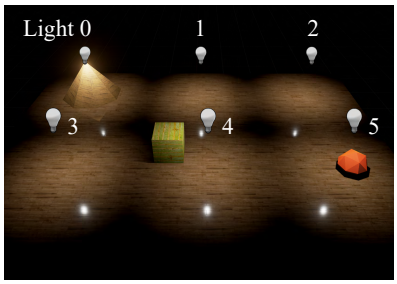
This limitation has not been a problem for Falcor, since Falcor implements a naive lighting integration algorithm that sums up lighting contribution from all light sources in the scene. In practice most lights only contribute to a small region of the scene and affects only a few pixels on the screen, so processing all lights is not efficient when rendering scenes with many lights.

A common optimizing strategy is to quickly filter out the lights that will not contribute to the color of a pixel and exclude them from lighting computation. Optimized deferred [54], and forward+ [36] renderers build culling data structures that provides a filtered subset of lights for each shading point. Moreover, optimized ray-tracing renderers compute lighting by *sampling* the light environment: the renderer picks a subset of lights based on the *weight* of each light, and renormalizes the computed lighting contribution to approximate the ground-truth. The weight of each light is determined by the power, proximity and the surface area of the light source. In both a deferred and a ray-tracing renderer, the shader needs random access to a subset of lights using one level of indirection, such as indices or pointers stored in the culling data structure.

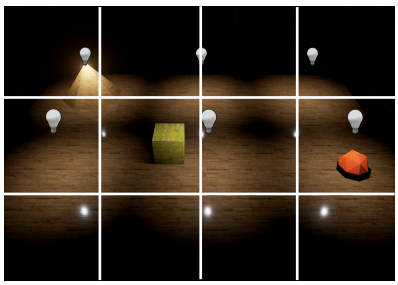
Can we extend our light collection representation to allow such random access without forfeiting its benefits (i.e., decoupling from light types and fully static dispatch)? Fortunately, the `ILightEnv` interface and the composite types defined in Listing 5.10 can be extended with a richer set of operations to support the more advanced access patterns in an optimized deferred or ray-tracing renderer. We present the solution in detail in the following subsections.

5.4.1 Compute Lighting Using a Subset of Lights

A deferred or forward+ renderer may choose to build a list of lights to use for lighting computation for each shading point. Fig. 5.6 illustrates a commonly used algorithm called screen-space tiled lighting. Assume we are going to render a scene containing 6 point lights, as shown in Fig. 5.6 (a). The lights are marked with numbers. The algorithm divides the screen into small tiles as shown in Fig. 5.6 (b). For each tile, the renderer builds a list of lights whose region of inference intersects with the world-space region represented by the screen-space tile. Fig. 5.6 (c) shows the resulting tiled light list. Each tile corresponds to a list of indices of the lights that



(a) example scene



(b) screen-space tiles

0	0, 1	1, 2	2
0, 3	0, 1, 3, 4	1, 2, 4, 5	2, 5
3	3, 4	4, 5	5

(c) tiled light list

Figure 5.6: Illustration of screen-space tiled lighting. (a) An example scene containing 6 point lights (marked by numbers). (b) The screen is divided into a 4×3 grid of tiles. (In practice, we use much smaller tile sizes such as 16×16 pixels per tile.) (c) A per-tile light list is constructed, which tells the shader what lights to use when computing lighting integration for pixels in each tile.

inference the tile. When computing lighting for each pixel, the shader only uses the light list corresponding to the screen-space tile containing the current pixel. Because the world region covered by the screen-space tile is small, the light list in each tile contains only a small fraction of lights, speeding up lighting computation by a large factor.

Since a light may interfere more than one tile, instead of duplicating the light data in many lists, the per-tile light list stores a single handle, through which the actual light data can be accessed from a light collection. When the light collection is represented as a flat array, this handle is simply a 0-based index into an array. However when the light collection is represented using the composite types in Listing 5.10, accessing a light from a handle becomes more complicated.

Listing 5.17 illustrates an extension to the `ILightEnv` interface that allows integrating a subset of lights. In this solution, a light is also indexed by a single integer. Each light in a composite light environment will be assigned an index starting from 0 in the natural order. For example, in a light environment represented by

```
LightPair<LightArray<PointLight, 10>, LightSingleton<DirectionalLight>>
```

the ten point lights will be assigned with indices 0 through 9, and the directional light can be accessed with index 10. This indexing scheme makes it easy to ensure consistency between the CPU code and the shader code: the host (CPU) code can still use a single array of `Light*` to represent the light collection, as long as the array is sorted by light type, and the composite light environment shader type is generated from the sorted host-site array.

The `indices` and `indexCount` parameters of `aggregateSparse` together define a list of indices

```
interface ILightEnv
{
    ...
    int getLightCount();
    TIntegrator.TRessult aggregateSparse<TIntegrator:IIntegrator>(
        TIntegrator integrator, // callback for each light
        TIntegrator.TResult initValue, // initial value to aggregate to
        int* indices, // pointer to the start of the list of indices
        int indexCount, // size of the list of indices
        int baseOffset, // for internal use
        out int numLightsProcessed); // returns the number of lights processed
}
```

Listing 5.17: Extension to the `ILightEnv` interface to allow computing light integration from a subset of lights in a deferred renderer. The `indices` parameter represents a list of 0-based light indices. Only lights specified by the `indices` list will be used for light integration.

that should be used to compute the aggregate result. The `baseOffset` parameter, which will be discussed shortly, is used for internal implementation only. The entry point should always pass 0 to `baseOffset` when calling `aggregateSparse`.

Listing 5.18 shows the extended `LightArray` implementation of the new `aggregateSparse` method. This implementation assumes that the list of indices is sorted in ascending order, which can be easily guaranteed when building the tiled light lists. The key in this implementation is to translate an index in `indices` into the local index that can be used to access the `lights` array. To do so, the function needs to know the *global* light index corresponding to the first light in `lights` array. This is provided via the `baseOffset` parameter, and in the case that the `LightArray` is wrapped inside a `LightPair`, the `LightPair` implementation is responsible for providing the correct `baseOffset` when it makes recursive calls into `LightArray.aggregateSparse`. The function simply iterates through the list of indices, subtracts the input index by `baseOffset` to obtain the *local* index, and uses it to access the `lights` array. The function also returns the number of lights that has been processed in the `indices` list via the `numLightsProceseed` parameter. Because we have already known that the indices come in ascending order, we can return immediately when we see an index that is out of the range of the `lights` array, since there can't be more indices in the list that falls within the range of current array.

Listing 5.19 shows the `LightPair`'s implementation of the `aggregateSparse` method. The source code should be self-explanatory. The key responsibility of `LightPair` is to make recursive calls to `head`'s and `tail`'s `aggregateSparse` method, and provide the adjusted `baseOffset` and `indices`

```
struct LightArray<TLight:ILight, let N: int> : ILightEnv
{
    TLight lights[N];
    int getLightCount() { return N; }
    TIntegrator.TRessult aggregateSparse<TIntegrator:IIntegrator>(
        TIntegrator integrator, TIntegrator.TResult initValue,
        int* indices, int indexCount, int baseOffset, out int numLightsProcessed)
    {
        TIntegrator.TResult result = initVal;
        numLightsProcessed = 0;
        for (int i = 0; i < indexCount; i++)
        {
            // adjust index for local array
            int idx = indices[i] - baseOffset;
            // if idx is beyond the range of local array,
            // we have already done processing this array.
            if (idx >= N)
                break;
            // otherwise integrate this light
            result = integrator.integrate(result, lights[idx]);
            numLightsProcessed++;
        }
        return result;
    }
}
```

Listing 5.18: Implementation of `LightArray` for indexed light access.

arguments to ensure correct index translation.

5.4.2 Sampling the Light Environment

Listing 5.20 shows further-extended light environment definitions that support sampling based light integration used by a ray-tracing renderer. First, the `ILight` interface is extended with a `getWeight()` method. A light with bigger weight is more likely to be sampled. Next, the `ILightEnv` interface is extended with `getSumWeights` and `sampleLight` methods, which can be used together to sample the light collection.

Listing 5.21 shows a sampling algorithm implemented using the new `ILightEnv` interface. The algorithm calls `ILightEnv.getSumWeights` to obtain the total weights summed from all lights in the collection. For each sample, a random value between 0 and `sumWeights` is generated, and this value is used to locate the corresponding light source, which is then used to accumulate

```
struct LightPair<H:ILightEnv, T:ILightEnv> : ILightEnv
{
    H head;
    T tail;
    ...
    int getLightCount()
    {
        return head.getLightCount() + tail.getLightCount();
    }

    TIntegrator.TRessult aggregateSparse<TIntegrator:IIntegrator>(
        TIntegrator integrator, TIntegrator.TResult initValue,
        int* indices, int indexCount, int baseOffset, out int numLightsProcessed)
    {
        int numProcessed1 = 0;
        int numProcessed2 = 0;

        // recursively call aggregateSparse in two sub-environments
        TIntegrator.TResult r1 = head.aggregateSparse(integrator, initValue,
                  indices, indexCount, baseOffset, numProcessed);
        TIntegrator.TResult r2 = tail.aggregateSparse(integrator, r1,
            indices + numProcessed1, indexCount - numProcessed1,
            baseOffset + head.getLightCount(), numProcessed2);

        numLightsProcessed = numProcessed1 + numProcessed2;
        return r2;
    }
}
```

Listing 5.19: Implementation of `LightPair` for indexed light access.

light contribution. For example, assume a light environment that contains three point lights with weights `0.5`, `0.3` and `0.1`. The algorithm first computes the sum of all weights, which will be `0.9`. Next, for each sample, a random value will be chosen in the range of `[0, 0.9)`. If the random value (`sample`) is `0.6`, then the second light will be picked for lighting computation, and its contribution will be accumulated to `lightResult`.

