



SIGGRAPH 2026
Los Angeles 19-23 JUL

The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques

Hands-On Course

Introduction to Slang

The Next-Generation Shading Language

Nia Bickford and Chris Hebert
NVIDIA



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.



Hello and welcome to our Slang workshop!

In this lab session, Chris Hebert and I are going to show you how to use Slang, the next-generation shading language.

Since this lab is interactive, we'll have you write and compile Slang programs on your own computer, and give you a few coding puzzles to solve.

Through this, you'll learn how Slang can help you write fast, cross-platform shaders, and we'll even cover some advanced language features like modules and autodifferentiation.

PRESENTER SETUP:

- Silence all notifications
- Clear desktop; place `vk_slang_editor` and other resources in the same locations as on the Docker image.
- Launch a text editor. Load in the file containing command lines for the slangc lab. In another tab, load in the GLSL file for the GLSL -> Slang demo. Switch back to the slangc tab and minimize the editor.
- Launch RenderDoc and fill in the path to `simple_polygons` (compiled for Vulkan 1.3), but don't capture it yet. Minimize RenderDoc.
- Open Slang Playground in a browser window; minimize the browser window.
- Open Visual Studio Code with both of Chris's notebooks. Minimize VS Code.

If You Know HLSL, You Can Write Slang

SIGGRAPH



But you might find that if you know a shading language like GLSL or HLSL, you can probably write Slang already!



```
RWTexture2D<float4> texFrame;

[shader("compute")]
[numthreads(16, 16, 1)]
void clear(uint2 thread: SV_DispatchThreadID)
{
    float4 color = float4(0.1, 0.75, 0.8, 1.0);
    texFrame[thread] = color;
}
```

Here's an example of a shader that clears an image, `texFrame`, to a solid color. Let's walk through it, line by line, and you'll probably recognize the parts.



```
RWTexture2D<float4> texFrame;  
  
[shader("compute")]  
[numthreads(16, 16, 1)]  
void clear(uint2 thread: SV_DispatchThreadID)  
{  
    float4 color = float4(0.1, 0.75, 0.8, 1.0);  
    texFrame[thread] = color;  
}
```

First, we define a 2D texture, named `texFrame`. The “RW” stands for “read” and “write”, and we’ll write RGBA colors to it using vectors of 4 floats.



```
RWTexture2D<float4> texFrame;

[shader("compute")]
[numthreads(16, 16, 1)]
void clear(uint2 thread: SV_DispatchThreadID)
{
    float4 color = float4(0.1, 0.75, 0.8, 1.0);
    texFrame[thread] = color;
}
```

Then we define the shader entrypoint, which is a function. Above it, we have two *attributes*.



```
RWTexture2D<float4> texFrame;  
  
[shader("compute")]  
[numthreads(16, 16, 1)]  
void clear(uint2 thread: SV_DispatchThreadID)  
{  
    float4 color = float4(0.1, 0.75, 0.8, 1.0);  
    texFrame[thread] = color;  
}
```

This first one says it's a compute shader,



```
RWTexture2D<float4> texFrame;  
  
[shader("compute")]  
[numthreads(16, 16, 1)]  
void clear(uint2 thread: SV_DispatchThreadID)  
{  
    float4 color = float4(0.1, 0.75, 0.8, 1.0);  
    texFrame[thread] = color;  
}
```

and this one says the compute shader covers the image using threads in 2D blocks of 16x16.



```
RWTexture2D<float4> texFrame;  
  
[shader("compute")]  
[numthreads(16, 16, 1)]  
→ void clear(uint2 thread: SV_DispatchThreadID)  
{  
    float4 color = float4(0.1, 0.75, 0.8, 1.0);  
    texFrame[thread] = color;  
}
```

Our shader's input is the index of the thread in the compute shader dispatch. It's a vector of 2 unsigned integers, x and y. That's the pixel we're writing to.



```
RWTexture2D<float4> texFrame;  
  
[shader("compute")]  
[numthreads(16, 16, 1)]  
void clear(uint2 thread: SV_DispatchThreadID)  
{  
    float4 color = float4(0.1, 0.75, 0.8, 1.0);  
    texFrame[thread] = color;  
}
```

Then we define a variable called `color`, which is a vector of 4 floats. We set it to an RGBA color.



```
RWTexture2D<float4> texFrame;

[shader("compute")]
[numthreads(16, 16, 1)]
void clear(uint2 thread: SV_DispatchThreadID)
{
    float4 color = float4(0.1, 0.75, 0.8, 1.0);
    texFrame[thread] = color;
}
```

And finally, we write it into the texture at the given pixel.



```
RWTexture2D<float4> texFrame;  
  
→ [shader("compute")]  
   [numthreads(16, 16, 1)]  
   void clear(uint2 thread: SV_DispatchThreadID)  
   {  
       float4 color = float4(0.1, 0.75, 0.8, 1.0);  
       texFrame[thread] = color;  
   }
```

If you've programmed shaders before, probably the only new thing here is this [shader("compute")] attribute. This allows you to define multiple shaders in a single file. The rest probably looks pretty familiar!



```
RWTexture2D<float4> texFrame;

[shader("compute")]
[numthreads(16, 16, 1)]
void clear(uint2 thread: SV_DispatchThreadID)
{
    float4 color = float4(0.1, 0.75, 0.8, 1.0);
    texFrame[thread] = color;
}
```

In exchange...

In exchange for compiling this with Slang, you get a *lot* of features.



Slang lets you compile your code for Vulkan, DirectX, OpenGL, Apple Metal, WebGPU, PyTorch, CUDA, OptiX, and even C++ code so you can run it on the GPU or on the CPU.

And it lets you use the *full* capabilities of each target – for instance, WebGPU doesn't support ray tracing, but Slang will let you use it when you're targeting Vulkan, DirectX, or Metal.

More Language Features

SIGGRAPH



- **Shader reflection**

- **Auto-differentiation**

- **Helps structure large codebases**

- Modules
- Interfaces
- Generics, like templates
- Specialization

- **Helpful type features:**

<i>Pointers</i>	<i>Bindless</i>	<i>ParameterBlock<T></i>
<i>Properties</i>	<i>Operators</i>	<i>Optional<T></i>
<i>Tuples</i>	<i>Lambdas</i>	<i>Cooperative vectors</i>
<i>Interfaces</i>	<i>Exceptions</i>	<i>and more</i>

- **15+ more years of wisdom in shader language design**

- **Runtime performance can meet or beat target-specific code**

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

14

And it gives you advanced language features to solve the needs of today's graphics developers.

It has shader reflection, for getting info about shaders.

Auto-differentiation, which Chris will talk about.

It helps you structure large codebases using modules, interfaces, and generics, as well as **specialization** to help with compile times;

types like pointers and parameter buffers that make shader I/O easier, properties for structs, which even C++ doesn't have, and many more type features to help you write shaders.

In sum, it's a shader language designed with 15+ more years of wisdom in how to create languages for computer graphics,

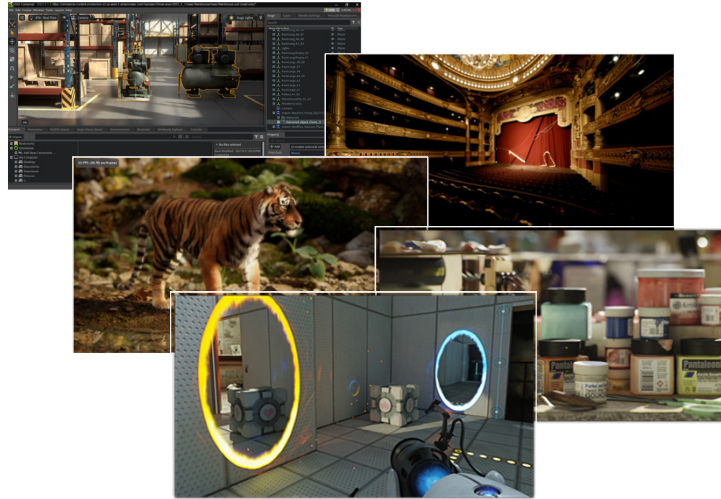
and its run-time shader performance can meet or beat handwritten code, even when using advanced features like generics.

Slang in Use

SIGGRAPH



- Open source
- Open governance
- Used in production:



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

15

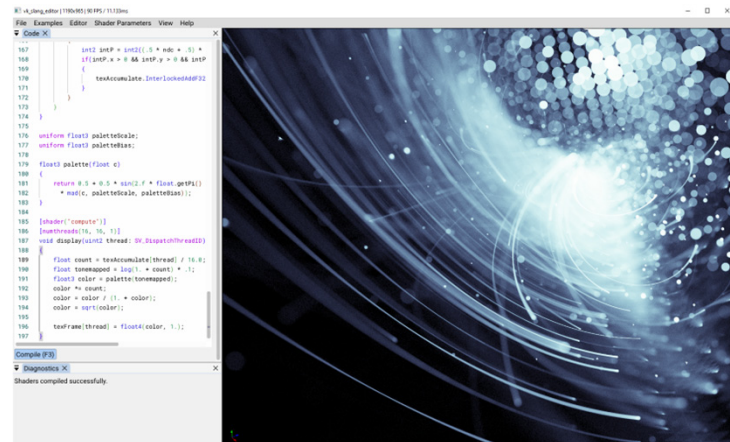
Slang is also open-source, and an open-governance project under the Khronos Group. And it's used in production in major engines. Valve ported their Source 2 engine to it. Autodesk Aurora uses Slang to render on many different platforms. And Slang powers NVIDIA Omniverse and Portal with RTX.

Agenda

SIGGRAPH



- Language Basics
- Using slangc
- Porting GLSL
- Shader I/O
- Debugging and Tools
- Structs, Modules, and Interfaces
- SlangPy
- Automatic differentiation



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

16

In today's session, we'll get you coding shaders as quickly as possible with an interactive tool.

Then we'll drop down to the command line to show you how to use the slangc compiler directly.

If you're approaching Slang from GLSL, we'll talk about what's involved in porting your code to Slang.

Passing data to and from shaders is an important topic, and one that Slang really innovates in.

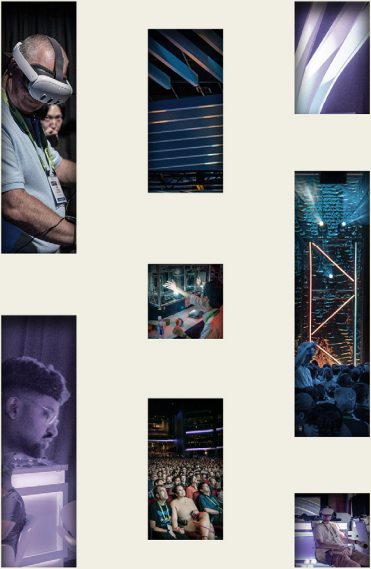
Then we'll go over debuggers and other tools and how they work with Slang.

We'll get into some advanced language features in the structs and modules section.

Then Chris will talk about SlangPy, which is really interesting for researchers, and finally one of Slang's standout features – its ability to automatically generate forwards and backwards derivatives of code.

Let's get started!

The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques



 **SIGGRAPH 2026**
Los Angeles 19-23 JUL

Lab: Language Basics

© 2026 SIGGRAPH. ALL RIGHTS RESERVED. 

Switch to desktop + notes; next slide will be “Experiment” with Mandelbrot



Windows or Linux

(Windows 10+, Ubuntu 22.04, or similar)

Download this .zip file:

<https://developer.download.nvidia.com/ProGraphics/nvpro-samples/SlangLab2026/Lab.zip>

Extract it, then enter the folder for your OS and run `vk_slang_editor`.

Web

(requires WebGPU: Chrome 113+, Edge 113+, Firefox 141+, or Safari 26+)

Open this URL:

<https://shader-slang.org/slang-playground/>

Note that some examples will need porting to the way Slang Playground works.

Build from Source

You will need Git, CMake, and a C++20 compiler.
To download and build the editor, run this:

```
git clone --recursive https://github.com/nvpro-samples/vk_slang_editor.git
```

```
cd vk_slang_editor
```

```
mkdir build
```

```
cd build
```

```
cmake ..
```

```
cmake --build . --config Release --parallel
```

Then run `_bin/<config>/vk_slang_editor`.

Lab: Getting Started

SIGGRAPH



- This one's intended to get participants coding with Slang as soon as we can. (The previous intro/motivation clocks in at about 5 minutes of presentation.)
- Have them open up `vk_slang_editor`; show where to find it on the filesystem.
- They'll see this code:

A screenshot of the vk_slang_editor application. On the left, a code editor window titled 'Code' shows Slang code. On the right, a preview window displays the rendered output of the code, which is a smooth color gradient from blue on the left to red on the right.

```
1 RWTexture2D<float4> texFrame; // Output texture
2 uniform float iTime; // In seconds
3 uniform float2 iResolution; // Screen size
4
5 [shader("compute")]
6 [numthreads(16, 16, 1)]
7 void main(uint2 thread: SV_DispatchThreadID)
8 {
9     float2 uv = float2(thread) / iResolution;
10    float4 color = float4(uv, 0.5 + 0.5 * sin(iTime), 1.0);
11    texFrame[thread] = color;
12 }
13
```

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

19

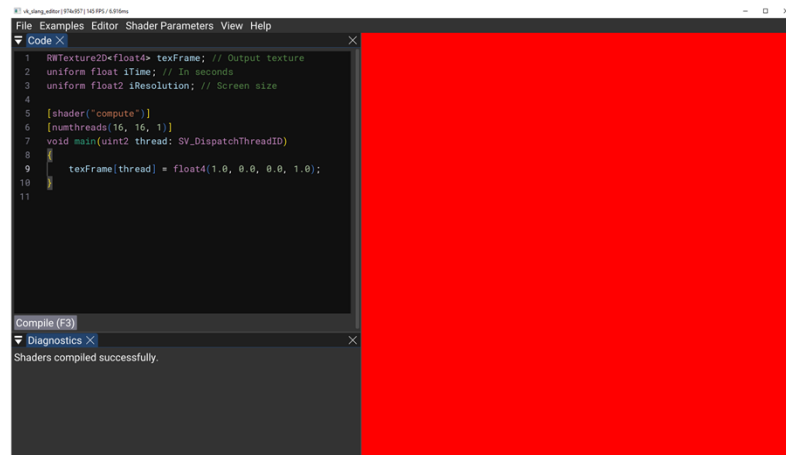
We've loaded a dev environment onto each of the systems in this room. Open up the file explorer by clicking on – and then double-click on `vk_slang_editor` to open it. You should see a gradient, like this:

Lab: Getting Started

SIGGRAPH



- Clear it to a solid color:



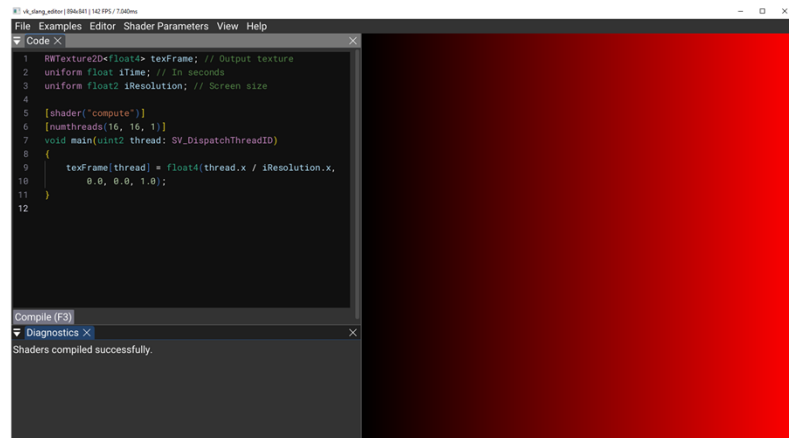
Let's start with something simple. Select the content of the main function, in between the curly braces, and delete it. Let's set all pixels of texFrame to a solid color. Etc. Then press F3 or click on the "Compile" button to compile. You should get a red screen, like this.

