SIGGRAPH
2025
The Premier Conference & Exhibition on
Computer Graphics & Interactive Techniques

HANDS-ON CLASS

INTRODUCTION TO SLANG

THE NEXT GENERATION SHADING LANGUAGE

NIA BICKFORD, CHRIS HEBERT, AND TRISTAN LORACH
NVIDIA

Hello and welcome to our Slang workshop!

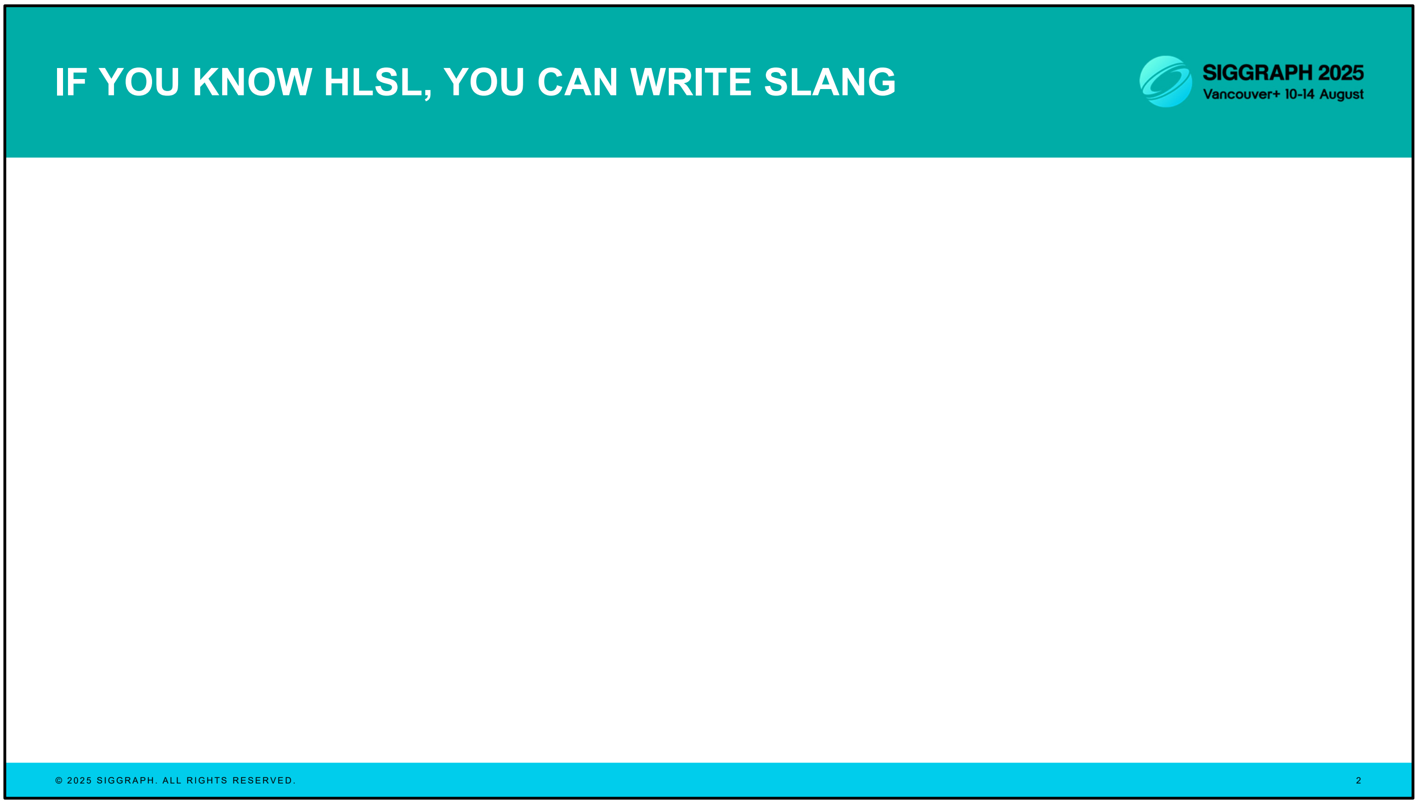
In today's lab session, Chris Hebert and I are going to show you how to use Slang, the next-generation shading language.

Since this lab is interactive, we'll have you write and compile Slang programs on the computers in front of you, 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.

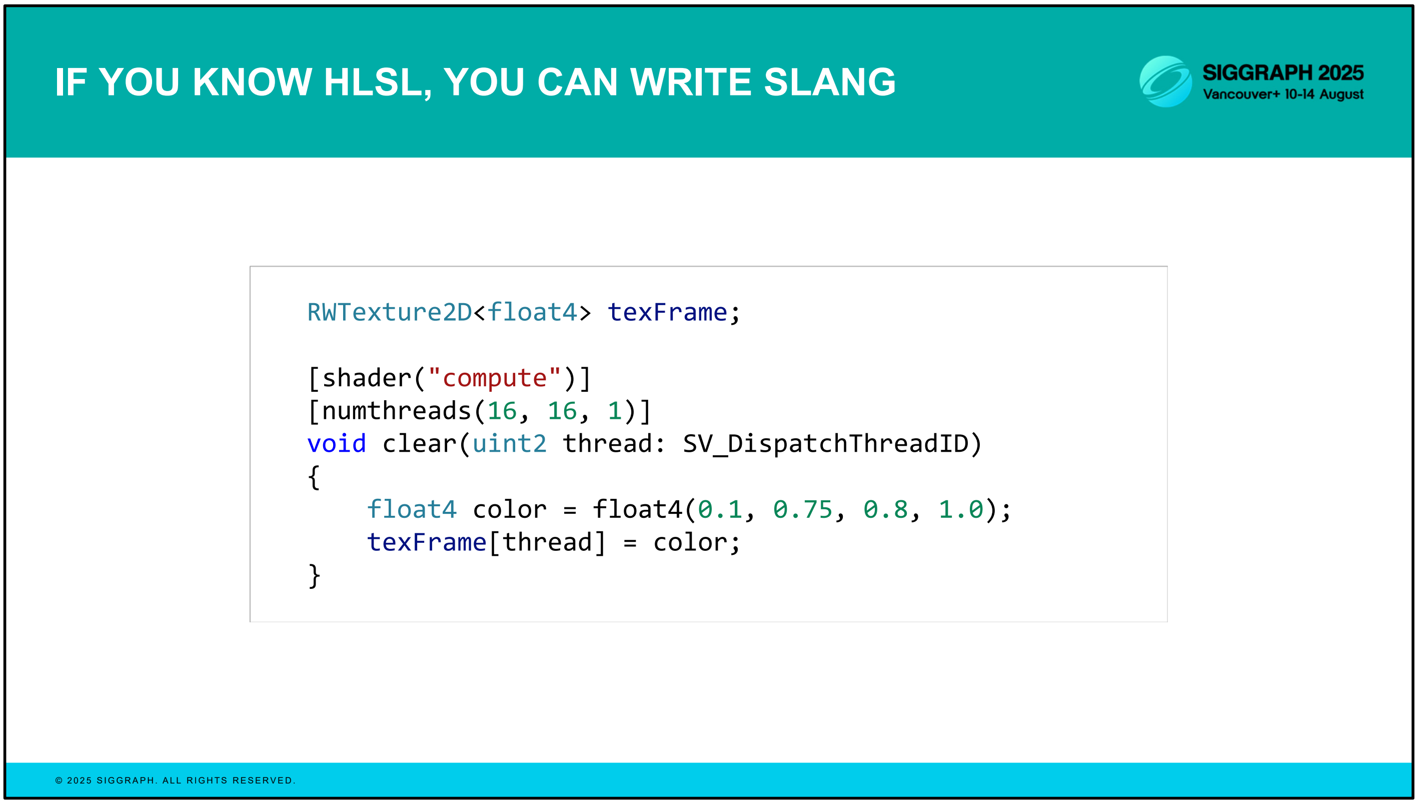
PRESENTER SETUP:

- 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

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

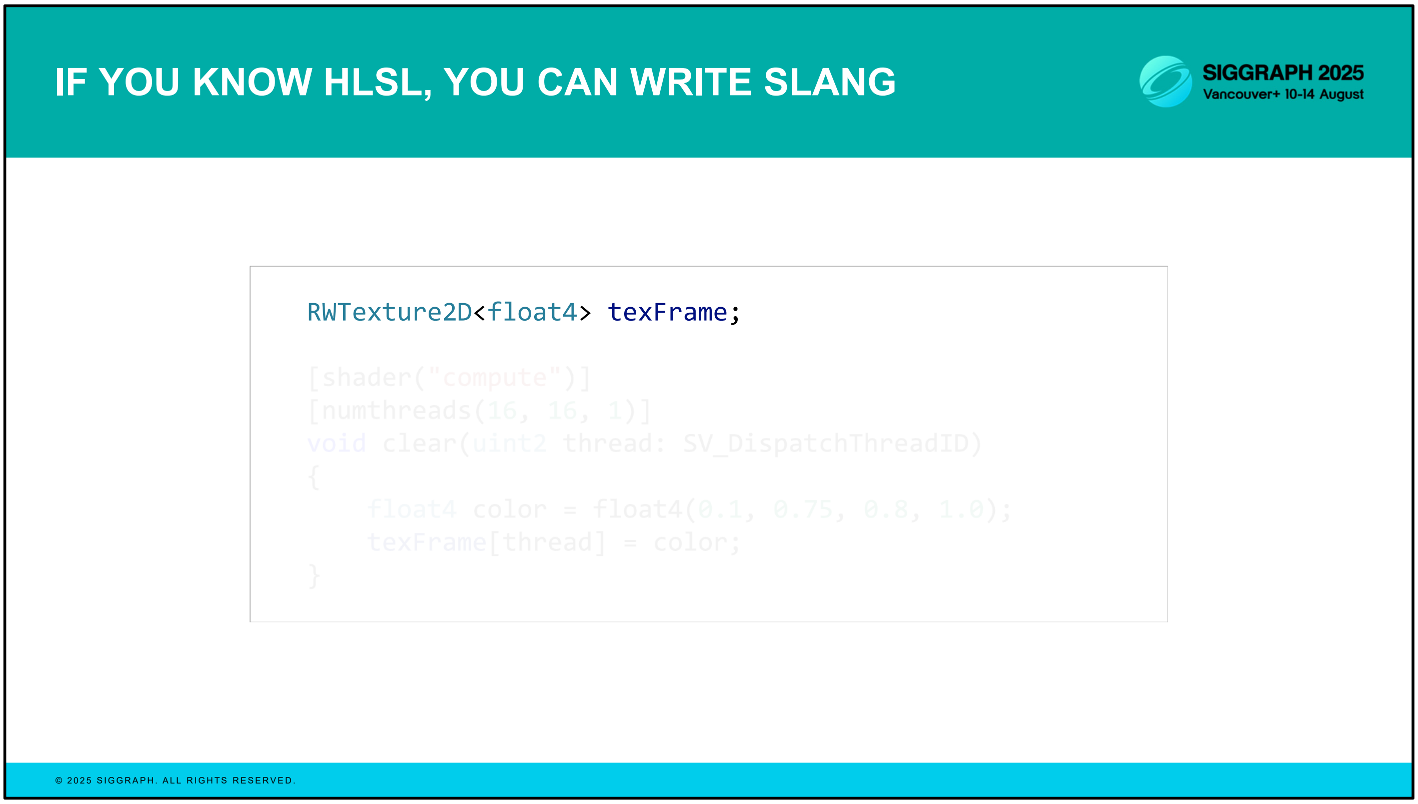


IF YOU KNOW HLSL, YOU CAN WRITE SLANG

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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.

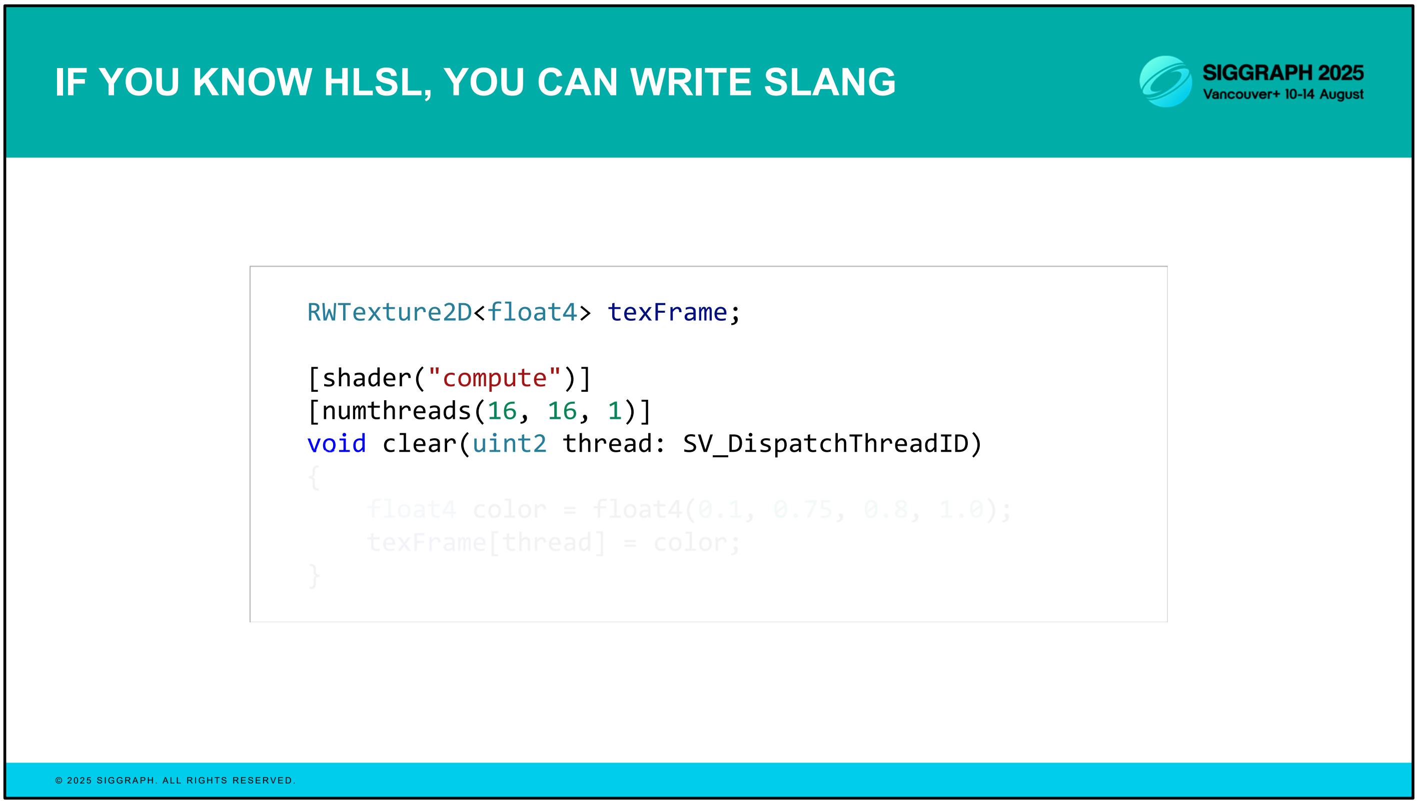


IF YOU KNOW HLSL, YOU CAN WRITE SLANG

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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.

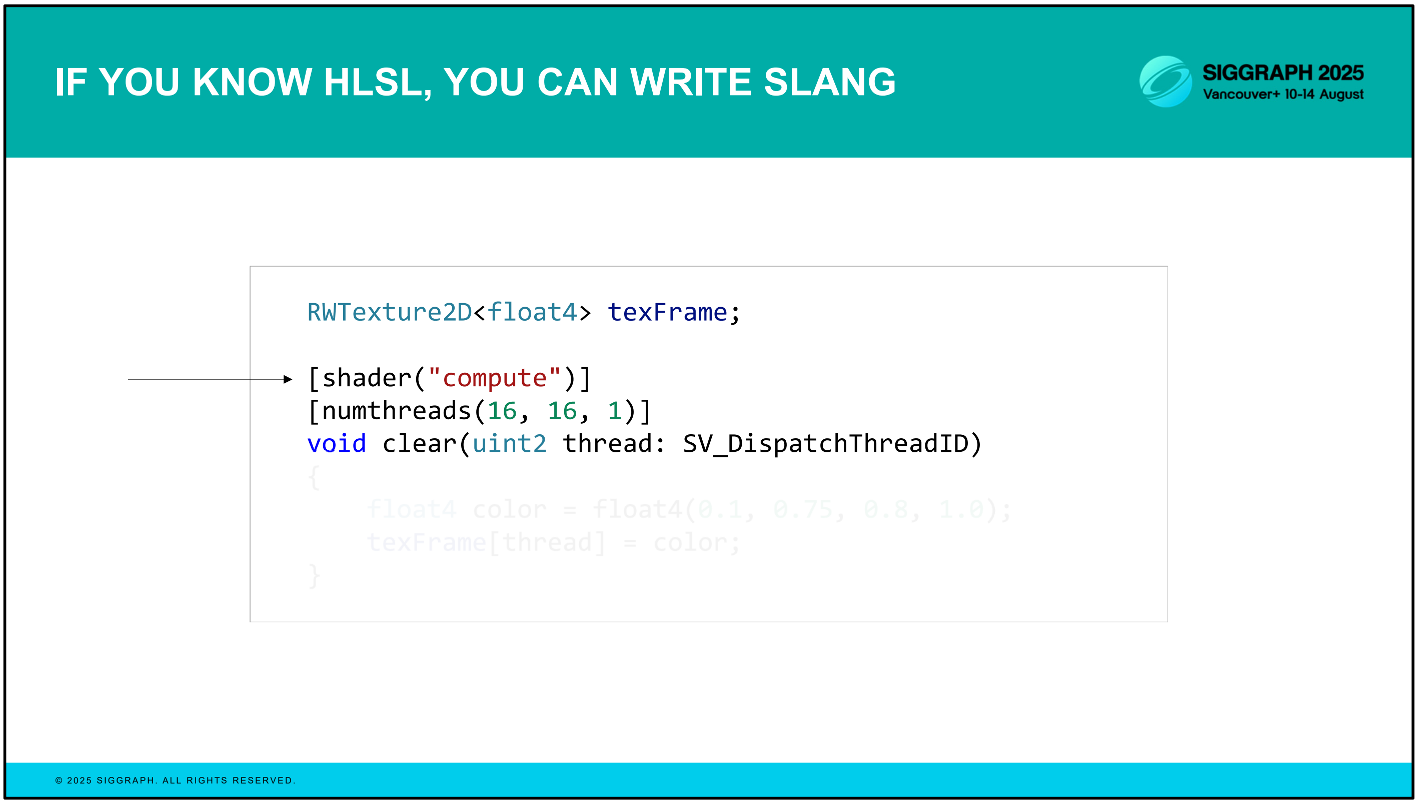


IF YOU KNOW HLSL, YOU CAN WRITE SLANG

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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*.

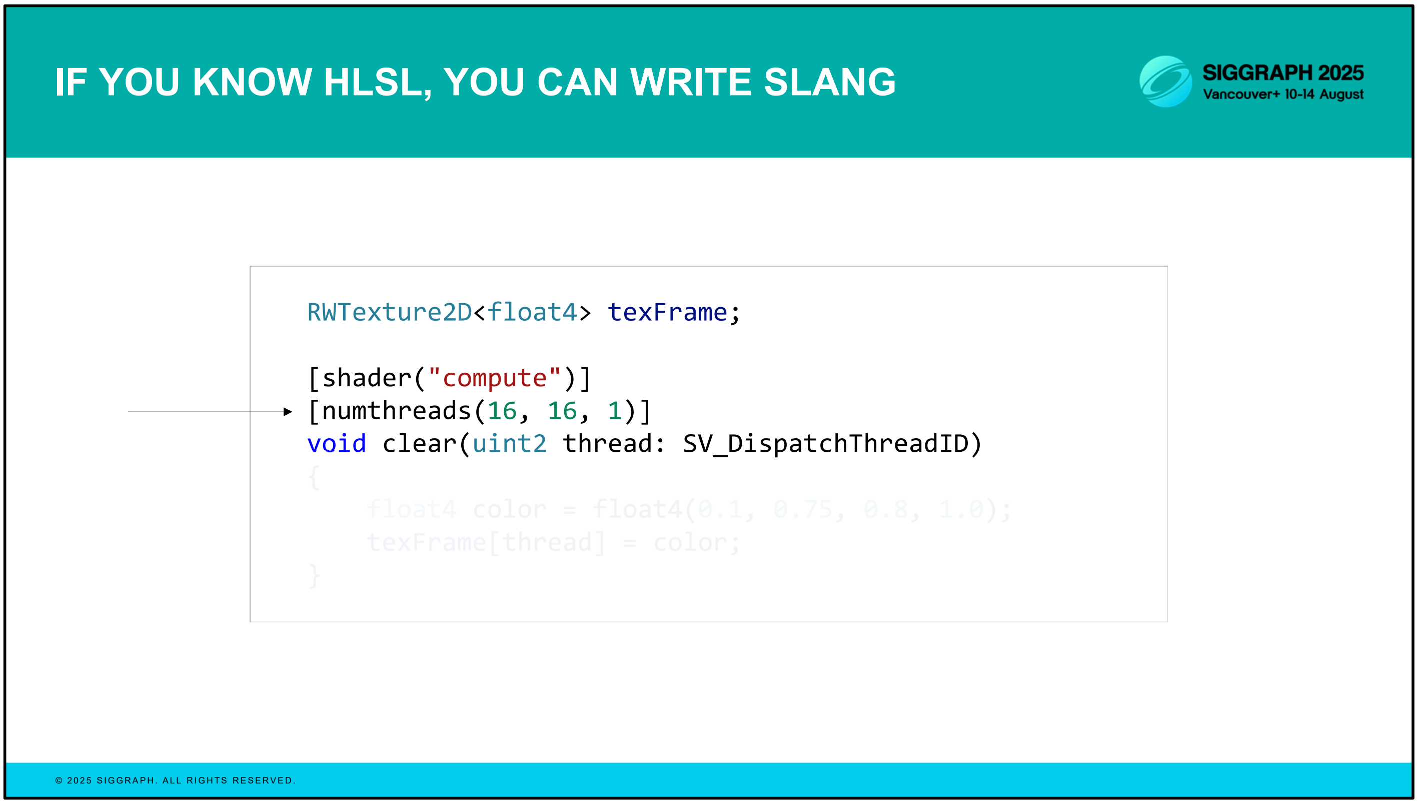


IF YOU KNOW HLSL, YOU CAN WRITE SLANG

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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,

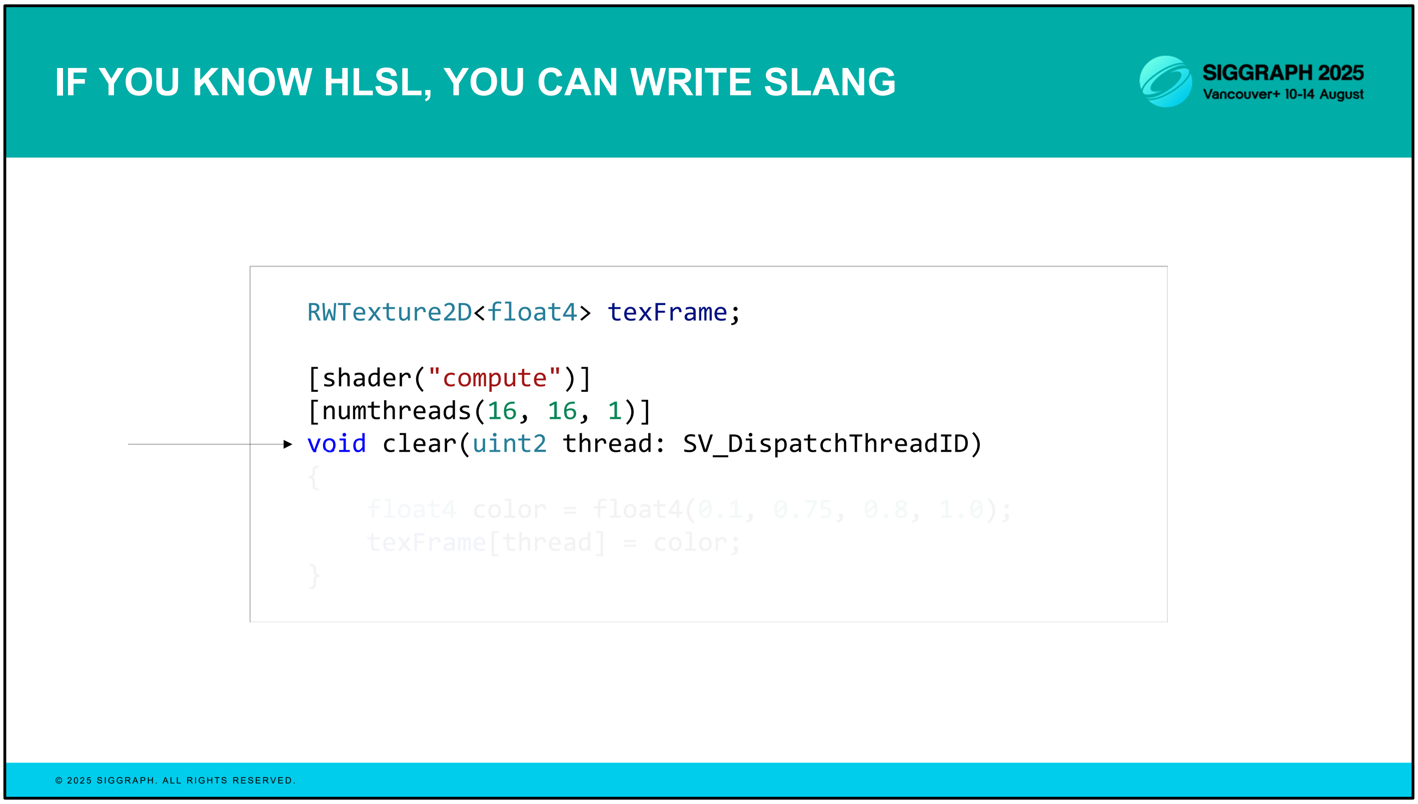


IF YOU KNOW HLSL, YOU CAN WRITE SLANG

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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.

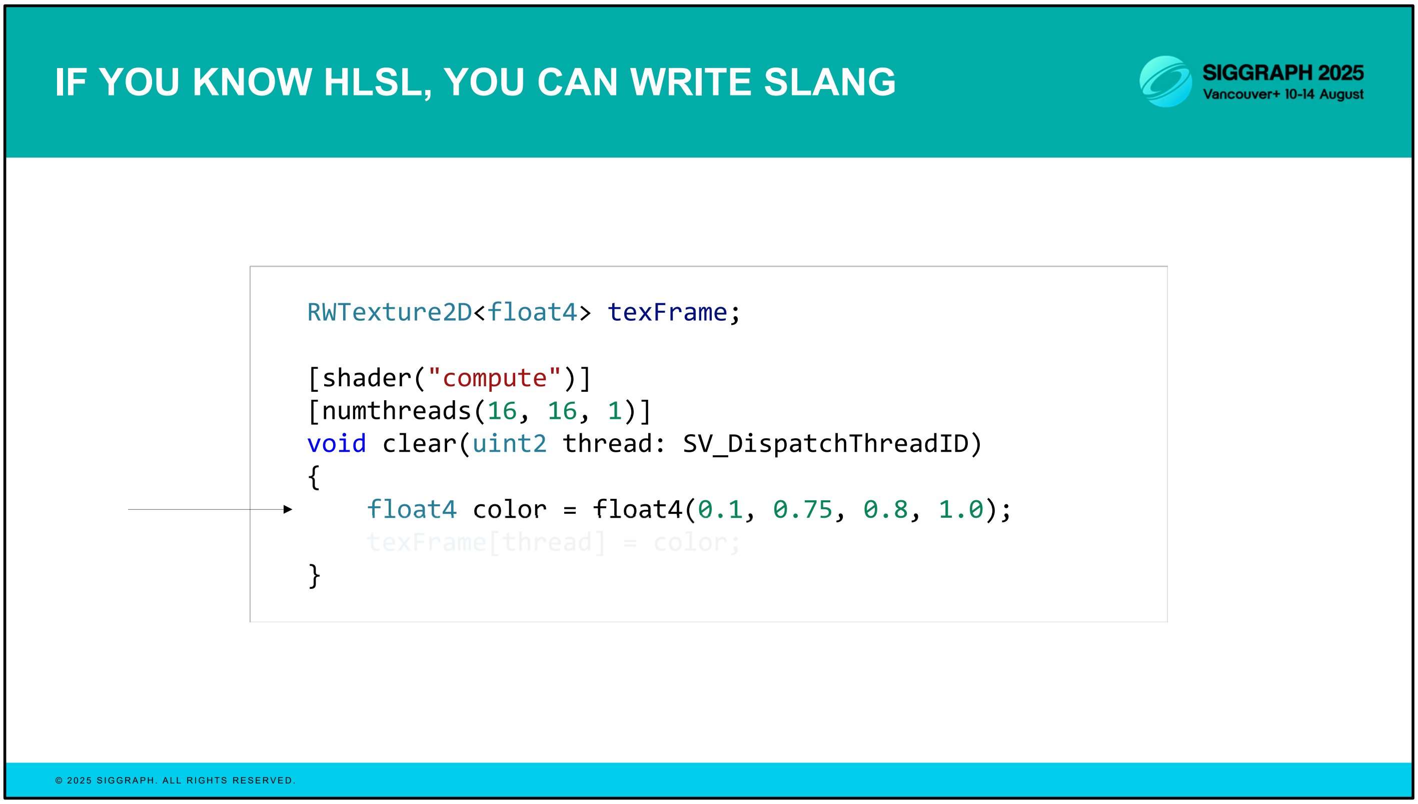


IF YOU KNOW HLSL, YOU CAN WRITE SLANG

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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.

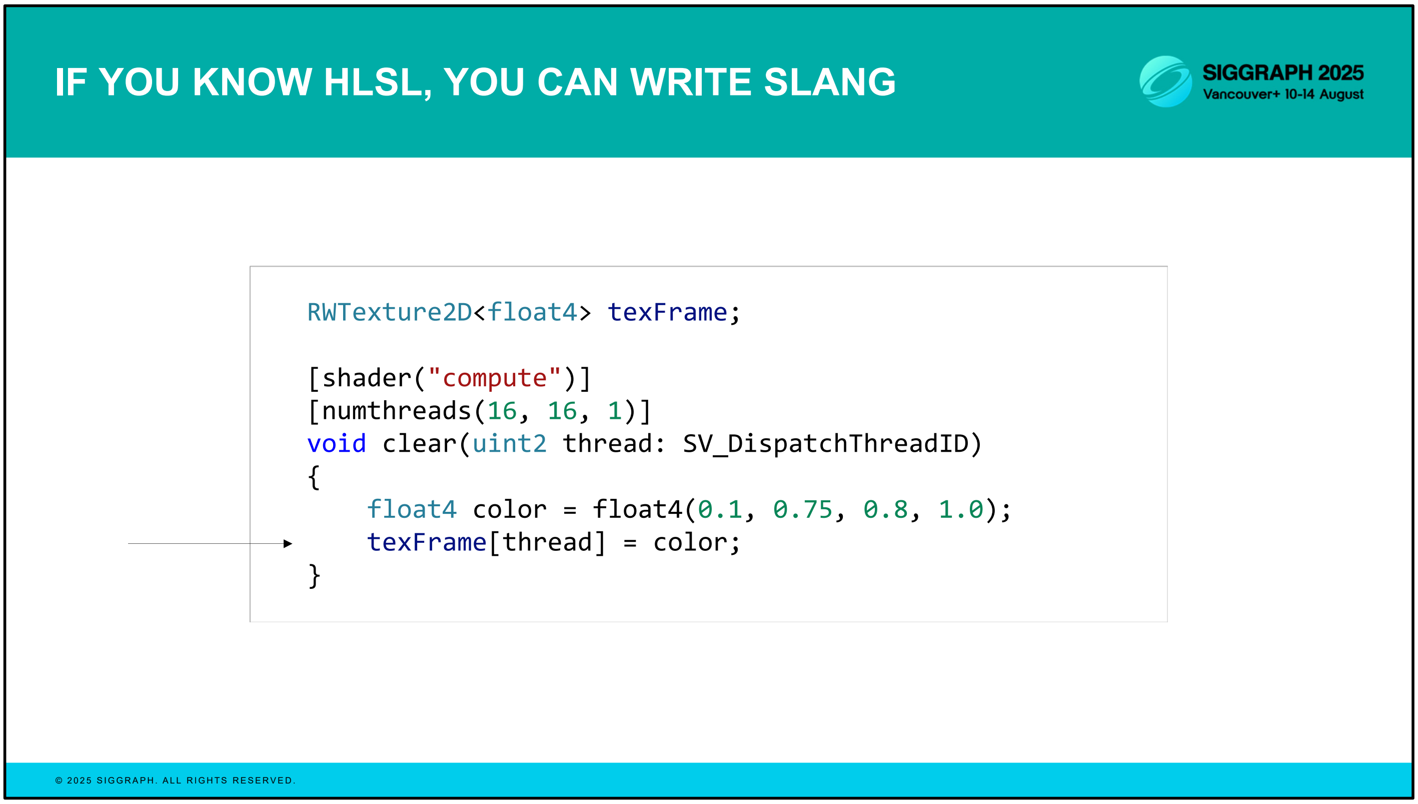


IF YOU KNOW HLSL, YOU CAN WRITE SLANG

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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.

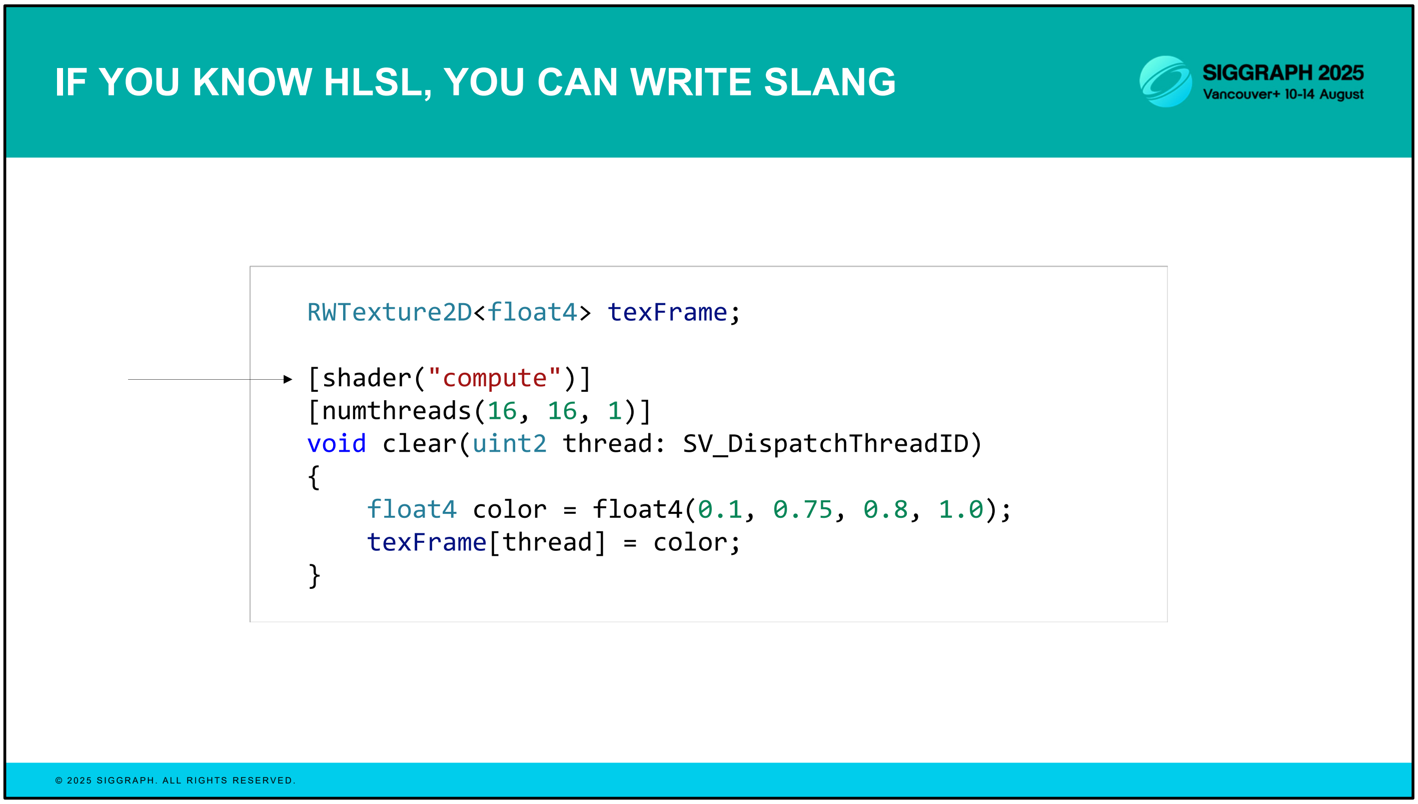


IF YOU KNOW HLSL, YOU CAN WRITE SLANG

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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.



