Shading Languages

Tim Foley

NVIDIA

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

Shader Graphs

Performance

Specialize code to data

Exploit specialized hardware

Based on model of solution domain

Rates of Computation

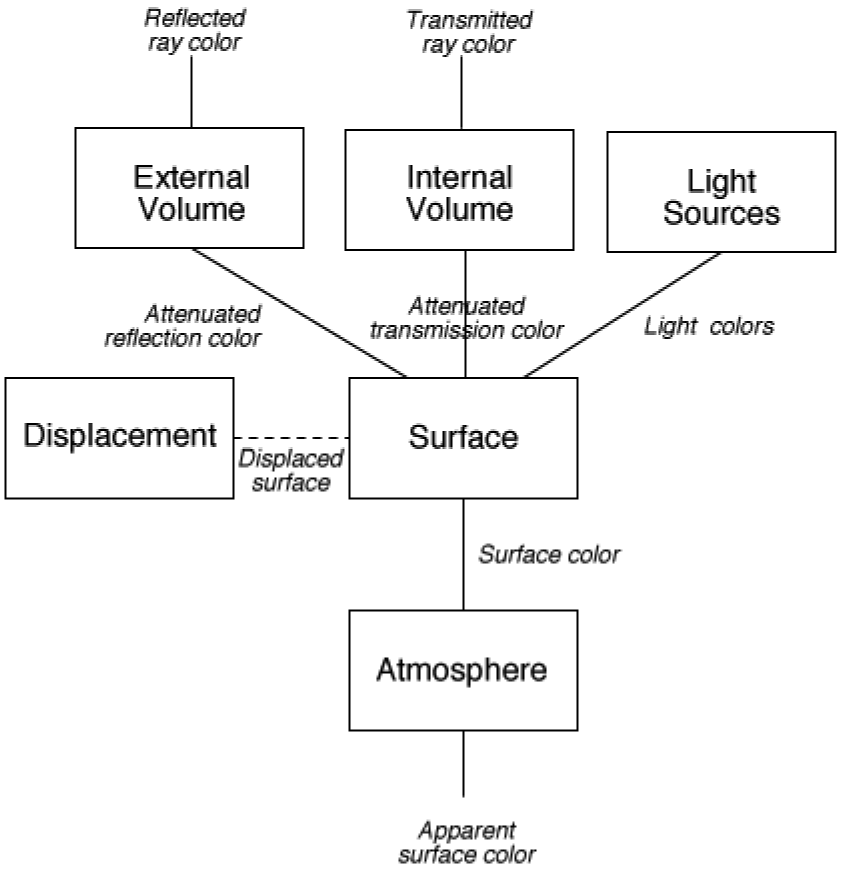
Building Shaders from Components

What kinds of components are needed?

What form do components take?

How do we combine components?

RenderMan Shader Types



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

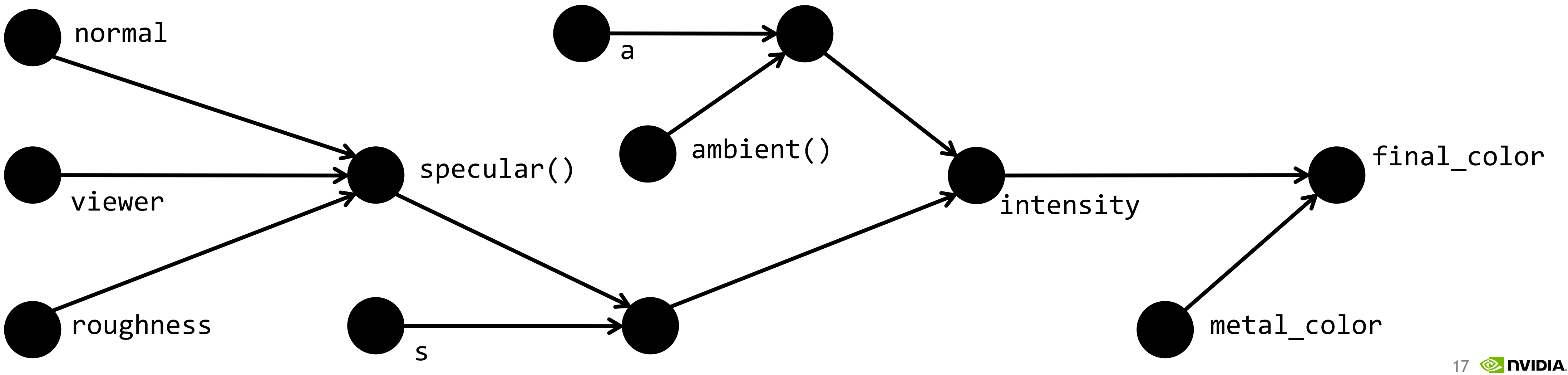
```
float a = 0.5, s = 0.5;
float roughness = 0.1;
float intensity;
color metal_color = (1,1,1);
intensity = a*ambient() +
    s*specular(normal,viewer,roughness);
final_color = intensity * metal_color;
```

Key:
type
constant

Shade Trees

```
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;
```

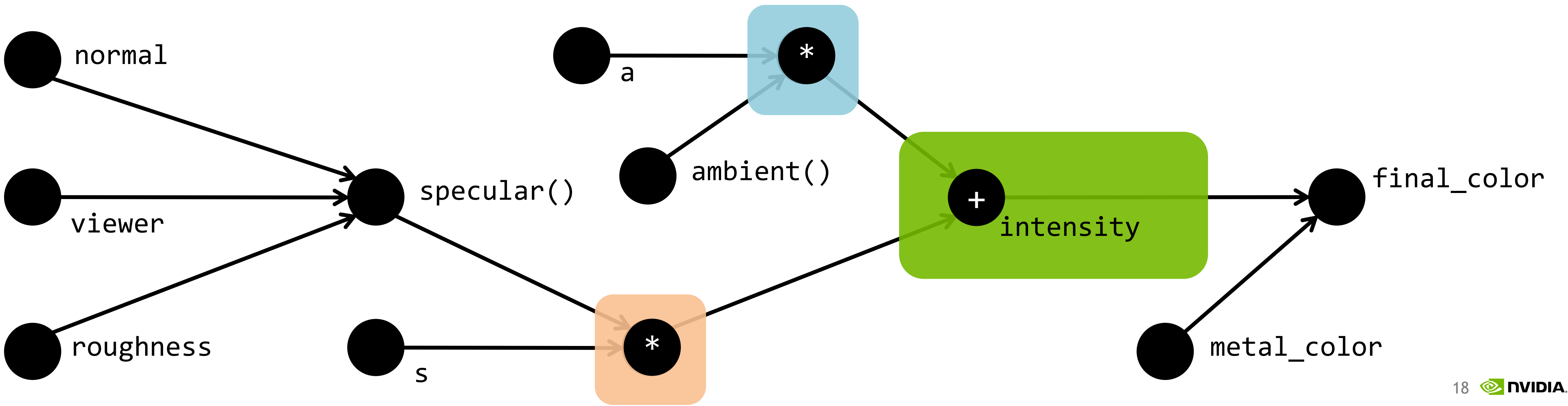
Key:
type
constant



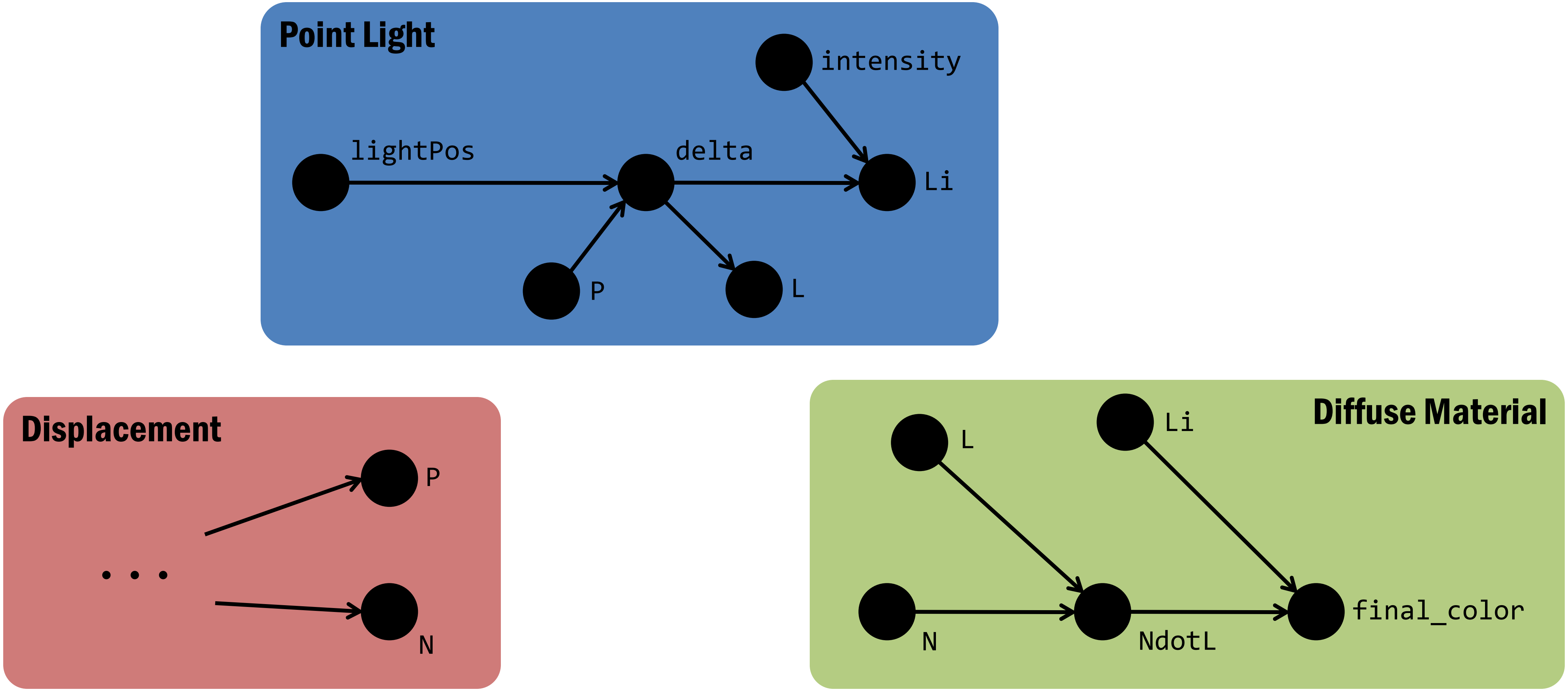
Shade Trees

```
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;
```

