Lecture 13:

The NeRF Explosion + Review of Traditional Graphics Pipeline

Visual Computing Systems
Stanford CS348K, Spring 2022

Volumetric representations

$$\sigma(\mathrm{p})$$

$$c(\mathrm{p}, \omega) = c(x, y, z, \phi, \theta)$$

Volume density and color at all points in space.

Regular 3D grid?

Storage requirements:
512^3 cells

Ignore directional dependency: rgbσ 4 bytes/cell
536 GB

Now directional dependency on (ϕ, θ) … much worse

Typical challenge: limited resolution

Credit: Voxel Ville NFT (voxelville.io)

Rendering volumes

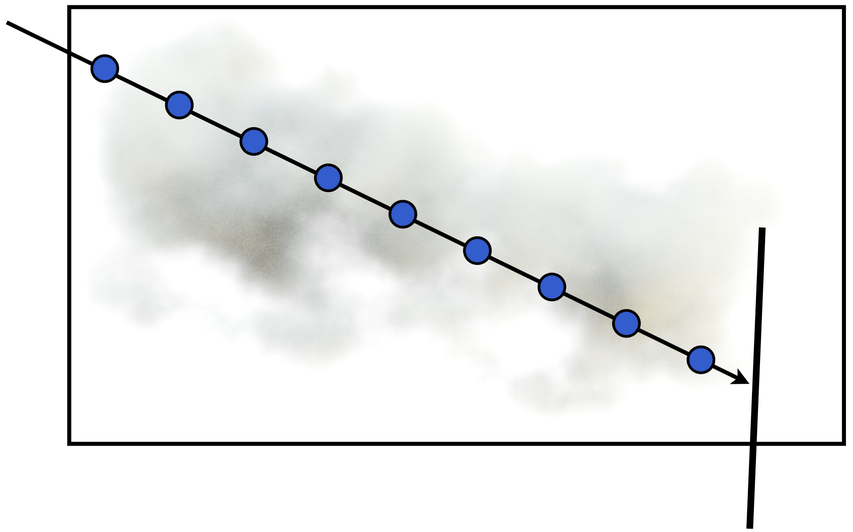
Given "camera ray" from point o in direction w….

$$\mathbf{r}(t) = \mathbf{o} + t\omega$$

And volume with density and directional radiance.

$\sigma(\mathrm{p})$

$c(\mathrm{p}, \omega)$ ← Volume density and color at all points in space.



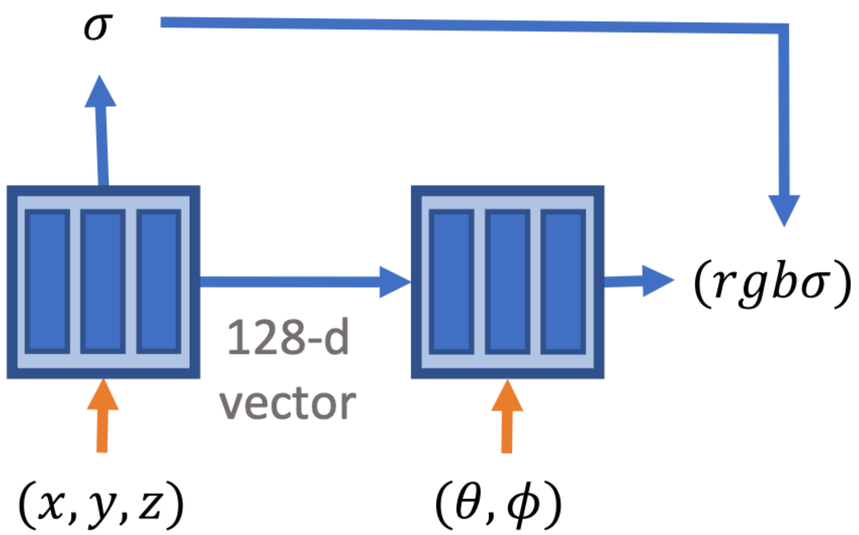
Step through the volume to compute radiance along the ray. (Easily differentiable computation)

$$C(\mathbf{r}) = \int_{t_n}^{t_f} T(t)\sigma(\mathbf{r}(t))\mathbf{c}(\mathbf{r}(t), \mathbf{d})dt\,, \text{ where } T(t) = \exp\left(-\int_{t_n}^{t} \sigma(\mathbf{r}(s))ds\right)$$

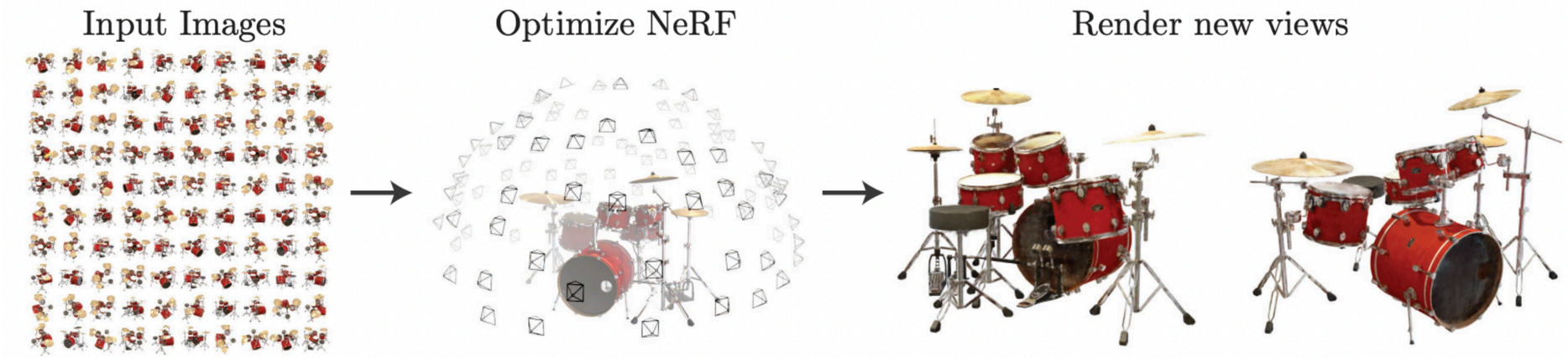
Learning (compressed) representations

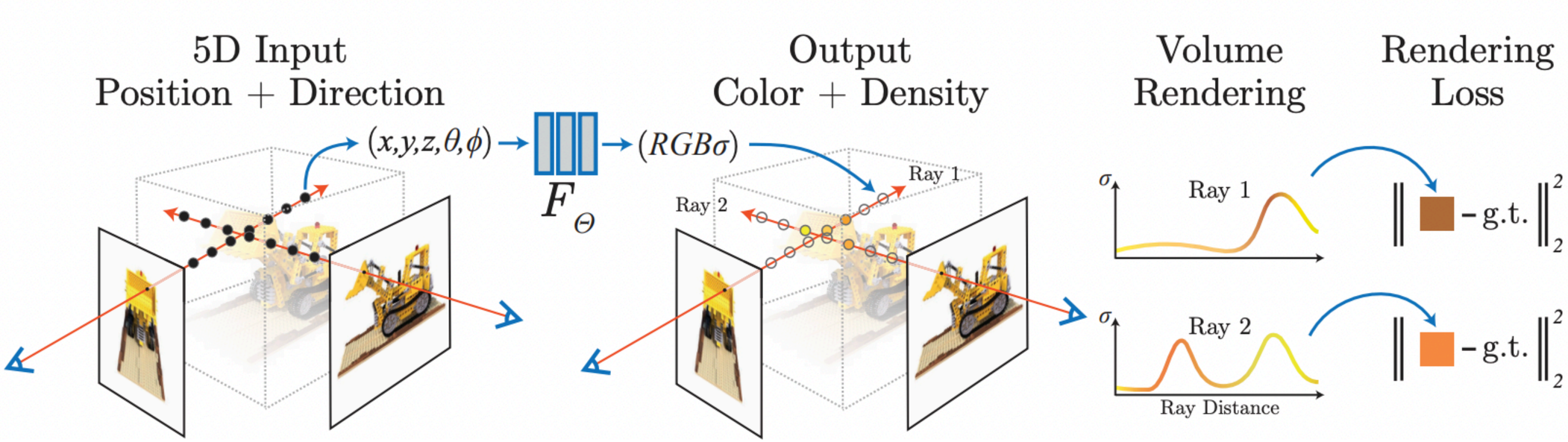
- Why not just learn an approximation to the continuous function that matches observations from different viewpoints?

$$(\mathrm{p}, \omega) \rightarrow F_\theta(\mathrm{p}, \omega) \rightarrow \begin{matrix} \sigma(\mathrm{p}) \\ c(\mathrm{p}, \omega) \end{matrix}$$



Learning neural radiance fields (NeRF)

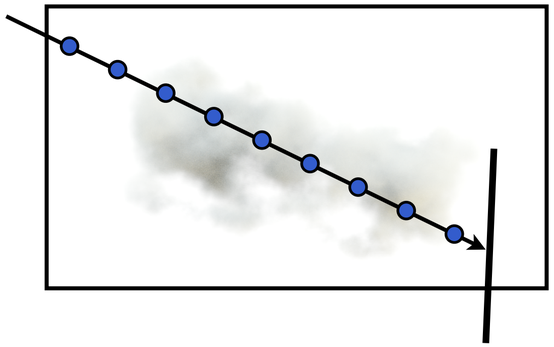




What just happened?

- Continuous coordinate-based representation vs regular grid: MLP "learns" to use weights to produce high-resolution output where needed given input data

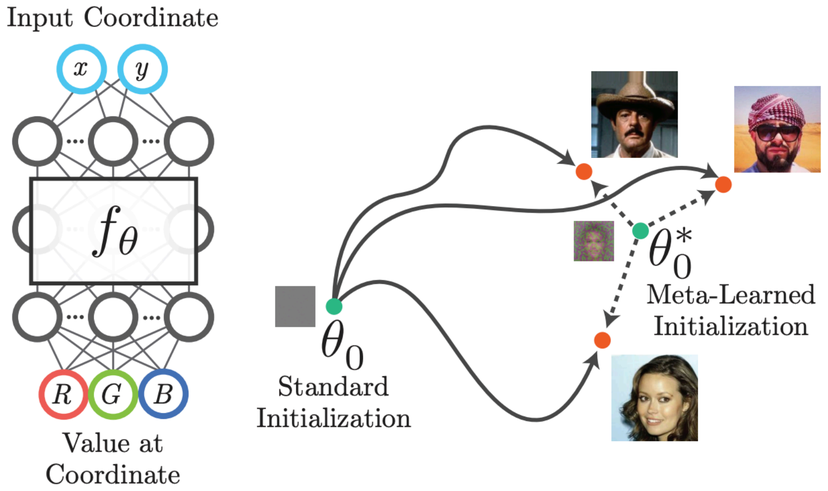
- Compact representation: trades-off space for expensive rendering
  - Good: a few MBs = effectively very high dense grid
  - Bad: must evaluate MLP every step
  - Bad: must step densely (don't know where the surface is)