`getSumWeights` and `sampleLight` methods can be implemented by induction in a similar way to `aggregate` and `aggregateSparse`. We omit the implementation for `getSumWeights` here, as it is mostly the same as `aggregate`. Listing 5.22 shows the implementation of sampleLight for both `LightArray` and `LightPair`. For `LightArray`, we simply call each light's getWeight until we have located the weight range in which the `sample` falls, and invoke the integrator using the current light. For

```
interface ILight
{
    float3 illuminate<B:IBxDF>(B bxdf, float3 wo);
    float getWeight();
}

interface ILightEnv
{
    ...
    float getSumWeights();
    TIntegrator.TResult sampleLight<TIntegrator:IIntegrator>(
        TIntegrator integrator, TIntegrator.TResult initValue, float sample);
}
```

Listing 5.20: Extended lighting environment interfaces for sampling based lighting. `ILight` is extended with the `getWeight` method, which determines how likely the light will be sampled. `ILightEnv` is extended with the `getSumWeights` method, which returns the sum of all lights' weights, and the `sampleLight` method, which accumulates lighting contribution from a light corresponding to a sampled position.

```
float3 sampleLighting<TLightEnv:ILightEnv, TBxDF:IBxDF>(TLightEnv lightEnv,
    int numSamples, TBxDF bxdf, float3 wo)
{
    StandardIntegrator integrator;
    integrator.viewDir = wo;
    integrator.bxdf = bxdf;
    float sumWeight = lightEnv.getSumWeights();
    float3 lightResult = 0.0;
    for (int i = 0; i < lightSamples; i++)
    {
        // pick a random value between 0 and sumWeight
        float sample = random(sumWeight);
        // find the light corresponding to the sampled location
        // and integrate lighting using the light
        lightResult = lightEnv.sampleLight(integrator, lightResult, sample);
    }
    return lightResult;
}
```

Listing 5.21: A light sampling algorithm using the new `ILightEnv` interface.

```
struct LightArray<TLight:ILight, let n:int> : ILightEnv
{
    ...
    TIntegrator.TResult sampleLight<TIntegrator:IIntegrator>(
        TIntegrator         integrator, // callback to process each light
        TIntegrator.TResult initValue, // initial result value at start of execution
        float               sample)    // a randomly generated sample value in [0,1]
    {
        // find the light corresponding to sample
        int range = 0.0;
        for (int i = 0; i < N; i++)
        {
            range += lights[i].getWeight();
            if (range > sample) // we found the light
            {
                return integrator.integrate(initValue, lights[i]);
            }
        }
        return initValue;
    }
}

struct LightPair<THead:TLightEnv, TTail: TLightEnv> : TLightEnv
{
    ...
    TIntegrator.TResult sampleLight<TIntegrator:IIntegrator>(
         TIntegrator         integrator, // callback to process each light
         TIntegrator.TResult initValue, // initial result value at start of execution
         float               sample) // a randomly generated sample value in [0,1]
    {
        // find the sub-part where sample falls in
        float range = head.getSumWeights();
        if (sample < range)
            return head.sampleLight(integrator, initValue, sample);
        else
            return tail.sampleLight(integrator, initValue, sample-range);
    }
}
```

Listing 5.22: Implementation of the new `ILightEnv` interface for light sampling.

`LightPair`, we call `getSumWeights` recursively on both `head` and `tail` to determine in which sub-part the sample falls, and make recursive `sampleLight` calls to the corresponding sub-part.

For clarity, we have presented a simplistic sampling algorithm which takes $O(n)$ time to locate a light object from a sample value. One can consider replacing the linear search process in Listing 5.22 with a more efficient binary search process($O(\log n)$ complexity) or the alias method ($O(1)$ complexity with an $O(n)$ preprocessing step) [95]. Regardless of the choice of sampling algorithms, the basic idea to incorporate sampled lighting into the composite light environment architecture stays the same.

5.5 Developer Feedback

The true evaluation of Slang is to use it in a major software code base. Although Slang has not yet been integrated in any large production renderers, it is being used as the sole shading language for Falcor at the time of writing. Falcor's entire shader library, including the shader code for ray tracing, totaling over 15,000 lines of code, is being compiled by the Slang compiler. Maintaining compatibility with HLSL is the key design decision that allowed successful adoption of Slang, since it allows existing shader code to continue working after switching the shader compiler to Slang. This makes it possible for the Falcor development team to incrementally adopt more advanced language features over time. Researchers who are using Falcor to experiment with various rendering algorithms are also willing to accept the switch from HLSL to Slang, since they will benefit from the improved extensibility and performance of the new Falcor system without needing to learn new language features or modify their existing shader code.

Upon seeing the benefits of the refactored branch, the Falcor development team is taking steps to merge changes we made in the refactored branch to the main development branch. Falcor is already using `ParameterBlock` to communicate shader parameters. Falcor's host- side logic has been modified to use Slang's runtime API to fill in parameter values into parameter blocks, and to generate shader kernels for both Direct3D 12 and Vulkan platforms. Developers have agreed that the use of `ParameterBlock` makes shader code cleaner and simpler, and improves the system performance in communicating parameters. As a next step, the Falcor development team is planning to integrate our refactored material shading logic into the main branch.

While we have demonstrated generics as an effective mechanism for shader specialization, developers have expressed concerns regarding Slang's generics syntax. Specifically, we heard from the Falcor development team that the refactored `NewMaterial` type as presented in Sec-

tion 5.1.3 is defining too many generic type parameters, which reduces code readability. This issue can be alleviated by adding syntax support for interface-typed parameters and fields that have a static polymorphism semantics, similar to Cg [75]. For example, the following material definition can be transformed automatically by the compiler into the same definition as in Listing 5.5:

```
struct NewMaterial : IMaterial
{
    IChannel baseColor;
    IChannel specular;
    IChannel emissive;
}
```

For this specific case, the syntactic transformation is straightforward: when the compiler sees a `struct` field that has an interface type, it turns the enclosing `struct` into a generic `struct` with a generic type parameter bound to the same interface, and replaces the type of the field with the generic type. While functionally equivalent in theory, adding support for interface-typed values is worthwhile as it greatly simplifies code that uses static polymorphism.

We have also heard from the Falcor development team that our refactored lighting environment representations makes it harder to access individual lights in the heterogeneous light collection. In the original branch, the lighting environment is a simple array, which is straightforward to iterate and access by index. However, in the refactored branch, implementing a new type of operation on the light environment requires adding new methods to the `ILightEnv` interface and implementing the new methods for the composite typoes (`LightSingleton`, `LightArray` and `LightPair`) by induction. While these composite types are necessary to enable static specialization of lighting environments, as will be discussed in detail in Section 7.2.2, language features such as pattern matching and higher-order functions can make it easier to work with them. Another source of complexity regarding the lighting environment is that the refactored shader library is using cascaded `LightPair` types to resemble a `tuple`. Adding compiler support for tuples will also simplify the lighting environment representation.

Despite the limitations in Slang's current implementations, developers from multiple groups within NVIDIA believe that the services provided by Slang are valuable to their projects. As a result, at the time of writing this thesis, two full-time developers now work on continued development of the Slang project to turn it into a more robust and useful system so that more teams and projects can adopt Slang for their shader compilation needs.

Chapter 6

Related Work

The key ideas of the Shader Components design pattern and the Slang shading language are inspired by extensive prior work on shading system and shading language design, general programming language mechanisms, and software development paradigms. This chapter begins with a brief review of the real-world systems and their solutions to shader programming. We summarize recent advances in shading languages and compilers in terms of improvements for shader modularity, shader specialization and shader-specific compiler optimizations. Next, we discuss the general language mechanisms that are adopted by Slang or related to the problems in shading systems. Finally, we discuss Slang's relationship with the aspect oriented programming paradigm and the multi-rate programing model.

6.1 Practices in Game Development

The example shading system design presented in Section 2.4 reflects the practices in many game engines such as Unreal Engine [28], Unity [92] and Lumberyard [3]. In particular, both Unreal Engine and Lumberyard Engine generate specialized shader variants by pasting different code snippets via preprocessor `#includes`. Almost all game engines leverage conditional compilation via preprocessor `#if` directives for fine grained specialization.

Similar to our example system design, the Unreal Engine exposes important shader features to host code as C++ objects, which holds the parameters for the shading feature and are used to generate specialized shader code. The shader components design pattern formalizes this practice with the component types and component instances.

Bungie's TFX shading system [91] uses a custom DSL that is highly integrated with a spe-

cific engine. The DSL is built on top of existing shading languages with ad-hoc string processing techniques to generate specialized shader variants. TFX requires the programmer to manually determine parameter block layouts for each shader entry point, which are then validated at run time by the system against the selected components. In contrast, the shader components design avoids such runtime validation by generating the parameter layout automatically from the definition of a component type, and statically checking that a component can be used with a shader entry point at compile time.

We hope that the language features built into Slang will eliminate the need for shading system programmers to write their own ad-hoc implementations of shader compilation tools and make it substantially easier for developers to build a shading system following either the shader components or Bungie's TFX design.

6.2 Advances in Shading Languages

6.2.1 Shader Modularity

Previous efforts have attempted to improve shader modularity by adding more-flexible language mechanisms to real-time shading languages, including Cg interfaces [75], HLSL classes and interfaces [61], GLSL shader subroutines [48], RTSL functions [80] and Sh classes [58].

Cg interfaces [57, 75], HLSL interfaces [61], and Spark [31] allow components to be expressed in a shading language using object-oriented classes that encapsulate both code and associated (parameter) data. Slang follows this path with its `struct` types.

GLSL shader subroutines [46, 48] 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).

RTSL [80] uses function syntax to represent components, but takes inspiration from the RenderMan Shading Language, where shaders are semantically treated as classes [35, Section 3.3]. However, the runtime API for RTSL conflates shader component classes and instances. The effect is as though only a single instance of each class is allowed.

The Sh shader metaprogramming system [59] supports the construction of abstractions like our surface shader separation (Section 5.1.3) and composite lighting environments (Section 5.1.4), using C++ templates. Examples similar to ours can be found in the companion book for Sh [58]. Sh is an embedded DSL which relies on runtime metaprogramming to generate its intermediate

representation (IR). In contrast, Slang is an extension of an existing shading language, and its generics can be statically checked at compile time.

None of the prior systems support associated types or retroactive extensions, which we found necessary to implement our modular and extensible shading system, as described in Chapter 5.

