Vulkanised 2024

The 6th Vulkan Developer Conference
Sunnyvale, California | February 5-7, 2024

TOWARD A NEXT-GEN VULKAN SHADING LANGUAGE: OUR JOURNEY WITH SLANG

Theresa “Tess” Foley
NVIDIA

WHO AM I?

~20 years experience working on GPU programming models

- Early “GPGPU” research @ Stanford (BrookGPU project)
- Contributor on CUDA 1.0
- Ph.D. @ Stanford: Spark shading language
- Neoptica -> Intel (Larrabee) -> NVIDIA Research
- Contributor on 1.0 specs for OpenCL, SPIR-V, and Vulkan

- Currently tech lead and manager for the Slang shading language @ NVIDIA
    - Ensuring that our real-time rendering developers have the best tools possible, to unblock innovation

VULKAN HAS A SHADING LANGUAGE PROBLEM

Key challenges not being addressed by GLSL and alternatives

- Increasing scale and complexity of shader codebases
  - Languages originally designed for 10s of lines of code are being used for 10s of **thousands** of lines of code
- Rapid pace of Vulkan API and SPIR-V extensions / innovation
  - No time to evolve language fundamentals – there is always a new extension that needs to be added
- The “shader combinatorics” problem
  - Numbers of compiled variants affects compile times, binary sizes, load times, etc.
  - Persistent problem for entire real-time rendering industry
- Readiness future of machine learning and graphics

GLSL EVOLUTION HAS STALLED

- For Vulkan 1.0, major changes to shading language were out of scope
  - Vulkan support retrofitted onto existing GLSL language
  - SPIR-V was intended to free Vulkan Working Group from language design responsibilities
- Chicken-and-egg problem
  - Developers won't adopt new language features unless they work "everywhere" and provide immediate benefit
  - IHVs incentivized to add features that expose exciting new hardware/API capabilities
- Slang was created, and continues to evolve, to address these challenges head-on

SLANG: A CROSS-PLATFORM SHADING LANGUAGE/COMPILER

Fast-moving language for the future of real-time rendering

Slang

CPU

OpenGL®

NVIDIA® CUDA®

Microsoft® DirectX®

Vulkan®

NVIDIA® OPTIX™

- **Open Source**
- Backwards-compatible with
    - HLSL 2020
    - GLSL (coming soon)

github.com/shader-slang/slang

Coming soon to:

WebGPU

SLANG: MODERN SOFTWARE ENGINEERING FOR GRAPHICS

Advanced type system inspired by Rust/Swift/C#

- Modules and visibility control
- Generics and interfaces
- Associated types and associated constants
- Namespaces, properties, extensions, operator overloading, automatic type inference…

MODULES

Provide separation compilation and control over visibility

material.slang ×

material.slang > Material > somePrivateMethod

```
module material;

public struct Material
{
    public float4 evalBRDF(float3 wi, float3 wo)
    {
        // ...
    }
    internal float4 somePrivateMethod()
    {
        // ...
    }
}
```

scene.slang 1 ×

scene.slang > Scene > compute

```
1   module scene;
2
3   import material;
4
5   struct Scene
6   {
7       StructuredBuffer<Material> materials;
8
9       void compute(float3 wi, float3 wo)
10      {
11          float4 result = materials[0].evalBRDF(wi, wo);
12          materials[0].somePrivateMethod();
13      }
14  }
15
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

scene.slang 1

'somePrivateMethod' is not accessible from the current context. (30600) [Ln 12, Col 22]

INTERFACES

Make requirements explicit

```
interface IMaterial
{
    associatedtype BRDF : IBRDF;
    BRDF sampleAt(SurfacePoint p);
}

interface IBRDF
{
     float4 evaluate(float3 lightDir, float3 eyeDir);
}

interface IGeometry { /*...*/ }
interface ILighting { /*...*/ }
```

You might already be familiar with:

Rust traits
Swift protocols
Haskell typeclasses
…

GENERICS

Improve code maintainability, diagnostics

test.slang 2 ●

test.slang > computeLighting

```
interface IMaterial
{
    float4 evalBRDF(float3 wi, float3 wo);
}

float4 computeLighting<M:IMaterial>(M material, float3 lightPos)
{
    
}

```

test.slang > computeLighting

```
interface IMaterial
{
    float4 evalBRDF(float3 wi, float3 wo);
}

float4 computeLighting<M:IMaterial>(M material, float3 lightPos)
{
    material.methodThatDoesNotExist();
}
```

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS

test.slang 1

'methodThatDoesNotExist' is not a member of 'M'. (30027) [Ln 8, Col 14]

- Early type checking prevents mistakes as code are being written
  - Never lose track of what a generic type has to offer
- Allows Intellisense to provide accurate assistance
- Faster front-end compilation time from reusing type checking results for generic functions

GENERICS ARE TRANSLATED TO EFFICIENT CODE

Start as a more maintainable alternative to preprocessor specialization

Slang w/ Generics

```
// IlluminationPass.slang

float4 shadeSurface<M : IMaterial>(M material)
{
    ...

    M.BRDF b = material.sampleAt(p);

    ...
}
```

same performance

HLSL w/ Preprocessor

```
// IlluminationPass.hlsl

float4 shadeSurface()
{
    ...

#if USE_METAL_MATERIAL
    MetalBRDF b = sampleMetalMaterial(p);
#elif USE_CLOTH_MATERIAL
    ClothBRDF b = sampleClothMaterial(p);
#elif ...
    ...
#endif

    ...
}
```

APPLICATION CAN CONTROL CODE GENERATION

Same shader code yields kernels with different trade-offs

```
// IlluminationPass.slang

float4 shadeSurface<M : IMaterial>(M material)
{
    ...

    M.BRDF b = material.sampleAt(p);

    ...
}
interface IMaterial
{
   associatedtype BRDF : IBRDF;
   BRDF sampleAt(SurfacePoint p);
}

struct MetalMaterial : IMaterial
{ ... }

...
```

```
void shadeSurface_staticallySpecialized(MetalMaterial material)
{
   ...
   MetalBRDF b = MetalMaterial_sampleAt(material, p);
}
```

statically specialized

APPLICATION CAN CONTROL CODE GENERATION

Same shader code yields kernels with different trade-offs

```
// IlluminationPass.slang

float4 shadeSurface<M : IMaterial>(M material)
{
    ...

    M.BRDF b = material.sampleAt(p);

    ...
}
interface IMaterial
{
   associatedtype BRDF : IBRDF;
   BRDF sampleAt(SurfacePoint p);
}

struct MetalMaterial : IMaterial
{ ... }

...
```

```
void shadeSurface_staticallySpecialized(MetalMaterial material)
{
   ...
   MetalBRDF b = MetalMaterial_sampleAt(material, p);
}
```

statically specialized

```
void shadeSurface_uberKernel(TaggedUnion material)
{
   ...
   switch(material.tag) {
   case 0: b = MetalMateral_sampleAt(material.payload, p);
   case 1: ...
   }
}
```

tagged-union-based “uber-kernel”

APPLICATION CAN CONTROL CODE GENERATION

Same shader code yields kernels with different trade-offs

```
// IlluminationPass.slang

float4 shadeSurface<M : IMaterial>(M material)
{
    ...

    M.BRDF b = material.sampleAt(p);

    ...
}
interface IMaterial
{
   associatedtype BRDF : IBRDF;
   BRDF sampleAt(SurfacePoint p);
}

struct MetalMaterial : IMaterial
{ ... }

...
```

```
void shadeSurface_staticallySpecialized(MetalMaterial material)
{
   ...
   MetalBRDF b = MetalMaterial_sampleAt(material, p);
}
```

statically specialized

