A System for Rapid Exploration of Shader Optimization Choices

Yong He
Carnegie Mellon University

Theresa Foley
NVIDIA

Kayvon Fatahalian
Carnegie Mellon University

Abstract

We present Spire, a shading language and compiler framework that facilitates rapid exploration of shader optimization choices (such as frequency reduction and algorithmic approximation) afforded by modern real-time graphics engines. Our design combines ideas from rate-based shader programming with new language features that expand the scope of shader execution beyond traditional GPU hardware pipelines, and enable a diverse set of shader optimizations to be described by a single mechanism: overloading shader terms at various spatio-temporal computation rates provided by the pipeline. In contrast to prior work, neither the shading language's design, nor our compiler framework's implementation, is specific to the capabilities of any one rendering pipeline, thus Spire establishes architectural separation between the shading system and the implementation of modern rendering engines (allowing different rendering pipelines to utilize its services). We demonstrate use of Spire to author complex shaders that are portable across different rendering pipelines and to rapidly explore shader optimization decisions that span multiple compute and graphics passes and even offline asset preprocessing. We further demonstrate the utility of Spire by developing a shader level-of-detail library and shader auto-tuning system on top of its abstractions, and demonstrate rapid, automatic re-optimization of shaders for different target hardware platforms.

Keywords: shading languages, real-time rendering

Concepts: •**Computing methodologies** $\rightarrow$ *Graphics systems and interfaces;*

1 Introduction

Developers of modern real-time rendering engines are faced with many choices when determining the techniques that yield the best performance-quality trade-offs for a given virtual world, target hardware platform, or specific set of scene viewing conditions. Many of these choices pertain to efficient implementation of shading calculations. For example, the process of optimizing a modern shading effect involves more than choosing a suitable material model and writing good GPU shader code. In the context of the entire rendering engine, optimization includes decisions such as identifying when components of a shader can be "baked" into textures or parameter buffers during offline preprocessing, selecting the spatial frequency (e.g., per-vertex or per-fragment) or coordinate space (e.g., uniformly on screen or in object-space) in which to evaluate the shading function, and deciding when to leverage multi-resolution shading techniques or forms of temporal reuse. Decisions can also span multiple objects: e.g., to constrain objects to share GPU state, so they can be "batched" for efficient rendering.

SIGGRAPH '16 Technical Paper, July 24-28, 2016, Anaheim, CA,
ISBN: 978-1-4503-4279-7/16/07
DOI: http://dx.doi.org/10.1145/2897824.2925923

Many of these choices have global impact on the design and implementation of a rendering pipeline used by an engine. Investigating a single optimization choice may require rewriting GPU kernel programs, host CPU application logic and data structures, and even code executed offline to prepare assets for the engine; the programming effort is not only substantial, but is distributed across multiple programming systems. This challenge is particularly acute for modern AAA games, which can feature many unique shaders, implementing a wide collection materials and multiple levels of material detail. (For example, Bungie's *Destiny* utilizes over 17,000 artist-authored materials compiled to over 180,000 unique vertex and fragment programs [Bungie 2014].) In a well-tuned system, many of these shaders may require different optimization choices. Further, retargeting the application to a wide range of hardware platforms (e.g., PC, console generation(s), mobile) requires new optimization decisions for each target.

In this paper, we describe Spire a shading language and compiler framework that supports rapid exploration of a wide space of shader optimization choices afforded by modern real-time graphics engines. Our design combines concepts from prior work in rate-based shader programming [Foley and Hanrahan 2011] with new language features that:

- Expand the scope of rate-based shaders beyond GPU hardware pipelines, to encompass further spatio-temporal rates for computation and storage. In particular, we support engine-specific pipelines that span multiple compute and graphics passes, and even offline asset processing.
- Allow shader terms to be overloaded: take on multiple definitions that permit varying algorithmic approximations or different rates of evaluation. This design allows shader optimization to be expressed as the choice of which term definition to use in a compiled shader.
- Decouple optimization decisions from shader code, and allow engine-specific optimization policies to be built as libraries and applied to many shaders.

In contrast to prior work, neither the shading language's design, nor our compiler's implementation, is specific to any one rendering pipeline, and thus our system establishes interfaces against which rendering engines may integrate to utilize its services.

We demonstrate use of this language and compiler framework to interactively explore algorithmic choice, frequency reduction, and texture data-layout optimizations for complex shaders, apply a consistent optimization policy to multiple shaders to create families of level-of-detail (LOD) shaders, and rapidly re-target shaders to alternative multi-rate rendering pipelines. We also demonstrate that Spire's representations facilitate the development of powerful optimization tools, such as a shader auto-tuning system that automatically optimizes shaders for different target hardware platforms.

2 Related Work

Many popular game engines provide abstractions and tools that assist shader authoring. For example, the Unreal Engine [Epic Games 2015a] provides a visual node-graph editing tool for defining expressions that compute input parameters for engine-defined material models. Each node in the expression graph corresponds to a pre-defined snippet of GPU shader (or engine) code, which is com-posed with engine-provided material model code during compilation. Such systems are specific to a particular rendering engine, do little to assist the process of shader optimization, and do not provide advanced language features for building libraries of shaders. Spire offers new shader representations that overcome these limitations and facilitate development of powerful shader optimization tools such as auto-tuners and shader level-of-detail libraries.

Systems such as GRAMPS [Sugerman et al. 2009] and Piko [Patney et al. 2015] allow developers to define custom graphics pipeline topologies using general stream programming primitives and compile them to different targets. These systems enable exploration of the design space of rendering pipelines, but do not generate output that is performance competitive with application specific solutions hand-authored by engine developers using conventional programming tools. Spire does not address the task of *implementing* efficient rendering pipelines. Instead, Spire focuses on the problem of efficiently *targeting* engine-specific pipelines with shaders.

Like EAGL [Lalonde and Schenk 2002], Spire shaders describe both offline and runtime computations and enable rendering assets to be specialized for shaders and stored for later runtime use. Whereas EAGL supplied a fixed set of offline preprocessing routines to specialize geometry buffers for shader needs, Spire enables shader-defined computations to be executed as part of a rendering pipeline's preprocessing phase. EAGL also leveraged knowledge of shaders to minimize state-change overhead in rendering pipeline draw loops. While Spire provides engine implementations with sufficient information to make such optimizations (Section 6.2), by design Spire does not assume responsibility for synthesizing pipeline implementations itself.

In contrast to Cg [Mark et al. 2003], GLSL [Kessenich et al. 2014], and HLSL [Microsoft 2016], which adopt a one-shader-per-pipeline-stage model of shader programming, Spire is based on the idea of rate-based programming, where a single shader defines computations that occur at multiple spatio-temporal *rates* (corresponding to pipeline stages). Mixed-rate shaders (also called *pipeline shaders*) are supported by RSL [Hanrahan and Lawson 1990] (via the `uniform` and `varying` rate qualifiers) and used in RTSL [Proudfoot et al. 2001] to map computation to multiple programmable pipeline stages (e.g., `primitive`, `vertex`, `fragment` qualifiers). Other mixed-rate shading systems do not require rate qualifiers, and rely on compiler policies and optimizations to select rates [McCool 2000; Austin and Reiners 2005]. Spire's rate-based dataflow programming model is most similar to that of Spark [Foley and Hanrahan 2011]. However, Spire and Spark differ greatly in implementation (Sections 4 and 5), and the two systems were motivated by different goals (Spire by shader optimization, and Spark from code modularity and shader composition).

Spire's component overloading features (Section 4.3) serve to separate the definition of a shader program from the set of optimization choices afforded by the shader and rendering pipeline. This design echoes the separation of algorithm from schedule enforced by programming languages for high-performance computing [Fatahalian et al. 2006; Bauer et al. 2012] and image processing [Ragan-Kelley et al. 2012]. However, since making shader optimization choices not only impacts shader performance, but also changes output quality and the algorithms used to evaluate shading, the line between the shader algorithm and its optimization decisions in Spire is less distinct. As in PetaBricks [Ansel et al. 2009], optimizing a Spire shader may involve choice of algorithm.

3 Design Goals

The design of Spire is motivated by the following goals.

Address a wide space of shader optimization options. Important shader optimization choices (spatio-temporal frequency reduction, algorithm simplification/approximation, buffer storage layout, coordinate space selection) affect code distributed throughout the engine (including running in GPU compute, or during offline asset processing.) To span a useful set of optimization choices, a system must encompass shader-related logic spanning the entire pipeline from art assets to pixels, not only the GPU rasterization pipeline. Our focus is on enabling high-level shader optimization choices that impact this end-to-end sequence of computations; exploring the space of low-level intra-kernel optimizations (register allocation, instruction orderings, etc.) is a non-goal of our system.

