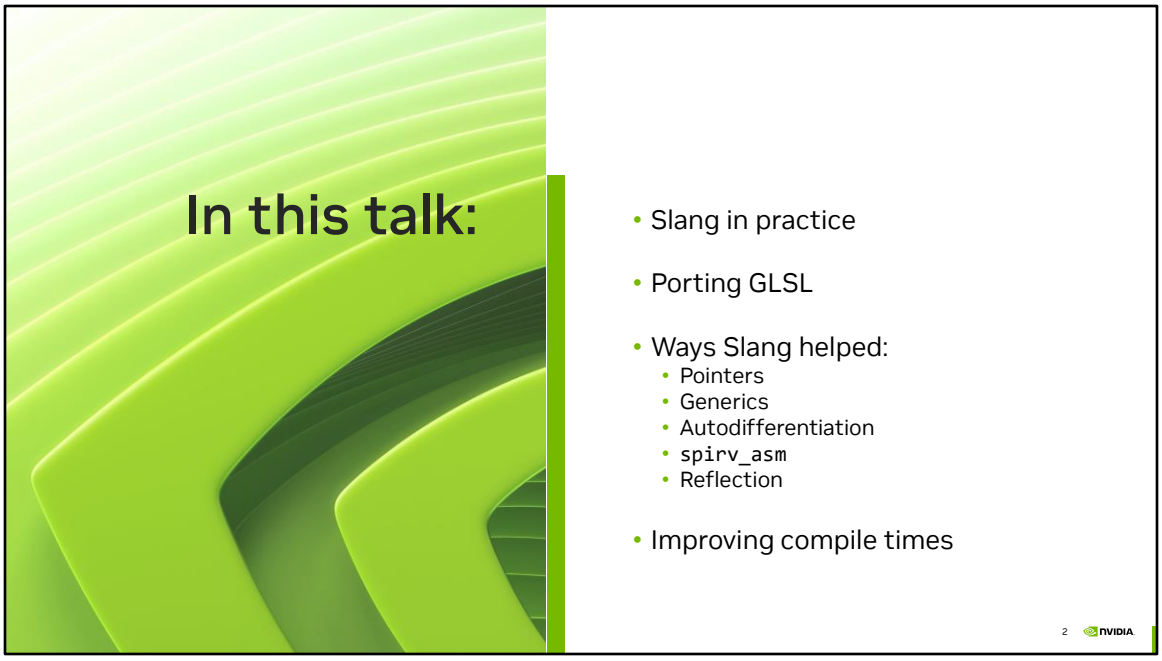
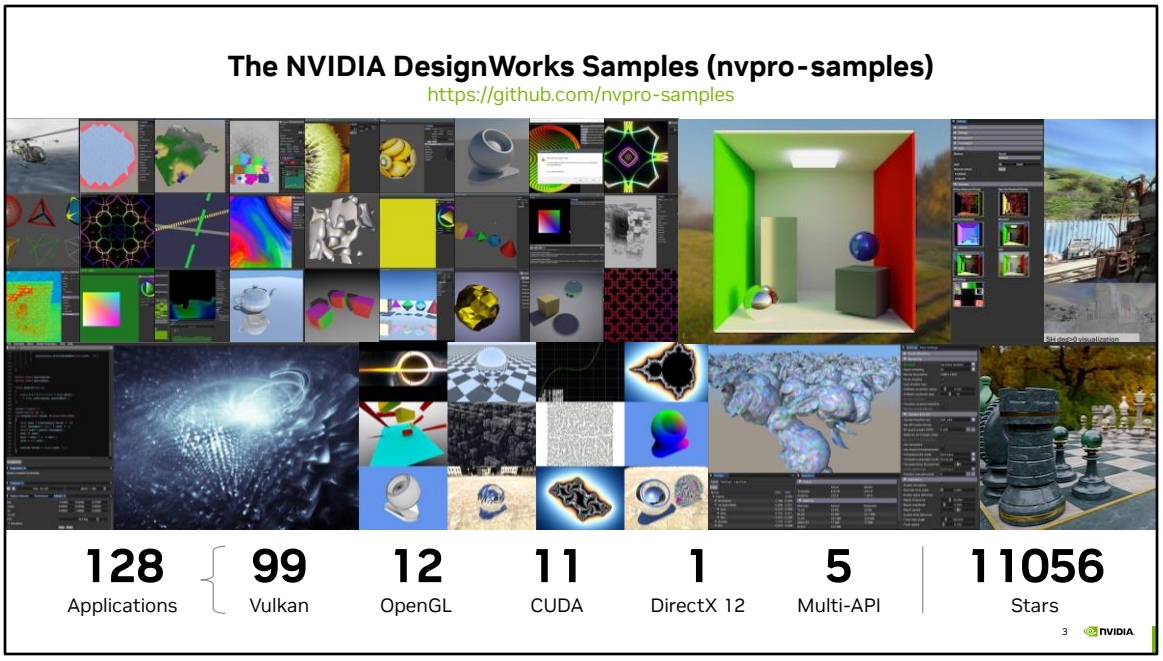
Hi! My team has ported dozens of applications to use the Slang shading language.



While Shannon talked about Slang from the compiler side, I'm hoping to talk about Slang in practice – what Slang looks like with real codebases.

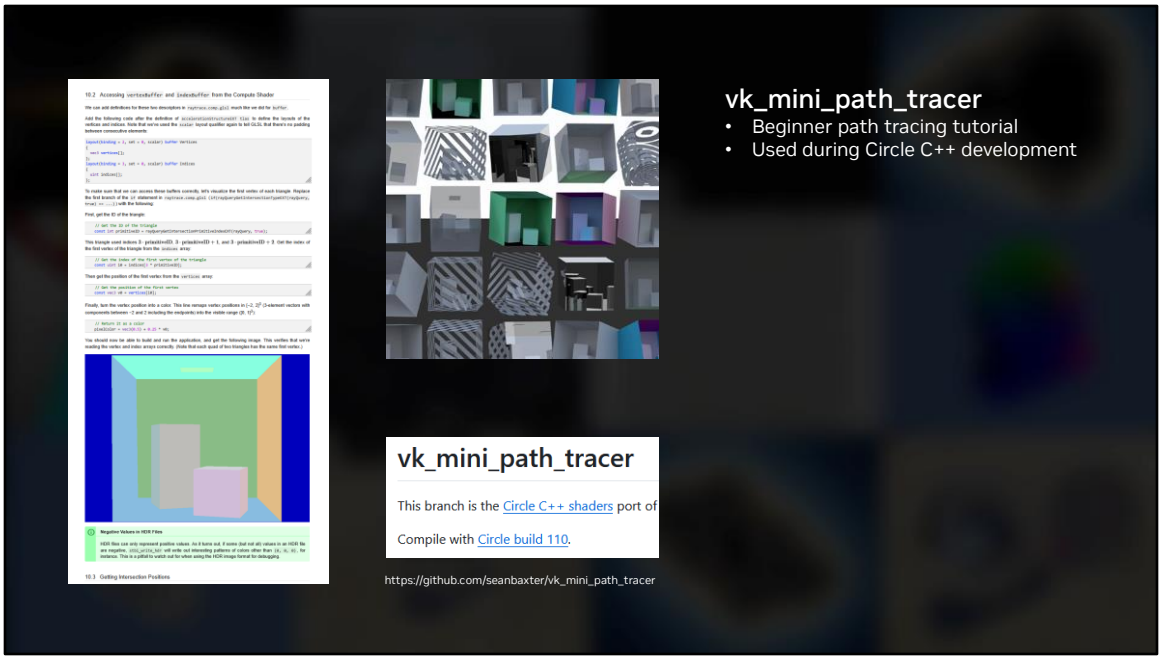
(click) I'll talk about some neat benefits we get out of it – like pointers, generics, autodifferentiation, SPIR-V assembly, and reflection;
(click) provide tips for efficiently porting and compiling code; and hopefully provide useful experience and ideas to engineers building other shading languages.



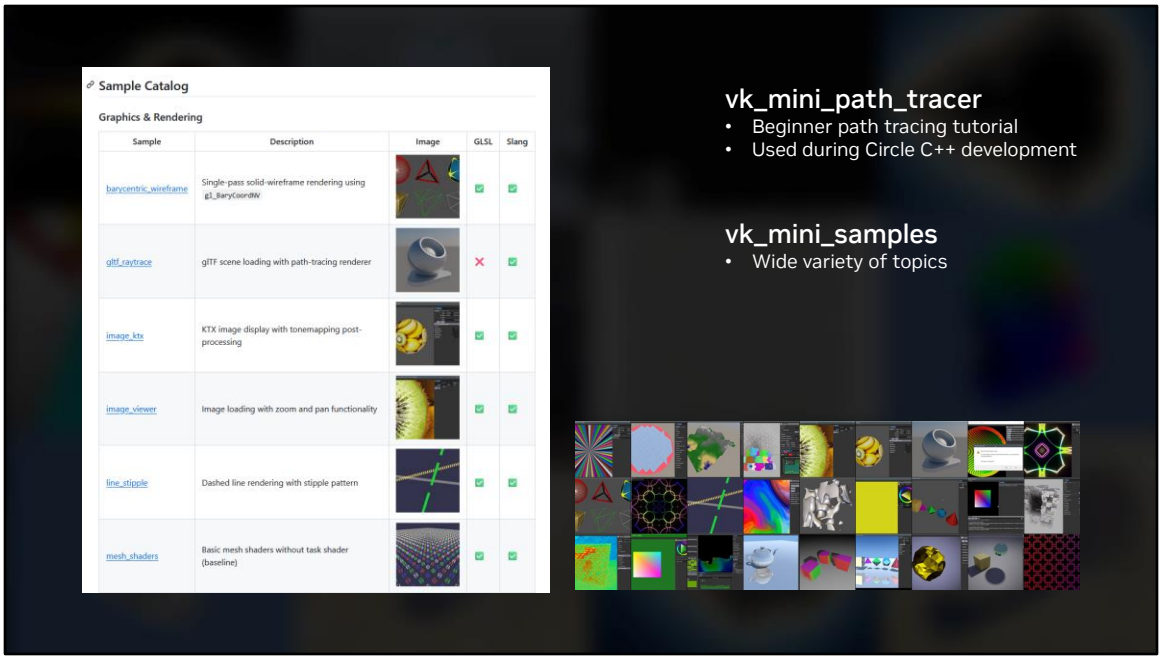
At work, my team develops the NVIDIA DesignWorks Samples. Over the past decade, we've built over 100 open-source applications covering all sorts of computer graphics techniques: from ray tracing to Gaussian splatting, texture compression, denoising, video encoding, Mega Geometry, and more. They're written for a variety of APIs; most are in Vulkan, but we've got samples for OpenGL, CUDA, and DirectX as well.

Our samples have been used by developers of all skill levels, all over the world, for all sorts of purposes. Together, our samples have received over 11,000 GitHub Stars.