```
void shadeSurface_uberKernel(TaggedUnion material)
{
   ...
   switch(material.tag) {
   case 0: b = MetalMateral_sampleAt(material.payload, p);
   case 1: ...
   }
}
```

tagged-union-based “uber-kernel”

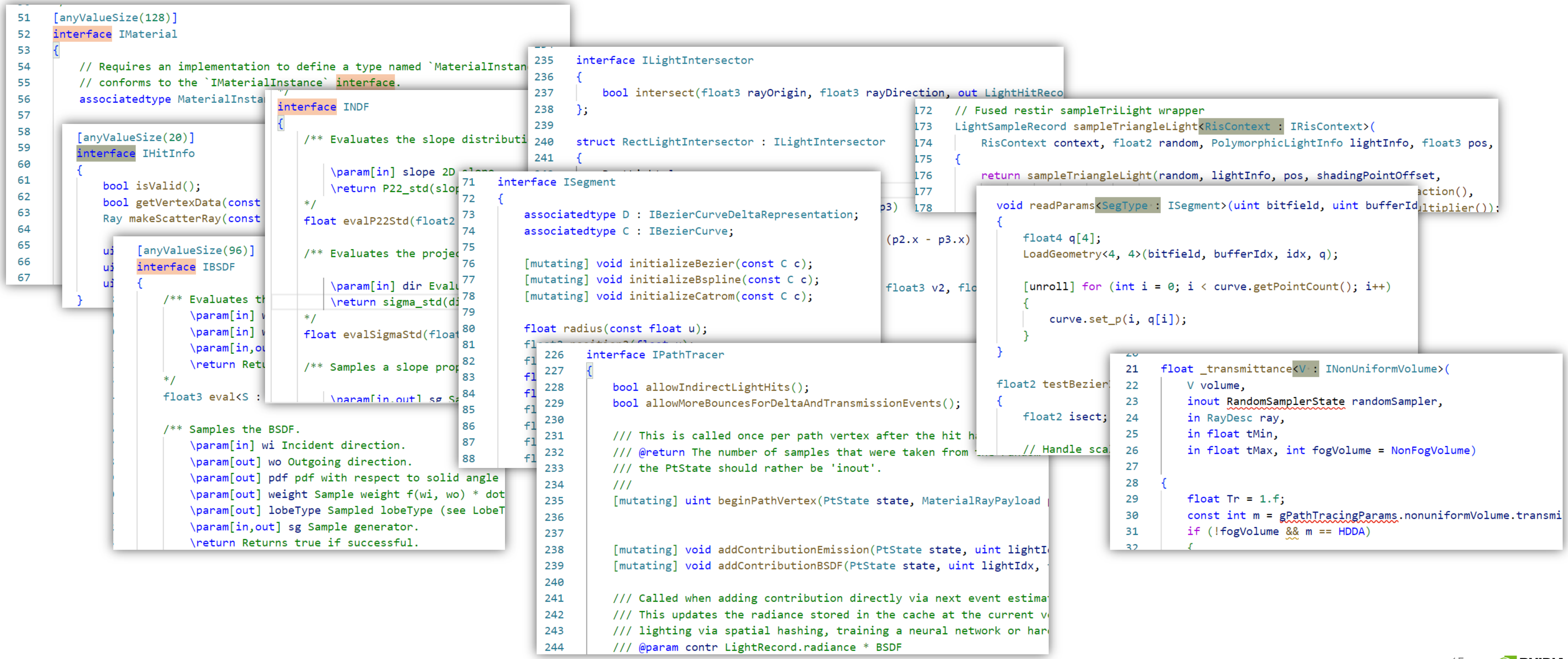
```
void shadeSurface_dynamicDispatch(void** vtbl, void* material)
{
   ...
   b = ((SampleAtFuncPtr)vtbl[0])(material, p);
}
```

dynamic / indirect dispatch
for platforms that support function pointers (CPU, CUDA, Metal, ...)

NVIDIA RENDERING INFRASTRUCTURE USES SLANG

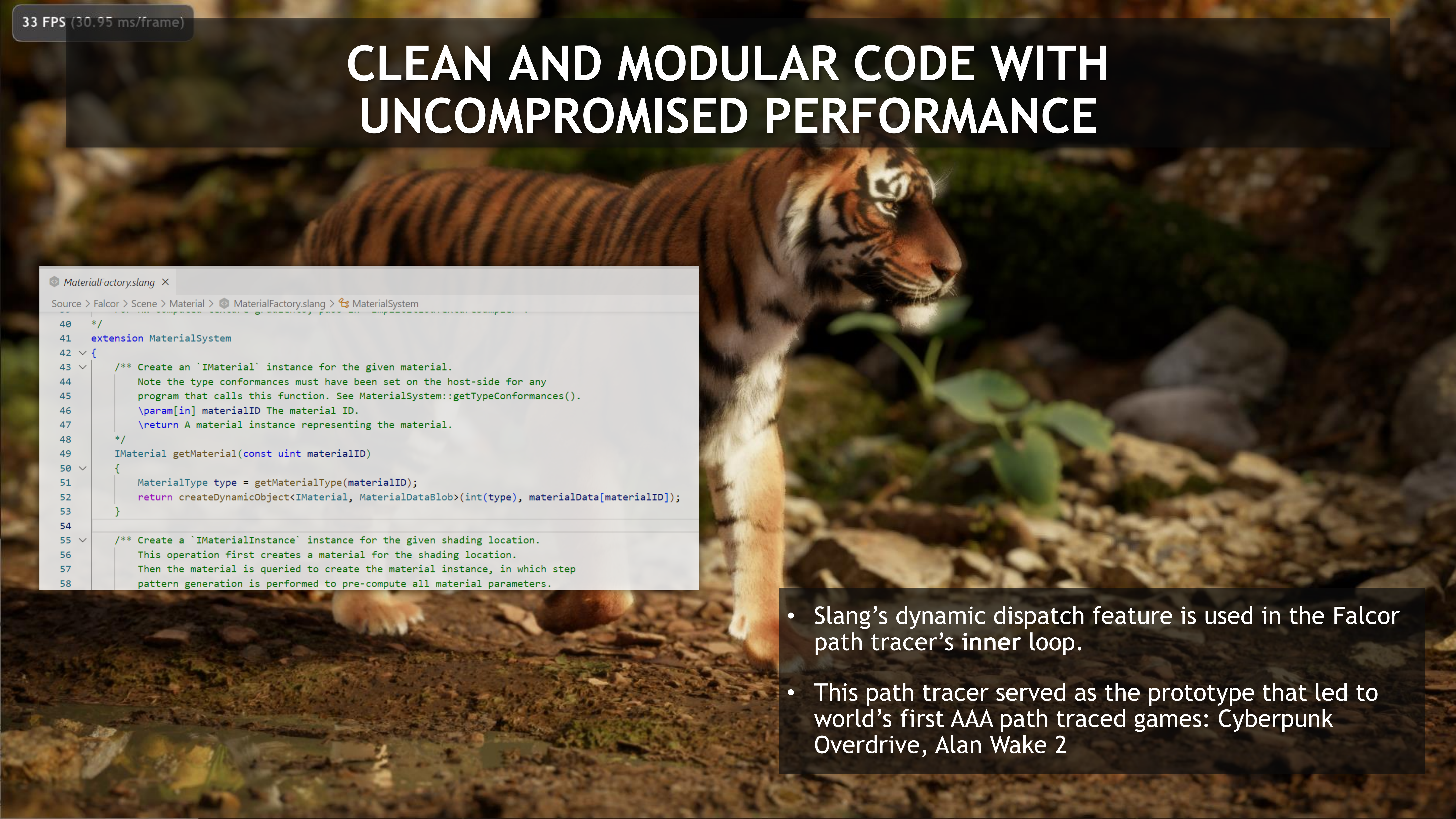
- Omniverse
  - RTX Renderer for real-time path tracing
  - Omniverse Kit application
- Portal RTX
- RTX Remix
- Falcor research rendering framework

NVIDIA PRODUCTS USE SLANG TO KEEP LARGE AND COMPLEX RENDERING CODEBASES MAINTAINABLE



NVIDIA

CLEAN AND MODULAR CODE WITH UNCOMPROMISED PERFORMANCE



MaterialFactory.slang

Source > Falcor > Scene > Material > MaterialFactory.slang > MaterialSystem

```
*/
extension MaterialSystem
{
    /** Create an `IMaterial` instance for the given material.
        Note the type conformances must have been set on the host-side for any
        program that calls this function. See MaterialSystem::getTypeConformances().
        \param[in] materialID The material ID.
        \return A material instance representing the material.
    */
    IMaterial getMaterial(const uint materialID)
    {
        MaterialType type = getMaterialType(materialID);
        return createDynamicObject<IMaterial, MaterialDataBlob>(int(type), materialData[materialID]);
    }