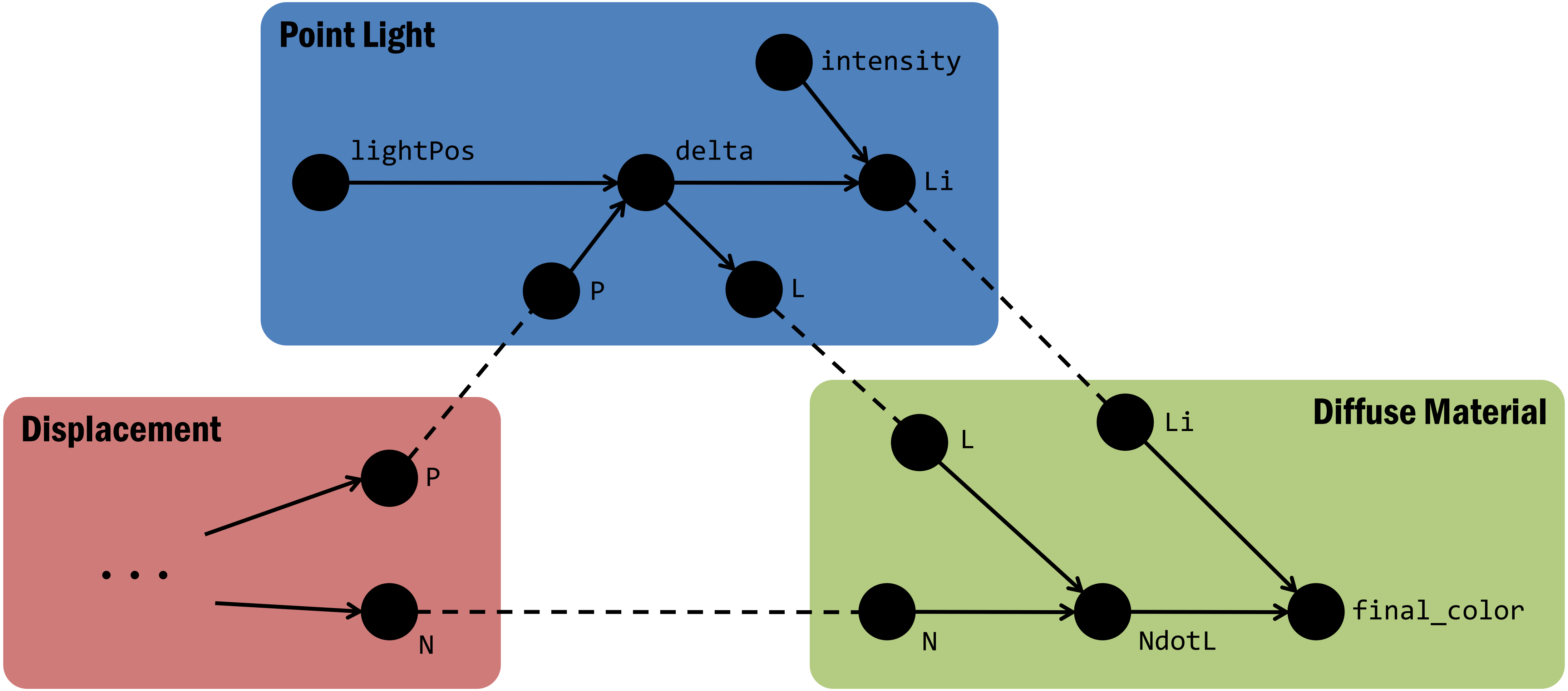
Key:
type
constant



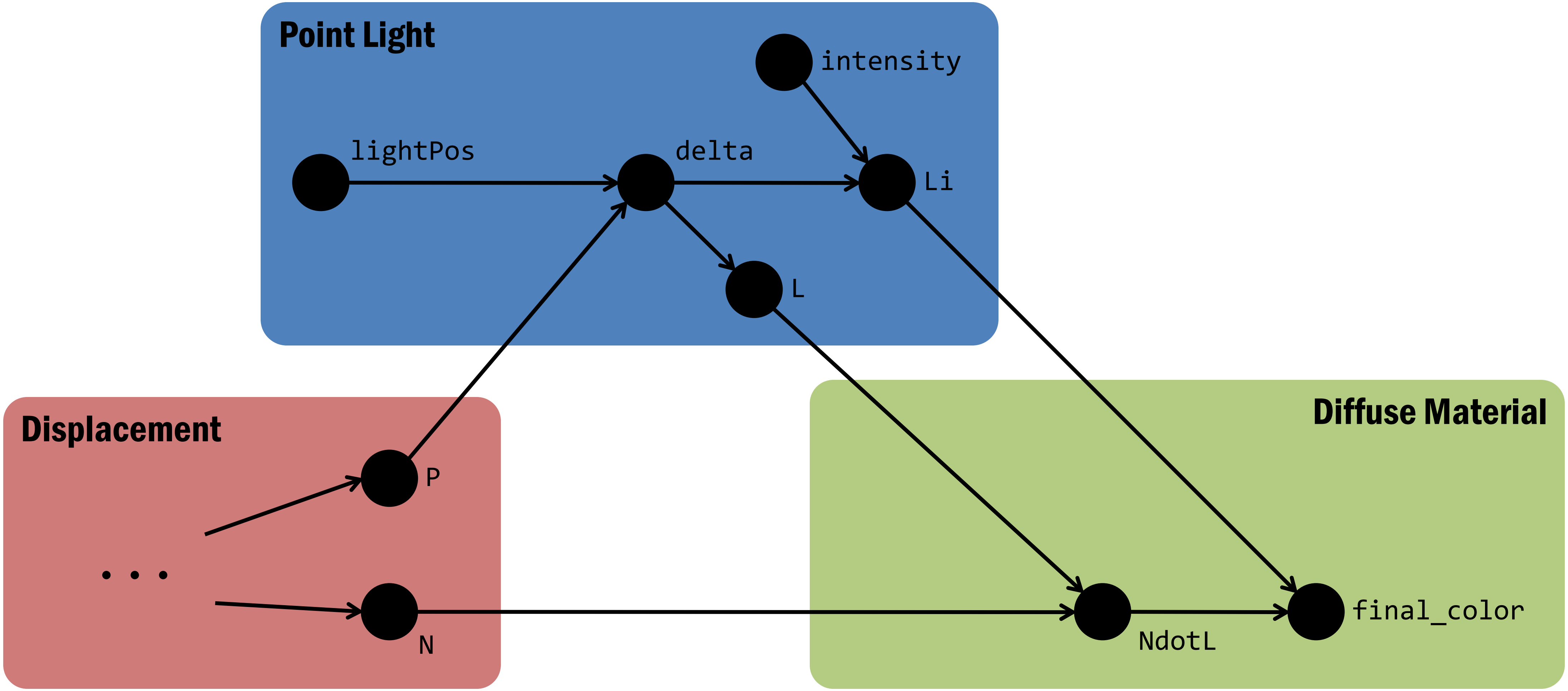
Shader Graphs are Composable



Shader Graphs are Composable



Shader Graphs are Composable



Exploiting Specialized Hardware

Specializing Code to Data

RenderMan Shading Language

[Hanrahan and Lawson 1990]

RenderMan Shading Language

```
uniform vector L;
varying vector N;

...

L = normalize(L);

...

N = normalize(N);
varying float NdotL = N . L;
```

Key:
type
rate

RenderMan Shading Language

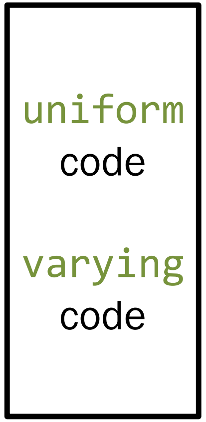
Key:
type
rate

```
uniform vector L;
varying vector N;

...

L = normalize(L);

...

N = normalize(N);
varying float NdotL = N . L;
```

