Slang - A Shader Compilation System for Productive Development of Efficient Real-time Shading Systems

Yong He

June 2017

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

Keywords: shading languages, real-time rendering

Abstract

To produce compelling images, a real-time renderer is responsible for simulating many real-world visual effects. These effects range from modeling the material properties of surfaces to evaluating complex lighting conditions, to the animation of surfaces. Given the diversity of scenes in modern graphics applications such as games, Industrial real-time renderers contain hundreds of thousands of lines of code that define these *shading* effects. Like any complex software system, it is desirable for these code bases to be implemented in a flexible and extensible framework to enable productive use and development. Additionally, real-time renderers must meet extreme performance requirements, which requires using the GPU graphics pipeline efficiently for the drawing tasks. As a result, the core logic for simulating various visual appearances must be written as shader code that runs at different stages of the GPU graphics pipeline. While traditional object-oriented programming principles work well for abstracting shading system concepts into an extensible and flexible framework, their current implementations in existing programming languages are not sufficient for generating high-performance GPU code. Specifically, renderers must make different trade-offs between shader compilation time and execution efficiency by implementing dispatch of shading features differently either via dynamic control flows in shader code, or via static shader specialization. Also, renderers must minimize and efficiently manage CPU-GPU communication. To meet this challenge, this thesis contributes the design of the Slang shading language and its compilation system, which adopts object-oriented programming concepts under GPU performance constraints. We address introduce a design pattern called shader components, which can be implemented with the enhanced language mechanisms in Slang, to decouple shading logic from the specific implementation choices of GPU code dispatching or CPU-GPU communication mechanisms. We demonstrate the effectiveness the Slang shader compilation system in a reference renderer implementation that is modular, flexible, extensible and fast. We further evaluate our work by adopting Slang into an existing research rendering framework to achieve higher performance with a more maintainable code base.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern real-time graphics applications, such as video games and 3D design visualization tools, are designed to produce realistic and high resolution images of virtual scenes at over 60 Hz refresh rate. These applications need to simulate a wide range of visual effects - including light response from various types of surfaces, shadowing and reflection, 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 doing this. 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 specified camera angle. A shading system must model many real-world concepts, such as light, materials, surfaces, camera lens, and etc., and must be capable to simulate an object's visual appearance as a result of arbitrary composition of these concepts.

Modern shading systems are complex software systems. For example, Unreal Engine 4, one of the most popular game engines, contains over 70,000 lines of shader code that runs on the GPU. Similar to any complex systems, a shading system must be designed to enable productive maintenance and extension of its features - it should have a framework that allows mix and matching different shading features in the system to produce a wide variety of visual effects, and also enables graphics programmers to easily extend the system by adding more shading features to simulate new types of natural phenomena, without affecting exiting code.

Shading systems for many off-line film renderers have been authored using object-oriented principles, which provide good abstractions for modeling a modular, extensible and flexible shading system framework. However, the challenge for authoring real-time rendering systems is to make use of these modular abstractions while meeting extreme performance demands. For example, consider a typical AAA video game which needs to draw 10,000 objects per frame. To render at 60 Hz, the game needs to draw 600,000 objects per second. A main-stream CPU core runs at 3 GHz, which leaves only 5,000 CPU cycles to draw a single object.

In order to achieve this performance goal, real-time shading systems must strive to undertake the following measures:

Accelerate drawing tasks using the GPU graphics pipeline. To draw objects with complex visual appearances in such a small time budget, graphics applications must off-load 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*, which are executed at different stages of the graphics pipeline, using C-like *shading languages* such as GLSL[4] and HLSL[8]. Besides greater challenges in modularizing and maintaining a heterogeneous code-base, developers are constrained by the GPU's performance characteristics, which requires the following commitments from a developer.

GPU code must be specialized to avoid dynamic dispatch when possible. Typically, a shading system implements a large library of shading features, but only a small set of features are enabled in a single draw command to achieve the specific visual appearance for an object. Existing object-oriented languages like C++ implements 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, shading languages do not support object-oriented constructs 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. 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.

CPU-GPU communication must be efficient. Even when using the GPU graphics pipeline to draw objects, 5,000 CPU cycles is still not an ample budget to compose and transfer necessary commands to the GPU. To prevent the CPU from being a severe performance bottleneck, the shading system should transfer as small amount of data to the GPU as possible, and reduce the CPU time needed in initiating the communications and sending the draw commands to the GPU. Unfortunately, many existing shading languages do not provide sufficient support to facilitate

GPU data layouts for efficient communication without adversely affect code maintainability, forcing shading systems to make trade-offs between better code maintainability and CPU-GPU communication efficiency.

In Chapter 2, we will discuss the challenges in implementing a high performance shading system in-detail. The crux of these issues is that developers want to think and program using object-oriented mechanisms, and require the shader compiler to provide different implementations to the object-oriented mechanisms to meet the performance goals. More importantly, there is no one-size-fit-all implementation to the language mechanisms. For example, shader code specialization may not always be a viable option and some developers may still want to generate dynamic dispatch code for more flexibility and faster compilation time; different shading systems may want to pack data on the GPU differently to facilitate a different CPU-GPU communication pattern. This means that the implementation of the shading language mechanisms should not be a fundamental decision in the shading language design. Instead, the implementation choice should be made available to the shading system by the shader compiler, to allow developers to easily switch to different implementations that best suites their needs.

1.1 Goals and Contributions

The goal of this thesis is to enable shading system designs that are modular, extensible and flexible while achieving high performance at the same time. We observed that the fundamental cause of the compromises in existing shading system implementations is the lack of necessary shading language features and shader compiler services to enable a modular and high performance shading system design. This is largely due to existing shading languages such as HLSL and GLSL are designed with a focus on exposing the hardware capabilities with minimal amount of programming language constructs, instead of on providing necessary support to facilitate high level shading system's tasks such as dispatching GPU execution to different shading features and efficient CPU-GPU communication.

In response, this thesis contributes:

1. A shading system design pattern centered around the *shader components* abstraction, which governs modularization of shader code as well as organizing the CPU logic for compiling and selecting specialized shader variants, and communicating shader parameters to GPU. Shader components aim to give developers an experience of object-oriented programming when abstracting shading logic for extensibility and flexible feature composition, while providing the opportunity for the shading system to implement the object-oriented constructs via either specialization or dynamic dispatch, and pack the data required by shader

kernels for efficient CPU-GPU communication.

2. Identification of the shading language mechanisms and shader compiler services missing in existing shading languages that prevents efficient implementation of the shader components abstraction.
3. Design and implementation of the Slang shading language, which adds several key object-oriented language constructs (including classes and interfaces with associate types) to existing shading languages, backed with multiple compiler implementations ranging from static specialization to dynamic dispatch, to enable efficient implementation of the shader components abstraction.
4. Design of the Slang shader introspection API for the shading system host code to efficiently compile and select specialized shader kernel variants, and to efficiently marshal and communicate parameter data to the GPU.
5. A reference shading system implementation that uses Slang and the shader components concept to achieve high performance with a maintainable and extensible code base.
6. A thorough evaluation of the Slang shading language and shader introspection API through a case study of adopting the shader components and the Slang system in a research rendering framework.
7. A discussion of the relationship between Slangs language mechanisms and prior rate-based shading language work aimed at addressing similar issues. The thesis provides a retrospect on how the most important goals achieved in these prior systems maps to disciplined uses of more general language mechanisms in Slang.

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. Implementing the shader components design pattern requires additional shading language features and shader compiler services that are missing in existing shading languages such as GLSL and HLSL. This chapter discusses the key language mechanisms in Slang and its associating introspection API that enable implementation of shader components.

Chapter ?? demonstrates a shading system design that uses the shader components design pattern and Slang shading language to achieve both system modularity and high performance goals.

Chapter 6 provides a case study of adopting Slang and shader components design pattern in a research rendering framework, as a further evaluation of this thesis.

Chapter 7 discusses the relationship of this thesis to previous works on shading language and shader compiler design.

Chapter 2

Background

The demand for extreme performance in real-time graphics applications has lead to development of highly specialized hardware architecture dedicated to solve the problem of graphics rendering - the real-time graphics pipeline. Most of the unique challenges in designing and implementing a real-time shading system stem from the need to efficiently use the GPU implementations of the graphics pipeline, and do so without sacrificing the flexibility and extensibility of the shading system. To illuminate how these challenges arise, this chapter first gives an introduction to the graphics pipeline architecture and the software constraints for achieving high performance on the GPU, then presents a detailed explanation on why the design goals of a shading system are hard to achieve given existing tools and techniques to program against the GPU graphics pipeline.

2.1 The Graphics Pipeline

On the 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. 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 to 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. Figure 2.1 illustrates the stages of a modern GPU graphics pipeline architecture (as defined by Vulkan [6], Direct3D 12 [9], and OpenGL 4.5 [11]). This figure omits the Input Assembly and

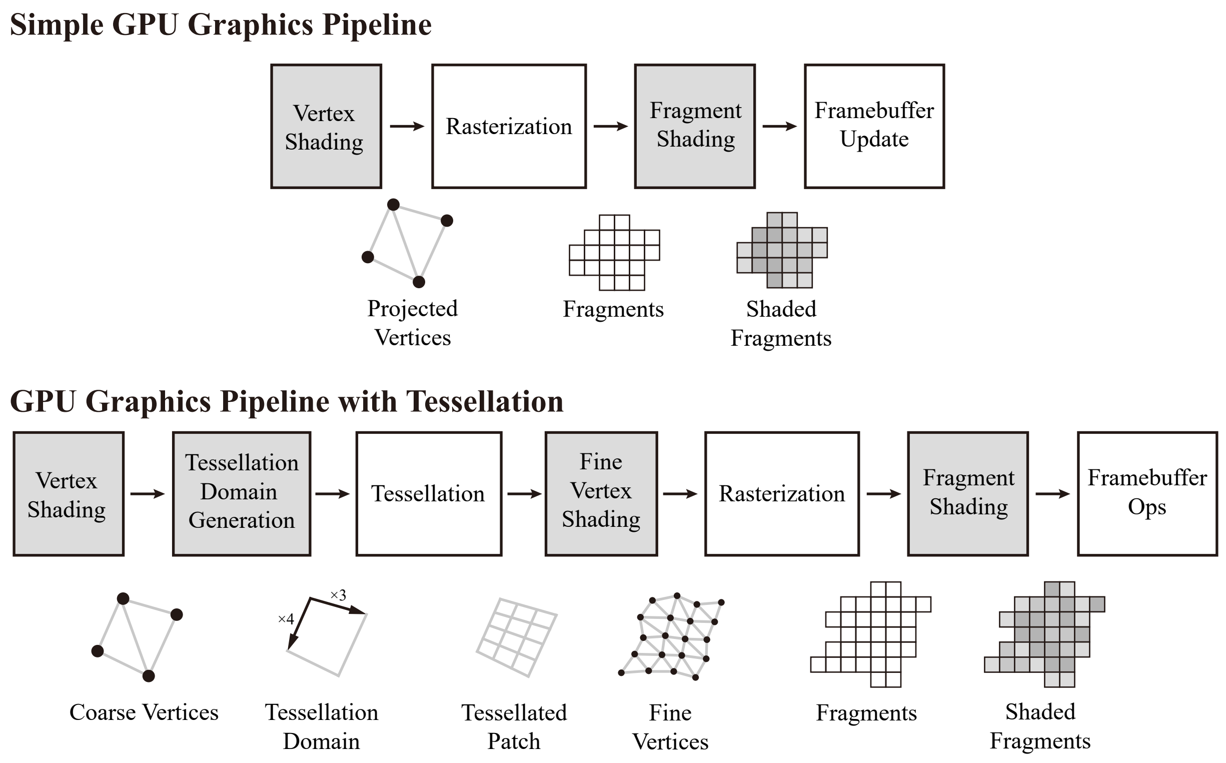


Figure 2.1: The GPU graphics pipeline architecture. Top: the simple GPU graphics pipeline configuration without tessellation stages. Bottom: the full pipeline configuration with tessellation enabled. Pipeline stages are shown in square boxes. Boxes in gray background represent programmable pipeline stages, and boxes in white background represent fixed-function stages.

Geometry Shader stages for simplicity, as 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 through out this thesis.

