

Render the Possibilities

# SIGGRAPH 2016

THE 43RD INTERNATIONAL  
CONFERENCE AND EXHIBITION ON



Computer Graphics  
Interactive Techniques

24-28 JULY

ANAHEIM, CALIFORNIA



# **A Modern Programming Language for Real-Time Graphics: What is Needed?**

**Tim Foley (NVIDIA Research)**

Hello. I'm Tim Foley from NVIDIA Research, and I'm here today to talk about the programming languages and tools we use for real-time graphics, and how we could possibly improve things.

# Who am I?

- **Real-time graphics researcher at NVIDIA**
- **Work on programming languages, models, and APIs for graphics**
  - GPU compute
  - Shading languages
  - Shader-style models for exploiting CPUs

Open Problems in Real-Time Rendering, SIGGRAPH 2016

3

Before I dive in, I'll go ahead and give a bit of background on myself.

I work in a real-time rendering research group at NVIDIA, but I try to straddle the line between graphics and languages/compiler.

My research projects to date have mostly been on languages, programming models, and APIs for graphics.

I've done work on some of the early GPU compute systems, alternative approaches to shading languages, and systems using shader-like programming to exploit CPUs (SIMD, etc.).

# Real-Time Graphics Programming Today

Open Problems in Real-Time Rendering, SIGGRAPH 2016

4

As a starting point, I'd like to spend some time talking about the state of the art in real-time graphics programming today.

This is largely based on my experience when I go and write application code.

## When we write the CPU part of an application

C/C++

clang

lldb

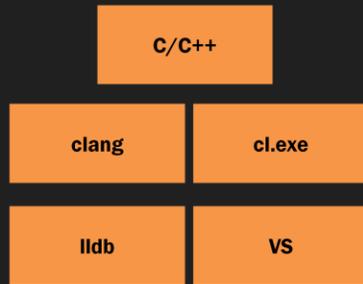
Open Problems in Real-Time Rendering, SIGGRAPH 2016

5

When I start out writing the CPU part of an application, I typically use a simple set of tools.

I'll write in C or C++ (depending on the requirements) and pick the front-end for my chosen platform.

## Maybe a few different compilers/debuggers



Open Problems in Real-Time Rendering, SIGGRAPH 2016

6

If I need to write code for multiple platforms, I might deal with a few front-ends, and a different debugger per-platform.  
Overall, though, the situation isn't too bad.

# GPU graphics requires its own languages

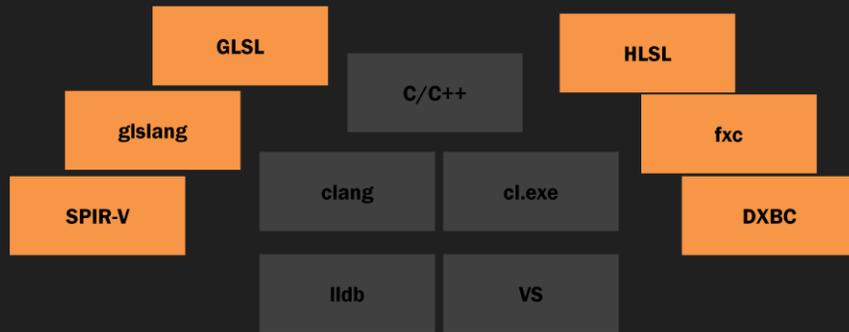


Open Problems in Real-Time Rendering, SIGGRAPH 2016

7

As soon as I start trying to use a GPU to do graphics, though, things get messier. I need to write shader code in its own special language, and different graphics APIs have their own languages.

## Different compilers and infrastructure

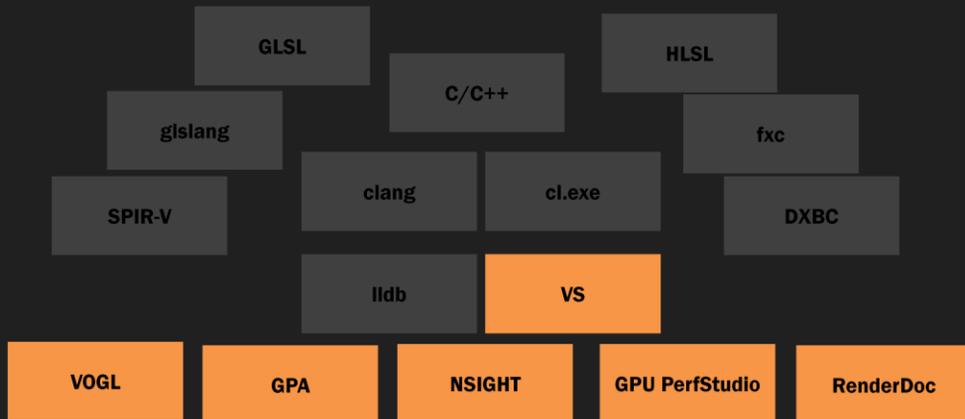


Open Problems in Real-Time Rendering, SIGGRAPH 2016

8

And those languages have their own supporting compilers and intermediate languages (ILs).

## Debugging tools vary by API/platform/HW



Open Problems in Real-Time Rendering, SIGGRAPH 2016

9

And when it comes time to debug my GPU code, or evaluate its performance, I have a bunch of different tools to consider.

Many of these tools are very powerful (please don't think I'm criticizing them), but they are often locked to specific graphics APIs, platforms, or HW vendors.

So where I had a relatively simple picture when I just need to run on CPU, as soon as I want to do GPU work I'm faced with this zoo.

## The way we program GPU graphics is outdated

- Different languages, different programs for CPU/GPU
- Specialization and composition with ad hoc string processing
- Unwieldy to build engine-specific compilation tools, DSLs

Open Problems in Real-Time Rendering, SIGGRAPH 2016

10

It isn't just about a proliferation of languages and tools, though; the way we program GPUs for real-time graphics is woefully outdated.

Not only do we write our GPU code in different languages, but we effectively write an entirely different *program* for CPU and GPU.

When it comes time to specialize and compose GPU shader code for performance or modularity reasons, we resort to ad hoc methods. We are really just banging strings together.

As a result, it is quite difficult to build more advanced tooling at the engine level – things like custom languages, optimizations, or code generators.