Lab: Getting Started

SIGGRAPH



- Draw a gradient:



Now let's draw a gradient. For this, we'll take the x component of the thread index, which ranges from 0 to the screen width minus 1, and divide it by the screen width to get a value between 0 and 1, Put it in the red channel, like this:
(Intentionally make a mistake here to show diagnostics)

Lab: Getting Started

SIGGRAPH



- Intentionally make a mistake to show diagnostics:

```
1 RWTexture2D<float4> texFrame; // Output texture
2 uniform float iTime; // In seconds
3 uniform float2 iResolution; // Screen size
4
5 [shader("compute")]
6 [numthreads(16, 16, 1)]
7 void main(uint2 thread: SV_DispatchThreadID)
8
9     float2 uv = float(thread) / iResolution;
10    texFrame[thread] = float4(uv, 0.0, 1.0);
11
12
```

Compile (F3)

Diagnostics

Error, line 9: expected an expression of type 'float', got 'vector<uint,2>'

float2 uv = float(thread) / iResolution;

(error code 38019)

Error, line 9: expected a function, got 'typeof(float)'

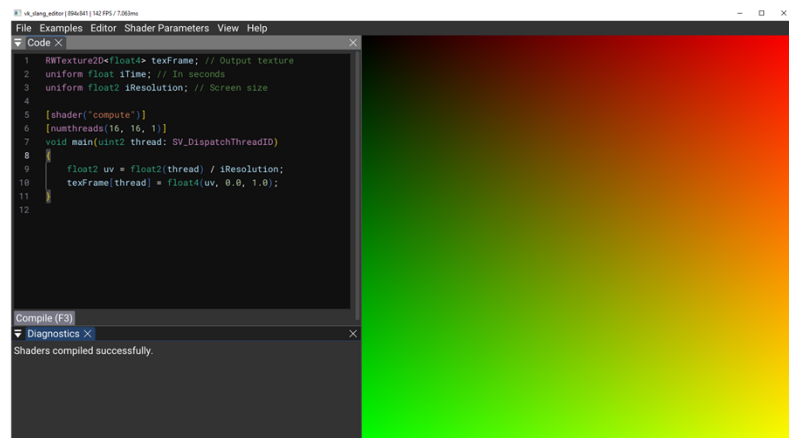
float2 uv = float(thread) / iResolution;

Lab: Getting Started

SIGGRAPH



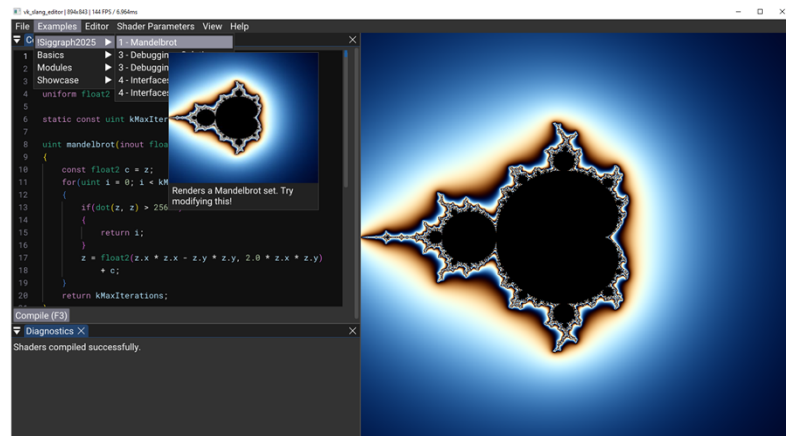
- 2D gradient and `float4(float2, float, float)` constructor:



Now let's extend this to 2 dimensions. Let's create a `float2` variable called `uv` and set it to the thread divided by the resolution.



- Load Mandelbrot set example:



Now let's skip ahead a bit to show off some more standard features. In the menu bar, click on Examples > !SIGGRAPH > 1 – Mandelbrot.

Point out:

- Compile-time constant (static const)
- inout parameter (copy in, copy out)
- For loop
- If statement
- Vector math
- Color palette

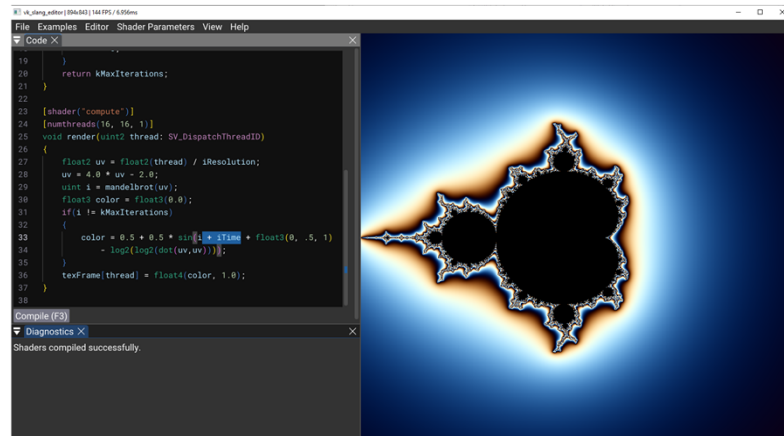
Lab: Getting Started

SIGGRAPH



- Use uniform float `iTime` to cycle colors:

(We are about to go back to the slides.)



Up above, you can see this shader defines several shader parameters with the `uniform` qualifier. One of these is `iTime`, which is a float and is the time in seconds since the shader started running. Let's use that to cycle the color palette.

Experiment!

SIGGRAPH

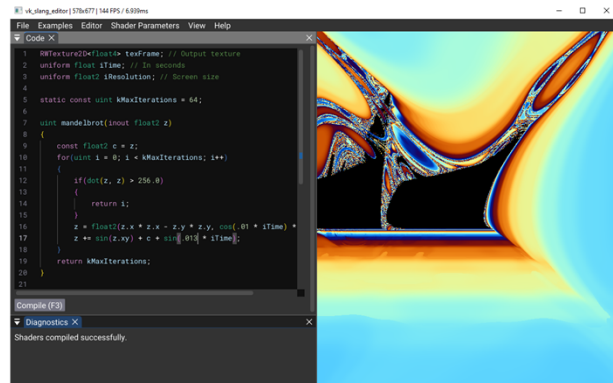


- **Modify the shader**

- Can you use `iTime` in other places?
- Can you use `iMouse`?
- Add uniform `float3 eye;` to the top and use it in your shader. Then click and drag on the viewport. What happens?
- Try to be fearless with your changes. It will autosave.

- **Stuck?**

- Reload *Examples > SIGGRAPH > Mandelbrot*



Try this on line 9!

```
const float2 c = iMouse.xy/iResolution;
```

Now, I'd like you to modify the shader on your own! The goal of this exercise is to get you practice with Slang by making changes to the code, compiling, and seeing the result. So I'd like you to try breaking the shader, changing the control flow, modifying the equations and experimentally seeing what you get.

I've put a few ideas up here. Try using the `iTime` or `iMouse` uniforms, or if you're feeling adventurous, try this third idea here.

Try to be fearless with your changes and break things; it will autosave next to the executable before every compile, so you won't lose anything.

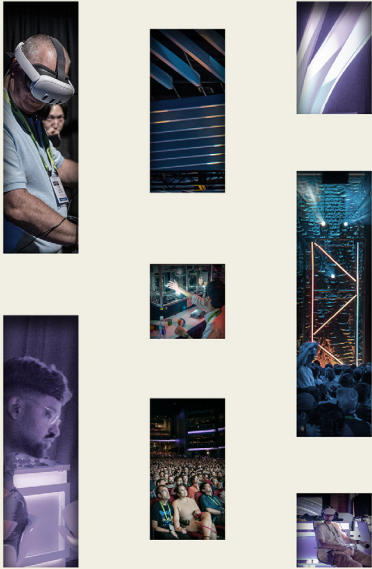
And if things aren't compiling and you're stuck, you can re-load the example. Or ask one of the TAs we have to help out.

I'll return in about 5 minutes to introduce the next section.

The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques



Using slangc



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.





- vk_slang_editor uses the Slang shared library to compile shaders for Vulkan:



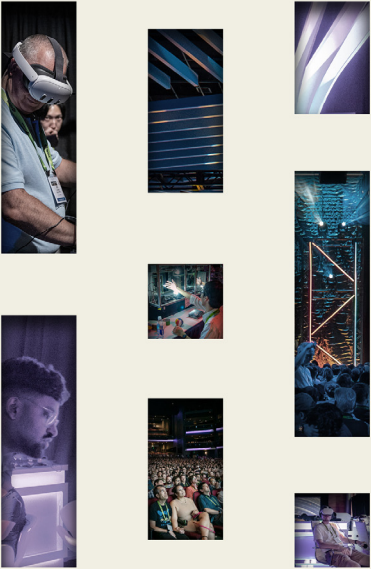
- Let's compile to different targets using the Slang command-line, slangc

My goal here is to show you how to work with the bare metal, in a sense. The tool you've been using doesn't do much* beyond taking your shader, passing it to the Slang library to compile it to Vulkan's SPIR-V intermediate representation, and then rendering with it.

Now let's compile to targets other than Vulkan by using the Slang command line, slangc.

**At least on the rendering side; readers who have seen the source code may note that it also works with reflection info, discussed in the next section, and spends quite a bit of code implementing an IDE.*

The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques



 **SIGGRAPH 2026**
Los Angeles 19-23 JUL

Lab: Compiling with slangc

© 2026 SIGGRAPH. ALL RIGHTS RESERVED. 

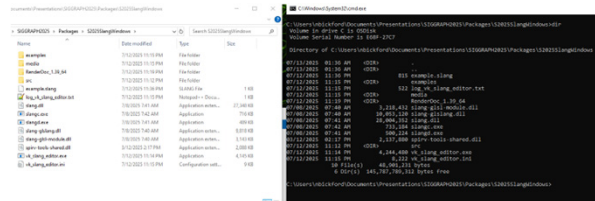
Switch to desktop + notes; next slide will be multiple inputs/outputs for Slang

slang Lab: Entering the Terminal

SIGGRAPH



- Switch to desktop
- In `vk_slang_editor`, save your shader in the same folder as `vk_slang_editor`
 - Or you can use `example.slang`, which I've created for you
- Open a command line in that folder
- Type `dir` to make sure you're in the right one





- Run `slangc -v`
- Let's compile to GLSL!
 - Run `slangc example.slang -target glsl`
 - Point out a few things
 - `#version 450`, so it's GLSL
 - uniforms and buffers are marked with a matrix layout; we'll talk about that later
 - Line directives
 - `RWTexture2D<float4>` became `layout(rgba32f) uniform image2D`
 - Slang automatically assigned bindings
 - Global shader parameters were moved into a constant buffer
 - `std140` layout
 - Slang transforms the code as a compiler would, but tries to keep the output readable.

slangc Lab: Getting Help

SIGGRAPH



- Run `slangc -h`
- Direct attention to the options for `-target`.
- Run `slangc -h target`.

slangc Lab: Other Targets

SIGGRAPH



- Run `slangc example.slang -target spirv`
 - Output to a file using `-o`
 - Open in a hex editor
- Just to show we can do other targets:
 - Run `slangc example.slang -target wgl`
 - Run `slangc example.slang -target metal`
 - Bindings now declared as function parameters

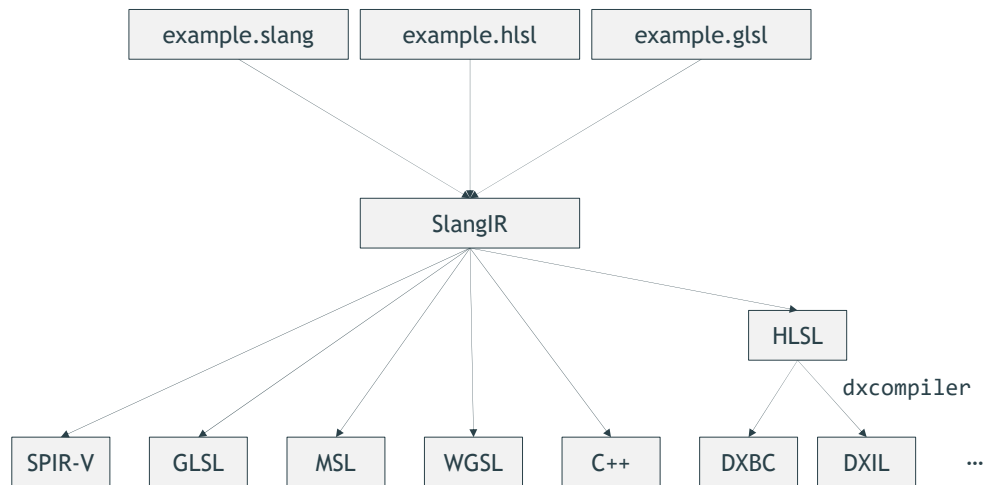


(this one is Windows-only!)

- Run slangc example.slang -target dxil
- That fails; we need to specify a *profile*.
 - slangc example.slang -target dxil -profile sm_6_6
 - Profiles enable capabilities; which shader features you can use.
 - For example, ray tracing is available in HLSL but not WGSL.
 - Warp operations aren't available in C++ code.
 - Advanced SPIR-V functionality depends on your GPU and driver.
 - Capabilities allow you to check that you're only using the features you know your platform supports.

How Does Slang Work?

SIGGRAPH



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

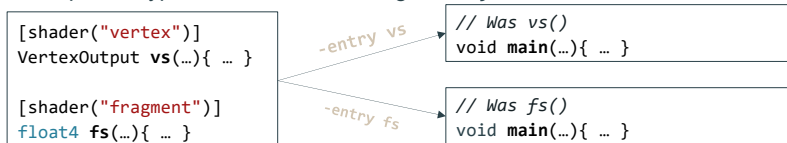
35

As you might have noticed, some of the outputs are very different than their inputs. Some studios use preprocessor-based systems to support multiple platforms, but for things like SPIR-V or WGSL, you really do need a full compiler with its parsing and codegen systems.

As a compiler, Slang works by taking the input and transforming to an intermediate representation called SlangIR. All its optimization and other passes operate on that IR, and then it outputs code for each target below. For some targets like DXIL, it'll output HLSL and then call a downstream compiler like DirectXShaderCompiler.



