SPARK: MODULAR, COMPOSABLE SHADERS FOR GRAPHICS HARDWARE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Timothy John Foley
June 2012

This dissertation is online at: http://purl.stanford.edu/wz483vv5440

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Patrick Hanrahan, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Alex Aiken

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Kurt Akeley

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Real-time computer graphics have become a ubiquitous part of modern life. Rich user interfaces and interactive games appear on screens ranging from mobile phones to stereo-3D TVs. Programmable graphics *pipelines* have been central to delivering these compelling experiences. *Shaders*—programs that describe the shape, movement, and appearance of rendered objects—run in the stages of these pipelines, and are used to define the "look and feel" of a production. The demand for rich, immersive experiences motivates the use of increasingly complex shaders.

In creating these complex real-time shaders, programmers should ideally be able to decompose code into independent, localized modules of their choosing. Current real-time shading languages, however, enforce a fixed decomposition into per-pipeline-stage procedures. Program concerns at other scales—including those that *cross-cut* multiple pipeline stages—cannot be expressed as reusable modules.

We present a shading language, Spark, that improves support for separation of concerns into modules. A Spark *shader class* can encapsulate a program feature that maps to more than one pipeline stage, and can be extended and composed using object-oriented inheritance.

We first discuss the design of this language: its origins, goals, and key design choices. We then describe our implementation of a compiler, standard library, and runtime system for Spark, targeting current programmable graphics hardware. Finally, we evaluate this implementation and demonstrate that it achieves our modularity goals without compromising performance: in our tests, shaders written in Spark achieve performance within 2% of Microsoft's High Level Shading Language (HLSL).

Acknowledgements

A great many people have provided opportunities and guided me to find this path; I am indebted to them all.

The impact that Pat Hanrahan has had on my academic and professional development cannot be overstated. He has changed the way I read, the way I write, and the way I *think* irrevocably and for the better. His support and guidance over these many years has gone beyond what I could have expected.

I also wish to thank Kurt Akeley, Alex Aiken, Vladlen Koltun and Juan Alonso for serving on my orals committee. Kurt's insightful questions and suggestions have greatly influenced my understanding, and subsequently the presentation in this dissertation. If I have achieved clarity, then he deserves much credit.

The Spark project would not have come to fruition without the input of many academic and professional colleagues. The genesis for the Spark system can be traced back to unpublished work done with Paul Lalonde at Neoptica. Together with Pat and Kurt, Bill Mark, Solomon Boulos and Henry Moreton were instrumental in the multiple rounds of presentations it took before the ideas crystalized into something explicable. Bill has contributed important ideas and understanding to the Spark system design, and improved my understanding of the context of prior work. Matt Pharr helped me to plan and effect large-scale organizational changes while this dissertation was in its early stages. Rahul Sathe generously gave his time as a user of the Spark system. Financial support for Spark was provided by Intel Corporation, the Stanford Pervasive Parallelism Laboratory, and the Gigascale Systems Research Center, one

of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. It is thanks to the generous support of Aaron Lefohn and Charles Lingle at Intel that I have been able return to Stanford and finish this dissertation.

My early education benefitted greatly from the influence of Richard Klier and Janene Scovel, who encouraged me to pursue my interests and learn in my own way (and looked the other way on a lot of missed assignments). As an undergraduate I appreciated the opportunity to work with and learn from Kent Wilson, Bill Griswold, and Geoff Voelker. In my career as a graduate student, Ian Buck, Jeremy Sugerman, Jonathan Ragan-Kelley, and Kayvon Fatahalian have become role models. The opportunities provided by Nick Triantos, Matt Pharr, and Craig Kolb have shaped the course of my career.

Finally, my heartfelt thanks go to my family. My parents fostered my interest in computers from the beginning: they provided me my first experience with Logo and HyperCard; they got me Basic and C/C++ compilers as Christmas presents; my mother introduced me to the Internet back when Gopher was still relevant. My wife, Erin, has been supportive of me ever since we met as teenagers, and has shown great patience over the years it has taken me to reach this point. Thank you.

To my parents, for raising me.
To my wife, for supporting me.
To my children, for loving me.

Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

The reach of real-time computer graphics has never been greater. Full-featured computing devices, with high-resolution displays and rich graphical capabilities, are now ubiquitous. Many of the most popular applications for these devices are real-time interactive games. Users can immerse themselves in 3D worlds on computers, dedicated game consoles, tablets, and phones. For developers, creating the kinds of rich, interactive graphics that users desire presents a dilemma: striking a balance between *generality* and *performance*.

Unlike the director of a computer-generated movie—who precisely controls the placement of their virtual lights, camera, and actors—the creators of a game must deal with one highly unpredictable element: the human player. A game must be ready to render a scene from any conceivable camera angle, and under any lighting conditions that the simulation allows. This requires highly *general* solutions, so that we can flexibly combine lights, surfaces, and materials as required.

Often in computer programming, we achieve flexibility by trading off *performance*. In order to maintain the illusion of continuous motion, however, a real-time rendering application must produce a complete rendered frame within 15 to 32 milliseconds.

Our work seeks to bring increased flexibility and generality to real-time rendering, while preserving the high performance that developers require.

1.1 Real-Time Rendering Architectures

Increasingly, high performance in computer applications is achieved through *parallelism*: by doing many things at once, the overall *throughput* can be increased. Implementations of real-time rendering architectures in *graphics hardware* chiefly rely on two kinds of parallelism:

- **Data Parallelism** A 3D model is composed of a multitude of vertices, and might cover many hundreds of on-screen pixels. Each of the vertices/pixels will require similar treatment, and by processing them together in batches, we can amortize out certain fixed costs and overheads.

- **Pipeline Parallelism** When rendering a surface, we must first determine the positions of its vertices, before we can compute colors for the pixels it covers. By processing these pixels in parallel with the vertices of other surfaces, we can render multiple surfaces with higher throughput than a single surface.

Most contemporary rendering architectures take the form of a *pipeline* with many distinct *stages*. This organization allows implementations to exploit significant pipeline parallelism. The stages communicate data on streams, in the form of *records*: individual vertices, fragments, control points, etc. Each stage consumes records produced by "upstream" stages, and produces new records for use by "downstream" stages. The overall pipeline *dataflow*—the connectivity of stages and streams, and type of records on each stream—is typically fixed.

Some stages perform *fixed-function* operations using dedicated hardware: for example, the assembly of transformed vertices into triangles or other primitives. An application may change some parameters of these stages—e.g., switching between triangle and quadrilateral primitives—but cannot otherwise override their behavior. In contrast, *programmable* stages are controlled by application-provided programs called shaders.

1.2 Shaders

A *shader* is a unit of application code that describes the appearance of rendered objects—shape, transformation, animation, color, etc.—and that runs in the context of a rendering system. A shader might compute transformed positions of vertices to simulate the movement of a walking character, or the color reflected by the character's skin under the influence of several light sources.

The code in shaders is often executed in a data-parallel fashion: e.g., on batches of vertex or fragment records. For this purpose, graphics hardware will typically have many general-purpose processing cores, and utilize techniques such as SIMD (single instruction, multiple data) execution. These cores allow an application's shaders to perform almost any computation, although the available input/output operations are typically constrained to those supported by the pipeline dataflow. In this way, a real-time rendering architecture can be seen as a *framework*, that is customized by the application's shaders.

By constraining the capabilities of shader programs in this fashion, interfaces to rendering architectures such as OpenGL [SAF+10] and Direct3D [Bly06, Mic10a] have traditionally been able to take advantage of data- and pipeline-parallel execution without exposing the complexity of parallel programming to user applications. For example, when a shader is computing the position of one vertex, it cannot "see" other vertices, and is thus oblivious to the order in which vertices are processed.

Having established the context in which shaders execute, we turn our attention to the challenge of authoring shaders for compelling real-time graphical effects.

1.3 The Challenge

> Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing.
>
> *On the Criteria to be Used in Decomposing Systems into Modules*
> David Parnas

In the first generation of programmable graphics hardware, shaders typically comprised tens of lines of code, targeting two programmable stages in a primarily fixed-function pipeline. Increasing hardware capabilities have seen an increase in both the number and complexity of programmable pipeline stages. For example, the Direct3D 11 architecture (hereafter D3D11) exposes a pipeline with five programmable stages. Achieving a particular effect requires coordination of shader code running in these stages, fixed-function hardware settings, and application code.

In light of the increasing scope and complexity of this programming task, we argue that the time is right to re-evaluate the design criteria for real-time shading languages. A modern shading language should support good software engineering practices, so that diligent programmers can create maintainable code. Our work focuses on the particular problem of *separation of concerns*: the factoring of logically distinct program features into localized and independent modules.

Separation of Concerns

Figure 1.1 shows a complex rendering effect that uses every programmable stage of the D3D11 pipeline. In a single pass through the rendering pipeline, an animated, tessellated, and displaced model is rendered simultaneously to all six faces of a cube map. In this visualization, the dashed boxes represent the five programmable stages of the D3D11 pipeline. The colored boxes represent logically distinct features or *concerns* in the program. Some concerns in this figure, such as tessellation, intersect multiple stages of the rendering pipeline. These are *cross-cutting concerns* in the terminology of aspect-oriented programming [KLM+97].

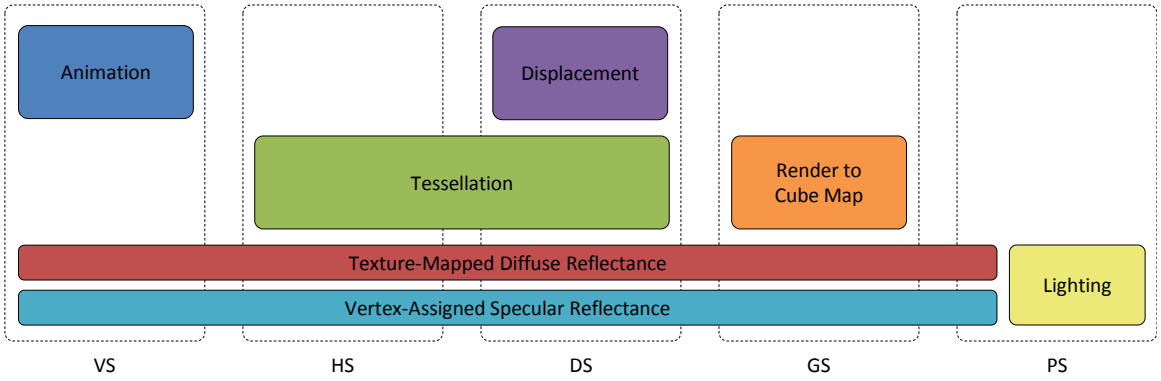

Figure 1.1: *A complex shading effect decomposed into user-defined modules in Spark. The dashed boxes show the programmable stages of the Direct3D 11 pipeline: the Vertex, Hull, Domain, Geometry, and Pixel Shader. The colored boxes show different* concerns *in the program. Some logical concerns* cross-cut *multiple pipeline stages.*

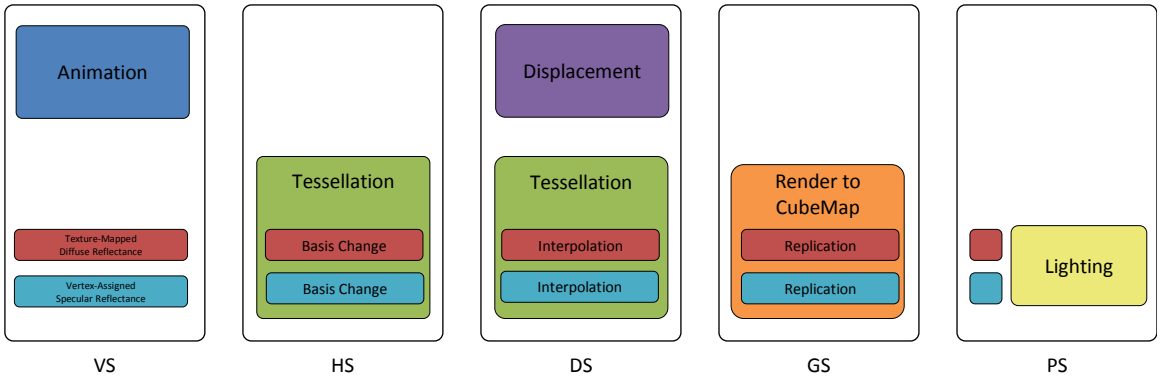

Figure 1.2: *The rendering effect in Figure 1.1, adapted to the constraints of shader-per-stage languages. The colored boxes represent subsets of the program pertaining to different features. Some logically coherent program features must be divided, and some orthogonal ones merged, to meet the constraints of shader-per-stage programming.*

Ideally, we would desire that a shading language would allow each logical concern to be defined as a separate, reusable module. In this way we could re-use existing modules to create new rendering passes. For example, we might want to create a shading effect like that in Figure 1.1, only without the displacement effect.

Modularity and reusability are increasingly important as more complex algorithms are expressed in shader code. For example, tessellation of approximate subdivision surfaces on the D3D11 pipeline requires a non-trivial programming effort: several weeks for a capable graphics programmer, from our experience. A programmer should be able to expend that effort once, and then re-use the resulting module many times. This, then, is a key goal of our work.

Several factors make achieving this goal challenging: most importantly, we identify challenges arising from *groupwise* shading code and *plumbing*.

Pointwise and Groupwise Code

Modern shaders comprise two kinds of code, which we will call *pointwise* and *groupwise*. The earliest programmable rendering architectures expose vertex and fragment processing with a simple mental model: a user-defined function is mapped over a stream of input records, one at a time. This ensures that individual vertex and fragment records may be processed independently (or in parallel), and so shading algorithms are defined pointwise: that is, independent of any particular record(s).

Instead of records, pointwise shading code is defined in terms of *attributes*: e.g., per-vertex or per-fragment points, vectors, colors, texture coordinates, etc. For example, pointwise shading code might compute the product of an interpolated per-vertex color and a per-fragment color, without making explicit reference to the particular vertices and fragment involved.

In contrast, groupwise operations, such as primitive assembly or rasterization, explicitly apply to an aggregation of records: e.g., all of the vertices that comprise

a primitive. Where historically groupwise operations have been enshrined in fixed-function stages, current real-time rendering pipelines such as D3D11 [Mic10a] and OpenGL 4 [SAF+10] include user-programmable stages that can perform groupwise operations: e.g., basis change, interpolation, and geometry synthesis.

Per-Stage Shaders

Today, the most widely used GPU shading languages are HLSL [Mic02], GLSL [KBR03], and Cg [MGAK03]. These are *shader-per-stage* languages: a user configures the rendering architecture with one shader procedure for each programmable stage of the pipeline. Figure 1.2 shows a possible mapping of the effect in Figure 1.1 to a shader-per-stage language. To meet the constraints of the programming model, cross-cutting concerns have been decomposed across multiple per-stage procedures.

More importantly, some pointwise and groupwise concerns are *coupled* in Figure 1.2. This coupling is driven by the need for *plumbing* code.

Plumbing

Each per-vertex attribute that is subsequently used in per-fragment computations requires code to *plumb* it through intermediate programmable stages. Figure 1.2 shows how this can create coupling. When tessellating a coarse mesh into a fine mesh, we must interpolate the values of each attribute for each new vertex; this plumbing code couples the implementation of the tessellation and texture-mapping concerns. The amount of plumbing code required increases with the number of attributes, and with the number of stages in the pipeline.

Plumbing code leads to coupling because the specification of *what* and *how* to plumb belong to logically separate concerns. A concern might introduce a new attribute in pointwise shading code (e.g., per-vertex color) that needs to be plumbed through the pipeline, but should be specified independently from the details of, e.g., tessellation.

Conversely, a tessellation effect determines the overall scheme for plumbing—basis change and interpolation—of attributes, but ought to be independent of the particular attributes that require interpolation.

In the search for a solution to this problem, we turn our attention to one of the earliest shading languages for real-time graphics hardware.

Pipeline Shaders

In contrast to shader-per-stage languages, a *pipeline-shader* language allows a single shader to target a programmable pipeline in its entirety. The idea of pipeline shaders originates in the Stanford Real-Time Shading Language (RTSL) [PMTH01]. The RTSL system uses pipeline shaders to generate high-performance code for the earliest generation of programmable graphics hardware. Despite some compelling results, the pipeline-shader approach has not seen broad adoption in industry. As such, the approach has not been updated to accommodate new rendering architectures with additional programmable pipeline stages.

In our work, we set out to explore whether a pipeline-shader language might yield better tools for separation of concerns on current real-time rendering architectures. To that end, we sought to take the key ideas of RTSL, and extend them to support the capabilities of modern pipelines: most notably dynamic control flow and user-defined groupwise operations.

Contribution

In this dissertation, we describe a shading language, Spark, and its implementation as a compiler and standard library targeting the D3D11 pipeline. Spark better supports separation of concerns than current shader-per-stage languages, allowing both point-wise and groupwise shading code to be factored into reusable modules. For example, Spark allows us to achieve the modularization depicted in Figure 1.1.

A Spark programmer may define independent *shader classes*, each encapsulating a logical concern—even those that cut across the inter-stage boundaries of a rendering pipeline. Shader classes can be extended and composed, using techniques from object-oriented programming.

When composing groupwise and pointwise shading code, our Spark compiler can automatically synthesize plumbing code by instantiating user-defined *plumbing operators*, thus avoiding coupling. In order to achieve good performance, we perform global (that is, inter-stage) optimization on composite shaders. In our experience with the system, we have found that Spark can achieve better separation of concerns than existing shading languages for effects like tessellation, while still achieving performance within 2% of HLSL shaders.

1.4 A Motivating Example

To order to illustrate the kind of benefits that can come from our solution, we will present a brief motivating example. Listing 1.1 gives a minimal shading effect in HLSL, which performs basic diffuse lighting under a single directional light source. This code defines per-stage shader procedures `VS()` and `PS()`, along with idiomatic *connector* structures to represent the input and output records of these procedures.

Listing 1.2 shows how the HLSL code in Listing 1.1 might be extended to support an additional feature: fetching diffuse reflectance from a texture map. The lines of code that were added or modified to support this feature are highlighted in red. Note how code has been changed or modified in both of the per-stage entry points, as well as in connector structures. This highlights the deficiencies of current shading languages:

- **Modularity** The implementation of the texture-mapping feature is spread throughout the code-base, rather than isolated in its own module.
- **Reuse** A programmer cannot use both the extended effect in Listing 1.2 and the unmodified effect in Listing 1.1 without resorting to "copy-paste programming."

```
cbuffer Uniform {
    float4x4 modelView;
    float4x4 modelViewProj;
    float3   L_world;
};

struct AssembledVertex {
    float3 P_model : Position;
    float3 N_model : Normal;
};

struct CoarseVertex {
    float3 N_world : Normal;
    float4 P_proj  : SV_Position;
};

struct Fragment {
    float4 color : SV_Target;
};

CoarseVertex VS( AssembledVertex input ) {
    CoarseVertex output;
    output.N_world = mul(input.N_model, float3x3(modelView));
    output.P_proj  = mul(float4(input.P_model, 1),
                         modelViewProj);
    return output;
}

Fragment PS( CoarseVertex input ) {
    Fragment output;
    float4 diffuse = 1;
    float  NdotL = dot(L_world, normalize(input.N_world));
    output.color = diffuse * saturate(NdotL);
    return output;
}
```

Listing 1.1: *A minimal HLSL shading effect. Positions and normals are transformed per-vertex, and simple diffuse lighting is computed per-fragment.*

```
cbuffer Uniform {
    float4x4 modelView;
    float4x4 modelViewProj;
    float3   L_world;
};
Texture2D<float4> diffuseTexture;
SamplerState      linearSampler;

struct AssembledVertex {
    float3 P_model : Position;
    float3 N_model : Normal;
    float3 uv      : TexCoord;
};

struct CoarseVertex {
    float3 N_world : Normal;
    float4 P_proj  : SV_Position;
    float3 uv      : TexCoord;
};

struct Fragment {
    float4 color : SV_Target;
};

CoarseVertex VS( AssembledVertex input ) {
    CoarseVertex output;
    output.N_world = mul(input.N_model, float3x3(modelView));
    output.P_proj  = mul(float4(input.P_model, 1),
                         modelViewProj);
    output.uv = input.uv;
    return output;
}

Fragment PS( CoarseVertex input ) {
    Fragment output;
    float4 diffuse = diffuseTexture.Sample(linearSampler,
                                           input.uv);
    float  NdotL = dot(L_world, normalize(input.N_world));
    output.color = diffuse * saturate(NdotL);
    return output;
}
```

Listing 1.2: *Extended HLSL shading effect. The code that has been added or modified to support texture mapping is highlighted in red.*

- **Plumbing** The programmer must write code in `VS()` to plumb texture coordinates through the vertex-processing stage, even though no per-vertex processing of texture coordinates is required.

Listing 1.3 shows Spark code implementing an effect equivalent to the HLSL in Listing 1.2. We do not expect that readers will fully understand the code at this point, but instead call attention to the ways in which it addresses the problems in the HLSL equivalent:

- **Modularity** The implementation of the texture-mapping feature is localized in a single `shader class` declaration.
- **Reuse** Both the original `Diffuse` effect and the extended `Texturing` effect are available for use. Copy-paste programming is not required.
- **Plumbing** No additional code is required to plumb the texture coordinate `uv` through the pipeline.

We will revisit this example in future chapters, as we explain the features of the Spark language and system.

1.5 Dissertation Road Map

In the remainder of this dissertation we discuss the Spark system in greater detail:

Chapter 2 provides background material that will facilitate understanding of our design. Our work draws not only on the history and state of the art in real-time shader languages, but also on several topics in the broader field of programming languages. We leverage ideas from several prior works to build a suite of language-design tools.

Chapter 3 discusses the design of the Spark language. We begin by documenting the design goals we sought to achieve, introduce the fundamental pipeline-programming

```
shader class Diffuse extends D3D11DrawPass
{
    input @Uniform float4x4 modelView;
    input @Uniform float4x4 modelViewProj;
    input @Uniform float3   L_world;

    input @Uniform VertexStream[float3] P_stream;
    input @Uniform VertexStream[float3] N_stream;

    @AssembledVertex float3 P_model = P_stream(IA_VertexID);
    @AssembledVertex float3 N_model = N_stream(IA_VertexID);

    @CoarseVertex float3 N_model =
        mul(N_model, float3x3(modelView), N_model);
    override RS_Position =
        mul(float4(P_model, 1.0f), modelViewProj);

    virtual @Fragment float4 diffuse = float4(1.0f);
    @Fragment float NdotL = dot(L_world, normalize(N_world));
    @Fragment float4 C = diffuse * saturate(NdotL);

    output @Pixel float4 target = C;
}

shader class Texturing extends Diffuse
{
    input @Uniform Texture2D[float4] diffuseTexture;
    input @Uniform SamplerState linearSampler;
    input @Uniform VertexStream[float2] uvStream;

    @AssembledVertex float2 uv = uvStream( IA_VertexID );

    override diffuse = Sample(diffuseTexture,
                              linearSampler,
                              uv);
}
```

Listing 1.3: *Spark shader code corresponding to the HLSL in Listing 1.2.*

abstraction that underlies our work, and then discuss several key design decisions that we made, explaining why we made the choices we did. This chapter ends with a set of brief example programs in Spark, demonstrating the key features of the language.

Chapter 4 discusses our implementation of the Spark compiler and runtime system. We present results from two suites of shaders, focused on illumination and geometry, respectively. Our results compare Spark and HLSL both on their ability to achieve separation of concerns, and on performance.

Finally, **Chapter 5** discusses forward-looking directions for our research. This includes connections between the Spark language and more formal work on Programming Language Theory (PLT), possible directions for improving the usability of the Spark language and system, and opportunities to evolve rendering architectures to better support our approach to shader programming.

Chapter 2

Background

In order to illuminate our design, we believe it is important to both situate Spark in the particular history of shading langauges, *and* in the broader context of programming languages in general. As such, this chapter covers a broad range of topics: shading languages, interfaces to graphics hardware, programming languages, and software engineering. As much as possible, we make note of how this background material informs our discussion in future chapters.

2.1 Real-Time Shading Languages

In this section, we present a brief history of real-time shading languages, with a focus on those works that mostly directly inform or illuminate our own design.

Shading langauges are examples of *domain-specific languages* (DSLs). While many DSLs are primarily concerned with accessibility or ease of use, shading languages are an important example of a *performance-oriented* DSL.

In general, shading languages support parallel execution, although parallelism is *implicit* rather than *explicit*. Typically multiple "instances" of a shader program are

run in parallel over multiple vertices, fragments, or shading samples. Instances are unable to communicate with one another, so the use of parallel execution is entirely transparent to programmers.

2.1.1 Declarative and Procedural Shaders

Modern shading languages derive from the early work on Cook's shade trees [Coo84] and Perlin's image synthesizer [Per85]. It is telling, then, to note that these two works differ on a crucial language design decision: Cook's language is declarative, while Perlin's is procedural.

Shade trees represent a shader in terms of its dataflow graph. Surface, light, atmosphere, and displacement shaders may be specified as separate, modular graphs. The rendering system then composes these modules by "grafting" one shade tree onto another. This notion of grafting motivates our support for composition of shaders in Spark (see Section 3.3.2).

In the shade tree system, shader graphs are authored using a declarative "little language." The language looks superficially similar to C, but lacks features such as control flow and mutable variables. These restrictions ensure that any program in the language can be represented as an equivalent dataflow graph, but in turn limit the kinds of programs that can be expressed.

In Perlin's image synthesizer, shaders are *procedures*: sequences of imperative statements. A shader procedure may perform almost arbitrary operations, including looping and conditional control flow, to compute its result. Support for modularity, however, is limited to procedural abstraction: simpler procedures may be used to define more complex ones. In particular, the image synthesizer does not support the modular specification and composition of surface and light shaders. Ultimately, the entire shading process for a pixel must be described by a single procedure.

This is a classic tradeoff: a procedural representation gives more power to the user (it can express any computation), but as a consequence a shader is effectively a black box to the rendering system. In contrast, a declarative, graph-based representation exposes more structure to the implementation, and is thus more amenable to analysis and transformation (e.g., Cook's grafting operation).

2.1.2 RenderMan Shading Language

Hanrahan and Lawson [HL90] describe the RenderMan Shading Language (RSL) as incorporating features of both Cook's and Perlin's work. RSL is a procedural language, but still separates the definition of surfaces and lights. In place of shade-tree style grafting, the interface between shaders is provided by specialized control-flow constructs (e.g., `illuminance` loops and `illuminate` statements).

RSL introduces two ideas that are relevant to our design. First is the idea of treating a shader program as an object-oriented class, from which shader objects are instantiated at run-time. Representing shaders as classes allows the application and rendering system to reason about and control the lifetime of shader instances, and in particular when (and how often) expensive operations like specialization and optimization are performed.

Second is the introduction of *rates of computation* in the form of the `uniform` and `varying` qualifiers. RSL allows a single shader to include computations at two different rates: per-batch and per-sample. For example, a vector representing the direction of a distant light source might be declared `uniform`, since the direction remains constant for every shading sample, while the diffuse reflectance of a surface might vary continuously and be declared `varying`. The expectation of the user is that computations involving only `uniform` values occur at a lower rate—that is, less often—and may thus be less costly.

2.1.3 Real-Time Shading Language

The Stanford Real-Time Shading Language (RTSL) [PMTH01] extends the concept of uniform and varying computation to a richer set of rate qualifiers, including `vertex` and `fragment`. These qualifiers allow a single pipeline shader to target both the vertex and fragment processors on early programmable GPUs, as well as a host CPU. Our Spark language further extends the notion of rates of computation, by making the set of rate qualifiers *extensible* (see Section 3.3.3).