## Start with a toy kernel on CPU

```
void add( int n, float* a, float* b ) {
    for( int i = 0; i < n; i++ ) {
        a[i] += b[i];
    }
}

int main() {
    int n = 100;
    float* a = (float*) malloc(n * sizeof(float));
    float* b = (float*) malloc(n * sizeof(float));

    // ...

    add(n, a, b);
}
```

Open Problems in Real-Time Rendering, SIGGRAPH 2016

11

To dig into some of these issues, let's start with a toy kernel that I might write on CPU.

Here I've got code for adding two arrays together. Let's suppose this isn't fast enough (because we can never add 100-element arrays fast enough!), and I decide to run it on my GPU.

## Let's run a simple kernel on GPU using D3D11

```
void add( int n, float* a, float* b ) {  
    for( int i = 0; i < n; i++ ) {  
        a[i] += b[i];  
    }  
}
```

suppose I want to run this on GPU

```
int main() {  
    int n = 100;  
    float* a = (float*) malloc(n * sizeof(float));  
    float* b = (float*) malloc(n * sizeof(float));  
  
    // ...  
  
    add(n, a, b);  
}
```

Open Problems in Real-Time Rendering, SIGGRAPH 2016

12

So I'm going to try to run this bit of code here on my GPU, and I'm going to use Direct3D 11 to do it (just because D3D 12 would be even *\*more\** code).

# First we just rewrite the kernel in HLSL

```
// add.hlsl
```

several different ways of passing parameters

```
cbuffer U {  
    uint n;  
}
```

```
RWStructuredBuffer<float> a;  
RWStructuredBuffer<float> b;
```

```
[numthreads(32,1,1)]  
void add( uint i : SV_DispatchThreadID ) {  
    a[i] += b[i];  
}
```

core logic didn't change much

Open Problems in Real-Time Rendering, SIGGRAPH 2016

13

First, I need to rewrite my kernel code into HLSL.

As you can see at the bottom of the slide, the core logic didn't change much.

But I've got all these extra boilerplate now, which mostly has to do with passing parameters into the kernel.

Every kind of parameter seems to have its own syntax, and so we end up with something a lot messier than we started with.

## Make sure to integrate HLSL into your build

```
fxc /T cs_5_0 /Fo add.dxbc add.hlsl
```

invoke stand-alone compiler from your build

```
FILE* sourceFile = fopen("add.hlsl", "rb");  
// load file to memory  
D3DCompile(source, strlen(source), path, NULL, NULL, "main", ...);
```

or coordinate compilation yourself using API

Next I need to integrate the HLSL compiler (fxc.exe) into my build, or use the provided API to orchestrate compilation myself.

## ...and load bytecode at runtime

```
FILE* bytecodeFile = fopen("add.dxbc", "rb");  
  
// load file to memory  
  
d3dDevice->CreateComputeShader(bytecode, bytecodeSize, NULL, &addShader);
```

And then I need some code to load the compiled bytecode at runtime.

## Okay, now we can actually invoke it!

```
// main.cpp

int main() {
    // ...

    context->CSetShader(addShader, 0, 0);
    context->CSetConstantBuffers(0, 1, &cb);
    ID3D11UnorderedAccessView* uavs[] = { a, b };
    context->CSetUnorderedAccessViews(0, 2, &uavs, 0);

    context->Dispatch((n + 31) & ~31, 1, 1);
}
```

this used to be a single function call

matching of arguments to parameters is highly implicit

Open Problems in Real-Time Rendering, SIGGRAPH 2016

16

And now I can finally get around to invoking my kernel.

But what used to be a single function call in the original code is now a bunch of state-setting logic followed by a dispatch.

And the parameter passing is really implicit here, so I have to try to remember which binding slot I used for `a` and `b`...

# Oops, we forgot something!

```
// add.hlsl
```

manually specify placement of every parameter

```
cbuffer U : register(b0) {  
    uint n;  
}  
  
RWStructuredBuffer<float> a : register(u0);  
RWStructuredBuffer<float> b : register(u1);  
  
[numthreads(32,1,1)]  
void add( uint i : SV_DispatchThreadID ) {  
    a[i] += b[i];  
}
```

Open Problems in Real-Time Rendering, SIGGRAPH 2016

17

...and of course the answer is undefined, because back on that earlier slide I forgot to add this extra cruft to my parameters to specify how they should get placed.

# I just wanted to add two arrays!

```
// add.hlsl
cbuffer U : register(b0) {
    uint n;
}

RWStructuredBuffer<float> a : register(u0);
RWStructuredBuffer<float> b : register(u1);

[numthreads(32,1,1)]
void add( uint i : SV_DispatchThreadID ) {
    a[i] += b[i];
}

FILE* sourceFile = fopen("add.hlsl", "rb");
// load file to memory
D3DCompile(source, strlen(source), path, NULL, NULL, "main", ...);

FILE* bytecodeFile = fopen("add.dxbc", "rb");
// load file to memory
d3dDevice->CreateComputeShader(bytecode, bytecodeSize, NULL, &addShader);

// main.cpp
int main() {
    // ...
    context->CSetShader(addShader, 0, 0);
    context->CSetConstantBuffers(0, 1, &cb);
    ID3D11UnorderedAccessView* uavs[] = { a, b };
    context->CSetUnorderedAccessViews(0, 2, &uavs, 0);

    context->Dispatch((n + 31) & ~31, 1, 1);
}
```

Open Problems in Real-Time Rendering, SIGGRAPH 2016

18

Okay. Finally.

We have something that works.

But I just wanted to add two arrays!

Is this really the best we can do?

Doesn't it seem like all of this is going to make it harder to experiment with moving our code to/from GPU compute?

# Specialize using string-based metaprogramming

## Preprocessor

```
float2 roughness;

#if ANISOTROPIC_ROUGHNESS_TEXTURE
    roughness = roughnessTex.Sample(...).xy;
#elif ROUGHNESS_TEXTURE
    roughness = roughnessTex.Sample(...).x;
#else
    roughness = uniformRoughness;
#endif
```

## String-Splicing

```
void generateMaterialCode(Material* m)
{
    emit("float4 evalMaterialLayers() {\n");
    emit("float4 albedo = 1.0f;\n");
    for( auto layer : m->layers )
    {
        emit("albedo = ");
        generateLayerCode(layer, "albedo");
        emit(";\n");
    }
    // ...
}
```

Next, when it comes time to specialize our kernels (whether compute or graphics) for performance, we are probably going to do one of two things.

First, we might run our code through a preprocessor, to generate specialized variants. Usually this is just the ordinary `#ifdef`, but I know that some of you out there have implemented your own preprocessors with fancy stuff like `#for`.

