SIGGRAPH ASIA 2019 BRISBANE

Staged Metaprogramming for Shader System Development

Kerry A. Seitz, Jr.,* Tim Foley,†
Serban D. Porumbescu,* and John D. Owens*

sa2019.siggraph.org

*UCDAVIS UNIVERSITY OF CALIFORNIA

†

What is a shader system?

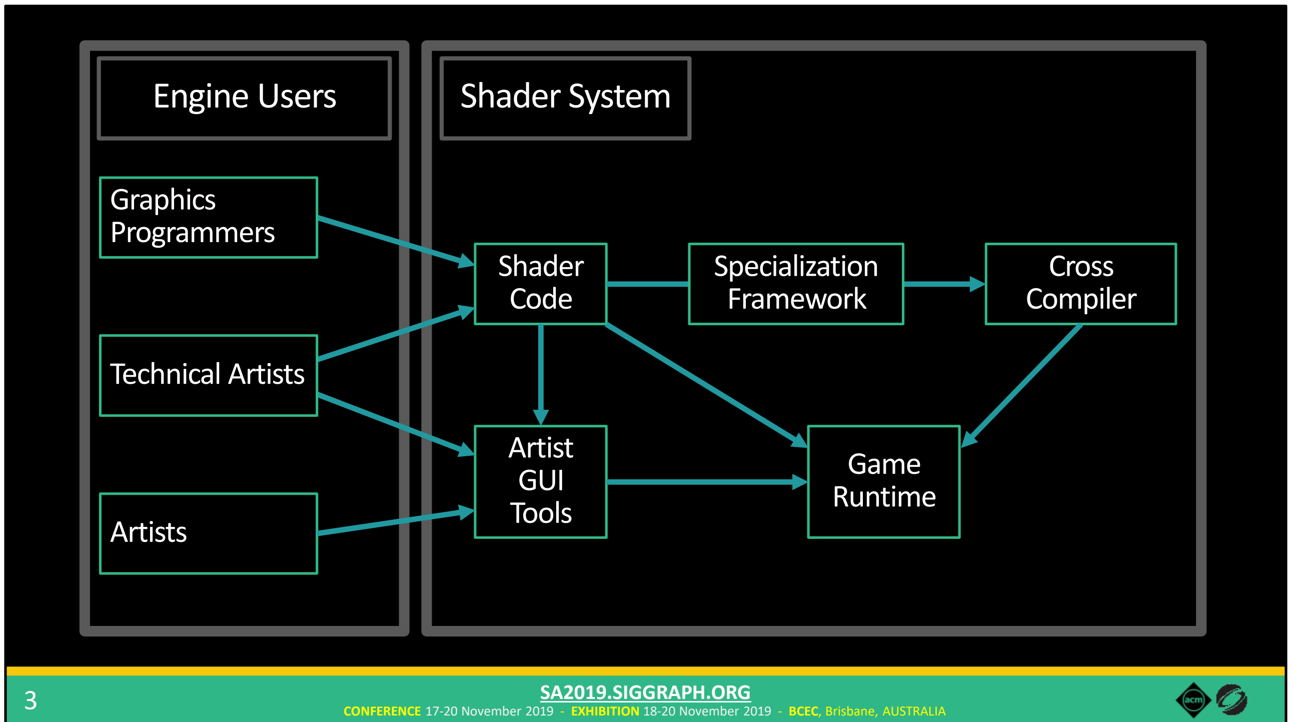
A game engine component that facilitates interacting with the rendering process

Let’s start by defining what I mean by a “shader system.”

A shader system is a game engine component that facilitates interacting with the rendering process.

Specifically, I’m talking about real-time 3D game engines like Unreal, Unity, and other in-house engines, which means that not only is performance critical, but so is enabling a wide variety of users to control different aspects of rendering.

Let’s look at an example of what I mean…



We have different types of engine users who need to use a shader system in different ways.

First, we have graphics programmers who need to be able to write shader code, in HLSL or GLSL for example.

Of course that shader code needs to be complied into executable kernels, and possibly cross-compiled if you're shipping on multiple platforms.

Then there's technical artists who also write shader code. Unlike graphics programmers, they are typically not experts in things like shader optimization. Maybe they'll use plain HLSL or GLSL too, or maybe an engine chooses to provide a custom Domain-Specific Language (or DSL).

That DSL might enable them to express which parameters to expose to a GUI that artists use to create and configure different materials.

Those configurations, along with the shader code and compiled kernels, need to be interfaced with the runtime engine code, which sets up and launches the rendering work.

And finally, shaders need to be specialized in order to achieve the best performance.

Specialization involves taking a shader that includes code and parameters for multiple different feature options, and then generating many different variants from that shader, each corresponding to a different subset of those features. As a result, expensive features do not impact performance when they are not needed.

Engine developers need to design these shader systems to both result in highly optimized final code while simultaneously providing the appropriate interfaces for each type of person involved in game development. But unfortunately…

Graphics APIs don't help with this task

Direct3D / OpenGL / etc. are singularly focused on providing robust, high-performance implementations on a wide range of hardware

In contrast, shader systems are ***multifaceted***

- They must provide a variety of interfaces for different users
- Engine devs are left to create these missing facets, layered on top of the APIs

Graphics APIs don't really help with this task. They are singularly focused on providing a robust, high-performance implementation on a wide range of hardware.

But as we've established, shader systems are multifaceted – they must provide a variety of interfaces for different users. Thus, engine developers are left to create layered implementations of these missing facets on top of the graphics APIs.

So how do they do that?

Current Methods

Let's look at some current methods used to implement shader systems.

Four methods to implement shader systems

Plain C++ and HLSL*

- Preprocessor #ifdefs + #defines, shared headers for data structures, manually-authored C++ class for each shader

A layered domain-specific language (DSL) with embedded HLSL*

- Unity's ShaderLab

A DSL that manipulates and generates HLSL*

- Bungie's TFX language [Tatarchuk and Tchou 2017]

Modifying HLSL*

- Slang [He et al. 2018]

*or any modern shading language (e.g., GLSL or Metal Shading Language)

One is to just the facilities provided by plain C++ and HLSL. (Quick aside: when I say HLSL here and for the rest of the talk, I could substitute any modern shading language like GLSL or Metal Shading Language). We could use preprocessor #ifdefs and #defines in the shader to express specialization options, create shared headers for data structure, and manually author a C++ class for each shader to provide an interface for CPU engine code.

Another is to implement a layered DSL that contains embedded HLSL. Unity's ShaderLab is an example of this approach.

You could also create a more sophisticated DSL that manipulates and generates HLSL, such as Bungie's TFX language used in Destiny.

And finally, you could go so far as to modify HLSL to implement custom features, like the Slang shading language, which added some modern programming language features to HLSL.

In the paper, we go into details on all of these, but let's briefly look at one in a little more detail.