RTSL is syntactically quite similar to RSL, and superficially looks like a procedural language. The language is, however, declarative: similarly to shade trees, sufficient restrictions are placed on RTSL shaders, such that they can be represented as DAGs. Like shade trees, RTSL allows graphs representing surface and light shaders to be defined separately and composed by the rendering system.

In order to generate code for a programmable graphics pipeline, the RTSL system takes the composed shader graphs and *partitions* them into sub-graphs according to computation rates. These sub-graphs are then used to generate (procedural) code for particular programmable pipeline stages. For example, all of the computations in the graph with the `vertex` rate are collected to generate an executable shader procedure for the vertex-processing pipeline stage. Edges that extend into or out of this per-vertex sub-graph correspond to the inputs and outputs of this procedure. This compilation strategy relies on an abstract pipeline programming model in which the rates in the RTSL language correspond to the programmable stages of the rendering pipeline. The approach to compilation in RTSL guides our implementation in Spark (see Section 4.1.3), but we rely on a more refined pipeline programming model described in Section 3.2.

The DAG representation in RTSL cannot express data-dependent control flow. This restriction is a good match for early programmable GPUs, which do not support data-dependent control flow in vertex or fragment processors. Similarly, an RTSL shader can express only pointwise shading operations (see Section 1.3), as the only groupwise operations on early GPUs are performed by fixed-function hardware.

One important decision in RTSL is that results computed at a low rate of computation can be implicitly converted to any higher rate, even if this might involve plumbing values through the pipeline. For example, a per-vertex color may be used in per-fragment computations; the RTSL compiler automatically exploits the rasterizer to perform interpolation and plumb the data through. This design choice motivates our own decision to support automatic plumbing in Spark (see Section 3.1.1).

The SMASH API [McC00] supports rate-qualified *sub-shaders*, with a DAG-based representation similar to RTSL. Renaissance [AR05] combines ideas from RTSL with a purely functional programming language in the style of Haskell. Data-dependent control flow is supported through higher-order functions like `sum`. Like RTSL, neither SMASH nor Renaissance supports the creation of user-defined groupwise operations.

2.1.4 Cg, HLSL, GLSL

Cg, HLSL, and GLSL share a common history and many design goals; we focus on the design of Cg as given by Mark et al. [MGAK03]. Cg consciously eschews any domain-specific factorization of shading into surface and light shaders, in favor of a general-purpose C-like procedural language. This decision means that Cg can express almost any algorithm, and is constrained only by hardware capabilities rather than any particular domain model.

For our discussion, the most important decision made in Cg was the choice of a shader-per-stage approach, rather than RTSL-like pipeline shaders. Several motivations are given for this decision. For example, with appropriate factoring of shader code—e.g., by putting all geometric computations in vertex shaders and all material and lighting computations in fragment shaders—an application might reuse a single vertex shader across a variety of materials.

Mark et al. further observe a problematic interaction between data-dependent control flow and RTSL's rate qualifiers. For example, it is unclear what semantics, if any, could be ascribed to a pipeline shader that modifies a per-vertex variable inside a

per-fragment loop (or vice versa). Mark et al. comment that auxiliary language rules could be used to ban such problematic cases, but the resulting programming model might be "unreasonably confusing." Resolving this apparent incompatibility between rate qualifiers and control flow was one of the most important, albeit simple, design decisions we made for Spark (see Section 3.3.1).

As the pipelines exposed by rendering architectures have grown in complexity, the shader-per-stage approach has been adapted to support new programmable pipeline stages, along with programmable groupwise operations. The use of explicit connector structures to represent records (see Section 1.4) allows these languages to directly express a per-stage procedure that, e.g., consumes an array of vertex records.

2.1.5 Shader Metaprogramming

Programmers often layer more flexible abstractions on shader-per-stage languages by metaprogramming. One common approach among game developers is to write an *über-shader*—a shader implementing the sum of all desired features—which may be pre-processed to strip out unused features and generate specialized shaders on demand. Other systems work by "stiching" together shaders from a library of code fragments. Effect systems [NVI10, Mic10b, LS02] allow a set of per-stage shaders to be encapsulated and parameterized as a single unit, but do not address program concerns at other scales. Shader metaprogramming frameworks [MQP02, LO04, KW09] apply host-language abstractions to shaders, and are examples of *embedded* domain-specific languages (EDSLs).

Pixel Bender 3D [Ado11] separates shader code into transformation and material concerns. A material shader uses separate procedural entry points to target vertex- and fragment-processing stages. The system does not target other programmable stages, nor allow shaders to be decomposed into arbitrary user-defined concerns.

The Vertigo system [Ell04] provides an EDSL in Haskell for authoring vertex shaders as pure functions. Vertigo leverages the inherent composability of pure functions to allow concise definitions of parametric surfaces (e.g., surfaces of revolution).

The GPipe system [Bex] allows shader programs to abstract over streams of vertices and fragments. In GPipe the type constructor `Fragment` acts as a *functor* and operations on, e.g., a `Fragment Float` are automatically *lifted* to happen per-fragment. We will see in Section 5.1.4 that a GPipe `Fragment Float` is quite similar to a Spark `@Fragment float`.

2.2 Interfaces to Graphics Hardware

Historically, the only function of graphics hardware was to support rendering architectures like Direct3D or OpenGL. Modern graphics hardware platforms, however, support both rendering and "compute" interfaces. In this section we discuss contemporary interfaces of both types.

2.2.1 The Direct3D 11 Rendering Pipeline

Figure 2.1 depicts the nominal structure of the D3D11 rendering pipeline Each pipeline stage is represented as a box, and user-programmable stages are shown in gray. We say this structure is "nominal" because in practice there are additional fixed-function stages not named in the D3D11 architecture.

We will briefly discuss the names and responsibilities of the pipeline stages, for the benefit of readers who may be unfamiliar with D3D11. Several stages produce vertex records of one type or another; to aid in disambiguation, we depict and name the type(s) or records produced at each stage on the right-hand side of Figure 2.1. We will continue to use this terminology (which is not standard to D3D11) throughout the dissertation.

The Input Assembler (IA) gathers attributes (such as positions, normals, or colors) from buffers in memory to create *assembled vertices*. The Vertex Shader (VS) maps an application-provided shader procedure over the assembled vertices to produce a stream of *coarse vertices*, representing a base mesh. Traditionally, shader code in D3D11 is authored as per-stage procedures in the HLSL language.

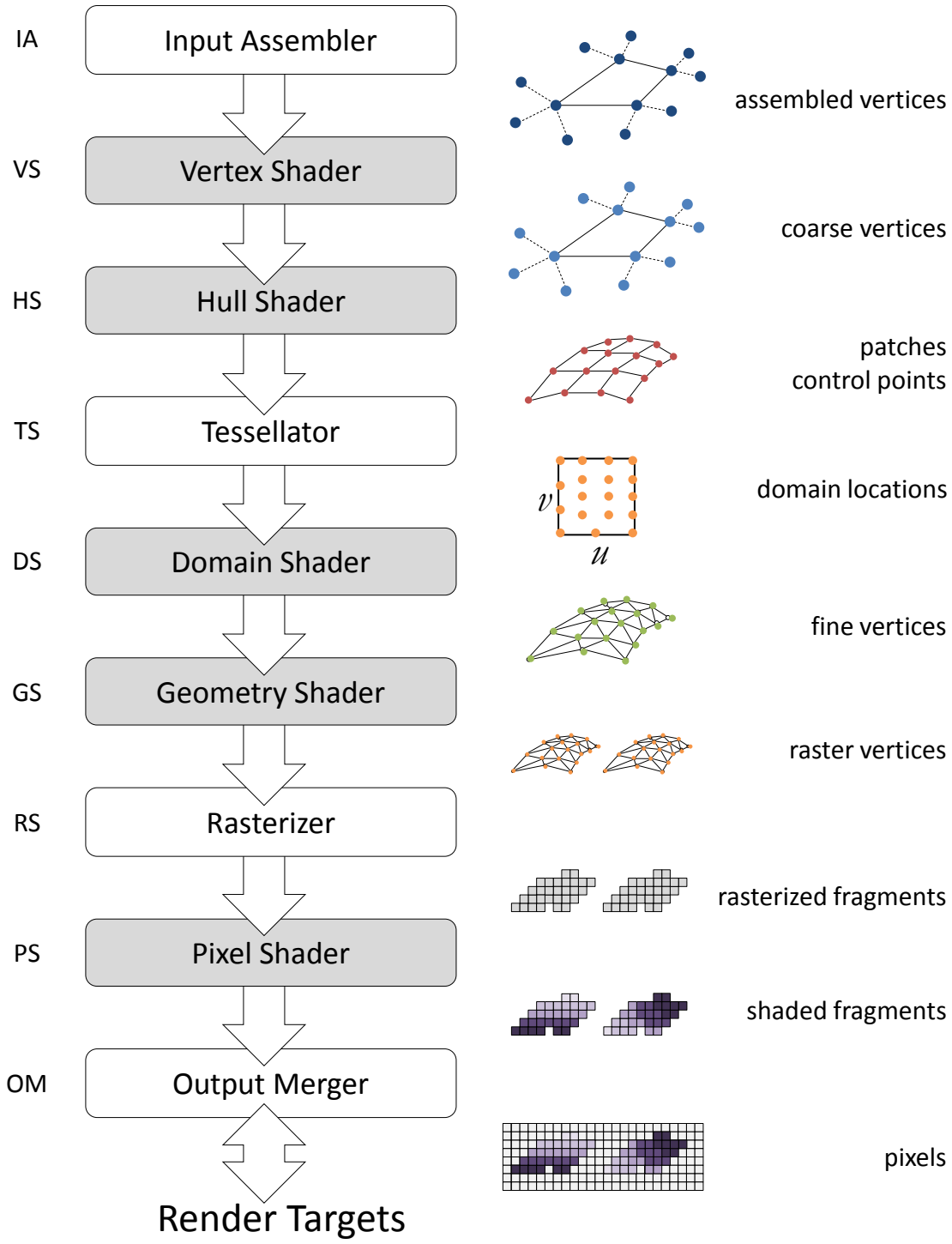

Figure 2.1: *Nominal structure of the Direct3D 11 rendering pipeline. Programmable stages are shown in gray. In practice, fixed-function primitive-assembly stages precede the HS and GS, but these are not included in the nominal pipeline.*

The coarse vertices produced by the VS are assembled into primitives by an unnamed fixed-function stage before being processed by the Hull Shader (HS). The HS can perform a basis transformation: e.g., it may convert each face of an input subdivision-surface mesh into control points for a bicubic Bézier patch. The HS stage makes use of two shader procedures: one to compute *control point* records, and another to compute the attributes for each *patch*, including per-patch-edge tessellation rates.

The control point and patch data flow past a fixed-function Tessellator (TS) stage, which augments them with a set of *domain locations* for new vertices in the tessellation parameter domain. For example, for a quadrilateral domain, these would be parametric (u,v) values.

The Domain Shader (DS) is responsible for interpolating the attributes of a patch and its control points at a parametric location—for example, by performing bicubic interpolation of positions, and bilinear interpolation of colors—to produce *fine vertices*. Along with the groupwise operation of interpolation, a DS may also perform pointwise operations for each fine vertex, such as displacement mapping.

Fine vertices are assembled into primitives by another unnamed stage, and these primitives are then processed by the Geometry Shader (GS). The GS applies a user-defined procedure to each primitive, and that procedure may perform almost arbitrary computation to generate a stream of *raster vertices* that describe zero or more output primitives. For example, a GS procedure may duplicate each input primitive up to six times and project each copy into a different face of a cube-map render target. These primitives are clipped, set up, and rasterized into fragments by the fixed-function Rasterizer (RS).

The Pixel Shader (PS) maps a function over the *rasterized fragments* to produce *shaded fragments*. Shaded fragments are composited onto render-target *pixels* by a fixed-function Output Merger (OM).

Since it may be a source of confusion for readers, we will make a brief digression: The terminology used by the Direct3D system does not distinguish the concepts of

fragments and *pixels*, and refers to both as "pixels." In our experience, however, this distinction is important, albeit subtle. The Spark system, and this dissertation, endeavor to use the more precise terminology. This leads to the potentially confusing situation that the D3D11 *Pixel* Shader stage in fact processes *fragments*.

Within this pipeline we can categorize the stages according to their communication patterns. For example, the VS and PS stages each perform a functional map of a shader procedure over their input stream, one record at a time, to produce output records. This one-to-one communication pattern means that user-defined shader code running in these stages can only implement pointwise operations.

In contrast, the remaining programmable stages each apply their shader procedures to an aggregated *group* of inputs. For example, the HS procedure that computes control-point records has access to all of the coarse vertices in the neighborhood of a base-mesh face, so that it can, e.g., transform attributes into a Bézier basis. Similarly, a shader procedure for the DS has access to all of the Bézier control points for a patch to perform many-to-one interpolation. Thus the HS and DS stages expose many-to-one communication patterns, and shader code for these stages may perform groupwise operations.

The fixed-function rasterizer and programmable GS stages implement even more general many-to-many communication. A shader procedure for the GS stage receives as input a group of fine vertices (representing an assembled primitive) and can emit zero or more raster vertices to an output stream. As such, the GS can perform almost arbitrary amplification, decimation, or synthesis of geometry.

Note, however, that stages such as the HS, DS, and GS need not exclusively perform groupwise operations. For example, displacement mapping of fine vertices is often performed in the DS, and when rendering to a cube map, projection of raster vertices into clip space is performed in the GS.

The stages of the D3D11 pipeline have diverse capabilities, roles, and communication patterns. This diversity presents a challenge, since we would ideally like for our

programming model to have as few unique concepts as possible, while still supporting the full generality of present and future pipelines. In Section 3.2.2 we discuss how our abstract pipeline model captures both the general and special cases (e.g., both the GS and VS stages).

2.2.2 Compute Interfaces

In recent years, much effort has gone into using graphics hardware for computations other than rendering. These "GPGPU" or "compute" approaches were initially implemented by abstracting over the behavior of rendering architectures like OpenGL. The shading system of Peercy et al. [POAU00] lays some groundwork by abstracting the fixed-function OpenGL pipeline (with some extensions) as a kind of CISC SIMD processor; each rendering pass serves as a single instruction. Purcell et al. [PBMH02, Pur04] abstract an early programmable graphics pipeline as a *stream processor*. Toolkits such as Brook for GPUs [BFH+04, Buc05], Sh [MQP02], and Glift [LKS+06, Lef06] provide assistance for writing programs that use this stream-programming abstraction.

In contrast, more recent systems for computation on graphics hardware, such as CUDA [NVI07], OpenCL [Khr], and D3D11 Compute Shaders, do not provide a layered abstraction over a rendering architecture, and do not conform to a strict stream-programming abstraction. In particular, they expose capabilities of graphics hardware that are not accessible through rendering interfaces of the same era: shared on-chip scratchpad memory, barrier synchronization, and general read-modify-write operations to shared global memory.

The challenges in designing a general-purpose parallel programming language for graphics hardware are, understandably, different from those of designing a language for shaders which target a rendering architecture like D3D11. As such, our work is intended to *complement* these "compute" languages; we concentrate only on the problem of real-time rendering.

One issue, however, stands in the way of achieving this idealized decoupling of rendering and "compute." The D3D11 rendering pipeline allows per-fragment computations in the Pixel Shader stage to perform the same kind of global-memory read-modify-write operations as Compute Shaders. The capability is referred to as Unordered Access Views (UAVs). The OpenGL pipeline supports an extension that brings UAV-like functionality to all programmable pipeline stages [BBL+10].

The addition of UAVs to a rendering architecture compromises some desirable properties: in particular, implementation details like the use of parallelism are no longer transparent to users of the architecture. In exchange, however, rendering passes are able to build more complex data structures than flat frame-buffers. As a result, current real-time rendering research often makes use of these capabilities.

We have not fully explored the impact of adding UAV support to our Spark language and system, but we discuss challenges, possible solutions, and directions for future work in Section 5.4.1.

2.3 Programming Languages

Having discussed the history of shading languages in particular, we now turn our attention to a variety of more general topics in programming languages.

Our work on Spark draws inspiration from a large number of existing languages. In particular, many of our language features are derived from the gbeta [Ern99], Dylan [Sha96], C# [HWG03], and Scala [OAC+04] languages. In this section, we present relevant background and terminology from work on these and other languages, that can help to illuminate our design decisions in Section 3.3.

For consistency, we use a common set of typographic conventions when formatting code across different languages:

Keywords: `class`, `if`

Types: `float`, `Color`

Comments: `// like this`

Key/Value Arguments: `f( someParam: value )`

2.3.1 Mixin Inheritance

The Spark language uses object-oriented inheritance to support the extension of shaders, and multiple inheritance to support the composition of two or more shaders (we will discuss this in Section 3.3.2). In this section we provide background on the particular "mixin" approach to multiple inheritance that we use.

We assume that readers are already familiar with the idea of *inheritance*, as it appears in object-oriented languages like C++ or Java. When a language supports *multiple inheritance*, it means that a single class may inherit from more than one *direct base class*. Many programmers may be familiar with the design of multiple inheritance and its implementation in C++ [Str89], but there are many alternative approaches that can be taken [Kro85, PW90, IB82].

One family of techniques relies on the concept of linearization [DH87], perhaps best known from Common LISP [DG87, KR91]. Typically, a *linearization* is a sequence of all the direct and indirect base classes of a class, such that the order of classes in the linearization respects the partial order defined by the inheritance graph. Some systems use another concept in place of classes—e.g., "flavors" [Moo86], "mixins" [BC90, Ern02], or "traits" [SDNB02]—but the mechanism of linearization is largely the same.

For a given class, the linearization of its bases forms a total order on the relevant part of the class hierarchy (for which the inheritance graph defines a partial order). This total order can then be used to determine which definitions of a `virtual` member override which others, or to resolve the target for a reference using `super` in a Java-like language. This means that when a class inherits two implementations of a given `virtual` function, which implementation will actually be called is determined by the linearization algorithm. It is thus important that a linearization algorithm do "sensible" things, so that a programmer can typically rely on the default behavior, but also exert control when required. The C3 linearization algorithm [BCH+96] tries to respect the declared order of base classes as much as possible (i.e., whenever it does not conflict with the partial order from the inheritance graph), and is the linearization algorithm used by Python [vR], Perl 6 [Tan], and Scala [OAC+04].

Multiple inheritance in general, and linearization-based approaches in particular, are not without problems. In dynamic languages, "name clashes" can result when a class inherits two *distinct* members with the same name; static languages like C++ and C# include mechanisms to cope with these issues. It has been noted [Sny87] that linearization-based multiple inheritance can work against modularity: changing the order of base classes listed for a class `C` (seemingly an implementation detail), can affect the linearization (and hence the behavior) of classes derived from `C`.

2.3.2 Virtual Classes

Briefly revisiting the motivating example from Section 1.4, we had an initial shader for diffuse lighting that used a structure to represent coarse vertex records, as follows:

```
// Diffuse Shader:
// ...
struct CoarseVertex
{
    float3 N_world : Normal;
    float4 P_proj  : SV_Position;
};
```

Extending the shader to support texture mapping required that we make a copy of this structure declaration and modify it:

```
// Diffuse + Texture Mapping Shader:
// ...
struct CoarseVertex
{
    float3 N_world : Normal;
    float4 P_proj  : SV_Position;
    float3 uv      : TexCoord;
};
```

A given shader might have many such "connector" structures, corresponding to types of records, and extensions to a shader will frequently require changes to several of the structures. In order to extend a shader without resort to copy-paste programming, we require a way to extend an entire "family" of connector structures. Fortunately, many solutions to this problem have been explored in the context of object-oriented languages.

Listing 2.1 sketches the shader from Section 1.4, Listing 1.1, in an object-oriented fashion, using nested classes. We will refer back to this listing throughout this section.

In a typical object-oriented language (e.g., Java), the interface in Listing 2.1 would allow us to instantiate the `Diffuse` shader and process vertices:

```
class Diffuse
{
    class AssembledVertex
    {
        float3 P_model;
        float3 N_model;
    }

    class CoarseVertex
    {
        float3 N_world;
        float4 P_proj;
    }

    // ...

    CoarseVertex VS( AssembledVertex input ) { /* ... */ }
    Fragment PS( CoarseVertex input ) { /* ... */ }
}
```

Listing 2.1: *Nested classes, corresponding to the diffuse shader and connector structures in Listing 1.1.*

```
Diffuse aShader = new Diffuse();
Diffuse.AssembledVertex av = ...;
Diffuse.CoarseVertex cv = aShader.VS( av );
```

The interface does *not*, however, stop clients from attempting certain invalid operations: e.g., trying to process a vertex output by one shader instance with the fragment shader of another instance:

```
Diffuse otherShader = new Diffuse();
// Runtime Error:
//     Input vertex came from a
//     different shader instance
Diffuse.Fragment result = otherShader.PS( cv );
//                                        ^^
```

Ernst [Ern01] introduces the notion of *family polymorphism*, which allows a program to express that several objects belong to a given family, and that mixing objects between families is not permitted. The basic idea is that *different* instances of the `Diffuse` class should have *different* types of `CoarseVertex`. So rather than a single type `Diffuse.CoarseVertex`, in our example we would have two types: `aShader.CoarseVertex` and `otherShader.CoarseVertex`. This distinction allows us to catch the earlier error statically:

```
Diffuse aShader = new Diffuse();
aShader.AssembledVertex av = ...;
aShader.CoarseVertex cv = aShader.VS( av );

Diffuse otherShader = new Diffuse();
// Compile-Time Error:
//     Expected otherShader.CoarseVertex,
//     found aShader.CoarseVertex
otherShader.Fragment result = otherShader.PS( cv );
//                                            ^^
```

At this point, one might note that the way that "nested" classes are referenced in typical object-oriented languages like Java and C# (e.g., `Diffuse.CoarseVertex`), is different from how other members of `Diffuse`, such as methods, are referenced using a particular object (e.g., `aShader.VS()`). It is as if nested classes are implicitly `static` in these languages. Family polymorphism removes this discrepancy, so that nested classes are referenced in the same fashion as any other member.

If nested classes are like any other member (fields, methods, etc.), it next becomes natural to ask whether a member class may be `virtual`. The idea of a virtual class may be difficult to grasp at first, but is easy to define by analogy to virtual methods:

> A **virtual method** `X` is a **function**-valued member of a class `C`. A class that extends `C` can provide a definition for `X` that **overrides** the definition from `C`. References to `X` are late-bound, and depend on the run-time type of the object used.

> A **virtual class** `X` is a **class**-valued member of a class `C`. A class that extends `C` can provide a definition for `X` that **further extends** the definition from `C`. References to `X` are late-bound, and depend on the run-time type of the object used.

In a language with virtual classes, we can not only define families of classes, but also *extend* them, e.g.:

```
class Texturing extends Diffuse
{
    extend class CoarseVertex
    {
        float2 uv;
    }
    // ...
}
```

In this example, `Texturing` extends the definition of `CoarseVertex` in `Diffuse` so that coarse vertices also have texture coordinates. In this way, virtual classes allow a programmer to extend a family—perhaps even an entire library—of related classes, while maintaining the established relationships between those classes. For example, the `VS()` in `Diffuse` was declared to return a `CoarseVertex`; the `VS()` method inherited by `Texturing` will itself return an instance of the *extended* `CoarseVertex` class. This same pattern has also been referred to as higher-order hierarchies [Ern03] and nested inheritance [NCM04].

The genesis of virtual classes can be traced back to Simula [DMN68], in which instances of classes are conceptually identified with activation records of functions. The Beta and gbeta languages follow this inspiration, using a single construct called a *pattern* to unify both classes and methods. Support for virtual methods (i.e., virtual patterns) led naturally to support for virtual classes in Beta [LMMP89].

Subsequent work has demonstrated that it is possible to incorporate virtual classes into a sound type system [EOC06], and has investigated efficient implementations

[BNE09]. The Scala language provides for the more restricted case of virtual *types*, which have also been proven sound [OCRZ03], even when combined with parametric polymorphism [MPO08a, MPO08b].

Virtual classes may initially seem esoteric, but the example developed in this section hints at how they underly our approach to modular and extensible shaders in Spark. We will discuss in Section 3.3.5, how we take inspiration from virtual classes in our language design. In Section 5.2 we attempt to make this connection more explicit by demonstrating a mapping from the core Spark language to Scala with an extension that adds virtual-class support.

2.3.3 Extensible Initialization

Along with inheritance, a defining feature of many popular object-oriented languages is the use of *constructors* to perform instance initialization. While constructors have an intuitive mental model, they also have numerous problems:

- During the execution of a constructor, the `this` object is not fully initialized. It may be passed to operations that rely on certain invariants that have not yet been established. Doing so is a semantic error that the compiler cannot detect.
- Creating an object (e.g., with a `new` expression) requires naming the class to be constructed *statically*, at compile time. Code cannot late-bind or parameterize over the class of objects to be created without using, e.g., the Factory pattern [GHJV95].
- Some languages do not allow a derived class to inherit base-class constructors, and even when this is possible (e.g., in C++11 [CPl11]), declaring an *extended* constructor requires the reiteration of the original constructor signature.

To make the issues with constructors and inheritance concrete, consider the C++ code in Listing 2.2. This example is derived from the code in Section 1.4, in which we define and then extend a connector structures for coarse vertices. Here we see a

```
class CoarseVertex {
public:
    CoarseVertex( float3 P, float3 N )
        : P_(P), N_(N)
    {}
private:
    float3 P_;
    float3 N_;
};

class ExtCoarseVertex {
public:
    ExtCoarseVertex( float3 P, float3 N, float2 uv )
        : Vertex(P, N), uv_(uv)
    {}
private:
    float2 uv_;
};

// Construct a CoarseVertex:
CoarseVertex cv( pos, nrm );
// Construct an ExtCoarseVertex:
ExtCoarseVertex ecv( pos, nrm, tex );
```

Listing 2.2: *Example of C++ constructor extension.*

base class `CoarseVertex` that requires both a position and normal for initialization, and a derived class `ExtCoarseVertex` that requires one additional parameter: a texture coordinate. Note that `ExtCoarseVertex` must reiterate (that is, re-declare) the same constructor parameters as `CoarseVertex`. This work is redundant, and must often be done for each of `CoarseVertex`'s derived classes (that is, the amount of work scales both with the number of parameters, and the number of derived classes). This code is also *fragile*: any change to the signature of `CoarseVertex`'s constructor requires changes to each of the derived classes (and not just call sites at which a `CoarseVertex` is constructed). This problem is compounded by features like multiple inheritance or virtual classes: a derived class may inherit initialization parameters from many sources.

```
define class <coarse-vertex> (<object>)
    slot P :: <float3>,
        required-init-keyword: P:;
    slot N :: <float3>,
        required-init-keyword: N:;
end class;

define class <ext-coarse-vertex> (<coarse-vertex>)
    slot uv :: <float2>,
        required-init-keyword: uv:;
end class;

// Construct a <coarse-vertex>:
let cv = make(<coarse-vertex>,
              P: pos, N: nrm);
// Construct an <ext-coarse-vertex>:
let ecv = make(<ext-coarse-vertex>,
               P: pos, N: nrm, uv: tex);
```

Listing 2.3: *The example in Listing 2.2 rendered in idiomatic Dylan.*

Alternatives to constructors exist which do not share some of these drawbacks. For example, the Common LISP Object System [DG87] and the Dylan language [Sha96] allow a class to define a number of initialization keywords, which can be used to provide values to data members ("slots") when an object is initialized. Listing 2.3 shows the same example as in Listing 2.2, rendered in an idiomatic Dylan style. We define a required initialization keyword for each slot introduced, and at the point where we construct an instance (the calls to `make()`), we provide a keyword argument for each of the inherited initialization keywords. Note that in this approach, the class `<ext-coarse-vertex>` is not required to re-iterate any information pertaining to the initialization behavior of `<coarse-vertex>`. The language implementation synthesizes the correct constructor signature for a derived class, even in the case of multiple inheritance. This is possible because the use of keyword argument passing removes the need to arrive at a fixed ordering for the constructor parameters (which might be inherited from several sources).