computed per-batch

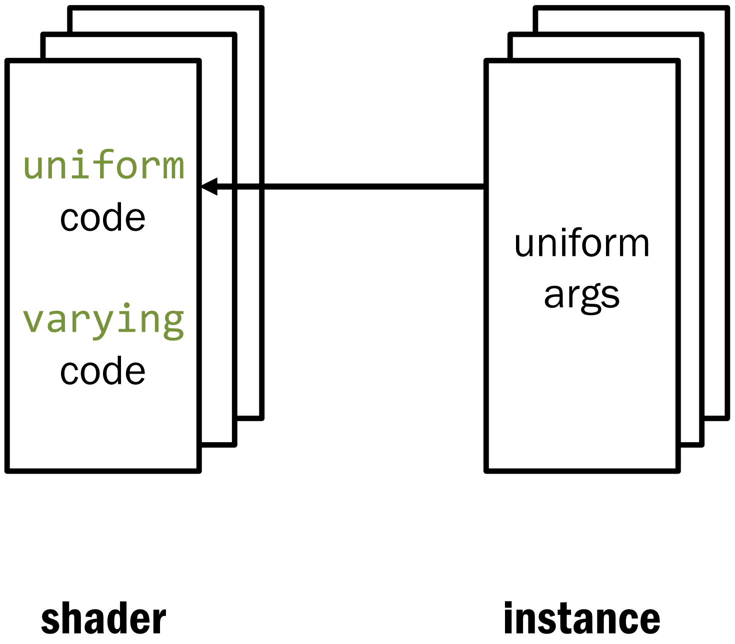
computed per-sample

Split shader into uniform and varying parts

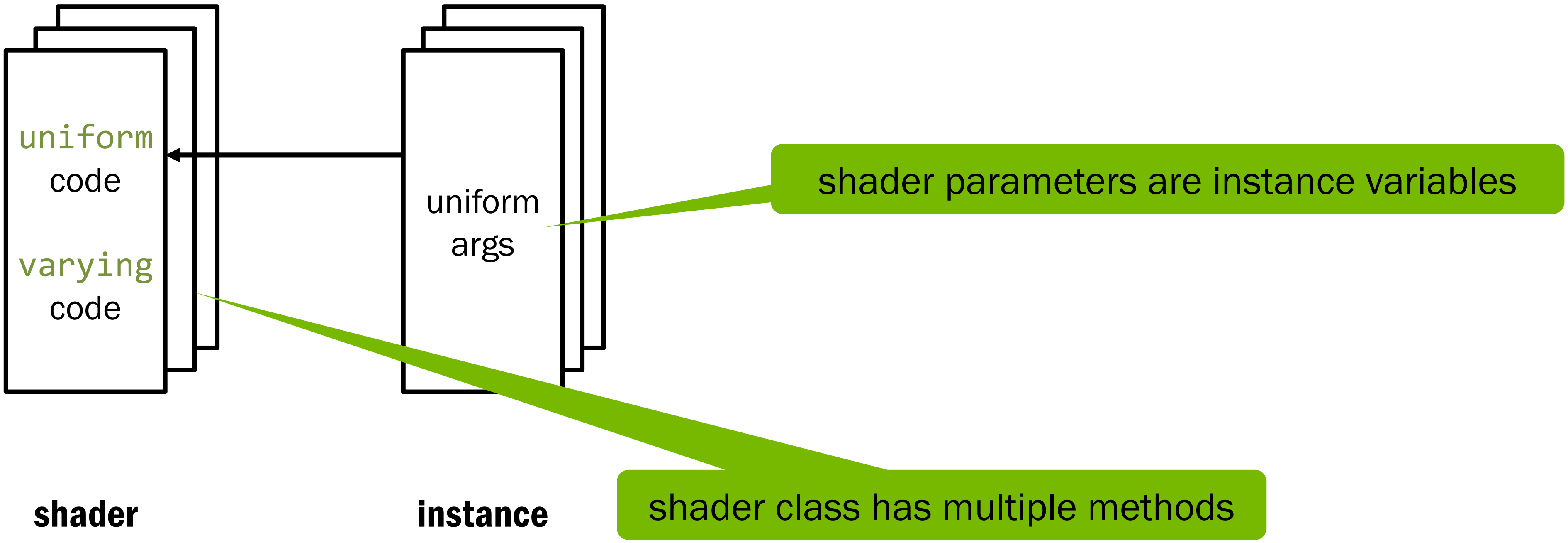


shader

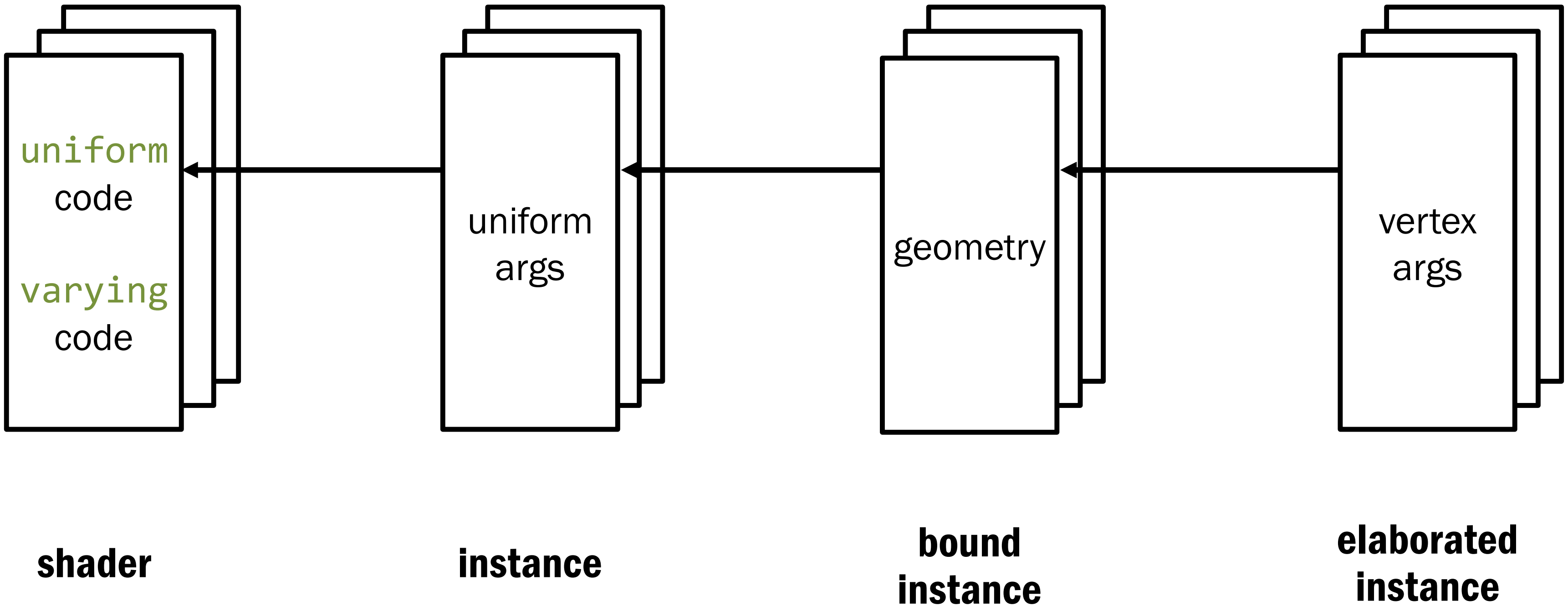
Create an instance of the shader class



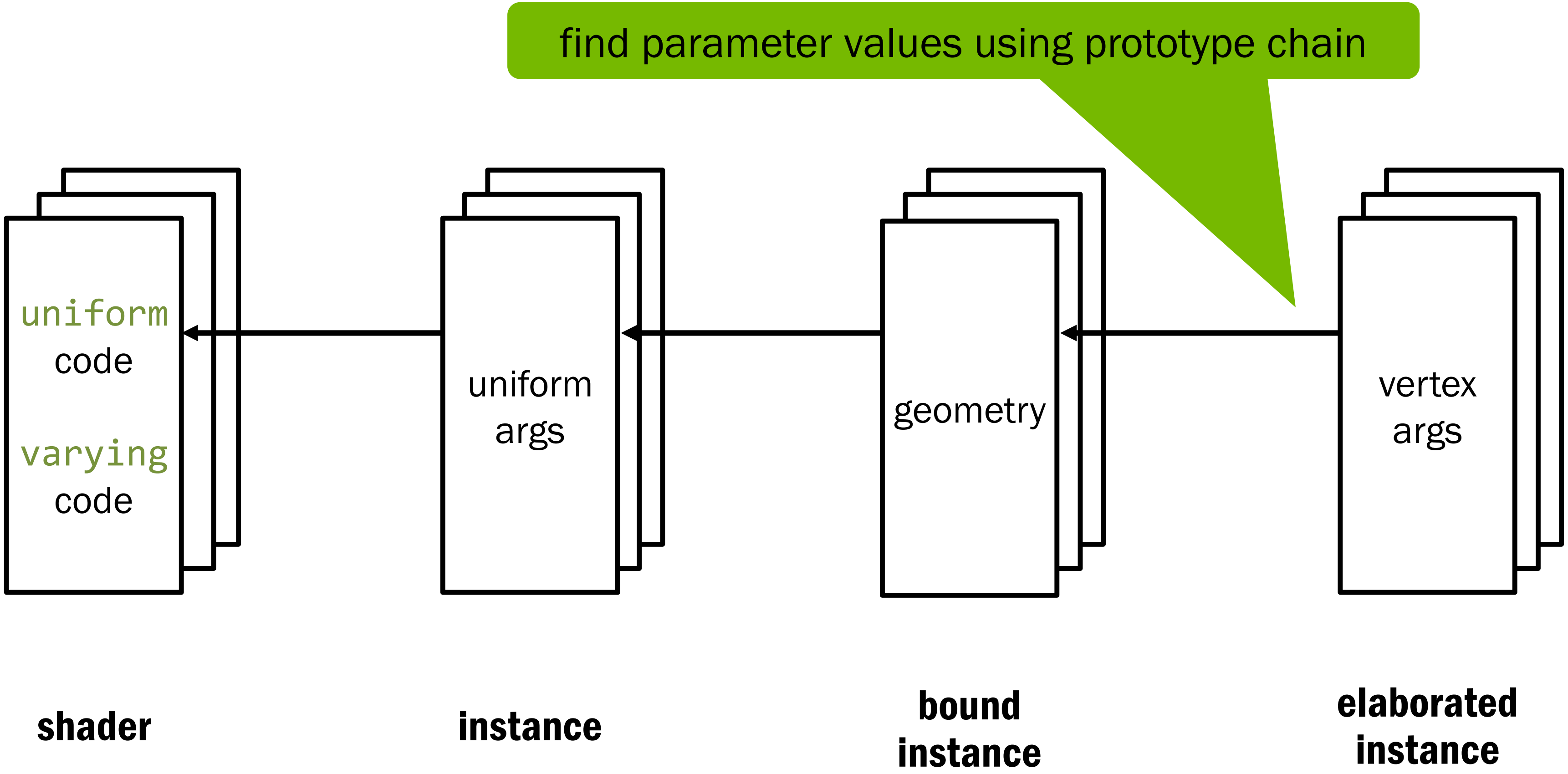
Create an instance of the shader class



Specialize as more information becomes known



Specialize as more information becomes known



Intermission:

Let’s Talk About Staging

Staging Transformations

[Jørring and Scherlis 1986]

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

Where x might represent information known “before” y

Compute functions f1(x) and f2(t,y)

Such that f2(f1(x),y) = f(x,y)

Can generalize to N stages

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

want to
compute
f(x, a)
f(x, b)
f(x, c)
…

Goal

Try to do “as much as possible” in f1

A Trivial Solution

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

end

function f1(x)
    return x


end

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

A Trivial Solution

```
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
```

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
```

first step returns a **closure**

second step applies it

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
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

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

```
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

```
delay(exp) <-> function() return exp end
force(exp) <-> 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;
}
```

Key:
keyword
type
constant
rate

```
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;
}
```

Key:
keyword
type
constant
rate

```
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;
}
```

Key:
keyword
type
constant
rate

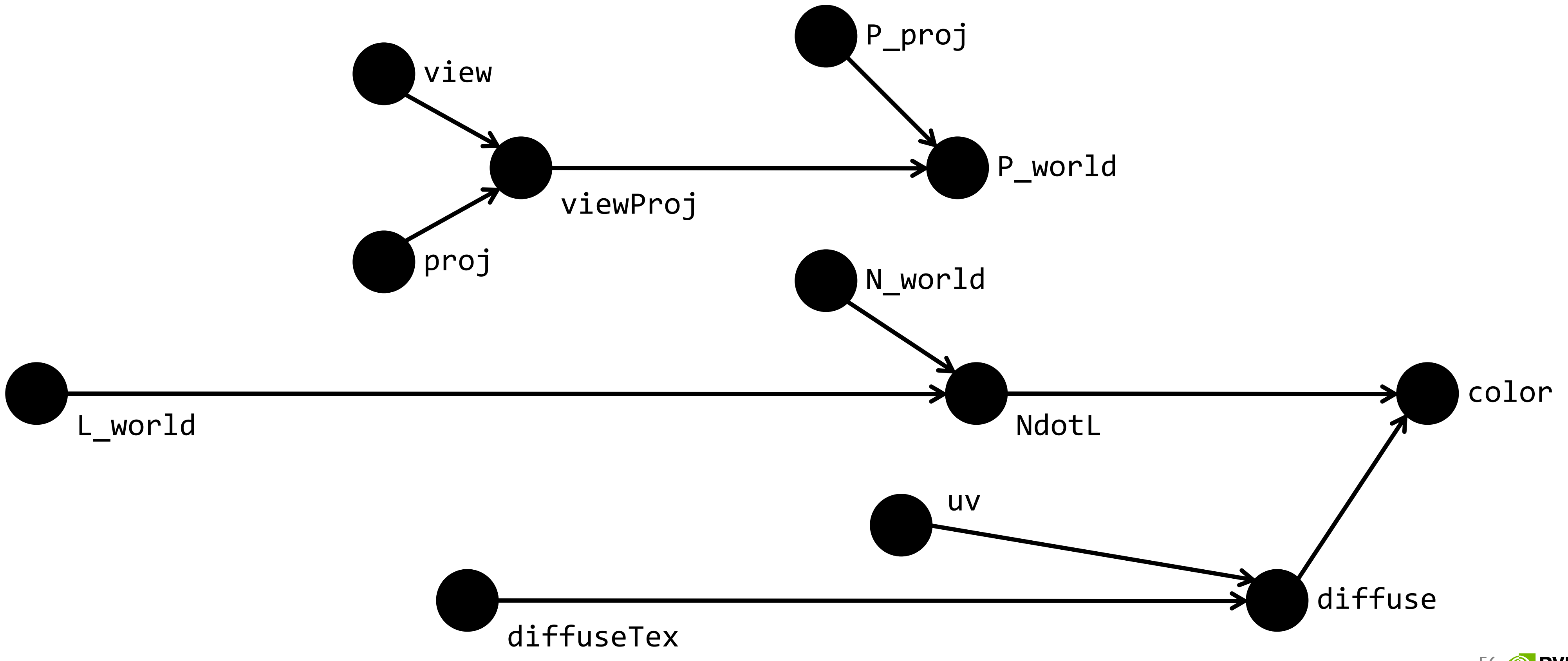
```
surface shader float4 Simple( ... )
{
    constant       float3   L_world  = normalize({1, 1, 1});

    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;
}
```

Key:
keyword
type
constant
rate

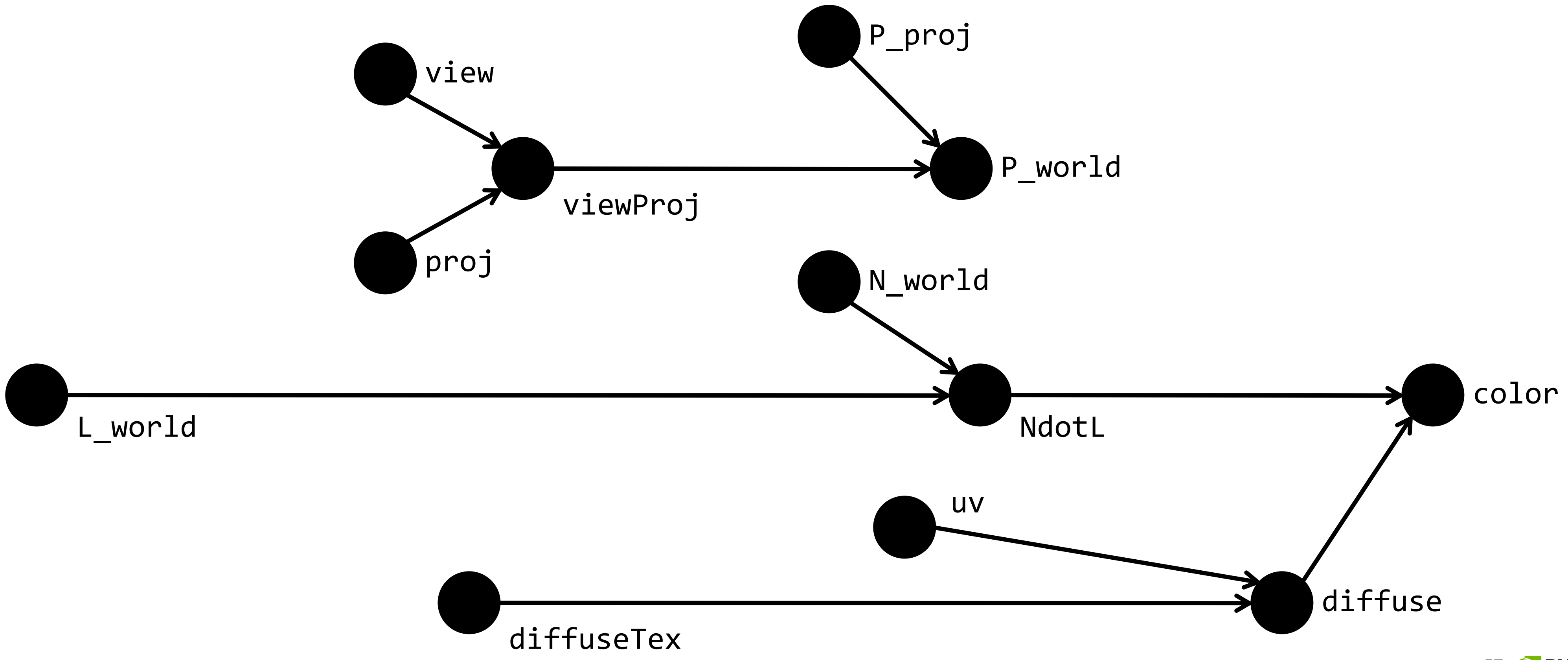
```
surface shader float4 Simple( ... )
{
    constant       float3   L_world  = normalize({1, 1, 1});

    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;
}
```