The graphics pipeline can be configured to include different set of stages. In its simplest form (as shown in the top half of Figure 2.1), the pipeline has four stages: Vertex Shading, Rasterization, Fragment Shading and Framebuffer Update. In this pipeline configuration, 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*.

The simple pipeline configuration requires the input geometry data is a stream of triangles. When the application uses more advances geometry representations, such as subdivision patches or Bezier patches, it needs to configure the graphics pipeline to use its tessellation stages. Figure 2.1 (bottom) illustrates the full graphics pipeline with tessellation stages enabled. The full pipeline adds three stages to the simple configuration: Tessellation Domain Generation, Tessellation and Fine Vertex Shading. Since the Rasterization stage can only process triangles, the role of these tessellation stages is to translate the more advanced geometry representation into a set of triangles. When using tessellation, the Vertex Shading stages is no longer responsible for projecting vertices onto image space, it is used as an additional stage to map the input geometry data into another form to use as input to the Tessellation Domain Generation stage. The Tessellation Domain Generation stage runs an application provided *tessellation domain shader kernel* to output a stream of tessellation domains. A tessellation domain defines how a patch should be subdivided into smaller triangles. For example, it can specify that a patch should be partitioned as a 3×4 grid of triangles. Next, the fixed-function tessellation stage takes a tessellation domain and output a stream of fine triangle vertices as the result of subdividing the tessellation domain. The Fine Vertex Shading then executes a *tessellation evaluation shader kernel* to compute the projected image space location of each fine triangle vertex. These projected fine vertex locations can then be used by the rest of the pipeline to complete the draw command.

2.1.1 Programming Shader Kernels

Shader kernels are expressed as C-like functions in shading languages like GLSL [4] and HLSL [8]. 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 compute the values that should be written to destination image at that fragment location.

Shader kernels for different pipeline stages are linked together to define the *shader program* that can be bound to the pipeline at once. This thesis uses the term *bind shader program* to mean configuring the programmable pipeline stages to use shader kernels defined by the shader program.

2.1.2 Managing Pipeline States

The pipeline needs to know many things to draw an object. In addition to what shader program to use, a programmer must tell the GPU 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.

2.1.3 Using the Graphics Pipeline

Application access to the graphics pipeline is made available via standardized graphics APIs, such as OpenGL, Direct3D and Vulkan. Theses 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, this provides concrete parameter values to the shader kernels.
4. Send a draw command.

2.2 GPU Architecture Characteristics

To use the GPU efficiently, it is important to understand the underlying GPU architecture that implements the real-time graphics pipeline, and its implications on optimizing a graphics application.

Drawing a frame typically requires executing a large number of draw commands, and a real-time graphics application needs to draw many frames (typically 30 or 60) per second. For this reason, GPUs are optimized for maximum processing throughput of draw commands, with architectures that exploit both pipelining and data parallelism. We discuss both aspects in this section.

2.2.1 Pipelining

GPU architectures are highly pipelined, executing draw commands in almost the same stages as the graphics pipeline abstraction described in Section 2.1. Pipelining enables the GPU to execute multiple draw commands at a time, but there is one caveat: some pipeline states, such as the shader program, cannot be changed when there are active draw commands being executed. This means that changing these states leads to a pipeline stall - if a draw command requires using another shader program, the GPU need to wait until all the currently executing draw commands to finish before it can start executing the new draw command. An application can suffer from serious performance degeneration when it requests to change pipeline states too often.

For performance, it is important to exploit the coherence between draw commands to avoid pipeline stalls as much as possible. For this concern, the graphics APIs are designed to make every state change explicit - the CPU must send dedicated state-change commands to the GPU. It is the application's responsibility to reduce state changes by reordering draw commands.

Typically, shader kernels stay the same for large groups of draw commands, while some shader parameter values change per object (thankfully, changing shader parameter values does not cause pipeline stall in most GPU pipeline implementations). Shader parameter values are usually updated at different frequencies: some parameters are shared by many draw commands, while other parameters are updated per-object. Listing 2.1 shows a typical GPU pipeline command stream produced by an application. The application first binds the shader program and shared shader parameter values. For each object, the application sets its geometry data location and the object-specific shader parameter values, followed by a draw command. Such ordering of draw commands reduces the number of state-change commands need to be communicated to the GPU, and amortizes the GPU performance overhead in changing key pipeline states (shader

```
/* pipeline states shared by first batch of draw commands */
SetShaderProgram(shader0)
SetShaderParameter(...)  // shared parameter values

/* object 0 */
SetShaderParameter(obj0.parameters)   // per-object shader parameters
SetInputGeometryDataLocation(obj0.geometryData)
Draw()  // issue a draw command

/* object 1 */
SetShaderParameter(obj1.parameters)   // per-object shader parameters
SetInputGeometryDataLocation(obj1.geometryData)
Draw()  // issue a draw command

...

/* pipeline states shared by second batch of draw commands */
SetShaderProgram(shader1)
SetFixedPipelineStates(...)
SetShaderParameter(...)  // shared parameter values

/* object 10 */
SetShaderParameter(obj10.parameters)   // per-object shader parameters
SetInputGeometryDataLocation(obj10.geometryData)
Draw()  // issue a draw command

/* object 11 */
SetShaderParameter(obj11.parameters)   // per-object shader parameters
SetInputGeometryDataLocation(obj11.geometryData)
Draw()  // issue a draw command

...
```

Listing 2.1: A typical GPU pipeline command stream for drawing scene objects.

program) that causes pipeline stalls. This listing exhibits two frequencies of state changes - per-batch and per-object. An actual application may divide state updates into more frequencies to further reduce state changes.

2.2.2 Data Parallelism

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. This is accelerated by the GPU with a multi-threading and wide SIMD architecture, with each SIMD lane processing one vertex or fragment. For example, a single core Stream Multiprocessor core of NVIDIA's Pascal GPU architecture supports up to 32 way multi-threading (executing two of them at a time), and each thread features 32 SIMD lanes. This enables a single core to process up to 1024 vertices or fragments simultaneously.

The GPU relies on multi-threading to hide memory latency. However, hardware multi-threading requires all thread's execution contexts (active local variables) being stored in fast

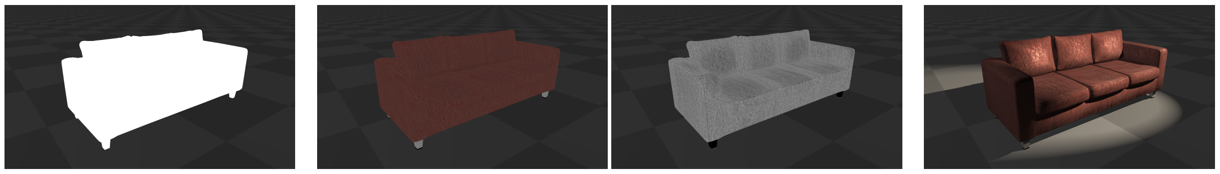

(a) geometry transform only
(b) surface pattern (base color, roughness)
(c) lighting

Figure 2.2: The shading features implemented in our example shading system.

on-chip memory. If a shader kernel requires a large execution context (has many live local variables), the GPU will not have enough on-chip memory to hold as many execution contexts as for simpler shader kernels. Having not enough active threads running on-chip reduces the GPU's capability to hide memory latency and results in degenerated performance.

On the other hand, wide SIMD architecture means that the GPU is not as efficient when executing code with divergent control flows. Therefore, developers always strive to avoid using dynamic control flows in the shader code.

2.3 Designing a Real-time Shading System

2.3.1 Typical Shading Features

To reveal the challenges in a real-time shading system, this section briefly introduces several most common shading features as seen in many typical 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.

Figure 2.2 illustrates renderings of a couch with different shading features. The simple pipeline configuration (that includes only two programmable stages: the vertex and fragment shading stage) is used to render the couch because the geometry is defined by triangles and the pipeline's tessellation stages are not needed. From left to right, the figure illustrates three categories of shading features: geometry transform, surface patterns, and lighting. The following text explains each of these categories in-detail.

Geometry Transform. The first step of rendering an image is to figure out where on the final image should the object appear. The geometry transform features in our system takes a camera setting (including camera position, view direction, and a field-of-view angle) as parameter, and compute the projected screen-space coordinates of each 3D vertex in the input geometry data as if it is viewed from the given camera. By performing geometry transform, we will see an image as shown in Figure 2.2 (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 white. This is because we have not included any shading features to determine the color of each pixel representing the couch surface.

Surface Pattern. Surface patterns are used to provide a sense of an object's physical material and depict the imperfect details - such as dents, scratches dirt and wear - that are essential to image realism. A surface pattern defines the light interaction properties at each location on the surface, such as how much light is reflected or absorbed.

Figure 2.3 shows six different surface patterns applied to the same sphere geometry and rendered with same lighting setup. When illuminated by lights, objects with various surface patterns appears vastly different, suggesting an object's material, such as metal, wood, brick and dirt. Therefore, many shading systems refer to a surface pattern as a *material*. Figure 2.4 is a close-up view of the sphere with a wood surface pattern. Details of imperfection such as dents and scratches can be seen clearly in this view.

Modern AAA video games include tens of thousands of unique surface patterns, which defines the detailed appearance for each type of scene object. For example, Bungie's Destiny features approximately 18,000 unique surface patterns. To aid development of this many surface patterns, many shading systems (such as Bungie's shader system and Unreal Engine 4) feature a GUI editor that enables artists to create surface pattern shading logic by mix and matching reusable building blocks from a predefined library.

To simulate the surface appearance of the couch in Figure 2.2, the renderer need to compute a surface pattern. Figure 2.2 (b) visualizes the patterns for two surface properties: base color and roughness. These renderings provide more details of the surface: they reveal the leather pattern, seams between patches of leather, and wear on the cushion.

Lighting. Despite more details, the renderings of Figure 2.2 (b) still look flat and unrealistic due to the lack of another important shading feature - lighting. In reality, the visual appearance of an object is determined by the amount of light reflected by the object's surface and perceived by the camera. The surface patterns only defines how the surface reflects light. For example, a red base color means the surface is reflecting mostly red light and absorbing lights in other frequencies. The lighting feature computes whether a surface location receives any light from light sources (shadowing), and determine how much light is reflected to the camera based on the surface properties at that location. Figure 2.2 (c) shows a final image with both surface pattern and lighting features computed. Lighting plays an essential part in revealing the details of a surface pattern. As can be seen in the final image, surface details such as wrinkles, wear and

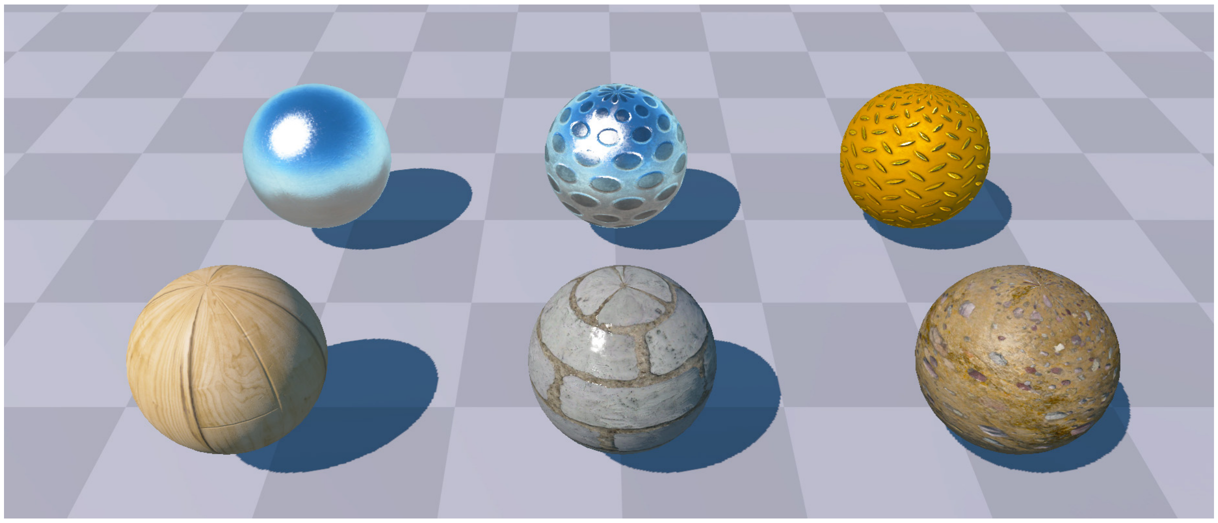


Figure 2.3: Examples of various surface patterns. Six different surface patterns are used to render the same sphere geometry, under the same lighting setup. Surface pattern defines how light is reflected at various points on the surface, giving a sense of the object's material.

