Writing Shaders Today: Complexity Abounds

- Shader codebases have become incredibly large & complex
- Developers need to deploy to many platforms
- Shader combinatorial explosion
- Fragile manual data binding models
- New graphics techniques & neural graphics discontinuity

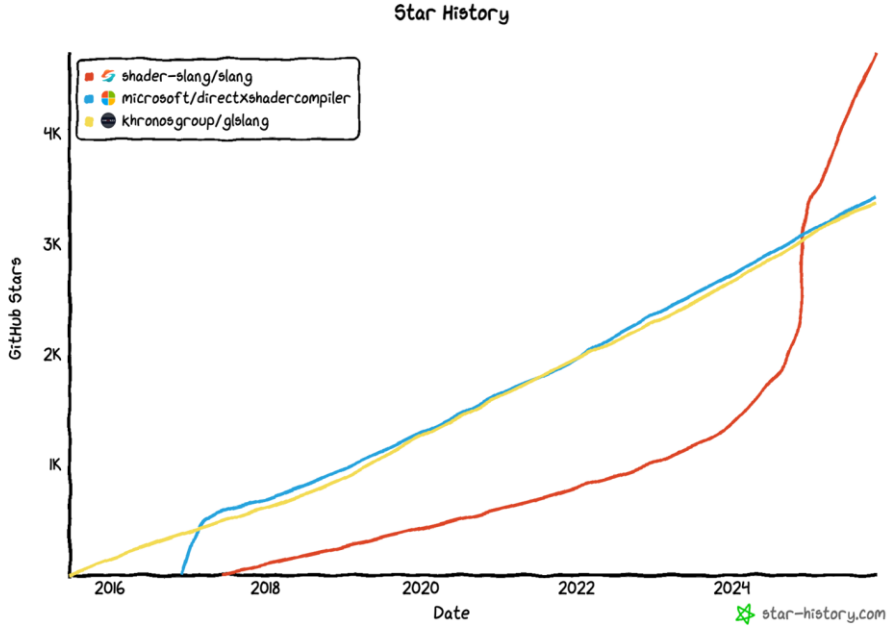
Slang: Open-Source and Cross-Platform

The Slang Developer Community

Developer interest continuing to grow!

- 4700+ GitHub Stars, >60 active contributors
- >1000 Discord members, lively discussion threads

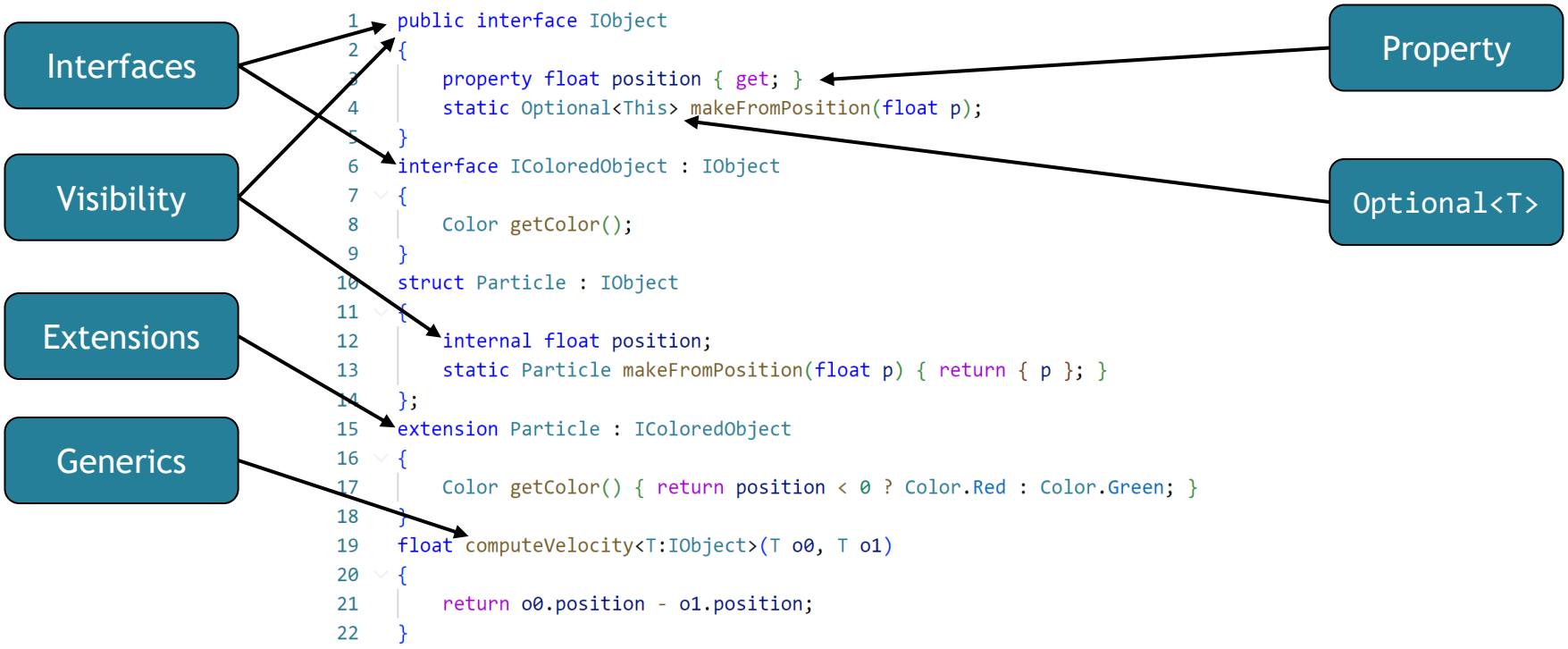
Join us on Discord!



