Shader Programming Models

State of the Art and Open Problems

Tim Foley

NVIDIA Research

Goal

- Share a mental model for thinking about shaders
  - Supporting terminology
- Not a mainstream viewpoint
  - Won’t get this perspective in tutorials, etc.
  - This is (part of) how I view the problem space

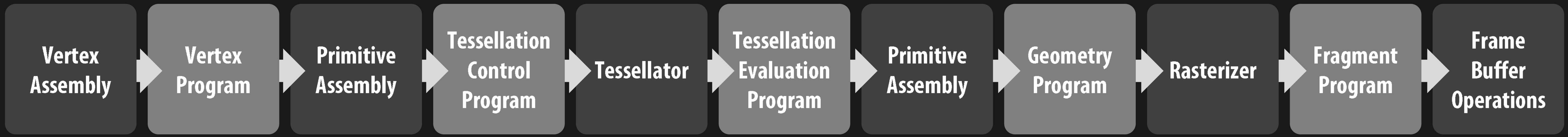
Agenda

- Hardware
  - Configuring a Programmable Graphics Pipeline
- Software
  - Graphics API Evolution
  - Pipeline Shaders: What and Why
- Future Directions

Programmable Graphics Pipeline

OpenGL 4.3 Pipeline

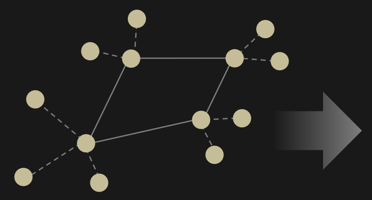
fixed-function
programmable



OpenGL 4.3 Pipeline

fixed-function

programmable

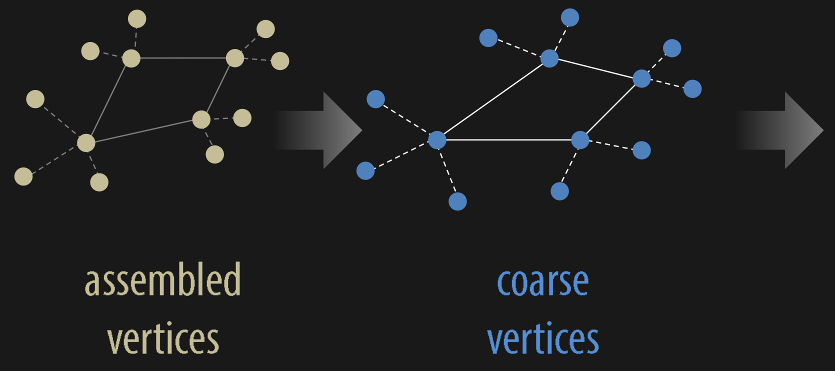


assembled vertices

Vertex Assembly → Vertex Program → Primitive Assembly → Tessellation Control Program → Tessellator → Tessellation Evaluation Program → Primitive Assembly → Geometry Program → Rasterizer → Fragment Program → Frame Buffer Operations

OpenGL 4.3 Pipeline

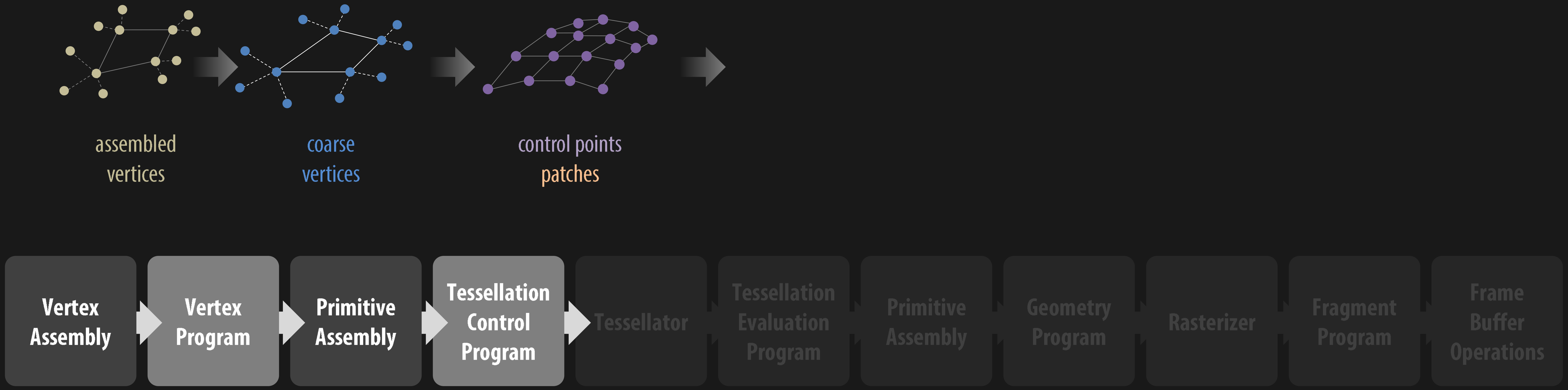
fixed-function
programmable



Vertex Assembly → Vertex Program → Primitive Assembly → Tessellation Control Program → Tessellator → Tessellation Evaluation Program → Primitive Assembly → Geometry Program → Rasterizer → Fragment Buffer Program → Frame Buffer Operations

OpenGL 4.3 Pipeline

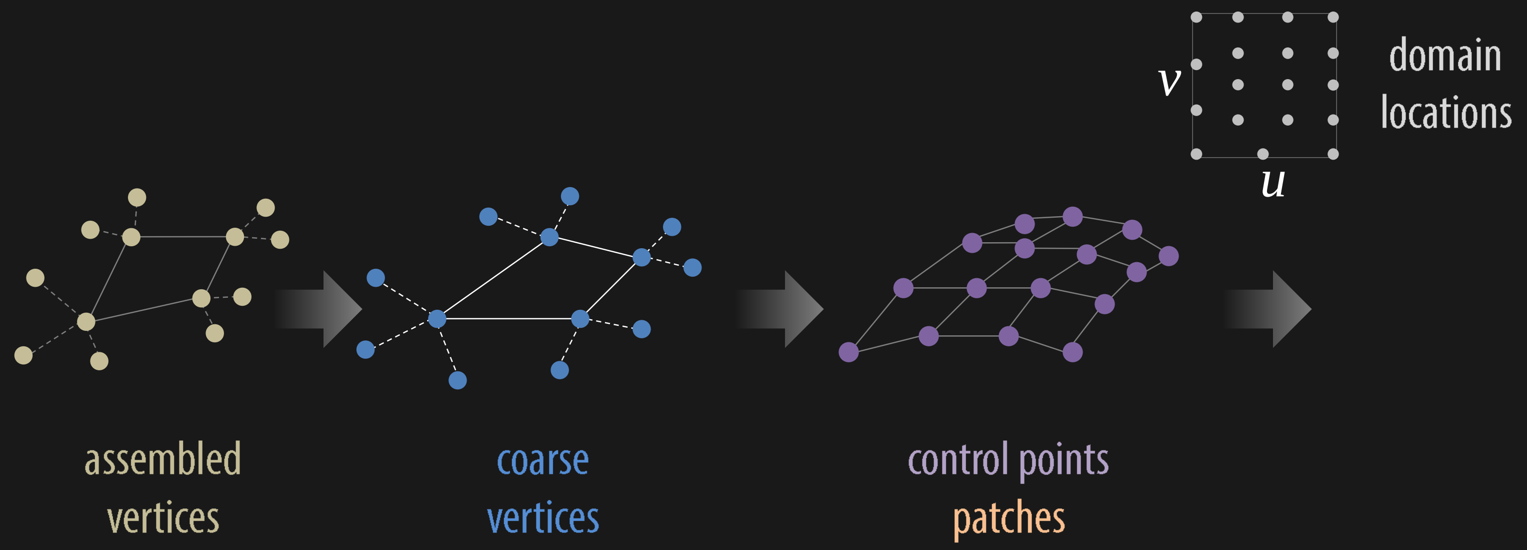
fixed-function
programmable



OpenGL 4.3 Pipeline

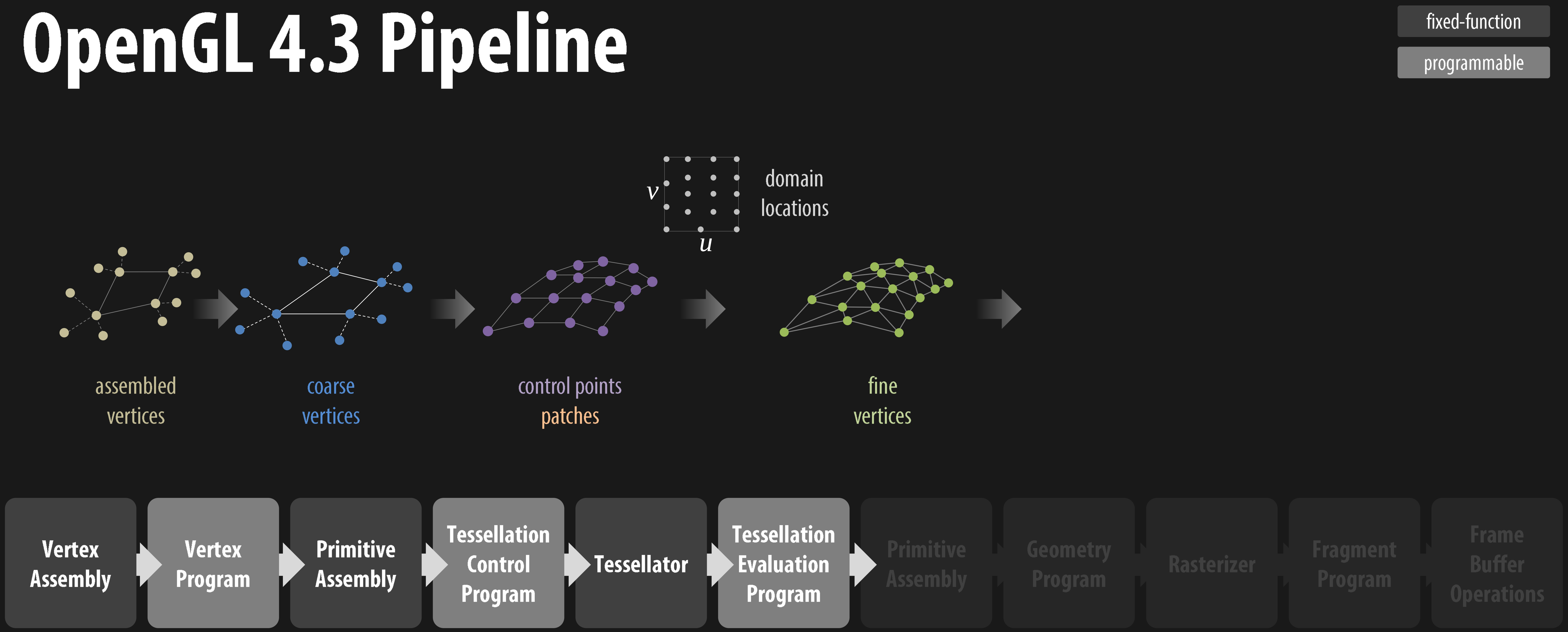
fixed-function

programmable



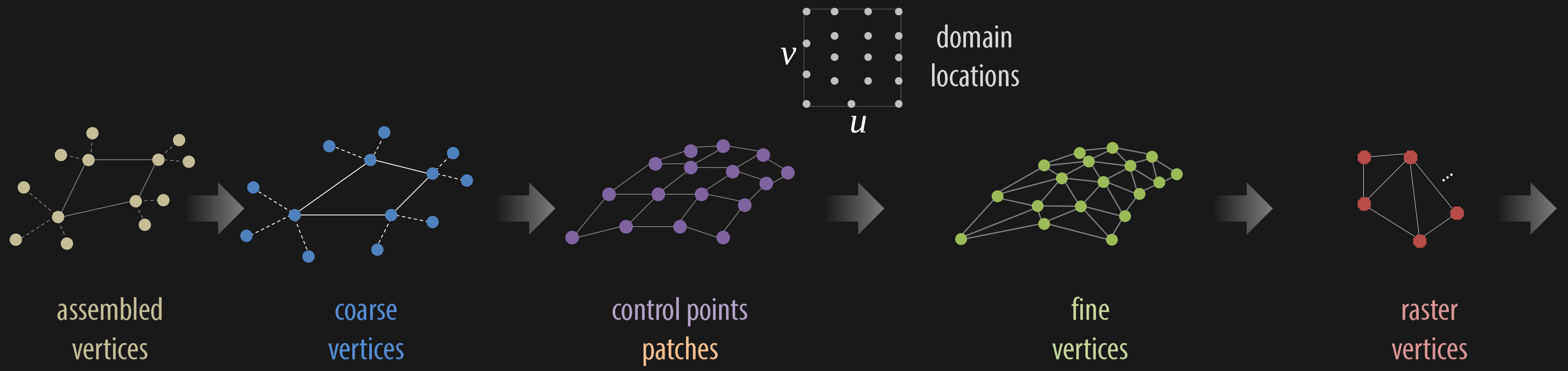
Vertex Assembly → Vertex Program → Primitive Assembly → Tessellation Control Program → Tessellator → Tessellation Evaluation Program → Primitive Assembly → Geometry Program → Rasterizer → Fragment Buffer Program → Frame Buffer Operations

OpenGL 4.3 Pipeline



OpenGL 4.3 Pipeline

fixed-function
programmable

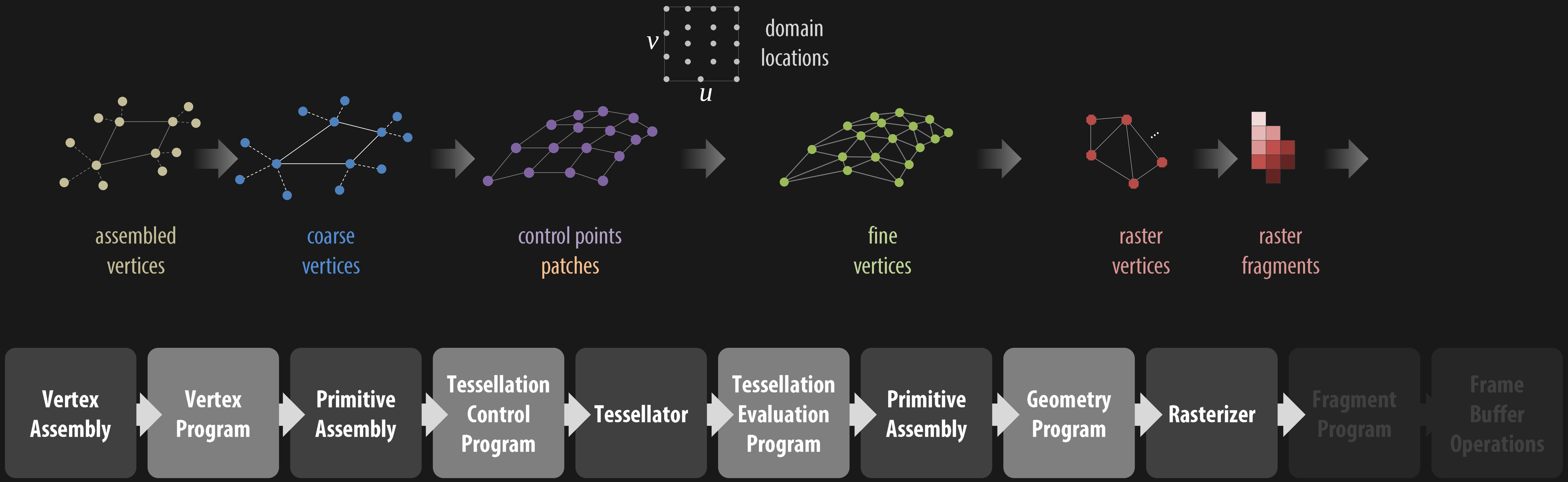


Vertex Assembly → Vertex Program → Primitive Assembly → Tessellation Control Program → Tessellator → Tessellation Evaluation Program → Primitive Assembly → Geometry Program → Rasterizer → Fragment Program → Frame Buffer Operations