- Slang renames the entrypoint to “main”
- GLSL: If you have multiple entrypoints, select one using `-entry <name>`:



- SPIR-V: Can keep entrypoint names using `-fvk-use-entrypoint-name`:



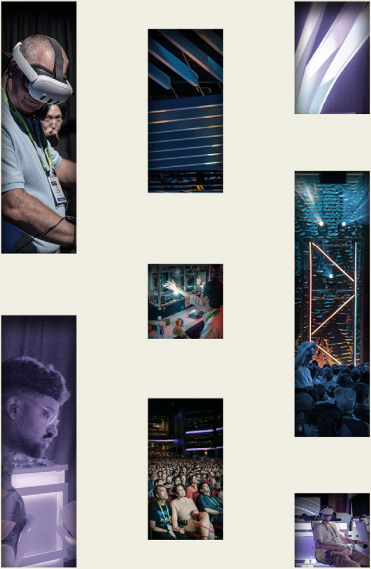
Now let's talk about the specifics of some targets you may have seen.

Earlier in our GLSL example, Slang renamed the entrypoint to “main”. This is because OpenGL only allows you to have one entrypoint, which must be named “main”, since GLSL doesn't have ways to mark functions as entrypoints.

So, if a Slang file contains multiple entrypoints, you have to select one at a time using the “-entry” argument.

(click) Vulkan and SPIR-V allow you to have multiple entrypoints, though. So, for that, I recommend using the `-fvk-use-entrypoint-name` argument to keep the original entrypoint names in the output.

The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques



 **SIGGRAPH 2026**
Los Angeles 19-23 JUL

Porting GLSL

© 2026 SIGGRAPH. ALL RIGHTS RESERVED. 

Slang also has features for taking GLSL-like code as input. Suppose you're a developer with, say, tens of thousands of lines of GLSL code, and you'd like to make use of some of Slang's features like reflection.

Almost Completely Works Out of the Box



- Make sure your shader has a `#version` directive
- Mark your entrypoints with `[shader]` attributes
- Domain, hull, and mesh shader attributes must be ported to HLSL equivalents, e.g.
 - `layout(vertices = 3) out;` → `[outputcontrolpoints(3)]`
 - `gl_TessLevelOuter` → `SV_TessFactor`
- That's all!
- Docs: <https://docs.shader-slang.org/en/latest/coming-from-glsl.html>

```
#version 450
#extension GL_EXT_shader_image_load_formatted : require

layout (local_size_x = 16, local_size_y = 16, local_size_z = 1) in;

// Uniforms
layout(binding=1) uniform image2D texframe;
layout(binding=2) uniform image2D teximg;
layout(binding=3) uniform image2D teximg;
layout(binding=4) uniform sampler2D texsmap;
uniform vec2 iResolution;
uniform float iTime;

#define PI 3.1415926535
#define MATTE_Y 0.39

[shader("compute")] // HLSL -> GLSL: Add entrypoint attribute
void mainImage()
{
    uvec2 thread = gl_GlobalInvocationID.xy;
    if (any(greaterThan(thread, iResolution))) return;
    vec2 fragCoord = vec2(thread);
    fragCoord.y = iResolution.y - fragCoord.y;

    vec2 uv = 2.0*(fragCoord.xy-0.5*iResolution.xy) / iResolution.x;

    // Matte to widescreen
    // No 11 matte again in the final pass;
    // this just saves some processing time
    vec2 uv2 = 2.0*(fragCoord-0.5*iResolution.xy)/iResolution.x;
    if (abs(uv2.y) > MATTE_Y)
    {
        imageStore(texframe, ivec2(thread), vec4(0.0));
    }
}
```

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

The good news is, Slang can compile GLSL shaders with almost no modification! You only have to do a few small things.

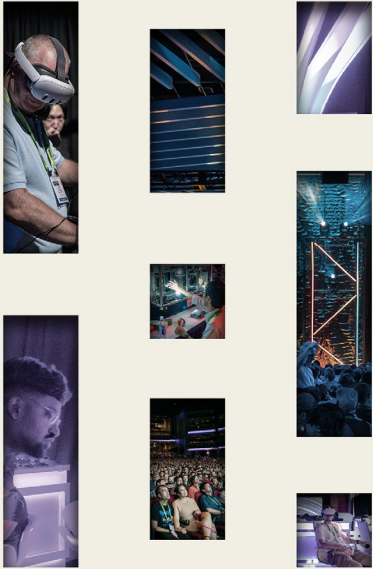
First off, make sure your shader uses the GLSL `#version` directive somewhere. This tells Slang to load in its GLSL compatibility module.

Then, you need to mark your entrypoints with `[shader]` attributes to tell Slang which functions are entrypoints for which shader types.

Most attributes are handled automatically, but if you're porting a domain, hull, or mesh shader, which is less common, you have to translate those specific GLSL attributes to HLSL equivalents. It's usually straightforward.

And then you're done! That's all you have to do.

The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques



 **SIGGRAPH 2026**
Los Angeles 19-23 JUL

Demo: Porting GLSL

© 2026 SIGGRAPH. ALL RIGHTS RESERVED. 

Here I'll show this process in action by porting a shader live. [...]

Demo

SIGGRAPH



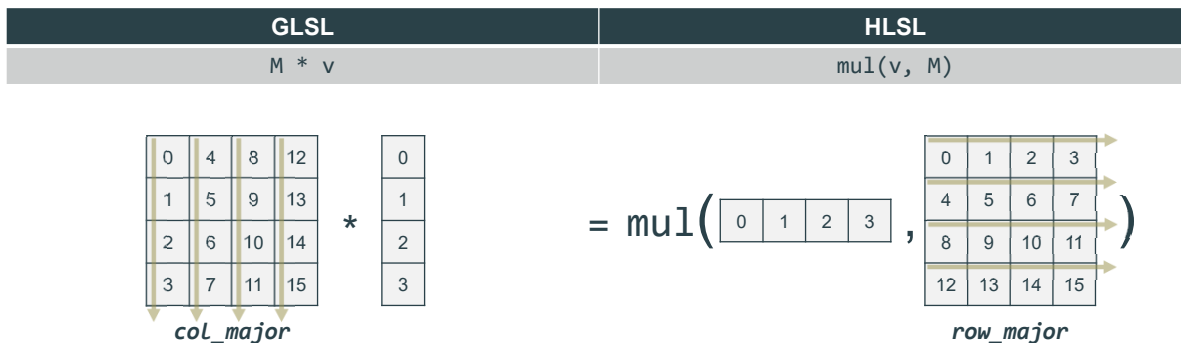
- *Goal is to prove that Slang really can do what was just described; to be done on presenter's screen.*
- Open a GLSL shader in `vk_slang_editor`
- Add shader attributes to entrypoints
- It compiles!

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

Here I'll show this process in action by porting a shader live. [...]

Matrix Layouts in GLSL vs. HLSL

SIGGRAPH



Slang translates for you: $M * v \rightarrow \text{mul}(v, M)$

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

41

One thing you may know about is that GLSL and HLSL typically have opposite matrix layouts and conventions for the order in which you multiply vectors and matrices.

In GLSL, matrices are column-major, and matrices usually appear on the left.

But in HLSL, matrices are *row-major*, and matrices usually appear on the *right*.

(click) The good news is that Slang's GLSL compatibility layer automatically handles this transposition for you: it'll translate GLSL $M * v$ to HLSL $\text{mul}(v, M)$.

Recommendation

SIGGRAPH



- **Use row-major layout** (-matrix-layout-row-major)
 - **No change needed to your GLSL code**
 - Don't spend time flipping the order or transposing your matrices
 - Using GLM or DirectXMath? memcpy
 - Natural to view in debugger
 - Slightly better access pattern
- **Whichever layout you choose, always specify it in the compiler flags**
 - Slang's biggest pitfall: slangc defaults to column-major, libslang defaults to row-major
 - Slangc expected to default to row-major in the future
 - SPIR-V annotation will appear opposite what you specify; Slang row-major/SPIR-V ColMajor has the better access pattern.

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

42

Now, Slang also lets you choose whether your matrices are stored row-major or column-major, and this can be a bit of a braintwister. My team at work was involved in porting several codebases from GLSL to Slang and went back and forth on different approaches. Based on that, we have a recommendation: **specify row-major layout**, and then don't mess with your code.

Because Slang's GLSL compatibility layer swaps the order for you, row-major CPU matrices, will show up as row-major HLSL matrices, and as column-major GLSL matrices. This is correct in both APIs.

So, if your GLSL matrix math was working before, it'll work now.

If you're using the GLM or DirectXMath libraries, you can memcpy the matrices to memory. No need for transposition.

And row-major CPU matrices are natural to view in debuggers, and have slightly better access patterns for GPUs.

So basically, specify row-major, and then don't touch your code.

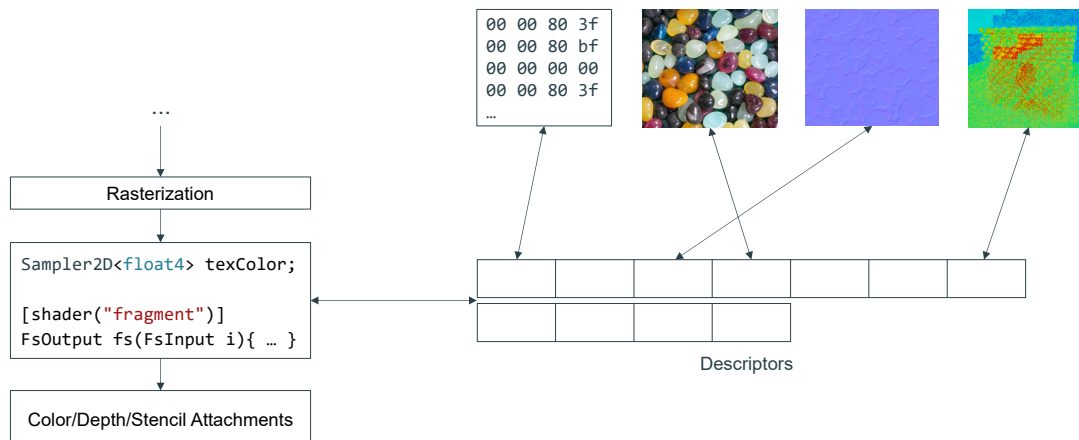
(click) The second most important thing is to *always* specify the matrix layout in the compiler flags. **(click)** Slang's biggest pitfall at the moment in my opinion is that the library and the executable compilers default to different layouts. The Slang team is hoping to make these both default to row-major in the future, but for now it's best to always specify it.

(Finally, earlier when we compiled to GLSL, you may have noticed that GLSL listed a row-major attribute, even though the command-line defaulted to column-major. This is because of that swap I mentioned in the last slide: the GLSL and SPIR-V layout will appear as the opposite of the Slang layout. Don't worry though: CPU row-major / SPIR-V ColMajor is still the better access pattern.)

(This slide assumes you're using scalar layout or are not copying to Slang matrices with at least 2 rows and 3 elements per row; in that case, for memcpy to work the padding has to match.)



This connects to a larger topic: shader input and output.



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

Most shaders operate on data in some way. E.g. a fragment shader might load material properties from one buffer, sample textures, and finally output a color to a framebuffer.

Typically, the way this works is resources like buffers and textures exist in video RAM somewhere. An app writes descriptors, which are like pointers with some extra info, into descriptor sets or descriptor heaps, and then binds these descriptor sets or heaps to the rendering pipeline.

The part we'll be talking about today is how we can write Slang code that accesses resources.



```
Texture2D<float4> tex;
RWTexture2D<float> weights;
Texture1D, Texture3D, TextureCube
Texture2DArray, Texture2DMSArray,
FeedbackTexture2D, ...

SamplerState sampler;

float4 rgba = tex.Load(uint3(x, y, mip));
weights[uint2(x, y)] = 1.0;

float4 rgba = tex.Sample(sampler, uv);

New!
Sampler2D<float4> combined;

float4 rgba = combined.Sample(uv);
```

There are many kinds of resources, and each one has a type.

For instance, here we define a Texture2D called tex. That will bind to a 2D texture object. **(click)** Calling Texture2D.Load loads the value of a single texel.

Texture2D objects are read-only. To read and write to them, we need **(click)** an RWTexture2D, like we saw in the intro example.

(click) You can also have 1D textures, 3D textures, and cubemap textures – **(click)** as well as more complex texture types like arrays, multisampled textures, feedback textures, and more.

Now, textures by themselves only give us point filtering. In order to get linear filtering, we need a **(click)** SamplerState object. That will bind to a descriptor saying what kind of filtering to use and how to interpolate between mips, as well as what to do if we sample outside the texture.

In Slang, we can write code like this more concisely by using the new **(click)** Sampler2D type, which is a Texture2D plus a SamplerState. To sample it, we just call **(click)** Sample. In Vulkan, this maps to a *combined texture sampler*, which is a single descriptor, while in

DirectX, this maps to a texture and a sampler.



- `ByteAddressBuffer` and `RWByteAddressBuffer` are raw buffers of bytes
 - Load/store at multiples of 4 bytes: `uint v = byteAddressBuffer.Load(word);`
- `StructuredBuffer<T>` is a buffer of `T`:

```
struct Material {  
    uint brdfType;  
    float roughness;  
}  
  
StructuredBuffer<Material> materials;  
...  
Material m = materials[5];
```

You can also have linear buffers of data. `ByteAddressBuffer` is a raw buffer of bytes. Like `RWTexture2D`, there's an `RWByteAddressBuffer` type that allows both reading and writing. **(click)** A `StructuredBuffer` is an array of a given type. For instance, here we define a material struct, and then a buffer of materials.

Constant Buffers

SIGGRAPH



- `ConstantBuffer<T>`
 - Read-only, usually up for 65,536 bytes
 - Good when all threads read the same value; depends on HW

```
struct DrawInfo {  
    float4x4 transform;  
    float    iTime;  
}  
  
ConstantBuffer<DrawInfo> info;  
...  
posOut = mul(posIn, info.transform);
```

- Slang places global `uniforms` in a global constant buffer

```
uniform float4x4 transform;  
uniform float    iTime;
```

Slang also has constant buffers, for read-only data limited to a maximum of usually 65,536 bytes. Constant buffers can be faster than storage buffers, especially when all threads in a wave read the same element at the same time, but it depends on how hardware implements them.

(click) In our earlier Mandelbrot sample, when you were reading uniform global constants like `iTime`, you were actually reading from a constant buffer! Slang places global uniforms like these in a global constant buffer for you.

Push/Root Constants

SIGGRAPH



- Push constants / root constants
 - Faster, smaller (128-256 bytes)
 - Best for data changing every draw call
- Slang makes **uniform** entrypoint parameters push constants (or ray tracing shader records)

```
[[vk::push_constant]]  
ConstantBuffer<DrawInfo> info;
```

New!

```
[shader("vertex")]  
VsOutput vsMain(...,  
    uniform DrawInfo info)  
{  
    ...  
}
```

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

48

Vulkan push constants, also known as root constants in DirectX, are even better than constant buffers when you have a small amount of data that changes every call. They're limited to 128-256 bytes usually, but they're good for things like IDs and model transform matrices.