Facilitate rapid exploration of shader optimization choices. We seek to enable both manual (by a programmer) and automatic (by auto-tuning tools) exploration of the performance-quality space for a shader (e.g., to find the best trade-off for a given platform). For example, a programmer should be able to express a high-level optimization choice, then have the system generate a full shader implementation for the rendering pipeline, even if the choice induces significant change in overall rendering pipeline logic. To further support rapid choice exploration, the system should also support mechanical application of choices (once they are made) to large numbers of shaders (e.g., to optimize thousands of shaders at once).

Maintain rendering pipeline independence. Expanding the notion of "rendering pipeline" to concerns beyond a single GPU draw call, presents the challenge of targeting a multitude of rendering pipelines tailored to the needs of particular applications. Rather than seek a one-size-fits-all shading solution (i.e., a shading language designed for a *single, universal* pipeline), the system must allow each engine to expose a different, specialized interface to shader programmers. The compiler must then be able to validate and compile shader code without full knowledge of how the engine will use it. Further, a framework that is independent of any specific pipeline implementation has the added benefit of allowing multiple engines to share the same language and compiler infrastructure, as well as offers the possibility of reusing shaders and tools (e.g., for auto-tuning) across engines.

Achieve high performance. The convenience of rapid exploration cannot come at the cost of significant overhead in generated code; the system must produce code that is competitive with expert developers making the same optimization choices by hand. Further, the system should only generate code for shaders, and not attempt to synthesize other parts of the engine; those tasks are left to highly specialized solutions authored by engine programmers.

4 Spire Concepts

Figure 1 provides an overview of the process of compiling and optimizing a Spire shader. In this section we describe the key concepts in this process, in which responsibilities are divided among three parties: the engine developer, the shader author, and the Spire compiler framework. As Spire adopts many mixed-rate shading language abstractions featured in Spark [Foley and Hanrahan 2011], we will relate concepts back to their Spark equivalents when possible, and make note of important differences in our model.

4.1 Defining Pipeline Topology

An important aspect of Spire's design is that the shading language and compiler do not dictate or assume any specific rendering pipeline structure. Instead, the engine developer is responsible for providing a definition of an engine's rendering pipeline(s) to the compiler front-end. The front-end is then responsible for validating authored shaders against the capabilities of the corresponding pipeline.

Figure 2 depicts an engine-specific rendering *pipeline* (named `EnginePipeline`) as a graph. Each node in the graph (e.g.,

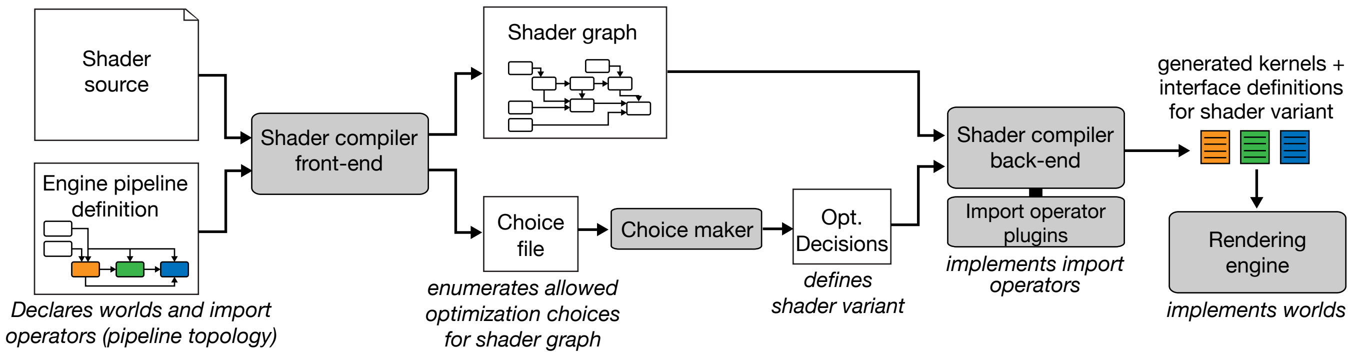

Shader source
Engine pipeline definition
Declares worlds and import operators (pipeline topology)
Shader compiler front-end
Shader graph
Choice file
enumerates allowed optimization choices for shader graph
Choice maker
Opt. Decisions
defines shader variant
Shader compiler back-end
Import operator plugins
implements import operators
generated kernels + interface definitions for shader variant
Rendering engine
implements worlds

Figure 1: *Overview of Spire shader compilation. The compiler front-end validates a shader against an engine-provided pipeline definition. The shader's space of component definition choices is exposed to a choice maker, which produces decisions that inform back-end code generation. The engine executes the compiled kernels to provide the semantics of the pipeline definition.*

`Fragment`) corresponds to a *world*: a logical place and time at which values can be computed and/or stored by the pipeline. For example, in `EnginePipeline`, the `Fragment` world represents conventional GPU fragment shading, while `PrebakeTex` represents precomputations whose results are stored into texels during offline asset processing. Worlds correspond to stages in the logical rendering pipeline of an engine, and to the notion of *rates* of computation in RTSL and Spark. Our use of a new term reflects the expanded scope of worlds in Spire; they express not just the rates of sampling or execution in a GPU hardware pipeline, but a myriad of places and times (spanning multiple frames, or even machines) at which an engine might compute or store shading results.

Abstract worlds are used to declare shader inputs provided from outside the language. Shaders cannot define computations at abstract worlds. `EnginePipeline` features three abstract worlds, shown as gray nodes in Figure 2. The `MeshVertex` world represents per-vertex attributes associated with geometry fed into the (asset processing and rendering) pipeline. The `MaterialUniform` world provides non-time-varying inputs (e.g., material parameters), while `FrameUniform` represents per-frame uniform shader parameters provided by the engine at runtime.

The directed edges that connect worlds in the pipeline graph correspond to *import operators*, which define how values are allowed to flow between worlds. For example, values computed in the `Vertex` world can be imported into `Fragment` computation (but not vice versa). Time-varying inputs from the `FrameUniform` world are not available to offline `PrebakeTex` computations. Import operators express a dependency relationship between worlds, such that some worlds are transitively *reachable* from others.

The pipeline topology only declares what worlds and dataflow paths a pipeline supports. The engine is responsible for defining and implementing the semantics of the worlds and import operators of its pipeline(s). For example, `EnginePipeline` could be implemented via two rendering passes. The first pass is an offline preprocess that renders a mesh, executing `PrebakeTex` computations in the GPU's fragment shader (with inputs from the `MeshVertex` and `MaterialUniform` worlds supplied via vertex attributes and uniform buffers respectively) and storing outputs in texture space of the mesh (as texels in texture buffers). The second pass is executed per-frame at runtime, mapping computations in the `Vertex` and `Fragment` worlds to the vertex- and fragment-processing stages of the GPU pipeline. In this pass `FrameUniform` data is now available via uniform buffers, and the outputs of `PrebakeTex` are available as textures bound to the GPU pipeline.

4.2 Defining Shaders

In Spire a *shader* is a function defined over an engine-defined domain. In the case of `EnginePipeline`, the domain is the surface of a mesh, and a shader computes the values of one or more signals on that surface (e.g., surface appearance) [Wang et al. 2014]. A shader author cannot express computations that depend on multiple points in the domain; in the terminology of Spark, a shader comprises only *point-wise* code. Operations which span multiple domain points (e.g., interpolation or resampling) are implemented by the rendering engine and typically exposed to the shader as import operators (e.g., interpolation of data imported from `Vertex` to `Fragment`).

From the point of view of the compiler, a shader is represented as a *shader graph*: a DAG of computation, defined against a particular pipeline. The nodes of a shader graph are *components*, corresponding to inputs, outputs, or intermediate computations of the shader. Spire components are similar to shader attributes in Spark.

Listing 1 defines a shader `Terrain` for the `EnginePipeline`, which computes the appearance of a terrain model by blending multiple layers of albedo and normal maps according to spatially varying weights encoded in a texture map. The body of this shader defines several components. Components placed in the `MaterialUniform` and `FrameUniform` worlds constitute shader input parameters that must be provided from outside of the shading language, on a per-material or per-frame basis, respectively. The remaining components in `Terrain` are given *definitions*. The definition of `mixFactor` is a simple expression, so it is written as an initializer. The computation of `albedo` is more involved, and is therefore written as a block-structured body statement which samples two texture maps and blends between the results. Similar to the restrictions in the Spark language, we allow use of control flow and mutable variables inside the definition of a single component, but not at the top level of a shader declaration.

A pipeline may also declare components, which are inherited by all shaders using the pipeline; these represent values that the pipeline either provides or requires. In Listing 1, the `vert_uv` component is provided by `EnginePipeline`, while the definitions of `projPos` and `output` are required by the pipeline. We discuss the definition of `EnginePipeline` later, in Section 5.1.