The gbeta language [Ern99] permits flexible initialization without the use of constructors, by allowing each class member to declare an initializer that might depend on any other class member. Because the type of one storage location might depend on the value in another location (as a result of virtual classes; see Section 2.3.2), it is important that the members be initialized in an order that respects their dependencies, without allowing any code to "see" a partially-initialized object (since this could render typing judgements invalid). In simple cases an ordering can be found at compile time, but in the general case the implementation falls back on lazy evaluation [Nie07]. We use this same basic approach in the Spark system.

2.3.4 Type Systems

In this section, we present a brief introduction to terminology and notation for type theory. We will use this notion when discussing the Spark type system in Section 5.1. Readers who are interested in a more detailed introduction to type theory are directed to Pierce's excellent introduction [Pie02]. Notation for type systems are not entirely standardized; whenever possible, we try to follow Pierce's conventions.

Types

We may classify terms in a programming language according to the types of values they may produce at run-time. We write typing judgements in the following fashion:

$$\texttt{1} : \texttt{int}$$
$$\texttt{float3(0, 1, 2)} : \texttt{float3}$$
$$\texttt{1.0f} : \texttt{float}$$

where : indicates that the expression on the left has the type on the right.

The responsibility of a type-checker is to determine the type of every expression. When no type can be assigned to an expression, a type error has been found. We may declare that a given expression has no type as follows:

$$\texttt{1 + true} : \bot$$

In addition to basic types like `int` or `float`, a formal type system will usually need a type to represent functions:

$$\texttt{cross} : \texttt{float3} \to \texttt{float3} \to \texttt{float3}$$

Here we say that the function `cross` takes two `float3` parameters, and returns a `float3`. The type `D` $\to$ `R` represents a function from domain `D` to range `R`. The $\to$ operator is right-associative, so that `A` $\to$ `B` $\to$ `C` is equivalent to `A` $\to$ (`B` $\to$ `C`).

The function `cross` above is given in a *curried* fashion: the two-parameter function has been turned into a function of one parameter, that returns another function that takes the remaining parameter.

Kinds

Many programming languages have some form of "generic" or "templated" types. For example, the C++ type `std::vector<T>` represents a sequence of values of type `T`. Such generic types present challenges for a type checker. We must be able to rule out nonsensical types like `int<std::vector>`, as well as ensure that generic types are applied to appropriate arguments before use (e.g., it makes no sense to have a variable of type `std::vector` in C++).

In response to these challenges, type theorists introduce the notion of *kinds*. Kinds can be seen as the "types of types": kinds classify types just as types classify expressions.

Two kinds are common to many type systems:

$$\begin{array}{lll} K & ::= & \\ & \mid \quad * & \texttt{// kind of proper types} \\ & \mid \quad K_1 \Rightarrow K_2 & \texttt{// arrow kind constructor} \end{array}$$

The kind $*$ is traditionally read "type" and is the kind of all *proper types*. Proper types are those that classify values, such as `int` and `std::vector<float>`. Other types like `std::vector`, which cannot classify any value, are *non-proper types*.

Much like typing derivations above, we write kinding derivations as:

$$\texttt{int} :: *$$
$$\texttt{std::vector<float>} :: *$$

where :: is read as "has kind."

The $\Rightarrow$ constructor is used to represent the kinds of so-called "generic" or "templated" types. For example, the (non-proper) C++ type `std::vector` can be kinded:

$$\texttt{std::vector} :: * \Rightarrow *$$

that is, `std::vector` is a type-level function that maps one data type (the type of elements `T`) to another (the type of vectors of `T`). As another example, the $\rightarrow$ type constructor for functions can be kinded:

$$\rightarrow :: * \Rightarrow * \Rightarrow *$$

since it maps two types—the domain type `D` and range type `R`—to the appropriate function type `D` $\rightarrow$ `R`.

The notation we have introduced, while limited, is sufficient to discuss the basic type-system features of commonly-used functional and object-oriented languages.

2.4 Software Engineering

Having discussed many *specific* programming language concepts and features, we now turn our attention to a broader topic: what does it mean for a programming model to support "modularity" and "composability"? If we are to evaluate the success or failure of the Spark system along such axes, we must find suitable definitions for these terms.

2.4.1 Modularity

We take as our starting point Parnas's discussion of modularity [Par72]. In contrasting different decompositions of a system, Parnas provides a concise definition for a "modularization" (emphasis ours):

> The system is divided into a number of modules with **well-defined interfaces**; each one is small enough and simple enough to be thoroughly understood and well programmed.

Simply being decomposed into modules with interfaces, however, is not sufficient for what we typically call modularity. We also wish to be able to change the *implementation* of one module (but not its interface) without impacting the implementation of other modules, or the correctness of the assembled program. Parnas proposes that a modularization should be driven by "information hiding" [Par71]. Here again we quote [Par72]:

> Every module in the [...] decomposition is characterized by its **knowledge of a design decision which it hides from all others**. Its interface or definition was chosen to reveal as little as possible about its inner workings.

This idea is similar to the notion of *separation of concerns* [Dij82]. These sources form the basis for our definition of "modularity": the physical decomposition of a program (into modules) reflects its logical decomposition (into concerns).

2.4.2 Composability

Good modularity allows one to change the implementation of a module without impacting others. If we further wish to support *multiple*, interchangeable implementations, then we are discussing *composability* of software *components* [McI68, Cox90].

Components are not unlike modules in that they encapsulate a related set of services behind an interface. They differ, however, in that multiple components implementing a given interface my co-exist in a single application.

Component interfaces are typically decomposed into those services that a component *provides* and those it *requires*. An important principle of software components is *substitutability*: the interface required by a component may be satisfied by any other component that provides that interface.

The Scala language is notable for identifying both modules and components with the objects of object-oriented programming. In Scala, the composition of modules and the composition of components are both described using (mixin) inheritance of classes.

2.4.3 Aspect-Oriented Programming

Sometimes, when code is decomposed so as to separate one set of concerns, the code for other concerns "cuts across" the modularization. A commonly-cited example of a cross-cutting concern is logging; a large application will have logging code spread across many modules. This observation motivates the approach known as *aspect-oriented programming* (AOP) [KLM+97, KHH+01]. We will discuss in Section 5.3 how the Spark programming language may be viewed as an AOP language.

An *aspect* is a unit of modularity (usually separate from more traditional units like classes) that is used to encapsulate a cross-cutting concern. In many formulations of AOP, an aspect intercedes in the execution of an "ordinary" program by means of *pointcuts* and *advice*. Pointcuts describe dynamic contexts in the executing program: e.g., around calls to functions with the prefix `Log_`. Advice then specifies how to modify the program behavior when particular pointcuts occur at runtime. Typically a final executable program is derived by *weaving* the code for aspects into the code for other modules (e.g., by modifying Java bytecode as it is loaded). This idea of intercepting and modifying the behavior of an existing program is similar in spirit to a metaobject protocol [KR91] in a dynamic language, but AOP is compatible with static languages.

2.5 Summary

In this chapter we have provided a brief history of real-time shading languages and rendering pipelines; described several features from existing languages that might not be familiar to readers; and introduced suitable notation and terminology for talking about type-theory and software-engineering concepts in future chapters. With these preliminary matters out of the way, we now turn our attention to presenting the design of the Spark language.

Chapter 3

The Spark Language

In this chapter, we discuss the design of the Spark shading language. We begin with an overview of our design goals, before discussing the abstraction that underlies our design. We then discuss some of the key design decisions we had to make in realizing our design, and conclude with a few small, illustrative examples of Spark code.

3.1 Design Goals

At a high level, our goal for the Spark language was to achieve a synthesis of the best aspects of RTSL—composition of complex effects from modular pipeline shaders—with those of more recent shading languages like Cg—flexibility, generality, and support for modern graphics pipelines. In this section, we illuminate our design goals by discussing how they differ from those of languages like Cg and RTSL, respectively.

3.1.1 Differences from Cg/HLSL/GLSL

Our goals are largely similar to those of Cg, HLSL, and GLSL, differing primarily in the introduction of three new goals:

Modularity

Programmers should be able to define orthogonal program features as separate modules. Changes to one module should not require modification of unrelated modules. What constitutes an "orthogonal feature" should be driven by the needs of the user (i.e., separation of concerns) and not those of the implementation.

In particular, logical concerns like tessellation, that would otherwise cross-cut the pipeline structure, should be expressible as a single module.

Composability

It should be possible to combine or extend thoughtfully designed modules to create new shaders, without resort to copy-paste programming. In particular, a developer should be able to define a library of shader components for different effects and combine them in a wide variety of rendering passes.

To demonstrate both modularity and composability, we determined that a Spark user should be able to write a complex D3D11 tessellation effect as a reusable module.

Automatic Plumbing

Attributes defined in pointwise shading code should be plumbed automatically as required. For example, a module that defines a per-vertex color and uses it in per-fragment computations should not require code to, e.g., interpolate that color across a higher-order surface being tessellated.

The need for automatic plumbing is really a consequence of the preceding two goals, if we are to support the separate definition and subsequent composition of concerns that involve pointwise and groupwise shading code. In particular, in order to define a tessellation effect as a reusable module, we need to be able to tessellate models with any number of per-vertex attributes (colors, texture coordinates, etc.).

3.1.2 Differences from RTSL

While our work builds on the pipeline-shader approach of RTSL, we do not share some of the goals that motivated that system. As such, we felt the need to enumerate some additional design goals that serve to clarify our mission to "modernize" the pipeline-shader approach and make it a suitable alternative to Cg, HLSL, and GLSL:

Support Modern Graphics Pipelines

The shading language should be able to expose the capabilities of modern rendering pipelines such as D3D11, including control flow and programmable groupwise operations. Furthermore, the same language should be usable with future extensions to the pipeline.

This latter point is quite important: it would not be enough to simply devise specific language constructs to support e.g., tessellation. We require more general mechanisms that could apply equally well to future pipelines and stages.

As discussed in Section 2.2.2, we do not concern ourselves with the "compute" interface in D3D11, as this is separate from the rendering pipeline. In particular, the compute interface does not expose a multi-stage pipeline, and so the issues that motivate our work would not seem to be a concern.

The current Spark system does not fully achieve this goal, as it omits one major feature of the D3D11 rendering pipeline: support for arbitrary read-modify-write operations to memory though "unordered access views" (UAVs). We discuss possible directions for future work to address this limitation in Section 5.4.1.

Domain-Motivated, Not Domain-Specific

Specialization to the domain of real-time shading is acceptable; as stated above, we are not trying to build a general-purpose programming language for graphics

hardware. The Spark language need not support arbitrary dataflow programming (as, e.g., Imagine's StreamC and KernelC [KDR+02]), since the topology of a rendering pipeline is typically fixed.

We recognize, however, that one of the key innovations in Cg is the lack of a particular domain model for things like surface/light interaction. In a similar fashion, our solution should not constrain user applications except as the pipeline requires. Programmers should be free to define their own application-specific interfaces for illumination, and other aspects of the graphics domain.

Performance

Shaders are effectively the inner-most loops of a renderer. Benefits to abstraction or modularity must be weighed against costs to performance. Our goal was to achieve performance similar to hand-tuned shaders in existing high-level languages.

Phase Separation

Flexibility in the shading language and runtime should not result in unexpected pauses for run-time compilation. There should be a clear *phase separation* between code generation and execution, and run-time parameter changes should not trigger recompilation or other expensive operations.

This goal stems from complaints that real-time graphics programmers frequently level at shading-language runtimes. While a system that optimizes or specializes shaders "on the fly" should achieve good amortized performance, the worst-case performance on particular frames can vary widely (e.g., when an object using a new combination of effects is first rendered). For many domains, including games, predictable worst-case frame rates are more important than better average-case performance.

3.2 Shader Programming Abstraction

This section introduces an abstraction for shader programming that underlies our work. This abstraction defines an interface between the declarative code in a shader graph, and procedural code comprising the rest of a rendering system.

Figure 3.1 illustrates this interface. A declarative shader graph is used to express pointwise shading code, while the stages of the pipeline execute procedural code that may perform groupwise operations. The interface between the shader graph and the pipeline is provided by a family of data types we call *record types*, which expose rates of computation to the pipeline, in addition to *plumbing operators*, which are invoked by the shader graph to plumb values between pipeline stages.

Spark is a shading language for programming a graphics pipeline; it is *not* a system for constructing new pipelines (as, e.g., GRAMPS [SFB+09]). While systems exist for more general dataflow programming (c.f., Imagine [KDR+02]), most rendering interfaces expose a fixed, or minimally configurable, pipeline topology. A fixed topology allows for the pipeline implementation to include highly optimized scheduling logic that is independent of particular applications. We assume in our work that shaders are authored against a fixed *pipeline model*, that might be defined as part of a standard library. The pipeline model defines the stages, record types, and rates of computation for a particular pipeline (e.g., D3D11). It is the role of the shading language, then, to expose the capabilities and constraints of the pipeline model to the user.

3.2.1 Shader Graphs

In our abstraction, shader graphs—DAGs representing shading computations—are used to express pointwise shading code. The right of Figure 3.1 depicts a shader graph. Each node in the graph represents either an input to the shader or a value it computes.

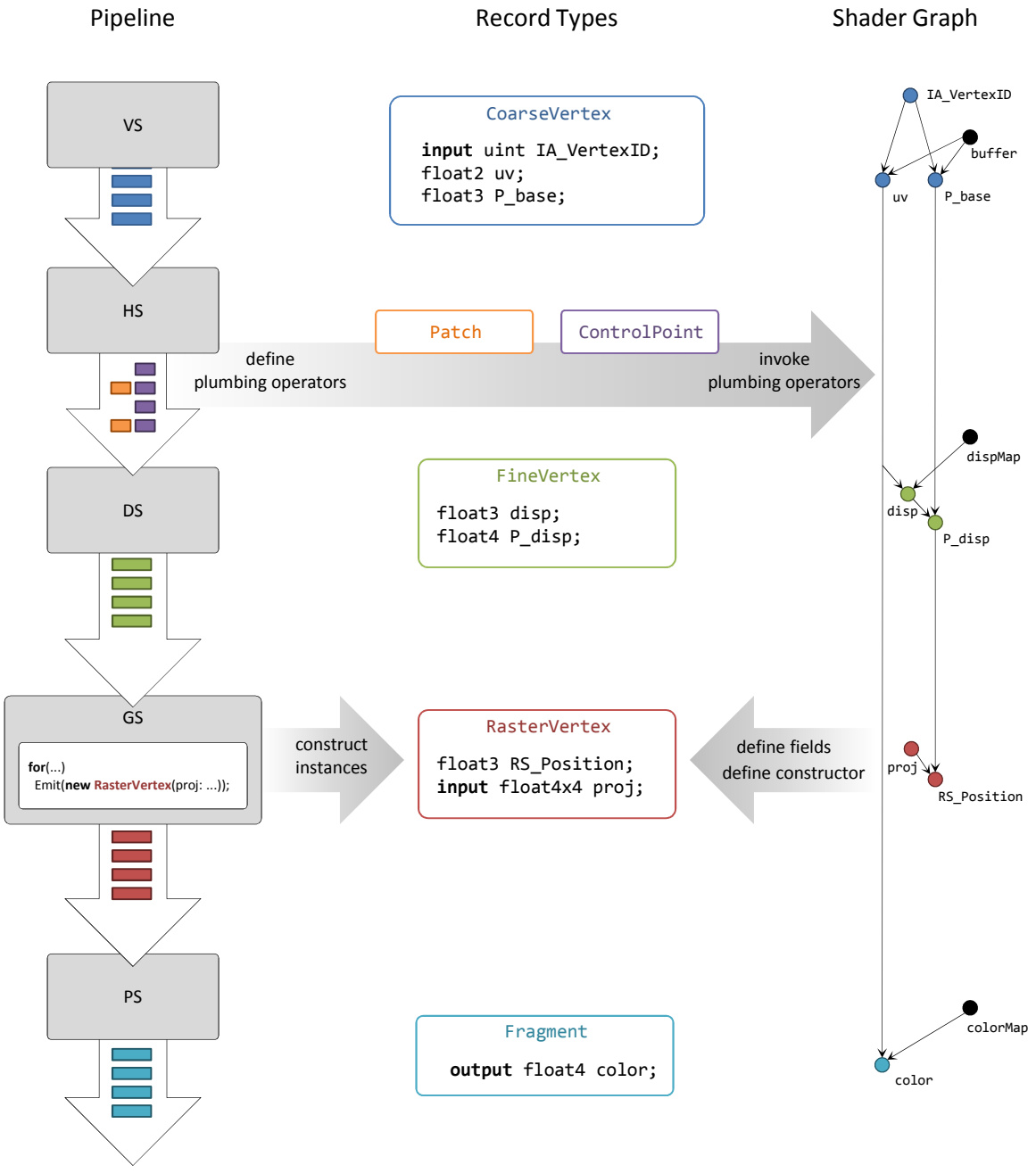

Figure 3.1: *Shader graphs are mapped to the rendering pipeline with the use of* record types. *Nodes in the shader graph are colored according to their rate of computation: each node corresponds to a field in the associated record type. Per-stage kernels and shader procedures running in the rendering pipeline construct, manipulate and communicate records.*

As in RTSL, every node is colored according to its rate of computation. For example, values that are computed for every fragment are given the per-fragment rate. Black nodes in this graph correspond to values with uniform rate: that is, uniform shader parameters. Shader-graph nodes represent attributes: e.g., a per-fragment color attribute.

The graph in Figure 3.1 represents the following computations:

- Per coarse vertex, position and texture-coordinate data are fetched from a buffer, using a system-provided vertex ID.
- Per fine vertex, a vector displacement map is sampled and used to compute a displaced position.
- Per raster vertex, the position is projected into clip space.
- Per fragment, a color map is sampled.

Some attributes (shader-graph nodes) are pre-defined for a particular rendering system. For example, D3D11 defines a per-coarse-vertex sequence number: this appears in our graph as the `IA_VertexID` input. System-defined attributes may also represent shader outputs used by the rendering system: for example, `RS_Position` represents the projected position consumed by the rasterizer.

3.2.2 Pipeline Model

The left of Figure 3.1 depicts the programmable *stages* of the D3D11 rendering pipeline. The stages are connected by *streams* that carry data in the form of *records*. Different stages and streams will make use of different *types* of records. In the case of the D3D11 pipeline, the record types correspond to the terms introduced in Section 2.2.1: coarse vertices, fragments, etc. For example, the stream connecting the DS and GS carries `FineVertex` records. Our abstraction does not assume that stages have only one input or output stream, nor does it assume the pipeline is cycle-free.

Each stage in the pipeline runs a system-defined *kernel* that defines its behavior (our use of this terminology is inspired by KernelC [KDR+02]). In general, a kernel can pop records from input streams, execute system- or user-defined operations, and push records onto output streams. A fixed-function stage can be understood as having a hard-coded kernel, perhaps parameterized on data, but not on code.

In contrast, the kernel for a programmable stage applies a per-stage shader procedure to compute results. For example, the kernel for the D3D11 GS stage pops fine vertex records (representing a primitive) from its input stream, and passes both these records and a handle to its output stream to a user-defined shader procedure. This shader procedure may then construct zero or more `RasterVertex` records and push them on the output stream.

Other stages follow a similar pattern. The kernels for the VS and PS stages always pop a single input record and apply a per-stage procedure to compute one output record, implementing their one-to-one communication pattern. The kernels for the HS and DS stages pop an aggregate of input records (e.g., all of the coarse vertices in the neighborhood of a base-mesh primitive) and then apply per-stage procedures one or more times to construct a number of independent output records.

In a shader-per-stage language like Cg, these per-stage procedures are authored directly by the user. Our abstraction supports this degree of flexibility, and indeed we make use of it for the GS stage, but for the other programmable stages in D3D11 it is not required. For example, the only thing that a VS-stage procedure can do is construct a coarse vertex given an assembled vertex, so the rendering system can provide a single "one size fits all" VS procedure, e.g.:

```
CoarseVertex VS( uint IA_VertexID, AssembledVertex av )
{
    return CoarseVertex( IA_VertexID, av );
}
```

In this case, the VS procedure delegates all of the shading work to the constructor for `CoarseVertex` records.

3.2.3 Rates and Record Types

A key property of our abstraction is that record types are in one-to-one correspondence with rates of computation. That is, for every record type there is a corresponding rate of computation, and vice versa. For example we have both a type `CoarseVertex`, as well as per-coarse-vertex computations in our shader graph.

A record type may be thought of like a C++ `struct` type: it has some number of *fields*, as well as a *constructor*. Continuing the identification of rates of computation and record types, for every node in the shader graph with a given rate, there is a field in the corresponding record type. For example, our shader graph in Figure 3.1 defines a per-raster-vertex position `RS_Position`, and the `RasterVertex` record type has a corresponding `RS_Position` field.

The constructor for a record type is similarly defined by the shader-graph nodes. For example, the per-coarse-vertex computations in the shader graph define the body of the `CoarseVertex` constructor. Input nodes in the graph (e.g., `IA_VertexID`) correspond to constructor parameters—that is, values which must be provided whenever a `CoarseVertex` record is created.

This identification of rates and record types provides the required interface between per-stage procedures in the pipeline and code in the shader graph. When the system-defined VS procedure above invokes the `CoarseVertex` constructor, it has the effect of performing all the per-coarse-vertex computation in the shader graph, and collecting the resulting values in a record.

This model also applies to a stage with a user-defined procedure, such as the GS. A user-defined GS procedure may be authored to perform a particular groupwise operation—e.g., duplicating primitives for rendering to a cube map—by constructing and pushing new `RasterVertex` records. This code only has to know about the input attributes of the shader graph: these define the signature of the `RasterVertex` constructor. Additional pointwise computations (i.e., additional non-input nodes) may be independently added to the shader graph without conflict (e.g., `RS_Position`). The

operations in the shader graph, in turn, remain oblivious to how many `RasterVertex` records the GS procedure might construct, or in what order it might emit them. Our abstraction thus allows groupwise code in a shader procedure to be decoupled from pointwise code in a shader graph.

Our approach here is a refinement of how rates are modeled in RTSL. The RTSL system maps shader graphs to programmable pipeline stages by identifying rates of computation with *stages* of the pipeline. For example, RTSL computations with the `vertex` rate map to instructions compiled for the vertex-processing stage of the pipeline. While this is an intuitive approach for a pipeline with programmable VS and PS stages, it does not map as easily to the needs of a current pipeline like D3D11. For example, we want to be able to express per-raster-vertex computations, but no pipeline stage executes at per-raster-vertex rate.

3.2.4 Plumbing Operators

When an attribute with per-coarse-vertex rate (e.g., `P_base`) is used as an input to a per-fine-vertex computation (`P_disp`), the attribute must be plumbed from one rate to the other. We can recognize plumbing in the shader graph in Figure 3.1 wherever an edge connects nodes with different colors. We refer to the operations that perform plumbing—that is, that create cross-rate edges—as *plumbing operators*. Each cross-rate edge in the graph (e.g., from `P_base` to `P_disp`) was created by invoking a plumbing operator. In practice, plumbing an attribute like `P_base` yields a new attribute with the same data type (e.g., `float3`) but a different rate; for clarity we omit the nodes that result from plumbing in Figure 3.1.

A shader graph specifies *where* plumbing must be performed, but does not define *how* plumbing operators perform this work. Some plumbing operators are defined as part of the pipeline model: for example, to expose interpolation from raster vertices to fragments by a fixed-function rasterizer. Additional plumbing operators might be defined by the shader programmer, as part of a shading effect like tessellation. These user-defined operations may then be *invoked*, perhaps implicitly, in the shader graph.

Plumbing might involve interpolation or more general resampling; as such the implementation of a plumbing operator will typically involve groupwise code, requiring access to explicit records. For example, an operator for plumbing attributes from coarse to fine vertices might operate on an aggregation of `CoarseVertex` records representing the neighborhood of an input primitive.

Different plumbing operators may apply to different types of attributes. For example, in Figure 3.1, positions might be plumbed from coarse to fine vertices using bicubic Bézier interpolation. Texture coordinates or colors, in turn, might be subjected to only bilinear interpolation.

In the process of plumbing an attribute from one rate to another, it might be resampled to intermediate rates. For example, to interpolate vertex positions from coarse to fine vertices, they might first be converted to per-control-point positions in a Bézier basis, and then interpolated. In this way, plumbing might introduce additional attributes (shader-graph nodes) not depicted in Figure 3.1.

3.3 Key Design Decisions

In this section, we review some of the most important design decisions we had to make in realizing the Spark language. In each case, we try to explain why we made the choices we did, and highlight alternative approaches we considered.

3.3.1 A Language with Declarative and Procedural Layers

We believe that programmers should be able to define and compose modules that might intersect multiple stages of the rendering pipeline. RTSL demonstrates that this is possible using declarative shader graphs. However, as discussed earlier, RTSL's shader graphs cannot express shaders with control flow, nor can they define the various kinds of groupwise shading operations we wanted to support.

We decided to tackle this problem by building a shading language with two layers. In the upper layer, the user defines declarative shader graphs. That is, instead of writing a shader as a procedure composed of statements, the user declares a set of shader-graph nodes. Each node is either a shader input, or defines its value as an expression of other nodes.

In the lower layer, the user defines procedural *subroutines*. Within a subroutine, the programmer can make use of local variables, control flow and other features of procedural shading languages. A subroutine can then be called in the definition of a new shader-graph node. The new node may encapsulate a complex computation, but will be assigned a single rate of computation. In this way we avoid the problematic interactions between control flow and rate qualifiers that were discussed in Section 2.1.4.

An alternative would have been to introduce looping and conditional constructs to the declarative language in the form of higher-order functions or recursion, as is done in Renaissance [AR05]. We were concerned, however, that a purely functional approach would alienate programmers who are more familiar with C-like procedural languages. Furthermore, while our original motivation for including procedural subroutines was the ability to express control flow, we soon discovered additional uses. One simple use was that we could express plumbing operators as a special kind of subroutine (although we do not allow control flow in plumbing operators). Another use was in expressing the per-stage procedure required by the GS.

As discussed in Section 3.2, our abstraction includes per-stage shader procedures running in the rendering pipeline; while system-defined procedures are sufficient for many stages, the GS stage needs a user-defined procedure. We model this in Spark as a procedural subroutine that the user must define, which may push zero or more raster vertex records onto an output stream.

A purely functional language could instead model the GS as yielding a variable-length list, but implementing this efficiently might be a challenge. Alternatively, the side-effects (emitting raster vertices) could be explicitly ordered with, e.g., monads [Wad90].

3.3.2 Shaders Are Classes

Shade trees and RTSL give shader graphs the *appearance* of procedures, even though the underlying shader-graph representation is quite different. Users might be surprised to find out that they cannot use data-dependent control flow in the body of a shader. We wanted to avoid this confusion in Spark, particularly because we also support subroutines in which control flow *is* allowed.

RSL shows that it can be valuable for an application to treat shaders as classes rather than procedures. For Spark, we treat shaders as classes both semantically and syntactically. This choice of representation brings benefits along multiple axes.

First, classes are a declarative rather than procedural construct: they describe what something *is*. Users are familiar with the idea that a class in C++ or Java directly contains declarations (of types, fields, methods, etc.), but does not directly contain statements (e.g., a class cannot directly contain a `for` loop, although a method in the class may). Similarly, a Spark shader class contains declarations of types, subroutines, and, most importantly, attributes (nodes in the shader graph). This decision aims to reduce the learning curve for the language, and make confusion between procedural and declarative code less likely.