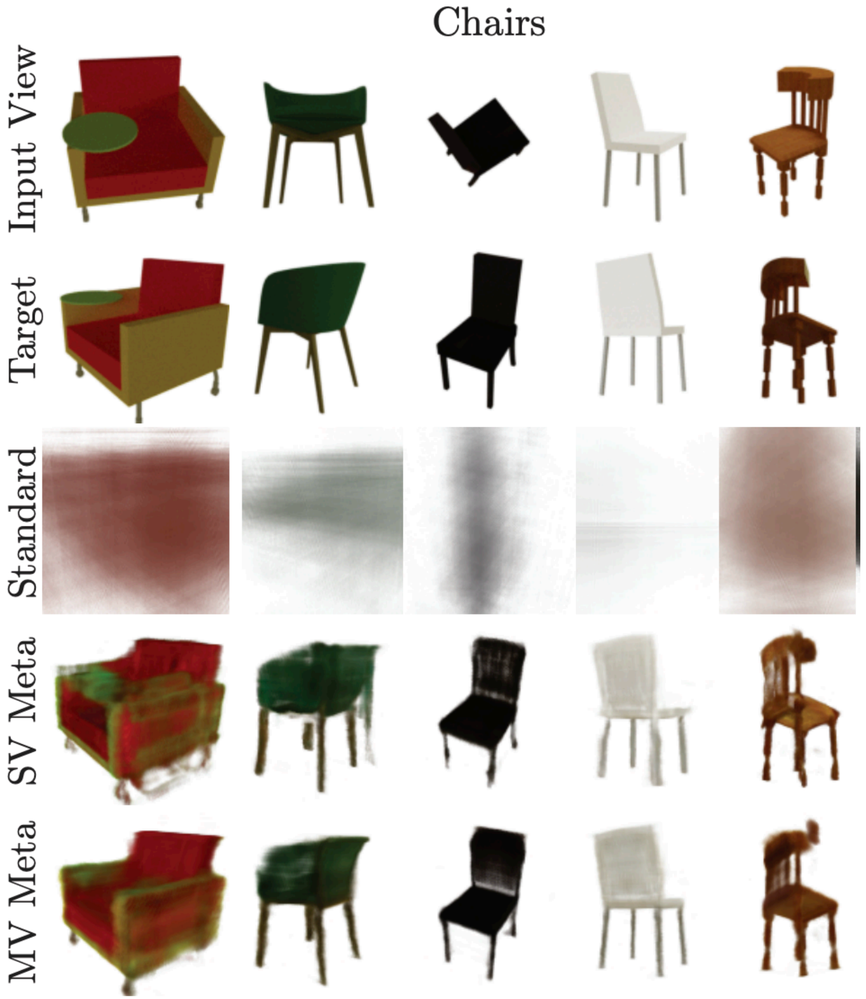


- Compact representation: optimization can learns to interpolate views despite complexity of volume density and radiance function
  - Only structural bias is the separation into positional σ and directional rgb
  - Training time: hours to a day to learn a good NeRF

Improving NeRF training

- Improve training speed AND reduce number of images needed using meta-learning





Improving rendering performance

- Main idea: move to a different point in the compression-compute trade-off space

- Two main ideas:
  - Avoid stepping densely through space (evaluating the MLP to find density = 0)
  - Avoid evaluating the MLP when you can

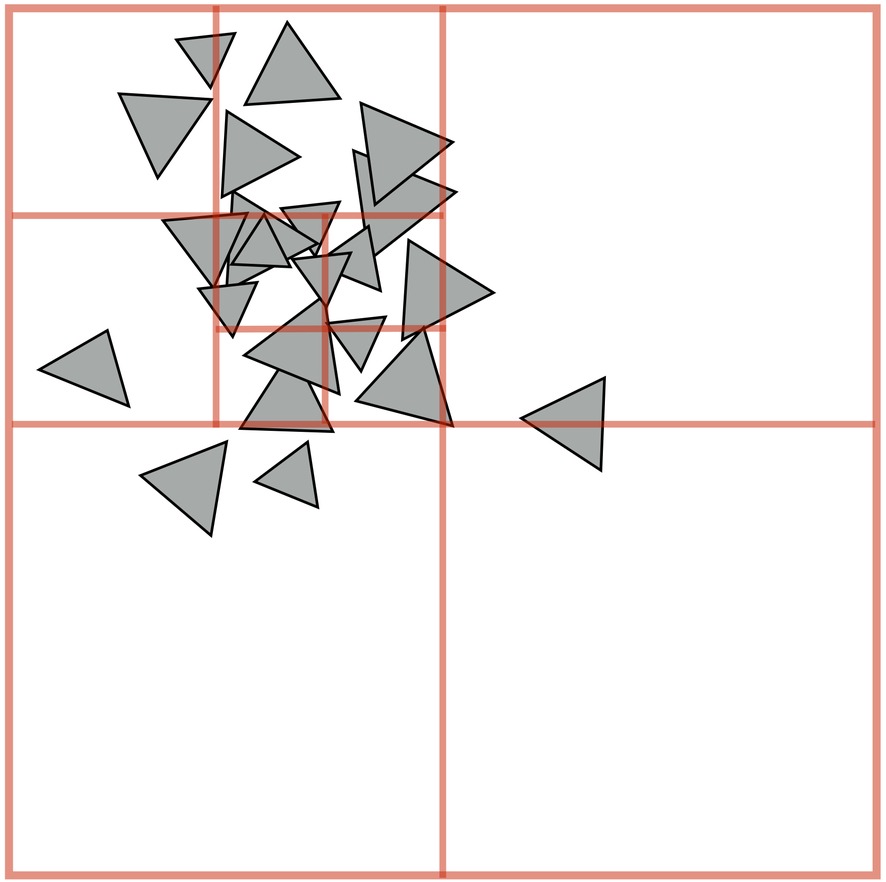
Quad-tree / octree

Quad-tree: nodes have 4 children (partitions 2D space)

Octree: nodes have 8 children (partitions 3D space)

Like uniform grid: easy to build (don't have to choose partition planes)

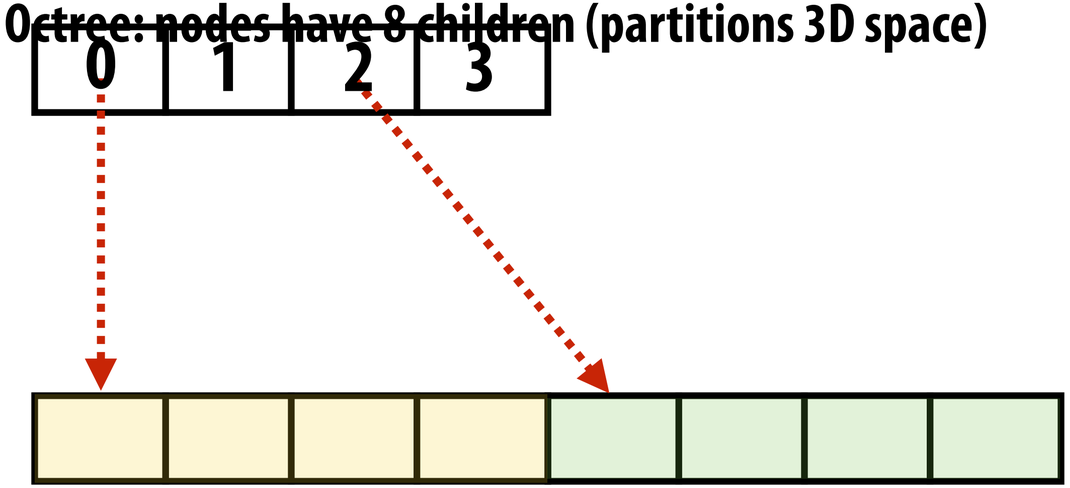
Has greater ability to adapt to location of scene geometry than uniform grid.

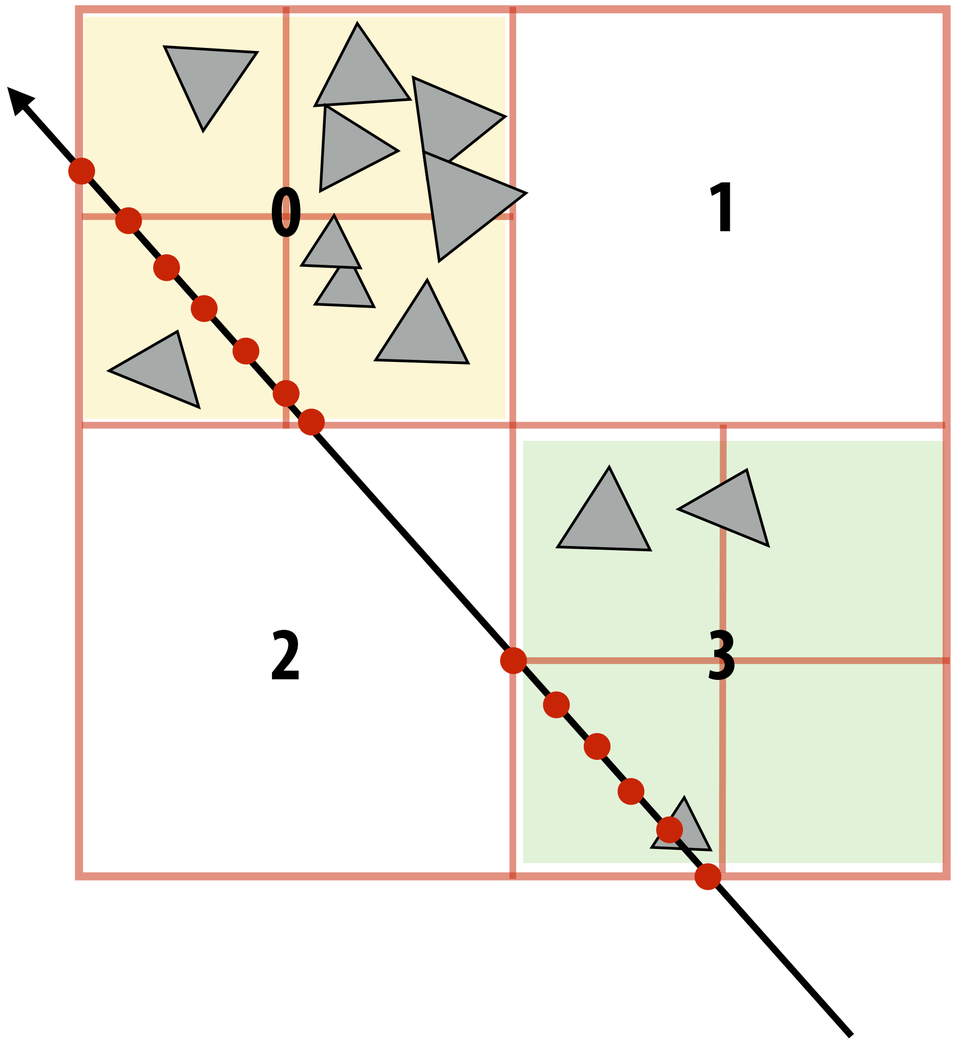


Simple two-level sparse quad tree

Quad-tree: nodes have 4 children (partitions 2D space)