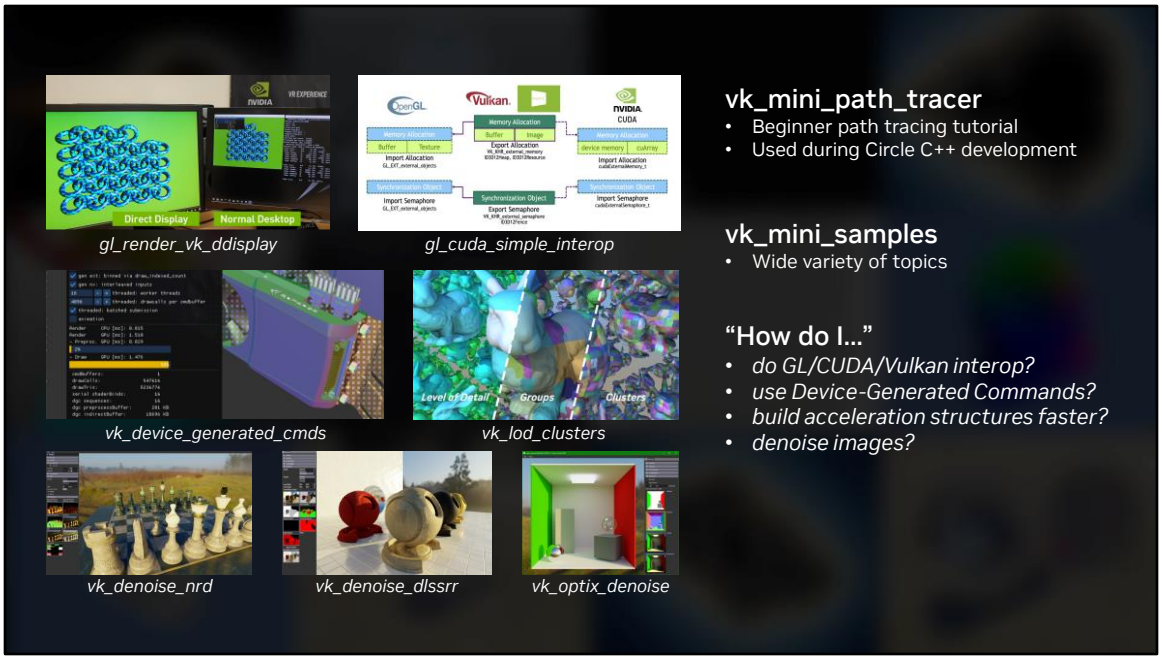
(Figure from executables produced by build_all's INSTALL target, plus 1 for vk_video_samples, 1 for vk_minimal_latest, and 1 for vk_gaussian_splatting)



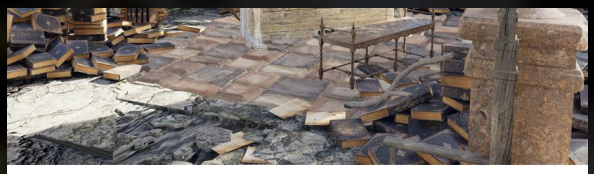
For beginners, we have tutorials like vk_mini_path_tracer, which attempts to teach Vulkan path tracing without requiring prior graphics knowledge. (click) But it was also used by Sean Baxter as a test application for the Circle C++ shader compiler.



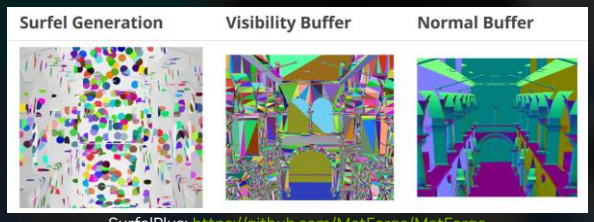
Many of our readers are computer graphics professionals, and for them we have samples on a wide variety of topics, like the 29 mini-samples written by Martin-Karl Lefrancois.



Sometimes a developer will come to us with a problem – like “how do I do interop between GL and Vulkan?” – we’ll write sample code showing how to do that, and then release it so people can learn from it.



MatForge: https://github.com/MatForge/MatForge
Forked from **vk_gltf_renderer**



SurfelPlus: https://github.com/MatForge/MatForge
Forked from **vk_raytrace**

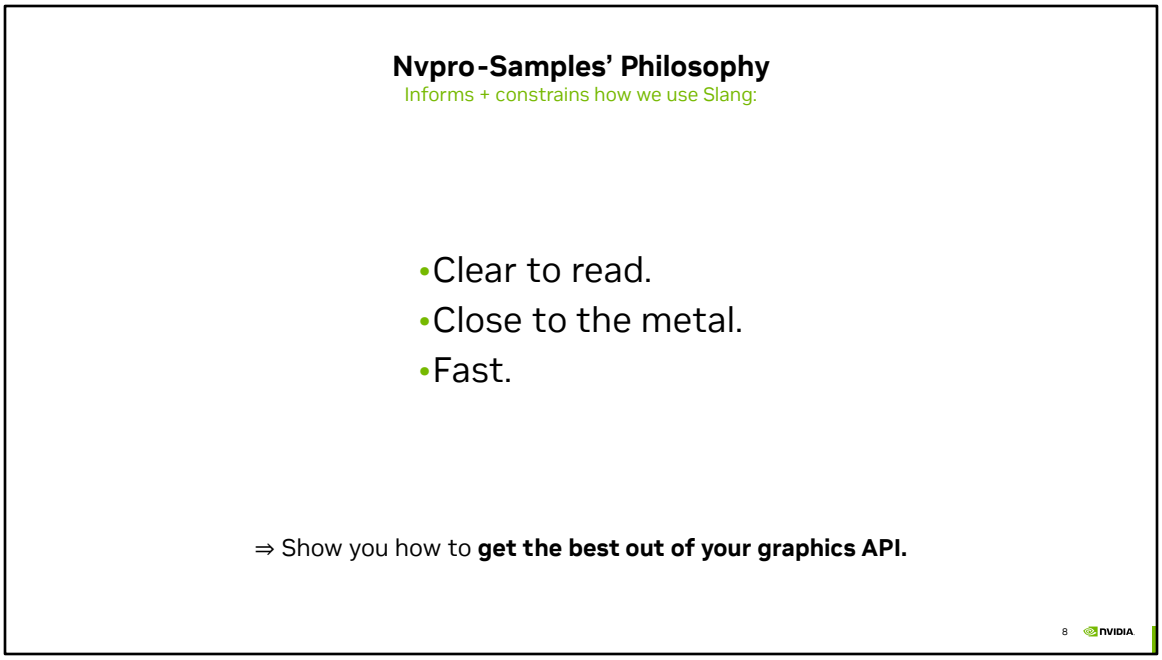
vk_mini_path_tracer
- Beginner path tracing tutorial
- Used during Circle C++ development

vk_mini_samples
- Wide variety of topics

"How do I..."
- *do GL/CUDA/Vulkan interop?*
- *use Device-Generated Commands?*
- *build acceleration structures faster?*
- *denoise images?*

Used for prototyping:
- MatForge
- SurfelPlus

And our samples are easy to modify as well; university students have used them to build projects like MatForge and SurfelPlus.



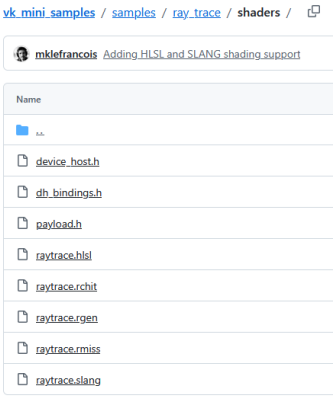
All our samples have a sort of unifying philosophy, which informs and constrains how we use Slang.

- First, they should be *clear to read*.
- Second, they should be *close to the metal*. In the parts that matter to each sample, you should be seeing raw Vulkan code. No deep abstractions you have to dig through; no engines, no RHIs; what you see is what it does.
    - For things that aren't important to the sample – such as swapchain creation in an SSAO sample – we're OK with abstractions, but they should still be easy to peel away.
- Most importantly, because we're a high-performance graphics team, it needs to be *fast*. For this, Slang's performance – as good as GLSL or better – is really important.

So, in other words, our goal **(click)** is to show you how to *get the best* out of the graphics API you're using.

Why did we choose Slang?

- 2023-2024:
  - Nia experimented with Slang's autodiff for texture compression
  - Martin-Karl ported vk_mini_samples from **GLSL** to **Slang** and **HLSL**



- Compared pros and cons of all 3 languages
  - We want a shading language that works + developers like
  - At the time, Slang was a research project

- Slang had these killer features:

9 NVIDIA

We started looking into using Slang in 2023-2024. I had been using Slang's automatic differentiation feature to do some GPU-accelerated texture compression research in 2023. Later that year Martin-Karl really kicked it off by porting his vk_mini_samples from GLSL to Slang and HLSL, to compare the pros and cons of all 3 languages.

(click) Now, we do work at NVIDIA, but we *do* want to choose a shading language that worked for us, and that developers will like. At the time, Slang was a research project, after all; it hadn't been adopted by Khronos yet. **(click)** But it had a few features that were really appealing for us.

NVIDIA

Pointers!

GLSL: gltf_pathtrace.comp.glsl + nvpro_core

```
struct RenderNode { … };

struct Scene
{
  uint64_t materialAddress;
  uint64_t renderNodeAddress;
  uint64_t renderPrimitiveAddress;
  uint64_t lightAddress;
  int      numLights;
};

layout(buffer_reference, scalar)
readonly buffer RenderNodeBuf {
  RenderNode _[];
};

// Get RenderNode i
RenderNode renderNode
  = RenderNodeBuf(renderNodeAddress)._[i]
```

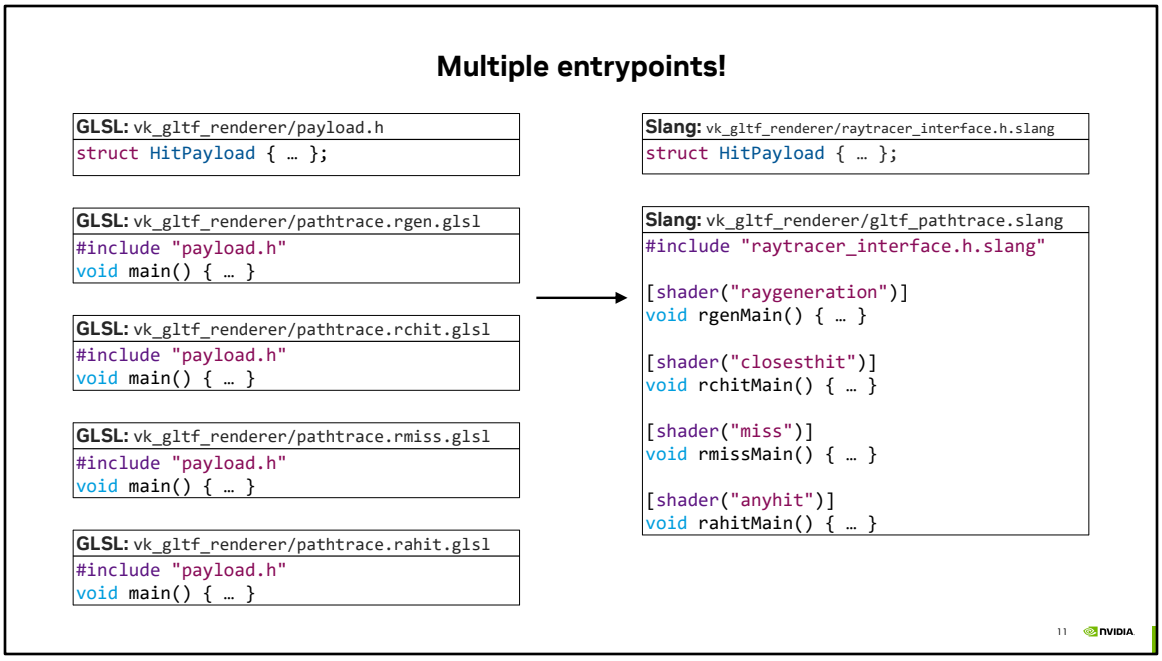

Slang: gltf_pathtrace.slang + nvpro_core2

```
struct RenderNode { … };

struct Scene
{
  ShadeMaterial*   materials;
  RenderNode*      renderNodes;
  RenderPrimitive* renderPrimitives;
  Light*           lights;
  int              numLights;
};

// Get RenderNode i
RenderNode renderNode = renderNodes[i];
```

