Shading Languages

Tim Foley
Why Shading Languages?
Why Shading Languages? DSLs?

Productivity
Performance
Productivity

Build shaders from re-usable components

Performance

Specialize code to data

Exploit specialized hardware
Productivity

Build shaders from re-usable components

Based on model of problem domain

Performance

Specialize code to data

Exploit specialized hardware

Based on model of solution domain
Productivity

- Build shaders from re-usable components
- Based on model of problem domain

Performance

- Specialize code to data
- Exploit specialized hardware
- Based on model of solution domain
Building Shaders from Components
What kinds of components are needed?

What form do components take?

How do we combine components?
RenderMan Shader Types

- External Volume
  - Reflected ray color
- Internal Volume
  - Transmitted ray color
  - Attenuated reflection color
- Light Sources
  - Attenuated transmission color
  - Light colors
- Displacement
  - Displaced surface
- Surface
  - Surface color
- Atmosphere
  - Apparent surface color
Shader Components in a Modern Game

- Materials (pattern generation / BSDFs)
- Lights / Shadows
- Volumes (e.g., fog)
- Animation
- Geometry (e.g., tessellation, displacement)
- “Camera” (rendering mode)

  2D/cubemap/stereo, color/depth output
What kinds of components are needed?

What form do components take?

How do we combine components?
What form do shader components take?

Function/procedure?

Dataflow graph?

Class?
Make a shader look like a procedure

Represent with a dataflow graph IR (shader graph)

Compose and specialize using class-like concepts
Shade Trees

[Cook 1984]
Shade Trees

```c
float a = 0.5, s = 0.5;
float roughness = 0.1;
float intensity;

// Color
float metal_color = (1,1,1);

// Calculate intensity
intensity = a * ambient() + 
    s * specular(normal, viewer, roughness);

final_color = intensity * metal_color;
```
float a = 0.5, s = 0.5;
float roughness = 0.1;
float intensity;
color metal_color = (1,1,1);
intensity = a*ambient() +
            s*specular(normal,viewer,roughness);
final_color = intensity * metal_color;
float a = 0.5, s = 0.5;
float roughness = 0.1;
float intensity;
color metal_color = (1,1,1);
intensity = a * ambient() +
    s * specular(normal, viewer, roughness);
final_color = intensity * metal_color;
Shader Graphs are Composable

Point Light

- lightPos
- delta
- intensity
- Li
- P
- L
- N

Displacement

- P
- N

Diffuse Material

- L
- Li
- N
- NdotL
- final_color
Shader Graphs are Composable

Point Light

Displacement

Diffuse Material

lightPos

delta

P

L

intensity

Li

L

N

final_color

N
delta

lightPos

P

N

NdotL

P

N

Li

Li
Shader Graphs are Composable

Point Light

- lightPos
- delta
- Li
- intensity

Displacement

- P
- N

Diffuse Material

- NdotL
- final_color
Exploiting Specialized Hardware

Specializing Code to Data
RenderMan Shading Language

[Hanrahan and Lawson 1990]
uniform vector L;
varying vector N;

...

L = normalize(L);

...

N = normalize(N);
varying float NdotL = N . L;
uniform vector L;
varying vector N;

...  
L = normalize(L);
...

N = normalize(N);
varying float NdotL = N . L;
Split shader into uniform and varying parts

```plaintext
uniform
code

varying
code

shader
```
Create an instance of the shader class

```
Create an instance of the shader class

uniform
code

varying
code

uniform
args

shader

instance
```
Create an instance of the shader class

- shader class has multiple methods
- shader parameters are instance variables
Specialize as more information becomes known

- shader
- instance
- bound instance
- elaborated instance

- uniform code
- varying code
- uniform args
- geometry
- vertex args
Specialize as more information becomes known

find parameter values using prototype chain
Intermission:

Let’s Talk About Staging
Staging Transformations

Given a function of two parameters $f(x,y)$

Where $x$ might represent information known “before” $y$

Compute functions $f_1(x)$ and $f_2(t,y)$

Such that $f_2(f_1(x),y) = f(x,y)$

Can generalize to $N$ stages

[Jørring and Scherlis 1986]
Examples of \( f(x,y) \)

**Regular expression matching**

\( x \) is regular expression, \( y \) is string to match against

**RenderMan Shading Language**

\( x \) is *uniform* parameters, \( y \) is *varying* parameters
Examples of $f(x,y)$