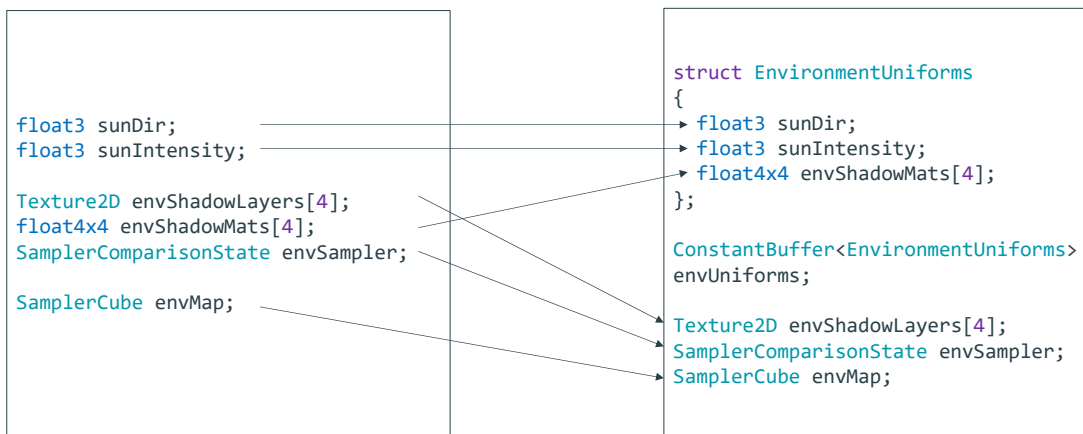
In HLSL, the syntax for push constants is kind of strange; you define a constant buffer, and then when you're creating your root signature you have to say "actually, this constant buffer isn't a constant buffer, it's a root constant". If you're targeting SPIR-V, **(click)** you have to add this `vk::push_constant` attribute.

(click)

Slang supports that, but it also provides cleaner syntax. If you mark an entrypoint parameter as `uniform`, then in SPIR-V it'll translate to a push constant, and for DirectX it'll show up as a root constant in reflection info, which we'll talk about in a few slides.

Parameter Blocks Help Organize

SIGGRAPH



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

49

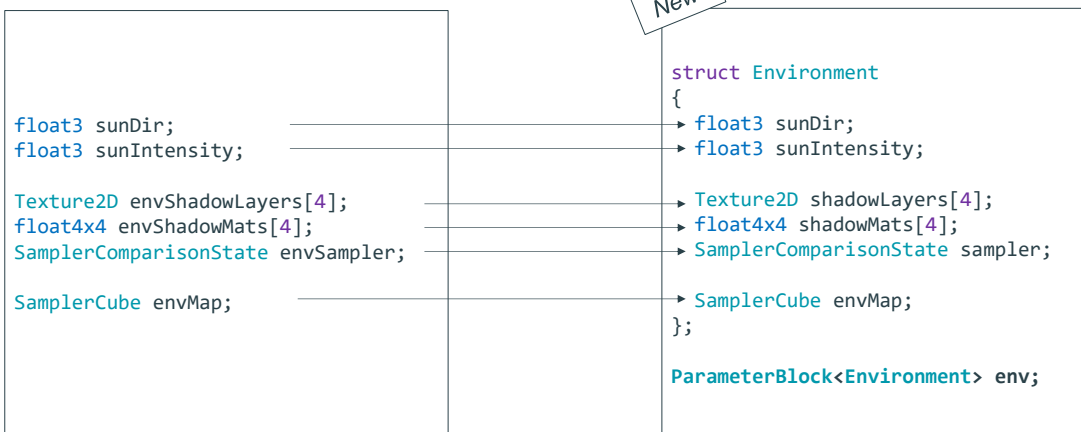
Graphics techniques often require multiple uniforms and resources. On the left we have the fields for a fairly typical environment technique. We've got a sun direction and color. Then the fields for a cascaded sun shadow map: four texture layers, the transformation matrix for each layer, and a sampler that does comparisons for percentage-closer filtering. And finally, a cubemap for the skydome.

In traditional shading languages, **(click)** we can't group these together. Because you can't have textures inside constant buffers, we're forced to separate uniforms into a struct and put them into a constant buffer. Because of this, these might be dozens of lines apart, or even in different files. We have to make sure their variable names don't conflict with other things in our codebase, and it gets messy.

Wouldn't it be cleaner if these were in a single struct?
Don't these *want* to be in a struct together?

Parameter Blocks Help Organize

SIGGRAPH



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

50

You can do this with Slang's `ParameterBlock` type. Now we can place all of our environment parameters in a struct, and place that in a `ParameterBlock`. When compiled, Slang will do the work of separating these into descriptors and constant buffers; while we're writing it, they're nicely visually grouped together.

We can even treat `ParameterBlocks` like other types – for instance, we can place them in structs or even nest them.

Explicit Bindings

SIGGRAPH



- HLSL-style:

```
Texture2D color : register(t3, space0);
```

- HLSL-style for Vulkan:

```
[[vk::binding(3, 0)]] Texture2D color;
```

- GLSL-style:

```
layout(binding=3, set=0) Texture2D color;
```

- Slang supports all 3!



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

51

So far, we've been talking about resource types.

The second part of shader I/O is defining where the shader expects the descriptor for each resource to be. If you want to *explicitly* specify this, there are 3 main ways.

DirectX uses different indices for textures, constant buffers, larger buffers (UAVs), and samplers. So in HLSL style, we can say that the `color` texture is bound to texture index 3 in space 0.

Vulkan is a bit simpler; it uses one index for the binding and one for the set. HLSL and GLSL have different ways of specifying this.

The good news is you can choose whichever one of these is your favorite; Slang supports all 3!

Too Many Bindings?

SIGGRAPH



```
[[vk::binding(0,0)]] Texture2D texture0; [[vk::binding(1,0)]] Texture2D texture1;  
[[vk::binding(2,0)]] Texture2D texture2; [[vk::binding(3,0)]] Texture2D texture3;  
[[vk::binding(4,0)]] Texture2D texture4; [[vk::binding(5,0)]] Texture2D texture5;  
[[vk::binding(6,0)]] StructuredBuffer<Vertex> mesh0; [[vk::binding(7,0)]] StructuredBuffer<Vertex> mesh1; ...
```

- Makes some graphics techniques infeasible
- Slow

When a shader needs a **lot** of resources – e.g. a ray tracing shader that needs access to every vertex buffer in the entire scene – using a separate binding for every resource is impossible. Or it can just be a lot of bindings to set, which slows things down.



- Solution: Bind/less!
 - Point to a buffer of descriptors in memory



New type!

```
[[vk::binding(0,0)]] StructuredBuffer<DescriptorHandle<Texture2D>> textures;
[[vk::binding(1,0)]] StructuredBuffer<DescriptorHandle<StructuredBuffer<Vertex>>> vertexBuffers;
```

- Use it like this:

```
float4 foo(DescriptorHandle<Texture2D> handle, SamplerState state, float2 uv) {
    return nonuniform(handle).Sample(state, uv);
}
```

The solution to this is to bind less! Instead of having a binding for each resource, we have an arbitrarily-sized buffer of descriptors somewhere in memory, and then we point to that and say “go read from resource number 5” for example.

Previously, one problem was that Vulkan and DirectX phrase bindless in different ways. And they use raw indices, which loses type safety.

Slang introduces the `DescriptorHandle` type, which converts to whichever bindless representation your API uses. So in the code here, we say that binding 0 in set 0 is a buffer of descriptor handles for `Texture2D` objects. And we do a similar thing for vertex buffers.

(click) You can use `DescriptorHandles` like the objects they point to. The only thing you need to make sure of is – if you can have different threads in the same wave accessing different resources, then you need to tell Slang by wrapping the handle in the *nonuniform* qualifier just before you use it.



- SPIR-V, C++, and CUDA have buffer addresses
- Slang lets you use pointers to global memory for these targets:

New!

```
struct Node
{
    float3* vertices;
    Node* next;
}

ConstantBuffer<Node> rootNode;
...

float3 pos = rootNode.next->next->next.vertices[10];
```

Additionally, if you're compiling to targets like SPIR-V, C++, and CUDA that support them, Slang will let you use pointers! This makes writing some data structures a *lot* easier. So for instance, here we're defining a linked list, just as easily as if we were writing CPU code, except this can run on the GPU.

Matching Binding Info

SIGGRAPH



shader.slang

```
[[vk::binding(1)]] Sampler2D color;  
[[vk::binding(3)]] Texture3D volume;  
  
[[vk::binding(4)]]  
StructuredBuffer<Vertex> vertices;
```

?

app.cpp

```
std::vector<VkDescriptorSetLayoutBinding>  
bindings =  
{  
    { .binding = 1,  
      .descriptorType =  
        VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER  
      , ... },  
    ...  
};
```

Finally, the app and the shader have to agree on the binding indices and types somehow, so that the app knows how to set up rendering pipelines and update uniform buffers correctly. This is a fairly common problem.

Matching Binding Info

SIGGRAPH



shader.slang

```
[[vk::binding(BINDING_COLOR)]]
Sampler2D color;
[[vk::binding(BINDING_VOLUME)]]
Texture3D volume;

[[vk::binding(BINDING_VERTICES)]]
StructuredBuffer<Vertex> vertices;
```

app.cpp

```
std::vector<VkDescriptorSetLayoutBinding>
bindings =
{
    {.binding = BINDING_COLOR,
     .descriptorType =
VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
, ...},
    ...
};
```

common.h

```
#define BINDING_COLOR 1
#define BINDING_VOLUME 3
#define BINDING_VERTICES 4

struct Vertex
{
    float3 pos; float padding;
    float2 uv;
};
```

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

56

The usual solution is to add a third file that defines all the binding indices and constant buffer structures, and then to include this in both the shader and the app. Since it's included by both Slang and C++ code in this case, it must be a *polyglot*: readable by both languages. And if you're familiar with layout rules in GLSL, you might notice that we have to be really careful about the alignment on the float2 field after this float3 field – or that we have to use scalar layouts, which are my preferred solution. That said, this tends to work pretty well.

*shader.slang*

```
Sampler2D color;  
Texture3D volume;  
StructuredBuffer<Vertex> vertices;
```



shader.spv



slang::ProgramLayout*

app.cpp

```
std::vector<VkDescriptorSetLayoutBinding>  
bindings = generateBindings(layout);
```



With Slang, another option is to use shader reflection! When you compile a shader with Slang, you can also get a `slang::ProgramLayout*` object that tells you every shader parameter's type, contents, and binding. Then you can use that to automatically generate your bindings on the app side – and on the shader side, your parameters look a lot simpler, since you don't have to explicitly specify binding indices any more.



- Reflection information includes:
 - Resource bindings + info
 - Struct layouts + info
 - Entrypoints + info
 - Much more
- This is how vk_slang_editor works!
- Slang Playground uses *custom attributes*
- Uniform buffer updates slower unless you JIT
- See Theresa Foley's talk, *Getting Started with Slang: Reflections API*

<https://youtu.be/OxOZ81N3NKw>

Reflection information includes resource types and binding indices, but it also includes a lot more. You can get how structs are laid out in memory and the names of each of their fields, each of the entrypoints and their attributes, and much more.

In fact, this is how the editor you've been using works! Whenever you compile a shader, it looks at the reflection info to figure out how to set up its pipelines and how to update its constant buffers. For instance, if you added a parameter named "eye" in the previous exercise, vk_slang_editor would look at the shader inputs, see that there was one named "eye", and would go "ah, that's asking for the camera position; I should add a camera widget and start updating that uniform with the camera position".

Slang Playground, which is an online Slang editor, uses a similar solution except it uses *custom attributes* instead of looking at variable names.

The one downside of reflection is that if you're using it to update uniform buffers, the additional lookups required will probably make it slower than the *common header* solution, unless you generate assembly on the fly.

Reflection is a big topic in and of itself; if you're interested in learning more, I recommend checking out Tess Foley's online talk about it.

Lab: Shader I/O

SIGGRAPH



- Can you figure out the missing types in *Examples > SIGGRAPH > 2 – Shader IO?*
- Use the comments as your guide
- Stuck?
 - The main keywords you'll need are in the box on the right →

We've talked about:

- `Texture2D`
 - `SamplerState`
 - `ByteAddressBuffer`
 - `StructuredBuffer<T>`
 - `uniform` (shader parameter)
 - `ConstantBuffer<T>`
 - `ParameterBlock<T>`
 - `DescriptorHandle<T>`
- } combined: `Sampler2D`

Answer

SIGGRAPH



```
// A shader parameter of type 'uint':
uniform uint kNumParticles;

// A readable and writable buffer of 'Particle's.
// (Slang lets us define the Particle struct anywhere in the file.)
// This one has been completed for you.
RWStructuredBuffer<Particle> particles;

// A readable and writable buffer of 'uint's:
RWStructuredBuffer<uint> tileParticleIds;

// A 2D readable and writable texture.
RWTexture2D texFrame;

// A 2D combined texture and sampler.
Sampler2D texPuzzleMedian;
```

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

60

By the way, I mentioned earlier that the editor you're using uses *reflection* to set up all its shader bindings.

It has some features to show what's going on there. If you open View > Reflection, you can see some of the reflection info Slang generates in a JSON view. Here's the global constant buffer for instance, and here's the kNumParticles field within it.

Vk_slang_editor uses this info to also create GUI controls for each of the parameters. If you click on "Shader Parameters" in the menu bar, you can see each of the resources. Here I can control the value of kNumParticles.



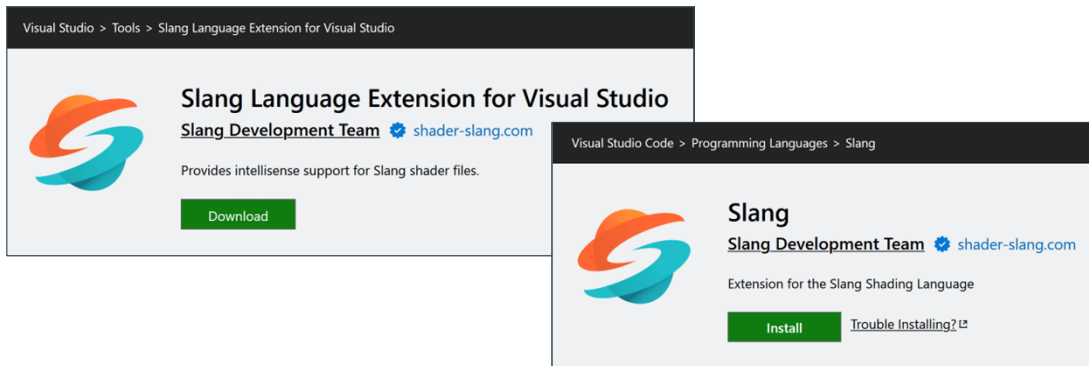
Now, let's talk about tooling. Slang has a wide amount of support in IDEs and debuggers.

Writing Slang

SIGGRAPH



- Slang extensions for Visual Studio and Visual Studio Code



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

If you're using Visual Studio or Visual Studio Code, Slang provides official language extensions.