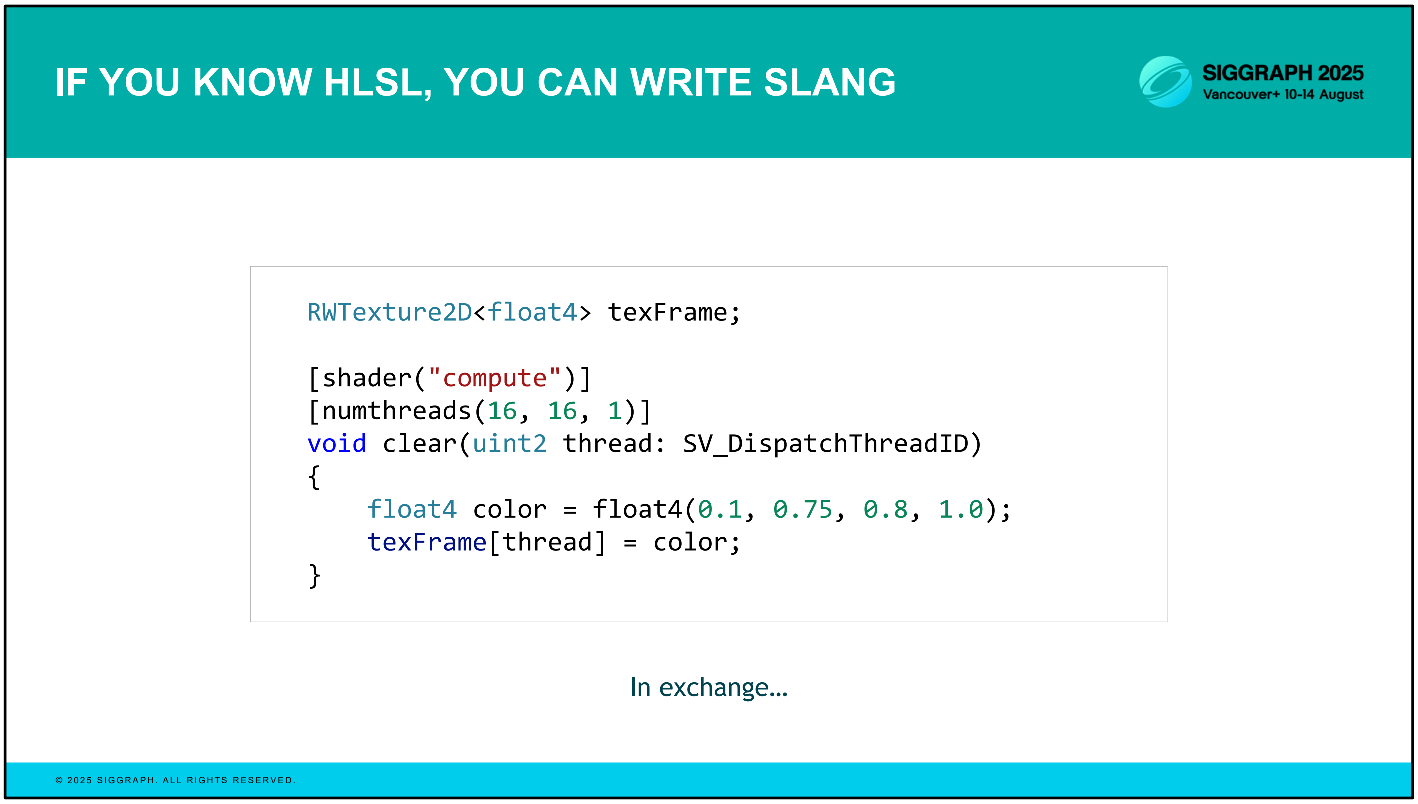
IF YOU KNOW HLSL, YOU CAN WRITE SLANG

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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!



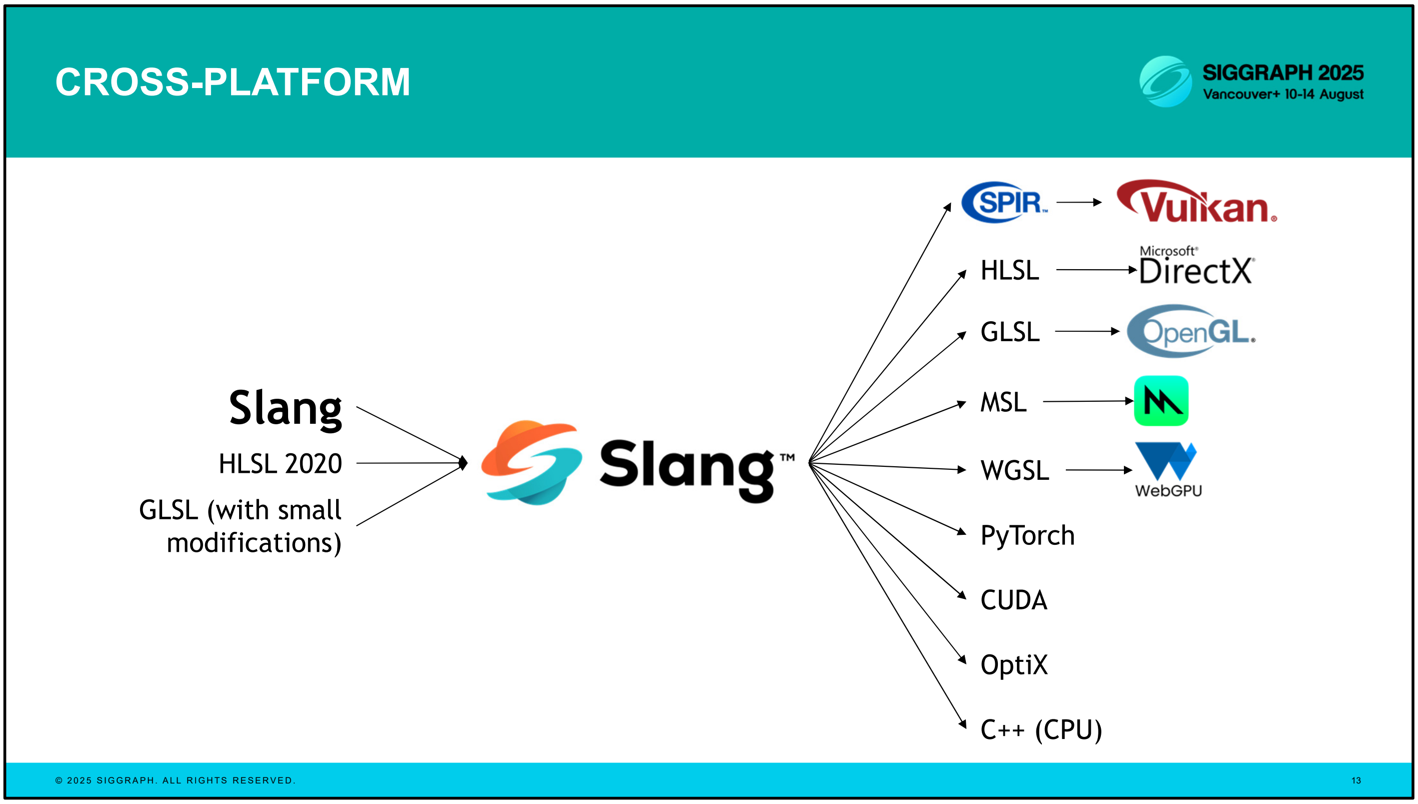
IF YOU KNOW HLSL, YOU CAN WRITE SLANG

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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.



CROSS-PLATFORM

Slang

HLSL 2020

GLSL (with small modifications)

Slang™

SPIR → Vulkan

HLSL → DirectX

GLSL → OpenGL

MSL

WGSL → WebGPU

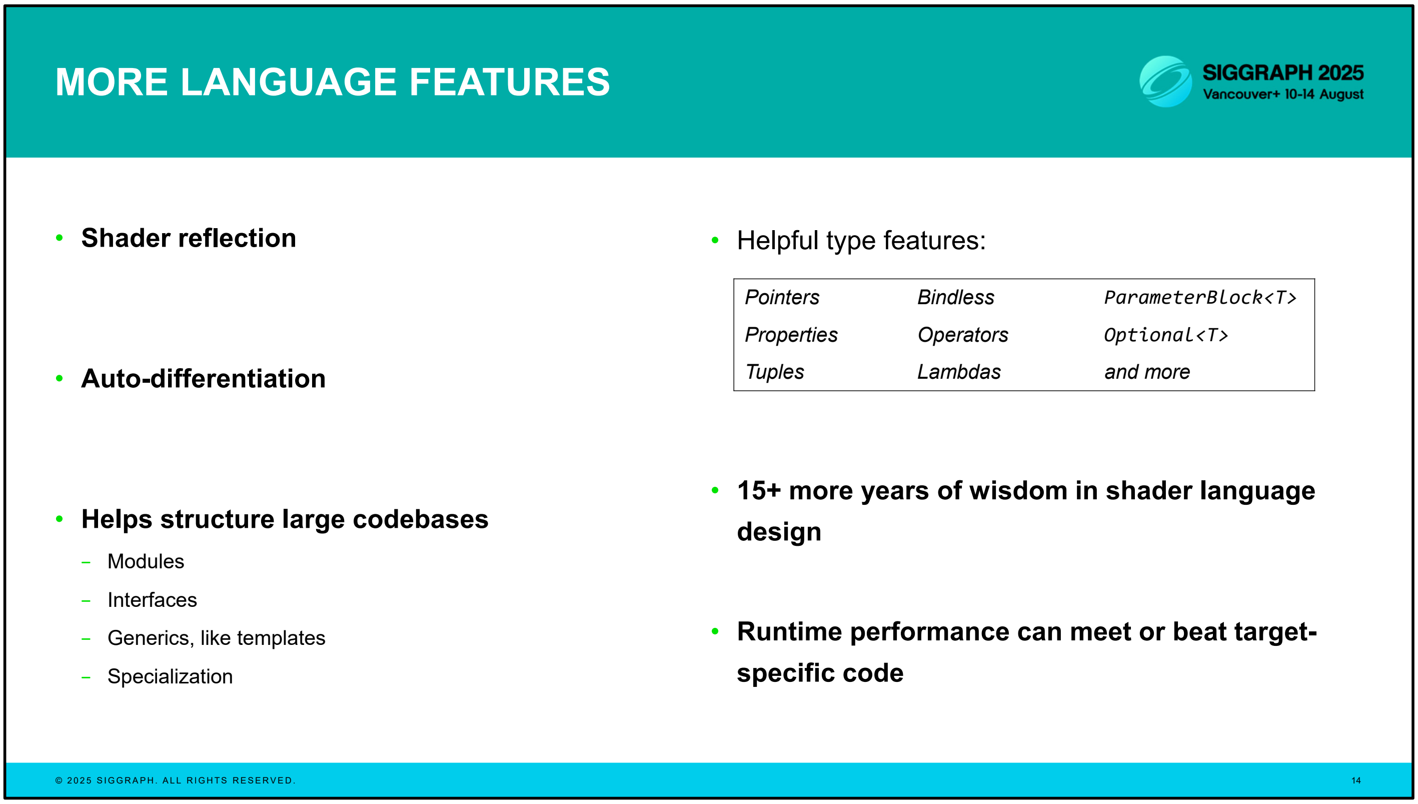
PyTorch

CUDA

OptiX

C++ (CPU)

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 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 2025
Vancouver+ 10-14 August

- **Shader reflection**
- **Auto-differentiation**
- **Helps structure large codebases**
  - Modules
  - Interfaces
  - Generics, like templates
  - Specialization
- Helpful type features:

Pointers	*Bindless*	`ParameterBlock<T>`
Properties	*Operators*	`Optional<T>`
Tuples	*Lambdas*	*and more*

- **15+ more years of wisdom in shader language design**
- **Runtime performance can meet or beat target-specific code**

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.

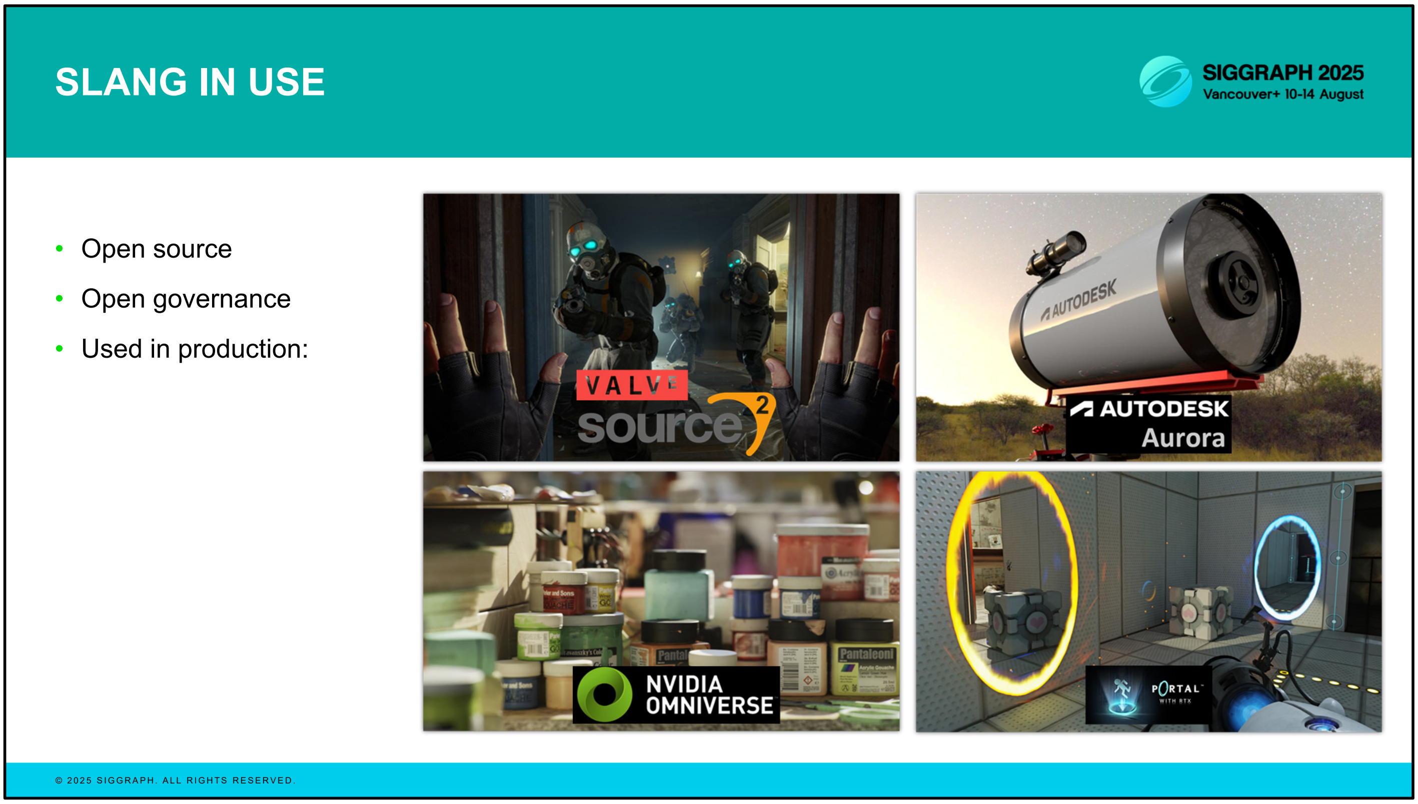
Auto-differentiation, which Chris will talk about.

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 2025
Vancouver+ 10-14 August

- Open source
- Open governance
- Used in production:

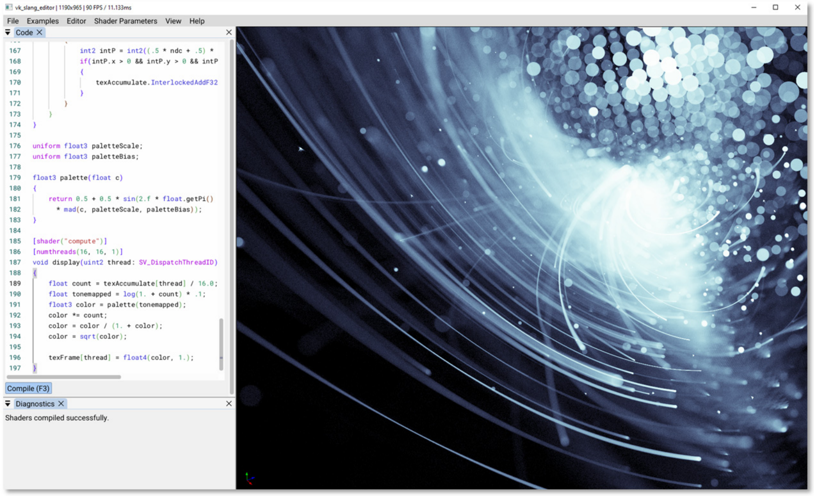
Slang is also open-source, and an open-governance project under the Khronos Group.

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 2025
Vancouver+ 10-14 August

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



```
            int2 intP = int2((.5 * ndc + .5) *
            if(intP.x > 0 && intP.y > 0 && intP
            {
                texAccumulate.InterlockedAddF32
            }
        }
    }
}

uniform float3 paletteScale;
uniform float3 paletteBias;

float3 palette(float c)
{
    return 0.5 + 0.5 * sin(2.f * float.getPi()
      * mad(c, paletteScale, paletteBias));
}

[shader("compute")]
[numthreads(16, 16, 1)]
void display(uint2 thread: SV_DispatchThreadID)
{
    float count = texAccumulate[thread] / 16.0;
    float tonemapped = log(1. + count) * .1;
    float3 color = palette(tonemapped);
    color *= count;
    color = color / (1. + color);
    color = sqrt(color);

    texFrame[thread] = float4(color, 1.);
}
```

Shaders compiled successfully.

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.

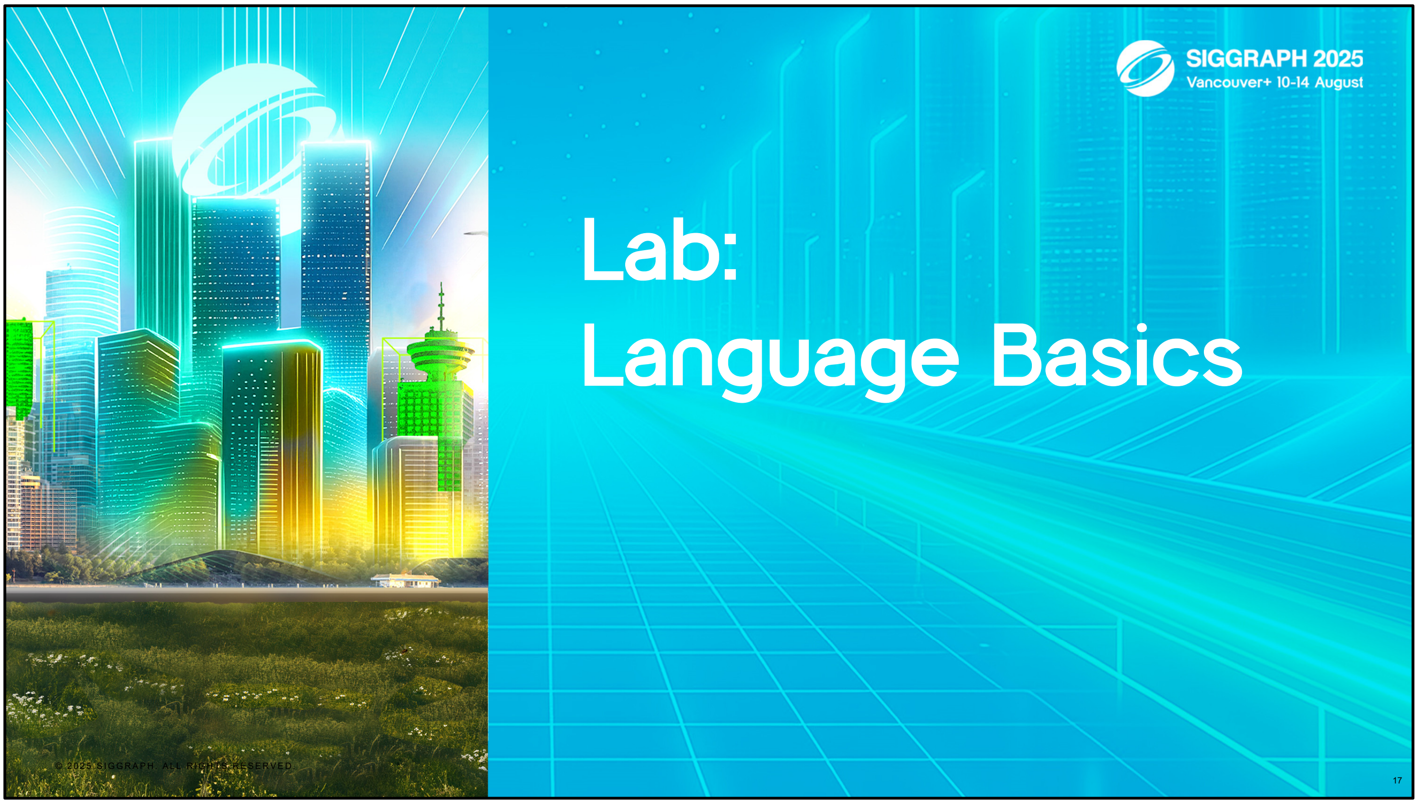
Passing data to and from shaders is an important topic, and one that Slang really innovates in.

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!



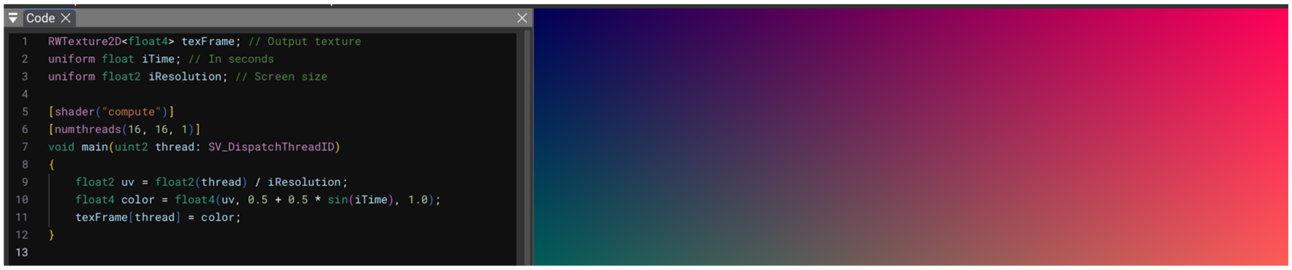
SIGGRAPH 2025
Vancouver+ 10-14 August

Lab:
Language Basics

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

LAB: GETTING STARTED

- *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:*



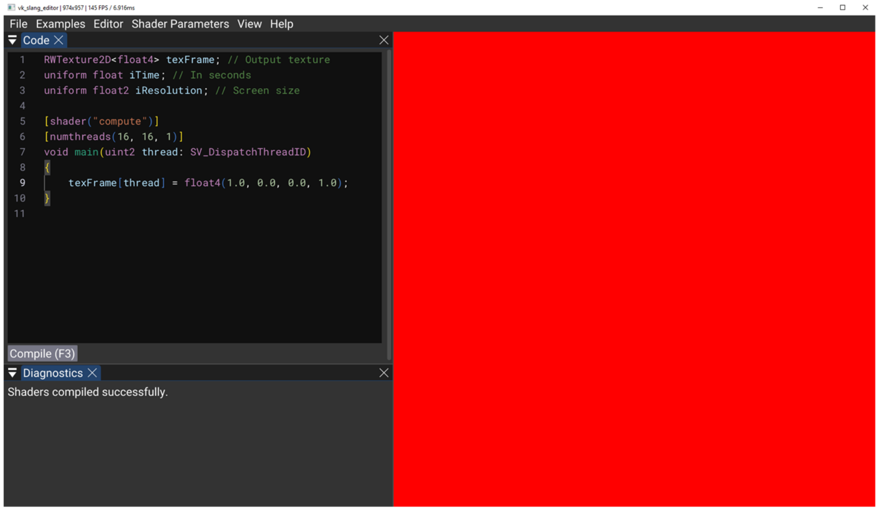
```
RWTexture2D<float4> texFrame; // Output texture
uniform float iTime; // In seconds
uniform float2 iResolution; // Screen size

[shader("compute")]
[numthreads(16, 16, 1)]
void main(uint2 thread: SV_DispatchThreadID)
{
    float2 uv = float2(thread) / iResolution;
    float4 color = float4(uv, 0.5 + 0.5 * sin(iTime), 1.0);
    texFrame[thread] = color;
}
```

We’ve loaded a dev environment onto each of the 60 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 2025
Vancouver+ 10-14 August

- Clear it to a solid color:


vk_slang_editor | 974x957 | 145 FPS / 6.916ms
File Examples Editor Shader Parameters View Help
Code
Compile (F3)
Diagnostics

```
RWTexture2D<float4> texFrame; // Output texture
uniform float iTime; // In seconds
uniform float2 iResolution; // Screen size

[shader("compute")]
[numthreads(16, 16, 1)]
void main(uint2 thread: SV_DispatchThreadID)
{
    texFrame[thread] = float4(1.0, 0.0, 0.0, 1.0);
}
```

```
Shaders compiled successfully.
```

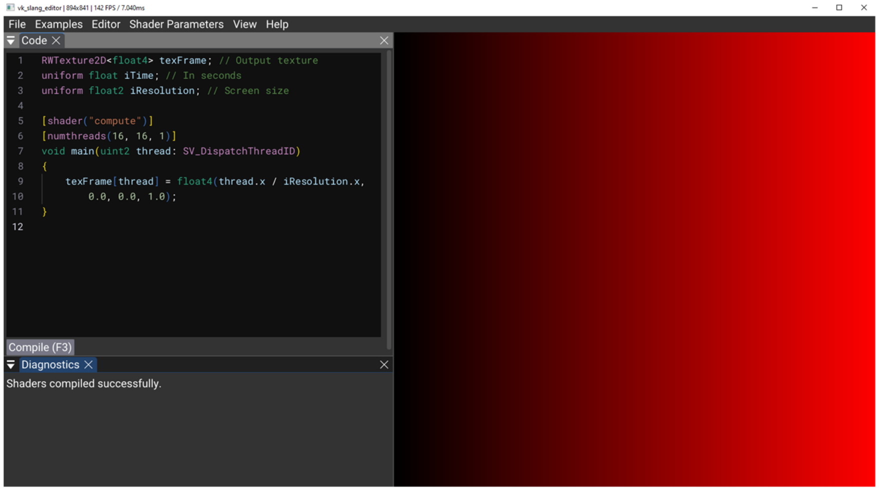
Let’s start with something simple. Select the content of the main function, in between the curly braces, and delete it. Let’s set all pixels of texFrame to a solid color. Etc.

Then press F3 or click on the “Compile” button to compile. You should get a red screen, like this.

LAB: GETTING STARTED

SIGGRAPH 2025
Vancouver+ 10-14 August

- Draw a gradient:


vk_slang_editor | 894x841 | 142 FPS / 7.040ms
File Examples Editor Shader Parameters View Help
Code
Compile (F3)
Diagnostics

```
RWTexture2D<float4> texFrame; // Output texture
uniform float iTime; // In seconds
uniform float2 iResolution; // Screen size

[shader("compute")]
[numthreads(16, 16, 1)]
void main(uint2 thread: SV_DispatchThreadID)
{
    texFrame[thread] = float4(thread.x / iResolution.x,
        0.0, 0.0, 1.0);
}
```

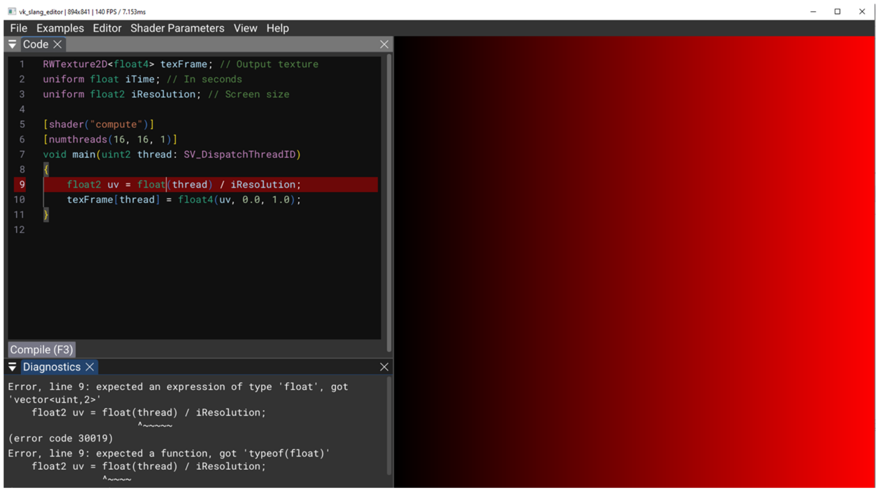
```
Shaders compiled successfully.
```

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:

(Make the mistake here to show diagnostics)

LAB: GETTING STARTED

- Intentionally make a mistake to show diagnostics:



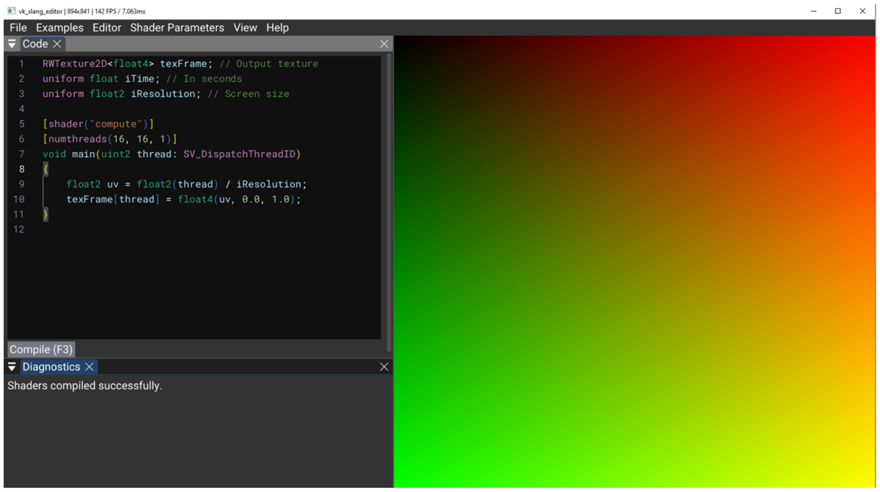
```
1   RWTexture2D<float4> texFrame; // Output texture
2   uniform float iTime; // In seconds
3   uniform float2 iResolution; // Screen size
4
5   [shader("compute")]
6   [numthreads(16, 16, 1)]
7   void main(uint2 thread: SV_DispatchThreadID)
8   {
9       float2 uv = float(thread) / iResolution;
10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

```
Error, line 9: expected an expression of type 'float', got
'vector<uint,2>'
    float2 uv = float(thread) / iResolution;
                     ^~~~~~
(error code 30019)
Error, line 9: expected a function, got 'typeof(float)'
    float2 uv = float(thread) / iResolution;
                ^~~~~
```

LAB: GETTING STARTED

SIGGRAPH 2025
Vancouver+ 10-14 August

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



```
RWTexture2D<float4> texFrame; // Output texture
uniform float iTime; // In seconds
uniform float2 iResolution; // Screen size

[shader("compute")]
[numthreads(16, 16, 1)]
void main(uint2 thread: SV_DispatchThreadID)
{
    float2 uv = float2(thread) / iResolution;
    texFrame[thread] = float4(uv, 0.0, 1.0);
}
```

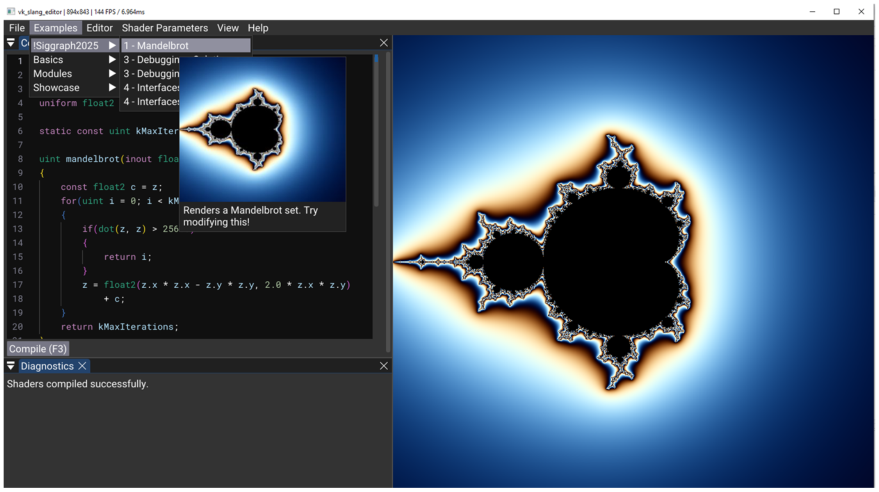
```
Shaders compiled successfully.
```

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.

LAB: GETTING STARTED

SIGGRAPH 2025
Vancouver+ 10-14 August

- Load Mandelbrot set example:



```
uniform float2

static const uint kMaxIter

uint mandelbrot(inout floa
{
    const float2 c = z;
    for(uint i = 0; i < kM
    {
        if(dot(z, z) > 256
        {
            return i;
        }
        z = float2(z.x * z.x - z.y * z.y, 2.0 * z.x * z.y)
            + c;
    }
    return kMaxIterations;
```

Renders a Mandelbrot set. Try modifying this!

Shaders compiled successfully.

Now let’s skip ahead a bit to show off some more standard features. In the menu bar, click on Examples > !Siggraph2025 > 1 – Mandelbrot.

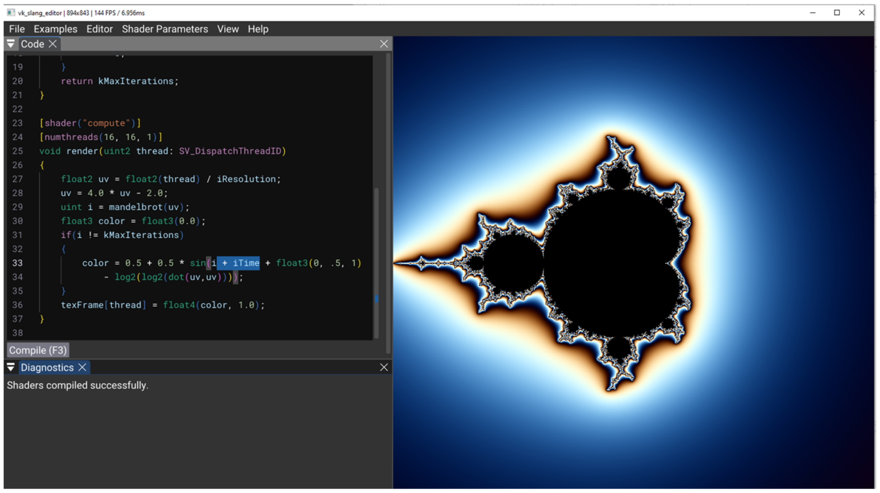
Point out:

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

LAB: GETTING STARTED

- Use `uniform float iTime` to cycle colors:

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



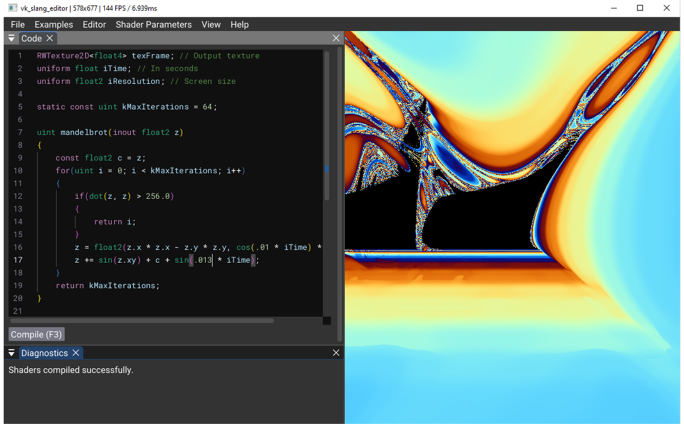
```
    }
    return kMaxIterations;
}