Key:
keyword
type
constant
rate

Map to Shader Graph

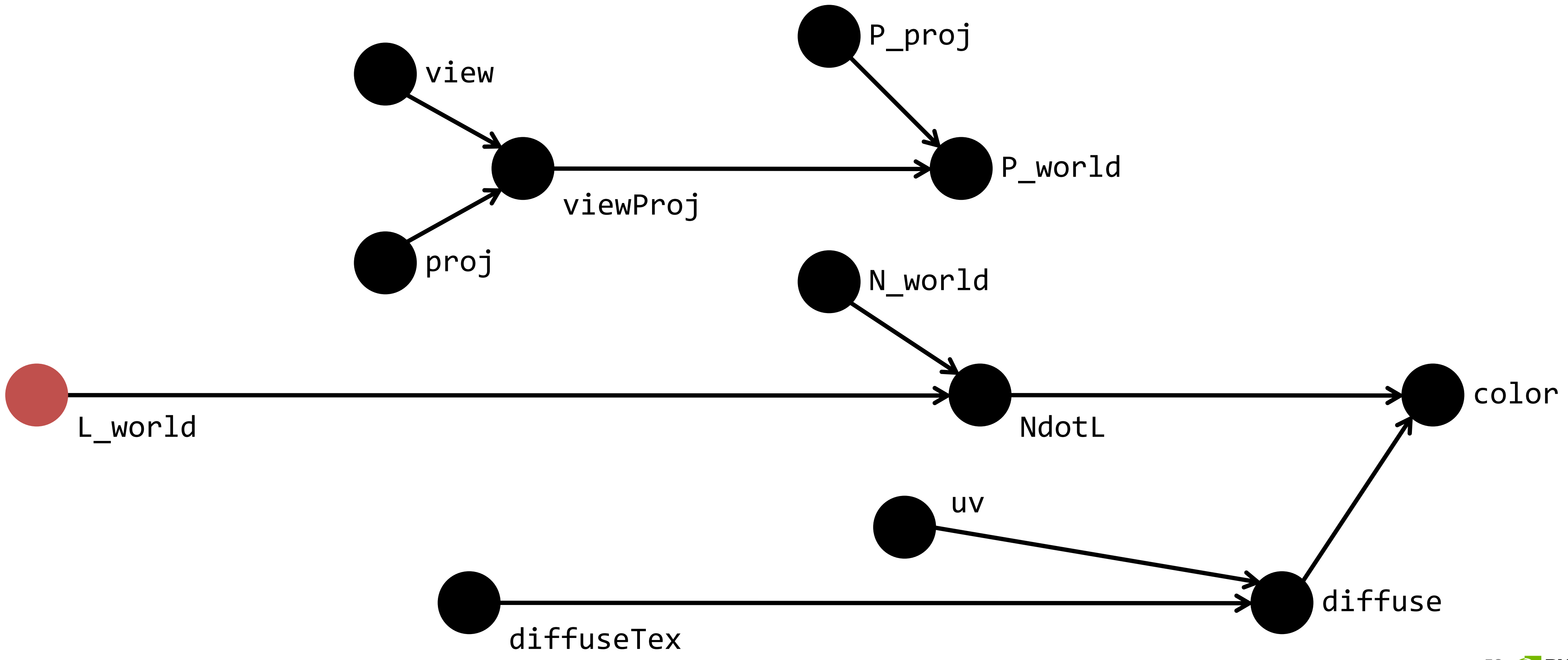


Color by Rates

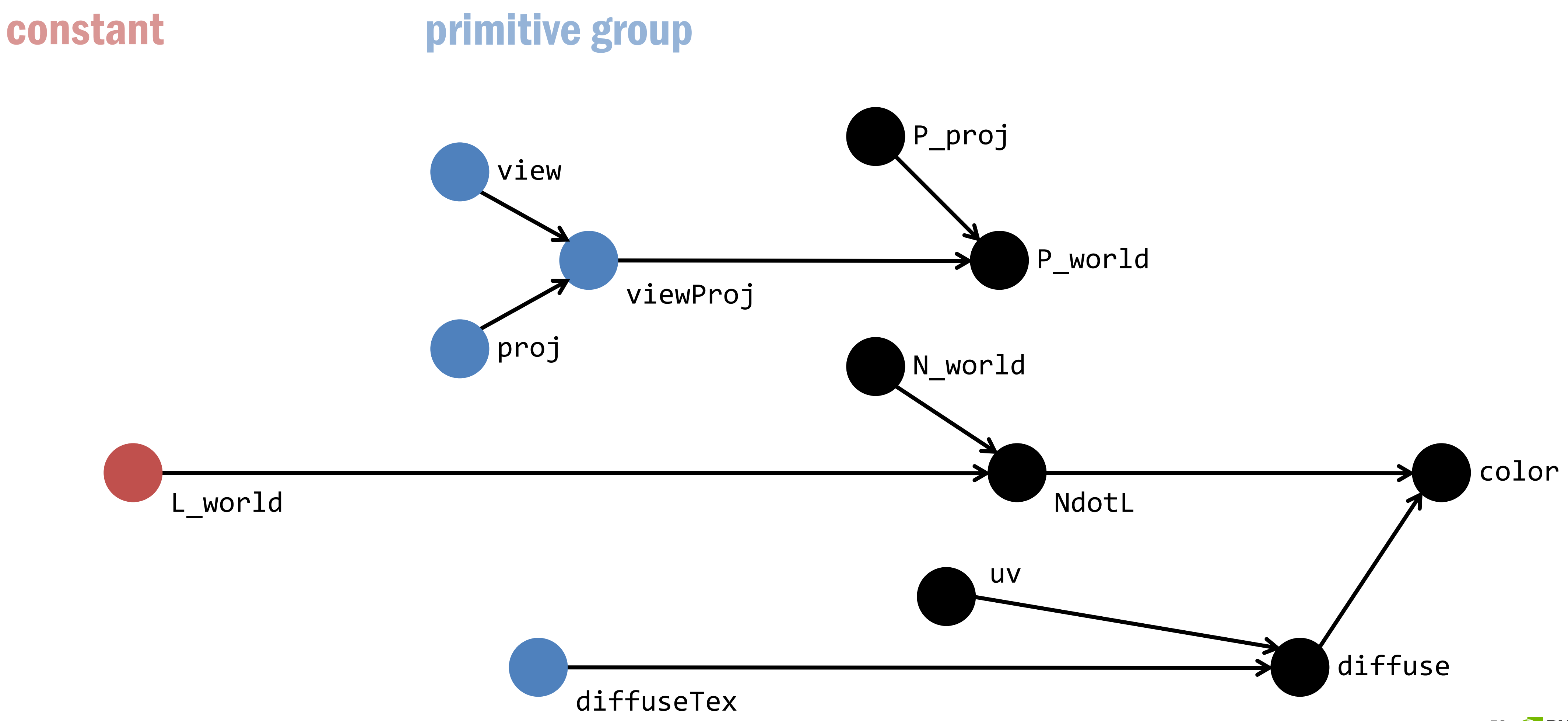


Color by Rates

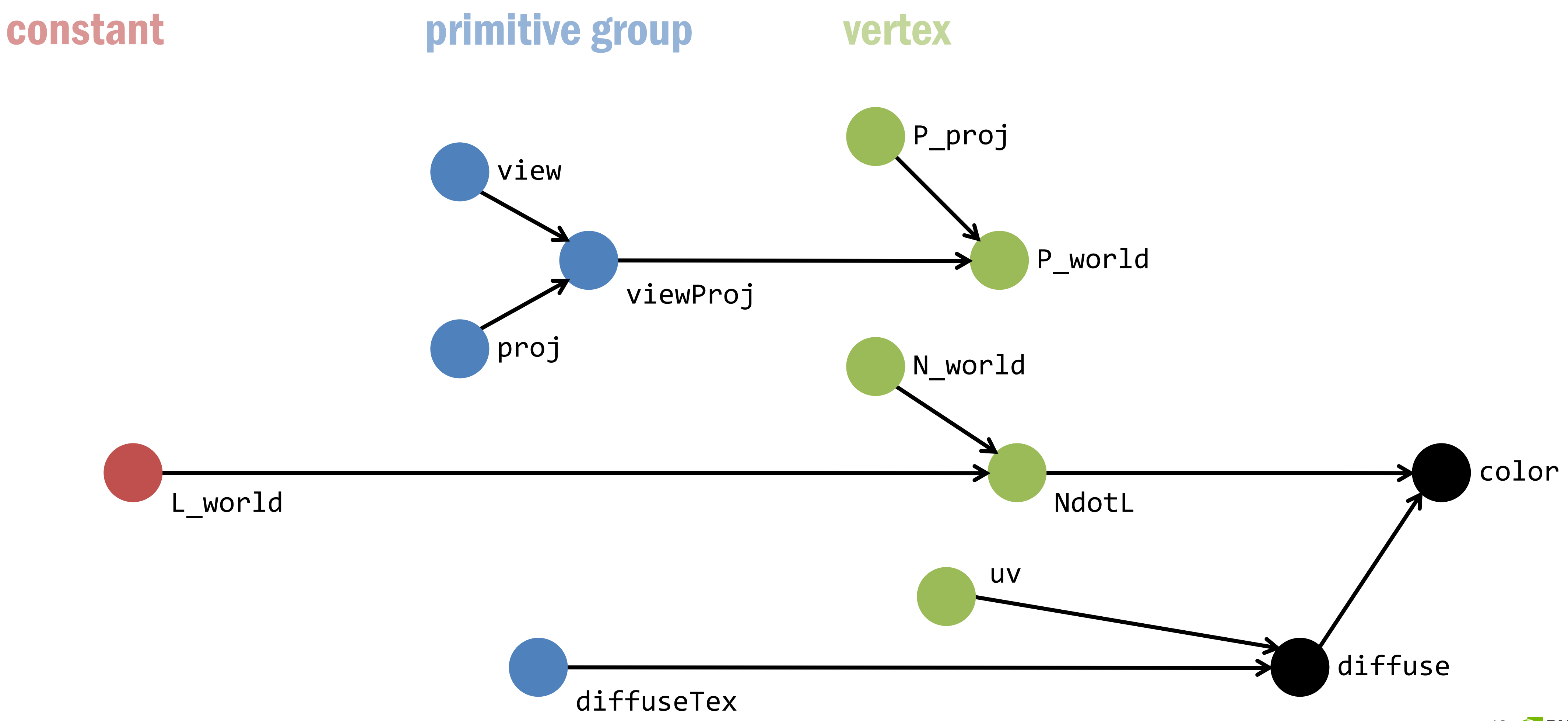
constant



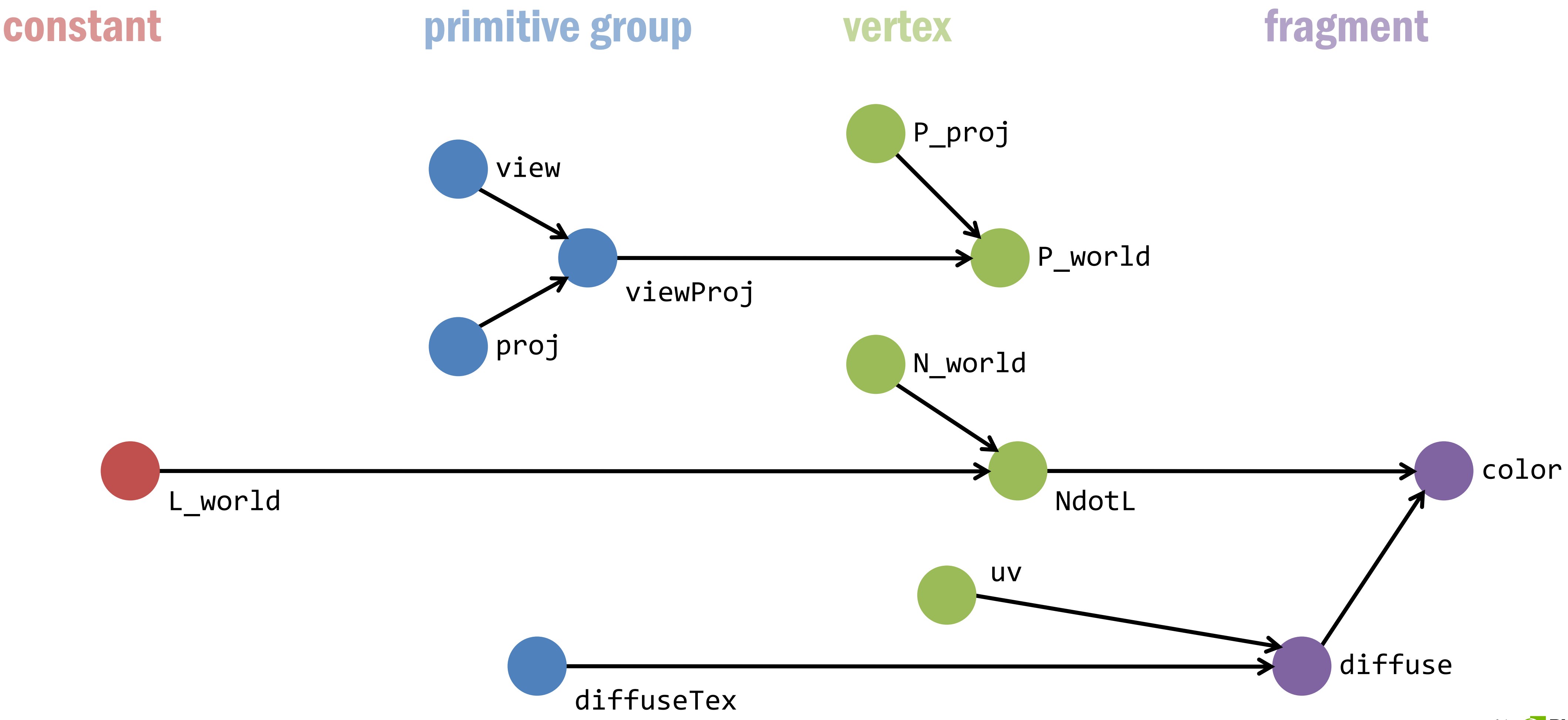
Color by Rates



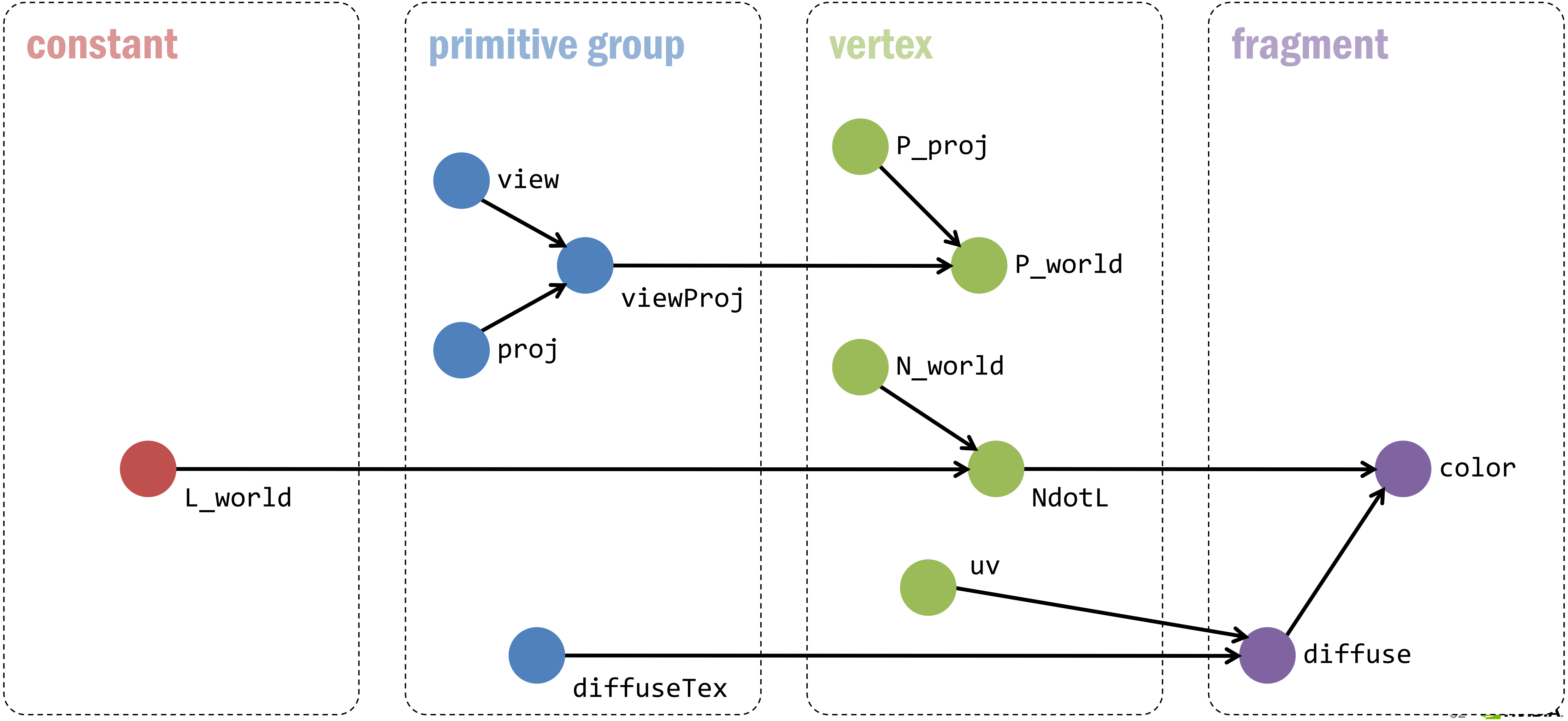
Color by Rates