4.3 Component Overloading

Through the mechanism of *component overloading*, Spire shaders can be authored to describe a rich space of evaluation choices. The component `normal` in Listing 1 is explicitly overloaded: it has different definitions for the `Vertex` and `Fragment` worlds. The first definition simply uses the per-vertex normal attribute of the

```
shader Terrain using EnginePipeline {
  @MaterialUniform sampler2D mixMap;
  @MaterialUniform sampler2D albedoMap1, albedoMap2;
  @MaterialUniform sampler2D normalMap1, normalMap2;
  @FrameUniform vec3 lightDir;
  @FrameUniform mat4 matMVP;

  float mixFactor = texture(mixMap, vert_uv).x;
  vec4 albedo {
    vec4 c1 = texture(albedoMap1,vert_uv*5);
    vec4 c2 = texture(albedoMap2,vert_uv*5);
    return mix(c1, c2, mixFactor);
  }
  @Vertex vec3 normal = vert_normal;
  @Fragment vec3 normal {
    vec4 n1 = texture(normalMap1,vert_uv*5);
    vec4 n2 = texture(normalMap2,vert_uv*5);
    vec3 n = mix(n1, n2, mixFactor).xyz;
    return n * 2.0 - 1.0;
  }
  float lighting:basic = phong(lightDir, normal, ...);
  float lighting:ggx   = ggx(lightDir, normal, ...);
  vec4 projPos = matMVP * vec4(vert_pos,1);
  vec4 output = albedo * lighting;
}
```

Listing 1: *A shader written against* `EnginePipeline` *from Figure 2 and Listing 3). The shader composes two layers of albedo and normal texture maps (normal map compositing is optional), then computes surface reflectance using one of two techniques.*

model being rendered, while the per-fragment definition samples and mixes two layers of normal maps.

Figure 3 (top) depicts the shader graph for `Terrain`. Graph nodes correspond to components, while the colored boxes within some nodes correspond to overloaded definitions. For example, the node for `normal` contains two definitions, each color-coded according to the `Vertex` and `Fragment` worlds for which it is defined (colors are consistent with those in Figure 2). The different definitions of a component may have different dependencies; we depict this using dashed lines. For example, the `Fragment` definition of `normal` depends on `vert_uv`, `mixFactor`, and the normal maps, whereas the `Vertex` definition only depends on `vert_normal`.

Although `albedo` has only a single declaration in Listing 1, the shader graph includes three definitions (one for each world). This is a key feature of Spire's design: a component declared without an explicit world is overloaded with a definition in every possible world allowed by its dependencies. (Observe `projPos` has no explicit world qualifier but is only defined in the `Vertex` and `Fragment` worlds, due to its dependence on the per-frame matrix `matMVP`.) Previous multi-rate languages either disallow component declarations without an explicit world qualifier, or automatically infer a single world during compilation. Implicit overloading defers these decisions until later, so that even simple shaders can expose many world placement optimization choices via overload selection.

While the overloading of `normal` provided unique definitions at different worlds, `lighting` is explicitly overloaded with multiple definitions for the *same* world. `Terrain` provides two (named) definitions of `lighting`, differentiated by *variation name*: one (`basic`) evaluates simple Phong shading, and the other (`ggx`) evaluates a more complex GGX specular lighting model on the surface. Thus intra-world overloading provides a mechanism for describing potential algorithmic approximations in the shader's definition. In this example, since the two definitions of `lighting` are not explicitly associated with specific worlds, they are implicitly defined at

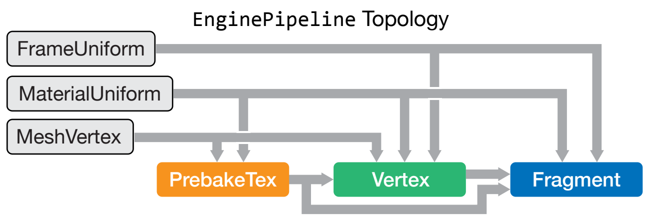


Figure 2: *An example engine-defined pipeline topology, consisting of* worlds *and their dataflow connections (*import operators*).*

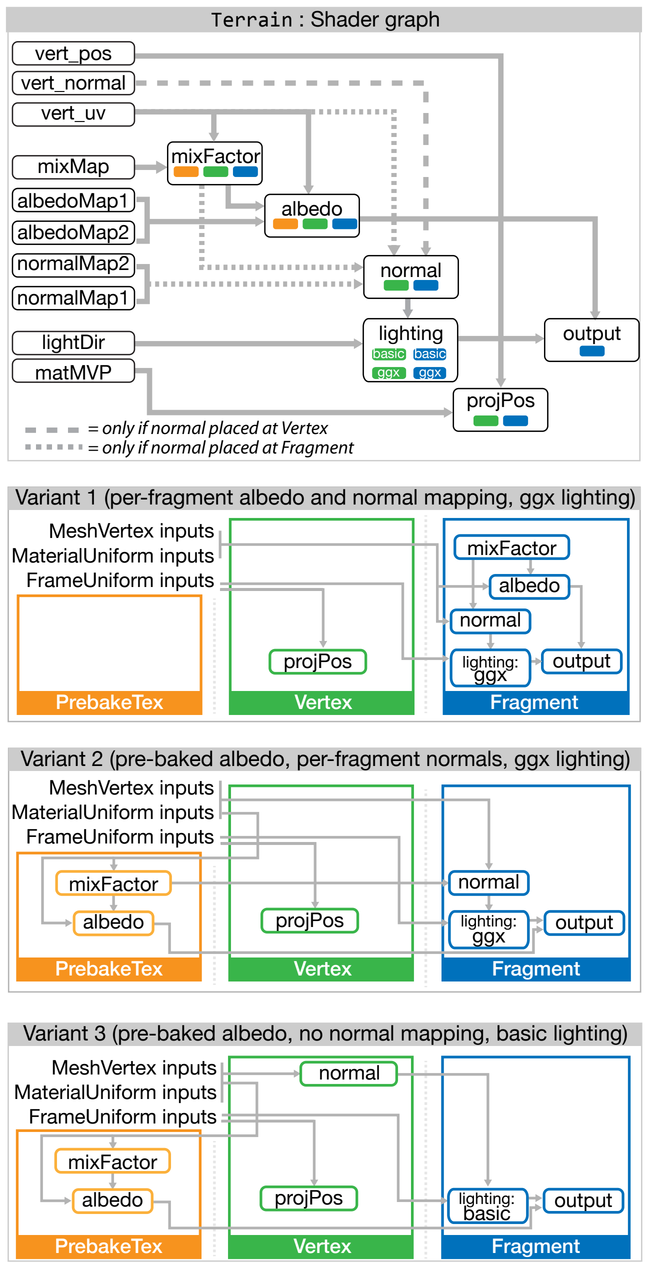


Figure 3: *Shader graph and three graph variants that result from making different optimization decisions for* `Terrain` *in Listing 1.*

```
Terrain.mixFactor = {PrebakeTex,Vertex,Fragment}
Terrain.albedo    = {PrebakeTex,Vertex,Fragment}
Terrain.normal    = {Vertex,Fragment}
Terrain.lighting  = {Vertex:basic,Fragment:basic,
                     Vertex:ggx,  Fragment: ggx}
Terrain.projPos   = {Vertex,Fragment}
-------------------------------------------------
Terrain.mixFactor = PrebakeTex
Terrain.albedo    = PrebakeTex
Terrain.normal    = Vertex
Terrain.lighting  = Fragment:basic
Terrain.projPos   = Vertex
```

Listing 2: *Top: choice file for the shader* `Terrain` *(Listing 1). Each line specifies the set of available definitions for a component. (The definition specifies the world in which the component is placed and how to compute its value.) Bottom: optimization decision file corresponding to the last variant of* `Terrain` *shader in Figure 3.*

all permitted worlds. (There are four definitions of `lighting` in `Terrain`'s shader graph.)

4.4 Making Optimization Choices

Since components may be overloaded with multiple definitions, a shader graph represents a space of possible choices for how to evaluate a shader (it represents a set of valid DAGs). The compiler front-end exposes the space of optimization choices afforded by a shader as a *choice file*, which enumerates overloaded components and their possible definitions. Listing 2-top shows a choice file for the `Terrain` shader.

In Spire, making shader optimization decisions is achieved by *placing* components in worlds (deciding when and where particular shader components should be computed or stored) and choosing which definition of the component to evaluate in that world. For example, `Terrain` presents the choice of blending `albedo` layers per fragment, per vertex, or baking the results of blending into a texture as a preprocess. Placing the `albedo` computation amounts to choosing one of its overloaded definitions.

The ability to explicitly overload component definitions (such as `normal` and `lighting` in the `Terrain` shader) means that choosing a component definition serves to both select a rate of computation and also algorithmic approximation as well (two forms of performance-quality trade-offs). For example, the user may provide an explicit overload for `albedo` in `PrebakeTex` world to offset the contrast bias when sampling at lower spatial rate.

In prior work, world placement choices were made either by the shader author (via explicit rate qualifiers) or the compiler (inferred rates). Spire models both world placement and algorithmic choices as overloaded definition selection, and the responsibility for making overload selection choices is given to a *choice maker* external to the compiler, as shown in Figure 1. The choice maker can be an automated tool, a programmer, or an interactive UI for engine developers or shader authors; we demonstrate implementations of each of these options in Section 7.

