Slang Virtual Meetup - October 21, 2025

Getting Started with Slang: Automatic Differentiation

KHRONOS GROUP

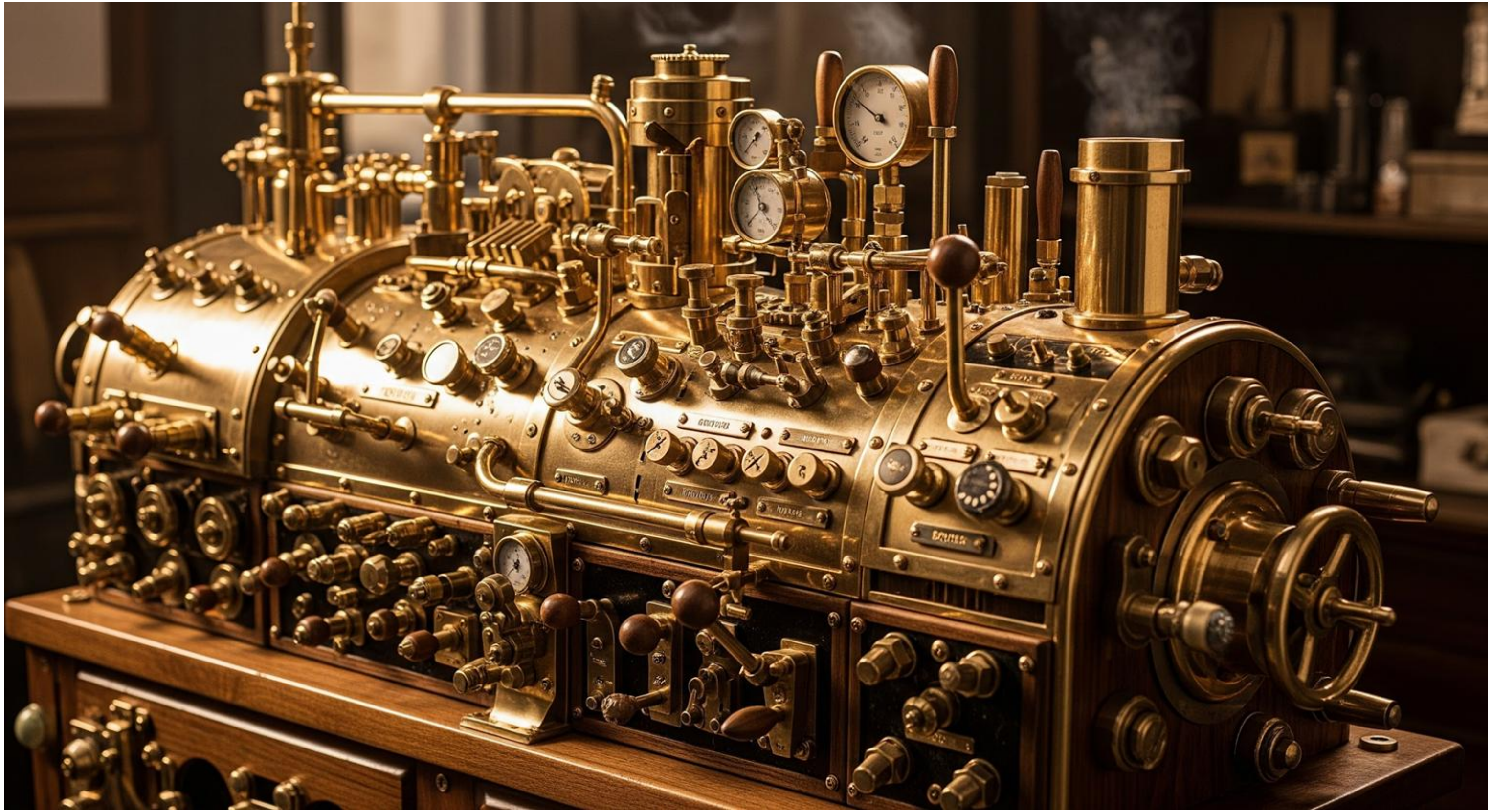
Slang™ shader-slang.org

How to Participate

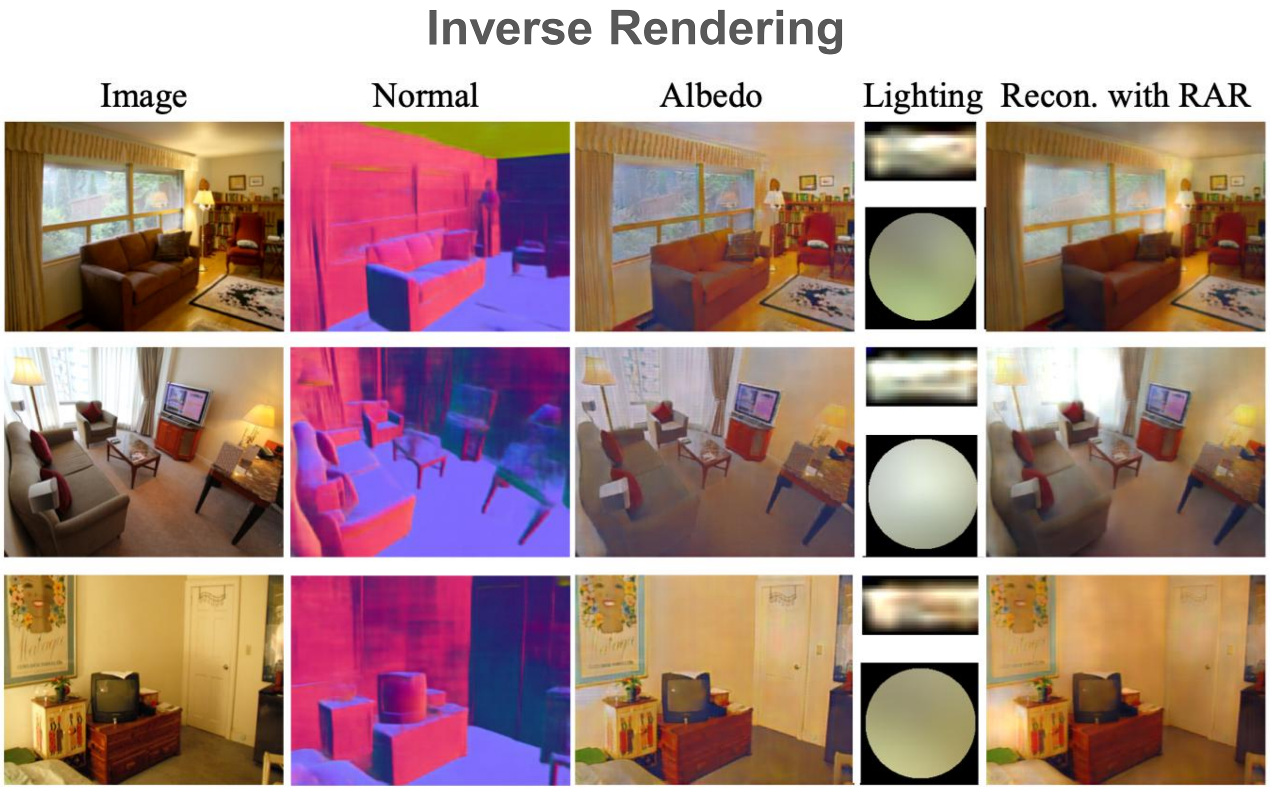
- Speaker Questions
  - During the presentations, please submit your questions to the speakers by using the Zoom Q&A feature, not the chat button
  - At the end of the talk, our moderator will put as many questions as possible to the speaker
- Recording
  - We are recording this webinar and will be sharing it via the event page on the Slang website
  - A direct link will be posted in chat: https://shader-slang.org/event/2025/10/06/getting-started-with-slang-automatic-differentiation
- Survey
  - To help us design future Slang events, we would appreciate it if you could complete the short survey form that will pop up at the end of the webinar

Introduction: Why Do We Care So Much About Derivatives?