    /** Create a `IMaterialInstance` instance for the given shading location.
        This operation first creates a material for the shading location.
        Then the material is queried to create the material instance, in which step
        pattern generation is performed to pre-compute all material parameters.
```

- Slang's dynamic dispatch feature is used in the Falcor path tracer's **inner** loop.
- This path tracer served as the prototype that led to world's first AAA path traced games: Cyberpunk Overdrive, Alan Wake 2

GRAPHICS DEVELOPERS NEED MACHINE LEARNING

Integrated into their shading language

- "Learning" == gradient-based optimization
- Powerful tool for solving many hard problems in graphics
  - LOD (geometry, texture, material…)
  - Compression
  - Approximation (shader LOD)
  - Parameter tuning (lighting, post-fx, …)

- Graphics devs would rather write code in shading language than typical ML frameworks
  - Control-flow-divergent kernels do not map well to bulk synchronous "tensor" ops of PyTorch etc.

SLANG HAS FIRST CLASS AUTOMATIC DIFFERENTIATION

Works with existing shader codebases in HLSL (and soon GLSL)

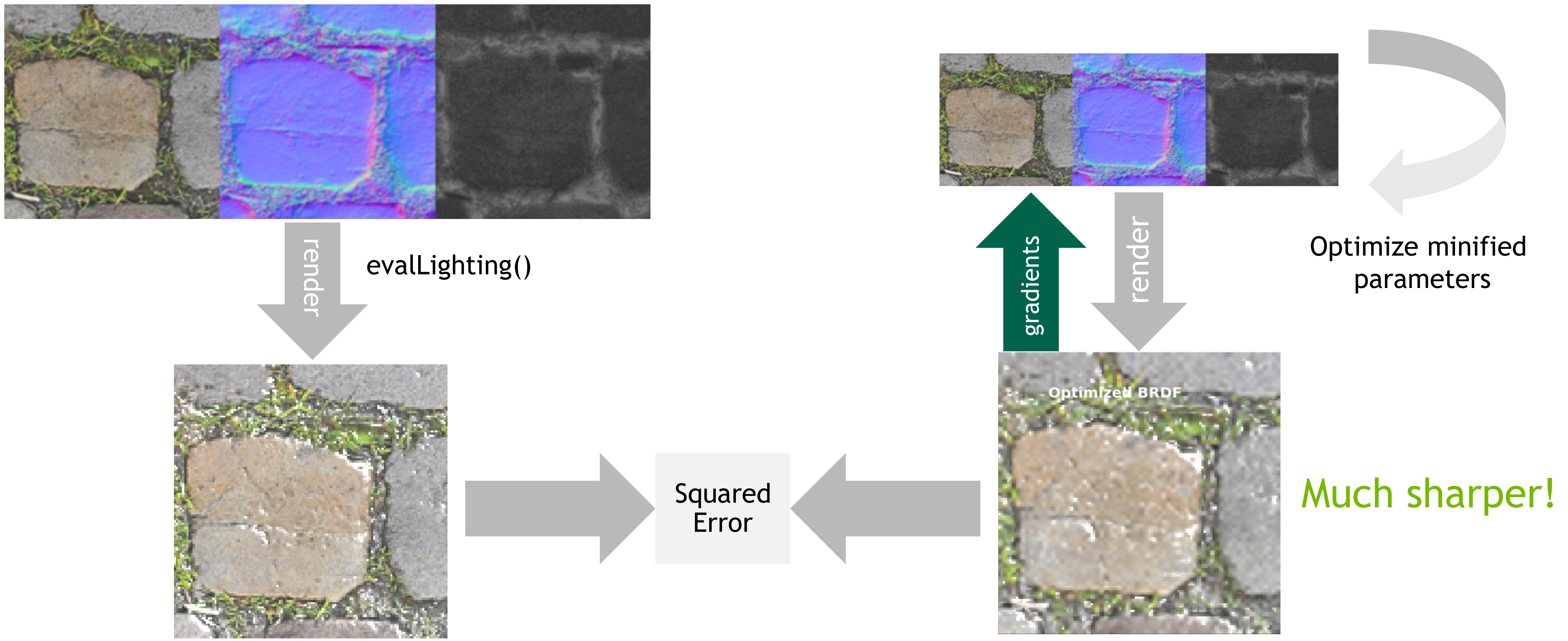
```
// Annotate methods to signal differentiability
[Differentiable]
float square(float x)
{
    return x * x;
}

float3 main()
{
    DifferentialPair<float> dpx = diffPair(4.f);
    // Call the derivative of the 'square' method
    bwd_diff(square)(dpx, 1.f);
    printf("d_square at x=4 is %d", dpx.d);
}
```

AUTODIFF APPLIES TO MORE THAN YOU THINK

- Even traditional graphics problems can benefit
  - Anything you can express as parameter optimization
- Examples
  - Mipmap generation
  - Texture compression

SLANG’S AUTO-DIFF ENABLES CREATIVE SOLUTIONS TO TRADITIONAL PROBLEMS

- **Ex. 1: Better Texture LOD Construction via “learning”**



MIPMAP BUILDER CODE IS MOSTLY TYPICAL SHADER CODE…

with auto-diff decorations, and learning can be driven from Python

BRDF.slang ×

brdf-appearance-optimize-example > BRDF.slang > SchlickFresnel

BRDFs (Slang)

```
[Differentiable]
inline float3 BRDF(float3 L, float3 V, float3 N, float3 baseColor, float roughne
{
    const float subsurface = 0;
    const float specularTint = 0;
    const float anisotropic = 0;
    const float sheen = 0;
    const float sheenTint = .5;
    const float clearcoat = 0;
    const float clearcoatGloss = 1;

    float NdotL = dot(N, L);
    float NdotV = dot(N, V);
    if (NdotL < 0 || NdotV < 0)
        return float3(0);

    float3 H = normalize(L + V);
    float NdotH = dot(N, H);
    float LdotH = dot(L, H);

    float3 Cdlin = baseColor;
    float Cdlum = .3f * Cdlin[0] + .6f * Cdlin[1] + .1f * C
```

Loss Kernel (Slang)

```
[Differentiable]
float loss(DiffTensorView<float> input,
           no_diff float2 uv,
           no_diff float3 reference,
           no_diff Params params)
{
    float3 diff = sqrt(max(computeLighting(input, uv, params), 0.0f
    return dot(diff, diff);
}

[AutoPyBindCUDA]
[CudaKernel]
[Differentiable]
void brdf_loss(DiffTensorView<float> input, DiffTensorView<float>
{
    uint3 globalIdx = cudaBlockIdx() * cudaBlockDim() + cudaThrea

    if (globalIdx.x >= output.size(0) ||
        globalIdx.y >= output.size(1))
        return;

    float2 uv = (globalIdx.xy + 0.5) / float2(output.size(0), out
    float3 referenceVal = float3(reference[int3(globalIdx.xy, 0)]
    float loss = loss(input, uv, referenceVal, PackInputParams(in

    output.store(globalIdx, loss);
}
```

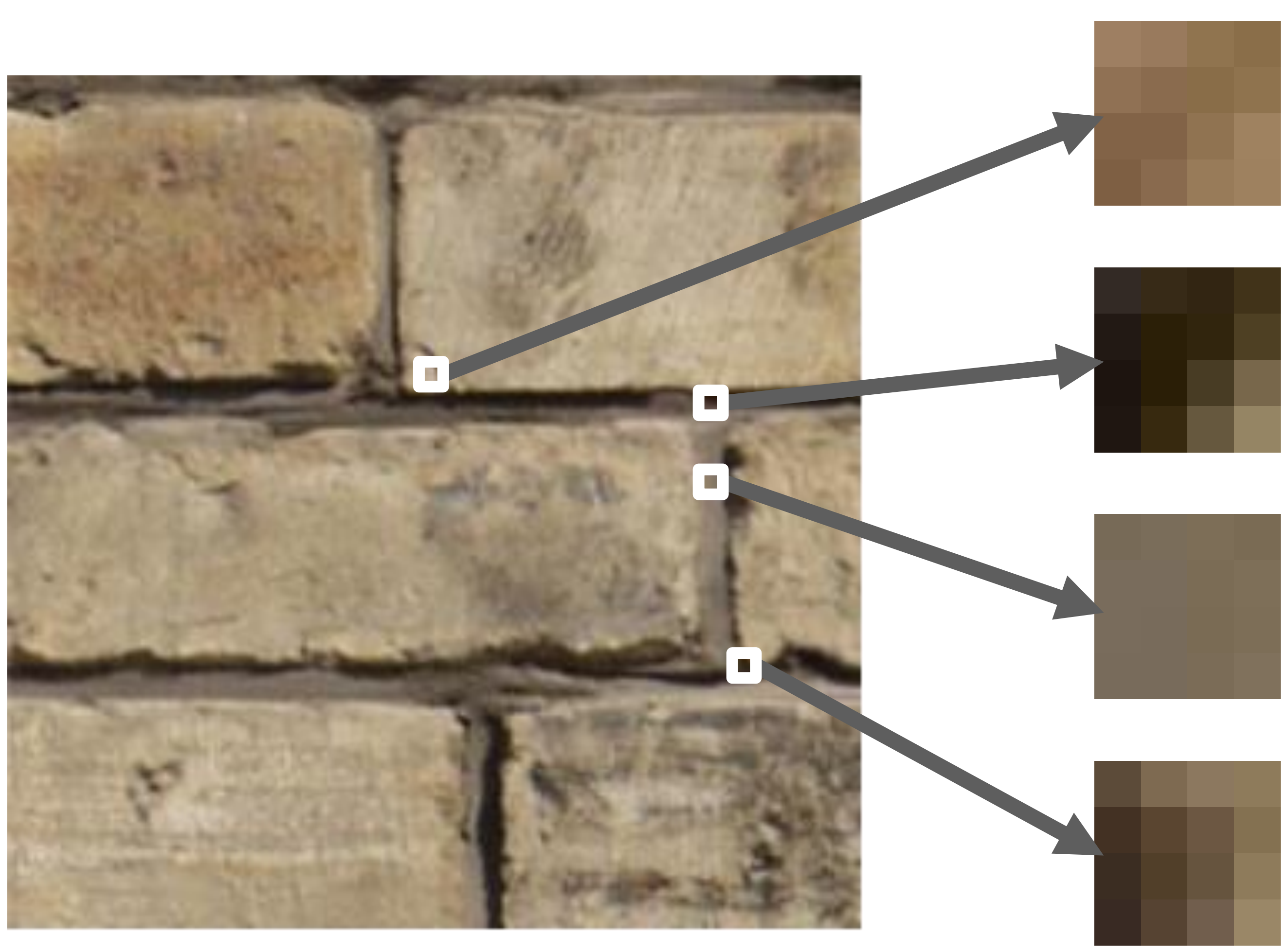
Learning code (Python)