Second, classes bring a rich set of mechanisms for modularity and composition from the discipline of object-oriented programming (OOP). Our formulation of OOP is heavily influenced by the Scala language [OAC+04]. Notable capabilities include:

- A shader class can *extend* another shader class, inheriting its declarations (attributes, etc.), and adding new nodes of its own—without changing the behavior of the original class.
- Multiple shader classes can be *composed* by using multiple mixin inheritance, implemented through *linearization* (we employ the C3 linearization algorithm; see Section 2.3.1).
- By declaring some shader-graph nodes `virtual`, a shader can allow parts of its behavior to be customized.

- Shader classes with `abstract` members can represent interfaces that a module either *requires* or *provides*.

Every Spark shader class inherits (directly or indirectly) from a base class that defines the particular pipeline being targeted (e.g., `D3D11DrawPass` for D3D11). From this base class, the shader class inherits pipeline-specific types, subroutines, and shader-graph nodes, which define the services that the pipeline provides and requires. Shaders targeting different pipelines will inherit different capabilities and responsibilities.

The representation of shaders as classes in Spark also benefits our run-time interface. For each Spark shader class, our compiler generates a C++ "wrapper" class. The interface of this class is generated statically, but the implementation might be generated at run-time. The wrapper is used to construct shader instances and set values for parameters. The wrapper also exposes a `Submit()` method that handles binding of shaders, resources and state. This is similar in spirit to existing effect systems, but generating wrapper code allows for a low-overhead, type-safe interface to shaders.

The object-oriented representation also helps us achieve a clear phase separation. In the Spark run-time interface, creating a shader instance is a heavy-weight operation: it may generate GPU code or allocate other resources. Once a shader instance has been created, however, setting its parameters and using it for rendering are lightweight operations that should not trigger recompilation.

Users may, of course, want to specialize shader code to particular parameter values. The RTSL system models this by having both `constant` (compile-time) and `primitive group` (i.e., uniform) rates. Spark follows this same approach and has both `@Constant` and `@Uniform` rates. The intention with this design is that users may specify values for constant parameters at the time a shader instance is created, although our current implementation lacks this feature.

Our use of an object-oriented abstraction seems to be a kind of design sweet spot, providing elegant solutions for a number of requirements, but it is by no means the only option. For example, rather than use OOP features in the shading language to

represent composition, we might instead use explicit operators at run-time (as with Cook's grafting operator, or the Sh shader algebra). One weakness that results from using linearization-based inheritance for composition is that a given module (shader class) in Spark can only compose one "copy" of any other module; that is, we can only express *is-a* and not *has-a* relationships between modules. In Section 5.4.4 we sketch a possible future direction for addressing this limitation.

3.3.3 Model Rates of Computation in Libraries, Not the Compiler

Existing shading languages with rates of computation represent them with a fixed set of keywords (e.g., `varying` in RSL or `fragment` in RTSL). However, modeling each rate as a language keyword wouldn't mesh well with our goals for Spark: different rendering pipelines will support different rates, and the introduction of a new pipeline stage should not require changing the language syntax.

We decided that Spark should have an extensible set of rate-qualifier names, rather than a fixed set of keywords. In particular, system libraries that expose different pipelines should be able to expose different rates. To distinguish them from other names in the language, we require that the names of rate qualifiers start with an `@` sign. Where RTSL has `fragment`, then, an equivalent Spark shader uses `@Fragment`. The intention is that `@` can be read as "per-," so that a `@Fragment Color` is a "per-fragment color."

As discussed in Section 3.2.3 every rate of computation in our abstraction is associated with a corresponding record type. In Spark, the record type corresponding to a rate qualifier has the same name without the `@` prefix. So, for example, the record type associated with the `@ControlPoint` rate is `ControlPoint`.

3.3.4 Expose Rate Conversion as Plumbing Operators

Simply allowing for an extensible set of rate-qualifier *names* is not sufficient. One of our key design goals is to allow automatic plumbing. For example, values with the `@CoarseVertex` rate qualifier should be usable in `@FineVertex` computations.

The Spark compiler performs plumbing by automatically inserting calls to plumbing operators. This design was inspired by languages that support user-defined implicit conversions. Both C++ and Scala allow users (and libraries) to define auxiliary functions that perform type conversion. Calls to these functions can be inserted by the type-checker as needed, according to well-defined rules.

In the case of Spark, plumbing operators take the form of subroutines with explicitly rate-qualified inputs and outputs, e.g.:

```
implicit @FineVertex float coarseToFine(
    @CoarseVertex float attr );
```

Here we have an operator that can convert a per-coarse-vertex `float` into a per-fine-vertex `float`. Invoking a plumbing operator—whether implicitly or explicitly—causes an attribute (a node in the shader graph) to be plumbed from one rate to another. When the compiler encounters a mismatch on rate qualifiers, it may insert calls to `implicit` plumbing operators to coerce a value from one rate to another.

In many cases, plumbing operators are defined as part of a rendering pipeline. For example, the D3D11 pipeline exposes plumbing operators which may be used to convert from `@RasterVertex` to `@Fragment` values. Because plumbing operators are named subroutines, the library can expose multiple methods of interpolation, e.g.:

```
implicit @Fragment float perspectiveInterpolate(
    @RasterVertex float attr );
@Fragment float centroidInterpolate(
    @RasterVertex float attr );
```

In this example, shader-graph code can opt in to centroid interpolation for a given value by calling the appropriate operator explicitly, or rely on implicit plumbing, which performs perspective-correct linear interpolation.

The signature of the plumbing operators above may be surprising. How can a function like `perspectiveInterpolate()` possibly take a single input when interpolation requires at least three values (for the three raster vertices that make up a triangle)?

These signatures, however, are correct from the point of view of pointwise shading code: `perspectiveInterpolate()` takes a per-raster-vertex attribute and converts it to a per-fragment attribute. More fundamentally, this function does not operate on particular concrete values (e.g., `2.0f`), but on *attributes*: nodes of the shader graph or, equivalently, fields of a record type. We can intuitively think of the parameter `attr` as representing the *name* of a graph node.

When defining a module that includes groupwise operations (e.g., a tessellation scheme) a programmer may also define plumbing operators that can interpolate per-coarse-vertex values to fine vertices. A user-defined plumbing operator may apply only to values of a specific type (e.g., `Point`s or `Normal`s), or can be "templated" to apply to any attribute. A tessellation scheme can thus define special-case interpolation for points, vectors, and normals, while also defining a templated operator to handle attributes of other types.

One important consideration when allowing libraries to define their own rates of computation is portability. For example, if a Direct3D 9 pipeline interface uses a `@Vertex` rate where D3D11 uses `@CoarseVertex`, then a single shader cannot trivially port between the two. Allowing shaders to port between different rendering pipelines will require conscientious library design.

Stage-Specific Operations

In order to support plumbing operators, we designed the Spark language to support functions with explicitly rate-qualified inputs and outputs. Once this support was in

place, it became clear that the same basic mechanism could also be used to define functions that are specific to a particular rate. For example, the screen-space derivative functions like `ddx()` should only be available to code running at per-fragment rate in the D3D11 pipeline:

```
@Fragment float ddx( @Fragment float value );
```

These "single-rate" functions can be used in the system-provided libraries to expose operations that should only be available to a certain rate, or set of rates. Users may also define their own single-rate functions; for example, a subroutine that needs to make use of the `ddx()` function must be marked as specific to the `@Fragment` rate.

3.3.5 Implement Record Types as Virtual Classes

As described in Section 3.2.3, our shader-programming abstraction relies on identifying rates of computation with record types. Thus, a declaration like:

```
@Fragment float nDotL = dot( N, L );
```

is viewed both as defining an attribute `nDotL` with `@Fragment` rate, but also as defining a `float`-type field `nDotL` in the `Fragment` record type.

We have already described how a derived shader class inherits all the attributes (shader-graph nodes) of its base class(es). If a `@Fragment` attribute in the base class corresponds to a field of its `Fragment` record type, then this means that the derived class's `Fragment` type must also have these fields. Any additional `@Fragment` attributes defined in the derived class serve to extend the derived definition of `Fragment`.

In order to provide a robust conceptual grounding for this behavior, we chose to model the semantics of Spark's record types as a restricted form of virtual classes, as introduced in Section 2.3.2.

As discussed in Section 3.2.3, the computations with, e.g., the `@RasterVertex` rate define the operations to be performed when constructing a `RasterVertex` value. We take inspiration from the flexible approaches to initialization described in Section 2.3.3, and synthesize initialization code for record types automatically from the corresponding shader-graph nodes. Attributes with the `input` qualifier correspond to constructor parameters for the corresponding record type. A given class might inherit multiple such attributes, with no clear ordering implied; Spark uses a `key:value` syntax for passing arguments when constructing records to avoid needing to define an order.

3.3.6 Define Plumbing Operators Using Projection

We will illustrate how Spark can be used to implement plumbing operators with a brief example. Suppose the user wishes to define an operator to plumb values from control points to fine vertices, using linear barycentric interpolation over triangles:

```
implicit @FineVertex float baryInterpolate(
    @ControlPoint float attr );
```

To support such a definition, the system library for the D3D11 pipeline exposes two built-in attributes (shader-graph nodes):

```
@FineVertex Array[ControlPoint, ...] DS_InputControlPoints;
@FineVertex float3 DS_DomainLocation;
```

The first of these declarations states that for every fine vertex (i.e., per-fine-vertex) there is an *array* of control points (comprising an input patch). The array size depends on the number of control points given by the user: in this case, three. The second declaration states that every fine vertex knows its barycentric location in the tessellation domain.

```
implicit @FineVertex float baryInterpolate(
    @ControlPoint float attr )
{
 // project 'attr' out of each control point
 @FineVertex float a0 = attr @ DS_InputControlPoints(0);
 @FineVertex float a1 = attr @ DS_InputControlPoints(1);
 @FineVertex float a2 = attr @ DS_InputControlPoints(2);

 // barycentric linear interpolation
 return a0 * DS_DomainLocation.x
      + a1 * DS_DomainLocation.y
      + a2 * DS_DomainLocation.z;
}
```

Listing 3.1: *Example Spark plumbing operator. The per-control-point attribute* ***attr*** *is* projected *out of three particular control points and then interpolated.*

Listing 3.1 shows how a user-defined plumbing operator takes advantage of these system-defined attributes to perform interpolation across several control points. We fetch each control point from the array—each of these values is a `ControlPoint` record. We then *project* out the value of a particular field by using `@` as an infix operator. Note that this is projection in the sense of the relational algebra [Cod70], rather than geometry.

Projection relies fundamentally on the interface between shader graphs and record types defined in Section 3.2.3. In particular, if `attr` represents a per-control-point attribute (that is, a shader-graph node with per-control-point rate), then it corresponds to a field in the `ControlPoint` record type. If, as in Section 3.3.3, we think of `attr` as holding the *name* of a particular field, then the projection `attr @ cp` fetches the field with that name for a particular control point.

When performing projection, the `@` character may be read as "at," so that, e.g., `color @ vertex` yields the value of a per-vertex color *at* a particular vertex. The use of `@` to both indicate rates of computation and to perform projection is meant to be similar in spirit to C, where `*` is used both when declaring pointer variables and when dereferencing them.

If record types can be thought of like `struct` types, then it might seem that we should instead use more conventional syntax like `cp.attr`. The scoping rules for our projection operation, however, do not match those for C's "dot operator." In particular, `attr` in Listing 3.1 is a function parameter in local scope. Each time `baryInterpolate()` is invoked, this parameter may refer to a *different* field of the `ControlPoint` type. As such, we use distinct syntax to reflect the distinct semantics.

Alternative Approaches

Our approach to exposing plumbing in Spark relies deeply on our choice to expose record types explicitly, and identify record types with rates of computation. However, at the earliest stages of our design process, we did not have a notion of record types and instead sought to expose plumbing entirely in terms of rates.

For example, rather than expose the many-to-one relationship between control points and fine vertices as an array of explicit `ControlPoint` records:

```
@FineVertex Array[ControlPoint, ...] DS_InputControlPoints;
```

we could instead expose a multi-parameter plumbing operator:

```
@FineVertex T CP2FV[type T]( @ControlPoint T attr,
                             @ControlPoint int idx );
```

With this `CP2FV()` operator available, a user can define a `baryInerpolate()` operator equivalent to the one in Listing 3.1, without the need for projection. For example, we could fetch the value of an attribute for the three control points by writing:

```
@FineVertex float a0 = CP2FV(attr, 0);
@FineVertex float a1 = CP2FV(attr, 1);
@FineVertex float a2 = CP2FV(attr, 2);
```

Another alternative would be to have a plumbing operator that returns an array:

```
@FineVertex Array[T,3] CP2FV[type T](@ControlPoint T attr);
```

in which case the code inside the plumbing operator becomes:

```
@FineVertex float a0 = CP2FV(attr)(0);
@FineVertex float a1 = CP2FV(attr)(1);
@FineVertex float a2 = CP2FV(attr)(2);
```

As discussed above, we initially employed approaches like this, which allow users to define their own plumbing operators without exposing record types or projection as language concepts. We ultimately abandoned these alternatives in favor of exposing record types and projection directly. Several factors motivated our choice:

- We needed the notion of record types anyway, at the very least to be able to expose the GS stage.
- Each of these alternative plumbing operators is potentially confusing: the first because it obscures the use of an array; the second because applying the operator changes the type (and not just the rate) of the operand.
- In order to implement operators like these in our system library, we would end up using the mechanisms of record types and projection anyway. Exposing the mechanisms directly allows users to define their own rates, if desired.

3.3.7 Drive Rate Conversion by Outputs, Not Inputs

An important design choice is where the compiler should insert implicit conversions. For example, given a snippet of code like:

```
@CoarseVertex float3 N = ...;
@CoarseVertex float3 L = ...;
@Fragment float nDotL = dot( N, L );
```

is the dot product computed per-coarse-vertex or per-fragment?

RTSL derives the rate of an operation from the rates of its inputs: the dot product is computed per-vertex. This rule has an appealing simplicity, and a similar flavor to the rules for type promotion in C.

We initially applied this approach in Spark, but found that it had unintuitive consequences. In cases where a programmer *wants* to compute the above dot product per-fragment, they need to insert an explicit conversion (cast). Such cases arise often, however, and we found that with these rules even simple shaders required several explicit casts to achieve the desired behavior. Of greater concern, when we forgot to insert a cast, we would get no diagnostic messages—errors or warnings—from the compiler. Instead, a shader would silently compute some results at less than the desired rate, leading to visual artifacts.

Why didn't the RTSL designers encounter this problem? We suspect that the answer has to do with the limited capabilities of GPU fragment processors at the time. Shader authors targeting early programmable GPUs would tend to compute intermediate results per-vertex whenever possible. In this case, language rules which err on the side of efficiency rather than quality were likely a good match.

For Spark, though, we found that users expected the dot product above to be computed per-fragment. We eventually concluded that the rate at which a computation is performed must be driven by its *output* rather than its inputs. This decision was made with reluctance, since it too has unexpected consequences. Most notably, moving a sub-expression from one place to another can change the rate at which it is evaluated. This choice also means every user-declared node in the shader graph must have a rate qualifier specified, whereas RTSL allows many qualifiers to be inferred. We have still found, though, that this new rule more closely matches programmer intuition, and eliminates many explicit casts that would otherwise be needed.

3.3.8 Move Computations When Pipeline Stages Are Disabled

The D3D11 pipeline allows the HS, DS, and GS stages to be disabled by binding a "null" kernel. In Spark, a shader class must opt in to use of these pipeline stages by inheriting from system-defined mixin shader classes `D3D11Tessellation` and/or `D3D11GeometryShader`. If a class does not inherit from these, directly or indirectly, the corresponding stages are disabled.

If the user disables, e.g., the GS, what should happen to `@RasterVertex` computations in their shader graph? In our interface to the D3D11 pipeline we take advantage of the fact that record types and pipeline stages are decoupled. When the GS is disabled, our D3D11 back-end moves the construction of `RasterVertex` records to the DS stage instead. If the tessellation (HS and DS) stages are also disabled, then all computation on the different flavors of vertices will be executed in the VS stage. This design has similar properties to the shader framework of Kuck and Wesche [KW09], where operations defined in the "Post Geometry" stage are executed in the VS if no GS effect is active.

Moving computations in this manner has the drawback that the mapping from the shader-graph abstraction to the rendering pipeline becomes more complicated. This complexity may make it difficult for a shader writer to decide what rate to give to particular computations. An important benefit, however, is that it is possible to write pointwise shading operations that can be used both with and without tessellation or GS effects. For example, a displacement effect that operates at `@FineVertex` rate will work with both tessellated and untessellated models.

3.3.9 A Language for Configuring the Entire Pipeline

A concrete Spark shader class (that is, one with no remaining `abstract` members) defines a complete configuration of the rendering pipeline—including both programmable and fixed-function stages. This is in contrast to most prior work.

When rendering with a shader-per-stage language, the configuration of the pipeline is driven both by the particular shaders that are bound, as well as by fixed-function state configured through a C or C++ API. The Cg and D3D "effect" systems allow per-stage shaders to be bundled together with state for *most* of the fixed-function stages, although the configuration of vertex assembly (the IA pipeline stage in D3D11) and binding of render targets are left to C/C++ code.

Our primary motivation for putting *all* configuration into the Spark language was to be able to statically check and ensure the validity of a pipeline configuration. We can ensure, for example, that the IA pipeline stage outputs all of the vertex attributes required by the VS stage, because we generate the configuration of both stages from the same program.

One challenge we faced in achieving this goal was that some fixed-function stages are "almost" programmable. Specifically:

- The IA stage configuration specifies zero or more vertex attributes, each with a name and type. Each attribute may be fetched from one of several user-defined vertex streams. The index used for the fetch is either the system-provided per-vertex ID, or the system-provided per-instance ID (optionally divided by a user-defined constant).

- The OM stage configuration specifies zero or more blending setups, each corresponding to a pair of a PS-stage output and a bound render target. For each target, and for both the color and alpha channels of that target, the user may specify a computation—e.g., sum, difference, minimum, or maximum—as well as two multiplicative factors, one each for the source fragment and destination pixel.

In each of these cases, the fixed-function stage produces/consumes records with attributes that must match—in number, name, and type—those consumed/produced by an adjacent programmable pipeline stage.

In order to simplify the configuration and validation of the IA and OM stages, we allow them to be configured in a Spark program *as if* the stages are programmable. For example, the following code implements a standard premultiplied alpha blend:

```
@Fragment float4 color = ...;
output @Pixel float4 target =
  (target @ OM_Dest) * color.w + color;
```

Here `OM_Dest` refers to the destination pixel for the blend operation (somewhat confusingly, of type `@Pixel Pixel`), so that `target @ OM_Dest` refers to the value stored in the render target `target` for that pixel. We scale that value by the last component (the alpha value) of our per-fragment color, and then add it to the color, to achieve the final value to be written back to the render target.

One down-side to our approach is that our compiler implementation must be able to map high-level Spark code for blending or vertex-assembly operations to the low-level configuration structures required by the rendering pipeline. We currently achieve this by a simple pattern-matching approach, but have found that it is difficult to provide good diagnostic messages to users when pattern-matching fails; simply telling a user that their blending logic is "too complex" is undesirable.

One additional benefit of providing a uniform interface to the rendering pipeline is that all input shader parameters in Spark take the form of `input` attributes declared in a shader class (e.g., `input @Uniform` in the case of D3D11) and all output parameters are `output` attributes (`output @Pixel` in D3D11). This highly uniform representation is in contrast to, e.g., HLSL where different syntax is used depending on the type of parameter: a uniform matrix, texture resource, and render target would each be declared differently.

3.4 Example Spark Shaders

In this section we present several complete Spark shaders, targeting key features of the D3D11 pipeline. With each example, we will try to highlight relevant features of the Spark language.

3.4.1 A Minimal Complete Shader

In order to impart the flavor of the Spark language, we present a brief code example in Listings 3.2 and 3.3. The `Base` shader class fetches and transforms vertices, while `Displace` and `Shade` extend `Base` with displacement mapping and simple texture mapping, respectively. This decomposition separates the concerns of displacement and texturing: each can be defined and used independently. The shader class `Example` composes the two concerns, and yields a shader graph similar to that in Figure 3.1.

Each shader class is declared with the `shader class` keywords. The `Base` class extends `D3D11DrawPass`, a shader class defined as part of our Spark system library, and implemented with support from the compiler. Inheriting from this class means that `Base` can make use of types, operations, and rates of computation defined by the D3D11 interface. This includes a number of types (e.g., `float4x4`) and operations (e.g., `mul`) that are familiar to users of HLSL. In addition, `Base` inherits a number of rates of computation, such as `@CoarseVertex` and `@FineVertex`. These types, operations, and rates are defined by the `D3D11DrawPass` class, rather than by the Spark language syntax.

Note that the `VertexStream` type is "templated" on a user-defined `struct` type. Spark uses square brackets `[]` rather than angle brackets `<>` to enclose type parameters. As a consequence indexing operations, such as fetching from `vertexStream`, use ordinary function-call syntax.

```
shader class Base extends D3D11DrawPass
{
    input @Uniform float4x4     modelViewProjection;
    input @Uniform uint         vertexCount;
    input @Uniform SamplerState linearSampler;

    // Stream of vertices in memory
    struct PNuv { float3 P; float3 N; float2 uv; }
    input @Uniform VertexStream[PNuv] vertexStream;

    // Bind number and type of primitives to draw
    override IA_DrawSpan = TriangleList(vertexCount);

    // Per-coarse-vertex - fetch from buffer
    @CoarseVertex PNuv assembled =
        vertexStream(IA_VertexID);
    @CoarseVertex float3 P_base  = assembled.P;
    @CoarseVertex float2 uv = assembled.uv;

    // Declare model-space position to be virtual
    virtual @FineVertex float3 P_model = P_base;

    // Bind clip-space position for rasterizer
    override RS_Position = mul(float4(P_model, 1.0f),
                               modelViewProjection);
}
```

Listing 3.2: *Example Spark shader class, for projection of vertex positions.*

The classes in Listings 3.2 and 3.3 define a number of attributes (shader-graph nodes). The `input` `@Uniform` attributes represent input shader parameters, while the `output` `@Pixel` attribute in `Shade` represents a shader output that should be captured in a render target. A shader class can `override` the definition of an inherited `abstract` or `virtual` attribute, whether the attribute is user- or system-defined (e.g., `P_model` and `RS_Position`, respectively).

```
mixin shader class Displace extends Base
{
    input @Uniform Texture2D[float3] displacementMap;

    // Per-fine-vertex - displace
    @FineVertex float3 disp =
        SampleLevel(displacementMap, linearSampler,
                    uv, 0.0f);

    override P_model = P_base + disp;
}

mixin shader class Shade extends Base
{
    input @Uniform Texture2D[float4] colorMap;

    // Per-fragment - sample color
    @Fragment float4 color = Sample(colorMap,
                                    linearSampler, uv);
    // Per-pixel - write to target
    output @Pixel float4 target = color;
}

shader class Example extends Displace, Shade {}
```

Listing 3.3: *Extensions of the shader class in Listing 3.2. The* `Example` *class corresponds approximately to the shader graph in Figure 3.1.*

3.4.2 C++ Interface

Listing 3.4 shows examples of C++ wrapper classes generated by the Spark compiler. The Spark shader classes `Base` and `Example` from Listings 3.2 and 3.3 are reflected as equivalent C++ wrapper classes. Note that we elide the `Displace` and `Shade` shader classes here; we will discuss the translation of `mixin` shader classes in Section 4.1.4.

Listing 3.5 shows how the C++ class `Example` can be used for rendering in an application. When creating an instance of the class, we specify a particular rendering device. Creating the instance might entail compilation of GPU code or allocation of other resources for the device. It is a heavyweight operation, and thus should be avoided at runtime.

```
class Base :
    public spark::d3d11::DrawPass
{
public:
  // @Uniform inputs
  void SetModelViewProjection( spark::float4x4 value );
  void SetVertexCount( UINT value );
  void SetLinearSampler( ID3D11SamplerState* value );
  void SetVertexStream( spark::VertexStream value );

  // ...
};

class Example :
    public Base
{
public:
  // @Uniform inputs
  void SetDisplacementMap( ID3D11ShaderResourceView* value );
  void SetColorMap( ID3D11ShaderResourceView* value );

  // @Pixel outputs
  void SetTarget( ID3D11RenderTargetView* value );

  // Submit
  void Submit( ID3D11Device* device,
               ID3D11DeviceContext* context );

  // ...
};
```

Listing 3.4: *Spark compiler-generated C++ wrapper classes. These classes correspond to the Spark shader classes* `Base` *and* `Example` *in Listings 3.2 and 3.3, respectively.*

```
// At startup:
ID3D11Device* d3dDevice = ...;
spark::IContext* context = ...;
Example* instance =
    context->CreateShaderInstance<Example>(d3dDevice);

// At runtime:
ID3D11DeviceContext* d3dContext = ...;
//     Set @Uniform inputs:
instance->SetModelViewProjection( mvp );
instance->SetVertexCount( 3 );
instance->SetVertexStream( vertexBuffer );
instance->SetLinearSampler( samplerState );
instance->SetDisplacementMap( dispTex );
instance->SetColorMap( colorTex );
//     Set @Pixel outputs:
instance->SetTarget( renderTarget );
instance->SetDepthStencilView( depthStencil );
//     Submit:
instance->Submit( d3dDevice, d3dContext );
```

Listing 3.5: *Rendering with a Spark shader, using compiler-generated C++ wrapper.*

Once the shader instance is created, its `input @Uniform` and `output @Pixel` parameters may be bound. This is done using the compiler-generated accessor functions in the C++ wrapper class.

To render with the shader instance, we submit it to a D3D11 rendering context (either immediate or deferred). No other parameters or state-setting operations are required; a concrete shader class specifies (or inherits) a complete configuration for the rendering pipeline.

3.4.3 Tessellation

Listing 3.6 shows a shader class `Tessellate` that implements a simple tessellation effect as a `mixin` shader class. This shader class depends only on a system-provided shader class, and so can easily be reused in a variety of contexts. In particular,

```
mixin shader class Tessellate extends D3D11TriTessellation
{
  // Take triangle 0-ring as input,
  // produce triangle patch as output
  override HS_InputCoarseVertexCount = 3;
  override HS_OutputControlPointCount = 3;

  // Uniform tessellation rate
  input @Uniform float tessFactor;
  override HS_EdgeFactor = tessFactor;
  override HS_InsideFactor = tessFactor;

  // Parameters for fixed-function tessellator
  override TS_Partitioning = IntegerPartitioning;
  override TS_OutputTopology = TriangleCWTopology;

  // Plumbing operators
  @ControlPoint T CoarseToControlPoint[type T](
    @CoarseVertex T value )
  {
    return value @ HS_InputCoarseVertices(HS_ControlPointID);
  }

  override implicit @FineVertex T CoarseToFine[
    type T, implicit Linear[T]](
    @CoarseVertex T value )
  {
    @ControlPoint T cpValue = CoarseToControlPoint(value);
    @FineVertex T v0 = cpValue @ DS_InputControlPoints(0);
    @FineVertex T v1 = cpValue @ DS_InputControlPoints(1);
    @FineVertex T v2 = cpValue @ DS_InputControlPoints(2);
    return v0 * DS_DomainLocation.x
         + v1 * DS_DomainLocation.y
         + v2 * DS_DomainLocation.z;
  }
}
```

Listing 3.6: *Example Spark tessellation effect, implementing linear barycentric interpolation over triangular patches. The* `Tessellate` *shader class can be freely combined with effects such as* `Displace` *in Listing 3.3 without explicit dependencies.*

we note that `Tessellate` can be combined with `Displace` from Listing 3.3 without either shader class having explicit knowledge of the other. Because the displacement computation in `Displace` is performed at `@FineVertex` rate, it will automatically apply to post-tessellation vertices when tessellation is enabled, and to untessellated vertices otherwise (see Section 3.3.8).