The output of the choice maker is a set of *optimization decisions*, specifying which definitions of the components should be used. Listing 2-bottom shows an example set of optimization decisions for the `Terrain` shader. When applied to the shader graph, the optimization decisions define a *shader variant*: a graph with only a single definition for each overloaded component. Figure 3 shows three variants of the `Terrain` shader. The first variant computes both `albedo` and `normal` per-fragment and the GGX lighting model (achieving high-quality output, but at high cost). The second composites the albedo texture layers offline in the `PrebakeTex` world, reducing run-time per-fragment evaluation of `albedo` to a single texture fetch. The final variant, corresponding to the decisions in Listing 2, further simplifies the computation to use per-vertex normals and employs a simple lighting model.

Each shader variant in Figure 3 corresponds to the notion of a "shader" in RTSL or Spark. These prior systems do not allow component definitions to be overloaded (whether explicitly or implicitly); thus no placement choices are left to make after compilation.

4.5 Generating Kernels

Given a shader variant (a shader graph plus optimization choices) defined against a particular pipeline, the engine requires an executable *kernel* which it can run to perform all the shader's computations for a given world. For example, if an engine implementing `EnginePipeline` needed to render a mesh using one of the `Terrain` shader variants in Figure 3, it might request kernels for the `Vertex` and `Fragment` worlds to execute in the vertex and fragment shading stages of a GPU hardware rasterization pipeline.

In Figure 3 (bottom), the blue box for the `Fragment` world encapsulates the logic of the GPU fragment-stage kernel. The nodes (component definitions) inside the box will become instructions in the kernel; dependency edges entering or leaving the box correspond to kernel inputs and outputs, respectively. These inputs and outputs correspond to (implicit) uses of import operators.

The compiler back-end is responsible for generating kernel code for a shader variant. Our code generation approach is similar to RTSL and Spark, with additional features to support engine-defined pipelines, which we will discuss in Section 5.1.

Given the generated kernels and information about their inputs and outputs, the engine orchestrates execution of a shader by invoking the various kernels at the appropriate places and times. The engine is responsible for storing any kernel outputs that will be used in downstream worlds. For example, the engine will need to allocate textures to store the outputs of a `PrebakeTex` kernel and bind these textures so that they are available as inputs to corresponding `Fragment` kernel invocations. The format and size of these required buffers is conveyed to the engine with the kernel.

5 Compiler Implementation

5.1 Defining and Generating Code for a Pipeline

While the definition of a pipeline and the generation of executable kernel code appear at opposite ends of Figure 1, these steps are intimately linked. For example, the abstract topology of a pipeline only determines what import operators exist, but code must be generated to interface with the engine that implements them. Does a given kernel input come via a buffer, texture, or preceding stage? Each of these options requires different kernel code.

In Spire, an engine programmer exposes both the capabilities of a pipeline (used when validating shaders) and important information about its behavior (used when generating code) by writing a *pipeline definition*. Listing 3 provides the definition of `EnginePipeline` corresponding to the topology illustrated in Figure 2. Each of the worlds in the pipeline topology is introduced with a `world` declaration, and some worlds are marked `abstract`. The definition also includes `import` operators declared between pairs of worlds: e.g., `Vertex->Fragment`. Finally, the pipeline declares components that all shaders using the pipeline will inherit. The `abstract` components must be defined by every concrete shader using this pipeline. The `output` component is marked `export` to indicate that it represents an output of the entire pipeline.

Spark also allows new worlds and rate-conversion operators to be declared in the language, and the capabilities of pipelines are ex-

```
pipeline EnginePipeline {
  abstract world FrameUniform;
  abstract world MaterialUniform;
  abstract world RootVertex;
  world PrebakeTex : "GLSL" export standardExport;
  world Vertex : "GLSL" export vertexExport(projPos);
  world Fragment : "GLSL" export standardExport;

  import MaterialUniform->PrebakeTex
      using uniformImport;
  import MaterialUniform->Vertex
      using uniformImport;
  import MaterialUniform->Fragment
      using uniformImport;
  import FrameUniform->Vertex using uniformImport;
  import FrameUniform->Fragment using uniformImport;
  import MeshVertex->Vertex using standardImport;
  import MeshVertex->PrebakeTex using standardImport;
  import Vertex->Fragment using standardImport;
  import PrebakeTex->Fragment
      using textureImport(vert_uv);

  @MeshVertex vec3 vert_pos;
  @MeshVertex vec3 vert_uv;
  @MeshVertex vec3 vert_normal;

  abstract @Vertex vec4 projPos;
  abstract export @Fragment vec4 output;
}
```

Listing 3: *Pipeline definition for* `EnginePipeline`*, with topology as depicted in Figure 2*

posed with definitions not unlike `EnginePipeline`. However, Spark worlds must either have their semantics fully implemented in the shading language, or be implemented with built-in logic in the compiler. That is, worlds that are translated to GPU kernels in GLSL or HLSL are all special-cased in the Spark compiler. There is no way for a user of Spark to add a world like `PrebakeTex`.

A key contribution of Spire is that engine-specific pipelines can introduce new worlds and implement the semantics of these worlds without having to modify the compiler. A non-abstract `world` declaration includes a *language specifier*; kernel code for this world will be output in the specified language. Our implementation currently supports `GLSL` kernels for both graphics and compute.

The semantics of an import operator typically involve generating specialized code in both the producing kernel and consuming kernel, for each component on which the operator is *invoked*: i.e., each component to be propagated between worlds. This is consistent with Spark, where user-defined plumbing operators are viewed as templates to be instantiated on demand. While Spark plumbing operators only generate more Spark code, Spire import operators generate kernel code (e.g., GLSL). An engine programmer specifies how this kernel code should be generated by selecting from built in *strategies* for import and export code generation in the compiler. Table 1 lists the strategies supported by our GLSL code generator.

To help explain how strategies impact code generation, consider the example where component `albedo` in the `Terrain` shader is placed in the `PrebakeTex` world, and used by the computation of `output` in the `Fragment` world. In this case, the `PrebakeTex->Fragment` import operator is invoked on `albedo`. The compiler back-end must export the value of `albedo` from `PrebakeTex` and import it into the `Fragment` world; this requires suitable code to be generated in each world.

Strategy	GLSL Code Generation Behavior
`uniformImport`	Declare imported components in a `uniform` block
`standardImport`	Declare imported components with `in` qualifier
`textureImport(uv)`	For each component, declare a `uniform sampler2d` and sample at texture coordinate `uv`
`bufferImport`	Declare imported components in a shader storage `buffer`
`standardExport`	Declare exported components with `out` qualifier
`vertexExport(v)`	Same as `standardExport`, but also assign `v` to `gl_Position`
`bufferExport`	Declare exported components in a shader storage `buffer`

Table 1: *Code generation* strategies *for import and export of components, supported by our GLSL back-end. A pipeline definition uses these strategies to specify code generation behavior.*

The declaration of the `PrebakeTex` world specifies that the `standardExport` strategy should be used when exporting components. In this case, the compiler generates a GLSL `out` declaration:

```
out vec4 albedo;
```

The declaration of the `PrebakeTex->Fragment` import operator specifies that the `textureImport` strategy should be used when components are imported. This strategy generates code to import `albedo` by declaring a texture sampler:

```
uniform sampler2D albedo_sampler;
```

and then emitting code to fetch the value of `albedo`:

```
vec4 albedo = texture(albedo_sampler, vert_uv);
```

Additional parameters required by the code generator (e.g., the texture coordinate in this example) are specified as arguments to the import strategy (e.g., `textureImport(vert_uv)`).

The export strategy of a world may also specify component to use for stage-specific outputs; e.g., the `Vertex` world specifies that `projPos` be assigned to the built-in GLSL variable `gl_Position`.

While new worlds and import operators can easily be declared by engine developers, our system does not currently allow new output languages or import/export strategies to be added as easily: new import or export strategies need to be defined via compiler plugins. An alternative would be to define new strategies as compile-time code, using an explicit metaprogramming interface.

While the import and export strategies, running in the compiler, are responsible for generating appropriate code for kernel inputs and outputs, the engine is responsible for any allocation, buffering, or movement of data required at runtime (e.g., filling in and binding a uniform buffer). To assist the engine, the compiler generates reflection data for kernels, allowing inputs and outputs to be enumerated.

5.2 Overloading Component Definitions

As described in Section 4.4, Spire exposes performance-quality optimization as overload selection choices: that is, choices between the definitions of overloaded components.

The Spire compiler represents each component in a shader as a set of definitions, each associated with a world. A component may have zero or multiple definitions for a given world; in the latter case we say that it is *defined* in that world. A component declaration with an explicit world qualifier simply adds a definition, associated with the specified world, to the set for the named component.