In the second case, maybe you have let your artists write surface shaders using some kind of “noodle graph,” so you need to generate the final shader code by pasting strings together.

# Design-space exploration

GI: baked, voxels, probes, screen-space?

Use GPU compute for culling/submission?

Tiled deferred or forward+ ?

Manually code as many options as you can

Skinning on CPU or GPU?

How to partition for multi-GPU?

Decoupled/texture-space shading?

Move some simulation/GI to servers, amortize across users?

Open Problems in Real-Time Rendering, SIGGRAPH 2016

20

Finally, I want to talk about design-space exploration.

There are a lot of high-level design choices I might have when I go to build a renderer. For example, I might ask myself whether I should be doing one of these new “forward +” things, or stick with tiled deferred.

I might ask myself whether it makes sense to use GPU compute to do my scene culling and generate data for submission like the “multi-draw-indirect” advocates talk about.

For each of these choices, the current state of the art is: you just manually code as many of the options you can, and see what is fastest.

But wouldn't it be better if we built tools that could help with this?

# The Future of Real-Time Graphics Programming

(one possibility)

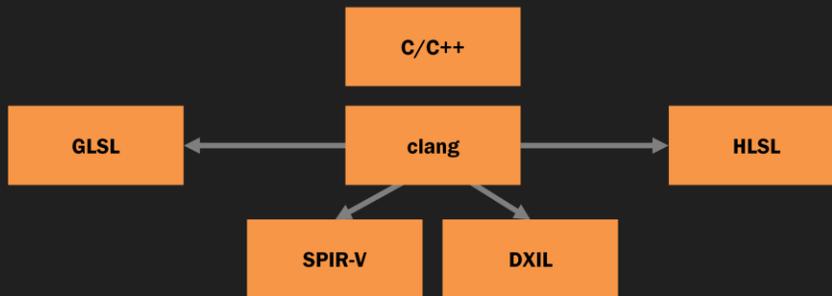
Open Problems in Real-Time Rendering, SIGGRAPH 2016

21

With some of the problems (or challenges) covered, I'm going to spend a little time sketching one possible vision for what the future of graphics programming might look like.

Later in the talk I'm going to go into more detail on these items, so for right now lets just consider these high-level aspirations.

## Step 1: Move to C/C++ for shaders



Open Problems in Real-Time Rendering, SIGGRAPH 2016

22

First, let's see if we, as an industry, and finally make the jump to using C/C++ for our shader code.

That way, instead of the zoo of tools I showed earlier, we might have a picture like this.

# Heterogeneous CPU + GPU Programming

```
// add.hlsl
cbuffer U : register(b0) {
    uint n;
}

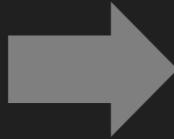
RWStructuredBuffer<float> a : register(u0);
RWStructuredBuffer<float> b : register(u1);

[numthreads(32,1,1)]
void add( uint i : SV_DispatchThreadID ) {
    a[i] += b[i];
}

FILE* sourceFile = fopen("add.hlsl", "rb");
// load file to memory
D3DCompile(source, strlen(source), path, NULL, NULL, "main", ...);

FILE* bytecodeFile = fopen("add.dxbc", "rb");
// load file to memory
d3dDevice->CreateComputeShader(bytecode, bytecodeSize, NULL, &addShader);

// main.cpp
int main() {
    // ...
    context->CSSetShader(addShader, 0, 0);
    context->CSSetConstantBuffers(0, 1, &cb);
    ID3D11UnorderedAccessView* uavs[] = { a, b };
    context->CSSetUnorderedAccessViews(0, 2, &uavs, 0);
    context->Dispatch((n + 31) & -31, 1, 1);
}
```



```
[[target(gpu)]]
void add( uint i, float* a, float* b ) {
    a[i] += b[i];
}

int main() {
    float* a = gpu_malloc(n * sizeof(float));
    float* b = gpu_malloc(n * sizeof(float));

    dispatch_gpu(n, [(uint i) { add(i, a, b); }]);
}
```

Open Problems in Real-Time Rendering, SIGGRAPH 2016

23

Next, instead of the mess of code I ended up with when I tried to add two arrays, what if we had a compiler that understood that an application is built out of both CPU and GPU code, and made it simple to invoke one from the other?

# First-Class Metaprogramming Constructs

```
float2 roughness;
#if ANISOTROPIC_ROUGHNESS_TEXTURE
roughness = roughnessTex.Sample(...).xy;
#elif ROUGHNESS_TEXTURE
roughness = roughnessTex.Sample(...).x;
#else
roughness = uniformRoughness;
#endif

void generateMaterialCode(Material* m)
{
    emit("float4 evalMaterialLayers() {\n");
    emit("float4 albedo = 1.0f;\n");
    for( auto layer : m->layers )
    {
        emit("albedo = ");
        generateLayerCode(layer, "albedo");
        emit(";\n");
    }
    // ...
}
```



```
FuncDef* generateMaterialCode(Material* m)
{
    return `{ float4 evalMaterialLayers()
    {
        float4 albedo = 1.0f;
        $for(auto layer : m->layers)
        {
            albedo = $(generateLayerCode(layer, `albedo));
        }
    }
}
```

Open Problems in Real-Time Rendering, SIGGRAPH 2016

24

And then, instead of banging strings together in order to generate optimized kernels, what if we had a language with more first-class support for quoting and generating code.

[Note to folks reading the slides after the fact: the syntax over there on the right isn't from any real language, but is just supposed to be a sketch of what a "quasiquote" feature might appear like in a C-like language. Don't read too much into it, since it wasn't mean to be covered in depth.]

# Domain-Specific Languages and Optimizations

## to enable rapid design space exploration

```
post_effect HexagonalBokeh {
    uniform w, h : int;

    input color : float4 { size: w,h }
    input depth : float { size: w,h }

    temp t : float4 { size: w/2, h/2 }

    pass Blur1 {
        // ...
    }
}
```

Open Problems in Real-Time Rendering, SIGGRAPH 2016

25

And finally, what if we could use those metaprogramming facilities to build our own engine-specific languages and optimization.

Like in this case, imagine that my engine has a lot of post FX, and so I create a custom DSL to express them in a declarative fashion.

Then I could write custom code generator and optimizers, that might allow me to schedule compositions of many effects more efficiently, or manage the lifetimes of the buffers they use.

[Note: again, the example on this slide is meant to be aspirational rather than an example of code you could write today in any system.]