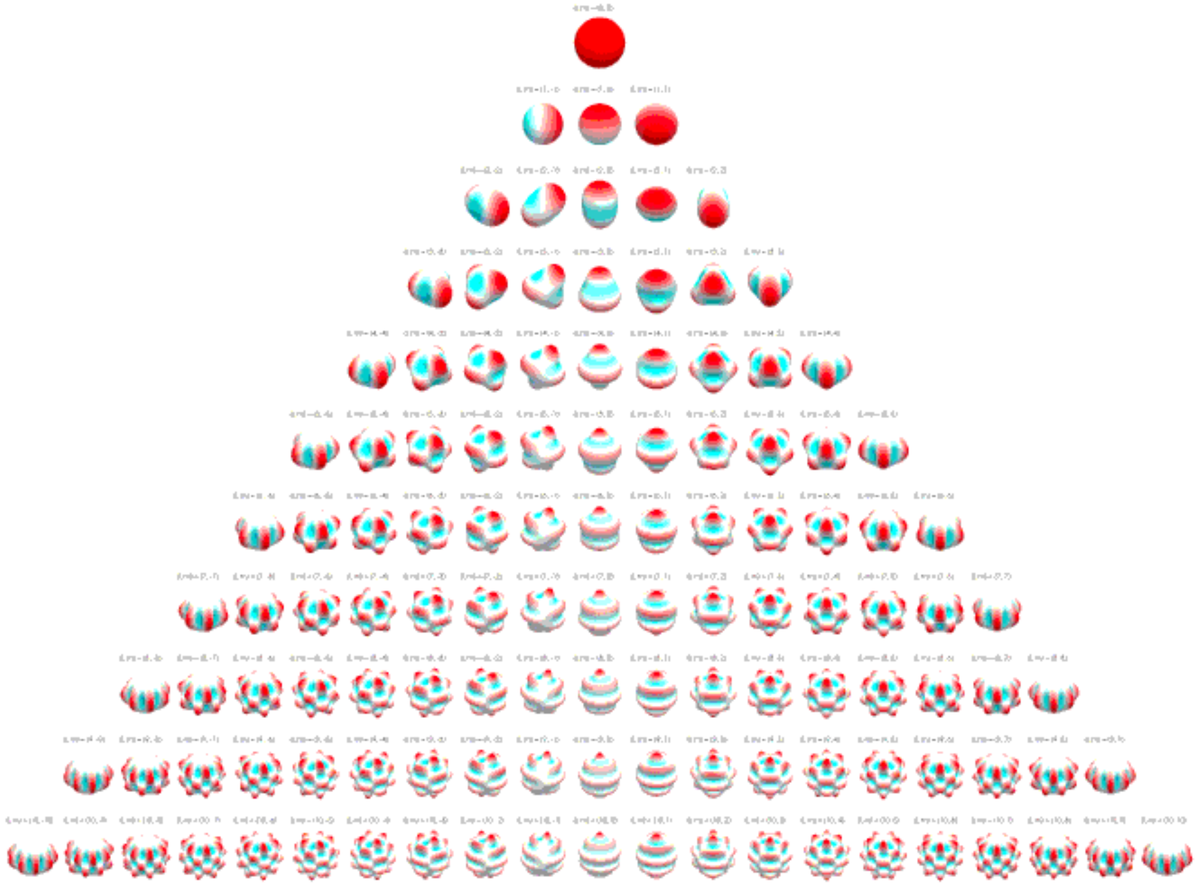
Octree: nodes have 8 children (partitions 3D space)

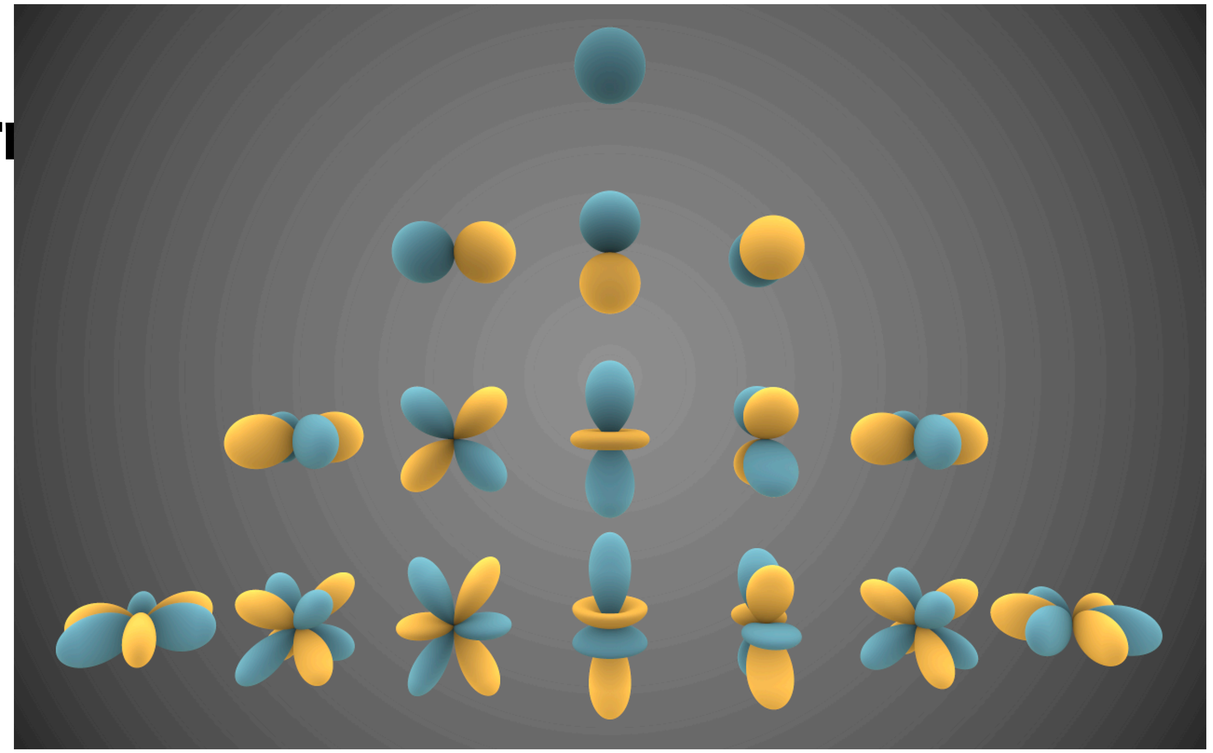




Spherical harmonics

- Useful basis for representing directional inform
- Analogy: cosine basis on the sphere



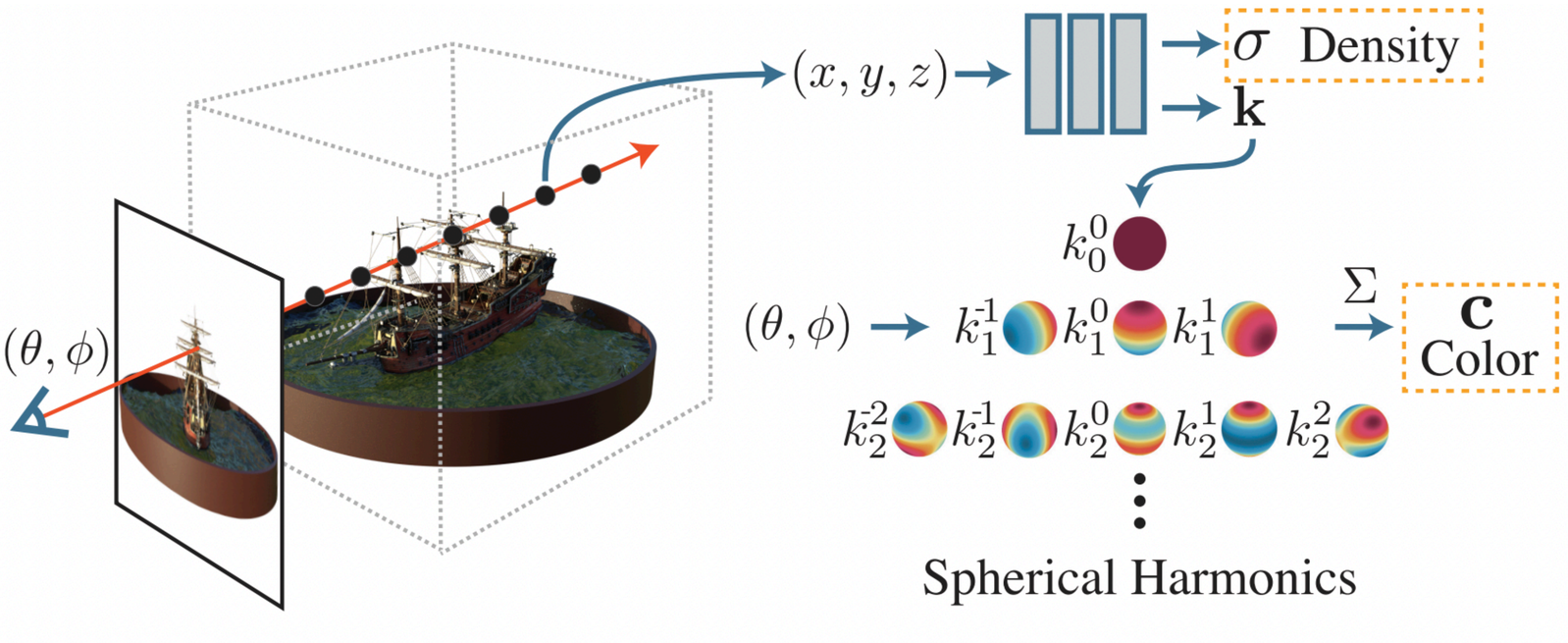


- Represent $c(\mathrm{p}, \omega)$ compactly by projecting into basis of SH.

$$\mathrm{Y}_l^m(\omega)$$

Okay… so let's have F_θ just predict SH coefficients

(We can compute $c(\mathrm{p}, \omega)$ from the coefficients as needed)

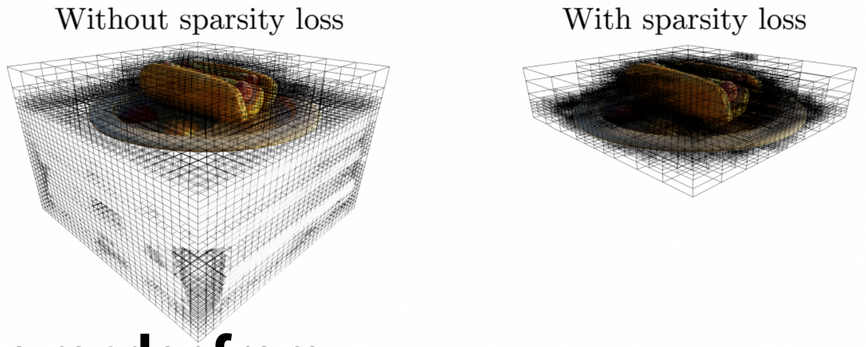


$$F_\theta(\mathrm{p}) \rightarrow \begin{matrix} \sigma(\mathrm{p}) \\ K(\mathrm{p}) \end{matrix}$$

(Vector of SH coefficients)

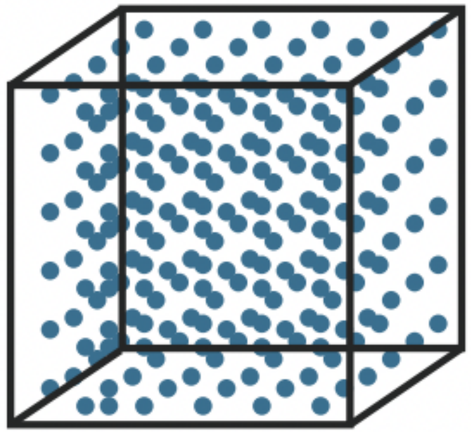
Note, now the MLP is just a function of 3D coordinates p.