Looking at the `Tessellate` shader in Listing 3.6 in more detail, we see that it inherits from the `D3D11TriTessellation` shader class, indicating that tessellation should be performed on a triangular (u, v, w) domain. Even when using a triangular *domain*, the tessellation stages in the D3D11 pipeline can operate on patches with any number of control points (`HS_OutputControlPointCount`), computed from a neighborhood of coarse-mesh vertices of any size (`HS_InputCoarseVertexCount`). By specifying concrete values for these inherited attributes, we control the amount of data that the HS stage consumes and produces on each invocation. In this case, we only care about a 0-ring neighborhood of each input triangle (three vertices), and will produce output patches with three control points (one for each vertex).

Next, we introduce a shader parameter to define a uniform tessellation rate, and bind this to both the edge and interior tessellation factors required by D3D11. We also bind values to the attributes that control the fixed-function tessellation (TS) pipeline stage: namely, the partitioning scheme and desired output primitive topology.

Finally, we define two plumbing operators, to help with plumbing data from coarse (pre-tessellation) to fine (post-tessellation) vertices. The `CoarseToControlPoint()` operator is an explicit operator to plumb data from the per-coarse-vertex to the per-control-point rate. Since in this example coarse vertices and control points are in one-to-one correspondence, the operator is trivial. The `CoarseToFine()` operator overrides an `abstract` operator introduced by the `D3D11TriTessellation` base class; every concrete shader that performs tessellation must define this operator. The `CoarseToFine()` operator proceeds in three steps:

- First, the desired attribute `value` is plumbed from coarse vertices to control points, using the previously-defined `CoarseToControlPoint()`.

- Second, the per-control-point attribute `cpValue` is used to *project* (see Sections 3.3.6 and 5.1.6) the value of this attribute out of the three patch control points provided to the DS.
- Third, the system-provided barycentric coordinates `DS_DomainLocation` are used to compute a linear combination of the per-control-point values.

Both of these plumbing operators are parameterized on a type `T`. The `CoarseToFine()` operator additionally includes a *constraint* that the type `T` supports the `Linear` *concept*. In this case, the use of `Linear` simply means that the plumbing operator only applies to types `T` which support addition, and multiplication by scalars. A more detailed discussion of constraints and concepts is beyond the scope of this dissertation, but interested readers should note that they are related to the proposed C++ concept feature [SGG+05], and implicit parameters in Scala [OMO10].

3.4.4 Geometry Shader

Listing 3.7 shows a shader class `PointSprites` that uses the Geometry Shader pipeline stage to implement the core of a "point sprite" rendering effect. In point sprite rendering, the application submits point primitives, which are then expanded into quadrilateral "billboards." Depending on the use-case, these billboards might be oriented to face the camera and aligned with the screen-space X and Y axes (e.g., to render particles), or have other constraints placed on their orientation (e.g., to render a level-of-detail "imposter" for a tree model).

Rather than implement a particular use-case for point sprites, the `PointSprites` class instead implements the core behavior shared by most effects. It defines a Geometry Shader procedure that takes in one fine vertex, and produces four raster vertices. To allow per-fine-vertex attributes to be plumbed to raster vertices, it defines the `FineToRaster()` plumbing operator. The implementation of this operator is trivial, as every raster vertex in a generated quad receives a copy of the fine vertex that defined the input point. In addition, each raster vertex in the quad receives a unique (u, v) parameter that identifies one corner of the quad. A particular concrete point-sprite effect would then mix in the `PointSprites` class and use the `uv` parameter to compute its projected position, sample texture data, etc.

```
mixin shader class PointSprites extends D3D11GeometryShader
{
  // take points as input, output quads (2-triangle strip)
  override GS_InputVertexCount = 1;
  override GS_MaxOutputVertexCount = 4;

  // Geometry Shader procedure
  override @GeometryOutput void GeometryShader()
  {
    @GeometryOutput FineVertex f = GS_InputVertices(0);
    Append( GS_OutputStream,
      RasterVertex( fv: f, uv: float2(0,0) ) );
    Append( GS_OutputStream,
      RasterVertex( fv: f, uv: float2(0,1) ) );
    Append( GS_OutputStream,
      RasterVertex( fv: f, uv: float2(1,1) ) );
    Append( GS_OutputStream,
      RasterVertex( fv: f, uv: float2(1,0) ) );
  }

  // Plumbing
  input @RasterVertex FineVertex fv;
  override @RasterVertex T FineToRaster[type T](
    @FineVertex T value )
  {
    return value @ fv;
  }

  // Point-sprite parameter (per-raster-vertex)
  input @RasterVertex float2 uv;
}
```

Listing 3.7: *Example Spark Geometry Shader effect, implementing the core of a "point sprite" rendering effect. The* `PointSprites` *shader uses the GS stage to expand points into quads, and provides a parameter* `uv` *that can be used to transform or shade the generated quads.*

Chapter 4

The Spark System

Having completed our discussion of the design of the Spark language, we now turn our attention to the system we have created for compiling, composing, and running Spark programs. In this chapter, we describe our implementation of the Spark compiler, library, and runtime, and discuss our experience using Spark to implement several shading workloads.

4.1 Implementation

In this section we describe our implementation: both the overall architecture, and important implementation details. Our compiler and runtime for the Spark language are implemented in a combination of C# and C++ code. In particular, the core of the compiler is implemented in C# so that we can take advantage of garbage collection, reflection, and other niceties of modern VM-based languages. Applications interface with Spark through a pure C++ API, and need not be aware of the use of C#.

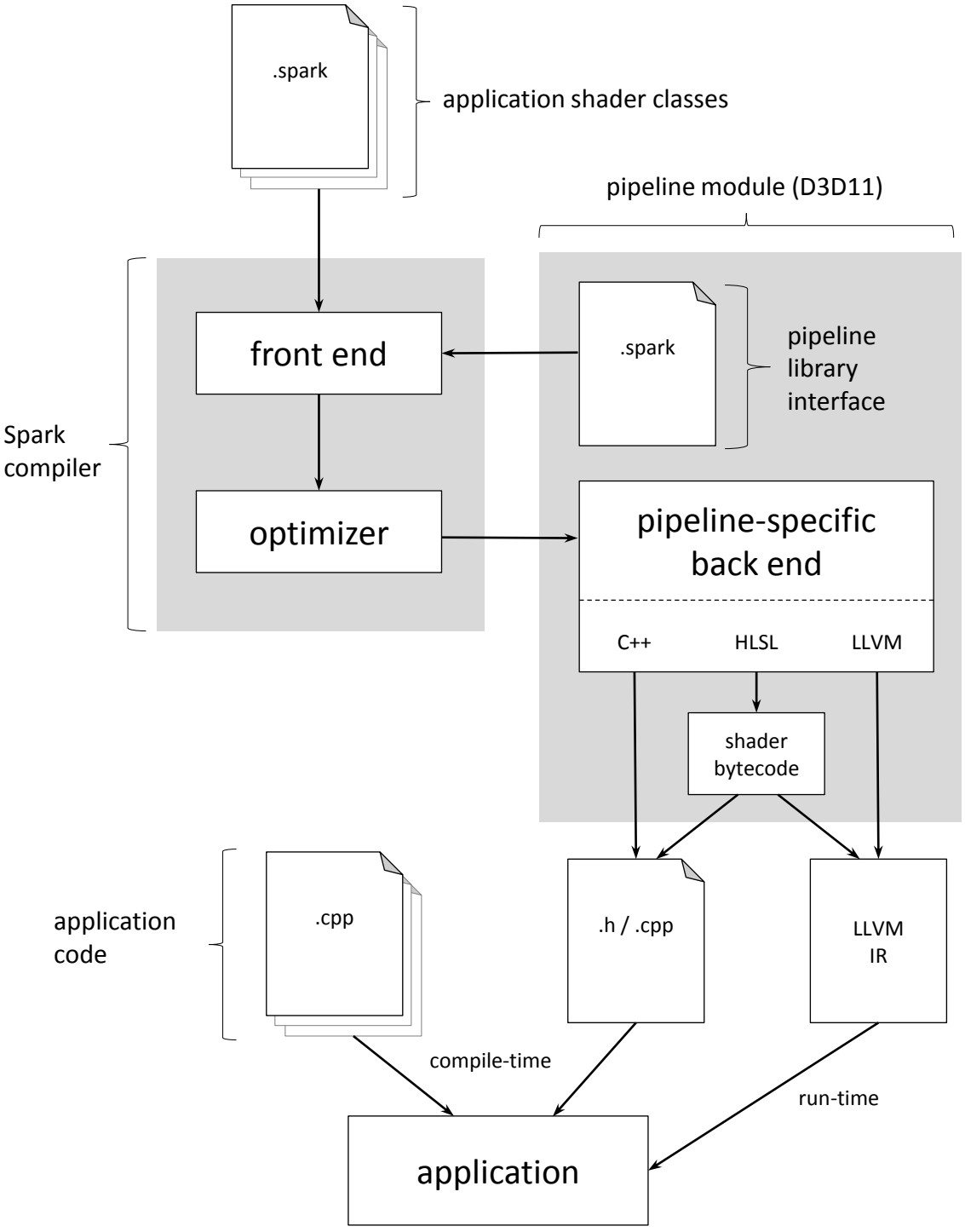

Figure 4.1: *System block diagram. Application shaders are type-checked against a pipeline-specific library interface. Global optimizations are applied to each shader class. Executable CPU and GPU code are generated by a pipeline-specific back end.*

4.1.1 Architecture

Figure 4.1 shows the structure of the Spark system. In order to compile a shader, the core Spark compiler coordinates with a *pipeline module* for a particular rendering architecture. The pipeline module defines the interface to a given rendering pipeline (e.g., D3D11) as a system library of Spark code, comprising one or more shader classes. These shader classes declare the types, functions, rates of computation, and plumbing operators for the pipeline. Some functions and operators are implemented in ordinary Spark code, but in other cases implementations are left out of this library and are instead provided by the pipeline-specific back end. We have so far implemented a pipeline module for the D3D11 rendering pipeline.

The user-defined shader classes for an application are parsed and type-checked together with the interface provided by the pipeline module. The type-checking step is responsible for assigning rates to all intermediate computations (see Section 3.3.7), and ensuring that any conversions between rates are supported by appropriate plumbing operators. Because all of the built-in rates of computation and plumbing operators for a pipeline are provided by the system library in the pipeline module, the type-checking logic in the Spark compiler is not tied to a particular rendering architecture, and can be re-used for new or extended pipelines.

4.1.2 Optimization

Once the shader code has been type-checked, the Spark compiler performs global optimizations on each shader class. The most important of these optimizations is *dead-code elimination* (DCE) over the shader graph.

It is important to note that by performing DCE on the shader graph we can eliminate code more aggressively than if we performed DCE only on per-stage shaders. For example, a shader might fetch and transform tangent vectors per-coarse-vertex. A shader-per-stage compiler cannot tell if tangent vectors output by a VS procedure will be used by downstream computations or not. In contrast, the Spark compiler can

easily see if *any* value computed in the shader graph has no uses, and eliminate it (we must be careful, of course, not to remove any operations that might have side-effects; see Section 5.4.1).

A rendering system based on per-stage shaders could, of course, perform inter-stage DCE at run-time, once the set of per-stage shaders is known. This optimization would come at the cost of a clear phase separation (see Section 3.1.2), and could result in unpredictable pauses when switching shaders at run-time.

We do not perform most simple expression optimizations (constant folding, algebraic simplifications, etc.) because our current back ends use source-to-source translation; we can rely on the more complete optimizers in the down-stream compilers to improve code quality. We do, however, perform simple *common subexpression elimination* (CSE), by using *value numbering* when building the shader graph. This optimization is motivated by a very specific problem: when our type-checker inserts implicit plumbing code, it may end up plumbing the *same* value at more than one place in the shader graph. Without our CSE optimization, we could end up communicating redundant data between pipeline stages, consuming both bandwidth, and storage in inter-stage streams.

In order to make our DCE and CSE optimizations more effective, we "flatten" the inheritance hierarchy before optimization. For each shader class, we perform a *deep copy* of the code it inherits, and then optimize this copy. The result of this approach is that use of inheritance and `virtual` members does not negatively impact the performance of generated code. It may, however, lead to increased compile times and memory usage.

4.1.3 Code Generation

Once a shader class has been optimized, it is passed to a back end in the pipeline module for code generation. The primary concern of our D3D11 pipeline module is to translate the optimized Spark shader graph into per-stage shader code in HLSL, which is then compiled to D3D11 shader bytecode.

At a high level, our code generation approach is similar to that of RTSL, but differs in how we map from shader-graph to per-stage code. RTSL partitions the shader graph directly according to rates of computation (e.g., mapping computations with the `vertex` rate to the vertex-processing stage and creating inter-stage interpolants for every `vertex`-to-`fragment` edge). In contrast, Spark relies on the dual representation between record types and rates of computation (see Section 3.2).

As a brief refresher:

- Each rate of computation in the shader graph is identified with a corresponding record type.
- The attributes (nodes) in the graph with a given rate correspond, conceptually, to fields of the corresponding record type (and in some cases, constructor parameters).
- Constructing an instance of record type `R` has the effect of performing those computation in the shader graph with rate `@R`, and collecting the result values in a record.
- The stages of the pipeline run per-stage shader procedures, which are responsible for constructing records as needed.

Given this background, we now describe the overall flow of our translation to HLSL.

Record Types

A Spark record type like `RasterVertex` is translated into an HLSL "connector" `struct`. For example, `RasterVertex` would be a connector output by the GS stage and input to the PS. As an optimization, only those attributes with `@RasterVertex` rate that are actually used in down-stream computations will be included in the HLSL struct. We also define an HLSL function for the `RasterVertex` constructor, which takes parameters corresponding to any `input` attributes with the corresponding rate; the body of the constructor executes the corresponding shader-graph code, stores the resulting values in a `RasterVertex` structure, and returns it.

Shader Procedures

In the general case, a stage like the GS makes use of a user-defined procedure that can construct and output zero or more `RasterVertex` records. Since we expose the GS procedure directly in Spark (see Section 3.3.1), we generate HLSL code for the GS stage simply by translating this user-defined procedure. Code in the GS procedure that constructs `RasterVertex` records will call the generated `RasterVertex` constructor function described above, and thereby execute any shader-graph code with the `@RasterVertex` rate.

As discussed in Section 3.2.2 we can consider other programmable pipeline stages as having system-defined procedures (e.g., a VS procedure that constructs a single `CoarseVertex`). We can thus generate HLSL code for these stages by translating the system-defined procedure with the same approach as used for the GS.

This approach means that ideally we should be able to define all of the per-stage procedures directly in the Spark language (whether in user or system code), and employ a uniform strategy for translation to HLSL. A number of practical issues get in the way of this approach:

- The syntax and rules of the HLSL language aren't always consistent: different shader stages require that their inputs and outputs be declared in specific, idiosyncratic styles.
- In the particular case of the HS stage, the HLSL compiler applies an auto-parallelization technique that requires a particular idiomatic style of input for best results.
- When optional pipeline stages are disabled, we need to change the per-stage procedures to move computation to the remaining stages.
- We expose some stages (the IA and OM) as "almost" programmable, but as they are not programmable in HLSL, we must employ another translation strategy.

We will now describe these implementation issues in more detail.

Hull Shader

The HS stage uses both a per-control-point procedure (constructing a `ControlPoint`) and a per-patch procedure. The per-patch procedure is responsible for performing the computations associated with several Spark rates of computation, corresponding to patch corners, edges, and interior axes (in the case of quadrilateral domains), as well as constructing an output `Patch`. The HLSL compiler employs an auto-parallelization approach that finds parallelizable loops in the per-patch procedure and translates them into fine-grained fork/join parallelism in the generated shader bytecode. For example, the compiler will detect when a loop over the corners of a patch is parallelizable, and generate bytecode which processes all of the corners in parallel. In order to exploit this facility, our compiler needs to generate HLSL loops with the particular kind of structure that the auto-parallelizer will detect. In practice, achieving this result is one of the most complex parts of our implementation, but we have confirmed that our code-generation strategy allows the HLSL compiler to detect fine-grained parallelism between independent patch corners and edges for Spark shaders.

One additional implementation detail that pertains to the HS stage is that we chose to expose a `@InputPatch` rate in Spark, that is convertible to both the `@ControlPoint` and `@Patch` rates. We currently support this rate by duplicating computation: we construct an `InputPatch` record as the first step in both the per-control-point and per-patch procedures that we generate. This abstraction has proved convenient for defining computations that should be available to both per-control-point and per-patch logic, even though the required dataflow is not directly supported by D3D11.

A similar modification could be used to express a rate for per-instance computation (e.g., `@Instance`) when using geometry instancing in the IA stage. Per-instance computation would ultimately map to the VS stage (the earliest programmable stage in the pipeline).

Disabling Stages

As described in Section 3.3.8, we allow the user to opt out of certain pipeline stages, and move computation accordingly. For example, if a shader makes no use of the HS, DS, or GS pipeline stages then all `@CoarseVertex`, `@FineVertex`, and `@RasterVertex` computation must be mapped to the VS pipeline stage. This can be achieved by generating code from different per-stage procedures depending on which pipeline stages are enabled. For example, a VS procedure that constructs a `CoarseVertex`, `FineVertex`, and `RasterVertex` in order—passing each into the constructor of the next—can be used when all of the HS, DS, and GS are disabled.

Input Assembler and Output Merger

As discussed in Section 3.3.9, Spark allows the fixed-function IA and OM pipeline stages to be configured using code rather than data. We currently generate configuration data for these stages from code with the `@AssembledVertex` and `@Pixel` rates using *ad hoc* pattern matching.

This has been sufficient for our needs so far, but is not entirely satisfactory. When code does not conform to the constraints of these stages, we issue an error diagnostic. Unfortunately, since we are working with optimized code at this point in the compiler, we often do not have useful source location information to report to the user.

Plumbing

Plumbing operators implemented entirely in Spark code do not cause complications for code generation; they are always inlined as part of our optimization process. Subsequently, the code generator only sees code that directly manipulates records, and applies projection to them. Both of these are trivial to map to HLSL, since we convert record types to HLSL `struct`s. In particular, projection in HLSL can always (after optimization) be translated into a simple field reference on an HLSL `struct`.

Some plumbing operators in the system library cannot be implemented directly in Spark, since they expose capabilities implemented in fixed-function stages. The most notable example of this is interpolation from `@RasterVertex` to `@Fragment` rate, which is provided by the RS stage, but declared as qualifiers on PS stage inputs. We handle these cases in an *ad hoc* fashion.

Host Code

In addition to generating D3D11 shader bytecode via translation to HLSL, we also generate code for the host CPU, as either C++ source or LLVM IR [LA04]. The generated code includes routines to be used by an application for initializing shader instances, and for submitting rendering operations.

The initialization code is responsible for:

- Performing any `@Constant` operations from the shader graph.
- Allocating a constant buffer and any fixed-function state objects required by the shader (e.g., IA, blending, sampler, rasterizer, or depth-stencil states).
- Loading shader bytecode through the D3D11 interface to generate optimized machine code for per-stage shaders.

In order to support this initialization code, we embed the compiled D3D11 shader bytecode into the generated C++/LLVM as static data.

The submission code is responsible for:

- Performing any `@Uniform` operations from the shader graph.
- Filling in the constant buffer with shader parameters.
- Binding constant buffers, textures, render targets, and other resources.
- Binding state objects for fixed-funtion pipeline stages.
- Binding shader procedures for programmable pipeline stages.
- Submitting a rendering command (e.g., a D3D11 `DrawPrimitive()` call).

```
shader class Base extends D3D11DrawPass
{
  input @Uniform float4x4 view;
}

mixin shader class ColorMixin extends Base
{
  input @Uniform float4 color;
}

shader class Derived extends Base, ColorMixin
{
  output @Pixel float4 target = color;
}
```

Listing 4.1: *Spark shaders using* `mixin` *inheritance.*

4.1.4 Wrapper Generation

An application that uses Spark is statically compiled against a header file generated by the Spark compiler. This header file defines the interface to each shader class, as discussed in Section 3.3.2. In particular, the C++ wrapper class exposes *setter* functions for each of the shader's parameters (e.g., `input @Uniform` attributes), along with a `Submit()` function for performing rendering.

One complication that arises in the generation of C++ wrappers is multiple inheritance. Spark implements multiple inheritance (of `mixin` shader classes) using a linearization approach, as described in Sections 2.3.1 and 3.3.2. To accurately reflect the Spark inheritance graph in our C++ wrappers, we would need to make heavy use of C++ multiple inheritance and `virtual` inheritance. Given the limitations of C++'s approach to multiple inheritance, we instead use the following strategy when generating C++ wrappers:

- The inheritance of primary (non-`mixin`) Spark shader classes is directly reflected in the inheritance of the C++ wrapper classes. Since Spark limits primary shader classes to a single-inheritance tree, this creates no complications.

- When a shader class `C` inherits from a `mixin` shader class `M`, we replace the inheritance (*is-a*) relationship with aggregation (*has-a*). That is, we give the C++ wrapper `C` a data member of type `M`.

This approach is illustrated in Listings 4.1 and 4.2. Because the C++ wrapper for a shader class `C` does not directly inherit from a mixin class `M`, we cannot directly use an instance of `C` where an `M` is expected. We work around this limitation in two ways. First, we generate functions in `C` for setting shader parameters that *forward* to the contained instance of `M`. Second, we provide a templated `StaticCast()` function that can be used on a `C` instance to cast it to any type it inherits from in Spark (e.g., `M`), whether or not it does so in C++.

4.1.5 Runtime Loading and Composition

The compiled implementations of shader classes can either be linked into the application as C++ code, or loaded at run-time as LLVM IR. Supporting dynamic loading of shader code is made difficult by the fact that application code is statically compiled using the header files generated by the Spark compiler. If the Spark compiler has already been run to generate the headers, why not just embed the generated shader code as well?

Two main use-cases motivate runtime loading of shader code. First, a programmer may wish to be able to alter the *implementation* of one or more shaders, in a way that does not affect their C++ *interface*. In this case, it should ideally be possible to modify and reload the Spark shader code on the fly without recompiling the application. Such rapid iteration can be important when fine-tuning or debugging shader code.

Second, in many cases a programmer can define a suite of shading effects as a library of many `mixin` shader classes in Spark, and generate C++ wrappers for these mixins. They can then load this shader code at runtime and use it to compose *new* shader classes on the fly. Since the new shader classes are strictly compositions of the existing mixins, no new wrapper interfaces need to be generated for these dynamically-composed classes.

```
class Derived : public Base
{
public:
    // Setter inherited from Base
    // void SetView( spark::float4x4 value );

    // Setter generated for parameter declared in Derived
    void SetTarget( ID3D11RenderTargetView* value );

    // Setter that forwards to ColorMixin
    void SetColor( spark::float4 value ) {
        this->StaticCast<ColorMixin>()->SetColor( value );
    }

    // Helper function to cast to other Spark
    // shader class types
    template<typename T>
    T* StaticCast()
    {
        return StaticCastImpl( (T*) NULL );
    }

protected:
    // Default case: allow cast to any type
    // that the C++ wrapper inherits from
    template<typename T>
    T* StaticCastImpl( T* dummy )
    {
        return this;
    }

    // Special case: allow case to mixin
    ColorMixin* StaticCastImpl( ColorMixin* dummy )
    {
        return &_M_ColorMixin;
    }

private:
    // Aggregate instance of mixin
    ColorMixin _M_ColorMixin;
};
// Primary class
```

Listing 4.2: *C++ wrapper code for `Derived` in Listing 4.1.*

Listing 4.3 shows a brief example of how the Spark runtime API can be used to compose shader classes on the fly. In this listing, a module of Spark code is loaded and compiled on the fly. Subsequently, the runtime representation of shader classes like `Base` and `ColorMixin` are looked up in this module, and used to create an array of shader classes to be composed. The runtime function `CreateShaderClass()` is used to construct a new, composed shader class from which we can construct and use instances. Because the new class is composed from pre-existing shader classes, we can interact with instances of the dynamically-generated class through the existing wrappers for `Base` and `ColorMixin`.

4.1.6 Limitations

While it is one of our design goals to support the full capabilities of the D3D11 rendering pipeline, our current implementation has some limitations. First we enumerate those limitations that pose no particular challenges to our design approach, and simply come from limited development time:

- At present, the Spark compiler assigns all `@Uniform` shader parameters to a single D3D constant buffer. Using multiple constant buffers is sometimes beneficial for performance.
- The current Spark system supports a `@Constant` rate, but doesn't support `input @Constant` parameters, even though their semantics are well-defined. All `@Constant` computations must be fully resolvable by the compiler.
- Our interface to D3D11 does not yet support the Stream Out (SO) pipeline stage.

As discussed briefly in Section 2.2.2, the Spark system does not currently expose atomic read-modify-write operations in shaders, through the Unordered Access View (UAV) feature of D3D11. Simply exposing the feature is not a challenge *per se*: Spark supports procedural subroutines, which may in theory perform operations with arbitrary side effects, and these subroutines may be invoked in the definition of the

```
spark::IContext* sparkContext = /* ... */;

// Load and compile shader-code module
spark::IModule* module =
    sparkContext->LoadModule("MyShaders.spark");

// Figure out what classes/mixins we want to use
spark::IShaderClass* mixins[MAX_MIXINS];
int mixinCount = 0;

mixins[mixinCount++] = module->FindShaderClass<Base>();
if( usingColorMixin )
    mixins[mixinCount++] =
        module->FindShaderClass<ColorMixin>();
// ...

// Create a new shader class from a list of classes/mixins
spark::IShaderClass* shaderClass =
    context->CreateShaderClass( mixins, mixinCount );

// Create an instance of the new shader class
spark::IShaderInstance* shaderInstance =
    shaderClass->CreateInstance( d3dDevice );

// Cast the instance to the generated C++ wrapper interfaces
Base* base = shaderInstance->DynamicCast<Base>();
ColorMixin* colorMixin =
    shaderInstance->DynamicCast<ColorMixin>();
```

Listing 4.3: *Runtime composition of a Spark shader class.*

declarative shader graph. If we were to use such an approach then the order of operations within such a procedure would be well-defined, but our implementation does not guarantee a particular evaluation order between shader-graph nodes. Determining whether such an implementation would suffice for the needs of users is future work; we discuss this situation and our conjectures in Section 5.4.1.

4.2 System Experience

In order to evaluate the Spark language, as well as our compiler and runtime implementation, we implemented several shading workloads. In this section, we will describe these workloads, and provide an in-depth discussion of two of them.

4.2.1 Workloads

During development of the Spark implementation, our initial workloads were derived by porting several example programs from the Microsoft Direct3D SDK:

BasicHLSL (Figure 4.2) Renders a single model with a diffuse texture map and directional light.

DetailTessellation (Figure 4.3) Uses tessellation, displacement, and normal mapping to add detail to a flat surface. It supports adaptive tessellation based on distance as well as local surface detail.

PNTriangles (Figure 4.4) Performs tessellation of PN triangle patches [VPBM01], with several schemes for controlling tessellation rates: distance-, orientation-, and screen-space-adaptive. Supports frustum and backface culling of patches.

CubeMapGS (Figure 4.5) Implements several techniques for rendering to a cube map: multipass rendering, instancing in the IA stage, looping in the GS stage, and instancing in the GS stage.

We also implemented a workload using the example Spark shaders given in Section 3.4 to confirm that they work as expected.