# What makes graphics programming interesting?

Open Problems in Real-Time Rendering, SIGGRAPH 2016

26

Before going into detail on how we might try to achieve some of these benefits, I'm going to take a step back and ask why we should be looking at the graphics programming problem in the first place.

After all, what makes graphics programming any different from other domains? Why should we be asking for these more advanced tools?

I'm going to try to answer this by describing what it is I think graphics programming is all about (this is me stepping up on my soapbox, so you are certainly all free to disagree):

**The task of a graphics programmer is  
to define and mediate the interface  
between content (art) and platform (HW)**

Open Problems in Real-Time Rendering, SIGGRAPH 2016

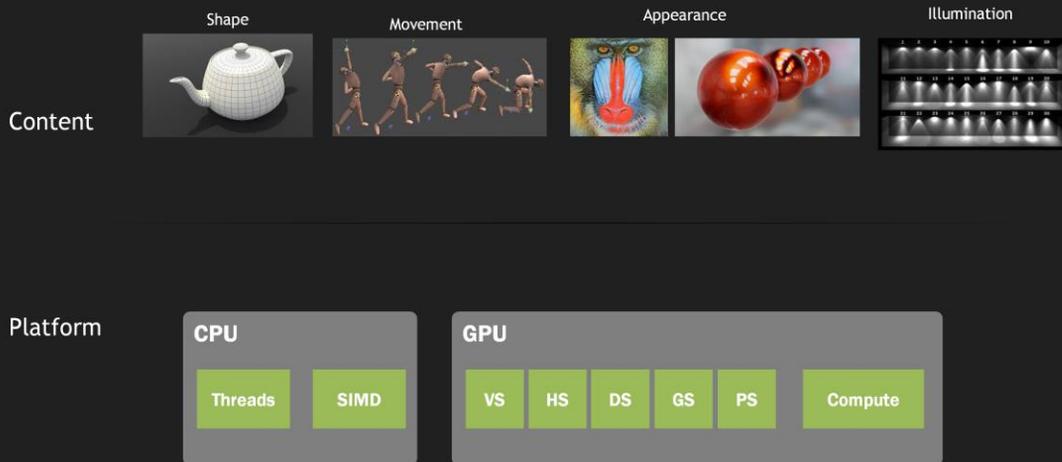
27

Now, this slide may look like a word salad right now, so I have a few figures coming up that I hope will explain it better.

I'm going to claim that the task of a graphics programmer – what we are all trying to do – is to *\*define\** and *\*mediate\** the interface between art content, and one or more HW platforms.

Let's look at some pictures to make that a bit more concrete...

# Content and Platform



Open Problems in Real-Time Rendering, SIGGRAPH 2016

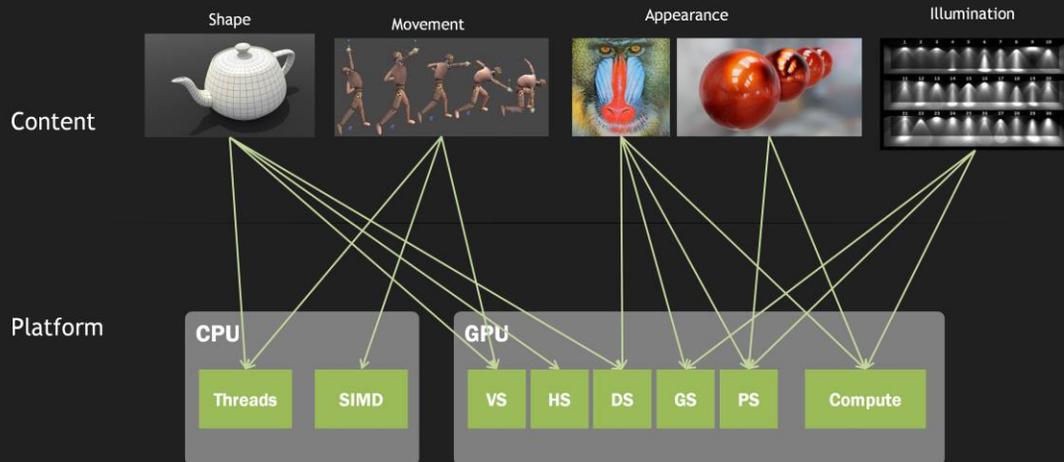
28

Here at the top of the slide we have “content,” which is all the concepts that artists are going to understand and author. We have stuff like shapes, movement, appearance, and illumination.

At the bottom we have the hardware platforms that we target, which these days look kinda similar: we have CPU cores with support for threads and SIMD, and GPU hardware that supports both a traditional rasterization pipeline, and some kind of general-purpose compute.

In order to get the stuff an artists produces up on the screen, we need to have a plan for how to map the concepts in the top row to the hardware platform on the bottom.

## Specify mapping per-asset, per-platform?

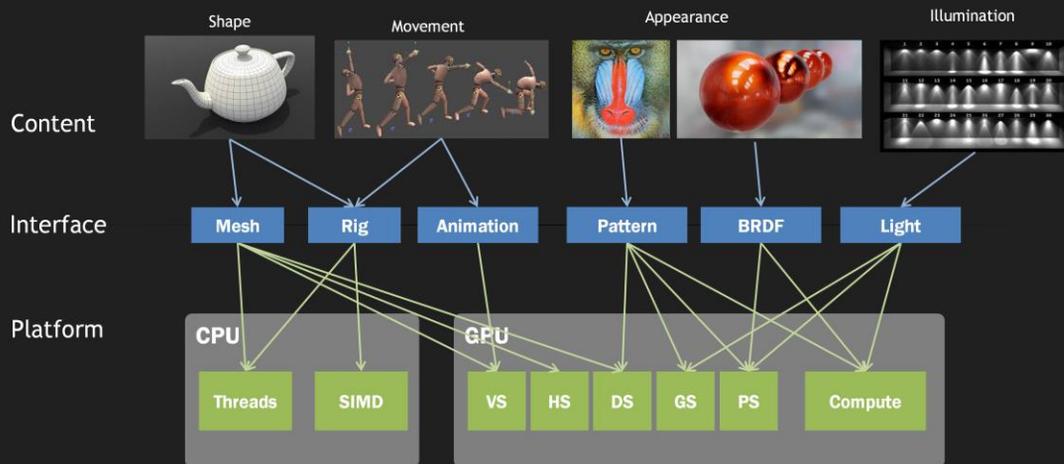


Open Problems in Real-Time Rendering, SIGGRAPH 2016

29

One (bad) way we could do that is by writing code on a per-asset, per-platform basis. This obviously wouldn't scale, and so it isn't what we actually do. And that gets us to the heart of my definition...