A Machine with a Thousand Levers



Practical Problems in Graphics and Beyond



Neural Inverse Rendering of an Indoor Scene From a Single Image, Sengupta et al, https://senguptaumd.github.io/Neural-Inverse-Rendering/

Practical Problems in Graphics and Beyond

3D Gaussian Splatting

3D Gaussian Splatting for Real-Time Radiance Field Rendering, Kerbl et al, https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/

Practical Problems in Graphics and Beyond

Procedural content optimization

Reference

Loss

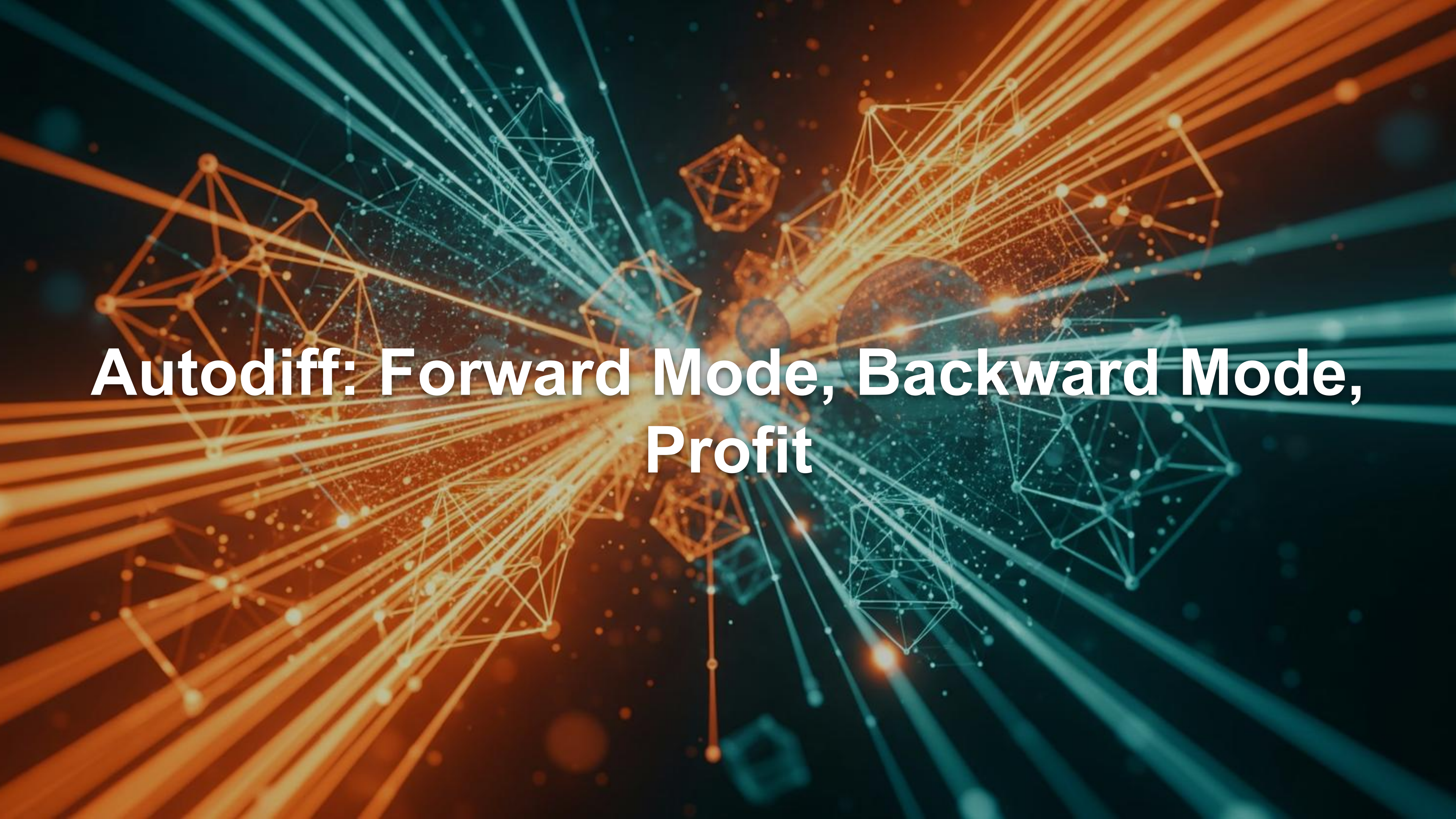
Render

Practical Problems in Graphics and Beyond

Differentiable Physics and

Fluid Simulation on Neural Flow Maps, Deng et al,
https://yitongdeng-projects.github.io/neural_flow_maps_webpage/

Autodiff: Forward Mode, Backward Mode, Profit



The Math Part: Intuiting About Derivative Propagation

Consider the following equation: $y = f(g(h(x)))$, which we could alternatively notate as:

$$\begin{aligned} x_0 &= x \\ x_1 &= h(x_0) \\ x_2 &= g(x_1) \\ x_3 &= f(x_2) \\ y &= x_3 \end{aligned}$$

Forward Mode

$$dx_0/dx = dx/dx = 1$$
$$dx_1/dx = dh/dx_0 * dx_0/dx$$
$$dx_2/dx = dg/dx_1 * dx_1/dx$$
$$dx_3/dx = df/dx_2 * dx_2/dx$$

$$y' = f'(g(h(x))) * g'(h(x)) * h'(x)$$

Reverse Mode