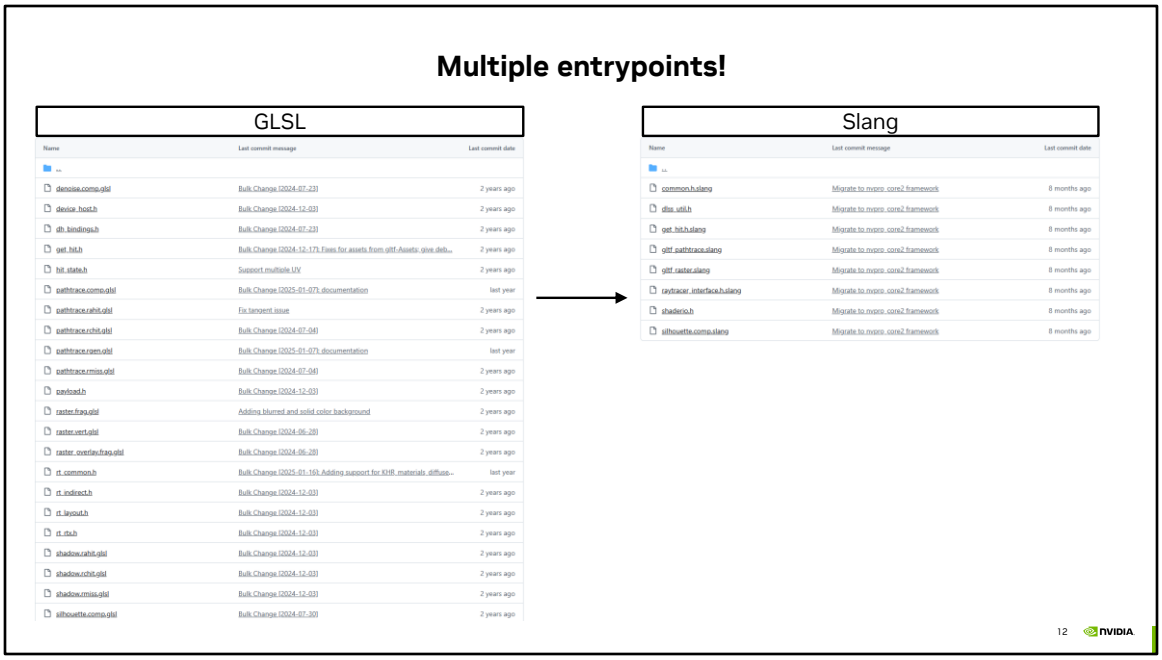
10 NVIDIA

The main one was **pointers**. We use GLSL buffer references quite heavily for complex data structures in our glTF and RTX Mega Geometry renderers. But the syntax for pointers is much nicer – and it maps well to what the hardware's doing, which is a load from an address in VRAM.

NVIDIA



Then there's the ability to define **multiple entrypoints in a single file**. In our ray tracing pipeline examples, we often needed to create different GLSL files for ray generation, closest-hit, miss, and sometimes any-hit and intersection shaders. These shaders all need to share the same ray payload, so it's much nicer to define them in a single file instead of splitting them across many files with correspondingly complex includes.

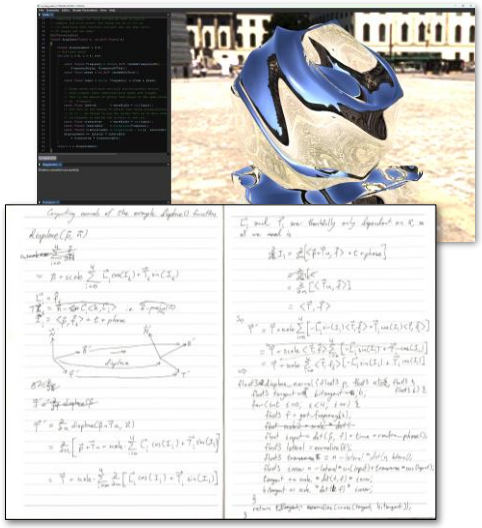


This significantly reduces the visual complexity of our shader folders. Here are the shaders for vk_gltf_renderer, a fully-featured glTF path tracer, before and after it was ported to Slang.

NVIDIA

Autodifferentiation!

```
float3 displace(float3 p, float3 n) {
  return <complex noise function of p and n>
}

[shader("vertex")]
VsOutput vertexMain(Vertex v) {
  float3 pos = displace(v.pos, v.normal);
  …
}

[shader("fragment")]
float4 fragmentMain(VsOutput v) {
  // Compute normal manually
  // Don't change `displace()` without updating
  // this code!
  float3 normal
    = <lots of complex manual derivative code>
  …


}
```



13 NVIDIA

There's also **autodifferentiation**.

My usual example for this is procedural displacement. Suppose I have a shader that displaces vertices, like a Perlin noise function, a wave function, or vertex
skinning. Then I also need to compute a new normal vector in the fragment shader for correct shading, which I can do by computing directional derivatives of the displacement function.

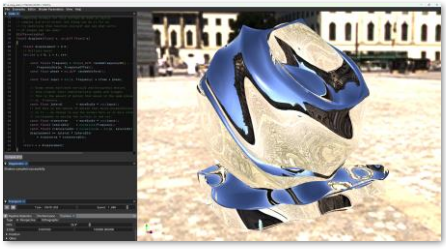
(click) I've written code to compute these analytical derivatives by hand many times before, but it always takes quite a bit of algebraic manipulation, it's easy to get wrong, is another code path to test, and is also hard to read. If I need backwards derivatives for optimization, it's another code path.

In addition, if an artist comes to me and asks if we can change the displacement function, then depending on how I've structured the derivative code I might have to re-derive and re-test a lot of it. This slows down iteration times, and weakens the creative process.

NVIDIA

Autodifferentiation!

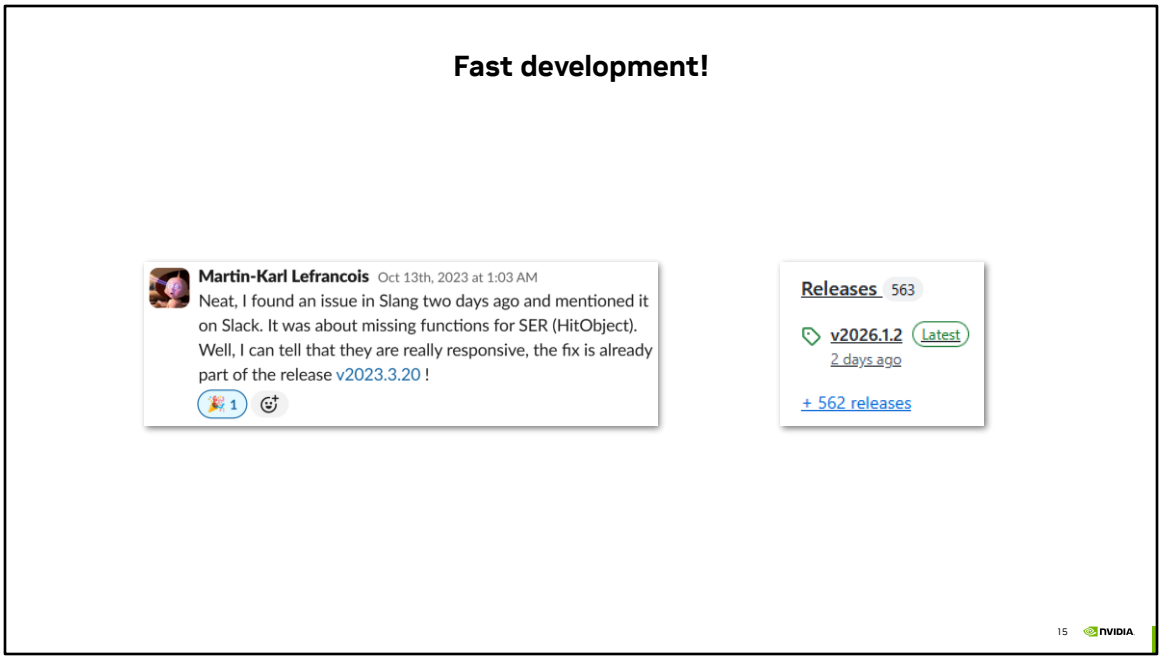
```
[Differentiable]
float3 displace(float3 p, float3 n) {
  return <complex noise function of p and n>
}

[shader("vertex")]
VsOutput vertexMain(Vertex v) {
  float3 pos = displace(v.pos, v.normal);
  …
}

[shader("fragment")]
float4 fragmentMain(VsOutput v) {
  // Compute displaced tangent and bitangent
  float3 T = fwd_diff(displace)(
    diffPair(v.pos, v.tangent), v.normal).d;

  float3 B = fwd_diff(displace)(
    diffPair(v.pos, v.bitangent), v.normal).d;

  float3 normal = normalize(cross(T, B)); // Done!
  …
}
```



14 NVIDIA

With Slang, I can mark a function as [Differentiable], and then it'll automatically generate code to compute analytical derivatives for me! Then if I change the displacement function, the normals update automatically.

NVIDIA



Finally, Slang has **fast feature development and response times**. Since we're developing samples for new technologies, we often need support for cutting-edge APIs. Often when Martin-Karl or I found a bug or had a missing feature, it'd be fixed or added in a few days, which is really remarkable for a compiler.

So, based on Martin-Karl's research and experience, we decided to port our samples to Slang

Porting GLSL to Slang, Without Much Modification

- 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

The good news is Slang makes porting code from GLSL to Slang really easy! You only have to do a few small things.

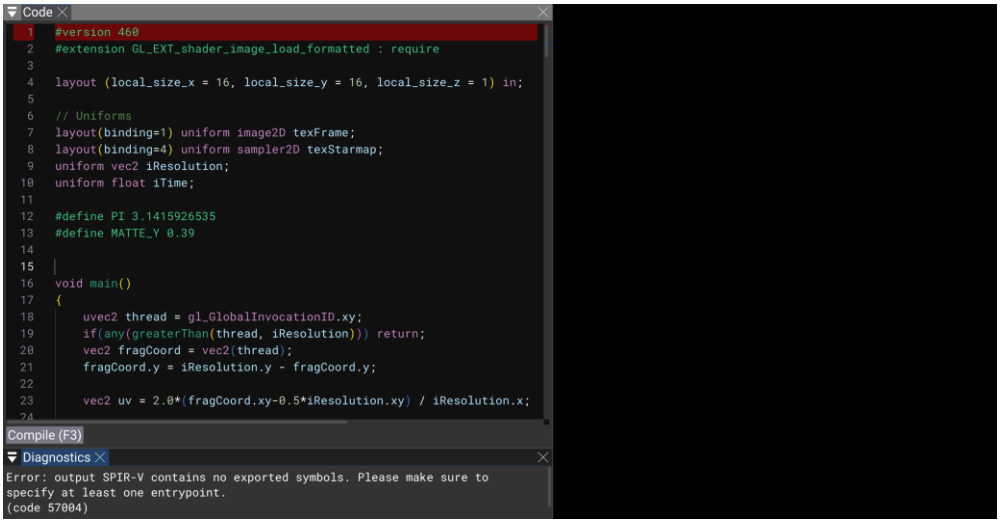
(click) First off, make sure your shader uses the GLSL #version directive somewhere. This tells Slang to load in its GLSL compatibility module, which defines type aliases and functions to map GLSL constructs to Slang.