```
full_res_brdf = brdf_input_torch.clone()
# Resize BRDF map to half resolution.
half_res_brdf = torch.tensor(cv2.resize(brdf_input_torch.cpu().numpy(), None, fx=0.5,

# Initialize outputs.
lighting_from_full_res_brdf = torch.zeros(output_shape).cuda()
lighting_from_half_res_brdf = torch.zeros(output_shape).cuda()
gradient_output = torch.zeros_like(half_res_brdf).cuda()
loss_output = torch.zeros((original_shape[0], original_shape[1], 1)).cuda()
output_grad = torch.ones_like(loss_output).cuda()

# Reference and the lighting from half-res resized BRDF.
m.brdf(input=full_res_brdf, output=lighting_from_full_res_brdf, input_params=input_par
m.brdf(input=half_res_brdf, output=lighting_from_half_res_brdf, input_params=input_par

# Backward gradient pass.
m.brdf_loss.bwd(input=(half_res_brdf, gradient_output), output=(loss_output, output_gr

# We want to display it together with outputs, resize.
gradient_resized = cv2.resize(gradient_output.cpu().numpy(), None, fx=2, fy=2)
plt.rcParams['figure.figsize'] = (12, 12)
plt.imshow(display(np.hstack((lighting_from_full_res_brdf.cpu(), lighting_from_half_re
plt.axis("off")
plt.show()
```

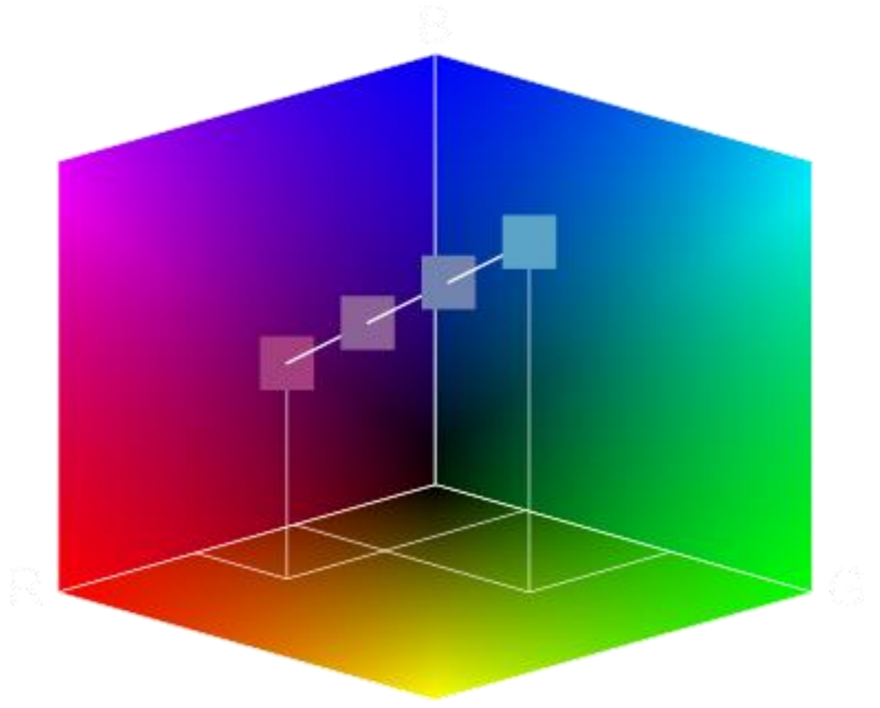
SLANG'S AUTO-DIFF ENABLES CREATIVE SOLUTIONS TO TRADITIONAL PROBLEMS

Ex 2: Use Gradient Descent to Find Block Compression of Textures

Block Compression (BC7-mode6)



For every 4x4 block



1. Find 1 pair of end-points in color-space
2. Linear interpolation coefficient for each texel

Can be framed as an optimization problem

All images from https://www.reedbeta.com/blog/understanding-bcn-texture-compression-formats/

Ex 2: Compressing Textures by “learning”, in 3 Steps

1. Implement a de-compresser

```
TextureBlock decompress(CompressedTextureBlock blockCoefficients)
{
      // Implement BC7 decompression here. Super easy!
}
```

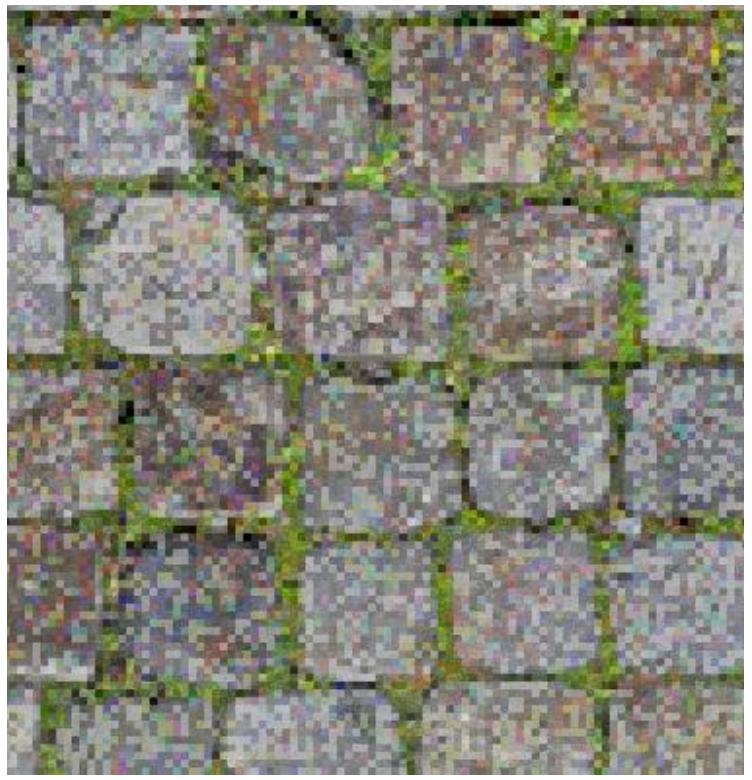
2. Implement a loss function

```
float loss(TextureBlock groundtruth, CompressedTextureBlock compressed)
{
    return distance(decompress(compressed), groundtruth);
}
```

3. Use gradient decent to find the compression coefficients