$$dy/dx_3 = dy/dy = 1$$
$$dy/dx_2 = dy/dx_3 * df/dx_2$$
$$dy/dx_1 = dy/dx_2 * dg/dx_1$$
$$dy/dx_0 = dy/dx_1 * dh/dx_0$$

$$y' = f'(g(h(x))) * g'(h(x)) * h'(x)$$

Which Mode To Use When

Forward Mode

- Requires a separate pass for **each input variable** for which you want to compute a derivative
- **Cost** is proportional to **number of inputs,** but scales well to large numbers of outputs

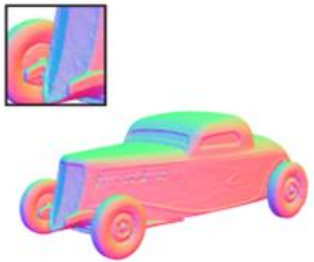
“How does changing this parameter affect all of these outcomes?”

Reverse Mode

- Requires **one forward pass** to compute and store intermediate values, and **one backward pass** to compute all gradients
- **Cost** is proportional to **number of outputs**.
- May also require storage space for intermediate values

“How is the outcome affected by each of these parameters?”

The most common use case is calculating derivatives WRT to a large number of inputs, and only one output – so reverse mode is usually used. But there are notable use cases for forward mode, e.g. calculating the normal of a SDF by differentiating with respect to position – just a 3-float vector.

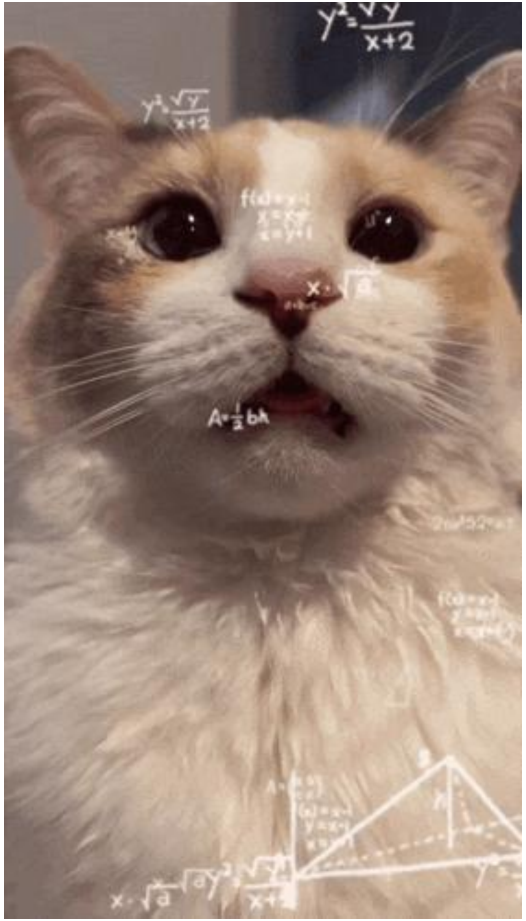


Takikawa et al, *Neural Geometric Level of Detail: Real-time Rendering with Implicit 3D Shapes*