Regular expression matching

- $x$ is regular expression, $y$ is string to match against

RenderMan Shading Language

- $x$ is uniform parameters, $y$ is varying parameters

...
Goal

Try to do “as much as possible” in f1
A Trivial Solution

```plaintext
function f(x, y)
    ...
end

function f1(x)
    return x
end

function f2(t, y)
    return f(t, y);
end
```
A Trivial Solution

```plaintext
function f(x, y)
    ...
end

function f1(x)
    return x
end

function f2(t, y)
    return f(t, y);
end
```

This isn’t “as much as possible”
Another Trivial Solution

function f(x,y)
    ...
end

function f1(x)
    return function(y)
        return f(x,y)
    end
end

function f2(t, y)
    return t(y);
end
function f(x,y)
    ...
end

function f1(x)
    return function (y)
        return f(x,y)
    end
end

function f2(t, y)
    return t(y);
end
A Terra Solution

terra f(x, y)
    ...
end

function f1(x)
    return terra y)
        return f(x, y)
    end
end

function f2(t, y)
    return t(y);
end
A Terra Solution

terra f(x,y)
    ...
end

function f1(x)
    return terra(y)
    return f(x,y)
end

function f2(t, y)
    return t(y);
end

compiler might do inlining, constant folding, etc.
More Idiomatic Terra

function f_staged(x,y)
    ...
end

function f1(x)
    return terra(y)
    return [f_staged(x,y)]
end
end

function f2(t, y)
    return t(y);
end
More Idiomatic Terra

```plaintext
function pow_staged(n, y)
    if n == 0 then return `1.0
    else return `(y * [pow_staged(n-1, y)])
end

function make_pow(n)
    return terra(y)
        return [pow_staged(n, y)]
    end
end

function f2(t, y)
    return t(y);
end
```
Explicit Staging Annotations

```python
function pow_staged(n,y)
    if n == 0 then return 1.0
    else return (y * [pow_staged(n-1,y)]) end
end
```
Staged vs. Unstaged

function pow (n,y)
    if n == 0 then return 1.0
    else return (y * pow (n-1,y)) end
end
Staged Programming
but not
Staged Metaprogramming
Old Goal

Try to do “as much as possible” in f1
Revised Goals

Try to do “as little as possible” in f2

Then, try to do “as little as possible” in f1

Then, try to do “as much as possible” when generating f1, f2
Explicit Staging Annotations

Quote and splice are one option

Delay and force is another

\[
\text{delay}(\text{exp}) \leftrightarrow \text{function()} \text{ return } \text{exp end}
\]

\[
\text{force}(\text{exp}) \leftrightarrow \text{exp()}
\]

Rate qualifiers are yet another

uniform and varying

Appear to be related to “world” type in modal type theories

[“Modal Types for Mobile Code” Muphy 2008]
Real-Time Shading Language

[Proudfoot et al. 2001]
surface shader float4 Simple( ... )
{
    constant float3 L_world = normalize({1, 1, 1});

    primitivegroup matrix4 viewProj = view * proj;

    vertex float4 P_proj = P_world * viewProj;
    vertex float NdotL = max(dot(N_world, L_world), 0);

    fragment float4 diffuse = texture(diffuseTex, uv);
    fragment float4 color = diffuse * NdotL;

    return color;
}
surface shader float4 Simple( ... )
{
    constant float3 L_world = normalize({1, 1, 1});

    primitivegroup matrix4 viewProj = view * proj;

    vertex float4 P_proj = P_world * viewProj;
    vertex float NdotL = max(dot(N_world, L_world), 0);

    fragment float4 diffuse = texture(diffuseTex, uv);
    fragment float4 color = diffuse * NdotL;

    return color;
}
surface shader float4 Simple( ... )
{
  constant float3 L_world = normalize({1, 1, 1});

  primitivegroup matrix4 viewProj = view * proj;

  vertex float4 P_proj = P_world * viewProj;
  vertex float NdotL = max(dot(N_world, L_world), 0);

  fragment float4 diffuse = texture(diffuseTex, uv);
  fragment float4 color = diffuse * NdotL;

  return color;
}
surface shader float4 Simple( ... )
{
    constant float3 L_world = normalize({1, 1, 1});
    primitivegroup matrix4 viewProj = view * proj;

    vertex float4 P_proj = P_world * viewProj;
    vertex float NdotL = max(dot(N_world, L_world), 0);

    fragment float4 diffuse = texture(diffuseTex, uv);
    fragment float4 color = diffuse * NdotL;

    return color;
}
surface shader float4 Simple( ... )
{
    constant float3 L_world = normalize({1, 1, 1});