# Map using application-specified interface



Open Problems in Real-Time Rendering, SIGGRAPH 2016

30

What I claim we actually do in practice is this:

A graphics programmer defines an *\*interface\** - a set of concepts that are specific to a particular engine or production.

*\*This\** is how we will express an animation rig.

*\*This\** is how we will represent reflectance functions.

Some of the representations in that interface might be pure data (e.g., for a mesh), and others might include code (e.g., if artists describe pattern generation as a “noodle graph”).

Either way, the task is to:

- map the assets produced by artists so that they conform to the chosen concepts in the interface, and
- map those concepts efficiently to one or more target platforms

As a previous speaker noted, because artist time is often more valuable than programmer time on a production, it may be reasonable for developers to implement different bespoke mappings to particular platforms.

**The task of a graphics programmer is  
to define and mediate the interface  
between content (art) and platform (HW)**

Open Problems in Real-Time Rendering, SIGGRAPH 2016

31

Now I'm coming back to this definition, and I hope it makes a bit more sense now what I mean.

The task of a graphics programmer is to define and mediate that interface between content and the platform(s).

## **A modern language for graphics programming should help with that task**

Open Problems in Real-Time Rendering, SIGGRAPH 2016

32

So the key thing we should be looking for when we evaluate our languages, tools and programming models is how well they assist us with that task.

With that bit of motivation out of the way, I'm going to dive into some concrete steps we could take to improve our languages and tools.

# Can we all just move to C/C++ for shaders now?

We are so close!

Open Problems in Real-Time Rendering, SIGGRAPH 2016

33

First on that list is a simple one:

Are we finally at the point where we can, as an industry, just move to writing our shaders in C/C++?

It seems like a year ago this was a no-go, but now it seems like we are *\*really\** close.

## Okay, I actually mean “C/C++”

- **Will need to subset the language for at least a while**
  - Prior efforts to define reasonable “static C++” for OpenCL, Metal
- **Stuff that is hard to support for performance**
  - Full memory model
  - Exceptions
  - Function pointers, virtual functions
  - Recursion
  - Dynamic allocation
  - Much of the stdlib

Open Problems in Real-Time Rendering, SIGGRAPH 2016

34

Actually, before I go any further, I should probably clarify that when I say C/C++, I mean “C/C++” with giant air quotes.

For the near future, we will probably need to be working in subsets of the language.

There have been a few attempts to define reasonable “static” subsets of these languages that throw out the constructs that are really hard to support with good performance on GPUs.

Some of this stuff might get resolved over time, but other stuff... maybe we don't ever need or want it (like exceptions).

## **C/C++ for shaders will improve quality of life**

- **Use mature broadly-deployed compilers**
- **Real pointers**
- **Static C++ features (if you're into that sort of thing)**
  - **Templates**
  - **Auto**
  - **Lambdas**

Open Problems in Real-Time Rendering, SIGGRAPH 2016

35

I'm going to claim that switching to C/C++ for shaders will greatly improve quality of life for graphics programmers.

We'd be able to use mature and broadly deployed compilers. I don't know about all of you, but I'd really like to get those great clang diagnostic messages in all my code.

For folks who are doing tricky atomic-based lock-free data structures, maybe we can switch to using real pointers and drop the level of semantic indirection forced by always working with buffers and indices.

And finally, for those of you who are into that sort of thing, there are some C++ features that can help with writing more compact code.

# The pieces are falling into place

- **Khronos: SPIR-V intermediate language**
  - OSS front-ends and SPIRV ↔ LLVM being developed on GitHub
  - Clang→SPIRV currently works for OpenCL
- **Microsoft: plans for DXIL, clang-based HLSL front-end**
  - Have promised OSS model
- **Apple: Metal has clang-based front-end**
  - IL not currently open

Open Problems in Real-Time Rendering, SIGGRAPH 2016

36

Part of why I feel like this might be a realistic goal for the next year is that many of the pieces are falling into place.

The Khronos folks have standardized the SPIR-V intermediate language, which is now supported in both Vulkan and OpenGL.

That community has been doing a lot of open-source work on front-ends, translation to and from LLVM, etc.

There is even a working clang->SPIR-V compiler, with the only problem being that it works for OpenCL, and not for graphics.

Meanwhile, Microsoft has publicly (though not loudly) talked about their plans to move to a clang-based HLSL front end, and a new LLVM-based IL.

They've promised that the project will follow an open-source model, and in my opinion that can't come out soon enough.

And Apple already has a clang-based C/C++ front-end for shaders, so they are ready to go. The only sticking point there is that their IL is currently not open or publicly documented.

## Legacy platforms will be the challenge

- **How soon can you drop D3D11 and GL ES (WebGL)?**
- **C/C++ → HLSL/GLSL translation is needed**
  - Some projects tackling parts of this (e.g., SPIRV-Cross)
- **How do we coordinate to get this done?**

Open Problems in Real-Time Rendering, SIGGRAPH 2016

37

Really, the main sticking point here will be how soon people can drop support for legacy platforms.

I'm not sure how many people can really leave behind D3D11, or WebGL in the next year or so.

My guess is that in the mean time, we are going to need some kind of C/C++ to HLSL/GLSL transpiler, based on a *really* restricted subset of C/C++.

There are some bits of code out there that could help with parts of this, so maybe this is something where we can coordinate between a few companies/teams to get this done.

**And once we have C/C++  
we are all done, right?**

Open Problems in Real-Time Rendering, SIGGRAPH 2016

38

And so, once we can write our shaders in C/C++ we are done, right?

That obviously isn't what I'm here to say...

## Getting to C/C++ is the (important) first step

- However, many of the pain points will remain
- Rest of this talk: capabilities not present in C/C++
- Some of these capabilities might come to C/C++ some day
- Some might come to new languages

Open Problems in Real-Time Rendering, SIGGRAPH 2016

39

I'd argue that getting to the point where we write shaders in C/C++ is a first step (and *\*important\** first step), but it won't really resolve all of the pain points I talked about earlier.

The rest of this talk is going to focus on capabilities that *\*aren't\** present in ordinary C/C++ today.

These capabilities might come to some future version of C/C++, or they might come to other new languages (which might interoperate with C/C++).

I'm not here to champion any particular language (today), so I really just want to talk about *\*capabilities\**.