Try it out on the web: https://try.shader-slang.org

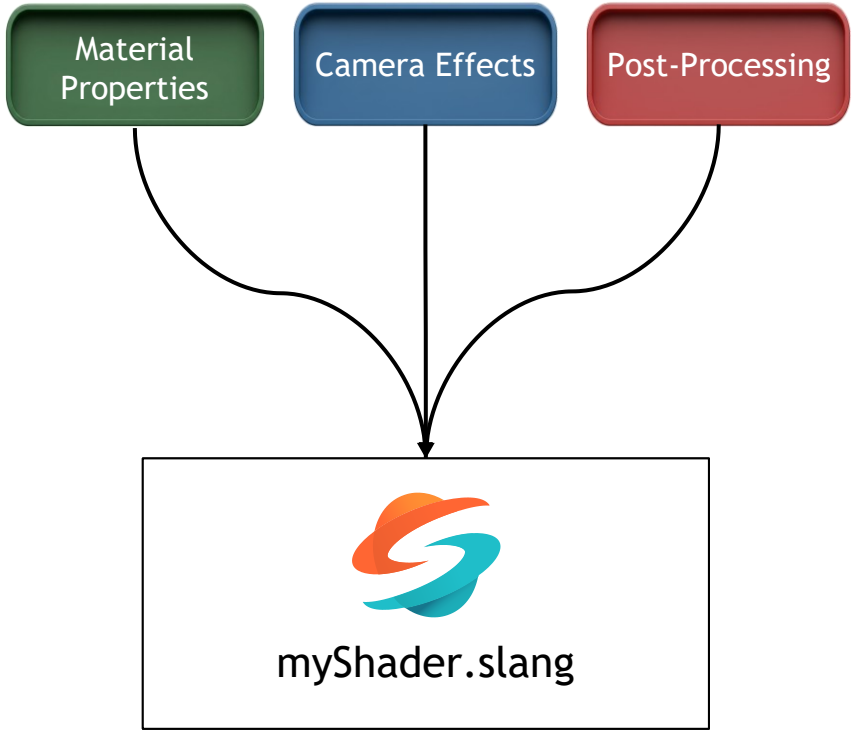
See the documentation: https://docs.shader-slang.org

Modern Features for Modern Shaders



Modules


#INCLUDE


Material Properties
Camera Effects
Post-Processing
myShader.slang

Modules: Organizing Your Code

```
// material.slang
module material;

public struct Material {
    public float4 evalBRDF(float3)
      { /* ... */ }
    internal float4 myPrivateMethod()
      { /* ... */ }
    private float4 m_someVar;

}
```

Begin your module definition with `module`

Use `public` declarations to create types, methods, and functions visible to importing modules

`internal` declarations are only visible inside the current module.

If no visibility modifier is provided, all declarations default to `internal`

`private` declarations are only visible inside the current *type*

Modules: Organizing Your Code

```slang
// material.slang
module material;

public struct Material {
    public float4 evalBRDF(float3)
      { /* ... */ }
    internal float4 myPrivateMethod()
      { /* ... */ }
}
```

```slang
// scene.slang
import material;

struct Scene {
    StructuredBuffer<Material> materials;

    void compute(float3 wi, float3 wo) {
        float4 result =
                materials[0].evalBRDF(wi, wo);

        materials[0].myPrivateMethod();
    }
}
```