Deciding on the set of worlds for an implicitly overloaded component (one without an explicit world) is more involved. Data dependencies limit the set of worlds for which a valid definition can be generated. The compiler restricts an implicitly-overloaded component to the set of non-abstract worlds where all of the components it depends on are *available*. A component is available in every world that is reachable (via zero or more import operators) from one of the worlds where it is defined. We also restrict an implicitly-overloaded definition to exclude the worlds used by explicit declarations of the same component with the same variation name.

The above rules are oblivious to whether the components that an implicitly-overloaded component depends on are themselves explicitly or implicitly overloaded. For example, in Listing 1, the implicitly-overloaded component `lighting` depends on the explicitly-overloaded `normal`, and so may be placed in either of the worlds where `normal` is available: `Vertex` or `Fragment`. Judicious use of explicit overloading, to express simplification choices for frequently-referenced components, in turn makes implicit overloading more useful, by enabling more overload selection choices for downstream components.

5.3 Enumerating Choices and Making Decisions

As described in Section 4.4, Spire allows shader optimization choices to be made by external, potentially engine-specific, tools between the execution of the compiler front- and back-ends. To support this process, the compiler front-end must be able to enumerate the space of choices in a shader, and the back-end must be able to generate a variant subject to specific optimization decisions.

Naïvely, we might just enumerate the overloaded components in a shader, then enumerate the worlds in which each component is defined. However, this naïve strategy results in potentially exposing choices that are not useful. For example, consider the shader code:

```
float x = 1.0;
@Vertex float y = x + 2.0;
```

Given Spire's rules for implicit overloading, the component `x` is defined in *every* non-abstract world, so the naïve strategy would include `Fragment` among the overload selection choices for `x`. In practice, though, trying to place `x` in the `Fragment` world would fail, because `x` must be available to the `Vertex` computation of `y`. Before enumerating overload selection choices to a choice file, the compiler first performs constraint propagation to remove such impossible alternatives.

Once the choice maker has made a set of optimization decisions, the compiler back-end tries to apply those decisions and produce a shader variant for code generation. Valid decisions must be in the space enumerated in the choice file, and should respect pipeline dependencies; when decisions are in conflict, we greedily satisfy overload selection constraints in order, skipping those that are impossible (rather than issue an error). Components for which no decision is given are automatically placed by the compiler. By default, the compiler chooses the component definition in the latest possible world, where "latest" is determined by pipeline topology. When more than one component definitions are provided for the latest world, the first one is chosen. This policy tends to favor higher visual quality, at the possible expense of performance, and is consistent with the design choice in SMASH [McCool 2000] and Renaissance [Austin and Reiners 2005].

While we describe the optimization interface in terms of choice and decision files, this is only one option. Our compiler system can also be used as a library, in which case an API is used to query the optimization space of a shader, and request particular variants.

6 Optimizing Large Numbers of Shaders

The system described so far allows users to explore and make optimization choices for individual shaders. We now discuss language extensions that support making optimization choices that affect multiple shaders. These extensions allow engine developers to express optimization policies as reusable libraries.

Spire extends the use of object-oriented inheritance for shaders in Spark by introducing two new mechanisms that allow a library of base shaders to implement optimization policy. First is the idea of shader groups, which allow a single derived shader definition to synthesize a family of related shaders. Second is the idea of locked worlds, which allow an engine to guarantee state coherence between related shaders. As a running example, we will consider the case where an engine developer wants to specify a policy for shader level-of-detail (LOD) optimization, to which all surface shaders should conform.

Listing 4 defines several base shaders for an engine with materials that support both high- and low-quality shaders. The features common to all shaders are defined in `BaseImpl`, while `BaseMaterialHQ` and `BaseMaterialLQ` inherit from `BaseImpl` and define logic appropriate to high- and low-quality shaders, respectively. The high-quality shader specifies that `albedo` should be computed per-fragment (e.g., from multiple texture layers); the low-quality shader specifies that it be baked into a texture offline.

6.1 Synthesizing Families of Related Shaders

The base shaders defined above allow engine-defined policies to be applied to many authored shaders, but do not address the issue of easily synthesizing multiple shaders from a single description, each conforming to *different* policies (e.g., generating both high and low LOD shaders from the same artist-authored description).

We address this challenge with *shader groups*. Listing 4 includes the definition of a shader group, `BaseMaterial`. A shader group collects a list of shaders into a single named entity. In this example, `BaseMaterialHQ` is a required member of the group, while `BaseMaterialLQ` is marked `optional`.

A derived shader declaration may inherit from a group rather than a single base shader; in fact, a shader writer may not even know they are using a group. Listing 4 presents an alternative definition of the `Terrain` shader from Listing 1, this time inheriting from the `BaseMaterial` group. When a `shader` declaration inherits from a group, it defines an entire shader group rather than a single shader. The derived group will contain one derived shader for each required entry in the base group; the declarations within the derived declaration must be valid in the context of *every* required base shader. For optional base shaders, the compiler attempts to compile a derived shader. On success, the shader is added to the derived group; on failure, no shader is added, and the failure is reported as a warning.

A shader group can be used to define a family of related shader permutations. `BaseMaterial` defines a very simple level-of-detail policy: every material has a high-quality shader, and may optionally have a low-quality shader with baked albedo. The single definition of `Terrain` in Listing 4 yields two shaders conforming to this policy, which we can think of as `TerrainHQ` and `TerrainLQ`.

Note that `Terrain` in Listing 4 makes no direct reference to `BaseMaterialLQ`, or the `PrebakeTex` world, but this declaration yields a shader that uses pre-baked textures. The author who writes `Terrain` might not know that they are inheriting from a shader group. While we have shown an example of using groups to implement an optimization policy for LOD, the same mechanism can be

```
shader BaseImpl using EnginePipeline {
  @FrameUniform mat4 matMVP;
  @FrameUniform vec3 lightDir;
  vec4 projPos = matMVP * vec4(vert_pos,1);
  abstract @Fragment vec3 albedo_in, normal_in;
  float nDotL = max(0,dot(lightDir, normal_in));
  vec4 output = albedo_in * nDotL;
}
shader BaseMaterialHQ : BaseImpl {
  abstract @Fragment vec3 normal, albedo;
  @Fragment vec3 normal_in = normal;
  @Fragment vec3 albedo_in = albedo;
}
shader BaseMaterialLQ : BaseImpl
    finalizes Vertex,Fragment
{
  abstract @PrebakeTex vec3 albedo;
  @Vertex vec3 normal_in = vert_normal;
  @PrebakeTex vec3 albedo_in = albedo;
}
group BaseMaterial {
  shader BaseMaterialHQ;
  optional shader BaseMaterialLQ;
}

shader Terrain : BaseMaterial {
  @MaterialUniform sampler2D mixMap;
  @MaterialUniform sampler2D albedoMap1, albedoMap2;
  @MaterialUniform sampler2D normalMap1, normalMap2;

  float mixFactor = texture(mixMap, vert_uv).x;
  vec4 albedo {
    vec4 c1 = texture(albedoMap1, vert_uv * 5);
    vec4 c2 = texture(albedoMap2, vert_uv * 5);
    return mix(c1, c2, mixFactor);
  }
  vec3 normal {
    vec4 n1 = texture(normalMap1, vert_uv * 5);
    vec4 n2 = texture(normalMap2, vert_uv * 5);
    vec3 n = mix(n1, n2, mixFactor).xyz;
    return n * 2.0 - 1.0;
  }
}
```

Listing 4: *A set of base shaders used to define an LOD policy.*

used to express other optimization spaces, such as platform: e.g., different optimizations for PC, console, and mobile targets.

6.2 Locking Worlds to Guarantee Coherence

Shader programming often involves walking a fine line between specialization and coherence. Aggressively specializing code to particular assets or viewing conditions (number of lights, etc.) can result in better performance via constant folding, loop unrolling, etc. However, this specialization often comes at the cost of coherence, since GPU pipeline state changes must be inserted between drawing objects requiring different (specialized) shaders.

For example, when drawing objects with simplified LODs, it is desirable to be able to group objects together into a small number of *batches* to reduce per-object overheads. To be in the same batch, simplified LOD shaders might need to have the same vertex and fragment kernels (since these impact GPU state), but could still be allowed different code in other worlds (e.g., it does not matter if objects execute different code when baking textures).

In Listing 4, the `BaseMaterialLQ` shader *locks* the `Vertex` and `Fragment` worlds using the `finalize` keyword. As a result, shaders which inherit from `BaseMaterialLQ` are not allowed to introduce additional component definitions in the locked worlds, nor request additional data to be imported from (or through) those worlds. Because the `Vertex` and `Fragment` worlds are locked, an engine can assume that all shaders that derive from `BaseMaterialLQ` share identical kernel code for these worlds, with an important caveat we will discuss next. The engine can thus use the same state to render a batch of objects using different shaders that inherit from `BaseMaterialLQ`, exploiting coherence among the related shaders. In contrast to, e.g., EAGL [Lalonde and Schenk 2002], our system does not perform any sorting, batching, or state-change optimizations; it merely exposes sufficient mechanisms for an engine developer to implement these optimizations.