- Any editor that supports the Language Server Protocol can use slangd

```

13 // See the license for the specific language governing permissions and
14 // limitations under the license.
15
16 * SPDX-FileCopyrightText: Copyright (c) 2022-2025, NVIDIA CORPORATION.
17 * SPDX-License-Identifier: Apache-2.0
18 */
19
20
21 #include "hdr_io.h.slang"
22 #include "constants.h.slang"
23 #include "functions.h.slang"
24 #include "sample_blur.h.slang"
25
26 // clang-format off
27 [TextureBinding(EnvDraw::idfrags_0)]..._RTexture2D(float2x2 gOutldr; Christoph Kubisch 202...
28 [[vk::binding(EnvDrawings::hdr, 1)]]...Sampler2D gInldr;
29 [[vk::push_constant]]...ConstantBuffer<HdrDomePushConstant> pc;
30 // clang-format on
31
32
33 [shader("compute")]
34 [numthreads(HDR_WORKGROUP_SIZE, HDR_WORKGROUP_SIZE, 1)]
35 void main(uint3_t threadId: SV_DispatchThreadID)
36 {
37     const float2 pixel_center = float2(threadId.xy) + float2(0.5);
38
39     int2 image_size;
40     gOutldr.GetDimensions(image_size.x, image_size.y);
41 }
  
```

QtCreator

```

150
151 p = float3(-2, -2, 2);
152 p /= clamp(dot(p, p), -4.5, 18.0);
153
154 p *= float2(1.0, 1.2) * 3.1; Defined in C:\lab\Gfx\Nv\upro
155 samples\develop\vk_slang_editor\examples\Showcase\Fractal Flame.
156 slang(35)
157
158 // Split the particle
159 float depth;
160 float2 ndc = projectParticle(p, de
161 ndc = kAperture
162 // Depth of field
163 * (1 - kFocusDepthAdjust *
164 * randomPointInDisk() // iho
165 // Depth rejection: replace with
166 if (depth > 0.)
167     defer
168     degrees
169     uint2 intP = int2((1.5 * ndc * depth
170     if (intP.x > 0 && intP.y > 0) DescriptorAccess
  
```

This lab's editor

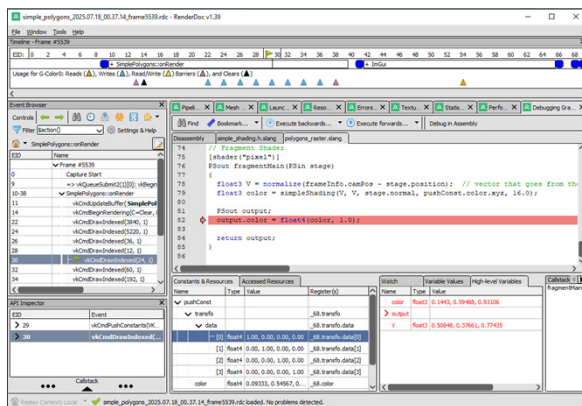
In addition, any editor that supports the Language Server Protocol, which powers things like syntax highlighting, Intellisense, jump to function, diagnostics, and much more for languages such as Python and Rust, can also use Slang's language server, slangd.

For instance, this is how autocompletion and function definitions work in the editor you've been using in this lab!

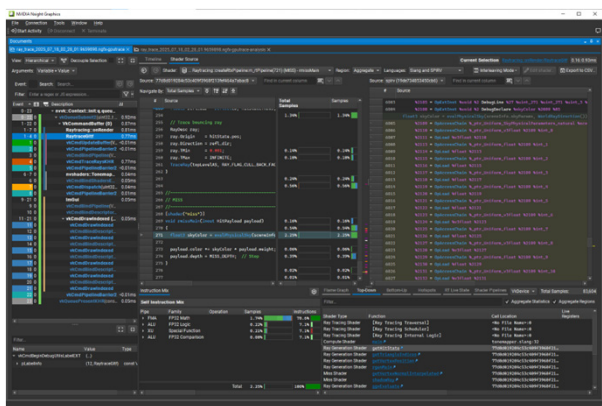
(Slangd and other language servers for languages like Python and Rust are just programs that run locally where the editor can pipe in source code and make requests, and the program responds with things like "here are the available autocompletions".)

Debugging Shaders

SIGGRAPH



RenderDoc



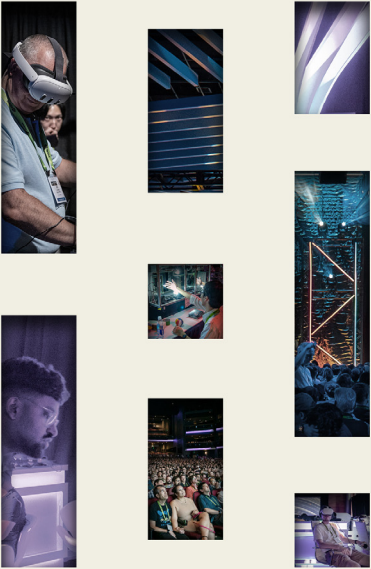
Nsight Graphics

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

64

Debugging Slang shaders also works like debugging other shading languages, so you get a pretty natural experience.

The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques



 **SIGGRAPH 2026**
Los Angeles 19-23 JUL

Demo: Shader Debugging

© 2026 SIGGRAPH. ALL RIGHTS RESERVED. 

Switch to desktop + notes; next slide will be “Experiment” with Mandelbrot

Demo: Shader Debugging

SIGGRAPH



- *Open Martin-Karl's `simple_raster` sample in RenderDoc*
- *Show draw calls*
- *Show how to view vertex data*
- *Show how to view pass constants*
- *Show "Debug this Vertex"; point out how variable names appear.*

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.



- Embed shader source code using `-g1`

```
15         ; Debug Information
16         %1 = OpString "RWTexture2D<float4> texFrame;
17     [shader(\"compute\")]
18     [numthreads(16, 16, 1)]
19     void main(uint2 thread: SV_DispatchThreadID) {
20         texFrame[thread] = float4(1.0, 0.0, 0.5, 1.0);
21     }
22     "
```

- For releases:
 - Remove debug info using `-line-directive-mode none`
 - Or split line info to a source map using `-line-directive-mode source-map -o out.zip`
 - SPIR-V supports `-separate-debug-info`

To make your app easier to debug, the most important thing you can do is to specify `-g1` on the Slang command line.

This will make formats like SPIR-V embed a copy of your shader code – so even if the debugger doesn't have shader search paths set up, it can still load the shader source. Slang will automatically generate line info, mapping from output instructions to input lines, by default.

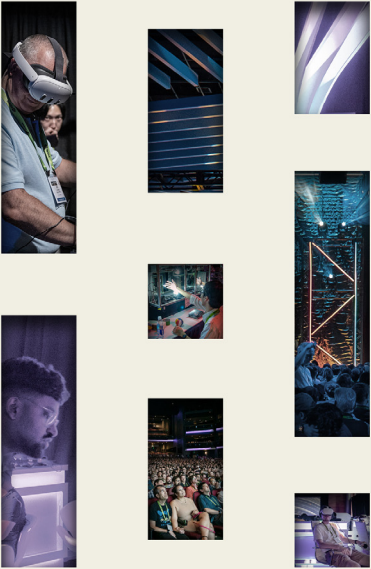
(click).

When releasing your app, you might want to make your shaders harder to debug. Slang gives you a few options here.

You can remove debug info by setting `-line-directive-mode` to `none`.

Or you can *split off* the debug info into a *source map*, or into a separate SPIR-V file.

The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques



 **SIGGRAPH 2026**
Los Angeles 19–23 JUL

Structs, Modules, and Interfaces

© 2026 SIGGRAPH. ALL RIGHTS RESERVED. 

In this section, we'll talk about some of Slang's advanced language features.



```
struct SdfInfo
{
    float m_t;
    uint* m_data;
}
```

Structs are all about organizing variables. For most of my examples, I'll use a struct that looks like this; it contains a float, and a pointer to some unsigned integer data in memory.

The first thing you might notice is there's no semicolon after this struct! This is purely a convenience feature; you don't have to have them in Slang.



```
struct SdfInfo  
{  
    float m_t;  
    uint* m_data;  
}
```

```
SdfInfo s;  
s.m_t    = INFINITY;  
s.m_data = nullptr;
```

Usually when you have a type like this, you always want to initialize it with some default values after you create it. HLSL and GLSL don't have constructors, so usually you have to initialize fields explicitly, or do some other workaround.



New!

```
struct SdfInfo
{
    float m_t;
    uint* m_data;

    __init()
    {
        m_t = INFINITY;
        m_data = nullptr;
    }
}
```

```
SdfInfo s = {};
// s now initialized
```

Slang lets you define constructors for types! Constructors are functions named `__init`, with two underscores. This is a default constructor that is called whenever you create a struct of this type.



New!

```
struct SdfInfo
{
    float m_t;
    uint* m_data;

    __init(uint* data)
    {
        m_t = INFINITY;
        m_data = data;
    }
}
```

```
SdfInfo s(pSky);
```

You can also provide parameters to constructors, like this

Default Values

SIGGRAPH



New!

```
struct SdfInfo
{
    float m_t      = INFINITY;
    uint* m_data = nullptr;

    // Slang auto-generates __init()
}
```

```
SdfInfo s;
```

Or you can have default values for struct members, and Slang will auto-generate the constructor.



```
struct SdfInfo
{
    float m_t    = INFINITY;
    uint* m_data = nullptr;

    float getT() { return m_t; }
    uint at(uint i) { return m_data[i]; }
}
```

Structs can also have functions, which work like member functions in other programming languages.



```
struct SdfInfo
{
    float m_t    = INFINITY;
    uint* m_data = nullptr;

    float getT() { return m_t; }
    uint at(uint i) { return m_data[i]; }

    static uint ID() { return 0; }
}
```

Can now write
`uint id = SdfInfo.ID();`

And you can have static member functions. Here `ID()` is a static function, and I can write `SdfInfo.ID()`.



```
struct SdfInfo
{
    float m_t    = INFINITY;
    uint* m_data = nullptr;

    float getT() { return m_t; }
    uint at(uint i) { return m_data[i]; }

    void update(float t, uint* data)
    {
        if(m_t > t) return;
        m_t = t;
        m_data = data;
    }
}
```

Functions are const by default. If a function changes struct values, **(click)**



New!

```
struct SdfInfo
{
    float m_t    = INFINITY;
    uint* m_data = nullptr;

    float getT() { return m_t; }
    uint at(uint i) { return m_data[i]; }

    [mutating]
    void update(float t, uint* data)
    {
        if(m_t > t) return;
        m_t = t;
        m_data = data;
    }
}
```

then you need to mark it as [mutating].



New!

```
struct SdfInfo
{
    private float m_t    = INFINITY;
    private uint* m_data = nullptr;

    float getT() { return m_t; }
    uint at(uint i) { return m_data[i]; }

    [mutating]
    void update(float t, uint* data)
    {
        if(m_t > t) return;
        m_t = t;
        m_data = data;
    }
}
```

Slang also lets you make member variables and functions private using the private keyword. This is useful for the same reasons as in C++ and other languages.



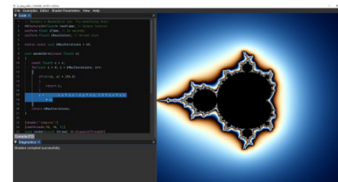
```
struct Complex
{
    float r, i;
}

Complex operator*(Complex a, Complex b)
{
    return Complex(a.r * b.r - a.i * b.i,
                  a.r * b.i + a.i * b.r);
}
```

```
Complex z, c;

// ...

return z * z + c;
```



You can also *overload* operators for structs in Slang. Earlier in our Mandelbrot example, we had some math on *float2s* that was really doing multiplication for complex numbers. But a *cleaner* way might have been – instead of using *float2s* – define a *Complex* number type, and then make it so that the multiplication operator does *complex* instead of elementwise multiplication. Then our Mandelbrot iteration would have been a lot simpler to write.

Include Files

SIGGRAPH



- Supported in Slang
- Text substitution
- If any header changes, all dependents must be recompiled
- No private types
- #define leaks out

there's something better

bsdf.slang

```
#ifndef BSDF_SLANG
#define BSDF_SLANG

float3 ggxSample(
    float3 wi, float2 a2, float2 xi)
{ ... }

// Private, please
float sqr(float x)
{ return x * x; }
#endif
```

main.slang

```
#include "bsdf.slang"

[shader("fragment")]
...
```

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

80

Structs help organize code on the small scale. Now let's talk about organizing larger codebases.

Slang lets you *include* other files, like other languages do. But this old technique has some downsides.

Since #include works by (virtually) substituting in the text of files you include and then compiling the result, if a single header changes then everything that depends on it has to be recompiled from scratch. **(click)**

GLSL and HLSL don't let you have private functions in headers. **(click)**

And there are smaller issues – for instance, preprocessor defines can leak out of the file they were defined in. **(click)**

There's something better.

Modules

SIGGRAPH



New!

- Compiled separately; linked together
- public, private, internal
- #define doesn't leak out

Public →

```
bsdf.slang
module bsdf;

public float3 ggxSample(
    float3 wi, float2 a2, float2 xi)
{ ... }
```

Only visible within the bsdf module →

```
internal float sqr(float x)
{ return x * x; }
```

```
main.slang
import bsdf;

[shader("fragment")]
...
```

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

81

Slang introduces support for *modules* in shader languages. These look a lot like standard include files, but they're compiled *separately* and then *linked* together. This can improve compile times, as we'll talk about in a bit.

You can also have public, private, and internal members. Private members are only visible within the current file, while internal members are visible to the rest of the module (if you have multiple files making up a module).

And, things like preprocessor macros don't affect other files.

(You don't need to specify `internal` on `sqr()` here if you use Slang language version 2025 or newer: specify `-lang 2025` on the command line.)

The Standard Libraries are Modules

SIGGRAPH



hlsl.meta.slang

slang / source / slang / hlsl.meta.slang

```
Code Blame 28438 lines (26617 loc) · 934 KB · 0
11090 }
11091
11092 // Compute base-10 logarithm.
11093 // @param x The input value.
11094 // @return The base-10 logarithm of 'x'.
11095 // @category math
11096 // @generic T : __BuiltinFloatingPointType
11097 [__readNone]
11098 [require(cpp_cuda_gsl_hlsl_metal_spirv_wgsl, sm_4_0_version)]
11099 T log10(T x)
11100 {
11101     __target_switch
11102     {
11103         case hlsl: __intrinsic_asm "log10";
11104         case metal: __intrinsic_asm "log10";
11105         case wgsl: __intrinsic_asm "(log($0) * $50( 0.43429448190325182765112891891661 ))";
11106         case glsl: __intrinsic_asm "(log($0) * $50( 0.43429448190325182765112891891661 ))";
11107         case cuda: __intrinsic_asm "$P_log10($0)";
11108         case cpp: __intrinsic_asm "$P_log10($0)";
11109         case spirv:
11110         {
11111             const T tmp = T(0.43429448190325182765112891891661);
11112             return spirv_asm (
11113                 NbaseFlog$5$T = OpExtInst glsl450 Log $x;
11114                 result = CET = OpFMA11 $baseFlog$5$T $x $tmp;
11115             );
11116         }
11117     }
11118 }
11119
11120
```

glsl.meta.slang

slang / source / slang / glsl.meta.slang

```
Code Blame 9850 lines (9062 loc) · 268 KB · 0
213 public in int gl_SampleID : SV_SampleIndex;
214 public in int gl_ViewIndex : SV_ViewID;
215 public in int gl_ViewportIndex : SV_ViewportArrayIndex;
216 public in int gl_BaseVertex : SV_StartVertexLocation;
217 public in int gl_BaseInstance : SV_StartInstanceLocation;
218
219
220 // Override operator* behavior to compute algebraic product of matrices and vectors.
221
222 [OverloadRank(15)]
223 [forceinline]
224 [require(cpp_cuda_gsl_hlsl_spirv, sm_4_0_version)]
225 public matrix<float, N, N> operator*(int N:int)(matrix<float, N, N> m1, matrix<float, N, N> m2)
226 {
227     return mul(m2, m1);
228 }
229
230 [OverloadRank(15)]
231 [forceinline]
232 [require(cpp_cuda_gsl_hlsl_spirv, sm_4_0_version)]
233 public matrix<half, N, N> operator*(int N:int)(matrix<half, N, N> m1, matrix<half, N, N> m2)
234 {
235     return mul(m2, m1);
236 }
237
```