Okay… just lets train for a bit…

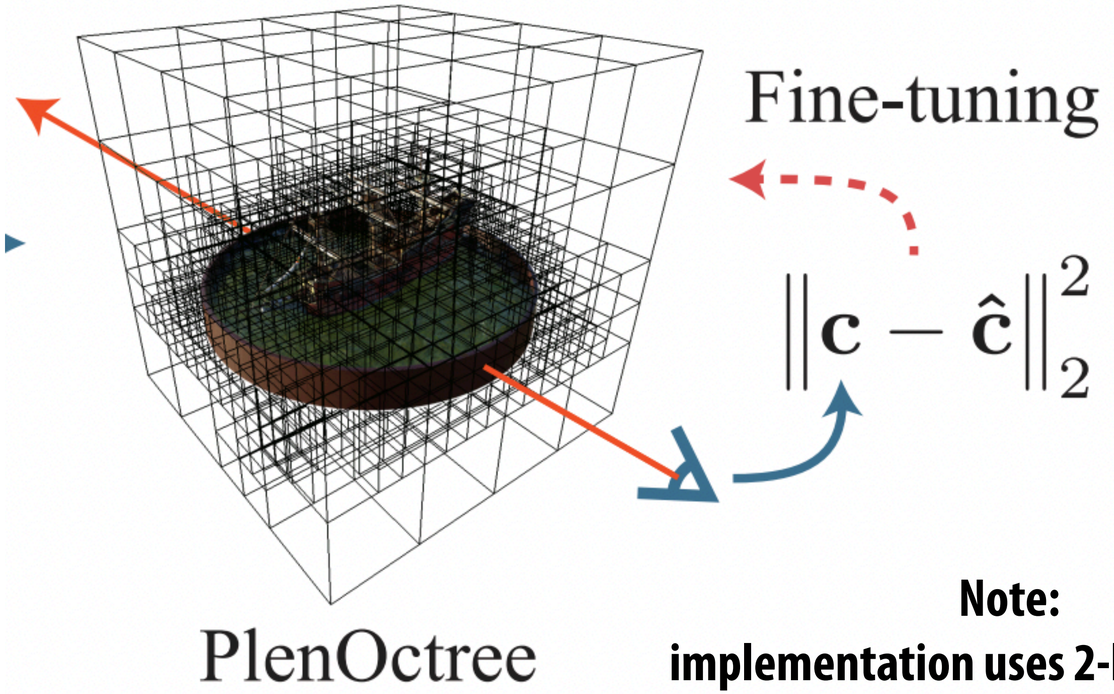


- Until we know where the empty space clearly is.

- Then move to an octree representation that's more efficient to render from…
- With the octree structure fixed, we can continue to optimize SH coefficients and density at leaves with SGD



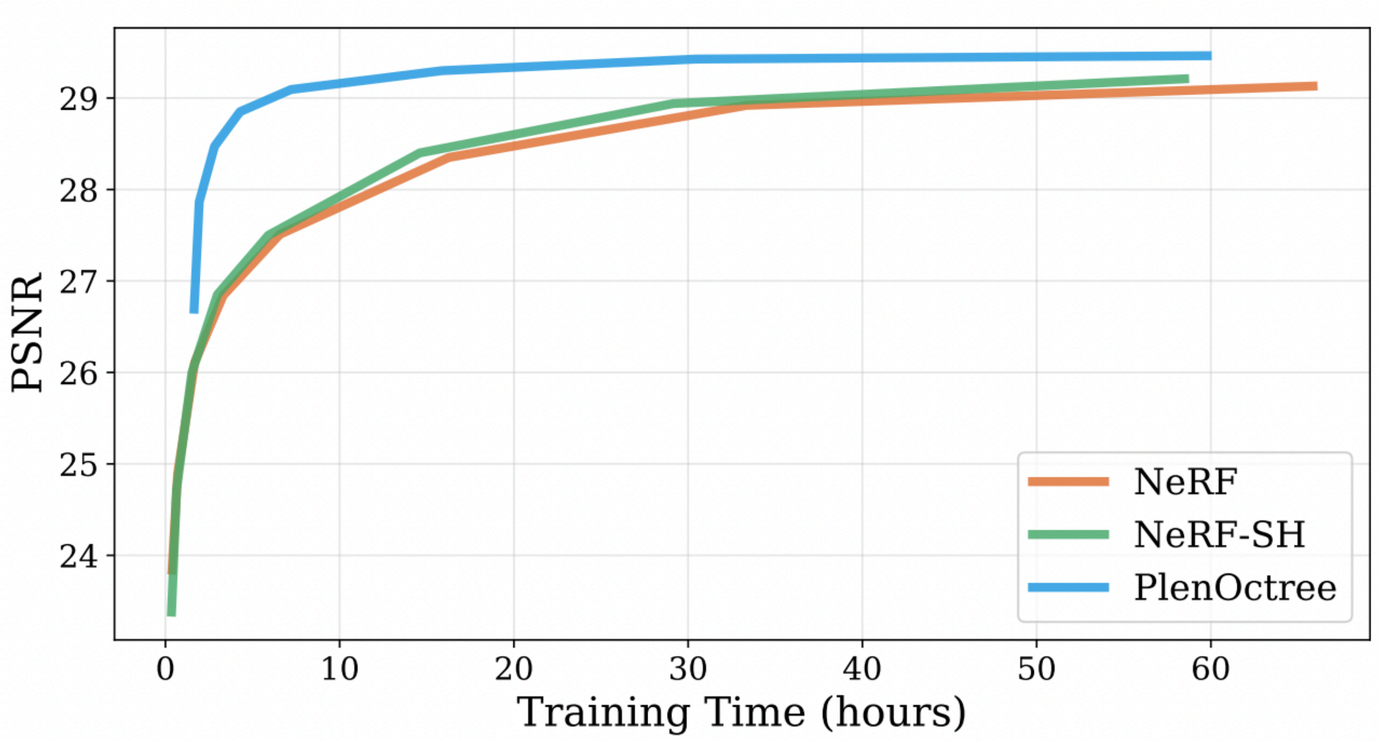
Use MLP to densely sample volume
(Find the empty space)



Note:
implementation uses 2-level octreee

What just happened?

- We performed initial training a la original NeRF
- Once we get a sense of the empty space, we add a traditional acceleration structure
- That structure speeds up rendering (a lot), and can even be a better structure to "fine tune" training on, since MLP need not be trained to convergence

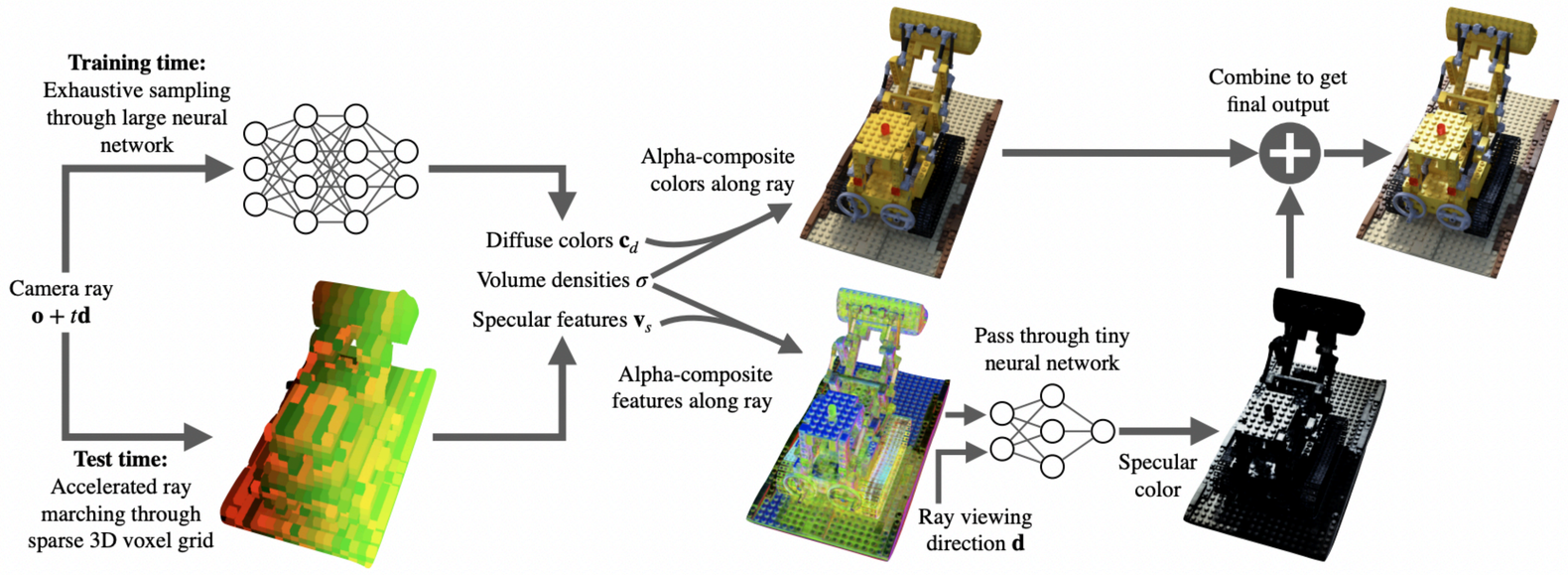


- Cost? octree structure now 100's of MBs instead of a few MBs for MLP

Another take (by other groups)

- Same idea: densely sample MLP to "bake" into sparse octree representation
- Instead of SH , MLP outputs density, diffuse color, and specular (directional) "features"

$$F_\theta(\mathrm{p}) \rightarrow \begin{matrix} \sigma(\mathrm{p}) \\ \mathrm{c}(\mathrm{p}) \\ \mathrm{v}(\mathrm{p}) \end{matrix}$$



- "Volume render" both the diffuse colors and the features, and use one MLP eval at the end to turn the diffuse color and features into a final color
  - Note: now 1 MLP eval per ray, instead of per step

Finally…back to where we began.

- Start with a dense 3D grid of SH coefficients, learn that at low resolution
- Now move to a sparse higher resolution representation
- Directly optimize for opacities and SH coefficients using differentiable volume rendering

- No neural networks. Just optimizing the octree representation of baked SH lighting

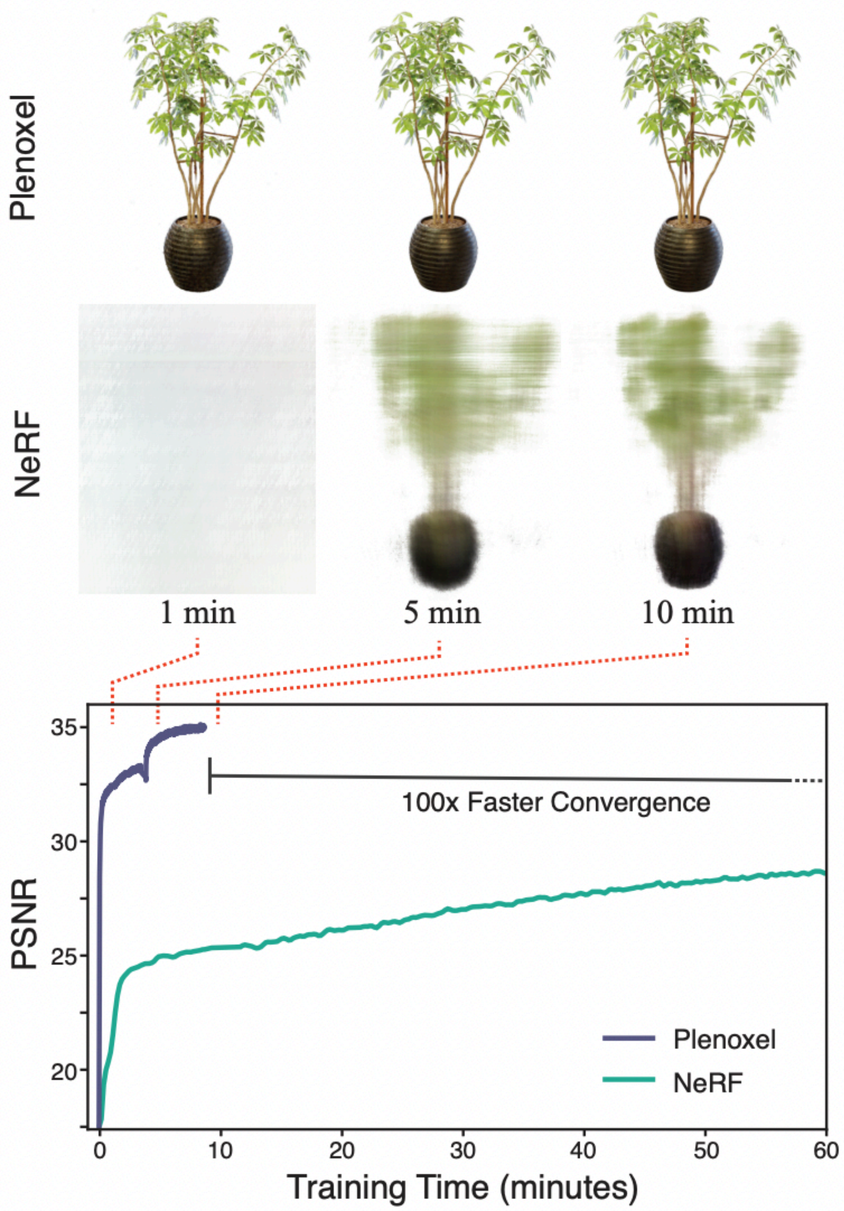
Light probe locations in a game

Here: SH probes sampled on a uniform grid

Finally…back to where we began.