    primitivegroup matrix4 viewProj = view * proj;

    vertex float4 P_proj = P_world * viewProj;
    vertex float NdotL = max(dot(N_world, L_world), 0);

    fragment float4 diffuse = texture(diffuseTex, uv);
    fragment float4 color = diffuse * NdotL;

    return color;
}
Map to Shader Graph

- L_world
- P_world
- N_world
- NdotL
- P_proj
- viewProj
- proj
- view
- color
- diffuse
- diffuseTex
- uv
- N_world
- L_world
- L_world
- N_world
- NdotL
- P_world
- P_world
- P_proj
- viewProj
- proj
- view
- L_world
- N_world
- NdotL
- P_world
- P_world
- P_proj
- viewProj
- proj
- view
- L_world
- N_world
- NdotL
- P_world
- P_world
- P_proj
- viewProj
- proj
- view
- L_world
- N_world
- NdotL
- P_world
- P_world
- P_proj
- viewProj
- proj
- view
Color by Rates

\[ \text{color} \leftarrow \text{diffuse} \leftarrow \text{diffuseTex} \leftarrow \text{uv} \leftarrow \text{NdotL} \leftarrow \text{P_world} \leftarrow \text{viewProj} \leftarrow \text{proj} \leftarrow \text{view} \leftarrow \text{L_world} \]

\[ \text{N_world} \leftarrow \text{proj} \]

\[ \text{P_proj} \leftarrow \text{viewProj} \]
Color by Rates

constant

\[ L_{\text{world}} \rightarrow \text{proj} \rightarrow \text{view} \rightarrow \text{viewProj} \rightarrow P_{\text{proj}} \rightarrow P_{\text{world}} \]

\[ N_{\text{world}} \rightarrow \text{NdotL} \rightarrow \text{uv} \rightarrow \text{diffuse} \rightarrow \text{diffuseTex} \rightarrow \text{color} \]
Color by Rates

Constant

Primitive Group

- view
- proj
- viewProj
- P_proj
- P_world
- N_world
- NdotL
- uv
- diffuseTex
- diffuse
- color
- L_world

NVIDIA
Color by Rates

constant

primitive group

vertex

\[
\text{view} \rightarrow \text{viewProj} \rightarrow \text{P_proj} \rightarrow \text{P_world} \rightarrow \text{N_world} \rightarrow \text{NdotL} \rightarrow \text{uv} \rightarrow \text{diffuse} \rightarrow \text{color}
\]

\[
L_{\text{world}} \rightarrow \text{diffuseTex}
\]
Color by Rates

constant

primitive group

vertex

fragment

L_world

view

proj

viewProj

P_proj

P_world

N_world

NdotL

uv

diffuseTex
diffuse

color
Partition
Partition

constant

primitive group

vertex

fragment

\( L_{\text{world}} \)

\( \text{view} \)

\( \text{viewProj} \)

\( \text{proj} \)

\( P_{\text{proj}} \)

\( P_{\text{world}} \)

\( N_{\text{world}} \)

\( \text{NdotL} \)

\( \text{uv} \)

\( \text{diffuseTex} \)

\( \text{color} \)

\( \text{diffuse} \)
Spark

[Foley and Hanrahan 2011]
Shader Components in a Modern Game

Materials (pattern generation / BSDFs)

Lights / Shadows

Volumes (e.g., fog)

Animation

Geometry (e.g., tessellation, displacement)

“Camera” (rendering mode)

2D/cubemap/stereo, color/depth output
define shader graphs as classes

compose with inheritance
Define Shader Graphs as Classes

abstract mixin shader class SimpleDiffuse : D3D11DrawPass
{
    input @Uniform float3 L_world;
    abstract @FineVertex float3 N_world;
    virtual @Fragment float4 diffuse = float4(1.0f);

    @Fragment float NdotL = max(dot(L_world, N_world), 0.0f);
    @Fragment float4 color = diffuse * NdotL;

    output @Pixel float4 target = color;
}
Define Shader Graphs as Classes

abstract mixin shader class SimpleDiffuse : D3D11DrawPass {
    input @Uniform float3 L_world;
    abstract @FineVertex float3 N_world;
    virtual @Fragment float4 diffuse = float4(1.0f);

