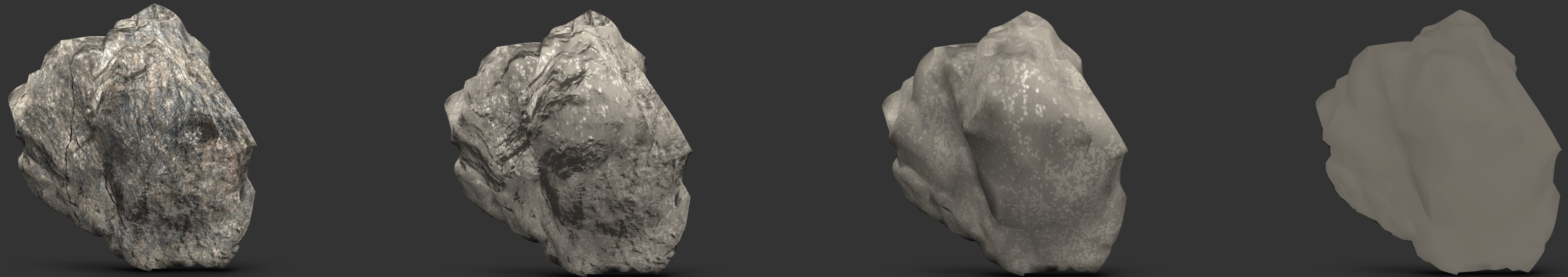


Tackling the level-of-detail problem through new shading languages and tools



Kayvon Fatahalian
Carnegie Mellon University

with Yong He (CMU), Tim Foley (NVIDIA Research), Natalya Tatarchuk (Bungie)

**Claim: advanced code generation tools can and should
take on a much larger role in enabling rapid**

(or even automatic)

exploration of key renderer optimization decisions

(even decisions that have global impact on the structure of the renderer)

Low-level representation (per pipeline stage code)

What you get from the system:

Low-level instruction scheduling

Register allocation

Standard compiler optimizations

(constant propagation, redundancy removal,
dead code elimination, etc.)

But no attempt at hard “graphics” optimizations:

Cross-pipeline-stage optimizations

CPU draw command optimization

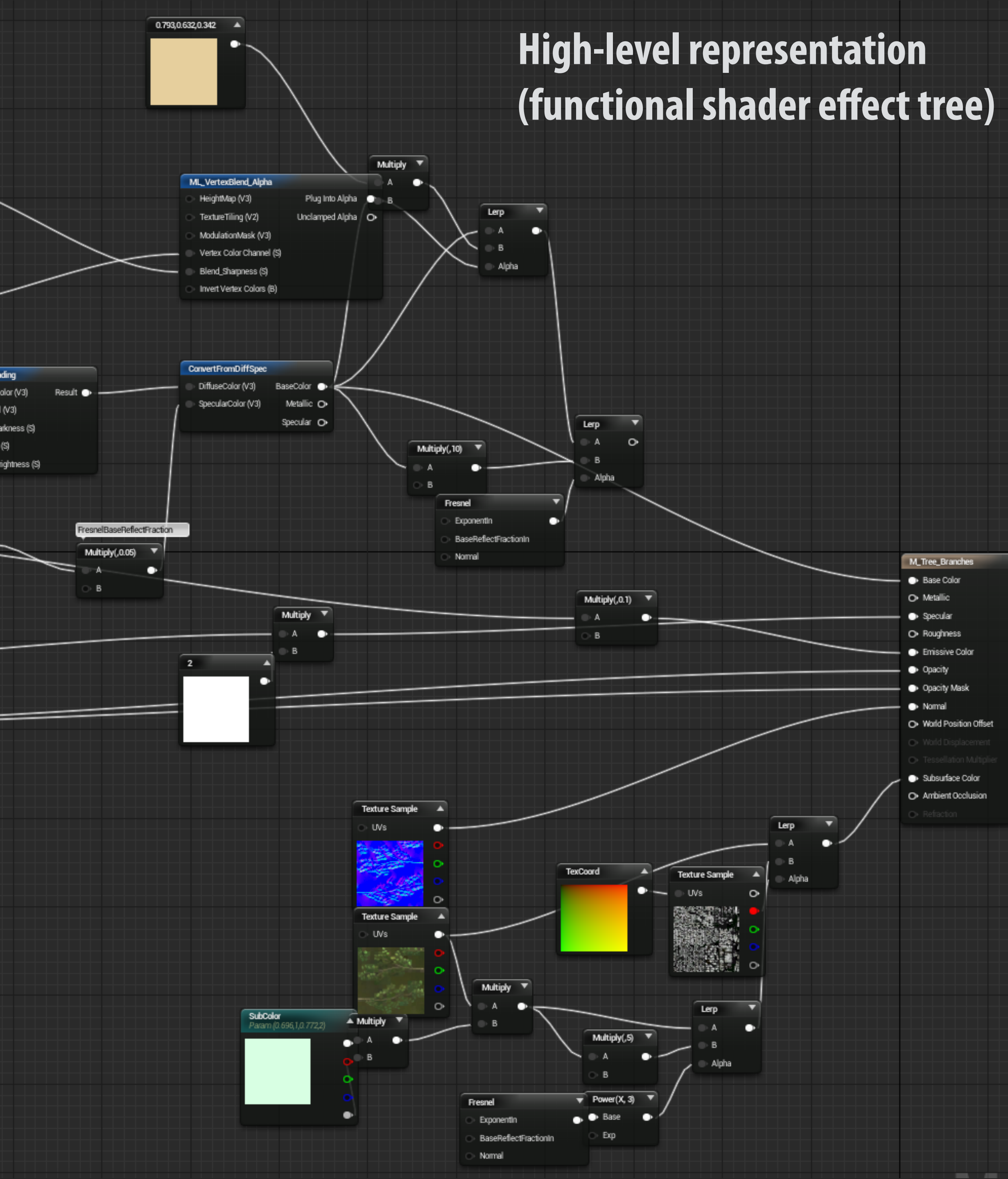
Choice of graphics technique (algorithm)

```
struct Vertex_In {
    float3 vertex, normal;
    float3 view, light_dir;
    float2 uv;
}
struct Coarse_Out {
    float diffuse, specular;
    bool diffuse_flag : SV_REFINE_FLAG_0;
    bool specular_flag : SV_REFINE_FLAG_1;
}
Coarse_Out coarse_shader(Vertex_In in) {
    Coarse_Out rs;
    rs.diffuse_flag = rs.specular_flag = false;
    float nDotL = dot(in.normal, in.light_dir);
    if (fwidth(nDotL) > DIFFUSE_THRESHOLD)
        rs.diffuse_flag = true;
    else
        rs.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
    if (dot(fwidth(N), fwidth(N)) < SPECULAR_THRESHOLD)
        rs.specular = spec_lighting(in, rs.specular_flag);
    else
        rs.specular_flag = true;
    return rs;
}

float4 fine_shader(Vertex_In vin, Coarse_Out cin) {
    if (cin.diffuse_flag) {
        float nDotL = dot(vin.normal, vin.light_dir);
        cin.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
    }
    if (cin.specular_flag) {
        bool tmp_flag;
        // unused flag, but needed by call below
        cin.specular = spec_lighting(vin, tmp_flag);
    }
    float lighting = cin.diffuse + cin.specular;
    float4 albedo = texture(texAlbedo, uv);
    return albedo * lighting;
}

float4 spec_lighting(Vertex_In in, out int flag) {
    // compute specular lighting
    // optionally set flag if refinement is needed.
    ...
    return specular;
}
```


High-level representation (functional shader effect tree)



Low-level representation (per pipeline stage code)

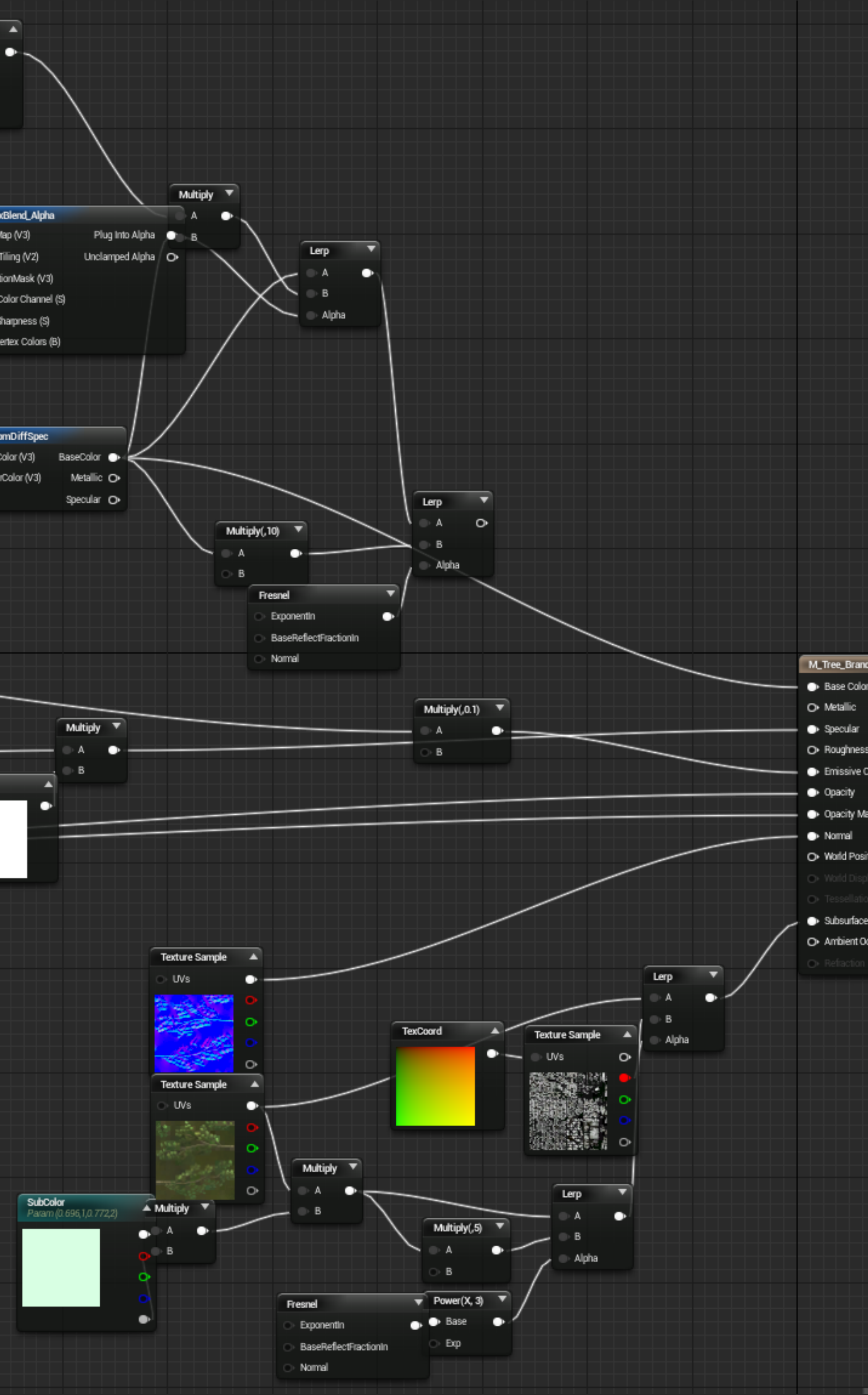
```
struct Vertex_In {
    float3 vertex, normal;
    float3 view, light_dir;
    float2 uv;
}

struct Coarse_Out {
    float diffuse, specular;
    bool diffuse_flag : SV_REFINE_FLAG_0;
    bool specular_flag : SV_REFINE_FLAG_1;
}

Coarse_Out coarse_shader(Vertex_In in) {
    Coarse_Out rs;
    rs.diffuse_flag = rs.specular_flag = false;
    float nDotL = dot(in.normal, in.light_dir);
    if (fwidth(nDotL) > DIFFUSE_THRESHOLD)
        rs.diffuse_flag = true;
    else
        rs.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
    if (dot(fwidth(N), fwidth(N)) < SPECULAR_THRESHOLD)
        rs.specular = spec_lighting(in, rs.specular_flag);
    else
        rs.specular_flag = true;
    return rs;
}
```

```
float4 fine_shader(Vertex_In vin, Coarse_Out cin) {
    if (cin.diffuse_flag) {
        float nDotL = dot(vin.normal, vin.light_dir);
        cin.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
    }
    if (cin.specular_flag) {
        bool tmp_flag;
        // unused flag, but needed by call below
        cin.specular = spec_lighting(vin, tmp_flag);
    }
    float lighting = cin.diffuse + cin.specular;
    float4 albedo = texture(texAlbedo, uv);
    return albedo * lighting;
}
```

```
float4 spec_lighting(Vertex_In in, out int flag) {
    // compute specular lighting
    // optionally set flag if refinement is needed.
    ...
    return specular;
}
```

```
// set of defines will pull in different code
// from header...
#define DEPTH_ONLY_PASS 0
#define ENABLE_NORMAL_MAPPING 1
#define ENABLE_FANCY_SPECULAR 1
#define ENABLE_XBOX_ONE_HACK 1
```

