*To appear in ACM TOG 30(4).*

# Spark: Modular, Composable Shaders for Graphics Hardware

Tim Foley*
Intel Corporation
Stanford University

Pat Hanrahan
Stanford University

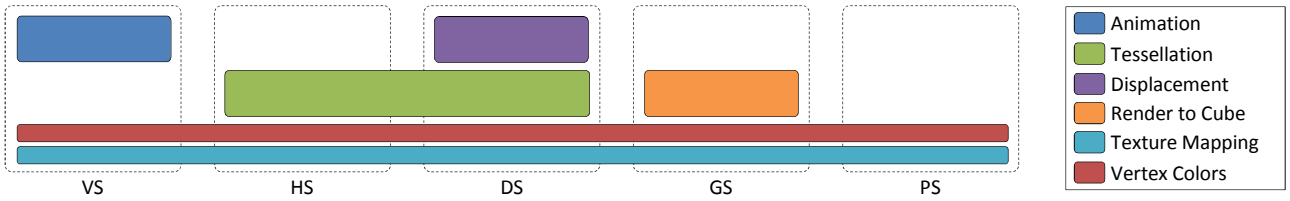


**Figure 1:** *A complex shading effect decomposed into user-defined modules in Spark. The dashed boxes show the programmable stages of the Direct3D 11 pipeline; the colored boxes show different* concerns *in the program. Some logical concerns* cross-cut *multiple pipeline stages.*

## Abstract

In creating complex real-time shaders, programmers should 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, and its implementation for modern graphics hardware that improves support for separation of concerns into modules. A Spark *shader class* can encapsulate code that maps to more than one pipeline stage, and can be extended and composed using object-oriented inheritance. In our tests, shaders written in Spark achieve performance within 2% of HLSL.

**Keywords:** shading language, graphics hardware, modularity

## 1 Introduction

Authoring compelling real-time graphical effects is challenging. Where once shaders comprised tens of lines of code targeting two programmable stages in a primarily fixed-function pipeline, increasing hardware capabilities have enabled rapid growth in complexity. Achieving a particular effect requires coordination of shaders, fixed-function hardware settings, and application code.

In light of the increasing scope and complexity of this programming task, 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 problem of *separation of concerns*: the factoring of logically distinct program features into localized and independent modules.

Figure 1 shows a complex rendering effect that uses every stage of the Direct3D 11 (hereafter D3D11) pipeline. In a single pass, an animated, tessellated and displaced model is rendered simultaneously to all six faces of a cube map. The dashed boxes represent the programmable stages of the D3D11 pipeline. The colored boxes represent logically distinct features or *concerns* in the program. Some concerns (such as tessellation) intersect multiple stages of the rendering pipeline. These are *cross-cutting concerns* in the terminology of aspect-oriented programming [Kiczales et al. 1997].

Ideally, a shading language would allow each logical concern to be defined as a separate, reusable module. 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. A programmer should expend that effort once, and re-use the resulting module many times.

Modern shaders comprise two kinds of code, which we will call *pointwise* and *groupwise*. Early programmable graphics hardware exposes vertex and fragment processing with a simple mental model: a user-defined kernel is mapped over a stream of input. This ensures that individual vertices and fragments may be processed independently (or in parallel), and so shading algorithms are defined pointwise for a single stream element. In contrast, groupwise operations, such as primitive assembly or rasterization, apply to an aggregated group of stream elements. Where historically groupwise operations have been enshrined in fixed-function stages, current rasterization pipelines such as Direct3D [Blythe 2006; Microsoft 2010a] and OpenGL [Segal et al. 2010] include user-programmable stages that can perform groupwise operations: e.g., basis change, interpolation, and geometry synthesis.