OpenGL 4.3 Pipeline

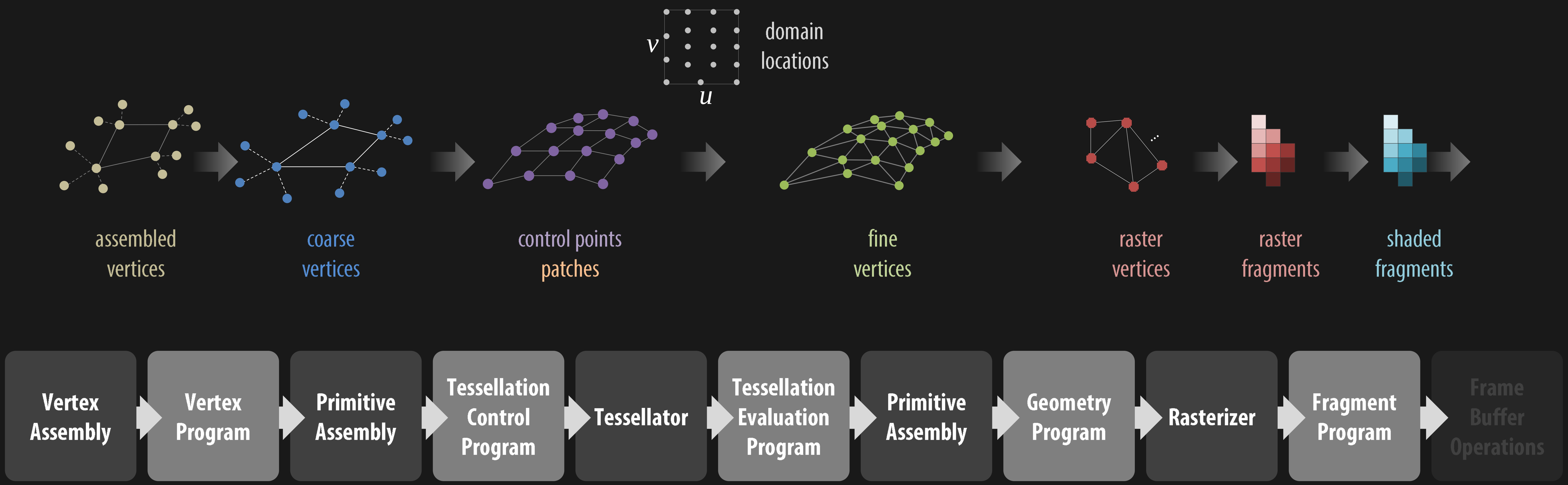
fixed-function

programmable



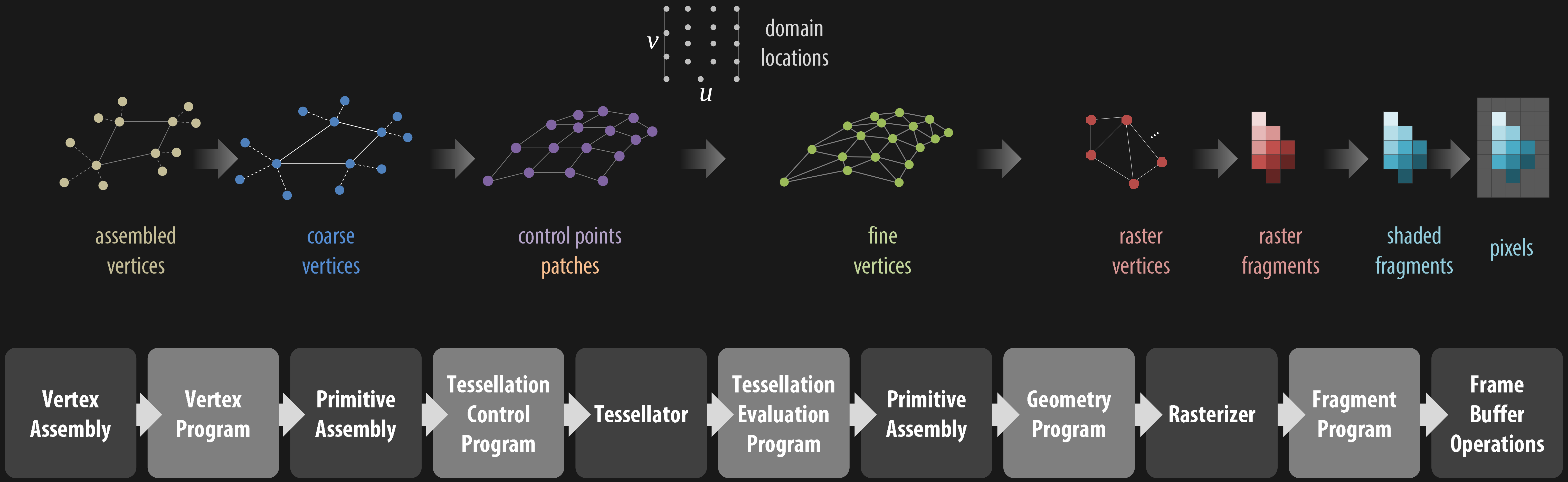
OpenGL 4.3 Pipeline

fixed-function
programmable

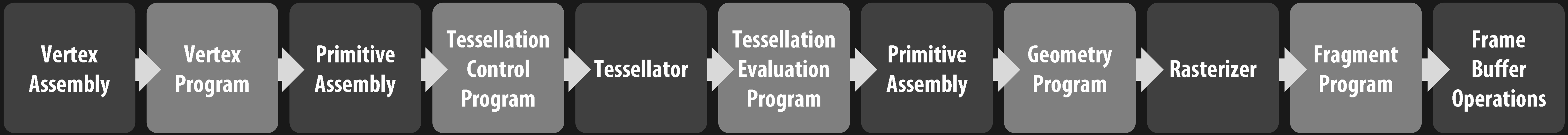


OpenGL 4.3 Pipeline

fixed-function
programmable

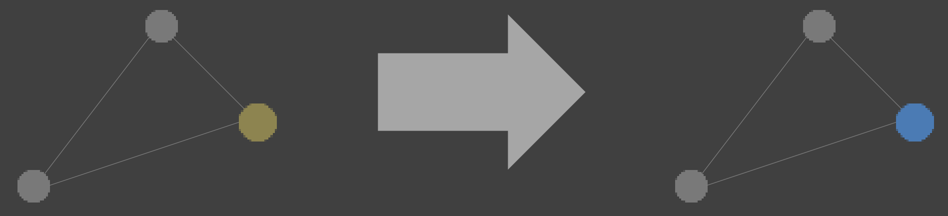


Pointwise
Groupwise

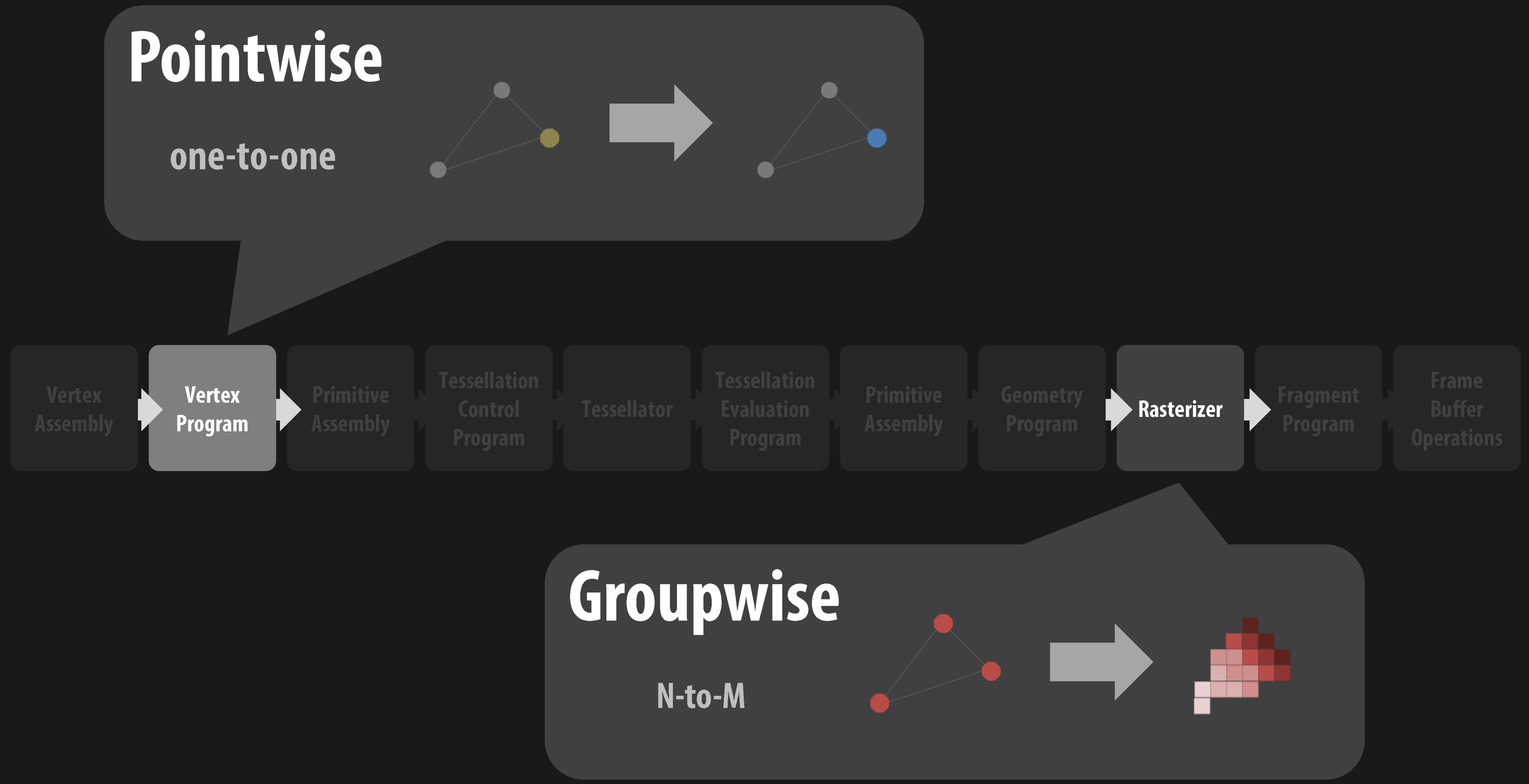

Vertex Assembly
Vertex Program
Primitive Assembly
Tessellation Control Program
Tessellator
Tessellation Evaluation Program
Primitive Assembly
Geometry Program
Rasterizer
Fragment Program
Frame Buffer Operations

Pointwise

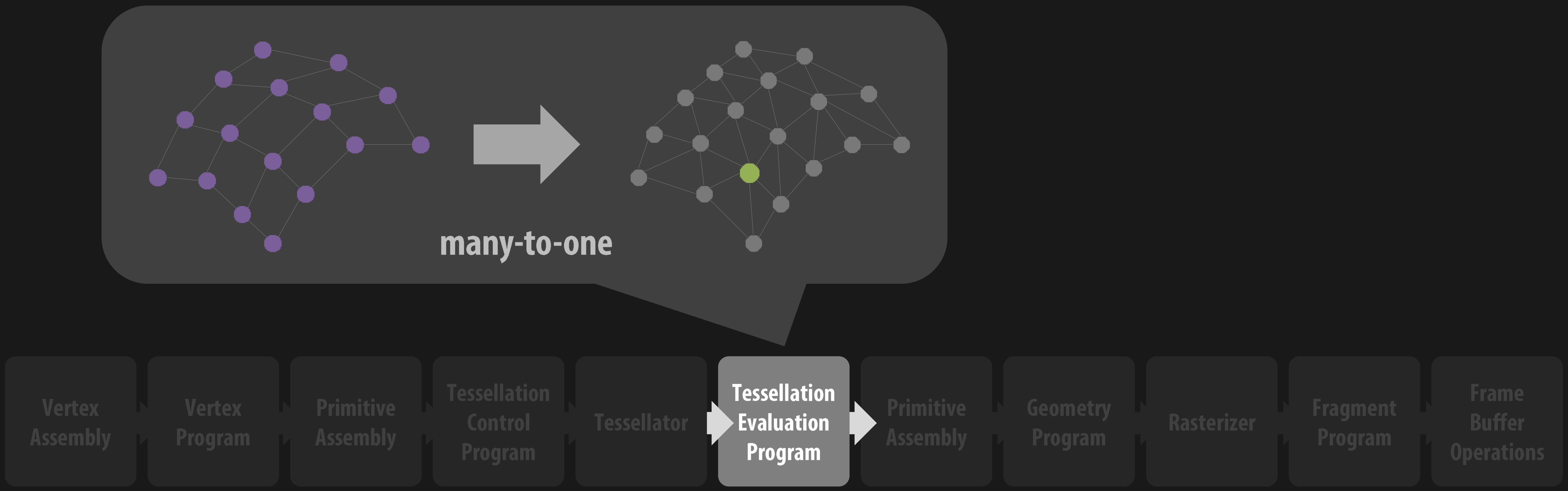
one-to-one

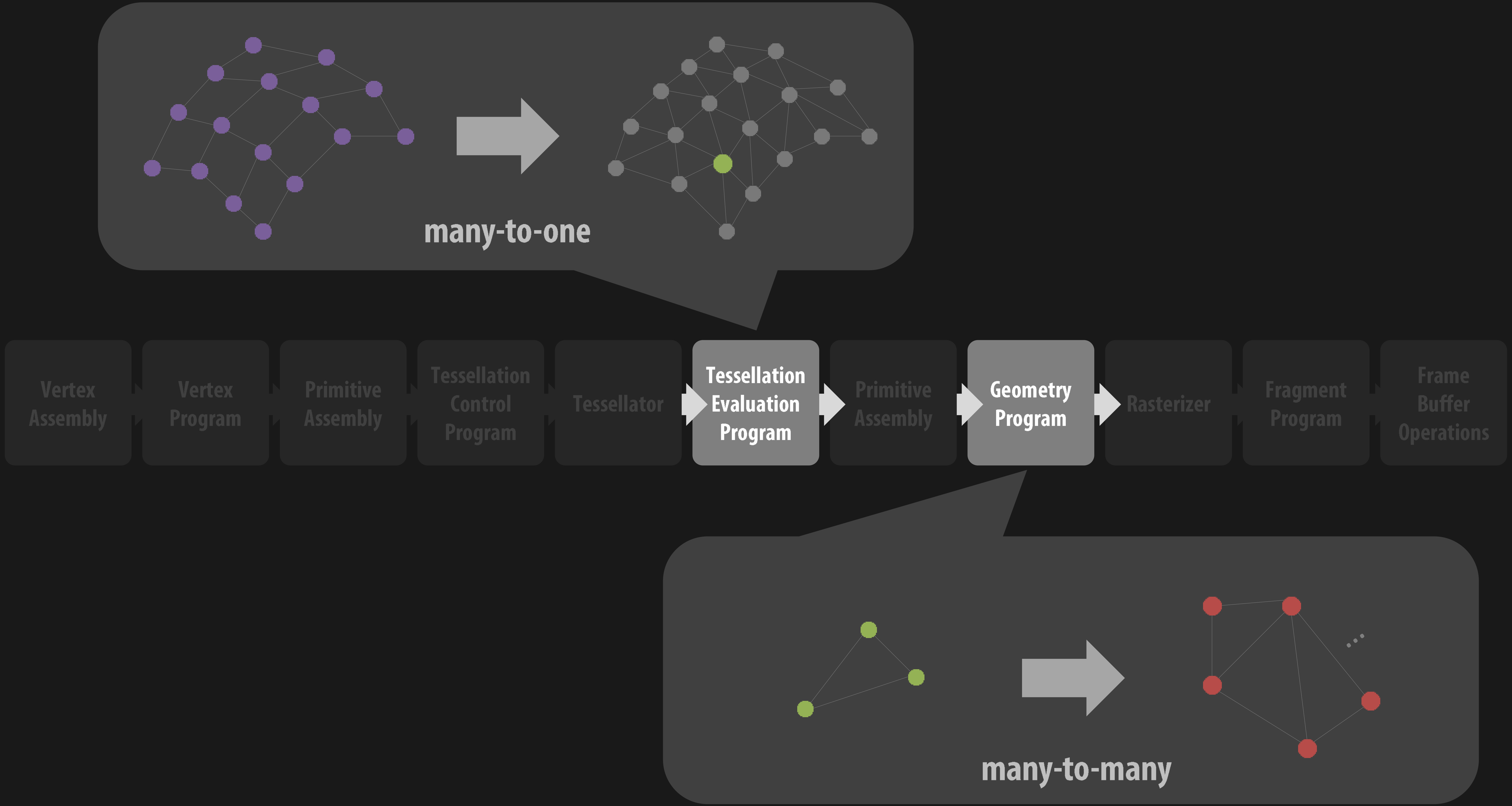


Vertex Assembly → Vertex Program → Primitive Assembly → Tessellation Control Program → Tessellator → Tessellation Evaluation Program → Primitive Assembly → Geometry Program → Rasterizer → Fragment Program → Frame Buffer Operations

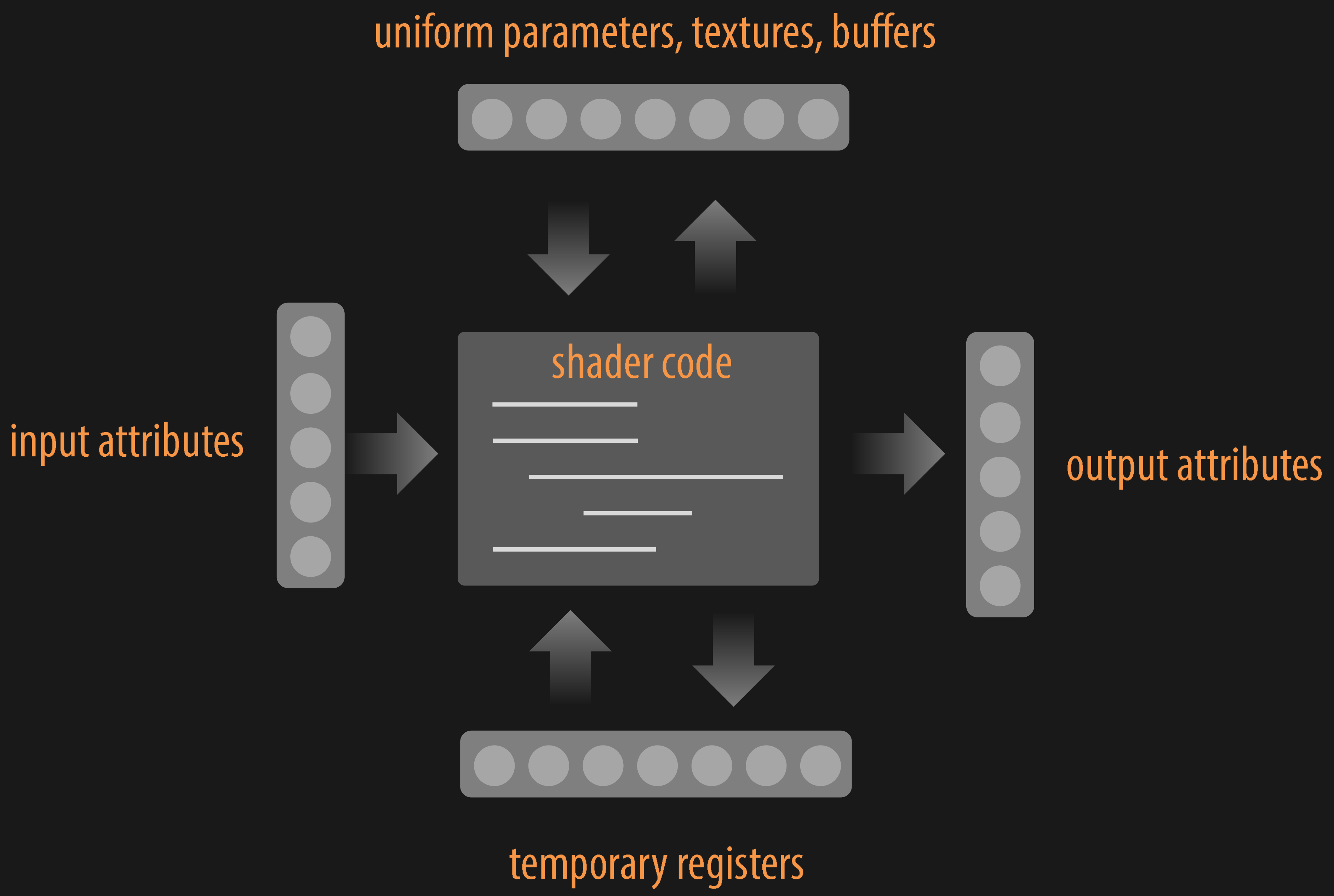

Pointwise
one-to-one
Vertex Assembly
Vertex Program
Primitive Assembly
Tessellation Control Program
Tessellator
Tessellation Evaluation Program
Primitive Assembly
Geometry Program
Rasterizer
Fragment Program
Frame Buffer Operations
Groupwise
N-to-M


Vertex Assembly
Vertex Program
Primitive Assembly
Tessellation Control Program
Tessellator
Tessellation Evaluation Program
Primitive Assembly
Geometry Program
Rasterizer
Fragment Program
Frame Buffer Operations


many-to-one
Vertex Assembly
Vertex Program
Primitive Assembly
Tessellation Control Program
Tessellator
Tessellation Evaluation Program
Primitive Assembly
Geometry Program
Rasterizer
Fragment Program
Frame Buffer Operations


many-to-one
Vertex Assembly
Vertex Program
Primitive Assembly
Tessellation Control Program
Tessellator
Tessellation Evaluation Program
Primitive Assembly
Geometry Program
Rasterizer
Fragment Program
Frame Buffer Operations
...
many-to-many

Inside a Programmable Stage



[Lindholm, Kilgard, and Moreton *A User-Programmable Vertex Engine* 2001]
[Blythe *The Direct3D 10 System* 2006]

Configuring the Pipeline

Programmable Pipeline Stages

Vertex Program

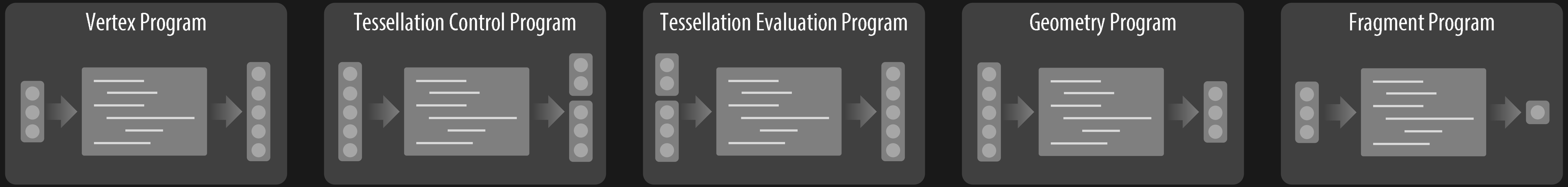
Tessellation Control Program

Tessellation Evaluation Program

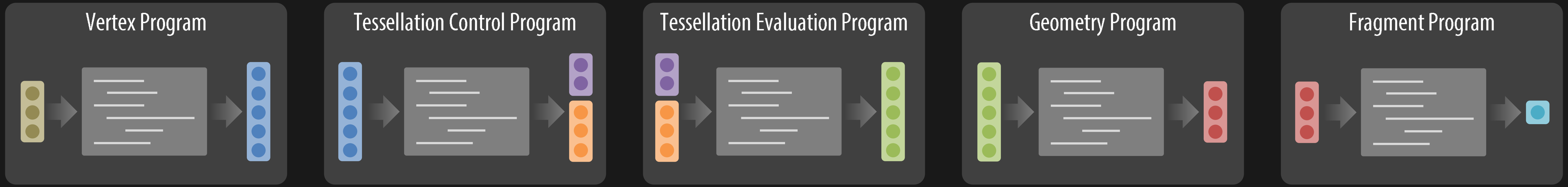
Geometry Program

Fragment Program

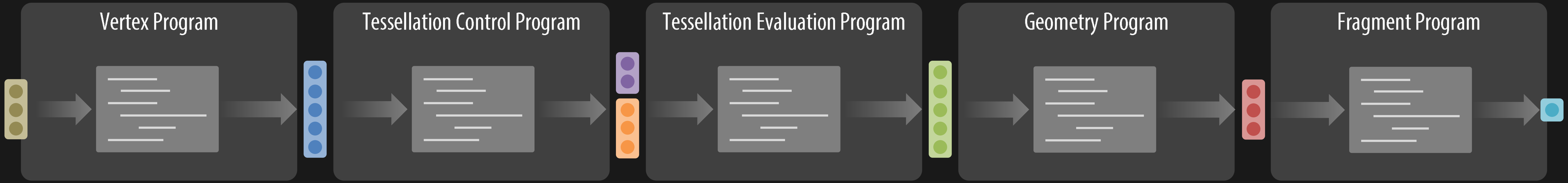
Bind One Shader to Each Programmable Stage