(click) 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 **(click)** domain, hull, or mesh shader, which is less common, you have to translate those specific GLSL attributes to HLSL equivalents. It's usually straightforward.

(click) And then you're done!

GLSL



As an example, here's a GLSL shader based off a Shadertoy I wrote back in college. It already has #version 460 at the top, so all that's needed is to add [shader("compute")] to its main() function.

Slang

Code

```
#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=4) uniform sampler2D texStarmap;
uniform vec2 iResolution;
uniform float iTime;

#define PI 3.1415926535
#define MATTE_Y 0.39

[shader("compute")]
void main()
{
    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;
```

Compile (F3)

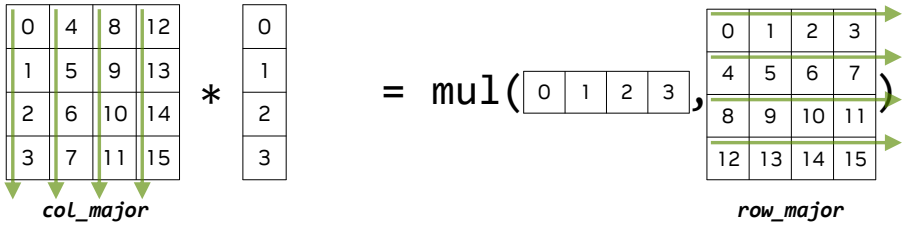
Diagnostics

Shaders compiled successfully.

Once we've done that, slangc successfully compiles it.

Matrix Layouts in GLSL vs. HLSL

GLSL	HLSL
`M * v`	`mul(v, M)`



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

19 NVIDIA

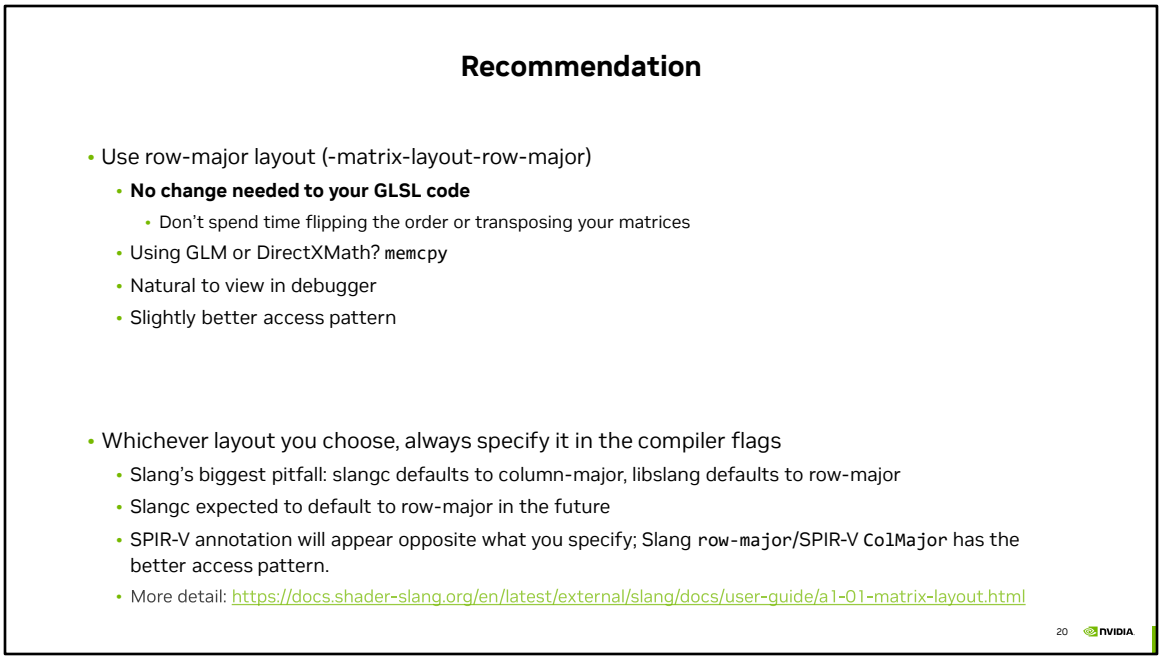
One potentially confusing thing you may know is that GLSL and HLSL typically have opposite matrix layouts and conventions for the order in which you multiply vectors and matrices.

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).

NVIDIA



However, uploading matrix data can be a bit of a braintwister. If you're looking at SPIR-V it's even more so because Slang also emits that transposed. My team went back and forth on different approaches when we were porting our first samples.

Based on our experience, we have a recommendation: **(click)** 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.

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

(click) And row-major CPU matrices are natural to view in the debugger, **(click)** and have slightly better access patterns for the GPU.

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

(click) The second most important thing is to always specify the matrix layout in the compiler flags. 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.

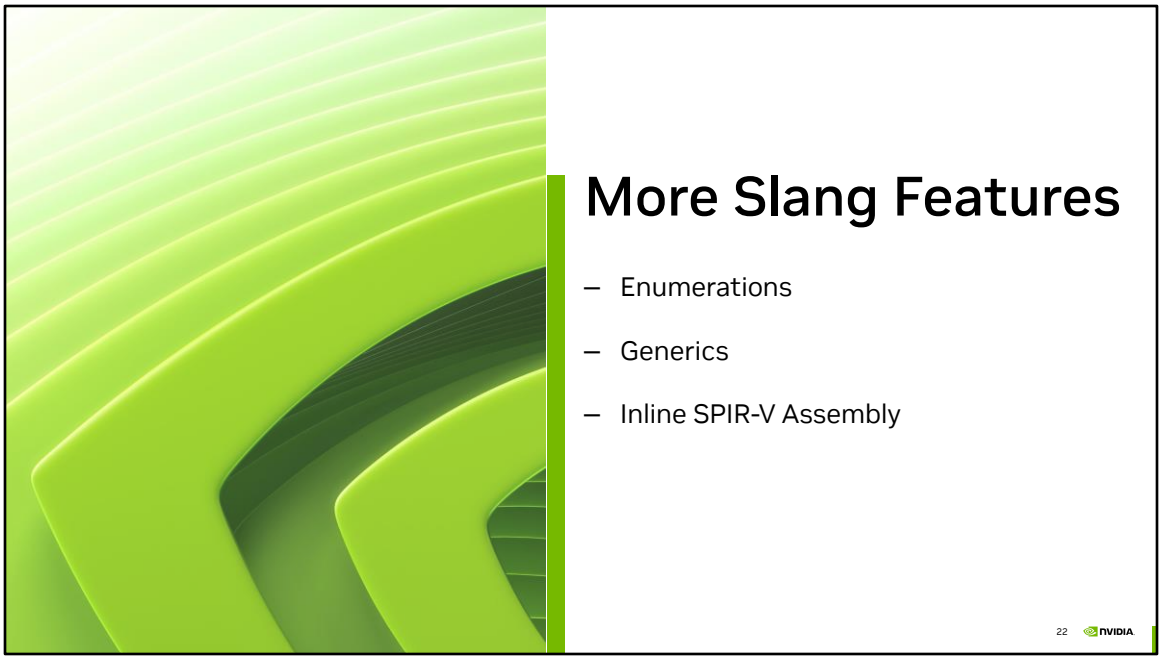
Porting GLSL to Slang, Without Much Modification

- 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`

- For our samples, we chose the hard route of manually porting code to idiomatic Slang instead!
  - Want to show how to **get the best out of Slang**, not GLSL-over-Slang

And that's it! This is the way I'd recommend people handle porting from large codebases in general. Use #version 460, mark your entrypoints, use row-major layouts, and you're done.

(click) For our samples, though, we chose *not* to do this! Instead, we chose to manually port our code over to idiomatic Slang, which looks more like HLSL.
(click) The reason for this goes back to our original principles: we want to avoid abstractions; and since we're teaching how to write samples using Slang, we want to show people how to *get the best* out of the Slang shading language – not how to use GLSL layered on top of Slang.



In addition to pointers, multiple entrypoints, and autodifferentiation, here are a couple of other neat ways our samples use Slang.

Enumerations

GLSL: nvpro_core/dh_hdr.h

```
#ifdef __cplusplus
#define START_BINDING(a) enum a {
#define END_BINDING() }
#else
#define START_BINDING(a) const uint
#define END_BINDING()
#endif

START_BINDING(EnvDomeBindings)
eHdrBrdf     = 0,
eHdrDiffuse  = 1,
eHdrSpecular = 2
END_BINDING();
```


Slang: nvpro_core2/hdr_io.h.slang

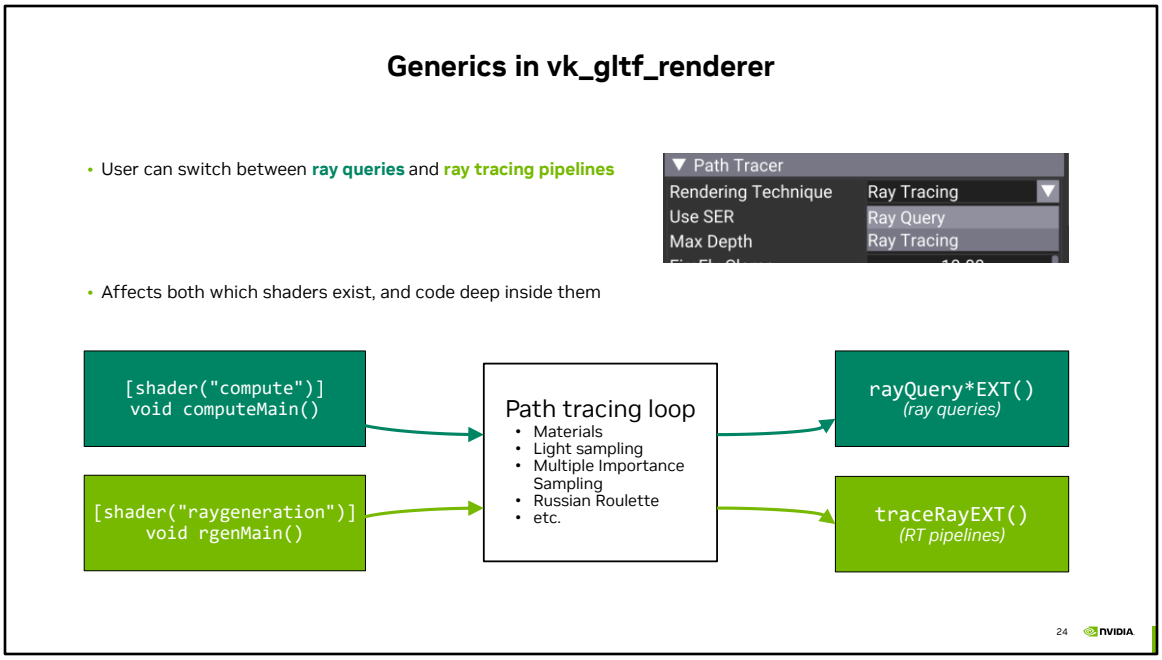
```
enum EnvDomeBindings {
  eHdrBrdf     = 0,
  eHdrDiffuse  = 1,
  eHdrSpecular = 2
};
```

- Scoped by default

23 NVIDIA

We use enums for things like algorithm types or light types. When we were using GLSL, we had to use some preprocessor magic to make enums work in both C++ and GLSL. Now with Slang, our enums are compatible with C++, and they're scoped by default, which is nice.