My slides will use C/C++ pseudo-code where possible, just to make them as readable as possible.

# Heterogeneous Programming

Open Problems in Real-Time Rendering, SIGGRAPH 2016

40

One big area where just switching to C/C++ for shaders doesn't help is heterogeneous programming.

Ordinary C/C++ is not a heterogeneous language.

# Remember: I just wanted to add two arrays!

```
// add.hlsl
cbuffer U : register(b0) {
    uint n;
}

RWStructuredBuffer<float> a : register(u0);
RWStructuredBuffer<float> b : register(u1);

[numthreads(32,1,1)]
void add( uint i : SV_DispatchThreadID ) {
    a[i] += b[i];
}

FILE* sourceFile = fopen("add.hlsl", "rb");
// load file to memory
D3DCompile(source, strlen(source), path, NULL, NULL, "main", ...);

FILE* bytecodeFile = fopen("add.dxbc", "rb");
// load file to memory
d3dDevice->CreateComputeShader(bytecode, bytecodeSize, NULL, &addShader);

// main.cpp
int main() {
    // ...
    context->CSetShader(addShader, 0, 0);
    context->CSetConstantBuffers(0, 1, &cb);
    ID3D11UnorderedAccessView* uavs[] = { a, b };
    context->CSetUnorderedAccessViews(0, 2, &uavs, 0);

    context->Dispatch((n + 31) & ~31, 1, 1);
}
```

Remember this example where I just wanted to add two arrays, but I had to write a bunch of code?

That isn't going to get better just because the kernel was written using C++ instead of HLSL.

# Is this really so bad?

Open Problems in Real-Time Rendering, SIGGRAPH 2016

42

You might be asking yourself “is it really so bad? People ship entire games/engines this way.”

## Let's try the same thing with CUDA

```
void add( int n, float* a, float* b ) {
    for( int i = 0; i < n; i++ ) {
        a[i] += b[i];
    }
}

int main() {
    int n = 100;
    float* a = (float*) malloc(n * sizeof(float));
    float* b = (float*) malloc(n * sizeof(float));

    // ...

    add(n, a, b);
}
```

Open Problems in Real-Time Rendering, SIGGRAPH 2016

43

As an answer to that, let's take a look at how people running on the *same* hardware, just in a different community (HPC/supercomputing folks) would do the same thing.

Suppose I'm a CUDA user and I want to run some of this code on my GPU. What do I do?

## Well, that was easy

```
device
void add( int n, float* a, float* b ) {
    int i = tid.x;
    if( i < n ) {
        a[i] += b[i];
    }
}

int main() {
    int n = 100;
    float* a; cudaMalloc(&a, n * sizeof(float));
    float* b; cudaMalloc(&b, n * sizeof(float));

    // ...

    add<<<(n + 31) & ~31, 32>>>(n, a, b);
}
```

Open Problems in Real-Time Rendering, SIGGRAPH 2016

44

Oh.... That wasn't hard at all.

And lest you think I'm just here to brag about NVIDIA...

## Similar concepts in C++ AMP

```
int main() {
    extent<1> n(100);
    array_view<float,1> a(n);
    array_view<float,1> b(n);

    // ...
    parallel_for_each(n, [=](index<1> i) restrict(amp) {
        a[i] += b[i];
    });
}
```

This is the same thing done in Microsoft's C++ AMP. It has a slightly more "modern C++" flavor to it, but the basic ideas are similar.

# What do we need?

```
// CUDA
__device__
void add( int n, float* a, float* b ) {
    int i = threadIdx;
    if( i < n ) {
        a[i] += b[i];
    }
}

int main() {
    int n = 100;
    float* a; cudaMalloc(&a, n * sizeof(float));
    float* b; cudaMalloc(&b, n * sizeof(float));

    // ...

    add<<<(n + 31) & ~31, 32>>>(n, a, b);
}

// C++ AMP

int main() {
    extent<1> n(100);
    array_view<float,1> a(n);
    array_view<float,1> b(n);

    // ...
    parallel_for_each(n, [=](index<1> i) restrict(amp) {
        a[i] += b[i];
    });
}
```

So what makes a heterogeneous programming model like this?

## A way to mark code for particular processors

```
// CUDA
__device__
void add( int n, float* a, float* b ) {
    int i = threadIdx;
    if( i < n ) {
        a[i] += b[i];
    }
}

int main() {
    int n = 100;
    float* a; cudaMalloc(&a, n * sizeof(float));
    float* b; cudaMalloc(&b, n * sizeof(float));

    // ...

    add<<<(n + 31) & ~31, 32>>>(n, a, b);
}

// C++ AMP

int main() {
    extent<1> n(100);
    array_view<float,1> a(n);
    array_view<float,1> b(n);

    // ...
    parallel_for_each(n, [=](index<1> i) restrict(amp) {
        a[i] += b[i];
    });
}
```

First, we need a way to mark code that belongs to a particular processor, so that we can have CPU functions and GPU functions living in the same file.

# Call from one processor to another

```
// CUDA
__device__
void add( int n, float* a, float* b ) {
    int i = threadIdx;
    if( i < n ) {
        a[i] += b[i];
    }
}

int main() {
    int n = 100;
    float* a; cudaMalloc(&a, n * sizeof(float));
    float* b; cudaMalloc(&b, n * sizeof(float));

    // ...

    add<<<(n + 31) & ~31, 32>>>(n, a, b);
}

// C++ AMP

int main() {
    extent<1> n(100);
    array_view<float,1> a(n);
    array_view<float,1> b(n);

    // ...
    parallel_for_each(n, [=](index<1> i) restrict(amp) {
        a[i] += b[i];
    });
}
```

Open Problems in Real-Time Rendering, SIGGRAPH 2016

48

Second, we need a way to kick off execution from one processor type to another.

CUDA does this with dedicated syntax, but C++ AMP shows that you can also do it with a higher-order function.

# Primitives to control placement of resources

```
// CUDA
__device__
void add( int n, float* a, float* b ) {
    int i = threadIdx;
    if( i < n ) {
        a[i] += b[i];
    }
}

int main() {
    int n = 100;
    float* a; cudaMalloc(&a, n * sizeof(float));
    float* b; cudaMalloc(&b, n * sizeof(float));

    // ...

    add<<<(n + 31) & ~31, 32>>>(n, a, b);
}
```

```
// C++ AMP

int main() {
    extent<1> n(100);
    array_view<float,1> a(n);
    array_view<float,1> b(n);

    // ...
    parallel_for_each(n, [=](index<1> i) restrict(amp) {
        a[i] += b[i];
    });
}
```

Open Problems in Real-Time Rendering, SIGGRAPH 2016

49

Finally, we need some way to allocate and place resources in the different memory spaces of a heterogeneous system.