[shader("compute")]
[numthreads(16, 16, 1)]
void render(uint2 thread: SV_DispatchThreadID)
{
    float2 uv = float2(thread) / iResolution;
    uv = 4.0 * uv - 2.0;
    uint i = mandelbrot(uv);
    float3 color = float3(0.0);
    if(i != kMaxIterations)
    {
        color = 0.5 + 0.5 * sin(i + iTime + float3(0, .5, 1)
            - log2(log2(dot(uv,uv))));
    }
    texFrame[thread] = float4(color, 1.0);
}
```

```
Shaders compiled successfully.
```

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!

- **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* > *_SIGGRAPH2025* > *Mandelbrot*



```
1   RWTexture2D<float4> texFrame; // Output texture
2   uniform float iTime; // In seconds
3   uniform float2 iResolution; // Screen size
4
5   static const uint kMaxIterations = 64;
6
7   uint mandelbrot(inout float2 z)
8   {
9       const float2 c = z;
10      for(uint i = 0; i < kMaxIterations; i++)
11      {
12          if(dot(z, z) > 256.0)
13          {
14              return i;
15          }
16          z = float2(z.x * z.x - z.y * z.y, cos(.01 * iTime) *
17          z += sin(z.xy) + c + sin(.013 * iTime);
18      }
19      return kMaxIterations;
20  }
21
```

```
Shaders compiled successfully.
```

Try this on line 9!

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

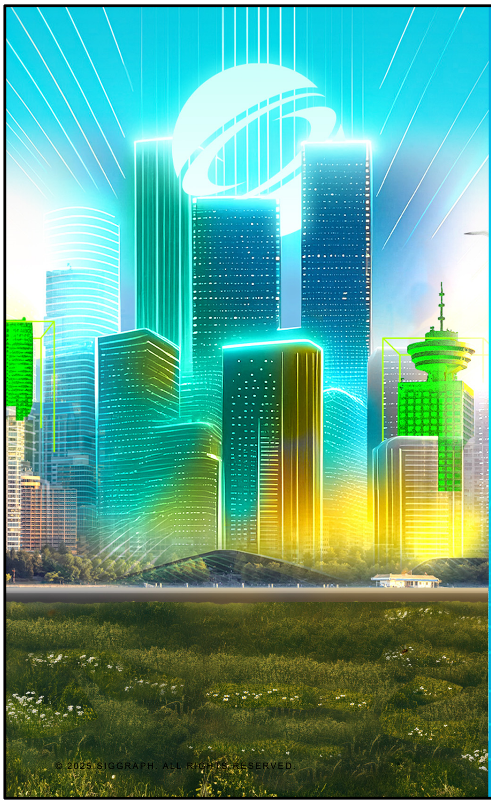
Now, I'd like you to modify the shader on your own! The goal of this exercise is to get you practice with Slang by making changes to the code, compiling, and seeing the result. So I'd like you to try breaking the shader, changing the control flow, modifying the equations and experimentally seeing what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

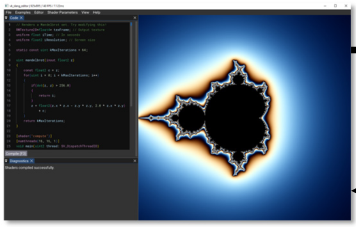


Using slangc

OVERVIEW

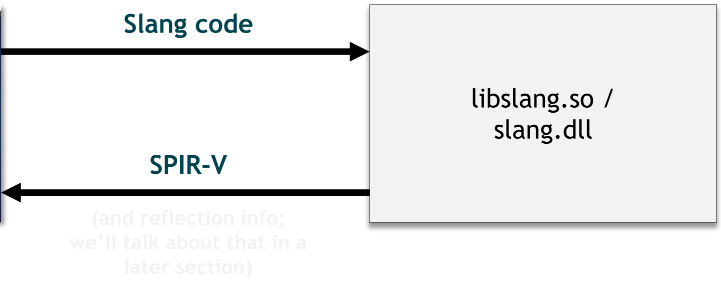
SIGGRAPH 2025
Vancouver+ 10-14 August

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



```
// Renders a Mandelbrot set. Try modifying this!
RWTexture2D<float4> texFrame; // Output texture
uniform float iTime; // In seconds
uniform float2 iResolution; // Screen size

static const uint kMaxIterations = 64;

uint mandelbrot(inout float2 z)
{
    const float2 c = z;
    for(uint i = 0; i < kMaxIterations; i++)
    {
        if(dot(z, z) > 256.0)
        {
            return i;
        }
        z = float2(z.x * z.x - z.y * z.y, 2.0 * z.x * z.y)
            + c;
    }
    return kMaxIterations;
}

[shader("compute")]
[numthreads(16, 16, 1)]
void main(uint2 thread: SV_DispatchThreadID)
```

Shaders compiled successfully.



Slang code

libslang.so / slang.dll

SPIR-V

(and reflection info; we'll talk about that in a later section)

- 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.*

SIGGRAPH 2025
Vancouver+ 10-14 August

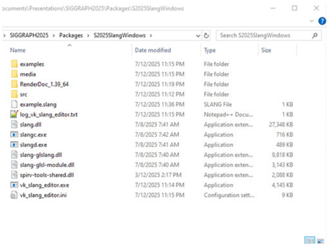
Lab: Compiling with `slangc`

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

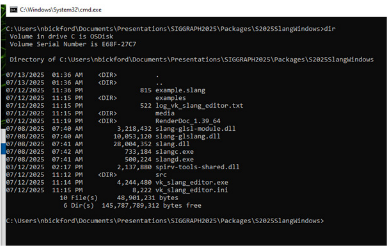
LAB: SLANGC

SIGGRAPH 2025
Vancouver+ 10-14 August

- 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 `ls` to make sure you're in the right one
- *(Screenshots show Windows but will be switched to Linux)*


ocuments\Presentations\SIGGRAPH2025\Packages\S2025SlangWindows
SIGGRAPH2025 › Packages › S2025SlangWindows ›
Search S2025SlangWindows

Name	Date modified	Type	Size
examples	7/12/2025 11:15 PM	File folder	
media	7/12/2025 11:15 PM	File folder	
RenderDoc_1.39_64	7/12/2025 11:19 PM	File folder	
src	7/12/2025 11:12 PM	File folder	
example.slang	7/12/2025 11:36 PM	SLANG File	1 KB
log_vk_slang_editor.txt	7/12/2025 11:15 PM	Notepad++ Docu...	1 KB
slang.dll	7/8/2025 7:41 AM	Application exten...	27,348 KB
slangc.exe	7/8/2025 7:42 AM	Application	716 KB
slangd.exe	7/8/2025 7:41 AM	Application	489 KB
slang-glslang.dll	7/8/2025 7:40 AM	Application exten...	9,818 KB
slang-glsl-module.dll	7/8/2025 7:40 AM	Application exten...	3,143 KB
spirv-tools-shared.dll	3/12/2025 2:17 PM	Application exten...	2,088 KB
vk_slang_editor.exe	7/12/2025 11:14 PM	Application	4,145 KB
vk_slang_editor.ini	7/12/2025 11:15 PM	Configuration sett...	9 KB



```
C:\Users\nbickford\Documents\Presentations\SIGGRAPH2025\Packages\S2025SlangWindows>dir
 Volume in drive C is OSDisk
 Volume Serial Number is E68F-27C7

 Directory of C:\Users\nbickford\Documents\Presentations\SIGGRAPH2025\Packages\S2025SlangWindows

07/13/2025  01:36 AM    <DIR>          .
07/13/2025  01:36 AM    <DIR>          ..
07/12/2025  11:36 PM               815 example.slang
07/12/2025  11:15 PM    <DIR>          examples
07/12/2025  11:15 PM               522 log_vk_slang_editor.txt
07/12/2025  11:15 PM    <DIR>          media
07/12/2025  11:19 PM    <DIR>          RenderDoc_1.39_64
07/08/2025  07:40 AM         3,218,432 slang-glsl-module.dll
07/08/2025  07:40 AM        10,053,120 slang-glslang.dll
07/08/2025  07:41 AM        28,004,352 slang.dll
07/08/2025  07:42 AM           733,184 slangc.exe
07/08/2025  07:41 AM           500,224 slangd.exe
03/12/2025  02:17 PM         2,137,880 spirv-tools-shared.dll
07/12/2025  11:12 PM    <DIR>          src
07/12/2025  11:14 PM         4,244,480 vk_slang_editor.exe
07/12/2025  11:15 PM             8,222 vk_slang_editor.ini
              10 File(s)     48,901,231 bytes
               6 Dir(s)  145,787,789,312 bytes free

C:\Users\nbickford\Documents\Presentations\SIGGRAPH2025\Packages\S2025SlangWindows>
```

SLANGC LAB: QUICK START

SIGGRAPH 2025
Vancouver+ 10-14 August

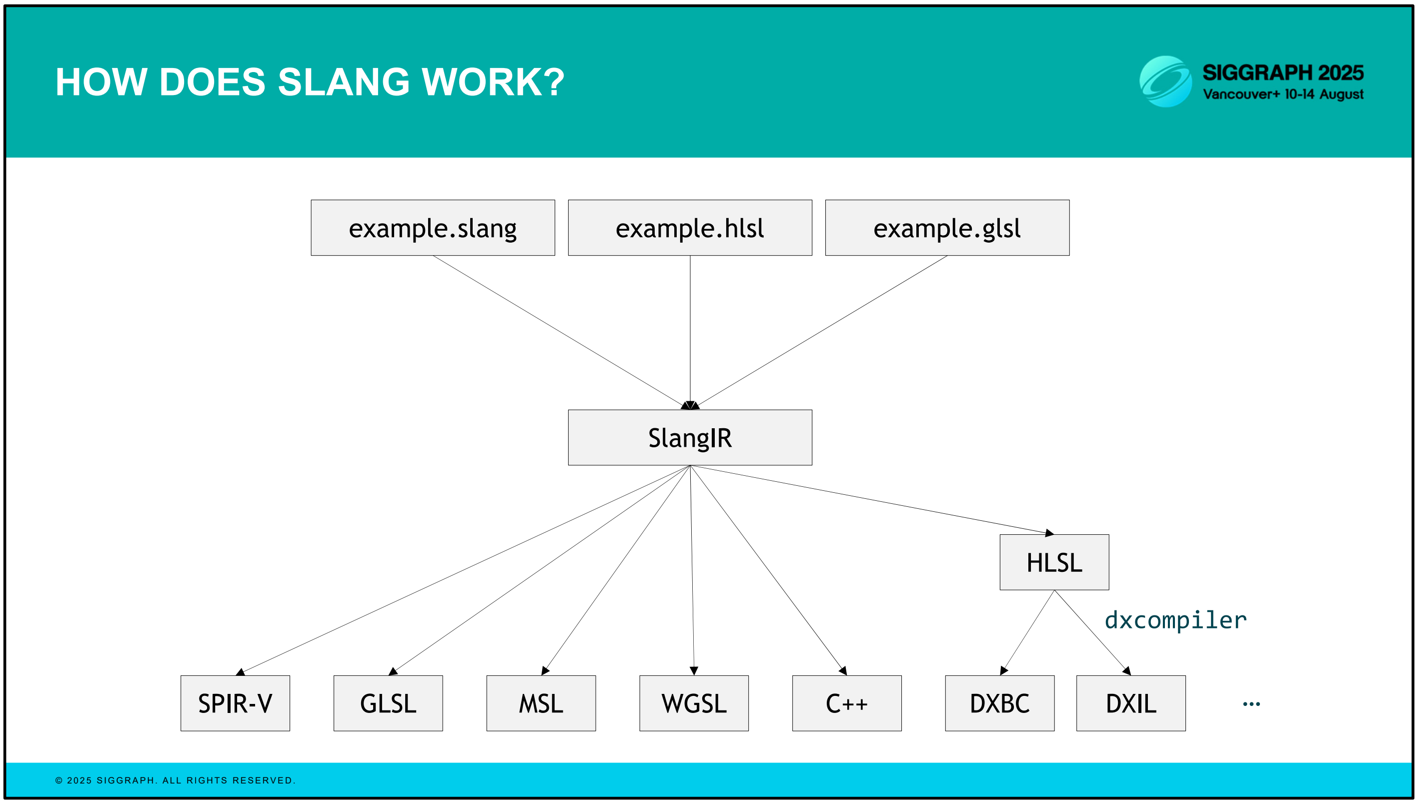
- Run `slangc -v`
  - Should get 2025.13.1
- Let’s compile to GLSL!
  - Run `slangc example.slang -target glsl`
  - Point out a few things
    - `#version 450`, so it’s GLSL
    - uniforms and buffers are marked with a matrix layout; we’ll talk about that later
    - Line directives
    - `RWTexture2D<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

- Run `slangc -h`
  - That’s a lot of text! Direct it to a file using `slangc -h > help.txt`
  - Open it in a text editor
- Direct attention to the options for `-target`.

SLANGC LAB: OTHER TARGETS

- 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 wgsl`
  - Run `slangc example.slang -target metal`
    - Bindings now declared as function parameters



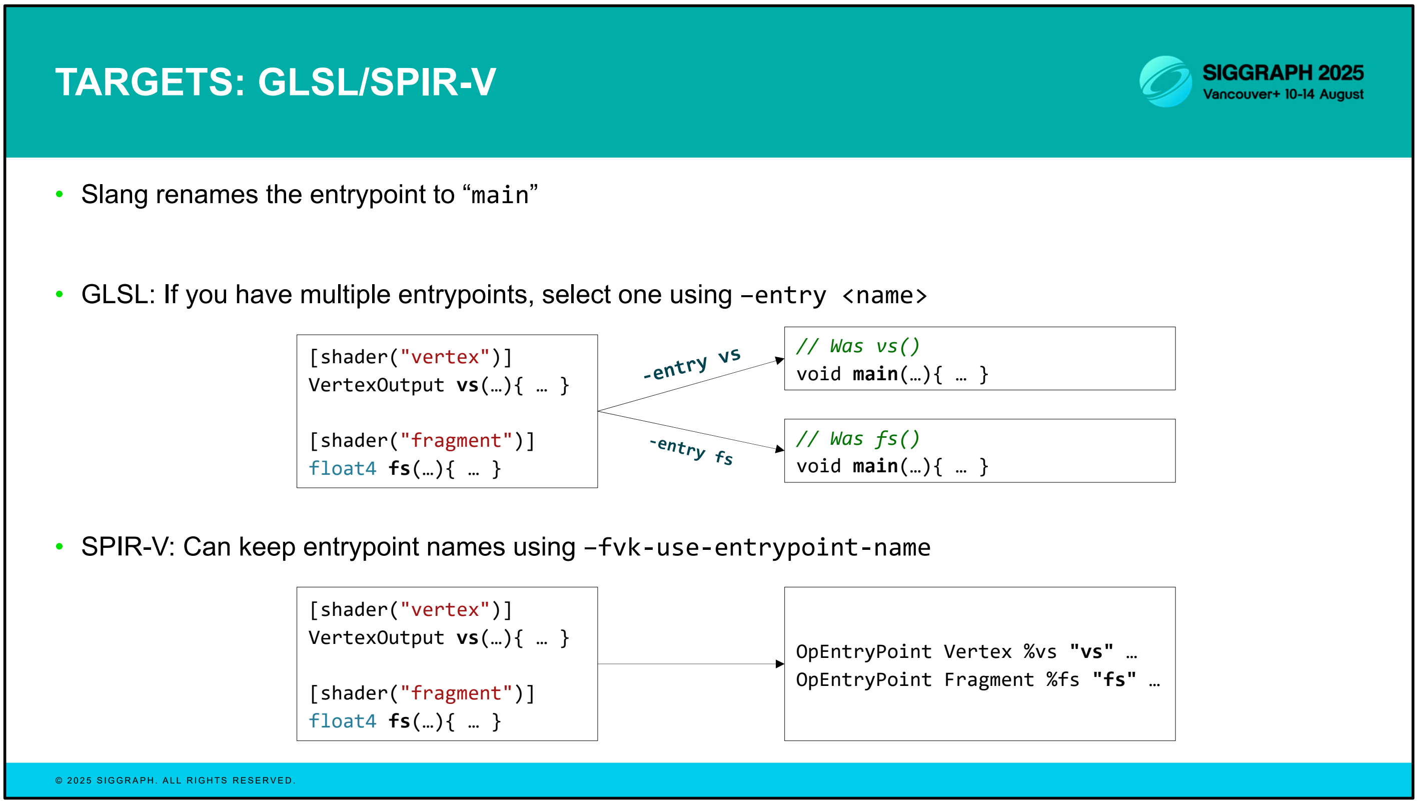
HOW DOES SLANG WORK?

SIGGRAPH 2025
Vancouver+ 10-14 August

example.slang
example.hlsl
example.glsl
SlangIR
HLSL
dxcompiler
SPIR-V
GLSL
MSL
WGSL
C++
DXBC
DXIL
...

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.



TARGETS: GLSL/SPIR-V

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

```
[shader("vertex")]
VertexOutput vs(…){ … }

[shader("fragment")]
float4 fs(…){ … }
```

`-entry vs`

```
// Was vs()
void main(…){ … }
```

`-entry fs`

```
// Was fs()
void main(…){ … }
```

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

```
[shader("vertex")]
VertexOutput vs(…){ … }

[shader("fragment")]
float4 fs(…){ … }
```

```
OpEntryPoint Vertex %vs "vs" …
OpEntryPoint Fragment %fs "fs" …
```

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 it 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.

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.

SIGGRAPH 2025
Vancouver+ 10-14 August

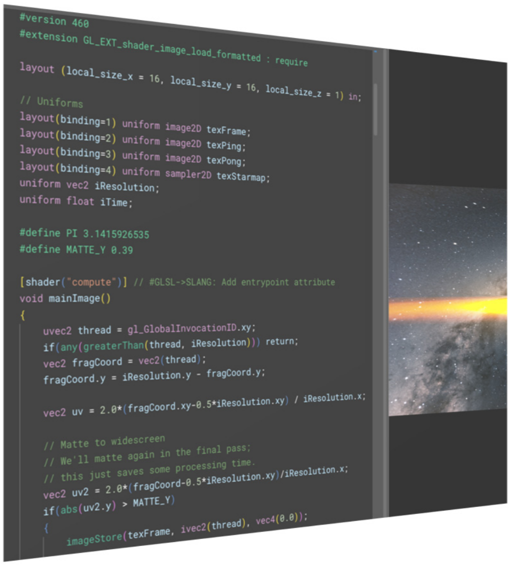
Porting GLSL

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

SIGGRAPH 2025
Vancouver+ 10-14 August

- 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 460
#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 texPing;
layout(binding=3) uniform image2D texPong;
layout(binding=4) uniform sampler2D texStarmap;
uniform vec2 iResolution;
uniform float iTime;

#define PI 3.1415926535
#define MATTE_Y 0.39

[shader("compute")] // #GLSL->SLANG: 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
    // We'll 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));
```

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.

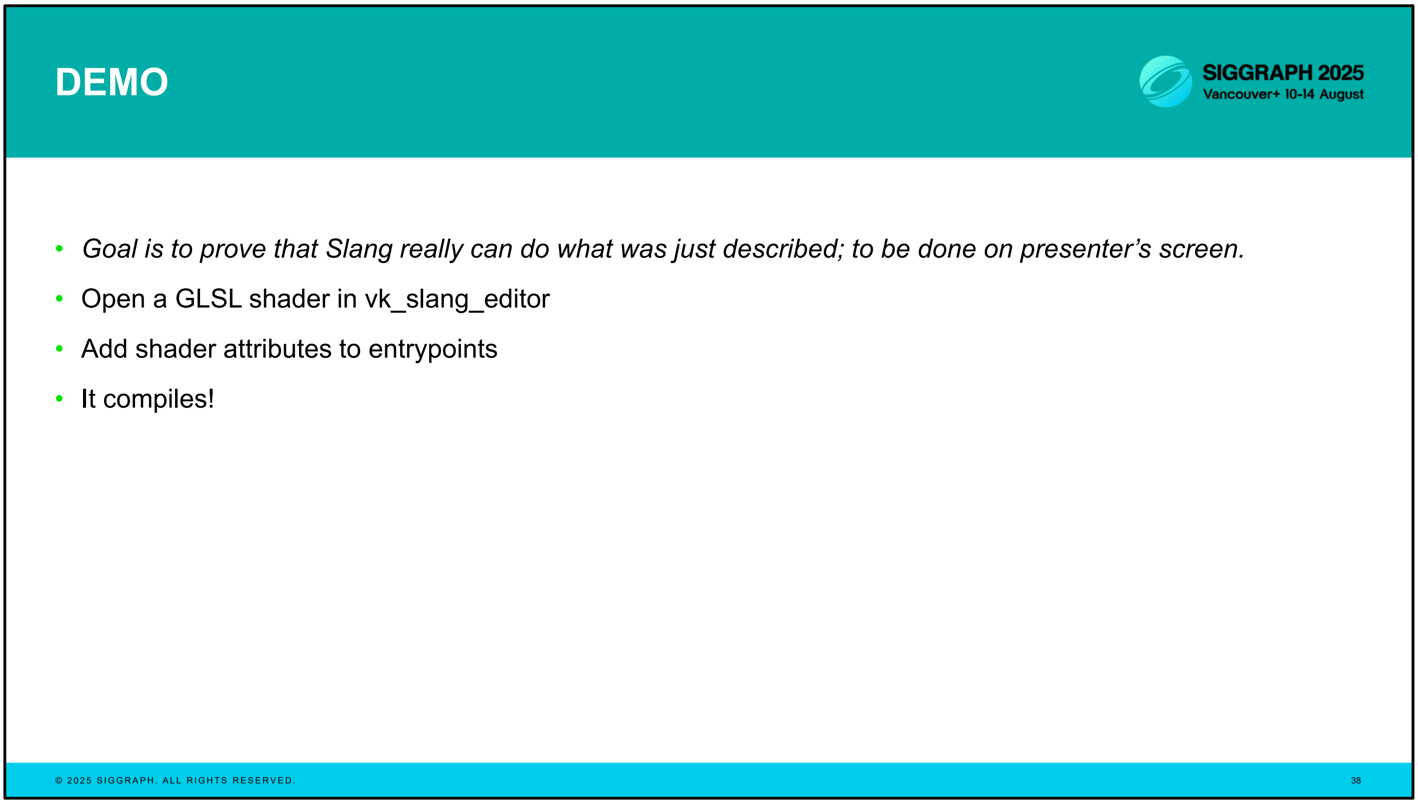
Then, you need to mark your entrypoints with [shader] attributes to tell Slang which functions are entrypoints for which shader types.

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.

Demo:
Porting GLSL

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

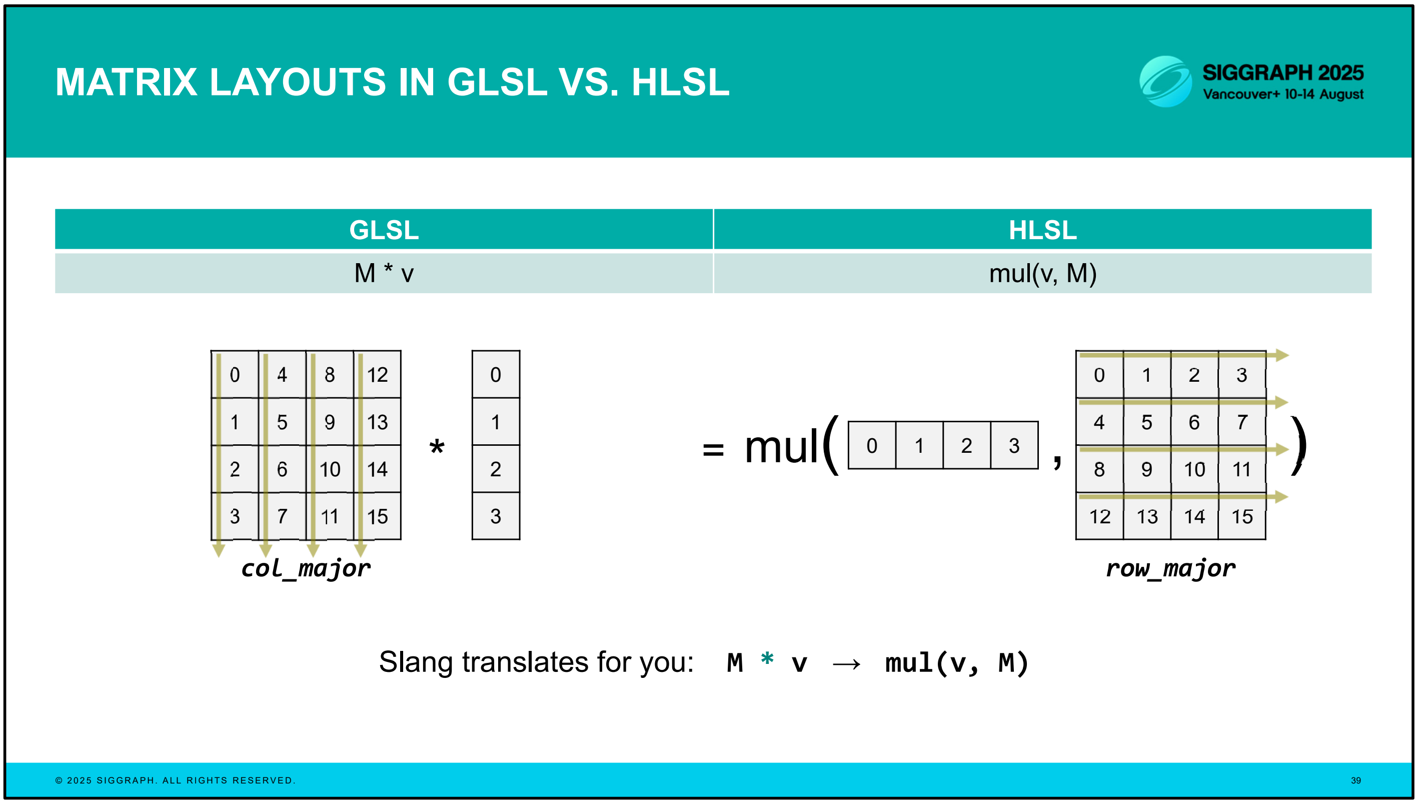


DEMO

SIGGRAPH 2025
Vancouver+ 10-14 August

- *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!

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



MATRIX LAYOUTS IN GLSL VS. HLSL

SIGGRAPH 2025
Vancouver+ 10-14 August

GLSL	HLSL
M * v	mul(v, M)

$$\begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix} * \begin{bmatrix} 0 \\ 1 \\ 2 \\ 3 \end{bmatrix} = \text{mul}\left(\begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix}, \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} \right)$$

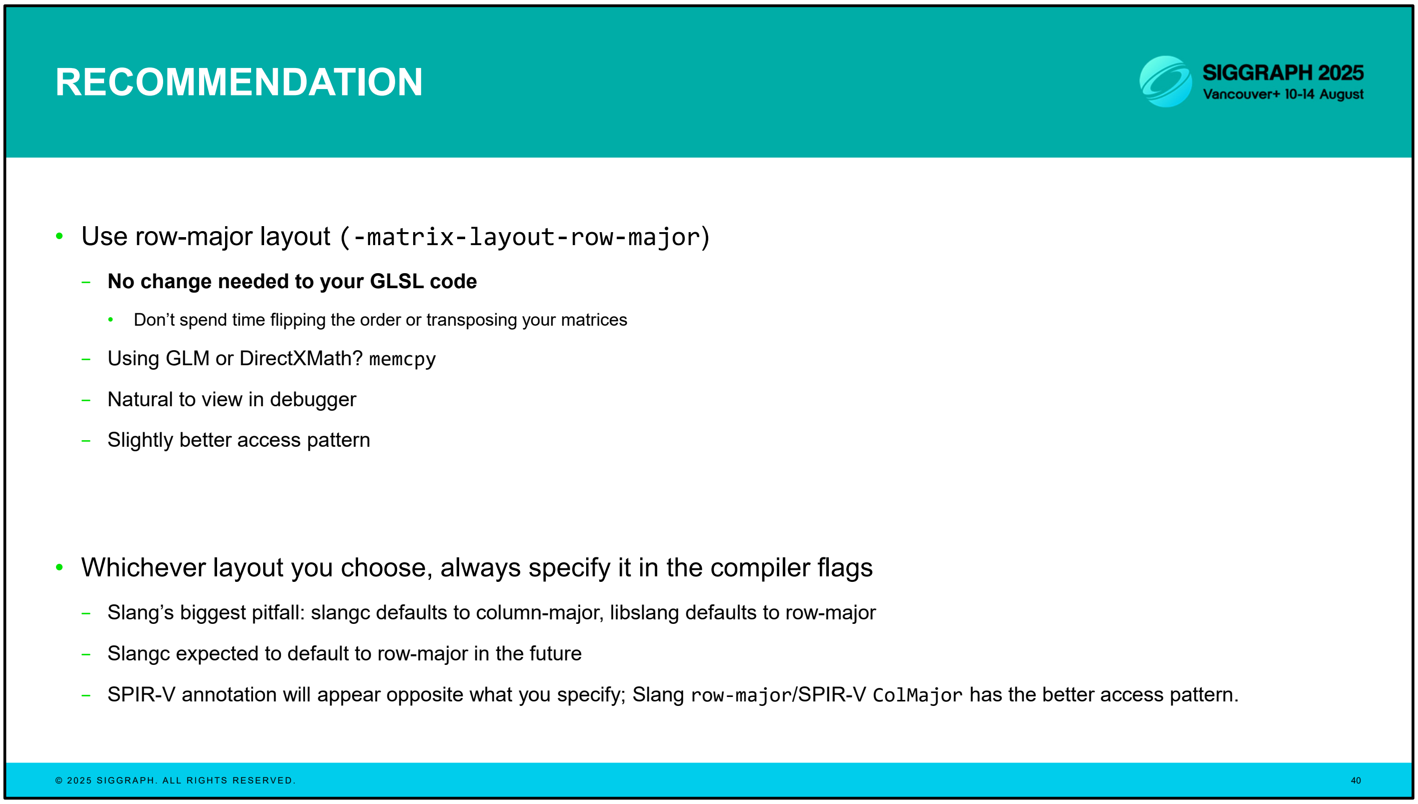
Slang translates for you: `M * v` → `mul(v, M)`

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 mul(v, M).



RECOMMENDATION

- 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.

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 the debugger, and have slightly better access patterns for the GPU.

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

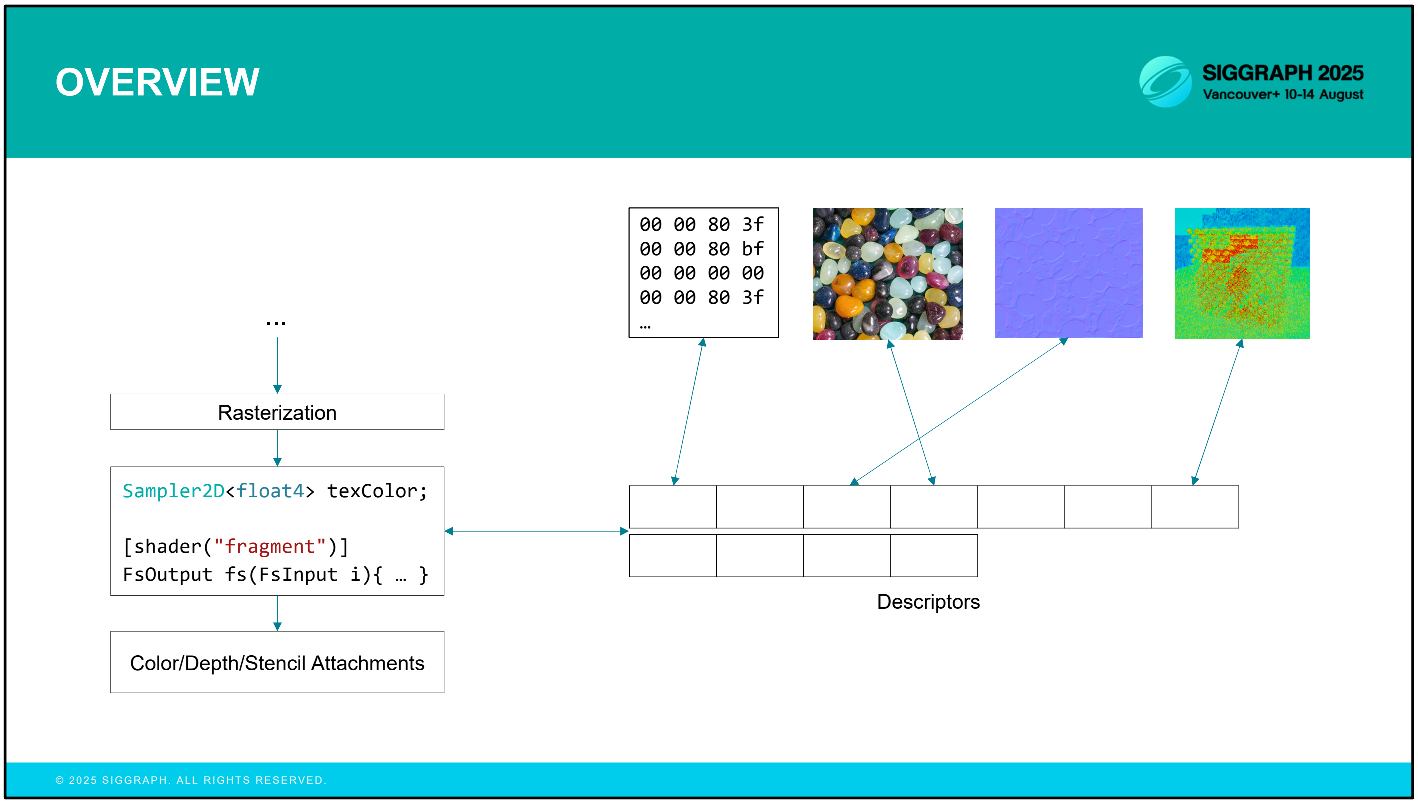
The second most important thing is to *always* specify the matrix layout in the compiler flags. Slang's biggest pitfall at the moment in my opinion is that the library and the executable compilers default to different layouts. The Slang team is hoping to make these both default to row-major in the future,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

Shader I/O

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



OVERVIEW

SIGGRAPH 2025
Vancouver+ 10-14 August

…

Rasterization

```
Sampler2D<float4> texColor;

[shader("fragment")]
FsOutput fs(FsInput i){ … }
```

Color/Depth/Stencil Attachments

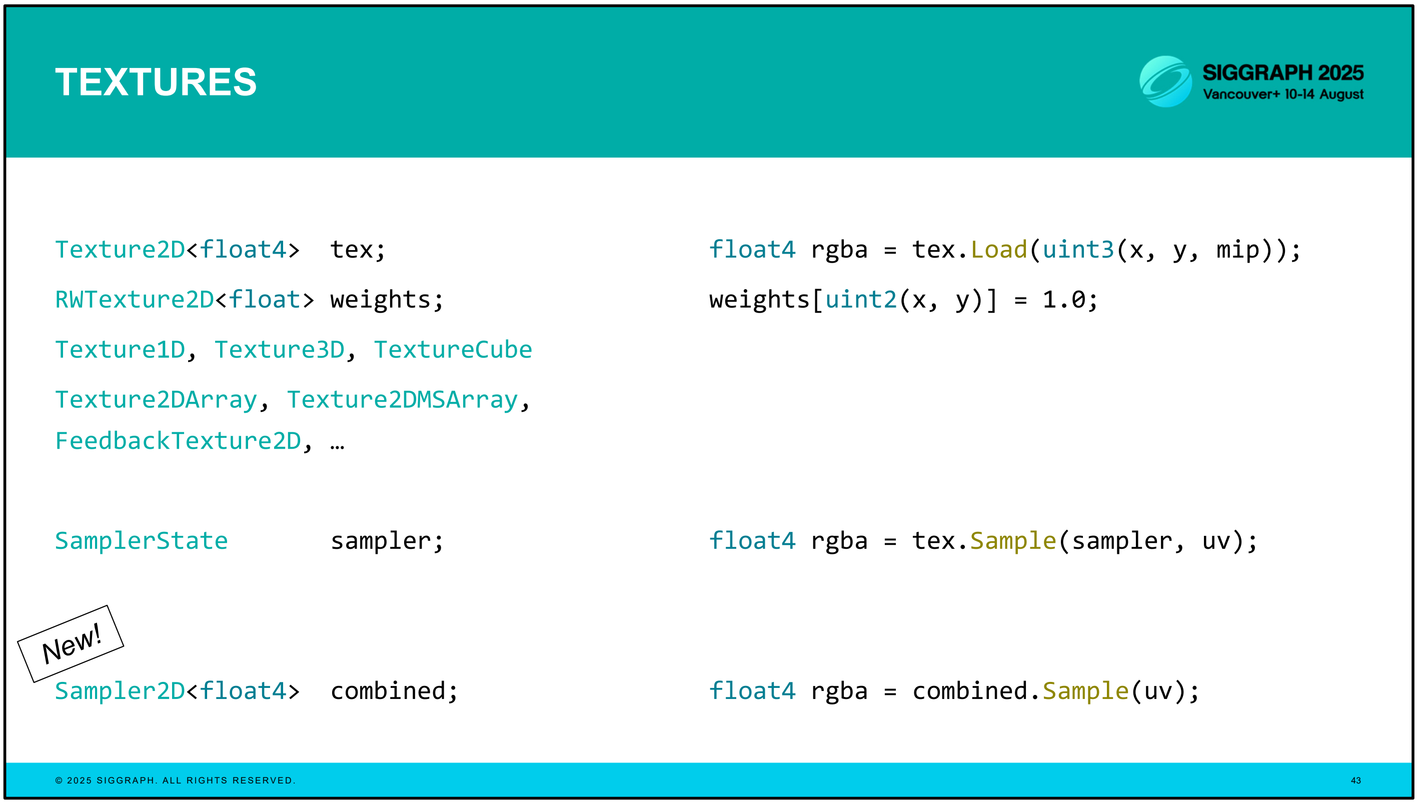
```
00 00 80 3f
00 00 80 bf
00 00 00 00
00 00 80 3f
…
```

Descriptors

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.