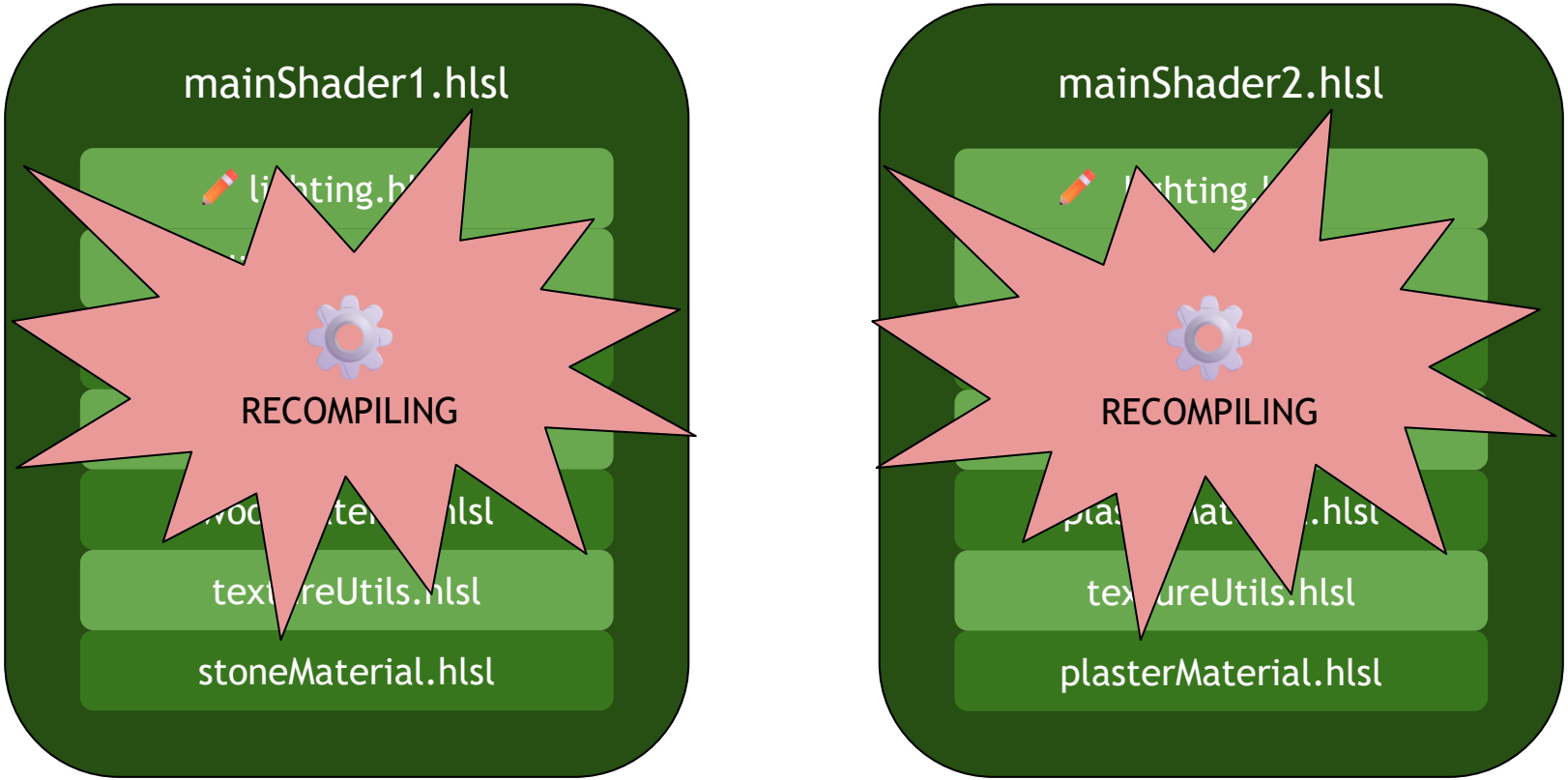
Pull in modules with `import`

Visibility enforcement as you type

ERROR: ‘myPrivateMethod’ is not accessible from the current context

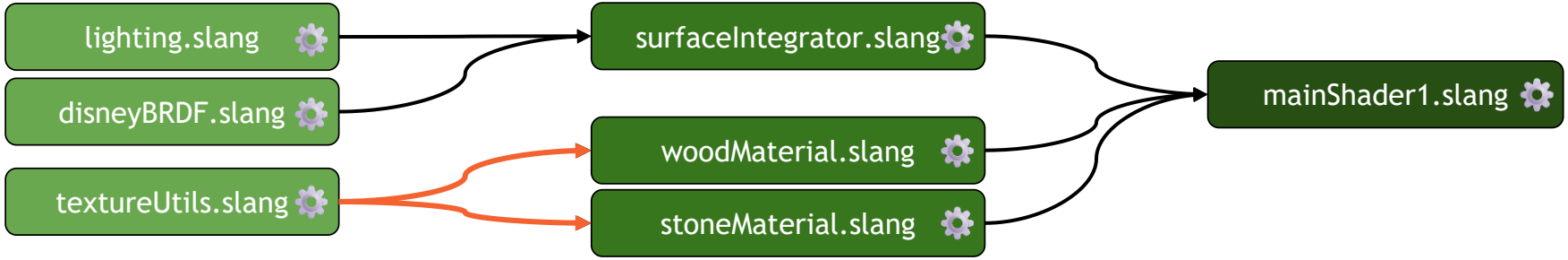
Modules: Streamline Compilation

`#include`-style with HLSL, GLSL, MSL



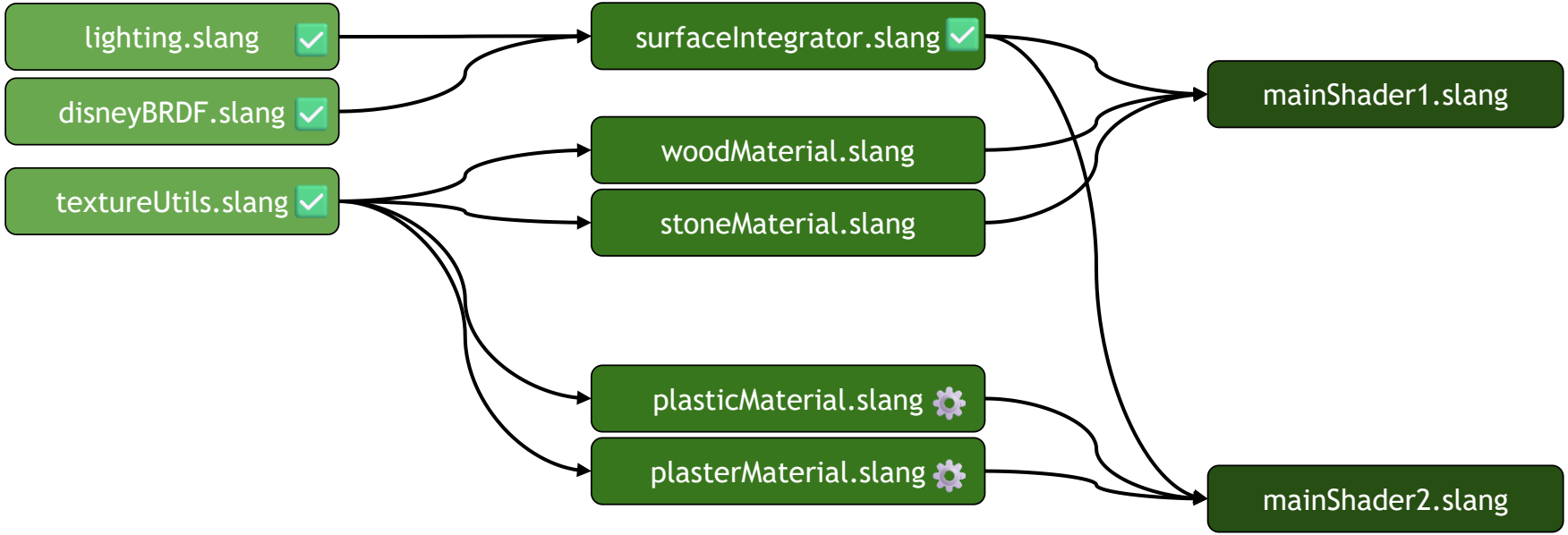
Modules: Streamline Compilation

`import`-style with Slang



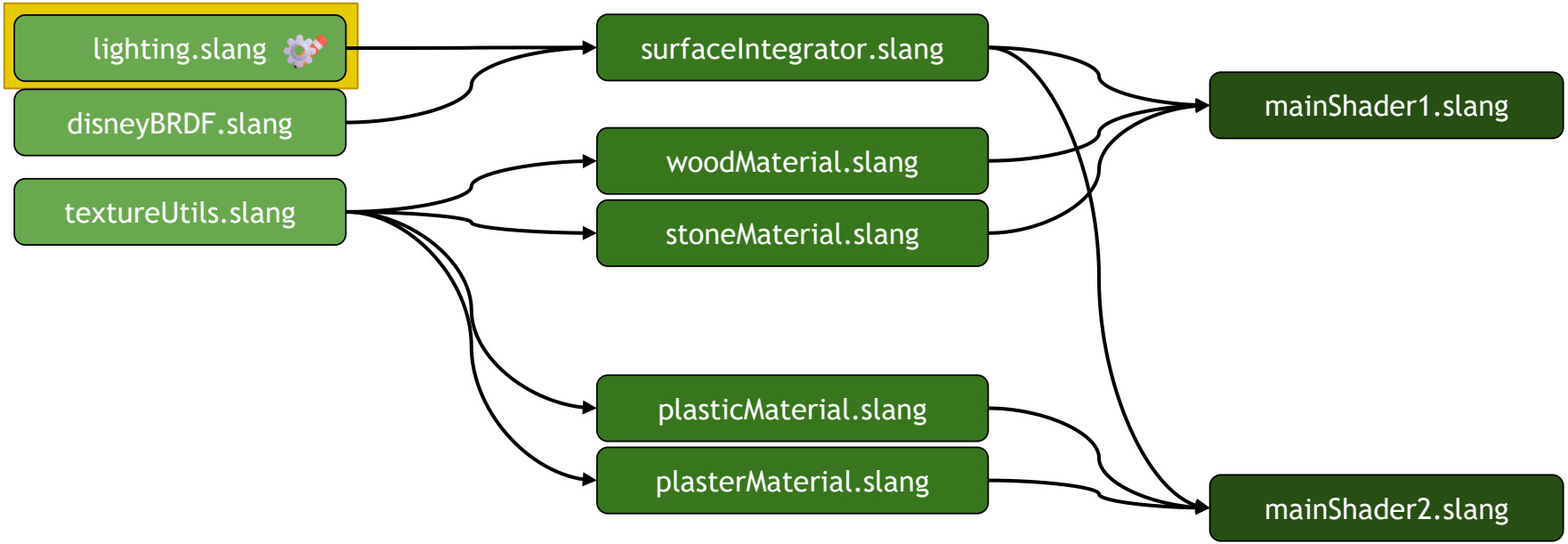
Modules: Streamline Compilation

`import`-style with Slang



Modules: Streamline Compilation

`import`-style with Slang



Specialization with Interfaces & Generics

Interfaces

- Define a contract for behavior
- Swap implementations without macros or copy/paste
- Enforce correctness at compile time

Generics

- Reuse algorithms over a constraint
- Specialize to concrete code per use

```
public interface IBrdf {
   // Required method for evaluating the BRDF
   float3 eval(float3 n, float3 v, float3 l, float3 a);
}
public struct Lambertian : IBrdf {
   float3 eval(float3 n, float3 v, float3 l, float3 a) {