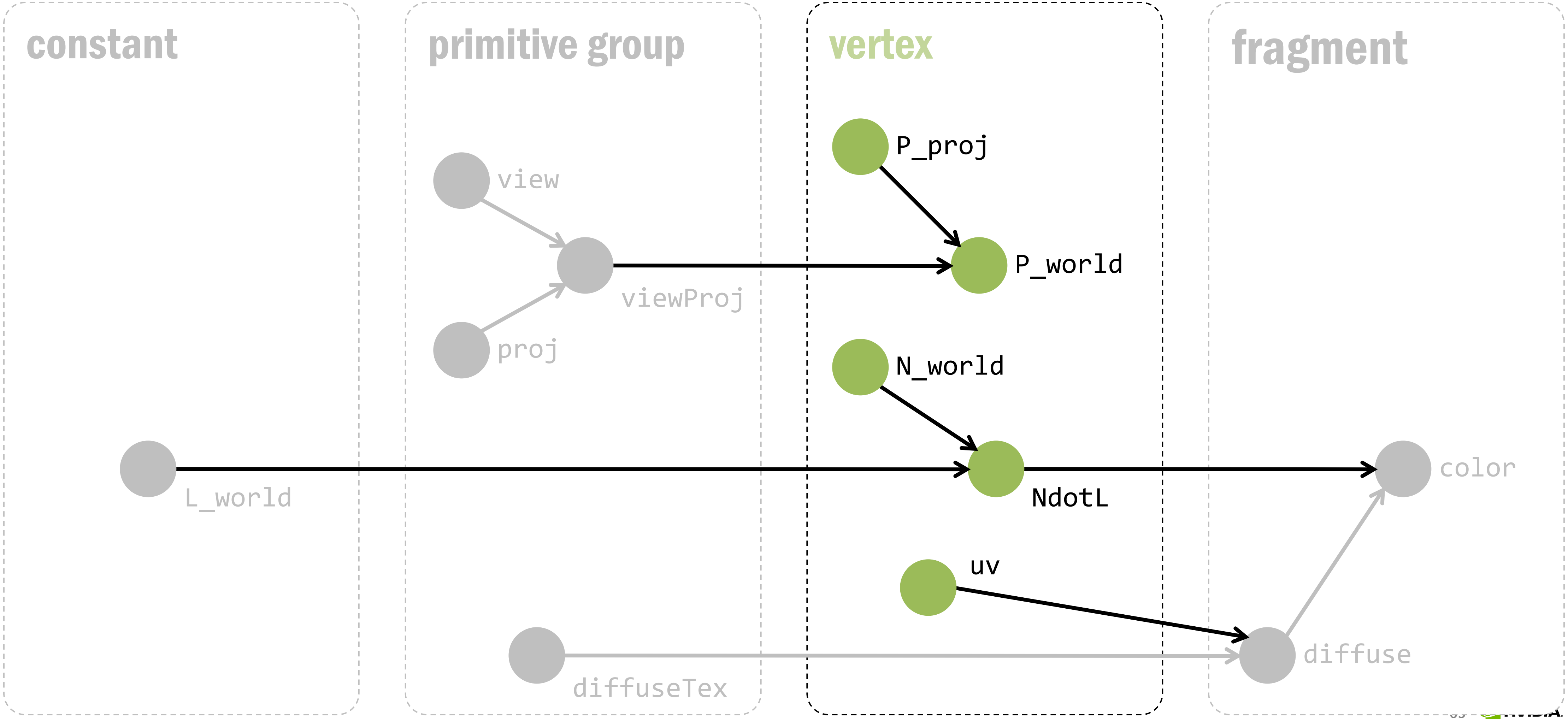
Color by Rates



Partition


constant
primitive group
vertex
fragment
view
viewProj
proj
P_proj
P_world
N_world
NdotL
uv
L_world
color
diffuse
diffuseTex

Partition


constant
primitive group
vertex
fragment
P_proj
view
P_world
viewProj
proj
N_world
L_world
NdotL
color
uv
diffuse
diffuseTex