Today, the most widely used GPU shading languages are HLSL [Microsoft 2002], GLSL [Kessinich et al. 2003], and Cg [Mark et al. 2003]. These are *shader-per-stage* languages: a user configures the rendering architecture with one shader procedure for each programmable stage of the pipeline. Figure 2 shows a possible mapping of the effect in Figure 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 2. Each per-vertex attribute (color, texture coordinate, etc.) that is subsequently used in per-fragment computations requires code to *plumb* it through each intermediate stage. When tessellating a coarse mesh into a fine mesh, for example, we must

*e-mail: tim.foley@intel.com, tfoley@graphics.stanford.edu

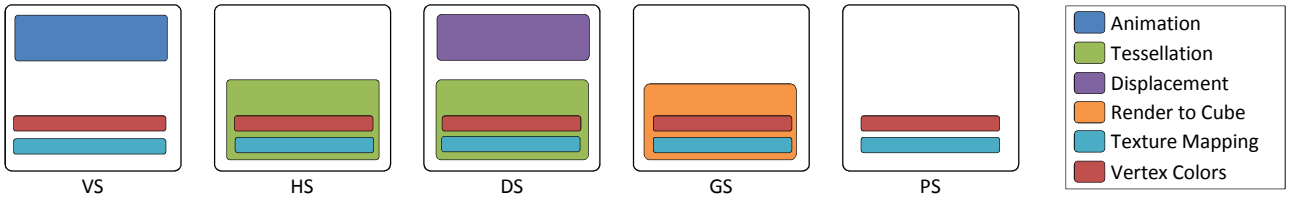


**Figure 2:** *The rendering effect in Figure 1, adapted to the constraints of shader-per-stage languages. The labeled boxes represent the procedures for the Direct3D 11 Vertex, Hull, Domain, Geometry, and Pixel Shader respectively. The colored boxes represent subsets of the program pertaining to different features. Some logically coherent program features must be divided, and some orthogonal ones merged, to meet the constraints of shader-per-stage programming.*

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.

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) [Proudfoot et al. 2001]. In our work, we set out to explore whether a pipeline-shader language might yield better tools for separation of concerns. We sought to take the key ideas of RTSL, and extend it to support the capabilities of modern rendering pipelines: most notably dynamic control flow and user-defined groupwise operations.

In this paper we describe a shading language, Spark, and its implementation for the D3D11 pipeline. Spark allows us to achieve the modularization depicted in Figure 1. A Spark programmer may define independent *shader classes*, each encapsulating a logical feature – even features 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 groupwise and pointwise features are composed, the Spark compiler automatically generates any required plumbing code by instantiating user-defined *plumbing operators*. In order to achieve good performance, we perform global (that is, inter-stage) optimization on composite shaders. We have found that shaders written in Spark achieve performance within 2% of HLSL shaders.

## 2 Background

### 2.1 Declarative and Procedural Shaders

Modern shading languages derive from Cook's shade trees [1984] and Perlin's image synthesizer [1985]. It is telling, then, that these two works differ on a key design decision: Cook's language is declarative, while Perlin's is procedural.

Shade trees represent a shader in terms of its dataflow graph. The graphs are authored using a declarative "little language." Surface, light, atmosphere, and displacement shaders may be specified as separate, modular graphs. The rendering engine then composes these modules by "grafting" one shade tree onto another. Note, however, that a shade tree cannot represent algorithms with looping or conditional control flow, nor can it represent access to read/write memory (e.g., mutable local variables).

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. 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).

### 2.2 RenderMan Shading Language

Hanrahan and Lawson [1990] 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 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 renderer to 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 *computation rates* 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. The expectation of the user is that uniform computations occur at a lower rate, and may thus be less costly.

### 2.3 Real-Time Shading Language

The Stanford Real-Time Shading Language (RTSL) [Proudfoot et al. 2001] 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.

RTSL is syntactically similar to RSL, and superficially looks like a procedural language. The language is, however, declarative: sufficient restrictions are placed on 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. Subsequently, these composed shader graphs are partitioned into sub-graphs according to computation rates, to generate executable code for each programmable pipeline stage.