       // ... (implementation details)
   }
}
```

```
public float3 shade<T : IBrdf>(T b, float3 n, float3 v,
                               float3 l, float3 a) {
   return b.eval(n, v, l, a);
}
```

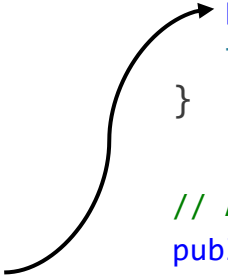
```
float3 c1 = shade(Lambertian(), n, v, l, albedo);
float3 c2 = shade(GGX(),        n, v, l, albedo);
```

Fine-Grained Module API Control

Properties

- Typed, declarative knobs (instead of `#define` toggles)
- Expose defaults – callers set them explicitly

Interfaces can make properties required



```
// material.slang
module material;

public interface IMaterial {
    property float roughness { get; }
    float4 evalBRDF(float3 n, float3 v, float3 l);
}


// A simple implementation of the interface
public struct SimpleMaterial : IMaterial {
    public property float roughness { get { return 0.8; } }
    public float4 evalBRDF(float3 n, float3 v, float3 l)
        { /* ... */ }
}
```

Fine-Grained Module API Control

Extensions

- Add methods to existing types without editing original code
- Great for optional features in separate packages

```
// material.slang
module material;

public interface IMaterial {
   property float roughness { get; }
   float4 evalBRDF(float3 n, float3 v, float3 l);
}
```

```
// In some other module…
import material;
extension IMaterial
{
   public float3 calcRefract(float3 view, float3 normal)
      { /* ... */ }
}
```

Presence Patterns

`Optional<T>`

- Runtime presence/absence with explicit checks
- Replaces sentinels/bitflags

```
import material;
float3 shadeObject(
    MyObject obj,
    Optional<IMaterial> overrideMaterial) {
    IMaterial matToUse;

    if (overrideMaterial.hasValue) {
        matToUse = overrideMaterial.value;
    }
    else {
        matToUse = obj.getDefaultMaterial();
    }

    return matToUse.evalBRDF(...);
}
```

Presence Patterns

`Conditional<T, bool flag>`

- Compile-time presence/absence; strips storage and code when false
- `public consts` act as specialization params: use a source default or provide a value when compiling