Figure 4.2: *BasicHLSL example, showing a single model with diffuse texture map and directional lighting.*

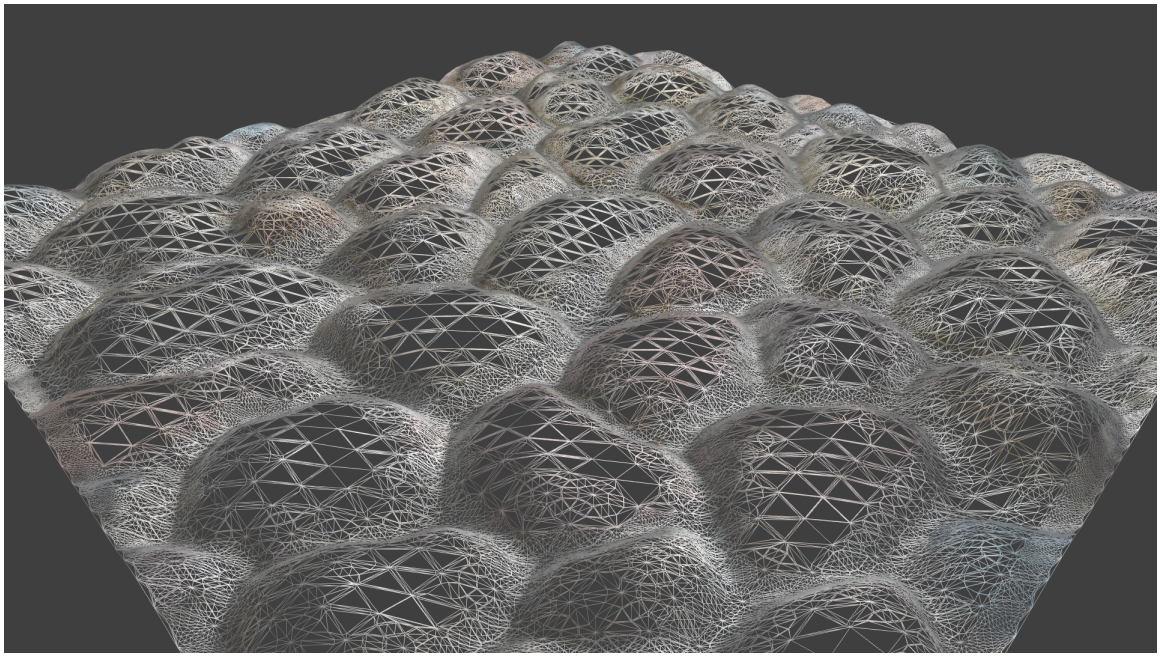

Figure 4.3: *DetailTessellation example, showing a tessellated and displaced terrain surface. The tessellation rate is determined in part by local surface detail.*

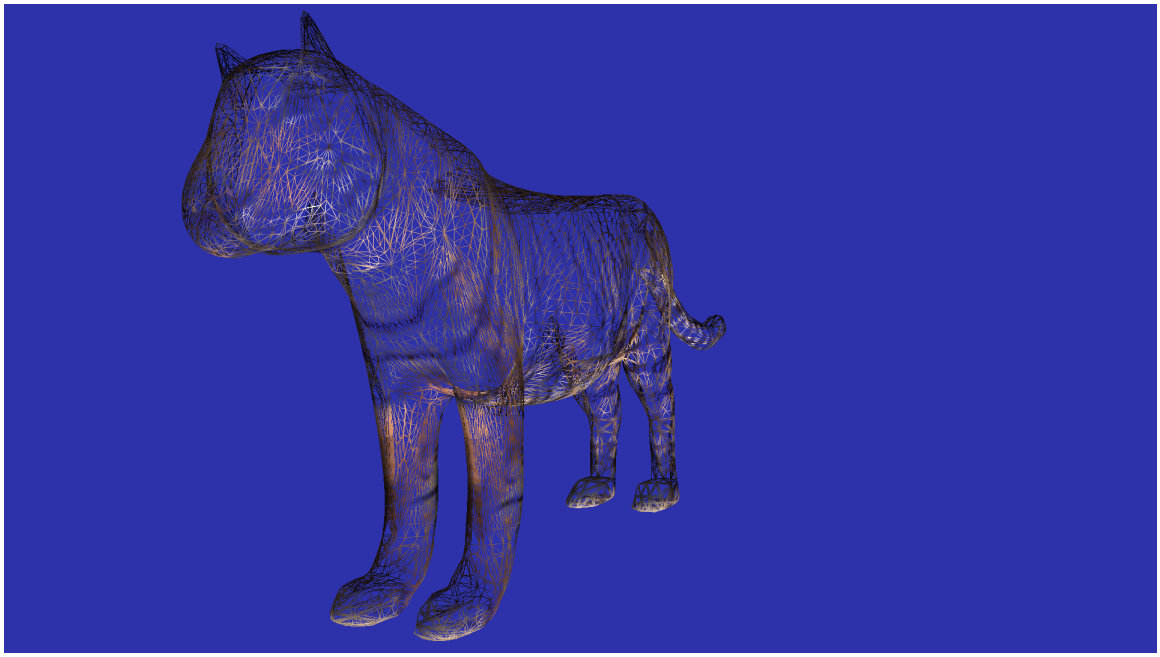

Figure 4.4: *PNTriangles example, demonstrating screen-space adaptive tessellation.*

Figure 4.5: *CubeMapGS example, showing a car model reflecting its environment, using a dynamic reflection cube map.*

Component	Description
`Base`	Various shader parameters
`SurfaceAttributes`	Surface-attribute interface
`PackGBuffer`	Pack surface attributes into g-buffer
`UnpackGBuffer`	Unpack surface attributes from g-buffer
`GenerateSurfaceAttributes`	Generate surface attributes in forward pass
`Light`	Light interface
`DirectionalLight`	Directional light
`SpotLight`	Spot light

Table 4.1: *Shader components used in surface-lighting application.*

In addition to these smaller examples, we have developed two more complicated applications that involve suites of components (implemented as Spark shader classes), intended to be composed. We will discuss these applications further in the following sections. The first application was authored by a user with some prior experience with D3D 11 and HLSL, but no experience with Spark. This example involves simple shading and lighting: implementing both a *forward* and *deferred* rendering strategy. The second application was authored by an experienced Spark user, and comprises a library of geometric effects: animation, tessellation, etc.

In looking at these applications, we are interested in two questions. First, does the Spark shader code demonstrate our goals of modularity and composability? Second, does our Spark shader code perform similarly to traditional shading languages like HLSL?

4.2.2 Library for Lighting Surfaces

The motivation for our first application was to see whether Spark could be used to encapsulate the difference between forward and deferred rendering, so that orthogonal concerns like light shaders could be written once and re-used across both rendering strategies.

Table 4.1 lists the key Spark shader components used in this application. The key feature of this design is that we define an abstract interface, `SurfaceAttributes`,

for the surface attributes in the material model, which other components *provide* or *consume*. Two components provide the interface: `GenerateSurfaceAttributes` and `UnpackGBuffer`. The interface is consumed by `PackGBuffer` and the two concrete subclasses of `Light`: `DirectionalLight`, and `SpotLight`. These components can be composed in three basic patterns:

- Composing `GenerateSurfaceAttributes` with one of the concrete `Light` components yields a forward-rendering pass.
- Composing `GenerateSurfaceAttributes` with `PackGBuffer` yields a g-buffer generation pass.
- Composing `UnpackGBuffer` with one of the concrete `Light` components yields a deferred-lighting pass.

Because the particular light shaders `DirectionalLight` and `SpotLight` depend only on the `SurfaceAttributes` interface, their definitions are independent of whether forward or deferred rendering is being used. This demonstrates that the Spark shader code as been able to achieve the desired decomposition of the problem, but how does this compare to existing shader-per-stage languages like HLSL? We will explore this question next.

The Spark Code

Figure 4.6 shows a zoomed-out view of the Spark code for this application. Different colors represent different concerns, and the dashed boxes outline individual Spark shader classes. For the most part, we see that separate concerns are encapsulated in separate shader classes, with two exceptions:

- The parameters of all the effects are bundled in a single `Base` shader class.
- The spot-light shader class includes code for things like sampling a shadow map, and additive blending that are not particular to spot-light illumination.

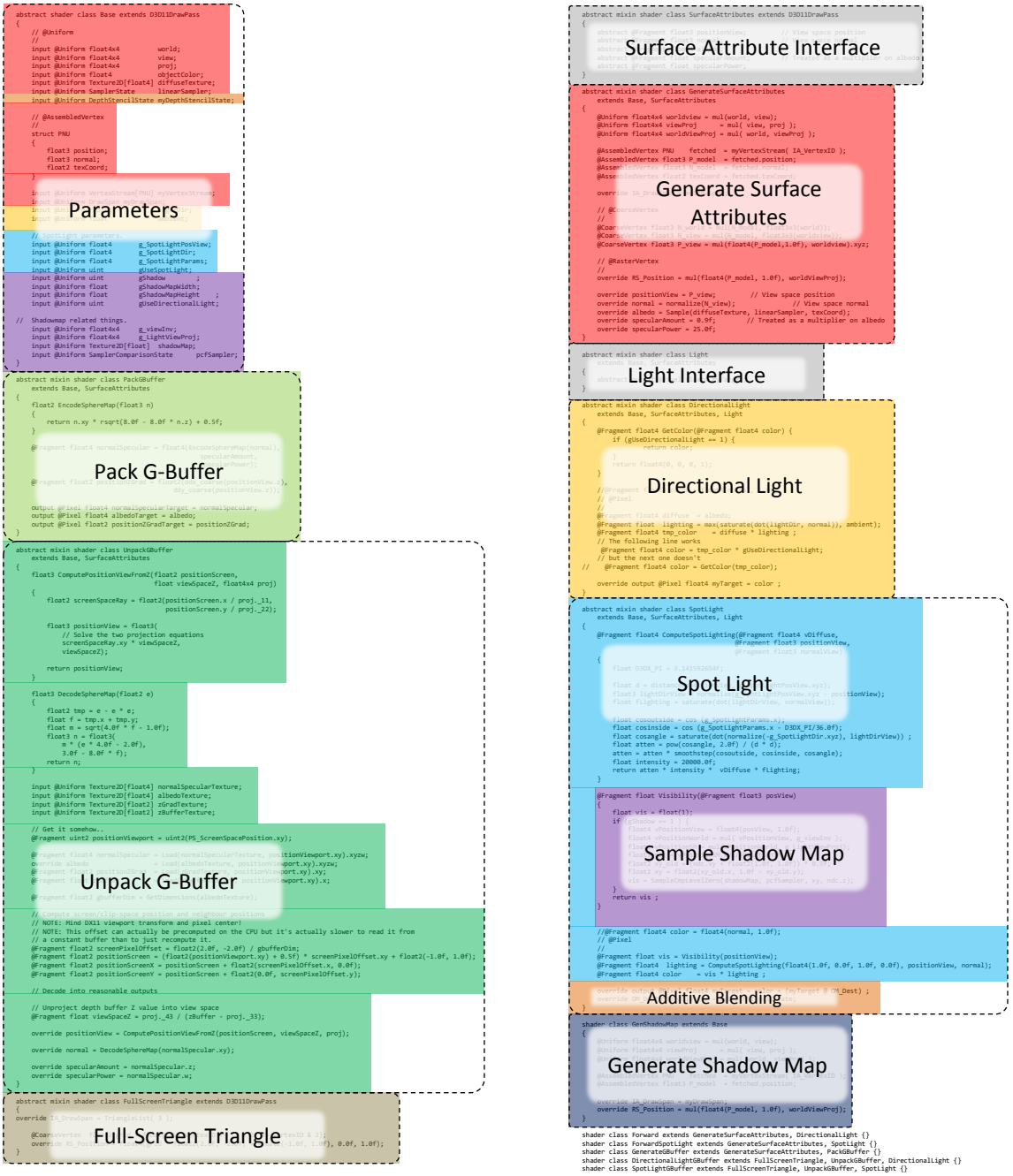

Figure 4.6: *Spark source code for deferred lighting application. The different colors indicate different concerns in the program. The dashed boxes indicate different Spark shader classes.*

In each case, it would be possible to refactor the code to remove the unwanted coupling: the parameters of each of the components could be moved into the corresponding shader class; the shadow-mapping and additive-blending code could be moved into their own independent shader classes. We have left the code as it was originally decomposed by the programmer in order to ensure that the comparison to HLSL is fair.

The HLSL Code

Figure 4.7 shows a zoomed-out view of the HLSL code for the lighting application. Again, different concerns have been color-coded – the colors correspond to those in Figure 4.6. The dashed lines here enclose each of the per-stage shader entry points.

We can see that the decomposition of the HLSL code is quite similar to the Spark code. This is partly due to how this application was developed: the programmer first wrote the HLSL shader code and then ported it to Spark. Rather than the Spark `SurfaceAttributes` interface, the HLSL code has a `SurfaceAttributes struct` type. The various forward- and deferred-rendering components are expressed in HLSL as subroutines that consume or produce values of this type.

The most salient difference between the Spark and HLSL code is that the HLSL code requires a specific Pixel Shader entry point for each of the concrete rendering passes (e.g., forward-rendered spot light). Each of these composed rendering passes in Spark also requires a distinct shader class, but each of these is a one-line declaration.

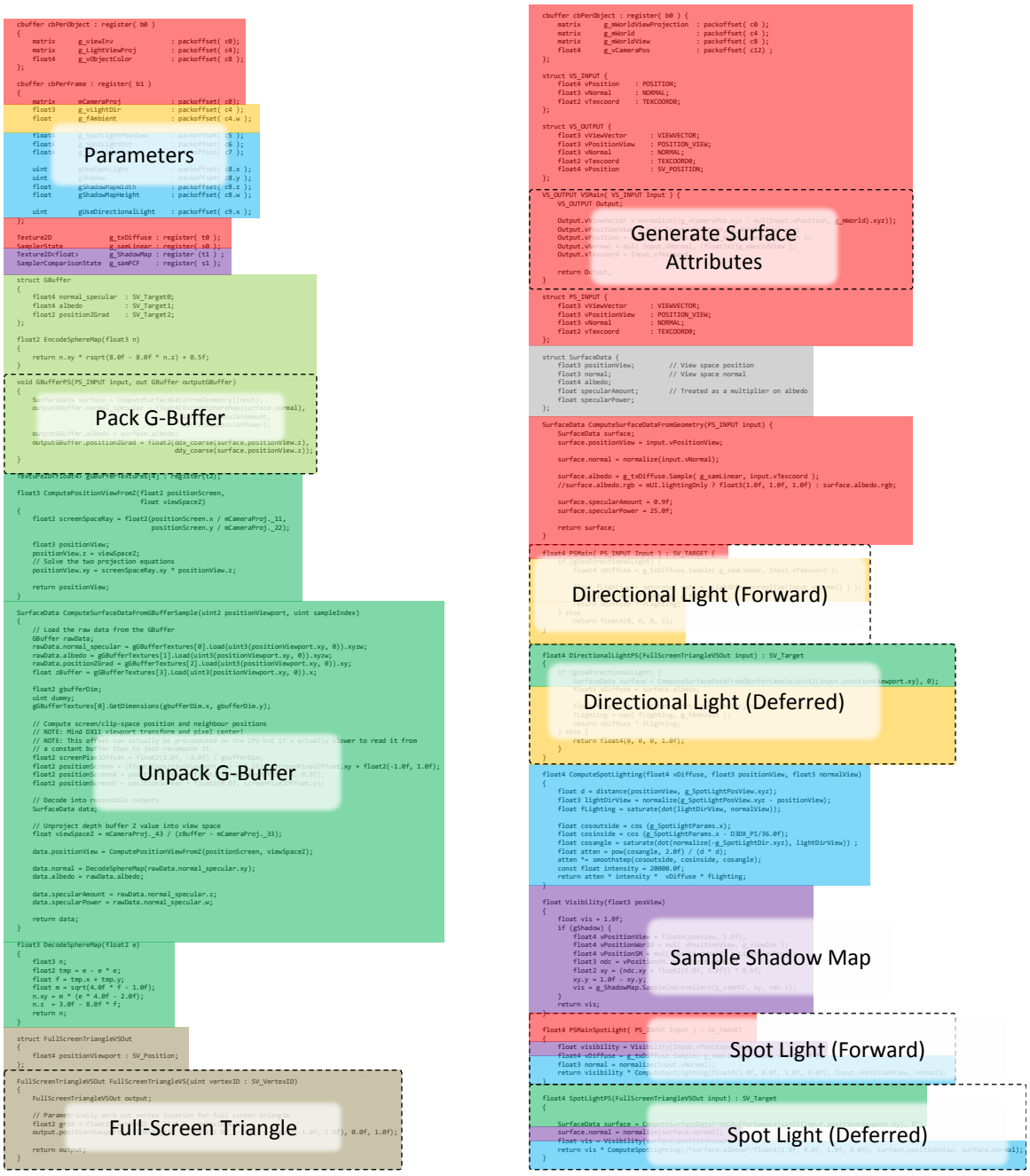

Figure 4.7: *HLSL source code for deferred lighting application. The different colors indicate different concerns in the program. The dashed boxes indicate per-stage shader entry points.*

Discussion

This application is something of a negative result: while the Spark decomposition is no worse than the HLSL, it is not significantly better. The reason for this is simple: the concerns being decomposed here primarily intersect a single pipeline stage (the Pixel Shader). There are no cross-cutting concerns that might benefit from expression in Spark. For our next example we specifically select an application area that is more likely to yield cross-cutting concerns.

4.2.3 Library for Geometric Effects

As a second application, we had an expert HLSL and Spark user develop a suite of shader classes that implement a variety of geometric effects. The goal was to demonstrate that different subsets of these effects could be composed to render models under different conditions, without writing additional code.

Table 4.2 summarizes the most important shader components in the suite. Among the shader components, `CubicGregoryQuads` and `RenderToCubeMap` include groupwise operations; all the others comprise only pointwise shading code. Table 4.3 shows which components are used by each of our models. The shading code for each model is simply a composition of existing shader classes; no additional shader code was written per-model.

Figure 4.8 shows a scene composed of these models. Note that the Lizard and Vortigaunt models are both rendered into the reflection cube map used by the Big Guy (this is most visible in the pink reflections on the right side of the Big Guy). Because `RenderToCubeMap` defines suitable plumbing operators, it may be combined with any of the other shader components to render a model into a cube map in a single pass, without additional code.

Figure 4.8: *Example models rendered with Spark shaders. From left to right the models are Lizard, Big Guy, and Vortigaunt. Lizard is a triangle mesh with skeletal animation, diffuse/normal maps, and Phong illumination. Big Guy is an approximate subdivision surface rendered with a dynamic reflection cube map (reflecting both the Lizard and Vortigaunt). Vortigaunt uses both skeletal animation and approximate subdivision surfaces. Vortigaunt model data provided courtesy of Valve Corp; Uffizi Gallery light probe image courtesy of Paul Debevec.*

Component	Description
`SkeletalAnimation`	Transformation by "bone" matrices
`CubicGregoryQuads`	Tessellate approximate subdivision surfaces [LSNCn09]
`RenderToCubeMap`	Uses GS instancing
`PhongMaterial`	Phong illumination [Pho73]
`EnvMapMaterial`	Sample a reflection cube map
`NormalMaterial`	Visualize normals

Table 4.2: *Shader components used in Figure 4.8.*

Model	Components
Lizard	`SkeletalAnimation`, `PhongMaterial`
Big Guy	`CubicGregoryQuads`, `EnvMapMaterial`
Vortigaunt	`SkeletalAnimation`, `CubicGregoryQuads`, `NormalMaterial`

Table 4.3: *Models used in Figure 4.8, and the shader components they use. For a given rendering pass, these may additionally be composed with shader components implementing light sources, or with the `RenderToCubeMap` component.*

The Spark Code

Figure 4.9 shows a zoomed-out view of the Spark code for this application. As in Section 4.2.2, dashed outlines indicate distinct Spark shader classes. We do not color-code individual concerns, but instead group concerns into broad categories:

- Vertex attributes (red). These might be provided by particular models being rendered, and are required by certain effects (e.g., animation requires bone weights and indices). We highlight texture coordinates, tangent vectors, and animation weights/indices, but do not call out positions or normals; these attributes are pervasive enough that we do not consider them as logically-orthogonal concerns.
- Skeletal animation (blue).
- Tessellation (green). The code supports the appropximation of Catmull-Clark subdivision surfaces by cubic Gregory patches [LSNCn09]. The code is factored into cases for triangles and quadrilaterals, and a base class for the common code. Computation of tessellation rates is specified through a separate component, although uniform tessellation is the only option implemented.
- Geometry Shader effects (teal). We only support a single use of the GS—single-pass rendering to a cube map—but we factor out the behavior of a "pass-through" GS as a base class.
- Material and lighting model (orange). We support a small number of materials, along with two types of lights.
- Material parameters (purple). Material parameters such as diffuse color may be specified per-model or through a texture map. In order to express this variability, we use several shader components.

At a glance, the decomposition here is more successful than that in Section 4.2.2: distinct categories of features do not cross shader-class boundaries. We will discuss a few aspects of this decomposition in more detail, with a focus on possible areas for future work.

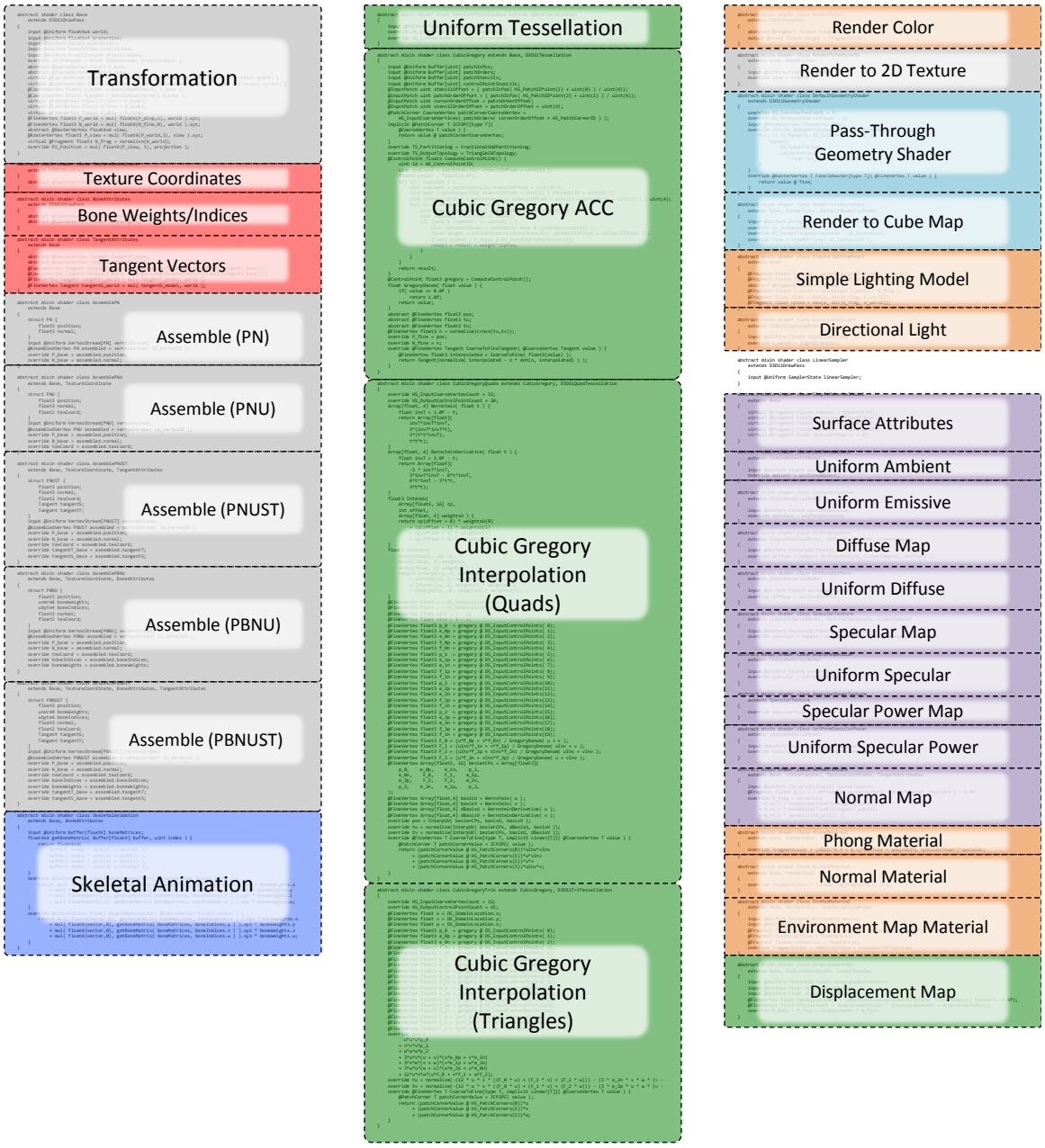

Figure 4.9: *Spark source code for geometric effect application. The different colors indicate different categories of concerns in the program. The dashed boxes indicate different Spark shader classes.*

The shader classes representing vertex attributes (e.g., "Texture Coordinates") represent an interface between classes that *provide* the particular attribute (e.g., "Assemble (PNU)") and those that *require* the attribute (e.g., "Diffuse Map"). The various "Assemble" components configure the IA pipeline stage, and correspond to different in-memory layouts for vertex data; this application supports five different layouts, and so there are five different classes for vertex assembly.

The proliferation of classes related to vertex assembly speaks to a possible weakness of our design choice to have Spark target the entire pipeline, including fixed-function stages. An application that configures the IA stage through a C++ API rather than shader code could more easily data-drive the configuration of this stage.

Ideally, Spark would allow vertex-assembly operations to be flexibly composed like other kinds of shader code, but this is made challenging by the need to support a well-defined in-memory layout for vertex data. To clarify this point, we note that it is (relatively) easy for a programmer to understand the memory layout of a C++ `struct` that uses single inheritance. In contrast, in the presence of multiple mixin inheritance, the in-memory layout would depend on details of the linearization algorithm. Spark currently avoids the potential confusion by drawing a sharp divide between `struct` types (which guarantee a memory layout, but do not support any kind of inheritance), and classes and record types (which support multiple mixin inheritance and virtual-class-based further extension, respectively).

The decomposition in Figure 4.9 also shows a large number of shader classes to support material parameters (e.g., ambient, diffuse, and specular colors). Depending on the model, these parameters might be computed at one of several rates (the current code supports material parameters as uniform shader parameter as well as texture maps; conceivably one might also wish to support per-vertex color channels). Currently, the concerns of *what* material parameters are available, and *how* those parameters are specified are coupled. Ideally, we should be able to define the notion of a "material parameter" once, and simply specify that both diffuse and specular reflectance are surface attributes.

The HLSL Code

For comparison, we also implemented an idiomatic HLSL über-shader for the same set of effects and compared performance results. In the über-shader, each of our shader components corresponds, approximately, to a preprocessor flag. For example, a preprocessor flag is used to enable or disable rendering to a cube map.

When creating an über-shader, certain global optimizations can be implemented using the preprocessor. For example, the material used on the Vortigaunt does not make use of interpolated positions or texture coordinates in the PS. Therefore, when this material is used, some plumbing code in earlier pipeline stages is dead code, and may be eliminated. In our HLSL über-shader, we perform dead-code elimination by conditionally defining per-stage shader outputs based on the needs of downstream stages.

Figure 4.10 shows a zoomed-out view of the HLSL shader code; some preprocessing directives have been stripped from this code to minimize the overall length. Dashed boxes mark per-stage shader entry points, and code pertaining to different concerns is color-coded to match Figure 4.9. In some cases, blocks of one color are "nested" inside of another color to make clear that both features are in the same subroutine.