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

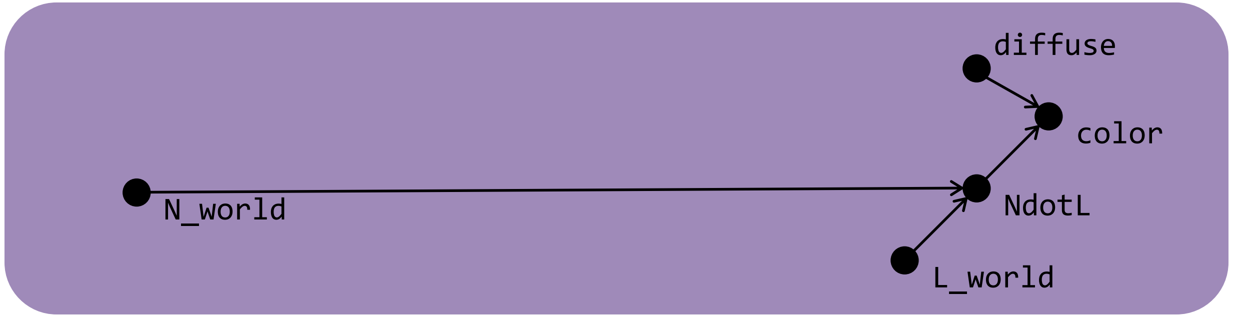
Define Shader Graphs as Classes

```
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;
}
```

Key:
keyword
type
constant
rate

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;
}
```

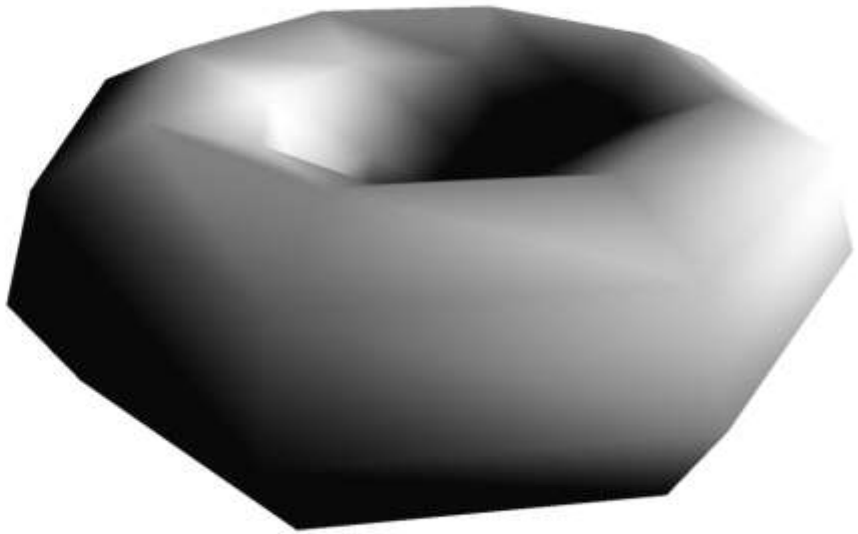


Define Shader Graphs as Classes

```
shader class SimpleDiffuse
{
    ...
}

shader class CubicGregoryACC     { ... }
shader class MyTextureMapping    { ... }
shader class ScalarDisplacement { ... }
...
```

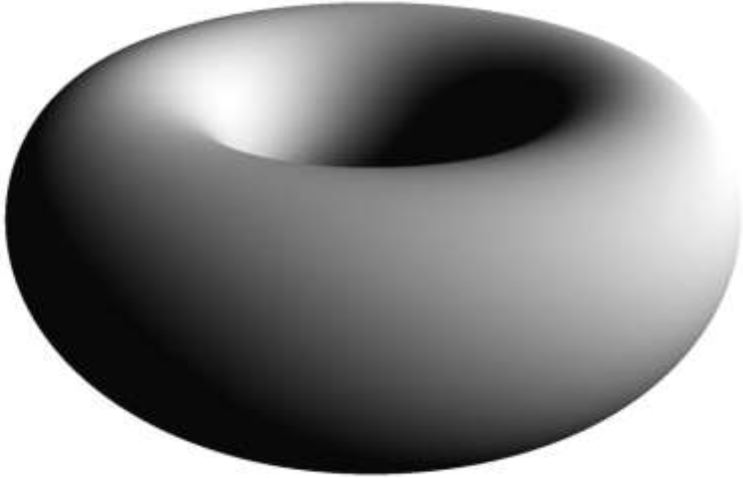
Compose With Inheritance

```
shader class Composed
    extends SimpleDiffuse



{}
```



Compose With Inheritance

```
shader class Composed
    extends SimpleDiffuse,
            CubicGregoryACC


{}
```



Compose With Inheritance

```
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 190]

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]

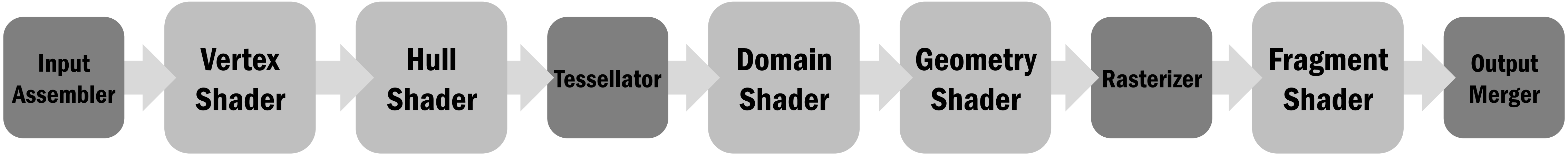
Mobile Types for Mobile Code

[Tom Murphy VII 2008]

Backup

Cg, HLSL, GLSL

The Direct3D 11 Pipeline



Programmable Stages

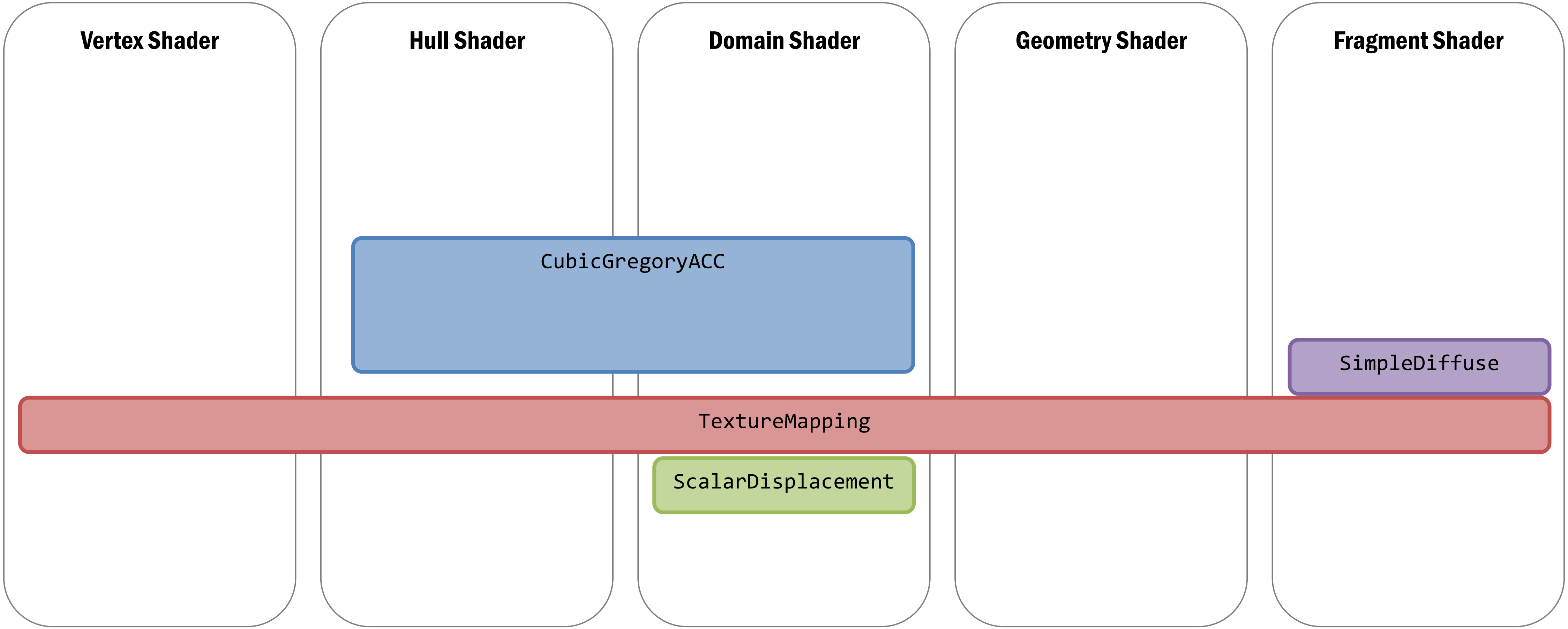
Programmable Stages

Vertex Shader	Hull Shader	Domain Shader	Geometry Shader	Fragment Shader

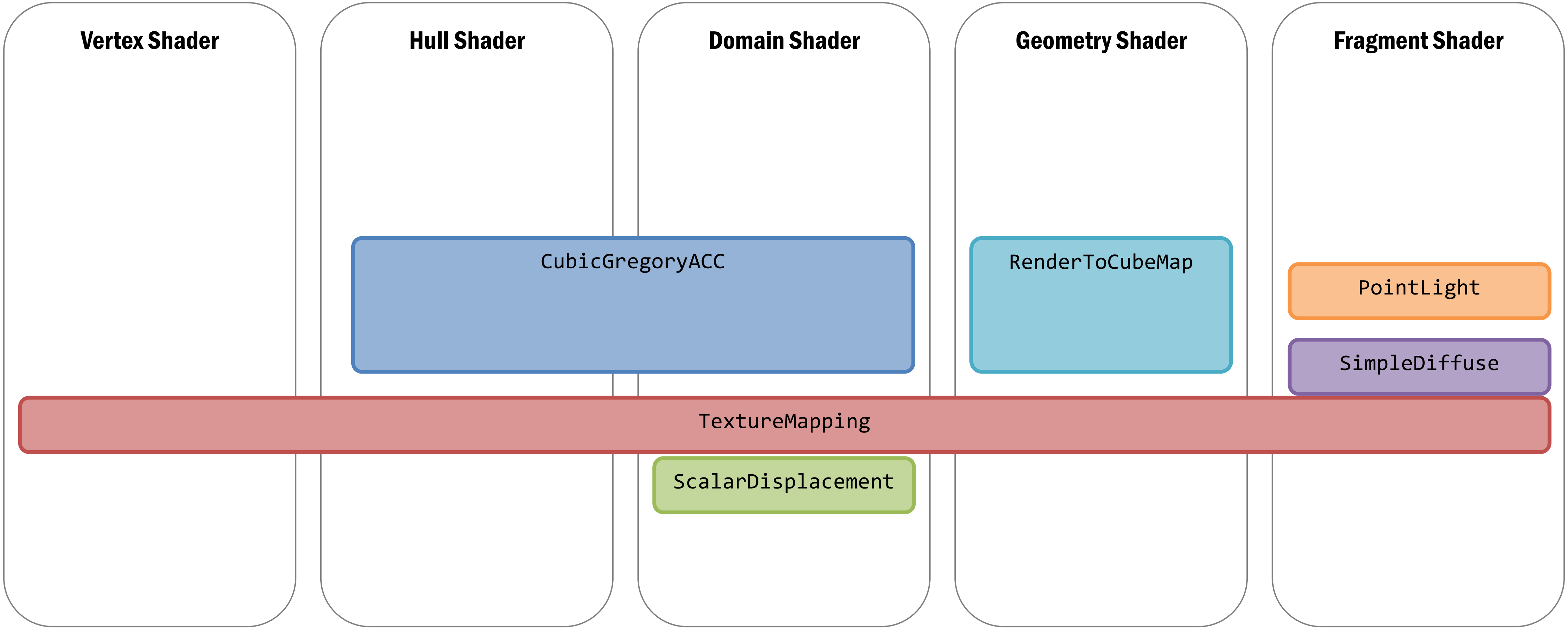
Problem-Domain Components

Vertex Shader	Hull Shader	Domain Shader	Geometry Shader	Fragment Shader

Problem-Domain Components



Problem-Domain Components



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
}
```

Key:
keyword
type
constant
rate

```
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
```

Key:
keyword
type
constant
rate