An example in Unity's ShaderLab

```
Shader "SurfaceShader" {
  Properties { lightDirection {"Light Direction", Vector} = (0,0,0) }
  …
  CGPROGRAM
  #pragma multi_compile STANDARD SUBSURFACE CLOTH

  float3 lightDirection;

  float4 surfaceShader(…) {
  …
  #if defined(STANDARD)
    color = evalStandardMaterial(shadingData);
  #elif defined(SUBSURFACE)
    color = evalSubsurfaceMaterial(shadingData);
  #elif defined(CLOTH)
    color = evalClothMaterial(shadingData);
  #endif
    return color * max(0, dot(shadingData.normal, lightDirection);
  }
  ENDCG
}
```

Express specialization options using #if

7

Here's an example shader written in Unity's ShaderLab DSL. I'm not going to discuss everything here, but I do want to highlight a few things.

First, specialization options are expressed using preprocessor #ifs, just like you would do if you were using plain HLSL. At first, that might seem fine, but what if (for some reason) you wanted to generate a specialized variant that contained both STANDARD and CLOTH material? You couldn't do that from this shader code. Why might you want to generate such a variant? Maybe you're using a deferred renderer, but more on that later.

One of the nice things that ShaderLab provides is…

An example in Unity's ShaderLab

```
Shader "SurfaceShader" {
  Properties { lightDirection {"Light Direction", Vector} = (0,0,0) }
  …
  CGPROGRAM
  #pragma multi_compile STANDARD SUBSURFACE CLOTH

  float3 lightDirection;

  float4 surfaceShader(…) {
  …
  #if defined(STANDARD)
    color = evalStandardMaterial(shadingData);
  #elif defined(SUBSURFACE)
    color = evalSubsurfaceMaterial(shadingData);
  #elif defined(CLOTH)
    color = evalClothMaterial(shadingData);
  #endif
    return color * max(0, dot(shadingData.normal, lightDirection);
  }
  ENDCG
}
```

Custom #pragma syntax to list specialization options

- Custom #pragma syntax enables ShaderLab compiler to automatically generate specialized variants

8

ShaderLab has a custom #pragma syntax to list specialization options.

This enables the ShaderLab compiler to automatically generate all specialized shader variants, rather than requiring users to manually generate them.

An example in Unity's ShaderLab

```
Shader "SurfaceShader" {
  Properties { lightDirection {"Light Direction", Vector} = (0,0,0) }
  …
  CGPROGRAM
  #pragma multi_compile STANDARD S[illegible]FACE CLOTH

  float3 lightDirection;

  float4 surfaceShader(…) {
  …
  #if defined(STANDARD)
    color = evalStandardMaterial(shadingData);
  #elif defined(SUBSURFACE)
    color = evalSubsurfaceMaterial(shadingData);
  #elif defined(CLOTH)
    color = evalClothMaterial(shadingData);
  #endif
    return color * max(0, dot(shadingData.normal, lightDirection);
  }
  ENDCG
}
```

Double declaration of artist-configurable parameters

- Custom #pragma syntax enables ShaderLab compiler to automatically generate specialized variants

In order to expose artist-configurable parameters to a GUI, ShaderLab has a special "Properties" listing. But unfortunately, each of these parameters must be declared twice – once in the Properties and again in the embedded HLSL.

An example in Unity's ShaderLab

```
Shader "SurfaceShader" {
  Properties { lightDirection {"Light Direction", Vector} = (0,0,0) }
  …
  CGPROGRAM
  #pragma multi_compile STANDARD SUBSURFACE CLOTH

  float3 lightDirection;

  float4 surfaceShader(…) {
  …
  #if defined(STANDARD)
    color = evalStandardMaterial(shadingData);
  #elif defined(SUBSURFACE)
    color = evalSubsurfaceMaterial(shadingData);
  #elif defined(CLOTH)
    color = evalClothMaterial(shadingData);
  #endif
    return color * max(0, dot(shadingData.normal, lightDirection);
  }
  ENDCG
}
```

Bug!!!
lightDirection is a
float3, not a float4

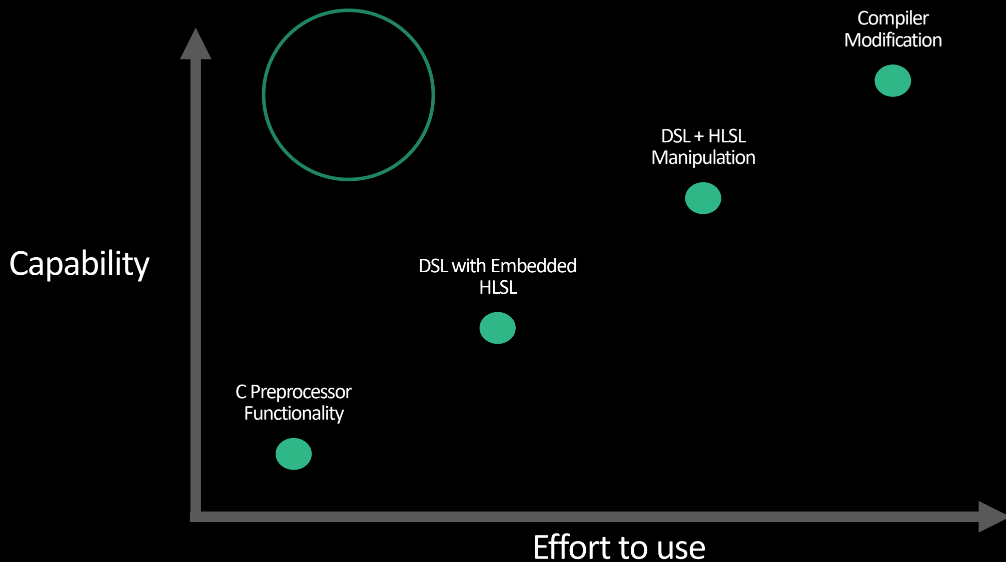
- Custom #pragma syntax enables ShaderLab compiler to automatically generate specialized variants
- Use a "stringly-typed" interface to set parameters:

```
Shader.SetVector("lightDirection",
  Vector4(1.0, 1.0, 1.0, 1.0);
```

10

Finally, runtime engine code sets parameters using a "stringly-typed" interface. This interface doesn't provide good error checking, which can lead to subtle bugs, such as here where I've accidentally used the wrong type when setting the lightDirection parameter.

Some of the other methods I mentioned improve upon these issues, but one important thing to note is that…



These methods are on an unfavorable continuum. Methods with greater capability require greater effort to use.

C preprocessor functionality is fairly simple to work with, but it's limited in the types of things you can implement with it.

At the other end of the spectrum, if you're willing to modify a compiler to add features to HLSL, you have a lot of flexibility but now you have a much larger codebase to maintain, especially as the core HLSL continues to evolve.

Engine developers today are faced with the problem of balancing between the benefits that new features might provide to users versus the effort required to implement those features.