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

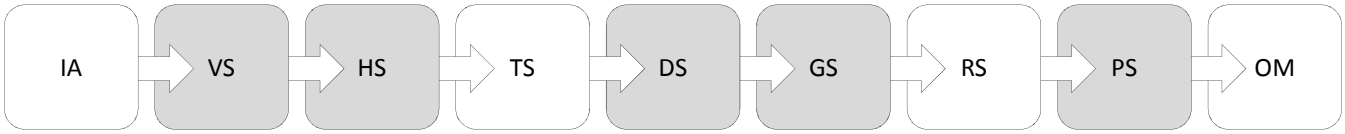


**Figure 3:** *Nominal structure of the Direct3D 11 rendering pipeline. Programmable stages are shown in gray. In practice, fixed-function primitive-assembly stages precede the HS and GS, but these are not included in the nominal pipeline.*

can express only pointwise operations, 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 intermediate pipeline stages. 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.

The SMASH API [McCool 2000] supports rate-qualified *sub-shaders*, with a DAG-based representation similar to RTSL. Renaissance [Austin and Reiners 2005] 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.

### 2.4 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. [2003]. 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., 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 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."

Programmers may layer more flexible abstractions on shader-per-stage languages by metaprogramming. An *über-shader*, a shader implementing the sum of all desired features, may be pre-processed to generate specialized shaders on demand. Effect systems [NVIDIA 2010; Microsoft 2010b; Lalonde and Schenk 2002] allow a set of per-stage shaders to be encapsulated, but do not address program concerns at other scales. Shader metaprogramming frameworks [McCool et al. 2002; Lejdfors and Ohlsson 2004; Kuck and Wesche 2009] apply host-language abstractions to shaders.

Pixel Bender 3D [Adobe 2011] separates shader code into transformation and material concerns. A material shader uses separate procedures 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.

### 2.5 Direct3D 11

Figure 3 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 will briefly discuss the names and responsibilities of these stages, for the benefit of readers who may be unfamiliar with D3D11. Several stages manipulate vertices of one kind or another; as such, we will also define names for the different kinds of vertices to aid in disambiguation.

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 a kernel over the assembled vertices to produce a stream of *coarse vertices*, representing a base mesh. These coarse vertices are then assembled into primitives 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. As such, the HS outputs *control points* as well as attributes for each *patch*, such as edge tessellation rates.

The control point and patch data flow past a fixed-function Tessellator (TS) stage, which augments them with a set of parametric locations for new vertices in the tessellation 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*. The DS may also perform pointwise operations like displacement mapping.

Fine vertices are assembled into primitives that are then processed by the Geometry Shader (GS). The GS applies a procedural kernel to each primitive, that may perform almost arbitrary computation to generate a stream of *raster vertices* that describe zero or more output primitives. For example, the GS may duplicate each input primitive six times, projecting 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 kernel over the *rasterized fragments* to produce *shaded fragments*. Shaded fragments are composited onto render-target *pixels* by a fixed-function Output Merger (OM).

The VS and PS stages each perform a functional map over their input stream. Thus, HLSL kernels for the VS and PS describe only pointwise (one-to-one) operations. In contrast, the remaining programmable stages apply a kernel to an aggregated *group* of inputs, and so HLSL allows for groupwise operations, both many-to-one and many-to-many. For example, the HS must have access to all the coarse vertices in the neighborhood of a primitive to transform it into a Bézier basis. Similarly, the DS must have access to all of the Bézier control points for a patch to perform many-to-one interpolation. Furthermore, the GS has the capability to emit zero or more raster vertices. As such the GS can perform almost arbitrary amplification, decimation, or synthesis of geometry.

Note, however, that 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. A single per-stage procedure for these languages, then, may end up combining both pointwise and groupwise concerns.

# 3 Design Goals

Our design 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.

**Composability** It should be possible to combine or extend thoughtfully designed modules to create new shaders, without resort to copy-paste programming.

**Automatic plumbing** Attributes defined in pointwise 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 for unrelated pipeline stages. This is a consequence of the preceding two goals, if we are to support separation of pointwise and groupwise concerns.

In addition, once we decided that our work would build upon the pipeline-shader approach of RTSL, we felt the need to enumerate some additional design goals. These goals served 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 must be able to expose the full capabilities of the D3D11 rendering pipeline. Furthermore, the same language should be usable with future extensions to the pipeline.