The composite light environment types in our refactored Falcor shader library (Section 5.1.4) embody the ideas from functional programming: the lighting computation from a complex light collection can be composed from simpler functions for a single light. There are many prior efforts in developing functional languages for shader code, such as Vertigo [26], Renaissance [12] and GPipe [15]. However, recent advances in graphics pipelines have given shader kernels the capability of generating side-effects such as writing to buffers in global memory. These prior efforts are either purely functional with no support for operations with side-effects, or rely on additional design patterns (such as monads) that are not familiar to most graphics developers. In contrast, Slang follows HLSL with a traditional C-like imperative syntax, and supports functional programming paradigms using features that are more familiar to the graphics community, such as generics and associated types.

Slang's `ParameterBlock<T>` feature represents the same idea as the `ArgumentBuffer` in the Metal Shading Language [7]. OpenGL's bindless texture extension [19] also offers similar functionality by allowing both resource and ordinary parameters to be communicated via a constant(uniform) buffer. Slang's `ParameterBlock` represents a layer of abstraction over different graphics APIs for cross-platform compatibility, while the Metal Shading Language's `ArgumentBuffer` and OpenGL's bindless texture extension are platform-specific features.

At the time of writing, there is also interest in adapting C++ for use in shader programming. While traditional C++ templates prohibit early type checking for template code prior to specialization, the incoming *concept* feature [90] that was recently voted into the C++20 standard will allow C++ to achieve the same benefits of the language mechanisms we introduced to Slang (e.g., generics and associated types). However, as previously stated, this thesis focuses on answering the more fundamental question of which language mechanisms are preferable for achieving performance and extensibility goals in the context of a shading system. We hope our efforts serve to highlight the language features that are essential or preferred in order to support extensible and high-performance shading systems.

6.2.2 Shader Specialization

The Shader Components design pattern streamlines the process of shader specialization and aligns it with a shading system's modular decomposition. Various efforts have been made to simplify shader specialization.

Cg and HLSL both allow a shader entry point, as well as arbitrary functions, to be parameterized on interfaces. Cg, Spark and HLSL 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. Slang's runtime API gives the programmer explicit control on when to perform specialization. The reason for this differentiation is that 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.

The Vulkan API allows shaders to use compile-time constant parameters called "specialization constants" [50] (Metal supports a similar feature [43]). 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.

6.2.3 Compiler Optimizations for Shader Code

Besides generating specialized code, various language and compiler mechanisms have been proposed to optimize a shader program's computation logic for faster execution time. These efforts usually generate a simplified version of the original shader code, which produces approximate results that represent a trade-off between accuracy (image quality) and shader execution performance. These techniques include (a) code transformations that are local to a single shader stage (intra-stage optimizations) [72, 74, 89] such as removing instructions from fragment shaders and automatically lowering the floating point precision [78]; (b) global compiler optimizations that simplifies the entire shader program for all stages of the graphics pipeline [37, 97]; and (c) shading languages that enable developers to rapidly explore different shader simplification choices [38].

While simplifying the core shading logic is useful in achieving higher performance, the overall shading system performance depends on both GPU efficiency (i.e., how fast the GPU can finish executing the shader code) and CPU efficiency (i.e., the host-side logic to change pipeline state and communicate parameter). This thesis is motivated by the observation that the major challenges to game developers are in architecting the shading system to achieve both CPU and GPU efficiency while retaining the modular and extensible code base, which is not well-studied in previous efforts.

6.3 General Language Mechanisms

One design principle of Slang is to extend HLSL with existing programming language mechanisms that are familiar to graphics developers. As discussed in Chapter 4, Slang's language design borrows from Rust and Swift, which in turn adopt many concepts from functional languages like Haskell. In this section, we discuss related efforts from the programming language community on two important features adopted by Slang: associated types and retroactive extensions.

6.3.1 Associated Types and Virtual Classes

Slang chooses to support associated types, a feature available in Swift [10], and similar to abstract type members in Scala [85]. This is not to be confused with a more expressive feature called virtual classes in Beta [56]. A virtual class is similar in spirit to virtual functions. Virtual functions enable dynamic dispatch to different function implementations from a single abstract object handle, based on the run-time type of the referenced object. Similarly, virtual classes are members of an object that are not functions, but types. The type member can be accessed from an abstract polymorphic handle, and therefore the actual type being used is dependent on the type of the referenced object at runtime.

Virtual classes are mostly used in conjunction with inheritance. Consider the implementation of two material types, `Metal` and `RustMetal`, as shown in Listing 6.1. The code uses inheritance to reuse the logic in `Metal` and to achieve polymorphism. By inheriting from `Metal`, `RustedMetal`'s `BxDF` type (line 10) is also inheriting from `Metal.BxDF` (line 3), so that `RustedMetal.BxDF` will contain both the `roughness` field and the `rustExtent` field. Functions that expect `Metal` can continue to work with `RustedMetal` without any change. Note that the type of `bxdf` (line 23) in the `evalMetalLighting` function is `m.BxDF`, indicating that its type is determined at runtime from the actual instance passed

```
 1: struct Metal : IMaterial
 2: {
 3:    struct BxDF
 4:    {
 5:        float roughness;
 6:        float evalBxDF(float3 wo, float3 wi) {...}
 7:    };
 8:    BxDF evalMaterial(SurfaceGeometry geom) {...}
 9: };
10: struct RustedMetal : MetalMaterial
11: {
12:     // RustMetalMaterial.BxDF automatically inherits from MetalMaterial.BxDF
13:     // so it will contain both roughness and rustExtend fields.
14:     struct BxDF
15:     {
16:         float rustExtent;
17:         float evalBxDF(float3 wo, float3 wi) {...}
18:     };
19:     BxDF evalMaterial(SurfaceGeometry geom) {...}
20: };
21: float3 evalMetalLighting(MetalMaterial m, SurfaceGeometry geom, PointLight l)
22: {
23:     m.BxDF bxdf = m.evalMaterial(geom);
24:     return evalLighting(bxdf, l);
25: }
```

Listing 6.1: An example of virtual classes. `RustedMetal` inherits from `Metal`, allowing it to override both the implementation of the `evalMaterial` method and the `BxDF` type member. All existing functions that accept a `Metal` can work with `RustedMetalMaterial` arguments since `BxDF` is a virtual class. Its concrete type is determined at runtime from the instance of the object, not from the compile time type of the function parameter.

in as parameter `m`.

Associated types are a more limited form of virtual classes in that all types must be statically resolvable in compile time. Slang requires that the concrete path to a type must be directly available at the time a generic function is specialized. Therefore clauses such as `m.BxDF` (Listing 6.1, line 23), where the actual type is not known until runtime, are not allowed.

While virtual classes have greater expressive power, it is difficult for the compiler to statically derive the uses of a virtual class to a known type at compile time. Since generating statically specialized code is a priority for Slang, and since Slang does not allow inheritance, we choose to support only associated types, which is sufficient for all use cases we have encountered so far.

As we will discuss in Section 6.5, one way to extend Slang to fully support the multi-rate programming model is to introduce virtual classes to the language.

6.3.2 The Expression Problem

Slang's extension feature addresses the fundamental software engineering problem known to the programming language community as the *expression problem* [23, 81, 94].

In graphics, a common instance of the expression problem is the selection of light integration logic. The light integration step takes as input both the BxDF of the surface and those of the light sources. It is desirable to make a shading system extensible on both the types of BxDFs and types of light sources. Depending on the concrete type of BxDF and light source in use, the shading system should select the light integration code that best handles the combination of inputs.

The technique used in Section 5.2.2 to dispatch on the type of both a light and BxDF is known as *double dispatch* [44]. Implementing new features with extensions can lead to issues in certain situations. Most notably, using a module that adds an interface requirement but does not provide implementations for all types conforming to that interface breaks compilation. For example, consider a module that wants to add a `getBaseColor` method to the `IBxDF` interface presented in Listing 4.3:

```
interface IBxDF_Ext
{
    float3 getBaseColor();
}
extension IBxDF : IBxDF_Ext {}
```

This module is expected to provide implementations of `getBaseColor` for all existing types that conforms to the `IBxDF` interface via extensions:

```
extension Lambertian : IBxDF_Ext
{
    float3 getBaseColor() {...}
}
extension DisneyBRDF : IBxDF_Ext { ... }
```

If this module does not provide implementations for all existing BxDF types, using the module would result in compilation error. This could happen when the developer of the core shader

library adds a new type of BxDF at the same time as another developer adds an module that extends the `IBxDF` interface. A possible solution is to support "default" implementations: when defining the `getBaseColor` requirement in `IBxDF_Ext`, we can provide a default implementation that will be used for types that do not provide a specific implementation. However, supporting default implementations means that the compiler will no longer issue error messages for types that do not provide an implementation to a requirement. This weakens the compiler's capability to ensure code completeness.

Another issue of using extensions is that they can introduce conflicts. For example, using two extensions that add a method of the same name to an existing type would result in invalid code. This can be addressed with more-advanced scoping and disambiguating rules for naming, but it also makes code more confusing.

Some languages in the tradition of Common Lisp, such as Julia [16], support the more general mechanism called *multi-methods* [67]. A multi-method is similar to a group of overloaded functions in C++. For example, the following code defines a set of implementations of `integrateLighting` that computes lighting from different combinations of BxDF types and light source types:

```
float3 integrateLighting(LambertianBxDF, PointLight);
float3 integrateLighting(LambertianBxDF, AreaLight);
float3 integrateLighting(DisneyBxDF, PointLight);
float3 integrateLighting(DisneyBxDF, AreaLight);
```

The difference between a multi-method with function overloading is that calls to a multi-method are dispatched at runtime based on the dynamic types of arguments, instead of being statically resolved at compile time. This allows authoring more flexible code that calls a multi-method using arguments of abstract types, such as the following call site:

```
float3 evalLighting(IBxDF bxdf, ILight1 l1, ILight2 l2)
{
    return integrateLighting(bxdf, l1) + integrateLighting(bxdf, l2);
}
```

This code is not valid in C++ (which does not support multi-methods), since the calls to the `integrateLighting` multi-method cannot be resolved at compile time without knowing the concrete type of `bxdf`, `l1` and `l2`.