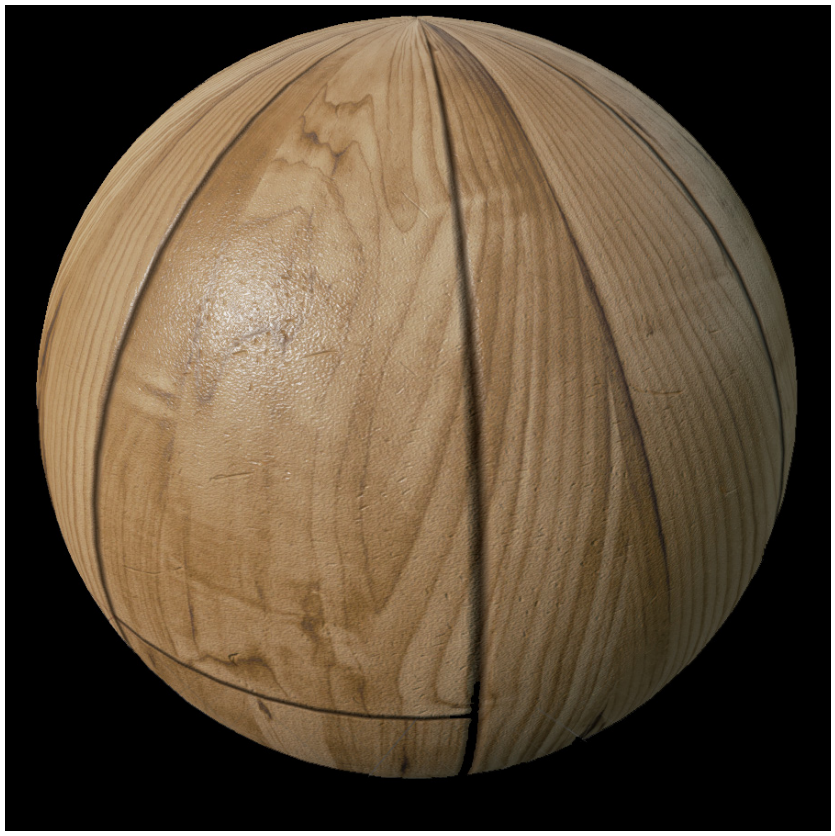


Figure 2.4: Close-up view of a sphere rendered with a wood surface pattern. The surface pattern depicts details of imperfection (such as dents and scratches) that are essential to provide a sense of realism.

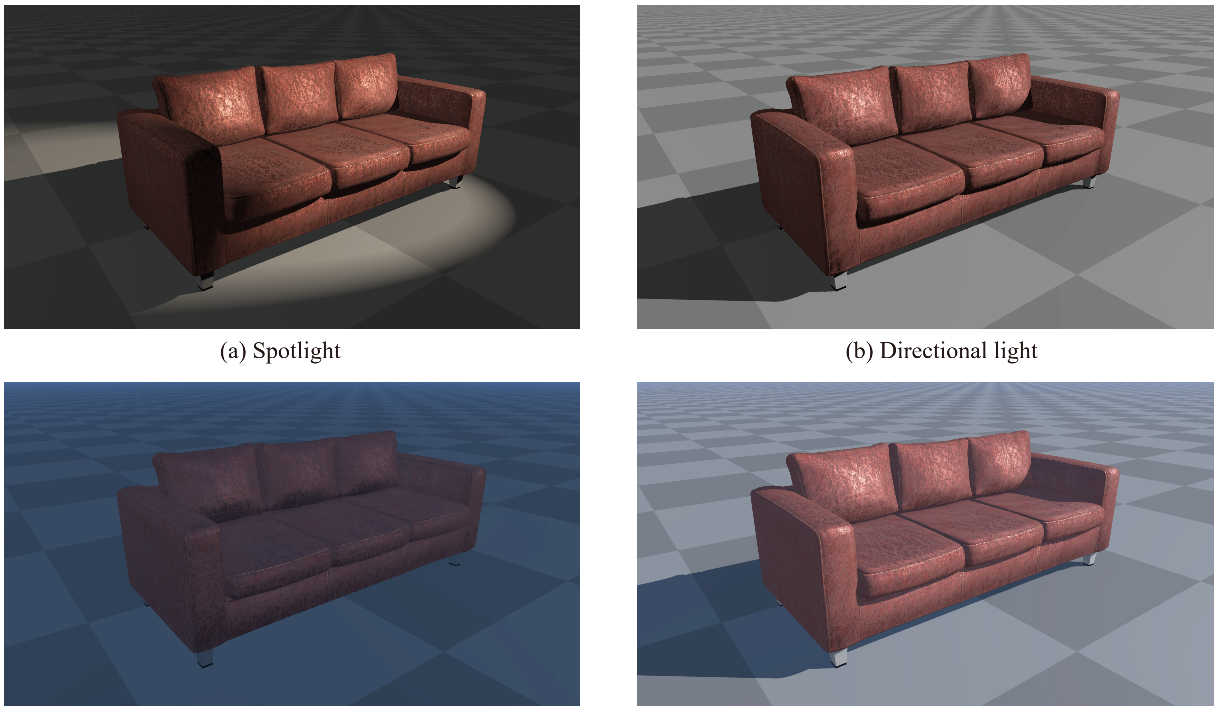


Figure 2.5: Renderings of the couch object illuminated by different types of light sources. (a) Illuminated by a spot light. (b) Illuminated by a directional light representing the sun. (c) Illuminated by a sky light. (d) Illuminated by a combination of the sky light and directional light, a common out-door lighting setup.

seams are more pronounced under the spotlight.

Modern shading systems support many types of light sources and can compute lighting from arbitrary collection of light sources. By setting up light sources, artists can create a diversity of scene atmospheres. Figure 2.5 shows the same couch object illuminated by different types of light sources.

All these shading features are computed in the programmable pipeline stages on the GPU. For example, geometry transform needs to be done in the vertex shading stage to provide the graphics pipeline the projected coordinates. The surface pattern is typically computed in the fragment shading stage, because the surface property values vary at different locations of the surface, and the fragment shader kernel by definition is executed once for each pixel fragment (which represents a different surface location). Lighting is also computed in fragment shading stage to provide pixel level details.

Drawing an object requires using a set of shader kernels in the programmable pipeline stages that implement a combination of these shading features depending on the object property and the lighting setup of the scene. For example, to render the couch in Figure 2.2 (c), the shader kernels must implement three shading features: camera view transformation, couch leather surface pattern, and lighting using a shadowed spot light.

2.3.2 Authoring Modular Shader Code

Achieving software extensibility and flexibility is a well-studied field in computer science. One of the most adopted technique in graphics applications is the object-oriented design. In this section, we examine a modular shading system framework based on the object-oriented concepts. We then discuss the challenges and workarounds in realizing this design due to limitations in existing shading languages.

A Modular Design

As established in Section 2.3.1, a typical shading problem involve interaction among three types of entities:

- A camera that defines how to transform the 3D object vertices to image-space coordinates.
- A surface pattern that defines the light response properties at each point on the object's surface.
- A lighting environment that determines the appearance of an object as affected by a collection of light sources.

Listing 2.2 outlines the framework of shader code that models the relationship among these three entities. The camera concept is encapsulated by the `Camera` class (line 1), which defines both the parameters (line 4-5) and computation for transforming object vertices into image-space. The `ISurfacePattern` interface (line 24) abstracts many different types of surface patterns. It defines that any surface pattern module must provide a `computeSurfaceProperty` method that returns the surface properties (defined by the `SurfaceProperty` struct in line 15) for a given location on the object surface (specified through the `uv` parameter).

The `Lighting` class (line 37) implements the lighting feature that computes the object's final appearance as affected by a collection of light sources. The shading system needs to support three different types of light sources: spotlight (as shown in Figure 2.2 (c)), directional light (typically used to model the sun), and sky light (models illumination from all directions of the sky). The `ILight` interface (line 30) abstracts the differences of light sources, and requires a light source implementation to provide a `computeLighting` method to compute the lighting contribution of the light source at a given surface location. The `Lighting` class simply sums up the lighting contributions from all input light sources stored in the `lights` array (line 39) to produce the final color output.

To draw objects, the shading system must provide to the GPU graphics pipeline the shader kernel functions that use the shading features in this framework. Listing 2.3 shows the entry-points of the vertex and fragment shader kernels. The `VertexShader` function (line 21) takes as input a `Camera` instance (`cam`) for transforming input vertices to image-space, and an vertex (`vert`) from input geometry data. The `VertexShader` kernel function will be executed once for each input vertex by the vertex shading stage of the GPU graphics pipeline. Therefore, the `vert` parameter, which represents an input vertex, varies in each invocation of `VertexShader` kernel. On the other hand, the same camera settings will be used transform all vertices from the input geometry data, so the `cam` parameter is qualified with the `uniform` keyword to indicate that it will not change across different kernel invocations. A *uniform* parameter is stored in a shared GPU memory location that is initialized by the CPU and is read-only to shader kernels.

The structure for an input vertex from the geometry data is defined in the `InputVertex` structure (line 2). In addition to defining a world space 3D coordinate (`position`, line 4), an input vertex also contains values of additional surface attributes at the vertex's location (including the normal and tangent vector, and a 2D coordinate, `uv`, used to identify the relative location of the vertex on the object's surface).

The `VertexShader` kernel function simply calls `Camera` class's `transformVertex` method to compute a projected coordinate of the input vertex. It passes through all attribute values from the input vertex as its output to make them accessible in the fragment shading stage.

The fragment shader kernel is defined by the `FragmentShader` function (line 37). The fragment shader kernel takes as input an interpolated vertex shader output at the fragment's location (`v`) that is varying with each fragment shader kernel invocation, and additional uniform parameters for camera settings (`cam`), surface pattern (`surfPattern`) and lighting environment (`lighting`). The fragment shader kernel first calls the `surfPattern`'s `computeSurfaceProperty` method to compute the surface properties at the fragment's location (line 44), then calls `lighting`'s `computeLighting` method with the computed surface properties to get the final color of the pixel (line 46).

This design uses the interface concept from object-oriented programming to achieve extensibility and flexibility. The entry-point kernel functions are written against the `ISurfacePattern` and `ILight` interfaces and are independent of which concrete surface pattern or light source is in-use. The developer can easily add new types of surface patterns or light sources to the shader system without modifying existing shading features or entry-point shader kernels. All possible combinations of different surface patterns and light sources are guaranteed to work. These are

```
/* The camera view transform module */
class Camera
{
    // matrices used for transforming vertices
    float4x4 projectionMatrix;
    float4x4 viewMatrix;
    // position of the camera
    float3 position;

    // a method to transform vertex coordinates to image-space
    float4 transformVertex(float3 input)
    {
        return projectionMatrix * viewMatrix * float4(input, 1.0);
    }
}

/* defines the surface properties at a single surface location */
struct SurfaceProperty
{
    float3 baseColor;
    float3 normal;
    float roughness;
    float metallic;
}

/* defines the interface for a surface pattern module */
interface ISurfacePattern
{
    SurfaceProperty computeSurfaceProperty(float2 uv);
}

/* defines the interface for a light source */
interface ILight
{
    float4 computeLighting(float3 pos, float3 viewDir, SurfaceProperty surf);
}

/* the lighting module, computes lighting at a surface location
   using a collection of lights. */
class Lighting
{
    ILight lights[];
    int numLights;
    float4 computeLighting(float3 pos, float3 viewDir, SurfaceProperty surf)
    {
        // sum up lighting results from individual light sources
        float4 result = 0.0;
        for (int i = 0; i < numLights; i++)
            result += lights[i].computeLighting(pos, viewDir, surf);
        return result;
    }
}
```

Listing 2.2: A conceptual shader framework written in a C++ like language that uses object-oriented mechanisms to model three shading concepts: camera, surface pattern, and lighting.

```
/* The content of a vertex from input geometry data*/
struct InputVertex
{
    float3 position;  // the world-space position of the vertex
    float3 normal;    // the surface normal vector at this vertex location
    float3 tangent;   // the surface tangent vector at this vertex location
    float2 uv;        // the surface local uv coordinate at this vertex location
}

/* Defines the per-vertex output in Vertex Shading stage */
struct VertexShaderOutput
{
    float4 projectedCoordinates;
    float3 worldPosition;
    float3 surfaceTangent;
    float3 surfaceNormal;
    float2 uv;
}

/* The entry-point kernel function for the Vertex Shading stage */
VertexShaderOutput VertexShader(
    uniform Camera cam,
    in InputVertex vert )
{
    VertexShaderOutput result;
    // compute the projected coordinate using camera
    result.projectedCoordinates = cam.transformVertex(vert.position);
    // pass through rest of vertex attributes
    result.worldPosition = vert.position;
    result.surfaceTangent = vert.tangent;
    result.surfaceNormal = vert.normal;
    result.uv = vert.uv;
    return result;
}

/* The entry-point kernel function for the Fragment Shading stage */
float4 FragmentShader(
    uniform Camera cam,
    uniform ISurfacePattern surfPattern,
    uniform Lighting lighting,
    in VertexShaderOutput v )
{
    // compute surface properties at this fragment's location
    SurfaceProperty prop = surfPattern.computeSurfaceProperty(v.uv);
    // compute lighting given the surface properties.
    return lighting.computeLighting(v.position, cam.position, prop);
}
```

Listing 2.3: A conceptual implementation of shader kernel entry-points for the vertex shading stage and fragment shading stage, using modules from the framework illustrated in Listing 2.2.

all good system properties that we seek to achieve.