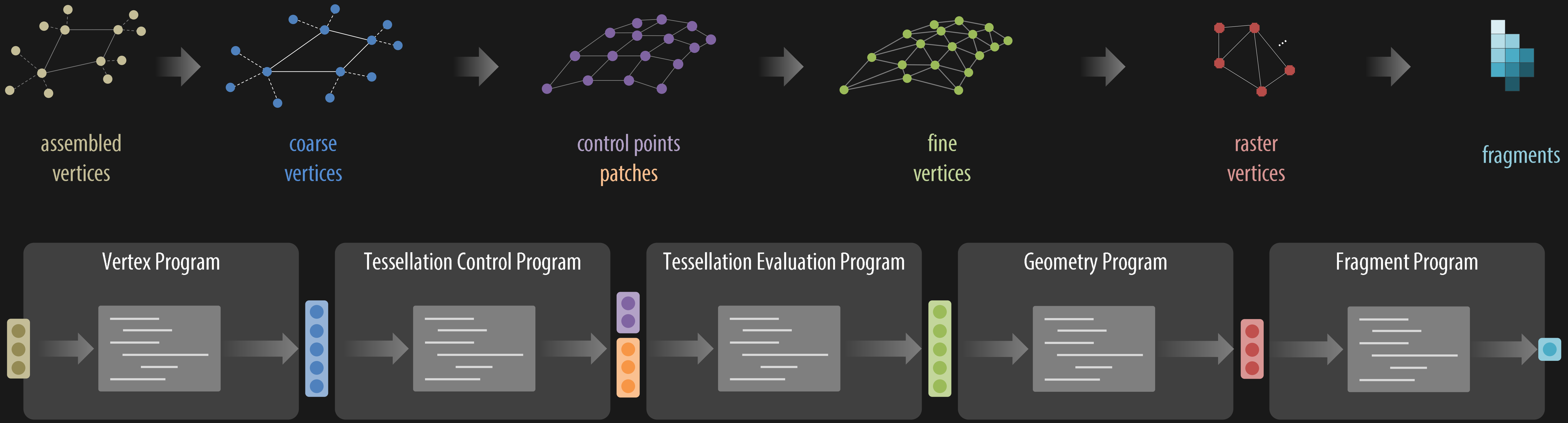
Signatures Must Match Along Boundaries



Signatures Must Match Along Boundaries



Boundaries Correspond to Types of Records



Stages Communicate Records

Record Types

assembled vertex

coarse vertex

control point

patch

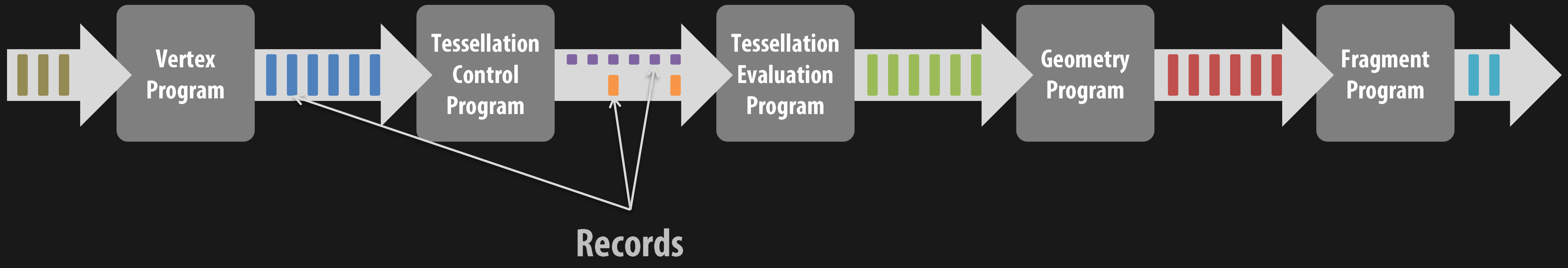
...

fine vertex

raster vertex

fragment

Pipeline



API Evolution

from

a framework for configuring a rendering pipeline

to

a framework for authoring pipeline configurations

The OpenGL State Machine

```
// pseudo-OpenGL:

EnableDepthTest();
SetDepthFunc(LESS);
DisableStencilTest();
EnableBlend();
SetBlendFactors(ONE, ONE_MINUS_SRC_ALPHA);
SetColorCombineOp(MODULATE);
// ...

SetTexture(0, diffuseTex);
SetTexture(1, specularTex);

DrawIndexed(indexCount);
```

Coarse-Grained State Object

```
// pseudo-D3D10/11:

SetDepthStencilState(depthStencilState);
SetBlendState(blendState);
SetVetexShader(vertexShader);
SetFragmentShader(fragmentShader);

SetTextures(0, textureCount, &textures[0]);

DrawIndexed(indexCount);
```

Pipeline State Object

```
// pseudo-Vulkan/D3D12/Metal/…:

SetPipelineState(pipelineState);

SetBindingTable(bindingTable);

DrawIndexed(indexCount);
```

“Stateless”

```
// pseudo- Some Hypothetical API:

DrawIndexed(pipelineConfig, bindingTable, indexCount);
```

“Stateless”

```
// pseudo- Some Hypothetical API:

DrawIndexed(pipelineConfig, bindingTable, indexCount);


// CUDA

KernelFunc<<gridDims, blockDims>>( arg0, arg1, ... );
```

Pipeline Configuration is a Fundamental Concept

- Granularity of work dispatch
- Granularity of final code generation
- New APIs starting to reflect this in state model

What does this mean for shading languages?

- Language for authoring pipeline configurations

- Want to enable good development practice
  - Modularity
  - Composability

Modularity

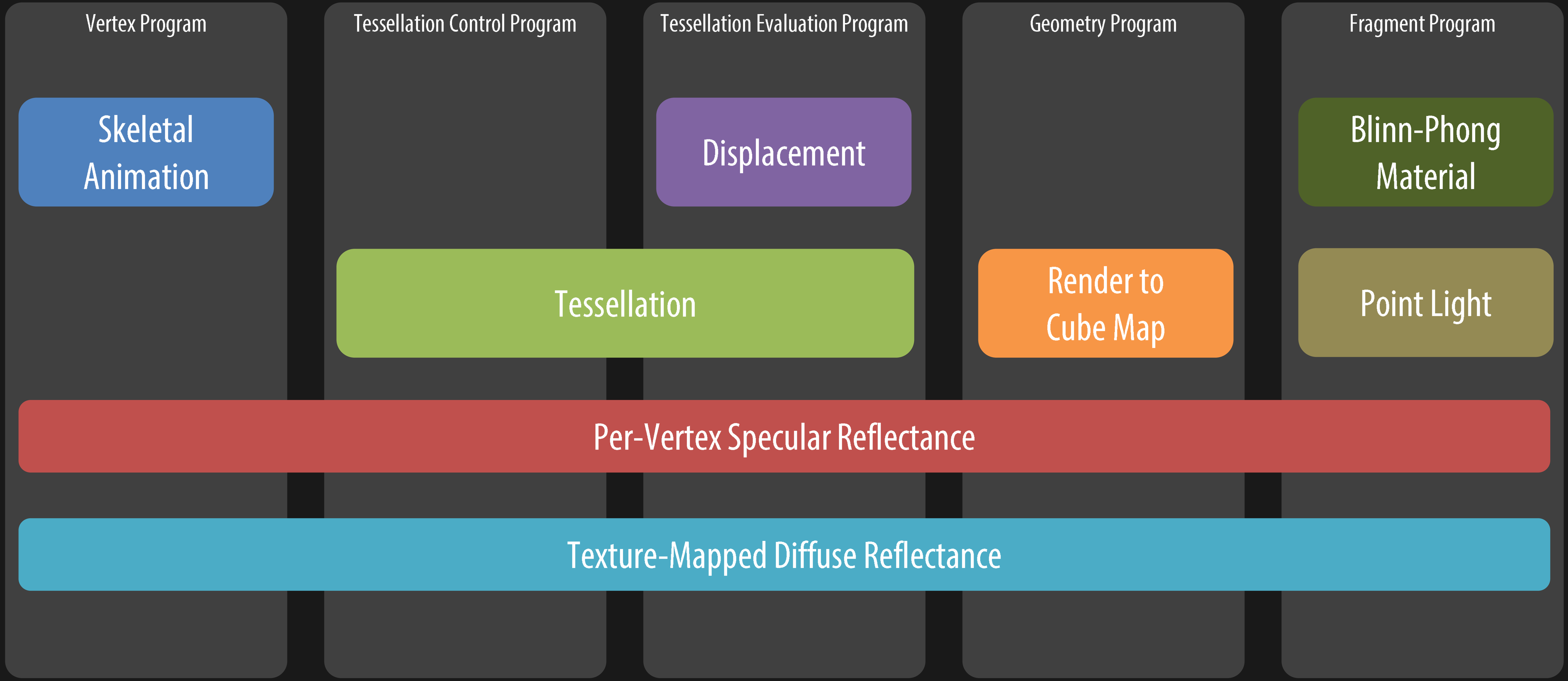
The physical decomposition of a program into modules reflects its logical decomposition into concerns.

[Parnas 1971, 1972]
[Dijkstra 1982]

Combinations of Features

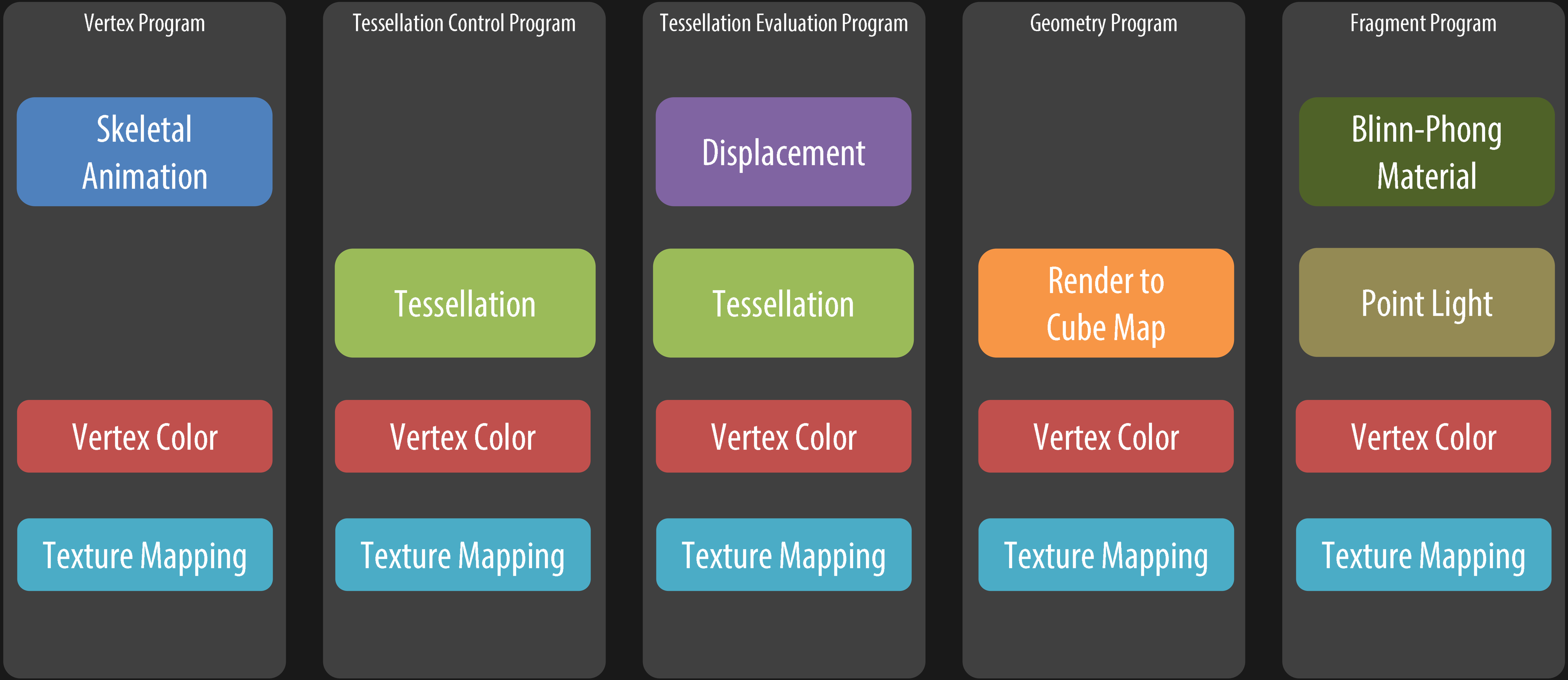
Geometry	Material	Lighting
Skeletal Animation	Normal Mapping	Point Light
Morph Target Animation	Blinn-Phong	Spot Light
Render to Cube Map	Cook-Torrance	Directional Light
ACC Tessellation	Oren-Nayar	Cascaded Shadow Map
Displacement Mapping	Kajiya-Kay	Variance Shadow Map
	…	…

Features Don’t Line Up With Stages



Cross-Cutting Concerns Impact Modularity

[Kiczales et al. 1997]



Cross-Cutting Concerns Impact Modularity

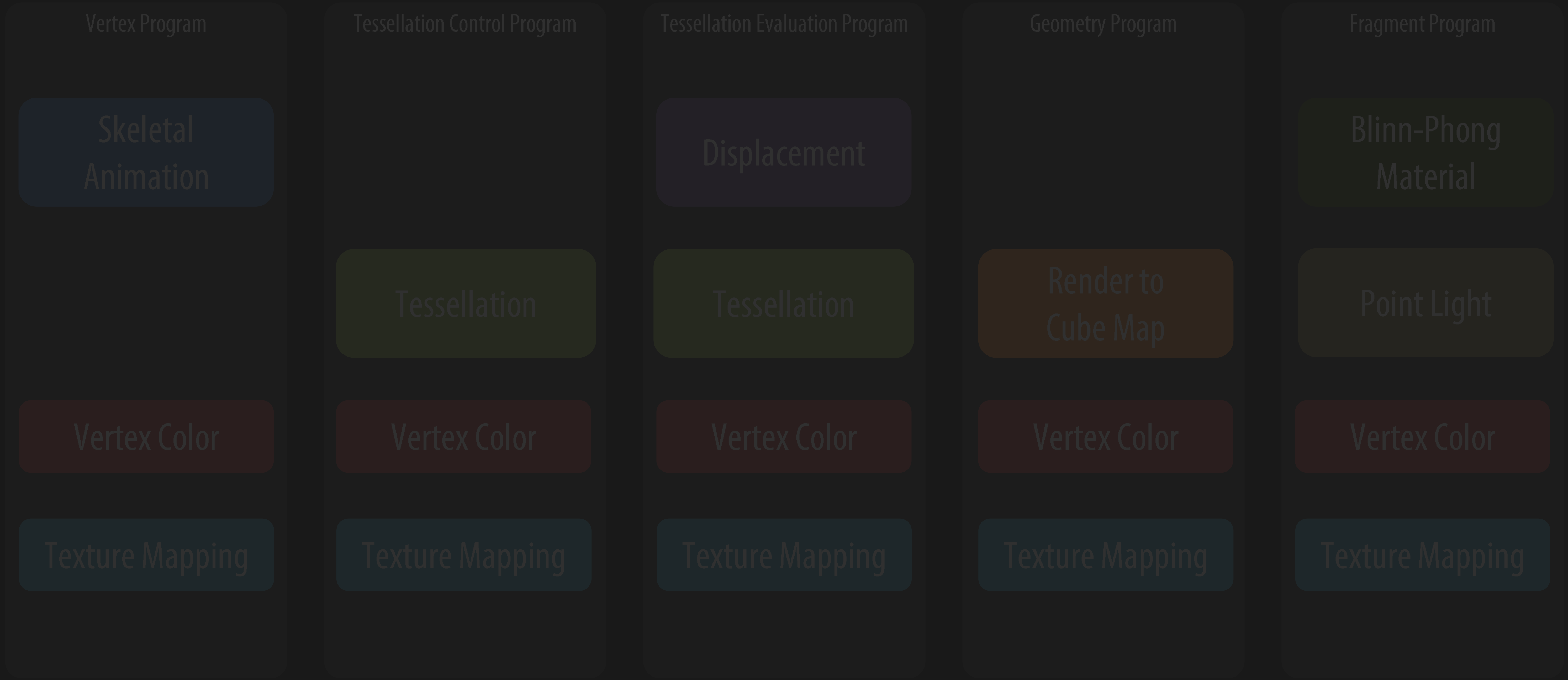
[Kiczales et al. 1997]

Vertex Program	Tessellation Control Program	Tessellation Evaluation Program	Geometry Program	Fragment Program
Skeletal Animation		Displacement		Blinn-Phong Material
	Tessellation	Tessellation	Render to Cube Map	Point Light
Vertex Color	Vertex Color	Vertex Color	Vertex Color	Vertex Color
Texture Mapping	Texture Mapping	Texture Mapping	Texture Mapping	Texture Mapping

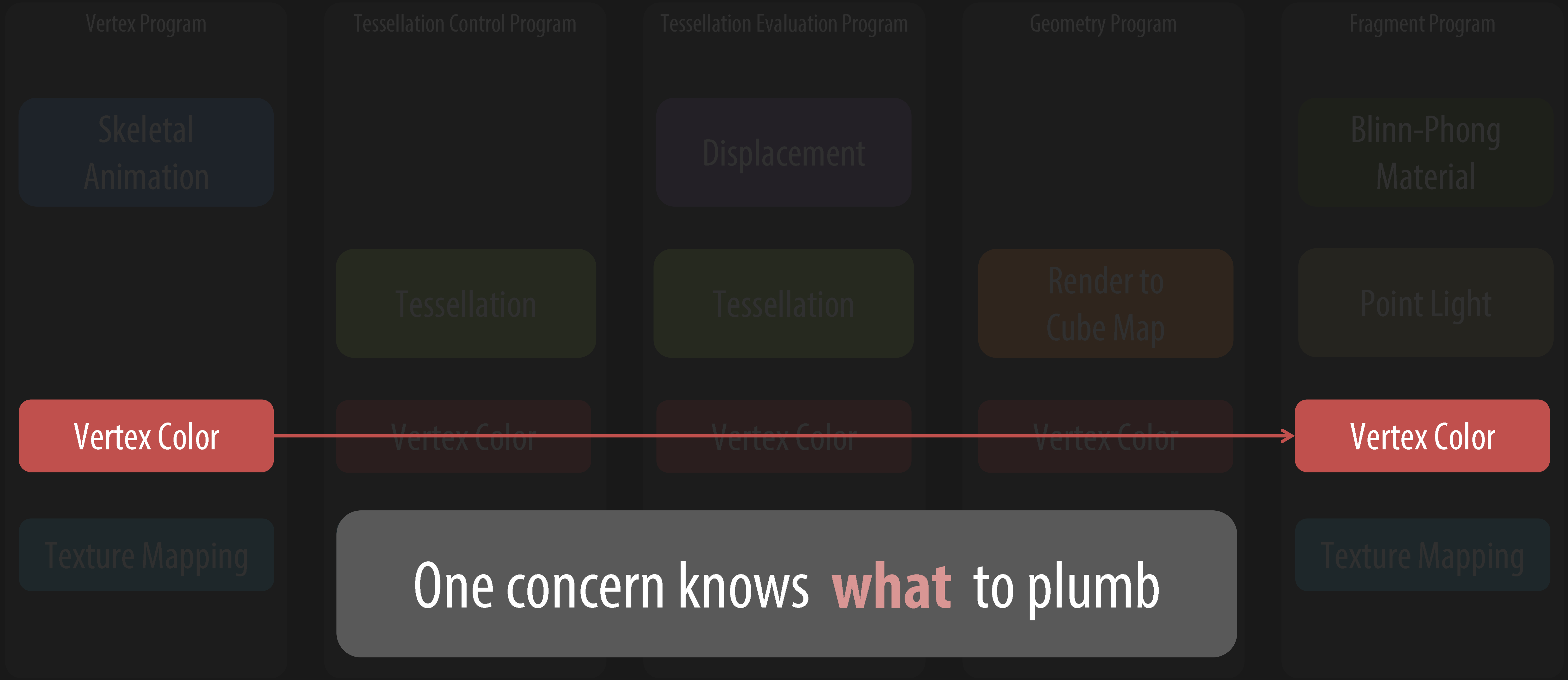
Composability

Thoughtfully-designed modules may be combined, without requiring changes to their implementation.