What we'd really like is a technique that sidesteps this trade-off – one that provides lots of capabilities while requiring only a modest effort to use.

Based on this observation…

Design Goals

(… Based on this observation)

as well as the other issues we've seen in modern shader systems, we came up with a set of design goals to guide our work.

Design goals

Minimize implementation effort and maintenance costs

- E.g., we don't want to build or modify a compiler

Early error detection

- Unlike "stringly-typed" interfaces

Don't repeat yourself (DRY)

- Avoid repeat declarations of shader parameters, constant buffers, etc.

Each engine requires a unique shader system, customized to the engine's design and the needs of its users. If we can minimize the effort required to build and maintain shader systems, we can better enable developers to create robust, feature-rich implementations.

We'd like to be able to catch errors earlier, in contrast to "stringly-typed" interfaces in ShaderLab and in graphics APIs.

Programmers shouldn't have to declare the same shader parameter, constant buffer, etc. more than once.

Design goals (cont.)

Performance

- Minimize overheads to CPU and GPU code

Productivity for artists and technical artists

- Don't hinder their workflows

Support options for static and dynamic feature composition

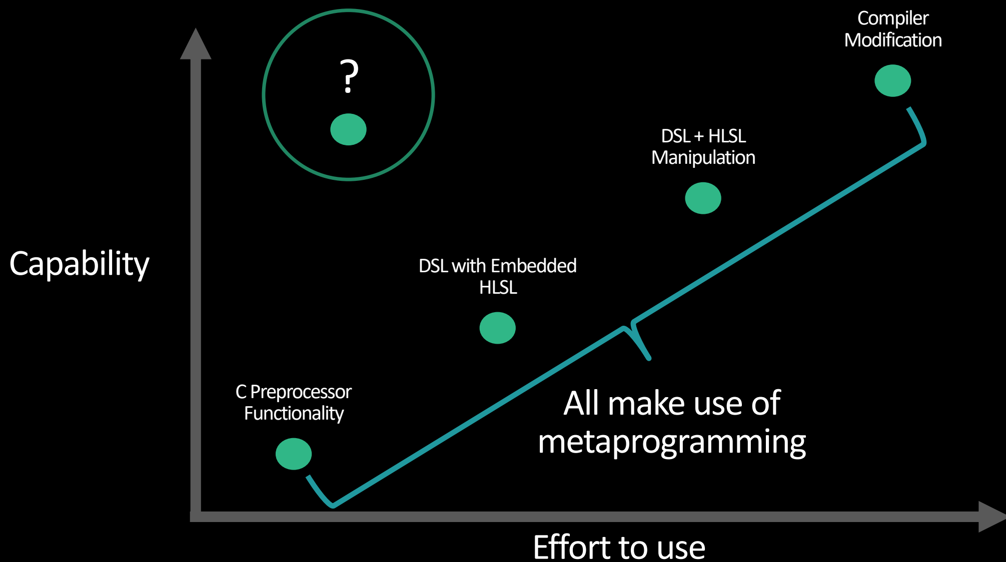
- To explore trade-offs between static and dynamic shader specialization

Performance is paramount in real-time graphics applications, so our system should strive to minimize overheads to GPU shader code and CPU engine code, as well as enable developers to explore opportunities to improve performance.

Productivity is key for artists, so a shader system must provide them with familiar workflows.

To achieve maximum performance, engines generate many specialized shader variants. However, complete static specialization can lead to additional overheads that decrease performance. So, we want our system to enable exploration of these trade-offs in the hopes of improving overall performance.

Given the landscape of existing solutions…



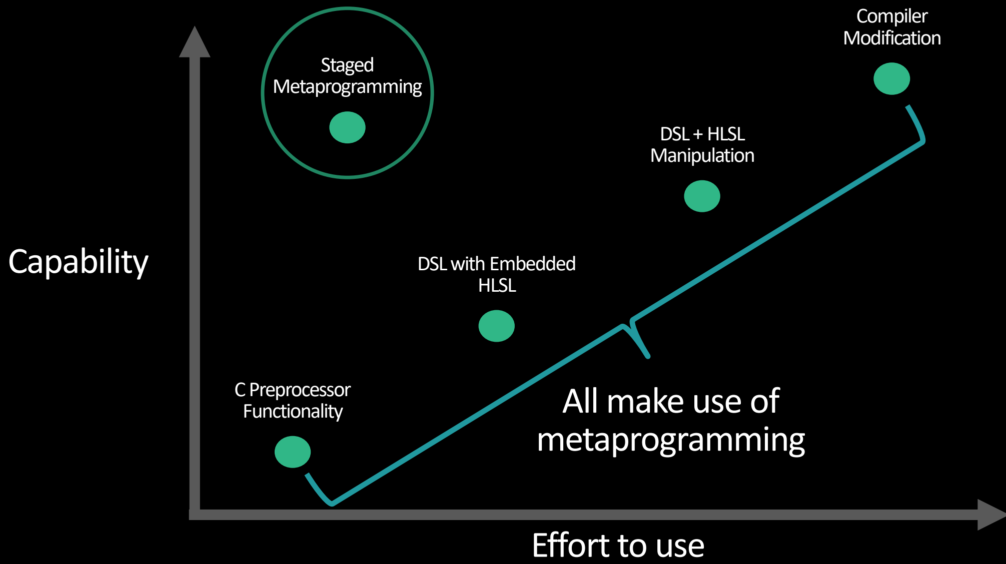
Our first goal of minimizing implementation effort seems at odds with some of our other goals. Certainly, we could achieve many of them by modifying a compiler and adding features to HLSL, but that requires a high implementation effort.

Is there a method that avoids this trade-off?

When examining these existing methods, we discovered that they all happen make extensive use of metaprogramming, whether they realize it or not. We broadly define metaprogramming as writing code that manipulates other code, including reading, analyzing, transforming, or generating code.

Using this key insight, we decided to make metaprogramming a fundamental design principle at the core of our shader system, and to find a metaprogramming technique that sidesteps this apparent trade-off between capability and complexity.

As I'm sure you can guess from the title of this talk…



The technique we identified is staged metaprogramming.

Staged Metaprogramming

So, what is staged metaprogramming…

Staged metaprogramming

Explicit stages of code execution

- E.g., compile-time stage / runtime stage

Code running in earlier stages can construct and manipulate code in later stages

Consistent with description of *multi-level languages* [Taha 1999]

- Also includes *multi-stage languages*

In staged metaprogramming, there are multiple explicit stages of code execution. For example, we could have a stage that conceptually executes at application compile time, versus a stage that executes at application run time.

Code running in an earlier stage of execution can construct and manipulate code that will run in a later stage.

Our definition of staged metaprogramming aligns with the description of a multi-level language and also includes multi-stage languages as well. If you're familiar with those terms, this might provide some extra context.

So what makes up a staged metaprogramming environment?