**Domain-motivated rather than domain-specific** Specialization to the domain of real-time shading is acceptable (we do not aim to build a general-purpose parallel programming language). It is not acceptable to impose a particular domain model (e.g., a system-defined interface for surfaces and lights).

**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.

# 4 Shader Programming Abstraction

In order to separate the specification of groupwise operations like tessellation from attributes defined in pointwise shading code, we must define an interface between them. For example, surface and light shaders should determine *which* per-vertex attributes are plumbed through the pipeline, but a tessellation scheme must define *how* interpolation is used to achieve that plumbing.

Figure 4 illustrates how our shader programming abstraction defines this interface. Pointwise shader code is expressed in a shader graph, while groupwise operations are expressed as procedures that run in the rendering pipeline. 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 (e.g., [Sugerman et al. 2009]). 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.

## 4.1 Shader Graphs

In our abstraction, pointwise shading code is defined using *shader graphs*: DAGs representing shading computations. The top of Figure 4 depicts a shader graph. Each node in the graph represents either an input to the shader or a value it computes.

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. We will sometimes refer to shader-graph nodes as *attributes*: e.g., a per-fragment color attribute.

The graph in Figure 4 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 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 nodes may also represent shader outputs used by the renderer: for example, `RS_Position` represents the projected position consumed by the rasterizer.

## 4.2 Pipeline Model

The bottom of Figure 4 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 2.5: coarse vertices, fragments, etc. For example, the stream connecting the DS and GS carries `FineVertex` records.

Each programmable stage repeatedly reads one or more records from its input stream, and applies a *kernel* to construct and emit zero or more records on its output stream. The stages of the D3D11 pipeline fall into a few categories. The VS and PS stages always read a single input record and construct one output record. The HS and DS stages read an aggregate of records (e.g., all of the coarse vertices in the neighborhood of a base-mesh primitive), and use this aggregate to construct a number of independent output records.

In each of the above cases, the kernel of the stage is system- rather than user-defined. The GS stage, however, takes a *user-defined* kernel: a procedure which may construct and output zero or more `RasterVertex` records.