NVIDIA



We also use Slang's generics in some interesting ways. In some of our ray tracing samples, we want to let the user switch between ray tracing pipelines and ray queries, so they can see which is faster with the sample's shader setup on their GPU.

(click) Typically, ray tracing happens in the innermost parts of the shader: calls to traceRayEXT(), for instance, are surrounded by the rest of the ray-gen or compute shader code to jitter rays, handle multiple bounces, perform multiple importance sampling, and so on.

Generics in vk_gltf_renderer

1. Define an interface for ray tracing backends:

```
interface IRaytracer
{
  void   Trace(RayDesc ray, inout HitPayload payload, inout uint seed);
  float3 TraceShadow(RayDesc ray, inout uint seed);
}
```

2. Implement it for **ray queries** and **ray tracing pipelines**:

```
struct RayQueryRaytracer : IRaytracer
{
  void Trace(…)
  {
    RayQuery rayQuery;
    rayQuery.TraceRayInline(topLevelAS, 0, 0xFF, ray);

    while(rayQuery.Proceed())
    {
      …
```

```
struct TraditionalRaytracer : IRaytracer
{
  void Trace(…)
  {
    HitObject hitObj = HitObject::TraceRay(
      topLevelAS, 0, 0xFF, 0, 0, 0, ray, payload);

    ReorderThread(hitObj);

    …
```

25 NVIDIA

Slang's interfaces and generics give a nice, type-safe way to do this. First, we define an IRaytracer interface. This has a Trace() function to trace a regular ray and store the results in a HitPayload struct, and a TraceShadow() function so the implementation can choose faster ray flags for shadow transmission.

Then we define two implementations of IRaytracer: one which uses ray queries, and one which uses ray tracing pipelines. We can define the closest-hit, miss, and any-hit shaders in the same file as the TraditionalRaytracer struct so our shaders are nicely bundled together. Or, we can use the new HitObject API so we don't need to invoke shaders. And that also lets us use Shader Execution Reordering.

Generics in vk_gltf_renderer

3. Make our path tracing code generic over `IRaytracer`:

```
void processPixel(IRaytracer raytracer, float2 samplePos, float2 imageSize)
{
  …
}
```

4. Call our path tracing code with `RayQueryRaytracer` or `TraditionalRaytracer`:

```
[numthreads(WORKGROUP_SIZE, WORKGROUP_SIZE, 1)]
[shader("compute")]
void computeMain(uint3 threadIdx: SV_DispatchThreadID)
{
  float2 samplePos = (float2)threadIdx.xy;
  uint2  imageSize;
  outImages[int(OutputImage::eResultImage)]
    .GetDimensions(imageSize.x, imageSize.y);

  RayQueryRaytracer raytracer;
  processPixel(raytracer, samplePos, imageSize);
}
```

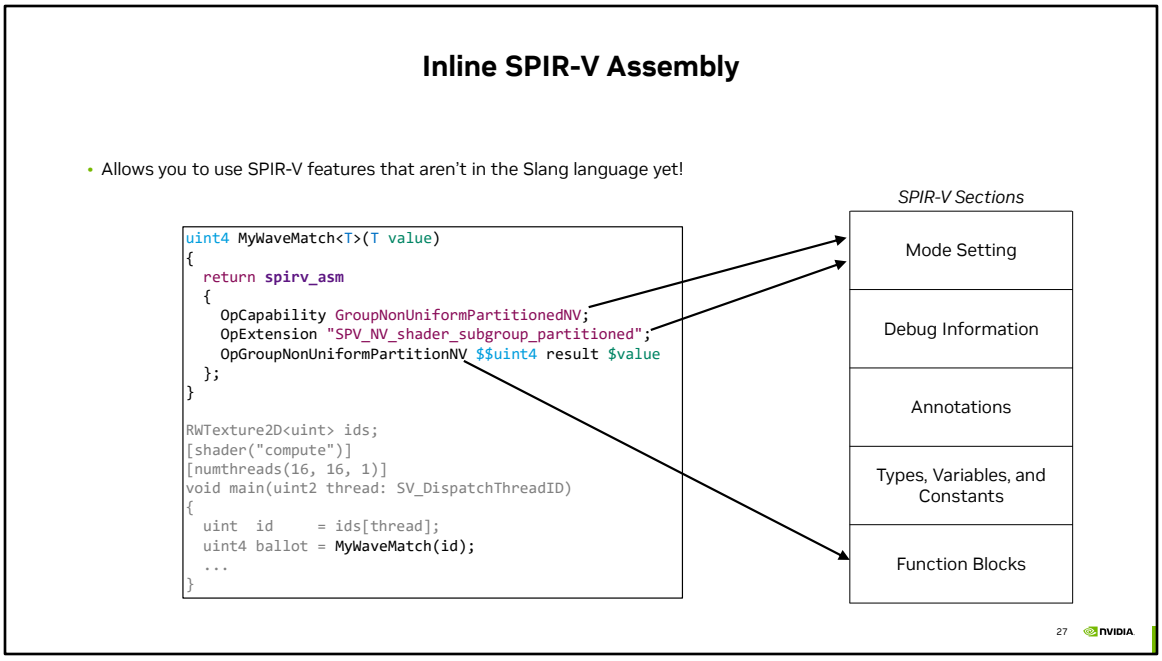
```
[shader("raygeneration")]
void rgenMain()
{
  float2 samplePos = (float2)DispatchRaysIndex().xy;
  float2 imageSize = (float2)DispatchRaysDimensions().xy;



  TraditionalRaytracer raytracer;
  processPixel(raytracer, samplePos, imageSize);
}
```

26 NVIDIA

Then, all we have to do is make our path tracing code generic over IRaytracer.

Now in our compute shader that uses ray queries, we call our path tracer with a RayQueryRaytracer object. And in our ray generation shader that uses ray tracing pipelines, we call our path tracer function with a TraditionalRaytracer object. And now we're done. Our code's now clean, type-safe, and doesn't depend on the preprocessor.

NVIDIA



Sometimes, we develop samples for features that are so new they don't have language support yet. Or, perhaps we're developing a new hardware feature and prototyping new SPIR-V opcodes. Amazingly, the Slang compiler allows you to use new SPIR-V features, even if the Slang language doesn't know about them yet, using *inline SPIR-V assembly*.

This is easiest to describe with an example. **(click)** Imagine if the NonUniformPartition opcode had just been introduced; we wanted to use it, but Slang didn't have it yet. **(click** for section placement)

Full shader example:

RWTexture2D<uint> texIn;
RWTexture2D<uint> texOut;

```
uint4 MyWaveMatch<T>(T value)
{
  return spirv_asm
  {
    OpCapability GroupNonUniformPartitionedNV;
    OpExtension "SPV_NV_shader_subgroup_partitioned";
    OpGroupNonUniformPartitionNV $$uint4 result $value
  };
}


[shader("compute")]
[numthreads(16, 16, 1)]
void main(uint2 thread: SV_DispatchThreadID)
{
  uint tid = thread.x + thread.y;
  uint4 ballot = MyWaveMatch(tid);

  uint value = texIn[thread];
  value = WaveMultiPrefixSum(value, ballot);
  texOut[thread] = value;
}
```

Slang's standard library uses `spirv_asm`

slang / source / slang / **hlsl.meta.slang**

Code | Blame | 32538 lines (30487 loc) · 1.08 MB ·

```
7204    /// Barrier for writes to all memory spaces.
7205    /// @category barrier Memory and control barriers
7206    __glsl_extension(GL_KHR_memory_scope_semantics)
7207    [require(cuda_glsl_hlsl_metal_spirv_wgsl, memorybarrier)]
7208    void AllMemoryBarrier()
7209    {
7210        __target_switch
7211        {
7212        case hlsl: __intrinsic_asm "AllMemoryBarrier";
7213        case glsl: __intrinsic_asm "memoryBarrier(gl_ScopeDevice, (gl_StorageSemanticsShared|gl_StorageSemanticsImage|gl_StorageSemanticsBuffer), gl_SemanticsAcquireRelease)";
7214        case cuda: __intrinsic_asm "__threadfence()";
7215        case metal: __intrinsic_asm "threadgroup_barrier(mem_flags::mem_device | mem_flags::mem_threadgroup | mem_flags::mem_texture | mem_flags::mem_threadgroup_imageblock)";
7216        case spirv: spirv_asm
7217            {
7218                OpMemoryBarrier Device AcquireRelease|UniformMemory|WorkgroupMemory|ImageMemory;
7219            };
7220        case wgsl: __intrinsic_asm "storageBarrier(); textureBarrier(); workgroupBarrier();";
7221        }
7222    }
7223
```

28 NVIDIA

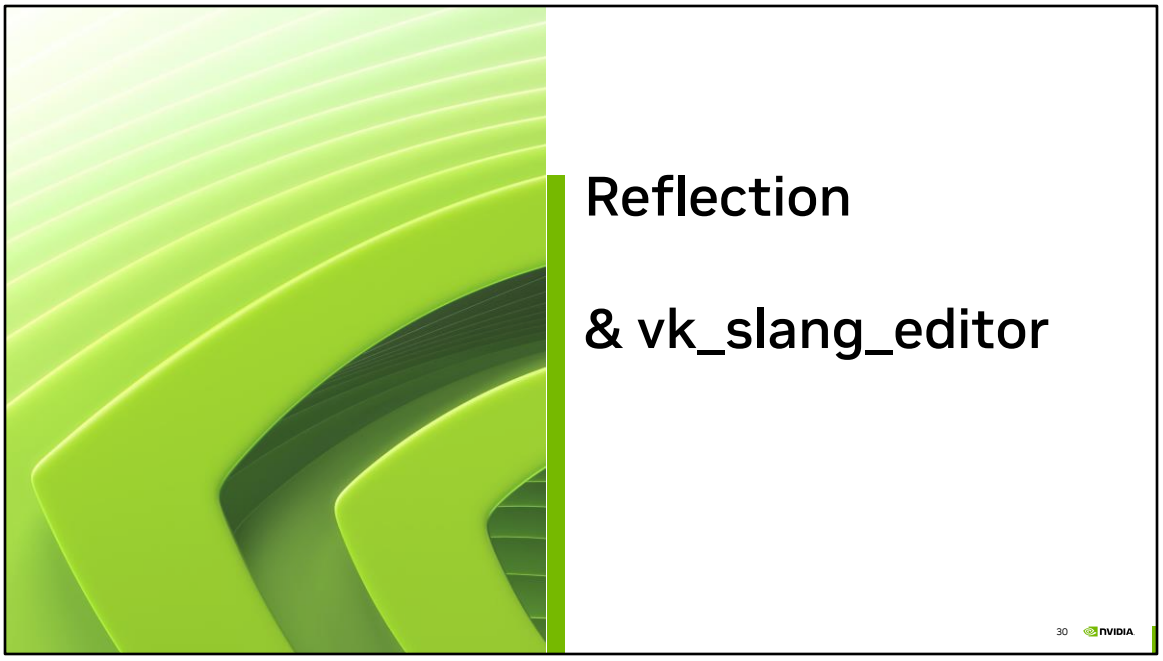
In fact, this is how much of Slang's standard library is defined! If you look at one of the meta.slang files inside Slang's repository, you'll see that most library functions switch depending on the target, then call the intrinsic for the target language or provide SPIR-V assembly.