Compared to the Spark code, individual features are more widely distributed in the code, and some features are coupled in the HLSL code that were separate in Spark. In particular, note how plumbing code related to vertex attributes (red) appears inside the animation (blue) and tessellation (green) features. In practice, adding support for a new vertex attribute (e.g., per-vertex diffuse color) to the über-shader could require modification at each of these red locations. Similarly, the tessellation effect could only be re-used in another shader by first decoupling it from the particular vertex attributes used here and copy-pasting the tessellation-related code (green) into a new shader.

Figure 4.10: *HLSL source code for geometric effect application. The different colors indicate different categories of concerns in the program. The dashed boxes indicate per-stage shader entry points.*

Model	Time HLSL (ms) no DCE	Time HLSL (ms) DCE	Time Spark (ms)		Compile Time (ms) HLSL	Compile Time (ms) Spark
Render to Cube Map (1024×1024×6)						
Lizard	0.507	0.506	0.507	(+0%)	155	568 (3.7×)
Vortigaunt	6.49	4.36	4.37	(+0%)	230	631 (2.7×)
Render to Screen (1792×512)						
Lizard	0.093	0.093	0.091	(−2%)	117	374 (3.2×)
Big Guy	1.12	0.990	1.01	(+2%)	178	387 (2.2×)
Vortigaunt	0.974	0.851	0.867	(+2%)	207	511 (2.5×)

Table 4.4: *Performance results comparing Spark and HLSL. We measure GPU execution time of each draw pass for the view shown in Figure 4.8, rendered on an ATI Radeon HD 5870, as well as cumulative shader compilation times for each pass, measured on an Intel Core i7 975. For the HLSL über-shader, we measure rendering performance without and with manual dead-code elimination (DCE). Spark-compiled shaders perform similarly to HLSL with this global optimization applied.*

These results serve as *qualitative* evidence that Spark is better suited to modular, reusable expression of these shading effects. In the next section, we present quantitative data to demonstrate that this improved expressiveness does not have an undue performance cost.

Performance

Table 4.4 shows performance results for the rendering passes in Figure 4.8, using both HLSL and Spark. We give results for our HLSL über-shader both with and without manual dead-code elimination. The benefits of dead-code elimination are most notable when rendering the Vortigaunt to our reflection cube map: performance is improved by 33%. Rendering passes using Spark-compiled shaders have similar performance to the HLSL über-shader with dead-code elimination: between 2% slower and 2% faster. This is because, as described in Section 4.1.2, the Spark system automatically performs global dead-code elimination as part of compilation.

Table 4.4 also shows HLSL and Spark compile times for the combination of features in each rendering pass. The per-combination cost of the Spark compilation is between two and four times that of HLSL. In the case of Spark, this cost includes global optimization of the composed shader class, source-to-source translation to HLSL, compilation of the generated HLSL, and generation of CPU code for shader binding and `@Uniform` computation. In addition to the per-combination cost, the Spark path also incurs a one-time startup cost of 6.6 seconds to parse and type-check the Spark shader suite. This startup cost could potentially be mitigated by performing type-checking at application compile time and instead loading a serialized representation of the Spark shader suite.

Language	Code	Preprocessor	Total
HLSL	478	260	738
Spark	501	0	501

Table 4.5: *Comparison of lines of code (non-whitespace, non-comment) in Spark and HLSL implementations of the shader suite.*

Table 4.5 compares the number of non-comment lines of code in the Spark and HLSL implementations of the shader suite. While both the Spark and HLSL implementations use similar amounts of code to express the shading algorithms themselves, the HLSL implementation also requires preprocessor directives to select between the different über-shader code paths, and to implement dead code elimination. In the case of Spark, components are defined as distinct shader classes, and so preprocessor directives are not required to enable or disable features. Considering both code and preprocessor directives, the Spark implementation requires 32% fewer lines to implement the same effects, with similar performance.

Chapter 5

Discussion

We have presented both the design of the Spark shading language, as well as our experience implementing and using a compiler and runtime for Spark. In this chapter, we will discuss possible directions for continuing or extending our work. We begin by sketching the connections between the type system of the Spark language and more familiar work on programming languages. We then turn our attention to various possible extension of the Spark language or system that could provide a better experience to future users.

5.1 Rates of Computation Are Functors

In this section, we will relate the core type system of Spark to the notation introduced in Section 2.3.4, and use this notation to demonstrate how Spark's rates of computation serve as mathematical functors.

As discussed in Section 2.3.4, our type system has two levels: expressions and their values are classified by types, while types are in turn classified by *kinds*.

5.1.1 Kinds

Our Spark type system has three fundamental kinds, and one kind constructor:

$$
\begin{array}{lll}
K & ::= & \\
 & | \quad * & \texttt{// kind of proper types} \\
 & | \quad K_1 \Rightarrow K_2 & \texttt{// arrow kind constructor} \\
 & | \quad \mathit{RecordType} & \texttt{// kind of record types} \\
 & | \quad \mathit{RateQualifiedType} & \texttt{// kind of rate-qualified types}
\end{array}
$$

The kinds $*$ and $\Rightarrow$ are as presented in Section 2.3.4: $*$ is the kind of all proper types, and $\Rightarrow$ is used to form the kinds of "generic" or "templated" types. In our concrete Spark syntax we use square brackets `[]` to represent the application of a generic type to arguments (e.g., `Array[float,6]`).

The *RecordType* kind is used to classify record types such as `Fragment`. All record types are also proper types (but not vice versa), since records (vertices, fragments, etc.) are values at run-time. Formally, we say that *RecordType* is a *sub-kind* of $*$.

5.1.2 Rate-Qualified Types

The remaining kind in our system is *RateQualifiedType*, and all attributes in our system have types of this kind. Rate-qualified types are created using the `@` type constructor, which takes two parameters: a record type and any data type, and returns an attribute type.

$$\texttt{@} :: \mathit{RecordType} \Rightarrow * \Rightarrow \mathit{RateQualifiedType}$$

The following set of kinding derivations give an example of how the `@` operator works:

$$
\begin{array}{rl}
\texttt{Fragment} & :: \mathit{RecordType} \\
\texttt{float4} & :: * \\
\texttt{@ Fragment float4} & :: \mathit{RateQualifiedType}
\end{array}
$$

In the concrete syntax of Spark, we write a type like `@ Fragment float4` in the stylized form `@Fragment float4`, and call the `@Fragment` part a rate of computation.

5.1.3 Rates of Computation

Since the two-argument `@` operator is given in a *curried* form, so it can be partially applied to a record type, and so a rate of computation like `@Fragment` is in fact a one-argument type constructor:

$$\texttt{@Fragment} \quad :: * \Rightarrow \mathit{RateQualifiedType}$$

5.1.4 Lifting

Most functions in Spark, whether built in or user-defined, are simply declared to operate on ordinary values (i.e., with types of kind $*$). These functions can then be automatically *lifted* by the compiler to operate on attributes (with types of kind *RateQualifiedType*).

That is, if a simple mathematical operator like floating-point addition:

$$\texttt{+} : (\texttt{float} \times \texttt{float}) \rightarrow \texttt{float}$$

is applied to arguments of type `@R float`, the compiler automatically synthesizes a version of the operator lifted to the `@R` rate:

$$\texttt{+}_{\texttt{@R}} : (\texttt{@R float} \times \texttt{@R float}) \rightarrow \texttt{@R float}$$

5.1.5 Plumbing Operators

While most functions must be lifted to work on attributes (rate-qualified types), plumbing operators, introduced in Section 3.2.4, are functions that work directly on values with rate-qualified types, e.g.:

$$\texttt{perspectiveInterpolate} : \texttt{@CoarseVertex float4} \rightarrow \texttt{@Fragment float4}$$

5.1.6 Projection

Section 3.3.6 shows how user-defined plumbing operators are typically implemented using *projection*. We briefly digress here to show how projection fits into our presentation of Spark's type system. In our concrete syntax for Spark, we overload the `@` character as an infix projection operator. To avoid ambiguity, however, we will use `#` here instead.

We treat `#` as a type-parametric infix operator:

$$\texttt{\#} \quad : \forall \texttt{R} :: \mathit{RecordType}.\forall \texttt{T} :: *.\texttt{@R T} \rightarrow \texttt{R} \rightarrow \texttt{T}$$

Here the $\forall$ symbol is used to introduce a type parameter, so that this definition may be read as: given a record type `R` and a data type `T`, the `#` operator is a function that maps an attribute of type `@R T` and a record of type `R` to a value of type `T`.

The following derivation illustrates use of projection (leaving out the type parameter of `#`, since it can be inferred from context):

```
      color  : @ControlPoint float4
         cp  : ControlPoint
color # cp   : float4
```

In general, project interacts in an orthogonal fashion with lifting, so that:

```
      color  : @ControlPoint float4
        cp0  : @FineVertex ControlPoint
color # cp0  : @FineVertex float4
```

Here we have lifted the projection operator to the `@FineVertex` rate. The result is a projection similar to that used in Listing 3.1.

5.1.7 Rates of Computation Are Functors

Given these preliminaries, we can now show that rates of computation in Spark are mathematical *functors*. We use the definition as given in Pierce's introduction [Pie91]:

Given categories $\mathbf{C}$ and $\mathbf{D}$, a functor $F : \mathbf{C} \to \mathbf{D}$ is a map taking:

- each $\mathbf{C}$-object A to a $\mathbf{D}$-object $F(A)$, and
- each $\mathbf{C}$-arrow $f : A \to B$ to a $\mathbf{D}$-arrow $F(f) : F(A) \to F(B)$

such that:

- $F(id_X) = id_{F(X)}$ for all $\mathbf{C}$-objects A, and
- $F(g \circ f) = F(g) \circ F(f)$ for all composable $\mathbf{C}$-arrows f and g.

That is, identity mappings and composition are preserved by F.

In the case of Spark, a rate of computation `@R` : $* \Rightarrow$ *RateQualifiedType* is a map taking:

- each data type `T` to a rate-qualified type `@R T`, and
- each function `f` : `A` $\to$ `B` to a lifted operator $\texttt{f}_{\texttt{@R}}$: `@R A` $\to$ `@R B`

That is, $\mathbf{C}$ is the category of types of kind $*$, and $\mathbf{D}$ is the category of types of kind *RateQualifiedType*. Our approach to lifting trivially preserves identity mappings and composition, since a a lifted operator $\texttt{f}_{\texttt{@R}}$ is applied pointwise.

We note that in the GPipe system [Bex], the `Vertex 'a` and `Fragment 'a` type constructors are similar in function to Spark's `@Vertex A` and `@Fragment A`. These types in GPipe are instances of the Haskell type-class `Functor`, which represents the mathematical functor concept (although it does not enforce the two equality constraints).

5.2 Record Types Are Virtual Classes

In the preceding section we concerned ourselves with the code that appears *inside* of a Spark shader class. We now turn our attention to the mechanisms by which shader classes are extended and composed.

As we discuss in Section 3.3.5, record types in Spark constitute a specialized case of the general concept of virtual classes, as introduced in Section 2.3.2. In particular, each record type can be seen as a virtual class, nested inside of the shader class. When inheriting from a shader class, we also inherit the record types it contains, and can *further extend* those record types with new members.

In this section, we will make this connection explicit by describing an approach for translating ordinary Spark shader code to a language with support for virtual classes. As the target of our translation we use the Scala language, with a proposed extension [Sca] that adds direct support for virtual classes (the current Scala language supports virtual *types* but not full virtual classes).

5.2.1 Spark

Listing 5.1 shows a Spark shader class `Base` that might be defined in a system library for a simple rendering pipeline, and a shader class `Derived` that might be defined in user code. We will briefly discuss the declarations in these classes before moving on to their translation.

The `Base` class defines three record types for a simple rendering pipeline: `Uniform`, `Vertex`, and `Fragment`. Two of these record types are marked `concrete`, which indicates to the Spark compiler that we need to be able to construct instances of them. In practice, this means that a class that inherits from `Base` cannot define new `input` attributes with per-vertex or per-fragment rates, because doing so would alter the signature of the `Vertex` or `Fragment` record constructor, respectively. We have not encountered the `concrete` keyword previously (e.g., in the examples in Section 3.4) because user code in Spark does not typically need to define new record types.

```
shader class Base
{
    record Uniform;
    concrete record Vertex;
    concrete record Fragment;

    input    @Vertex   uint   VS_VertexID;
    abstract @Vertex   float4 VS_Position;
    abstract @Fragment float4 PS_Color;

    input @Vertex Uniform __u2v;
    implicit @Vertex T UniformToVertex[type T](
        @Uniform T value )
    { return value @ __u2v; }

    input @Fragment Uniform __f2v;
    implicit @Fragment T UniformToFragment[type T](
        @Uniform T value )
    { return value @ __f2v; }

    input @Fragment Vertex v2f;
    implicit @Fragment T VertexToFragment[type T](
        @Vertex T value )
    { return value @ __v2f; }
    // ...
}

shader class Derived extends Base
{
    input @Uniform float4x4              modelViewProj;
    input @Uniform VertexStream[float4] positionStream;
    input @Uniform VertexStream[float4] colorStream;

    @Vertex float4 P_model = positionStream( VS_VertexID );
    @Vertex float4 C       = colorStream( VS_VertexID );

    override VS_Position = mul( modelViewProj, P_model );
    override PS_Color    = C;
}
```

Listing 5.1: *Spark shader classes to illustrate translation to virtual classes.* `Base` *represents a pipeline interface that might appear in a system library, and* `Derived` *a shader using that interface.*

The `Base` class then defines attributes that a derived shader can use. A shader can use `VS_VertexID` to fetch per-vertex data, and must provide values for `VS_Position` and `PS_Color` to be used in rasterization and blending, respectively.

Finally, the `Base` class defines several plumbing operators, in order to allow data to be plumbed from `@Uniform` rate to `@Vertex` or `@Fragment`, and also from `@Vertex` to `@Fragment`. If this were an interface to a real pipeline, such plumbing operators might be implemented with support from a pipeline module in the compiler (see Section 4.1.1), but here we provide dummy implementations with the help of additional "private" `input` attributes (marked with leading underscores on their names—e.g., `__u2v`—since Spark does not support private members).

The `Derived` class extends `Base` to define some shading computations. We assume here that types such as `float4` and `VertexStream` have been defined in `Base`, but left out of this example to conserve space. The code in `Derived` makes (implicit) use of the plumbing operators defined by `Base`, to plumb `@Uniform` parameters like `modelViewProj` into per-vertex computation, and to plumb the per-vertex attribute `C` to a per-fragment computation.

5.2.2 Scala with Virtual Class Support

Listing 5.2 shows a translation of the Spark shader class `Base` to the Scala language extended with direct support for virtual classes (see the Scala project site [Sca] for details on this notation). Listing 5.3 shows the matching translation of `Derived`.

We now describe the steps of this translation:

- Each Spark shader class is translated to a Scala class. The Scala class is `abstract` iff the Spark class was.
- If the Spark shader class inherited from any base classes, the Scala class inherits from the corresponding classes.

```
abstract class Base
{   // record Uniform;
    trait Uniform <: {}

    // concrete record Vertex;
    abstract class Vertex(
        // input attributes:
        val u2v : Uniform,
        val VS_VertexID : uint ) <:
    {   // other attributes:
        def VS_Position : float4;
        // plumbing operators:
        def UniformToVertex[T](value : Uniform=>T)
            = value(u2v);
    }

    // concrete record Fragment;
    abstract class Fragment(
        // input attributes:
        val u2f      : Uniform,
        val v2f      : Vertex ) <:
    {   // other attributes:
        def PS_Color : float4;
        // plumbing operators:
        def UniformToFragment[T](value : Uniform=>T)
            = value(u2f);
        def VertexToFragment [T](value : Vertex=>T)
            = value(v2f);
    }
}
```

Listing 5.2: *Translation of the shader class* `Base` *in Listing 5.1 to Scala, extended with direct support for virtual classes.*

- Each `concrete` record type maps to a nested Scala class, and each non-concrete record type maps to a Scala trait. If the Spark record type has any `abstract` attributes, the corresponding Scala class is marked `abstract`.
- The class/trait corresponding to a record type is declared virtual. This is marked in Scala by the `<:` token (which in Scala represents the "is subtype of" relation).
- Each of the `input` attributes of a concrete record type is translated into a constructor parameter on the corresponding class (c.f., `VS_VertexID`).
- Every other attribute of a record type is translated into a zero-argument method in the body of the corresponding Scala class/trait. Abstract attributes translate to abstract methods, while concrete attributes translating their defining expression from Spark to Scala.
- Plumbing operators in Spark are translated to ordinary Scala methods. Parameters with rate-qualified types (e.g., `@Uniform T`) are translated to functions (e.g., `Uniform=>T`). Projection in Spark (e.g., `value @ u2f`) is translated to function application in Scala: `value(u2f)`.
- Applications of plumbing operators are all made explicit, and their parameters are turned into explicit functions. For example, when passed to a plumbing operator, the `@Vertex float4` attribute `C` is replaced with `(v:Vertex) => v.C`.

Each of the nested classes/traits inside of the Scala translation of `Derived`, which correspond to record types in the original Spark shader, *further extends* the equivalent declaration in `Base`. That is, the type `Derived.Vertex` includes all the members of `Base.Vertex`, as well as any new ones declared (like `P_model`).

The Scala code here is more verbose than the equivalent Spark. This is in part because of making invocation of plumbing operators explicit, and also because the concrete `Vertex` and `Fragment` types must re-declare the constructor signature of the equivalent classes in `Base` (see Section 2.3.3 for a discussion of this issue with traditional constructors).

```
class Derived extends Base
{   // Uniform:
    trait Uniform <:
    {
        val modelViewProj   : float4x4;
        val positionStream  : VertexStream[float4];
        val colorStream     : VertexStream[float4];
    }
    // Vertex:
    class Vertex(
        // input attributes:
        val u2v : Uniform,
        val VS_VertexID : uint ) <:
    {   // apply implicit plumbing:
        def positionStream = UniformToVertex(
            (u : Uniform) => u.positionStream );
        def colorStream    = UniformToVertex(
            (u : Uniform) => u.colorStream );
        def modelViewProj  = UniformToVertex(
            (u : Uniform) => u.modelViewProj );
        // attributes:
        def P_model     = positionStream( VS_VertexID );
        def C           = colorStream( VS_VertexID );
        def VS_Position = mul( modelViewProj, P_model );
    }
    // Fragment
    class Fragment(
        // input attributes:
        val u2f      : Uniform,
        val v2f      : Vertex ) <:
    {   // apply implicit plumbing:
        def C = VertexToFragment( (v : Vertex) => v.C );
        // attributes:
        def PS_Color = C;
    }
}
```

Listing 5.3: *Translation of the shader class `Derived` in Listing 5.1 to Scala, extended with direct support for virtual classes.*

The same resource that introduces the Scala virtual class notation also introduces a strategy for encoding most use cases for virtual classes in terms of Scala's existing support for virtual types. We have applied this strategy to the example given in this section (the resulting code is quite verbose, and not especially interesting), and confirmed that the resulting Scala code type-checks, runs, and behaves as we expect given the high-level Spark.

5.2.3 Summary

The translation process outlined in this section gives us reason to believe that the semantics of Spark record types are indeed a constrained form of virtual classes. We have not, however, *proved* that the results of our translation process are semantically equivalent to the original Spark. Formalizing this translation is a possible direction for future research.

We remark that it is an important benefit that a Spark programmer need not understand the relationship between record types and virtual classes, or the details of functors as given in the preceding section. In fact, the original shader class `Derived` in Listing 5.1 is written entirely in terms of rates of computation; this shader could in fact have been easily written in the RTSL system. The relationships sketched here have proved valuable in implementing and understanding the fundamental abstractions of the Spark language, and how it expands upon the programming model of RTSL, but ultimately do no interfere with day-to-day programming in our system.

5.3 Spark and Aspect-Oriented Programming

Section 2.4.3 introduced the aspect-oriented programming (AOP) paradigm. Having discussed the design of Spark, we now describe how the language can be viewed as an AOP language, by relating Spark concepts to standard AOP terminology [KLM+97].

Spark's shader classes serve the role of *aspects*: each can encapsulate a program concern that might cross-cut the pipeline structure. Rates of computation are then *pointcuts*, describing places in the execution of a program that we might want to intercept: per-vertex, per-fragment, etc. We introduce *advice* at a given pointcut (e.g., a rate `@R`) by declaring attributes: each attribute modifies the program behavior by augmenting the shader graph.

The dual representation between rates of computation and record types in Spark reflects the essence of *weaving* in AOP. The user specifies their program primarily according to one decomposition (rates of computation), and then our compilation strategy (as outlined in Section 4.1.3) transforms the program into an a different decomposition for execution.

This connection between Spark and AOP was not a planned one, but it provides yet another lens through which our approach can be understood.

5.4 Future Work

> A language designed to give programmers what they want may initially succeed but create pernicious problems as it catches on. However, a language designed to give programmers what they really need may never catch fire at all
>
> *Seven Paradoxes of Object-Oriented Programming Langauges*
> David Ungar

In this section, we sketch several potential extensions to the Spark language and programming model. These extensions address some of the difficulties we have found when programming in Spark, and demonstrate possible directions for future shading languages that build on our results.

5.4.1 Improved Support for Procedural Operations

In Section 2.2.2 we describe how recent rendering pipelines such as D3D11 have begun to incorporate features of "compute" interfaces like CUDA, OpenCL, and Compute Shader: in particular, atomic read-modify-writes to global memory (using UAVs).

Our declarative shader-graph language design was originally motivated by rendering pipelines without support for these operations. As discussed in 4.1.6, our implementation does not prohibit the use of side-effecting operations in procedural shading code, but it does not specify a particular evaluation order for side effects in distinct shader-graph nodes.

Given that the use of UAVs is likely to increase in future rendering applications, we must ask: will our design choices in Spark be good ones moving forward? We have not yet implemented workloads that make use of UAVs in Spark, so we cannot answer this question with certainty, but we have reason to believe the answer is "yes."

In our experience with advanced shading algorithms written in HLSL, most uses of atomic operations in rendering pipelines fall into two categories:

Debugging The ability to output data to a globally-visible buffer from any point in the pipeline is an invaluable tool for debugging. In particular, by atomically appending to a buffer, it is possible for a shader programmer to implement a kind of "poor man's `printf()`".

Data Structure Construction Atomic operations can be used to construct data structures based on shader invocations: for example, to construct a linked list for each pixel, collecting all the fragments generated over that pixel [YHGT10].

In both of these cases, there is a well-defined operation that must be performed atomically (outputting to a stream, appending to a list), but the order of that operation relative to other shading computations is not important. The simple approach to atomic operations in Spark described above would be sufficient for these examples. We hope that gathering further experience with the Spark language will allow us to uphold or refute this intuition.

If greater control over order of operations is required, we could investigate incorporating Spark's most important features (rates of computation, plumbing operators, etc.) into a purely procedural shading language. We would, of course, have to find another way to rectify the problematic interactions between control flow and rates of computation identified by the creators of the Cg system (see Section 2.1.4). We conjecture that the most crucial restrictions in such a language would be:

- Under a control-flow construct that depends on an expression with rate `@R`, only allow assignment to variables with rate `@R`.
- Do not allow nesting of control-flow constructs with different rates.

If these restrictions are sufficient, then such a language might not be as confusing as Mark et al. feared. Proving out such a language is a possible direction for future work.

5.4.2 Rate-Based Overloading

It is common in writing Spark shaders to have the same nominal attribute, e.g., a world-space normal `N_world`, be defined at multiple places along the pipeline. For example, we might initially compute the normal per-coarse-vertex:

```
@CoarseVertex float3 N_world = ...;
```

but need to re-normalize the vector before using it in per-fragment shading:

```
@Fragment float3 N_world_frag = normalize( N_world );
```

These two declarations create a bit of a problem: both represent a world-space normal, but only one may be called `N_world`.

Of course, users can work around this limitation by renaming one of the two attributes, as we have done here with `N_world_frag`. In doing so, however, we create a risk that a user will forget to refer to the correct attribute `N_world_frag` in some per-fragment computation:

```
@Fragment float NdotL = saturate( dot( N_world, L_world ) );
```

In this case, the dot product `NdotL` will be computed using the interpolated (but not normalized) value of `N_world`, which is almost certainly *not* what the programmer intended. We have encountered problems of this type in our use of Spark, and they have been somewhat difficult to diagnose, since they typically manifest as reduced image quality rather than obviously-incorrect results.

One possible solution would be to define a custom type (e.g., `Normal`) to use for a value like `N_world` and give it a custom vertex-to-fragment plumbing operator that provides the renormalization. This would obviate the need for the declaration of a `@Fragment` normal attribute. Such an approach might appeal to some users (and is implementable in the current Spark language), but it has the potentially-undesirable effect of "hiding" the call to `normalize()` inside of an implicit plumbing operator. Some programmers would prefer to make all such operations explicit in their shader.

Given this situation, it would be an interesting extension of the Spark language to allow for multiple attributes to be declared with the same name and data type, but different rates, e.g.:

```
@CoarseVertex float3 N_world = ...;
@Fragment float3 N_world = normalize( interpolate(N_world) );
@Fragment float NdotL = saturate( dot( N_world, L_world ) );
```

Here we define an attribute `N_world` at both per-coarse-vertex and per-fragment rates. When an expression refers to `N_world`, such as in the definition of `NdotL`, the compiler can choose the version of the attribute that is "closest" to the desired rate (as measured by number of implicit rate conversions required, similar to Section 3.3.7). If the user needs to override this inferred decision, they may explicitly invoke a plumbing operator to make their intention clear; for example, the call to `interpolate()` in the definition of the per-fragment normal makes it clear that the programmer is referring to the interpolated per-coarse-vertex normal.

5.4.3 Type-System Support for Coordinate Spaces

The RenderMan Shading Language has a built-in notion of coordinate *spaces*. Shader inputs such as positions and vectors are all defined in "shading space," and geometric quantities may be converted between this an other spaces (e.g., world or view space) using the built-in `transform()` function.

It is easy for a shader programmer to incorrectly combine values defined in different coordinate spaces. One way to avoid such problems is to perform all computations in a "coordinate-free" fashion [MLD97], but real-time shaders often make use of specific coordinate spaces for performance or precision reasons. Alternatively, a system can exploit static checking to ensure that quantities in different spaces are not combined [OP10]. Static checking of coordinate spaces is quite similar to checking units of measure, a feature directly supported by the F# language [Ken97, Ken].

We can imagine extending the Spark shading language with similar features to make coordinate spaces explicit, e.g.:

```
space World;
space View;

// shader input: world-space light vector
input @Uniform Vector#World L;

// world-to-view transformation matrix
input @Uniform Transform#(View/World) view;

// compute view-space normal
@Fragment Normal#View N = ...;

// Error: dot() - Cannot combine values
// in 'World' space and 'View' space
@Fragment float NdotL_bad = saturate( dot( N, L ) );

// OK:
@Fragment float NdotL = saturate( dot( N, view*L ) );
```

5.4.4 Composing Classes vs. Objects

The final code that must be run for a given rendering pass may be influenced by many factors: camera, lights, material, geometry, and participating media. One strength of the Spark language is that we can separate out each of these concerns into separate modules. Each of these modules is a shader *class*, and the code for each rendering pass is derived by composing *classes*.

The representation of shader-graph composition as class inheritance in Spark is a key design decision, and follows closely from our goal of having a clear phase separation

(see Section 3.1). Because global optimizations are required to achieve good performance, composition in Spark is an *explicit* and *static* operation, so that the user can decide when to spend the time. Each unique combination of shader classes must be composed, compiled, and optimized separately, at a cost in both space and time. This problem is already known to users of über-shaders as the "shader combinatorics" or "permutation management" problem: a simple library of components may produce an overwhelming number of compiled shaders. While Spark does not eliminate this combinatorial explosion, it can still increase the manageability of shader code by allowing components to be specified and type-checked independently.