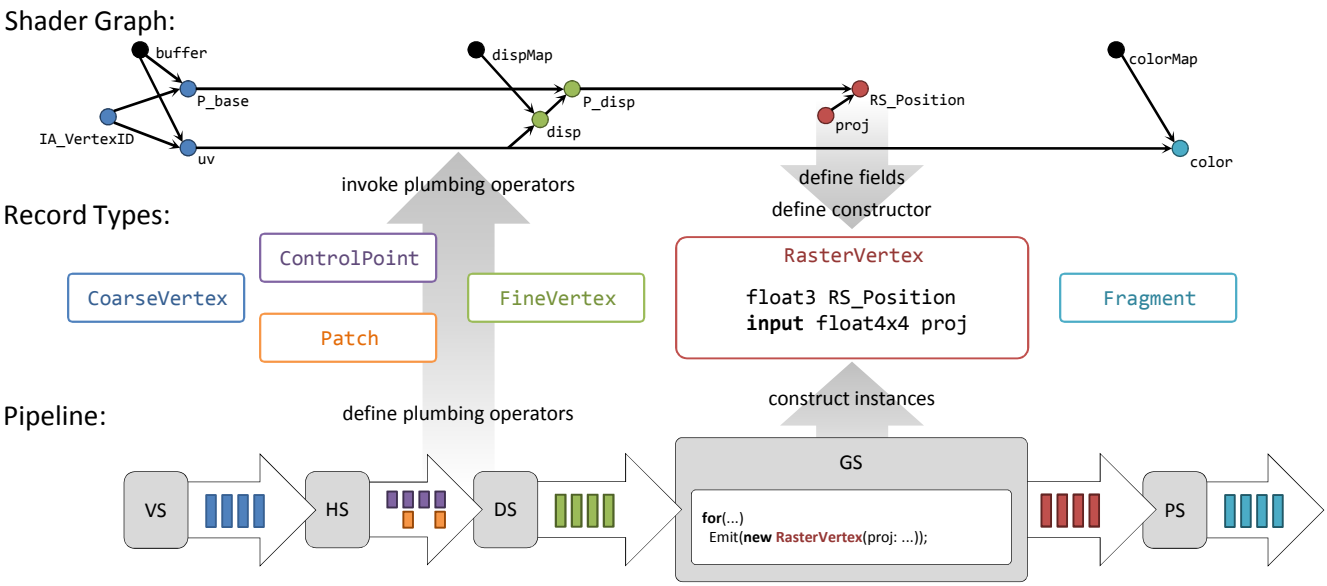


**Figure 4:** *Shader graphs are mapped to the rendering pipeline with the use of* record types. *Nodes in the shader graph are colored according to their rate of computation: each node corresponds to a field in the associated record type. Procedural kernels running in the rendering pipeline construct, manipulate and communicate records.*

### 4.3 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 4 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 is the key to mapping a shader graph to the rendering pipeline. When the system-defined kernel for the VS stage 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.

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 intuitive, this approach does not accurately reflect some new pipeline stages. For example, the HS stage constructs both patches and control points, and so performs computations at two different rates.

The identification of rates and record types also allows for a clean model of the GS stage. A user-defined GS kernel is written to perform a particular groupwise operation – e.g., duplicating primitives for rendering to a cube map – by constructing and emitting new `RasterVertex` records. The groupwise 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 like `RS_Position`) may be independently added to the shader graph without conflict. The operations in the shader graph, in turn, remain oblivious to how many `RasterVertex` records the GS kernel might construct, or in what order it might emit them.

### 4.4 Plumbing Operators

When a node with per-coarse-vertex rate (e.g., `P_base`) is used as an input to a per-fine-vertex computation (`P_disp`), the value must be plumbed from one rate to the other. We can recognize plumbing in the shader graph in Figure 4 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.

The 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 pointwise shading code.

Plumbing might involve interpolation or more general resampling; it constitutes groupwise rather than pointwise code. Like per-stage kernels, plumbing operators are defined in terms of record types. For example, an operator for plumbing values 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 4, positions might be plumbed from coarse to fine vertices using bicubic Bézier interpolation. Texture coordinates, however, 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 intermediate shader-graph nodes not depicted in Figure 4.

# 5 Key Design Decisions

## 5.1 A language with declarative and procedural layers

We wanted programmers to 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. As discussed, though, 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, flow control 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 2.4.

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 [Austin and Reiners 2005]. 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.

In particular, the procedural layer is also used to define groupwise shading operations. The kernel that drives the GS, for example, is modeled as a procedural subroutine that the user must define. The user also defines new plumbing operators as subroutines.

A purely functional language could 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 [Wadler 1990]

## 5.2 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 has been fortuitous, bringing 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, nodes in the shader graph. This may 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 [Odersky et al. 2004]. Notable capabilities include:

- A shader class can *extend* another shader class, inheriting its declarations (graph nodes, 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 (à la Scala's *traits*). Mixins support a robust form of multiple inheritance.
- 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 graphics 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 graphics pipeline provides and requires. All of the configuration for a rendering pass – both programmable and fixed-function – is encapsulated in a shader class. 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.

## 5.3 Model rates of computation in libraries, not the compiler

Existing shading languages with rates of computation represent them with 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."

The implementation of a rate of computation (e.g., translation of computations to GPU code) might involve dedicated support in the compiler. While a Spark shader programmer can *define* their own rate qualifiers (e.g., a hypothetical `@Light` rate), they may not be able to *implement* the desired semantics directly in Spark code.

As discussed in Section 4.3 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`.

Simply allowing for an extensible set of rate-qualifier *names* is not sufficient, however. 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 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 );
```

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 to convert from `@RasterVertex` to `@Fragment` values. Because plumbing operators are functions, the library can expose different methods of interpolation, e.g.:

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

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 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 value and converts it to a per-fragment value. More fundamentally, this function does not operate on 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.