Plumbing

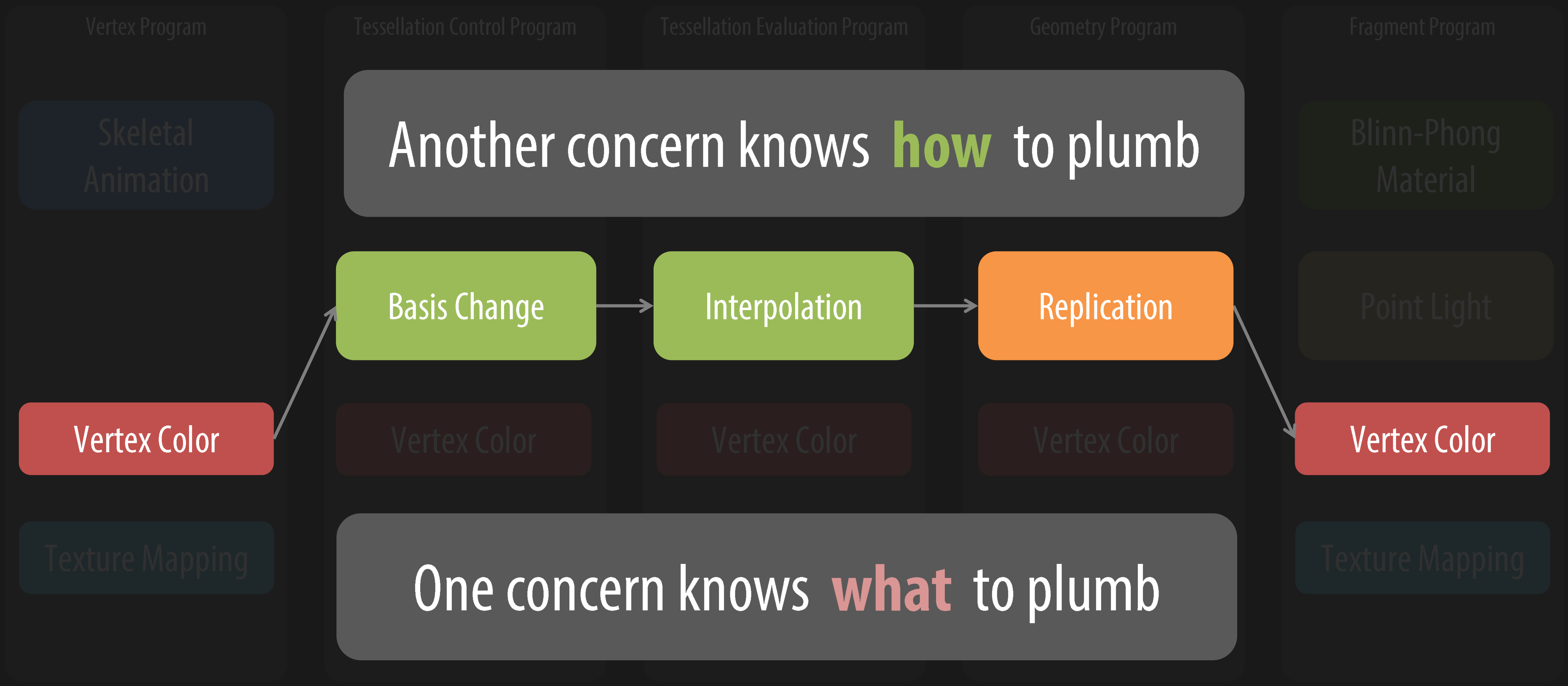

Vertex Program
Tessellation Control Program
Tessellation Evaluation Program
Geometry Program
Fragment Program
Skeletal Animation
Displacement
Blinn-Phong Material
Tessellation
Tessellation
Render to Cube Map
Point Light
Vertex Color
Vertex Color
Vertex Color
Vertex Color
Vertex Color
Texture Mapping
Texture Mapping
Texture Mapping
Texture Mapping
Texture Mapping

Plumbing



One concern knows **what** to plumb

Plumbing


Vertex Program
Tessellation Control Program
Tessellation Evaluation Program
Geometry Program
Fragment Program
Skeletal Animation
Another concern knows how to plumb
Blinn-Phong Material
Basis Change
Interpolation
Replication
Point Light
Vertex Color
Vertex Color
Vertex Color
Vertex Color
Vertex Color
Texture Mapping
One concern knows what to plumb
Texture Mapping

Brief?

History of Real-Time Shading Languages

Shader Graphs

Rates of Computation

Pipeline Shaders

Shade Trees

[Cook 1984]

shader graphs

Shade Trees

[Cook 1984]

```
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;
```

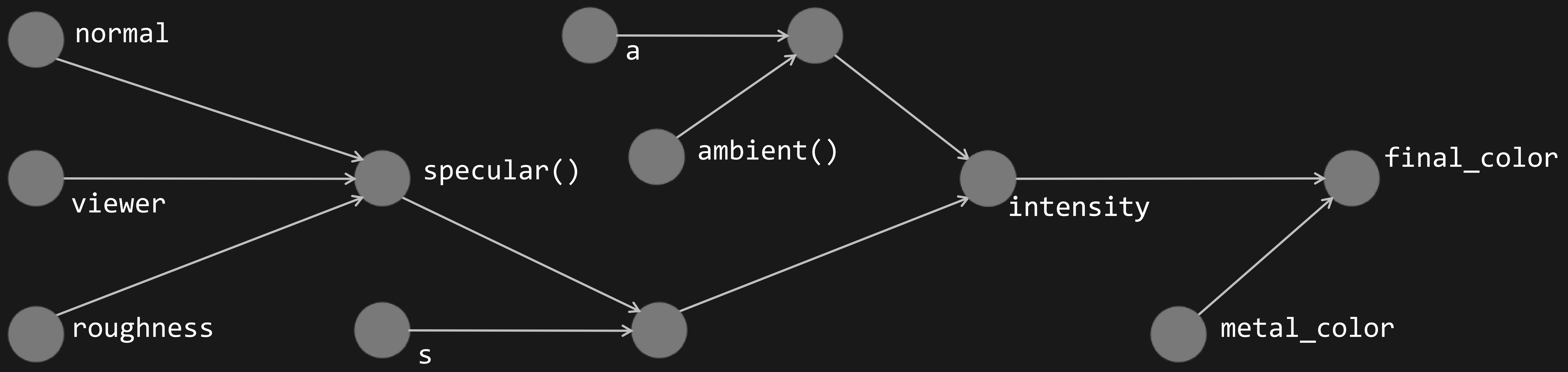
Key:
type
constant

Shade Trees

[Cook 1984]

```
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

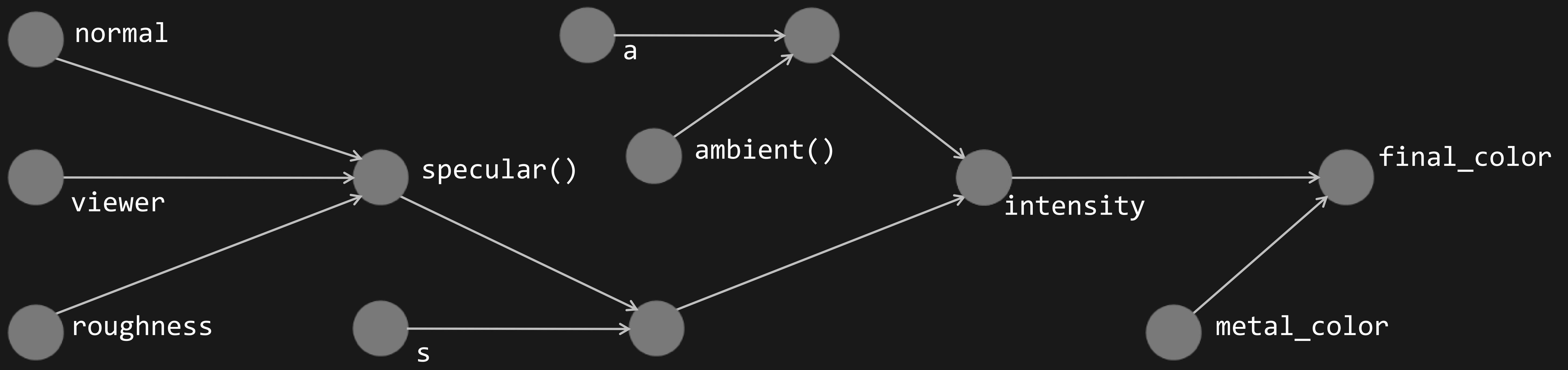


Shade Trees

[Cook 1984]

```
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

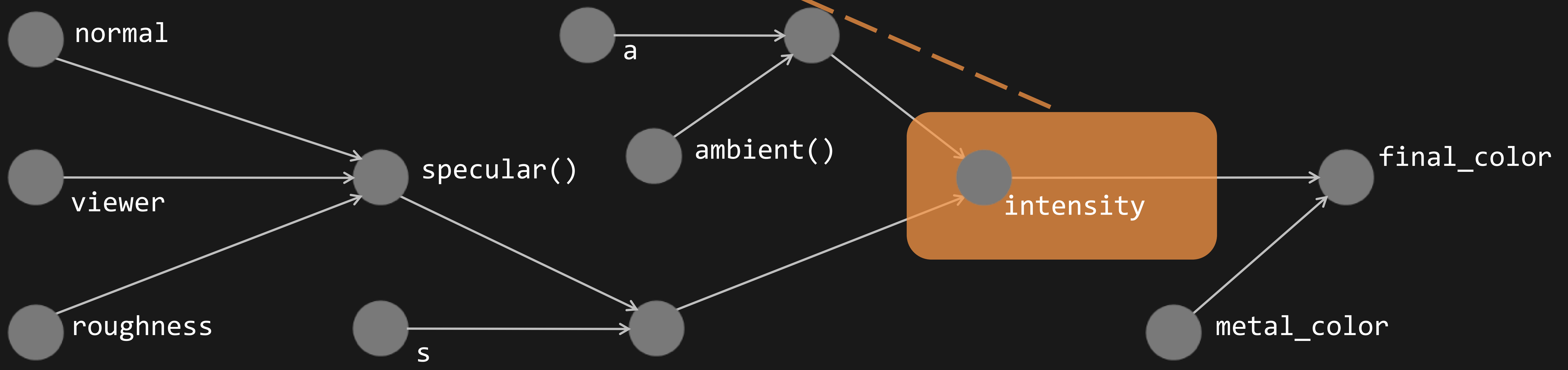


Shade Trees

[Cook 1984]

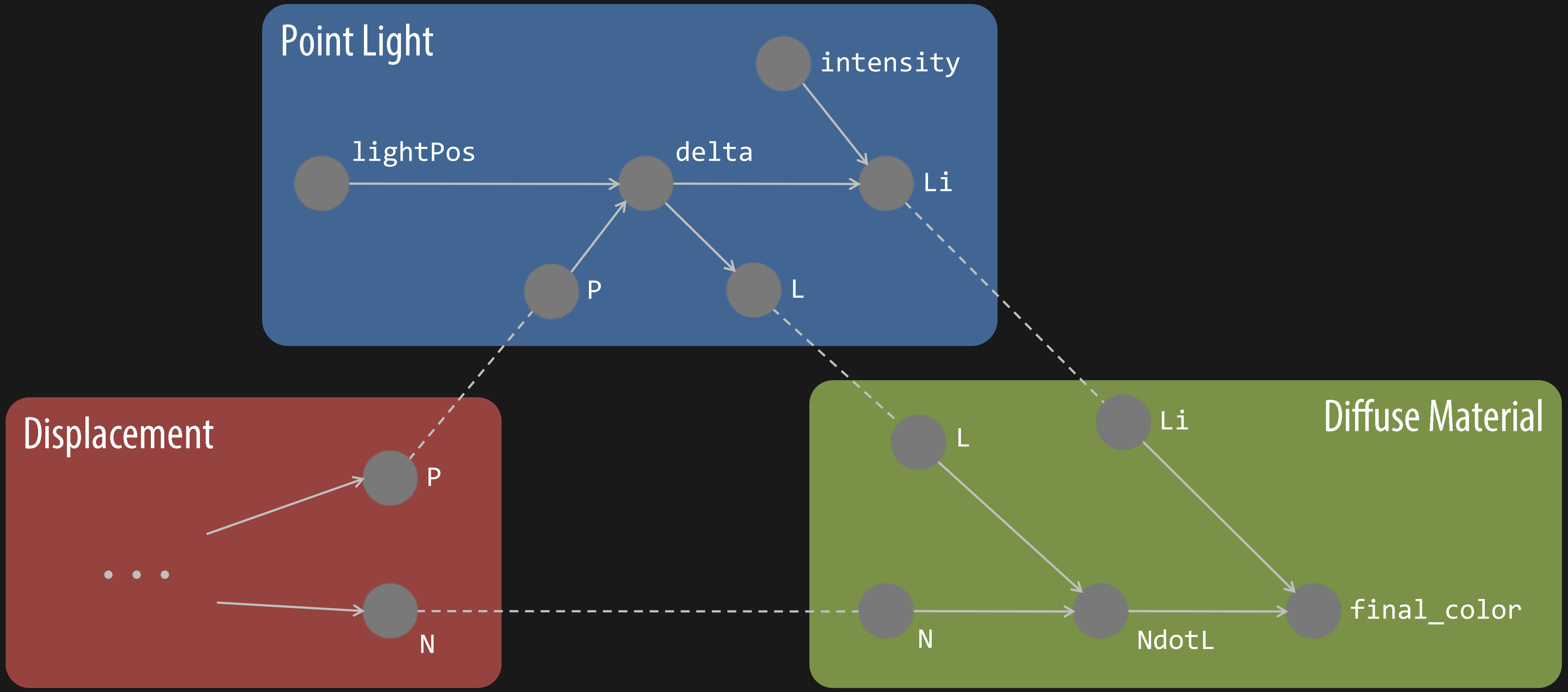
```
float a = 0.5, s = 0.5;
float roughness = 0.1;
float intensity;
color metal_color = (1,1,1);
intensity = a*ambient() +
     s*specular(normal,viewer,roughness);
final_color = intensity * metal_color;
```

Key:
type
constant



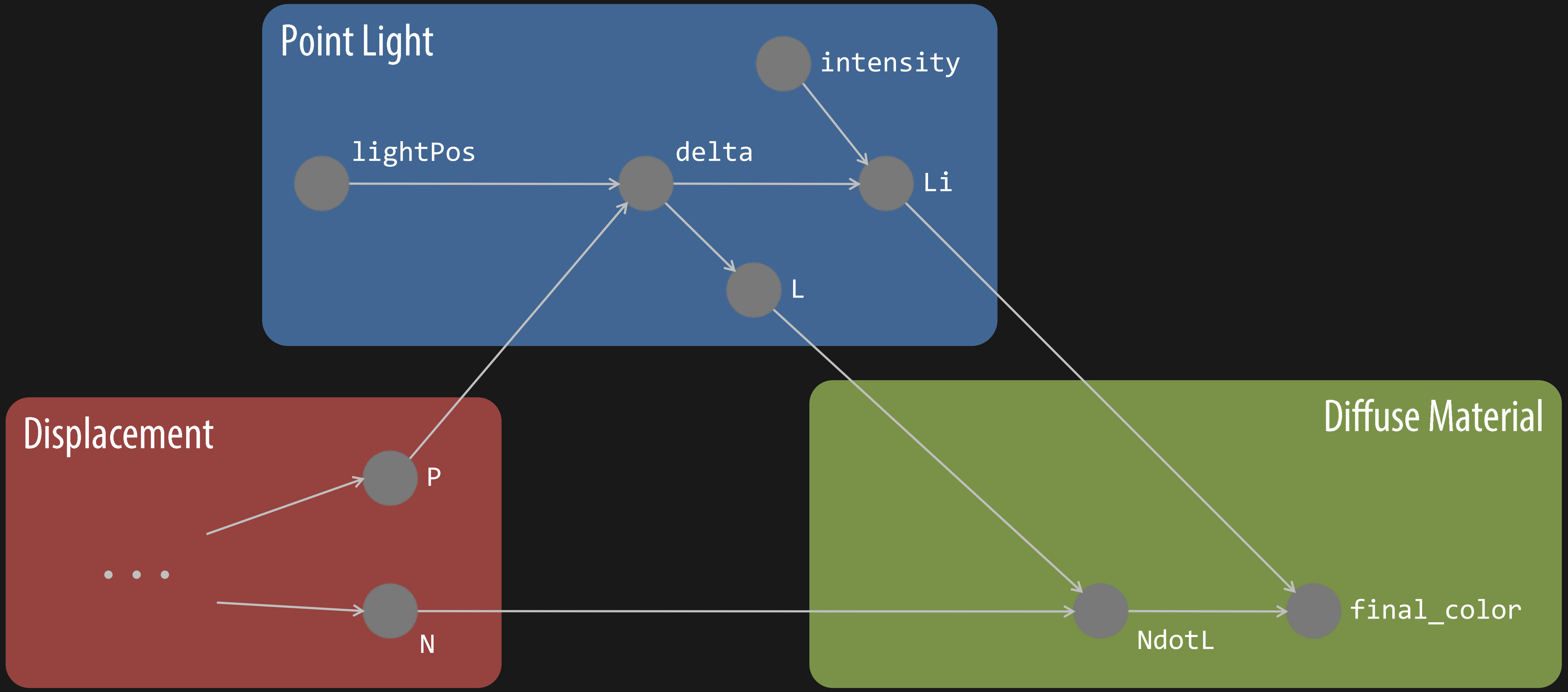
Shader Graphs are Composable

[Cook 1984]



Shader Graphs are Composable

[Cook 1984]



RenderMan Shading Language

[Hanrahan and Lawson 1990]

rates of computation

Key:
type
rate

computed per-batch

```
uniform vector L;
varying vector N;

...

L = normalize(L);

...

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

computed per-sample

Real-Time Shading Language

[Proudfoot et al. 2001]

pipeline shaders