Key features of staged metaprogramming

Code is a first-class citizen

- Pass as arguments, return from functions, store in structs

Code is constructed using regular language syntax using *quasi-quote*

- myCode = quote var outColor = diffuse + specular end
- Quasi-quotes are hygienic and lexically scoped (but you can violate this)

Unquote inserts quasi-quoted code into the runtime application

- [myCode]

Quasi-quotes can be specialized to generate different version

Let's talk about the key features.

Code is a first-class citizen. Programs can operate on code in the same way that they can operate on other constructs, including passing code as arguments, returning code from functions, and storing code in data structures.

Code is constructed using regular language syntax using quasi-quote. In this example, we have the keywords "quote" and "end" to denote that we're constructing code. The code between these keywords is expressed using regular syntax, but this code is not executing here. Instead, it is stored in the "myCode" variable for use later.

These quotes are hygienic and lexically scoped, so the definition of "outColor" here would not conflict with another variable using the same name elsewhere (unlike preprocessor macros). However, you intentionally can violate this property when needed.

The unquote operator splices quoted code into the runtime application. Here we're taking the myCode variable and inserting its contents into the program.

Also, quotes can be specialized to generate different versions, similar to how shaders can be specialized into different variants.

If you're used to working in C++ or another popular systems language, you're probably not used to seeing these quote and unquote constructs. Well that's because…

Lua-Terra: a research substrate for staged metaprogramming

C++ and HLSL don't have the features of staged metaprogramming

So, we used Lua-Terra [DeVito et al. 2013] to explore our ideas

- Multi-stage language
- Uses Lua code in the first stage to manipulate next-stage Terra code

Lua – high-level scripting language
Terra – simple, low-level, C-like language

SA2019.SIGGRAPH.ORG
CONFERENCE 17-20 November 2019 - EXHIBITION 18-20 November 2019 - BCEC, Brisbane, AUSTRALIA

These languages don't have the features required for staged metaprogramming.

We used a language called Lua-Terra to demonstrate how staged metaprogramming is useful for shader systems. Lua-Terra is a multi-stage language that uses Lua code in the first stage to manipulate and generate next-stage Terra code

You might be familiar with Lua. It is a high-level scripting language commonly used in game development already.

In contrast, Terra is simple, low-level, C-like language. We chose Lua-Terra specifically because Terra models the lower-level systems language environment that is commonly used in engine development.

However, high-level scripting and code generation can be expensive operations, and as we know performance is critical for game engines…

Compile-time staged metaprogramming

Performance is important, so all metaprogramming occurs at application compile time

- Avoids overhead of generating code at runtime
- In contrast to prior work (e.g., Sh [McCool et al. 2002] and Vertigo [Elliott 2004])

All Lua code executes at compile time

Runtime application and shader code are written in Terra

SA2019.SIGGRAPH.ORG
CONFERENCE 17-20 November 2019 - EXHIBITION 18-20 November 2019 - BCEC, Brisbane, AUSTRALIA

So in our system, all metaprogramming occurs at application compile time. We aren't generating any code while the game is running, all of that happens beforehand. This is in contrast to prior work, which generates code at runtime.

On the last slide, I mentioned that the Lua code metaprogrammings the Terra code, and since all of the metaprogramming happens at compile time, all of the Lua code executes at compile time as well. We don't run any Lua code during game runtime (although we could if we wanted to add Lua scripting into the application).

What's left at runtime is just the C-like Terra code. The game runtime, as well as all shader code are written in Terra.

Now let's take a look at a shader in our system, which we call Selos.

Example shader in our system (called Selos)

```
shader SurfaceShader {
  ConfigurationOptions {
    MaterialType = MaterialSystem.MaterialTypeOption.new()
  }
  …
  uniform LightData {
    @UIType(Slider3) lightDirection : vec3
  }
  …
  fragment code
    …
    color = [MaterialType:eval()](shadingData)
    return color *
      max(0, dot(shadingData.normal, lightDirection);
  end
}
```

Single declaration of parameter

22

Again, I'm just going to highlight a few key points.

Unlike in ShaderLab, in Selos we can express GUI controls directly alongside the parameter declaration, avoiding the double-declaration problem.

Example shader in our system (called Selos)

```
shader SurfaceShader {
  ConfigurationOptions {
    MaterialType = MaterialSystem.MaterialTypeOption.new()
  }
  …
  uniform LightData {
    @UIType(Slider3) lightDirection : vec3
  }
  …
  fragment code
    …
    color = [MaterialType:eval()](shadingData)
    return color *
      max(0, dot(shadingData.normal, lightDirection);
  end
}
```

- Statically-checked interface to shaders:

```
var myShader = SurfaceShader.new()
var lightData =
         myShader.LightData:map(…)
lightData.lightDirection = vec4(…)
```

Compile-time error:
lightDirection is a vec3

23

Our system generates a statically-checked interface for shaders, meaning that that bug from my ShaderLab code before is instead reported as a compile-time error in Selos.

Instead of using preprocessor #if…

Example shader in our system (called Selos)

```
shader SurfaceShader {
  ConfigurationOptions {
    MaterialType = MaterialSystem.MaterialTypeOption.new()
  }
  …
  uniform LightData {
    @UIType(Slider3)
  }
  …
  fragment code
    …
    color = [MaterialType:eval()](shadingData)
    return color *
      max(0, dot(shadingData.normal, lightDirection);
  end
}
```

Specialization expressed and controlled through ConfigurationOptions

- Statically-checked interface to shaders:

```
var myShader = SurfaceShader.new()
var lightData =
         myShader.LightData:map(…)
lightData.lightDirection = vec4(…)
```

- Automatically generate variant (like ShaderLab)
- Opportunity to explore more specialization options (we'll return to this)

24

Shader specialization is expressed and controlled through ConfigurationOptions.

This allows Selos to automatically generate all variants (like in ShaderLab).

And it also enables us to explore other options for shader specialization. I mentioned before that you couldn't generate a shader with both STANDARD and CLOTH materials from the ShaderLab code, but in ours we easily can. We'll return to this part later.

Staged metaprogramming is the principle design decision in our system…

Example shader in our system (called Selos)

```
shader SurfaceShader {
  ConfigurationOptions {
    MaterialType = MaterialSystem.MaterialTypeOption.new()
  }
  …
  uniform LightData {
    @UIType(Slider3) lightDirection : vec3
  }
  …
  fragment code
    …
    color = [MaterialType:eval()](shadingData)
    return color *
      max(0, dot(shadingData.normal, lightDirection);
  end
}
```

25

But if you'll notice, those quote and unquote mechanisms I described earlier don't show up in this code. In fact, this shader looks pretty similar to a shader written in GLSL or HLSL, and it doesn't really exhibit aspects of staged metaprogramming directly. This design is intentional.

While staged metaprogramming underlies our system's implementation, it also introduces some new and unfamiliar programming constructs. How do we cope with that?

