Slang + Vulkan

Using Slang with Vulkan
Introduction

• Hai Nguyen
• DevTech Engineer @ NVIDIA
• Previously
  - Graphics Tech Lead @ Google/Android XR
  - Graphics Tech Lead @ Google/Stadia (RIP)
• Open Source Contributions
  - Slang
  - SPIRV-Reflect
  - Previously DXC/SPIR-V
Agenda

- Slang overview
- Slang in 2024
- Slang shading language
- Vulkan graphics and compute with Slang
  - Graphics and compute shaders
  - Mesh shaders
  - Ray-tracing shaders
- Slang compiler
- On-ramping from existing code bases
- Slang: Forging Ahead
- Closing
What is Slang?
Slang Overview

• What is Slang?
  - Slang is a high-level shading language bringing modern language features to real-time graphics and bridging it into AI.

• What does Slang bring to the table?
  - Backwards compatible with existing GLSL and HLSL code
  - Module system as a standard language feature
    - Enables better organization of code
    - Improves compilation time
  - Automatic differentiation as a first-class language feature
  - Generics and interfaces provide a straightforward path for shader specialization

• Slang features that we’re not covering today
  - Automatic differentiation, generics and interfaces

• Slang Exploratory Forum in Khronos
  - Proposing to continue open source work under multi-company governance
  - Build industry trust by ensuring that Slang is not “NVIDIA Only”
  - Enhance responsiveness of the current open source project
Slang Source Language, Platforms, and Targets

Slang

Open Source Compiler

github.com/shader-slang/slang

GLSL → Slang

HLSL → Slang

HLSL → Direct3D
GLSL → OpenGL|ES

CUDA
Optix
C (CPU)
PyTorch
Metal Shading Language (beta)
WGSL for WebGPU (coming soon)

x86-64
arm64
Who’s using Slang?
Slang in 2024

• Previous Slang Talks
  - Toward a Next-Gen Vulkan Shading Language: Our Journey with Slang
    - Theresa Foley @ Vulkanised 2024
  - Slang - A Shading Language for the AI-Accelerated Future of Rendering
    - Theresa Foley, Ivan Fedorov @ GDC 2024

• The road to SIGGRAPH 2024
  - Created a plan after GDC to address specific features and issues for Vulkan
  - Used internal and external code bases to suss out compilation related issues
  - Filed 38 Github issues, 16 of which were high priority for a release close to SIGGRAPH 2024
  - Our aim was to hit the following goals
    - Improve on-ramping experience from existing code bases
      - Added missing common uses cases and syntax
      - Added missing variants of functions
      - Added flags to relax some of Slang’s stricter requirements for on-ramping
    - Clarify language usage between Slang and existing shading languages
    - Fix as many of crashy bits as possible before you get to them 😊

• We’re not yet done!
  - Expect more updates at Vulkanised 2025!
Slang Shading Language

Slang - the Language!
Slang Shading Language

- What does Slang’s syntax look like?

- What common language feature does Slang support?
  - Let’s go over a few of these...no need to do a full language review here...
    - User struct types
    - Entry points
    - Resource declarations
    - Functions
    - Namespaces
    - Enums

- How does one use Slang’s module system?

- How does type casting work in Slang?
  - In general, type casting in Slang works the same way as HLSL...
  - ...however, Slang has stricter casting requirements in some cases
    - Stricter requirements helps prevent users from tripping on things like float being implicitly casted to bool
    - Compiler will issue a warning or an error message when casting is not allowed
  - We won’t go into details about this today due to time constraints
  - We are definitely discussing ways to make this more ergonomic for users!
Basic Shader In Slang

- Good news! You’re probably familiar with Slang’s syntax!
- Syntax is similar to HLSL for POD types, resources, struct types, entry points, etc.
- Supports automatic variable type inference using `var` keyword
- Full intellisense support is available in Visual Studio and VSCode

```slang
// BasicVertexShader.slang

struct SceneParameters {
    float4x4 MVP;
};

ConstantBuffer<SceneParameters> SceneParams;

struct VSOutput {
    float4 PositionCS : SV_Position;
    float2 TexCoord : TEXCOORD;
};

[shader("vertex")]
VSOutput vsmain(float3 PositionOS : POSITION, float2 TexCoord : TEXCOORD) {
    var posCS = mul(SceneParams.MVP, float4(PositionOS, 1));
    VSOutput output = (VSOutput)0;
    output.PositionCS = posCS;
    output.TexCoord = TexCoord;
    return output;
}

// BasicComputeShader.slang

#include "pbr.slang"

Texture2D<float4> InTex;
RWTexture2D<float4> OutTex;

[shader("compute")][numthreads(16, 16, 1)]
void main(uint2 dtid : SV_DispatchThreadID) {
    var value = InTex[dtid.xy];
    OutTex[dtid.xy] = value;
}
```
Descriptor Binding Declarations