```
CompressedTextureBlock compress(TextureBlock data)
{
      CompressedTextureBlock result = random_init();
      for (int i = 0; i < N_STEPS; i++) {
          derivative = computeDerivativeOfLoss(result, data);
          result += derivative * learning_rate;
      }
      return result;
}
```

Ex 2: Compressing Textures by “learning”



Initialization

5 Steps

20 Steps

Full optimization loop within a single shader invocation:

4K Texture compressed in **2.4ms** on RTX 4090

FALCOR RENDERER IS NOW DIFFERENTIABLE

github.com/NVIDIAGameWorks/Falcor

- Existing Slang shader codebase developed over 5+ years
  - 5,000+ lines of Slang code
  - Samplers, materials, path-tracers, lighting models, denoisers, geometry models
- Differentiability added with ~200 lines of annotations

RGB image

derivative w.r.t. camera position

Forward Render: **2.4ms**

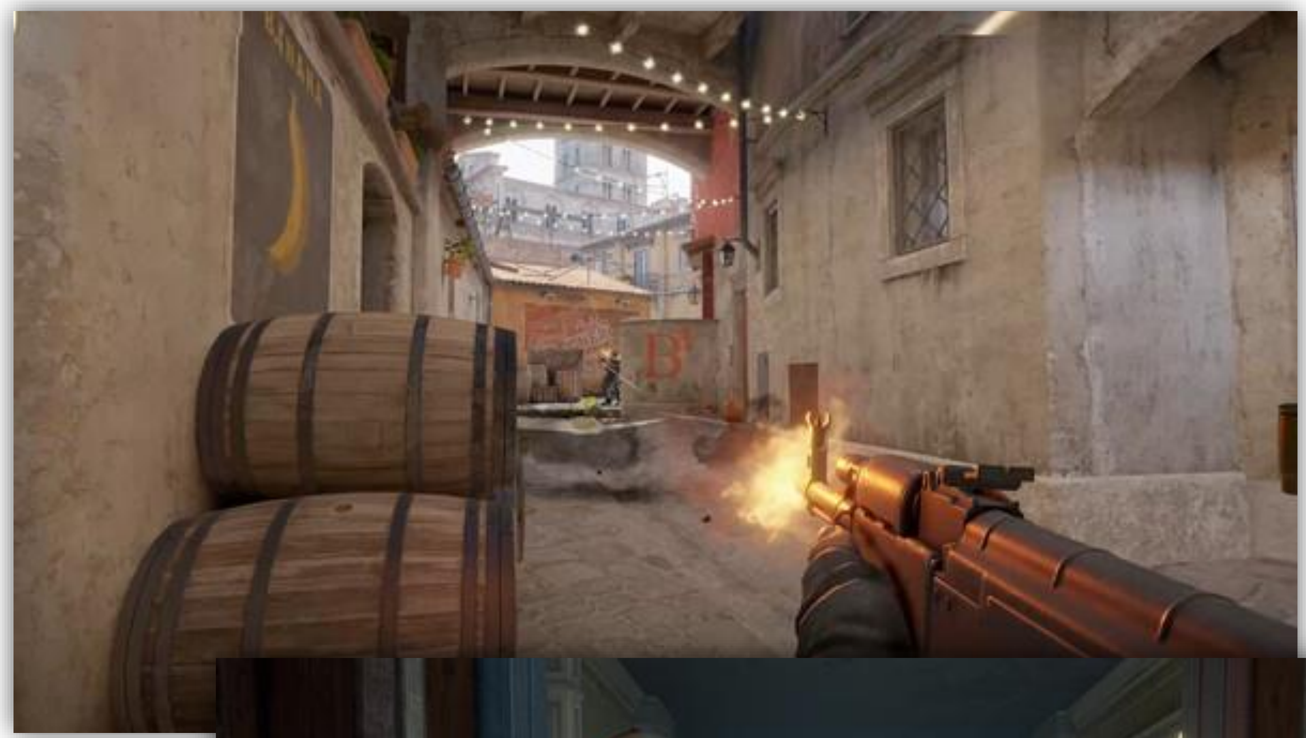
Back-prop Pass: **54ms**

SLANG IS READY FOR THE VULKAN COMMUNITY

SOURCE 2 SHADER CODEBASE MIGRATED TO SLANG

In collaboration with Valve

- Source 2
  - Used by games such as Dota 2, Half Life: Alyx, and Counter Strike 2
  - Target platform: D3D 11, Vulkan, Vulkan Ray Tracing (Hammer Editor)
  - Existing shaders are written in HLSL with Vulkan specific extensions: Raytracing, Subgroup Wave Ops, Bindless
- Shader statistics:
  - ~452 individual materials (.vfx files)
  - > 10 million shader variants
- Current shader compilation pipeline:
  - Vulkan - DXC + spirv-opt + SPIRV-Reflect
  - DX11 – FXC



VALVE ADOPTING SLANG IN SOURCE 2

- Motivation: Access state-of-the-art hardware features sooner via Slang
  - Use Slang's ConstBufferPointer<T> type instead of DXC's vk::RawBufferLoad
  - Plans to leverage more slang-specific features in the future

- Converting entire HLSL codebase to work with Slang
  - All Dota/CS2 shaders rendering correctly (including GPU Path Tracing in Hammer for CS2 Workshop Tools)
  - Existing spirv-opt + SPIRV-Reflect pipeline continues to work

- Minimal shader code changes
  - Changed ~10 lines of code to get everything to compile
  - Fixed additional Slang warnings for better code quality (uninitialized variables etc., to prevent bugs)

- Valve has plans to move Source 2's shader compiler to Slang

SLANG SUPPORTS VULKAN AS A TIER-1 PLATFORM

Levels of Support

Tier 1 (co-evolving)

- Core language features evolving with the platform
- Latest features / extensions are exposed as strongly-typed language constructs/types that are easy to use
- New language features might be available only for the platform, while Slang tries its best to cross-compile to other platforms when possible

Tier 2 (cross-platform coverage)

- Making sure all features that are available on other platforms can be cross-compiled to this platform
- Features specific to this platform are added in a way that leads to minimal change to the language (e.g. as attributes)

FIRST-CLASS LANGUAGE FEATURES FOR VULKAN

Combined Texture Sampler

```
Sampler2D<float4> myCombinedSampler;
void test(float2 uv)
{
    float4 color = myCombinedSampler.Sample(uv);
}
```

Custom UAV Data Layout

```
StructuredBuffer<float3, ScalarDataLayout> scalarBuffer;
StructuredBuffer<float3, Std430DataLayout> std430Buffer;
```

Buffer Pointer

```
struct MyData {
    int flags;
    float values;
    ConstBufferPointer<MyData> pNext;
}
StructuredBuffer<MyData> objects;
void test()
{
    let obj = objects[0].pNext.get().pNext.get();
    let f = obj.flags;
}
```

SSBO

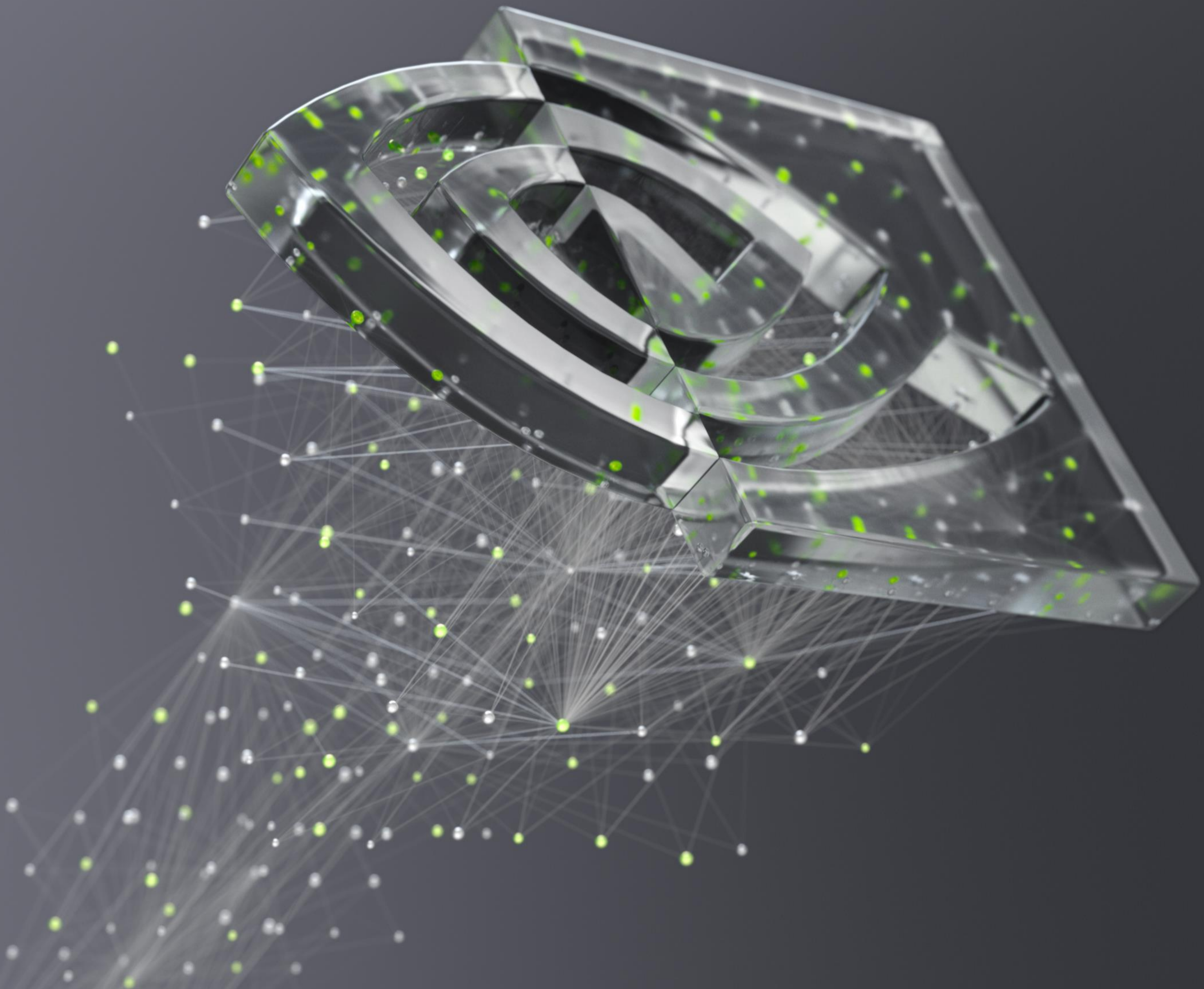
```
struct SSBO {
    int flags;
    float values[];
}
GLSLShaderStorageBuffer<SSBO> ssbo;
void test()
{
    let f = ssbo.flags;
    let v = ssbo.values[0];
}
```

Embedded SPIRV Assembly