TEXTURES

SIGGRAPH 2025
Vancouver+ 10-14 August

```
Texture2D<float4>  tex;                   float4 rgba = tex.Load(uint3(x, y, mip));
RWTexture2D<float> weights;               weights[uint2(x, y)] = 1.0;
Texture1D, Texture3D, TextureCube
Texture2DArray, Texture2DMSArray,
FeedbackTexture2D, …

SamplerState       sampler;               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.

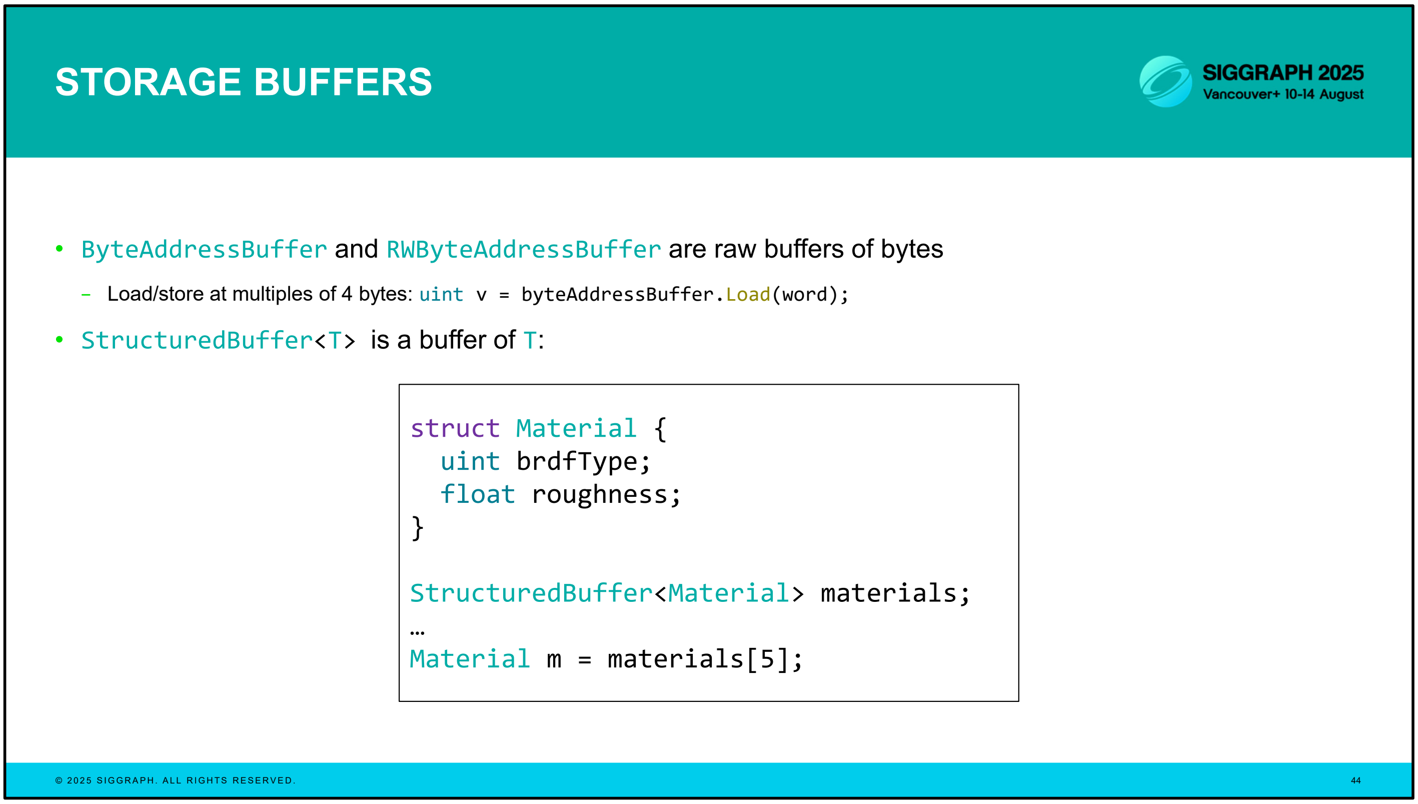
Texture2D objects are read-only. To read and write to them, we need **(click)** an RWTexture2D, like we saw in the intro example.

(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 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.



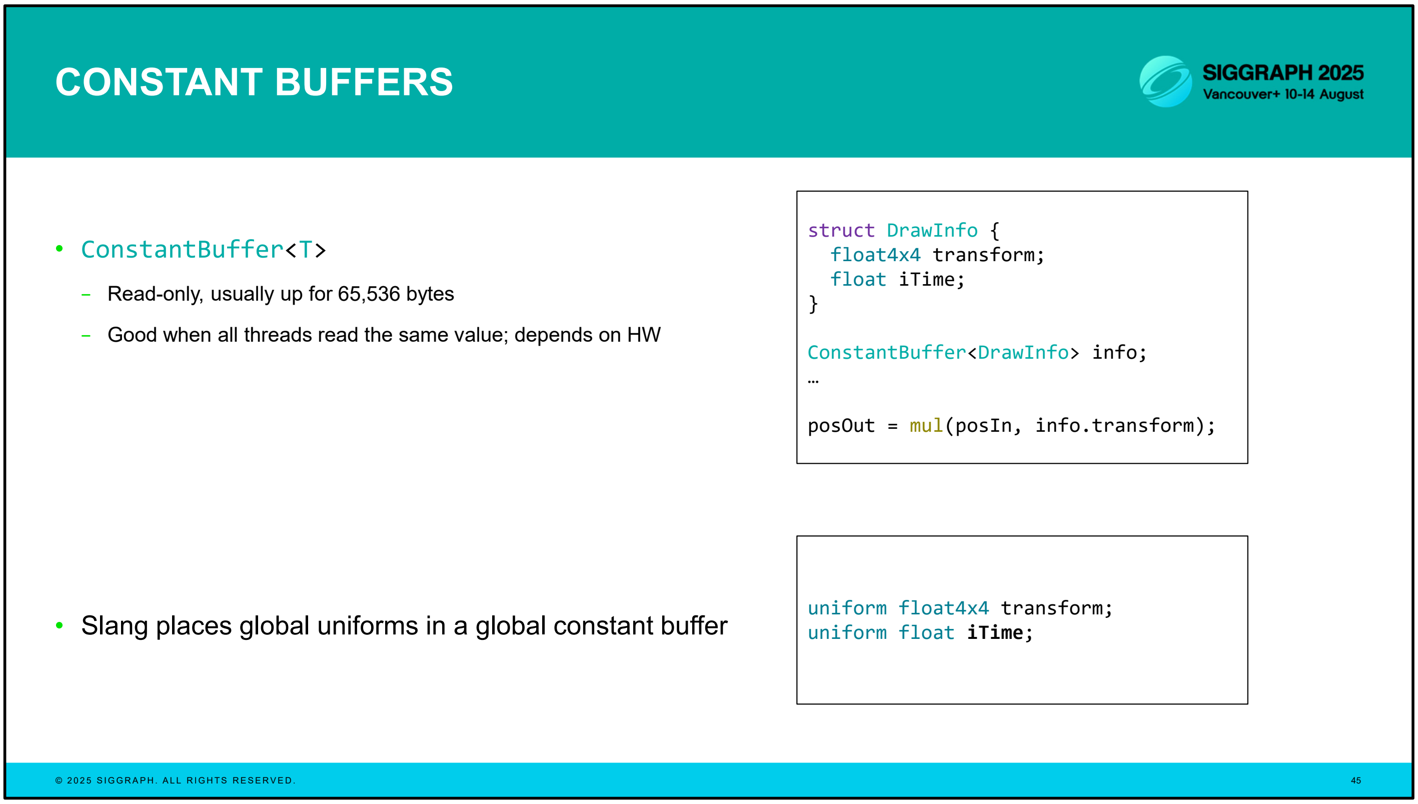
STORAGE BUFFERS

- 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 2025
Vancouver+ 10-14 August

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

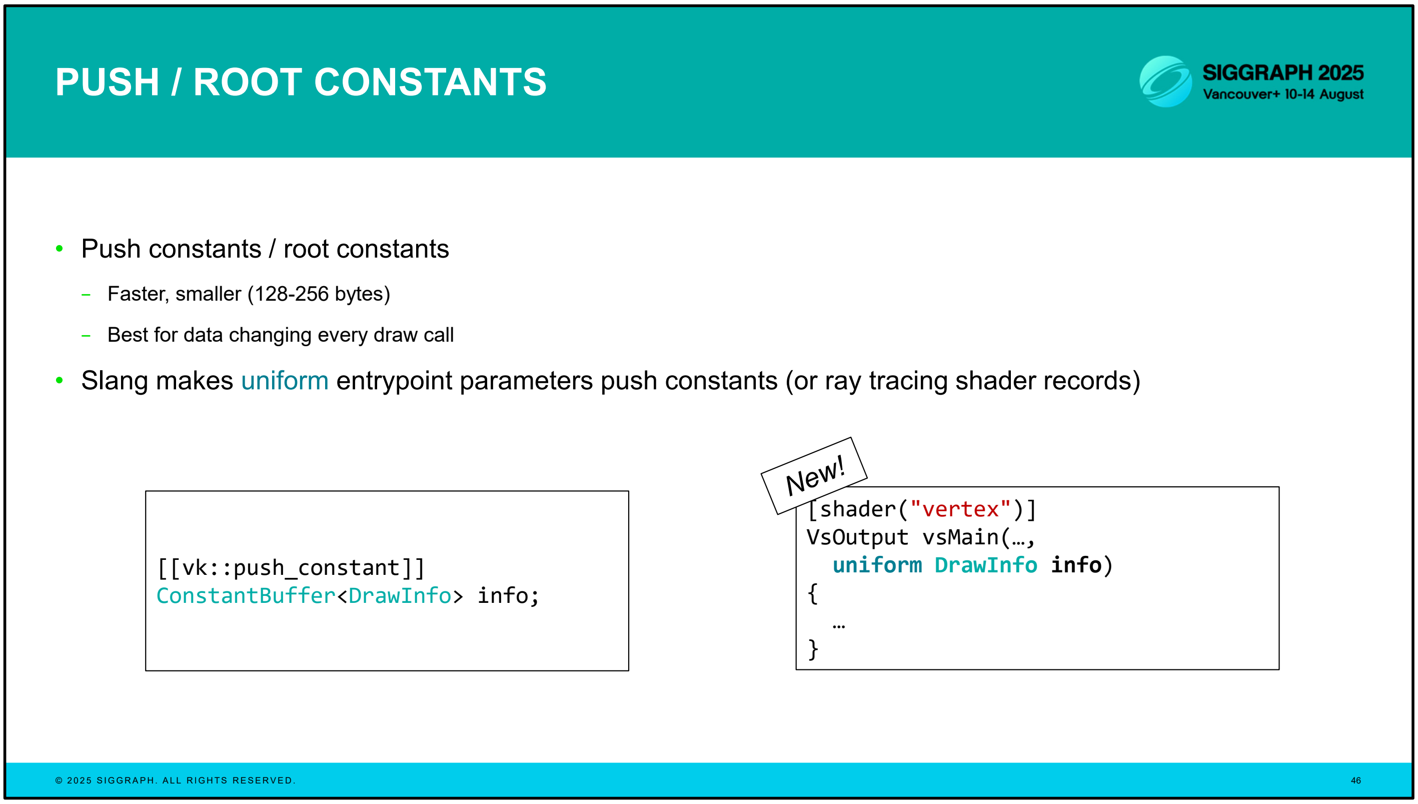
```
struct DrawInfo {
  float4x4 transform;
  float iTime;
}

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.

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 2025
Vancouver+ 10-14 August

- 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!

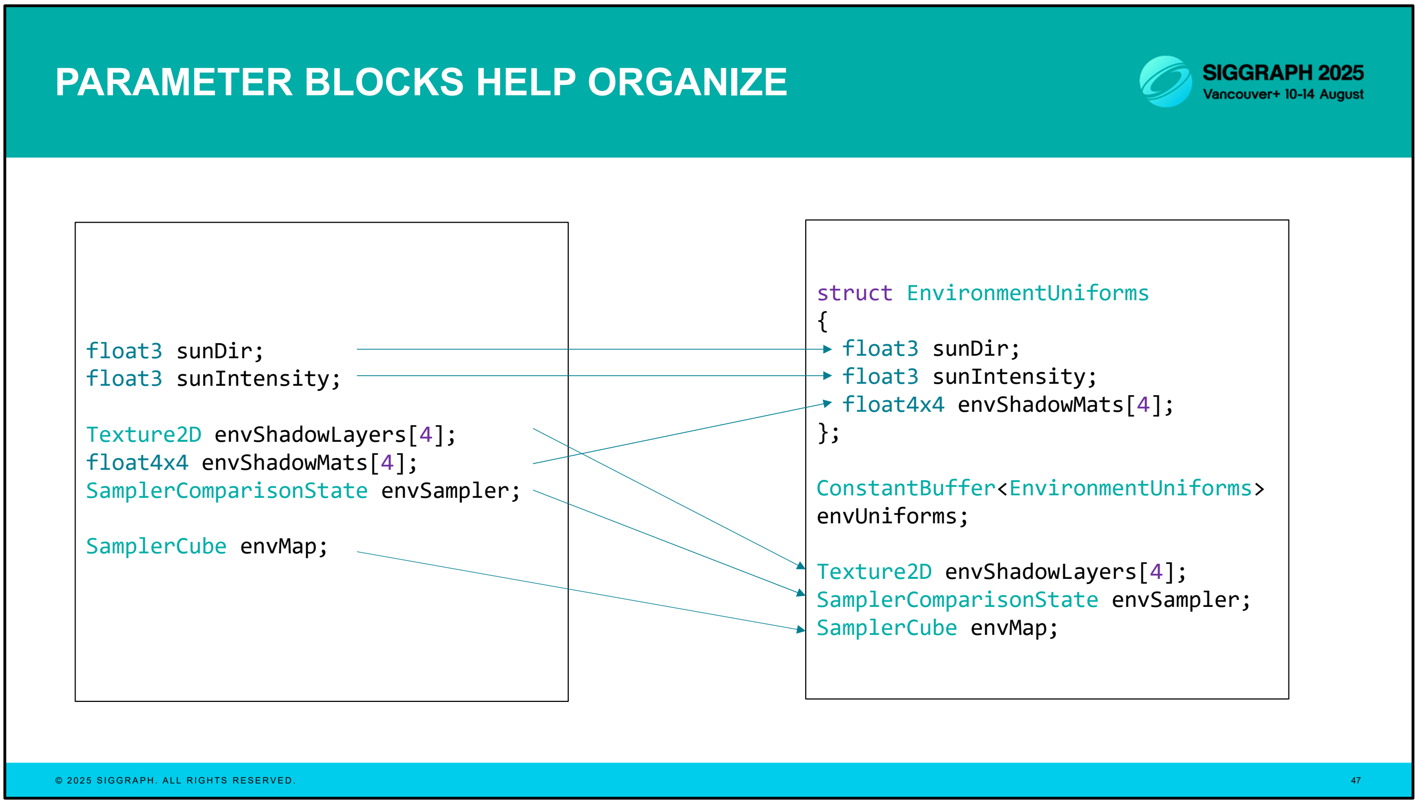
```
[shader("vertex")]
VsOutput vsMain(…,
  uniform DrawInfo info)
{
  …
}
```

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 2025
Vancouver+ 10-14 August

```
float3 sunDir;
float3 sunIntensity;

Texture2D envShadowLayers[4];
float4x4 envShadowMats[4];
SamplerComparisonState envSampler;

SamplerCube envMap;
```

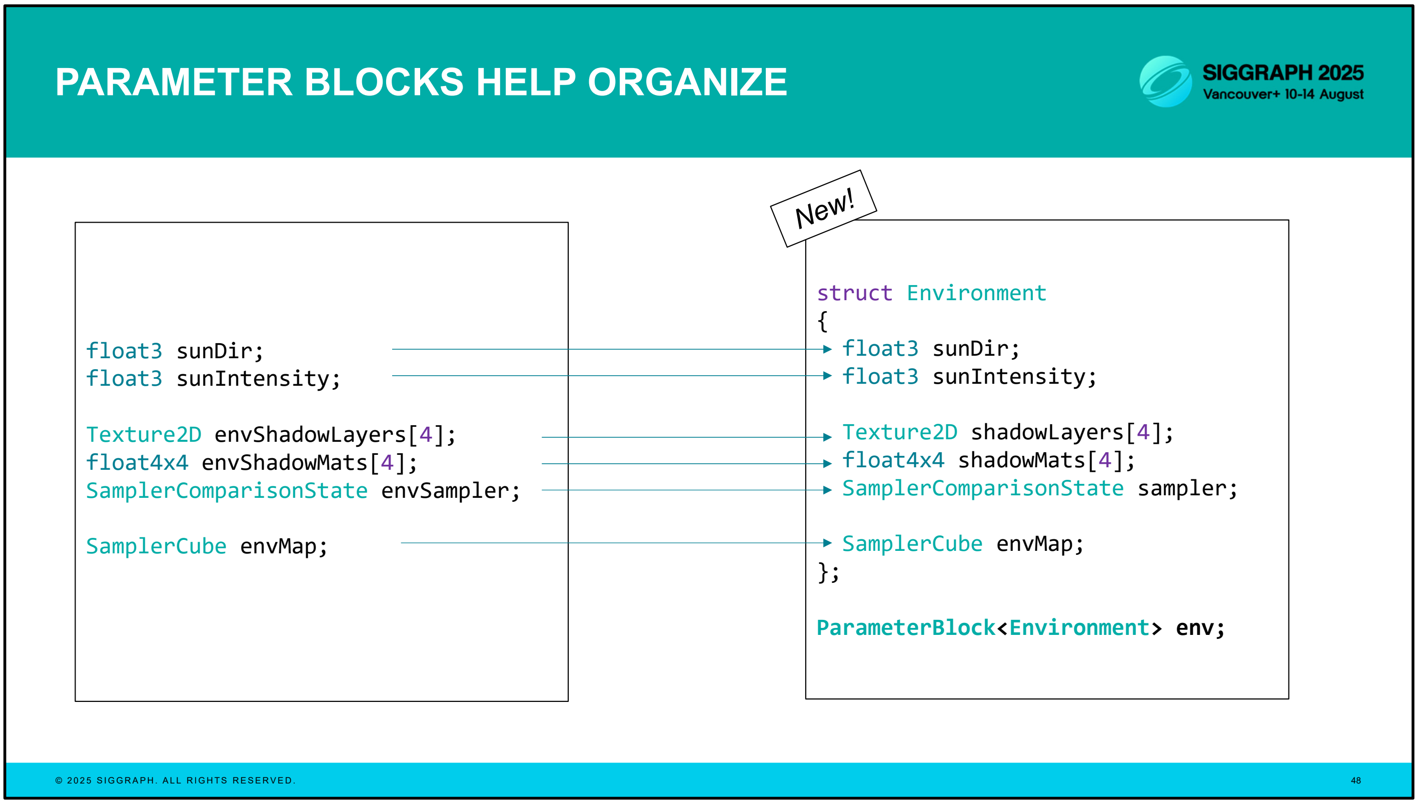
```
struct EnvironmentUniforms
{
  float3 sunDir;
  float3 sunIntensity;
  float4x4 envShadowMats[4];
};

ConstantBuffer<EnvironmentUniforms>
envUniforms;

Texture2D envShadowLayers[4];
SamplerComparisonState envSampler;
SamplerCube envMap;
```

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, 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 2025
Vancouver+ 10-14 August

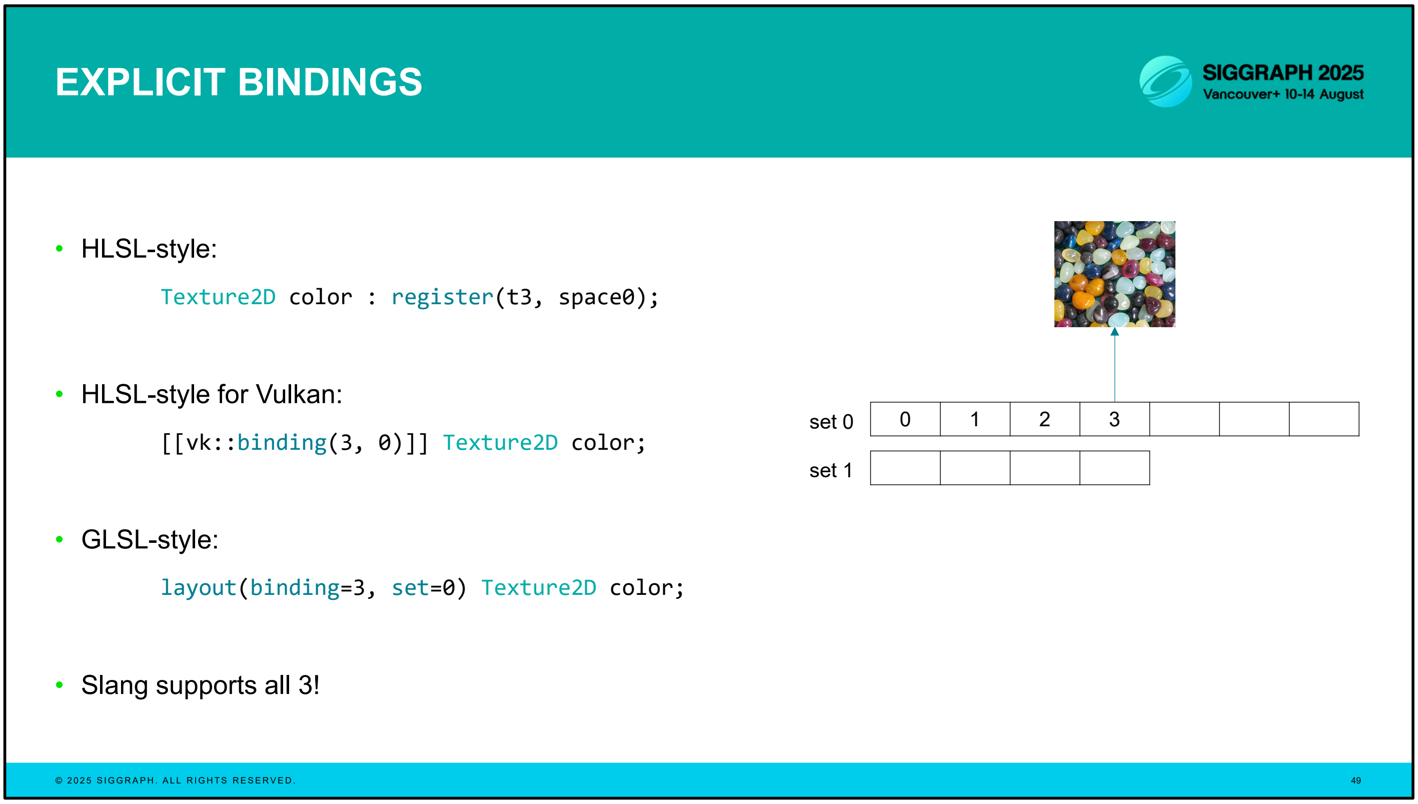
```
float3 sunDir;
float3 sunIntensity;

Texture2D envShadowLayers[4];
float4x4 envShadowMats[4];
SamplerComparisonState envSampler;

SamplerCube envMap;
```

New!

```
struct Environment
{
  float3 sunDir;
  float3 sunIntensity;

  Texture2D shadowLayers[4];
  float4x4 shadowMats[4];
  SamplerComparisonState sampler;

  SamplerCube envMap;
};

ParameterBlock<Environment> env;
```

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

- 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!

set 0	0	1	2	3			
set 1							

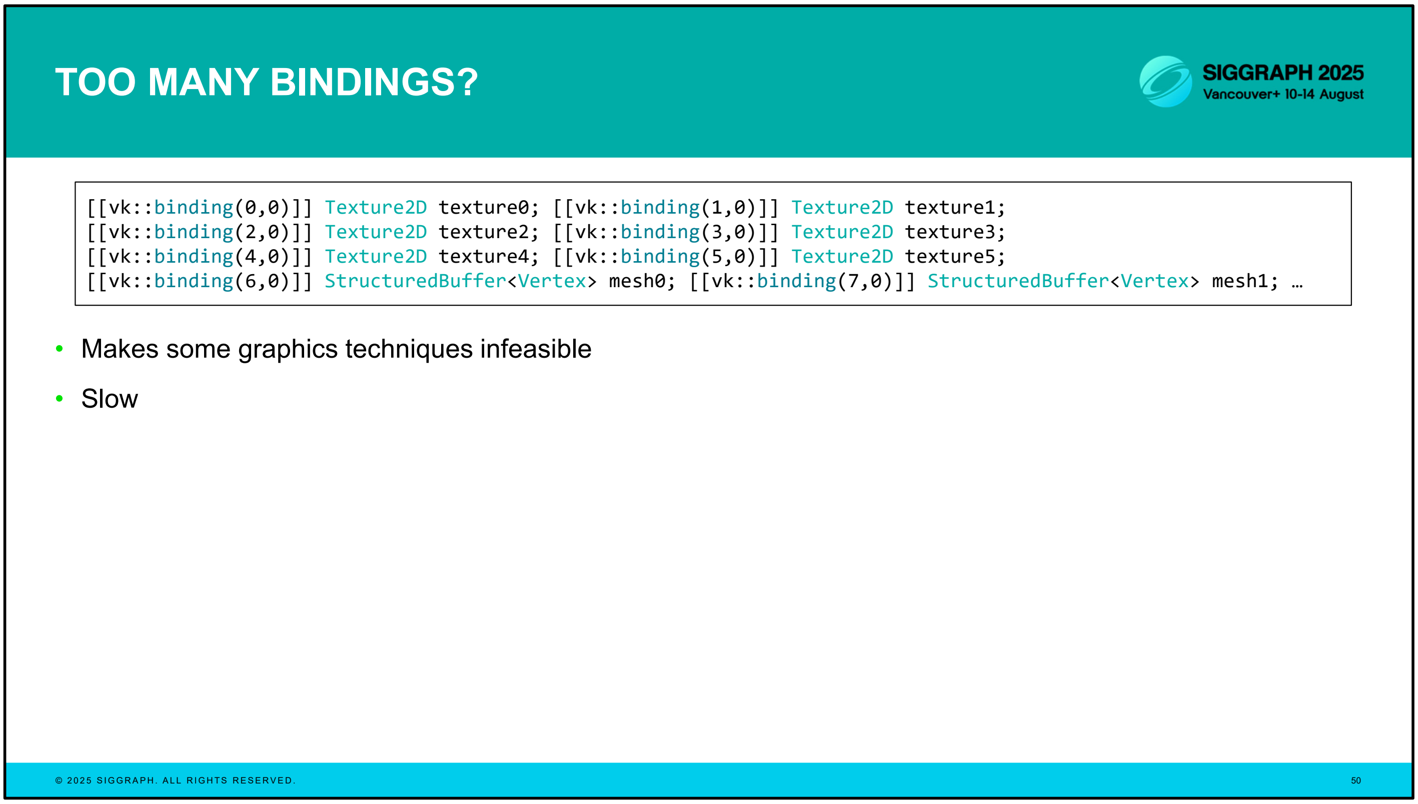
So far, we’ve been talking about resource types.

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!



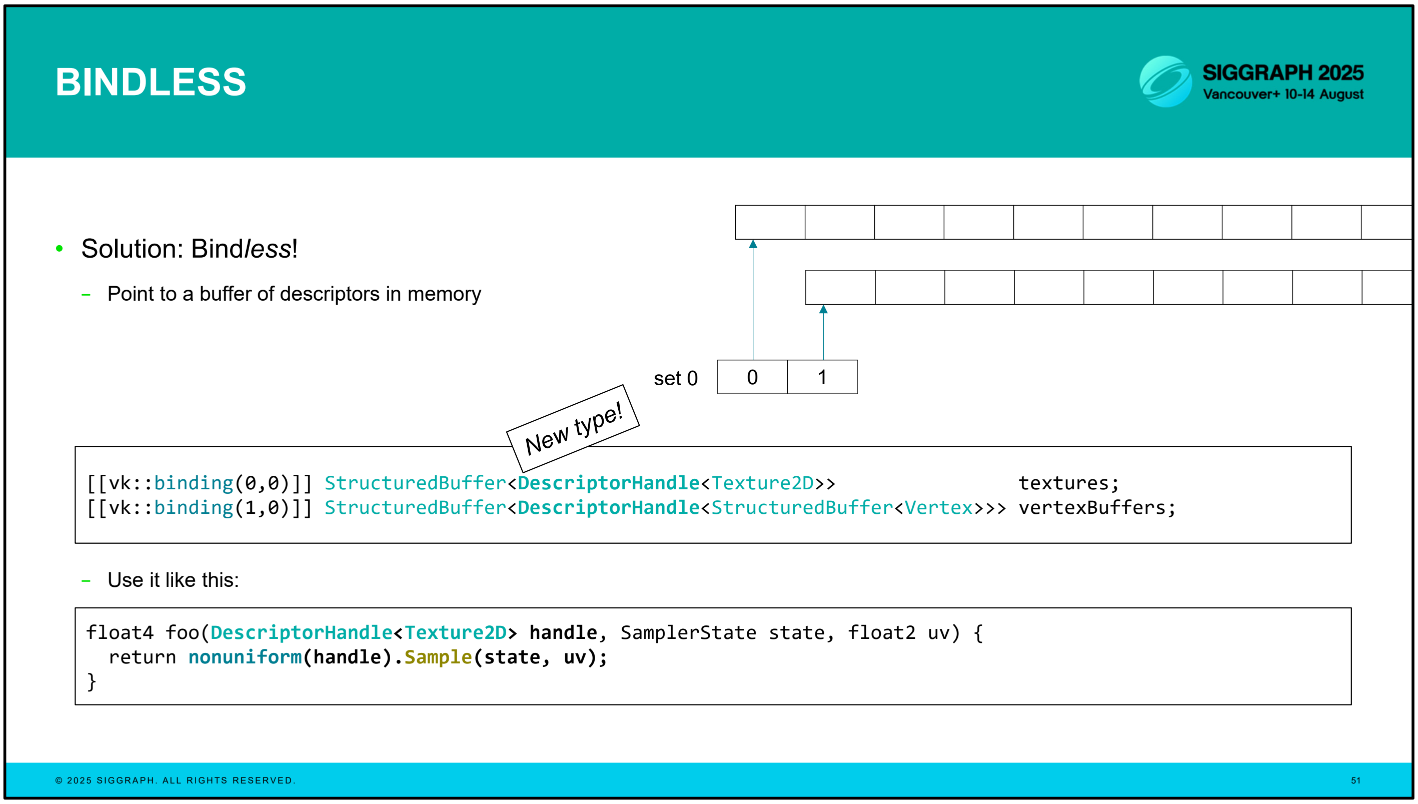
TOO MANY BINDINGS?