Other systems (including, e.g., some versions of Cg and Cg Effects), defer machine code generation and optimization until the time that a draw call is issued, maintaining a cache of previously-compiled shader code to amortize these costs. The resulting unpredictable pauses have been criticized by interactive graphics programmers, for whom worst-case rather than average-case performance is typically most important.

The decision to express shader composition through class inheritance in Spark brings about an unfortunate limitation. A typical rendering application might represent each camera, light, and geometric object in the scene as a distinct object in a *scene graph*; each object might be an instance of a particular C++ class. In the RenderMan system, each of these objects could be associated (in a one-to-one fashion) with an RSL shader *object* instantiated from some RSL shader class. When the properties of an object in the scene graph change (e.g., the brightness of a light source), these changes can easily be propagated to the corresponding shader object.

In the Spark system, however, camera, lighting, and material concerns (expressed as shader classes) must be composed to create composite effects *before* instances are created. This breaks the one-to-one relationship between objects in a scene graph and shader objects. An application using Spark must instead maintain a single Spark shader object for each *combination* of camera, light, and material classes. The corresponding logic to propagate parameter values from objects in the scene graph to shader objects becomes more complex.

While our design decision in Spark followed naturally from our goal to ensure a clear phase separation, we believe that future real-time shading languages should endeavor to support more flexible composition of shader *objects* rather than shader *classes*. Our shader-programming abstraction in Section 3.2 provides a foundation that is still relevant in such a scenario. The particular use of object-oriented idioms like inheritance and `virtual` members to represent provided and required interfaces, however, might not be tenable. Instead, we believe it could be valuable to explore more explicitly *component-oriented* language features (c.f., ArchJava [ACN02]). The most important challenge, however, in supporting composition of objects rather than classes would seem to be phase separation; how can a system avoid unpredictable runtime pauses for shader specialization?

5.4.5 Minimizing State Changes

Spark shader objects are *monolithic*: they represent the configuration of the entire rendering pipeline, including both fixed-function and programmable stages. With our current implementation strategy, we submit corresponding hardware state-change commands for the entire pipeline state each time a Spark shader object is submitted. This was appropriate as our concern so far has primarily been about GPU, rather than CPU performance, but a more serious implementation would likely need to consider CPU costs more thoroughly.

We can envision two main ways that the CPU cost of rendering with Spark shaders could be reduced: first by modifying the Spark language and runtime to exploit coherence between successive rendering operations, and second by modifying the interface provided by rendering architectures to better suit our approach. We discuss the former option in this section, and the latter in the next.

Since current rendering architectures expose fine-grained incremental state changes, it would be advantageous if a system like Spark could take advantage of this capability, only submitting changes to the hardware when needed.

A simple approach to optimizing state changes is to *shadow* the graphics hardware state in CPU memory, and compare new state to the shadowed copy before submitting changes to the hardware. This approach reduces the number of state changes that are submitted, but at a cost of additional CPU overhead. This trade-off is unlikely to be acceptable for most high-performance rendering applications.

Instead, we propose to exploit coherence in a less reactive fashion. We can observe that most rendering applications bind different kinds of state at different rates. For example, the same render-target and camera-parameter bindings will typically be used to render an entire scene. In turn, a pass-per-light forward renderer will use the same lighting parameters and shadow-map resource bindings across many scene objects.

We can imagine a system for composing shader objects (as in Section 5.4.4) using a *stack*. Shader objects could be pushed onto and popped from the stack, with longer-lived objects (e.g., representing the camera) remaining near the bottom of the stack. Rendering operations could be submitted based on the combination of shader objects presently on the stack. Between two successive rendering operations, only state pertaining to the shader objects that have been pushed and popped needs to be submitted to the hardware; any state related to objects further up the stack would remain unchanged. With suitable constraints (e.g., a parent/child hierarchy over shader objects) it would be possible for a compiler to pre-compute the state changes involved in pushing/popping shader objects of a particular type, thus allowing for a reduction in state-change overhead without heavy run-time CPU costs.

5.4.6 Evolving Rendering Achitectures

In this section, we discuss possible changes to future rendering architectures. While Spark has motivated these proposals, we believe that they can be beneficial for future rendering architectures no matter what kind of shading languages dominate.

Traditionally, the interface between a host application and a graphics processor takes the form of a *command-buffer processor* [Ake93]. An application running on a CPU

thread packs incremental state-change commands and rendering operations into a command-buffer, to be submitted and consumed by the command-buffer processor, which in turn dispatches work to the rendering pipeline. Even as the degree of parallelism in both CPUs and GPUs has increased, this sequential interface remains as a bottleneck between the two.

This implementation choice has influenced the application-facing interface of rendering architectures like OpenGL and Direct3D. Both architectures support APIs that allow for incremental state-change commands to be submitted to the architecture; the OpenGL architecture is explicitly described in terms of a state machine. This availability of such fine-grained state changes has some unfortunate consequences, which we will discuss in more detail.

Parallelization

From the perspective of an application, the rendering architecture (e.g., the OpenGL state machine) is *global*, *mutable* state. Attempting to parallelize an application that uses a rendering architecture is, in spirit, similar to parallelizing one that makes heavy use of global variables. Historically, OpenGL has addressed this issue by only allowing a rendering context to be accessed from one thread at a time.

More recently, D3D11 allows multiple application threads to create *commands lists* of "deferred" commands. These command lists may be submitted to the rendering architecture, on a single thread, for subsequent execution. The pipeline is reset to a default configuration before and after each command list that is submitted. Each new parallel thread that creates a command list must configure the entire pipeline from this default thread.

We see two weaknesses to this approach, as parallelism increases:

- There is still a global, sequential stream of commands from the application to the rendering architecture. This global point of sequencing is a potential bottleneck and source of contention.

- Because each parallel thread must set up the pipeline "from scratch," the use of incremental deltas saves less CPU time as parallelism increases.

Validation and Optimization

Consider the task of implementing a rendering architecture like OpenGL or Direct3D. An implementation receives a stream of incremental state-change commands, punctuated by rendering commands (e.g., to draw some triangles). In practice, the implementation cannot know *a priori* whether the next command to arrive will cause a state change, or will do some rendering with the current state.

Certain combinations of state might require validation or enable optimizations. For example, we might need to validate that the number of type of attributes output by a GS shader procedure match the inputs declared in a PS shader procedure. If we detect that a PS shader procedure does not modify the depth value of fragments, we might wish to enable an optimization in which we perform a conservative depth test earlier in the pipeline. Note that in general, validation and optimization may require information about the configuration of more than one pipeline stage.

The combination of fine-grained state changes, and the need for validation/optimization, creates implementation challenges. Because the stream of state-change commands is not known in advance, an implementation will typically track "dirty" bits when state changes, and perform validation/optimization only when it encounters a rendering command. Because optimization might be requested several times per frame, an implementation is limited in the kinds of optimization it can perform; a time-consuming optimization could actually end up wasting time overall.

Some implementations may amortize out the cost of time-consuming optimizations by caching their results (e.g., optimized shader code). If the same pipeline configuration is encountered again later, the previously optimized results can be re-used. Caching approaches can improve average-case performance, but they do nothing to improve worst-case performance. When a new object in a game scene first moves into view, the

frame-rate may "stutter" while the rendering architecture implementation performs expensive one-time optimizations. If an application also maintains a cache—e.g., a cache of specialized shaders derived from an über-shader—then there is a chance that the policies of the two caches will conflict and lead to thrashing.

The Direct3D 10 architecture [Bly06] coarsens the granularity of state changes (as compared to, e.g., OpenGL) by introducing *state objects* which encapsulate the configuration of all or part of a pipeline stage (e.g., the IA stage, or the blending part of the OM stage). Optimization and validation that pertains to only one state object can be performed when the object is created—perhaps at application startup time—and need not impact run-time performance. However, this approach does not affect the situation for inter-stage validation or optimization.

The OpenGL system provides for *display lists*, which may be used to record and play back sequences of state-change and rendering commands. In principle, an implementation of OpenGL can inspect the pipeline configuration that would result from playing back a display list and validate/optimize that configuration. Since the representation of a display list is opaque to an application, the implementation could transparently cache the optimized data in the display list. In practice, however, display lists have proven incapable of cleanly encapsulating pipeline configurations. The chief problem is that the meaning and effect of a display list may be changed by almost any state-change commands that precede it.

Proposal

Given the above issues, we propose a fundamental change in the interface to rendering architectures. Rather than submitting incremental state changes, followed by rendering commands that rely on the previously-configured state, the interface should allow programmers to construct and submit *complete* rendering commands. These rendering command should unambiguously describe the pipeline configuration to be used. Spark shader objects (instances of shader classes) are an example of complete rendering commands, but are by no means the only possible realization.

An interface based on complete rendering commands can address the issues we identify above:

- Because the configuration used by a command no longer depends on previously-submitted commands, single-threaded submission is no longer required. Multiple threads can submit independent rendering commands in parallel.
- If complete rendering commands are created from templates (e.g., Spark shader classes) that describe an entire pipeline configuration, then an implementation can perform costly inter-stage validation and optimization at the time these templates are created, rather than relying on run-time caching.

A Spark shading class can be seen as describing a class of these complete rendering commands. Like an OpenGL display list, a Spark shader class can describe an entire pipeline configuration. Unlike a display list, though, a shader class is *explicitly* parameterized: the only things that may affect the results of a Spark shader are its parameters.

This distinction means that an implementation can perform optimization on a Spark shader class (or similar representation) that were not possible on a display list. Any part of the pipeline configuration that does not depend on an explicit parameter is a candidate for optimization; nothing the user can do at run-time may invalidate such an optimization. In turn, any part of the configuration that is parameter-dependent might change at run-time in response to parameter changes, so an implementation should avoid expensive optimizations.

A complete rendering command can be seen as simply the combination of a particular shader class with its parameter data. In the simplest case it might be represented as two pointers—one each for code and data—in a *closure*.

Of course, achieving such an interface to a rendering architecture is not without challenges. For example, in current architectures the sequential nature of command submission provides a benefit, in insulating the programmer from the details of parallel execution; an implementation may exploit parallelism in executing commands,

but must ultimately preserver the *appearance* of sequential execution. If an interface switches to explicitly parallel submission, then programmers must take responsibility for scheduling rendering operations, so as to avoid data races.

In addition, by dispensing with the notion of a single global, mutable pipeline configuration, we are effectively *virtualizing* that state. Carrying that virtualization through to hardware implementation will have costs. For example, rather than store configuration state for a rasterizer in dedicated storage and update it incrementally, a GPU might instead store multiple state objects for the rasterizer in general-purpose memory, and rely on caches to load the correct state into the rasterizer as needed. Such caches would introduce costs in peformance, area, and power.

Despite these challenges, we have reason to believe that our proposal is in keeping with general trends in GPU hardware architectures. For example, the OpenGL extension which introduces UAV-like atomics read-modify-write operations [BBL+10] also introduces explicit memory *barrier* operations into the OpenGL interface; programmers are expected to take responsibility for some parallel scheduling decisions. Additionally, NVIDIA's Kepler architecture [NVI11] supports "bindless" textures; texture-related state is virtualized and stored in the memory hierarchy.

Given these trends, we believe that a rendering architecture that completely eschews fine-grained state changes will be practical and beneficial in the near future.

Chapter 6

Conclusion

Spark is a language for real-time shading that enables greater modularity and composability than current procedural shader-per-stage languages. It extends prior work on declarative, graph-based shader representations by supporting algorithms that require control flow, enabling user-defined units of modularity, and supporting programmable groupwise operations on modern pipelines. Our experience using Spark has shown that it can be effectively used to encapsulate complex effects like tessellation into reusable modules.

While procedural shading languages achieve good performance by directly exposing the topology of the rendering pipeline, our results indicate that equivalent performance can still be achieved with a language at a higher level of abstraction, after applying appropriate global optimizations on shader programs.

We hope that our discussion of the design of Spark will be useful to future language designers and rendering system architects. In particular, we believe that Spark can serve as an example of synthesis between the modularity and composability of declarative programming and the expressive power of procedural shading languages.

Mainstream computing is increasingly driven by low-power devices like phones and tablets. In order to achieve higher performance with low power consumption, these

devices make use of optimized processing pipelines that combine fixed-function and programmable operations – not just for 3D rendering, but also for media processing during video capture and playback. In order to harness the power of such pipelines, software developers need rendering architectures to expose a programming model that achieves both flexibility and performance.

We hope that Spark can lead the way.

Appendix A

Glossary

attribute A node in a *shader graph*, representing a computation to be performed as well as the result of that computation (e.g., a per-fragment `float` computed as the dot product of normal and light vectors). Every attribute has both a *rate of computation* (e.g., per-fragment) and a data type (`float`).
Through the identification of rates of computation and *record types*, an attribute with rate `@R` is also a field of the record type `R`. The value of such a field—that is, the value of the attribute for a particular record—is accessed through *projection*.

groupwise An operation or dataflow that considers multiple *records* of input or output: e.g., interpolation from multiple control points to a tessellated vertex.

kernel A system-provided procedure that defines the behavior of a particular *pipeline stage*. In general, a kernel pops *records* off of input streams, applies fixed-function or user-defined operations to produce new records, and pushes the results to output streams.

per-stage shader A variety of *shader* that describes a function or procedure that will run in a single *stage* of a *pipeline*.

pipeline A collection of *stages* which communicate *records* through dataflow streams.

pipeline shader A variety of *shader* that may include code that will run in more than one *stage* of a *pipeline*.

plumbing The process of transporting or converting data computed at one rate (e.g., a per-vertex texture coordinate) to make it available at another rate (e.g., per-fragment). In Spark, plumbing is achieved by invoking *plumbing operators*.

plumbing operator In general, any function with *rate-qualified types* for input and output, where not all of the rates are the same. Plumbing operators may be invoked—either explicitly by a programmer or implicitly by a compiler—to perform *plumbing* by converting data from one rate to another.

pointwise Operations that are defined in terms of *attributes*, without reference to particular *records*. For example, taking the dot product of two per-fragment vectors, or plumbing a per-vertex color to a per-fragment color.

rate of computation A qualifier of an *attribute* that conceptually describes how often that attribute will be computed: e.g., per-vertex, per-control-point, per-fragment, etc.
In Spark, every rate of computation is in one correspondence with a *record type*. For example, the rate of per-fragment computation, `@Fragment`, corresponds to the type of fragment records `Fragment`.

rate-qualified type A kind of type, that combines a *rate of computation* (e.g., `@Fragment`) with a data type (e.g., a proper type like `float`). *Attributes* in Spark have rate-qualified types.

record An instance of a *record type*. Code running in the *stages* of a *pipeline* can construct records, and the stages communicate records over streams.

record type A kind of abstract data type used to define the overall structure and dataflow of a *pipeline*. The streams that connect different pipeline *stages* each carry records of particular types: vertices, control points, fragments, etc.
In Spark, every record type is in one-to-one correspondence with a *rate of computation*, and the *attributes* with rate `@R` correspond to the fields of the record type `R`.

shader A unit of application code that describes the appearance of rendered objects (shape, transformation, animation, color, etc.) and that runs in the context of a rendering system (e.g., a *pipeline*).

shader graph A representation of *shader* code as a dataflow graph. Nodes in a shader graph represent *attributes* to be computed, and each has a *rate of computation*.

stage A given stage in a *pipeline* is connected to other stages by input and output streams which carry *records*. The processing at a particular stage is defined by its *kernel*.

Bibliography

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *Proceedings of ICSE 2002: International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM.

[Ado11] Adobe. Pixel Bender 3D. `http://labs.adobe.com/technologies/pixelbender3d/`, 2011.

[Ake93] Kurt Akeley. Reality Engine graphics. In *Proceedings of SIGGRAPH 1993*, pages 109–116, New York, NY, USA, 1993. ACM.

[AR05] Chad Austin and Dirk Reiners. Renaissance: A functional shading language. In *Proceedings of Graphics Hardware 2005*, pages 1–8, New York, NY, USA, 2005. ACM.

[BBL+10] Jeff Bolz, Pat Brown, Barthold Lichtenbelt, Bill Licea-Kane, Eric Werness, Graham Sellers, Greg Roth, Nick Haemel, and Pierre Boudier Piers Daniell. OpenGL extension EXT_shader_image_load_store. `http://www.opengl.org/registry/specs/EXT/shader_image_load_store.txt`, 2010.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. *SIGPLAN Notices*, 25:303–311, September 1990.

[BCH+96] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for Dylan. *SIGPLAN Notices*, 31(10):69–82, October 1996.

[Bex] Tobias Bexelius. GPipe. `http://www.haskell.org/haskellwiki/GPipe`.

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *Transactions on Graphics*, 23(3):777–786, August 2004.

[Bly06] David Blythe. The Direct3D 10 system. *Transactions on Graphics*, 25(3):724–734, July 2006.

[BNE09] Anders Bach Nielsen and Erik Ernst. Virtual class support at the virtual machine level. In *Proceedings of VMIL 2009: Workshop on Virtual Machines and Intermediate Languages*, pages 1:1–1:10, New York, NY, USA, 2009. ACM.

[Buc05] Ian Buck. *Stream computing on graphics hardware*. PhD thesis, Stanford, CA, USA, 2005. AAI3162314.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13:377–387, June 1970.

[Coo84] Robert L. Cook. Shade trees. In *Proceedings of SIGGRAPH 1984*, pages 223–231, New York, NY, USA, 1984. ACM.

[Cox90] Brad J. Cox. Planning the software industrial revolution. *IEEE Software*, 7:25–33, November 1990.

[CPl11] C++11 standard. `http://www.iso.org/iso/catalogue_detail?csnumber=50372`, October 20111. 14882:2011.

[DG87] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An overview. In *Proceedings of ECOOP 1987*, pages 151–170, London, UK, UK, 1987. Springer-Verlag.

[DH87] Roland Ducournau and Michel Habib. On some algorithms for multiple inheritance in object-oriented programming. In *Proceedings of ECOOP 1987*, pages 243–252, London, UK, UK, 1987. Springer-Verlag.

[Dij82] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[DMN68] Ole-Johan Dahl, Bjørn Myhrhaug, and Kristen Nygaard. *SIMULA 67 Common Base Language*. Norwegian Computing Center, 1968.

[Ell04] Conal Elliott. Programming graphics processors functionally. In *Proceedings of Haskell 2004: ACM SIGPLAN Workshop on Haskell*, pages 45–56, New York, NY, USA, 2004. ACM.

[EOC06] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proceedings of POPL 2006*, pages 270–282, New York, NY, USA, 2006. ACM.

[Ern99] Erik Ernst. *gbeta – a language with virtual attributes, block structure, and propagating, dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[Ern01] Erik Ernst. Family polymorphism. In *Proceedings of the ECOOP 2001*, pages 303–326, London, UK, UK, 2001. Springer-Verlag.

[Ern02] Erik Ernst. Safe dynamic multiple inheritance. *Nordic Journal of Computing*, 9:191–208, September 2002.

[Ern03] Erik Ernst. Higher-order hierarchies. In *Proceedings of ECOOP 2003*, pages 303–328, Darmstadt, Germany, July 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[HL90] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. In *Proceedings of SIGGRAPH 1990*, pages 289–298, New York, NY, USA, 1990. ACM.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *C# Language Specification*. Addison-Wesley, Boston, MA, USA, 2003.

[IB82] Alan H. Ingalls and Daniel H. H. Borning. Multiple inheritance in Smalltalk-80. In *Proceedings of the National Conference on Artificial Intelligence*, pages 234–237, Pittsburgh, PA, USA, August 1982.

[KBR03] John Kessinich, Dave Baldwin, and Randi Rost. The OpenGL® Shading Language, version 1.05. `http://www.opengl.org`, February 2003.

[KDR+02] Ujval Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. The Imagine stream processor. In *Proceedings ICCD 2002: International Conference on Computer Design*, pages 282–288. IEEE, September 2002.

[Ken] Andrew Kennedy. Units of measure in F#. `http://blogs.msdn.com/b/andrewkennedy/archive/2008/08/29/units-of-measure-in-f-part-one-introducing-units.aspx`.

[Ken97] Andrew J. Kennedy. Relational parametricity and units of measure. In *Proceedings of POPL 1997*, pages 442–455, New York, NY, USA, 1997. ACM.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of ECOOP 2001*, pages 327–353, London, UK, 2001. Springer-Verlag.

[Khr] Khronos OpenCL Working Group. The OpenCL specification, version 1.0. `http://www.khronos.org/registry/cl/specs/opencl-1.0.pdf`.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of ECOOP 1997*. Springer-Verlag, 1997.

[KR91] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[Kro85] Stein Krogdahl. Multiple inheritance in SIMULA-like languages. *BIT Numerical Mathematics*, 25:318–326, 1985. 10.1007/BF01934377.

[KW09] Roland Kuck and Gerold Wesche. A framework for object-oriented shader design. In *Proceedings of ISVC 2009: International Symposium on Advances in Visual Computing*, pages 1019–1030, Berlin, Heidelberg, 2009. Springer-Verlag.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of CGO 2004: International Symposium on Code Generation and Optimization*, Palo Alto, California, Mar 2004.

[Lef06] Aaron Lefohn. *Glift: Generic Data Structures for Graphics Hardware*. PhD thesis, Computer Science, University of California, Davis, September 2006.

[LKS+06] Aaron Lefohn, Joe Kniss, Robert Strzodka, Shubhabrata Sengupta, and John Owens. Glift: Generic, efficient random-acccess GPU data structures. *Transactions on Graphics*, 25(1):60 – 99, 2006.

[LMMP89] Ole Lerhman Madsen and Birger Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA 1989*, pages 397–406, New York, NY, USA, 1989. ACM.

[LO04] Calle Lejdfors and Lennart Ohlsson. PyFX – an active effect framework. In *Proceedings of SIGRAD 2004*, pages 17–24, Gävle, Sweden, 2004. Linköping University Electronic Press.

[LS02] Paul Lalonde and Eric Schenk. Shader-driven compilation of rendering assets. *Transactions on Graphics*, 21(3):713–720, 2002.

[LSNCn09] Charles Loop, Scott Schaefer, Tianyun Ni, and Ignacio Castaño. Approximating subdivision surfaces with Gregory patches for hardware tessellation. *Transactions on Graphics*, 28:151:1–151:9, 2009.

[McC00] Michael D. McCool. SMASH: A next-generation API for programmable graphics accelerators. Technical Report CS-2000-14, University of Waterloo, August 2000.

[McI68] Douglas McIlroy. Mass-produced software components. In P. Naur and B. Randell, editors, *Software Engineering, Report on a Conference Sponsored by the NATO Science Committee*, pages 138–155, Brussels, Belgium, 1968. Scientific Affairs Division, NATO.

[MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: A system for programming graphics hardware in a C-like language. *Transactions on Graphics*, 22:896–907, 2003.

[Mic02] Microsoft. Shader model 1 (DirectX HLSL). `http://msdn.microsoft.com`, 2002.

[Mic10a] Microsoft. Direct3D 11 reference. `http://msdn.microsoft.com`, 2010.

[Mic10b] Microsoft. Effect format (Direct3D 11). `http://msdn.microsoft.com`, 2010.

[MLD97] Stephen Mann, Nathan Litke, and Tony DeRose. A coordinate free geometry ADT. Technical report, Computer Science Department, University of Waterloo, June 1997.

[Moo86] David A. Moon. Object-oriented programming with Flavors. *SIGPLAN Notices*, 21(11):1–8, June 1986.

[MPO08a] Adriaan Moors, Frank Piessens, and Martin Odersky. In *Proceedings of the 2008 International Workshop on Foundations of Object-Oriented Langauges*, FOOL '08, 2008.

[MPO08b] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. *SIGPLAN Notices*, 43:423–438, October 2008.

[MQP02] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proceedings of Graphics Hardware 2002*, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics.

[NCM04] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. *SIGPLAN Notices*, 39:99–115, October 2004.

[Nie07] Anders Bach Nielsen. Ensuring that user defined code does not see uninitialized fields. In *Proceedings of ICOOOLPS 2007*, Berlin, Germany, July 2007.

[NVI07] NVIDIA. CUDA technology. `http://www.nvidia.com/cuda`, 2007.

[NVI10] NVIDIA. Introduction to CgFX. `http://developer.nvidia.com`, 2010.

[NVI11] NVIDIA. Kepler GK110. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2011.

[OAC+04] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[OCRZ03] Martin Odersky, Vinvent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings of ECOOP 2003*, pages 201–224, Darmstadt, Germany, July 2003. Springer.

[OMO10] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. *SIGPLAN Notices*, 45:341–360, October 2010.

[OP10] Jiawei Ou and Fabio Pellacini. SafeGI: Type checking to improve correctness in rendering system implementation. *Computer Graphics Forum*, 29(4):1269–1277, 2010.

[Par71] David L. Parnas. Information distribution aspects of design methodology. In *Proceedings of IFIP Congress*, 1971.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, December 1972.

[PBMH02] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *Transactions on Graphics*, 21(3):703–712, July 2002.

[Per85] Ken Perlin. An image synthesizer. In *Proceedings of SIGGRAPH 1985*, pages 287–296, New York, NY, USA, 1985. ACM.

[Pho73] Bui Tuong Phong. *Illumination for Computer-Generated Images*. PhD thesis, 1973.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, USA, 2002.

[PMTH01] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics

hardware. In *Proceedings of SIGGRAPH 2001*, pages 159–170, New York, NY, USA, 2001. ACM.

[POAU00] Mark S. Peercy, Marc Olano, John Airey, and P. Jeffrey Ungar. Interactive multi-pass programmable shading. In *Proceedings of SIGGRAPH 2000*, pages 425–432, New York, NY, USA, 2000. ACM.

[Pur04] Timothy John Purcell. *Ray tracing on a stream processor*. PhD thesis, Stanford, CA, USA, 2004. AAI3128683.

[PW90] William Pugh and Grant Weddell. Two-directional record layout for multiple inheritance. *SIGPLAN Notices*, 25:85–91, June 1990.

[SAF+10] Mark Segal, Kurt Akeley, Chris Frazier, Jon Leech, and Pat Brown. The OpenGL® graphics system: A specification (version 4.0 (core profile) - march 11, 2010). `http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf`, 2010.

[Sca] Adding virtual classes to Scala. `https://wiki.scala-lang.org/display/SIW/VirtualClassesDesign`.

[SDNB02] Nathanael Shärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. Technical report, Oregon Graduate Institute School of Science & Engineering, 2002.

[SFB+09] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. GRAMPS: A programming model for graphics pipelines. *Transactions on Graphics*, 28(1):1–11, 2009.

[SGG+05] Jeremy G. Siek, Douglas Gregor, Ronald Garcia, Jeremiah Willcock, Jaakko Järvi, and Andrew Lumsdaine. Concepts for C++0x. Technical Report N1758=05-0018, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2005.

[Sha96] Andrew Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[Sny87] Alan Snyder. *Inheritance and the development of encapsulated software systems*, pages 165–188. MIT Press, Cambridge, MA, USA, 1987.

[Str89] Bjarne Stroustrup. Multiple inheritance for C++. *Computing Systems*, 2(4):367–95, 1989.

[Tan] Audrey Tang. Perl 6 method resolution order. `http://use.perl.org/~autrijus/journal/25768`.

[VPBM01] Alex Vlachos, Jörg Peters, Chas Boyd, and Jason L. Mitchell. Curved PN triangles. In *Proceedings of I3D 2001*, pages 159–166, New York, NY, USA, 2001. ACM.

[vR] Guido van Rossum. Unifying types and classes in Python 2.2. `http://www.python.org/download/releases/2.2.3/descrintro/#mro`.

[Wad90] Philip Wadler. Comprehending monads. In *Proceeding of LFP 1990: ACM Conference on LISP and Functional Programming*, pages 61–78, New York, NY, USA, 1990. ACM.

[YHGT10] Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-time concurrent linked list construction on the gpu. *Computer Graphics Forum*, 29(4):1297–1304, 2010.