...

```
#include all_my_shaders.h
```

[shader program here...]

```
struct Vertex_In {
    float3 vertex, normal;
    float3 view, light_dir;
    float2 uv;
}

struct Coarse_Out {
    float diffuse, specular;
    bool diffuse_flag : SV_REFINE_FLAG_0;
    bool specular_flag : SV_REFINE_FLAG_1;
}

Coarse_Out coarse_shader(Vertex_In in) {
    Coarse_Out rs;
    rs.diffuse_flag = rs.specular_flag = false;
    float nDotL = dot(in.normal, in.light_dir);
    if (fwidth(nDotL) > DIFFUSE_THRESHOLD)
        rs.diffuse_flag = true;
    else
        rs.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
    if (dot(fwidth(N), fwidth(N)) < SPECULAR_THRESHOLD)
        rs.specular = spec_lighting(in, rs.specular_flag);
    else
        rs.specular_flag = true;
    return rs;
}

float4 fine_shader(Vertex_In vin, Coarse_Out cin) {
    if (cin.diffuse_flag) {
        float nDotL = dot(vin.normal, vin.light_dir);
        cin.diffuse = clamp(nDotL, 0.0, 1.0) * Kd + Ka;
    }
    if (cin.specular_flag) {
        bool tmp_flag;
        // unused flag, but needed by call below
        cin.specular = spec_lighting(vin, tmp_flag);
    }
    float lighting = cin.diffuse + cin.specular;
    float4 albedo = texture(texAlbedo, uv);
    return albedo * lighting;
}

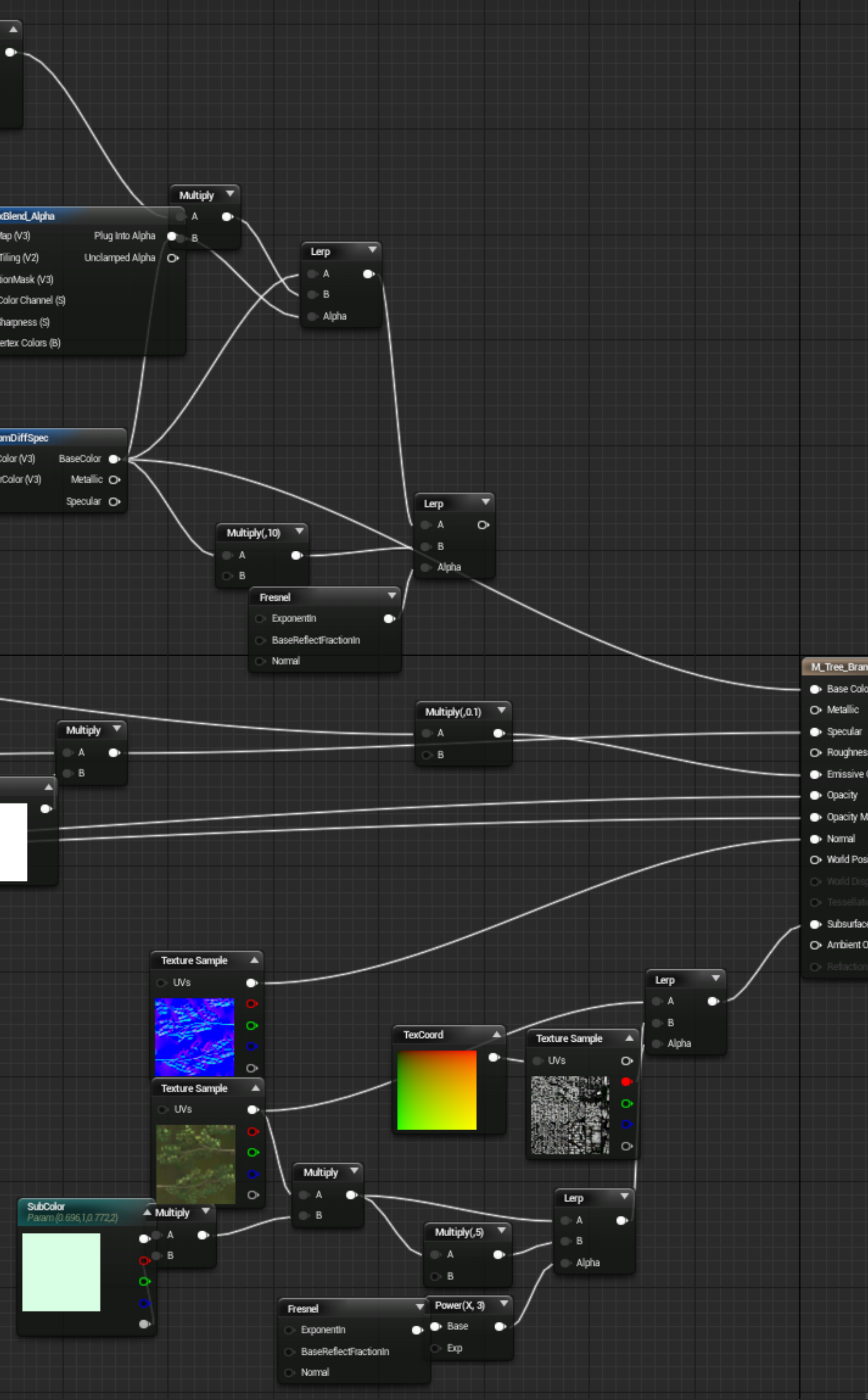
float4 spec_lighting(Vertex_In in, out int flag) {
    // compute specular lighting
    // optionally set flag if refinement is needed.
    ...
    return specular;
}
```



SPIR-V is Kronos' proposed low-level "intermediate representation" (IR) for shader and compute programs.

SPIR-V will simplify the mechanics of implementing an advanced compiler for new high-level representations.

But it does not provide help with performing advanced graphics optimizations, since it is a low-level machine IR.



Two open problems

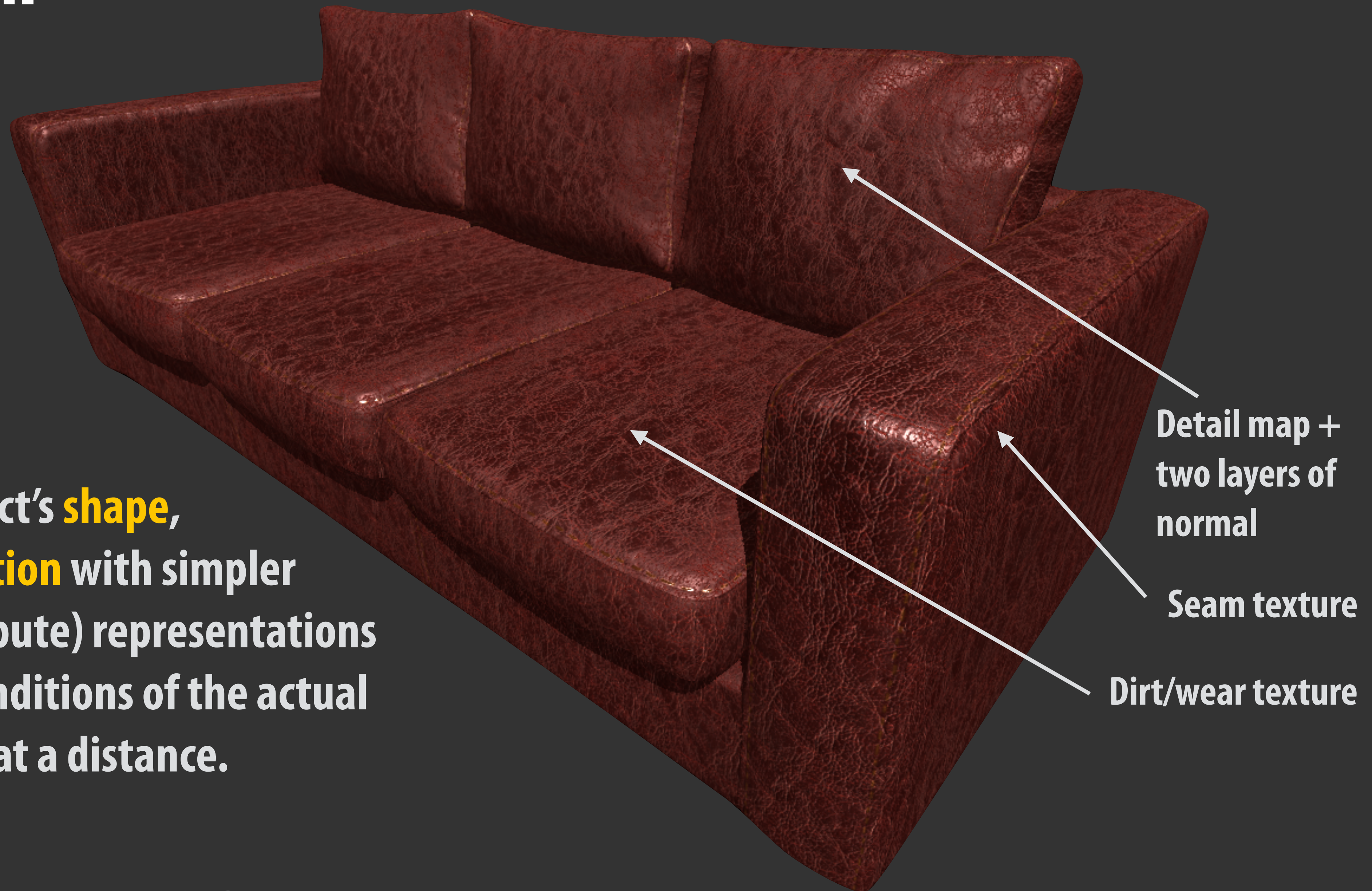
Is there a better representation for real-time rendering computations that will enable a compiler to perform sophisticated graphics optimization decisions such as...