The OptiX ray tracing framework [73] and the DirectX Ray Tracing API [84] use a mechanism that is similar to multi-methods to select intersection handling logic based on both the

object's material and the type of the intersecting ray (e.g., shadow ray or camera ray) at runtime. Multi-methods could potentially be used to simplify the approach to light/surface interaction presented in this thesis, and may become a useful mechanism to model the dispatching behavior in the ray tracing systems.

However, adding multi-methods to Slang raises challenges in static type checking. In the context of Slang, the above `evalLighting` function would be written as a generic function that implies static dispatch:

```
float3 evalLighting<TBxDF:IBxDF, TLight1:ILight, TLight2:ILight> (
    TBxDF bxdf, TLight1 l1, TLight2 l2)
{
    return integrateLighting(bxdf, l1) + integrateLighting(bxdf, l2);
}
```

Recall that Slang is designed with the goal of fully type-checking generic functions and reporting code errors before a generic function is specialized. Supporting multi-methods conflicts with this goal and would potentially result in most type checks being deferred at specialization time, similar to C++ templates. This slows down compilation speed and can produce confusing error messages.

6.4 Aspect-Oriented Programming

Aspect-oriented programming [51] is a programming paradigm that enables implementing a concern that cross-cuts different parts of an existing code-base in a standalone module, without modifying existing code. A typical example of aspect oriented programming is the implementation of logging functionality: instead of directly modifying various existing functions to insert logging logic, a programmer can define an *aspect* as a standalone code module, which contains all the logging logic that will be injected to various functions by the compiler.

The use of extensions to implement a new BxDF dependent light type in our case study shares the spirit of aspect-oriented programming: all light shading and integrating logic related to the new area light type is implemented in a standalone code module as an extension to existing code base. A difference in terms of mechanisms is that most aspect-oriented programming languages (e.g. AspectJ[1]) support injecting code snippets into existing function bodies, while extensions in Slang only operate at function granularity, by injecting more requirements to interfaces or functions to existing types.

Aspect oriented programming is considered to be more closely related to multi-rate programming discussed previously [32], where a multi-rate component can be viewed as an *aspect*, and computations defined in the component correspond to *advises* that are injected to different stages of pipeline execution. Through this lens, authoring multi-rate components in Slang can be viewed as an instance of aspect oriented programming.

6.5 Multi-Rate Shading Languages

Some research shading languages propose novel language mechanisms based on *multi-rate programming* that can improve the modularity or extensibility of shader code. The idea of multi-rate programming languages is that a single expression may contain terms that are executed at different graphics pipeline stages. RTSL [80], Spark [31] and Spire [38] allow shader code to be divided into separately-defined components, and allow a single component to define code in multiple pipeline stages.

For example, Listing 6.2 illustrates a shader component implemented in Spark that computes lighting contribution from a point light. The point light logic is expressed as a `mixin` that extends the `Base` shader entry point (which already defines `inputVertex`) by injecting additional lighting computation logic. Note that the `MultiRatePointLight` shader module is defining computation that should happen in more than one shading stages: the incident light direction `L` is computed at the vertex shading stage, and the dot product `nDotL` as well as the final lighting result is computed at the fragment shading stage. The Spark compiler will automatically split the logic defined in `MultiRatePointLight` into two shader kernels.

The multi-rate nature of Spark makes it easy to exploit a graphics pipeline's different execution rates. In a typical scenario, the number of vertices processed by a graphics pipeline is less than the number of fragments. Thus, doing computation in the vertex shading stage can reduce the total amount of computation.

When designing Slang we chose not to adopt the idea of multi-rate programming in order to maintain compatibility with HLSL and familiarity to shading system developers. We found that in most cases, it is preferable to implement a shading feature in a single shading stage instead of separating its computation into multiple stages, because the amount of reduced computation is usually not sufficient to offset the overhead of additional communication required to pass the intermediate result between shading stages.

Despite the limited practical use of multi-rate programming, we note that most of its uses

```
mixin shader class MultiRatePointLight extends Base
{
    input @Uniform float3 lightPosition;
    input @Uniform float3 lightIntensity;

    @Vertex float3 N = inputVertex.normal;
    @Vertex float3 L = normalize(lightPosition - inputVertex.pos);
    @Fragment float nDotL = max(0, dot(N, L));
    @Fragment float lightingResult = lightIntensity * nDotL;
}
```

Listing 6.2: A shader component written in Spark that computes lighting contribution from a point light. By using rate qualifiers (e.g., `@Vertex` and `@Fragment`), the code computes the incident light direction `L` in vertex shader, and the dot product (`nDotL`) in the fragment shader. Shader parameters (`lightPosition` and `lightIntensity`) are specified via the `@Uniform` qualifier.

can be encoded using associated types, since both are related to the notion of *virtual classes* as in Beta [56]. In Slang, we can define an interface for component types that contain computation spanning more than one shading stage, even though the definition of the encapsulated logic itself is partitioned into two "single-rate" functions:

```
interface IMultiRateComponent
{
    associatedtype PerVertexResult;
    PerVertexResult perVertexComputation(InputVertex vtx);
    float3 perFragmentComputation(PerVertexResult v);
}
```

The `IMultiRateComponent` defines two methods for a multi-rate component type to implement — `perVertexComputation`, which will be called from a vertex shader kernel, and `perFragmentComputation`, which will be called from a fragment shader kernel. To allow the vertex and fragment shader entry points to name and communicate the values computed by the component, `IMultiRateComponent` defines an associated types called `PerVertexResult` to represent the values computed in the vertex shading stage.

Given the definition of `IMultiRateComponent`, we can author the vertex and fragment shader entry points to make use of a multi-rate lighting component, as shown in Listing 6.3. In contrast to languages like Spark and Spire where the compiler automatically generates the entry points that communicate per vertex values between vertex and fragment stages, a Slang programmer explicitly states this cross-stage communication by routing the `PerVertexResult` in the `VertexOutput`

```
type_param TLighting : IMultiRateComponent;
ParameterBlock<TLighting> light;
struct VertexOutput
{
   float4 projectedCoord : SV_POSITION;
   TLighting.PerVertexResult lightingVertexResult;
};
VertexOutput vertexMain(InputVertex v)
{
    VertexOutput result;
    result.projectedCoord = ...;
    result.lightingVertexResult = light.perVertexComputation(v);
    return result;
}
float3 fragmentMain(VertexOutput v)
{
    return light.perFragmentComputation(v.lightingVertexResult);
}
```

Listing 6.3: Vertex and fragment shader entry points using a generic, multi-rate lighting component.

```
struct MultiRatePointLight : IMultiRateComponent
{
    float3 lightPosition;
    float3 lightIntensity;

    struct PerVertexResult { float3 N, L; };
    PerVertexResult perVertexComputation(InputVertex vtx)
    {
        PerVertexResult rs;
        rs.N = vtx.normal;
        rs.L = normalize(lightPosition - vtx.position);
        return rs;
    }
    float3 perFragmentComputation(PerVertexResult v)
    {
        float nDotL = max(0, dot(v.N, v.L));
        return lightIntensity * nDotL;
    }
}
```

Listing 6.4: An equivalent implementation of `MultiRatePointLight` in Slang.

```
mixin shader class ShadowedPointLight extends MultiRatePointLight
{
    input @Uniform Texture2D shadowMap;
    @Vertex float shadowCoord = calcShadowCoord(...);
    @Fragment float shadowFactor = shadowMap.Sample(shadowMap, shadowCoord);
    override @Fragment float lightingResult = lightIntensity * nDotL * shaowFactor;
}
```

Listing 6.5: An example of composing multi-rate computations in Spark. In this example, `MultiRatePointLight` is extended with multi-rate shadow computation. Shadows are computed by first computing the coordinate for sampling the shadow map in thevertex shader, and sampling the shadow map in the fragment shader. Computation for `lightResult` is overridden to combine with the shadow factor. Implementing the same component in Slang would require the virtual classes feature.

struct returned by the vertex shader entry point.

The Slang equivalent implementation of `MultiRatePointLight` is given in Listing 6.4. Instead of qualifying a computation with rates, the Slang implementation partitions the computation into different methods by pipeline stages, which involves more boiler-plate code than the Spark equivalent.

Although the implementation in Listing 6.4 captures the essence of multi-rate programming, a full mapping of the multi-rate programing model, particularly the composition of multi-rate components as in Spark, would require the more-advanced virtual classes feature as discussed in Section 6.3.1 [31]. Consider the example in Listing 6.5, which extends `MultiRatePointLight` with shadows. This example computes shadow in two steps: first, the coordinate to sample the shadow map (`shadowCoord`) is computed at the vertex shading stage. Next, this coordinate is used to sample the shadow map at the fragment shading stage. The resulting shadow factor is modulated with the lighting result by overriding the computation of `lightingResult`.

The closest map of the `ShadowedPointLight` component in Slang's programming model is to represent it as a `struct` that inherits from `MultiRatePointLight`. However, since the shadow algorithm computes a new term (`shadowCoord`) in the vertex shader and passes it to the fragment shader, the term must be included in the associated `PerVertexResult` type (as in Listing 6.4). Therefore, `ShadowedPointLight` type must define a new `PerVertexResult` type that overrides `MultiRatePointLight`'s `PerVertexResult`. Since we use inheritance, any existing functions accepting `MultiRatePointLight` must also accept `ShadowedPointLight`, which defines a different `PerVertexResult` type. To support this scenario, the compiler needs to be extended to support virtual classes.

Chapter 7

Discussion

We have presented the Slang compilation system and described how its language mechanisms and runtime services can be used to adapt an existing shading system to follow the shader components design pattern for improved extensibility and performance. In this chapter, we summarize the key design decisions we made in Slang, and lay out future work for the Slang project.

7.1 Key Design Decisions

The most important insight of this thesis is that performance and extensibility requirements can be met simultaneously when the shading system is architected in a way that aligns modularity boundaries with the granularities in which key performance operations (including shader code specialization, shader variant lookup and shader parameter communication) are performed. We studied the design rules that are key to achieving this alignment and motivates the shader compilation services required to implement this design.