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

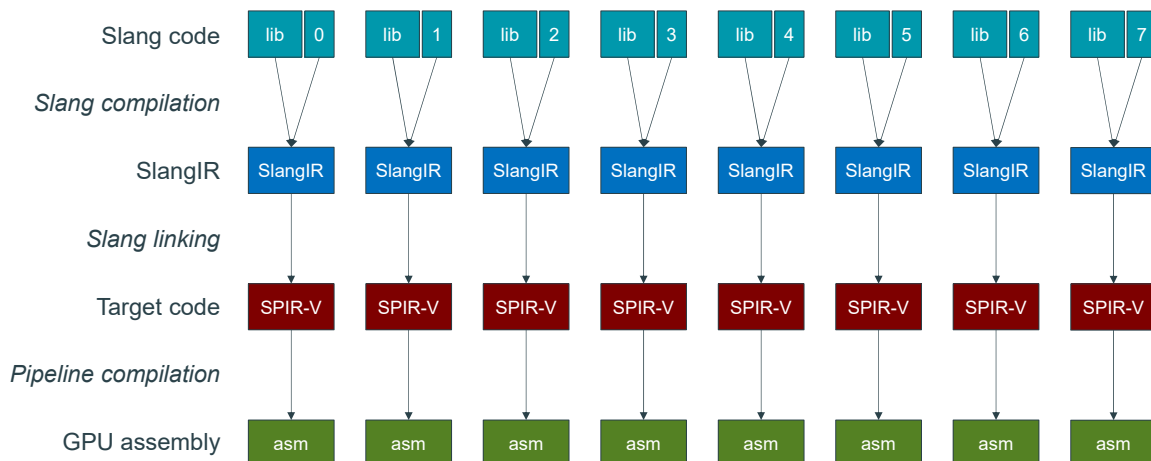
82

Notably, Slang’s standard library uses modules. All Slang files *link* with `hlsl.meta.slang`. And when you enable GLSL compatibility by adding `#version`, Slang links with `glsl.meta.slang` as well.

If you’re ever wondering how Slang’s intrinsic functions work, you can look at their implementations! Reading these files is also useful for learning about advanced features like *inline assembly*, which lets you use features even *Slang* doesn’t know about yet.

Now, a word on compile times.

Improving Compile Times: Modules + Specialization



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

83

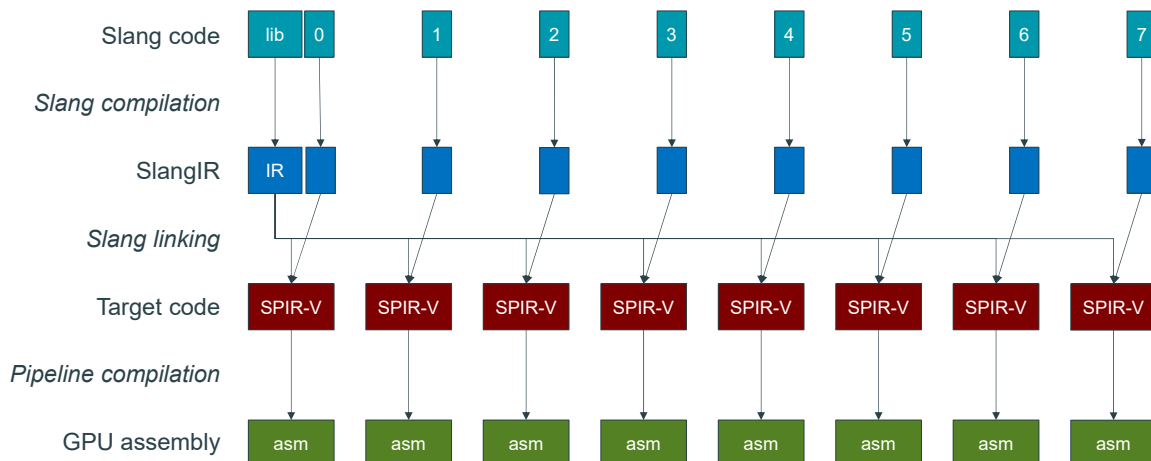
Modules can help with compile times if you use them more than once.

Here's a diagram of an app compiling 8 graphics pipelines. The input for each one is a shader that links against a larger library.

Slang compiles shaders to SlangIR, then links them together and emits target code – like SPIR-V. When the app creates a pipeline at runtime using a SPIR-V module, the graphics driver will usually do its own set of optimizations, producing GPU assembly.

If you use modules, **(click)**

Improving Compile Times: Modules + Specialization



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

84

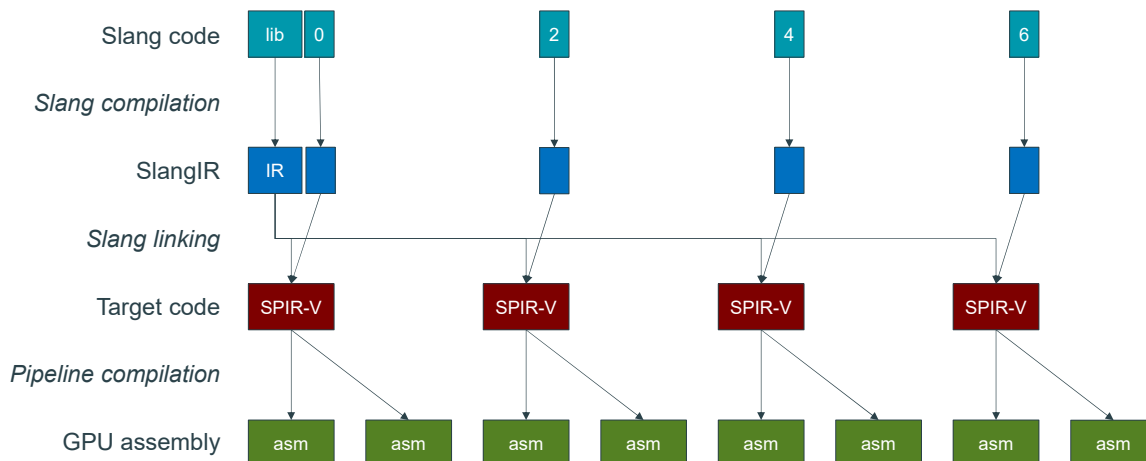
you only have to compile the shared library once, and then you can link each shader against it, saving time.

In practice, for a hot-reload benchmark I put together for a path tracer, I saw this reducing compile times by about 30%.

Another thing that can help is **(click)**

(Benchmark: <https://github.com/NBickford-NV/slang-compile-timer>)

Improving Compile Times: Modules + Specialization



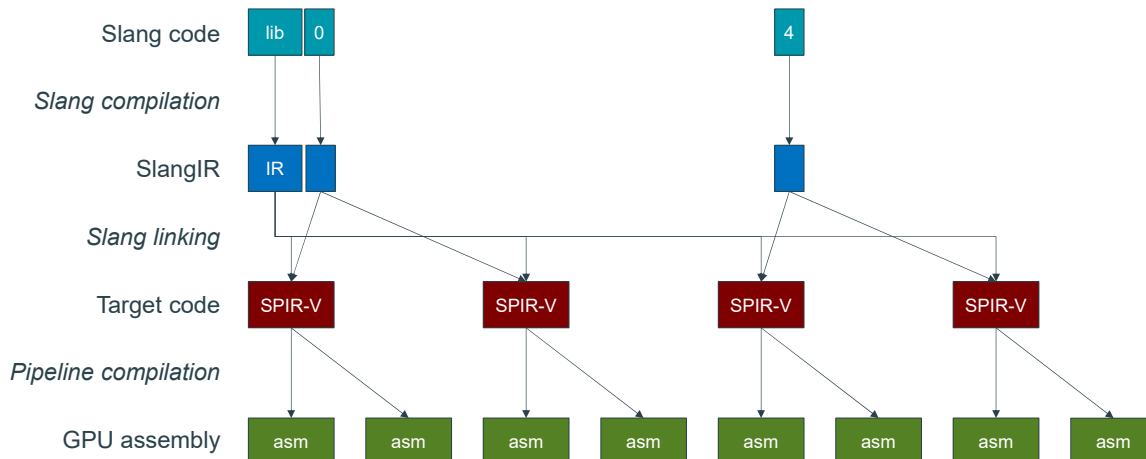
© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

85

specialization constants, like in Vulkan, which are designed to reduce *shader permutations*. Instead of say, compiling different shaders for 1 light, 2 lights, or 3 lights per pixel, you can use a variable at the Slang level, and only specify its value when the driver needs to do final optimizations.

New in Slang, you can now do **(click)**

Improving Compile Times: Modules + Specialization



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

86

link-time specialization, where you can specify *types* for things *after* compiling but *before* linking.

All of these taken together reduce the total amount of work quite significantly! But you can see we're still paying the load-time cost for the driver to optimize and generate GPU assembly for each of these pipelines. So we're not quite in an ideal world for shader compilation yet.



```
struct PointLight
{
    float3 m_color;
    float3 m_pos;

    float3 irradiance(float3 pos)
    {
        return color
            / dot(pos - m_pos, pos - m_pos);
    }
}
```

```
struct DirectionalLight
{
    float3 m_color;

    float3 irradiance(float3 pos)
    {
        return m_color;
    }
}
```

```
float3 pointLightSum(
    PointLight* lights,
    uint numLights
)
{
    // Same code
}
```

```
float3 directionalLightSum(
    DirectionalLight* lights,
    uint numLights
)
{
    // Same code
}
```

Finally, let's talk about interfaces. It's pretty common in computer graphics to have multiple objects that conform to some sort of higher-level interface. For instance, point lights, directional lights, and spotlights are all kinds of lights. Mirror, Lambert, and GGX are all BRDFs. And so on.

Here we have one struct for a point light and another for a directional light. If we want to sum over all point lights, we can write a function to do that. And if we want to sum over all directional lights, we can also write a function to do that. But the code inside these functions will look exactly the same. So, ideally, we'd like to *generalize* this function somehow.



```
struct PointLight : ILight
{
    float3 m_color;
    float3 m_pos;

    float3 irradiance(float3 pos)
    {
        return color
            / dot(pos - m_pos, pos - m_pos);
    }
}
```

```
struct DirectionalLight : ILight
{
    float3 m_color;

    float3 irradiance(float3 pos)
    {
        return m_color;
    }
}
```

```
interface ILight
{
    float3 irradiance(float3 pos);
}
```

```
float3 lightSum<LightType>(
    LightType* lights,
    uint numLights
)
where LightType : ILight
{
    // ...
}
```

Slang provides a solution to this. First we define an *interface* in the box on the top-right. This is saying that any type that conforms to the ILight interface must have a function called “irradiance” that takes in a float3 position and returns a float3.

Then over on the left, we say that PointLight implements the ILight interface by adding a colon followed by ILight, and we do the same for DirectionalLight. **(click)**

Now we can write a function called lightSum that can take in *any* light type!

This is a *generic* function – and in fact, we’ve been seeing generics this whole time! Texture2D and StructuredBuffer were generic *types*, for instance.

Generics are a lot like C++ templates; really, the only difference is we need to say what interfaces our types conform to. This works *really* well with modules.

In C++, you usually have to put all your templates in these massive headers, so that compilers can look at their definitions. With generics, **(click)** these can be in separate modules; no need for headers!

On the left, the compiler only needs to check that Point and DirectionalLight conform to ILight. And on the right, it only needs to check that lightSum uses only what ILight gives it.

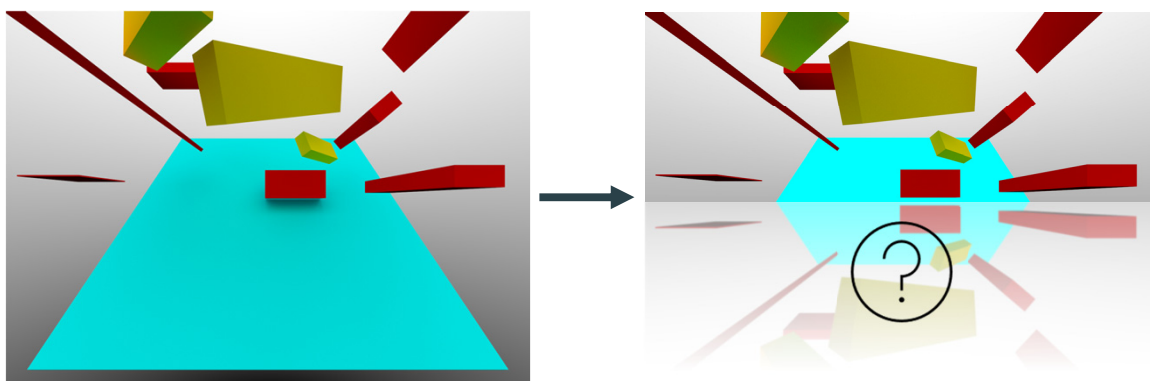
And you also get better compile-time errors as a result.

Puzzle: Implement an Interface

SIGGRAPH



- Can you implement a mirror BRDF to reveal the pattern in the reflection?



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

89

Let's put this into practice with my final puzzle of the day.

In `vk_slang_editor`, if you load Examples > SIGGRAPH > 3 – Interfaces, you'll see a path-traced scene with several colored boxes, all using a diffuse BRDF.

Now, this seemingly random arrangement of boxes in fact contains a hidden image.

Your challenge is to reveal this hidden image by turning the cyan plane here into a mirror.

To do this, you'll need to remove line 13, and then write a struct called `MirrorBRDF` that conforms to the `IBrdfs` interface and reflects the input direction along the hit normal.

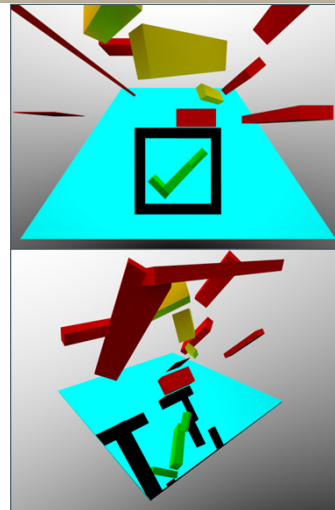
I'll be back in 5 minutes to show how to solve this puzzle, and then Chris will cover `SlangPy` and autodifferentiation. Good luck!

Solution

SIGGRAPH



```
struct MirrorBrdf : IBrdf
{
    static float3 sample(HitInfo hit,
                        out float3 wOut,
                        inout uint rngState)
    {
        wOut = reflect(hit.direction, hit.normal);
        return hit.baseColor;
    }
}
```

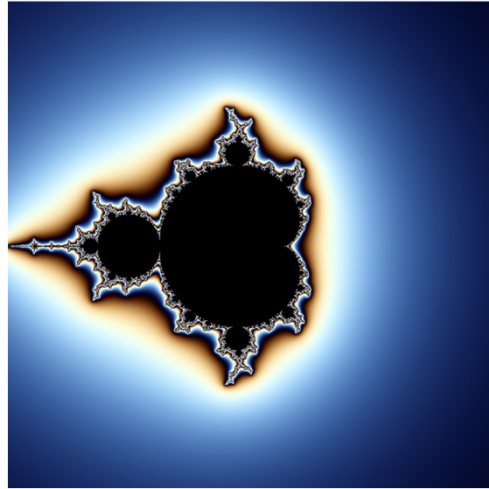




Thanks, Nia. So you've just seen Slang itself — the language. What I want to show you for the next half hour is **SlangPy**: the Python layer that lets you drive all of that from a notebook, with almost none of the usual host-side ceremony. I'm going to start by breaking a promise every graphics talk makes, which is that the demo will work. Because we're going to do the demo *first*.