Synthesizing and managing levels of detail.

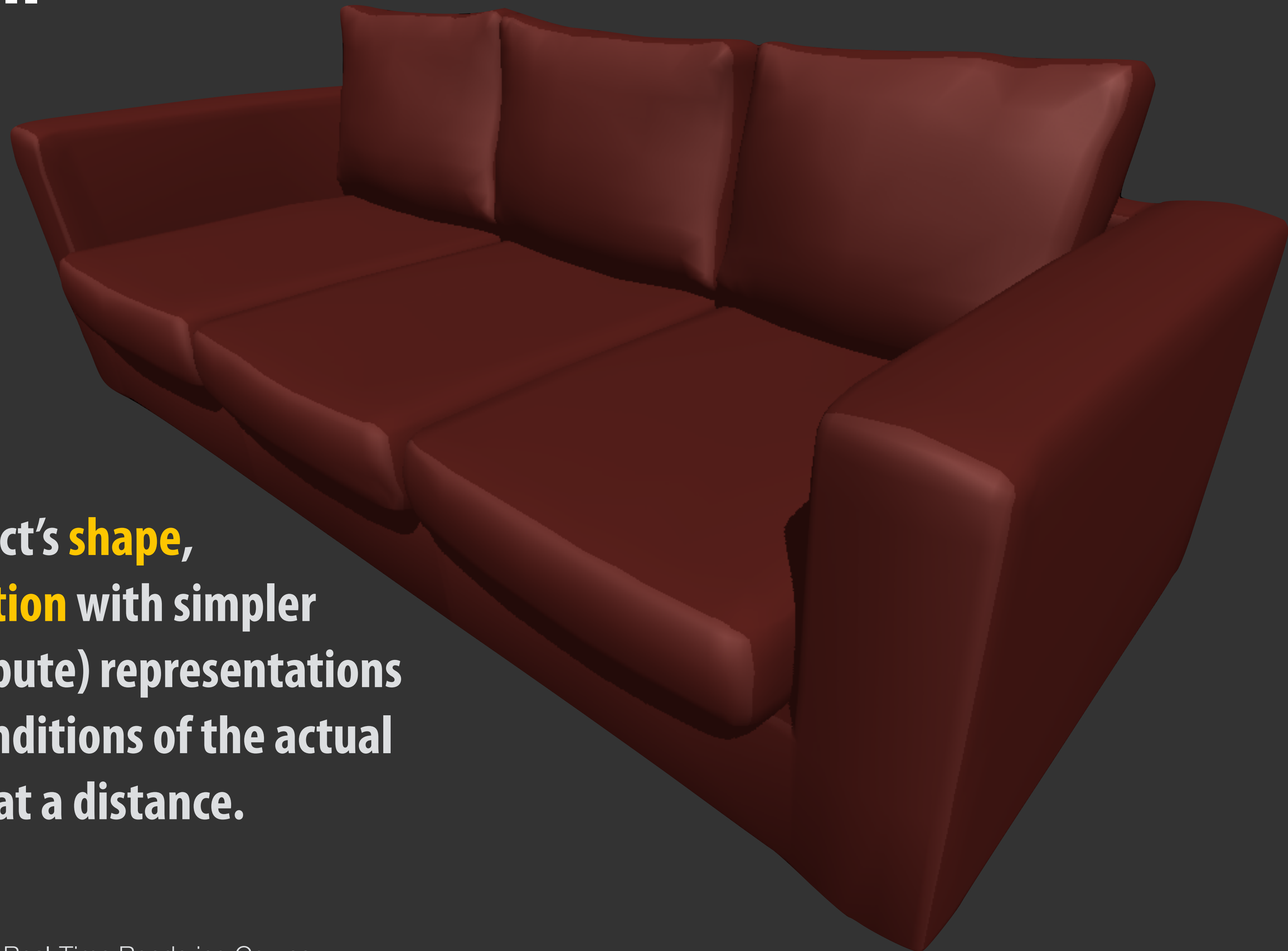
(today's topic)

Level of detail

Approximate an object's **shape**, **appearance**, and **motion** with simpler (and cheaper to compute) representations that are plausible renditions of the actual object when viewed at a distance.



Level of detail



Approximate an object's **shape**, **appearance**, and **motion** with simpler (and cheaper to compute) representations that are plausible renditions of the actual object when viewed at a distance.

Level of detail

Original



Simplified



Approximate an object's **shape**, **appearance**, and **motion** with simpler (and cheaper to compute) representations that are plausible depictions of the actual object when viewed at a distance.

Level of detail is critical for...

- **Reducing GPU load**

Reduce vertex count

Utilize simpler GPU pipeline shaders

- **Reducing CPU load**

Reduce the number of state changes

Reduce the number of scene draw commands

- **Improving image quality**

Simplification process involves pre-filtering that reduces aliasing caused by undersampling detailed objects at a distance.

No LOD (Baseline)

8.92 ms



LOD (w/ Transitions)

7.95 ms



Level of detail is critical for...

- **Reducing GPU load**

Reduce vertex count

Utilize simpler GPU pipeline shaders

- **Reducing CPU load**

Reduce the number of state changes

Reduce the number of scene draw commands

- **Improving image quality**

Simplification process involves pre-filtering that reduces aliasing caused by undersampling detailed objects at a distance.

In all cases, must avoid discontinuous transitions when switching between detail levels.

Level of detail applies to many aspects of rendering an object

GEOMETRY

Decimate: use simplified mesh with reduced triangle count

Amplify: generate detail via tessellation

ANIMATION

Reduce number of bones at a distance; reduce number of influences

Simplify/eliminate secondary motions

APPEARANCE

MATERIAL

Manually author simplified shaders

LIGHTING

Not aware of significant (shipping) solutions in games



Destiny (2014)

Credit: Bungie

Level of detail in Destiny

GEOMETRY:

3-4 levels of mesh detail for characters

(100%, 40%, 15%, 8% of base vertices)

2-3 levels for “environmental” objects

(100%, 40%, 15% of base vertices)

Skinning LOD: engine targets ~4M skinned vertices per frame across all draw calls / render passes
(more objects in scene → bias towards lower levels of detail to meet target.)

Rough estimate: for any given frame, about 50% of on-screen objects are in a simplified detail state (LOD yields 2.5x increase in number of possible instances in the scene)

ANIMATION:

Reduce bone count at lower levels of detail (e.g., 75 to 40 bones)

Engine stops animating “less important” bones at a distance

Reduce per-vertex influences from 4 to (1 or 2)

Level of detail in Destiny

MATERIAL APPEARANCE:

Characters:

One level of simplified shaders

(drops normal maps to reduce vertex processing cost: no need for processing tangents)

Environment objects:

Simplified shader with “flattened” albedo and specular maps, no normal maps

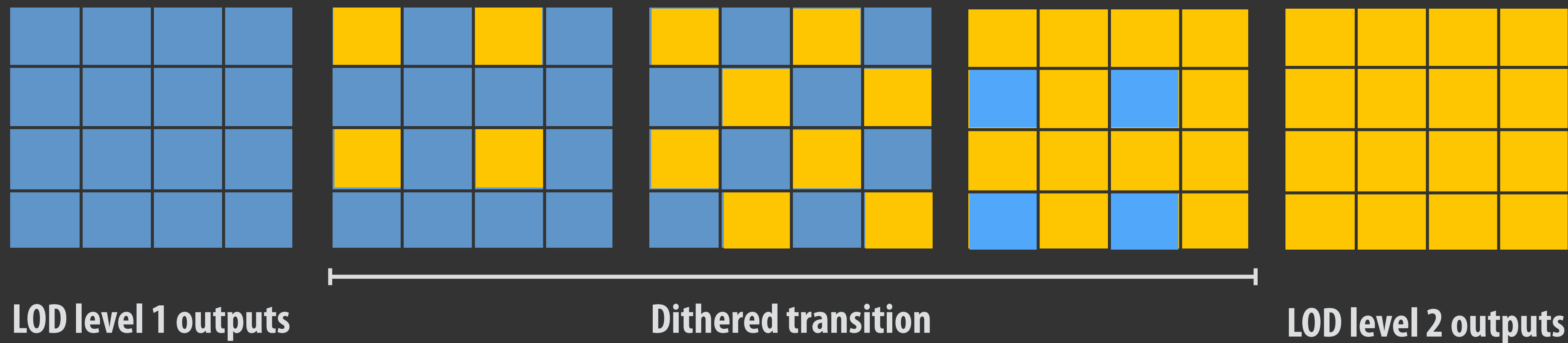
Same shader used for models with $\leq 15\%$ vertices

Implication — many distant objects can be rendered without changing shader state since they use the same simplified shader (reduces CPU work)



Smooth transitions between levels of detail

- Status quo: stencil buffer based dithering between outputs of two levels of detail
- Requires rendering object at two levels of detail (and throwing out pixels of work)
- General solution for all forms of LOD: geometry, animation, appearance







**Dither transition
between LODs**

Credit: Bungie



**Dither transition
between LODs**

Credit: Bungie



**Dither transition
between LODs**

Credit: Bungie



**Dither transition
between LODs**

Credit: Bungie



**Dither transition
between LODs**

Credit: Bungie



**Dither transition
between LODs**

Credit: Bungie



**Dither transition
between LODs**

Credit: Bungie



**Same dithering
technique is used to
fade out objects at a
distance.**

Credit: Bungie



**Same dithering
technique is used to
fade out objects at a
distance.**

Credit: Bungie



**Same dithering
technique is used to
fade out objects at a
distance.**

Credit: Bungie



**Same dithering
technique is used to
fade out objects at a
distance.**

Credit: Bungie

Significant diversity of shaders in Destiny

17,800 artist-authored shader graphs ← many unique materials!
(although most are tweaks to standard shader templates)

Shader graphs compiled to: (on Xbox One)

90,756 unique vertex shaders

92,590 unique fragment shaders

Recap: [a few of] the tasks required for level-of-detail

- Geometry and animation simplification (impacts asset generation)
- Shader simplification (decisions cross-cutting multiple GPU stages)
- Shader interface changes (modifies uniform parameters, vtx-2-fragment, etc.)
- Draw command merging (CPU logic)
- Smooth transition management (CPU and GPU logic)

Note how level-of-detail functionality bleeds through entire rendering system

Offline asset generation

CPU-side processing

GPU processing

A challenge!

Can we automatically generate appearance shader levels of detail from a specification of the full detailed shader?