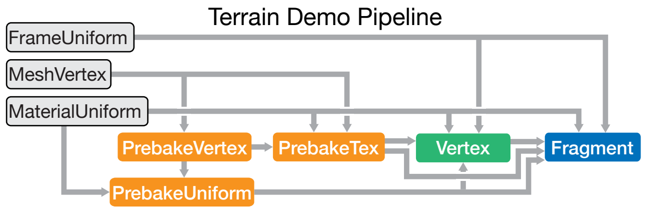


Figure 4: *Pipeline topology used for the* `Terrain` *scene, LOD system, and auto-tuning example. This pipeline extends* `EnginePipeline` *from Figure 2 with additional worlds for pre-baking vertex data and uniform parameters.*

The caveat alluded to above is that two shaders derived from `BaseMaterialLQ` will share identical vertex and fragment kernels *so long* as they make consistent optimization decisions for inherited components that may be placed in those worlds. If one shader is optimized to compute `nDotL` in the `Vertex` world while another computes it in `Fragment`, then clearly the resulting kernels will not be coherent. Our compiler is not responsible for making overload selection decisions, and so cannot guarantee the desired property. Instead, the simple mechanism of choice files can be used to enforce an appropriate policy at the engine level. Choices from inherited components are explicitly prefixed in the file (e.g., `BaseMaterialLQ.nDotL`), and a tool can either not expose these choices, or ensure that consistent decisions are made across all shaders.

Locking is most useful when defining frequently-used base shaders for an engine, since it allows an engine developer to a define policy that will be applied to many shaders authored by less technical users. Shader authors may not even be aware of the constraints, in the common case; for implicitly overloaded components in a derived shaders, the compiler will simply skip locked worlds when generating overloaded definitions.

7 System Experience

To evaluate the design of Spire, and to gain experience using its abstractions, we have implemented a real-time rendering engine that supports several distinct rendering pipelines. Our implementation adopts many best practices of modern production game engines (shader level-of-detail, precompiled command lists, sorting to minimize GPU pipeline state changes, etc.) to achieve high rendering performance. The engine partitions scene geometry spatially into blocks, and pre-generates a command list (via the `NV_command_list` OpenGL extension) for all geometry in a block sharing the same GPU pipeline vertex and fragment kernels. Each frame, the engine selects a desired shader level-of-detail for each block and appends the corresponding command lists for the block to a rendering task queue. To reduce GPU state changes, the engine sorts the command lists in the task queue by shader prior to drawing.

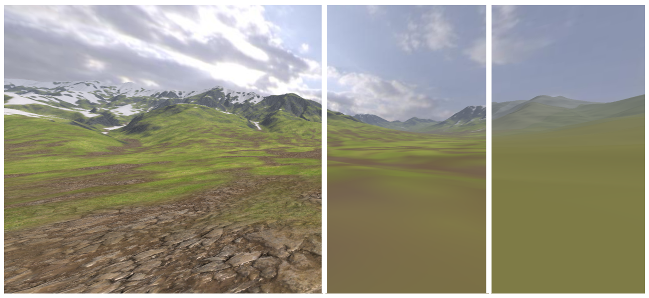


Figure 5: *`Terrain` scene rendered with three shader variants with significantly different performance-quality trade-offs.*

The engine can be configured to draw geometry using several different rendering pipelines. These pipelines contain multiple offline and runtime worlds that implement varied rendering functionality (e.g., object-space and screen-space shading, multi-resolution rendering). We describe these pipelines in detail in the following subsections. Unless otherwise stated, all performance results described in this section were obtained on a machine with an Intel i7-5820K CPU and an NVIDIA GeForce GTX980 Ti GPU.

Our compiler framework is implemented as C++ libraries, including a GLSL back end for compute and graphics kernels. The GLSL back end utilizes plugins for import and export strategies as described in Section 5.1, and can generate kernel code for all pipelines described in this section using the implemented strategies.

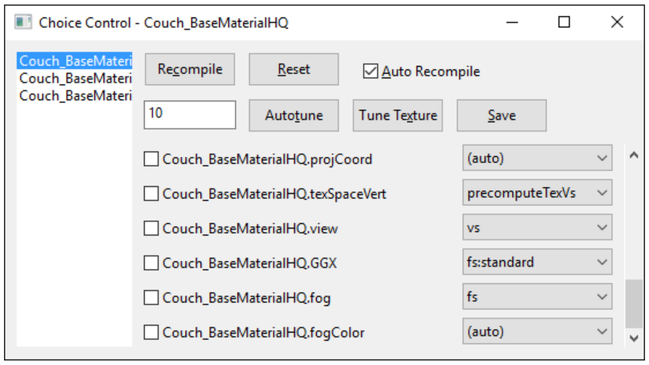
7.1 Exploring a Diverse Set of Optimizations

To demonstrate the wide scope of shader optimizations enabled by overload selection choices, we created a rendering pipeline that consists of five worlds: `PrebakeUniform`, `PrebakeVertex`, `PrebakeTex`, `Vertex`, and `Fragment`. This pipeline, shown in Figure 4, is an extended version of the simple `EnginePipeline` introduced in Section 4.1. The additional `PrebakeVertex` world is used to bake components as additional vertex attributes.

The `PrebakeUniform` world is used to bake components into uniform values at scene load time. Similar to He et al.'s [2015] *parameter shader* stage, our engine uses the `PrebakeUniform` world to support aggressively simplified shaders. The engine runs the `PrebakeUniform` code as an OpenGL compute kernel, to evaluate component values at the vertices of a mesh. The resulting values are averaged over the mesh to generate a per-mesh uniform inputs for subsequent per-frame rendering.

Figure 5 shows a terrain scene, exported from the Unreal Marketplace [2015b], running on this pipeline. The terrain is rendered with a complex shader that blends five material layers (soil, cliff, snow, and two types of grass) and computes additional effects such as fog, color variation and specular lighting. The full shader has 39 components, 27 of which have more than one overload selection choice, yielding millions of possible shader variants. The `Terrain` shader shown in Section 4 is a greatly simplified version of this shader.

As an alternative to making optimization decisions by manually editing a choice file, we developed a shader optimization tool that uses the compiler's choice enumeration API to query for a list of choices available for the current shader, then displays them in an interactive UI (Figure 6). As the user makes optimization decisions, the engine invokes the compiler back-end to generate new GLSL kernels for the resulting shader variants. The tool dynamically loads newly compiled kernels, making performance and quality changes immediately visible to the user.


Choice Control - Couch_BaseMaterialHQ

☐ Couch_BaseMaterialHQ.projCoord	(auto)
☐ Couch_BaseMaterialHQ.texSpaceVert	precomputeTexVs
☐ Couch_BaseMaterialHQ.view	vs
☐ Couch_BaseMaterialHQ.GGX	fs:standard
☐ Couch_BaseMaterialHQ.fog	fs
☐ Couch_BaseMaterialHQ.fogColor	(auto)

Figure 6: *A GUI for exploring shader optimization choices. The user can modify decisions for each choice, and the engine instantly recompiles and applies a new shader variant.*

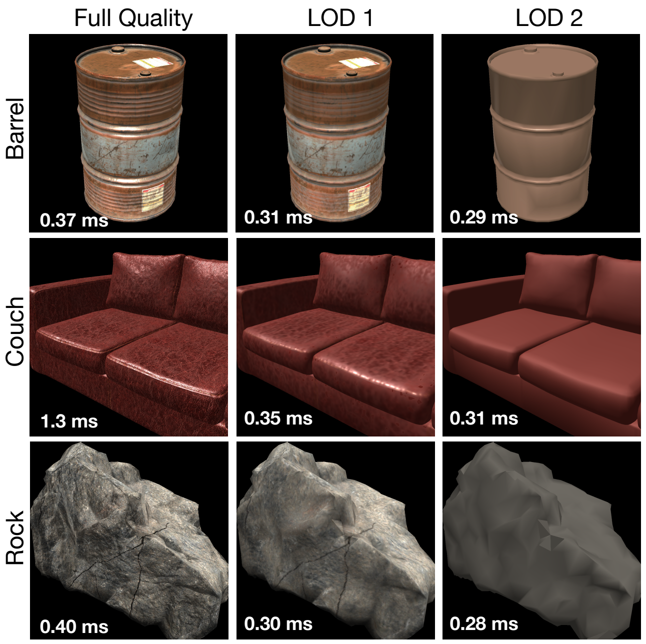


Figure 7: *Shader LODs generated for objects with different surface shaders, all inheriting from a common shader group.*

We have used the UI to create an LOD policy for the terrain shader, consisting of three variants of decreasing runtime cost. While the UI makes it simple to explore the space of choices afforded by a shader, each decision potentially yields significantly different kernel code and engine behavior. For example, different decisions will not only cause vertex, fragment, and compute shaders to change, but will determine which compute passes are needed, cause different numbers of command lists to be submitted, and potentially lead to different allocations of textures and uniform buffers.