An HLSL Implementation

In practice, a shader library implementation is subject to many additional constraints due to performance goals and shading language restrictions.

The most commonly used shading languages that come with modern graphics APIs, are low-level C like languages and does not include full support for object-oriented programming mechanisms. For example, most shading languages do not support interfaces. HLSL supports classes and interfaces in a limited way, and does not allow an uniform parameter to have an abstract interface type or defining an array of interface-typed objects (such as line 39 of Listing 2.2). GLSL has no support for classes or interfaces. Furthermore, the requirement for an application to run on multiple software/hardware platforms using different graphics APIs has forced shader code be written using the common set of shading language mechanisms that exist on all platforms to simplify porting shader code to different platforms.

Depending on how a shading system trades off between performance and flexibility, dispatch of computation to concrete shading features is either implemented as a run-time operation via dynamic control flow (`if` or `switch` statements), or by specializing shader kernels at shader compile-time via meta-programming or preprocessor directives to generate a *variant* of shader kernel that contains only the code for required shading features. Dispatch using dynamic control flow in shader code requires minimal CPU work to compile and use the shader kernels (there is only one version of shader code), but incurs GPU execution overhead because the same code dispatching logic is repetitively performed at each shader kernel invocations (which occurs millions of times per frame), and because large shader kernels tend to use more registers which leads to less efficient GPU utilization. On the other hand, dispatch via specialization incurs more CPU work to compile and manage specialized shader kernel variants, and is infeasible when total number of variants is unbounded.

Figure 2.6 shows an HLSL implementation of the example shader library. This implementation uses plain C-like functions and structs to mimic a `class`.

Dispatch of surface pattern is done at shader compile-time to reduce GPU execution overhead. The HLSL implementation of the shading features is separated into several HLSL source files. `ShaderLibrary.hlsl` contains the implementation for the camera transform and lighting features, and includes a forward declaration of the `SurfacePattern` struct and their associated functions. Concrete implementations to the surface pattern feature are not included in `ShaderLibrary.hlsl`, instead, they are provided in separate files, such as `CouchPattern.hlsl` that implements the surface pattern computation for the couch, or `WoodPattern.hlsl` that im-

`ShaderLibrary.hlsl` (common features and declarations)

```
struct Camera
{
    float4x4 projectionMatrix;
    float4x4 viewMatrix;
    float3 position;
}

float4 transformVertex(Camera cam, float3 input)
{
    return cam.projectionMatrix * cam.viewMatrix * float4(input, 1.0);
}

// forward declaration of a surface pattern module
struct SurfacePattern;
SurfaceProperty computeSurfaceProperty(SurfacePattern p, float2 uv);

// a unioned struct that contains parameters for all types of light sources
struct Light
{
    int lightType;
    DirectionalLightParams dirLight;
    SpotlightParams spotLight;
    SkyLightParams skyLight;
}
// functions that compute lighting from each types of light sources
float4 computeDirectionalLighting(DirectionalLightParams light,
    float3 pos, float3 camPos, SurfaceProperty prop) {...}
float4 computeSpotLighting(SpotLightParams light,
    float3 pos, float3 camPos, SurfaceProperty prop) {...}
float4 computeSkyLighting(SkyLightParams light,
    float3 pos, float3 camPos, SurfaceProperty prop) {...}

// implementation of lighting feature
struct Lighting
{
    int numLights;
    Light lights[MAX_LIGHTS];
}
float4 computeLighting(Lighting l, float3 pos, float3 camPos, SurfaceProperty prop)
{
    float4 result = 0.0;
    for (int i = 0; i < l.numLights; i++)
    {
        Light light = l.lights[i];
        if (light.lightType == 0)
            result += computeDirectionalLighting(light.dirLight, pos, camPos, prop);
        else if (light.lightType == 1)
            result += computeSpotLighting(light.spotLight, pos, camPos, prop);
        else if (light.lightType == 2)
            result += computeAreaLighting(light.skyLight, pos, camPos, prop);
    }
    return result;
}
```

`CouchPattern.hlsl` (couch surface pattern implementation)

```
struct SurfacePattern
{
    Texture2D baseColorMap;
    Texture2D normalMap;
}
SurfaceProperty computeSurfaceProperty(
    SurfacePattern p, float2 uv)
{ ... }
```

`WoodPattern.hlsl` (wood surface implementation)

```
struct WoodPattern
{
    Texture2D baseColorMap;
    Texture2D detailedColorMap;
    Texture2D normalMap;
}
SurfaceProperty computeSurfaceProperty(
    SurfacePattern p, float2 uv)
{ ... }
```

Figure 2.6: The same shading framework design as illustrated by Listing 2.2, implemented in HLSL using dynamic control flows and preprocessor techniques to simulate the interface dispatching mechanism.

`EntryPoint.hlsl` (shader kernel entry-points)

```
 1 #include <ShaderLibrary.hlsl>
 2 // include a concrete implementation of surface pattern
 3 #include SURFACE_PATTERN_FILE
 4
 5 // define uniform parameters
 6 cbuffer cb
 7 {
 8     Camera cam;
 9     SurfacePattern surfPattern;
10     Lighting lighting;
11 }
12
13 /* The entry-point kernel function for the Vertex Shading stage */
14 VertexShaderOutput VertexShader(in InputVertex vert )
15 {
16     VertexShaderOutput result;
17     // compute the projected coordinate using camera
18     result.projectedCoordinates = transformVertex(cam, vert.position);
19     // pass through rest of vertex attributes
20     result.worldPosition = vert.position;
21     result.surfaceTangent = vert.tangent;
22     result.surfaceNormal = vert.normal;
23     result.uv = vert.uv;
24     return result;
25 }
26
27 /* The entry-point kernel function for the Fragment Shading stage */
28 float4 FragmentShader(in VertexShaderOutput v)
29 {
30     // compute surface properties at this fragment's location
31     SurfaceProperty prop = computeSurfaceProperty(surfPattern, v.uv);
32     // compute lighting given the surface properties.
33     return computeLighting(lighting, v.position, cam.position, prop);
34 }
```

Figure 2.7: The shader kernel entry-points as illustrated by Listing 2.3, implemented in HLSL.

plements a wood pattern.

To select a surface pattern implementation, the developer includes its corresponding HLSL source file when compiling the shader kernels. Figure 2.7 shows the HLSL source file for the shader kernel entry-points. First, the programmer includes `ShaderLibrary.hlsl` (as in line 1). Depending on the actual scenario, the programmer selects different implementations of surface pattern feature by specifying the macro value of `SURFACE_PATTERN_FILE` at shader compile-time to include the an appropriate HLSL source file, such as `CouchPattern.hlsl` or `WoodPattern.hlsl`.

Contrarily, Dispatch of lighting computation for different types of light sources is implemented using run-time control flow because the system needs to support unlimited light source combinations. Our example shading system supports three types of light sources, spotlight, directional light and sky light. As shown in Figure 2.6 (line 18), the parameters of all types of light sources are aggregated into a single `Light` struct. The `Light` struct has a `lightType` field indicated the type of the light and which of the specific parameters is valid (`dirLight`, `spotLight` or `skyLight`). The library includes functions that compute lighting for each type of light sources (line 26-31). The lighting feature implementation (line 39) iterates through the input light sources array. For each light source, the code uses a dynamic `if` statement to call the light source's corresponding lighting function based on the value of `lightType` field.

None of the dispatching implementations (for surface pattern and for light sources) are perfect. The current implementation for dispatching surface patterns makes it not possible for a shader kernel to use two different surface patterns at the same time, because including two HLSL source files that implement concrete surface patterns into the entry-point source would result in redefinition of the `SurfacePattern` struct and `computeSurfaceProperties` function. Meanwhile, extending the current implementation with a new type of light sources involve changing code at multiple places: in addition to implementing a new function to compute the lighting contribution from the light source, the developer also needs to modify the `Light` struct definition to include a new set of light source parameters, assign a new ID to use in `lightType` field to represent this new type of light and ensure this ID is not clashing with existing types of light sources, and modify the dynamic dispatch logic in `computeLighting` function (Figure 2.6, line 39) to call the newly added lighting function. Furthermore, the `lights` array (Figure 2.6, line 37) is to be filled by the CPU with light source parameters. Since adding a new type of light source changes the definition of `Lighting` struct, the CPU code must also be updated in order to fill in the `lighting` array correctly.

With current shading languages, developers are forced to make decisions on how to implement dispatching when writing the shader code because this decision dramatically affects the shader code and the host-side CPU code that uses the shaders. Once the decision is made, it is not possible to change without significantly rewrite most part of the shading system. As a result, it is very difficult for a shading system to adapt to different applications or GPU architectures that can be more efficient with a different dispatching implementation.

To draw objects using the GPU graphics pipeline, the shading system needs to run CPU logic to setup the GPU pipeline state to use the compiled shader program, and communicate the parameter values for the shading features implemented in the shader program to the GPU. While workarounds exist for authoring modular shader code, the bigger challenge is to achieve high CPU performance in these tasks while maintaining shader code extensibility and flexibility, which is discussed in the following sections.

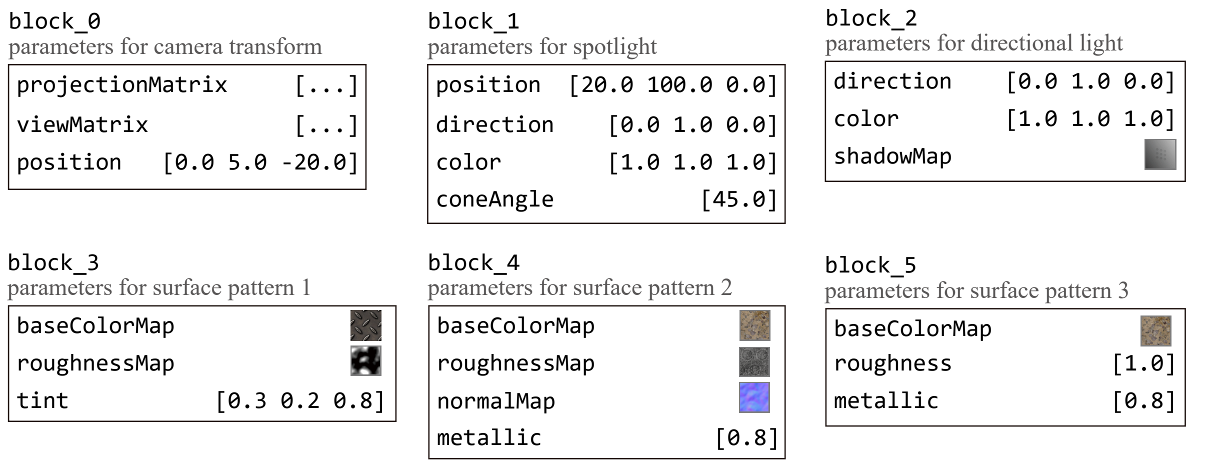
2.3.3 Compiling and Using Shader Programs

Because the dispatch of surface pattern is implemented via compile-time specialization, our shading system need to compile the shader kernels for each object that uses a different surface pattern. This is done at initialization time after the scene is loaded, so no more shader compilation work is necessary when the drawing the scene. For each surface pattern, our shading system invokes the HLSL shader compiler to compile `EntryPoint.hlsl` with the `SURFACE_PATTERN_FILE` macro specified to the corresponding source file that implements the surface pattern in the compiler options, and stores the compiled shader program in a shader cache. When drawing an object, the system finds the compiled shader program that implements the object's surface pattern from the shader cache, and bind it to the graphics pipeline. Because surface pattern is the only identification of a compiled shader program, the system uses the name of the surface pattern (a string) as the key to lookup from the shader cache.