- **Generate simplified vertex and fragment shaders**
- **Determine distance ranges at which it is acceptable to use them**
- **Solve the above problems in a small amount of time, so as not to disrupt artist workflows (recall 17K unique shader graphs)**

Example outputs from our system

Simplification time: 114 sec



Baseline Shader

Albedo Map
Normal Map
Detail Normal Map
Wear
Seam
Roughness Map

Simplification time: 114 sec



LOD 1

Albedo Map
Normal Map
Detail Normal Map
Seam
Roughness Map

Wear removed

Simplification time: 114 sec



LOD 2

Albedo Map
Normal Map
Seam
Roughness Map

Detail Normal removed

Simplification time: 114 sec



LOD 3

Albedo Map
Roughness Map

Seam removed

Simplification time: 114 sec



LOD 4

Roughness Map

Albedo Map removed

Simplification time: 114 sec



LOD 5

Constant Material

Simplification time: 95 sec

Baseline Shader

Detail Normal Map
Normal Map
Reflection
Fresnel Blend



LOD 1

Normal Map
Reflection
Fresnel Blend



Simplification time: 95 sec



LOD 2

Normal Map
Fresnel Blend

Reflection removed

Simplification time: 95 sec

LOD 3

Simplified Fresnel Blend

Normal Map removed



Simplification time: 95 sec

LOD 4
Constant Color

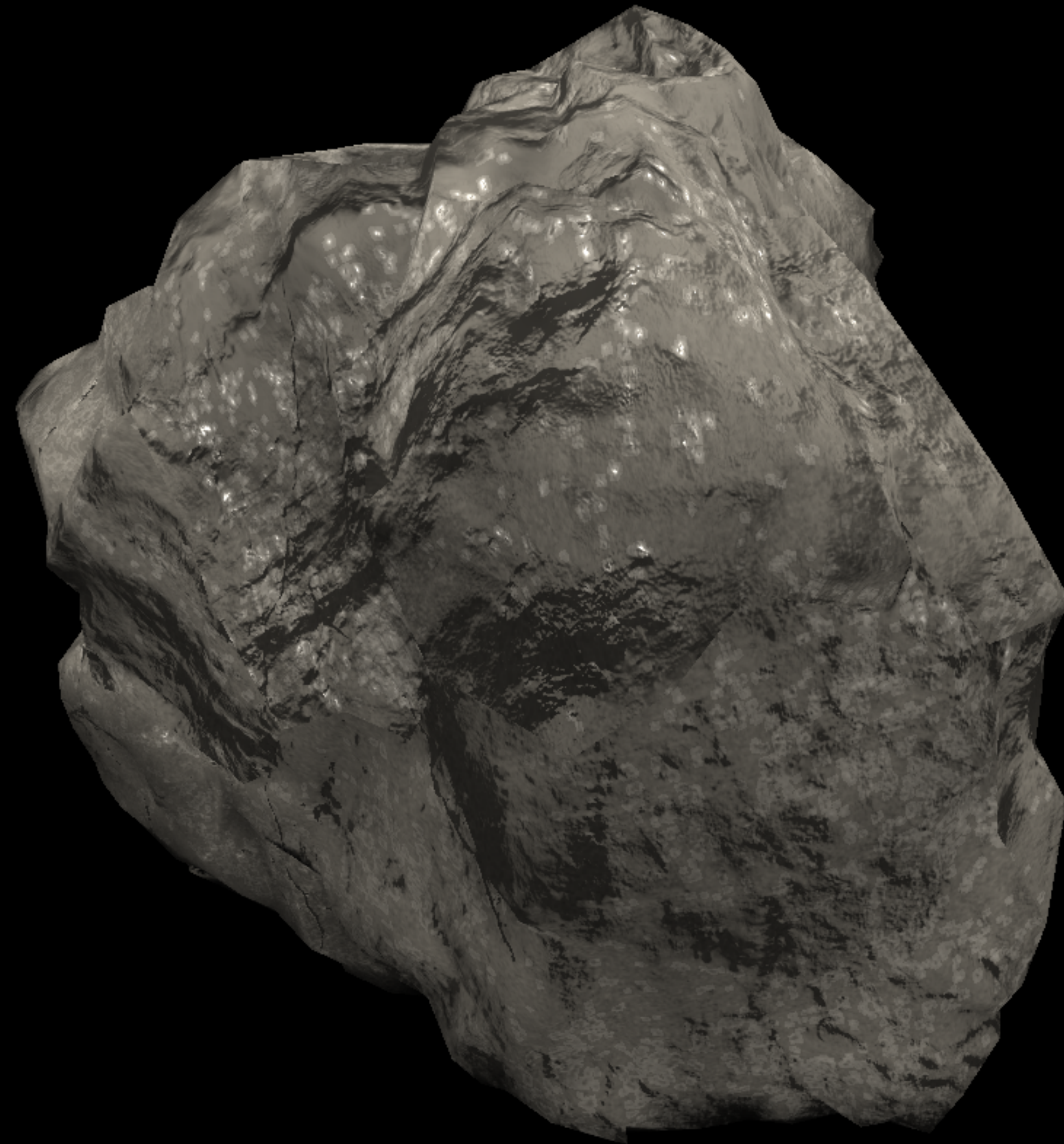


LOD 0 (original shader)