- Both `[[vk::binding()]]` and `:register()` syntax are supported for resource binding declaration.
- Slang also supports auto assignment for resource binding declarations if that’s your preference.

```slang
// BasicVertexShader.slang

struct SceneParameters {
    float4x4 MVP;
};

[[vk::binding(3, 1)]] ConstantBuffer<SceneParameters> SceneParams;

struct VSOutput {
    float4 PositionCS : SV_Position;
    float2 TexCoord : TEXCOORD;
};

[shader("vertex")]
VSOutput vsmain(float3 PositionOS : POSITION, float2 TexCoord : TEXCOORD) {
    VSOutput output = (VSOutput)0;
    output.PositionCS = mul(SceneParams.MVP, float4(PositionOS, 1));
    output.TexCoord = TexCoord;
    return output;
}

// BasicComputeShader.slang

Texture2D<float4> InTex : register(t1, space0);
RWTexture2D<float4> OutTex : register(u2, space0);

[shader("compute")]
[numthreads(16, 16, 1)]
void main(uint2 dtid : SV_DispatchThreadID) {
    OutTex[dtid.xy] = InTex[dtid.xy];
}
```

Resource declaration with `[[vk::binding()]]`
Resource declarations with `:register()`
Functions, Namespaces, Enums, etc

- **Functions**
  - Functions are globally visible so forward declarations are not necessary
  - Function definition and declaration must happen at same time
  - Functions can also be defined using modern syntax with `func` keyword

- **Namespaces**
  - Works similarly to HLSL
  - Nested namespaces are supported:
    ```cpp
    namespace NS0::NS1::NS2 {}
    ```
  - Namespaces can use dot notation
    ```cpp
    namespace NS0.NS1.NS2 {}
    ```
  - Cannot declare function in namespace and implement it later

- ** Enums**
  - Enums by default are scoped if there is a declared name
  - Use `[UnscopedEnum]` attribute to allow unscoped enum usage

```
// PixelShader.slang

enum Selector { First = 0, Second }

namespace MyNS {
    float4 Add(float4 a, float4 b) {
        return a + b;
    }
}

float4 MyNS::Sub(float4 a, float4 b) { return a - b; }

[shader("pixel")]
float4 psmain(float4 value : V, int select : S, int bump : B) : SV_Target {
    if ((select == Selector::First) && (bump == ProcBump))
        return MyNS::Add(value, (float4)0);
    else
        return MyNS::Add(value, (float4)1);
}
```
User Struct Types

- **Constructors (subject to change)**
  - Constructor without any parameters is the default constructor
  - Default constructors are called automatically at definition of variable

- **Member Functions**
  - `[mutating] attribute required if function modifies member vars
  - Static functions can be called using both the scope resolution operator and the dot accessor operator

- **Properties**
  - Similar to C# and Swift
  - `property::set` accessor is Implicitly `[mutating]`
  - Alternative modern syntax for declaration:
    ```python
    property float4 r_mask {
        get { return float4(x.x, 0, 0, 0); }
        set { x = float4(newValue.x, x.y, x.z, x.w); }
    }
    ```
  - Static member function
    ```python
    static float4 Add(float4 a, float4 b) { return a + b; }
    ```

- **Operator Overloading**
  - See Basic Convenience Features in Slang docs

- **class types are not supported**
  - `class` is a reserved keyword for future use
  - Usage of `class` is invalid for GPU code

```python
// PixelShader.slang
struct Pusher {
    float4 x; // Member variable

    __init(float4 _x) { x = _x; }
    _init() { x = (float4)0; }

    void inc(float4 amt) { x += amt; }

    property float4 r_mask {
        get { return float4(x.x, 0, 0, 0); }
        set { x = float4(newValue.x, x.y, x.z, x.w); }
    }

    static float4 Add(float4 a, float4 b) { return a + b; }

    shader("pixel")
    float4 psmain(float4 value : V) : SV_Target {
        var p = Pusher(value);
        p.inc((float4)1);
        p.dec((float4)2);
        p.r_mask = float4(value.y, 0, 0, 0);
        float4 y = p.r_mask;
        return Pusher::Add(p.x, Pusher::Add(y, y));
    }
}
```
**Modules**