2.3.4 Communicating Shader Parameters

When drawing large number of objects in a scene, the shader parameter values are updated at different frequencies: the same shader parameter values for camera view transformation are used to draw all objects in a scene, while the parameter values for surface pattern vary per object. An efficient renderer exploits this property to minimize the CPU-GPU communication needed to properly setup the shader parameters for each draw command.

To allow efficient parameter communication, Direct3D 12 and Vulkan introduce the concept of *parameter blocks* (called descriptor tables and descriptor set respectively). Figure 2.8 illustrates an efficient use of parameter blocks for parameter communication. During initialization, the renderer creates four parameter blocks to store parameter values for different shading features on the GPU: `block_0` stores the parameter values for the camera transform, `block_1` and `block_2` stores the parameter values for differnet lighting environments (representing a spotlight and a directional light, respectively), `block_3`, `block_4` and `block_5` stores the parameter values for different surface patterns. The camera transform parameter values in `block_0` will be used to draw all objects in the scene, and other parameter blocks are used in different scenarios. Figure 2.8 (bottom) shows a list of GPU commands required to draw three objects in the scene. To draw the first object, the renderer binds all parameter blocks for all the shading features (camera, lighting and surface pattern) needed by the shader kernel entry-points. The second object is affected by the same spotlight as the first object, and requires only a different surface pattern, so the renderer only needs to bind a different parameter block containing the parameters to the surface pattern feature. Similarly, the third object uses a different surface pattern and affected



block_0
parameters for camera transform

projectionMatrix	[...]
viewMatrix	[...]
position	[0.0 5.0 -20.0]

block_1
parameters for spotlight

position	[20.0 100.0 0.0]
direction	[0.0 1.0 0.0]
color	[1.0 1.0 1.0]
coneAngle	[45.0]

block_2
parameters for directional light

direction	[0.0 1.0 0.0]
color	[1.0 1.0 1.0]
shadowMap	

block_3
parameters for surface pattern 1

baseColorMap	
roughnessMap	
tint	[0.3 0.2 0.8]

block_4
parameters for surface pattern 2

baseColorMap	
roughnessMap	
normalMap	
metallic	[0.8]

block_5
parameters for surface pattern 3

baseColorMap	
roughness	[1.0]
metallic	[0.8]

```
BindParameterBlock(0, block_0);
BindParameterBlock(1, block_1);
BindParameterBlock(2, block_3);
Draw(object1);

BindParameterBlock(2, block_4);
Draw(object2);

BindParameterBlock(1, block_2);
BindParameterBlock(2, block_5);
Draw(object3);
```

Figure 2.8: An efficient way to communicate shader parameter values to the GPU using parameter blocks. Top: the shading system creates parameter blocks for different shading features at initialization. Bottom: a sequence of GPU commands to draw objects using the parameter blocks. Parameter blocks that contain shared parameter values are reused by different draw commands.

by a different directional light instead of the spotlight, so the renderer needs to bind the two parameter blocks for the surface pattern and lighting features.

This design has two major performance benefits. First, all parameter values for a shading feature are bound to the graphics pipeline in a single bind parameter block operation. This reduces the CPU overhead of repetitive API calls to communicate individual parameter values. Second, the parameter blocks created for shading features that are shared by multiple objects, e.g., camera and lighting, can be reused when drawing these objects without being repetitively transferred to the GPU.

To ensure correctness, the CPU logic and the compiled shader kernels must assume the same parameter block layout, so that the parameter values filled in by the shading system can be read correctly by the shader. When using OpenGL (with the bindless texture extension) and Metal, graphics programmers can encapsulate all shader parameters of a shading feature in a `struct` type, as shown in section 2.3.2. These graphics APIs provide the mechanism to map a `struct` type in the shader code to a parameter block. The layout of such a `struct` type is defined by shading language compiler rules and can be queried by the host application to determine how to populate the parameter block.

Unfortunately, the OpenGL bindless texture extension and Metal API are only available on specific hardware and software platforms. The language mechanism that maps a `struct` type to a parameter block is not a standard feature available in HLSL (the de facto standard shading language used in video games), GLSL (without the bindless texture extension) and SPIRV (shader IR used by Vulkan). Although Direct3D 12 and Vulkan support creating parameter blocks in the host application, their corresponding shading languages have no first-class construct to map a collection of shader parameters into a parameter block. By default, the HLSL compiler assumes all shader parameters are in the same parameter block. To implement efficient use of parameter blocks, the application developer must annotate each shader parameter in the entry-point shader to explicitly specify the parameter block it belongs to. But doing so compromises the modularity of a shading feature implementation, since the developer must know the content of each shading feature's parameters to properly write the layout annotation in the entry-point shader file.

Due to lack of universal shading language support for parameter blocks, it is not easy for a shading system that needs to target Direct3D 12 or Vulkan to use parameter blocks as intended. One common workaround is to communicate parameters by allocating and filling in a single parameter block for each draw command (in the format expected by the shader kernel entry-point), which incurs CPU overhead and unnecessary CPU to GPU communication.

```
1  float4 computeLighting(Lighting l, float3 pos, float3 camPos, SurfaceProperty prop)
2  {
3      float4 result = 0.0;
4  #if defined(ONE_DIRECTIONAL_LIGHT)
5      // special case 1: one directional light only
6      result = computeDirectionalLighting(l.lights[0].dirLight, pos, camPos, prop);
7  #elif defined(ONE_SPOT_LIGHT)
8      // special case 2: one spot light only
9      result = computeSpotLighting(l.lights[0].spotLight, pos, camPos, prop);
10 #elif defined(DIRECTIONAL_SKY_LIGHT)
11     // special case 3: one directional light and one sky light
12     result = computeDirectionalLighting(l.lights[0].dirLight, pos, camPos, prop);
13     result += computeSkyLighting(l.lights[0].skyLight, pos, camPos, prop);
14 #else
15     // general case, use dynamic control flow
16     for (int i = 0; i < l.numLights; i++)
17     {
18         Light light = l.lights[i];
19         if (light.lightType == 0)
20             result += computeDirectionalLighting(light.dirLight, pos, camPos, prop);
21         else if (light.lightType == 1)
22             result += computeSpotLighting(light.spotLight, pos, camPos, prop);
23         else if (light.lightType == 2)
24             result += computeAreaLighting(light.skyLight, pos, camPos, prop);
25     }
26 #endif
27     return result;
28 }
```

Listing 2.4: A new `computeLighting` function implementation that supports specialization. By specifying additional preprocessor macro definitions at compile time, this function can be specialized into four variants, computing lighting for one directional light, one spot light, combination of one directional light and one sky light, and for general combinations of lights, as controlled by the preprocessor directives in line 4, 7, 10 and 14.

2.3.5 Specializing Shader Kernels

While our shading system implementation uses dynamic control flow to dispatch lighting computation for different types of light sources to render scenes with arbitrary composition of light sources, most objects are affected by a simple lighting setup comprising only one or two light sources. For example, in an our-door scene, most objects are affected by only the sun light and the sky light. The shading system can avoid the overhead of dynamic control flow in these cases by using shader programs that are specialized to the concrete lighting setup when rendering an object affected by a simple set of light sources.

Existing shading languages including HLSL provide no direct support to specialize shader code. As a result, developers typically rely on the preprocessor to activate different parts of shader code. To support specialization of the lighting feature, the `computeLighting` shader function (as shown in Figure 2.6, line 39) is rewritten to allow conditional compilation into different variants. Listing 2.4 shows the new `computeLighting` function. This function can be compiled into four variants depending on preprocessor macros defined at compile-time. Defining `ONE_DIRECTIONAL_LIGHT` specializes `computeLighting` function to compute on directional light only (activating line 6), defining `ONE_SPOT_LIGHT` yields a variant that computes one spot light only, and defining `DIRECTIONAL_SKY_LIGHT` results in another variants that computes one directional light and one spot light. If none of these macros are defined at compile-time, the code for general code (line 16-25) is used, which computes lighting from arbitrary combinations of light sources with dynamic control flow.

This preprocessor based implementation of lighting feature specialization complicates shader program compilation and lookup. To enable specialization of lighting, the shading system need to compile the a shader program for each distinct combination of surface patterns and lighting environments. Caching and lookup of shader programs also gets more complicated, as a shader program is identified by a combination of surface pattern and lighting setup. This means increased CPU overhead in using more complicated keys to lookup shader programs from the shader cache when drawing each object.

2.3.6 Summary of Challenges

In summary, it is very difficult to implement a shading system that is extensible and efficient using existing shading languages due to the following challenges:

Writing modular and extensible shader code that uses different types of dispatching. Classes and interfaces are widely used language constructs in modeling complex software. However, many existing shading languages lack support for similar language constructs. Although HLSL does feature limited support for classes and interfaces, it provides no guarantee on how dispatching is implemented. Since implementation of dispatching (either via specialization or via dynamic control flow) is critical to GPU performance, it must be controllable by the shading system. Because of the lack of this shading language feature, existing shading system implementations resort on preprocessor directives to generate specialized shader code, which leads to reduced extensibility and maintainability.

Looking-up a specialized shader program with low CPU overhead. The shading system need to lookup a precompiled shader program to use when drawing each object. Since this lookup happens at per-object frequency, it must be highly optimized. Existing shading system implementations often choose the preprocessor macros used to compile the shader program as the lookup key, which is often a long string whose comparison is not efficient.

Communicating shader parameters efficiently using parameter blocks. Efficient shader parameter communication requires the shader code to group parameters in different parameter blocks based on frequency of update. It also requires the shader compiler to provide necessary introspection API for the shading system to populate parameter blocks correctly. However, HLSL does not provide a modular language construct that maps to a parameter block, and has no introspection API for the shading system to query the parameter block layout required by a shader program. This language limitation makes efficient parameter communication a conflicting goal with modular shader code authoring.

Chapter 3

Shader Components

As discussed in Chapter 2, developers wish to author shader code using object-oriented constructs for modularity and system flexibility, but the programming language implementation that maps these object-oriented constructs to dynamic dispatching is insufficient to meet the performance requirement in many cases. In fact, there is no single implementation of the object-oriented language constructs that meets all developers' needs: systems that require high rendering performance need to implement shader code polymorphism by Just-In-Time compiling and caching specialized variants; systems that have fewer combinations of different shader modules, but want to avoid the CPU cost of Just-In-Time compilation, need to statically populating all possible shader code combinations and generate all specialized shader variants offline; systems that involve flexible compositions that are too complex to be statically specialized, or systems that prefer minimum shader compilation overhead for fast development iteration, need the modularity constructs being implemented with dynamic dispatch. One of the challenges is to design a shading language that provide the object-oriented language constructs, but allow the shading system to control how dispatching is implemented. At the same time, the new shading language should also provide support for implementing efficient parameter communication.

The design of a shading language that aims to help a shading system to meet these high level performance and productivity goals must be driven by the constraints and requirements from actual shading systems. In this chapter, we present a shading system design pattern centered around a concept that we call *shader components*, which allows authoring and using modular shader code without being aware of how dispatching is implemented (thus making the implementation choice available as a late-bound decision after both the CPU and shader code are written), while at the same time facilitates efficient shader parameter communication. We then derive the shading language and shader compiler features needed to facilitate this design. To illustrate the design pattern, this chapter uses code examples written in the shading language we created called *Slang*, whose detailed design is presented in Chapter 4.