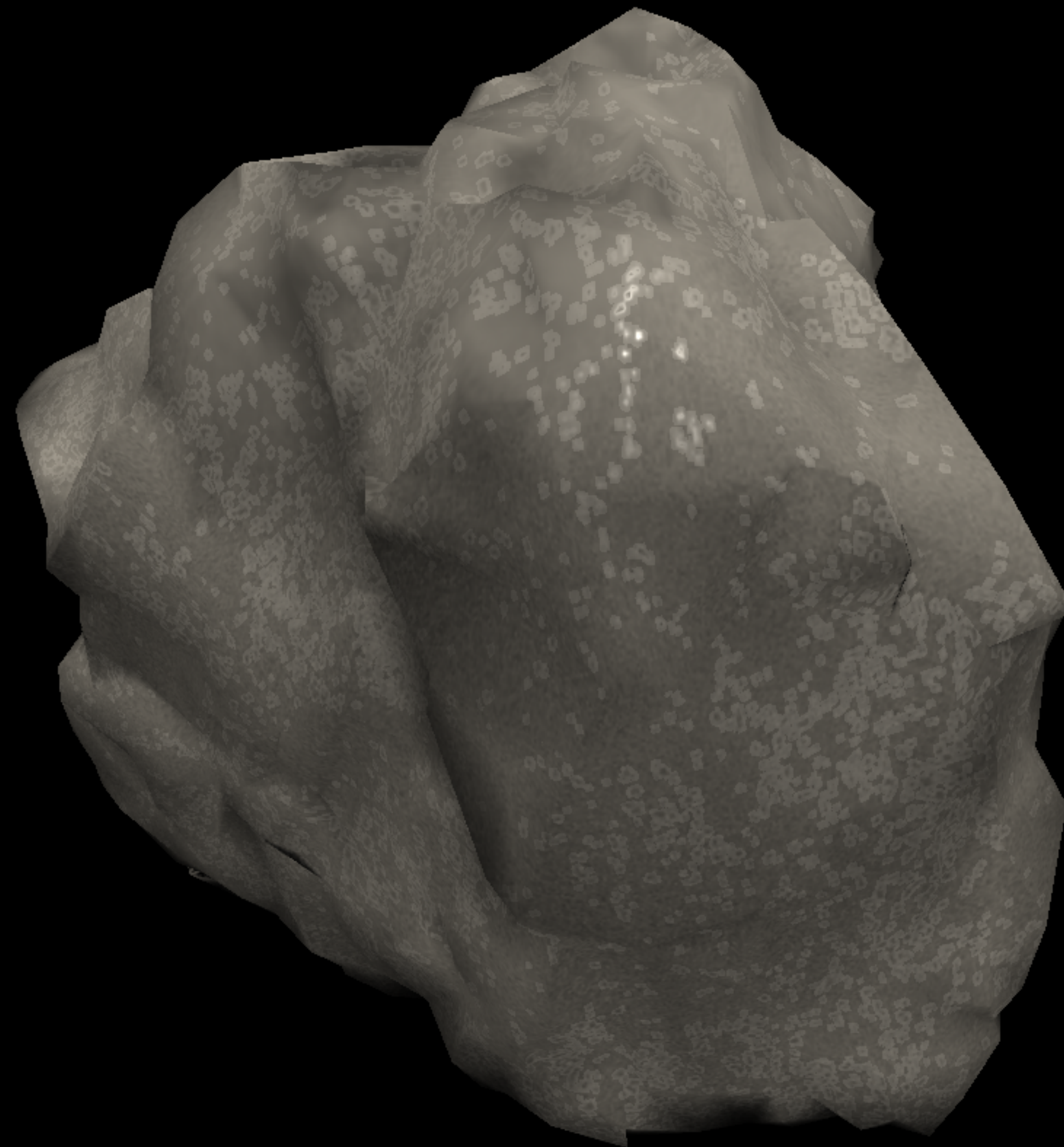
LOD 1

Simplification time: 49 sec



LOD 2

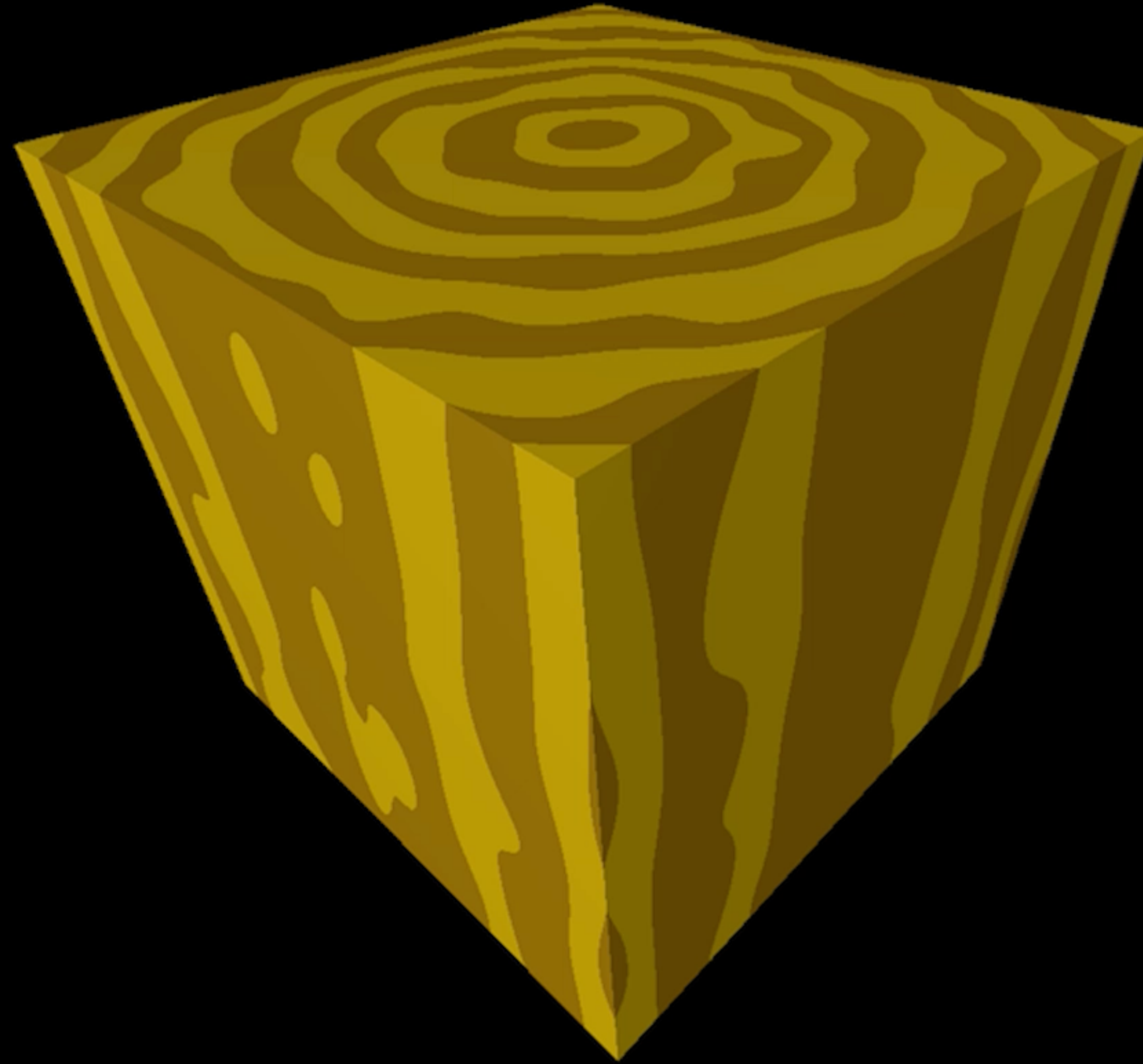
Simplification time: 49 sec



LOD 3

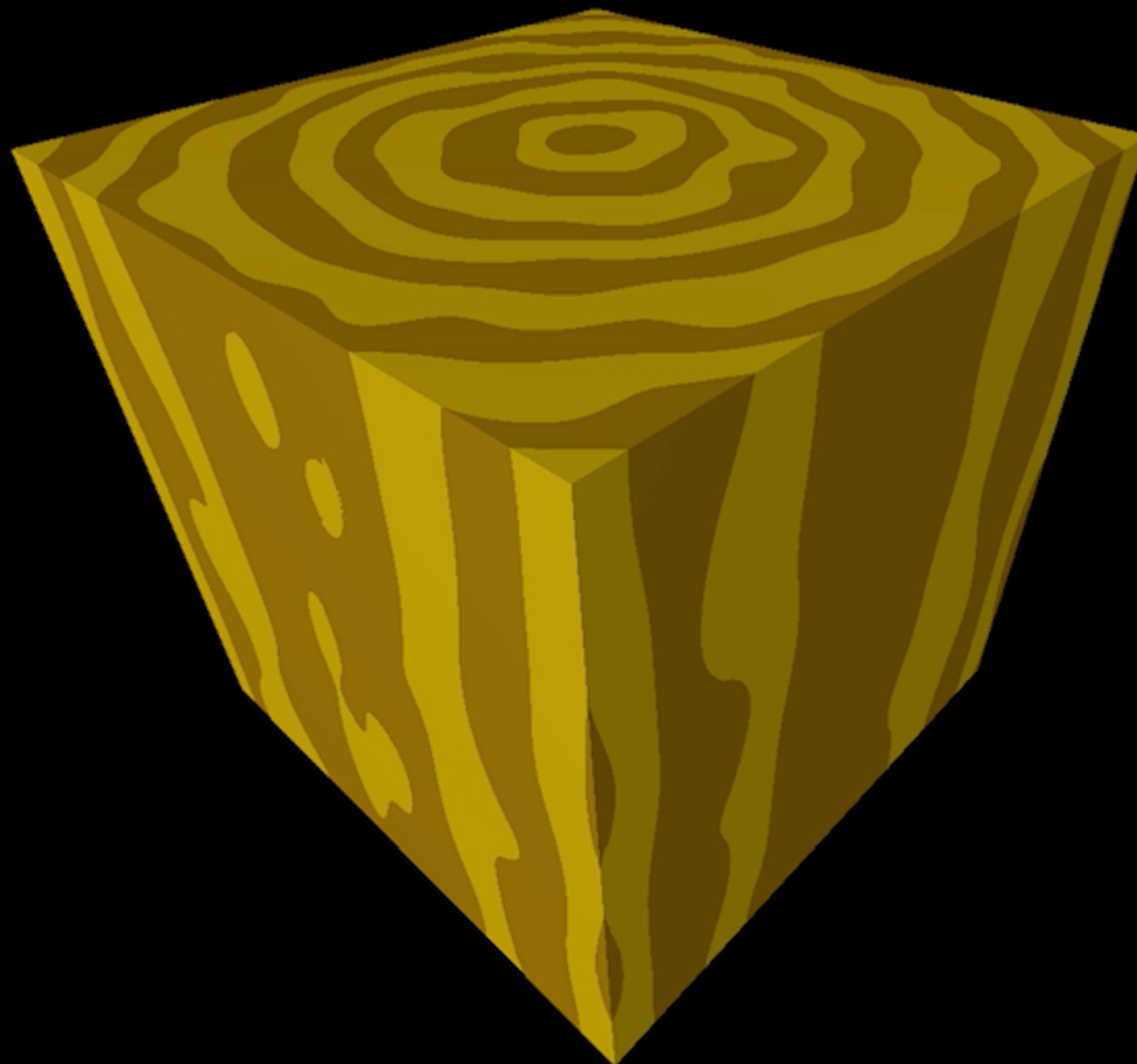
Simplification time: 49 sec



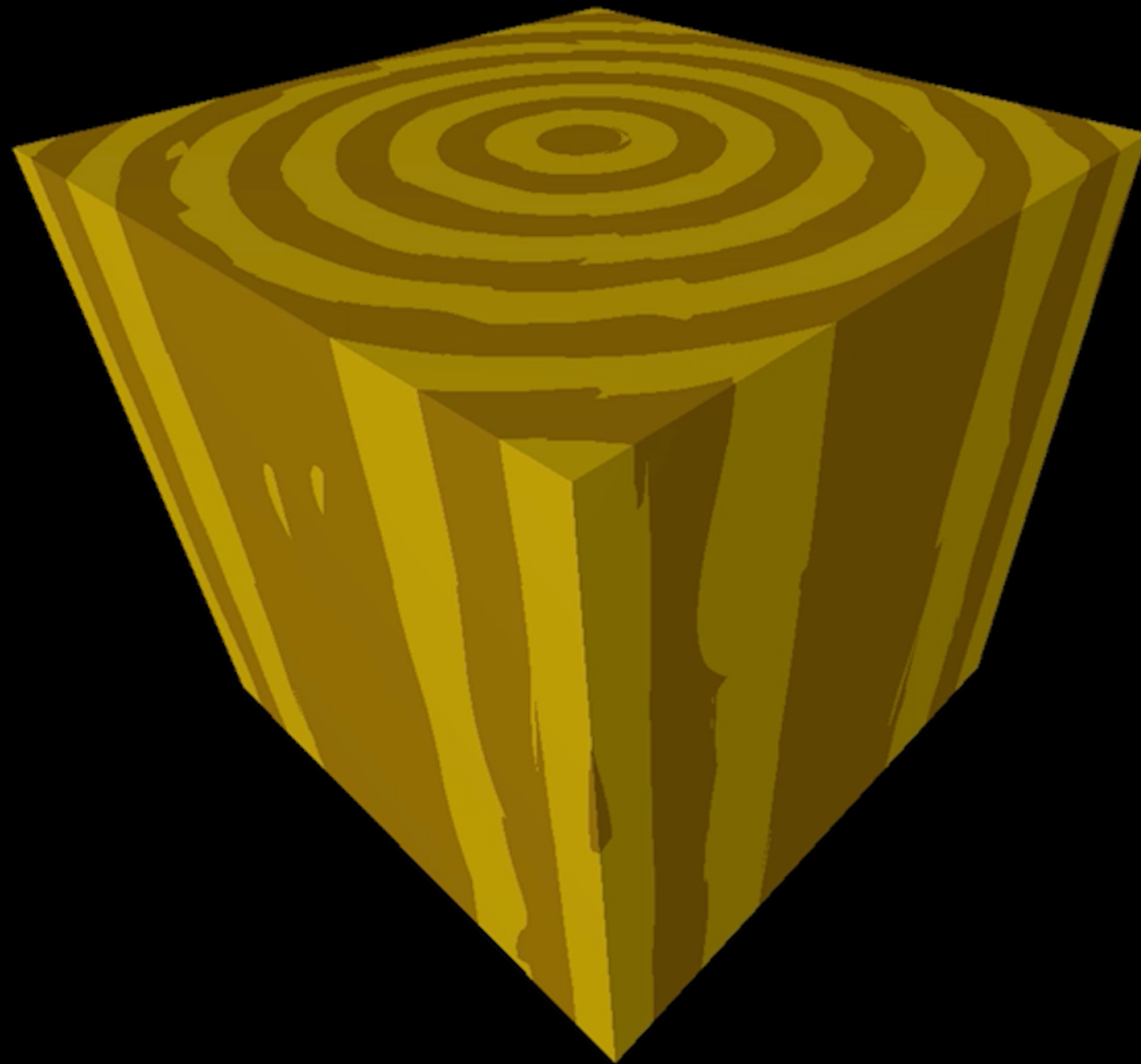


Baseline Shader

Perlin Noise Pattern

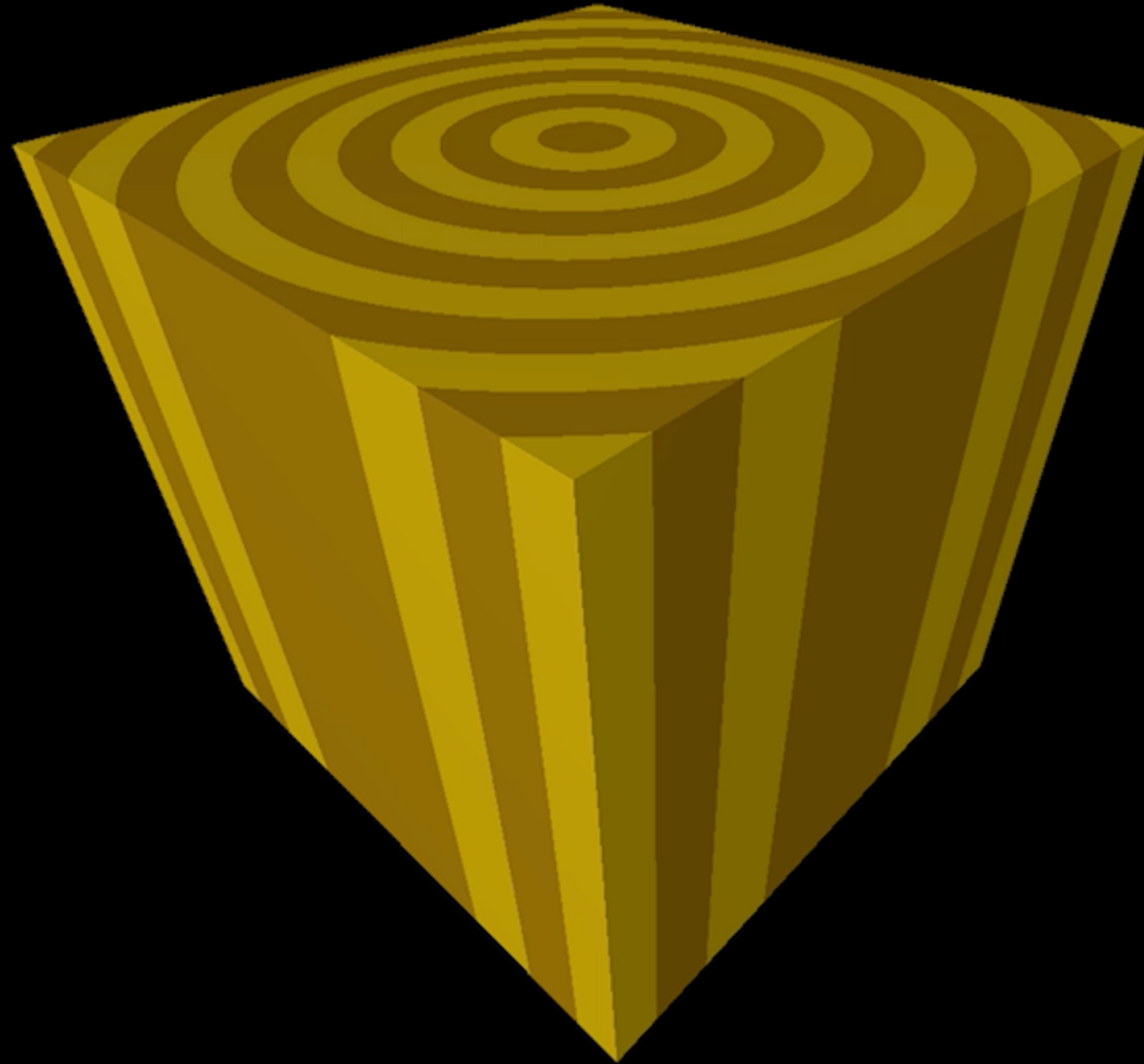