- Start with a dense 3D grid of SH coefficients, learn that at low resolution
- Now move to a sparse higher resolution representation
- Directly optimize for opacities and SH coefficients using differentiable volume rendering
- No neural networks. Just optimizing the octree representation of baked SH lighting

- Takeaway: conventional computer graphics representations are *much* more efficient representations to learn on.



What have we learned?

Let's talk a little "traditional" 3D graphics…

Reminder: what is an “architecture”?

(not distinguishing between software or hardware architecture)

A system architecture is an abstraction

- **Entities (state)**
  - Registers, buffers, vectors, triangles, lights, pixels, images
- **Operations (that manipulate state)**
  - Add two registers, copy buffers, multiply vectors, blur images, draw triangles
- **Mechanisms for creating/destroying entities, expressing operations**
  - Execute machine instruction, make API call, express logic in a programming language

Notice the different levels of granularity/abstraction in my examples

Key course theme: choosing the right level of abstraction for system's needs

Decision impacts system's expressiveness/scope and potential for efficient implementation

Example: x86 architecture?

- State:
  - Maintained by execution context (registers, PC, VM mappings, etc.)
  - Contents of memory

- Operations:
  - x86 instructions (privileged and non-privileged)

The 3D rendering task

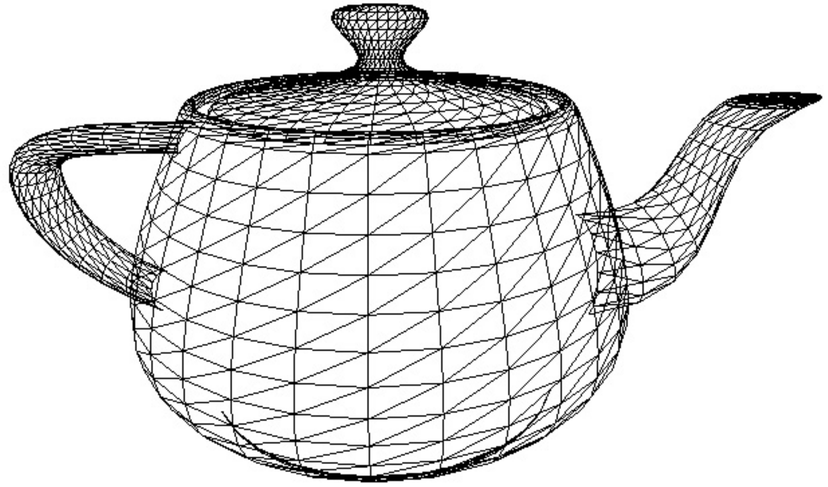


Image credit: Henrik Wann Jensen

Input: description of a scene

3D surface geometry (e.g., triangle meshes)
surface materials
lights
camera

Output: image

Problem statement: Determine how each geometric element contributes to the appearance of each output pixel in the image, given a description of a scene's surface properties and lighting conditions?

Goal: render high complexity 3D scenes, in real-time

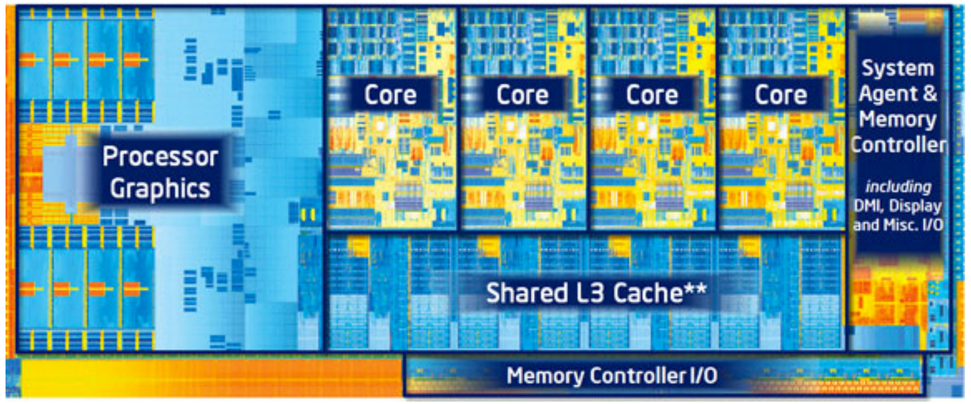
- 100's of thousands to millions of triangles in a scene
- Complex material, lighting, and animation computations
- High-resolution screen outputs (2-4 Mpixel + supersampling)
- 30-60 fps

Unreal Engine Kite Demo (Epic Games 2015)

Graphics pipeline implementation: GPUs

Specialized processors for executing graphics pipeline computations

Discrete GPU card
(NVIDIA GeForce Titan X)



Integrated GPU: part of modern Intel CPU die

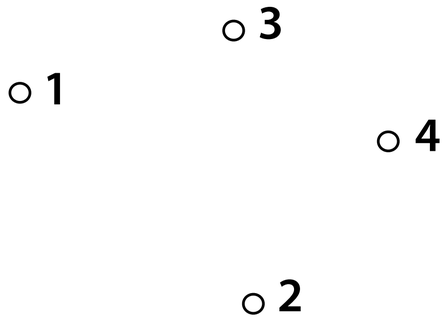
The real-time graphics pipeline architecture

(GPU-accelerated OpenGL/D3D graphics pipeline, from a systems perspective)

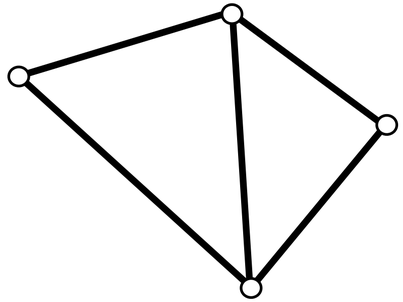
The graphics pipeline is an architecture for driving modern GPU execution

(Note to CUDA programmers: graphics pipeline was the original interface to GPU hardware. Compute mode execution came later…)

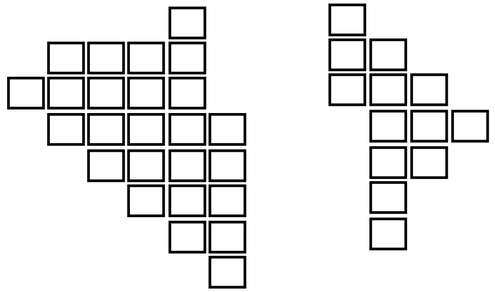
Real-time graphics pipeline entities



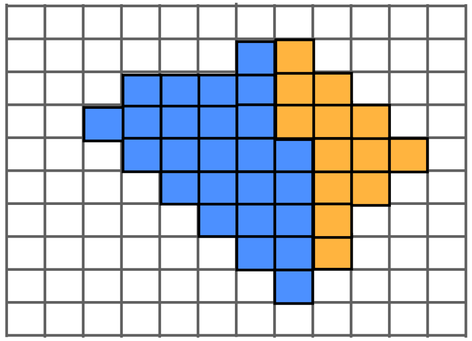
Vertices



Primitives
(triangles, points, lines)