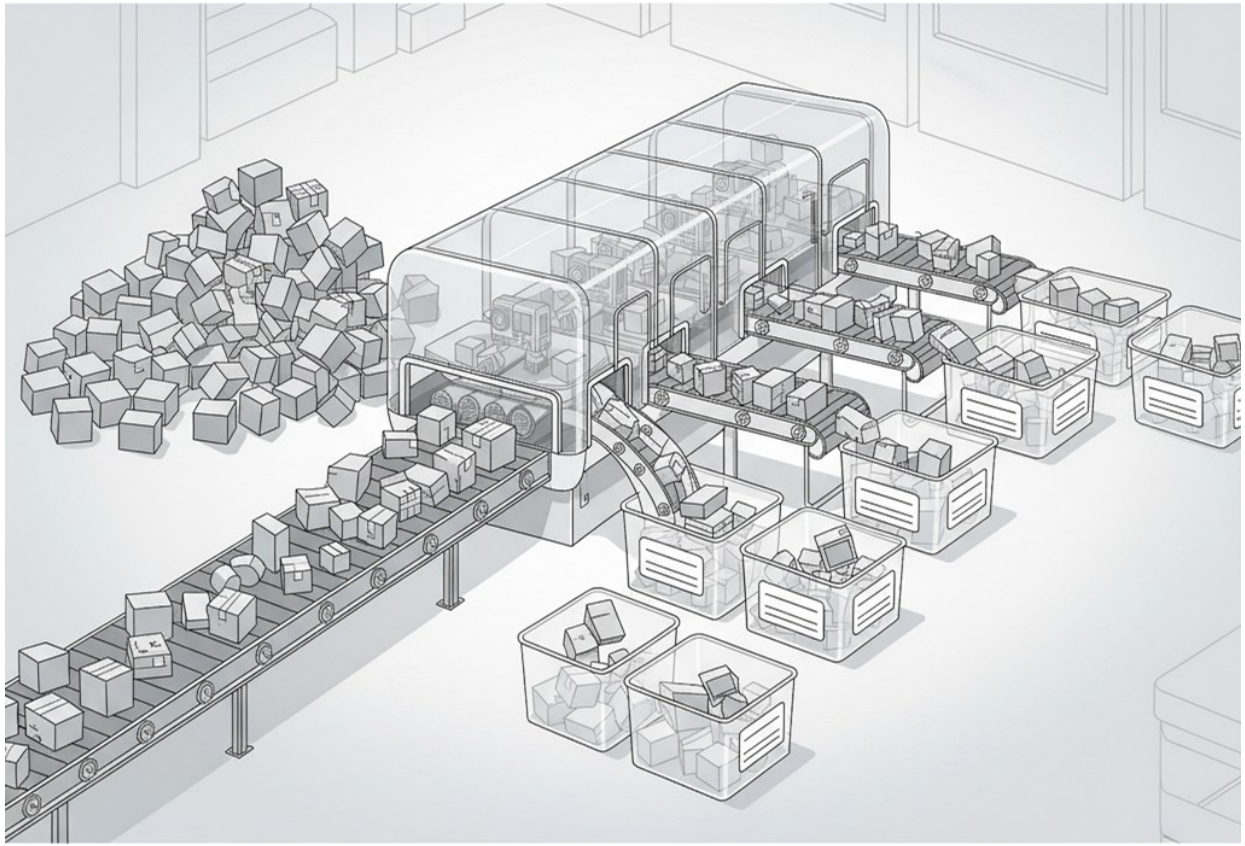
`fullMaterial` contains `m_albedoMap` and `m_normalMap`

`simpleMaterial` contains only `m_normalMap`

```
public struct SimpleMaterial<bool hasNormalMap,
                             bool hasAlbedoMap> {
   Conditional<Texture2D, hasAlbedoMap>
m_albedoMap;
   Conditional<Texture2D, hasNormalMap>
m_normalMap;
}

// in some other file...
StandardMaterial<true, true> fullMaterial;
StandardMaterial<false, true> simpleMaterial;
```

Shader Parameters & Binding Complexity



Manual Slots vs. Logical Reflection

HLSL

```
// myShader_v2.hlsl
[[vk::binding(3, 2)]]
Texture2D g_normalMap  : register(t5, space2);
[[vk::binding(2, 2)]]
SamplerState g_sampler : register(s3, space2);
```

```
LoadShader("MyShader_v1.hlsl");

myCmdList->SetTexture(1, g_normalMapTexture);
myCmdList->SetSampler(0, g_mySampler);
```

Engine-side code needs to know the binding points for shader parameters

And breaks if changes aren't made in parallel

Slang

```
// myShader.slang
struct MyScene
{
   Texture2D g_normalMap;
   SamplerState g_sampler;
};

ParameterBlock<MyScene> g_scene;
```

```
layout = myProgram->getLayout();

normalVar =
    layout->findEntryByName("g_scene.g_normalMap");
samplerVar =
    layout->findEntryByName("g_scene.g_sampler");


myCmdList->BindParameter(normalVar, g_normalMapTexture);
myCmdList->BindParameter(samplerVar, g_mySampler);
```