```
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;
}
```

**Figure 5:** *Example Spark plumbing operator. The per-control-point attribute* `attr` *is* projected *out of three particular control points and then interpolated.*

When defining a module for a groupwise shading operation – e.g., a tessellation scheme – a programmer also defines 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.

### 5.4 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 declaration 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 states that every fine vertex knows its barycentric location in the tessellation domain.

Figure 5 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 [Codd 1970], rather than geometry.

Projection relies fundamentally on the interface between shader graphs and record types defined in Section 4.3. In particular, if `attr` represents a per-control-point attribute (that is, a shader-graph node), then it also represents a field in the `ControlPoint` type. If, as in Section 5.3, 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 the scoping rules for C's "dot operator." In particular, `attr` in Figure 5 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 decided to use distinct syntax to reflect the distinct semantics.

### 5.5 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 error messages 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 came to the conclusion 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.

### 5.6 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 [2009], 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.

## 6 Example Program

In order to impart the flavor of the Spark language, we present a brief code example in Figure 6. 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 4.

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 values from `vertexStream`, use ordinary function-call syntax.

The classes in Figure 6 define a number of shader-graph nodes. The **`input`** `@Uniform` nodes represent shader parameters, while the **`output`** `@Pixel` node represents a shader output that should be captured in a render target. A shader class can **`override`** the definition of an **`abstract`** or **`virtual`** node, whether user- or system-defined (e.g., `P_model` and `RS_Position` respectively).