7.2 Shader Level-of-Detail Library

Modern game scenes often contain many different types of objects, with different shaders, so manually exploring LOD optimization choices for each object can be tedious. In practice, the same optimization choices may apply to many different shaders when creating LOD policies. To reuse optimization decisions for LOD, we encode the optimization decisions explored in Section 7.1 into a library of base shaders, similar to the approach shown in Listing 4.

The base shader group contains three shaders. The first base shader produces the highest-quality output and computes all important

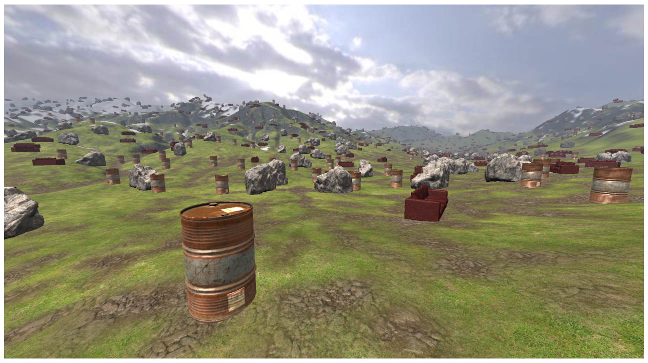


Figure 8: *Terrain scene featuring 10,000 instances of three different object types. Shaders share the same LOD policy.*

shader components in the `Fragment` world; the second base shader computes the `albedo` component in the `PrebakeTex` world and directly uses the mesh's vertex normal for lighting computations; the last base shader (corresponding the lowest level of detail) bakes albedo and lighting parameters into per-object uniform values by placing these components in the `PrebakeUniform` world.

In addition to the terrain shader, we also implemented three other different material surface shaders imported from the Unreal Marketplace: `Barrel` (23 components), `Couch` (28 components), and `Rock` (23 components). Figure 7 shows these shaders rendered at our three LODs. All shaders (including the terrain shader) inherit from this base shader group, so that our single LOD policy was applied to shaders differing significantly in how they compute components such as `albedo` and surface `normal`.

Since the low-detail shaders use baked resources (generated by the `PrebakeUniform` and `PrebakeTex` worlds), these base shaders use locked worlds (Section 6.2) to ensure that all low-detail shaders share the same vertex and fragment kernel code.

To exercise the shader LOD library, we placed 10,000 instances of three different object types in the terrain scene from Section 7.1. With shader LOD enabled, the frame shown in Figure 8 is rendered in 17 ms, compared to 33 ms without LOD.

A well-written engine is able to use information provided by the compiler to optimize rendering performance. Recall that our engine generates a single OpenGL command list for all objects in a scene block that use the same GPU pipeline vertex and fragment shader kernels (these objects are drawn in the batch). When using locked worlds in our base LOD shaders, the shader compiler is able to inform the engine when two different variants share GLSL vertex and fragment kernels. The engine uses this information to aggressively group objects into command lists, reducing the number of command list submissions needed to render the scene from 1024 to 328. This yields a 2 ms (10%) reduction in rendering time compared to using the same three shaders without world locking. Our demo scene only includes three unique types of objects (three unique shaders), and the performance benefits of world locking would increase with the number of unique shaders in the scene.

7.3 New Pipelines and Worlds

To demonstrate Spire's ability to target a variety of rendering pipelines, and to highlight the value of an engine-agnostic shading system, we have implemented a number of advanced rendering pipelines in our engine. Pipelines supporting multi-rate screen-space rendering, and object-space shading are shown in Figure 9.

Our engine implements the multi-rate shading pipeline using two rendering passes [Yang et al. 2008; Shopf 2009]. In the first pass, the engine performs low-resolution shading computations as defined by logic in the `LowResVertex` and `LowResFragment` worlds (which map to GPU vertex and fragment kernels). Components computed in the low-resolution pass are stored into quarter-screen-resolution textures. Code in the subsequent high-resolution rendering pass accesses low-resolution shading results by performing texture fetches parameterized by screen coordinate.

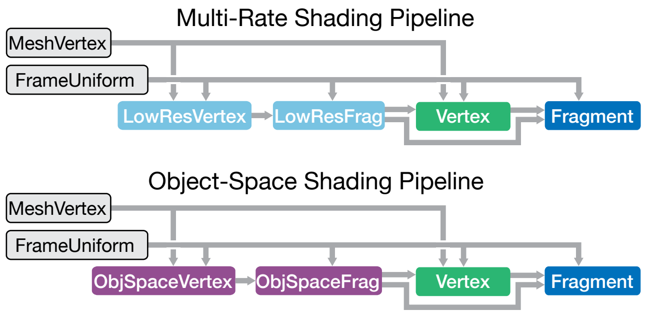


Figure 9: *Pipeline topologies for screen-space multi-rate shading and object space shading. The pipelines have identical topology, but differ in their engine implementations.*

The object-space shading pipeline is also implemented via two-pass rendering. During initialization, the engine allocates textures for all `ObjSpaceFrag` outputs. When rendering a mesh, the first pass uses `ObjSpaceVertex` and `ObjSpaceFrag` logic (again, mapping it to GPU pipeline vertex and fragment shading) to compute and store components into textures in the object's UV space. The second pass fetches these results using the object's texture coordinates.

Although they employ different implementation techniques, the multi-rate shading and object-space shading pipelines feature the same topology. This is because a pipeline topology only describes data dependencies between worlds; it does not reflect the execution semantics of the pipeline itself. The rendering engine is fully responsible for defining and implementing the meaning of each world. For example, the engine could adopt a different implementation of the multi-rate shading pipeline: executing `LowResFrag` world computations once-per-pixel and `Fragment` world computations once per screen multi-sample to achieve anti-aliasing.

As another example, shaders written against the object-space shading pipeline (which shares object-space shading results across fragments) could also be used to share shading results between eyes in a stereo rendering pipeline. Although not shown in Figure 9, we have also implemented this pipeline in our engine. The pipeline runs `Vertex` and `Fragment` world logic once per eye and `ObjSpaceVertex` and `ObjSpaceFrag` logic one per frame when configured to render stereo. In each of these cases, the change of pipeline execution semantics is transparent to the shader, and retargeting shaders to these new pipelines only requires a reinterpretation of overload selection decisions.

We were able to compile the shaders described in Sections 7.1 and 7.2 against our multi-rate and object-space shading pipelines without modifying their source code; optimization choices enabled by the new worlds became available once the shaders were recompiled. If future GPU hardware architectures evolve to natively support screen-space multi-rate [Vaidyanathan et al. 2014; He et al. 2014] or object-space shading [Clarberg et al. 2014], Spire will make it easier to retarget existing shaders to exploit these new capabilities.

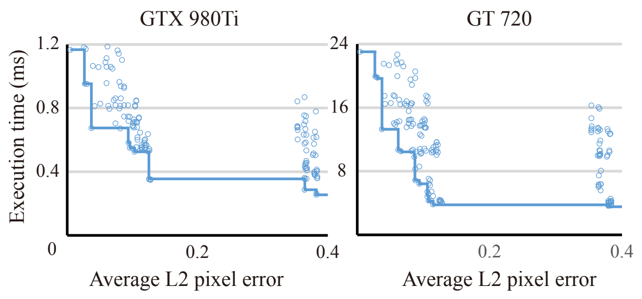


Figure 10: *Performance-quality Pareto curve for* `Couch` *shader variants explored by our shader auto-tuner. Results are plotted for a high-end and low-end GPU. Each blue circle represents a shader variant generated by the auto-tuner.*

7.4 Auto-Tuning Tool

Our compiler's ability to enumerate the space of component overload selection choices for a shader facilitates the implementation of auto-tuning tools that search for shader variants that meet particular performance-quality requirements. Using the compiler's API we were able to quickly implement a powerful auto-tuning tool for our engine. This tool goes beyond prior work in shader auto-tuning [He et al. 2015; Wang et al. 2014; Sitthi-Amorn et al. 2011] to search the space of rate reduction options, term simplifications, and even make decisions about the storage format of shader assets.

Automatic component overload selection. To leverage a developer's intuition about the most important choices needed to optimize a shader (which components are most likely to have significant performance-quality impacts), the tool allows the user to specify a set of components for which overload selection decisions should be explored. The auto-tuner enumerates all possible decisions for the selected components and evaluates visual error and performance of the resulting shader variants. (It uses L2 pixel distance from a high-quality reference image as an error metric.)