- **Modules**
  - Can be made up of more than one file
  - Can be compiled separately to Slang IR
  - Can be easily imported by shader code

- **Module Function and Struct Visibility**
  - Access control modifiers
    - `public`
      - Visible to other files and other modules
    - `internal` (default visibility)
      - Visible only to files within module
    - `private`
      - Visible only within parent struct

- **Compiling source files to Slang IR modules**
  - Compile `.slang` files to `.slang-module`:
    - `slangc -o pbr.slang-module pbr.slang`
    - `slangc -o aces.module.slang aces.slang`

- **Compiling to final SPIR-V**
  - Compile `.slang` file that uses `.slang-module` modules to SPIR-V:
    - `slangc -target spirv -profile ps_6_0 -entry psmain shader1.slang`
    - `slangc -target spirv -profile ps_6_0 -entry psmain shader2.slang`
  - Slang will look in current directory and `-I <dir>` paths for modules
    - `pbr.module.slang` and `aces.module.slang` in this case

---

```slang
// pbr.slang
module pbr;

float3 Fresnel(float3 x) {
    return x;
}

public float3 ShadeMat(float3 pos) {
    return Fresnel(pos);
}
```

```slang
// aces.slang
module aces;

public struct MyStruct {
    private int x;
}

public float3 ACESFilm(float3 x) {
    return ApplyACES(x);
}
```

```slang
// shader1.slang
import pbr;
import aces;

[shader("pixel")]
float4 psmain(float3 pos : P) : SV_Target {
    float3 c = ShadeMat(pos);
    c = ACESFilm(c);
    return float4(c, 0);
}
```

```slang
// shader2.slang
import aces;

[shader("pixel")]
float4 psmain(float3 color : C) : SV_Target {
    return float4(ACESFilm(color), 0);
}
```
Modules

- **Compilation Time Reduction**
  - Precompiled modules can reduce compile time
  - Compilation to module only happens once
    - Lexing / Parsing
    - Preprocessing
    - AST building
    - Module IR code generation
  - 1.3x speed up using modules vs monolithic (#4661)
    - ~9k LoC using MDL OmniSurface_BushedMetal

- **Code Organization Benefits**
  - Allows for improved logical code organization
    - All PBR functions can go into a pbr module
    - All ACES functions can go into an aces module
  - Remember a module can span multiple source files providing more precision in code organization

- **Legacy Behavior**
  - You can also use modules directly from source...but will lose out on compilation time reduction
    - > slangc -target spirv -profile ps_6_0 -entry psmain shader1.slang

- **How does slang search for modules?**
  - Slang looks in current dir and any -I <dir> paths
  - Slang looks for .slang-module modules first then falls back to .slang modules

---

```slang
// pbr.slang
module pbr;

float3 Fresnel(float3 x) {
  return x;
}

public float3 ShadeMat(float3 pos) {
  return Fresnel(pos);
}

// aces.slang
module aces;

public struct MyStruct {
  private int x;
}

public float3 ACESFilm(float3 x) {
  return ApplyACES(x);
}

// shader1.slang
import pbr; import aces;  
[shader("pixel")] float4 psmain(float3 pos : P) : SV_Target {
  float3 c = ShadeMat(pos);
  c = ACESFilm(c);
  return float4(c, 0);
}

// shader2.slang
import aces;  
[shader("pixel")] float4 psmain(float3 color : C) : SV_Target {
  return float4(ACESFilm(color), 0);
}
```
Writing Vulkan Shaders in Slang
Vulkan Graphics and Compute with Slang

- Highlight of Slang features for Vulkan
  - Compiles Slang, GLSL, and HLSL shader source to SPIR-V
  - `[[vk::*]]` style attributes are supported
  - Most of DXC’s `-fvk-*` flags are supported
  - For buffer layouts `-fvk-use-scalar-layout` and `-fvk-use-g1-layout` are supported
  - Use `-fvk-use-entrypoint-name` so output SPIR-V has source entry point(s) instead of `main()`
  - Inline SPIR-V assembly
  - Pointers (limited)

- Vulkan features that are currently not supported
  - `-fvk-use-dx-layout` is coming soon! (#4126)
  - Specialization constants are not supported
Supported shader stages for Vulkan

- All traditional graphics pipeline stages supported
  - Vertex [shader("vertex")]
  - Hull / Tessellation Control [shader("hull")]
  - Domain / Tessellation Evaluation [shader("domain")]
  - Geometry [shader("geometry")]
  - Pixel / Fragment [shader("pixel")]