Shader 1
Simplified Perlin Noise



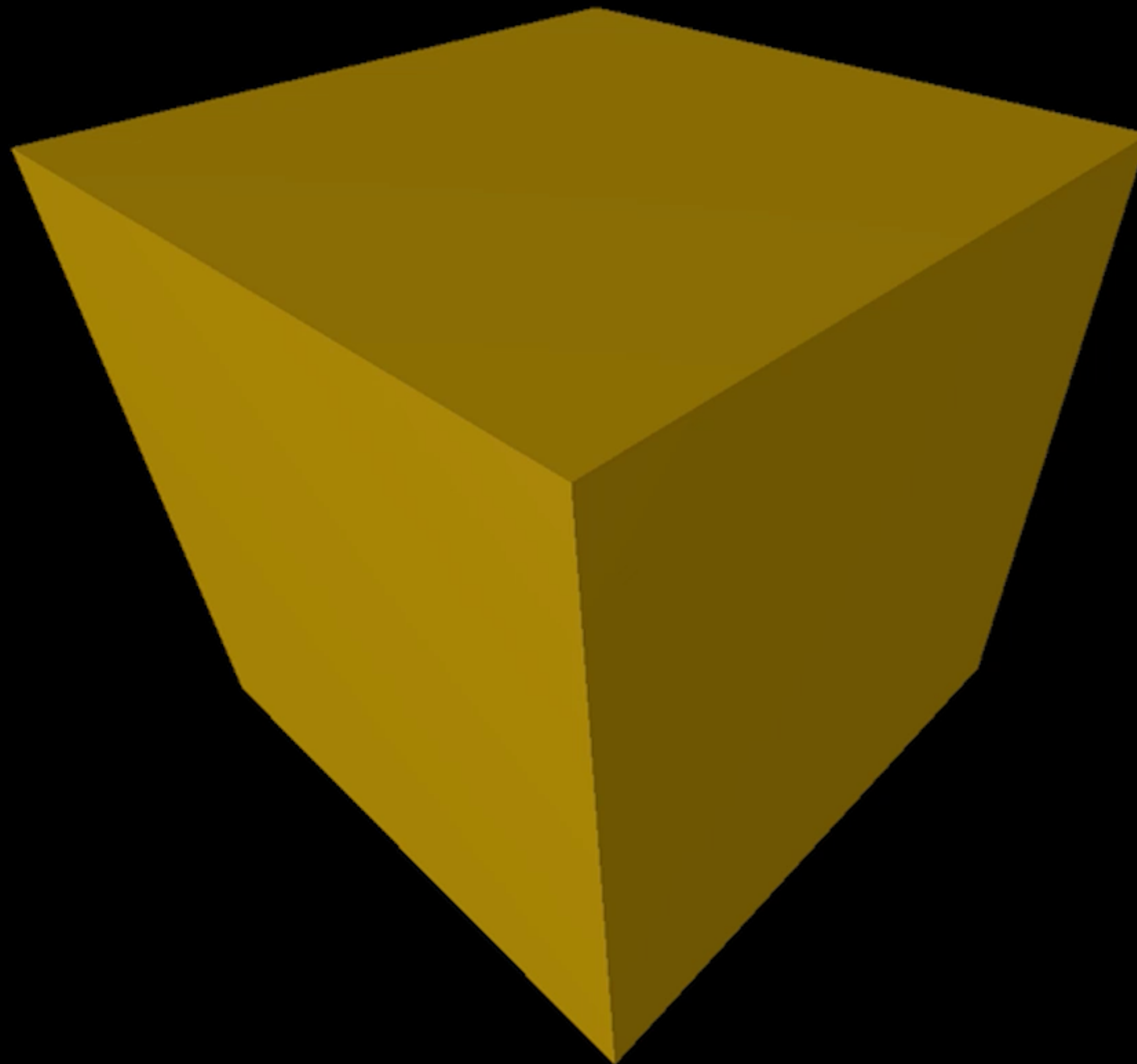
Shader 2

Simplified Perlin Noise



Shader 3

Perlin Noise Removed



Shader 4
Constant Color

No LOD (Baseline)

8.92 ms



LOD (w/ Transitions)

7.95 ms



No LOD (Baseline)

5.62 ms



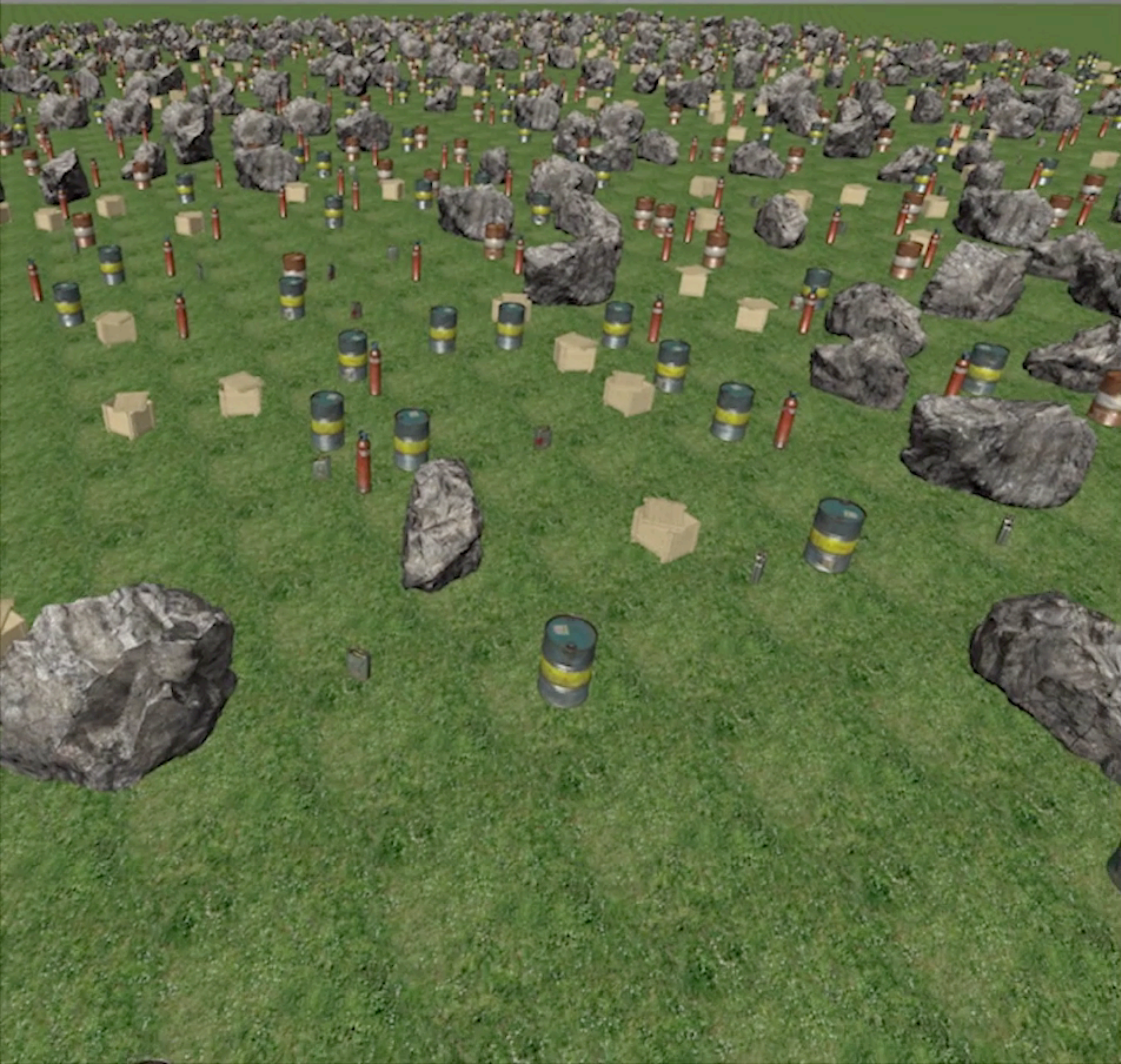
LOD (w/ Transitions)