Other Key Design Decisions

That leads me to our other key design decisions.

Other key design decisions

Write shader definitions using a DSL

- Present a familiar interface to technical artists
- Don't expose the metaprogramming directly

Represent shaders as compile-time Lua objects

- Consistent interface to manipulate shader code
- Compile-time only, so it doesn't add runtime overhead

First, shaders are written in a DSL that's similar to GLSL. This provides a familiar interface to technical artists, so that they can be productive without worrying about those new metaprogramming constructs. So how do we use staged metaprogramming then?

Internally in the system, we represent shaders as compile-time Lua objects. This provides a consistent interface to manipulate shader code, since we can store code directly in data structures. Because it only exists at compile time, this representation does not add overhead to the runtime application.

While shader-specific features are expressed through our DSL…

Other key design decisions (cont.)

Write shader logic and application code in the same language

- Share types and functions between CPU and GPU code

Generate runtime data structures for shaders

- Statically-checked interface – catches errors at application compile time
- Downside: must recompile application whenever a shader's interface changes
  - But we can hot reload if only the logic changes

SA2019.SIGGRAPH.ORG
CONFERENCE 17-20 November 2019 - EXHIBITION 18-20 November 2019 - BCEC, Brisbane, AUSTRALIA

Core shader logic as well as CPU-side code is written in Terra. This allows us to share types and functions between CPU and GPU code. You get that for free in our system.

As I mentioned earlier, we generate runtime data structures for shaders, which helps us catch more errors at compile time. However, this means that the game must be recompiled if the interface to a shader changes, like if you add a parameter for example. But if only the core logic changes, we can still hot reload shaders. It's a bit of a trade-off – the application needs to be recompiled more often, but it does provide better error checking.

I want to take a second to point out that…

Our implementation required only a modest effort

System Component	Language(s)	Lines of Code
Unity ShaderLab DSL	Flex/Bison/other	~2000
Slang Compiler	C++	~67,000
Selos (Ours)		
Core	Lua-Terra	~2300
HLSL/GLSL Backend	Lua-Terra	~2200

Improvements over ShaderLab, but with similar lines of code

Backends are reusable

Our implementation of these features required only a modest effort. The lines of code of the Selos core is comparable to that of Unity's ShaderLab implementation, while also improving on some of ShaderLab's issues. And both are much smaller than building or modifying a compiler.

We also had to implement backends to convert Terra to HLSL and GLSL, but we believe that these components are not engine-specific and could be shared across shader systems as an open source component.

In the previous design decisions, we recommend hiding the complexities of metaprogramming from many shader writers behind our shader DSL. But the power of raw metaprogramming provides the ability to implement some interesting features…

Other key design decisions (cont.)

Implement complex specialization options using raw metaprogramming

- Allows expert graphics programmers to explore the specialization design space

SA2019.SIGGRAPH.ORG
CONFERENCE 17-20 November 2019 - EXHIBITION 18-20 November 2019 - BCEC, Brisbane, AUSTRALIA

So we encourage expert graphics programming to use the features of staged metaprogramming directly when implementing specialization frameworks.

So let's look at a case study of something interesting we can do in Selos using staged metaprogramming…

Case Study:
Exploring the Specialization Design Space

Which is to explore the shader specialization design space. Specifically, we're going to look at specialization in a deferred renderer.

Specialization in deferred rendering

Forward Lighting Shader

```
float4 surfaceShader(…) {
…
#if defined(STANDARD)
  color = evalStandardMate
#elif defined(SUBSURFACE)
  color = evalSubsurfaceMa
#elif defined(CLOTH)
  color = evalClothMateria
#endif
…
}
```

Statically specialized variants generated at shader compile time

Dynamically branch based on material ID at shader run time

Deferred Lighting Shader

```
float4 surfaceShader(…) {
…
if(isStandardMaterial(materialID)
  color = evalStandardMaterial(…);
else if(isSubsurfaceMaterial(materialID)
  color = evalSubsurfaceMaterial(…);
else if(isClothMaterial(materialID)
  color = evalClothMaterial(…);

…
}
```

32

Here's what a material shader might look like in a forward renderer. We're using preprocessor #ifs to denote different code paths based on the type of material we're rendering, and then we can generate statically specialized variants at shader compile time for each material, one variant per material.

However, when performing shading in a deferred renderer, different pixels in the GBuffer might require different material features. So, the shader must be able to dynamically enable or disable features per-pixel at shader run time, based on material ID in this case.

Even when complete static specialization is not feasible, some specialization can still be beneficial.

Deferred lighting specialization in Uncharted 4

Generated per-tile bitmask of features needed in that tile

- E.g., This 16x16 tile contains metal and fabric

Dispatch tiles using shaders variants specialized for different feature combinations

- E.g., Render this tile with a shader that just has metal and fabric code

If all pixels in a tile are the same material, use a “branchless” variant

- E.g., This tile is all fabric, so dispatch using a fabric-only shader that omits checking materialID

33 [El Garawany 2016]

The deferred lighting pass in Uncharted 4 made use of partial specializations.

First, it splits the screen space into 16 by 16 pixel tiles and generates a per-tile bitmask of all of the material features present in that tile. For example, let’s say a given tile contains some pixels that should be rendered as metal and others that should be rendered as fabric.

Then, it dispatches tiles using different shaders, specialized for particular feature combinations. So that tile containing both metal and fabric would be rendered using a shader that just contains metal and fabric code. Code for other types of materials would be striped away from that variant.

As an optimization, if all pixels in a given tile are the same material, they dispatch it using a “branchless” variant. So if a tile is only fabric, for example, the shader can skip checking the material ID.

This design significantly improved performance for Uncharted 4. But, as you might realize, this results in many shader variants, and it turns out that…

Overspecialization can hurt performance!

Lots of shader variants!

- Increases shader switching overhead, dispatch overhead, game load time, etc.

Do we need all of these specializations?