- All mesh shading pipeline stages supported
  - Amplification / Task [shader("amplification")]
  - Mesh [shader("mesh")]

- Compute shading stage supported
  - Compute [shader("compute")]

- All ray-tracing pipeline stages supported
  - Ray Generation [shader("raygeneration")]
  - Miss [shader("miss")]
  - Closest Hit [shader("closesthit")]
  - Any Hit [shader("anyhit")]
  - Intersection [shader("intersection")]
  - Callable [shader("callable")]

This work is licensed under a Creative Commons Attribution 4.0 International License
Graphics and Compute Shaders

- Traditional graphics and compute shaders in Slang
  - Look more or less like equivalent HLSL shader
  - Shader state attributes are highly suggested but are optional
    - True for all shader stages
  - Entry points can be custom or simply main()
    - True for all shader stages
- Descriptor binding declarations
  - Will get warnings for :register() but binding numbers will be correct in SPIR-V
    - True for all shader stages
- sampler keyword is not supported
  - Use SamplerState instead

```cpp
struct CameraProps { /* matrices */ }

ConstantBuffer<CameraProps> Camera : register(b0);
Texture2D Texture0 : register(t1);
SamplerState Sampler0 : register(s2);

Matrix4x4 World, View, Projection;

struct VSOutput
{
    float4 PositionWS : POSITIONWS;
    float4 PositionCS : SV_POSITION;
    float2 TexCoord : TEXCOORD;
    float3 Normal : NORMAL;
};

[shader("vertex")]
VSOutput vsmain(...) {}

[shader("pixel")]
float4 psmain(VSOutput input) : SV_Target
{
    float3 lightPos = float3(1, 3, 5);
    float3 lightDir = normalize(lightPos - input.PositionWS.xyz);
    float diffuse = 0.8 * saturate(dot(input.Normal, lightDir));
    float ambient = 0.2;

    float3 color = Texture0.Sample(Sampler0, input.TexCoord).xyz;
    color = (ambient + diffuse) * color;
    return float4(color, 1);
}
```
Mesh and Amplification Shaders

- Mesh and amplification shaders
  - Looks like HLSL mesh and amplification shader
- Interpolation of uint vertex variables
  - Flat interpolation of uint vertex variables from mesh to pixel work correctly in Slang
  - Bug currently in DXC that causes flat interpolation of uint vertex variables that causes artifacting

```c
struct Payload { uint MeshletIndices[AS_GROUP_SIZE]; };  
groupshared Payload sPayload;

[shader("amplification")][numthreads(AS_GROUP_SIZE, 1, 1)]
void asmain(uint gtid : SV_GroupThreadID, uint dtid : SV_DispatchThreadID)
{
    sPayload.MeshletIndices[gtid] = dtid;
    DispatchMesh(AS_GROUP_SIZE, 1, 1, sPayload);
}

struct MeshOutput { float4 Position : SV_POSITION; }

[shader("mesh")][outputtopology("triangle")][numthreads(128, 1, 1)]
void msmain(uint gtid : SV_GroupThreadID, uint gid : SV_GroupID, in payload Payload payload, out indices uint3 triangles[128], out vertices MeshOutput vertices[64])
{
    uint meshletIndex = payload.MeshletIndices[gid];
    Meshlet m = Meshlets[meshletIndex];
    SetMeshOutputCounts(m.VertexCount, m.TriangleCount);

    if (gtid < m.TriangleCount) triangles[gtid] = uint3(vIdx0, vIdx1, vIdx2);

    if (gtid < m.VertexCount) vertices[gtid].Position = mul(Cam.MVP, Position);
}
```
Ray-Tracing Shaders

Casting return values of DispatchRaysIndex() and DispatchRaysDimensions()
- Both functions return a uint3 which HLSL and DXC will let you cast and truncate to a float2
- Slang’s doesn’t allow truncation between vector types
  - An error will result if you cast uint3 to a float2
  - Swizzle the results to uint2 then cast to a float2
  - This may change in the future and a warning will be issued instead but for now: swizzle then cast

Compiling ray tracing shaders to a library
- Compile commands - any of these will work:
  - slangc -target spirv -fvk-use-entrypoint-name ray_trace_shaders.slang
  - slangc -target spirv -fvk-use-entrypoint-name -profile lib_6_3 ray_trace_shaders.slang
  - slangc -target spirv -fvk-use-entrypoint-name -profile sm_6_3 ray_trace_shaders.slang
- If you specify an entry point only that entry point will appear in the final SPIR-V