This is the part that current graphics APIs might already have a good enough answer to.

## **A heterogeneous compiler improves ease-of-use**

- **Invoking GPU code can be as simple as a function call**
- **Can share type/constant/function declarations**
  - **With guarantees about layouts matching**
- **Integrate into existing build process**

Having a true heterogeneous compiler could improve ease-of-use. As we've seen, it can make invoking a GPU kernel as simple as a function call.

More importantly, it can allow us to confidently share type, constant and function declarations between processors, with some level of guarantee that their layouts match.

I've seen lots of people try to share structs declarations between HLSL and C++ for their constant buffers, only to get bitten by some annoying layout difference. We shouldn't have to put up with that stuff.

# Open Problems

- **Can this be implemented as a layered tool?**
  - Minimize dependency on C++ standards process
- **Challenge: how does this apply to vertex/fragment shaders?**
  - Harder to express as “just a function call”
- **Extra credit: how similar are heterogeneous and distributed?**

Open Problems in Real-Time Rendering, SIGGRAPH 2016

51

Of course, there are some challenges to realizing this vision.

First and foremost, I think, is the question of whether this is something that can be done in a completely layered fashion, or whether we'd need to engage with the C++ standards process.

My (real) concern is that the real-time graphics community really just needs something minimal to let them get their work done, and wouldn't want to wait on each and every platform they target to get caught up with the latest language standard.

Meanwhile, the tendency of the standards body will be toward adding a bunch of library constructs that each and every compiler will need to implement (with varying quality).

Another challenge (and this one is more directed at the researchers out there) is how any of this applies to the graphics pipeline: vertex and fragment shaders, and so forth. It is all well and good for me to say that invoking the GPU should be “just a function call,” but that isn't the reality of how graphics programming works today, where there is a lot of additional state to contend with.

In fact, the entire direction we are taking with the new “modern” graphics APIs is towards explicitly exposing command buffers, so that the simplicity of the “just a function call” view is hard to align with the reality of the APIs.

# First-Class Support for Specialization and Composition

Open Problems in Real-Time Rendering, SIGGRAPH 2016

52

The next topic on our list of pain points was specializing and composing shader code.

## Current practice is ad hoc

- **Textual preprocessing, macros, and splicing**
  - Prone to typos/errors in generated code, variable capture
- **If we switch to C++: could use templates**
  - Replacing one ad hoc kludge with another

As a reminder, the current state of the art is purely text-based approaches, whether macros or pasting strings together.

These approaches are prone to typos, and errors in the generated code, which may not be immediately visible (e.g., if a variable defined under one `#define` is used under a different `#define`, but you almost always enable both together).

Also, macros have well-known issues such as variable capture.

If we switch to C++ for shaders, when we can also apply templates to these problems, but templates have a lot of issues that make them less than ideal for metaprogramming.

## Trends in other domains and languages

- **Run arbitrary code at compile time**
  - Varying levels of support in Rust, Scala, D, Haskell, Racket, Jai, ...
- **Quasiquotation to easily construct/manipulate code**
  - Lisp/Racket, Scala, Template Haskell, Rust, ...
- **Extensible syntax for simple DSL construction**
  - Research trend in performance-oriented DSLs

If we look at other languages in development, we can see a trend toward enabling metaprogramming by allowing arbitrary user-defined code to be run during compilation (the languages list this have a variety of very different mechanisms for this).

Similarly, many of these languages are beginning to support *quasiquotation* as a way to more easily construct and manipulate code during compile time. That sounds like a very fancy word (after all, it comes from the Lisp world), but really it just means having first-class language support for constructing and splicing code.

Finally, if we look over at the research world, we see that there is a trend toward languages with some kind of extensible syntax, which can be used to construct custom domain-specific languages.

There is a lot of recent work on what are being called “performance-oriented DSLs”.

# Performance-oriented DSLs for rapid design space exploration

Open Problems in Real-Time Rendering, SIGGRAPH 2016

55

I want to take the time here to draw a concrete connection, and point out that domain-specific languages can be an enabler for rapid design-space exploration. I'll use the next few slides to dig into what I mean...

# Design-space exploration

GI: baked, voxels, probes, screen-space?

Use GPU compute for culling/submission?

Tiled deferred or forward+ ?

**Manually code as many options as you can**

Skinning on CPU or GPU?

How to partition for multi-GPU?

Decoupled/texture-space shading?

Move some simulation/CI to servers, amortize across users?

Open Problems in Real-Time Rendering, SIGGRAPH 2016

56

As we covered earlier, the state of the art in real-time graphics today is that when you are faced with a tough design choice, you try to code as many options as you can and pick the best.

## A good DSL can allow for rapid iteration

- **A more compact notation (often declarative)**
  - For a class of workloads
- **Separate target-specific scheduling information**
  - Automated tools can synthesize or explore space of schedules
- **Generate final code from the compact description**
  - Leverage: small change in input can lead to large changes in output

So how does a DSL help you with that?

Well, in the simplest terms you can think of a DSL as *\*just\** a more compact notation for a particular class of workloads. Very often these representations are declarative (so they focus on *\*what\** not *\*how\**)

Many performance-oriented DSLs also separate out some of the target-specific scheduling information, so that you can apply different schedules to the same workload, or have tools automatically explore a space of schedules.

As a result of the compactness of the notation, you get a certain kind of “leverage,” where small changes to the input (the workload code, scheduling info, etc.) leads to a large change in the code that gets generated as output.

Because of this leverage, you can more rapidly try out different options that might otherwise be very far apart in the design space (in terms of “number of lines of code that I need to touch”)

## DSL approach paying off in other domains

- **Halide: generate high-performance image processing kernels**
  - Automatically schedule composition of many filters (SIGGRAPH 2016 paper)
- **TensorFlow: machine learning pipelines**
  - Automatically distribute and scale across multiple machines

This is an idea that is paying off in other domains.

The poster child for performance-oriented DSLs is probably the Halide project, for developing high-performance image processing kernels.

Halide allows a simple description of a kernel to be scheduled to produce optimized code for a variety of different CPU and GPU architectures.

This year at SIGGRAPH, we even have a paper that shows how a compiler to automatically schedule compositions of different Halide filters, without needing an expert programmer to tune each composition.