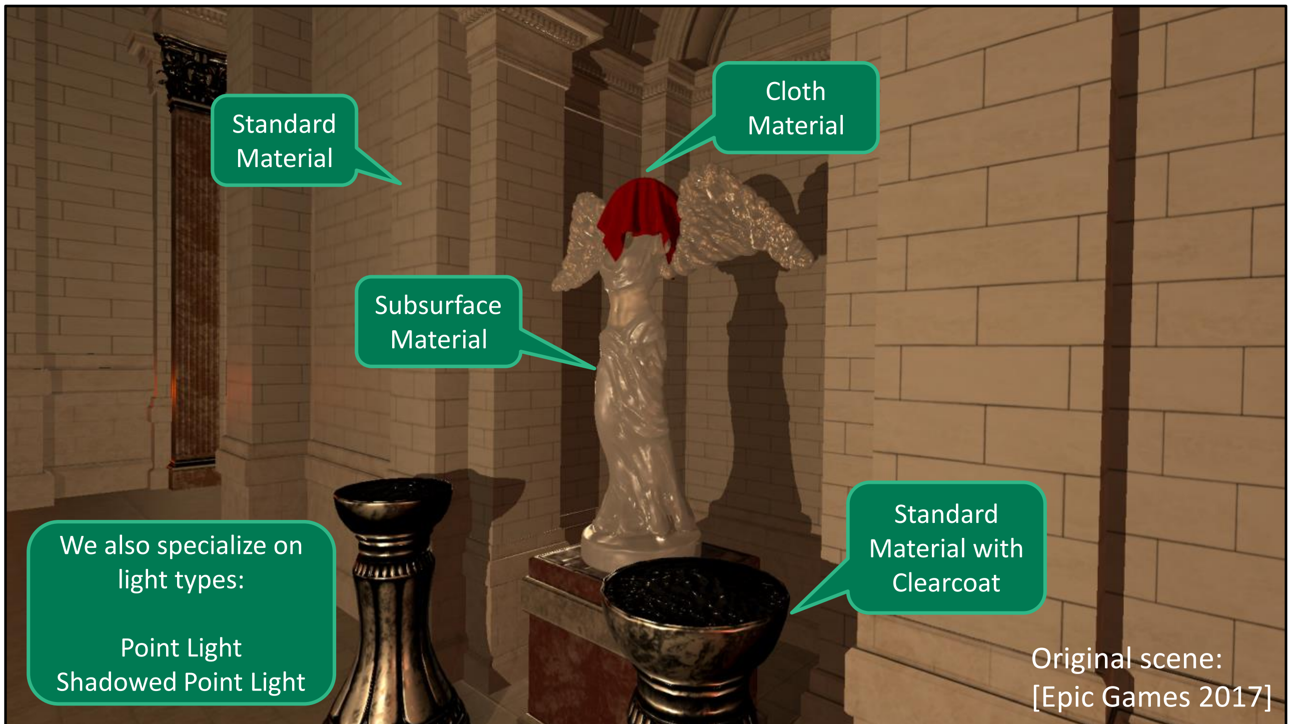
Staged metaprogramming enables exploration of this design space

Overspecialization can actually hurt performance!

Having lots of variants increases shader switching overhead, dispatch overhead, and game load time.

Do we really need all of these specialized variants? It's likely that some materials are more important to specialize than others.

We can explore this design space in Selos using staged metaprogramming.



We implemented a deferred renderer and also implemented specialization similar to Uncharted 4's for our deferred lighting shader.

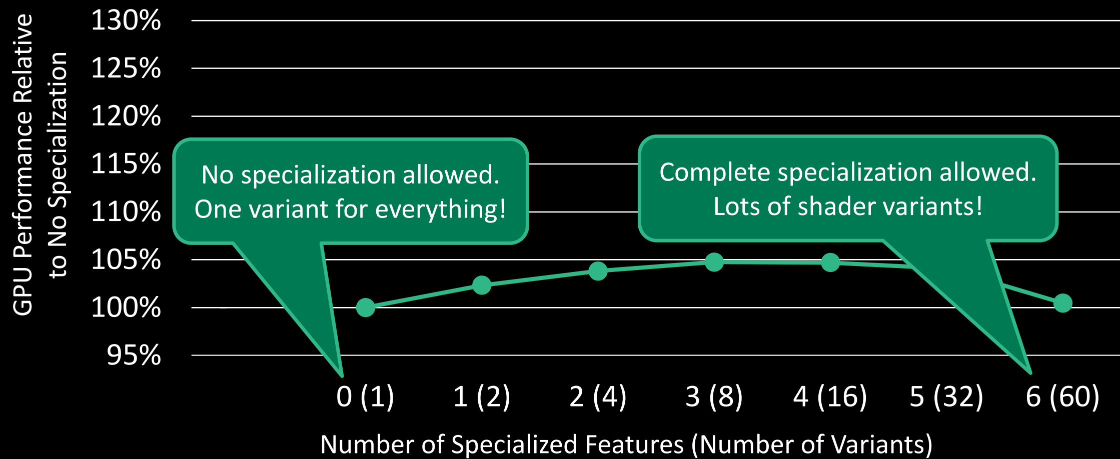
We used our system to render the Sun Temple scene from the ORCA repository, but we had to make a few modifications.

Like most other widely available scenes, this scene does not specific what type of BRDF to use for each material. So we chose to render certain objects with different BRDF types in order add some material variation. We also added some cloth geometry, which isn't in the original scene.

We also specialize based on light types by doing light culling, and then determining whether a tile does or does not contain at least one of a given light type.

This gives us six different features that we can specialize – four material features and two light features.

So we generated all of the possible variants, but then we also restricted our system to only specialize some of the material and light features.

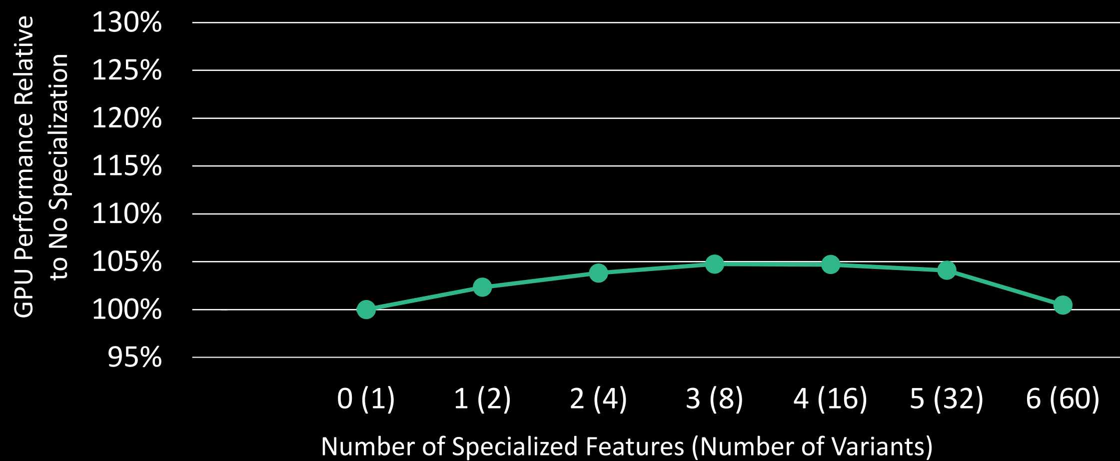


What we found is that partial specialization achieves the best performance.

This graph measures the GPU performance of our deferred lighting pass.