Slang ray tracing shader examples
- github.com/Autodesk/Aurora/tree/main/Libraries/Aurora/Source/Shaders

```cpp
RWTexture2D<float4> RenderTarget;

[shader("raygeneration")]
void MyRaygenShader ()
{
    const float2 pcNS = (float2)DispatchRaysIndex();
    const float2 uvNS = pcNS/(float2)DispatchRaysDimensions();
    const float2 pc = (float2)DispatchRaysIndex().xy;
    const float2 uv = pc/(float2)DispatchRaysDimensions().xy;
    RenderTarget[DispatchRaysIndex().xy] = float4(uv, 0, 0);
}

[shader("miss")]
void MyMissShader (inout RayPayload payload) {
    payload.color = float4(0, 0, 0, 1);
}

[shader("closesthit")]
void MyClosestHitShader (inout RayPayload payload, in MyAttributes attr) {
    float3 P = WorldRayOrigin() + (RayTCurrent() * WorldRayDirection());
    float3 bc = float3(1 - attr.barycentrics.x - attr.barycentrics.y, attr.barycentrics.x, attr.barycentrics.y);
    float3 N = N0 * bc.x + N1 * bc.y + N2 * bc.z;
    payload.color = float4(N + P, 1);
}
```

Error: cannot convert vector<uint,3> to float2
Swizzle to vector<uint, 2> and then cast

Optional but HIGHLY SUGGESTED
Compiling with Slang

Slang - the Compiler!
slangc for offline compilation

- Slang command line options for Vulkan
  - `slangc -target spirv -profile spirv_1_4 -fvk-use-entrypoint-name -entry psmain -o shader.spv shader.slang`
  - `slangc -target spirv -profile ps_6_5 -fvk-use-entrypoint-name -entry psmain -o shader.spv shader.slang`
  - No need to specify `-lang slang` or `-emit-spirv-directly` since they’re the defaults
  - If no `-entry` options are given, Slang will use `[shader(...)]` attributes to detect entry points

- Profiles
  - Use SPIR-V profiles when targeting SPIR-V
    - `-profile spirv_{1_0, 1_1, 1_2, 1_3, 1_4, 1_5, 1_6}`
  - HLSL style profiles if you need get more specific
    - `-profile sm_{4_0, 4_1, 5_0, 5_1, 6_0, 6_1, 6_2, 6_3, 6_4, 6_5, 6_6, 6_7}`
      - This also applies to vs, hs, ds, gs, ps
      - For example: `-profile sm_6_5`
    - `-profile lib_{6_1, 6_2, 6_3, 6_4, 6_5, 6_6, 6_7}`
    - `-profile as_/ms_{6_5, 6_6, 6_7}`
      - Amplification and mesh shaders
slangc for offline compilation

- Preprocessor macros and __target_switch
  - Slang has __SLANG__ preprocessor macro to help isolate code for Slang compiler
  - Slang does not have a __spirv__ preprocessor macro like DXC
    - Add -D__spirv__ when compiling if needed
  - Slang has a mechanism called __target_switch that lets you customize code for various targets (§4307)
    - See __target_switch in Slang docs

- Compiling specific entry points
  - If you don’t specify an entry point Slang will look for main()
    - slangc -target spirv -profile vs_6_3 two_entry_points.slang results in an error since no entry was specified and main() is not in the file
  - If a file has only one entry point that’s decorated with a stage attribute
    - slangc -target spirv one_entry_point.slang will produce a SPIR-V 1.5 binary with one entry point (same as compiling a library)
    - Note the compile command did not specify an entry point
  - If a file has multiple entry points with [shader(...)] and -entry is not specified, use -fvk-use-entrypoint-name to retain source entry point names
  - Specifying -stage requires -entry or the file must contain a single entry point called main()
slangc for offline compilation

• Compiling to SPIR-V libraries
  - Following commands will compile raytrace.slang to a SPIR-V library
    - `slangc -target spirv -fvk-use-entrypoint-name raytrace.slang` (SPIRV 1.5)
    - `slangc -target spirv -profile lib_6_3 -fvk-use-entrypoint-name raytrace.slang` (SPIRV 1.4)
    - `slangc -target spirv -profile sm_6_3 -fvk-use-entrypoint-name raytrace.slang` (SPIRV 1.5)
  - Don’t specify an entry point if you’re trying to compile a file to a library
    - If you specify an entry point only that entry point will appear in the final SPIR-V