```
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
```

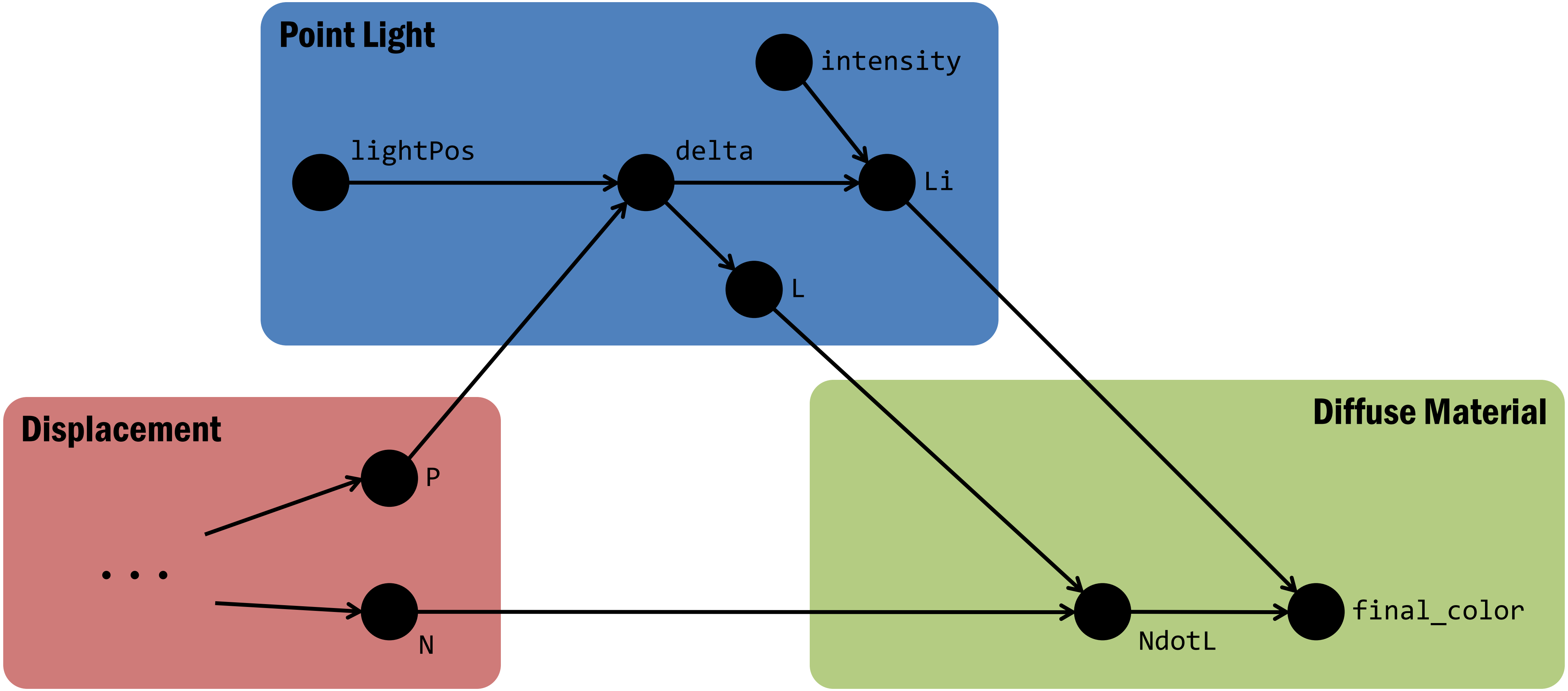
Key:
keyword
type
constant
rate

```
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
```

Key:
keyword
type
constant
rate

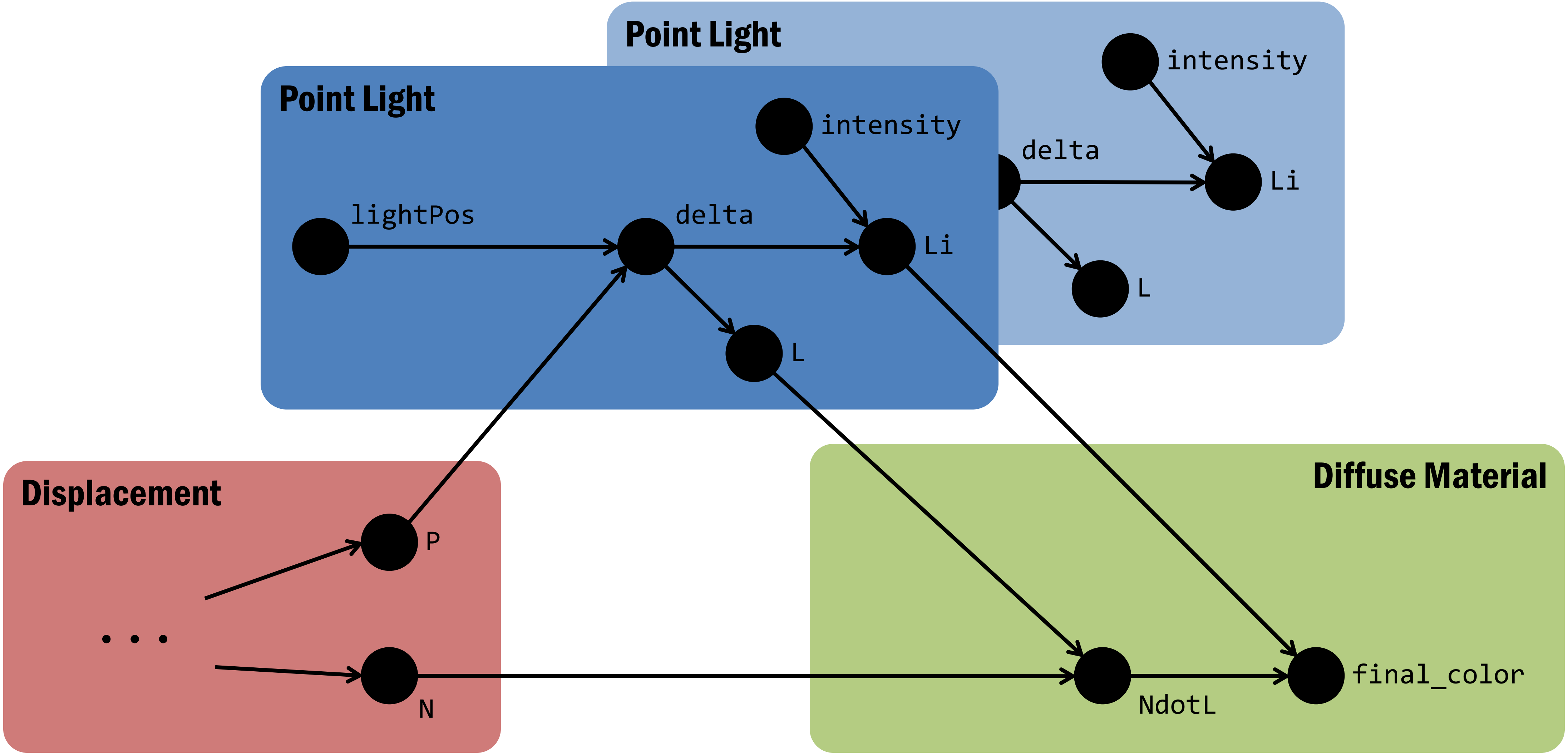
Staging Isn't Always For Performance

Shade Trees

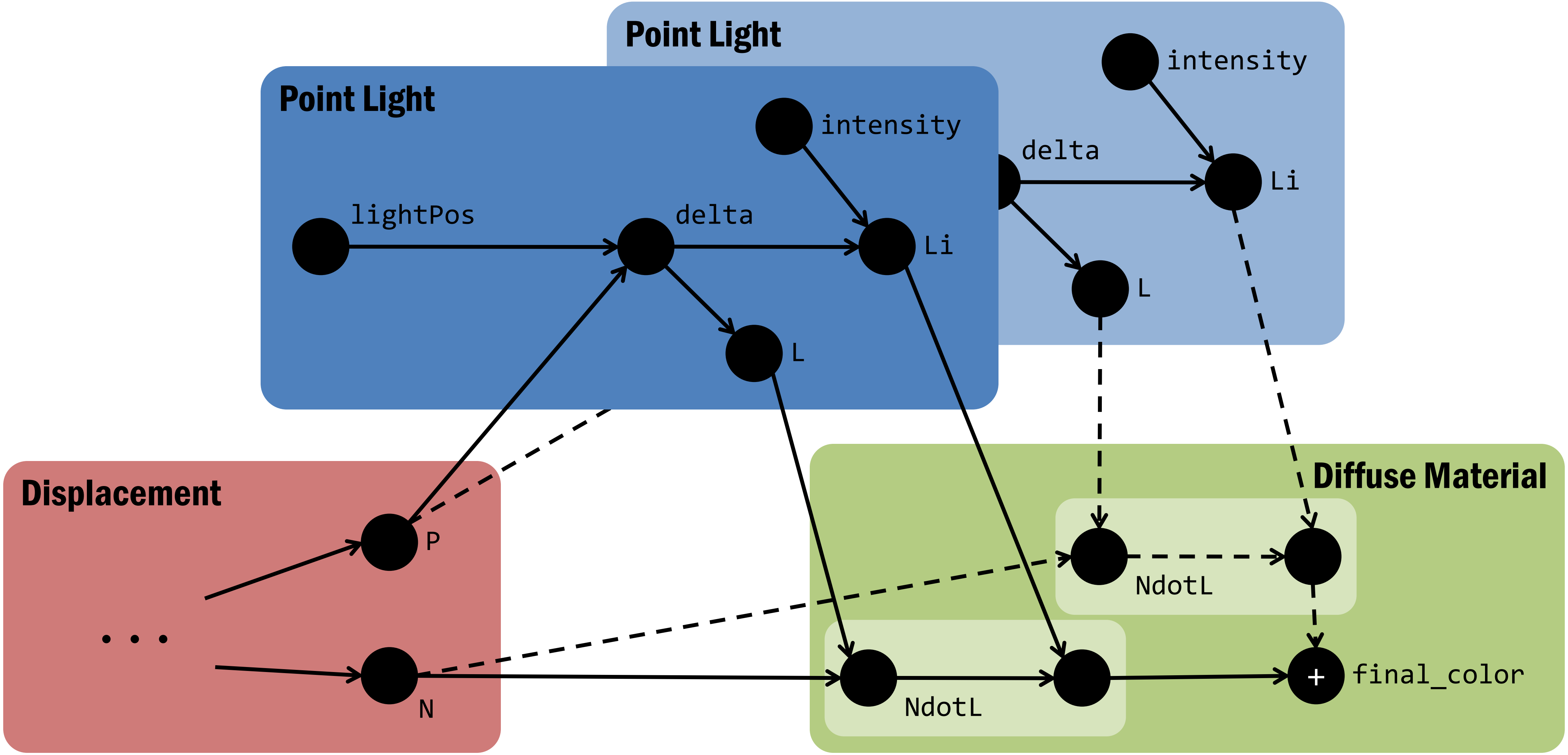


Point Light
intensity
lightPos
delta
Li
L
Displacement
P
...
N
Diffuse Material
NdotL
final_color

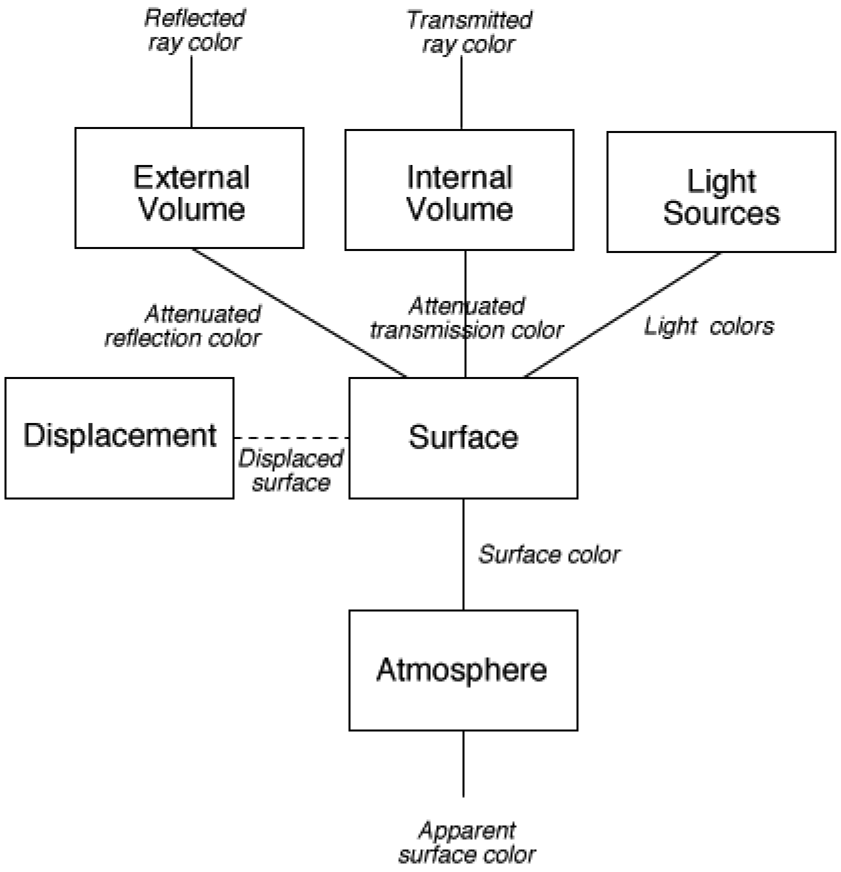
What if we have multiple lights?



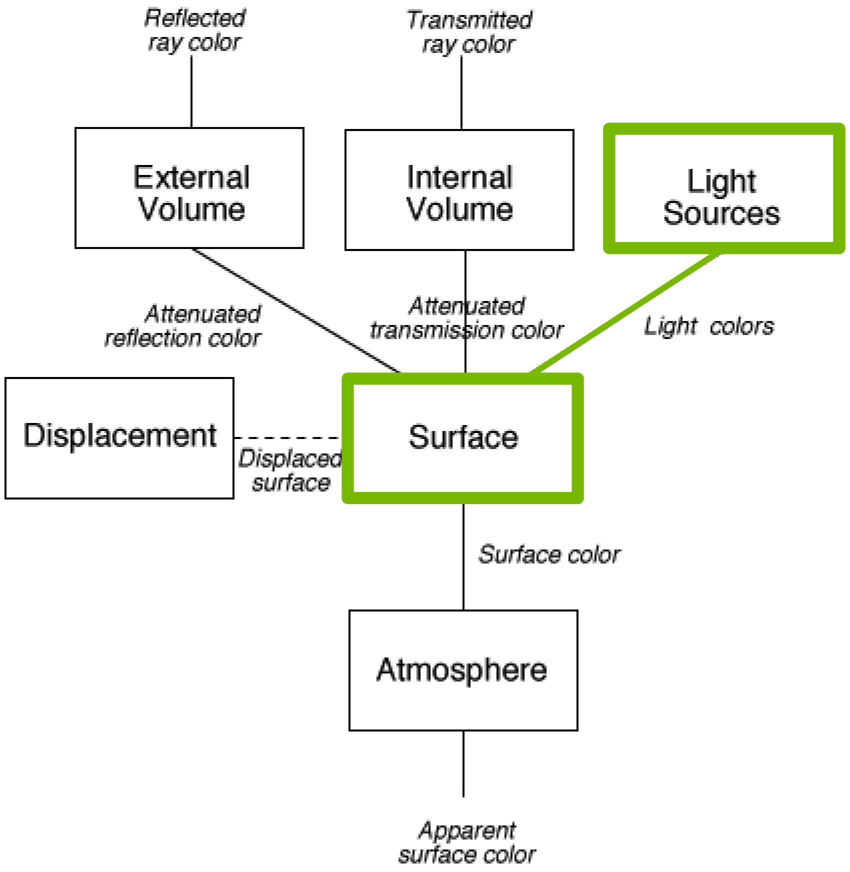
What if we have multiple lights?



RenderMan Shading Language



Surface and Light Shaders



Linkage via control-flow constructs

Key:
keyword
type
constant

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

Key:
keyword
type
constant

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

Key:
keyword
type
constant

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

Key:
keyword
type
constant

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

Key:
keyword
type
constant

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

Key:
keyword
type
constant

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);
}
```

