An interesting time in graphics programming

Many platforms, and even more APIs

- Vulkan 1.0 released just weeks ago

Still programming shaders like it’s 2002

- Author shaders in D3D9-era HLSL
- Translate to GLSL, Metal, etc.
Isn’t SPIR-V going to solve all of this?

Vulkan’s intermediate language (IL) for shaders

- Documented binary format, consumed by drivers
- Intended to support alternative front-ends

May encourage other APIs to follow suit

Not a panacea today

- Somebody still has to write all the new front- and back-ends
Benefits of building your own tools

A shader compiler tailored to your engine

Not a language designed for somebody else’s API

Target a wide range of platforms

Including future rendering pipelines for, e.g., VR

Transform and manipulate shaders

Example: Mechanically generate shader LODs
This Talk

Two in-progress research projects

   Exploring possible approaches to building shader tools

Rapid exploration of scheduling choices

   In context of future/extended pipelines

Framework for rapidly building custom shader language/tools

   Shaders as a performance-oriented domain-specific language (DSL)
Rapid Exploration of Shader Scheduling Choices

Collaborators: Yong He, Kayvon Fatahalian (CMU)
Pipelines with new shading rates

- Vertex Shader
- Coarse-Rate Shader
- Fragment Shader
- Pixels

texture space?
quarter-resolution screen space?
“foveated” rendering?
Shading reuse for stereo (VR)

Vertex Shader → Eye-Shared Shader → Per-Eye Shader → Pixels

share diffuse, compute specular per-eye?
Define engine-specific pipeline

Source Assets → Vertex Baking → Texture Baking → Vertex Shader → Fragment Shader → Pixels

- Offline asset processing
- Runtime (GPU) processing
Write shaders against that pipeline

```glsl
shader Terrain using MyEnginePipeline
{
    @MeshVertex vec2 uv;
    @MaterialUniform sampler2D albedoMap1, albedoMap2;
    @MaterialUniform sampler2D mixMap;

    float mixFactor = texture(mixMap, uv);

    vec4 albedo
    {
        vec4 c1 = texture(albedoMap1, uv);
        vec4 c2 = texture(albedoMap2, uv);
        return mix(c1, c2, mixFactor);
    }

    // ...
}
```
Shader = Dataflow Graph

lightDir → nDotL
normal → nDotL
uv → mixFactor
mixMap → mixFactor
albedoMap1 → mixFactor
albedoMap2 → mixFactor

nDotL → albedo
mixFactor → albedo
albedo → output
Scheduling = Placing Computation in Stages

Frame Uniform
- lightDir

Mesh Vertex
- normal
- uv

Material Uniform
- mixMap
- albedoMap1
- albedoMap2

Vertex Baking
- nDotL

Vertex Shader
- mixFactor

Texture Baking
- albedo

Fragment Shader
- output

- Frame Uniform: lightDir
- Mesh Vertex: normal, uv
- Material Uniform: mixMap, albedoMap1, albedoMap2
- Vertex Baking
- Vertex Shader: nDotL
- Texture Baking: mixFactor, albedo
- Fragment Shader: output
Even lower LOD? Do lighting per-vertex.
What can you do with a representation like this?

Rapidly explore performance/quality trade-offs

  Use a UI tool to explore different shader variants

Auto-tune scheduling choices

  Meet performance constraints on a particular target

Mechanically apply LOD choices to many shaders

  Guarantee that low LODs share same vertex/fragment code
Exploring Choices
(Multi-Rate / Object Space Pipeline)
Rapid Development of Engine-Specific Shading Tools

Collaborators: Kerry Seitz, John Owens (UC Davis)
Building engine-specific shader infrastructure

Front-end language

Integration with other systems

Back-end code generation
Shaders as domain-specific languages (DSLs)

Performance-oriented DSLs

Halide (for image processing) is poster child

Terra: framework for easily constructing DSLs

Low-level C-like language

Use Lua scripts to extend language/compiler, generate code

Supports LLVM code generation out of the box
Building Shader Tools in Terra

Write shader code directly in the Terra language

Can re-use types, functions between app (CPU) and shaders (GPU)

Shader-specific features implemented as syntax extensions

Lua scripts that run inside the compiler
Building Shader Tools in Terra

Built-in support for HLSL/GLSL code generation

Additional tools can parse/inspect/manipulate shaders

Using a Lua library API

Artist UI

Statically-typed engine interface

Asset processing tools
What is the right shader infrastructure for you?

Looking ahead: opportunities and challenges

Don’t expect Khronos, Microsoft, etc. to build all the pieces

Opportunity to build what you really want

Researchers can help accelerate and guide your efforts

Programming models that are ready for new HW and SW pipelines

Frameworks to rapidly build engine-specific compilers and tools
Thank You

tfoley@nvidia.com