Manual Slots vs. Logical Reflection

HLSL

```
// myShader_v2.hlsl
[[vk::binding(3, 2)]]
Texture2D g_normalMap  : register(t5, space2);
[[vk::binding(2, 2)]]
SamplerState g_sampler : register(s3, space2);
```

```
LoadShader("MyShader_v1.hlsl");

myCmdList->SetTexture(1, g_normalMapTexture);
myCmdList->SetSampler(0, g_mySampler);
```

Slang

```
// myShader.slang
struct MyScene
{
   Texture2D g_normalMap;
   SamplerState g_sampler;
};

ParameterBlock<MyScene> g_scene;
```

```
layout = myProgram->getLayout();

normalVar =
    layout->findEntryByName("g_scene.g_normalMap");
samplerVar =
    layout->findEntryByName("g_scene.g_sampler");


myCmdList->BindParameter(normalVar, g_normalMapTexture);
myCmdList->BindParameter(samplerVar, g_mySampler);
```

Manual Slots vs. Logical Reflection

Group parameters logically in a struct

Wrap in a ParameterBlock

Host-side code queries binding by **name**

Slang

```
// myShader.slang
struct MyScene
{
    Texture2D g_normalMap;
    SamplerState g_sampler;
};

ParameterBlock<MyScene> g_scene;
```

```
layout = myProgram->getLayout();

normalVar =
    layout->findEntryByName("g_scene.g_normalMap");
samplerVar =
    layout->findEntryByName("g_scene.g_sampler");


myCmdList->BindParameter(normalVar, g_normalMapTexture);
myCmdList->BindParameter(samplerVar, g_mySampler);
```

A Unified, Composable System

Use a struct to implement an interface

```
public interface IBrdf {
    float3 eval(float3 n, float3 v, float3 l);
}

public struct Lambertian : IBrdf {
    Texture2D g_albedoMap;
    float3 g_tint;

    public float3 eval(float3 n, float3 v, float3 l) {
        float3 albedo = g_albedoMap.Sample(...) * g_tint;
        return albedo / 3.14159;
    }
}
```

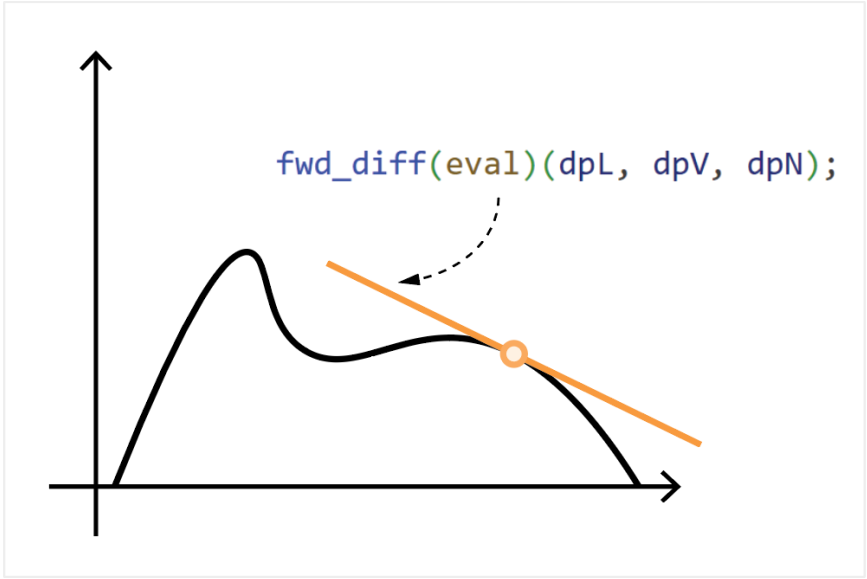
...and also in your `ParameterBlock`

```
ParameterBlock<Lambertian> g_lambertianMaterial;
```