NVIDIA

Two `spirv_asm` Notes

- `spirv_asm` can't currently create new opaque Slang types
  - E.g. trying to create EXT hit objects in old Slang versions
- If defining new SPIR-V ops, you must rebuild Slang with new SPIR-V headers
  - So that Slang knows in which region each opcode must be placed

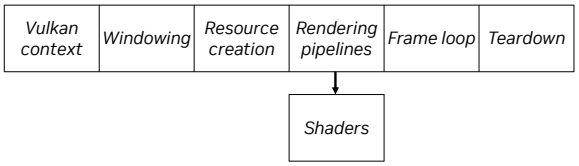
There are two things to note with SPIR-V assembly. First, you can't create new object types this way; Slang doesn't know how to handle opaque SPIR-V types it has no definition for. Secondly, if you're creating new SPIR-V opcodes, you need to define them in the SPIR-V Headers and then recompile Slang, so that Slang knows in which section to put each new SPIR-V opcode.



One of Slang's big features is its powerful shader reflection API, which allows us to write new kinds of apps.

Shader-Defined Renderers

- When writing a Vulkan app, most time is spent on the C++ infrastructure.
  - Pipelines, resources, lifetime management, etc.
  - Shaders are small things that C++ code declares.



- But shaders are (often) the interesting part!

- vk_slang_editor inverts this entire structure.
  - The shader defines the renderer!

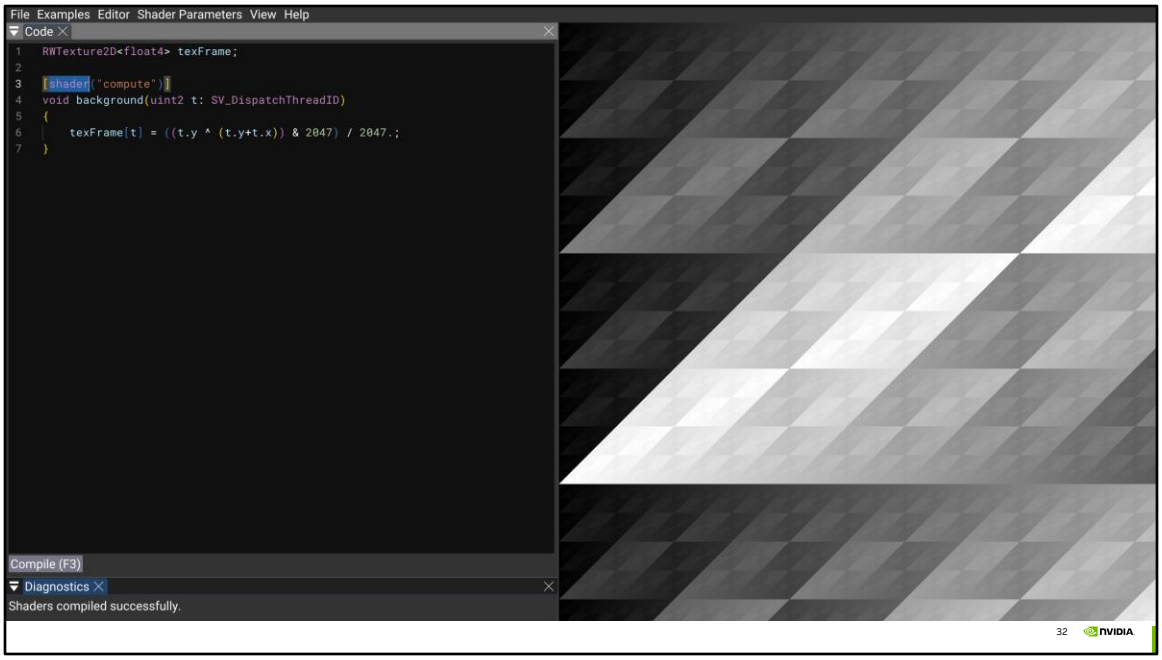
31 NVIDIA

Most of our samples are set up in a very classical way. Typically, when you're writing Vulkan, you spend a lot of time writing C++ code that defines pipelines, resources, and how to manage them. Shaders are the small inner loops within this larger system, with some small bridge headers to match structs between C++ and GLSL.

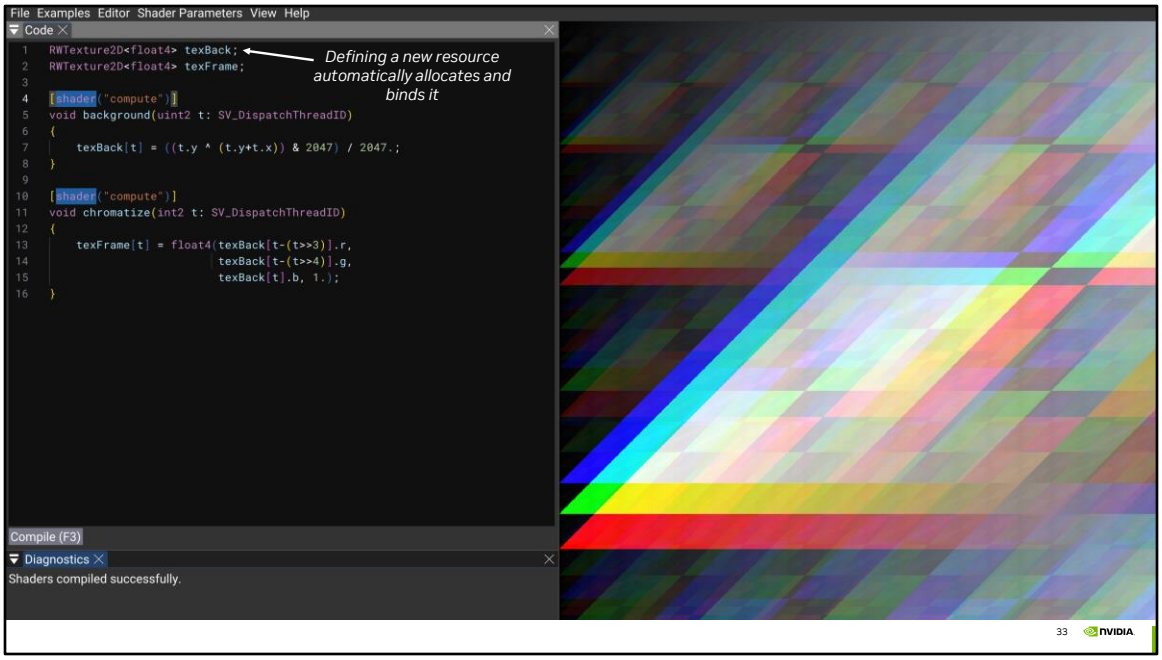
(click) But shaders are usually the interesting part!

So for SIGGRAPH 2025, we wrote **(click)** vk_slang_editor, which is a sample that inverts this entire structure. Instead of C++, you spend your time writing shaders. And based on the Slang shader you write in this interface, the rest of the Vulkan renderer reconfigures itself to run the shader you wrote.

NVIDIA

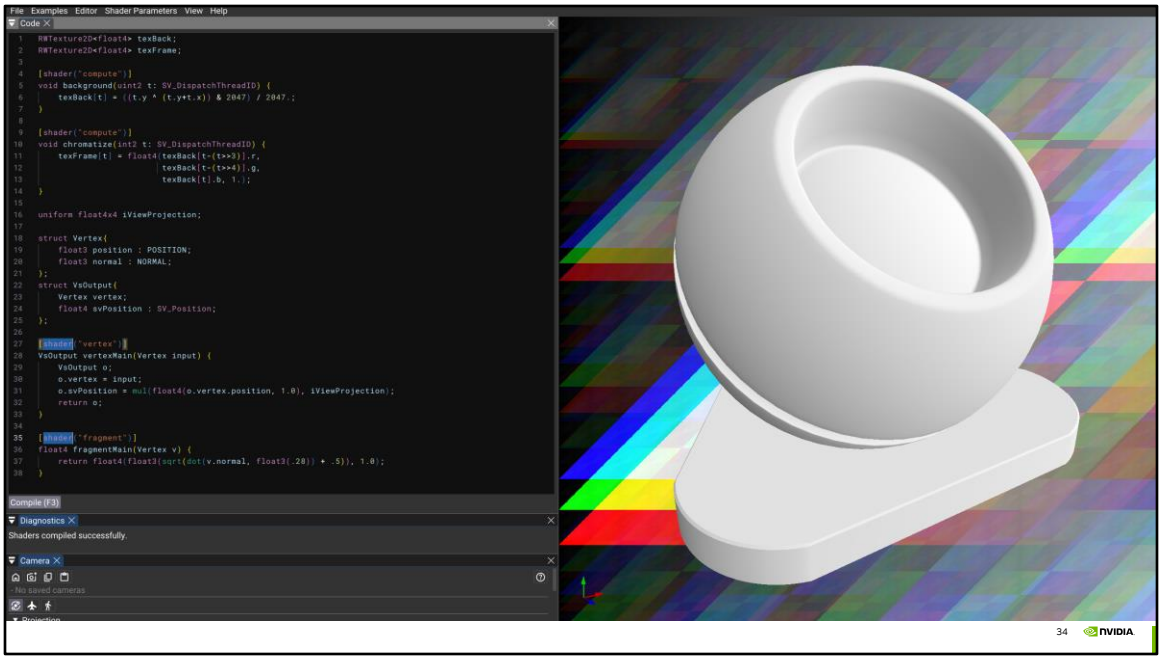


Here's a few examples. Suppose you write a compute shader. Then vk_slang_editor will run it as a full-screen compute pass.

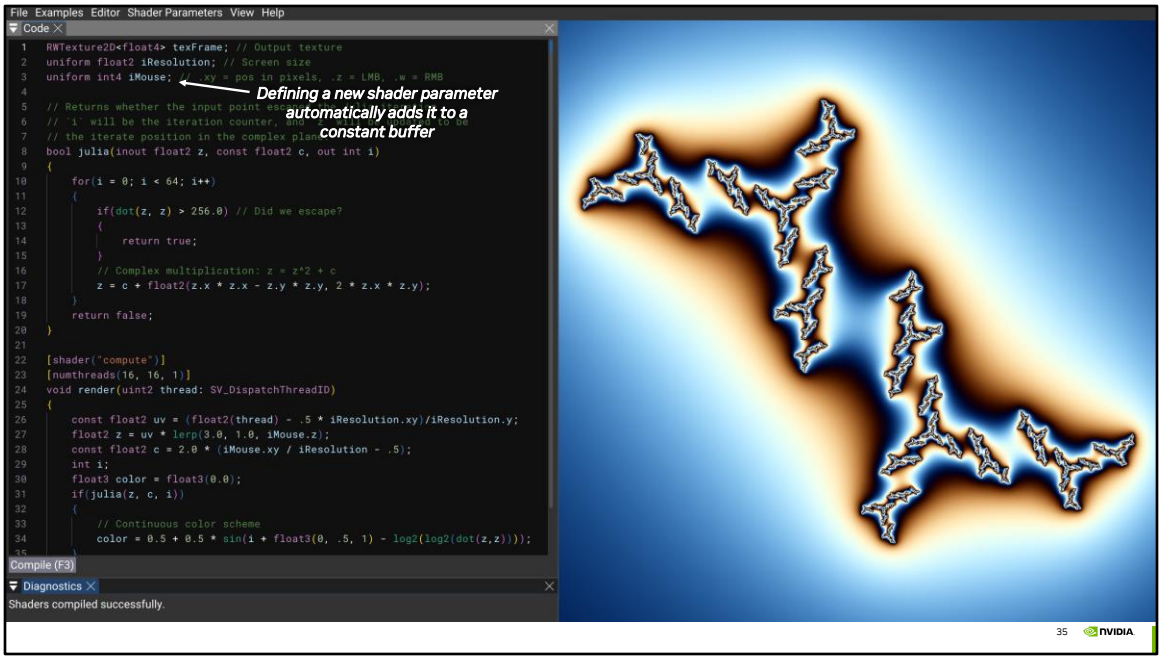


If you want to create a two-pass algorithm, create two compute shaders; we'll run each one in turn.