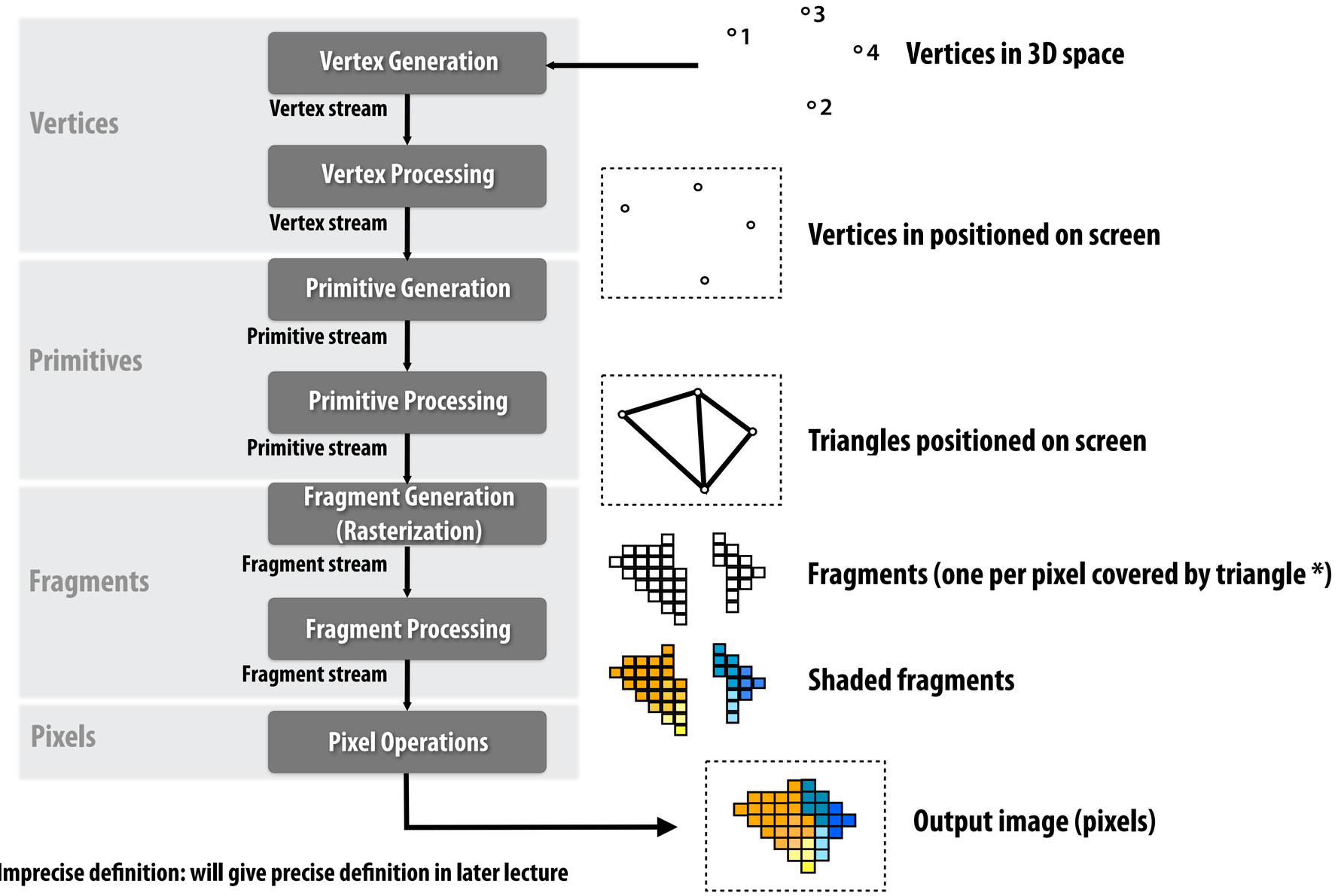


Fragments



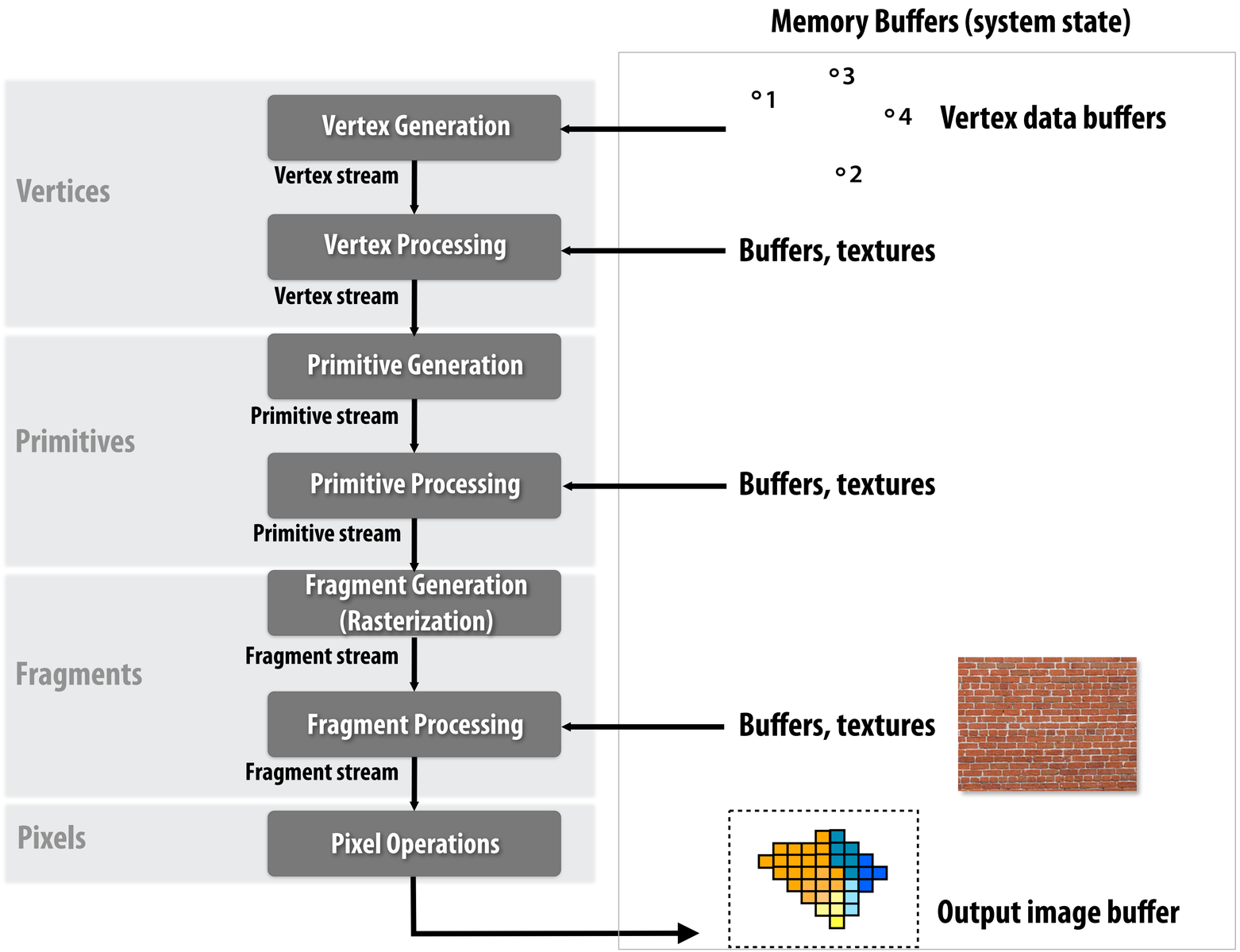
Pixels

Real-time graphics pipeline operations



* Imprecise definition: will give precise definition in later lecture

Real-time graphics pipeline state

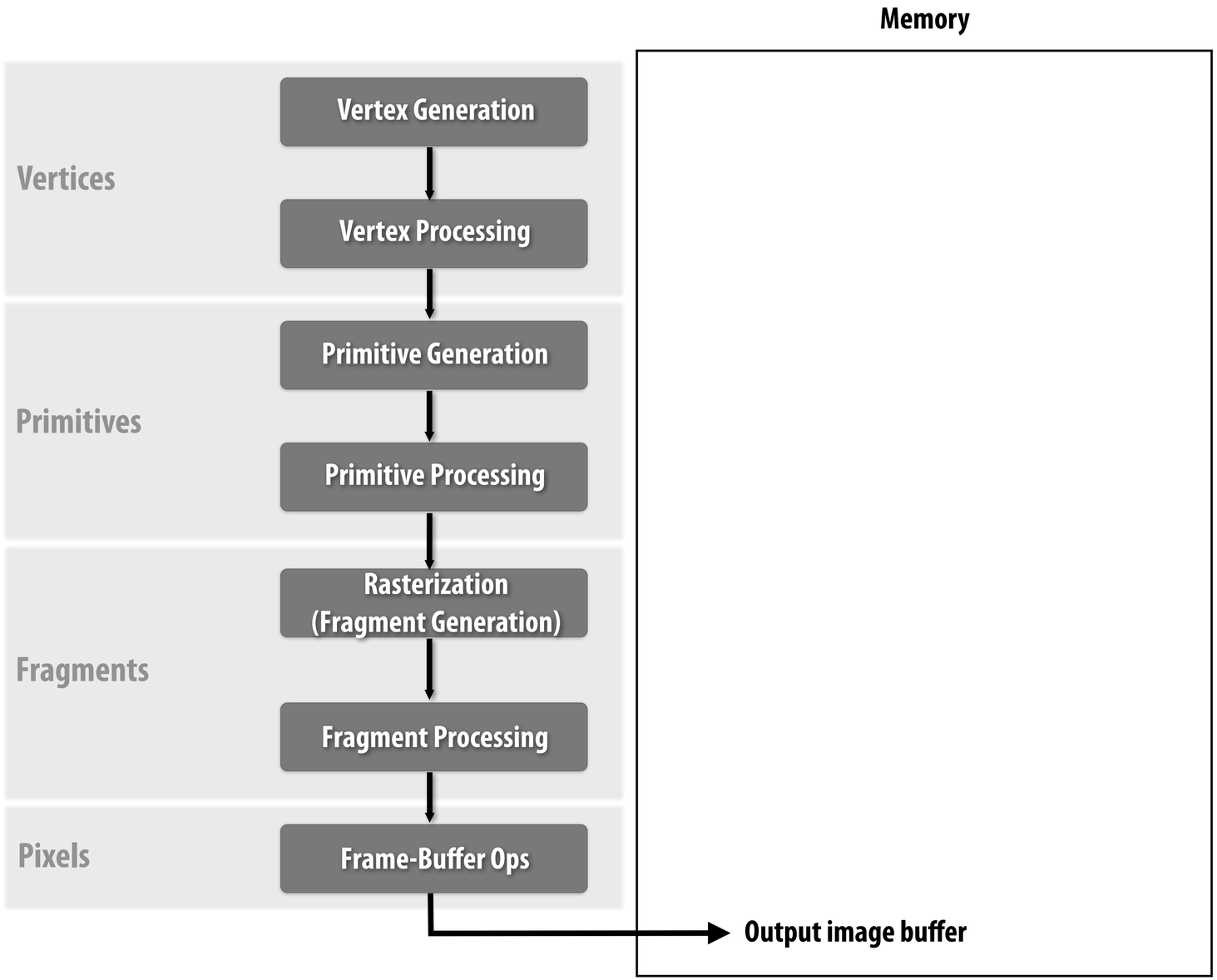


Issues to keep in mind during this overview*

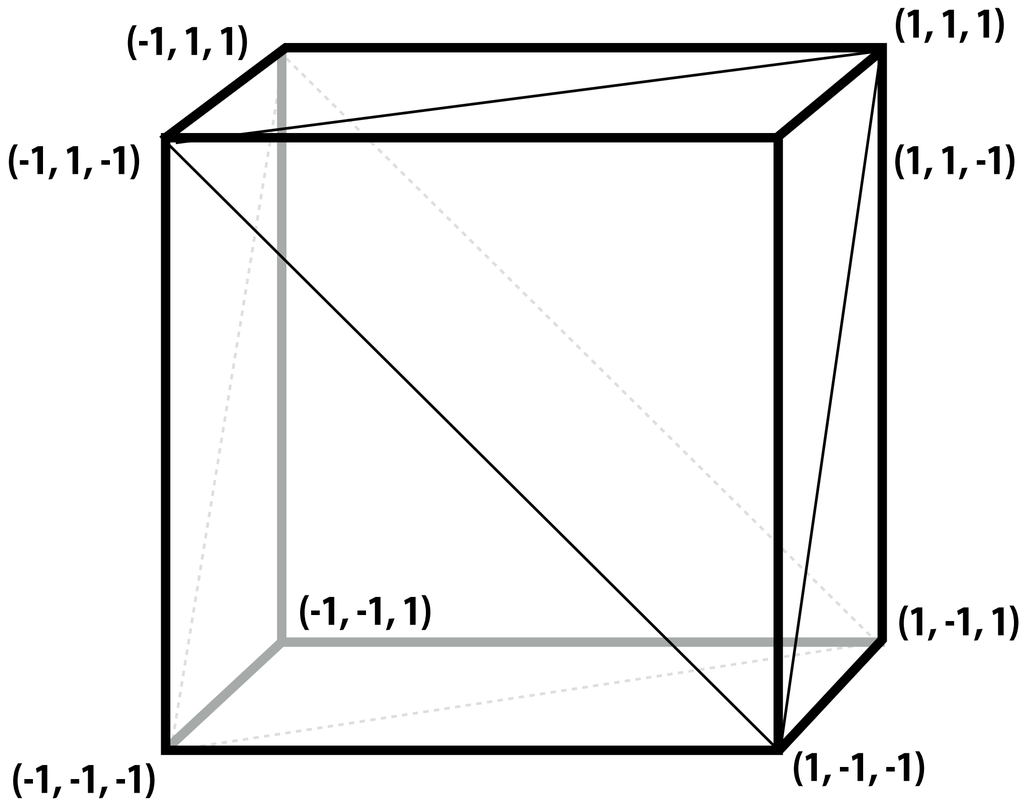
- Level of abstraction
- Orthogonality of abstractions
- How is the pipeline designed for performance/scalability?
- What the pipeline does and DOES NOT do