SIGGRAPH 2025
Vancouver+ 10-14 August

```
[[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.



BINDLESS

SIGGRAPH 2025
Vancouver+ 10-14 August

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

set 0 | 0 | 1

New type!

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

- Use it like this:

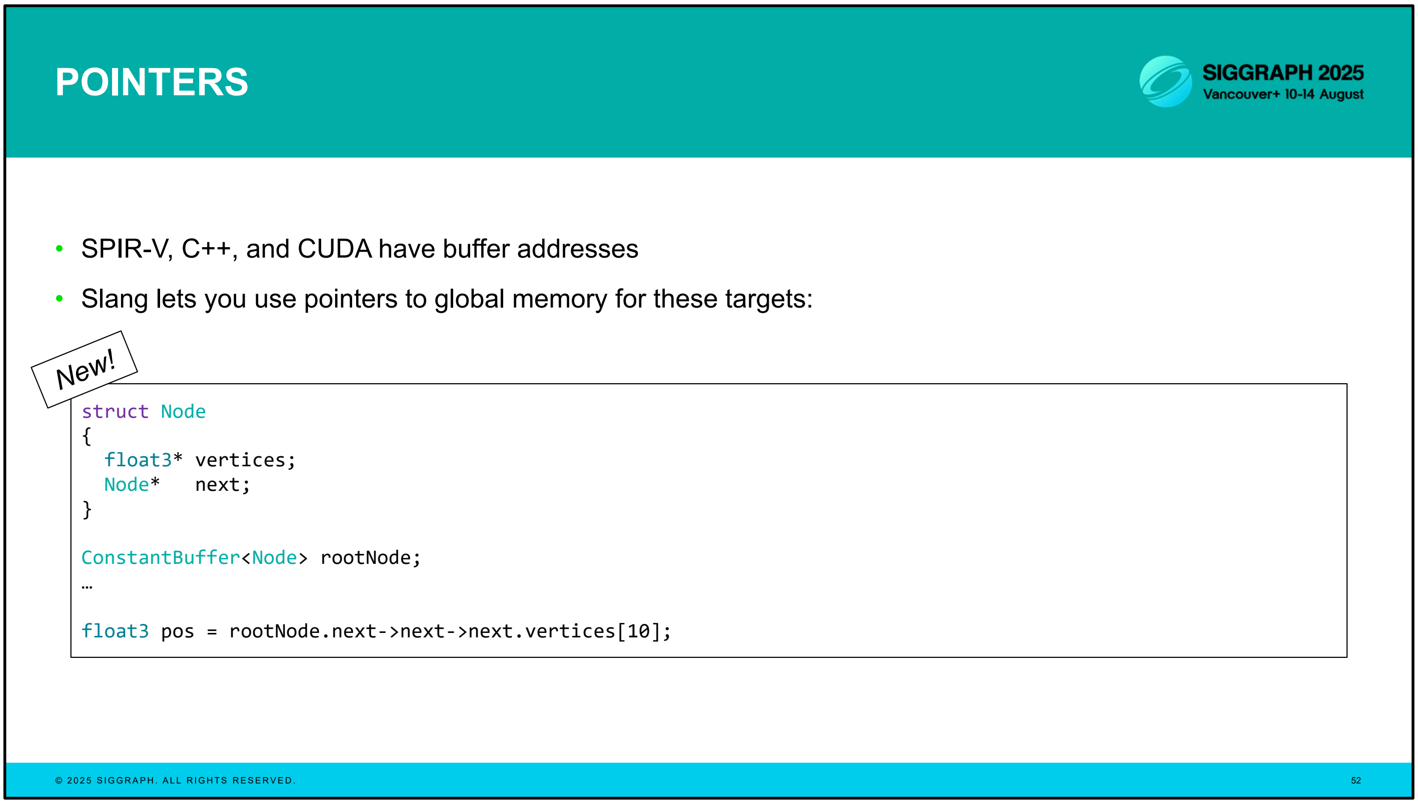
```
float4 foo(DescriptorHandle<Texture2D> handle, SamplerState state, float2 uv) {
  return nonuniform(handle).Sample(state, uv);
}
```

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.

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.



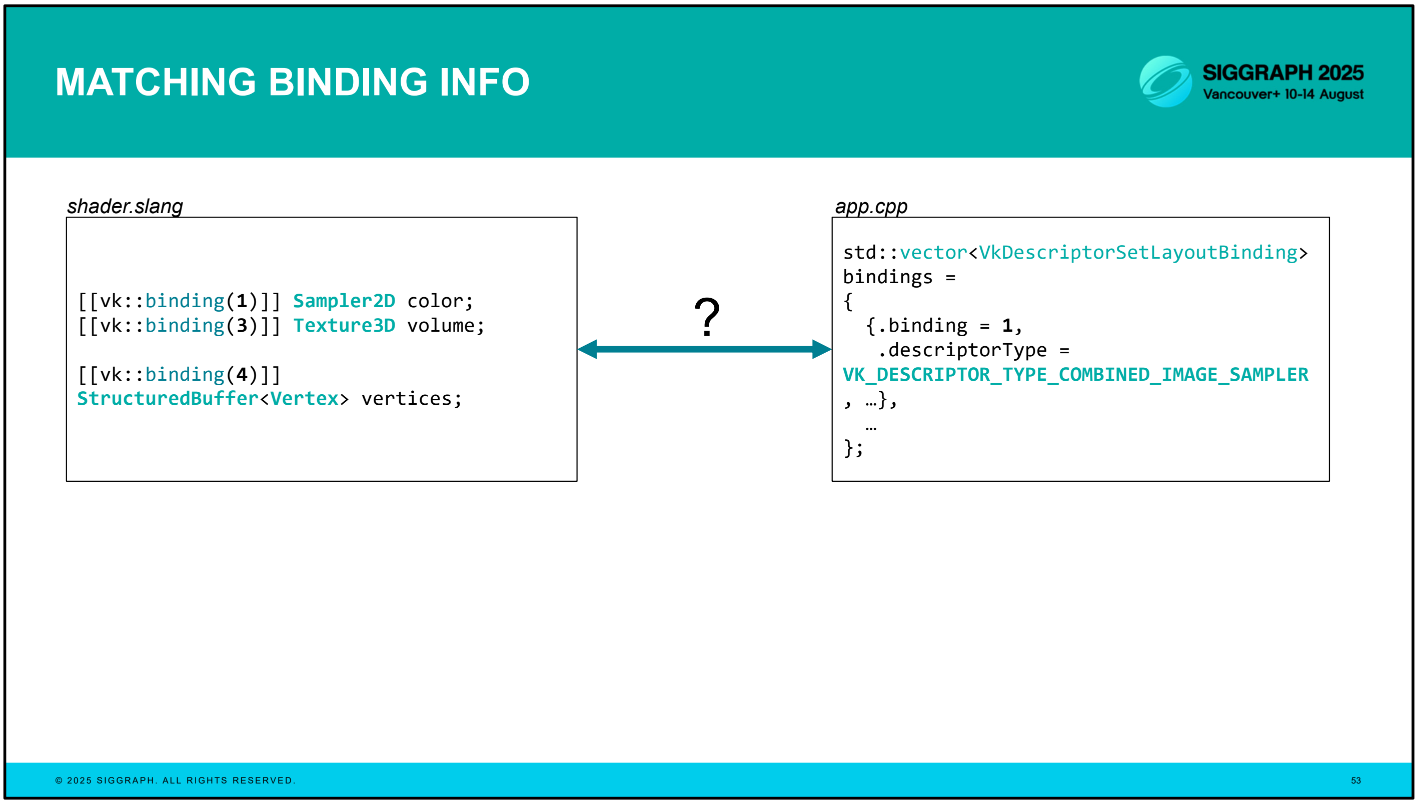
POINTERS

- 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];
```

Finally, 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 2025
Vancouver+ 10-14 August

shader.slang

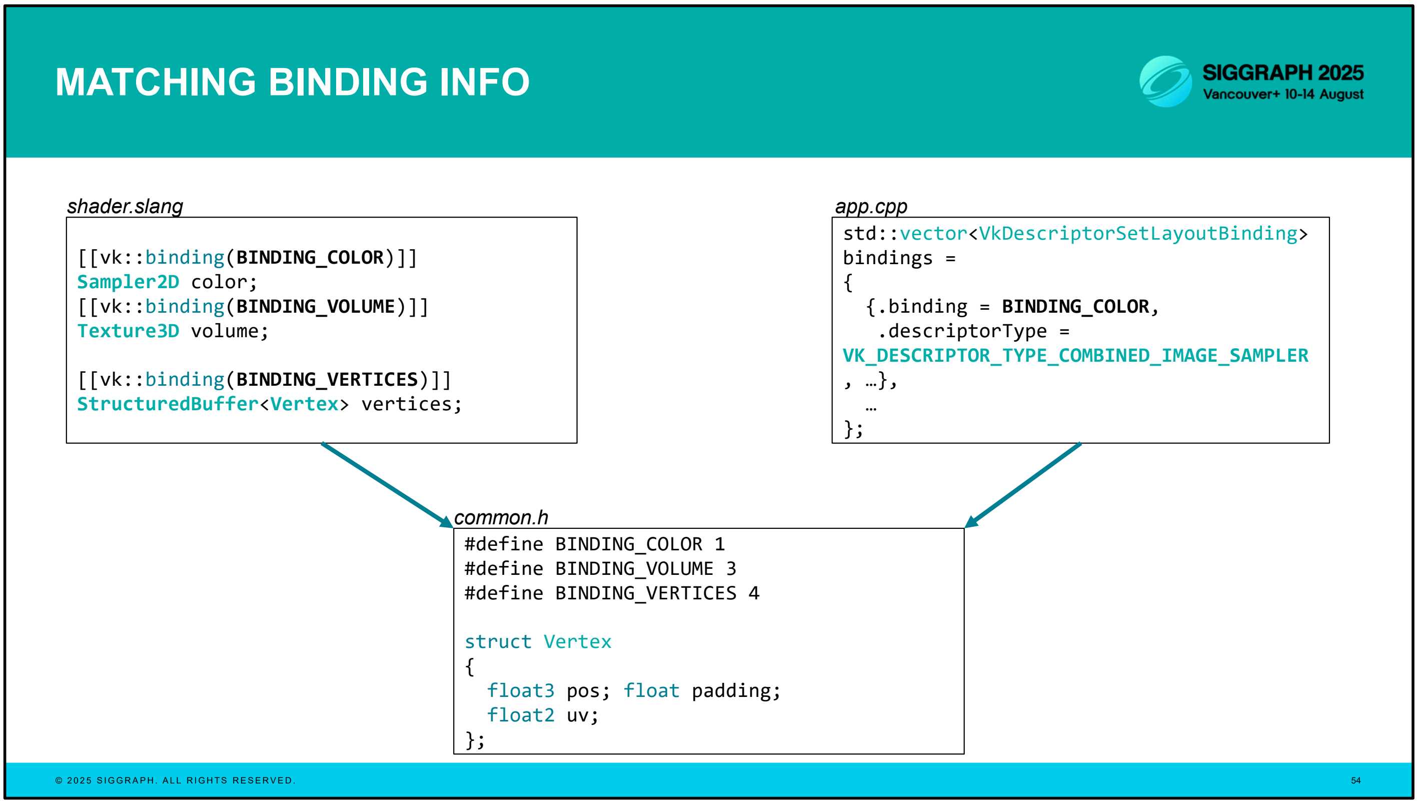
```
[[vk::binding(1)]] Sampler2D color;
[[vk::binding(3)]] Texture3D volume;

[[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 2025
Vancouver+ 10-14 August

shader.slang

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

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

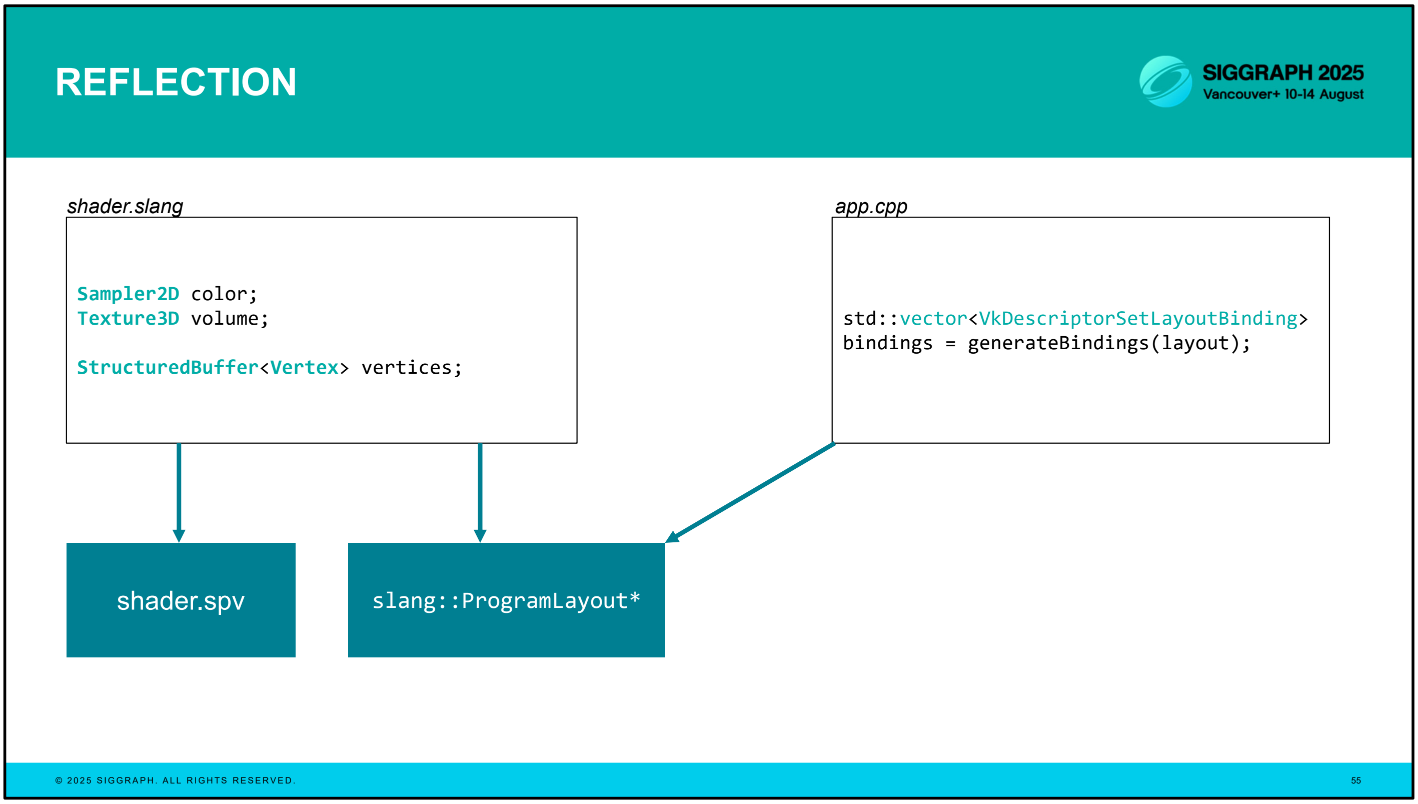
app.cpp

```
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;
};
```

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.



REFLECTION

SIGGRAPH 2025
Vancouver+ 10-14 August

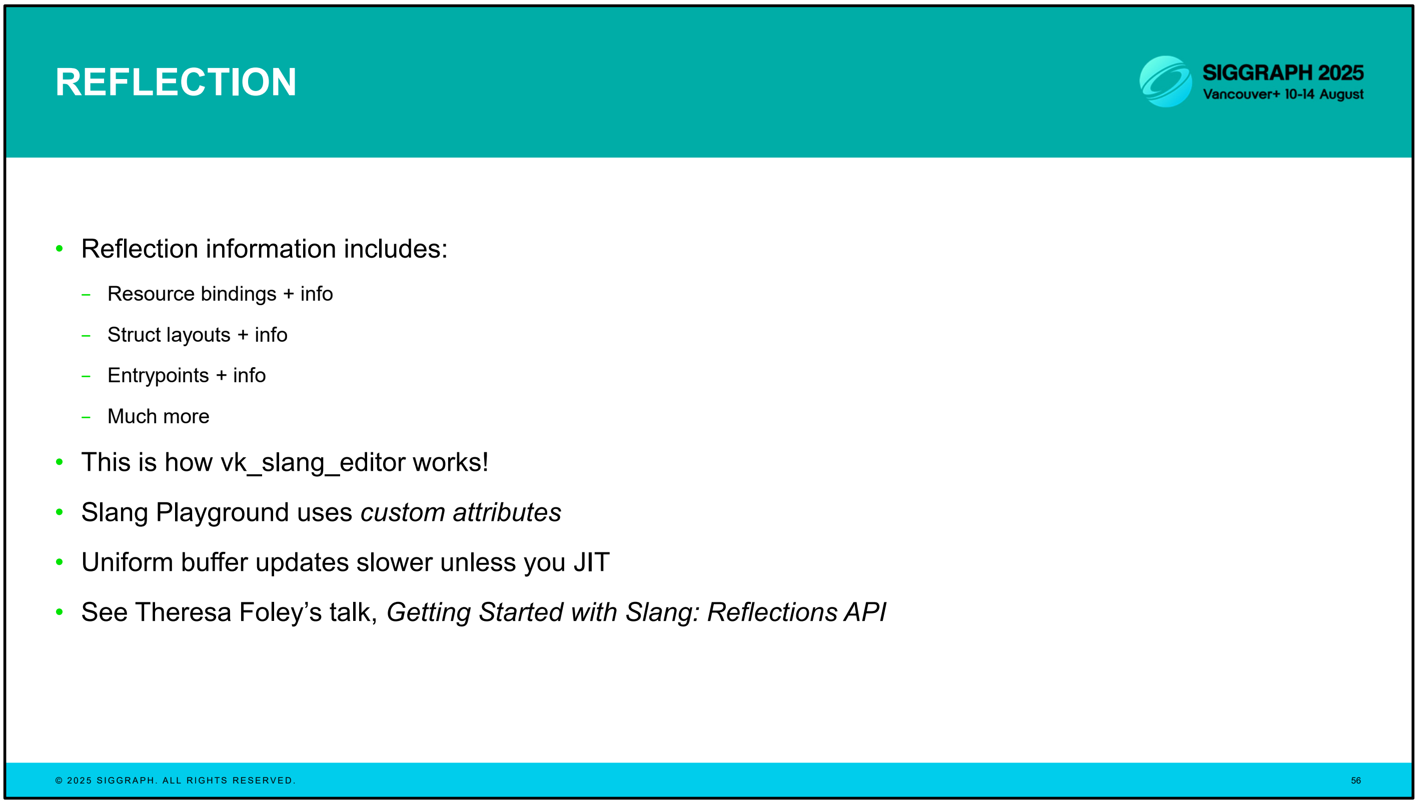
shader.slang

```
Sampler2D color;
Texture3D volume;

StructuredBuffer<Vertex> vertices;
```

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

SIGGRAPH 2025
Vancouver+ 10-14 August

- 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*

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 2025
Vancouver+ 10-14 August

- Can you figure out the missing types in *Examples > !SIGGRAPH2025 > 2 – Shader IO*?
- Use the comments as your guide
- Stuck?
  - Ask one of the TAs or the people around you! We're here to help.
  - The main keywords you'll need are in the box on the right →

We've talked about:

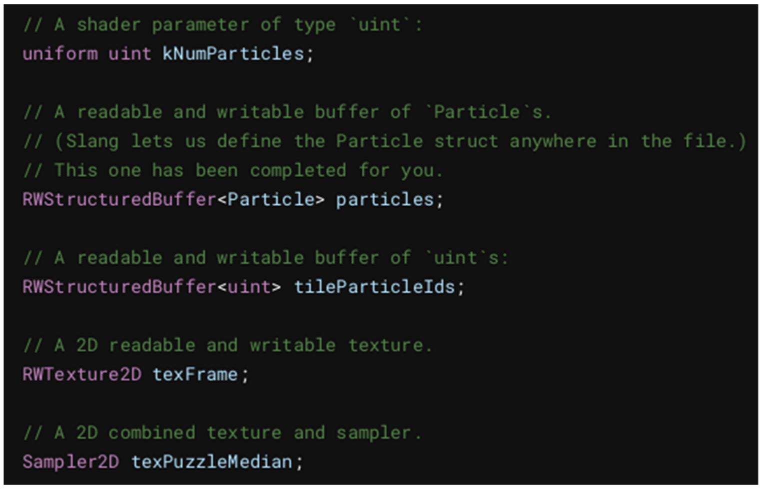
- `Texture2D`
- `SamplerState`

combined: `Sampler2D`

- `ByteAddressBuffer`
- `StructuredBuffer<T>`
- `uniform` *(shader parameter)*
- `ConstantBuffer<T>`
- `ParameterBlock<T>`
- `DescriptorHandle<T>`

ANSWER

SIGGRAPH 2025
Vancouver+ 10-14 August



```
// 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;
```

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.

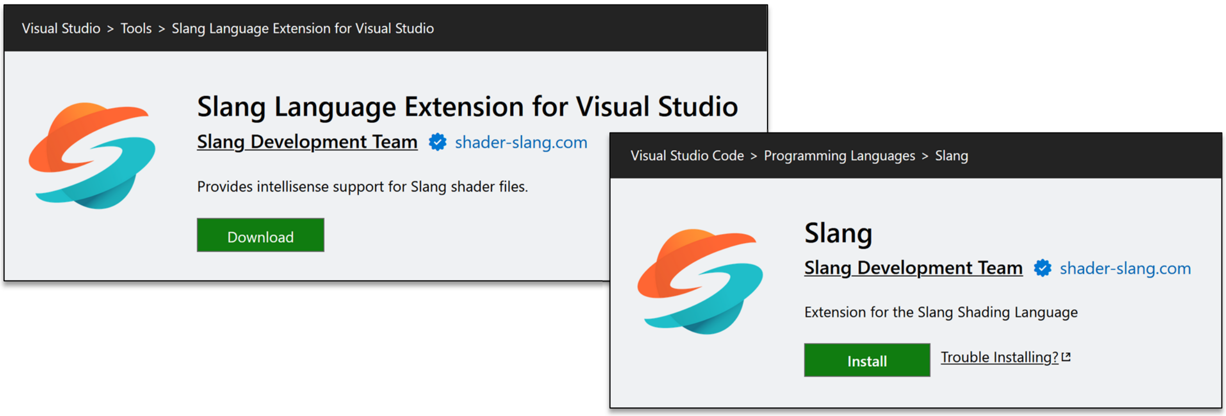
SIGGRAPH 2025
Vancouver+ 10-14 August

Tools & Debugging

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

WRITING SLANG

- Slang extensions for Visual Studio and Visual Studio Code



Visual Studio > Tools > Slang Language Extension for Visual Studio

Slang Language Extension for Visual Studio

Slang Development Team shader-slang.com

Provides intellisense support for Slang shader files.

Download

Visual Studio Code > Programming Languages > Slang

Slang

Slang Development Team shader-slang.com

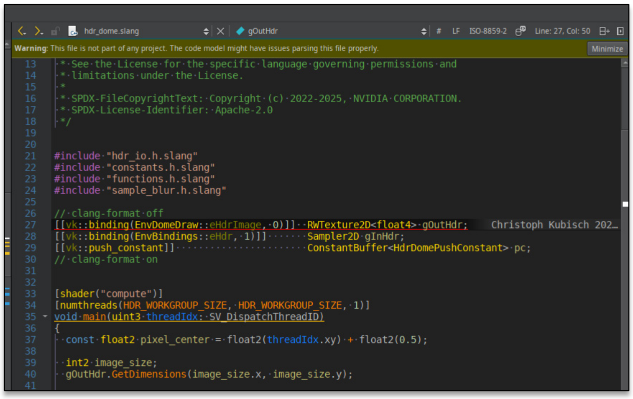
Extension for the Slang Shading Language

Install Trouble Installing?

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

WRITING SLANG

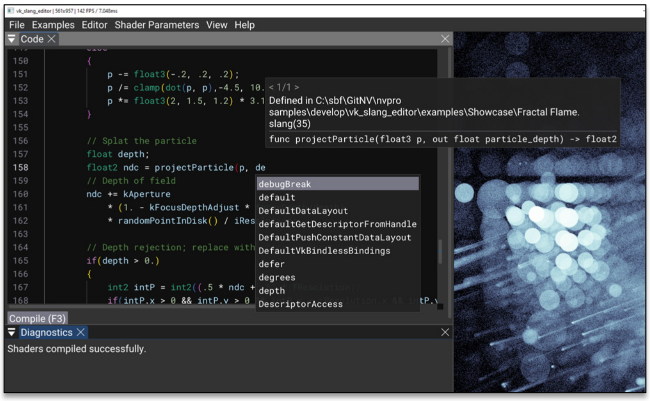
- **Any editor** that supports the Language Server Protocol can use **`slangd`**



Warning: This file is not part of any project. The code model might have issues parsing this file properly.

```
 * See the License for the specific language governing permissions and
 * limitations under the License.
 *
 * SPDX-FileCopyrightText: Copyright (c) 2022-2025, NVIDIA CORPORATION.
 * SPDX-License-Identifier: Apache-2.0
 */


#include "hdr_io.h.slang"
#include "constants.h.slang"
#include "functions.h.slang"
#include "sample_blur.h.slang"

// clang-format off
[[vk::binding(EnvDomeDraw::eHdrImage, 0)]]  RWTexture2D<float4> gOutHdr;     Christoph Kubisch 202…
[[vk::binding(EnvBindings::eHdr, 1)]]       Sampler2D gInHdr;
[[vk::push_constant]]                       ConstantBuffer<HdrDomePushConstant> pc;
// clang-format on


[shader("compute")]
[numthreads(HDR_WORKGROUP_SIZE, HDR_WORKGROUP_SIZE, 1)]
void main(uint3 threadIdx: SV_DispatchThreadID)
{
  const float2 pixel_center = float2(threadIdx.xy) + float2(0.5);

  int2 image_size;
  gOutHdr.GetDimensions(image_size.x, image_size.y);

```

QtCreator



```
        {
            p -= float3(-.2, .2, .2);
            p /= clamp(dot(p, p),-4.5, 10.
            p *= float3(2, 1.5, 1.2) * 3.1
        }

        // Splat the particle
        float depth;
        float2 ndc = projectParticle(p, de
        // Depth of field
        ndc += kAperture
            * (1. - kFocusDepthAdjust *
            * randomPointInDisk() / iRes

        // Depth rejection; replace with
        if(depth > 0.)
        {
            int2 intP = int2((.5 * ndc +
            if(intP.x > 0 && intP.y > 0                              tP.\
```

```
< 1/1 >
Defined in C:\sbf\GitNV\nvpro
samples\develop\vk_slang_editor\examples\Showcase\Fractal Flame.
slang(35)
func projectParticle(float3 p, out float particle_depth) -> float2
```

```
debugBreak
default
DefaultDataLayout
defaultGetDescriptorFromHandle
DefaultPushConstantDataLayout
DefaultVkBindlessBindings
defer
degrees
depth
DescriptorAccess
```

Shaders compiled successfully.

This lab's editor

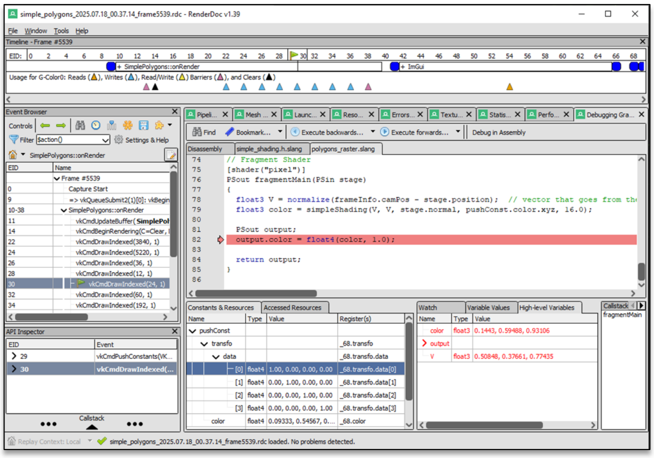
In addition, any editor that supports the Language Server Protocol, which powers things like syntax highlighting, Intellisense, jump to function, diagnostics, and much more for languages such as Python and Rust, can also use Slang's language server, slangd.

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

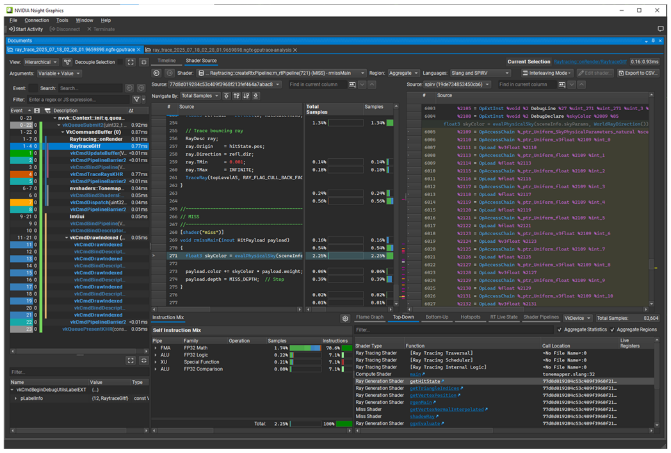
SIGGRAPH 2025
Vancouver+ 10-14 August



```
// Fragment Shader
[shader("pixel")]
PSout fragmentMain(PSin stage)
{
  float3 V = normalize(frameInfo.camPos - stage.position);  // vector that goes from the
  float3 color = simpleShading(V, V, stage.normal, pushConst.color.xyz, 16.0);

  PSout output;
  output.color = float4(color, 1.0);

  return output;
}
```

Name	Type	Value
color	float3	0.1443, 0.59488, 0.93106
output		
V	float3	0.50848, 0.37661, 0.77435

RenderDoc



Nsight Graphics

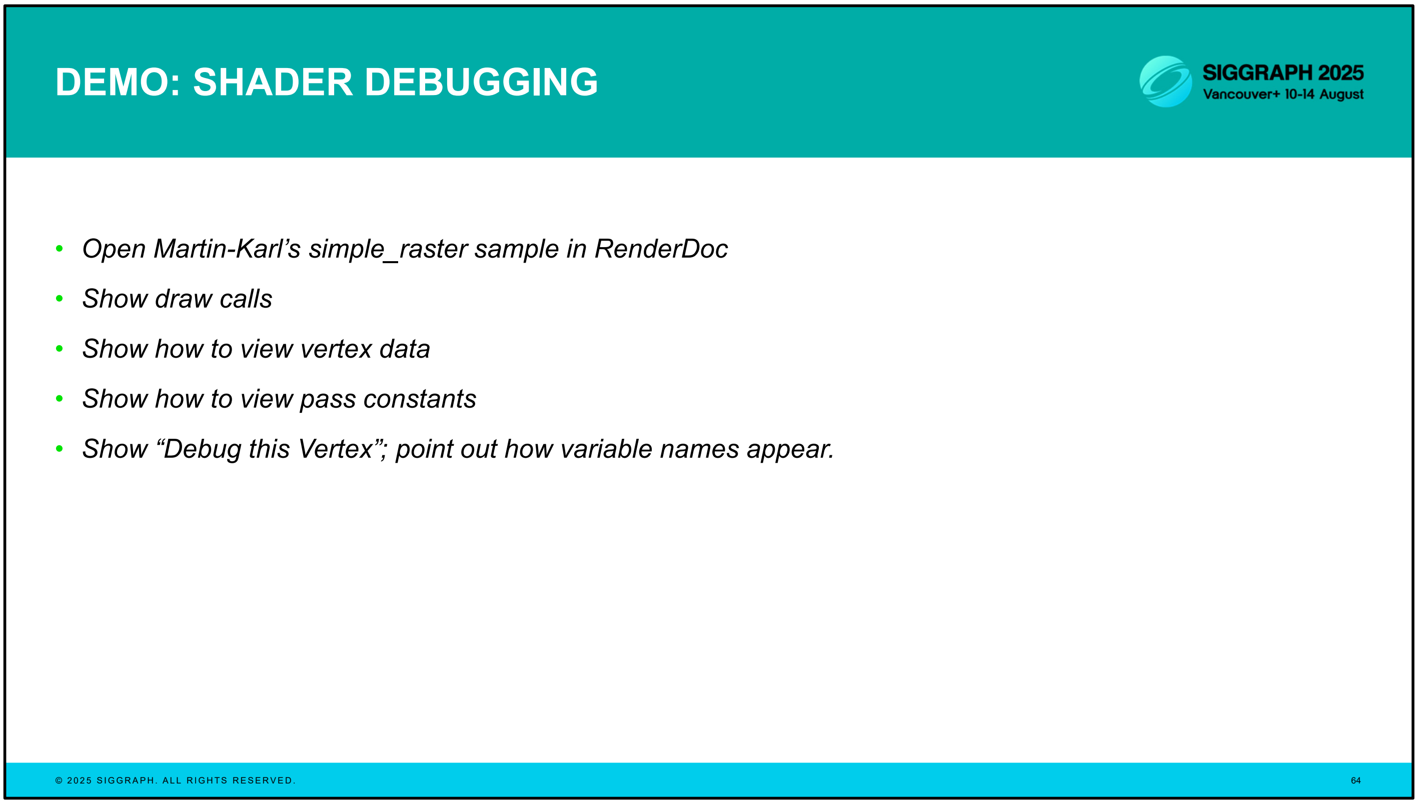
Debugging lab at SIGGRAPH: *NVIDIA Nsight Graphics in Action*, Wednesday, 9–10:30 AM, Room 116–117

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

SIGGRAPH 2025
Vancouver+ 10-14 August

Demo:
Shader Debugging

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



DEMO: SHADER DEBUGGING

SIGGRAPH 2025
Vancouver+ 10-14 August

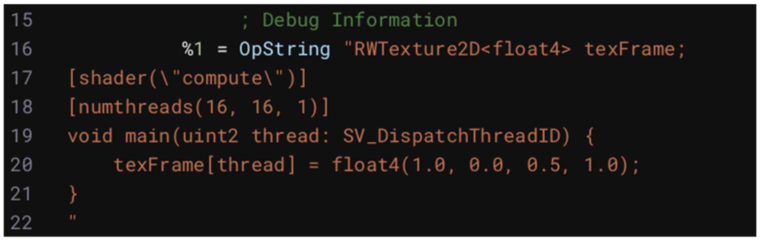
- *Open Martin-Karl’s simple_raster sample in RenderDoc*
- *Show draw calls*
- *Show how to view vertex data*
- *Show how to view pass constants*
- *Show “Debug this Vertex”; point out how variable names appear.*

Here’s an example!

DEBUG INFO

SIGGRAPH 2025
Vancouver+ 10-14 August

- Embed shader source code using `–g1`



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

- 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.

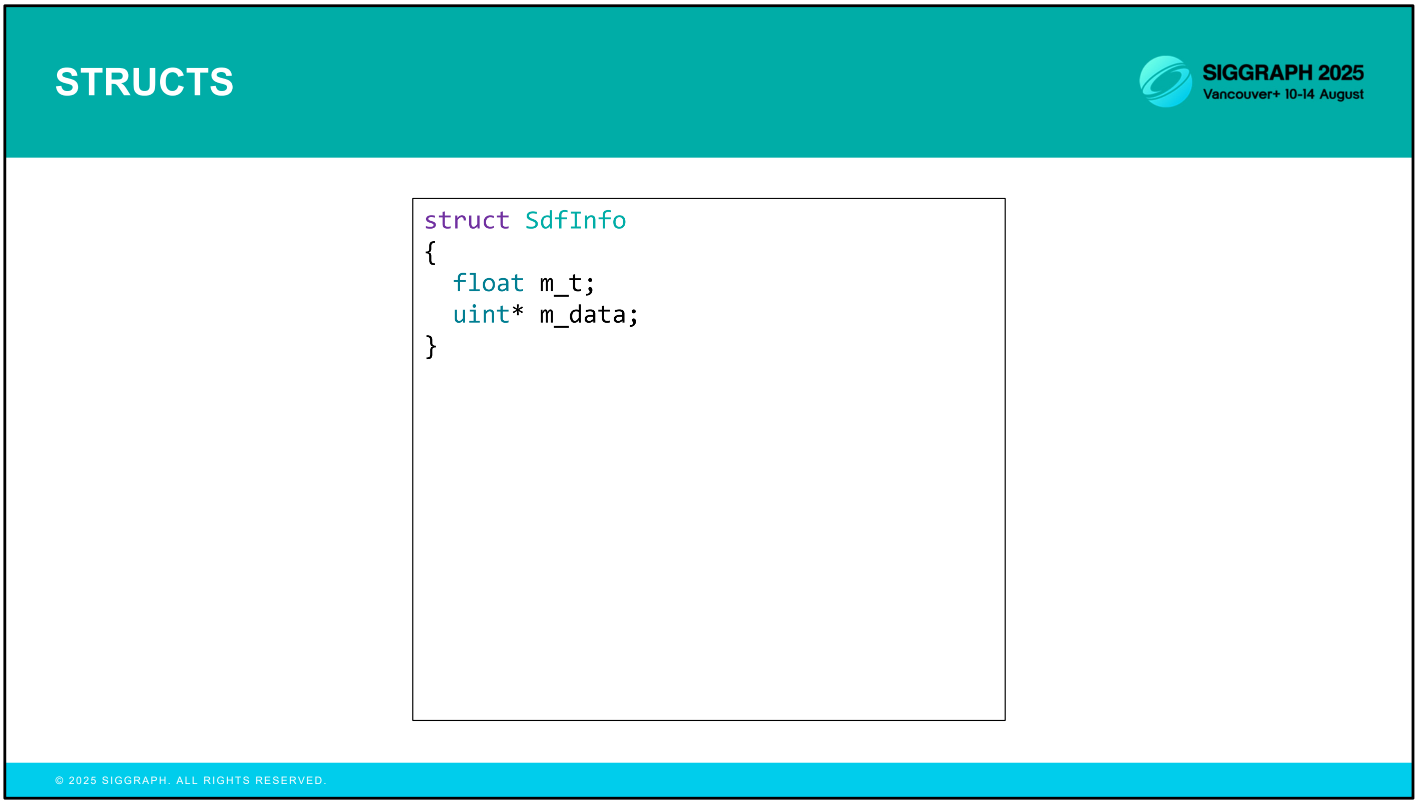
You can remove debug info by setting –line-directive-mode to none.

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

SIGGRAPH 2025
Vancouver+ 10-14 August

Structs, Modules, and Interfaces

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



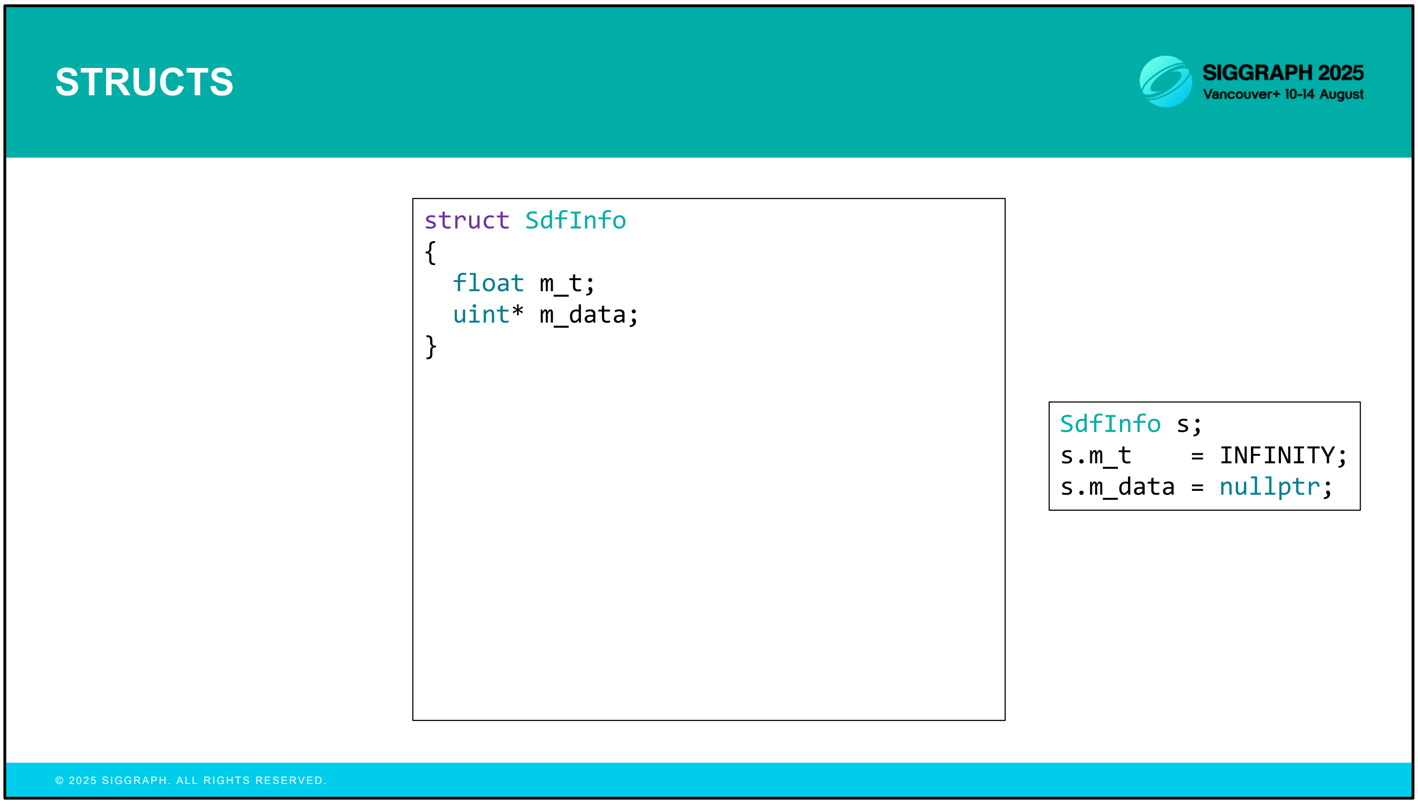
STRUCTS

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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.



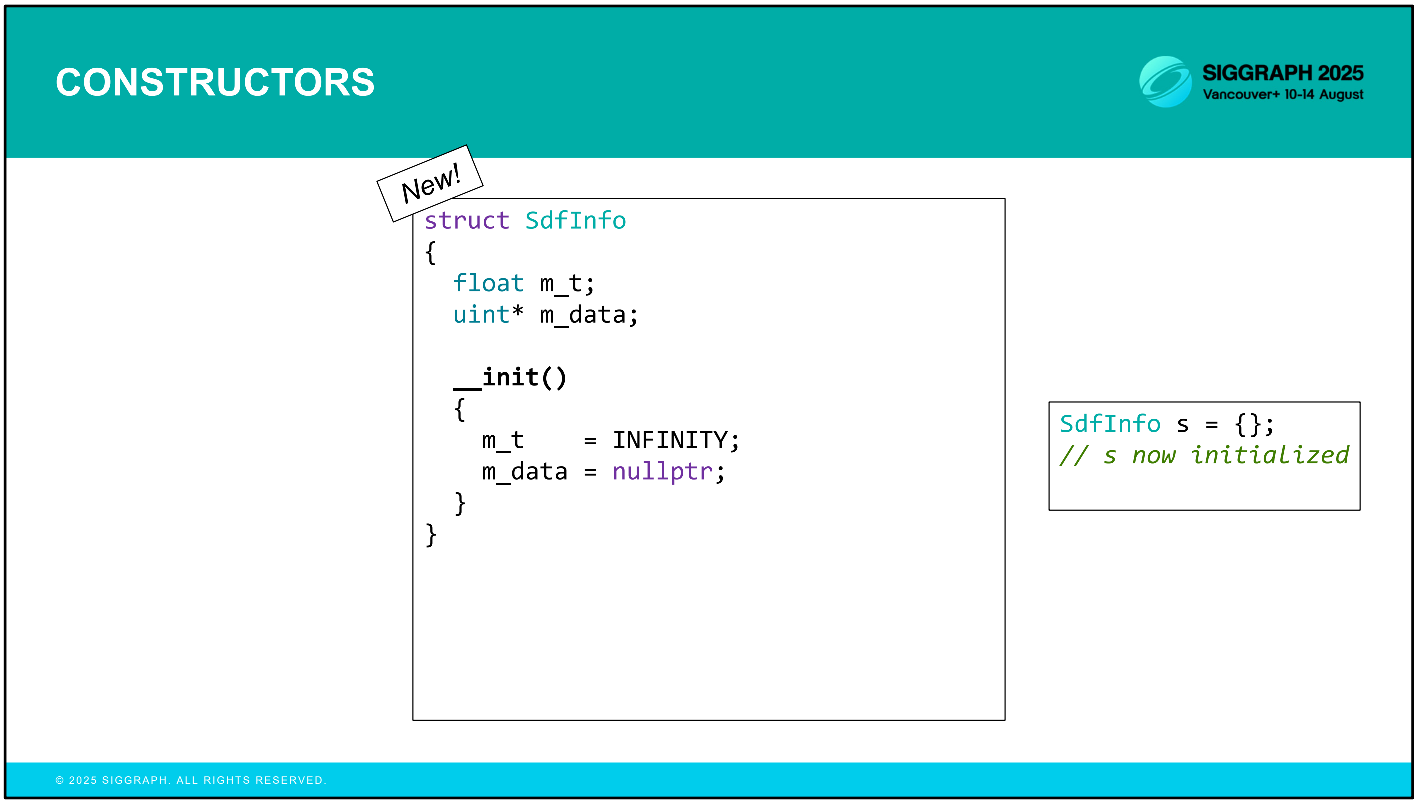
STRUCTS

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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.

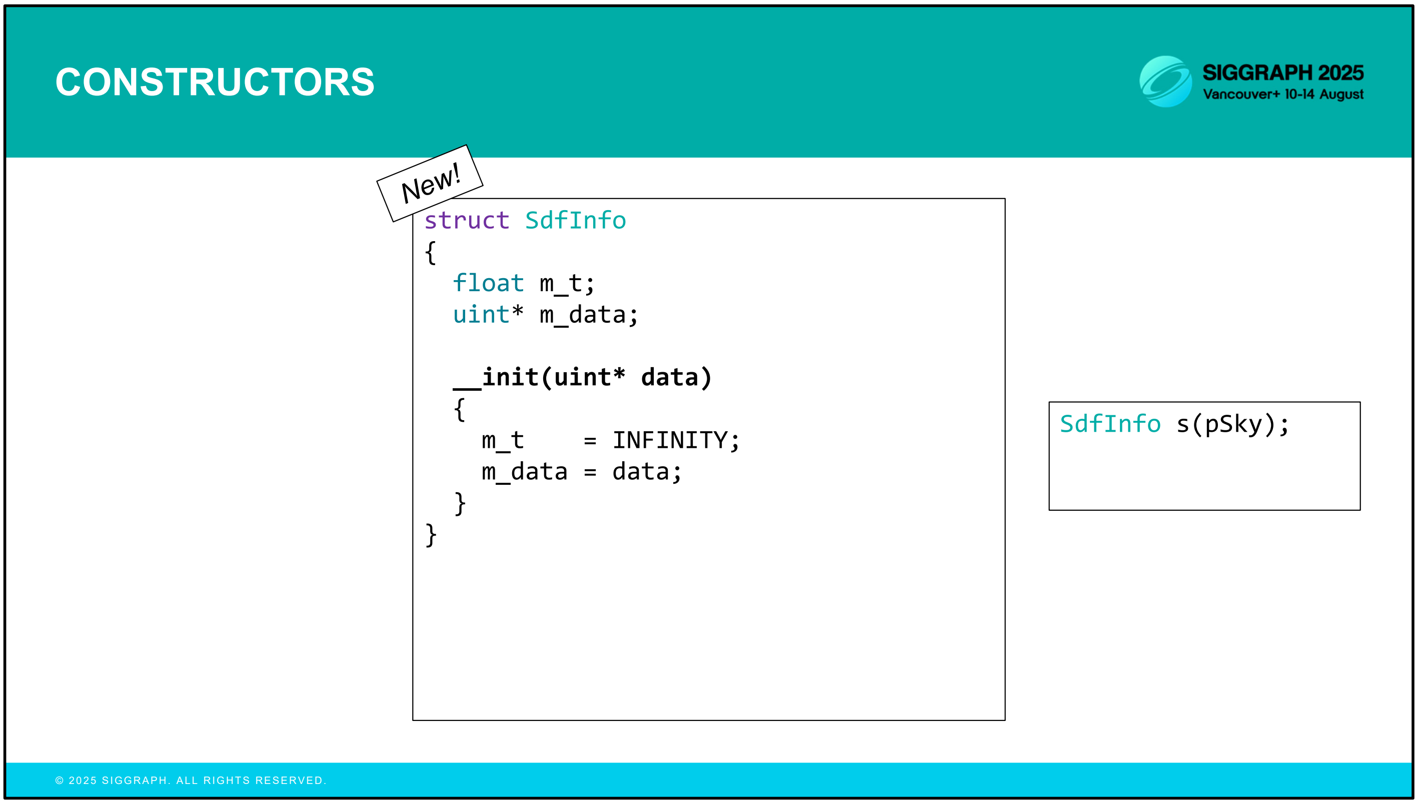


CONSTRUCTORS