```
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

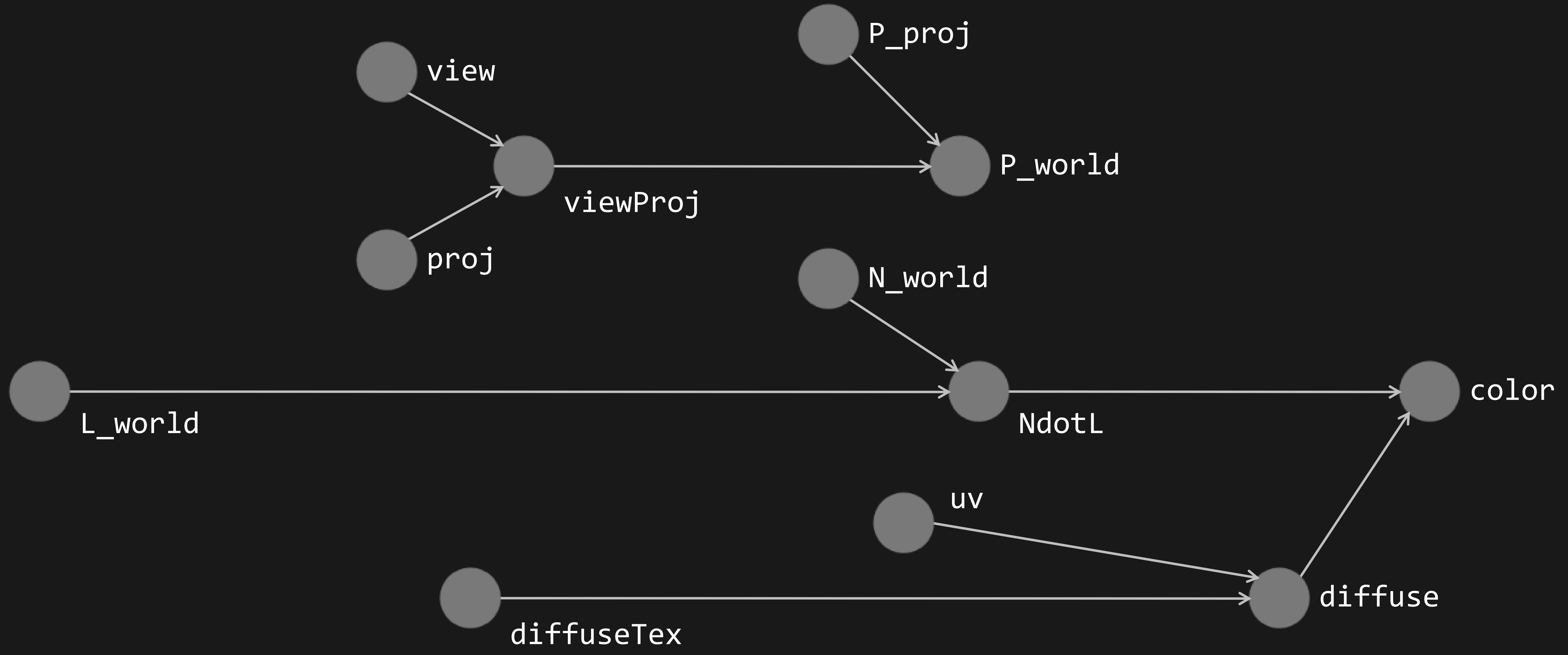
```
surface shader float4 Simple( ... )
{
    constant       float3  L_world  = normalize({1, 1, 1});

    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

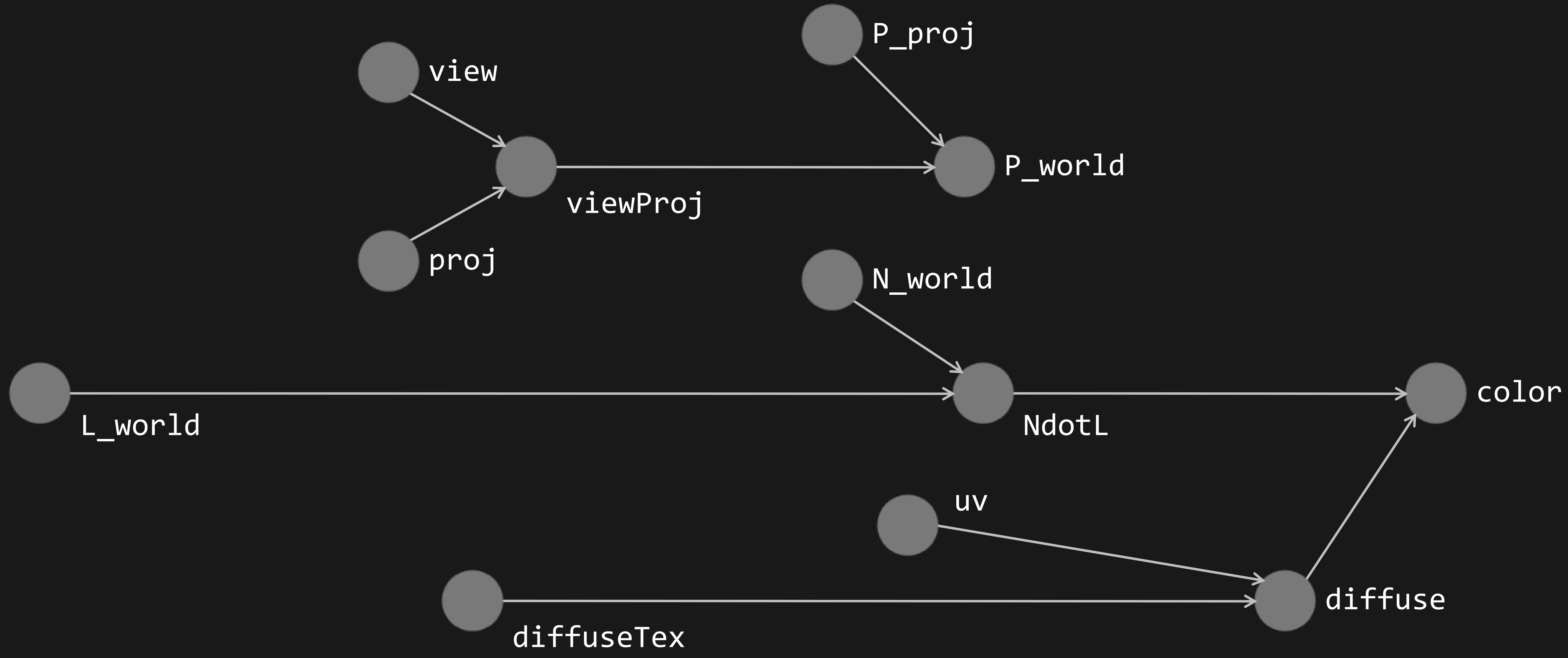
```
surface shader float4 Simple( ... )
{
    constant       float3  L_world  = normalize({1, 1, 1});