3.1 Overview

Modularizing shading features requires a modularity abstraction that spans both host-side CPU code and GPU shader 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 the correct shader program implementing the required code path is used by the graphics pipeline. We call this abstraction a *shader component*.

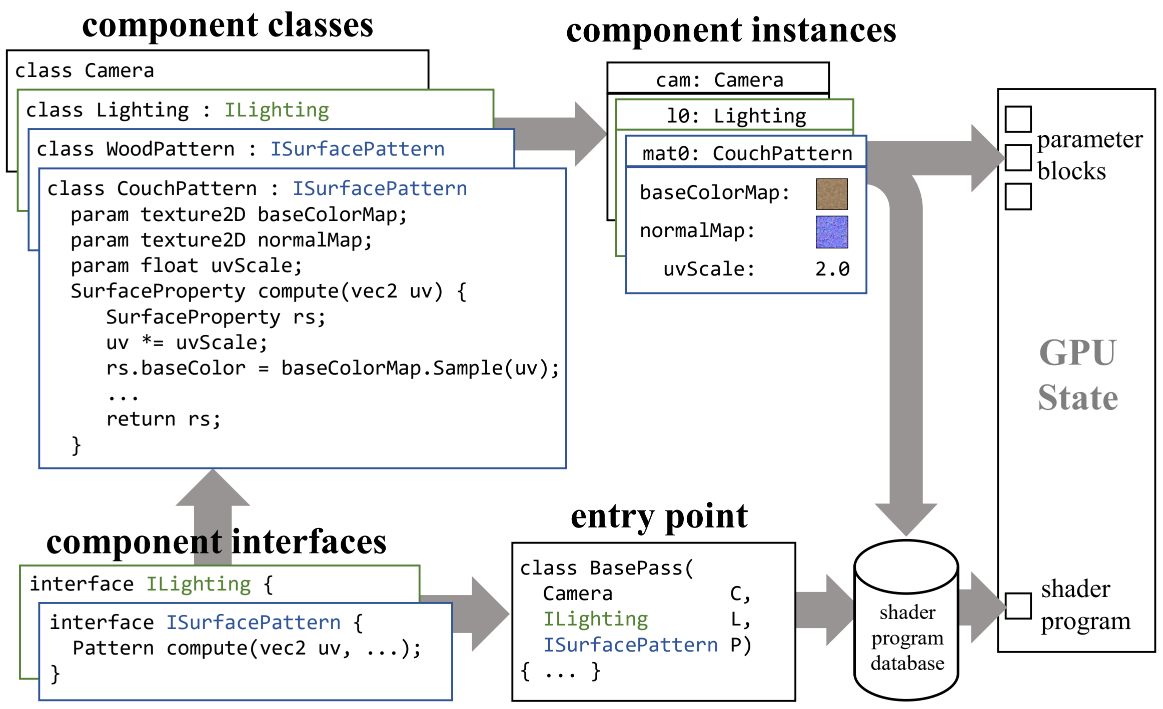
Figure 3.1 illustrates the key concepts of the shader components abstraction, using the same examples we have shown in Chapter 2. The remainder of this chapter describes the concepts in this figure, and has been organized around key principles that underlie the design.

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

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

In our design, a shader *component class* represents a shading language modular unit that encapsulates both the GPU shader code and parameters of a particular shading feature. For example, `CouchPattern` in Figure 3.1 is a component class with parameters for a base color texture map, a normal texture map, and a scaling factor to apply to texture coordinates. Another surface pattern component class, like `WoodPattern`, will in general have different parameters.



```
class Camera
class Lighting : ILighting
class WoodPattern : ISurfacePattern
class CouchPattern : ISurfacePattern
  param texture2D baseColorMap;
  param texture2D normalMap;
  param float uvScale;
  SurfaceProperty compute(vec2 uv) {
     SurfaceProperty rs;
     uv *= uvScale;
     rs.baseColor = baseColorMap.Sample(uv);
     ...
     return rs;
  }
```

```
interface ILighting {
interface ISurfacePattern {
  Pattern compute(vec2 uv, ...);
}
```

```
class BasePass(
  Camera          C,
  ILighting       L,
  ISurfacePattern P)
{ ... }
```

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 classes*. Instances of these classes encapsulate parameter values, and map to parameter blocks in modern APIs. An entry point and component arguments together determine a shader variant. Component classes and instances have been color-coded to match the interfaces they implement.

3.1.2 Holding and Communicating Shader Parameters

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

Building on the idea of a shader component class, our design allows host application code to allocate objects called *component instances* from these classes. An instance stores concrete values for the parameters declared in the class. For example, `mat0` in Figure 3.1 is an instance of the `CouchPattern` class that 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 an engine allocates is backed by a parameter block in the target graphics API. In Figure 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 *classes* are implemented in shader code (which requires the shading language to provide necessary mechanisms), while shader component *instances* are created by the shading system at runtime. 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 Chapter 4.

3.1.3 Composing a Shader Program from Shading Features

In our design, a shader *entry point* coordinates the overall execution and dataflow of shader code when rendering an object.

For example, the `BasePass` entry point in Figure 3.1 (and Listing 3.1) invokes surface pattern and lighting features to compute object appearances. The main shader logic in Listing 3.1 is almost the same as in Listing 2.3, except it wraps all the kernel function entry-points in a single class so that the entry-point can be more easily referred to by the shading system.

A key point of our design is that an entry point should be thought of as incomplete, with "holes" where specific components will be plugged in (these holes are parameters in a technical sense, but should not be confused with *shader* parameters, so we avoid the term). The shape of a hole can be given by concrete component classes (e.g., `Camera` in Listing 3.1, line 1), or by component *interfaces*. For example, the `BasePass` entry point requires a surface pattern

```
class BasePass(Camera cam, ILighting lighting, ISurfacePattern surfPattern)
{
    /* The content of a vertex from input geometry data*/
    struct InputVertex
    {
        float3 position;  // the world-space position of the vertex
        float3 normal;    // the surface normal vector at this vertex location
        float3 tangent;   // the surface tangent vector at this vertex location
        float2 uv;        // the surface local uv coordinate at this vertex location
    }

    /* Defines the per-vertex output in Vertex Shading stage */
    struct VertexShaderOutput
    {
        float4 projectedCoordinates;
        float3 worldPosition;
        float3 surfaceTangent;
        float3 surfaceNormal;
        float2 uv;
    }

    /* Vertex Shading kernel */
    VertexShaderOutput VertexShader(InputVertex vert )
    {
        VertexShaderOutput result;
        // compute the projected coordinate using camera
        result.projectedCoordinates = cam.transformVertex(vert.position);
        // pass through rest of vertex attributes
        result.worldPosition = vert.position;
        result.surfaceTangent = vert.tangent;
        result.surfaceNormal = vert.normal;
        result.uv = vert.uv;
        return result;
    }

    /* Fragment Shading kernel */
    float4 FragmentShader(VertexShaderOutput v )
    {
        // compute surface properties at this fragment's location
        SurfaceProperty prop = surfPattern.computeSurfaceProperty(v.uv);
        // compute lighting given the surface properties.
        return lighting.computeLighting(v.position, cam.position, prop);
    }
}
```

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

component (`pattern`) that must implement the `ISurfacePattern` interface.

A component interface represents a kind of feature (e.g., surface pattern, lighting) in a shader library, and declares the methods that all implementations of that feature must provide. For example, the `ISurfacePattern` interface in Figure 3.1 declares a method named `compute` that returns the surface properties at a given surface location.

The terminology we use for shader components uses object-oriented concepts like classes and interfaces, which are typically associated with dynamic dispatch (e.g., virtual function tables). However, the 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:

```
class BasePass( ILight light, ISurfacePattern surfPattern ) {...}
```

as being syntactic sugar for the following “templated” definition:

```
class BasePass<L : ILight, P : ISurfacePattern>( L light, P surfPattern ) {...}
```

An alternative approach would be to implement dispatch for components dynamically. As discussed in Chapter 2, real-time shader code typically benefits from aggressive specialization, so a shading language that supports classes and interfaces should ensure the default dispatch implementation results in fast GPU code. However in certain cases such as the lighting feature of our example shading system discussed in Chapter 2, generation of dynamic dispatch code is required to provide flexibility. One key insight of shader components is to decouple the implementation of dispatching from the how shader code and CPU code are written; this enables developers to optimize for different use scenarios without re-implementing the system.

3.1.4 Shader Program Variant Lookup and Caching

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

The performance of lookup in a shader program database is critical, because shader programs may be selected in the inner-most rendering loop. Rather than try to implement a one-size-fits-all shader program database in a language runtime library, our design leaves the responsibility for caching and lookup up shader program to the shading system, and requires the shader compiler to provide specific services to enable efficient implementation. We discuss one implementation strategy used in our shading system in Chapter **??**.

3.2 Shading System Workflow

The shader components concept aims at simplifying shading system tasks into a minimal set of operations that can be implemented efficiently with shader compiler support. In this section, we review the tasks that need to be performed by the shading system and highlight the key places that are improved by adopting shader components its corresponding compiler services.

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 classes and shader entry-points:

```
ShaderLibrary* lib = loadLibraryFromFile("shaderlib.shader");
```

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

```
ComponentClass* couchClass = findComponentClass(lib, "CouchPattern");
Entrypoint* basePass = findEntryPoint(lib, "BasePass");
```

The handles to component classes 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 a parameter block. A shading system can represent a component instance as a C++ class like the following:

```
class ComponentInstance
{
    ComponentClass* componentClass;
    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 class. This allows the shading system to allocate a parameter block and fill in concrete parameter values in the same format anticipated by the shader.