```
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.



CONSTRUCTORS

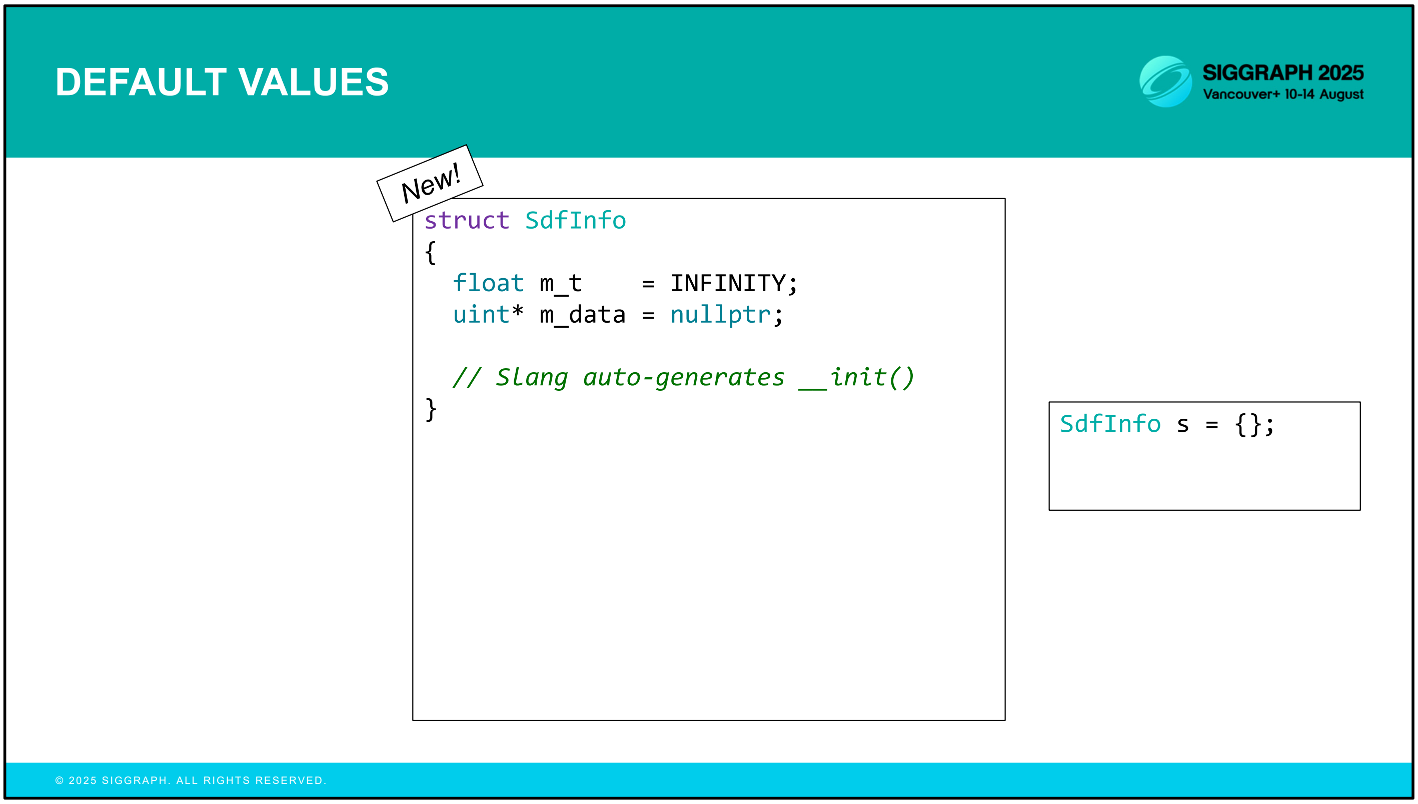
SIGGRAPH 2025
Vancouver+ 10-14 August

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



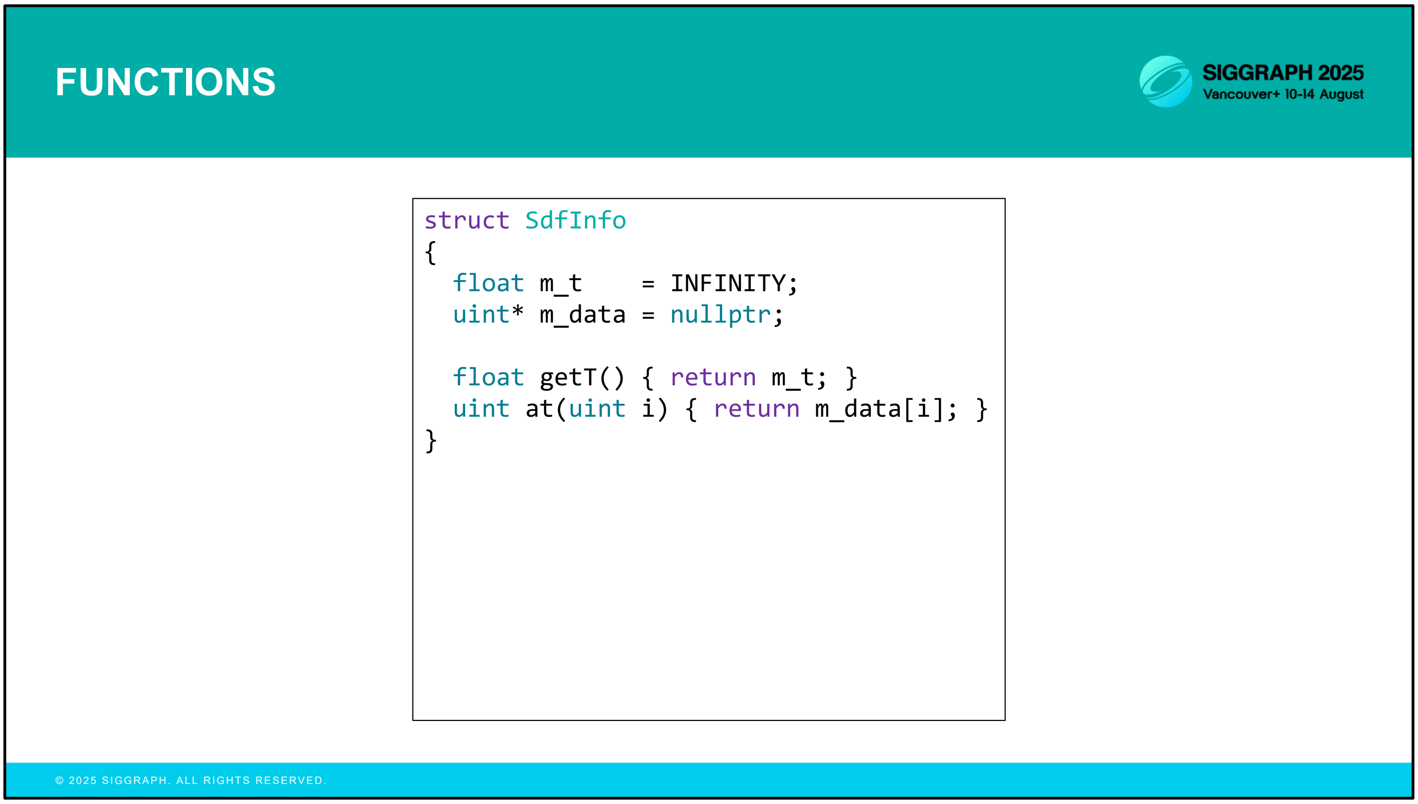
DEFAULT VALUES

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.

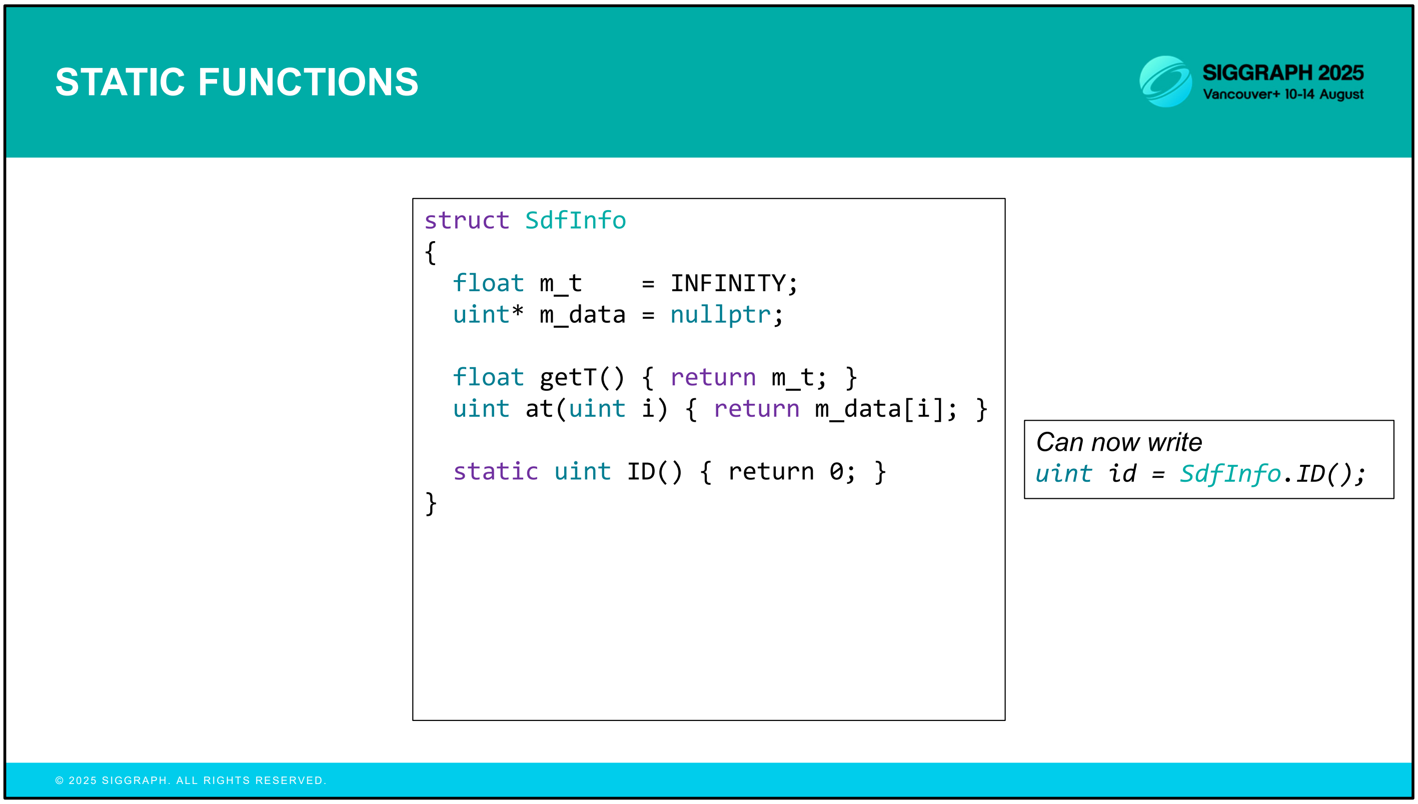


FUNCTIONS

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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.



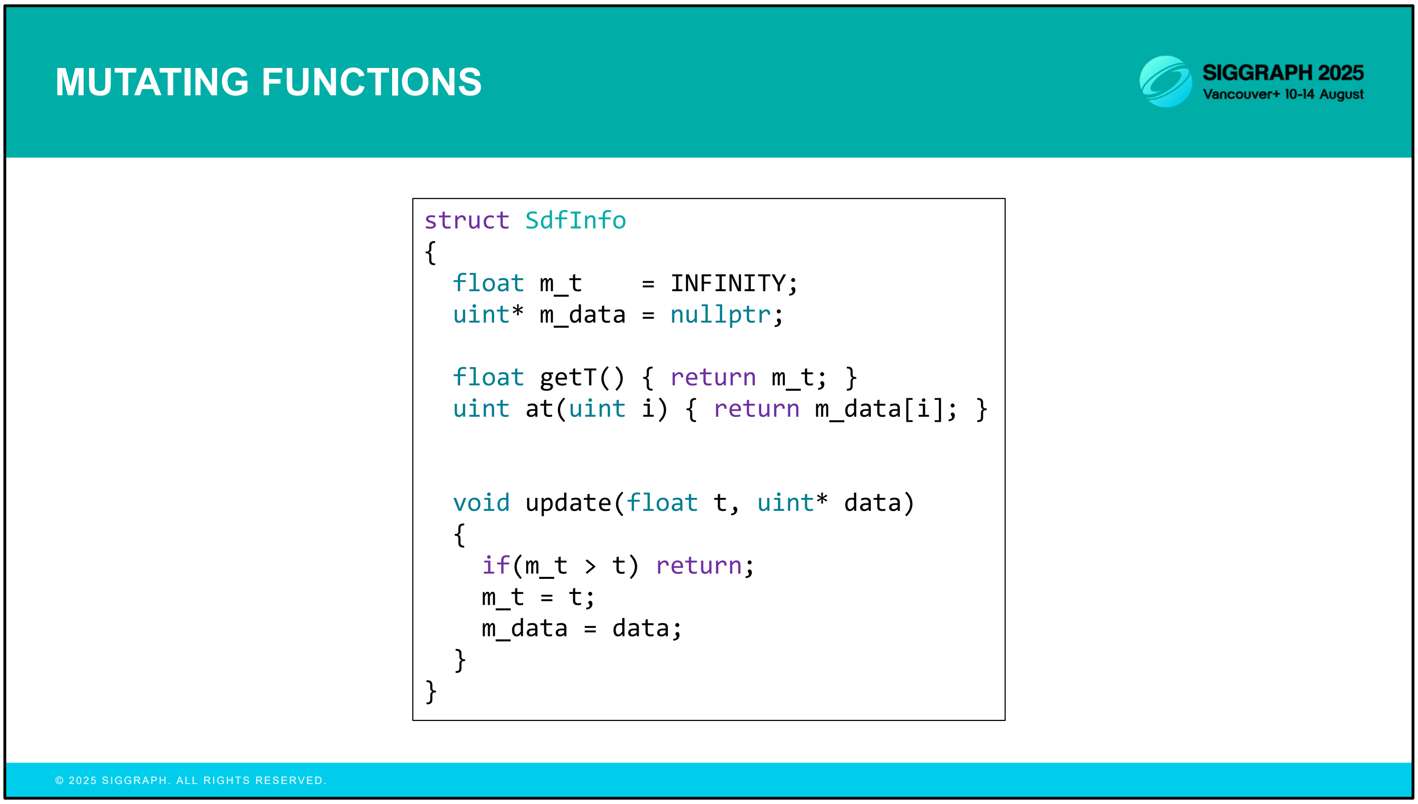
STATIC FUNCTIONS

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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().

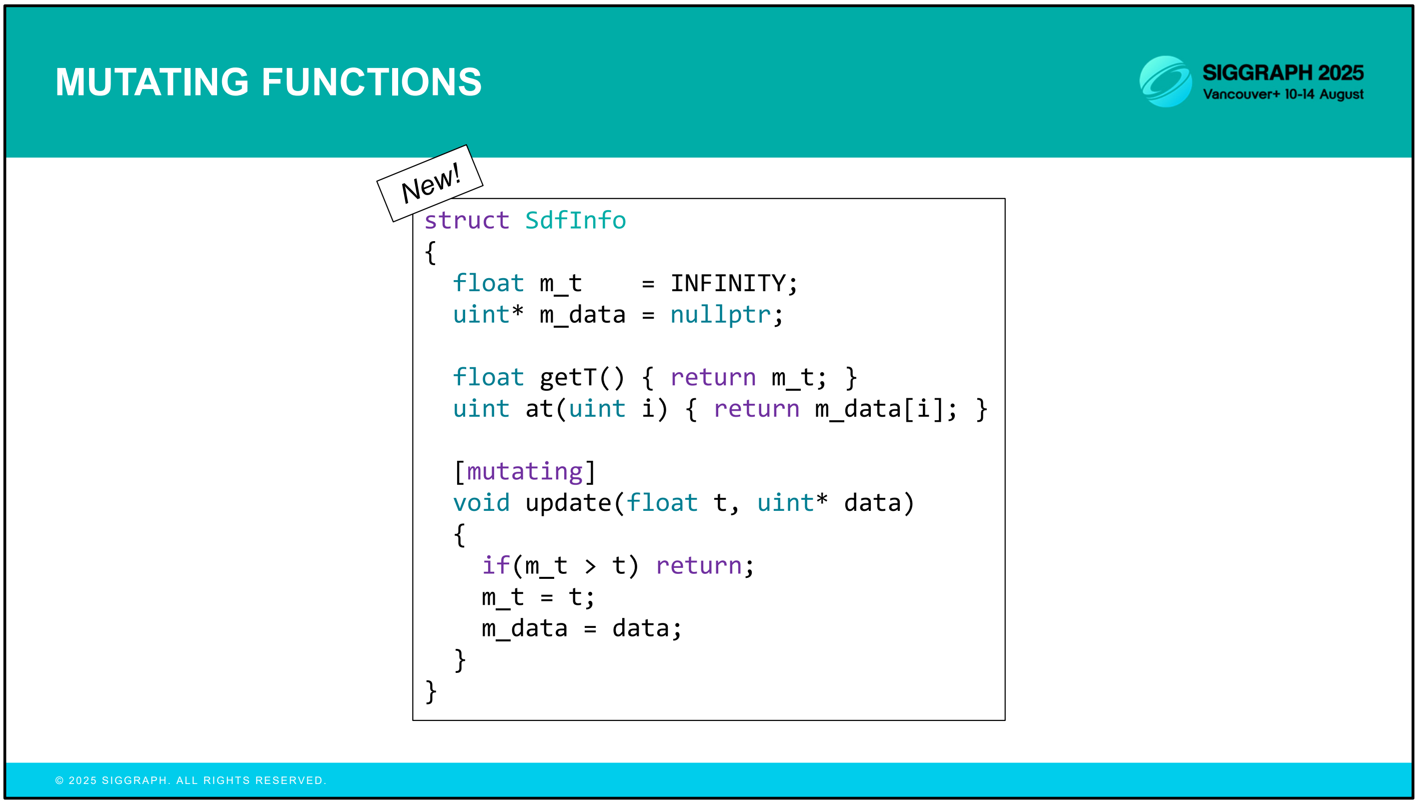


MUTATING FUNCTIONS

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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)**

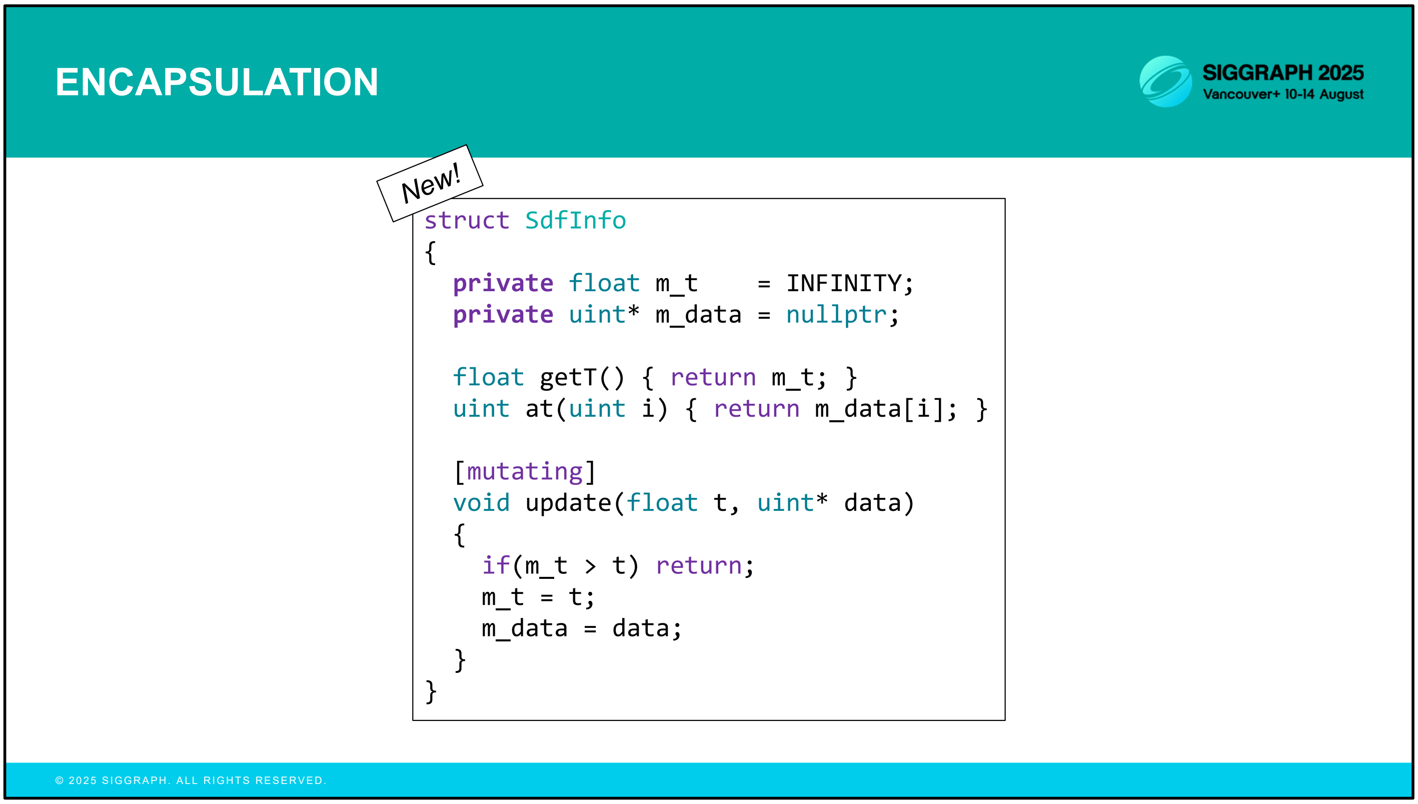


MUTATING FUNCTIONS

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].



ENCAPSULATION

SIGGRAPH 2025
Vancouver+ 10-14 August

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;
  }
}
```

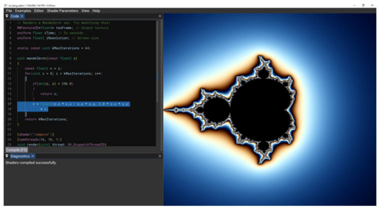
Slang also lets you make member variables and functions private using the private keyword. This is useful for the same reasons as in C++ and other languages.

OPERATOR OVERLOADING

SIGGRAPH 2025
Vancouver+ 10-14 August

```
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;
```



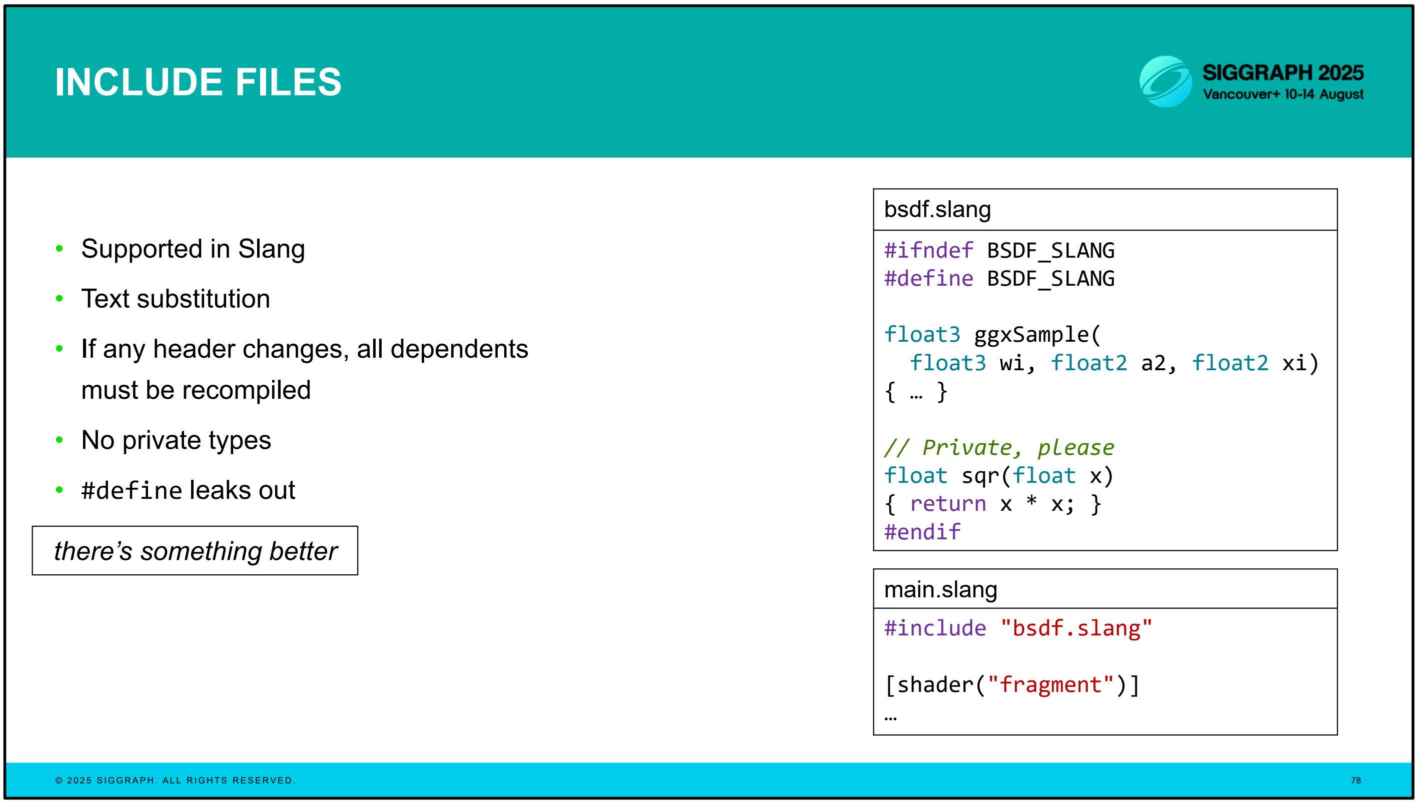
```
// Renders a Mandelbrot set. Try modifying this!
RWTexture2D<float4> texFrame; // Output texture
uniform float iTime; // In seconds
uniform float2 iResolution; // Screen size

static const uint kMaxIterations = 64;

uint mandelbrot(inout float2 z)
{
    const float2 c = z;
    for(uint i = 0; i < kMaxIterations; i++)
    {
        if(dot(z, z) > 256.0)
        {
            return i;
        }
        z = float2(z.x * z.x - z.y * z.y, 2.0 * z.x * z.y)
          + c;
    }
    return kMaxIterations;
}

[shader("compute")]
[numthreads(16, 16, 1)]
void render(uint2 thread: SV_DispatchThreadID)
```

Shaders compiled successfully.

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.



INCLUDE FILES

SIGGRAPH 2025
Vancouver+ 10-14 August

- 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")]
…
```

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

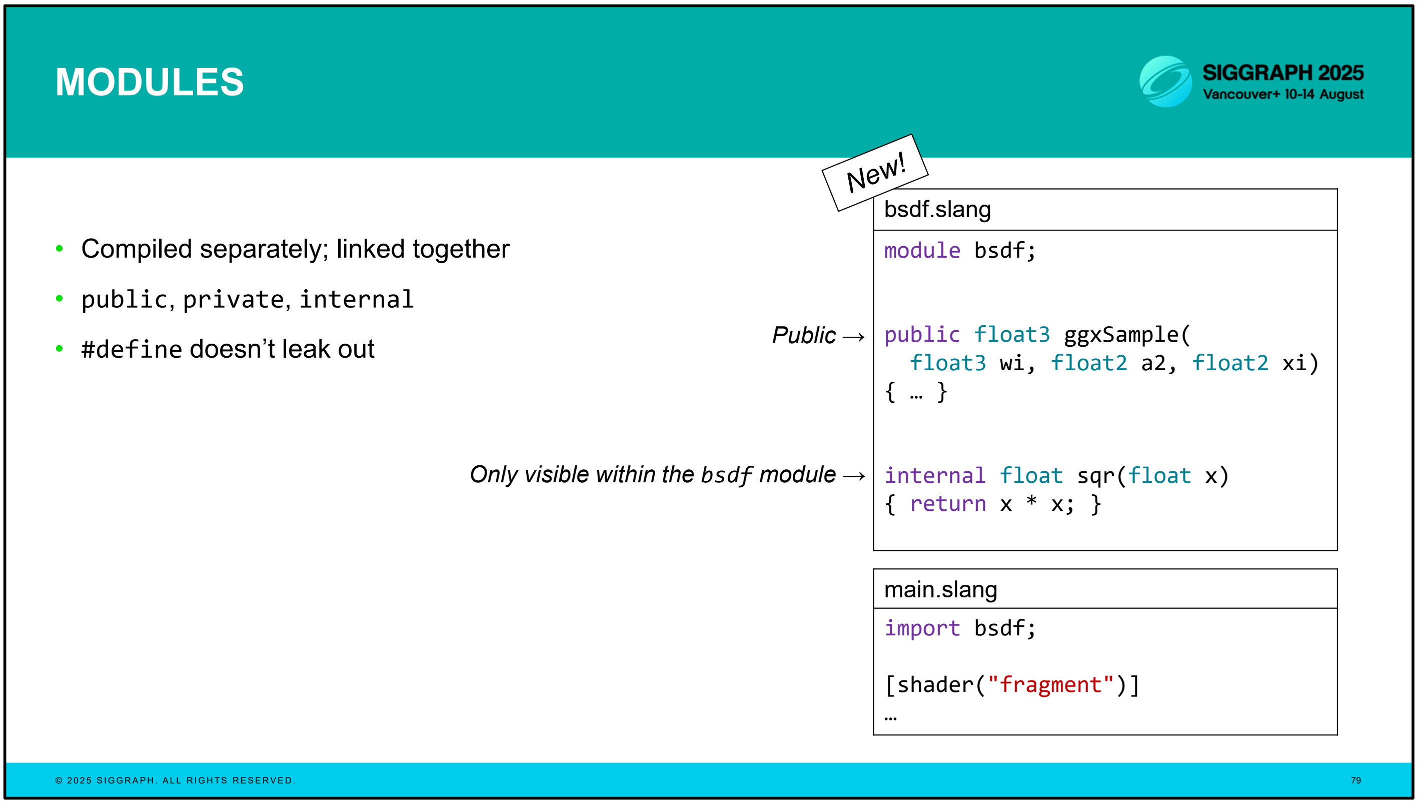
Slang lets you *include* other files, like other languages do. But this old technique has some downsides.