The Sticky Part: Hand-Writing Derivatives Is Hard

```
// The primal function to compute noise
float3 computeNoise(float2 uv, float freq, float amp, float rough)
{
   float noise_val = 0.0;
   for (int i = 0; i < 5; i++) {
       float i_f = (float)i;
       float current_freq = freq * pow(2.0, i_f);
       float current_amp = amp * pow(rough, i_f);
       float term = sin(uv.x * current_freq) * exp(uv.y * current_freq) +
                    pow(uv.x, 2.0) * cos(uv.y * current_freq);
       noise_val += current_amp * term;
   }
   return float3(noise_val);
}

// The loss function for optimization
float calcLoss(float2 uv, float3 target, float freq,
               float amp, float rough)
{
   float3 noise_color = computeNoise(uv, freq, amp, rough);
   float3 diff = noise_color - target;
   return dot(diff, diff);
}
```



Generating Derivatives with Slang Is Easy

```
// The primal function to compute noise
[Differentiable]
float3 computeNoise(no_diff float2 uv, float freq, float amp, float rough)
{
   float noise_val = 0.0;
   for (int i = 0; i < 5; i++) {
       float i_f = (float)i;
       float current_freq = freq * pow(2.0, i_f);
       float current_amp = amp * pow(rough, i_f);
       float term = sin(uv.x * current_freq) * exp(uv.y * current_freq) +
                    pow(uv.x, 2.0) * cos(uv.y * current_freq);
       noise_val += current_amp * term;
   }
   return float3(noise_val);
}

[Differentiable]
// The loss function for optimization
float calcLoss(no_diff float2 uv, no_diff float3 target,
               float freq, float amp, float rough)
{
   float3 noise_color = computeNoise(uv, freq, amp, rough);
   float3 diff = noise_color - target;
   return dot(diff, diff);
}
```

"Differentiable" annotation

```
bwd_diff(calcLoss)(uv, target, dpFreq, dpAmp,
                   dpRough, 1.0);
```

"no

```
void calcLoss_Backward(float2 uv, float3 target,
                       DifferentialPair<float> freq,
                       DifferentialPair<float> amp,
                       DifferentialPair<float> rough,
                       float)
{
      /* ... */
}
```

Getting Your Hands On the Gradients

```
var dpFreq = diffPair(freq);     // dpFreq.p (the primal value) is set to freq
var dpAmp = diffPair(amp);       // dpAmp.p is set to amp
var dpRough = diffPair(rough);   // dpRough.p is set to rough

bwd_diff(calcLoss)(uv, target, dpFreq, dpAmp, dpRough, 1.0);
// gradients stored in dpFreq.d, dpAmp.d, dpRough.d

dpFreq = diffPair(freq, 1.0);     // Set dpFreq.d to 1.0 to find the derivative WRT freq
dpAmp = diffPair(amp, 0.0);            // Other variables are held constant, so their derivatives
are 0
dpRough = diffPair(rough, 0.0);

DifferentialPair<float> dpOut = fwd_diff(calcLoss)(uv, target, dpFreq, dpAmp, dpRough);
// dpOut.d contains the derivative of calcLoss WRT freq
// dpOut.p contains the result of calcLoss when called normally with freq, amp, rough as params
```

Differentiable Types: The Compiler's Got Your Back

The `IDifferentiable` Interface

- Slang will only generate differential code for values of a type conforming to the `IDifferentiable` interface.
- Both built-in and user-defined types can implement this interface
- This interface requires that any type implementing it has an associated type, which the compiler will use to carry the derivative.
- This associated type is accessed via `Type.Differential`
- A type may (and often does) have itself as the associated `Differential` type.
- The compiler is almost always able to generate the associated `Differential` type