    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

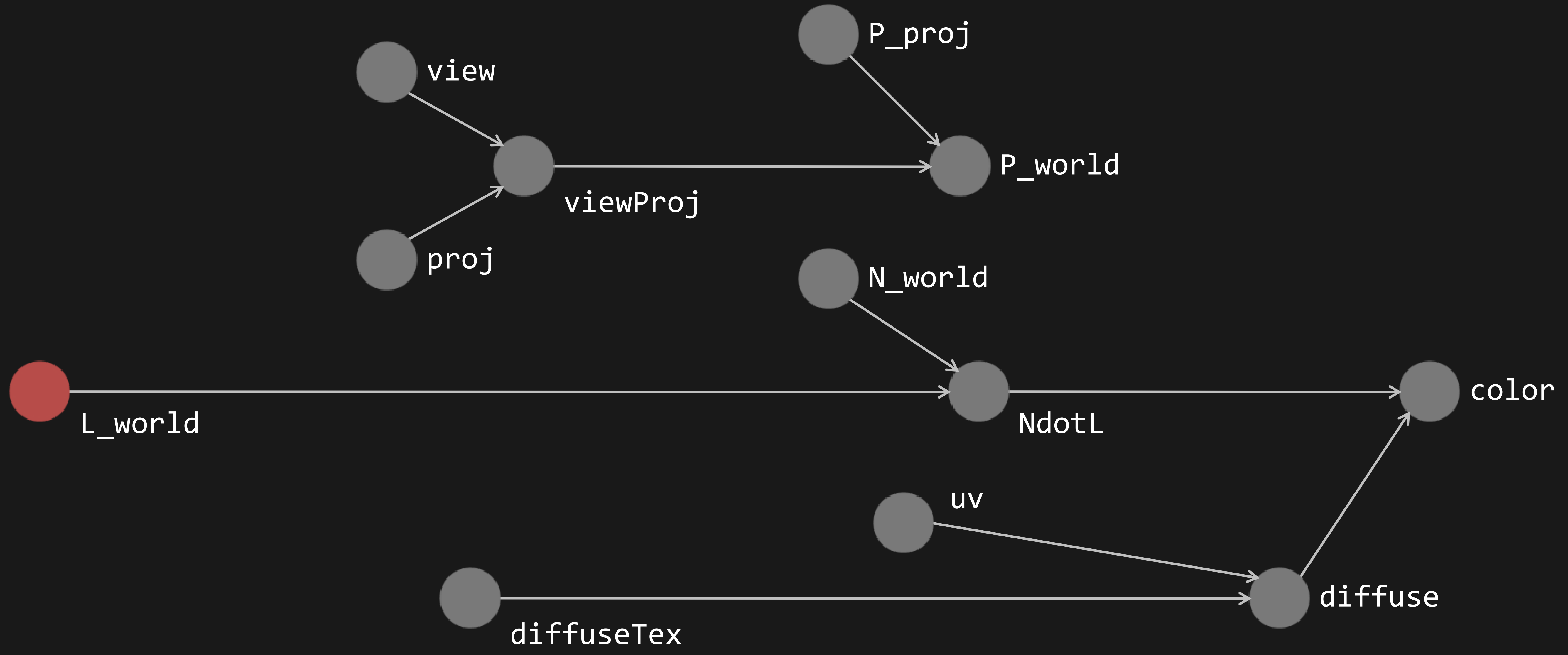


Color by Rates



Color by Rates

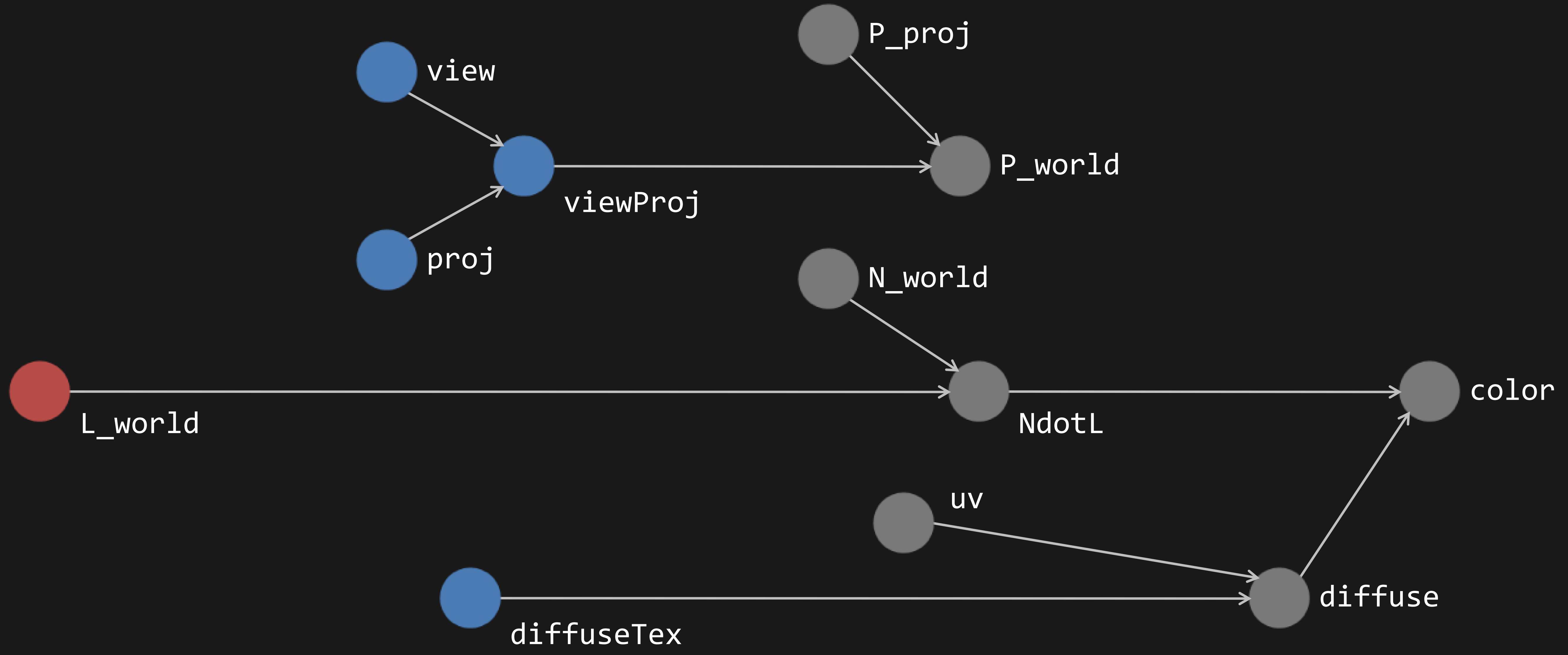
constant



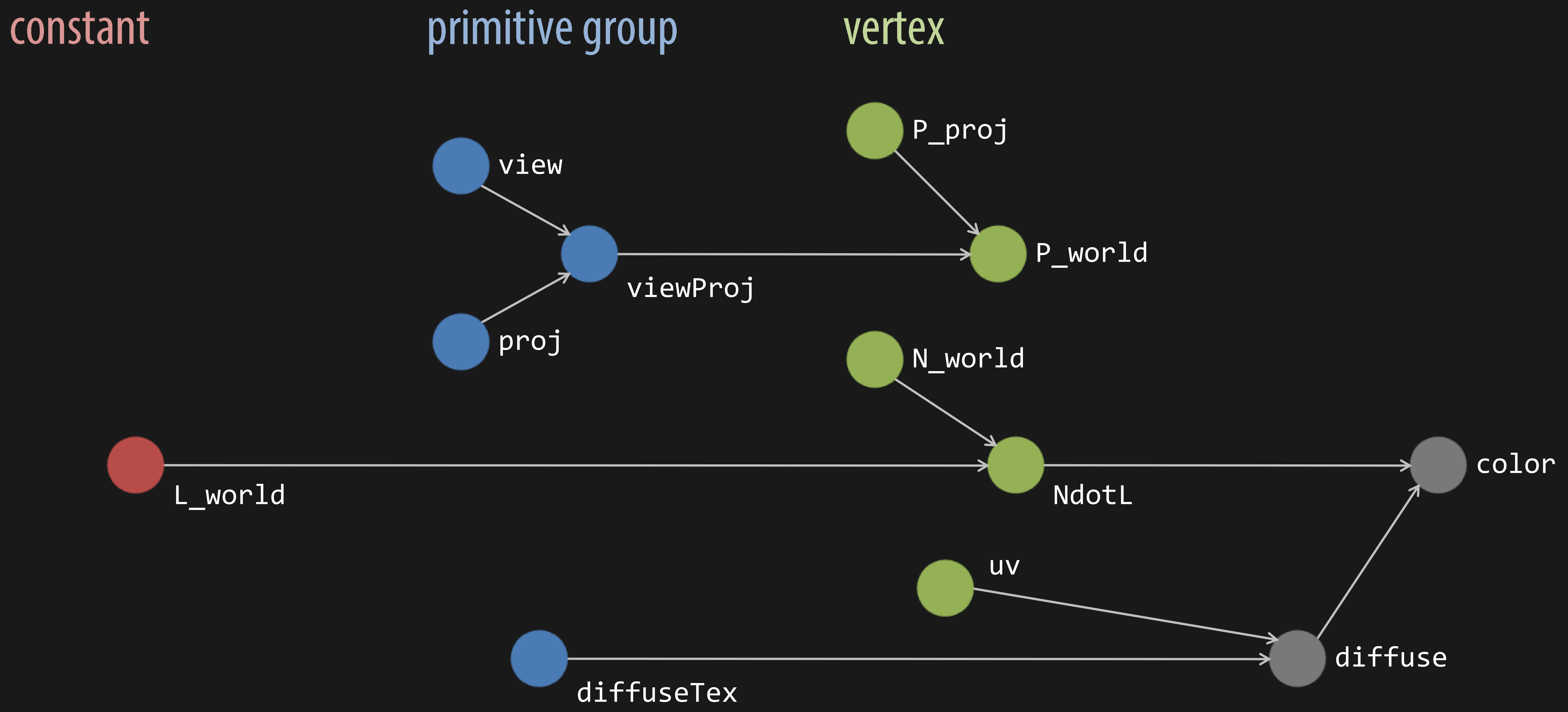
Color by Rates

constant

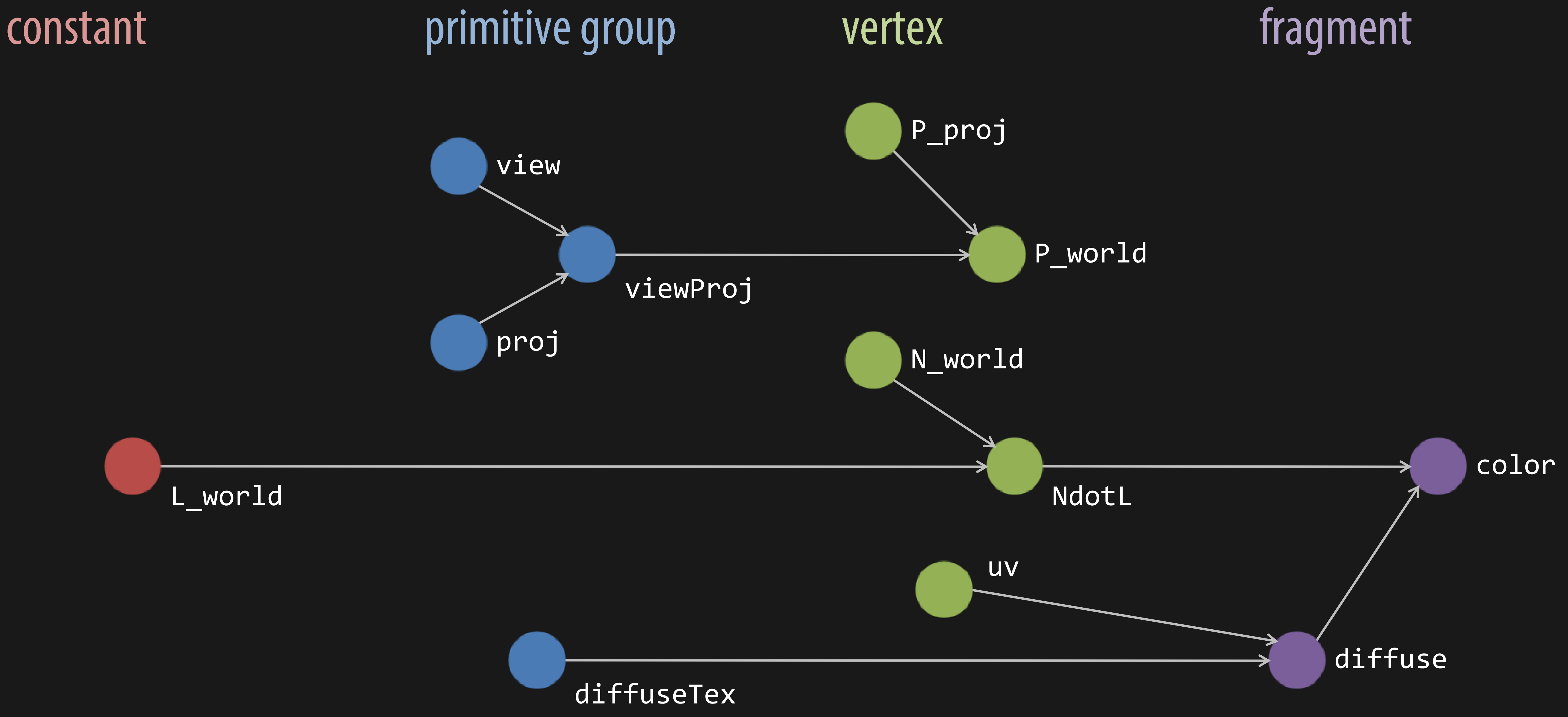
primitive group



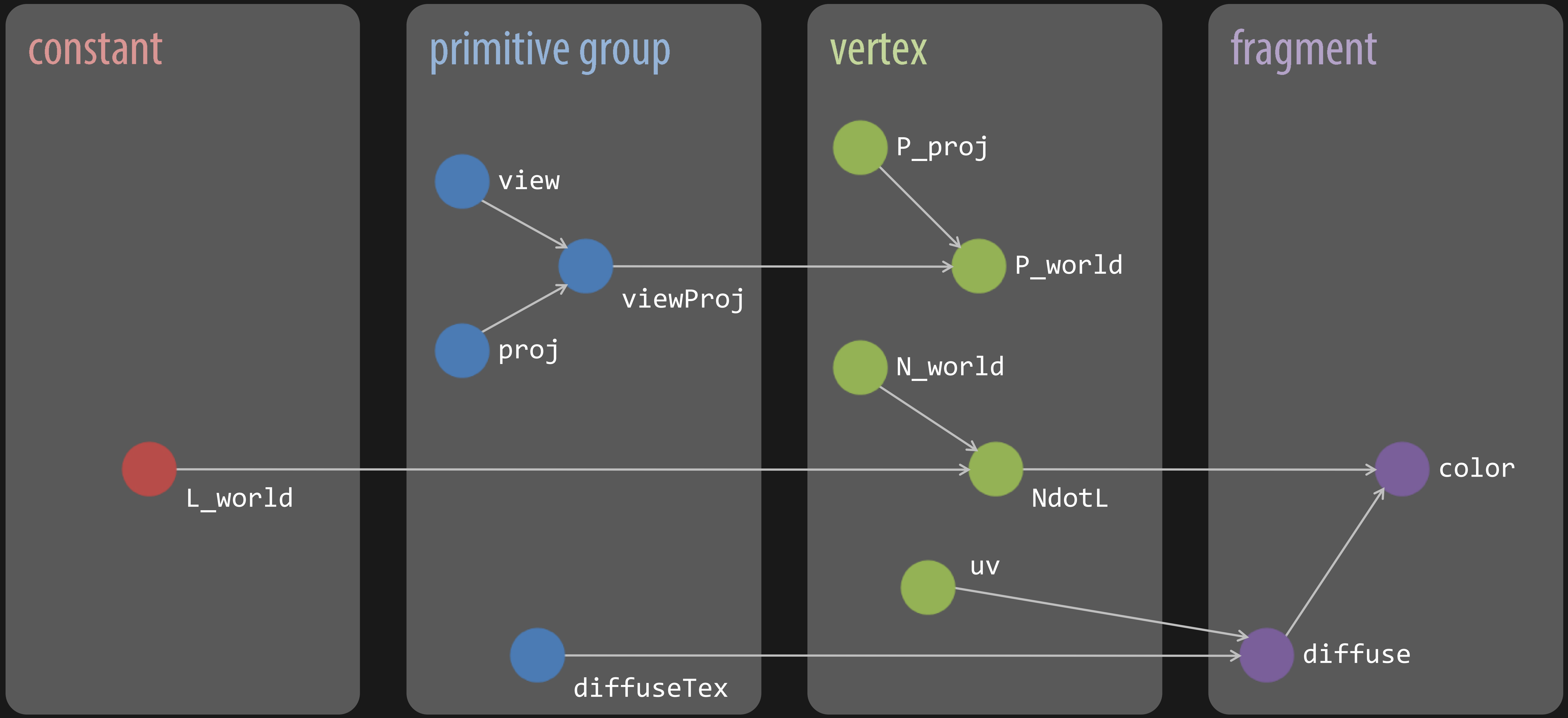
Color by Rates



Color by Rates

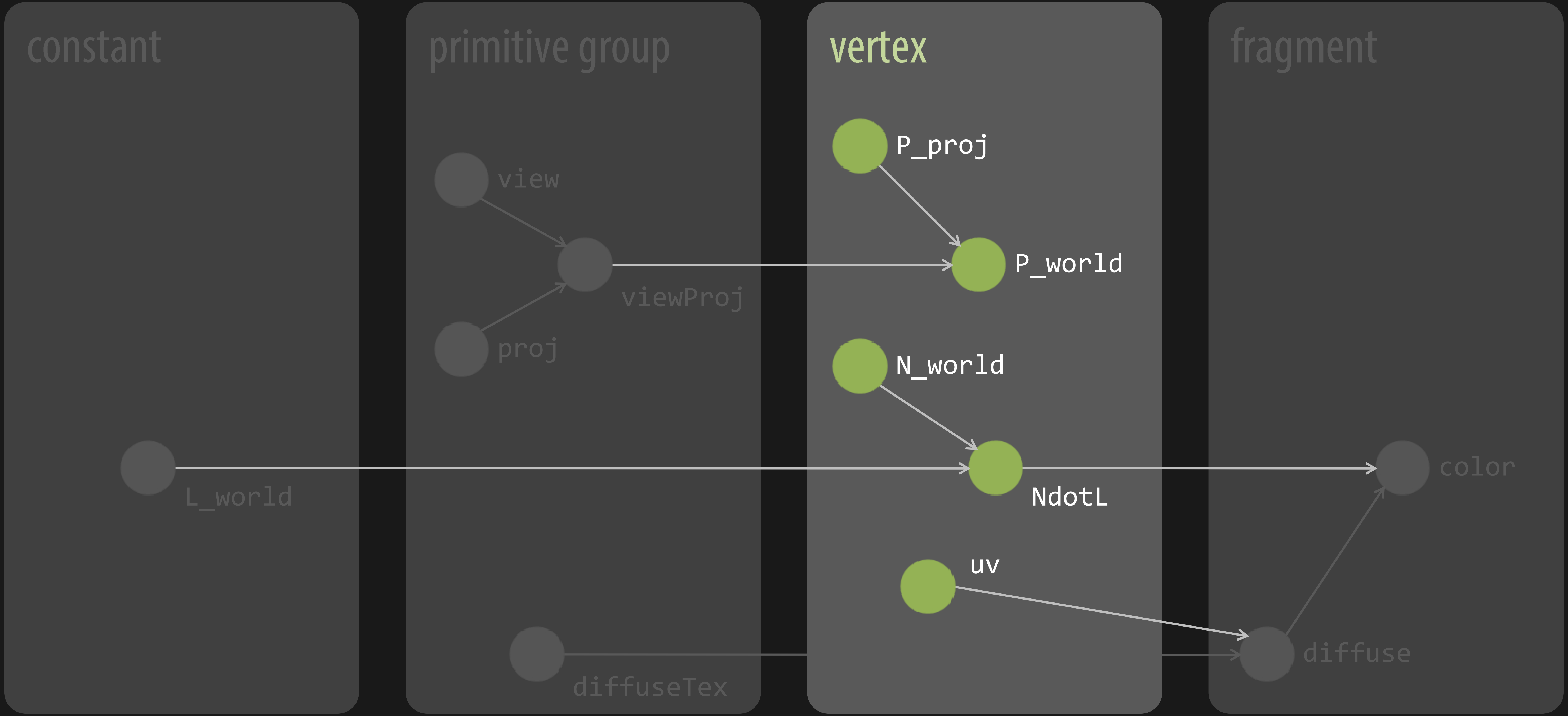


Partition

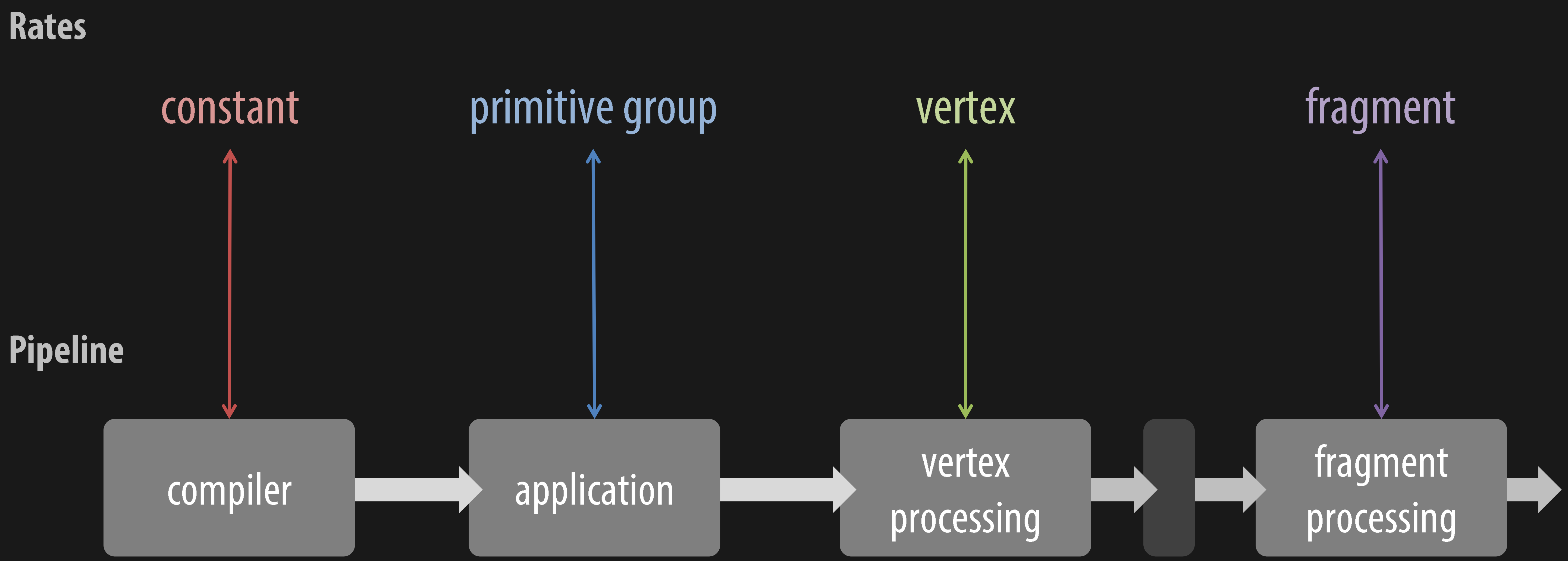


constant
primitive group
vertex
fragment
view
viewProj
proj
P_proj
P_world
N_world
NdotL
L_world
uv
diffuseTex
color
diffuse

Partition


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

Rates correspond to programmable stages



Cg, HLSL, and GLSL

[Mark et al. 2003]
[Microsoft]
[OpenGL ARB]

shader-per-stage languages

Cg, HLSL, GLSL

[Mark et al. 2003]
[Microsoft]
[OpenGL ARB]

- No pipeline shaders
  - One shader per stage
- No shader graphs
  - Procedural, C-like language
- Why?

Why one shader per stage?

[Mark et al. 2003]

- Performance transparency
- Mix-and-match flexibility
- Data-dependent control flow

Metaprogramming

- **Effect systems** [NVIDIA 2010; Microsoft 2010; Lalonde and Schenk 2002]
- **EDSLs** [McCool et al. 2002; Lejdfors and Ohlsson 2004; Kuck and Wesche 2009]
- **Über-shaders**

Rationale, revisted

- Performance transparency
- Mix-and-match flexibility
- Data-dependent control flow

with an über-shader, we have already given up on these

Rationale, revisted

- Performance transparency
- Mix-and-match flexibility
- Data-dependent control flow
- More complex pipelines

with an über-shader, we have already given up on these

Spark

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

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;
}
```

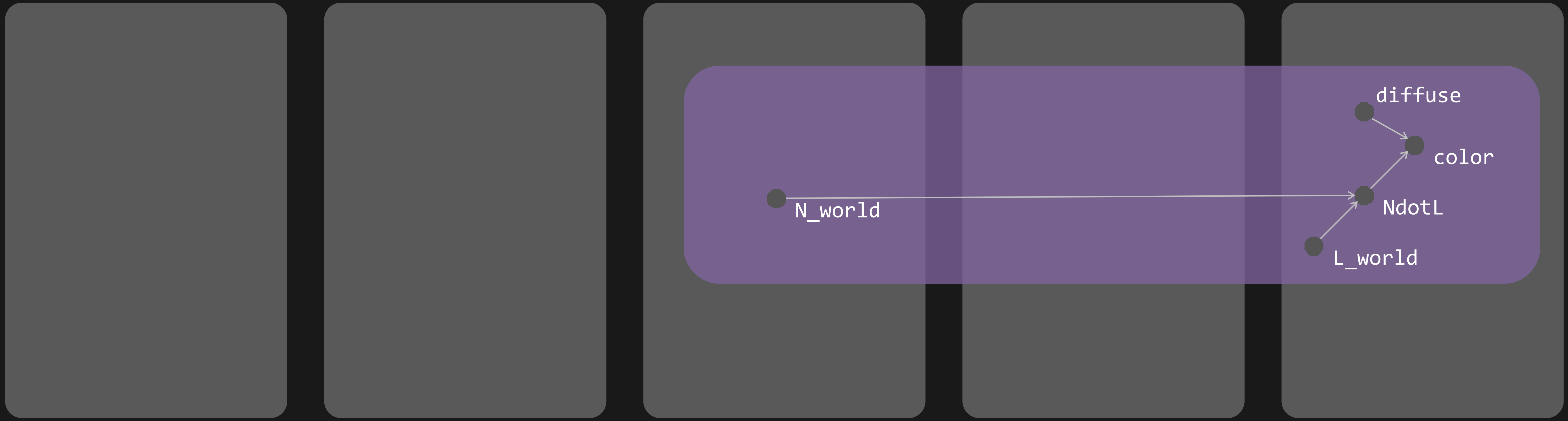
Key:
keyword
type
constant
rate

Define Shader Graphs as Classes

```
shader class SimpleDiffuse
{
    ...
}
```

Define Shader Graphs as Classes

```
shader class SimpleDiffuse
{
    ...
}
```

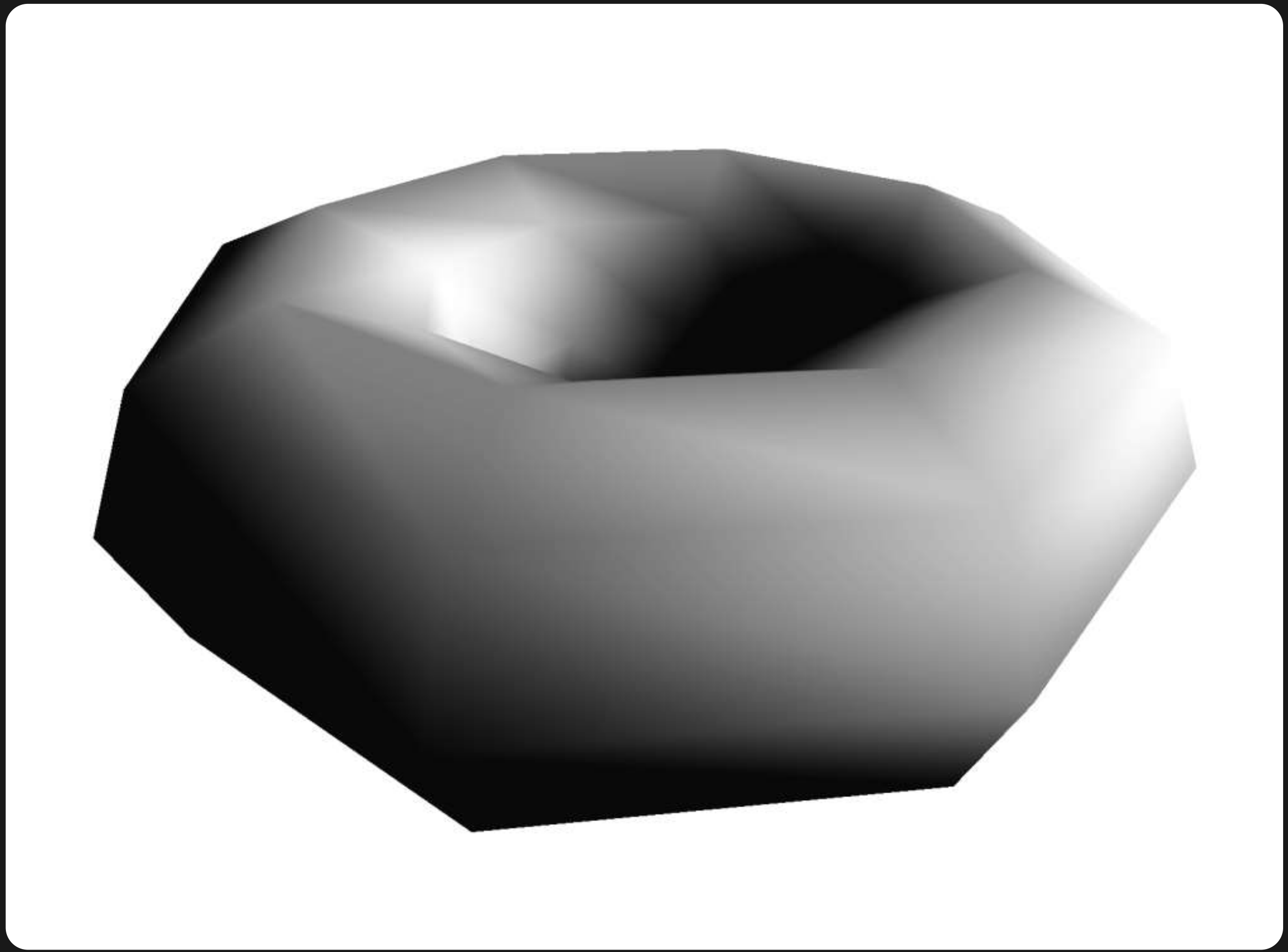


Define Shader Graphs as Classes

```
shader class SimpleDiffuse
{
    ...
}

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

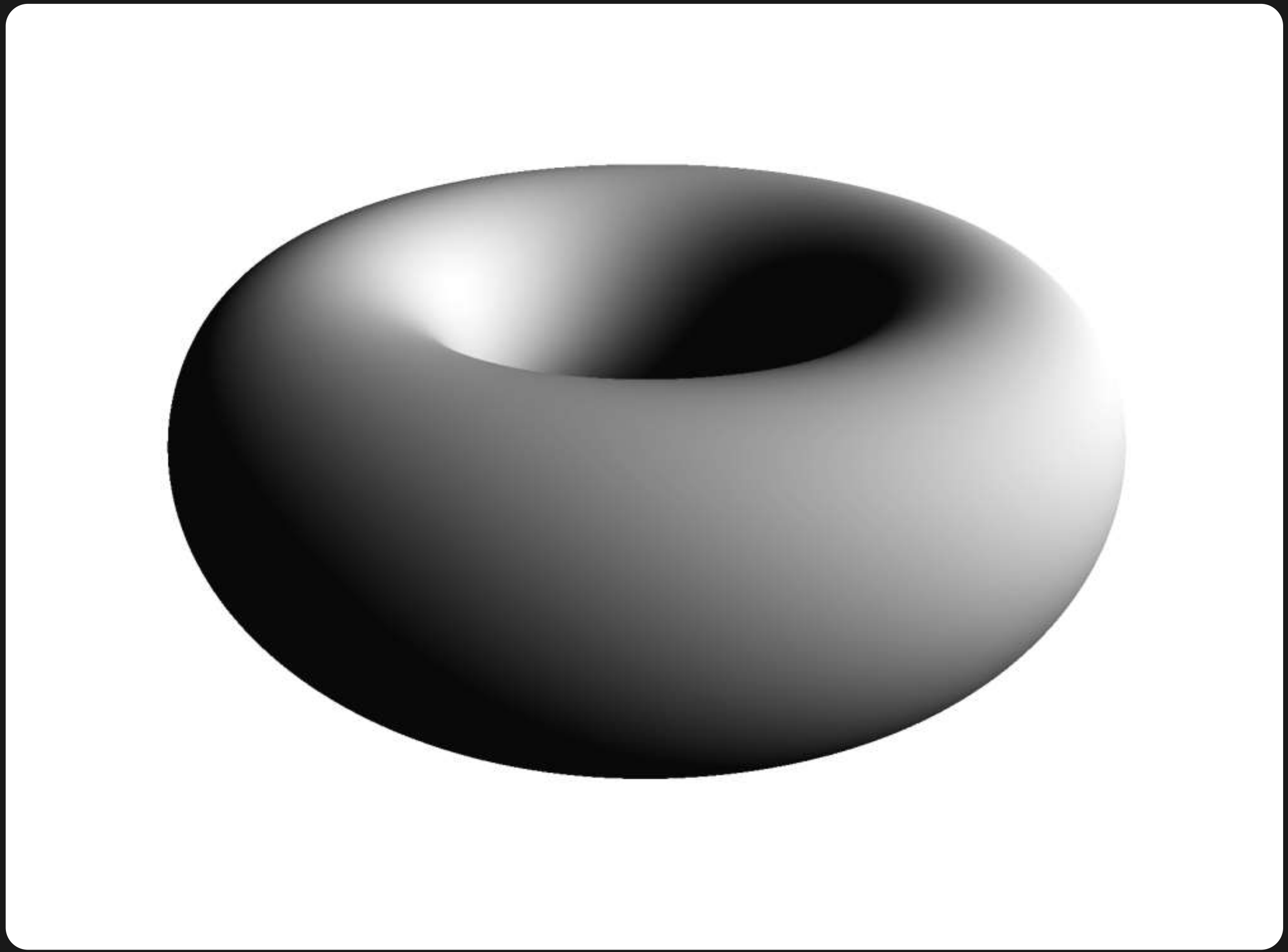
Compose With Inheritance

```
shader class Composed
    extends SimpleDiffuse



{}
```