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 2025
Vancouver+ 10-14 August

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

New!

bsdf.slang

```
module bsdf;


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


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

Public →

Only visible within the `bsdf` module →

main.slang

```
import bsdf;

[shader("fragment")]
…
```

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).

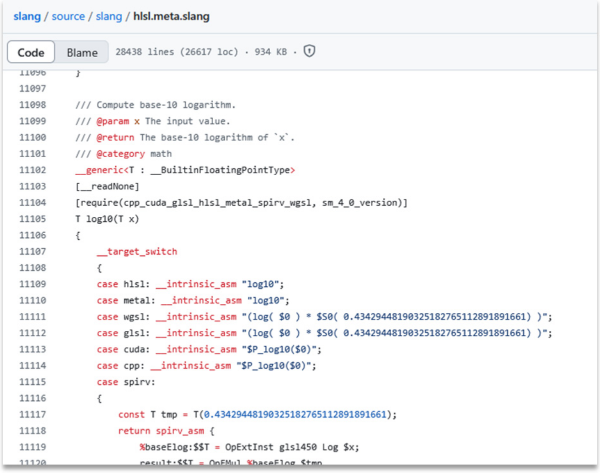
And, things like preprocessor macros don't affect other files.

(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 2025
Vancouver+ 10-14 August

hlsl.meta.slang

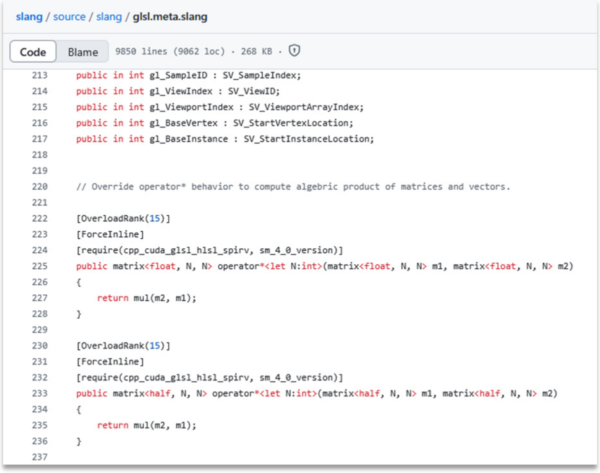


slang / source / slang / hlsl.meta.slang

28438 lines (26617 loc) · 934 KB ·

```
}

/// Compute base-10 logarithm.
/// @param x The input value.
/// @return The base-10 logarithm of `x`.
/// @category math
__generic<T : __BuiltinFloatingPointType>
[__readNone]
[require(cpp_cuda_glsl_hlsl_metal_spirv_wgsl, sm_4_0_version)]
T log10(T x)
{
    __target_switch
    {
    case hlsl: __intrinsic_asm "log10";
    case metal: __intrinsic_asm "log10";
    case wgsl: __intrinsic_asm "(log( $0 ) * $S0( 0.43429448190325182765112891891661) )";
    case glsl: __intrinsic_asm "(log( $0 ) * $S0( 0.43429448190325182765112891891661) )";
    case cuda: __intrinsic_asm "$P_log10($0)";
    case cpp: __intrinsic_asm "$P_log10($0)";
    case spirv:
    {
        const T tmp = T(0.43429448190325182765112891891661);
        return spirv_asm {
            %baseElog:$$T = OpExtInst glsl450 Log $x;
            result:$$T = OpFMul %baseElog $tmp
```

glsl.meta.slang



slang / source / slang / glsl.meta.slang

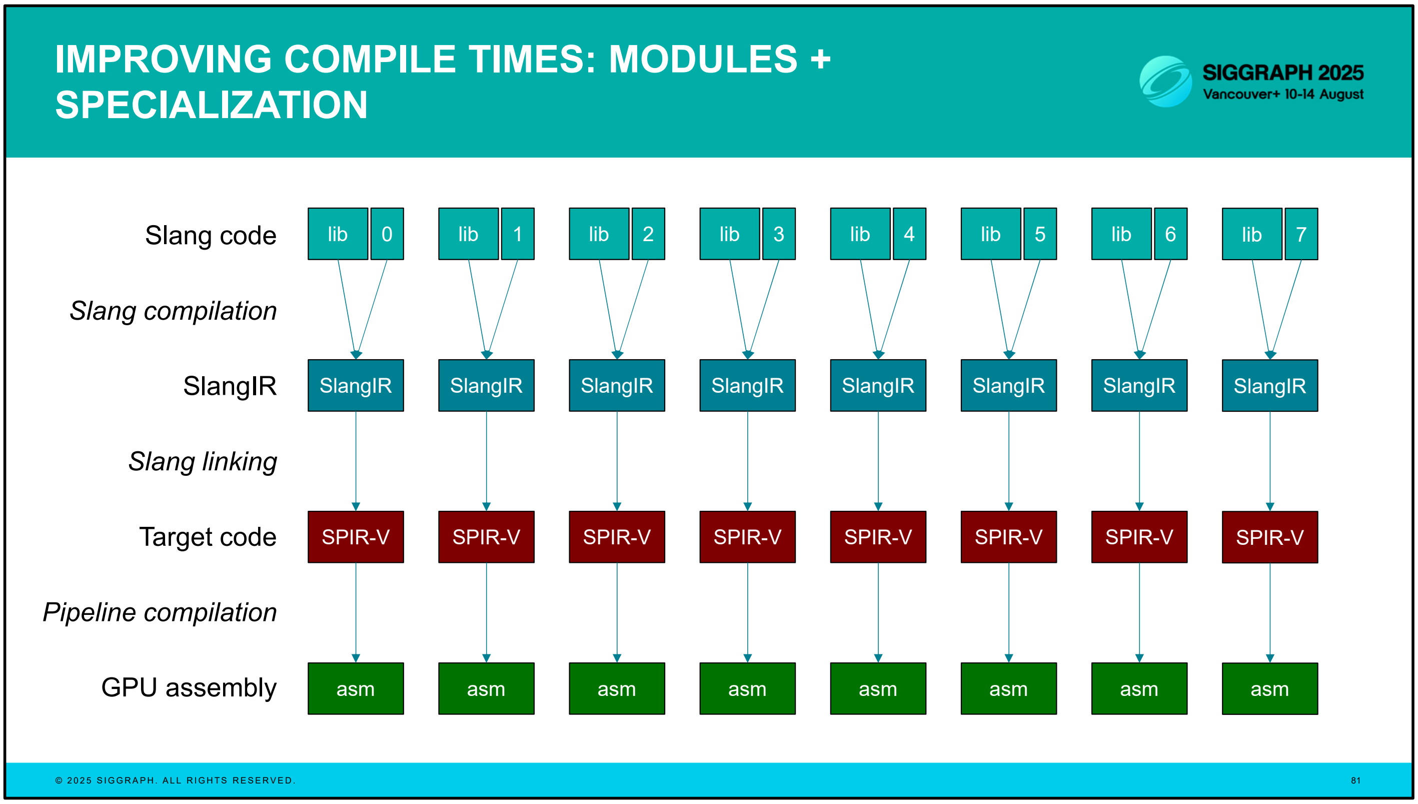
9850 lines (9062 loc) · 268 KB ·

```
public in int gl_SampleID : SV_SampleIndex;
public in int gl_ViewIndex : SV_ViewID;
public in int gl_ViewportIndex : SV_ViewportArrayIndex;
public in int gl_BaseVertex : SV_StartVertexLocation;
public in int gl_BaseInstance : SV_StartInstanceLocation;


// Override operator* behavior to compute algebric product of matrices and vectors.

[OverloadRank(15)]
[ForceInline]
[require(cpp_cuda_glsl_hlsl_spirv, sm_4_0_version)]
public matrix<float, N, N> operator*<let N:int>(matrix<float, N, N> m1, matrix<float, N, N> m2)
{
    return mul(m2, m1);
}

[OverloadRank(15)]
[ForceInline]
[require(cpp_cuda_glsl_hlsl_spirv, sm_4_0_version)]
public matrix<half, N, N> operator*<let N:int>(matrix<half, N, N> m1, matrix<half, N, N> m2)
{
    return mul(m2, m1);
}

```

Notably, Slang's standard library uses modules. All Slang files *link* with hlsl.meta.slang. And when you enable GLSL compatibility by adding #version 460, 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

SIGGRAPH 2025
Vancouver+ 10-14 August

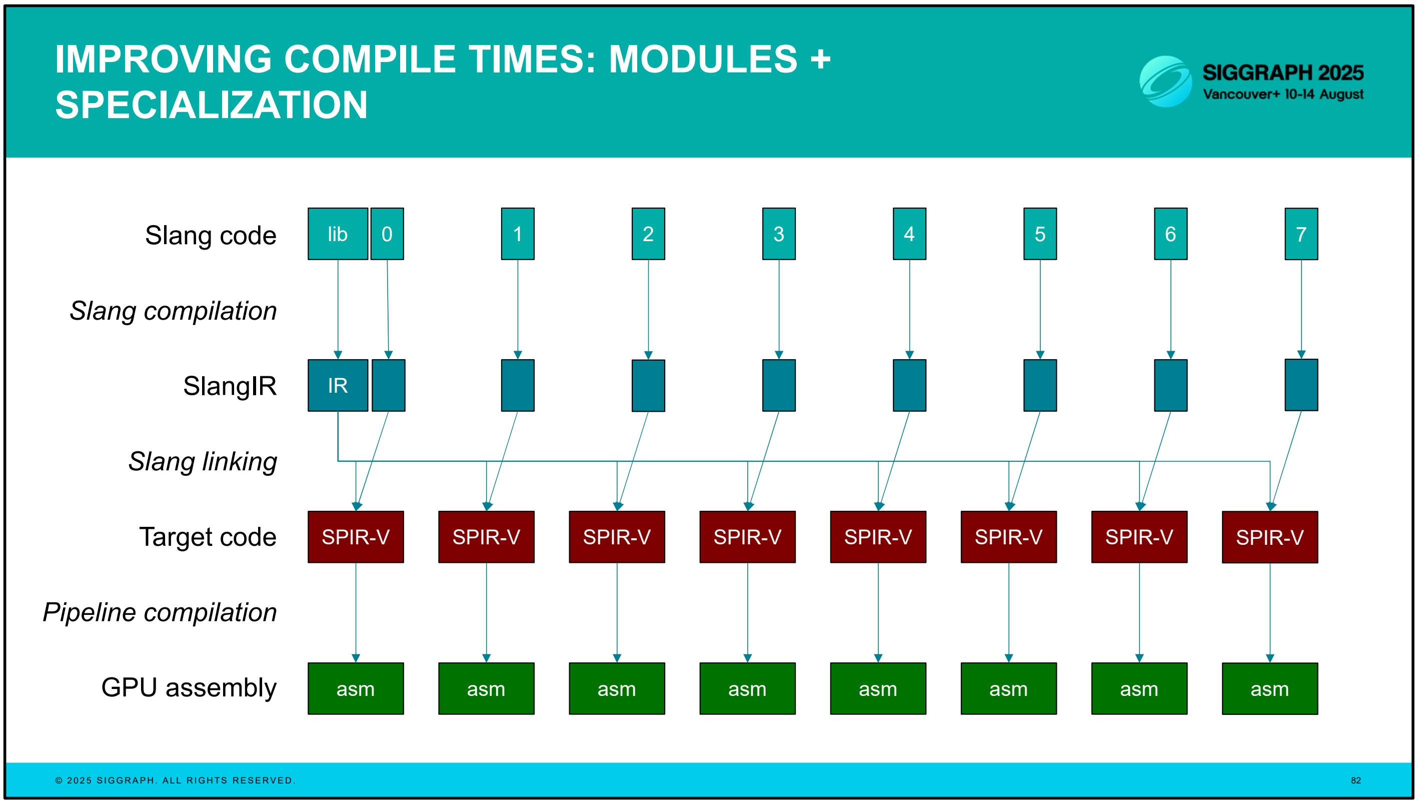
Slang code	lib 0	lib 1	lib 2	lib 3	lib 4	lib 5	lib 6	lib 7
Slang compilation								
SlangIR	SlangIR	SlangIR	SlangIR	SlangIR	SlangIR	SlangIR	SlangIR	SlangIR
Slang linking								
Target code	SPIR-V	SPIR-V	SPIR-V	SPIR-V	SPIR-V	SPIR-V	SPIR-V	SPIR-V
Pipeline compilation								
GPU assembly	asm	asm	asm	asm	asm	asm	asm	asm

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

Slang code: lib, 0, 1, 2, 3, 4, 5, 6, 7

Slang compilation

SlangIR: IR

Slang linking

Target code: SPIR-V, SPIR-V, SPIR-V, SPIR-V, SPIR-V, SPIR-V, SPIR-V, SPIR-V

Pipeline compilation

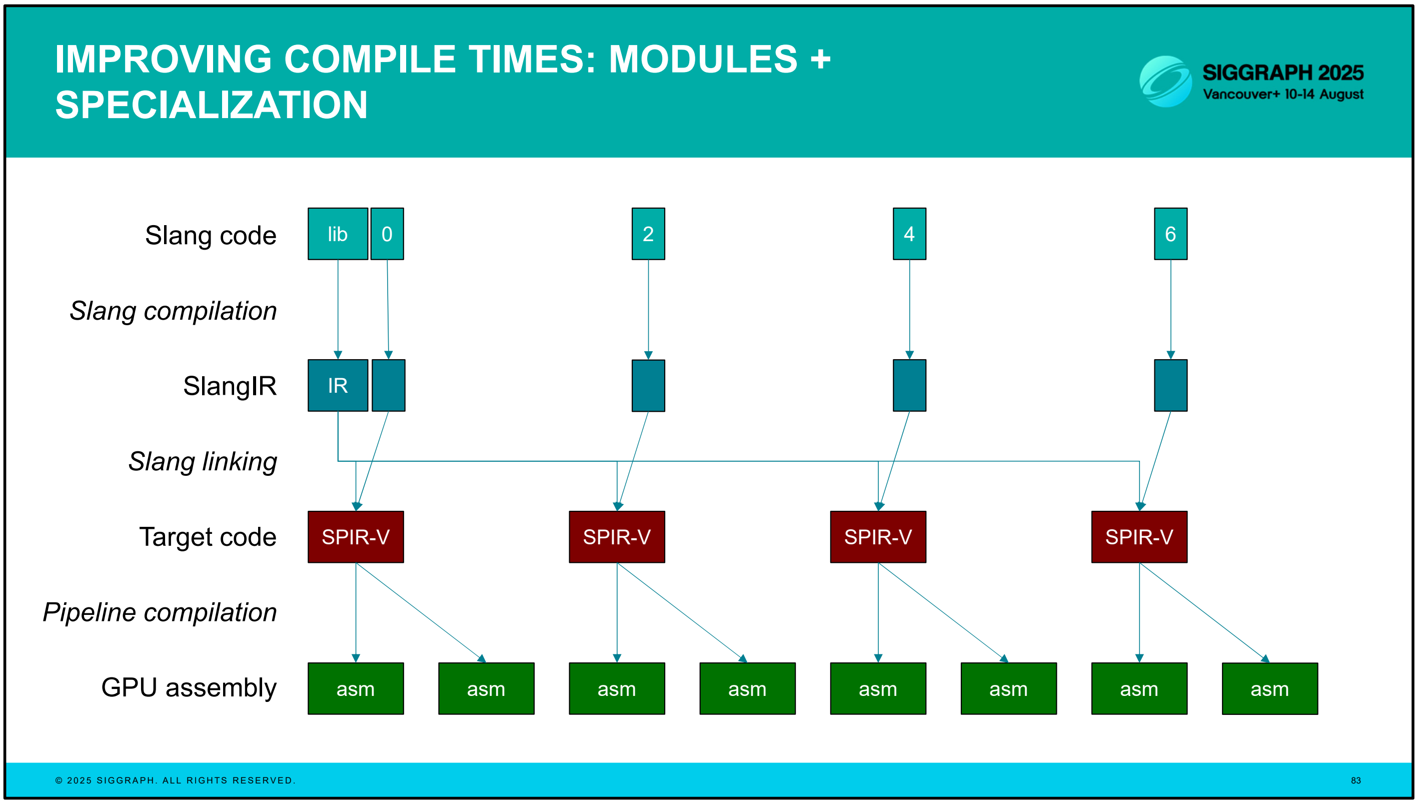
GPU assembly: asm, asm, asm, asm, asm, asm, asm, asm

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

SIGGRAPH 2025
Vancouver+ 10-14 August

Slang code: lib, 0, 2, 4, 6

Slang compilation

SlangIR: IR

Slang linking

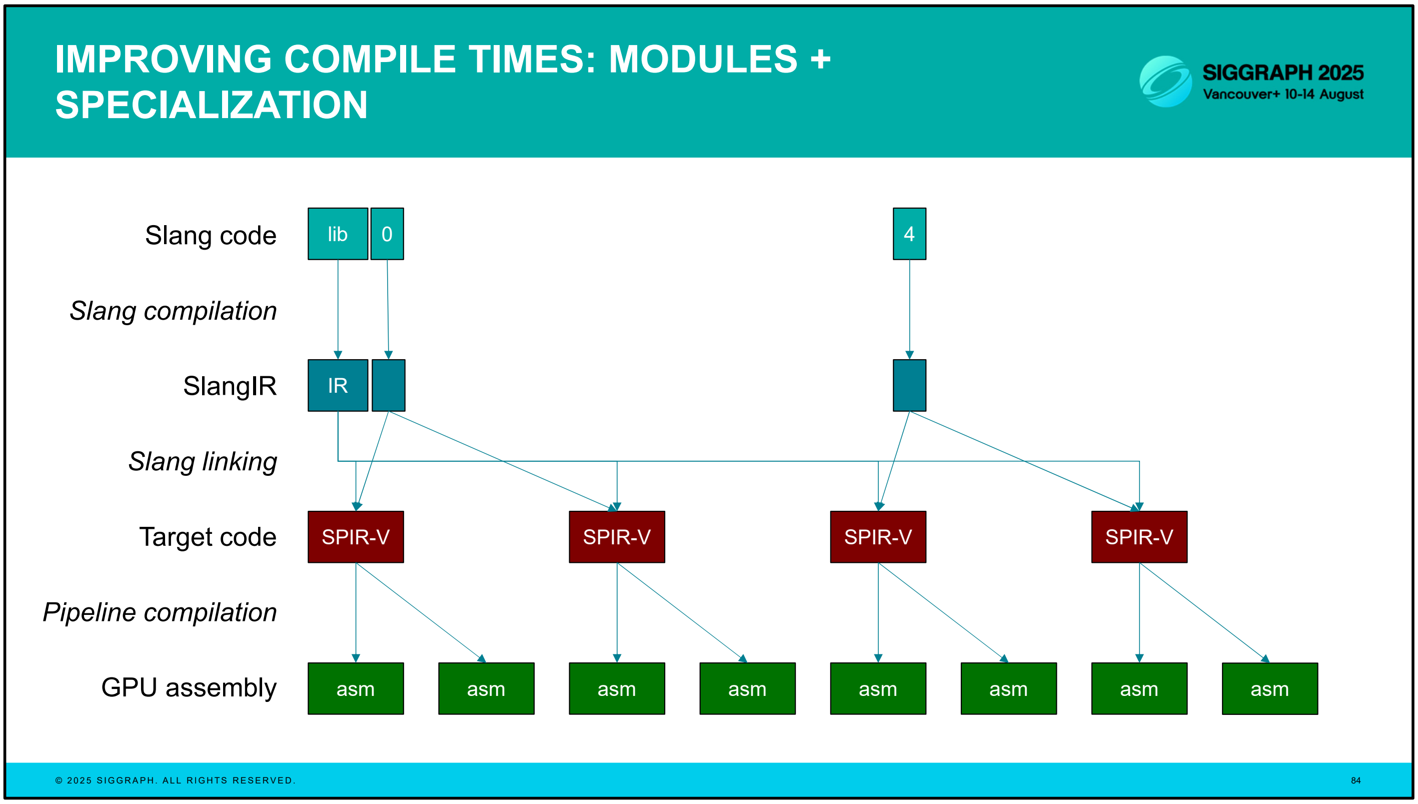
Target code: SPIR-V, SPIR-V, SPIR-V, SPIR-V

Pipeline compilation

GPU assembly: asm, asm, asm, asm, asm, asm, asm, asm

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

SIGGRAPH 2025
Vancouver+ 10-14 August

Slang code: lib, 0, 4

Slang compilation

SlangIR: IR

Slang linking

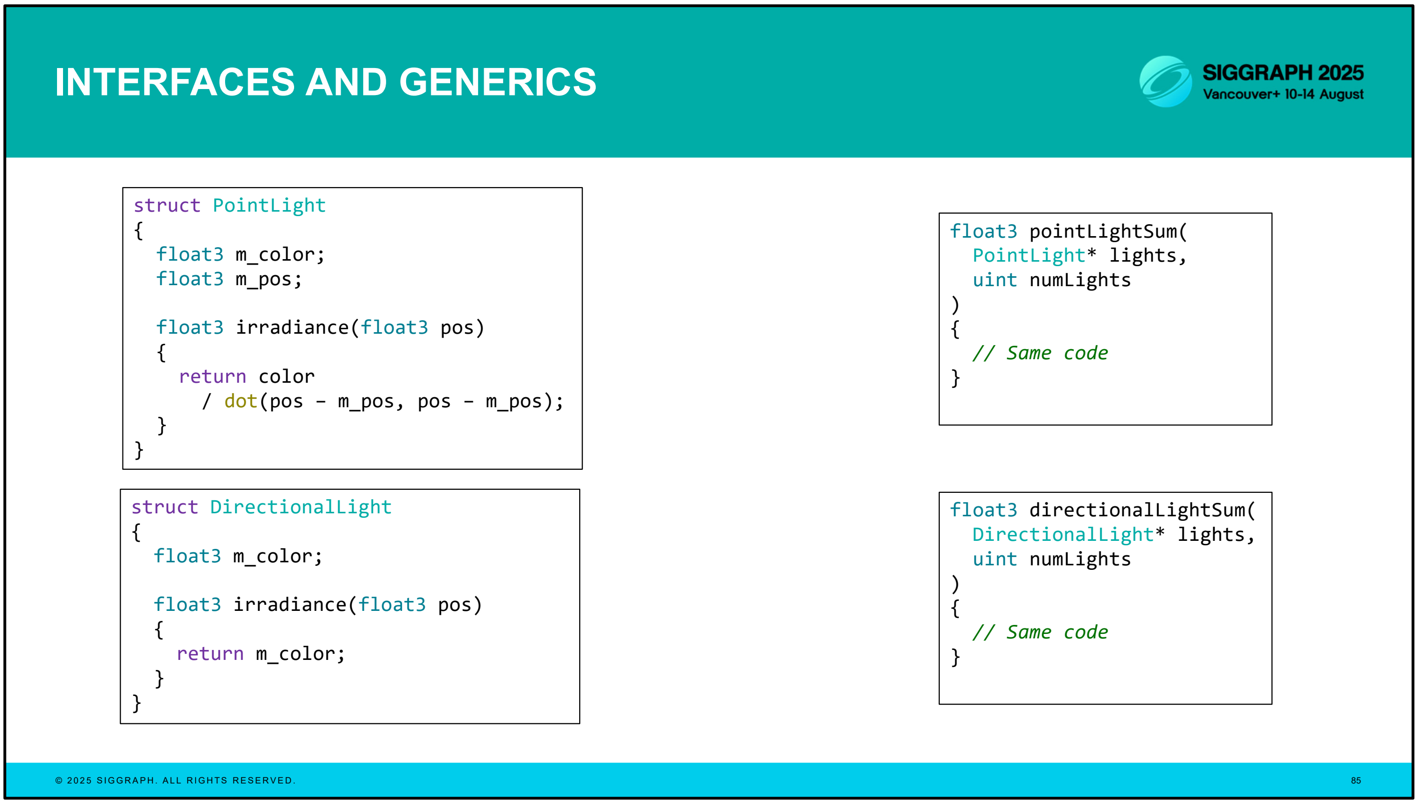
Target code: SPIR-V, SPIR-V, SPIR-V, SPIR-V

Pipeline compilation

GPU assembly: asm, asm, asm, asm, asm, asm, asm, asm

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.



INTERFACES AND GENERICS

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

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

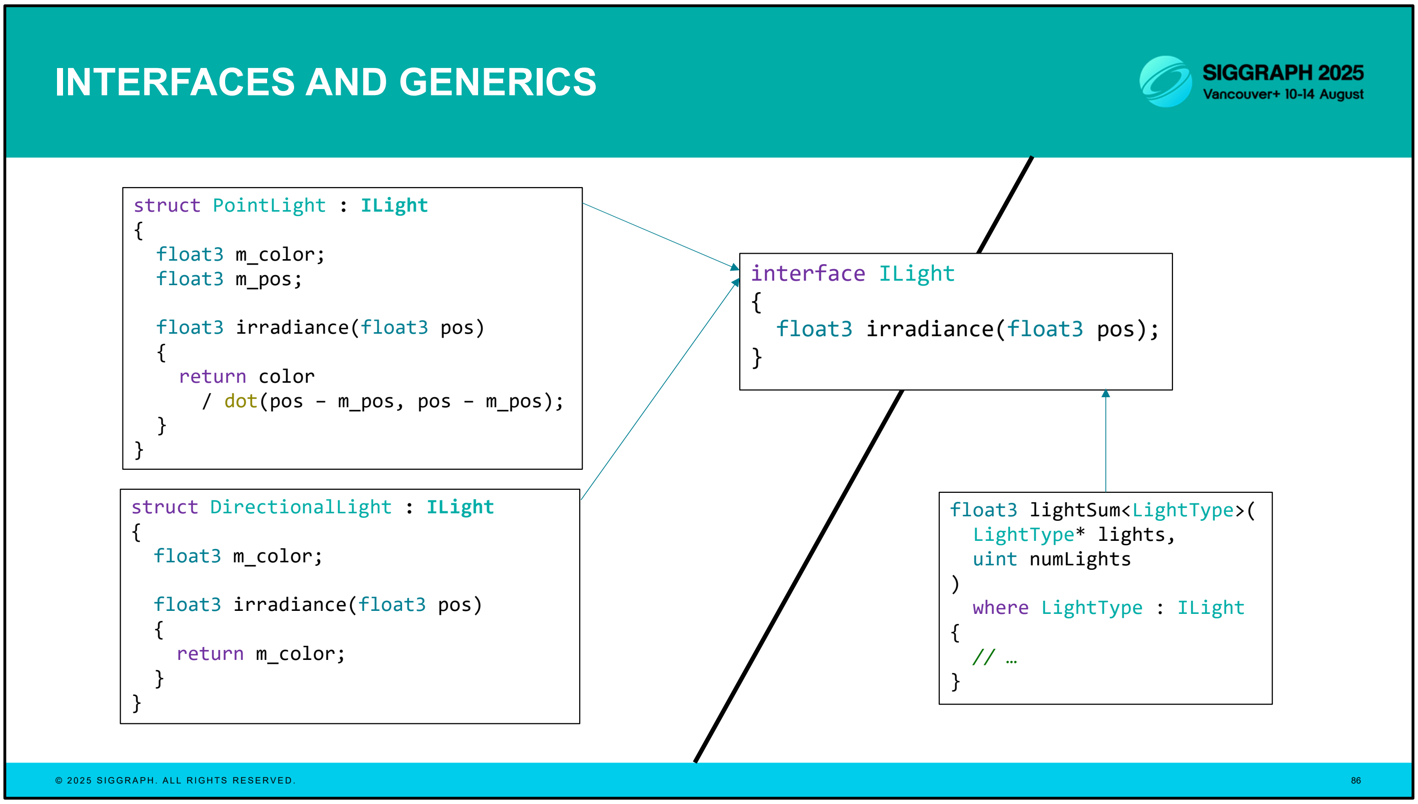
```
struct DirectionalLight
{
  float3 m_color;

  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.



INTERFACES AND GENERICS

```
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)**

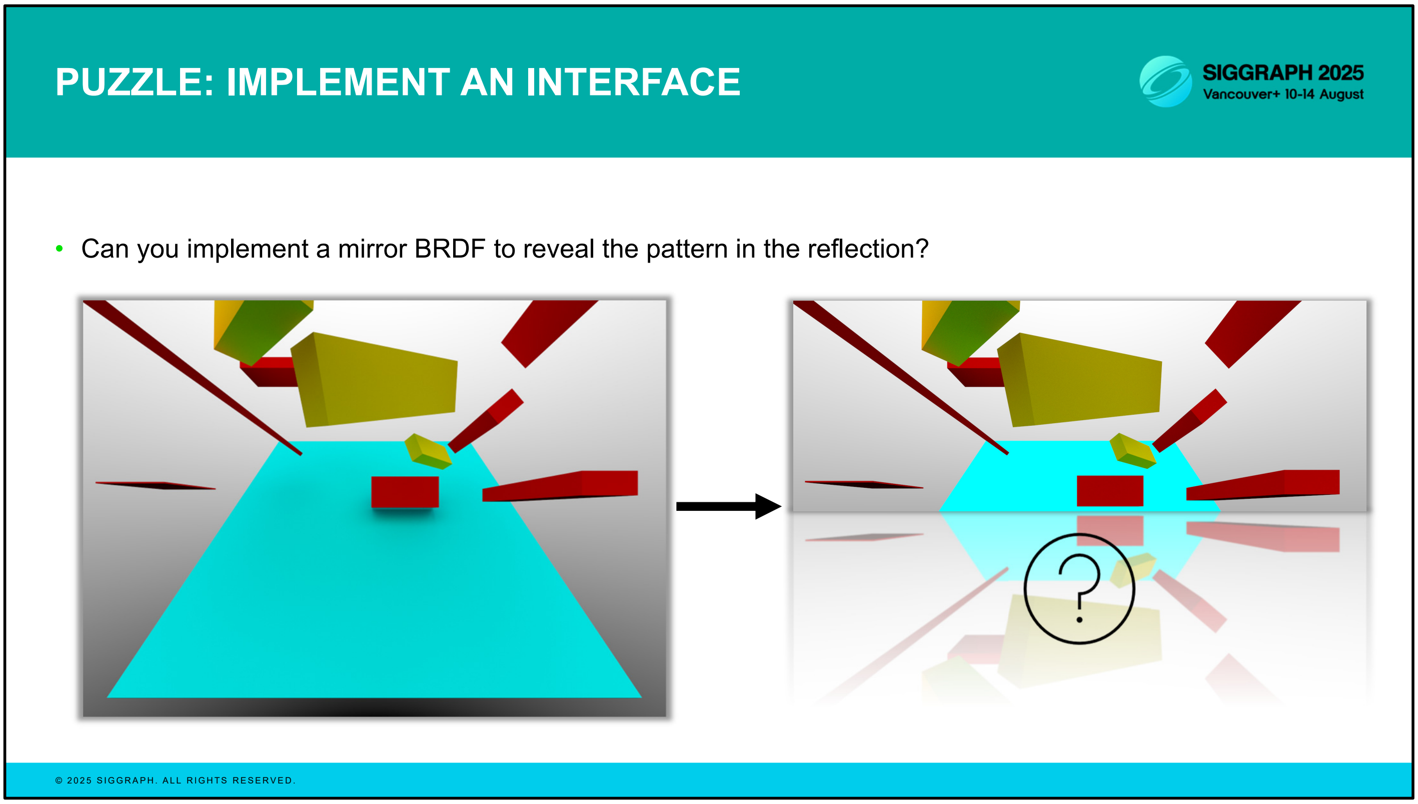
Now we can write a function called lightSum that can take in *any* light type!

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 a result.



PUZZLE: IMPLEMENT AN INTERFACE

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

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

In vk_slang_editor, if you load Examples > !SIGGRAPH2025 > 3 – Interfaces, you’ll see a path-traced scene with several colored boxes, all using a diffuse BRDF.

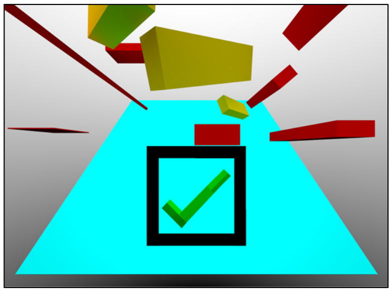
Now, this seemingly random arrangement of boxes in fact contains a hidden image.

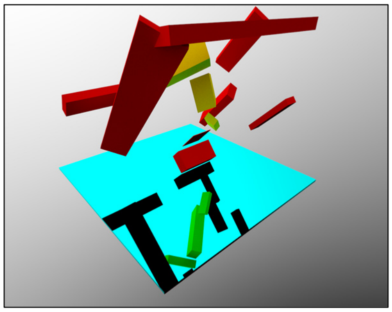
Your challenge is to reveal this hidden image by turning the cyan plane here into a mirror. To do this, you’ll need to remove line 13, and then write a struct called MirrorBRDF that conforms to the IBrdf interface and reflects the input direction along the hit normal.

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

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





SIGGRAPH 2025
Vancouver+ 10-14 August

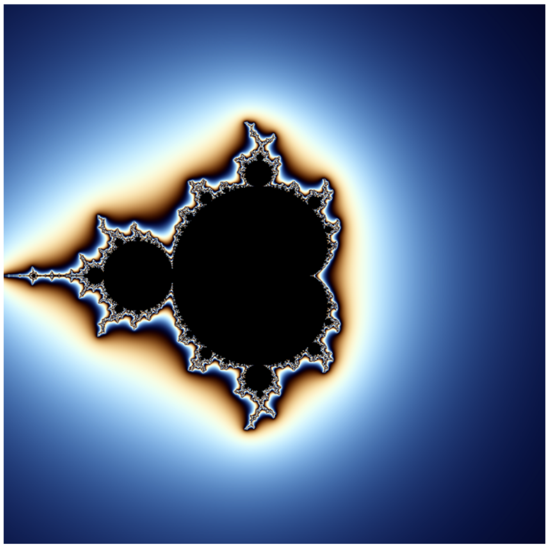
SlangPy

Chris Hebert

FIRST, A DEMONSTRATION

SIGGRAPH 2025
Vancouver+ 10-14 August

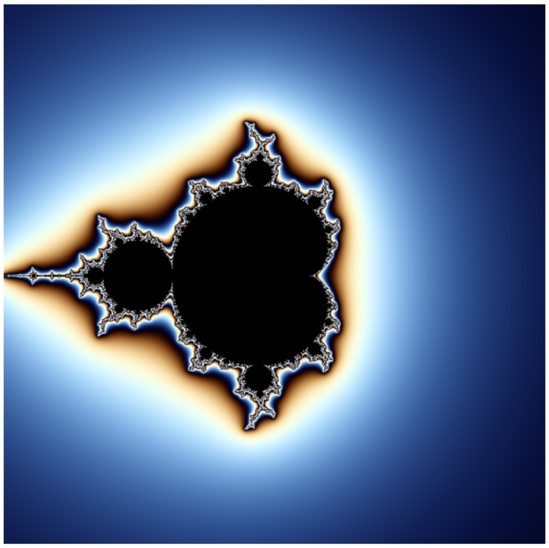
Remember this?



FIRST, A DEMONSTRATION

SIGGRAPH 2025
Vancouver+ 10-14 August

Remember this?



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

TO THE NOTEBOOK!!

IPYTHON NOTEBOOKS IN VS CODE

SIGGRAPH 2025
Vancouver+ 10-14 August

IPython notebooks are a collection of code cells and markdown cells.

They allow you to experiment with and document snippets of Python code.

They integrate seamlessly within VS code.

They can also run in a web browser.

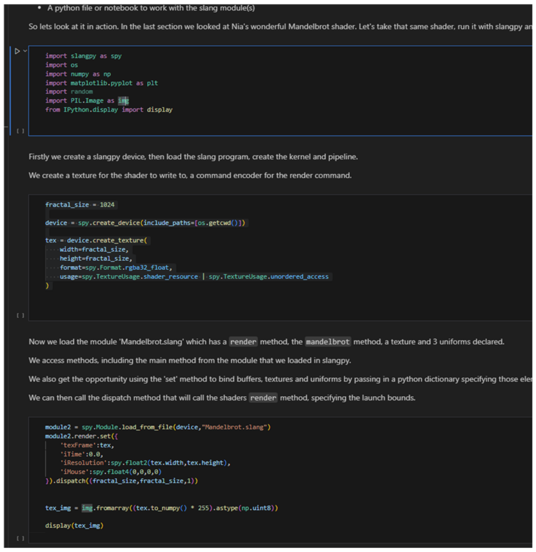
To run a code cell, place your cursor in the cell, click the **play** button in the top left corner.

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 of the cells.



- A python file or notebook to work with the slang module(s)

So lets look at it in action. In the last section we looked at Nia's wonderful Mandelbrot shader. Let's take that same shader, run it with slangpy an

```
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
```

Firstly we create a slangpy device, then load the slang program, create the kernel and pipeline.

We create a texture for the shader to write to, a command encoder for the render command.