Could re-cast as higher-order functions

Key:
keyword
type
constant

surface shader

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

light shader(s)

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

Could re-cast as higher-order functions

Key:
keyword
type
constant

surface shader

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

light shader(s)

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

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 `@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.

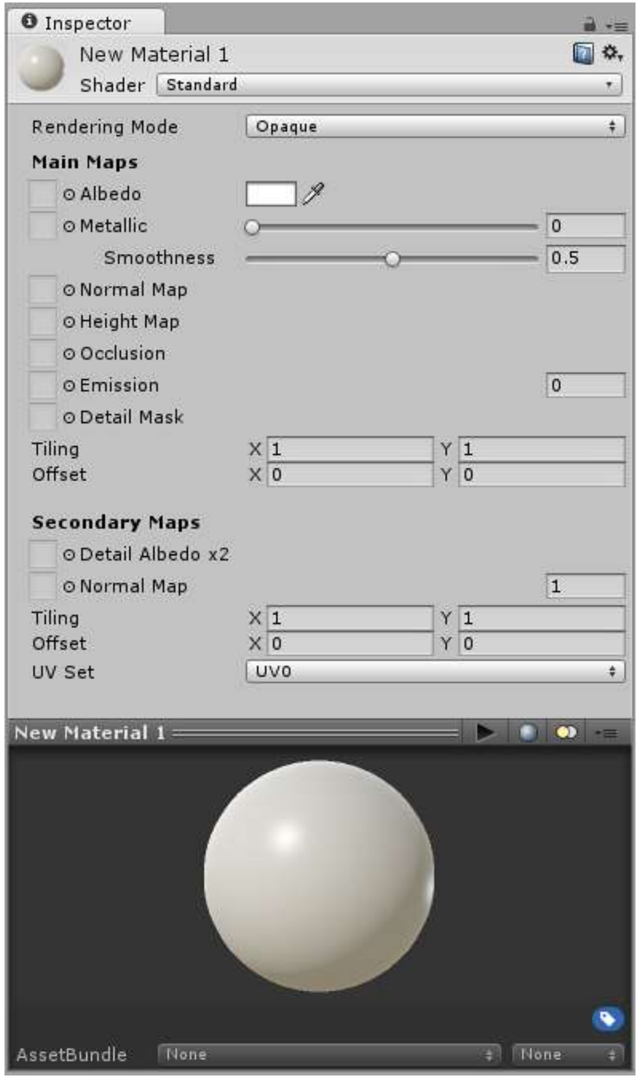
Authored by programmers, or technical artists

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

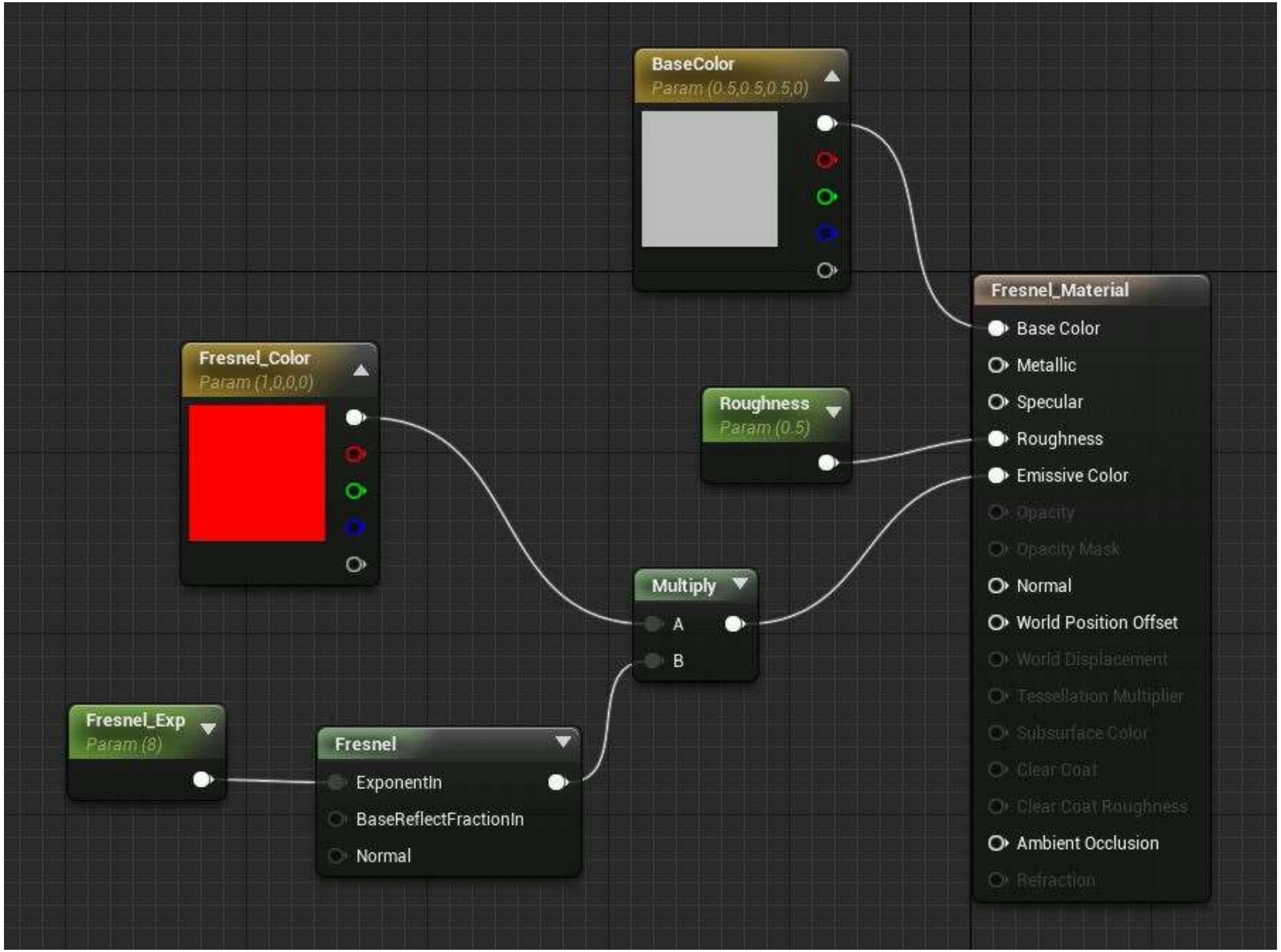
Also used for

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 = FresnelDieletric(I, N, ior);
    bsdf = Kd * (fr*reflection(N) + (1-fr)*refraction(N, ior));
}
```

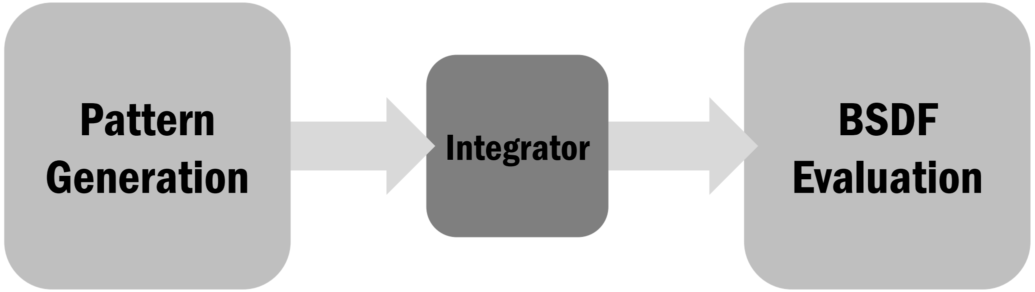
Open Shading Language (OSL)

shader outputs a "radiance closure,"
to be scheduled by renderer

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

closures created with built-in functions,
then combined with operators like +

Two-Stage Material Shading Pipeline



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