    @Fragment float NdotL = max(dot(L_world, N_world), 0.0f);
    @Fragment float4 color = diffuse * NdotL;

    output @Pixel float4 target = color;
}
Define Shader Graphs as Classes

shader class SimpleDiffuse
{
    ...
}
shader class CubicGregoryACC { ... }
shader class MyTextureMapping { ... }
shader class ScalarDisplacement { ... }
...
Compose With Inheritance

shader class Composed
extends SimpleDiffuse

{}
Compose With Inheritance

shader class Composed
extends SimpleDiffuse,
CubicGregoryACC

{}
shader class Composed
   extends SimpleDiffuse, CubicGregoryACC, TextureMapping
{
}
Compose With Inheritance

shader class Composed
extends SimpleDiffuse,
CubicGregoryACC,
TextureMapping,
ScalarDisplacement

{}
Cg, HLSL, GLSL, etc.

[Mark et al. 2003] [Microsoft] [OpenGL ARB] [Sony, Apple, ...]
Assumption so far...

Decompose program according to problem domain

Material, lights, animation, etc.

Compiler uses rates/staging to map to solution domain

Specialization

SIMD, GPU
New assumption

Decompose program according to solution domain

Programmable stages of the GPU graphics pipeline

Programmer must use ??? to align with problem domain

Procedural abstraction?
Object-oriented programming?
Preprocessor?
New assumption

Decompose program according to solution domain

Programmable stages of the GPU graphics pipeline

Programmer must use ??? to align with problem domain

Procedural abstraction?

Object-oriented programming?

Preprocessor?
Looking Ahead
“Just write shaders in C++”
“Just write shaders in C++”
the same language as your application
Most shader code is “just code”
Most shader code is “just code”

Works for CPU, GPU compute, graphics
Write “just code” part in language X

Write shader-specific parts in EDSL, implemented in language X
Write “just code” part in Terra
Write shader-specific parts in EDSL, implemented in Terra
Conclusion

Productivity

Decompose program according to problem domain

Performance

Use rates (staging) to guide code generation

Generality

Embed shaders into applications languages as EDSLs
Thank You
tfoley@nvidia.com
References

Shade Trees
[Robert L. Cook 1984]

A Language for Shading and Lighting Calculations
[Pat Hanrahan and Jim Lawson 1984]

Compilers and Staging Transformations
[Ulrik Jørring and William L. Scherlis 1986]

A Real-Time Procedural Shading System for Programmable Graphics Hardware
[Kekoa Proudfoot, William R. Mark, Svetoslav Tvetkov, and Pat Hanrahan 2001]

Spark: Modular, Composable Shaders for Graphics Hardware
[Tim Foley and Pat Hanrahan 2011]

Cg: A System for Programming Graphics Hardware in a C-like Language
[William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard 2003]

Mobile Types for Mobile Code
[Tom Murphy VII 2008]
Backup
Cg, HLSL, GLSL
The Direct3D 11 Pipeline
Programmable Stages

Vertex Shader → Hull Shader → Domain Shader → Geometry Shader → Fragment Shader
## Programmable Stages

<table>
<thead>
<tr>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Shader</td>
</tr>
<tr>
<td>Hull Shader</td>
</tr>
<tr>
<td>Domain Shader</td>
</tr>
<tr>
<td>Geometry Shader</td>
</tr>
<tr>
<td>Fragment Shader</td>
</tr>
</tbody>
</table>
### Problem-Domain Components

<table>
<thead>
<tr>
<th>Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Shader</td>
</tr>
<tr>
<td>Hull Shader</td>
</tr>
<tr>
<td>Domain Shader</td>
</tr>
<tr>
<td>Geometry Shader</td>
</tr>
<tr>
<td>Fragment Shader</td>
</tr>
</tbody>
</table>

![Image](image_url)
Problem-Domain Components

- **Vertex Shader**
- **Hull Shader**
  - CubicGregoryACC
- **Domain Shader**
  - ScalarDisplacement
- **Geometry Shader**
  - RenderToCubeMap
- **Fragment Shader**
  - SimpleDiffuse
  - PointLight

TextureMapping
Cross-Cutting Concerns

- **Vertex Shader**: TextureMapping
- **Hull Shader**: CubicGregoryACC, TextureMapping
- **Domain Shader**: CubicGregoryACC, TextureMapping, ScalarDisplacement
- **Geometry Shader**: RenderToCubeMap, TextureMapping
- **Fragment Shader**: PointLight, SimpleDiffuse, TextureMapping
Coupling