```
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);
}

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 {}
```

**Figure 6:** *Example Spark shader classes. The* `Example` *class corresponding approximately to the shader graph in Figure 4.*

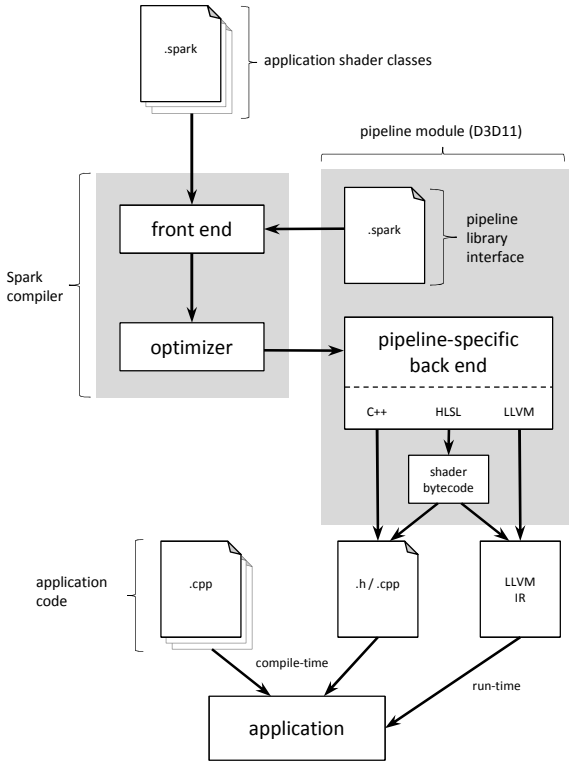


**Figure 7:** *System block diagram. Application shaders are type-checked against a pipeline-specific standard library. Global optimizations are applied to each shader class. Executable CPU and GPU code are then generated by a pipeline-specific back end.*

# 7 System Experience

## 7.1 Implementation

We have implemented a compiler and runtime for the Spark language in a combination of C# and C++ code. Figure 7 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 the rendering pipeline as one or more Spark shader classes. These shader classes declare the types, operations, and rates of computations for the pipeline. A user-defined shader, then, is type-checked against the interface declared in a particular pipeline module.

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 over the shader graph. In order to make optimization more effective, we "flatten" the inheritance hierarchy at this step. 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.

Once a shader class has been optimized, it is passed to a back end in the pipeline module for code generation. We have currently implemented a pipeline module for the D3D11 rendering pipeline. The back end in this module generates a combination of CPU code (as either C++ source or LLVM IR [Lattner and Adve 2004]) and shader bytecode (via translation to HLSL). The GPU shader bytecode is embedded in the CPU code.

Application code 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 5.2. The compiled implementations of shader classes can either be linked into the application as C++ code, or loaded at run-time as LLVM IR.

## 7.2 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. At present, the Spark compiler assigns all `@Uniform` shader parameters to a single D3D constant buffer. In

| Component | Description |
|---|---|
| `SkeletalAnimation` | Transformation by "bone" matrices |
| `CubicGregoryQuads` | Tessellation of approximate subdivision surfaces [Loop et al. 2009] |
| `RenderToCubeMap` | Uses GS instancing |
| `PhongMaterial` | Phong illumination [Phong 1973] |
| `EnvMapMaterial` | Sample a reflection cube map |
| `NormalMaterial` | Visualize normals |

**Table 1:** *Shader components used in Figure 8.*

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

**Table 2:** *Models used in Figure 8, and the shader components they use. For a given rendering pass, these may additionally be composed with light-source shaders or* `RenderToCubeMap`*.*

addition, we do not support the Stream Out (SO) pipeline stage, nor the use of Unordered Access Views (UAVs). The use of UAVs to express atomic read-modify-write operations (side effects) presents an interesting question: is it possible to encapsulate typical uses of UAVs into reusable and composable modules?

### 7.3 Results

In order to explore the value of Spark in modularizing shading effects, we developed a suite of shader classes that implement a variety of effects. Our goal was to demonstrate that Spark allows effects to be written as localized, reusable modules. We then compose different subsets of those modules to render models under different conditions, without writing additional code.

Table 1 summarizes the most important shader components in our suite. Among the shader components, `CubicGregoryQuads` and `RenderToCubeMap` implement groupwise operations; all the others are pointwise. Table 2 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 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. 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.

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.

Table 4 shows performance results for the rendering passes in Figure 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 7.1, the Spark system automatically performs global dead-code elimination as part of compilation.

Table 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 3:** *Comparison of lines of code (non-whitespace, non-comment) in Spark and HLSL implementations of the shader suite.*

Table 3 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.

## 8 Discussion

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.

While procedural shading languages achieve good performance by directly exposing the topology of the rendering pipeline, our results indicate that performance can still be achieved with higher levels of abstraction, by performing global optimizations.

Our design is not, however, without limitations. Because global optimizations are required to achieve good performance, composition in Spark is currently modeled as a *static* operation, in terms of class inheritance. 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.

**Figure 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.*

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