* These are great questions to ask yourself about any system you study

The graphics pipeline



Surface geometry representation: triangles

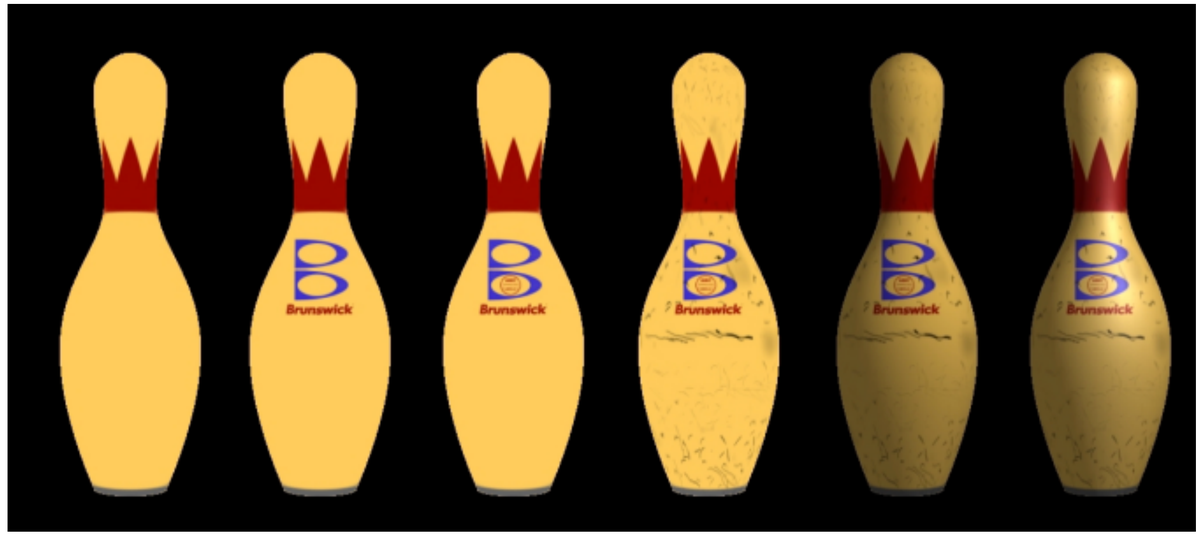


Texture mapping adds detail to surface

Pattern on ball

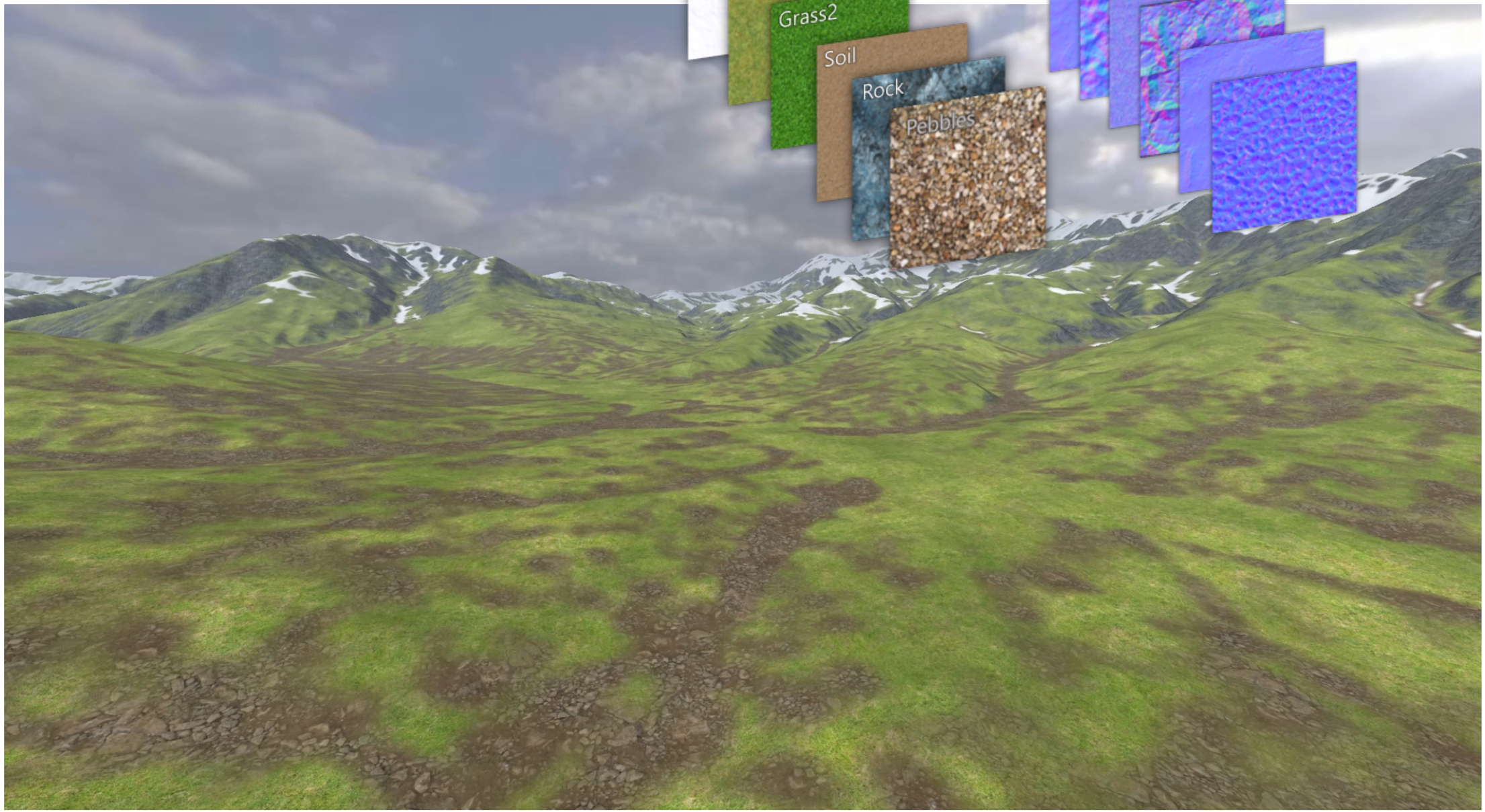
Wood grain on floor

Describe surface material properties



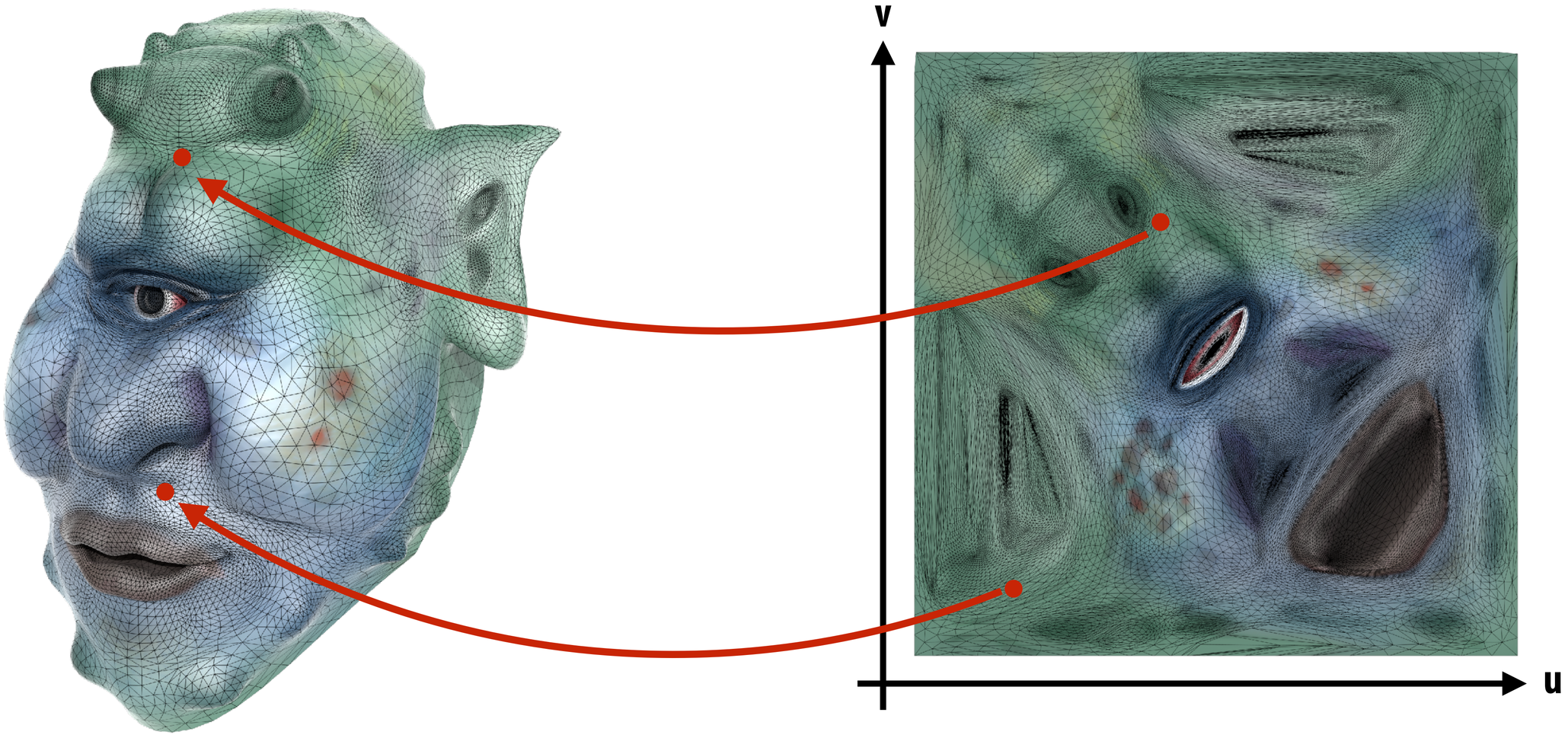
Multiple layers of texture maps for color, logos, scratches, etc.

Layered material



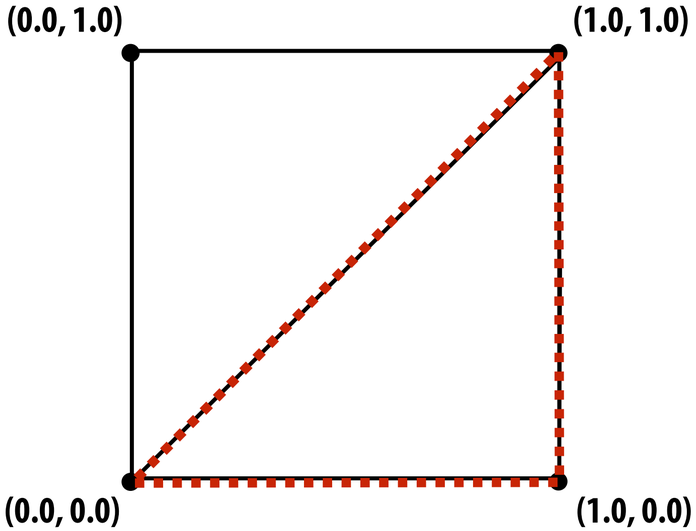
Texture mapping adds detail

Sample texture map at specified location in *texture coordinate space* to determine the surface's color at the corresponding point on surface.

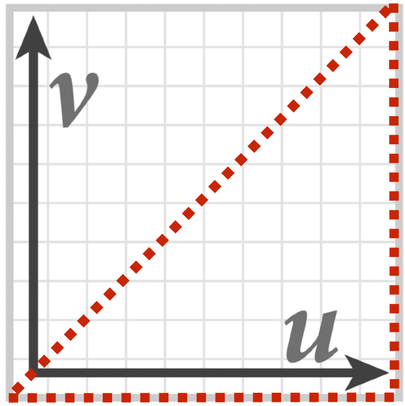


Representing surface detail: texture

"Texture coordinates" define a mapping from surface coordinates (points on triangle) to points in texture domain.

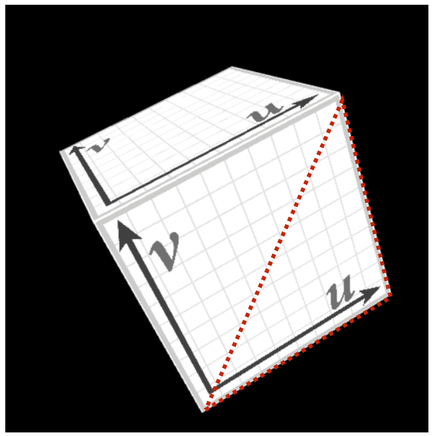


Two triangles (one face of cube) with surface parameterization provided as per-vertex texture coordinates.



`myTex(u,v)` is a function defined on the $[0,1]^2$ domain (represented by 2048x2048 image)

Location of highlighted triangle in texture space shown in red.