For more information on the behavior of IDifferentiable, see the Slang reference: https://shader-slang.org/stdlib-reference/interfaces/idifferentiable-01/index

Built-In Differentiable Types

Built-in types which are differentiable:

- Scalars: `float`, `double`, and `half`
- `vector` and `matrix` of differentiable scalars
- Arrays of a differentiable type
- `Tuple<T, U, V, … >` where all composing types are differentiable

Non-continuous value types are **not differentiable**

- `int`, `uint`, `bool`
- `void`

Pointer and reference types are a special case…

- This includes resource types like `RWStructuredBuffer` and `Texture2D`

Roll Your Own: User-Defined Differentiable Types

```
struct myType : IDifferentiable
{
    float field;
    float3 vecField;
    float3x3 matrixField;

    typealias Differential = myType;
}
```

Most of the time, all you need to do is specify the interface

The compiler can synthesize the differential

Although in this case, the type is identical, so instead, it will simply use the existing type as its own Differential

Roll Your Own: User-Defined Differentiable Types

```
struct myType : IDifferentiable
{
    int field;
    float3 vecField;
    float3x3 matrixField;

    typealias Differential = myType;
}
```

Alternatively, if you did want the differential type to include the `int` field, you can still do so by explicitly providing the `typealias` definition yourself.

Roll Your Own: User-Defined Differentiable Types

```
struct myType<T : IDifferentiable, U> : IDifferentiable
{
    T t;
    U u;
};
```

```
struct myType<T, U>.Differential
{
    T.Differential t;
};
```

Generic types can also be `IDifferentiable`

The synthesized differential type will carry through all the differentiable fields, just like a concrete aggregate type

Types that aren't constrained to `IDifferentiable` will not be carried through

Roll Your Own: User-Defined Differentiable Types

```
[Differentiable]
float myFunc(myType x)
{ /* ... */ }
```

User-defined differentiable types are passed to differentiable functions much like built-in types.

```
myType primal_val;
myType.Differential diff_val;
DifferentialPair<myType>  x_pair;
x_pair = diffPair(primal_val);

float dOutput = 1.0f;
bwd_diff(myFunc)(x_pair, dOutput);
```

Wrap the primal and differential values in a differentiable pair.

`diffPair()` can still be used with a single parameter to zero-initialize the differential, if a constructor exists.


Getting Fancy: Custom Derivatives

The Why: When Autodiff Isn't Enough

- Opaque functions

```
Texture2D texture;
texture.Sample(/* ... */);
```

- Buffer accesses

```
RWStructuredBuffer<float> myBuffer;
let value = myBuffer[idx];
```

- Numerical instability

```
tan(x);
log(x);
sqrt(x);
```

The How: Writing a Custom Derivative

```
[Differentiable]
float square(float x, float y)
{
    return x * x + y * y;
}
```

```
[ForwardDerivativeOf(square)]
DifferentialPair<float> squareFwd(DifferentialPair<float> x,
                                  DifferentialPair<float> y)
{
    return diffPair(square(x.p, y.p),
                    2.0f * (x.p * x.d + y.p * y.d));
}
```

```
[BackwardDerivativeOf(square)]
void squareFwd(inout DifferentialPair<float> x,
               inout DifferentialPair<float> y,
               float dOut)
{
    x = diffPair(x.p, 2.0f * x.p * dOut);
    y = diffPair(y.p, 2.0f * y.p * dOut);
}
```

The How: Writing a Custom Derivative

```
RWStructuredBuffer<float> yBuffer;

[Differentiable]
float square(float x, int yIdx)
{
    let y = getY(yIdx);
    return x * x +  y * y;
}
```