In the machine learning world, we have systems like TensorFlow, which use a declarative representation (in the middle ground between “library” and “DSL”), which allows it automatically schedule machine learning workloads across the CPU and GPU resources in your machine, or even across a distributed network of machines.

## Frameworks can speed up DSL creation

- **Terra: use Lua to metaprogram a C-like language**
  - Image processing (Darkroom), simulation (Ebb), optimization (opt)...
  - Effort to extend Terra to support graphics shaders (Kerry Seitz @ UC Davis)
- **AnyDSL/Impala: compile DSLs with partial specialization**
  - Image processing, multigrid solver, ray-tracing ...
- **C-Mera: use Lisp macros to metaprogram C/C++/GLSL/...**
  - Ray tracing, ...

Open Problems in Real-Time Rendering, SIGGRAPH 2016

59

Because of these kinds of benefits, we are also seeing research efforts towards building frameworks that allow new performance-oriented DSLs to be constructed more easily.

There's the Terra system, which uses Lua to metaprogram a low-level C-like language (called Terra). That system has been applied to a variety of problems, resulting in multiple SIGGRAPH papers.

Lua-Terra allows for arbitrary compile-time computation, and uses quasiquotation to generate and splice code.

I should call out that one of my collaborators, Kerry Seitz at UC Davis, has been working on extending Terra so that it can also be used for graphics shaders.

There's the AnyDSL project and the Impala language, which uses a technique known as "partial specialization" for metaprogramming; this is similar in its results to quasiquotation, but some would argue it is a more lightweight notation.

Finally, the C-Mera project goes right back to the roots of all this metaprogramming stuff, and lets you write C/C++ code using a Lispy syntax, so that you can metaprogram it using Lisp macros, and then generate ordinary

C/C++/GLSL/CUDA/etc. at the end

# Can we apply the DSL approach to graphics?

GI: baked, voxels, probes, screen-space?

Use GPU compute for culling/submission?

Tiled deferred or forward+ ?

**Rapidly generate and test many options**

Skinning on CPU or GPU?

How to partition for multi-GPU?

Decoupled/texture-space shading?

Move some simulation/GI to servers, amortize across users?

Open Problems in Real-Time Rendering, SIGGRAPH 2016

60

The question, then, for researchers is whether we could apply this same approach to real-time graphics.

Is it possible to build DSLs so that when faced with design choices like these, we can rapidly generate and test many options, instead of coding them all from scratch?

## Spire project is a first step

- **Intermediate representation for surface/material shaders**
- **Paper at SIGGRAPH this year (Wednesday)**
  - Yong He (CMU), Kayvon Fatahalian (CMU), Tim Foley (NVIDIA)
- **Explore scheduling choices that span multiple passes**
  - Compute shading terms at full resolution, or decoupled screen- or object-space
  - Composite BRDF layers per-fragment, or bake offline
  - Auto-tune for different HW targets, LOD, etc.
- **Challenge: scaling beyond surface/material shading**

Open Problems in Real-Time Rendering, SIGGRAPH 2016

61

As a small example, I'd like to point out the Spire project, being presented at SIGGRAPH this year, as a small step toward this goal, from some of my collaborators at CMU: Yong He and Kayvon Fatahalian.

Spire is an intermediate representation for surface/material shaders, that allows users to explore a space of scheduling choices that can span multiple rates in space/time, and even multiple rendering passes.

For example it is possible, with Spire, to rapidly try out computing particular shading terms at reduced resolution using decoupled screen- or object-space shading.

Looking forward, an important challenge is how we scale from a DSL like this for surface/material shading, to more complex parts of an engine's functionality.

# Conclusion

Open Problems in Real-Time Rendering, SIGGRAPH 2016

62

So, to wrap up...

# It is a great time for language innovation

- **Lot's of promising efforts out there**
  - Rust, Swift, Scala, Terra, Racket, ...
- **Very little focus on the needs of game/graphics developers**
  - Almost no recognition of heterogeneous CPU+GPU reality
- **Time for real-time graphics programmers to be heard**
  - Or, start coordinating to build the solutions we want

Open Problems in Real-Time Rendering, SIGGRAPH 2016

63

Now is a great time to be looking at languages. There is a lot of innovation going on out there, and a lot of new language projects that arise from developers in various fields deciding to scratch their own itch.

Unfortunately, when I survey the work going on out there, I find that there is almost no attention being paid to the needs of game and real-time graphics developers. Even more to the point: none of these language projects seems to accept the reality that we are all programming for heterogeneous architectures, with (at least) a CPU and GPU.

So as a small call to action, I will say that it is high time for real-time graphics programmers to make our voices heard, and get involved in the evolution of these languages.

Or, if none of them is suited to our needs, it is time for us to start coordinating on building the solutions we want for ourselves.

# Acknowledgements

- **Yong He, Kayvon Fatahalian (CMU)**

- **Kerry Seitz, John Owens (UC Davis)**

- **Adrian Sampson (Cornell)**

"Weep for Graphics Programming" <http://adriansampson.net/blog/opengl.html>

- **Aaron Lefohn and co. @ NVIDIA**

Open Problems in Real-Time Rendering, SIGGRAPH 2016

64

I'd like to thank my collaborators at various academic institutions, and my team at NVIDIA for their feedback that helped shape this presentation.

As a small parting shot, I'll direct people to this blog post from Adrian Sampson, entitled "Weep for Graphics Programming" which gives a really great example of what happens when somebody from another field (in this case, a programming language researcher) looks at what we put up with every day.

# Thank You

[tfoley@nvidia.com](mailto:tfoley@nvidia.com)

[@TangentVector](https://twitter.com/TangentVector)

Open Problems in Real-Time Rendering, SIGGRAPH 2016

65

Thanks for your time.

If you'd like to reach me, you can contact me by email at NVIDIA, or on Twitter.

## References

- **“Weep for Graphics Programming”**: <http://adriansampson.net/blog/opengl.html>
  - Adrian Sampson
- **Halide**: [halide-lang.org](http://halide-lang.org)
- **TensorFlow**: [tensorflow.org](http://tensorflow.org)
- **Terra**: [terralang.org](http://terralang.org)
- **AnyDSL/Impala**: [anydsl.github.io](https://github.com/anydsl)
- **C-Mera**: [github.com/kiselgra/c-mera](https://github.com/kiselgra/c-mera)
- **Spire**: [github.com/csyonghe/Spire](https://github.com/csyonghe/Spire)