4.07 ms



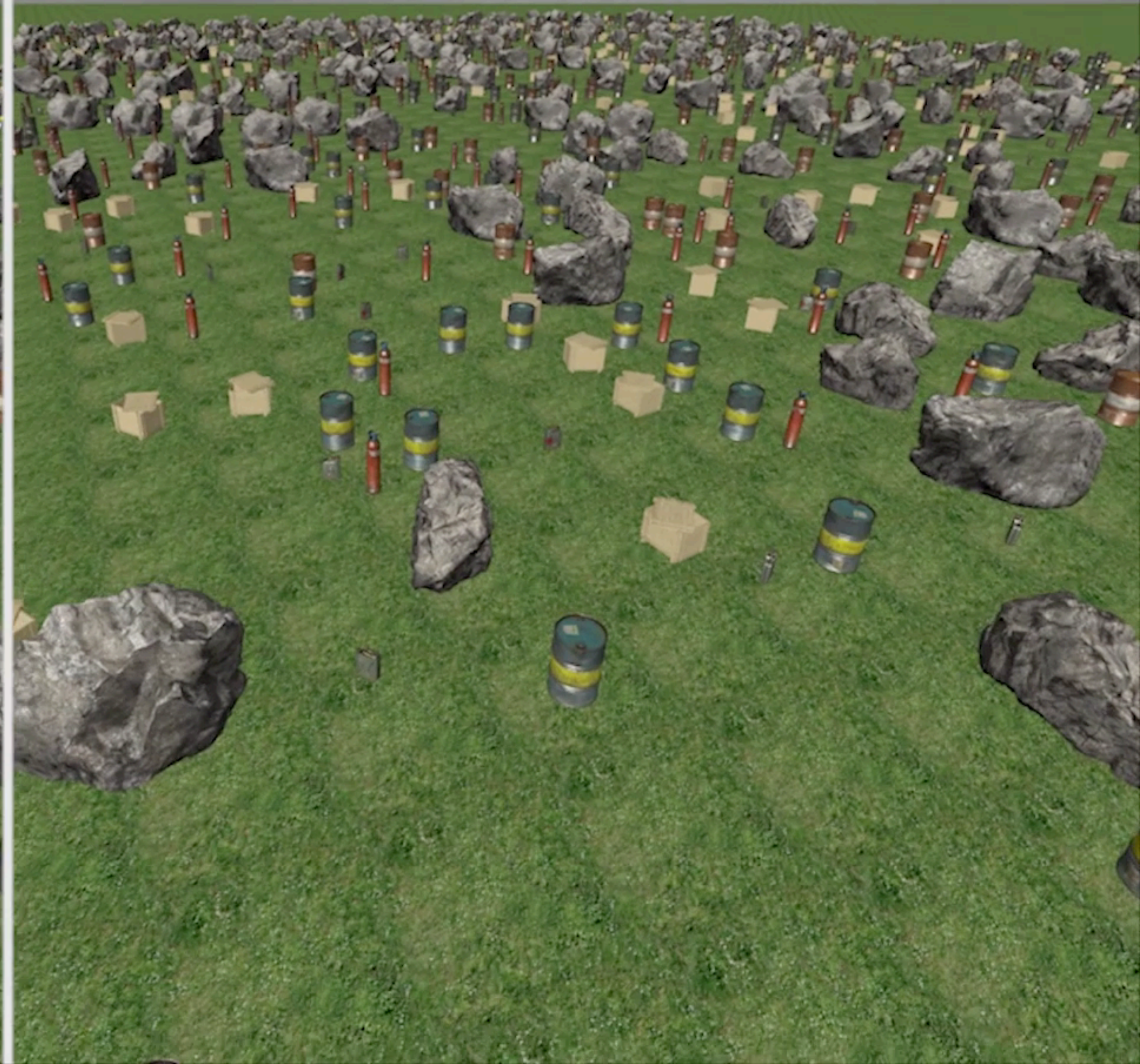
No LOD (Baseline)

5.45 ms



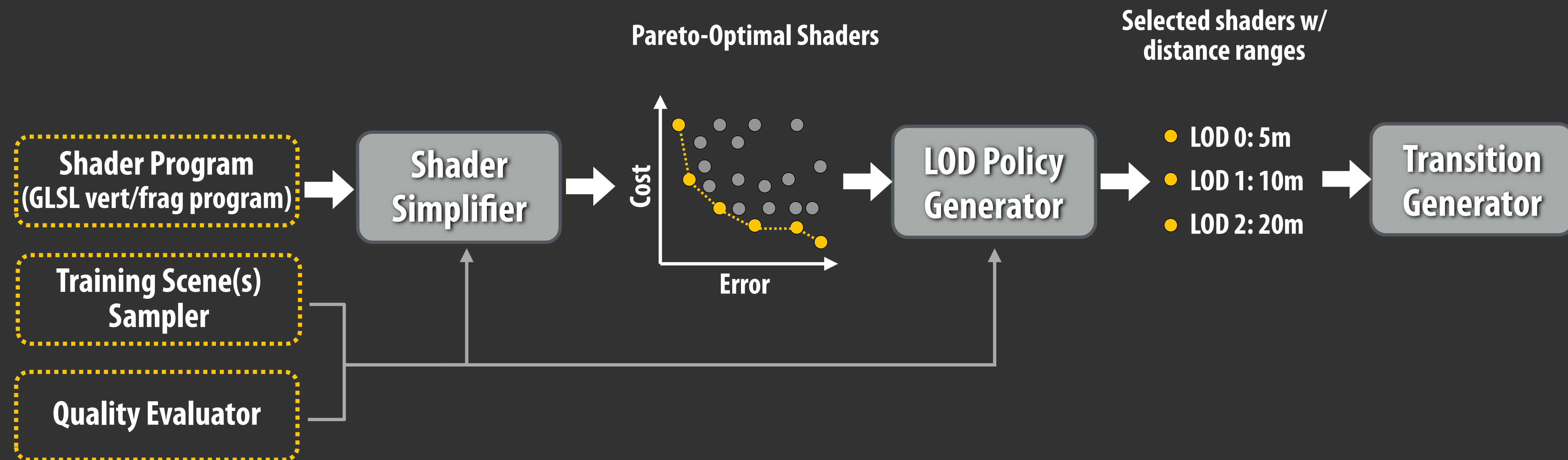
LOD (w/ Transitions)

4.74 ms



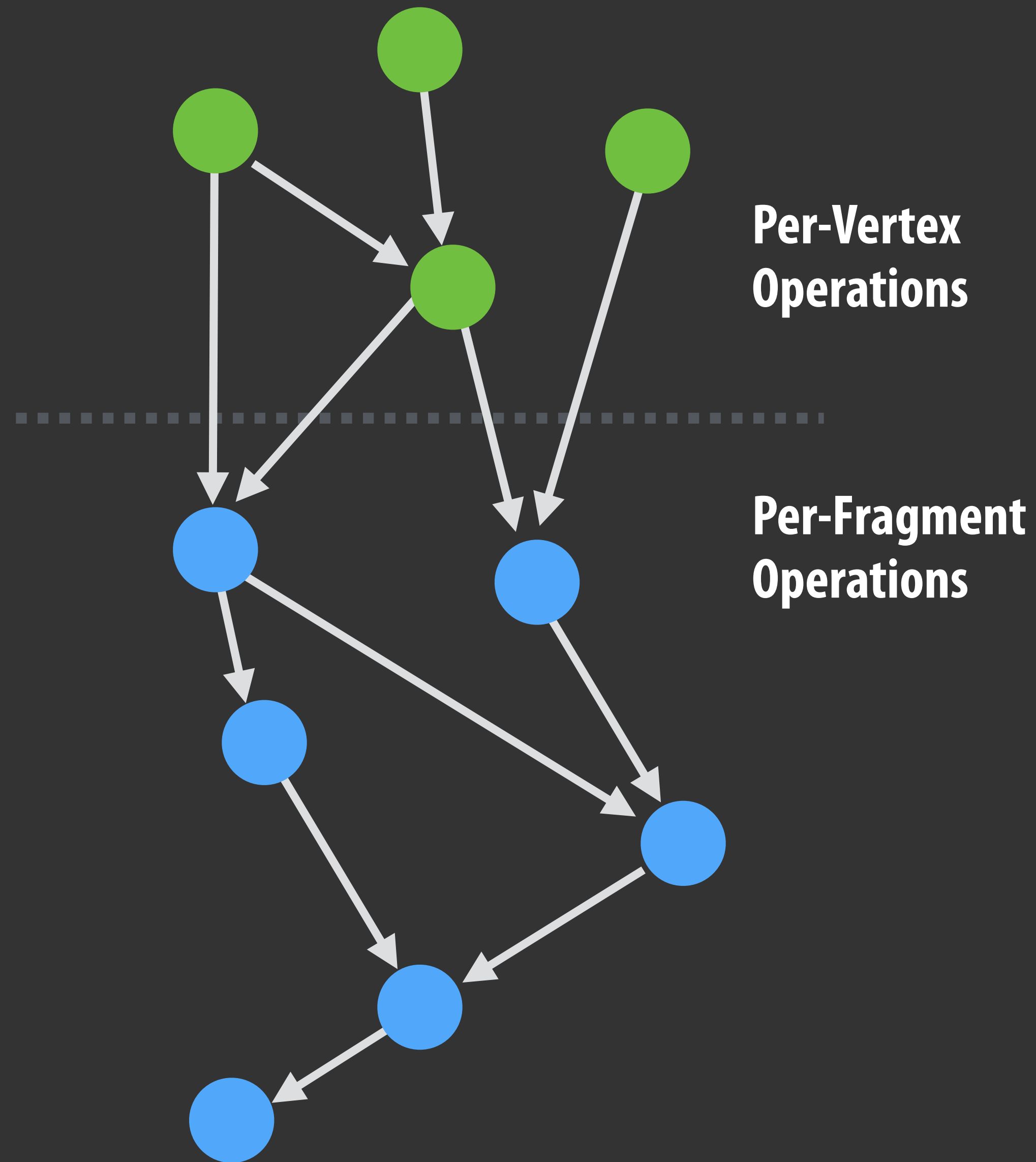
Data-driven shader compiler: system components

Tightly integrate instrumentation capability with compiler optimization framework