Remember this?



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

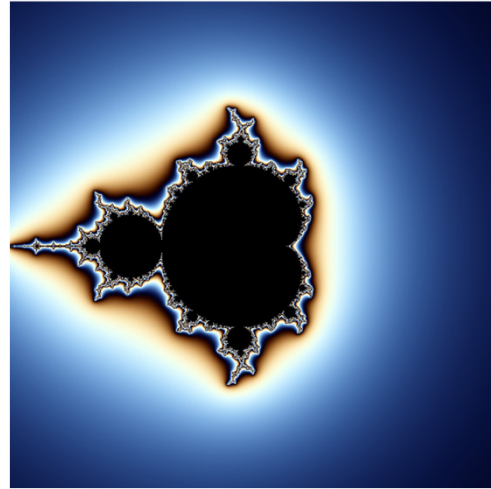
Remember this? This is the shader Nia just walked you through — the Mandelbrot set, written in Slang, completely standard. Nothing here is special yet. Hold that thought.



Remember this?

Well, let's take Nia's shader,
completely unmodified,
and run it with SlangPy...

To the notebook!



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

Here's the claim. We're going to take Nia's shader — *unmodified*, not one character changed — and I'm going to stand it up and run it from Python, live, in front of you. No C++. No swapchain. No descriptor sets. Let's just go to the notebook.
[Do NOT switch yet — one primer slide first so the room can follow along.]

IPython Notebooks in Visual Studio Code



IPython notebooks contain **code cells** and **markdown cells**.

They can run in Visual Studio Code –
or in a web browser.

To run a code cell, place your cursor on the cell,
and click the **play button**
in the top-left corner.

If you get behind (because you are experimenting
because you love SlangPy as much as we do),
go to the top of the page and
click **Restart**, then **Run All** to run all the cells.

code cell

```
import slangpy as spy
import os
import numpy as np
import matplotlib.pyplot as plt
import random
import PIL.Image as img
from IPython.display import display
```

markdown cell

Firstly we create a slangpy device, then load the slang pro
We create a texture for the shader to write to, a command

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

94

Quick orientation for anyone following along on their own machine.

A notebook is just a stack of cells. Two kinds: **code cells**, which run, and **markdown cells**, which are the notes explaining what's going on. You run a code cell by clicking into it and hitting the play button top-left. This runs in VS Code, or in a browser — your choice.

And the one survival tip: if you fall behind — because you're off experimenting, because you love SlangPy as much as we do — just go to the top, hit **Restart**, then **Run All**, and you'll catch right back up.

One more thing before we run anything: **installing SlangPy is a single line**. The very first cell in the notebook does it for you —

```
python -m pip install slangpy
```

That's the whole install. If you're following along on your own machine, run that first cell and you're ready. There's a conda path too, but pip is all we need today.

[NOW switch to the Mandelbrot notebook.]

LIVE — Mandelbrot notebook (~2 min)

[Screen: Mandelbrot notebook. Run All, or step the imports → dispatch → display cells.]

Talk track while it runs:

[Run the first cell — the pip install slangpy.] First cell just installs SlangPy — that one line

from a moment ago. Takes a couple of seconds. **[If it's already installed in your environment, note that so nobody panics when it returns instantly.]**

Next cell is just imports — slangpy as spy, numpy, matplotlib, PIL. Nothing exotic.

[Run the setup + dispatch cell.] Watch what this does: it creates a device, allocates a 1024×1024 float texture, loads Nia's .slang file straight off disk, sets the parameters, and dispatches.

[Image appears.] And there it is. Same shader, now driven from Python, rendered on the GPU, handed back as a NumPy array, displayed inline. That whole round trip is what's on the next slide.

[Switch back to slides.]

First, a Demonstration

SIGGRAPH

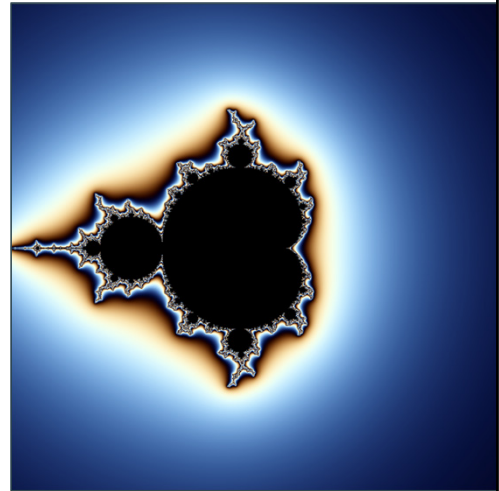


```
device = spy.create_device(include_paths=[os.getcwd()])

tex = device.create_texture(
    width  = 1024,
    height = 1024,
    format = spy.Format.rgba32_float,
    usage  = spy.TextureUsage.shader_resource |
            spy.TextureUsage.unordered_access
)

spy.Module.load_from_file(device, 'Mandelbrot.slang').render.set({
    'texFrame' : tex,
    'iTime'    : 0.0,
    'iResolution': spy.float2(tex.width, tex.height),
    'iMouse'   : spy.float4(0, 0, 0, 0)
}).dispatch((tex.width, tex.height, 1))

display(img.fromarray((tex.to_numpy()*255).astype(np.uint8)))
```



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

Let me show you what actually did the work, because it's shorter than you're expecting.

`spy.create_device` — that's your whole device init. One line.

`device.create_texture` — allocate the output, RGBA32 float, marked shader-resource and unordered-access so the compute shader can write it.

`spy.Module.load_from_file(...).render.set({...}).dispatch(...)` — load the shader, bind the uniforms by name — `iTime`, `iResolution`, `iMouse`, the frame texture — and dispatch one thread per pixel.

Last line pulls the texture to NumPy, scales to 8-bit, and displays it.

That's it. That's the entire host program.

First, a Demonstration

SIGGRAPH



```
device = spy.create_device(include_paths=[os.getcwd()])

tex = device.create_texture(
    width = 1024,
    height = 1024,
    format = spy.Format.rgba32_float,
    usage = spy.TextureUsage.shader_resource |
            spy.TextureUsage.unordered_access
)

spy.Module.load_from_file(device, 'Mandelbrot.slang').render.set({
    'texFrame' : tex,
    'iTime' : 0.0,
    'iResolution': spy.float2(tex.width, tex.height),
    'iMouse' : spy.float4(0, 0, 0, 0)
}).dispatch((tex.width, tex.height, 1))

display(img.fromarray((tex.to_numpy()*255).astype(np.uint8)))
```

Well, that's just 4 lines of code to:

- Initialize a device
- Allocate a texture
- Compile + dispatch a shader
- Display the result

... just sayin'.

So strip it down and that's really four things:

Initialize a device.

Allocate a texture.

Compile *and* dispatch a shader.

Display the result.

Four lines that do what is normally, what, a few hundred lines of Vulkan or DX before you see a single pixel. Same shader Nia wrote. Zero changes.

[beat]

...just sayin'.

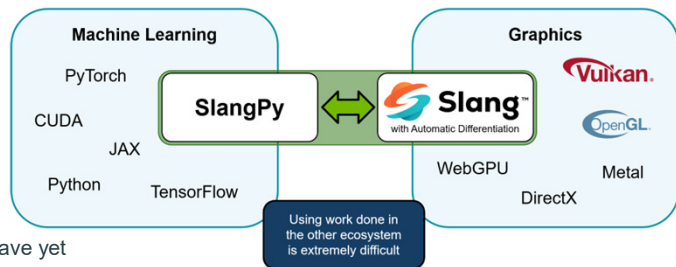
© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

Why SlangPy?

SIGGRAPH



- Provides a simple path to working with Slang
 - No need to wrestle with 1000s of lines of C++ code.
 - Dispatch a Slang shader with as few as 4 instead!
 - Makes testing prototypes and ideas as much simpler
- Helps bridge the gap between graphics and ML
 - Use Slang alongside Python's huge library ecosystem
 - E.g. design and test new operators PyTorch doesn't have yet



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

97

So why does this exist? Two reasons.

First, it's the shortest path to actually *working* with Slang. You don't have to wrestle thousands of lines of C++ to try an idea — you dispatch a shader in four lines and iterate. Prototyping goes from an afternoon to a coffee break.

Second — and this is the bigger one — it bridges two worlds that normally don't talk to each other. On the left, the ML ecosystem: PyTorch, CUDA, all of Python. On the right, graphics: Vulkan, DirectX, WebGPU, shading languages. Historically, using work from one side inside the other is *painful*. SlangPy puts Slang right in the middle of your Python session, so you can design and test a new operator — something PyTorch doesn't ship yet — and run it on the GPU immediately.

Why SlangPy?

SIGGRAPH

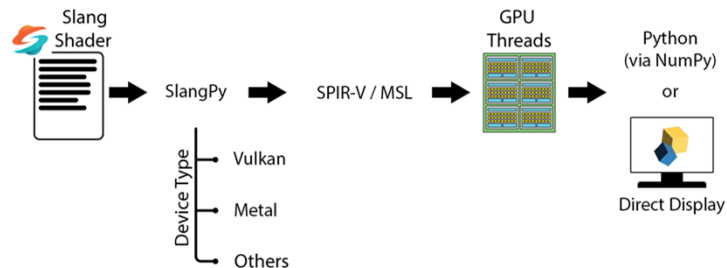


- Abstracts away the details of dispatching threads
 - But it's still there if you want it.
 - You can use it at both a high level and at a low level.

- Installs with a single call to your package manager.

- Supports a large array of backends.

- Vulkan
- Metal
- DirectX
- WebGPU
- CUDA



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

98

A few practical things.

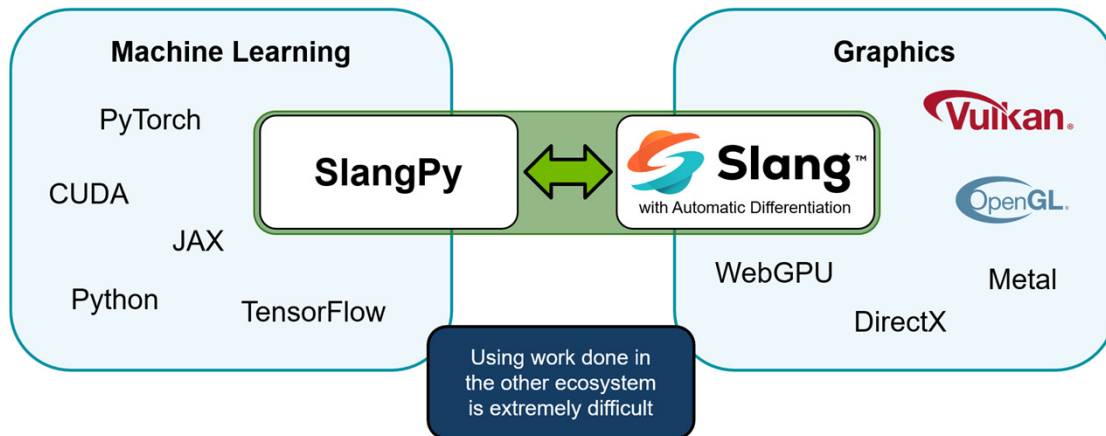
It abstracts away thread dispatch — but it doesn't *hide* it. You can work at a high level and let it handle the plumbing, or drop down and control dispatch yourself. You'll see both in the next notebook: the high-level `.dispatch` call, and the lower-level buffer and reflection API right next to it.

It installs with a single call to your package manager — you saw that a few minutes ago, one `pip install` line and we were dispatching shaders. That's the whole setup.

And it runs on a large spread of backends — Vulkan, Metal, DirectX, WebGPU, CUDA. Write once, dispatch onto whatever's in the machine.

Why SlangPy?

SIGGRAPH



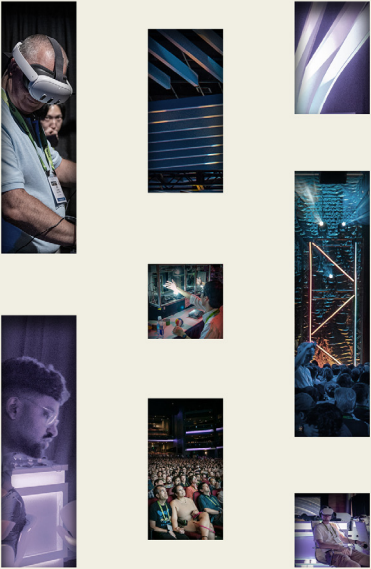
© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

99

Same idea, bigger picture. Two rich ecosystems, and the thing that's been missing is a comfortable bridge between them. That's the whole pitch: bring the graphics pipeline into Python, and bring Python's libraries to the GPU.

That bridge really opens up once your shaders can be *differentiated* — which is where I want to spend the rest of our time.

The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques



 **SIGGRAPH 2026**
Los Angeles 19-23 JUL

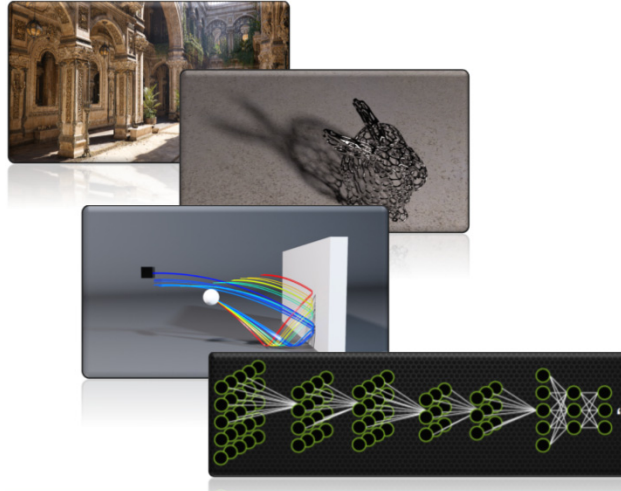
Autodifferentiation

© 2026 SIGGRAPH. ALL RIGHTS RESERVED. 

So. Autodiff.



- Neural rendering
 - Gradients for Neural Radiance Fields
 - Generating BSDFs from real images
- Differentiable rendering
 - Fitting 3D models to images
 - Automatic lighting
- Physics simulation
 - Finite element method derivatives
 - Inverse kinematics
- Optimized procedural noise
- Neural network training



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

Before the mechanics — why should a graphics person care?

Because once a function is differentiable, it becomes *trainable*, and that unlocks a huge amount of graphics.

Neural rendering — gradients for NeRFs, learning BSDFs from real photos.

Differentiable rendering — fitting 3D models to images, solving for lighting automatically.