The first key decision we made is to introduce a small set of general-purpose language constructs that are widely supported by modern programming languages (e.g., Swift, Rust and C$^{\#}$) as an extension to HLSL to support implementation of shader components design. The use of general language mechanisms such as generics and interfaces instead of ad-hoc code generation techniques (e.g., custom code generation tools and preprocessor hacks) enables cleaner code structure and allows the compiler to perform early type checking and to provide better error messages. These language constructs also afford the possibility of many other benefits, such as faster compilation of shader code and the ability to abstract over static specialization and dynamic dispatch, as we will discuss in the next section. Compared to shader compilation systems

that involve domain specific language constructs as in Spire [38] and Bungie's TFX shading system [91], our design maintains compatibility with HLSL, which is critical to Slang's successful adoption in existing code bases such as Falcor. Besides, the choice of general language mechanisms ensures Slang's familiarity to developers, which should reduce the amount of learning effort required to make use of these more advanced language features.

Another important decision was to clearly separate the language mechanisms from a shading system's runtime policies. Slang's runtime API is designed to provide services associated with the language mechanisms without dictating how a certain task should be performed. For example, Slang provides an API for the shading system to generate specialized shader kernels using specific types, but leaves the decision of *when* to specialize shader kernels to the shading system — it can either generate specialized kernels on-demand when rendering a scene, or up-front in an offline preprocessing step. Details regarding to how specialized shader kernels are stored and cached are also responsibilities of the shading system, not Slang. As another example, as discussed in Section 4.3.2, a shading system can choose to use the reflection API to obtain the parameter layout required to fill in parameter blocks, or to enforce a standard parameter layout rule in shader code so that the host code can infer the layout without using the reflection API. Although we prefer to use the reflection API for parameter communication, Slang allows both shading system implementations. This design decision is intended to allow Slang to be a generally useful shader compilation service for many shading systems with different goals and constraints, instead of being a dedicated tool for one specific shading system.

7.2 Limitations and Improvements

In this section, we discuss several aspects in which Slang can be improved in the short term. These ideas either originated from the issues arising from our experiences of using Slang, or were inspired by the latest advances in hardware architectures for real-time ray-tracing.

7.2.1 De-specialize Code for Dynamic Polymorphism

Slang's implementation of `interface` assumes static polymorphism, where the compiler must be able to infer what code to run when generating kernel code for a specialized shader variant. In most of the scenarios, this is the desired behavior, and the resulting shader variant should include only the code that is needed for a given set of shading features.

However, under certain circumstances a developer may want the compiler to generate dy-

```
struct GBufferElement
{
    int bxdfType;
    float4 data[GBUFFER_PIXEL_SIZE];
};
interface IMaterial
{
    associatedtype BxDF : IBxDF;
    BxDF evalPattern(SurfaceGeometry geom);
    GBufferElement packGBuffer(BxDF bxdf);
    BxDF unpackGBuffer(GBufferElement elem);
}
```

Listing 7.1: The extended `IMaterial` definition for deferred rendering. This new definition adds two method requirements: `packGBuffer`, which packs the parameters of a BxDF closure into a data structure that can be stored in the G-buffer; and `unpackGBuffer`, which extracts all the parameters from a G-buffer element.

namic dispatch code. For example, using de-specialized code reduces the compilation time since it results in fewer shader variants, which is helpful in debugging or experimenting with the visual effects of a new shader logic.

Another detailed circumstance where de-specialization is useful is the light integration phase of a deferred renderer. Recall that a deferred renderer produces images in two phases. The first phase renders a G-buffer image in which each pixel stores the surface pattern (including both the surface properties and a BxDF associated with the material) of the object overlapping that pixel. The second phase launches a single GPU kernel that reads this surface pattern and integrates lighting contributions using the BxDF of the material for all pixels in the G-buffer image. Note that different objects in the scene may use different materials (and thus different BxDFs). Therefore, the number and types surface property values stored in the G-buffer image, as well as the BxDF to use in lighting integration, can be different for each pixel.

This is currently implemented in Falcor by extending the `IMaterial` type with `packGBuffer` and `unpackGBuffer` methods, as shown in Listing 7.1. The `GBufferElement` type represents a single pixel in the G-buffer image, which contains a `type` field to indicate the type of BxDF associated with the pixel, and a `data` field for storing the surface property values of an evaluated surface pattern (BxDF closure). The `packGBuffer` method of each method is then used to "serialize" a BxDF closure into a `GBufferElement`, and `unpackGBuffer` is used to "deserialize" the BxDF from a `GBufferElement`.

```
...
type_param TMaterial : IMaterial;
ParameterBlock<TMaterial> gMaterial;
GBufferElement patternGenPass(SurfaceGeometry geom)
{
    ...
    TMaterial.BxDF bxdf = gMaterial.evalPattern(geom);
    return gMaterial.packGBuffer(geom);
}
```

Listing 7.2: The pattern generation phase shader entry point for deferred rendering.

Listing 7.2 shows the shader entry point for the surface pattern generation phase. The entry point calls `evalPattern` method to evaluate the surface pattern, then calls `packGBuffer` method to pack the resulting surface pattern into a `GBufferElement`, which is then returned as an output pixel in the G-buffer image.

The challenge is in implementing the shader entry point for the lighting integration phase. Since the same shader kernel is used to compute lighting for all pixels in the G-buffer, the shading system cannot specialize the shader entry point to any specific material implementation. Instead, the shader code must use the `bxdfType` field of a `GBufferElement` and dynamically dispatch to the actual BxDF. With current Slang, this is implemented by manually creating a new `DynamicMaterial` type that dispatches to concrete material types, as shown in Listing 7.3. The `DynamicMaterial` type wraps instances of all existing material implementations, such as `LambertMaterial` and `DisneyMaterial`, as its members. Similarly, the associated `BxDF` type wraps the associated `BxDF` types of `LambertMaterial` and `DisneyMaterial`. The implementation of `BxDF.eval` and `unpackGBuffer` simply dispatches to appropriate implementations based on `bxdfType`. With `DynamicMaterial` type, the shader entry point for lighting integration phase can be authored as in Listing 7.4.

Note that the implementation of `DynamicMaterial` type follows a mechanical process: we first define a `typeId` field to indicate the actual type represented by this dynamic type. Then, for each existing type that conforms to `IMaterial`, we add a field of that existing type to `DynamicMaterial`, and finally implement each method by dispatching execution to the one of the existing types based on `typeId`. Instead of requiring a developer to implement this manually, a compiler has all the information it needs to generate such an implementation. For example, Slang can potentially support the following syntax to produce a dynamic implementation that dispatches execution to all existing types that conforms to `IMaterial`:

```
struct DynamicMaterial : IMaterial
{
    int typeId;
    LambertMaterial lambertMaterial;
    DisneyMaterial disneyMaterial;
    ...
    struct BxDF : IBxDF
    {
        int bxdfType;
        LambertMaterial.BxDF lambertBxDF;
        DisneyMaterial.BxDF disneyBxDF;
        ...
        float3 eval(float3 wi, float3 wo)
        {
            switch (bxdfType)
            {
            case BXDF_TYPE_LAMBERT:
                return lambertBxDF.eval(wi, wo);
            case BXDF_TYPE_DISNEY:
                return disneyBxDF.eval(wi, wo);
            ...
            }
        }
    };
    ...
    BxDF unpackGBuffer(GBufferElement elem)
    {
        BxDF bxdf;
        bxdf.bxdfType = typeId;
        switch (typeId)
        {
        case BXDF_TYPE_LAMBERT:
            bxdf.lambertBxDF = lambertMaterial.unpackGBuffer(elem);
            break;
        case BXDF_TYPE_DISNEY:
            bxdf.disneyBxDF = disneyMaterial.unpackGBuffer(elem);
            break;
        ...
        }
        return bxdf;
    }
}
```

Listing 7.3: To implement the light integration phase, a developer has to manually create a `DynamicMaterial` type that dispatches to concrete material implementations based on `bxdfType`.

```
type_param TLightEnv : ILightEnv;
ParameterBlock<TLightEnv> lightEnv;
float3 deferredLightingPass(GBufferElement elem)
{
    DynamicMaterial mat;
    mat.typeId = elem.bxdfType;
    DynamicMaterial.BxDF bxdf = mat.unpackGBufferElem(elem);
    return integrateLighting(lightEnv, bxdf);
}
```

Listing 7.4: The shader entry point for light integration phase, which uses `DynamicMaterial` to dynamically dispatch to different BxDFs based on the `bxdfType` field of the G-buffer pixel.

```
LightData gLights[N];
for (int i = 0; i < N; i++)
{
  if (gLights[i].lightType == PointLight)
      <Handling Logic For PointLight>
  else if (gLights[i].lightType == DirLight)
      <Handling Logic For DirectionalLight>
}
```

Listing 7.5: Accessing a light collection represented by a single array is straightforward: the developer writes a `for` statement to iterate through the array and dispatch to different logic based on the type of each light.

```
dynamic DynamicMaterial : IMaterial;
```

The developer can then use `DynamicMaterial` in places where an `IMaterial`-conformed type is expected to generate *de-specialized* shader code.

7.2.2 Simplify the Syntax for Static Polymorphism

As mentioned in Section 5.5, our implementation of a specialized lighting environment makes it harder for developers to access lights inside the composite light collection. When lighting environment is represented as a single array of lights, a developer can use a `for` loop to iterate through the lights in the array, as illustrated by Listing 7.5.

In contrast, working with the composite light type requires more development effort. As shown in Listing 7.6, to implement a new type of operation on lights, the developer must extend the `ILight` interface and implement the operation(`myOp`) on each type of light. Next, the developer

```
interface IMyOp
{
    float myOp();
}
// extend ILight to implement the new operation on each type of light
extension ILight : IMyOp;
extension PointLight : IMyOp
{
    float myOp()
    {
        <Handling Logic For PointLight>
    }
}
extension DirectionalLight : IMyOp
{
    float myOp()
    {
        <Handling Logic For DirectionalLight>
    }
}
// extend ILightEnv to dispatch to the new operation
extension ILightEnv : IMyOp;
extension LightArray<T:ILight, let N:int> : IMyOp
{
    float myOp()
    {
        for (int i = 0; i < N; i++)
          lights[i].myOp();
    }
}
extension LightPair<T1:ILightEnv, T2:ILightEnv> : IMyOp
{
    float myOp()
    {
        ...
    }
}
extension LightSingleton<TILight:ILight> : IMyOp
{
    ...
}
```

Listing 7.6: Accessing light collection represented by a composite type is complicated: the developer needs to extend the `ILight` interface and implement the new handling logic in each light type, and extend the `ILightEnv` interface to inductively dispatch the calls to the new handling logic.

```
ILightEnv gLights;
gLights.foreach((ILight l) =>
{
   switch (l)
   {
   case (PointLight pl):
      <Handling Logic For PointLight>
   case (DirectionalLight dl):
      <Handling Logic For DirectionalLight>
   }
});
```

Listing 7.7: Accessing composite light collection type can be simplified with advanced language features: a developer calls the `foreach` operation with a lambda expression that implements the handling logic in a pattern matching clause.

must extend the `ILightEnv` interface with the same operation, and implement it in `LightArray`, `LightPair` and `LightSingleton` by induction, which simply dispatches the calls to each light's `myOp` method. With all extensions in place, the use-site can invoke the new operation by calling `TLightEnv.myOp` method.