We used the auto-tuner to find the best-quality variant of the `Couch` shader that generates images within an 8 ms budget on two different GPU platforms: NVIDIA GTX 980Ti and GT 720. We set the auto-tuner to explore choices for four of the shader's 28 components (`albedo`, `normal`, `metallic`, `roughness`) that are most compute-intensive and likely to have significant performance impact. The shader also exposes an algorithmic choice of three microfacet model implementations: `standardPhong` implements a simplified Blinn-Phong distribution, `standardGGX` evaluates the GGX model directly in shader, and `GGX_tex` implements the GGX model using precomputed lookup tables for the distribution terms. The auto-tuner generated and measured the performance and quality of 243 shader variants; this process took 45 seconds on the GTX 980Ti and 233 seconds on the GT 720. This auto-tuning time is similar to compilation times reported by He et. al's [2015] auto-tuning shader simplifier, despite our implementation considering a wider optimization space that results in more aggressively optimized shaders (prebaking components into textures and vertex attributes were not optimizations considered by this prior work). Our implementation uses a brute-force search over potential shader variants, which is acceptable because decisions are made per component, instead of per-instruction as in prior work. It also leverages human guidance (telling the auto-tuner which components are likely to be most important) to further limit the search space. A more advanced auto-tuning tool could adopt more sophisticated search strategies, like those of He et al. [2015], if faster auto-tuning is needed.

On the GTX 980Ti, the tool chooses to compute auto-tuned components in the `Fragment` world, and use the `standardGGX` microfacet implementation. (Performance is not an issue, so the system generates a high-quality shader.) To meet the selected performance constraint on the GT 720, the tool makes the same overload selection decisions as our manually chosen medium-quality variant in Figure 7 (which computes auto-tuned components in `PrebakeTex` world), and chooses the faster `standardPhong` microfacet implementation. The tool does not choose the `GGX_tex` implementation on any of the platforms, as the savings from reduced arithmetic operations is not worth the additional texture fetch. Figure 10 shows performance-quality Pareto curves for all explored shader variants on both platforms.

Data layout selection. In addition to making choices about how and where to evaluate components, the auto-tuning tool also makes decisions about the size and format of textures created as a result of these decisions. The auto-tuning tool analyzes the contents of precomputed texture data to determine the best storage format. For example, if a 3-channel texture populated by the `PrebakeTex` world contains only values between -1 and 1, the auto-tuner will attempt to treat the component as a normal map and choose an appropriate packed texture representation. Although not described in this paper for brevity, Spire supports the ability to annotate shader components with attributes that are passed to the code generator and rendering engine as meta data. For example:

```
[Normal][TextureSize: "512"]
vec3 normal = ...;
```

The auto-tuning tool emits these attributes as a result of auto-tuning (they are included in the Spire optimization decision file). For the `Couch` shader, the auto-tuner correctly injects the attributes so that the `normal` is stored in a `RGB10_A2` format texture, and albedo term in standard `RGB8` format.

8 Discussion

In this paper we presented a new shader language and compiler framework that enables rapid exploration of shader optimization choices. By extending the scope of a shader to target engine defined worlds with diverse semantics, and by introducing a component overloading mechanism to the language, we achieved many optimizations that would have involved global code changes if shaders were written using traditional systems. We further extend our language with shader groups and locked worlds, mechanisms that support the implementation of libraries that can be applied to large numbers of shaders. Our design allows creating pipeline-agnostic shaders which can remain unchanged even when the underlying rendering pipeline evolves. We believe this capability is exciting since GPU hardware evolution to support a wider range of rendering techniques and emerging applications (e.g., virtual reality) stands to trigger changes in design of game rendering engines.

Our ideas are based on the assumption that shaders evaluate signals at local surface locations and it is this assumption that allows components to migrate between worlds. Other operations, such as vertex data optimization [Kavan et al. 2011], geometry processing, and image post processing, do not satisfy this assumption, but could also benefit from a system that exposes their optimization choices. We are interested in establishing a connection between our existing framework and a more diverse set of rendering algorithms.

Last, our current design focuses on defining computation locations, but does not provide an explicit mechanism for managing storage locations. The current abstraction assumes each world implicitly defines a storage space that holds its outputs. In a more general setting, storage could be decoupled from computation, and we would like to explore the system-scope implications of adding a first-class notion of storage.

9 Acknowledgments

Support for this research was provided by the National Science Foundation (IIS-1253530) and by a NVIDIA Faculty Partnership. We would like to thank Nicolas Feltman and Adrian Sampson for valuable conversations.

References

ANSEL, J., CHAN, C., WONG, Y. L., OLSZEWSKI, M., ZHAO, Q., EDELMAN, A., AND AMARASINGHE, S. 2009. Petabricks: A language and compiler for algorithmic choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM, New York, NY, USA, PLDI '09, 38–49.

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

BAUER, M., TREICHLER, S., SLAUGHTER, E., AND AIKEN, A. 2012. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, IEEE Computer Society Press, SC '12, 66:1–66:11.

BUNGIE, 2014. Destiny computer game. Available at http://www.destinythegame.com.

CLARBERG, P., TOTH, R., HASSELGREN, J., NILSSON, J., AND AKENINE-MÖLLER, T. 2014. Amfs: Adaptive multi-frequency shading for future graphics processors. *ACM Trans. Graph. 33*, 4 (July), 141:1–141:12.

EPIC GAMES, 2015. Unreal Engine 4 documentation. Available at http://docs.unrealengine.com.

EPIC GAMES, 2015. Unreal Engine 4 Marketplace Web Site. http://www.unrealengine.com/marketplace.

FATAHALIAN, K., HORN, D. R., KNIGHT, T. J., LEEM, L., HOUSTON, M., PARK, J. Y., EREZ, M., REN, M., AIKEN, A., DALLY, W. J., AND HANRAHAN, P. 2006. Sequoia: programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ACM, SC '06.

FOLEY, T., AND HANRAHAN, P. 2011. Spark: modular, composable shaders for graphics hardware. *ACM Trans. Graph. 30*, 4 (July), 107:1–107:12.

HANRAHAN, P., AND LAWSON, J. 1990. A language for shading and lighting calculations. *SIGGRAPH Comput. Graph. 24*, 4 (Sept.), 289–298.

HE, Y., GU, Y., AND FATAHALIAN, K. 2014. Extending the graphics pipeline with adaptive, multi-rate shading. *ACM Trans. Graph. 33*, 4 (July), 142:1–142:12.

HE, Y., FOLEY, T., TATARCHUK, N., AND FATAHALIAN, K. 2015. A system for rapid, automatic shader level-of-detail. *ACM Trans. Graph. 34*, 6 (Oct.), 187:1–187:12.

KAVAN, L., BARGTEIL, A. W., AND SLOAN, P.-P. 2011. Least squares vertex baking. In *Proceedings of the Twenty-second Eurographics Conference on Rendering*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGSR '11, 1319–1326.

KESSENICH, J., BALDWIN, D., AND ROST, R., 2014. The OpenGL shading language language version 4.4. Available at https://www.opengl.org.

LALONDE, P., AND SCHENK, E. 2002. Shader-driven compilation of rendering assets. *ACM Trans. Graph. 21*, 3 (July), 713–720.

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. *ACM Trans. Graph. 22*, 3 (July), 896–907.

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

MICROSOFT, 2016. HLSL shader model 5 documentation. Available at https://msdn.microsoft.com.

PATNEY, A., TZENG, S., SEITZ, JR., K. A., AND OWENS, J. D. 2015. Piko: A framework for authoring programmable graphics pipelines. *ACM Trans. Graph. 34*, 4 (July), 147:1–147:13.

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 01, Annual Conference Series*, ACM, New York, NY, USA, 159–170.

RAGAN-KELLEY, J., ADAMS, A., PARIS, S., LEVOY, M., AMARASINGHE, S., AND DURAND, F. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph. 31*, 4 (July), 32:1–32:12.

SHOPF, J., 2009. Mixed resolution rendering. Game Developers Conference 2009 slides.

SITTHI-AMORN, P., MODLY, N., WEIMER, W., AND LAWRENCE, J. 2011. Genetic programming for shader simplification. *ACM Trans. Graph. 30*, 6 (Dec.), 152:1–152:12.

SUGERMAN, J., FATAHALIAN, K., BOULOS, S., AKELEY, K., AND HANRAHAN, P. 2009. GRAMPS: A programming model for graphics pipelines. *ACM Transactions on Graphics 28*, 1, 4:1–4:11.

VAIDYANATHAN, K., SALVI, M., TOTH, R., FOLEY, T., AKENINE-MÖLLER, T., NILSSON, J., MUNKBERG, J., HASSELGREN, J., SUGIHARA, M., CLARBERG, P., JANCZAK, T., AND LEFOHN, A. 2014. Coarse pixel shading. In *High Performance Graphics 2014*, 10.

WANG, R., YANG, X., YUAN, Y., CHEN, W., BALA, K., AND BAO, H. 2014. Automatic shader simplification using surface signal approximation. *ACM Trans. Graph. 33*, 6 (Nov.), 226:1–226:11.

YANG, L., SANDER, P. V., AND LAWRENCE, J. 2008. Geometry-aware framebuffer level of detail. In *Proceedings of the Nineteenth Eurographics Conference on Rendering*, Eurographics Association, EGSR'08, 1183–1188.