**Table 4:** *Performance results comparing Spark and HLSL. We measure GPU execution time of each draw pass for the view shown in Figure 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.*

A more flexible approach would be to express composition as a dynamic operation, by aggregating shader *instances* at run-time. Existing shader-per-stage languages allow dynamic composition of separately-compiled per-stage shaders. Limited dynamic composition is possible *within* shader stages using the HLSL Shader Model 5 `interface` feature. Allowing for dynamic composition of Spark-style components that cross-cut the pipeline structure, while maintaining good performance, might require support from the underlying rendering system. We are interested in the possibility of co-design of a rendering architecture and high-level shading language to allow for more flexible on-the-fly shader composition, while still maintaining high rendering performance.

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.

## Acknowledgements

We thank the members of the Stanford Graphics Lab and Intel's Advanced Rendering Technology group who supported this project. In particular, Bill Mark (Intel) and Solomon Boulos (Stanford) provided extensive feedback on the Spark system design and this paper.

The Vortigaunt model was provided to us by Jason Mitchell of Valve Corp. The Uffizi Gallery light prove image was provided by Paul Debevec. The Lizard and Big Guy models were obtained from the Microsoft DirectX SDK.

This work was performed as part of the Stanford Pervasive Parallelism Laboratory affiliates program supported by NVIDIA, Oracle, AMD, and NEC. The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

## References

ADOBE, 2011. Pixel Bender 3D. `http://labs.adobe.com/technologies/pixelbender3d/`.

AUSTIN, C., AND REINERS, D. 2005. Renaissance: A functional shading language. In *Proceedings of Graphics Hardware 2005*, ACM, New York, NY, USA, 1–8.

BLYTHE, D. 2006. The Direct3D 10 system. *Transactions on Graphics 25*, 3, 724–734.

CODD, E. F. 1970. A relational model of data for large shared data banks. *Commun. ACM 13* (June), 377–387.

COOK, R. L. 1984. Shade trees. In *Proceedings of SIGGRAPH 1984*, ACM, New York, NY, USA, vol. 18, 223–231.

HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. In *Proceedings of SIGGRAPH 1990*, ACM, New York, NY, USA, 289–298.

KESSINICH, J., BALDWIN, D., AND ROST, R., 2003. The OpenGL® shading language, version 1.05. `http://www.opengl.org`, February.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of ECOOP 1997*, Springer-Verlag.

KUCK, R., AND WESCHE, G. 2009. A framework for object-oriented shader design. In *Proceedings of ISVC 2009: International Symposium on Advances in Visual Computing*, Springer-Verlag, Berlin, Heidelberg, 1019–1030.

LALONDE, P., AND SCHENK, E. 2002. Shader-driven compilation of rendering assets. *Transactions on Graphics 21*, 3, 713–720.

LATTNER, C., AND ADVE, V. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of CGO 2004: International Symposium on Code Generation and Optimization*.

LEJDFORS, C., AND OHLSSON, L. 2004. PyFX – an active effect framework. In *Proceedings of SIGRAD 2004*, Linköping University Electronic Press, Gävle, Sweden, 17–24.

LOOP, C., SCHAEFER, S., NI, T., AND CASTAÑO, I. 2009. Approximating subdivision surfaces with Gregory patches for hardware tessellation. *Transactions on Graphics 28*, 151:1–151:9.

MARK, W. R., GLANVILLE, R. S., AKELEY, K., AND KILGARD, M. J. 2003. Cg: A system for programming graphics hardware in a C-like language. *Transactions on Graphics 22*, 896–907.

MCCOOL, M. D., QIN, Z., AND POPA, T. S. 2002. Shader metaprogramming. In *Proceedings of Graphics Hardware 2002*, Eurographics, Aire-la-Ville, Switzerland, Switzerland, 57–68.

MCCOOL, M. D. 2000. SMASH: A next-generation API for programmable graphics accelerators. Tech. Rep. CS-2000-14, University of Waterloo, August.

MICROSOFT, 2002. Shader model 1 (DirectX HLSL). `http://msdn.microsoft.com`.

MICROSOFT, 2010. Direct3D 11 reference. `http://msdn.microsoft.com`.

MICROSOFT, 2010. Effect format (Direct3D 11). `http://msdn.microsoft.com`.

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

ODERSKY, M., ALTHERR, P., CREMET, V., EMIR, B., MANETH, S., MICHELOUD, S., MIHAYLOV, N., SCHINZ, M., STENMAN, E., AND ZENGER, M. 2004. An overview of the Scala programming language. Tech. Rep. IC/2004/64, EPFL Lausanne, Switzerland.

PERLIN, K. 1985. An image synthesizer. In *Proceedings of SIGGRAPH 1985*, ACM, New York, NY, USA, vol. 19, 287–296.

PHONG, B. T. 1973. *Illumination for Computer-Generated Images*. PhD thesis.

PROUDFOOT, K., MARK, W. R., TZVETKOV, S., AND HANRAHAN, P. 2001. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of SIGGRAPH 2001*, ACM, New York, NY, USA, 159–170.

SEGAL, M., AKELEY, K., FRAZIER, C., LEECH, J., AND BROWN, P., 2010. The OpenGL® graphics system: A specification (version 4.0 (core profile) - march 11, 2010). `http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf`.

SUGERMAN, J., FATAHALIAN, K., BOULOS, S., AKELEY, K., AND HANRAHAN, P. 2009. GRAMPS: A programming model for graphics pipelines. *Transactions on Graphics 28*, 1, 1–11.

WADLER, P. 1990. Comprehending monads. In *Proceeding of LFP 1990: ACM Conference on LISP and Functional Programming*, ACM, New York, NY, USA, 61–78.