```
fractal_size = 1024

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

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

Now we load the module 'Mandelbrot.slang' which has a `render` method, the `mandelbrot` method, a texture and 3 uniforms declared.

We access methods, including the main method from the module that we loaded in slangpy.

We also get the opportunity using the 'set' method to bind buffers, textures and uniforms by passing in a python dictionary specifying those ele

We can then call the dispatch method that will call the shaders `render` method, specifying the launch bounds.

```
module2 = spy.Module.load_from_file(device,"Mandelbrot.slang")
module2.render.set({
    'texFrame':tex,
    'iTime':0.0,
    'iResolution':spy.float2(tex.width,tex.height),
    'iMouse':spy.float4(0,0,0,0)
}).dispatch((fractal_size,fractal_size,1))


tex_img = img.fromarray((tex.to_numpy() * 255).astype(np.uint8))

display(tex_img)
```

Code Cell

Markdown Cell

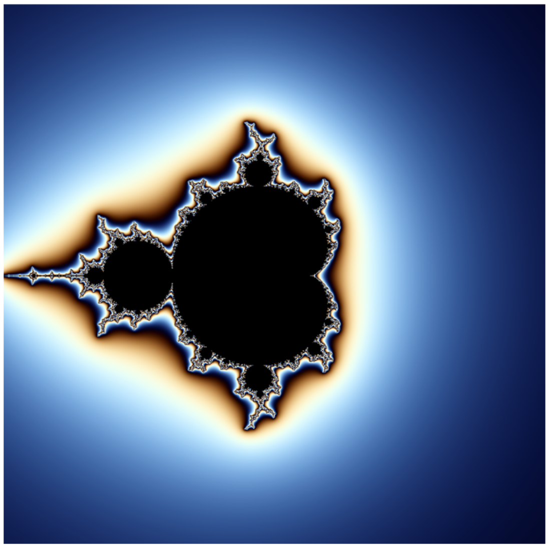
Code Cell

Markdown Cell

Code Cell

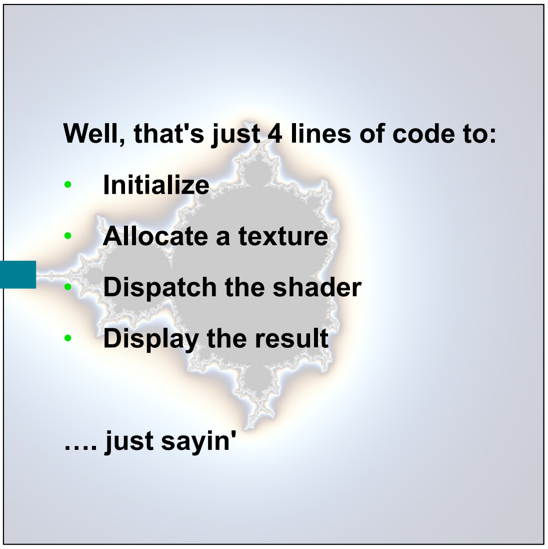
FIRST, A DEMONSTRATION

```
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)))
```



FIRST, A DEMONSTRATION

```
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:

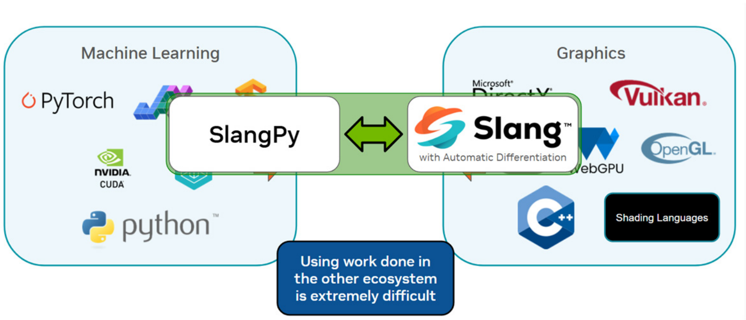
- **Initialize**
- **Allocate a texture**
- **Dispatch the shader**
- **Display the result**

…. just sayin'

WHY SLANGPY?

SIGGRAPH 2025
Vancouver+ 10-14 August

- Provides a very simple path to working with Slang
  - No need to wrestle with 1000s of lines of C++ code to test a 100-line shader
  - Dispatch a Slang shader with as little as 3 lines of code.
  - Makes iterating on prototypes and ideas much simpler
- Helps bridge the gap between Graphics and ML
  - You can use Slang alongside Python's huge ecosystem of libraries
  - e.g., Design and test new operators not currently provided by PyTorch

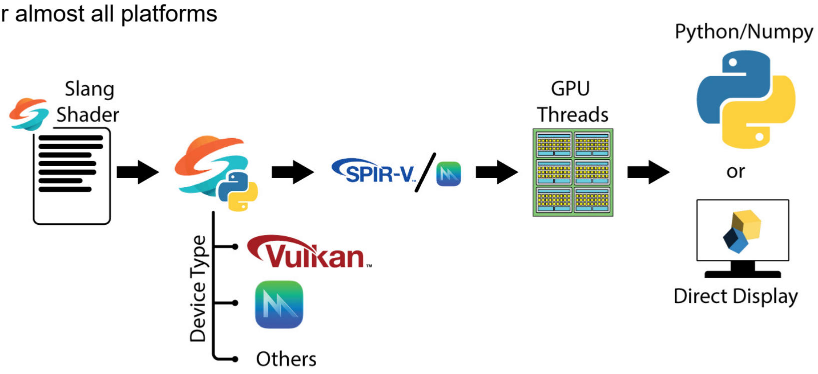


Using work done in the other ecosystem is extremely difficult

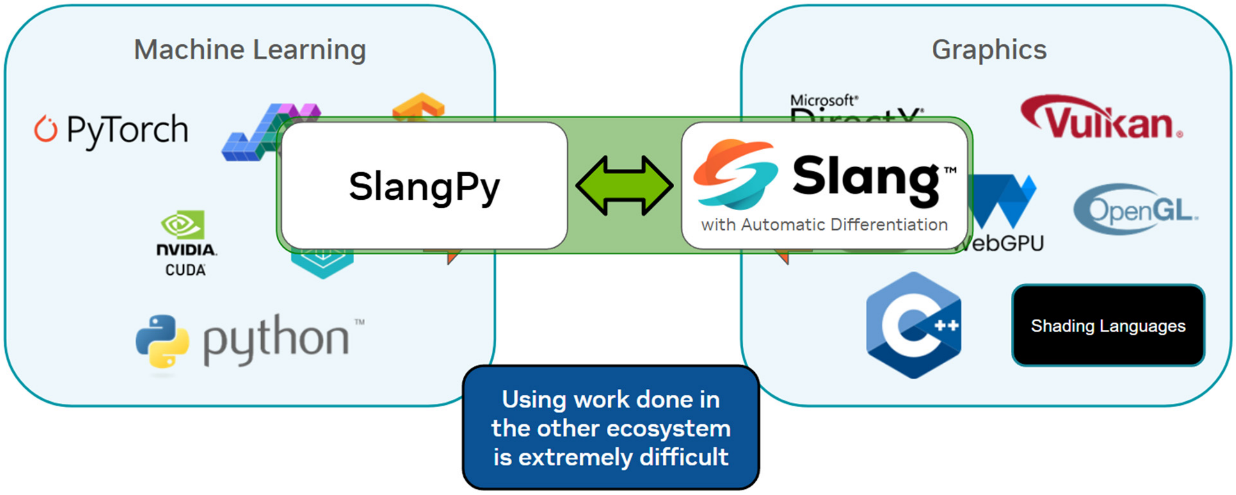
WHY SLANGPY?

SIGGRAPH 2025
Vancouver+ 10-14 August

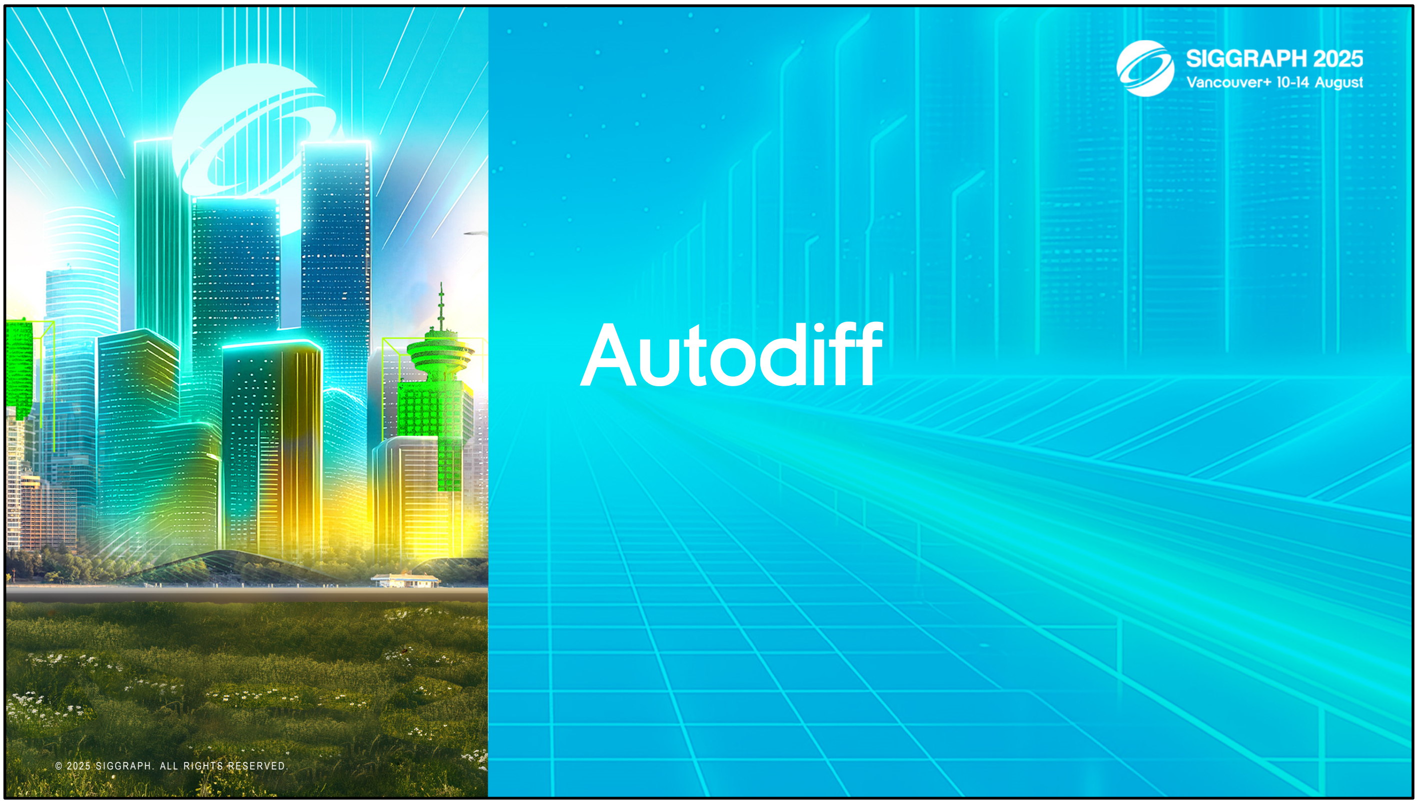
- Abstracts away the details of dispatching threads
  - But it's still there if you want it.
  - Use SlangPy at both a 'high' level and a 'low' level.
- Installs with a single call to Python Pip.
- Supports a large array of backends for almost all platforms
  - Vulkan
  - Metal
  - DirectX
  - WebGPU
  - CUDA



WHY SLANGPY?



Using work done in the other ecosystem is extremely difficult

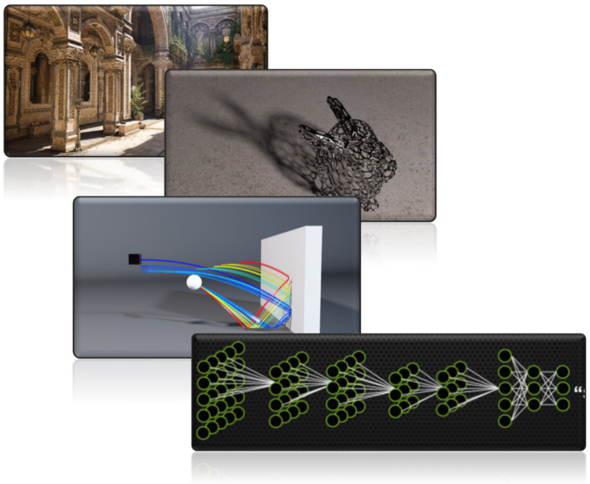


SIGGRAPH 2025
Vancouver+ 10-14 August

Autodiff

AUTODIFF

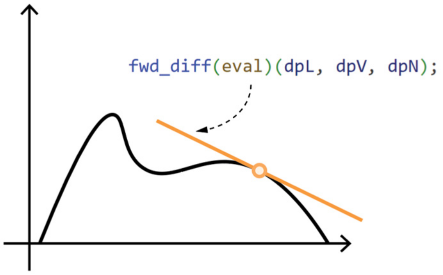
- Neural Rendering
  - Computing gradients for NeRFs (Neural Radiance Fields)
  - Gaussian splatting
- Differentiable Rendering
  - Fitting 3d models to images
  - Optimizing material properties
  - Automatic lighting
- Physics simulation
  - Finite element method derivatives
  - Inverse kinematics & other optimization problems
- Procedural content generation
  - Optimized procedural noise
- AI
  - Training neural networks large or small makes use of derivatives



AUTODIFF

SIGGRAPH 2025
Vancouver+ 10-14 August

- What Is Autodiff?
  - Automatically computes exact derivatives of any function
  - No manual gradient derivation required
  - Supports arbitrary control flow & dynamic dispatch
  - Enables any graphics function to become trainable
- Why Is This Important?
  - You only need to maintain 1 version of your functions
  - Slang generates the differentiable versions for you
  - Less chance of errors because Slang maintains consistency for you



```
fwd_diff(eval)(dpL, dpV, dpN);
```

AUTODIFF

- Forward mode vs Backwards mode

Forward Mode (fwd_diff)

Direction: Input → Output

Efficient when: Few inputs, many outputs

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

Reverse Mode (bwd_diff)

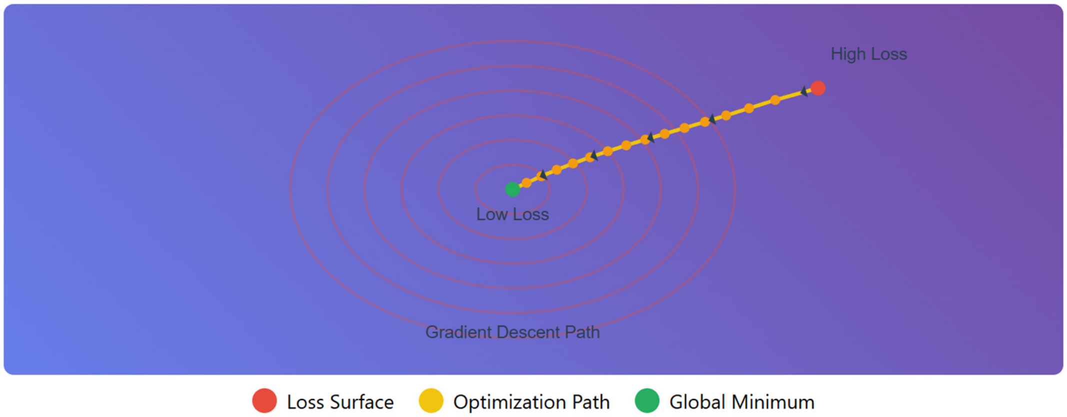
Direction: Output → Input

Efficient when: Many inputs, few outputs

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

AUTODIFF

- EXAMPLE : Gradient Descent



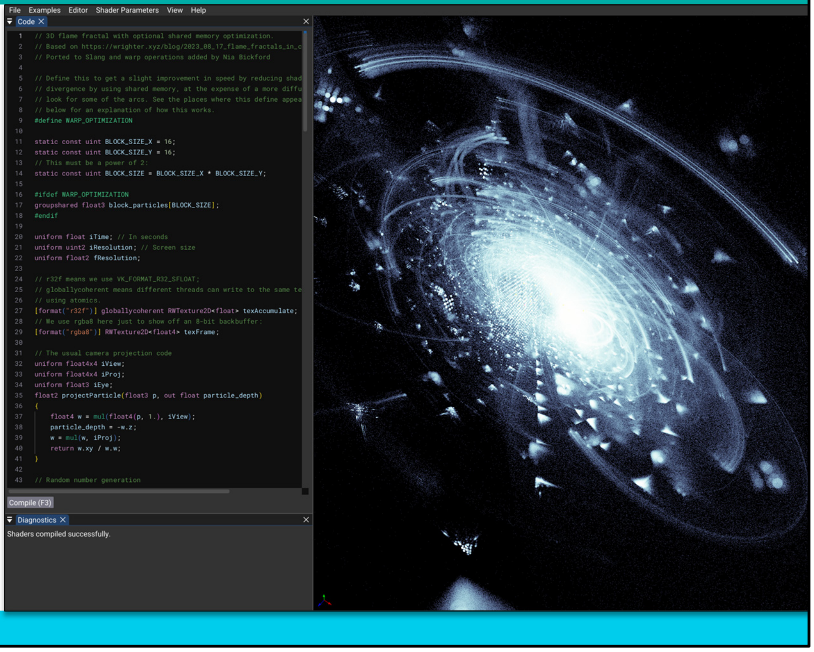
SIGGRAPH 2025
Vancouver+ 10-14 August

Lab: Autodiff

RECAP

- 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
- Autodifferentiation

https://docs.shader-slang.org



```
// 3D flame fractal with optional shared memory optimization.
// Based on https://wrighter.xyz/blog/2023_08_17_flame_fractals_in_c
// Ported to Slang and warp operations added by Nia Bickford

// Define this to get a slight improvement in speed by reducing shad
// divergence by using shared memory, at the expense of a more diffu
// look for some of the arcs. See the places where this define appea
// below for an explanation of how this works.
#define WARP_OPTIMIZATION

static const uint BLOCK_SIZE_X = 16;
static const uint BLOCK_SIZE_Y = 16;
// This must be a power of 2:
static const uint BLOCK_SIZE = BLOCK_SIZE_X * BLOCK_SIZE_Y;

#ifdef WARP_OPTIMIZATION
groupshared float3 block_particles[BLOCK_SIZE];
#endif

uniform float iTime; // In seconds
uniform uint2 iResolution; // Screen size
uniform float2 fResolution;

// r32f means we use VK_FORMAT_R32_SFLOAT;
// globallycoherent means different threads can write to the same te
// using atomics.
[format("r32f")] globallycoherent RWTexture2D<float> texAccumulate;
// We use rgba8 here just to show off an 8-bit backbuffer:
[format("rgba8")] RWTexture2D<float4> texFrame;

// The usual camera projection code
uniform float4x4 iView;
uniform float4x4 iProj;
uniform float3 iEye;
float2 projectParticle(float3 p, out float particle_depth)
{
    float4 w = mul(float4(p, 1.), iView);
    particle_depth = -w.z;
    w = mul(w, iProj);
    return w.xy / w.w;
}

// Random number generation
```

Shaders compiled successfully.

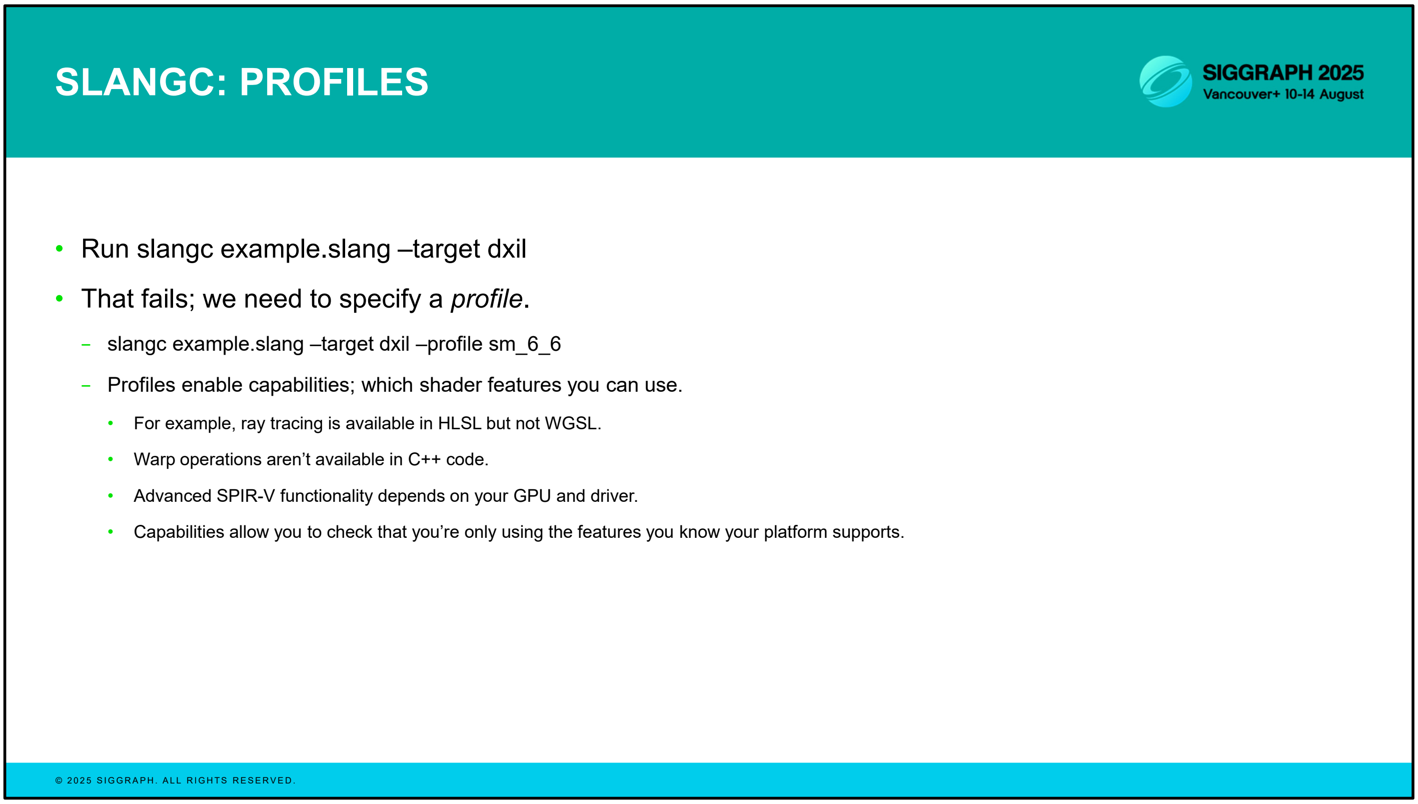
THANK YOU!

- Lab materials: https://shader-slang.org/landing/siggraph-25
- Related sessions:
  - **Nsight Graphics Gaussian Splatting Sessions**
    - Wednesday, 9am – 10:30am & 1pm – 2:55pm, Room 116-117
  - **Birds of a Feather: Developing with Slang: Tools, Techniques, and Future Directions**
    - Wednesday, 2:30pm – 3:30pm, Vancouver Marriot Pinnacle Hotel
  - **An Introduction to Neural Shading**
    - Thursday, 9am – 12:15pm, West Building, Room 109-110
  - **Hands-on Vulkan® Ray Tracing With Dynamic Rendering**
    - Thursday, 11:45am – 1:15pm, West Building, Exhibit Hall B

https://shader-slang.org/landing/siggraph-25

SIGGRAPH
2025

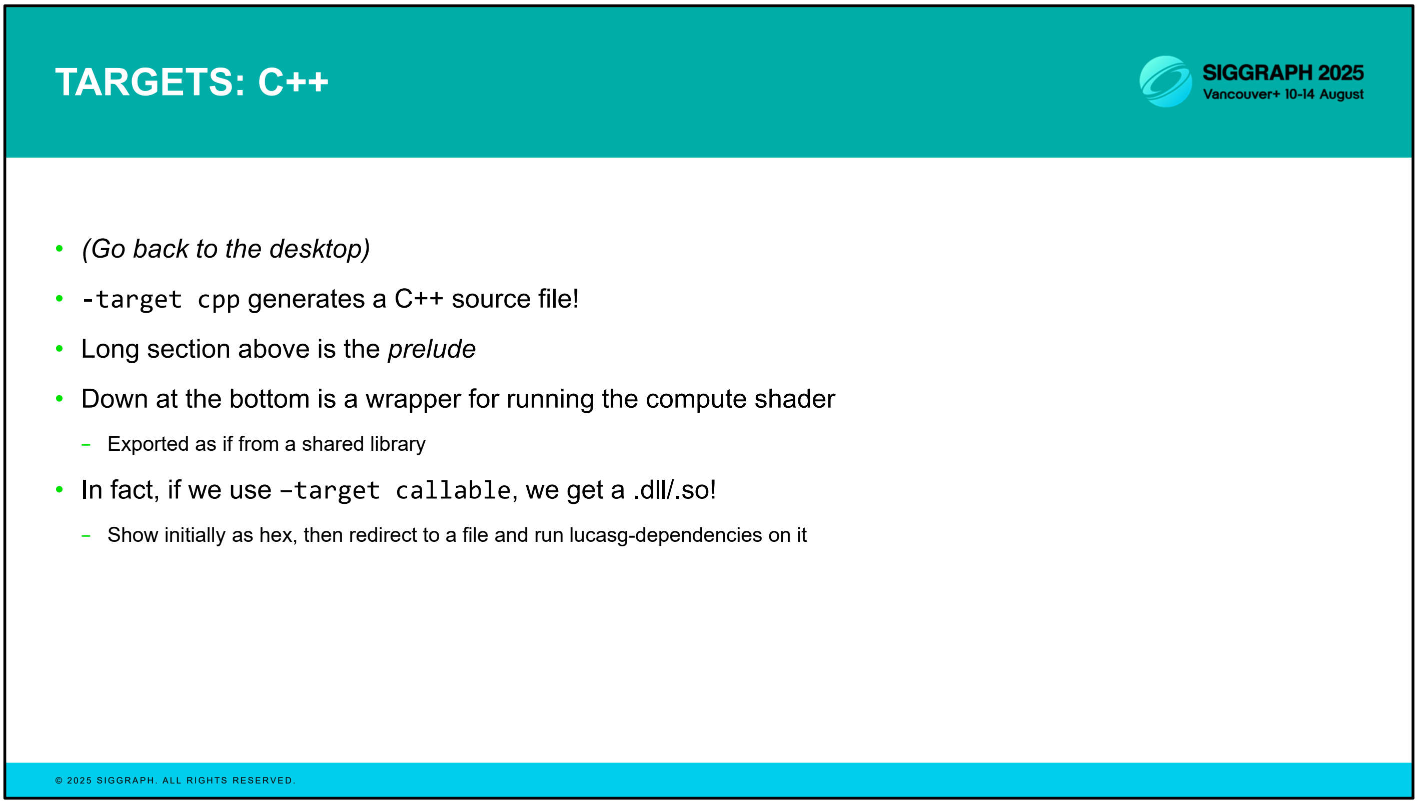
Backup Slides



SLANGC: PROFILES

- 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.

Incidentally, this is what allows Slang to avoid falling into the lowest-common-denominator trap like GLES. Slang can support the latest graphics features, and then capabilities restrict it to those the system can use.



TARGETS: C++

SIGGRAPH 2025
Vancouver+ 10-14 August

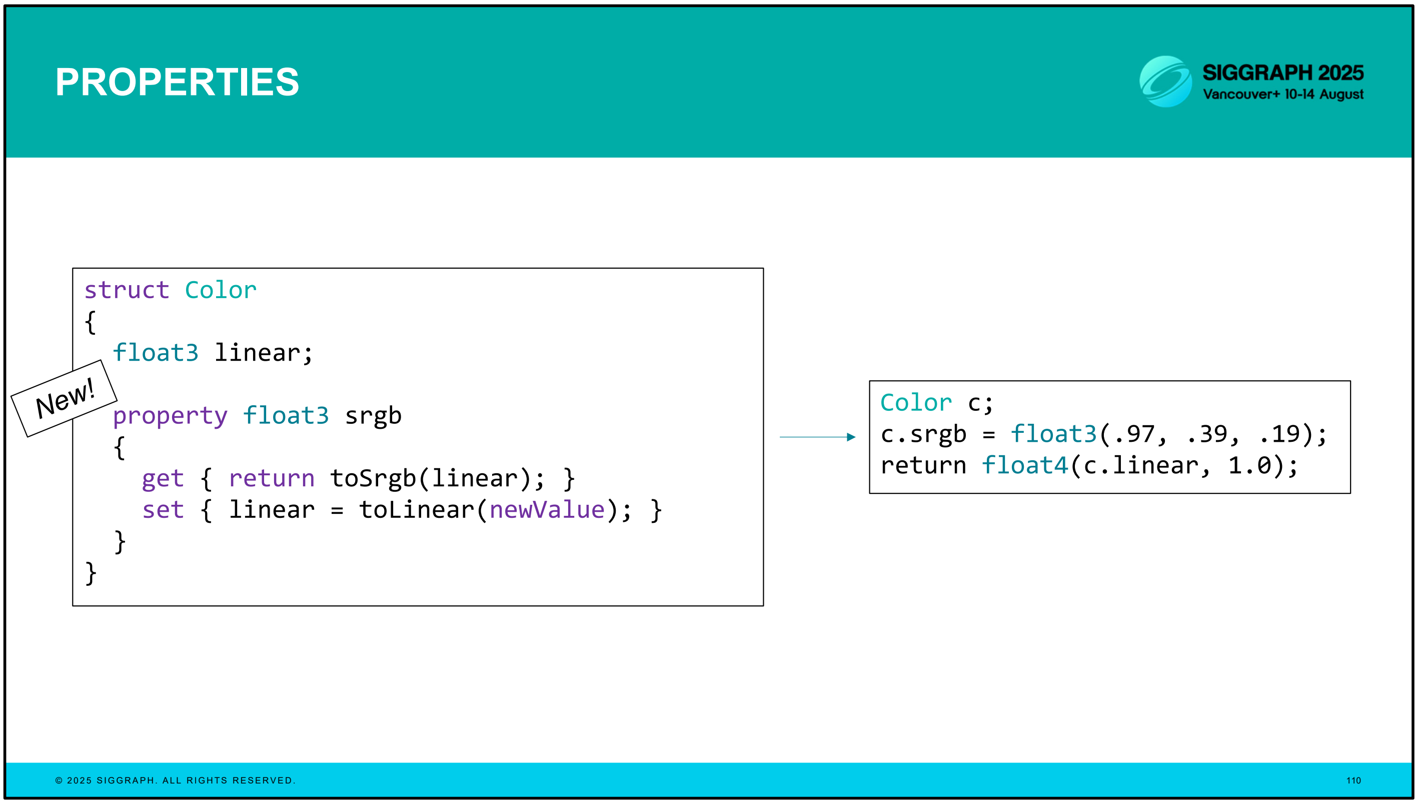
- *(Go back to the desktop)*
- `-target cpp` generates a C++ source file!
- Long section above is the *prelude*
- Down at the bottom is a wrapper for running the compute shader
  - Exported as if from a shared library
- In fact, if we use `–target callable`, we get a .dll/.so!
  - Show initially as hex, then redirect to a file and run lucasg-dependencies on it

At this point, it’s easiest to go back to the desktop to show it off.

“C++ is probably the most interesting out of those 8 targets – it generates a C++ source file, so you can run your compute shader on the CPU!”

…

“The intent here is that you’d dynamically load this .dll using your OS’ API, then call the entrypoint function and pass an appropriate pointer to the parameter struct.”



PROPERTIES

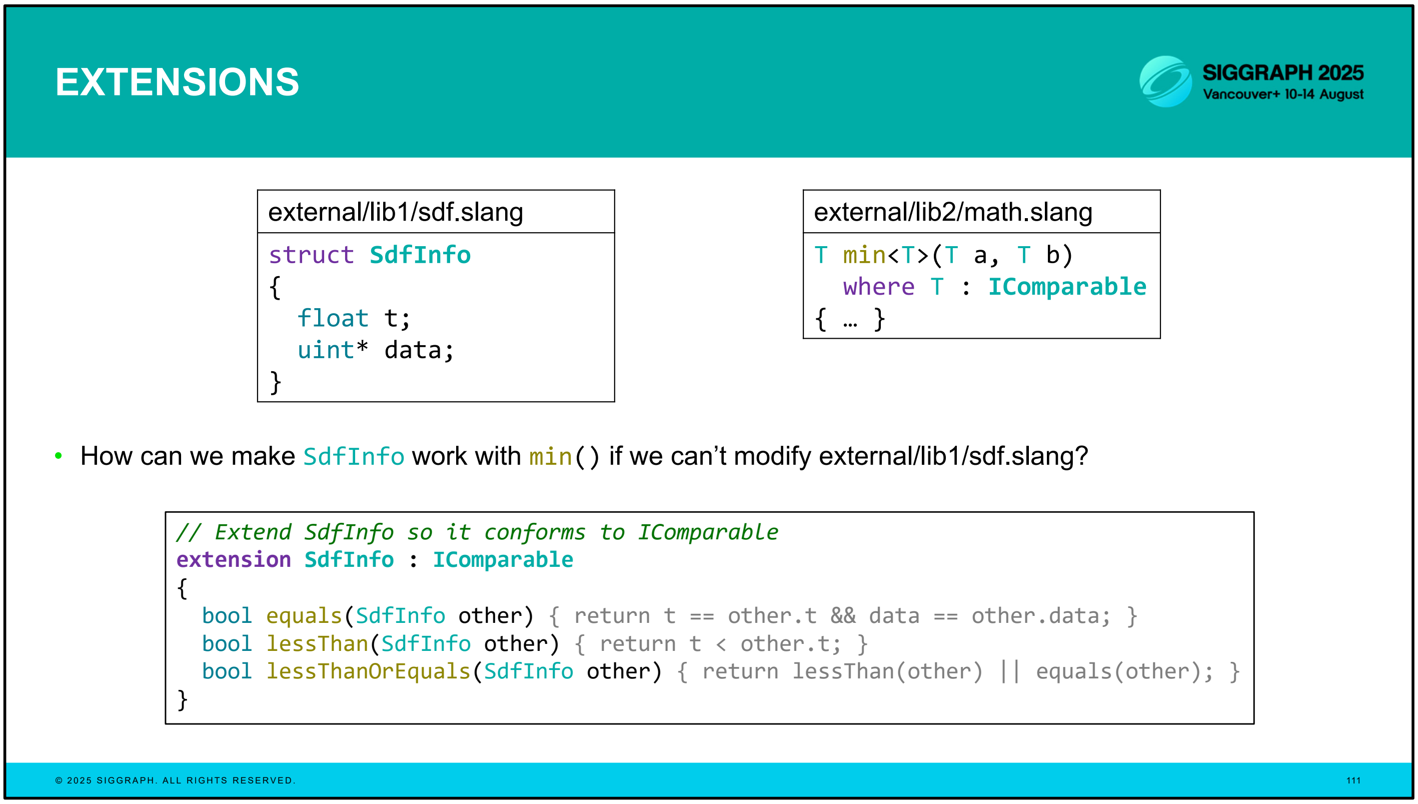
SIGGRAPH 2025
Vancouver+ 10-14 August

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.



EXTENSIONS

SIGGRAPH 2025
Vancouver+ 10-14 August

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?

```
// 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!