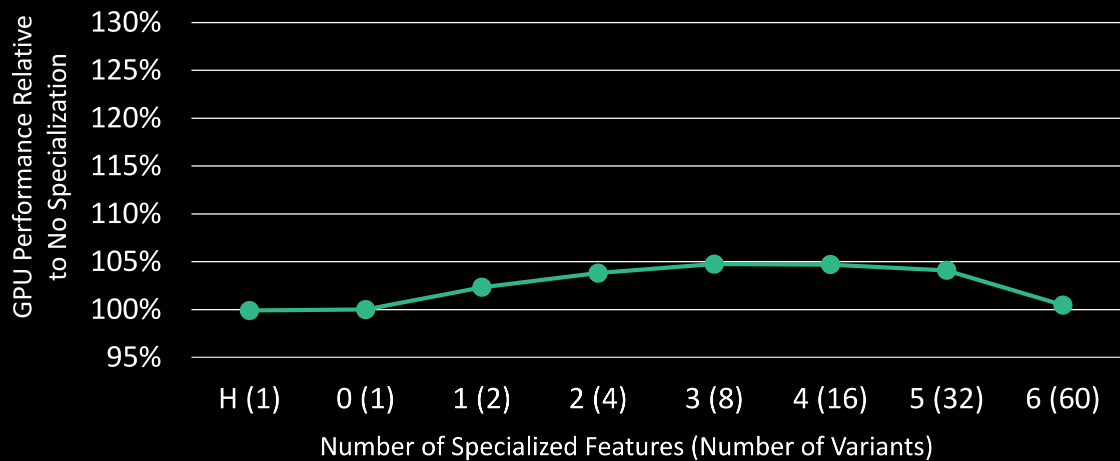
On the X axis, we have the number of material and light features we're allowing to be specialized, with the total number of generated variants in parentheses. For example, the zero case mean we're not allowing any features to be specialized, so the only shader variant is the typical deferred lighting shader, which uses dynamic branches for all feature selection. On the other end, we allow specialization for all features, which generates variants for all possible combinations of features, resulting in 60 total variants.

On the Y axis, we have the relative GPU performance compared against the baseline deferred lighting shader, which again has no specialization.



What we observe is that increasing the amount of specialization increases performance, but only to a point. Then, performance starts to degrade. So overspecialization decreases performance, and the sweet spot is in the middle.

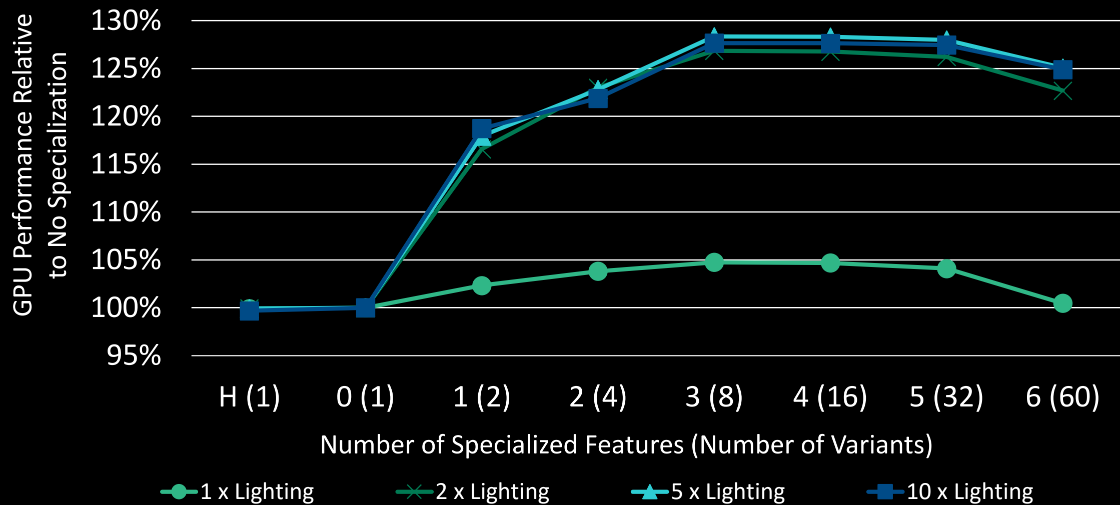
As a sanity check, we also handwrote an HLSL shader for the typical deferred shader case, to make sure our abstractions weren't adding overhead.



(As a sanity check, we also handwrote an HLSL shader for the typical deferred shader case, to make sure our abstractions weren't adding overhead.)

As you can see, it performs similarly to the version generated by our system.

Our test scene only has 14 lights, whereas game often have many more. We wanted to see how performance changes as we increases the amount of lighting computations to be more in line with modern 3D games…

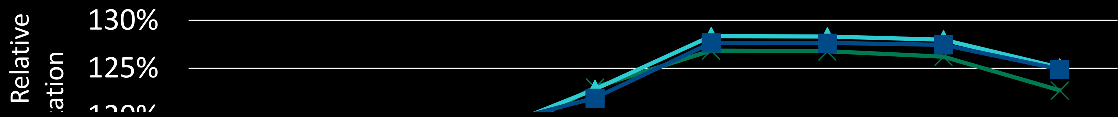


What we found is that as the amount of lighting work increases, the effects of specialization is even more pronounced. And still, partial specialization produced the best results.

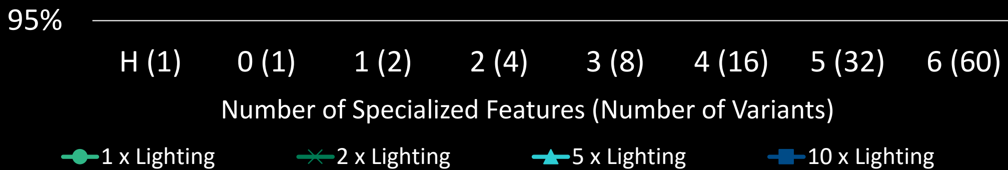
This and other types of exploration is something a shader system should help you with, and…

// Note: The reason why there are only 60 variants in the last case, rather than 64 – When enabling specialization for all 6 features, the system generates variants for all possible combinations of material and lighting features. For each feature, there's basically a choice of whether to include it in the shader or to omit it, hence there will be 2^6 = 64 variants. However, in the variants where all material features are omitted from the shader, there's no material to shade, so those shaders are effectively invalid. There are four such cases – when all of the material and light features are omitted, when only both light features are enabled, or when either one or the other light feature is enabled. So we're left with 60 total valid variants.

Partial specialization achieves the best performance



Staged metaprogramming enabled this exploration in our system



40

We could easily perform this exploration in our system because of our principled use of staged metaprogramming.

Wrapping Up

Summary

Staged metaprogramming is a key methodology for shader system development

Using it, we build the Selos shader system

- Same language for CPU & GPU code
- Statically-checked shader interface
- Performance improvements through design space exploration

We build everything in user-space code, using off-the-shelf Lua-Terra

We identified staged metaprogramming as a key methodology to aid in shader system development.

We used it to build a shader system that uses the same language for both CPU and GPU code, provides statically-checked interfaces to shaders to catch more errors earlier. And we were able to improve performance of our deferred renderer by exploring the shader specialization design space.