This workflow can be greatly simplified with interface-typed values, higher order functions (lambda expressions), and pattern matching. With these language features, we can define a `foreach` operation in `ILightEnv`:

```
interface ILightEnv { void foreach(Func<ILight, void> handler); }
```

With this representation, the same operation implemented in Listing 7.5 can be written as a call to `foreach` operation, as shown in Listing 7.7. The lambda function takes an interface-typed parameter `l`, and dispatch to different handling logic with a pattern matching clause. The mental complexity of writing code in Listing 7.7 is mostly on par with Listing 7.5. The difference is that the compiler can generate specialized code without any dynamic dispatching from Listing 7.7.

7.2.3 Support Generic Entry Point Types, Not Global Generic Parameters

Recall that in shader components design pattern, a shader entry point is a generic function that can be specialized with concrete component types. In most HLSL code, a shader parameter is not defined as a parameter of the entry point function, but as a global variable. To maintain consistency with HLSL, Slang supports global generic type parameters to allow defining generic typed shader parameters as global variables, as discussed in Section 4.2.6. In practice, global

```
type_param T;
struct D
{
    float doSomething()
    {
        T t; ...
    }
};
```

Listing 7.8: A type that is implicitly dependent on global generic type parameter. Attempting to use `D` as argument for type parameter `T` will result in an error that could be confusing to developers who are not aware of the implementation of `doSomething`.

generic type parameters have created many problems and should be avoided if possible.

In Slang, the clause `type_param T;` defines a global type symbol `T` that can be used in any place where a type is expected. As such, a programmer can define a global generic shader parameter that has type `T` or `ParameterBlock<T>` to implement the shader components pattern. When compiling an entry point into final target code, the host application must provide to Slang the actual type for `T` to properly specialize shader code. Since the definition `T` is treated as a normal type symbol, from the type system perspective it is legal to use `T` itself as the actual type argument for the generic type parameter `T`. However, doing so will result in illegal code being generated.

While it appears that using `T` as type argument for type parameter `T` is an obvious error, more confusing scenarios arise when the programmers define types that implicitly depend on `T`. For example, Listing 7.8 defines a valid type in Slang.

However, attempting to use `D` as type argument for `T` will also result in illegal code. This creates a confusing scenario where the developer must look into the definition of `D` to understand the compiler error.

A more appropriate way to represent generic shader entry points is to wrap the entry point function in a generic `struct` type. Listing 7.9 shows an example entry point type that wraps an entry-point function for the fragment shading stage. Instead of using global generic type parameters, the generic type parameters of the entry-point function are defined on the generic `MyEntryPoint` type. Confusing circular dependency scenarios are avoided since it is no longer possible to construct a type that is implicitly dependent on the generic type parameters.

Note that `IFragmentEntryPoint` will be a compiler-defined interface. A conformance decla-

```
struct MyEntryPoint<M:IMaterial, L:ILightEnv> : IFragmentEntryPoint
{
    ParameterBlock<M> material;
    ParameterBlock<L> lightEnv;
    struct FragmentResult
    {
        float3 color;
    };
    FragmentResult runFragment()
    {
        ...
    }
};
```

Listing 7.9: An entry point function for the fragment shading stage wrapped in a generic `struct` type. The fields of `MyEntryPoint` represent shader parameters.

```
interface IFragmentEntryPoint
{
    associatedtype FragmentResult;
    FragmentResult runFragment();
};
```

Listing 7.10: The compiler-defined `IFragmentEntryPoint` interface for signaling a type as an entry point type. It uses associated types to represent the return value of the entry point function.

ration to this interface signals that the corresponding type represents an entry point. This will instruct the compiler to interpret the fields of `MyEntryPoint` as the parameters for the shader entry point.

Listing 7.10 shows the definition for the `IFragmentEntryPoint` interface. It makes use of associated types to allow different fragment shader entry points to return different types of values.

Once we have represented entry points as types, we can easily extend this representation to define complete shader programs consisting of multiple entry points. For example, we can use a single type to represent a shader containing both the vertex shader kernel and the fragment shader kernel. Such a type can be bounded by an `IShaderProgram` interface in Listing 7.11. This is a particularly useful representation for the *hit group* concept introduced in the DirectX Ray Tracing API. A hit group defines different entry point shader functions that will be executed at different scenarios, such as at the exact ray intersection point (closest hit), or when a ray misses (miss shader) or hits any objects in the scene (any hit). To represent a hit group, we can use a

```
interface IShaderProgram
{
    associatedtype VertexResult;
    associatedtype FragmentResult;

    VertexResult runVertex(VertexInput vert_in);
    FragmentResult runFragment(VertexResult vrs);
};
```

Listing 7.11: The `IShaderProgram` interface can be used to bound a type that represents a combination of vertex and fragment entry points that together form a complete shader program. This interface definition makes use of associated types to represent the return types of the vertex and fragment shading stages.

single type to encapsulate all the required entry points.

7.2.4 Support Shared Shader Parameters

In most cases, shader parameters are associated with a component instance. For example, different `PointLight` instances are expected to have different `position` parameter values. In Slang, these per-instance parameters are represented as fields of `struct` types.

However, some shading features require parameters that are `shared` among all instances of the feature. One example is the polygon area lights, whose implementation requires access to a precomputed lookup table communicated to the GPU as a texture. Since the content of this lookup texture is independent of area light instances, encapsulating the texture parameter as a field of the `AreaLight` type and communicating the same texture handle for each area light instance is wasteful. In Slang, these shared parameters must be defined as global variables and communicated separately. Use of global variables leads to less maintainable code and imposes additional responsibility on the host code to correctly communicate their parameter values.

Slang has the opportunity to provide more-principled support of shared parameters with special layout rules and additional reflection API. Specifically, the developer should be able to define all types of parameters as fields of a `struct` type, and mark those shared parameters as `static` so that they are placed in a separate parameter block dedicated for all shared parameters. The layout of the dedicated shared parameter block is specific to a specialized shader variant. By querying its layout, the host application can determine what features are in use and set their shared parameters accordingly. Since shared parameters typically represent “static” values (e.g., lookup

tables) that do not frequently change, the application can create and fill in the shared parameter block upon initialization of a specialized shader variant.

7.2.5 Improve Shader Compilation Performance

Many modern video games generate a large number of specialized shader variants. These shader variants are typically compiled in an offline process, which takes up to hours of time. Speeding up shader compilation is valuable as it will help reduce the turn-around time during shader development and improve productivity.

In our case study, adopting Slangto Falcor 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, one second is spent in Slang, while 4.5 seconds are spent in HLSL compilation. This is slightly slower than the original (light specialization) branch, which takes 4.1 seconds to compile all the shader variants.

Slang's language features are carefully chosen to support separate compilation of individual modules and allow amortizing across multiple entry points and variants (as discussed in Section 4.3.3), which reduces the time required to compile many shader variants.

However, since the majority of shader compilation time (approximately 80%) is currently in the down-stream HLSL compiler, further optimizing Slang will not result in significant speed-up in overall compilation time. As a result, it is important to investigate alternative down-stream code generation techniques that allow us to eliminate the overhead of outputting HLSL and invoking a second compiler, such as by directly translating Slang's IR to formats like SPIR-V and DXIL [65].

7.3 Future Directions

Besides the short-term improvements that can make Slang a better shading language, our research has inspired us to tackle additional challenges in the context of programming languages for graphics applications.

7.3.1 Unified Language for Host and Shader Logic

Slang represents a first step towards the ultimate goal to allow both host side and shader side logic to be authored in a unified language, which would allow shading systems to be written similarly to how heterogeneous applications are written in CUDA [71].

A unified programming model has many benefits for development productivity. First, it eliminates the mental overhead for developers to switch languages when programming the shading system. Second, it preserves type information in both host-side and shader code, and eliminates the need to use reflection API to ensure layout consistency between CPU and GPU. A variable of a component type on the CPU can be used to invoke a shader entry point directly, since the compiler will be responsible for enforcing the same layout rule for both CPU code and GPU code. Finally, it allows the compiler to perform global optimizations across different invocations of the graphics pipeline and shader kernels to achieve higher performance not possible with the existing programming model.

The challenge is that the GPU state for the graphics pipeline is significantly more complicated than the context for general GPU compute tasks. In particular, the shading system is responsible for carefully managing the use of GPU memory (e.g., when to load and unload a texture image to GPU memory). In many shading systems, memory management critically impacts runtime performance. Besides, the execution of rendering commands is typically asynchronous: in most cases the CPU composes a list of commands, sends it to the GPU for execution, and never waits for any return values from the GPU. The programming model for expressing and coordinating asynchronous GPU command execution and complex memory management operations still remains to be studied.

7.3.2 Programming Model for Ray Tracing Hardware Architectures

To implement the recently proposed hardware ray-tracing API [84] efficiently, the next generation GPU architectures might add support for making recursive function calls and a new form of hardware accelerated dynamic dispatch mechanism similar in spirit to virtual functions in modern object-oriented programming languages. Future GPUs will likely have the ability to dynamically generate, schedule and execute heterogeneous tasks on the fly during a single draw command, opening up new possibilities to utilize a GPU's parallelism. An interesting research direction is to study how Slang's programming model can be extended to support advanced ray tracing workloads and to efficiently utilize the next-generation GPU architecture.