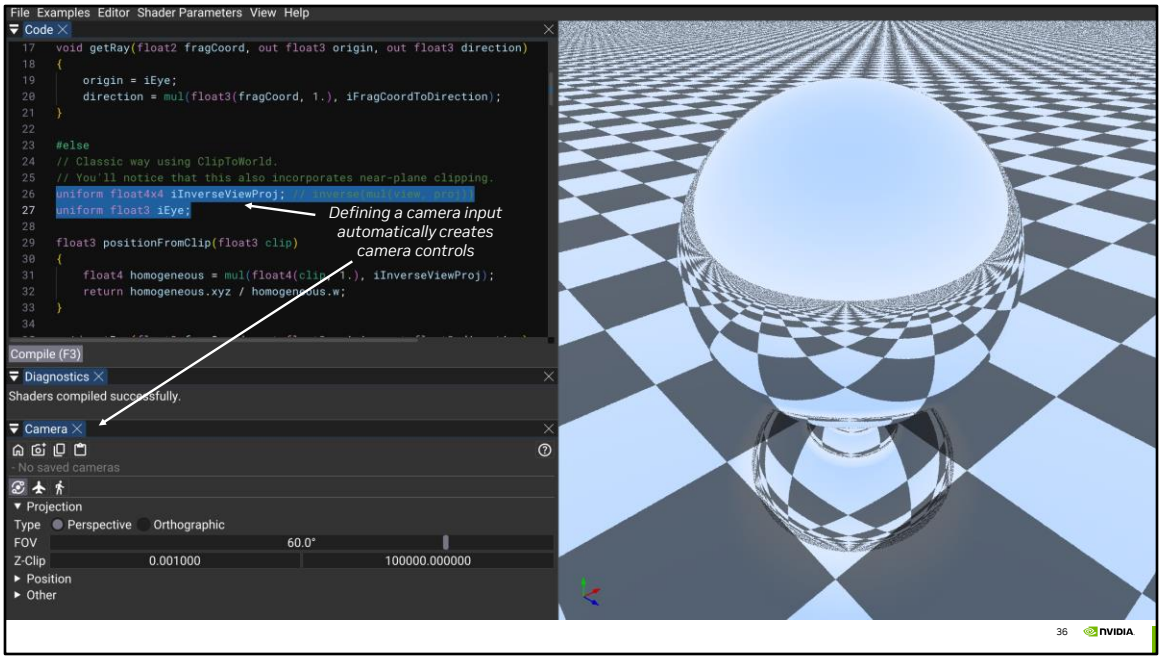
If you need another offscreen backbuffer? **(click)** Define it in the shader, and vk_slang_editor will create it for you.



Suppose you add a vertex and a fragment shader. Then vk_slang_editor will say “ah, that’s a graphics pipeline”, and combine it into one.

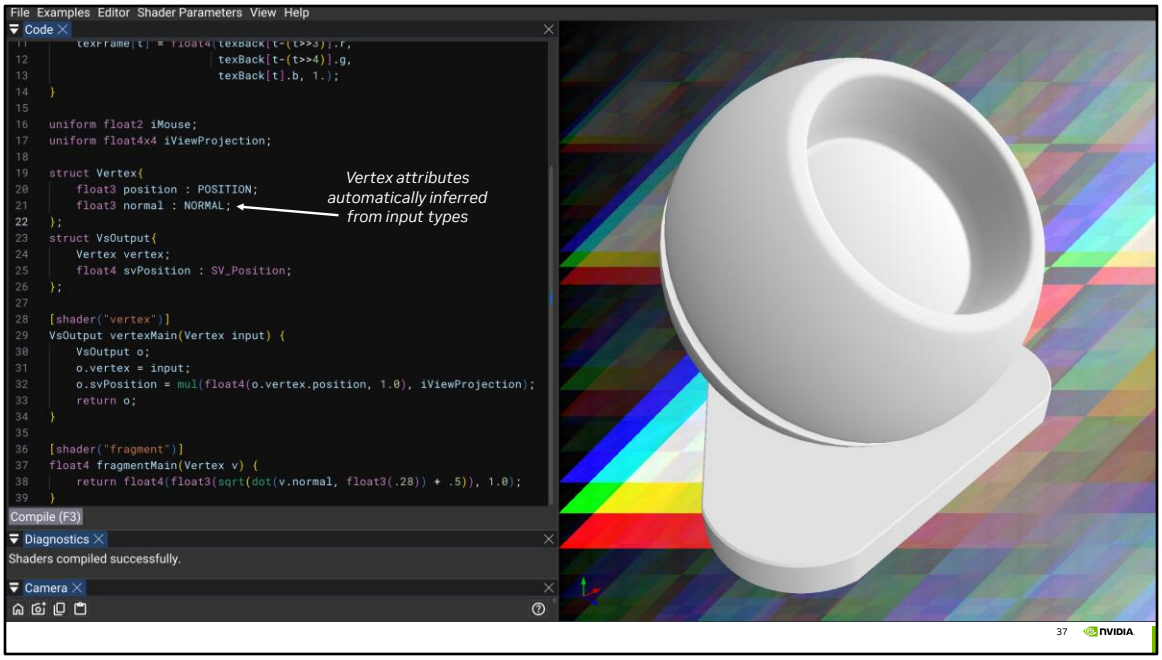


If you need a new shader uniform, like the mouse position, define it with an appropriate name, and we'll start populating it with values. Here we declare `uniform int4 iMouse` at the top of the shader, and now we can use the mouse to control Julia sets.



If you define a camera uniform, we'll even create a camera widget so you can navigate around your shader.

NVIDIA

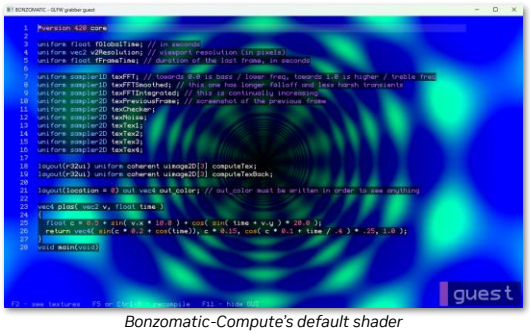


And this even extends to structs. Vertex input attributes, for instance, are defined by your vertex type. If you don't need tangents, we won't bind them. Everything is dynamic.

Why?

- Built for the SIGGRAPH 2025 *Introduction to Slang* hands-on lab
  - Learning a shading language should be about using the language, not building the infrastructure around it

- Inspired by numerous other shader editors:
  - Shadertoy
  - Gigi
  - Bonzomatic
  - compute.toys
  - FX Composer
  - Fragmentarium
  - SHADERed



Bonzomatic-Compute's default shader

38 NVIDIA

We built this for our SIGGRAPH 2025 course where we taught beginners Slang, because learning a shading language should be about using the language, rather than building the infrastructure around it.

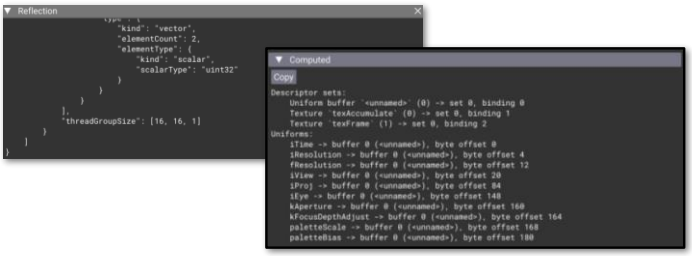
(click) It's inspired by numerous shader editors that came before it, like Shadertoy and the livecoding tool Bonzomatic, which try to minimize the distance between your shader code and the output onscreen.

NVIDIA

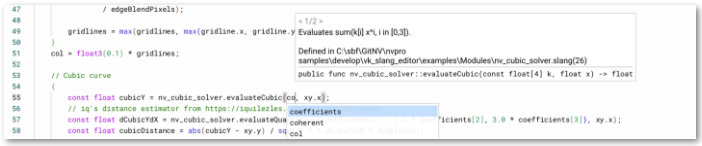
Slang Reflection

Reflection information includes:

- Resource bindings + info
- Struct layouts + info
- Entrypoints + info
- Much more



- vk_slang_editor walks through this to set up all its pipelines and resources.
- Surprisingly small!



- See Theresa Foley's talk, *Getting Started with Slang: Reflections API* online https://shader-slang.org/event/2025/06/04/getting-started-with-slang-reflections-api/

39 NVIDIA

Behind the scenes, this is all possible thanks to Slang's powerful shader reflection features. After a shader has been compiled, we can look at its reflection info to get all of its resources, bindings, structs, struct layouts, entrypoints, functions, and much more. Vk_slang_editor then walks through this info to set up all its pipelines and resources.

(click) Surprisingly, the code to do this is pretty small given that it's so flexible! The code to implement the Vulkan renderer and to do Slang reflection is in fact slightly shorter **(click)** than the code that implements Intellisense in the text editor using the Slang Language Server.

NVIDIA

Other Editor Approaches

- https://shader-slang.org/slang-playground/ is a WebGPU-based editor that uses *custom user attributes:*

```
1  import playground;
2  import rendering;
3
4  [playground::RAND(131072)]
5  RWStructuredBuffer<float> randBuffer;
6
```

playground.slang

```
// Initialize a \`float\` buffer with
uniform random floats between 0 and 1.
[__AttributeUsage(_AttributeTargets.Var)]
public struct playground_RANDAttribute
{
    int count;
};
```

+

Slang API

```
struct VariableReflection {
  unsigned int getUserAttributeCount();

  Attribute* getUserAttributeByIndex(
    unsigned int index
  );
}
```

40 NVIDIA

Now, vk_slang_editor's approach isn't the only possible one. Slang Playground is a web-based editor that also uses reflection, but handles this a different way, using *custom user attributes*. For instance, **(click)** here's a custom attribute that says this buffer has a certain size, and should be filled with random values.

(click) The way this works is that every Slang Playground shader imports a module that looks like this. These structs can then be used with attribute syntax. After the shader has been compiled, Slang Playground uses reflection info to get the custom user attribute struct types and values.

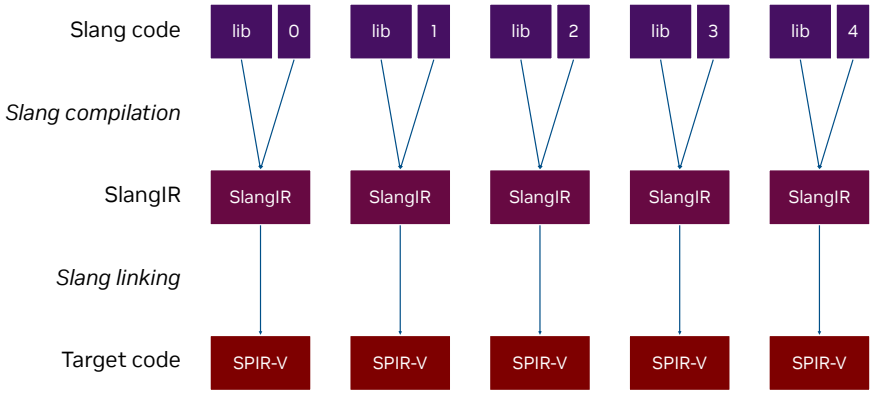
Electronic Arts' Gigi editor supports Slang shaders and has an entirely different and much more general approach to defining shaders and pipelines, and does a number of other impressive things. Alan Wolfe will be presenting on Gigi later today, so please make sure to stick around for that.