I want to emphasize that we build everything in user-space code, using off-the-shelf Lua-Terra. We didn't modify the Lua-Terra compiler at all.

Unfortunately, popular systems languages today don't have all the features required for staged metaprogramming, but…

The future of metaprogramming

Systems languages are trending towards better metaprogramming facilities

- Rust
- Various C++ proposals (e.g., [Sutter 2018], [Chochlik et al. 2018])
- Circle compiler for C++ [Baxter 2019]

They are trending in the right direction. Rust has some interesting features. There's various proposals about metaprogramming to the C++ committee, such as metaclasses and better support for compile-time reflection. And there's also the Circle compiler, which adds new introspection, reflection, and compile-time execution features to C++.

So hopefully in the future, the features of staged metaprogramming will be available in modern systems languages.

But beyond shader systems…

Heterogeneous programming for graphics

CUDA gives GPU compute code first-class, heterogeneous treatment in C++

How can we achieve the same for GPU graphics code?

- Many additional challenges!

I'm interested in thinking more broadly about about heterogeneous programming for graphics.

CUDA gives GPU compute code first-class, heterogeneous treatment in C++. How can we achieve the same for GPU graphics code? Graphics has many additional challenges!

Our work is a step in the right direction. We can use the same language for CPU and GPU code and provide some nice shader interfaces, but it's far from achieving true heterogeneity.

And furthermore...

Heterogeneous programming for graphics … and other domains?

Graphics is a challenging domain!

- Can we apply the lessons to other areas?

Potentially many future processor types

- We need programming models to support them

Staged metaprogramming allowed us to add support for a different processor type (GPU) purely as library code

- No fundamental language changes → don't have to involve standard bodies

What about other domains?

Graphics provides a complex and well-explored area in which to investigate the broader concept of heterogeneity. Can we apply the lessons we learn about graphics programming in other domains?

In a future with potentially many different processor types, we need programming models to support them.

What I think is really interesting is that staged metaprogramming allowed us to add support for a different processor type purely as library code. We didn't have to modify Terra at all. And I think that's a very powerful property of staged metaprogramming.

Acknowledgments

Discussions and advice

- Anjul Patney, Ahmed Mahmoud, Alex Kennedy, Angelo Pesce, Aras Pranckevičius, Brett Lajzer, Brian Karis, Chuck Lingle Dave Shreiner, Hugues Labbe, Joe Forte, Michael Vance, Padraic Hennessy, Paul Lalonde, Wade Brainerd, Zachary DeVito, and the reviewers

Feedback on the presentation

- Owens Group Members

Early code contributions

- Francois Demoullin

Financial Support

- Intel Corporation, National Science Foundation Graduate Research Fellowship Program, NVIDIA

There's many people I would like to thank for discussions and advice, feedback on the presentation, early code contributions, and financial support.

Thank you

Kerry A. Seitz, Jr.
kaseitz@ucdavis.edu
github.com/kseitz/selos

sa2019.siggraph.org

And also the source code is available on GitHub.

Thank you for your attention!

References

Sean Baxter. 2019. Circle. https://github.com/seanbaxter/circle

Matus Chochlik, Axel Naumann, and David Sankel. 2018. Static reflection. C++ Standards Committee Papers. http://wg21.link/p0194

Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A Multi-Stage Language for High-Performance Computing. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13). 105–116. https://doi.org/10.1145/2491956.2462166

Ramy El Garawany. 2016. Advances in Real-time Rendering, Part I: Deferred Lighting in Uncharted 4. In ACM SIGGRAPH 2016 Courses (SIGGRAPH '16). http://advances.realtimerendering.com/s2016/index.html

References (cont.)

Conal Elliott. 2004. Programming Graphics Processors Functionally. In Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell (Haskell ’04). 45–56. https://doi.org/10.1145/1017472.1017482

Epic Games. 2017. Unreal Engine Sun Temple, Open Research Content Archive (ORCA). https://developer.nvidia.com/ue4-sun-temple

Yong He, Kayvon Fatahalian, and Tim Foley. 2018. Slang: Language Mechanisms for Extensible Real-time Shading Systems. ACM Transactions on Graphics 37, 4, Article 141 (July 2018), 13 pages. https://doi.org/10.1145/3197517.3201380

Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. 2002. Shader Metaprogramming. In Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS ’02). 57–68. http://dl.acm.org/citation.cfm?id=569046.569055

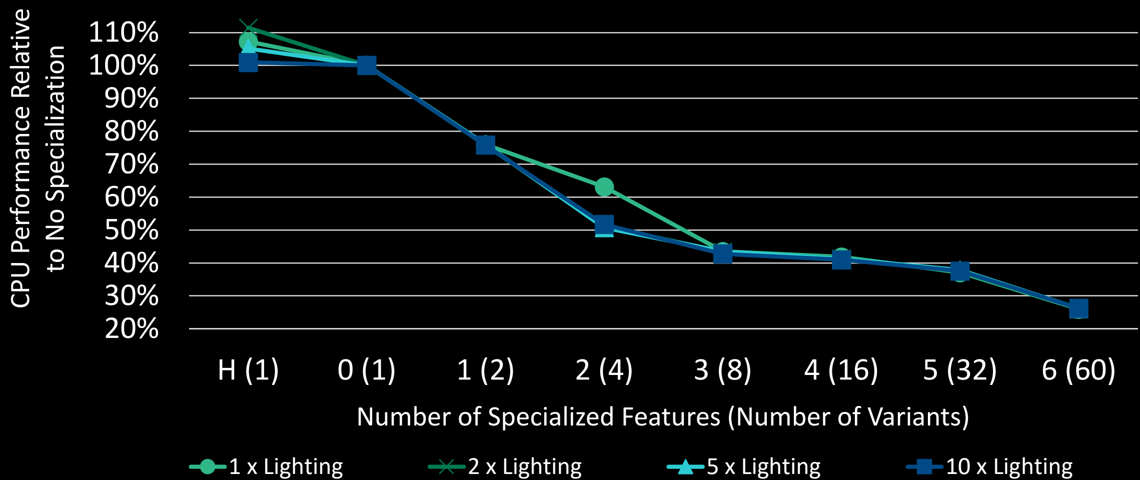
References (cont.)

Herb Sutter. 2018. Metaclasses: Generative C++. C++ Standards Committee Papers. https://wg21.link/P0707

Walid Mohamed Taha. 1999. Multistage Programming: Its Theory and Applications. Ph.D. Dissertation. Oregon Graduate Institute of Science and Technology.

Natalya Tatarchuk and Chris Tchou. 2017. Destiny Shader Pipeline. Game Developers Conference 2017. http://advances.realtimerendering.com/destiny/gdc_2017/

Extra Slides



More specialization means more variants. So, as we expected, there is more CPU overhead needed to bind the variants and dispatch tiles. So, CPU performance decreases with increased specialization.