```
1  BindEntryPoint(basePass);
2
3  BindComponentInstance(0, cameraInstance);
4  BindComponentInstance(1, lightingInstance0);
5  BindComponentInstance(2, surfacePatternInstance0);
6  Draw(object0);
7
8  BindComponentInstance(2, surfacePatternInstance1);
9  Draw(object1);
10
11 BindComponentInstance(1, lightingInstance1);
12 BindComponentInstance(2, surfacePatternInstance2);
13 Draw(object2);
```

Listing 3.2: Sequence of operations to draw objects using shader components.

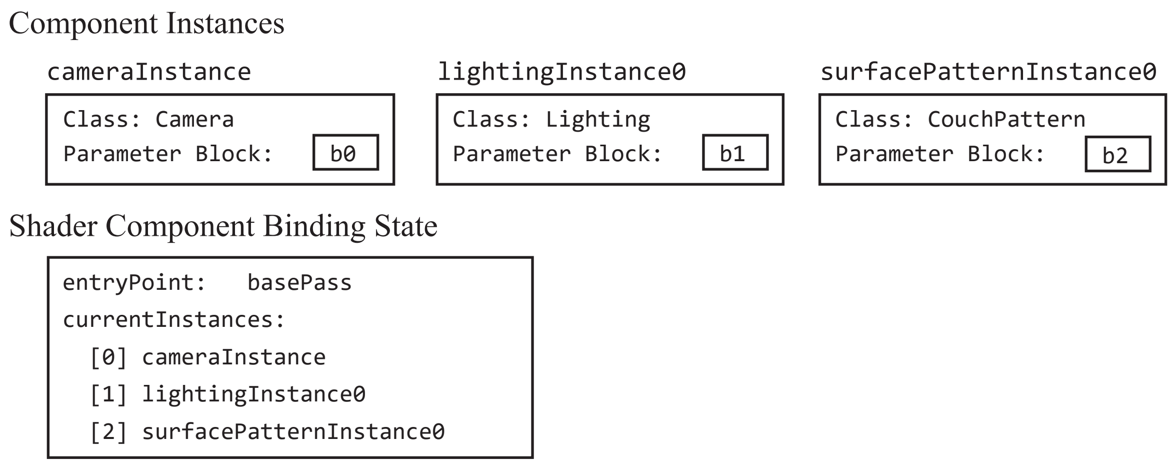


Figure 3.2: Shading-system-maintained states used to configure graphics pipeline for a draw command. `entryPoint` tracks the currently selected shader entry-point. `currentInstances` tracks the currently selected component instances.

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 3.2 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. `BindEntryPoint` function only changes a shading-system-maintained state tracking the currently selected shader entry point (`entryPoint` in Figure 3.2).

The shading system `BindComponentInstance` to attach component instances to entry-point parameters (line 3-5). This function modifies another shading-system-maintained state tracking the currently selected component instances at various slots (`selectedInstances` in Figure 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 Figure 3.2, are used to prepare the GPU pipeline for rendering.

In order to configure the GPU for rendering, the framework must both determine the final compiled shader kernels to use, and bind suitable parameter blocks that will provide parameter data to those shader kernels. With shader components, these two tasks are unified. The currently bound shader entry-point (`entryPoint`), together with the component classes of all currently selected component instances (`currentInstances`) fully defines an executable shader program. The shading system can lookup a shader program cache to see if a shader program resulting from this combination of entry-point and component classes 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. On the other hand, since each component instance also encapsulates a parameter block containing the required parameter values for the component, the shading system directly binds the parameter block to the GPU pipeline state to complete parameter communication.

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:

Enable 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 more 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 class can be used with any compiled shader programs that use the component class. 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 program 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, a shader program can be uniquely identified by the entry-point and the component classes used to compile the program.

This allows very compact key structure (such a short array of IDs of the entry-point and component classes) for looking-up shader programs from a cache. In contrast, a shading system that uses 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.

Shading system controls the implementation of dispatch. Shader code implementing the shader component concepts uses classes and interfaces to achieve polymorphism. The implementation of code dispatch - via dynamic control flow or via static specialization is controlled by a compiler option. This implementation decision is even decoupled from the host-side logic of the shading system. The shading system always follows the same sequence of operation - bind entry point, bind component instance, draw - regardless of how shader code dispatch is implemented. This enables shading systems to easily adapt to different hardware and scene configurations and make the right decision per use-case.

Chapter 4

The Slang Shading Language (Proposed Work)

This chapter presents the Slang shading language, which extends HLSL with several key language features to support implementation of the Shader Components shading system design.

4.1 Design constraints and principles

We want Slang to be a shading language that replaces HLSL in production environments. This implies that Slang should not only provide software engineering and performance benefits to graphics developers, but also designed to be easy to learn and to 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. Slang is expected to be used by graphics developers who are familiar with popular object-oriented languages such as C++, C$^{\#}$, Go and Swift. For this reason, we prefer using well-known language features from these languages, and avoid introducing exotic language constructs that would require non-trivial learning effort. This ensures any shader code written is Slang can be easily understood by developers even if they have not used Slang before.

No presumed policies. Shading systems often impose a series of policies on organizing shading features, using some (but not other) mechanisms to communicate parameter data, and when or whether to use specialized shader code for certain shading features. For example, the shader components design assumes one policy that one component corresponds to one parameter block. While Slang is designed to enable implementation of shader components, we still want the language itself to be provide only necessary mechanism for developers to implement any policies they want, instead of forcing a shading system that uses Slang to opt-in a certain policy.

Compatible with existing HLSL. Adopting a new shading language in an existing shading system can be a very challenging task. We would like to make this adopting process easy by allowing developers to adopt the new language features for a small part of the code-base at a time 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.

Report errors early and accurately. We would like the Slang compiler to be able to find most code errors at earliest possible stage and report error messages that pin-points the true source of problem. To improve development productivity, Slang is designed to run full check on individual modules, and provide guarantee that any valid composition of individually checked modules 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, and not having 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 advises 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 omit the features that are already in HLSL unless it interacts the new features in an interesting way.

```
1  // defining a generic struct type]
2  struct Pair<TKey, TValue>
3  {
4      TKey key;
5      TValue value;
6  };
7
8  // defining a generic function returning a generic struct type
9  Pair<TKey, TValue> makePair<TKey, TValue>(TKey k, TValue v)
10 {
11     Pair<TKey, TValue> rs;
12     rs.key = k;
13     rs.value = v;
14     return rs;
15 }
16
17 // explicit specialization
18 Pair<int, float> pair1 = makePair<int, float>(1, 2.0);
19
20 // implicit specialization
21 Pair<int, float> pair2 = makePair(1, 2.0);
```

Listing 4.1: Example of generics in Slang.

4.2.1 Generics

Slang supports defining generic types and functions, as shown in Listing 4.1. The syntax is similar to Java and C#. Line 2 defines a generic `struct Pair`, which has two generic parameters: `TKey` and `TValue`. `makePair` (line 9) is a generic function with `TKey` and `TValue` as its generic parameters and returns a `Pair`. Slang supports two ways to call a generic function, either with explicit generic type arguments (line 18) or relying on the compiler to deduce the type arguments from argument list (line 31).

One important use of generics is to define shader entry-points that are parameterized on shader components. Listing 4.2 defines a shader entry-point for a typical forward lighting render pass as a generic struct type `ForwardPass`. The struct defines both vertex shader and fragment shader kernel functions (line 10 and line 15), as well as three generic parameters: `TGeometry`, `TMaterial` and `TLight`. These generic parameters stands for three categories of shader component classes that can be used to specialize this entry-point: geometry transform, material, and lighting. Line 6-8 declares the shader parameter for each shader component as a member field. These member field represents references to component instances.

4.2.2 Interfaces as generic constraints

Similar to many other languages, an `interface` in Slang defines the methods that a type needs to implement in order to claim its conformance to the interface. For example, Listing 4.3 defines

```
struct ForwardPass<TGeometry, TMaterial, TLight>
{
    struct InputVertex {...};
    struct VertexOutput {...};

    TGeometry geom;
    TMaterial material;
    TLight light;

    VertexOutput vertexShader(InputVertex vertIn)
    {
        ...
    }

    float4 fragmentShader(VertexOutput vsIn)
    {
        ...
    }
};
```

Listing 4.2: Example of generics in Slang.

```
 1  interface IMaterial
 2  {
 3      float4 getColor();
 4  }
 5
 6  struct SolidColorMaterial : IMaterial
 7  {
 8      float4 getColor()
 9      {
10          return float4(1.0);
11      }
12  };
```

Listing 4.3: Example of defining an `interface` and a concrete type that implements an `interface` in Slang.

a `IMaterial` interface and a type `SolidColorMaterial` that implements the interface with its implementation of the `getColor()` method.

Declaring an interface conformance in the definition of a concrete type (as in line 5 in Listing 4.3) triggers the compiler to check the type definition and verify all required methods are indeed provided in the concrete type. An error message will be generated if the actual type definition does not contain required methods as required by the declared conformance.

In Slang, interfaces are used to define constraints on generic type parameters. Listing 4.4 shows the same entry-point shader as in Listing 4.10, but augmented the global generic parameters with interface conformances declarations. For example, line 17 states that the global generic parameter `TGeometry` must be a type that conforms to the interface `IGeometry`. With this knowledge, the compiler can verify that calling `computeGeometry` method on global variable `geom` (line 27) is valid, since `geom` must have a concrete type that conforms to `IGeometry` interface,

```
interface IGeometry
{
    VertexOutput computeGeometry(InputVertex vin);
}
interface IMaterial
{
   float4 getColor();
}
interface ILight
{
    float4 computeLighting(float4 surfaceColor);
}
struct ForwardPass<TGeometry : IGeometry, TMaterial : IMaterial, TLight : ILight>
{
    struct InputVertex {...};
    struct VertexOutput {...};

    TGeometry geom;
    TMaterial material;
    TLight light;

    VertexOutput vertexShader(InputVertex vertIn)
    {
       return geom.computeGeometry(vertIn);
    }

    float4 fragmentShader(VertexOutput vsIn)
    {
       float4 color = material.getColor();
       return light.computeLighting(color);
    }
};
```

Listing 4.4: Using interfaces as generic constraints.

which does provide `computeGeometry`.

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, which is in contrast to the Go language, where interface conformance is derived by the compiler from the use site of the concrete type. While automatic interface conformance deduction saves programmer's time in writing the declarations, it delays the checking of interface conformance until a use of the concrete type is seen. One of our design principles is to allow the compiler to check as early as possible: 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. 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.

```
interface IGeometry
{
    VertexOutput computeGeometry(InputVertex vin);
}
interface IMaterial
{
    float4 getColor();
}
interface ILight
{
    float4 computeLighting(float4 surfaceColor);
}

struct ForwardPass<TGeometry : IGeometry, TMaterial : IMaterial, TLight : ILight>
{
    struct InputVertex {...};
    struct VertexOutput {...};

    ParameterBlock<TGeometry> geom;
    ParameterBlock<TMaterial> material;
    ParameterBlock<TLight> light;

    VertexOutput vertexShader(InputVertex vertIn)
    {
        return geom.computeGeometry(vertIn);
    }

    float4 fragmentShader(VertexOutput vsIn)
    {
        float4 color = material.getColor();
        return light.computeLighting(color);
    }
};
```

Listing 4.5: Using `ParameterBlock<T>` to specify that the parameters of each shader components comes from a parameter block.

4.2.3 Explicit `ParameterBlock<T>` construct

As discussed in Chapter 3, we need a modular construct in the shading language that maps to a parameter block. HLSL already features a mechanism to represent parameters that comes from a constant buffer: a global variable of type `ConstantBuffer<T>` stands for a reference to a constant buffer resource, whose content is defined by type `T`. The rest of the shader code can treat this global variable as a pointer to `T` and access the data members defined by `T`. This is a good mechanism to specify parameter data communication, except that `ConstantBuffer<T>` works only when `T` contains only ordinary typed parameters and no resource parameters. If `T` contains any field of a resource type, such as `Texture2D`, the field will be separated out into a standalone global variable declaration, and the shading system must still query for the binding indices of these individual resource typed fields to complete parameter communication.

We added `ParameterBlock<T>`, which extends the semantics of `ConstantBuffer<T>` for ordinary parameters to include resource parameters as well. A `ParameterBlock<T>` variable stands for a reference to a parameter block, which may encapsulate both ordinary parameters and resource typed parameters. Listing 4.5 shows the same entry-point shader as in Listing 4.4. The difference is in line 19-21: instead of declaring global variables of type `TGeometry`, `TMaterial` and `TLight`, we wrap these global variables in a `ParameterBlock<T>` type to explicitly specify that these parameters come from parameter blocks.

`ParameterBlock<T>` is an abstract concept that maps to different implementations on different target platforms: on Direct3D 12, it maps to a descriptor table that has one constant buffer entry to a constant buffer that stores all the ordinary typed fields of `T`, and one shader resource view or unordered access view entry for each resource typed field of `T`. On Vulkan, it maps to a descriptor set with similar layout - a uniform buffer entry for all the ordinary type fields, and one entry for each resource typed field. `ParameterBlock<T>` can also be compiled to legacy Direct3D 11 / OpenGL platforms as well. On these platforms, a `ParameterBlock<T>` behaves like a `ContantBuffer<T>` - it encapsulates all ordinary fields in a constant buffer, and separates out each resource typed fields and assign them a global binding index, which can be queried through the introspection API. On all platforms, Slang will not attempt to eliminate unused parameters before assigning binding indices, or removing them from reflection information. This guarantees that the layout of a parameter block is the same across all shader variants that uses the parameter block, so that the shading system can safely reused a parameter block for all shader variants that requires it.

4.2.4 Associated types as interface requirement

Some shading features involve computations that needs to be done in multiple stages. For example, the following code illustrates the invocation of two-stage task:

```
void runTask<Task : ITwoStageTask>(Task t)
{
    int inputData[N];

    // do first stage and store partial results in context
    Context context = t.doStage1();

    int rs = 0;
    // do second stage once for each input
    for (int i = 0; i < N; i++)
        rs += t.doStage2(context, input[i]);
}
```

In this example, `t` represents a task that needs to be done in two stages. The `runTask` function invokes the first stage to perform some initial computation and stores the partial result in `context`. `runTask` then invokes the second stage `doStage2` method for each input element, which reuses

```
1 interface ITwoStageTask
2 {
3     associatedtype Context;
4     Context doStage1();
5     int doStage2(Context ctx, int element);
6 }
```

Listing 4.6: Defining an associated type in a Slang `interface`.

the partial result computed in the first stage. The problem is how to define the `ITwoStageTask` interface?

We can start by defining the `ITwoStageTask` interface as following:

```
interface ITwoStageTask
{
    Context doStage1();
    int doStage2(Context ctx, int element);
}
```

This definition restricts all implementations of `ITwoStageTask` to store its partial results in a concrete type `Context`. However, different implementations may compute different terms for the partial result, thus would return a different type as context. In fact, `runTask` function do not even need to know the concrete type of `context`, it is treating `context` as an opaque object and passing it directly from stage 1 to stage 2.

To enable defining such an interface without restricting all implementations to use a common `Context` type, Slang allows an interface to define *associated type* requirements in addition to method requirements. Listing 4.6 shows the definition of `ITwoStageTask` that uses an associated type. The declaration in line 3 states that in order for a concrete type to satisfy the requirement of `ITwoStageTask`, it must provide a type definition `Context`, such that the `doStage1` method returns `Context` and `doStage2` method takes `Context` as the first argument.

Listing 4.7 shows an actual concrete type implementing the `ITwoStageTask` interface. To satisfy the `associatedtype` requirement, `ATwoStageTask` type defines a nested `struct` type named `Context`. Alternatively, the `associatedtype` requirement can be satisfied via a nested `typedef` clause, as shown in Listing 4.8.

With associated types, we can now use `ITwoStageTask.Context` to refer to the context type of `ITwoStageTask` in `runTask` when declaring the `context` variable:

```
void runTask<Task : ITwoStageTask>(Task t)
{
    int inputData[N];

    // do first stage and store partial results in context
    Task.Context context = t.doStage1();
```

```
struct ATwoStageTask : ITwoStageTask
{
    struct Context
    {
        int x;
    };
    Context doStage1()
    {
        Context rs;
        rs.x = 5;
        return rs;
    }
    int doStage2(Context ctx, int element)
    {
       return element * element + ctx.x;
    }
}
```

Listing 4.7: Implementing associated type requirement in a concrete type.