On a rasterization graphics pipeline, it is almost always preferable to use specialized shader code over dynamic dispatch. However, this decision is more complicated in a ray tracing renderer. Although specialized shader kernels execute faster, using different shaders for different objects reduces the GPU's capability to group rays in batches and process them in parallel: the rays that hit different objects using different shader code cannot be processed simultaneously by a GPU execution unit. For optimized performance, a developer may want to explore different choices of static specialization or dynamic dispatch. Using a programming language that allows the developer to write shader code once and generate either static specialization or dynamic dispatch implementations could be valuable. Slang is designed to support this future scenario.

Chapter 8

Conclusion

While modular shader development and high rendering performance might appear to be at odds, this is not actually the case. We formalized a set of shading system design rules into a design pattern called shader components. We demonstrated that a popular real-time shading language can be extended with carefully chosen mechanisms from modern general-purpose languages to support implementing shader components, without the need for layered preprocessor and DSL tools.

By refactoring the Falcor shading system to adopt these mechanisms, we were able to achieve improvements in CPU and GPU performance (by exploiting the shader components pattern) while making the framework more aligned with developer's mental model of the graphics concepts and easier to extend with a new BxDF-dependent light type.

The most important lesson we learned from the development of Slang is that a real-time shading language needs to be designed from the perspective of a shading system's task. In contrast to existing shading languages such as HLSL and GLSL, we derived Slang's language and runtime API design from the requirements for implementing a modular, extensible and high performance shading system (the shader components pattern). This eliminates the need for shading systems to implement various hacks and work-arounds regarding shader compilation.

Slang's design demonstrates that it is possible to maintain performance and compatibility with existing HLSL code bases, while moving in the direction of modern general-purpose languages with strong support for good software development practices. We hope that all real-time graphics programmers will benefit from a new generation of shader compilation tools informed by ideas in this thesis.

The development of Slang has led us to many exciting future directions regarding programming models for graphics applications. We are particularly interested in extending Slang to support rapid exploration of choices between dynamic dispatch and static specialization; to incorporate the more advanced task-graph execution model in up-coming ray-tracing pipelines; and to become a unified programming language spanning both CPU and GPU.

Bibliography

[1] The AspectJ Project. http://www.eclipse.org/aspectj/, 2018. 6.4

[2] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering, Third Edition*. CRC Press, 2008. ISBN 9781439865293. URL https://books.google.com/books?id=V1k1V9Ra1FoC. 5.2.1

[3] Amazon. Lumberyard engine. https://aws.amazon.com/lumberyard/, 2016. 2.4.2, 2.4.2, 5.3.1, 6.1

[4] Solid Angle. Arnold renderer. https://www.solidangle.com/arnold/, 2018. 1, 2.1

[5] Thomas Annen, Tom Mertens, Hans-Peter Seidel, Eddy Flerackers, and Jan Kautz. Exponential shadow maps. In *Proceedings of Graphics Interface 2008*, pages 155–161, Toronto, Ont., Canada, 2008. Canadian Information Processing Society. ISBN 978-1-56881-423-0. URL http://dl.acm.org/citation.cfm?id=1375714.1375741. 5.2.1

[6] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999. ISBN 1558606181. 1, 2.1

[7] Apple Inc. *Metal Documentation*. 2014. URL https://developer.apple.com/documentation/metal. 6.2.1

[8] Apple Inc. Understanding argument buffers. In *Metal Documentation*. 2018. URL https://developer.apple.com/documentation/metal/fundamental_components/gpu_resources/understanding_argument_buffers. 4.2.5

[9] Apple Inc. *The Swift Programming Language*. 2018. URL https://docs.swift.org/swift-book/. 2, 4.1, 4.2.1

[10] Apple Inc. Generics. In *The Swift Programming Language Guide*. 2018. URL https://docs.swift.org/swift-book/LanguageGuide/Generics.html. 4.2.3, 6.3.1

[11] Apple Inc. Extensions. In *The Swift Programming Language Guide*. 2018. URL https://docs.swift.org/swift-book/LanguageGuide/Extensions.html. 4.2.4

[12] Chad Austin and Dirk Reiners. Renaissance : A functional shading language. Master's thesis, Iowa State University, 2005. 6.2.1

[13] David Baraff and Andrew Witkin. Large steps in cloth simulation. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, pages 43–54, New York, NY, USA, 1998. ACM. ISBN 0-89791-999-8. doi: 10.1145/280814.280821. URL http://doi.acm.org/10.1145/280814.280821. 2.1.1

[14] Nir Benty, Kai-Hwa Yao, Tim Foley, Anton S. Kaplanyan, Conor Lavelle, Chris Wyman, and Ashwin Vijay. The Falcor rendering framework, July 2017. URL https://github.com/NVIDIAGameWorks/Falcor. https://github.com/NVIDIAGameWorks/Falcor. 1, 5, 5

[15] Tobias Bexelius. Gpipe: Typesafe functional GPU graphics programming. http://hackage.haskell.org/package/GPipe, 2017. 6.2.1

[16] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012. URL http://arxiv.org/abs/1209.5145. 6.3.2

[17] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977. ISSN 0097-8930. doi: 10.1145/965141.563893. URL http://doi.acm.org/10.1145/965141.563893. 2.1.2

[18] David Blythe. The Direct3D 10 system. *ACM Transactions on Graphics*, 25(3):724–734, July 2006. ISSN 0730-0301. doi: 10.1145/1141911.1141947. 2.4.6

[19] Jeff Bolz and Pat Brown. Bindless texture OpenGL extension. https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_bindless_texture.txt, 2013. 6.2.1

[20] Bungie. Destiny computer game. http://www.destinythegame.com, 2014. 2.1.2

[21] Brent Burley. Physically-based shading at disney. https://disney-animation.s3.amazonaws.com/library/s2012_pbs_disney_brdf_notes_v2.pdf, 2012. 2.1.2, 2.1.2

[22] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–13, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040306. URL http://doi.acm.org/10.1145/1040305.1040306. 4.2.3

[23] William R. Cook. Object-oriented programming versus abstract data types. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 151–178, London, UK, UK, 1991. Springer-Verlag. ISBN 3-540-53931-X. URL http://dl.acm.org/citation.cfm?id=648142.749835. 6.3.2

[24] William Donnelly and Andrew Lauritzen. Variance shadow maps. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, pages 161–165, New York, NY, USA, 2006. ACM. ISBN 1-59593-295-X. doi: 10.1145/1111411.1111440. URL http:

//doi.acm.org/10.1145/1111411.1111440. 5.2.1

[25] ECMA International. C# language specification (ECMA-334:2017), December 2017. 2, 4.1, 4.2.1

[26] Conal Elliott. Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop*. ACM Press, 2004. URL http://conal.net/papers/Vertigo/. 6.2.1

[27] Woflgang F. Engel. Cascaded shadow maps. In Woflgang F. Engel, editor, *ShaderX5 - Advanced Rendering Techniques*, chapter 4, pages 197–206. Boston, Massachusetts, 2006. 5.2.1, 5.2.1

[28] Epic Games. Unreal Engine 4 Documentation. http://docs.unrealengine.com, 2015. 1, 2.4.1, 6.1

[29] Epic Games. Unreal Engine 4.10 source code. https://www.unrealengine.com/, 2017. 1

[30] Kayvon Fatahalian and Mike Houston. GPUs: A closer look. *Queue*, 6(2):18–28, March 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365498. URL http://doi.acm.org/10.1145/1365490.1365498. 2.3.1

[31] Tim Foley and Pat Hanrahan. Spark: Modular, composable shaders for graphics hardware. *ACM Transactions on Graphics*, 30(4):107:1–107:12, July 2011. 6.2.1, 6.5, 6.5

[32] Timothy Foley. *Spark: modular, composable shaders for graphics hardware*. dissertation, Stanford University, 2012. URL http://http://graphics.stanford.edu/~tfoley/papers/tfoley-dissertation.pdf. 6.4

[33] Google. Interface types, the Go language specification, 2018. URL https://golang.org/ref/spec#Interface_types. 4.2.2

[34] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. The Java language specification, Java SE 10 edition. https://docs.oracle.com/javase/specs/jls/se10/jls10.pdf, 2018. 4.2.1

[35] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. *SIGGRAPH Comput. Graph.*, 24(4):289–298, September 1990. 6.2.1

[36] Takahiro Harada, Jay McKee, and Jason C. Yang. Forward+: Bringing Deferred Lighting to the Next Level. In Carlos Andujar and Enrico Puppo, editors, *Eurographics 2012 - Short Papers*. The Eurographics Association, 2012. doi: 10.2312/conf/EG2012/short/005-008. 5.4

[37] Yong He, Tim Foley, Natalya Tatarchuk, and Kayvon Fatahalian. A system for rapid, automatic shader level-of-detail. *ACM Transactions on Graphics*, 34(6):187:1–187:12, October 2015. ISSN 0730-0301. doi: 10.1145/2816795.2818104. URL http://doi.acm.org/10.1145/2816795.2818104. 6.2.3

[38] Yong He, Tim Foley, and Kayvon Fatahalian. A system for rapid exploration of shader optimization choices. *ACM Transactions on Graphics*, 35(4):112:1–112:12, July 2016. 4, 6.2.3, 6.5, 7.1

[39] Yong He, Tim Foley, Teguh Hofstee, Haomin Long, and Kayvon Fatahalian. Shader components: Modular and high performance shader development. *ACM Transactions on Graphics.*, 36(4):100:1–100:11, July 2017. ISSN 0730-0301. doi: 10.1145/3072959.3073648. URL http://doi.acm.org/10.1145/3072959.3073648. 1

[40] Eric Heitz, Jonathan Dupuy, Stephen Hill, and David Neubelt. Real-time polygonal-light shading with linearly transformed cosines. *ACM Transactions on Graphics*, 35(4):41:1–41:8, July 2016. ISSN 0730-0301. doi: 10.1145/2897824.2925895. URL http://doi.acm.org/10.1145/2897824.2925895. 2.4.5, 5.2.2, 5.2.2

[41] Damien Hinsinger, Fabrice Neyret, and Marie-Paule Cani. Interactive animation of ocean waves. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 161–166, New York, NY, USA, 2002. ACM. ISBN 1-58113-573-4. doi: 10.1145/545261.545288. URL http://doi.acm.org/10.1145/545261.545288. 2.1.1