• Compiling to modules
  - Following command produces a Slang IR module called pbr.slang-module
    - `slangc -target spriv -o pbr.slang-module pbr.slang`
  - The extension .slang-module is required for Slang to locate the module for import
    - If .slang-module IR isn’t found for an import statement, Slang will fall back to .slang source module

• Slang’s module searching
  - Slang looks in current directory and any -I <dir> paths for modules
    - `slangc -target spirv -profile lib_6_3 -I some/dir -I ../some/other/dir raytrace.slang`
  - Slang looks for .slang-module modules first then falls back to .slang

• Modules saves on precious compile time!
Slang and Existing Code Bases
On-ramping from GLSL code bases

- **Use the** `–lang glsl` **option to set language mode to GLSL**
  - Pulls in GLSL specific intrinsics

- **Buffer Layouts**
  - Slang defaults to std140 for constant buffers and std430 for storage buffers
  - For scalar block layout either `-force-glsl-scalar-layout` or `-fvk-use-scalar-layout` will work
    - `-fvk-use-scalar-layout` is an alias for `-force-glsl-scalar-layout`

- **GLSL code should “just compile” successfully...**
  - ...but we might have missed some corner cases
  - Please report any issues ([github.com/shader-slang/slang/issues](https://github.com/shader-slang/slang/issues))
On-ramping from HLSL code bases

- **Use `-unscoped-enums` to ease on-ramping**
  - Slang expects scoped enums by default
- **Slang does not have `__spirv__` preprocessor macro**
  - Add `-D__spirv__` to command line if needed
- **Buffer Layouts**
  - Slang defaults to std140 for constant buffers and std430 for structured buffers and related
  - Use `-fvk-use-scalar-layout` set buffer layout to scalar block layout
  - Use `-fvk-use-gl-layout` to set std430 layout for raw buffer load/stores
  - `StructuredBuffer` and related objects can also take a per resource layout
    - `StructuredBuffer<MyStruct, Std140DataLayout>`
    - `StructuredBuffer<MyStruct, Std430DataLayout>`
    - `StructuredBuffer<MyStruct, ScalarDataLayout>`
- **#pragma pack_matrix() is not currently supported (#4202)**
  - If possible, use `-matrix-layout-column-major` or `-matrix-layout-row-major` to force matrix packing style
- **Will probably need to add some explicit type casting here and there**
  - Some casting issues will show as errors while others will show as warnings
  - We’re discussing how to improve the ergonomics of this
On-ramping from HLSL code bases

- 1-dimensional matrix types are not currently supported (#4395)
  - float1x1, float1x2, float1x3, float1x4
  - float2x1, float3x1, float4x1

- out and inout parameter decorations behave differently in Slang (#4430)
  - In particular this can affect how resources are passed around

- Matrix variant of select, and, and or are not currently supported (#4442)
  - Was new to me too
  - Vector variants are supported

- No support for inheritance
  - Limited syntax may compile but functionality is UB

- unsigned int currently does not work (#4458)
  - May seem nonsensical, but in shader code uint is almost exclusively used

- Forward declaration is not required and is not supported (#4446)
  - Will require some shader code change if code makes use of forward declarations
In Closing
Slang: Forging Ahead

- **Slang’s development philosophy**
  - All key technical work done in the open
  - Language discussions take place in the open
  - Users and contributors can help shape both the language and the compiler

- **Remain committed to productization of Slang for industrial strength**
  - Keep improving support for current and future targets
    - Vulkan, Direct3D, Metal (beta), OpenGL, CUDA, Optix, CPU, PyTorch
    - WebGPU coming soon!
  - Add new language and compiler features relevant to Slang’s user base

- **Slang Exploratory Forum in Khronos**
  - Proposing to continue open source work under multi-company governance
  - Build industry trust by ensuring that Slang is not “NVIDIA Only”
  - Enhance responsiveness of the current open source project

- **We would love contributions from you** ([github.com/shader-slang/slang](https://github.com/shader-slang/slang))
  - We welcome all bug reports, questions, and pull requests
Questions?!

- [github.com/shader-slang/slang](https://github.com/shader-slang/slang)
- Thanks to the following folks for their help and support with this talk!
  - Nat Duca
  - Shannon Woods
  - Brandon Mills
  - Yong He
  - Theresa Foley
  - Neil Trevett
  - Ariel Glasroth
  - Ellie Hermaszewska
  - Ivan Federov
  - Daniel Koch
  - Ryan Prescott
  - Michael Songy
  - Brad Loos