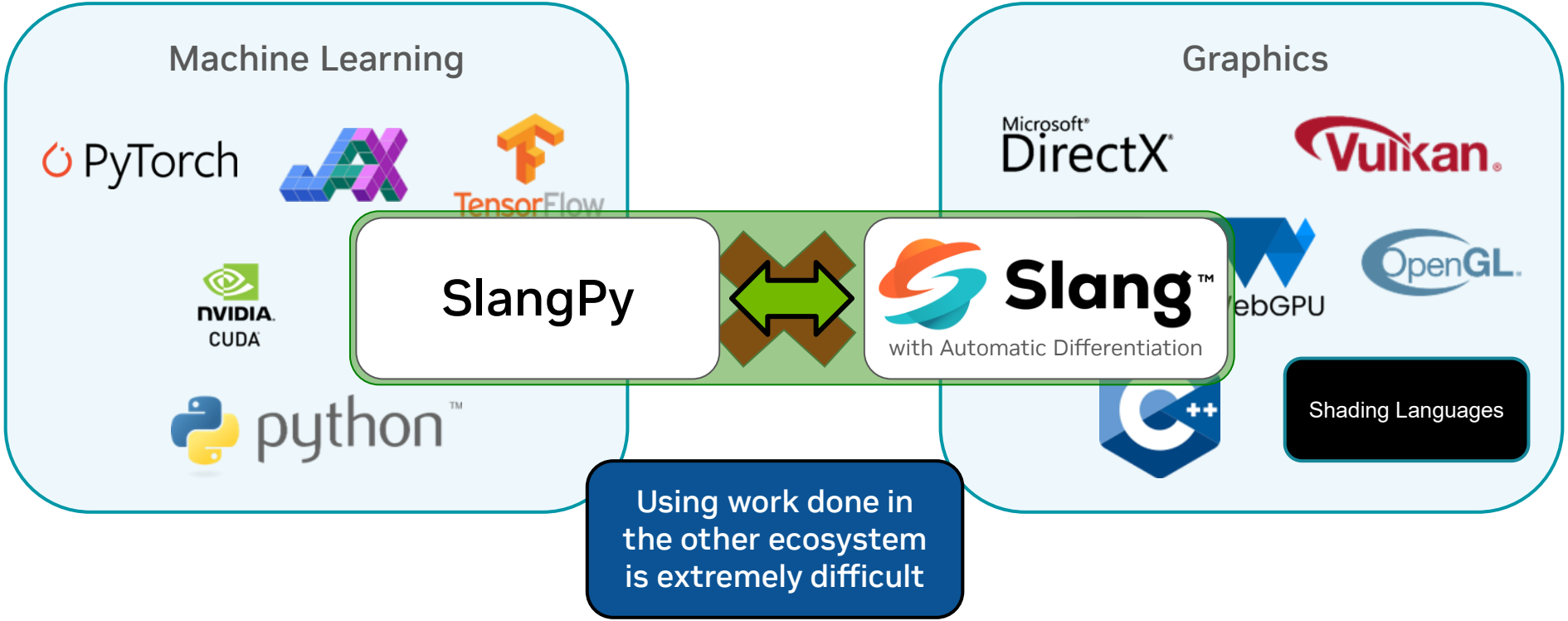
Automatic Differentiation

Your secret weapon for neural graphics

- Automatically compute exact derivatives for any function
- No manual gradient derivation required
- Supports arbitrary control flow and dynamic dispatch
- Enables any graphics function to become trainable



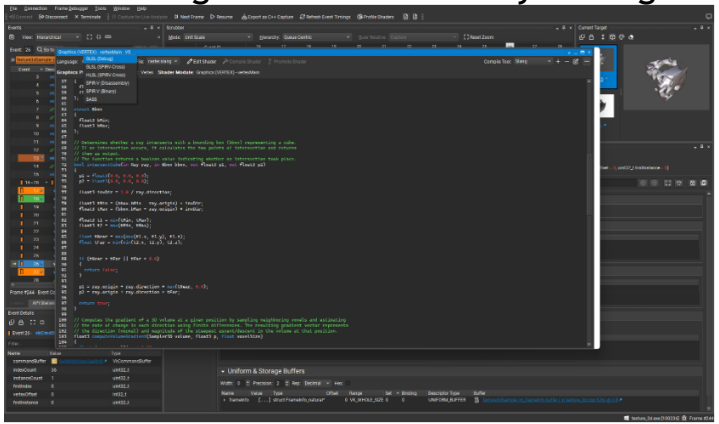
SlangPy


Machine Learning
PyTorch
TensorFlow
NVIDIA
CUDA
python
Graphics
Microsoft
DirectX
Vulkan
OpenGL
WebGPU
SlangPy
Slang™
with Automatic Differentiation
Shading Languages
Using work done in the other ecosystem is extremely difficult

Slang Works With Your Favorite Tools

RenderDoc step-through debugging

Follow Slang->SPIR-V->assembly in Nsight



Visual Studio & VSCode Extensions and Language Server

D: > test.slang > computeMain

```
1   struct MyType
2   {
3       /** returns a value.
4       */
5       int getValue(int val) {return val * 2;}
6   }
7
8   void computeMain(uint3 threadIdx)
9   {
10
11  }
```

Resources

Find all these details and more at shader-slang.org

Try it out in your browser

try.shader-slang.org

Watch Slang Videos

khr.io/slangvideos

Dive into docs

docs.shader-slang.org

Check out SlangPy

slangpy.shader-slang.org

Watch the Neural Shading Course and other SIGGRAPH Materials

Join the Discord