Compose With Inheritance

```
shader class Composed
    extends SimpleDiffuse,
            CubicGregoryACC


{}
```

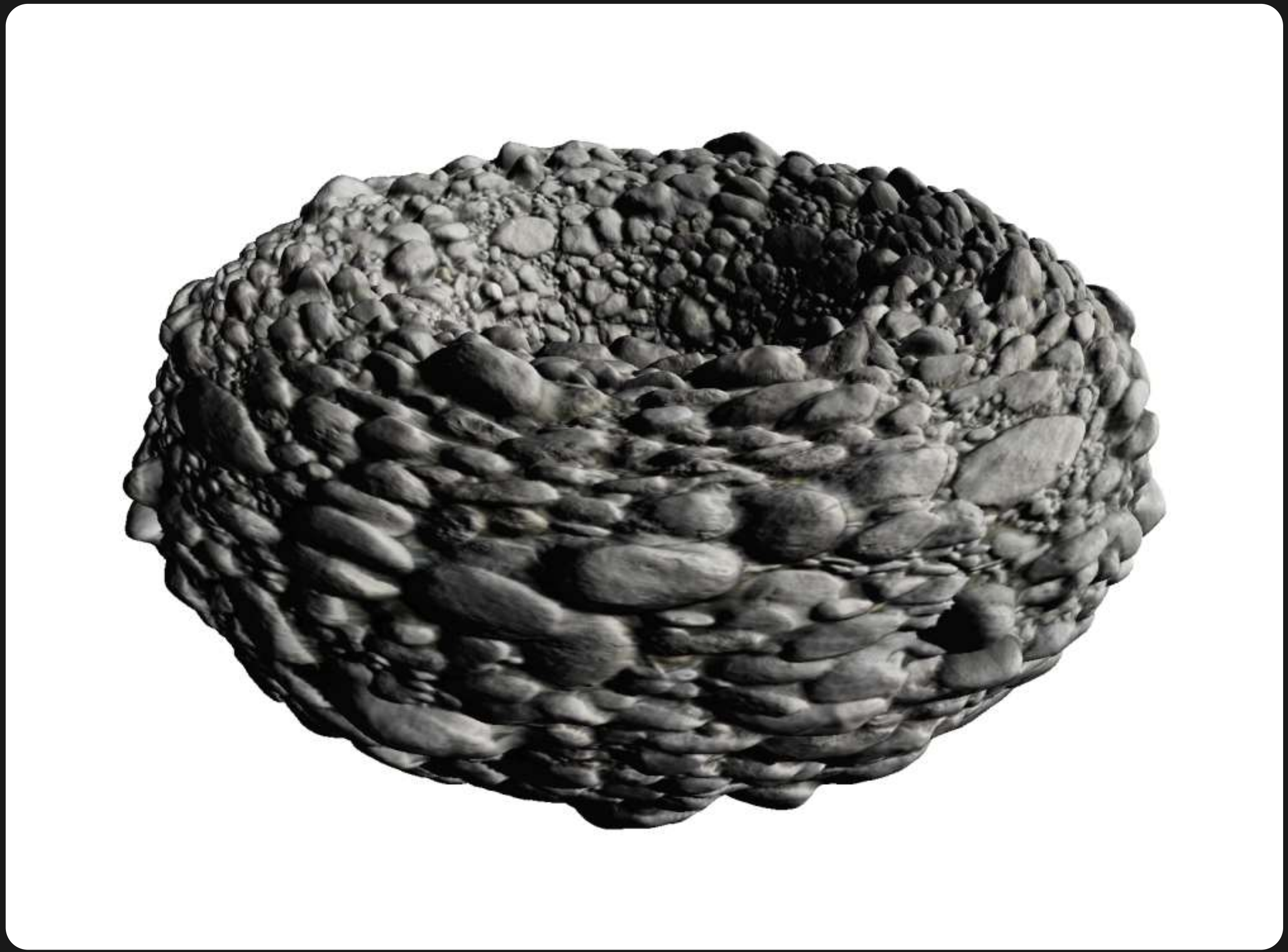


Compose With Inheritance

```
shader class Composed
    extends SimpleDiffuse,
            CubicGregoryACC,
            MyTextureMapping

{}
```

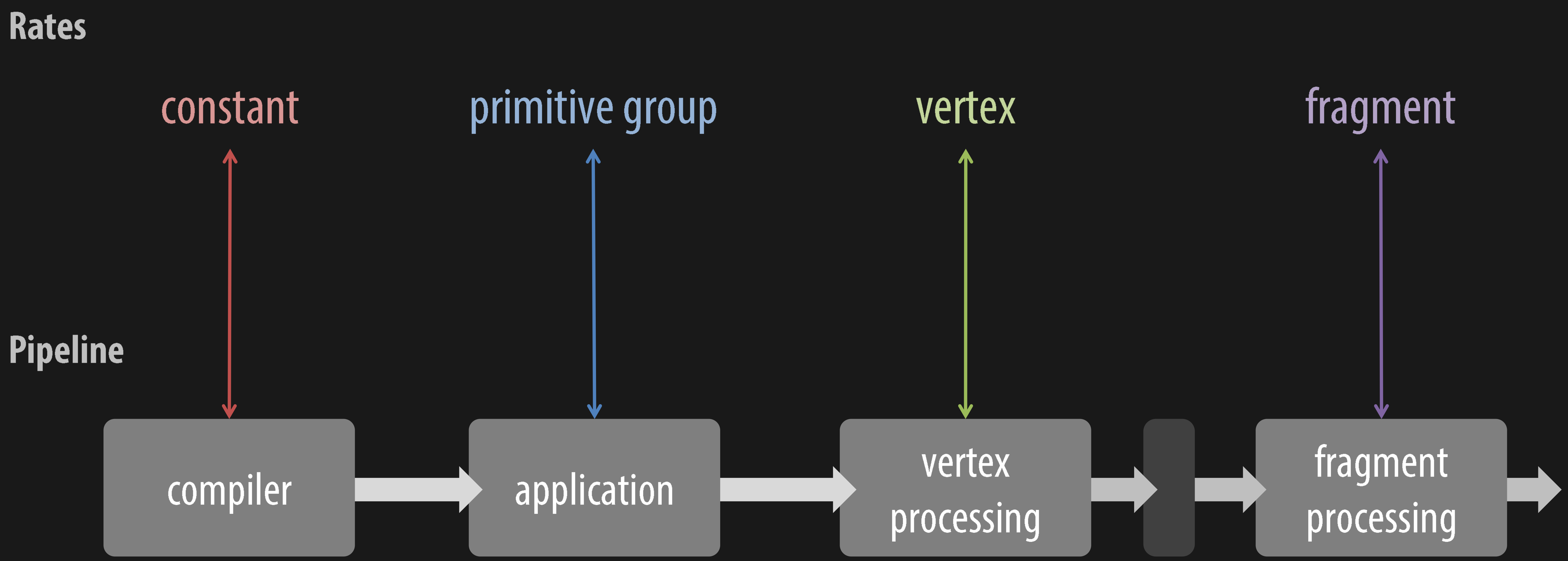
Compose With Inheritance

```
shader class Composed
    extends SimpleDiffuse,
            CubicGregoryACC,
            MyTextureMapping,
            ScalarDisplacement
{}
```

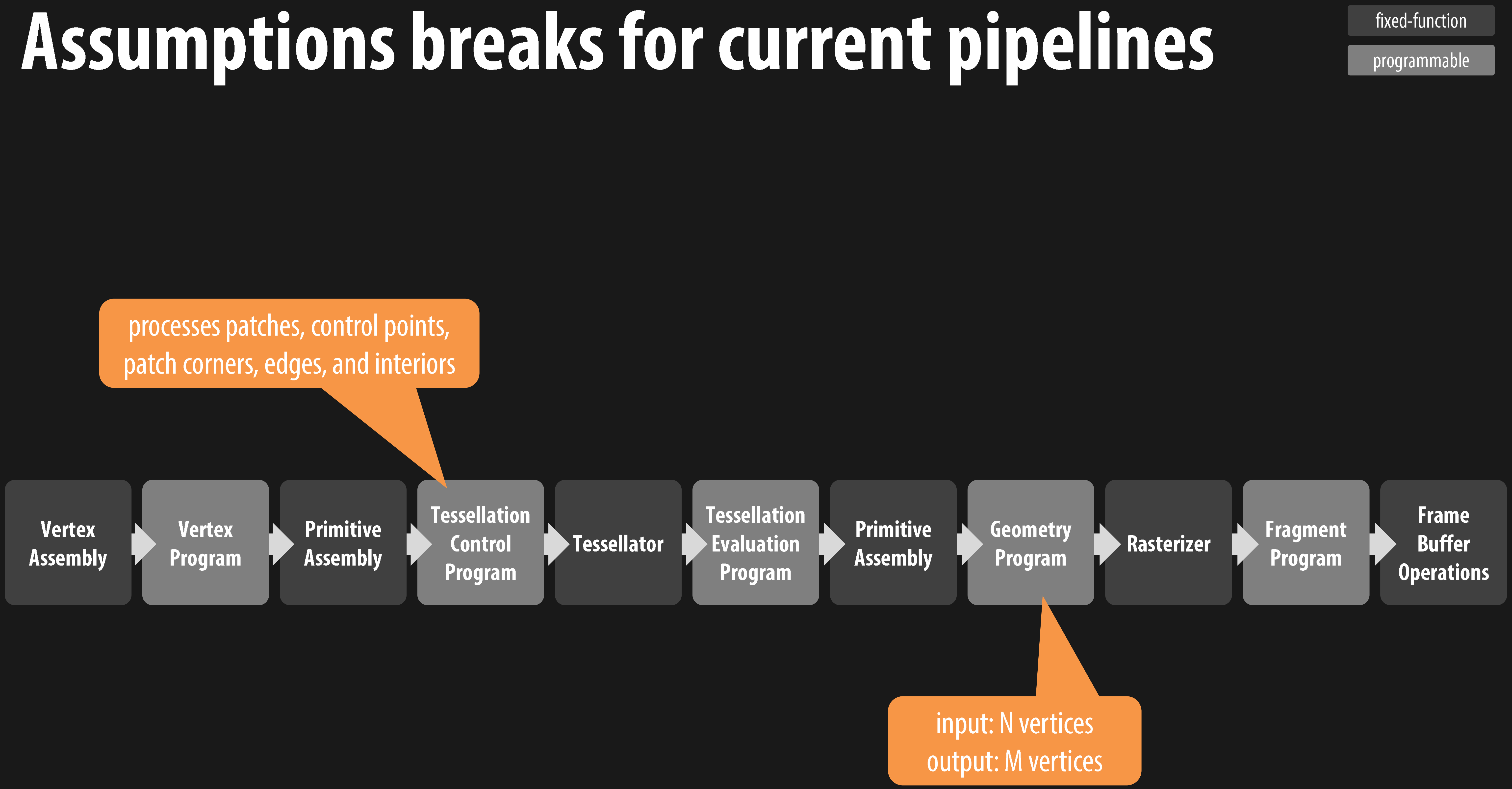


New Pipeline Programming Abstraction

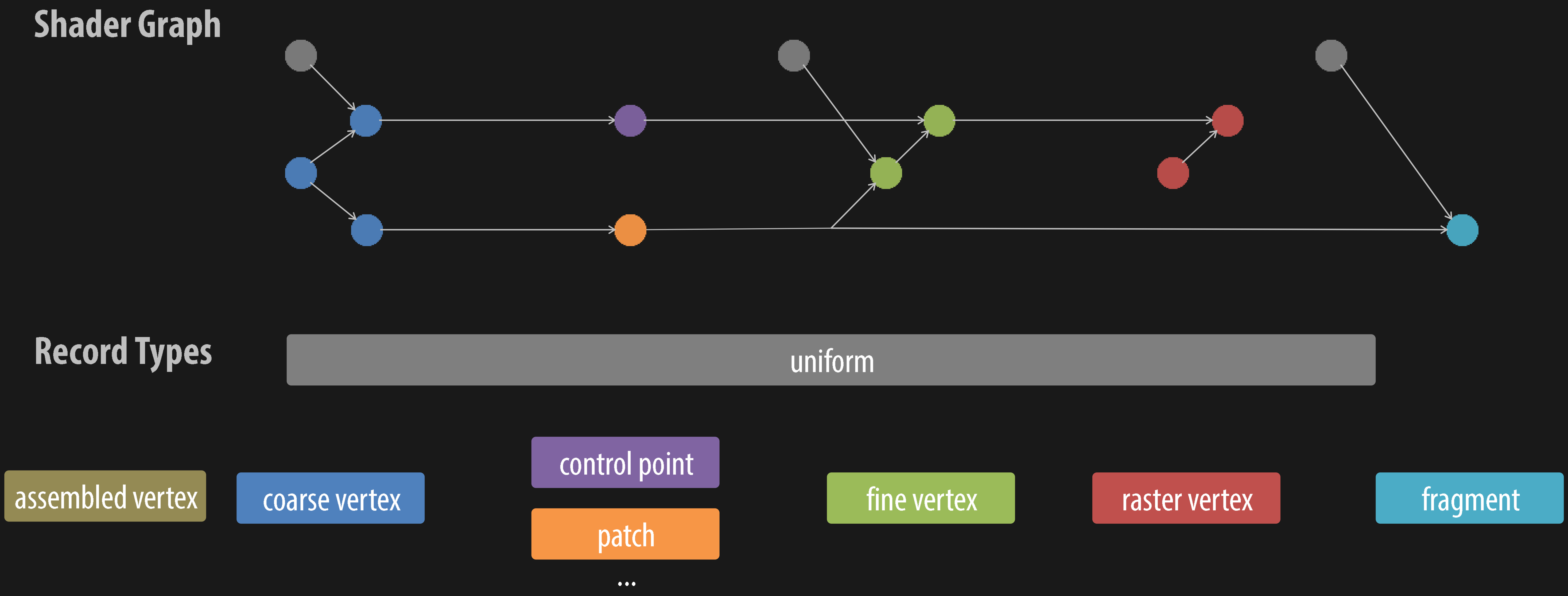
RTSL: correspondence between rates, stages



Assumptions breaks for current pipelines

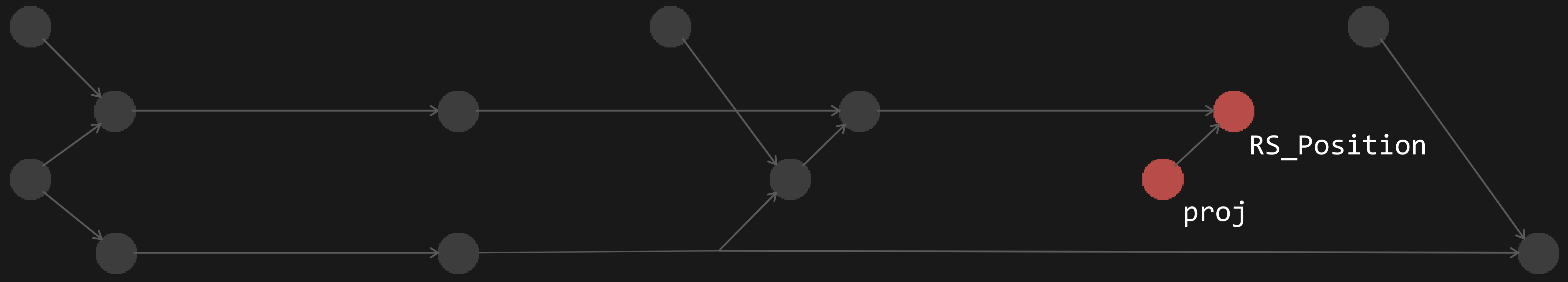


Rates Correspond to Record Types



Nodes Correspond to Attributes

Shader Graph



Record Types

uniform

assembled vertex | coarse vertex | control point | patch | ... | fine vertex | raster vertex | fragment

raster vertex

```
float3 RS_Position;
float4x4 proj;
```

Stages Create and Communicate Records

Record Types

assembled vertex

coarse vertex

control point

patch

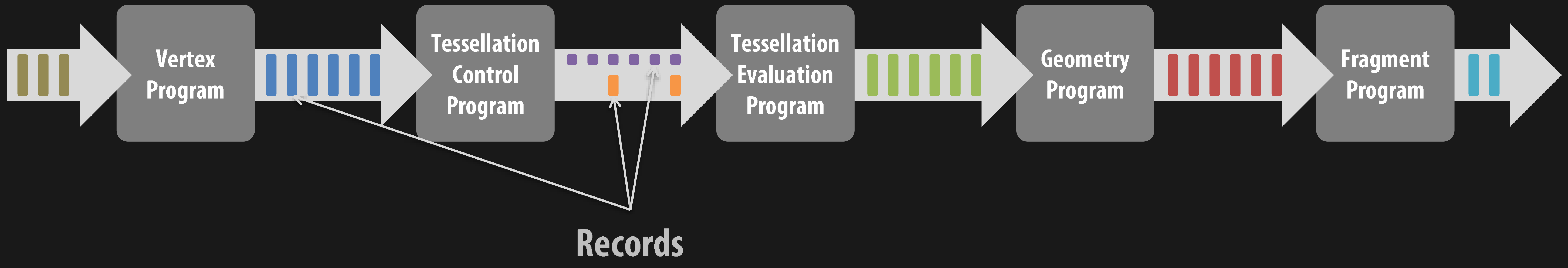
...

fine vertex

raster vertex

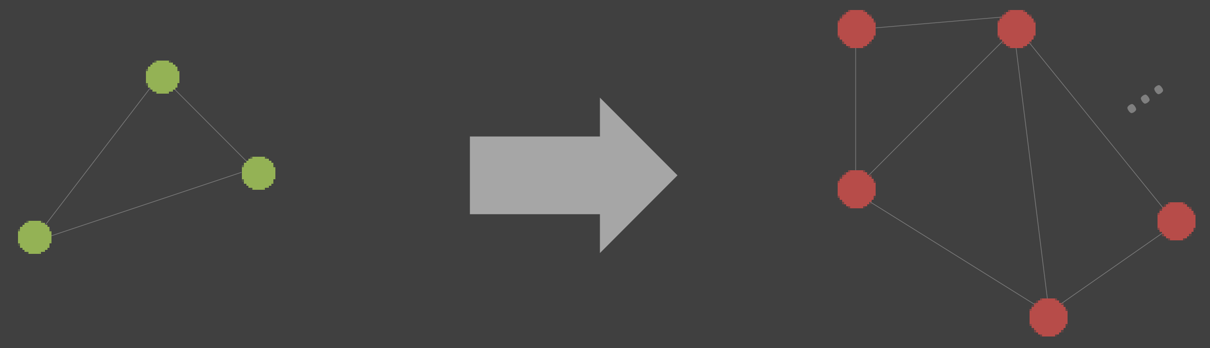
fragment

Pipeline



Groupwise Code Operates on Explicit Records

Geometry Program



`Array[FineVertex,3]` → `Stream[RasterVertex]`

Construct Records to Run Shader-Graph Code

Geometry Program

```
void RenderToCubeMap( Array[FineVertex, 3] input,
                      Stream[RasterVertex] output )
{
    for( i in Range(0,6) )
        output.Emit( new RasterVertex(proj: projMatrices(ii)) );
}
```

Construct Records to Run Shader-Graph Code

Geometry Program

```
void RenderToCubeMap( Array[FineVertex, 3] input,
                      Stream[RasterVertex] output )
{
    for( i in Range(0,6) )
        output.Emit( new RasterVertex(proj: projMatrices(ii)) );
}
```

Record Types

assembled vertex

coarse vertex

control point

patch

...

fine vertex

raster vertex

```
float3 RS_Position;
float4x4 proj;
```

fragment

Construct Records to Run Shader-Graph Code

Geometry Program

```
void RenderToCubeMap( Array[FineVertex, 3] input,
                      Stream[RasterVertex] output )
{
    for( i in Range(0,6) )
        output.Emit( new RasterVertex(proj: projMatrices(ii)) );
}
```

Record Types

assembled vertex

coarse vertex

control point

patch

...