In some sense, the power of Slang's reflection is that it makes this whole range of approaches to shader I/O possible.

Finally, let's talk about modules, and how they can help with compile times.

Modules can Reduce Recompilation

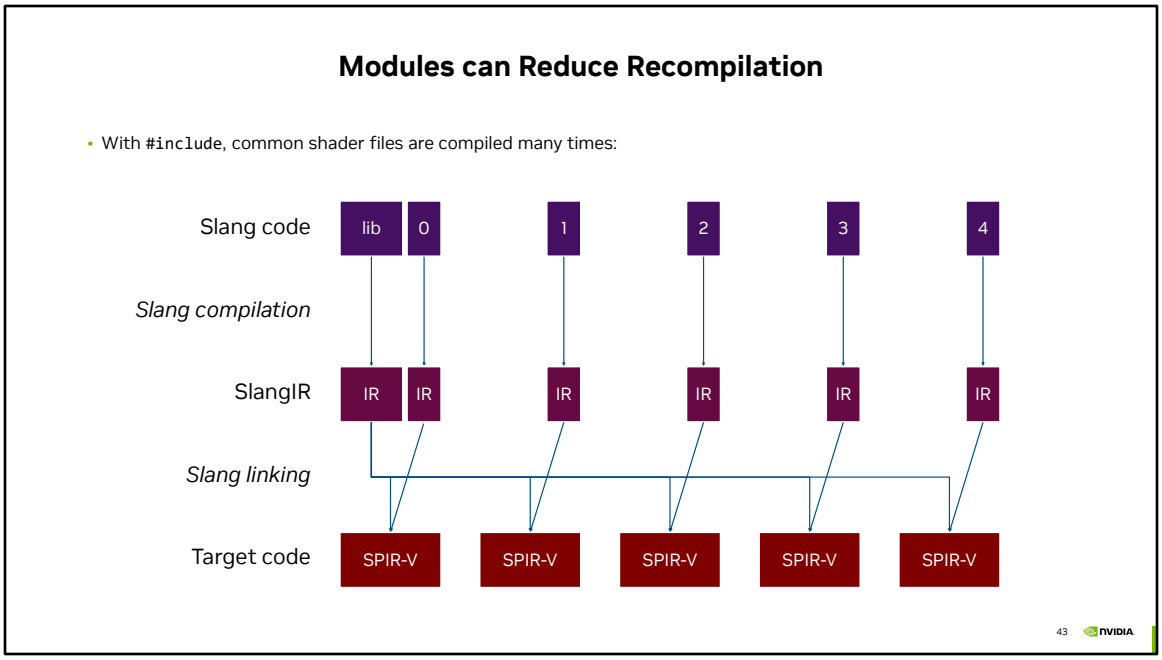
- With `#include`, common shader files are compiled many times:



42 NVIDIA

In our core library, nvpro_core2, we have common shader files that are used across many samples, like a physically-based BSDF library. Most samples have several shaders, and each one includes different subsets of these files. If a sample has 5 Slang shaders, all of which include our PBR material library, Slang will wind up compiling that PBR material library 5 times, which is slow.

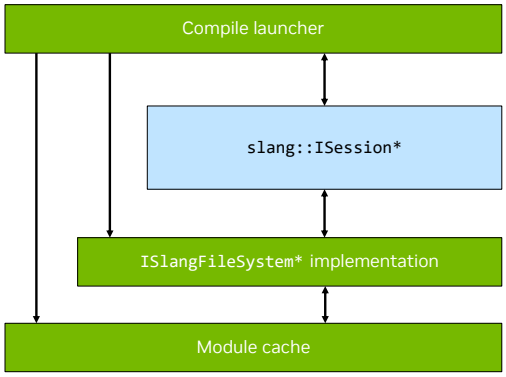
NVIDIA



Modules can help with this. Unlike include files, Slang modules can be compiled to SlangIR separately, and then linked together. This can save compile time because now we only need to compile our PBR material library to SlangIR once.

Now, doing this with the Slang library is nontrivial, because we only know what files a Slang shader imported after the shader has been compiled.

A Module Caching System



- Provide `ISlangFileSystem`
- When Slang searches for a .slang-module:
  - If it's in the cache, provide it
  - Slang will deserialize it into a `slang::IModule*`
- After Slang finishes compiling a shader:
  - Iterate over `slang::ISession::getLoadedModule(i)`
  - Serialize and cache each one not already cached
- Also works with multiple threads (assuming the module cache is thread-safe)

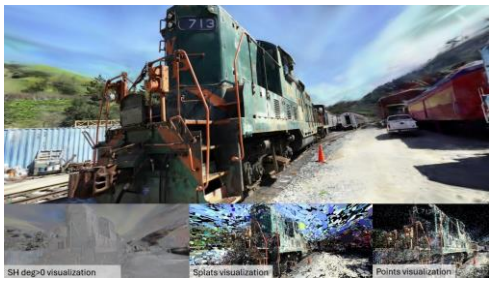
One solution is to intercept Slang's IFileSystem interface. Whenever a Slang module is imported, Slang will first look for a precompiled .slang-module with the same name, before looking for the source .slang file. So if we've already compiled a module, we can provide the .slang-module when Slang looks for it. Once we've finished compiling a Slang shader, we can call — to get all the modules it loaded, then take all the ones that aren't in the cache yet and serialize them to the .slang-module binary format.

(click) This also works with multithreaded compiles. While the Slang::IModule* interface can't be shared across different Slang sessions, because it internally references structures inside its session, .slang-module blobs can be, because they're serialized.

Module Caching Benchmark: vk_gaussian_splatting

- Ported vk_gaussian_splatting to use modules
  - Slang v2025.22.1
  - vk_gaussian_splatting:15 shader files
  - nvpro_core2's core library: 32 header files

Approach	Slang-to-SPIR-V Time
`#include`	673 ms
One module per file	597 ms; 11.3% faster
One module for all of nvpro_core2/nvshaders	744 ms; 10.5% slower



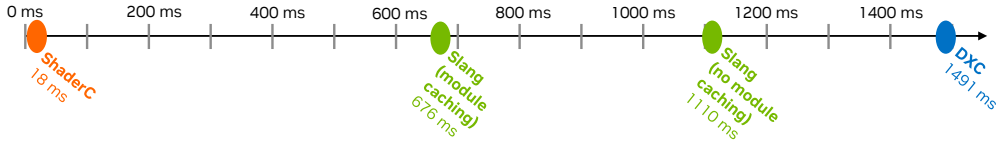
45 NVIDIA

We ran a couple of benchmarks, and got decent results from module caching. In one experiment, I ported our Vulkan Gaussian Splatting example to use modules. Turning on module caching gave a 11% compile time improvement. Interestingly, what seems to work best is turning each header file into its own module; if we instead combine all the headers into a single large module, compile times get 10% *slower*, possibly because we're deserializing lots of SlangIR we don't need.

NVIDIA

Module Caching Benchmark: Hot-Reloading

- https://github.com/NBickford-NV/slang-compile-timer
- 161'720 byte pathtracer from vk_gltf_renderer, 16 modules + 1 main file
- All optimizations and validation turned off, everything in memory, Slang v2026.1.2-5-ga83a63bcb
- Module caching improves hot-reload compile times on this machine and benchmark by 39%



- Also ported this benchmark to GLSL (ShaderC) and HLSL (DirectXShaderCompiler) from Vulkan SDK 1.4.335.0
- All compilers target SPIR-V with fastest compile flags set to make comparisons fair

46 NVIDIA

I also put together an open-source benchmark that simulates *hot-reloading* with and without module caching. For this, I took the main ray query shader from our glTF path tracer along with our core shader library, and ported it to use modules. I figure this is a relatively decent example of a medium-sized codebase; it implements a full PBR material system along with every glTF material extension that existed at the time, and comes out to about 161 KB of shader code.

To simulate hot-reloading, we recompile the shader, acting as if the main shader file had been edited. When module caching is on, we reuse modules for all the other files. On my machine, module caching reduces hot-reload compile times by about 39%.

(optional) One reason this might not be as high a reduction as expected is because although module caching lets us avoid compiling Slang to SlangIR, Slang still has to link the SlangIR modules together and compile SlangIR to SPIR-V. The Slang team has talked about experimental future work to reduce this time by linking together modules after they've been compiled to SPIR-V.

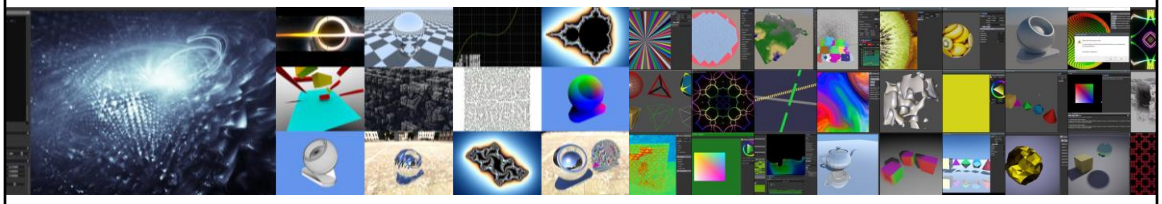
NVIDIA

(click) Finally, I also ported this hot-reload benchmark by hand to GLSL using glslangValidator, and HLSL using DirectXShaderCompiler, simulating an average developer writing code in each of these languages. To make this comparison fair, I had all 3 compile to SPIR-V, and set appropriate compiler flags to make compiles as fast as possible.

(click) Here are the results on this particular benchmark, versions of these compilers, and my machine. You can see that in this case, Slang is faster than DXC, but both are much slower than ShaderC and glslangValidator when it comes to SPIR-V. So Slang's ok, but since my team comes from GLSL, I would *love* to see both Slang and DXC improve their performance and get closer to the left here.

Conclusion

- Slang's **pointers, generics, autodifferentiation,** and more made our code **cleaner and easier to write** while **maintaining high performance**.
- **Porting from GLSL to Slang** is fast: use `#version 460`, mark your entrypoints, and use row-major layout.
- **Inline SPIR-V assembly** lets you use leading-edge features even if they aren't in the standard library yet.
- Slang's **reflection system** is exceptionally detailed and powerful.
- **Module caching can improve compile times**, in addition to modules' other benefits, like encapsulation.



https://github.com/nvpro-samples

Thank you!

47 NVIDIA

This about wraps up our whirlwind tour of some of the neat things we've been getting from using Slang in the nvpro-samples.

Slang's pointers, generics, autodifferentiation, and more made our code cleaner and easier to write, while continuing to have the performance we need.

There's **(click)** an easy route to port from GLSL to Slang, which is what I recommend for most projects starting out: if you have a version directive, the GLSL Compatibility Layer will take care of most of it. You only need to mark your entrypoints, fix up some hull and domain shader declarations, and choose a matrix layout.

We talked about some things unique to Slang, like its **(click)** inline SPIR-V assembly, which allows high levels of control and usage of new features, and its **(click)** exceptionally detailed reflection system, which can be used to make exceptionally dynamic renderers.

Finally, we discussed how **(click)** module caching can improve compile

times, although you'll always want to do performance measurement to ensure it benefits your app.

All of our samples are open-source, and you can find them at github.com/nvpro-samples.

(click) Thank you very much!


NVIDIA.