```
// the context type for ATwoStageTask
struct ATwoStageTaskContext
{
    int x;
};

struct ATwoStageTask : ITwoStageTask
{
    // indicate that we are using ATwoStageTaskContext as the associated Context type.
    typedef ATwoStageTaskContext Context;

    Context doStage1()
    {
        ATwoStageTaskContext rs;
        rs.x = 5;
        return rs;
    }
    int doStage2(Context ctx, int element)
    {
       return element * element + ctx.x;
    }
}
```

Listing 4.8: Implementing associated type requirement in a concrete type.

```
1  interface IMaterial
2  {
3      associatedtype Pattern;
4      Pattern evalPattern(...);
5      float4 evalReflectance(Pattern p, LightSample light, float3 viewDir);
6  };
7
8  float4 evalAppearance<TMaterial : IMaterial>(TMaterial mat, LightSample lights[])
9  {
10     float4 result = 0.0;
11     TMaterial.Pattern pattern = mat.evalPattern(...);
12     for (int i = 0; i < lights.size; i++)
13         result += mat.evalReflectance(mat, lights[i], viewDir);
14     return result;
15 }
```

Listing 4.9: Interface definition for materials. Surface appearance is evaluated in two stages. Stage 1 invokes the `evalSurfacePattern` method and store the surface pattern result in a `Pattern` variable. Stage 2 invokes `evalReflectance` method for each incident light sample, given the surface pattern evaluated in stage 1.

```

    int rs = 0;
    // do second stage once for each input
    for (int i = 0; i < N; i++)
        rs += t.doStage2(context, input[i]);
}
```

A typical example of this two stage pattern is the evaluation of materials. Generally, the appearance of an object is computed in two steps: evaluating the surface pattern (parameters to a BRDF) at a shading location and accumulating the lighting result (by evaluating the BRDF for each incoming light sample). Conceptually, a material defines both the surface pattern (which is inherent to the surface and invariant to lighting environment), and the BRDF (which evaluates reflectance from an incident light sample). Therefore, evaluation of surface appearance is a two-stage process: stage 1 evaluates the surface pattern, and stage 2 evaluates lighting for each light sample, using the same surface pattern as input to the BRDF.

Listing 4.9 shows a possible `IMaterial` interface definition. The `evalAppearance` function (line 8) demonstrates the two-stage process of evaluating surface appearance given a material and an array of light samples. First, it invokes `IMaterial.evalPattern` method to evaluate the surface pattern, whose result is stored in the `pattern` variable. The type of `pattern` is an associated type of `IMaterial`, which stands for a placeholder for a concrete surface pattern type. In the second stage, `evalAppearance` function loops over the array of light samples and call `evalReflectance` for each light sample, reusing the surface pattern evaluated in stage 1 and accumulate lighting result into `result`.

An alternative to associated types for defining multi-stage tasks is to make `ITwoStageTask` a generic interface, and expose the context type as a generic parameter:

```
interface ITwoStageTask<TContext>
{
    TContext doStage1();
    int doStage2(Context ctx, int element);
}
```

A concrete type then implements a specialized `ITwoStageTask` interface:

```
struct ATwoStageTask : ITwoStageTask<ATwoStageTaskContext>
{
    ATwoStageTaskContext doStage1() { ... }
    int doStage2(ATwoStageTaskContext context, int element) { ... }
};
```

Because `ITwoStageTask` is a generic interface, the `runTask` function also need to include a generic parameter for the context type:

```
void runTask<TContext, Task: ITwoStageTask<TContext>>(Task t)
{
    ...
    TContext context = t.doStage1();
    ...
}
```

As we can see, when using generic interfaces, the context type becomes a generic parameter that propagates to the use site of the interface. Furthermore, any use site of `runTask` function would also become a generic function with a additional `TContext` generic parameter. In contrast, the use of associated type results much cleaner code: it contains the type dependency within the interface definition and prevents the declaration of the `TContext` generic parameter to propagate into the use sites of the interface.

4.2.5 Global generic parameters

As described in section 4.2.1, Slang allows defining generic functions and generic `struct` types. To implement the shader components design, a shader entry-point is represented as a generic struct that wraps both the shader parameters and the kernel functions. This is a non-trivial change from how shader are authored in HLSL, where kernel functions and shader parameters are all defined in global scope. To support an incremental adopt path of existing HLSL shader code, Slang allows defining generic parameters for the global scope, as shown in Listing 4.10.

Line 4-6 defines three global generic parameters for the shader program: `TGeometry`, `TMaterial` and `TLight`. For each generic type parameter, we declare a global variable of that type in Line 8-10 to represent the parameters required by each type of shader component. The `genericparam`

```
struct InputVertex {...};
struct VertexOutput {...};

genericparam TGeometry : IGeometry;
genericparam TMaterial : IMaterial;
genericparam TLight : ILight;

ParameterBlock<TGeometry> geom;
ParameterBlock<TMaterial> material;
ParameterBlock<TLight> light;

VertexOutput vertexShader(InputVertex vertIn)
{
   ...
}

float4 fragmentShader(VertexOutput vsIn)
{
   ...
}
```

Listing 4.10: Example of global generics parameters in Slang.

declarations in global scope turns the entire shader program into a generic program that needs to be specialized in order to generate executable shader code.

Global generic parameters can be thought of a syntax sugar to wrapping everything in global scope into a generic `struct`. For example, Listing 4.2 is semantically equivalent to Listing 4.10.

With this feature, developers that wish to adopt the shader components idea can simply add `genericparam` declarations to their shader code and keep most of the existing code structure unchanged. As we will discuss in section 4.3, the shading system can continue to view a shader program as a collection of parameter declarations and kernel function definitions when the code is using global generic parameters, so many existing shading system logic interacting with the shader reflection API can also stay unchanged.

4.3 Introspection API

In this section, we will discuss the design of Slang's shader introspection API, that is used by the shading system to prepare parameter data for CPU-GPU communication and to generate specialized shader variants. We will discuss why this new API design reduces CPU overhead.

Chapter 5

Implementing Shader Components in Slang (Proposed Work)

In this chapter, we present an example shading system that implements the shader component design using Slang. This section breaks down into following subsections:

5.0.1 Authoring modular shading features.

Introduces how the shading features of our example shading system is implemented in Slang as individual modules, and how they are composed together to form a complete shader program.

5.0.2 Creating component instances using the introspection API.

Discusses how the shading system implements component instances at run time and use it as means to specify what shader code to run, as well as the shader parameter values to use.

5.0.3 Composing and selection of shader variants from entry point and components.

Discusses the implementation of a shader variants cache, which is used to store and lookup specialized shader variants with small CPU overhead.

5.0.4 Using de-specialized components.

Discusses how the shading system switches between different decisions of whether to use specialization or dynamic dispatch for certain categories of shading features.

Chapter 6

A Case Study of Adopting Shader Components and Slang (Proposed Work)

In this chapter, we will share our experiences in adopting Slang in a research renderer called *Falcor*. We report our changes made to Falcor as a result of the adoption, and evaluate the performance and modularity properties of the refactored code-base.

This chapter will include following sections.

6.1 Falcor Introduction

Introducing the features and goals of Falcor, and its current implementation of shader specialization and parameter communication.

6.2 The Refactored Shader Library.

Present the decisions we made, and the resulting refactored shader library framework.

6.3 Material Specialization

Discuss how we refacator the material system in Falcor to properly generate specialized shader variants for different configurations of materials.

6.4 Retrospection

Discuss the other findings as a result of this case study. One most notable observation is that we find our refacatored system is producing surprisingly low number of shader variants, which makes massive shader precompilation tooling that once thought as important in improving compilation time irrelavent.

Discuss how Slang makes it easier to implement systems like Bungies's TFX shading system [12].

Chapter 7

Relation to Previous Work (Proposed Work)

7.1 Language Constructs for Shader Modularity

Discuss relationship to modular constructs in prior shading language work, such as Cg interfaces [10], HLSL classes [7], GLSL subroutines [5].

7.2 Relationship to alternate rate-based solutions

We published several papers on shading languages that uses a rate-based type system [1, 2, 3]. We discuss why we've gone a different direction for Slang. We prove that the most important features of a rate-based type system has its correspondence in Slang. In some places it is not as elegant as in a rate-based type system, but the language mechanisms in Slang is more general and accessible to developers who are not aware of rate-based programming. We discuss why we dropped the exploration of choices as introduced by He et al. [3] - they do not scale with modularity, and most interesting choices lies in the entry point shader which is not hard to write.

Chapter 8

Timeline

December 2017 Finish implementation of key Slang language features.

January 2018 Complete the case study of integrating Slang into Falcor.

February 2018 Implement an example shading system to demonstrate the shader components design using Slang.

March 2018 - May 2018 Fix compiler issues and implement any remaining features that are found to be useful during evaluation.

March 2018 - June 2018 Finish thesis writing and defense.

Bibliography

[1] Tim Foley and Pat Hanrahan. Spark: Modular, composable shaders for graphics hardware. *ACM Trans. Graph.*, 30(4):107:1–107:12, July 2011. 7.2

[2] Yong He, Tim Foley, Natalya Tatarchuk, and Kayvon Fatahalian. A system for rapid, automatic shader level-of-detail. *ACM Trans. Graph.*, 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`. 7.2

[3] Yong He, Tim Foley, and Kayvon Fatahalian. A system for rapid exploration of shader optimization choices. *ACM Trans. Graph.*, 35(4):112:1–112:12, July 2016. 7.2

[4] 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, 2.1.1

[5] Khronos Group, Inc. ARB_shader_subroutine. `https://www.opengl.org/registry/specs/ARB/shader_subroutine.txt`, 2009. 7.1

[6] Khronos Group, Inc. *Vulkan 1.0.38 Specification*, 2016. 2.1

[7] Microsoft. Interfaces and classes. `https://msdn.microsoft.com/en-us/library/windows/desktop/ff471421.aspx`, 2011. 7.1

[8] Microsoft. HLSL shader model 5 documentation. `https://msdn.microsoft.com`, 2016. 1, 2.1.1

[9] Microsoft. Direct3D 12 programming guide. `https://msdn.microsoft.com/en-us/library/windows/desktop/dn899121(v=vs.85).aspx`, 2017. 2.1

[10] Matt Pharr. An introduction to shader interfaces. In Randima Fernando, editor, *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004. ISBN 0321228324. 7.1

[11] Mark Segal and Kurt Akeley. 2.1

[12] Natalya Tatarchuk and Chris Tchou. Destiny shader pipeline. 6.4