- **Vertex Shader**: TextureMapping
- **Hull Shader**: CubicGregoryACC, TextureMapping
- **Domain Shader**: CubicGregoryACC, TextureMapping, ScalarDisplacement
- **Geometry Shader**: RenderToCubeMap, TextureMapping
- **Fragment Shader**: PointLight, SimpleDiffuse, TextureMapping
Combinatorial Explosion

Vertex Shader
- TextureMapping

Hull Shader
- CubicGregoryACC
- TextureMapping

Domain Shader
- CubicGregoryACC
- TextureMapping
- ScalarDisplacement

Geometry Shader
- RenderToCubeMap
- TextureMapping

Fragment Shader
- PointLight
- SimpleDiffuse
- TextureMapping
Terra-Integrated Shaders
local pipeline ToyPipeline {
    uniform Uniforms {
        mvp : mat4;
    }
    input P_model : vec3;
    input N_model : vec3;
    output C : vec4;

    varying N = N_model;

    vertex code
        gl_Position = mvp * make_vec4(P_model, 1.0);
    end

    fragment code
        C = make_vec4(normalize(N), 1.0);
    end
}
local pipeline ToyPipeline {
    ...
}

terra init()
    ...
    GL.glShaderSource(vertexShader, 1, [ToyPipeline.vertex.glsl], nil);
    ...
end

terra render()
    GL.glVertexAttribPointer([ToyPipeline.P_model.__location], ...);
    GL.glBindBufferBase(
        GL.GL_UNIFORM_BUFFER,
        [ToyPipeline.Uniforms.__binding],
        uniformBuffer);
end
local pipeline ToyPipeline {
  ...
}

terra init()
  ...
  toyPipeline = ToyPipeline.new();
  toyPipeline.P_model.set( vertexData, ... );
end

terra render()
  toyPipeline.Uniforms.mvp.set( camera.modelViewProj );
  Gfx.push(toyPipeline);
  Gfx.draw();
end
local pipeline Camera { ... }
local pipeline SkeletalAnimation { ... }
local pipeline PhongMaterial { ... }
local pipeline PointLight { ... }

... 

terra render()
    for m = 0, materialCount do
        var mat = &materials[m];
        Gfx.push(mat.pipeline);
        for n = 0, mat.meshCount do
            Gfx.push(mat.meshes[n].pipeline);
            Gfx.draw();
            Gfx.pop();
        end
        Gfx.pop();
    end
end
end
Staging Isn’t Always For Performance
Shade Trees

Point Light

Displacement

... 

Diffuse Material

final_color

lightPos

intensity

delta

Li

Li

lightPos

delta

L

N

P

NdotL

...
What if we have multiple lights?

Point Light
- lightPos
- delta
- Li
- intensity

Displacement
- P
- N

Diffuse Material
- NdotL
- final_color
What if we have multiple lights?

Point Light

lightPos ➔ delta ➔ Li

Point Light

intensity ➔ delta ➔ Li

Diffuse Material

NdotL ➔ + ➔ final_color

Displacement

P ➔ N

...
RenderMan Shading Language
Surface and Light Shaders
Linkage via control-flow constructs

surface shader

color C = 0;
illuminance( P, N, Pi/2 ) {
    L = normalize(L);
    C += Kd * Cd * Cl * length(L ^ T);
}

light shader(s)

illuminate( P, N, beamangle ) {
    Cl = (intensity*lightcolor)/(L . L);
}

illuminate( P, N, beamangle ) {
    ...
}
Linkage via control-flow constructs

**Surface shader**

```plaintext
color C = 0;
illuminance( P, N, Pi/2 ) {
    L = normalize(L);
    C += Kd * Cd * Cl * length(L ^ T);
}
```

**Light shader(s)**

```plaintext
illuminant( P, N, beamangle ) {
    Cl = (intensity*lightcolor)/(L . L);
}
illuminant( P, N, beamangle ) {
    ...
}
```
Linkage via control-flow constructs

color C = 0;

illuminance( P, N, Pi/2 ) {
    L = normalize(L);
    C += Kd * Cd * Cl * length(L ^ T);
}

illuminance( P, N, beamangle ) {
    Cl = (intensity*lightcolor)/(L . L);
}

illuminate( P, N, beamangle ) {
    ... 
}

Key:

- keyword
- type
- constant
Linkage via control-flow constructs

surface shader

color C = 0;
illuminance( P, N, Pi/2 ) {
    L = normalize(L);
    C += Kd * Cd * Cl * length(L ^ T);
}

light shader(s)

illuminate( P, N, beamangle ) {
    Cl = (intensity*lightcolor)/(L . L);
}

illuminate( P, N, beamangle ) {
    ...
}
Linkage via control-flow constructs

surface shader

color C = 0;
illuminance (P, N, Pi/2) {
  L = normalize(L);
  C += Kd * Cd * Cl * length(L ^ T);
}

light shader(s)

illuminate (P, N, beamangle) {
  Cl = (intensity*lightcolor)/(L . L);
}

illuminate (P, N, beamangle) {
  ...
}
Could re-cast as higher-order functions

surface shader

color C = 0;
illuminance( P, N, Pi/2 ) {
    L = normalize(L);
    C += Kd * Cd * Cl * length(L ^ T);
}

light shader(s)

illuminant( P, N, beamangle ) {
    Cl = (intensity*lightcolor)/(L . L);
}
Could re-cast as higher-order functions

**surface shader**

```plaintext
color C = 0;
illuminance( P, N, Pi/2, function(L, Cl) {
    L = normalize(L);
    C += Kd * Cd * Cl * length(L ^ T);
});
```

**light shader(s)**

```plaintext
illuminate( P, N, beamangle, function(L) {
    Cl = (intensity*lightcolor)/(L . L);
});
```
Could re-cast as higher-order functions

**surface shader**

```plaintext
color C = 0;
illuminance( P, N, Pi/2, function(L, Cl)
   L = normalize(L);
   C += Kd * Cd * Cl * length(L ^ T);
});
```

**light shader(s)**

```plaintext
illuminate( P, N, beamangle, function(L) {
   Cl = (intensity*lightcolor)/(L . L);
});
```

---

**Key:**
- **keyword**
- **type**
- **constant**

**closure** to apply to each illumination sample
RTSL perlight rate

Computations that depend on both surface and light

System instantiates this sub-graph for each light

Sums results when converting per-light to fragment

In Spark, can implement \texttt{@Light} in user space
Modern renderers need different decomposition

Physically-based rendering

Want to guarantee energy conservation, etc. of BSDFs

Ray tracing

Renderer wants to control sampling, integration, scheduling of rays
Decompose surface shader into

Pattern generation

May be authored by artists

Might not even need a language

BSDF evaluation, integration, etc.

Authored by programmers, or technical artists

Typically only need a few (diffuse, dielectric, skin, ...)
Unity Standard Shader

Artist only sets textures, colors

Covers most use cases
Unreal Engine Material Editor

Graphical DSL

Also used for

- Animation
- Scripting
- Audio mixing

...
Open Shading Language (OSL)

surface Glass(
    color Kd = 0.8,
    float ior = 1.45,
    output closure color bsdf = 0)
{
    float fr = FresnelDielectric(I, N, ior);
    bsdf = Kd * (fr*reflection(N) + (1-fr)*refraction(N, ior));
}
Open Shading Language (OSL)

```plaintext
surface Glass(
  color Kd = 0.8,
  float ior = 1.45,
  output closure color bsdf = 0)
{
  float fr = FresnelDielectric(I, N, ior);
  bsdf = Kd * (fr*reflection(N) + (1-fr)*refraction(N, ior));
}
```

shader outputs a “radiance closure,” to be scheduled by renderer

closures created with built-in functions, then combined with operators like +
Two-Stage Material Shading Pipeline

Pattern Generation → Integrator → BSDF Evaluation
Automatic Rate Placement
Rates are a way to express scheduling
Decouple algorithm from schedule?

Automatically generate a good schedule?
A System for Rapid, Automatic Shader Level-of-Detail
[He, Foley, Tatarchuk, Fatahalian 2015]
Shader Simplification

1.7ms/frame 0.8ms/frame
Observation:

The best simplifications tend to come from reducing the rate at which a term is computed.
Move fragment code to vertex shader

Move vertex code to “parameter” shader
Project started with vertex+fragment shaders

Next step is “rate-less” pipeline shaders
Encoding algorithm choice
expensive = computeExpensiveBRDF(N, L, p1, p2, ...)

color = expensive
expensive = computeExpensiveBRDF(N, L, p1, p2, ...)
cheap = computeCheapBRDF(N, L, param)
color = [choice(`cheap, `expensive)]
expensive = computeExpensiveBRDF(N, L, p1, p2, ...)

color = [choice(`expensive, moveToVertex(`expensive))]
expensive = computeExpensiveBRDF(N, L, p1, p2, ...)
cheap = computeCheapBRDF(N, L, [fitParameter(`expensive)])
color = [choice(`cheap, `expensive)]
Explicit choices can make auto-tuning tractable