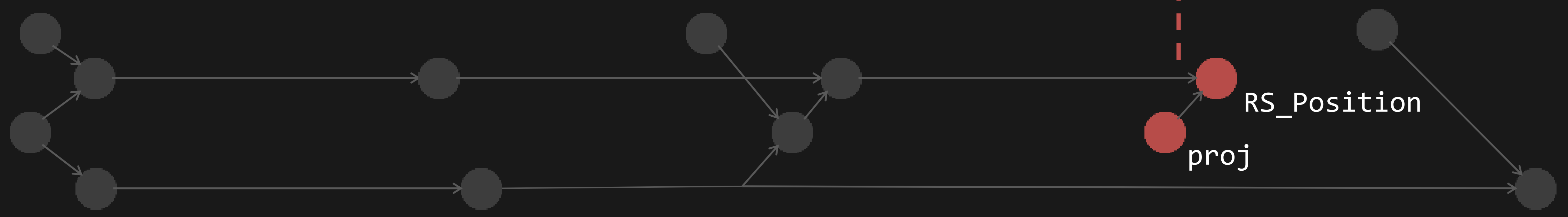
fine vertex

raster vertex

```
float3 RS_Position;
float4x4 proj;
```

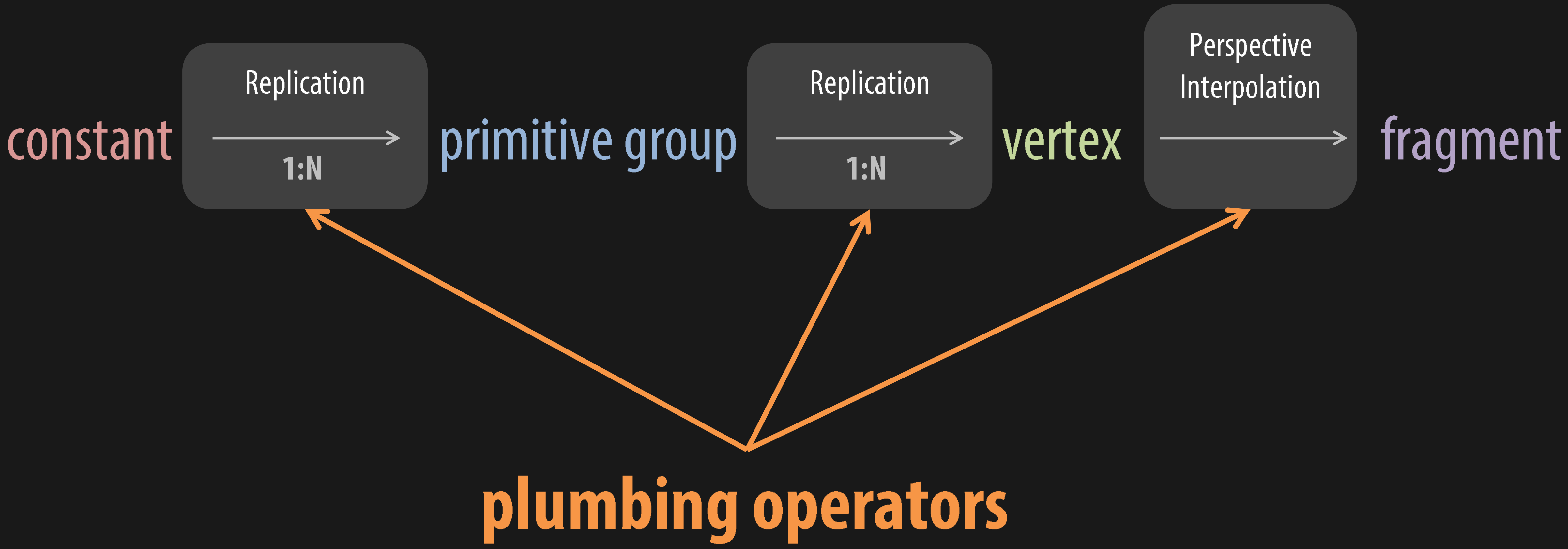
fragment

Shader Graph



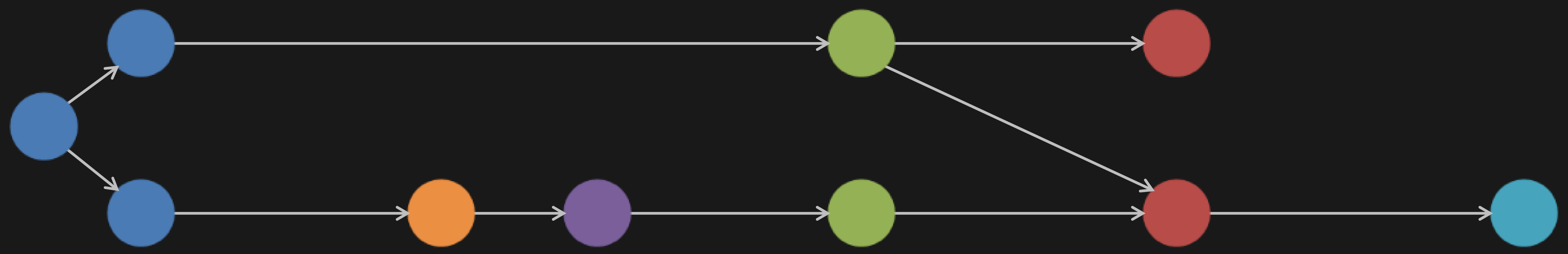
Plumbing Operators

All plumbing is implicit in RTSL



Built-In Plumbing Operators

Shader Graph



Record Types / Rates

coarse vertex

control point

patch

...

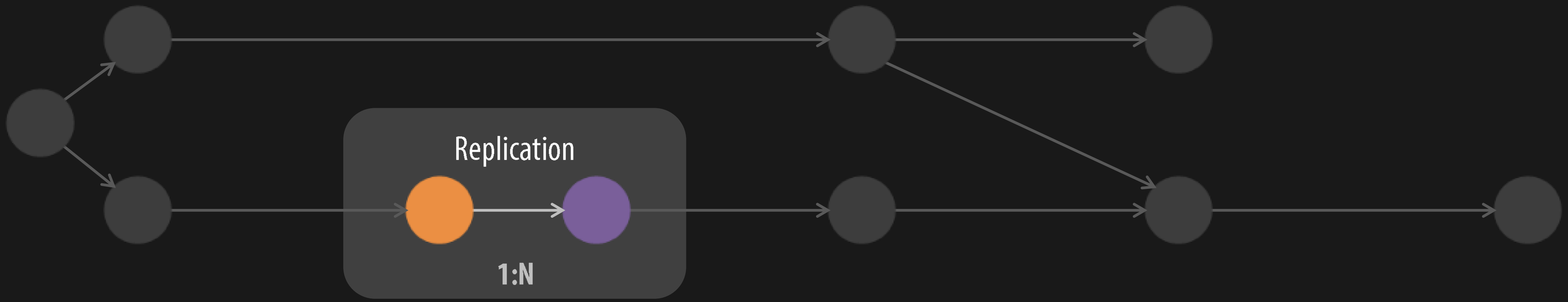
fine vertex

raster vertex

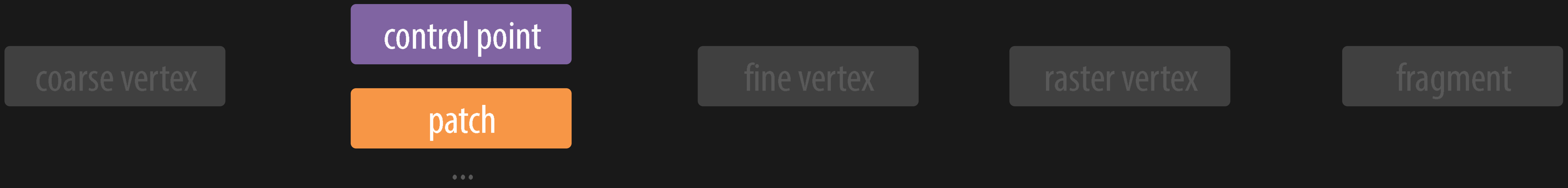
fragment

Built-In Plumbing Operators

Shader Graph

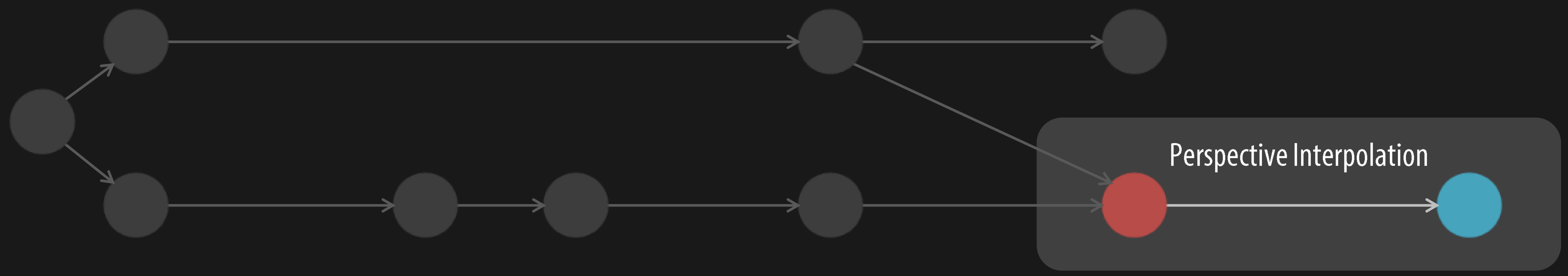


Record Types / Rates



Built-In Plumbing Operators

Shader Graph



Record Types / Rates

coarse vertex

control point

patch

...

fine vertex

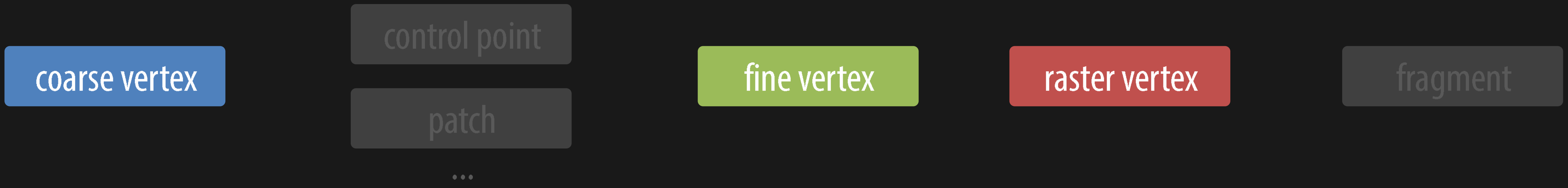
raster vertex

fragment

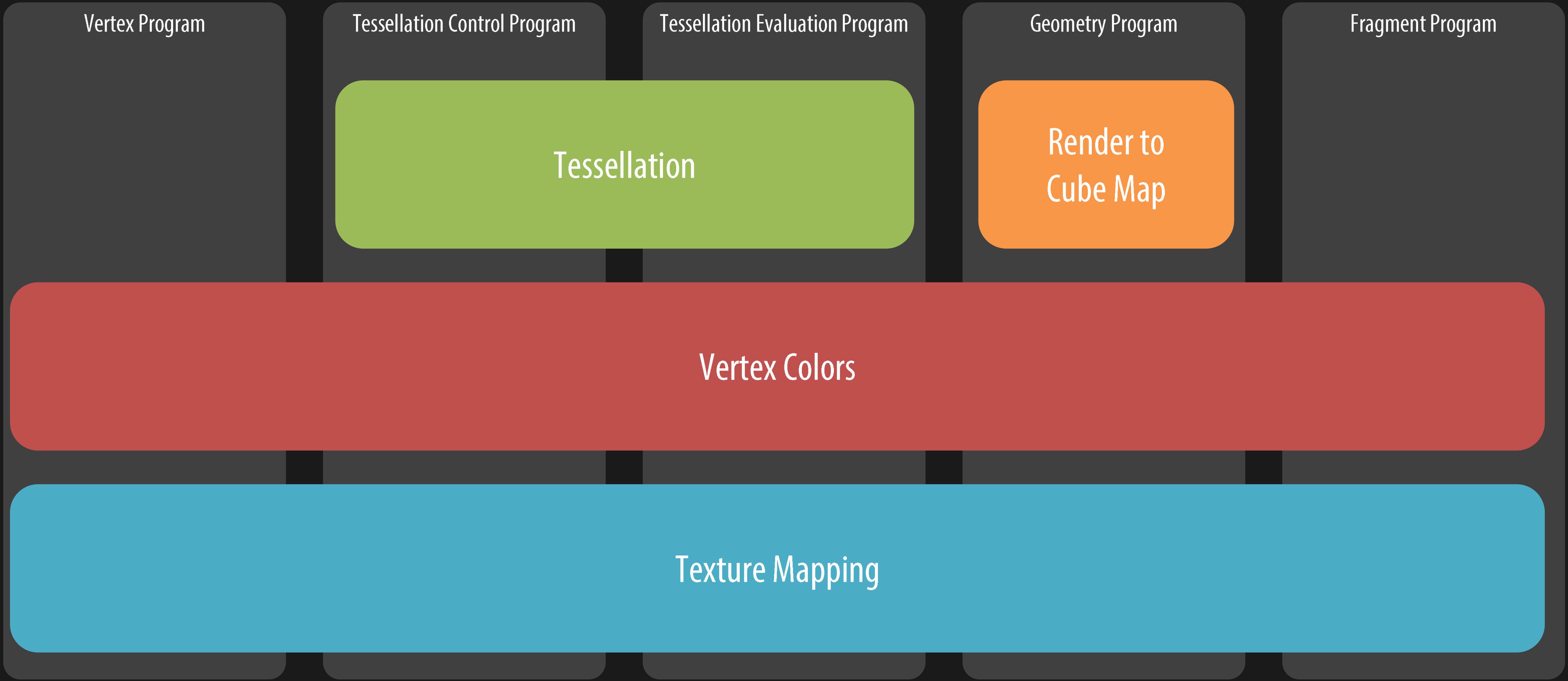
No Built-In Plumbing Operator

Shader Graph

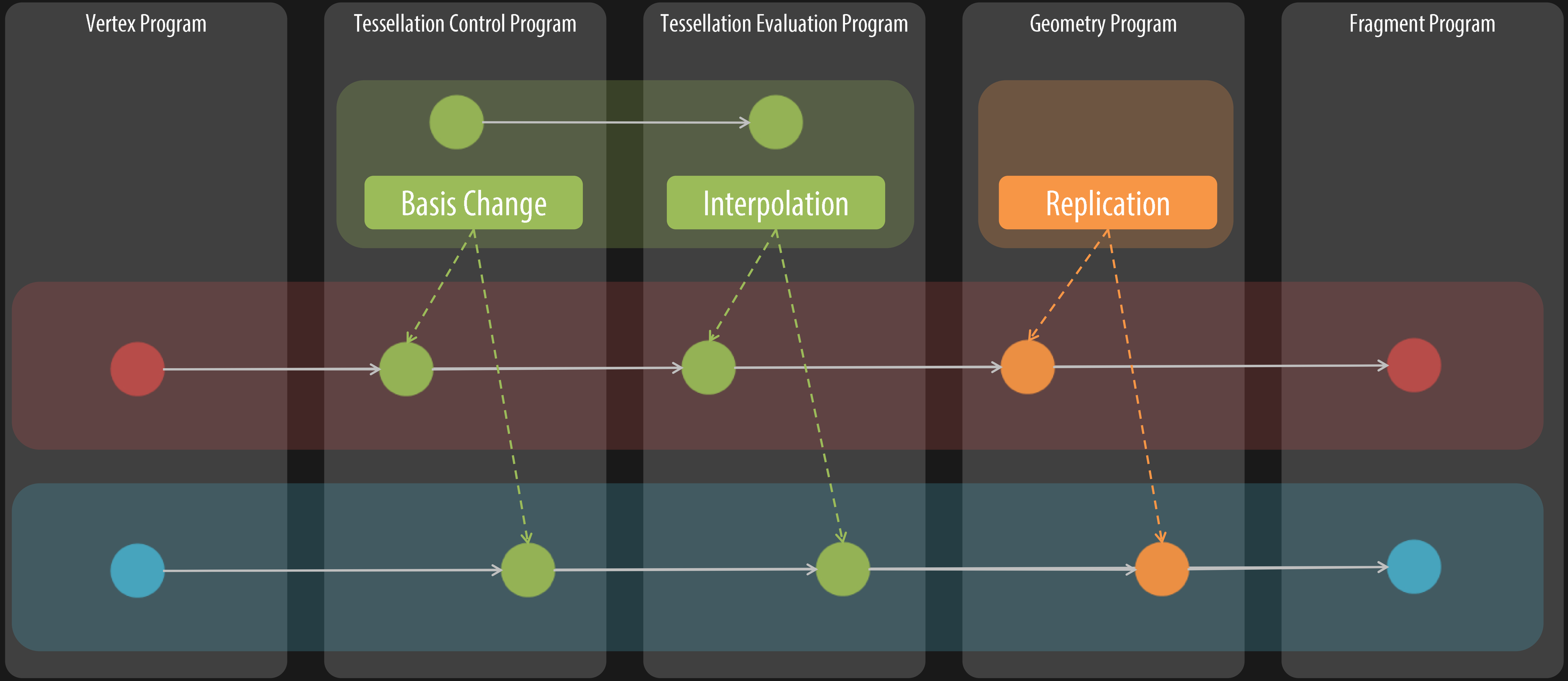
Record Types / Rates



User-Defined Plumbing Operators



User-Defined Plumbing Operators



Future Directions

Content Pipeline

Dynamic Composition

New and Flexible Pipelines

Content Pipeline

Content Pipeline

- Mediate interface between content and renderer
  - Current graphics programming models often ignore
- Extend rendering pipeline with earlier “rates”
  - Computations that run before applications ships
- Optimize assets to code, and vice versa
  - Some prior work in EAGL [Lalonde and Schenk 2002]

Dynamic Composition

Spark Does Not Address Dynamic Composition

- Each combination of features statically compiled
- Combinatorial explosion in some apps
- Worse load times, memory usage, etc.

Why is dynamic composition hard?

- "Just use function pointers"
  - Jumping to a computed address is not the issue

- Resource-based scheduling
  - Use more registers, get fewer threads, less latency hiding

Dynamic composition needs either…

- Dynamic computation of resource usage
  - Later, but still before launching threads

- Escew resource-based scheduling
  - Give every thread the same number of registers?

New and Flexible Pipelines

Alternative Rendering Pipelines in Research

- Decoupled shading

- Stochastic rasterization

- Ray traicng

Reactive Scheduling

- Don’t know how much work you will generate
  - Hits on a post-visibility decoupled shading cache
- Don’t get batches of coherent work “for free”
  - Rays might all hit the same surface, or many different ones
- Very different from current raster pipeline
  - Scheduling is almost entirely proactive

Questions

and/or

Discussion