Step 1: Wrap the buffer access

Step 2: Provide custom derivative

```
RWStructuredBuffer<Atomic<float>> yGradBuffer;

[BackwardDerivativeOf(getY)]
void getYBwd(int yIdx, float dOut)
{
    yGradBuffer[yIdx] += dOut;
}
```

```
float getY(int yIdx) { return yBuffer[yIdx]; }
```

The How: Writing a Custom Derivative

```
[Differentiable]
float square(float x, float y)
{
    let x2 = x * x;
    let y2 = debugGrad(y * y);
    return x2 + y2;
}
```

Step 1: Wrap the variable to be debugged

```
[BackwardDerivativeOf(debugGrad)]
void debugGradBwd(inout DifferentialPair<float> x,
                        float dOut)
{
    printf("Gradient is %f\n", dOut);
    x = diffPair(x.p, dOut);
}
```

Step 2: Add the printf custom derivative

```
float debugGrad(float x) { return x; }
```

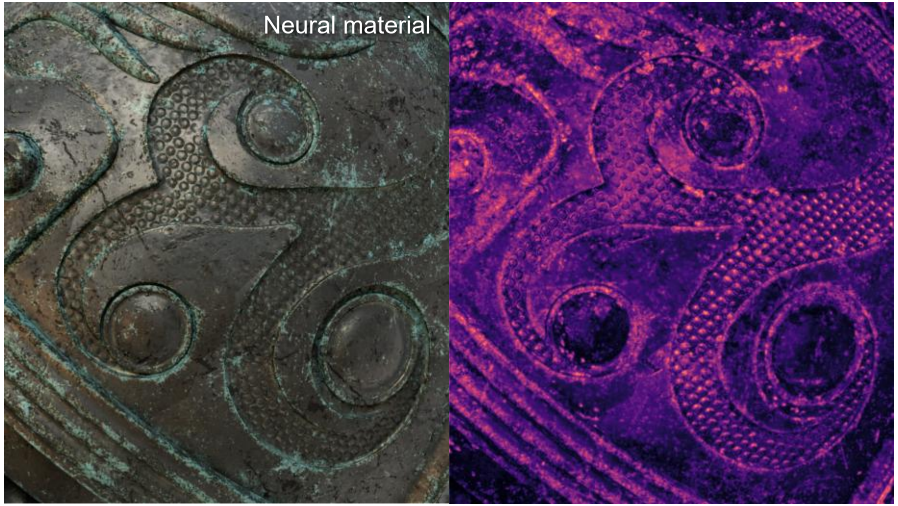
Resources

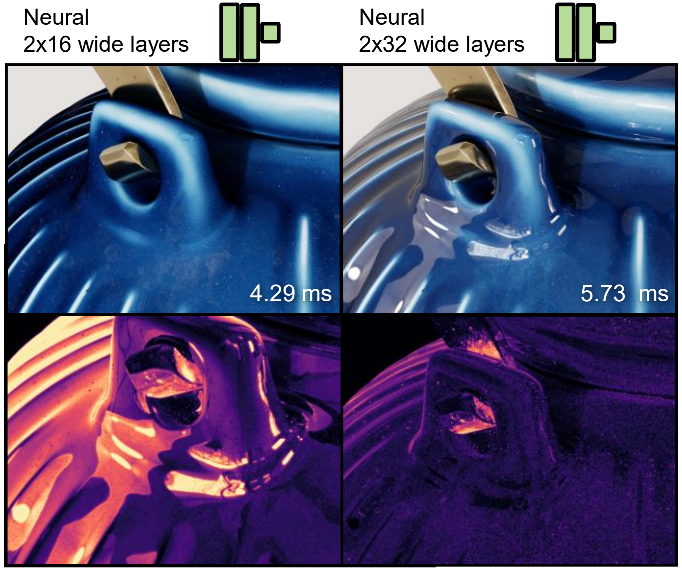
Join us on Discord!

[Neral Shading - SIGGRAPH 2025]

[Slang Autodiff Documentation]

[Slang Playground: try.shader-slang.org]





A recording of this presentation will be available at
shader-slang.org/news/

For more information on Slang, please visit
shader-slang.org

Email: slang@lists.khronos.org