```
[numthreads(1,1,1)]
void computeMain(uint3 tid : SV_DispatchThreadID)
{
    float3 localVar = float3(1, 2, 3);
    float3 val = spirv_asm
    {
        %tmp:$$float3 = OpConvertUToF $tid;
        result:$$float3 = OpFAdd %tmp $localVar;
    };
    // val == float3(tid) + localVar
}
```

SLANG IS READY FOR THE VULKAN COMMUNITY

- Fast-moving language built for the future of real-time rendering
  - Enables our in-house Vulkan developers to innovate
  - Continually evolving in collaboration with users
- Supports Vulkan/SPIR-V as a first-class platform
  - Direct SPIR-V code generation
  - First-class language features designed for Vulkan
- Easy path for adoption in existing codebases
  - HLSL/GLSL syntax compatibility
  - Compiles all of Valve's Source 2 shaders
- We want to hear from the Vulkan community
  - What other challenges would you like to see addressed in Slang?
  - Try Slang and give us your feedback!



NVIDIA

WHAT IS SLANG?

github.com/shader-slang/slang

- Open-source shading language and compiler
- Compatible with existing shader codebases
  - HLSL 2020 dialect
  - GLSL 4.x dialect under development

SLANG IS WIDELY USED INSIDE NVIDIA

in both products and research

NVIDIA >

Omniverse

NVIDIA RTX Remix

PORTAL™ WITH RTX

NVIDIA'S SHADING LANGUAGE INNOVATION PLATFORM

github.com/shader-slang/slang

- Gives us the tools needed to build large, maintainable, and high-performance real-time renderers
- Same shader code can be used across multiple platforms and APIs
  - Vulkan (SPIR-V), D3D12 (DXIL), CUDA, Optix, debuggable CPU code (C++)
- Extends HLSL/GLSL with carefully chosen constructs from modern general-purpose languages
  - Proven language constructs from Rust, Swift, C#, etc.
  - Adapted and implemented with focus on GPU performance

KEY CHALLENGES, REVISITED

Let’s look at how Slang addresses them

- Increasing scale and complexity of shader codebases
  - Languages originally designed for 10s of lines of code being used for 10s of **thousands** of lines of code
- Rapid pace of Vulkan API and SPIR-V extensions / innovation
  - No time to evolve language fundamentals – there is always a new extension that needs to be bolted on
- The “shader combinatorics” problem
  - Numbers of compiled variants affects compile times, binary sizes, load times, etc.
  - Persistent problem for entire real-time rendering industry
- AI/ML is a powerful new tool in real-time rendering
  - Metaprogramming-based ML frameworks are a poor fit for shader code

PLUS A BONUS CHALLENGE

Something you may not yet realize you want…

- Increasing scale and complexity of shader codebases
  - Languages originally designed for 10s of lines of code being used for 10s of **thousands** of lines of code
- Rapid pace of Vulkan API and SPIR-V extensions / innovation
  - No time to evolve language fundamentals – there is always a new extension that needs to be bolted on
- The “shader combinatorics” problem
  - Numbers of compiled variants affects compile times, binary sizes, load times, etc.
  - Persistent problem for entire real-time rendering industry
- AI/ML is a powerful new tool in real-time rendering
  - Metaprogramming-based ML frameworks are a poor fit for shader code

KEY CHALLENGES, REVISITED

Let’s look at how Slang addresses them

- Increasing scale and complexity of shader codebases
  - Languages originally designed for 10s of lines of code being used for 10s of **thousands** of lines of code
- Rapid pace of Vulkan API and SPIR-V extensions / innovation
  - No time to evolve language fundamentals – there is always a new extension that needs to be bolted on
- The “shader combinatorics” problem
  - Numbers of compiled variants affects compile times, binary sizes, load times, etc.
  - Persistent problem for entire real-time rendering industry
- AI/ML is a powerful new tool in real-time rendering
  - Metaprogramming-based ML frameworks are a poor fit for shader code

SLANG SCALES TO LARGE SHADER CODEBASES

Adopts well-understood constructs from modern languages

- Modules
- Interfaces
- Generics

MODULES

Provide separation compilation and control over visibility

material.slang ×

material.slang > Material > somePrivateMethod

```
module material;

public struct Material
{
    public float4 evalBRDF(float3 wi, float3 wo)
    {
        // ...
    }
    internal float4 somePrivateMethod()
    {
        // ...
    }
}
```

scene.slang 1 ×

scene.slang > Scene > compute

```
1  module scene;
2
3  import material;
4
5  struct Scene
6  {
7      StructuredBuffer<Material> materials;
8
9      void compute(float3 wi, float3 wo)
10     {
11         float4 result = materials[0].evalBRDF(wi, wo);
12         materials[0].somePrivateMethod();
13     }
14 }
15
```

PROBLEMS 1 | OUTPUT | DEBUG CONSOLE | TERMINAL | PORTS

scene.slang 1

ⓧ 'somePrivateMethod' is not accessible from the current context. (30600) [Ln 12, Col 22]

INTERFACES

Make requirements explicit

```
interface IMaterial
{
    associatedtype BRDF : IBRDF;
    BRDF sampleAt(SurfacePoint p);
}

interface IBRDF
{
    float4 evaluate(float3 lightDir, float3 eyeDir);
}

interface IGeometry { /*...*/ }
interface ILighting { /*...*/ }
```

You might already be familiar with:

Rust traits
Swift protocols
Haskell typeclasses
...

USER-DEFINED TYPE DECLARES CONFORMANCE

Compiler checks that it meets all requirements

```
interface IMaterial
{
   associatedtype BRDF : IBRDF;
   BRDF sampleAt(SurfacePoint p);
}

interface IBRDF
{
    float4 evaluate(float3 lightDir, float3 eyeDir);
}

interface IGeometry { ... }
interface ILighting { ... }
...
```

```
struct MetalMaterial : IMaterial
{
    Texture2D alebdoMap;
    Texture2D roughnessMap;
    float glossiness;

    struct BRDF : IBRDF
    {
        float4 albedo;
        float roughness;
        float glossiness;

        float4 evaluate(float3 lightDir, float3 eyeDir)
        { ... }
    }

    BRDF sampleAt(SurfacePoint p)
    { ... }
}
```

USER-DEFINED TYPE DECLARES CONFORMANCE

Compiler checks that it meets all requirements

```
interface IMaterial
{
   associatedtype BRDF : IBRDF;
   BRDF sampleAt(SurfacePoint p);
}

interface IBRDF
{
    float4 evaluate(float3 lightDir, float3 eyeDir);
}

interface IGeometry { ... }
interface ILighting { ... }
...
```

```
struct MetalMaterial : IMaterial
{
    Texture2D alebdoMap;
    Texture2D roughnessMap;
    float glossiness;

    struct BRDF : IBRDF
    {
        float4 albedo;
        float roughness;
        float glossiness;

        float4 evaluate(float3 lightDir, float3 eyeDir)
        { ... }
    }

    BRDF sampleAt(SurfacePoint p)
    { ... }
}
```

GENERICS

Enable re-usable shader code

Slang

```
// IlluminationPass.slang

float4 shadeSurface<M : IMaterial>(M material)
{
    ...

    M.BRDF b = material.sampleAt(p);

    ...
}
```

same performance

Plain HLSL

```
// IlluminationPass.hlsl

float4 shadeSurface()
{
    ...

#if USE_METAL_MATERIAL
    MetalBRDF b = sampleMetalMaterial(p);
#elif USE_CLOTH_MATERIAL
    ClothBRDF b = sampleClothMaterial(p);
#elif ...
    ...
#endif

    ...
}
```

GENERICS

Enable re-usable shader code

Slang

```
// IlluminationPass.slang

float4 shadeSurface<M : IMaterial>(M material)
{
    ...

    M.BRDF b = material.sampleAt(p);

    ...
}
```

```
float4 shadeSurface(IMaterial material)
{
    ...
    let b = material.sampleAt(p);
    ...
}
```

same performance

same performance

Plain HLSL

```
// IlluminationPass.hlsl

float4 shadeSurface()
{
    ...