[42] Sony Pictures Imageworks. Open shading language 1.9 language specification. https://github.com/imageworks/OpenShadingLanguage/blob/master/src/doc/osl-languagespec.pdf, 2017. 2.4.3

[43] Apple Inc. LOD with function specialization. From Metal documentation, https://developer.apple.com/documentation/metal/advanced_techniques/lod_with_function_specialization, 2018. 6.2.2

[44] Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 347–349, New York, NY, USA, 1986. ACM. ISBN 0-89791-204-7. doi: 10.1145/28697.28732. URL http://doi.acm.org/10.1145/28697.28732. 6.3.2

[45] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, pages 143–150, New York, NY, USA, 1986. ACM. ISBN 0-89791-196-2. doi: 10.1145/15922.15902. URL http://doi.acm.org/10.1145/15922.15902. 2.1.2

[46] John Kessenich, Dave Baldwin, and Randi Rost. *The OpenGL© Shading Language (Version 4.50)*, 2014. https://www.opengl.org/registry/doc/GLSLangSpec.4.50.pdf. 1, 2.2.1, 6.2.1

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

SPIRV.pdf. 4.3.3

[48] Khronos Group, Inc. ARB_shader_subroutine. https://www.opengl.org/registry/specs/ARB/shader_subroutine.txt, 2009. 6.2.1

[49] Khronos Group, Inc. *Vulkan 1.0.38 Specification*, 2016. 2.2, 2.2.3

[50] Khronos Group, Inc. Specialization Constants. In *Vulkan 1.0.82 Specification*. 2018. URL https://www.khronos.org/registry/vulkan/specs/1.0/html/vkspec.html#pipelines-specialization-constants. 6.2.2

[51] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 220–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg. ISBN 978-3-540-69127-3. 6.4

[52] Christoph Kubisch. Vulkan shader resource binding. https://developer.nvidia.com/vulkan-shader-resource-binding, 2016. 2.4.6

[53] Samuli Laine, Tero Karras, and Timo Aila. Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 137–143, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2135-8. doi: 10.1145/2492045.2492060. URL http://doi.acm.org/10.1145/2492045.2492060. 2.3.1

[54] Andrew Lauritzen. Deferred rendering for current and future rendering pipelines. In *SIGGRAPH 2010 Course: Beyond Programmable Shading*, 2010. 5.4

[55] J. P. Lewis, Matt Cordner, and Nickson Fong. Pose space deformation: A unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pages 165–172, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. ISBN 1-58113-208-5. doi: 10.1145/344779.344862. URL http://dx.doi.org/10.1145/344779.344862. 2.1.1

[56] O. L. Madsen and B. Moller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. *SIGPLAN Not.*, 24(10):397–406, September 1989. ISSN 0362-1340. doi: 10.1145/74878.74919. URL http://doi.acm.org/10.1145/74878.74919. 6.3.1, 6.5

[57] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *ACM Transactions on Graphics*, 22 (3):896–907, July 2003. 6.2.1

[58] Michael D. McCool and Stefanus Du Toit. *Metaprogramming GPUs with Sh.* A K Peters, 2004. ISBN 978-1-56881-229-8. 6.2.1

[59] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hard-*

ware, pages 57–68, September 2002. ISBN 1-58113-580-7. URL http://dl.acm.org/citation.cfm?id=569046.569055. 6.2.1

[60] John McDonald. High performance Vulkan: Lessons learned from Source 2. In *GPU Technology Conference 2016 (GTC)*, 2016. http://on-demand.gputechconf.com/gtc/2016/events/vulkanday/High_Performance_Vulkan.pdf. 2.4.6

[61] Microsoft. Interfaces and classes. https://msdn.microsoft.com/en-us/library/windows/desktop/ff471421.aspx, 2011. 4.3.5, 6.2.1

[62] Microsoft. Extension methods. In *The C# Programming Language Guide*. 2015. URL https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods. 4.2.4

[63] Microsoft. HLSL shader model 5 documentation. https://msdn.microsoft.com, 2016. 1, 3, 2.2.1

[64] Microsoft. Direct3D 12 programming guide. https://msdn.microsoft.com/en-us/library/windows/desktop/dn899121(v=vs.85).aspx, 2017. 2.2, 2.2, 2.2.3

[65] Microsoft. DirectX intermediate language. https://github.com/Microsoft/DirectXShaderCompiler/blob/master/docs/DXIL.rst, 2017. 7.2.5

[66] Microsoft. Resource binding overview, Direct3D 12 Programming Guide. https://docs.microsoft.com/en-us/windows/desktop/direct3d12/resource-binding-flow-of-control, 2018. 2.4.6

[67] Todd D. Millstein and Craig Chambers. Modular statically typed multimethods. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 279–303, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66156-5. URL http://dl.acm.org/citation.cfm?id=646156.679834. 6.3.2

[68] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365500. URL http://doi.acm.org/10.1145/1365490.1365500. 2.3.1

[69] NVIDIA. NVIDIA Tesla P100 white paper. http://www.nvidia.com/object/pascal-architecture-whitepaper.html, 2016. 2.3.1

[70] NVIDIA. ORCA: Open Research Content Archive. http://developer.nvidia.com/orca, 2017. 5.3.1

[71] NVIDIA. CUDA C programming guide. https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2018. 7.3.1

[72] Marc Olano, Bob Kuehne, and Maryann Simmons. Automatic shader level of detail. In *Proceedings of Graphics Hardware*, pages 7–14. ACM/Eurographics, 2003. 6.2.3

[73] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David

Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics*, August 2010. 6.3.2

[74] Fabio Pellacini. User-configurable automatic shader simplification. *ACM Transactions on Graphics*, 24(3):445–452, July 2005. 6.2.3

[75] Matt Pharr. An introduction to shader interfaces. In *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004. ISBN 0321228324. 4.3.5, 5.5, 6.2.1

[76] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010. ISBN 9780123750792. 1, 2.1, 2.1.2

[77] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6): 311–317, June 1975. ISSN 0001-0782. doi: 10.1145/360825.360839. URL http://doi.acm.org/10.1145/360825.360839. 2.1.2

[78] Jeff Pool, Anselmo Lastra, and Montek Singh. Precision selection for energy-efficient pixel shaders. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 159–168, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0896-0. doi: 10.1145/2018323.2018349. URL http://doi.acm.org/10.1145/2018323.2018349. 6.2.3

[79] Aras Pranckevičius. Porting Unity to new APIs. In *SIGGRAPH 2015 Course Notes: An Overview of Next-generation Graphics APIs*, 2015. doi: 10.1145/2776880.2787704. http://nextgenapis.realtimerendering.com/presentations/7_Pranckevicius_Unity.pptx. 2.4.6

[80] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of SIGGRAPH 01, Annual Conference Series*, pages 159–170, New York, NY, USA, 2001. ACM. 6.2.1, 6.5

[81] John C. Reynolds. Theoretical aspects of object-oriented programming. chapter User-defined Types and Procedural Data Structures As Complementary Approaches to Data Abstraction, pages 13–23. MIT Press, Cambridge, MA, USA, 1994. ISBN 0-262-07155-X. URL http://dl.acm.org/citation.cfm?id=186677.186680. 6.3.2

[82] Rust Project Developers. *The Rust Programming Language*. 2015. https://doc.rust-lang.org/book/. 2, 4.2.1

[83] Rust Project Developers. Advanced traits. In *The Rust Programming Language*. 2018. URL https://doc.rust-lang.org/book/2018-edition/ch19-03-advanced-traits.html. 4.2.3, 4.2.4

[84] Matt Sandy. DirectX raytracing. Talk at Game Developers Conference, 2018. URL https://msdnshared.blob.core.windows.net/media/2018/03/GDC_DXR_deck.pdf. 6.3.2, 7.3.2

[85] Scala Development Team. Tour of Scala: Abstract Types. https://docs.scala-lang.org/tour/abstract-types.html, 2018. 4.2.3, 6.3.1

[86] Mathias Schott and Lars M. Bishop. High-performance low-overhead rendering with OpenGL and Vulkan. Talk at Game Developers Conference, 2016. 2.3.2

[87] Mark Segal and Kurt Akeley. The OpenGL® Graphics System: A Specification, 2016. 2.2, 2.2, 2.2.3

[88] Peter Shirley and Steve Marschner. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 3rd edition, 2009. ISBN 1568814690, 9781568814698. 2.1.2

[89] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics*, 30(6):152:1–152:12, December 2011. 6.2.3

[90] Andrew Sutton and Bjarne Stroustrup. Design of concept libraries for C++. http://www.stroustrup.com/sle2011-concepts.pdf, 2011. 6.2.1

[91] Natalya Tatarchuk and Chris Tchou. Destiny shader pipeline. Talk at Game Developers Conference, 2017. URL http://advances.realtimerendering.com/destiny/gdc_2017/index.html. 1, 4, 4.3.5, 6.1, 7.1

[92] Unity Technologies. Unity game engine. https://unity3d.com/, 2018. 6.1

[93] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75283. URL http://doi.acm.org/10.1145/75277.75283. 4.2.2

[94] Philip Wadler. The expression problem. From the Java Genericity mailing list, http://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt, 1998. 6.3.2

[95] A. J. Walker. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters*, 10(8):127–128, April 1974. ISSN 0013-5194. doi: 10.1049/el:19740097. 5.4.2

[96] Bruce Walter, Stephen R. Marschner, Hongsong Li, and Kenneth E. Torrance. Microfacet models for refraction through rough surfaces. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, pages 195–206, Aire-la-Ville, Switzerland, Switzerland, 2007. Eurographics Association. ISBN 978-3-905673-52-4. doi: 10.2312/EGWR/EGSR07/195-206. URL http://dx.doi.org/10.2312/EGWR/EGSR07/195-206. 2.1.2

[97] Rui Wang, Xianjin Yang, Yazhen Yuan, Wei Chen, Kavita Bala, and Hujun Bao. Automatic

shader simplification using surface signal approximation. *ACM Transactions on Graphics*, 33(6):226:1–226:11, November 2014. 6.2.3