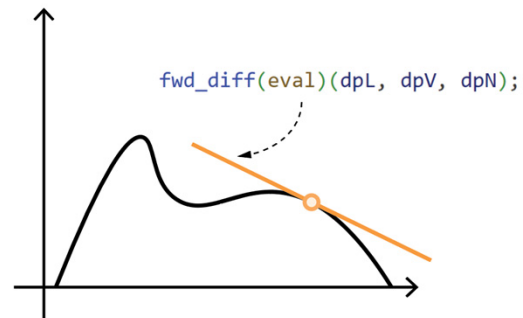
Physics — finite element derivatives, inverse kinematics.

Optimized procedural noise, and of course training neural networks directly.

And this list isn't exhaustive — pretty much anywhere you'd otherwise hand-derive a gradient, this applies.



- What is autodiff?
 - Automatically computes exact **derivatives** of any function
 - Supports arbitrary control flow and function calls
 - Enables any graphics function to become trainable
- Why's this important?
 - You only need to write and maintain 1 version of your functions.
 - No need to write a derivative function! Slang generates it for you.
 - Less chance of errors because it's always consistent and up-to-date.



So what is it, concretely?

Autodiff automatically computes the **exact** derivative of a function — not a finite-difference approximation, the real thing. It handles arbitrary control flow, loops, function calls. Any graphics function you write can become trainable.

And why it matters in practice: you write and maintain **one** version of your function. You never need to hand-write a derivative — Slang generates it from your original code. Which means far fewer bugs, because the derivative can't drift out of sync with the function it came from. It's always consistent, always up to date.

Forwards Mode vs. Backwards Mode



Forward Mode (fwd_diff)

Direction: Input $\rightarrow$ Output

Efficient when: Few inputs, many outputs

Computes: How outputs change w.r.t. one input at a time

Reverse Mode (bwd_diff)

Direction: Output $\rightarrow$ Input

Efficient when: Many inputs, few outputs

Computes: How one output changes w.r.t. all inputs

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

103

There are two flavors, and knowing when to use which is most of the practical skill.

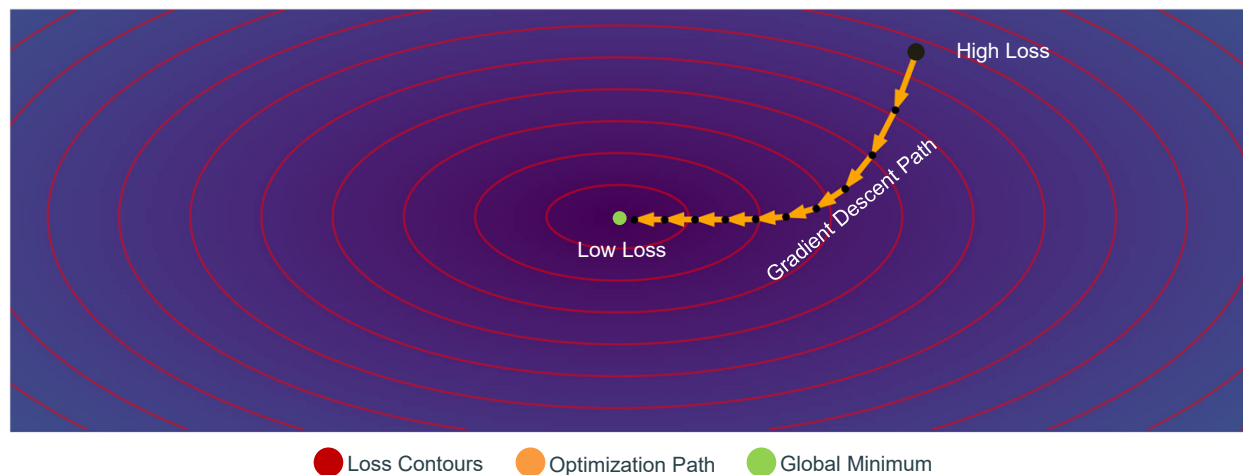
Forward mode — fwd_diff. Runs input to output. It answers: if I nudge *one* input, how does *everything* downstream change? So it's efficient when you have **few inputs and many outputs**.

Reverse mode — bwd_diff. Runs output back to input. It answers: for *one* output, how sensitive is it to *all* the inputs at once? Efficient when you have **many inputs and few outputs**.

Here's the intuition for why we care: training is the second case. You've got one number — a loss — and thousands of parameters feeding it, and you want the gradient with respect to all of them in a single pass. That's reverse mode. That's exactly what the notebook we're about to run uses.

Example: Gradient Descent

SIGGRAPH



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

104

So let's make it concrete with gradient descent — the simplest, most visual way to see gradients do something useful.

The picture: your loss surface is a bowl. You start somewhere up on the high-loss rim, red. Each step, you compute the gradient of the loss, take a small step *downhill* — opposite the gradient, scaled by a learning rate — and repeat. The yellow path walks you down to the green dot, the minimum.

The whole trick is: something has to give you that gradient at every step. Normally that's derivative code you write by hand. We're going to let Slang generate it — and watch it fit a curve to noisy data, live.

[Switch to the autodiff notebook: `slangpy_autodiff.ipynb`.]

Setup — imports, device, module (*cell 3*)

[Run it.] Same opening move as the Mandelbrot demo — import SlangPy, create a device, and load our .slang module. This module has our sigmoid math and, importantly, a differentiable loss function. We'll come back to that code.

The problem setup — noisy data (*cells 5, 7*)

[Run the jitter helper, then the data-setup cell.] Here's the game we're playing. I have a *target* sigmoid — coefficients a, b, c, d — and I generate 128 points along it, but I **jitter** the y -values with noise so it's not a clean curve. That's our ground truth data.

Then I set up a *separate* set of coefficients — the **trained** ones — starting from a deliberately bad guess, [0, 0, 0, 1]. The whole point: the trainer only ever sees the noisy *points*. It never sees the coefficients that made them. It has to recover the shape from the data alone.

Buffers — the low-level API (cell 9)

[Run it.] Notice this is the *low-level* path I mentioned earlier. Instead of the high-level dispatch, we use `create_buffer` directly and let SlangPy's **reflection** tell us the exact data layout for each buffer — `dataPoints`, `coeffs`, `lossHistory`. That's SlangPy reading the shader's own type information so the Python side and the GPU side agree, with no manual struct-packing. Both levels live side by side; you pick per situation.

See the target (cell 11)

[Run it — green plus-marks appear.] There's our noisy target data. That fuzzy S-shape is what we're trying to fit.

The starting guess (cells 13 → 14)

[Run the `compute_and_plot` definition, then call it.] `compute_and_plot` just dispatches `test_sigmoid` — evaluate the current coefficients across all the x's — and overlays the result in blue. **[Blue points appear, way off.]** And... our initial guess is *terrible*. Which is fine — it was a guess. Now let's fix it with `autodiff`.

One training step — where the autodiff actually is (cell 16)

[Show the `run_gradient_descent` definition — this is the moment to switch to the `.slang` file for 20 seconds if you want.] This loop dispatches `train_sigmoid` once per epoch. And *this* is the payoff of the whole talk. Inside that shader, in Slang, the update is basically: Wrap the coefficients in a `DifferentialPair`.

Call `bwd_diff(computeLoss)(...)` — reverse-mode, because we have one loss and four parameters.

Read the gradients back out of the pair's `.d` field.

Step each coefficient a little bit downhill.

I never wrote a derivative of `computeLoss`. I wrote the loss *once*, tagged it [`Differentiable`], and Slang generated the backward pass. That's the entire idea from slide 12, running on the GPU.

A few steps (cell 18)

[Run 5 epochs + plot.] Just five steps. Already the blue is peeling off the flat line and bending toward the data. It's moving in the right direction.

Full run (cell 20)

[Run 100 epochs + plot.] Now the full hundred. **[Fitted curve appears.]** And there's our curve — recovered purely from noisy points, no knowledge of the original coefficients. That's the sigmoid, fit by gradient descent, powered by `autodiff`.

Loss history (cell 22)

[Run it.] And the receipt: loss over time. Steep drop early, then it flattens out — converges nicely after roughly 30 epochs. That's the yellow path from the gradient-descent slide, plotted for real.

Wrap (~1–2 min)

[Back to a title/summary slide, or hold on the loss plot.]

So — to close the loop. We started by running a completely unmodified Slang shader from four lines of Python. We ended by *training* one, with a gradient we never had to write, on the GPU, in a notebook.

That's SlangPy: the graphics pipeline and the ML toolbox in the same Python session, and every function you write available in its differentiable form for free.

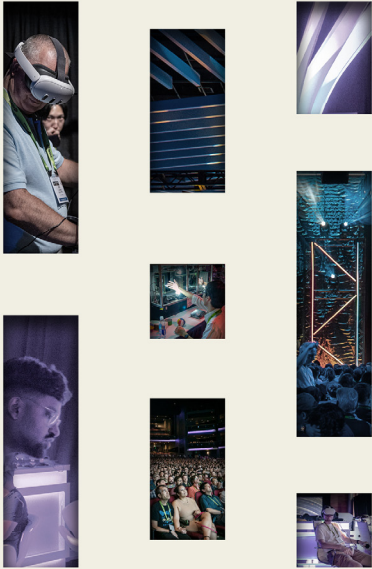
The notebooks are yours to keep — take them home, break them, retrain them on your own curves.

And now, back to Nia to wrap up the session.

The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques



Lab: Autodiff



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.



Recap

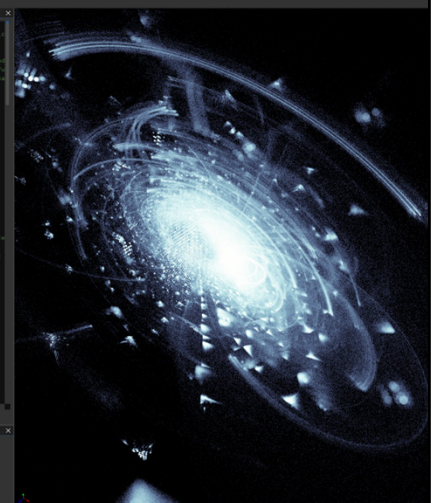
SIGGRAPH



- Language Basics
 - control flow, `inout`, types, etc.
- Using the slangc Command-Line
- Porting GLSL
- Shader I/O
- Debugging and Tools
- Structs, Modules, and Interfaces
- SlangPy
- Automatic differentiation

<https://docs.shader-slang.org>

```
File Examples Editor Shader Parameters View Help
1 // File: trivial.vert, options: shader memory optimization.
2 // Based on trivial_shader.vert from the Vulkan SDK.
3 // Purpose: to show and test operations, used by the ShaderLab
4
5 // Define this to get a single component to avoid by reducing about
6 // divergence by using shared memory, at the expense of a more diffi
7 // task for some of the code. For the purpose of this demo, we
8 // follow for an optimization of the this code.
9 #define SHARED_MEMORY
10
11 static const int BLOCK_SIZE_X = 16;
12 static const int BLOCK_SIZE_Y = 16;
13 //This must be a power of 2.
14 static const int BLOCK_SIZE = BLOCK_SIZE_X * BLOCK_SIZE_Y;
15
16 #ifdef SHARED_MEMORY
17
18 groupshared float4 block_particle[BLOCK_SIZE];
19
20 #endif
21
22 uniform float tTime; // in seconds
23 uniform vec3 fPosition; // screen size
24 uniform float fResolution;
25
26 // If fResolution is not 0.0, then we use it to scale the size to
27 // using screen.
28 [shader(100)] globalUniform int numParticles; // number of particles
29 // We use (float) here just to show off an int to float conversion.
30 [shader(100)] int numParticles; // number of particles
31
32 // The model matrix, projection matrix
33 uniform float4x4 mModel;
34 uniform float4x4 mProj;
35 uniform float4x4 mView;
36 float4 projectParticle(float p, out float particle_depth)
37 {
38     float4 x = mul(float4(p, 1.0), mModel);
39     particle_depth = -x.z;
40     x = mul(x, mProj);
41     return x.xy / w.x;
42 }
43
44 // Random number generation
```



© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

Thank you, Chris. Now let's go ahead and close out this lab.

We started with language basics, control flow like `if` and `for` loops, `inout` qualifiers on parameters, types, and so on.

Then we showed how to use the lower-level `slangc` command line to compile shaders to different languages, as well as some of the options it has.

Then we covered porting GLSL, which only involves a few changes to your code, shader I/O, and how Slang makes this easier, debugging tools,

as well as the demo of how Slang worked with RenderDoc, structs, modules, interfaces, and even a bit of generics.

Then Chris covered SlangPy, and finally, automatic differentiation.

The official Slang docs are in this link below, which have a lot more information about all of the topics that we talked about.

Thank you!

SIGGRAPH



- Lab materials: <https://shader-slang.org/landing/siggraph-26>
- Related sessions:
 - **Hands-On Machine Learning with Slang: Building High-Performance GPU Inference Pipelines**
 - Immediately after this session, same place!
 - **Birds of a Feather: Real-Time Shading**
 - Today, 1:00 – 2:30pm, Room 518
 - **Applied Research for Ray Tracing Massive Scenes in Real Time**
 - Wednesday, 9:00 – 9:20 am, Room 502A
 - **Birds of a Feather: Slang in Action**
 - Wednesday, 1:30 – 2:00pm, JW Marriott, Plaza 1



<https://shader-slang.org/landing/siggraph-26>

© 2026 SIGGRAPH. ALL RIGHTS RESERVED.

If you've enjoyed Slang, there are a few more SIGGRAPH sessions this week building on this course.

Thank you very much!





These slides aren't shown during the course, but maybe they'll be interesting for you if you're reading this afterwards.



New!

```
struct Color
{
    float3 linear;
    property float3 srgb
    {
        get { return toSrgb(linear); }
        set { linear = toLinear(newValue); }
    }
}
```

```
Color c;
c.srgb = float3(.97, .39, .19);
return float4(c.linear, 1.0);
```

In Slang, you can also add properties to classes, which look like member variables but are in fact a getter and a setter function. Here I have a struct that holds a linear RGB color. If I wanted to add a way to automatically convert sRGB colors, I'd usually have to add a `getSrgb` function and a `setSrgb` function. Here I'm doing this instead with a property, which lets me write this really concise code on the right.



```
external/lib1/sdf.slang
```

```
struct SdfInfo
{
    float t;
    uint* data;
}
```

```
external/lib2/math.slang
```

```
T min<T>(T a, T b)
    where T : IComparable
{ ... }
```

- How can we make SdfInfo work with `min()` if we can't modify `external/lib1/sdf.slang`?

New!

```
// Extend SdfInfo so it conforms to IComparable
extension SdfInfo : IComparable
{
    bool equals(SdfInfo other) { return t == other.t && data == other.data; }
    bool lessThan(SdfInfo other) { return t < other.t; }
    bool lessThanOrEquals(SdfInfo other) { return lessThan(other) || equals(other); }
}
```

Finally, a neat thing is that you can tack on interfaces to structs after the fact. Imagine you're writing code that depends on two external libraries. One defines an `SdfInfo` struct, but doesn't implement any interfaces for it. And another defines a `min()` function that works for types that implement `IComparable`, which is Slang's built-in interface for types you can use less-than, equals, and greater-than operators on.

(click) If you want to use the `SdfInfo` struct from library 1 with `min()` from library 2, normally in C++ you'd be out of luck and have to modify library 1.

But in Slang, **(click)** you can add on an `IComparable` implementation to `SdfInfo` like this!