#if USE_METAL_MATERIAL
    MetalBRDF b = sampleMetalMaterial(p);
#elif USE_CLOTH_MATERIAL
    ClothBRDF b = sampleClothMaterial(p);
#elif ...
    ...
#endif

    ...
}
```

KEY CHALLENGES, REVISITED

Let’s look at how Slang addresses them

- Increasing scale and complexity of shader codebases
  - Languages originally designed for 10s of lines of code being used for 10s of **thousands** of lines of code
- Rapid pace of Vulkan API and SPIR-V extensions / innovation
  - No time to evolve language fundamentals – there is always a new extension that needs to be bolted on
- The “shader combinatorics” problem
  - Numbers of compiled variants affects compile times, binary sizes, load times, etc.
  - Persistent problem for entire real-time rendering industry
- AI/ML is a powerful new tool in real-time rendering
  - Metaprogramming-based ML frameworks are a poor fit for shader code

KEEPING PACE WITH API EVOLUTION

Slang project design philosophy

- Portable / multi-platform language, but allow for target-specific constructs
  - Don’t reduce everything to lowest common denominator
  - Example: support for inline SPIR-V assembly
- Don’t just keep bolting on more `layout` qualifiers and `[[annotations]]`
  - Step back to consider design alternatives
  - Look for opportunities to leverage existing language constructs
- Example: parameter blocks

PER-API PARAMETER ANNOTATION MESS

Plain HLSL

Vulkan, D3D12

```
// MetalMaterial.hlsl.h

[[vk::binding(0,1)]] Texture2D gAlbedoMap    : register(t0, space1);
[[vk::binding(1,1)]] Texture2D gRoughnessMap : register(t1, space1);
[[vk::binding(2,1)]] cbuffer MaterialParams  : register(b0, space1);
{ float gGlossiness; }
...


float4 shadeSurface()
{
    ...
    float4 albedo = gAlbedoMap.Sample( ... );
    ...
}
```

PER-API PARAMETER ANNOTATION MESS

Plain GLSL

Vulkan

```
// MetalMaterial.glsl.h

layout(binding=0, set=1) texture2D gAlbedoMap;
layout(binding=1, set=1) texture2D gRoughnessMap;
layout(binding=2, set=1) uniform MaterialParams
{ vec4 gGlossiness; };
...


vec4 shadeSurface()
{
    ...
    vec4 albedo = texture(gAlbedoMap, ...);
    ...
}
```

Plain HLSL

Vulkan, D3D12

```
// MetalMaterial.hlsl.h

[[vk::binding(0,1)]] Texture2D gAlbedoMap    : register(t0, space1);
[[vk::binding(1,1)]] Texture2D gRoughnessMap : register(t1, space1);
[[vk::binding(2,1)]] cbuffer MaterialParams  : register(b0, space1);
{ float gGlossiness; }
...


float4 shadeSurface()
{
    ...
    float4 albedo = gAlbedoMap.Sample( ... );
    ...
}
```

PER-API PARAMETER ANNOTATION MESS

Slang

Vulkan, D3D12, D3D11, OpenGL, CUDA, CPU, …

```
struct MetalMaterial
{
    Texture2D albedoMap;
    Texture2D rougnessMap;
    float glossiness;
    ...
}

float4 shadeSurface(
    ParameterBlock<MetalMaterial> material)
{
    ...
    float4 albedo = material.albedoMap.Sample( ... );
    ...
}
```

Plain HLSL

Vulkan, D3D12

```
// MetalMaterial.hlsl.h

[[vk::binding(0,1)]] Texture2D gAlbedoMap    : register(t0, space1);
[[vk::binding(1,1)]] Texture2D gRoughnessMap : register(t1, space1);
[[vk::binding(2,1)]] cbuffer MaterialParams  : register(b0, space1);
{ float gGlossiness; }
...


float4 shadeSurface()
{
    ...
    float4 albedo = gAlbedoMap.Sample( ... );
    ...
}
```

Leverage existing language constructs instead of adding yet more annotations

KEY CHALLENGES, REVISITED

Let’s look at how Slang addresses them

- Increasing scale and complexity of shader codebases
    - Languages originally designed for 10s of lines of code being used for 10s of **thousands** of lines of code
- Rapid pace of Vulkan API and SPIR-V extensions / innovation
    - No time to evolve language fundamentals – there is always a new extension that needs to be bolted on
- The “shader combinatorics” problem
    - Numbers of compiled variants affects compile times, binary sizes, load times, etc.
    - Persistent problem for entire real-time rendering industry
- AI/ML is a powerful new tool in real-time rendering
    - Metaprogramming-based ML frameworks are a poor fit for shader code

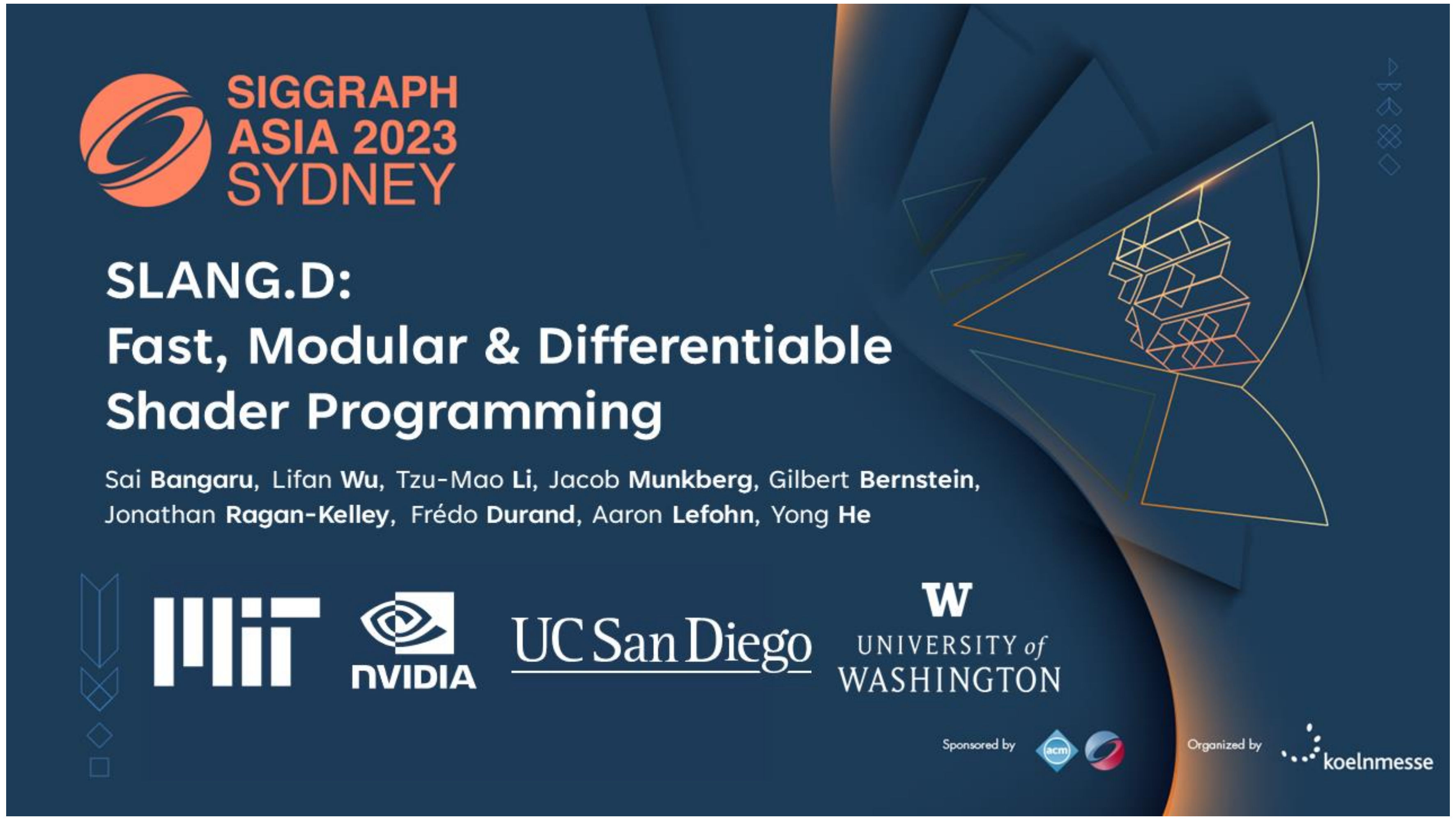
KEY CHALLENGES, REVISITED

Let’s look at how Slang addresses them

- Increasing scale and complexity of shader codebases
    - Languages originally designed for 10s of lines of code being used for 10s of **thousands** of lines of code
- Rapid pace of Vulkan API and SPIR-V extensions / innovation
    - No time to evolve language fundamentals – there is always a new extension that needs to be bolted on
- The “shader combinatorics” problem
    - Numbers of compiled variants affects compile times, binary sizes, load times, etc.
    - Persistent problem for entire real-time rendering industry
- AI/ML is a powerful new tool in real-time rendering
    - Metaprogramming-based ML frameworks are a poor fit for shader code

SLANG IS READY FOR AI-ACCELERATED RENDERING

Automatic differentiation is **built into** the language and compiler



LANGUAGE-INTEGRATED AUTOMATIC DIFFERENTIATION

Works with existing shader codebases in HLSL (and soon GLSL)

```
// Annotate methods to signal differentiability
[Differentiable]
float square(float x)
{
    return x * x;
}



float3 main()
{
    DifferentialPair<float> dpx(4.f);

     // Call the derivative of the ‘square’ method
    bwd_diff(square)(dpx, 1.f);

    printf(“d_square at x=4 is %d”, dpx.d);
}
```

DIFFERENTIABILITY IS UNDERSTOOD BY TYPE SYSTEM

User-defined types can opt in to being differentiable

```
struct MyRay : IDifferentiable
{
    float3 origin;
    float3 dir;
    int nonDifferentiablePayload;
}
```

DIFFERENTIABILITY IS UNDERSTOOD BY TYPE SYSTEM

User-defined types can opt in to being differentiable