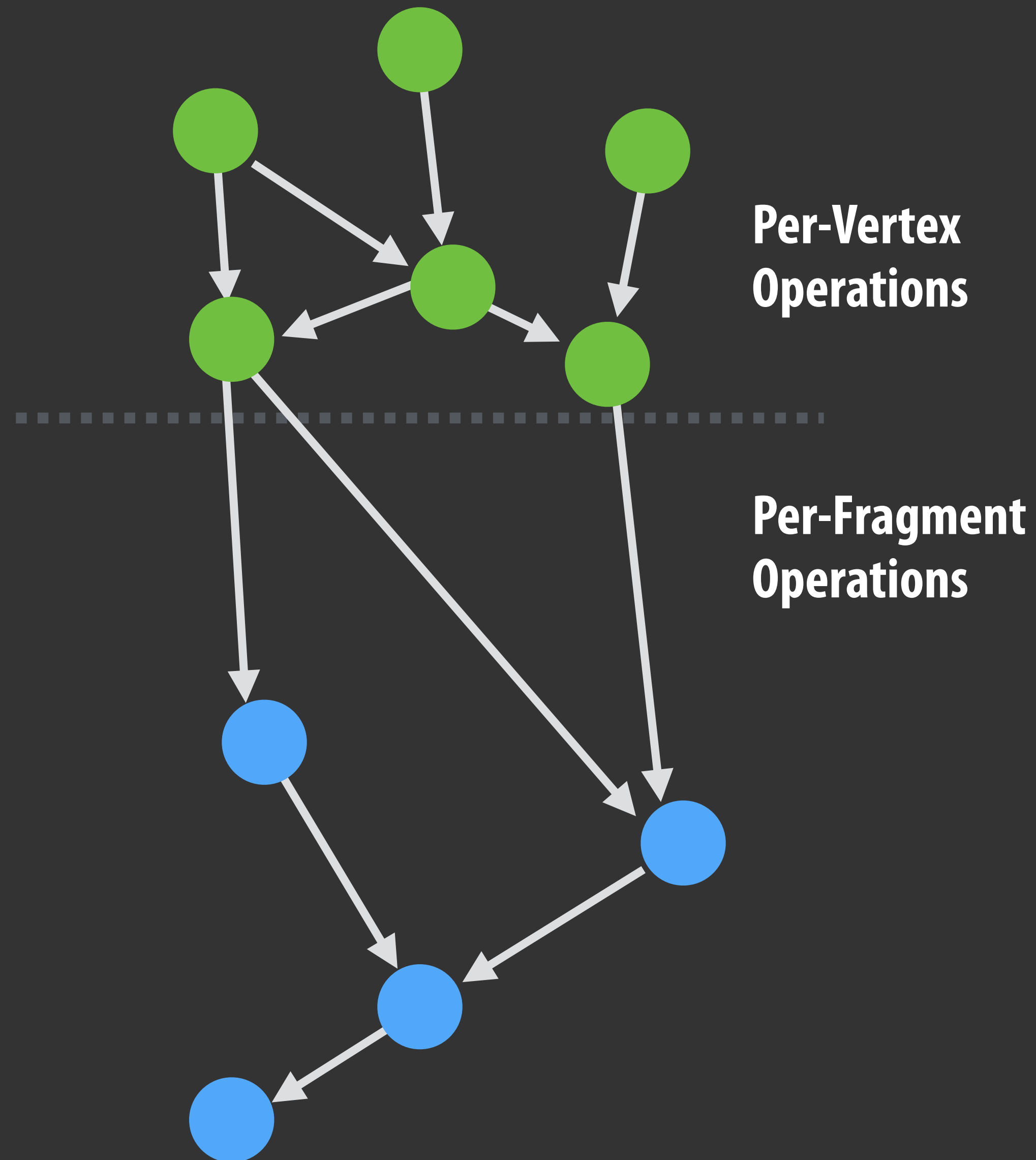
Simplification operations

Simplification operations



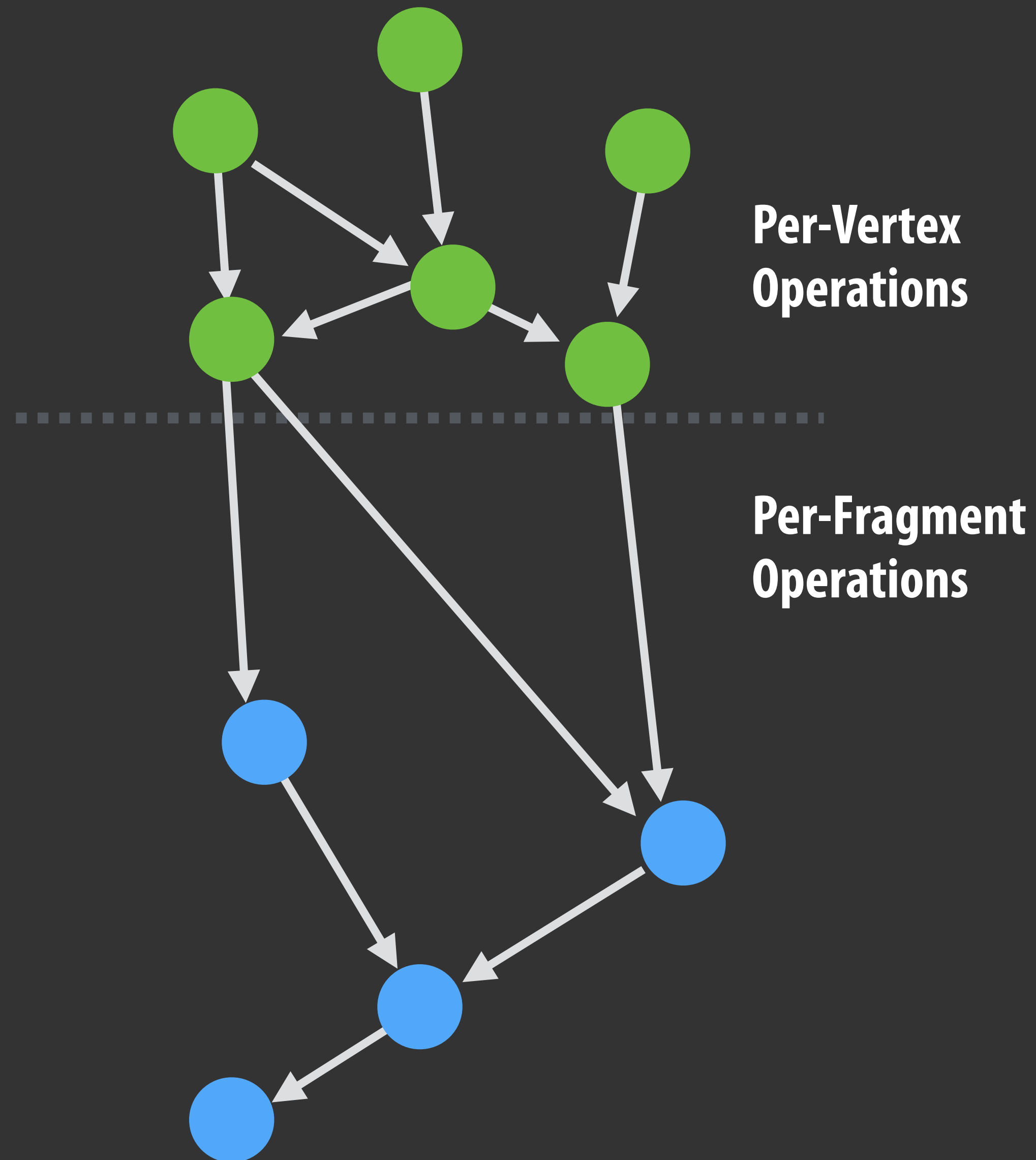
#1: move per-fragment operations to per-vertex operations when interpolation of coarsely sampled results is sufficient.

Simplification operations



#1: move per-fragment operations to per-vertex operations when interpolation of coarsely sampled results is sufficient.

Simplification operations



#2: move operations to CPU when a term is observed to be invariant over the entire object (when viewed at a distance)

Simplification operations

CPU-executed per-parameter
bind time operations

Per-Vertex
Operations

Per-Fragment
Operations

#2: move operations to CPU when a term is observed to be invariant over the entire object (when viewed at a distance)

Simplification operations

CPU-executed per-parameter
bind time operations

Per-Vertex
Operations

Per-Fragment
Operations

**#3: remove operations at any
stage that do not significantly
impact the final output**

Simplification operations

CPU-executed per-parameter
bind time operations

Per-Vertex
Operations

Per-Fragment
Operations

**#3: remove operations at any
stage that do not significantly
impact the final output**

Automatic simplification process

1. Consider applying stage movement or removal transformation to each instruction remaining in the program
 - Evaluate change in output as a result by rendering object from many different views and using random sampling of input parameter values.
 - Estimate performance of resultant shader
2. Pick optimization that provides best performance-quality trade-off
3. Repeat
 - Key technical innovation: new data-driven heuristics for pruning search space to enable rapid simplification**
(entire process completes in seconds to a few minutes)
[See He et al. SIGGRAPH Asia 2015]

A few notes

- **Generate smooth transitions by generating a single “transition shader” that lerps output of LOD_n and LOD_{n+1} terms**
 - Need only draw object once, but use transition shader
 - **Overhead of evaluating two shaders in the transition shader is often small because shaders share common subexpressions that are optimized by the compiler via redundancy removal techniques**
(8% higher than LOD_n on average, never more than 20% in initial tests)

Summary: what we've done

- **Shader LOD problem requires compiler to reason about CPU and GPU code (cross-stage representation is fundamental in IR)**
- **Tightly integrated instrumentation and data collection ability gives insight into nature of shader computations**
 - **Our goal: system should “be like a programmer”: make programmer-like decisions based on results of fine-scale instrumentation**
 - **Careful engineering to achieve high compiler performance (output in seconds-minutes)**

What we have not done (a lot)

- 1. Draw command merging: how do we force two detailed shaders to become the same as they simplify (the most important optimization in a material LOD system)**
- 2. Programmer control: what if the automatic system doesn't do what I want?**
- 3. Leverage programmer insight: simplification also employs switching to other appearance models, not just decimating the original detailed one**

Key graphics programming system challenge

Combining domain insights of an expert programmer

Knowledge of space of choices and algorithms

Identification of points where optimizations are most important

Assistance with evaluating quality

With representations that maintain strengths of automated system

Rapid exploration of many choices

Tabulating performance and output quality statistics of execution

Synthesis of efficient implementations of a choice

Separating definition of choices from functional specification

Idea successfully embodied in the Halide language for image processing [Ragan-Kelley 2012]

Also appears in other scientific programming systems:

Sequoia [Fatahalian 2006], Petabricks [2009], Legion [Bauer 2012]

```
in vec3 vert_tangent, vert_binormal, vert_normal;  
Vec3 tnormal = texture(normalMap, uv) * 2.0 - 1.0;  
Vec3 normal = mat3(vert_tangent, vert_binormal, vert_normal) * tnormal;  
normal = choose(normal, vert_normal);
```

Example 1: no explicit specification of pipeline stage for a term. (only provide a list of “permitted stages” for a term)

```
Vec3 lighting = ComputeLighting(normal,...);  
Vec3 color = FresnelBlend(color0, color1, color2, View, normal);  
color = choose(color, moveToCPU(color));  
Result = color * lighting;
```

Example 2: Use either the per-vertex normal or the result of normal mapping.

Example 3: Use the fully computed result or the average color for the object that was computed a prior on the host and available as a uniform parameter.

Bigger picture

Problem: huge space of choices in a modern rendering engine

- Often the possible choices are well understood, but what choice is most appropriate for current situation is a complex problem: render mode, platform, asset, etc.
- Implementation complexity is too high to explore these options rapidly
- Space of choices is growing:
 - More complex worlds, user generated content in worlds
 - Level of detail (and relationship to anti-aliasing and prefiltering)
 - Platform-specific implementation differences
 - Context: geometry pass / beauty pass, forward / deferred components
 - Dynamic branching vs. CPU-side binning
 - **Shading space decisions: fragment space vs. object/texel space**
 - **Variable shading rate optimizations: coarse fragment shading, foveated rendering**
 - **Spatio-temporal rates: high-frame rate for VR**

What I think about what we're doing: exploring a "rendering system IR"

**Seeking more powerful graphics programming representations that
embody space of decisions a programmer wishes to explore
(But never relinquishing ability for programmer to make the decision when desired)**

**Building compiler technology for generating and using results of large-
scale instrumentation to guide automation of some choices
(analyzing outputs, assessing quality, detecting aliasing, etc.)**

Not clear what forms of choices are amenable to this approach

Thank you

kayvonf@cs.cmu.edu

**Yong He is the primary graduate student on this work. He knows the details.
(www.csyong.net)**