```
struct MyRay : IDifferentiable
{
    float3 origin;
    float3 dir;
    int nonDifferentiablePayload;
}
```

compiler generates

```
struct MyRayDifferential
{
    float3 d_origin;
    float3 d_dir;
}
struct MyRay : IDifferentiable
{
    typealias Differential = MyRayDifferential;
    [DerivativeMember(MayRayDifferential.d_origin)] float3 origin;
    [DerivativeMember(MayRayDifferential.d_dir)] float3 dir;
    int nonDifferentiablePayload;
    static MyRayDifferential dzero()
    { return {float3(0.0), float3(0.0)}; }
    static MyRayDifferential dadd(MyRayDifferential v1,
                                  MyRayDifferential v2)
    {
        MyRayDifferential result;
        result.d_origin = v1.d_origin + v2.d_origin;
        result.d_dir = v1.d_dir + v2.d_dir;
        return result;
    }
    static MyRayDifferential dmul(MyRay p, MyRayDifferential d)
    {
        MyRayDifferential result;
        result.d_origin = p.origin * d.d_origin;
        result.d_dir = p.dir * d.d_dir;
        return result;
    }
}
```

AUTOMATIC DIFFERENTIATION SUPPORT

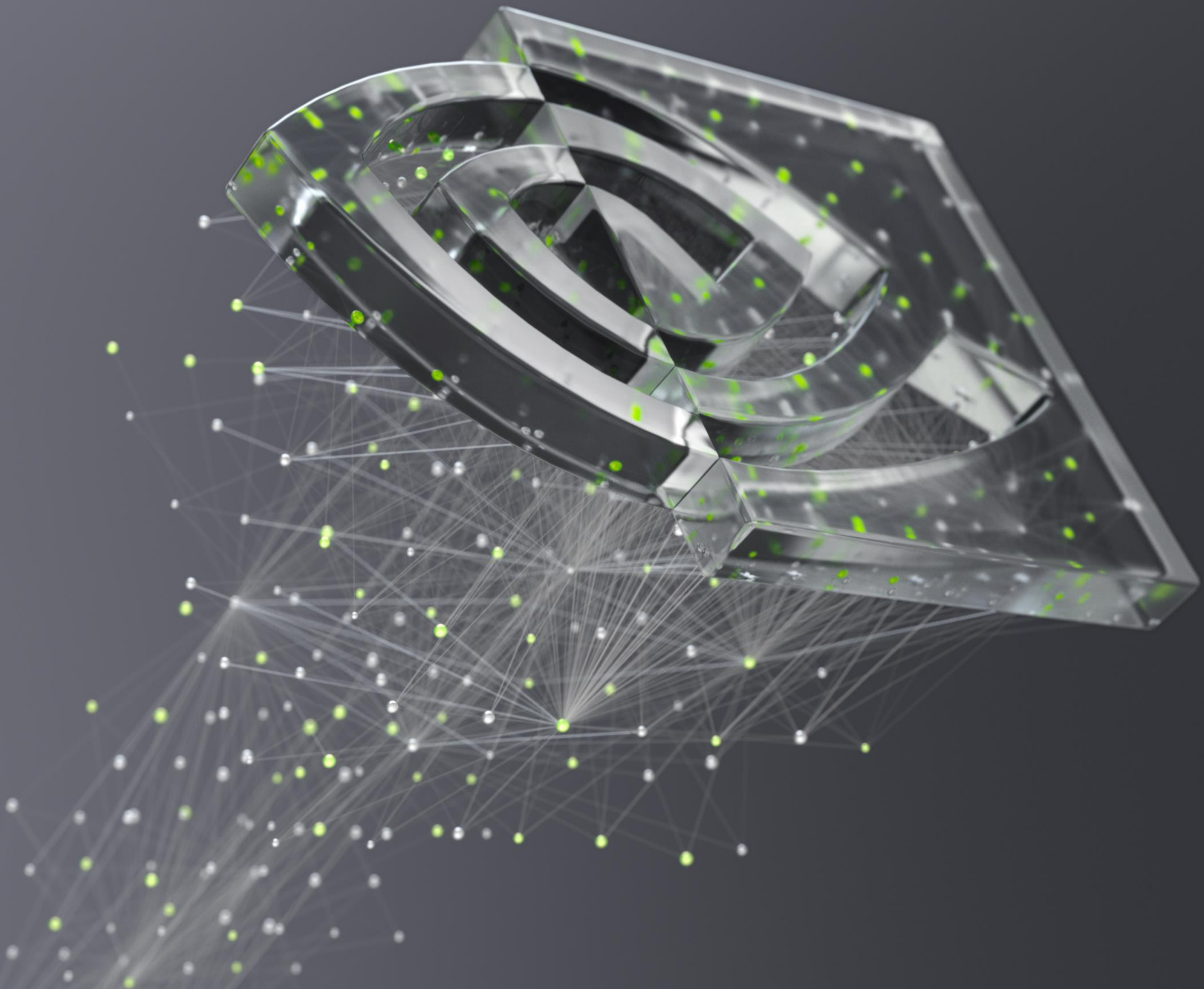
Too much to go over it all today

- Works with existing shader code, including codebases built for modularity
  - Supports mutable local variables and control flow (including loops)
  - Works with generics, interfaces, modules, etc.
- Forward, backward, and higher-order differentiation
- Application can specify hand-written derivatives for individual functions
- Slangpy module (available through pip) allows loading .slang files into PyTorch

Slang PyTorch

SLANG ADDRESSES THE KEY CHALLENGES

- Increasing scale and complexity of shader codebases
  - Modules, interfaces, generics, …
- Rapid pace of Vulkan API and SPIR-V extensions / innovation
  - Parameter blocks, inline SPIR-V assembly, …
- The “shader combinatorics” problem
  - Compiler supports multiple code generation strategies for interface dispatch
- AI/ML is a powerful new tool in real-time rendering
  - Automatic integration deeply integrated into language, type system, and compiler



NVIDIA

KEY CHALLENGES

Not being addressed by GLSL and alternatives

- Increasing scale and complexity of shader codebases
  - Languages originally designed for 10s of lines of code being used for 10s of **thousands** of lines of code
- Rapid pace of Vulkan API and SPIR-V extensions / innovation
  - No time to evolve language fundamentals – there is always a new extension that needs to be bolted on
- The “shader combinatorics” problem
  - Numbers of compiled variants affects compile times, binary sizes, load times, etc.
  - Persistent problem for entire real-time rendering industry

STATE OF THE ART IN REAL-TIME RENDERING EVOLVES

Increasing complexity of shader codebases

- Physically-Based Rendering
- (Hybrid) Ray Tracing
- Full Path Tracing + Reconstruction
- AI/ML-Accelerated Rendering

NEED LANGUAGE AND COMPILER THAT CAN SCALE

- Larger shader codebases
  - [TODO: comparison of lines of code in version N and version N+k of some engine shader codebase]
- Modular and reusable rendering techniques
  - Production developers shouldn’t have to reinvent the wheel
  - Ecosystem of rendering components shared through package managers, github, …
- Multiplatform development
  - Don’t let Vulkan path be reduced to lowest common denominator

KEY CHALLENGES

Not being addressed by GLSL and alternatives

- Increasing scale and complexity of shader codebases
  - Languages originally designed for 10s of lines of code being used for 10s of **thousands** of lines of code
- Rapid pace of Vulkan API and SPIR-V extensions / innovation
  - No time to evolve language fundamentals – there is always a new extension that needs to be bolted on
- The “shader combinatorics” problem
  - Numbers of compiled variants affects compile times, binary sizes, load times, etc.
  - Persistent problem for entire real-time rendering industry

HLSL/DXC

Velocity + Varying Priorities

- HLSL/DXC a top NV priority for new language features
  - But...it takes **time** to properly implement these important features in HLSL
  - Priorities and schedule for deliverables don't always line up across companies
  - Example of feature delay: Auto-differentiation
- DXC -> Vulkan/SPIR-V paths will continue to be extremely important
  - This does require investment by the ecosystem to maintain feature parity at required industry pace
  - Doesn't always happen as fast as folks need...
  - Example of feature delay of Vulkan/SPIR-V support in DXC: <NEED EXAMPLE>
- We'll continue to bring everything discussed here to HLSL/DXC at high priority, but it's interesting to consider alternative ways of getting this language tech to folks faster...

KEY CHALLENGES

Not being addressed by GLSL and alternatives

- Increasing scale and complexity of shader codebases
  - Languages originally designed for 10s of lines of code being used for 10s of **thousands** of lines of code
- Rapid pace of Vulkan API and SPIR-V extensions / innovation
  - No time to evolve language fundamentals – there is always a new extension that needs to be bolted on
- The “shader combinatorics” problem
  - Numbers of compiled variants affects compile times, binary sizes, load times, etc.
  - Persistent problem for entire real-time rendering industry