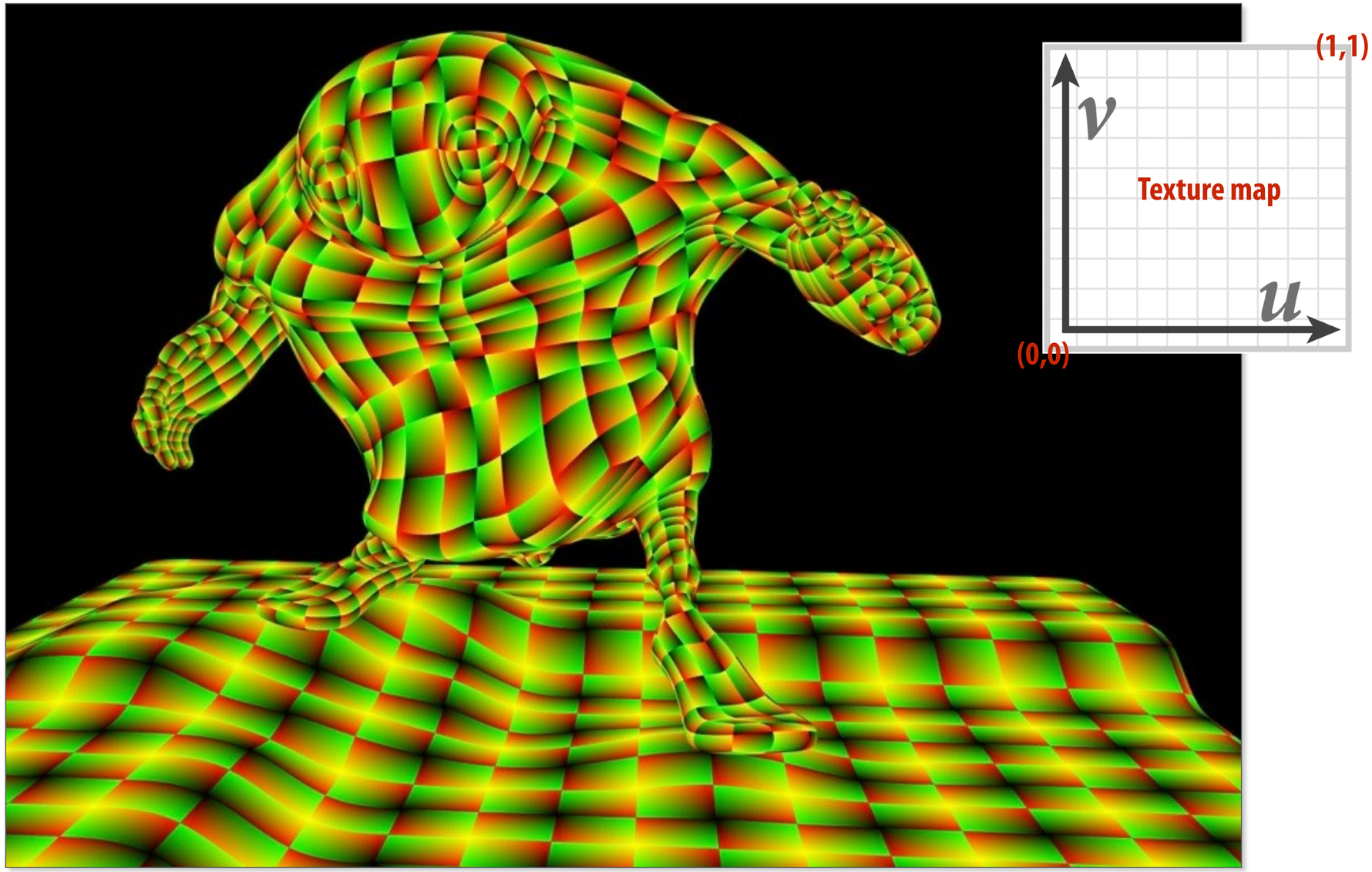
Final rendered result (entire cube shown).

Location of triangle after projection onto screen shown in red.

(We'll assume surface-to-texture space mapping is provided as per vertex values)

Texture coordinate visualization

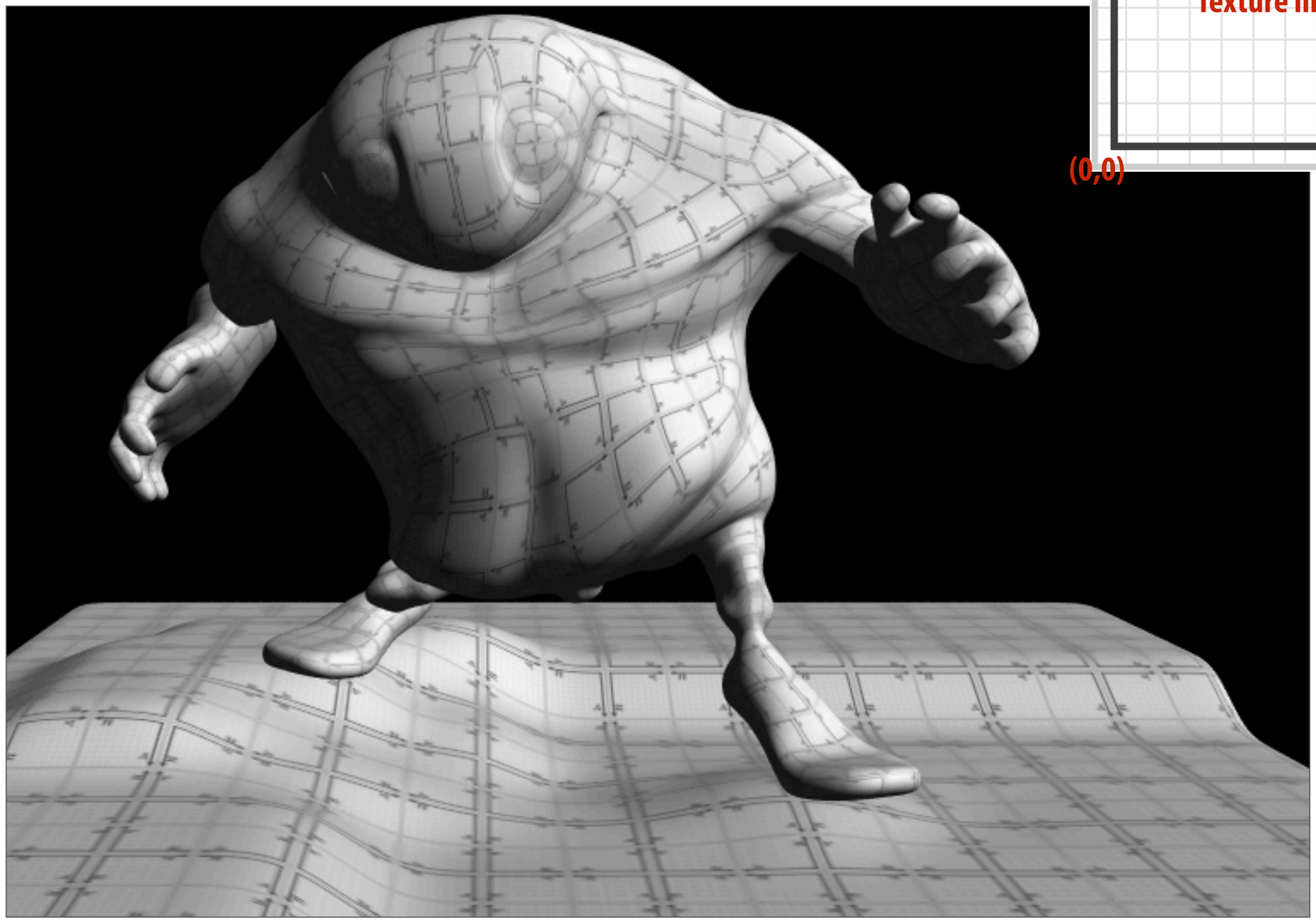
Defines mapping from point on surface to point (uv) in texture domain

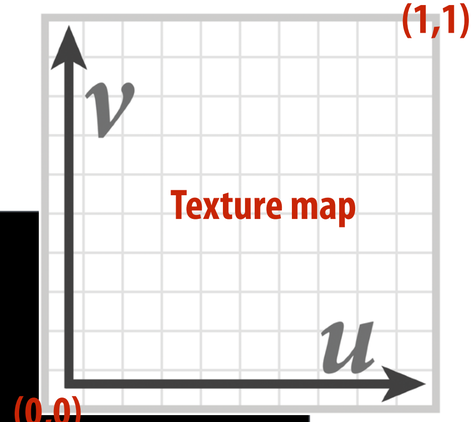


Red channel = u, Green channel = v

So uv=(0,0) is black, uv=(1,1) is yellow

Rendered result



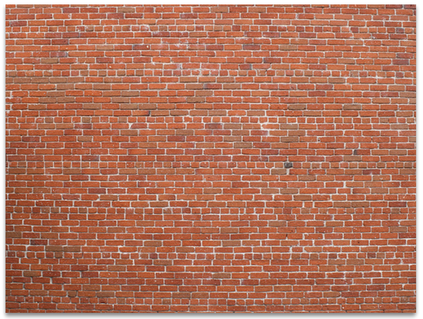


Command: draw these triangles!

Inputs:

```
list_of_positions = {
    v0x, v0y, v0z,
    v1x, v1y, v1x,
    v2x, v2y, v2z,
    v3x, v3y, v3x,
    v4x, v4y, v4z,
    v5x, v5y, v5x   };
```

```
list_of_texcoords = {
    v0u, v0v,
    v1u, v1v,
    v2u, v2v,
    v3u, v3v,
    v4u, v4v,
    v5u, v5v   };
```



Texture map

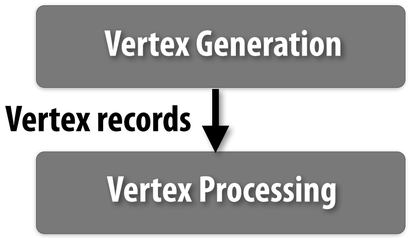
Object-to-camera-space transform: $\mathbf{T}$

Perspective projection transform $\mathbf{P}$

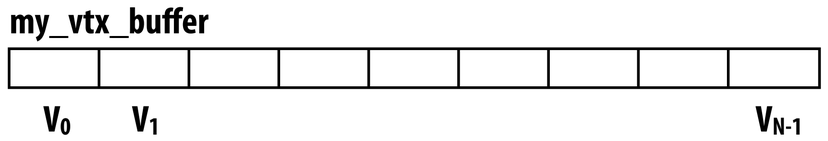
Size of output image (W, H)

Use depth test /update depth buffer: YES!

Constructing (“assembling”) vertices

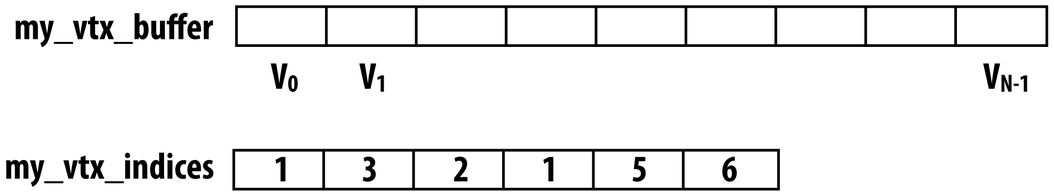


Contiguous version data version



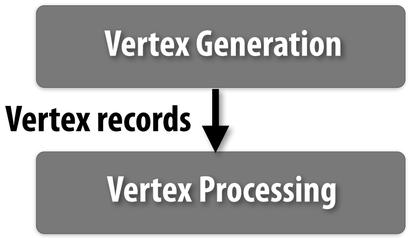

```
glBindBuffer(GL_ARRAY_BUFFER, my_vtx_buffer);
glDrawArrays(GL_TRIANGLES, 0, N);
```

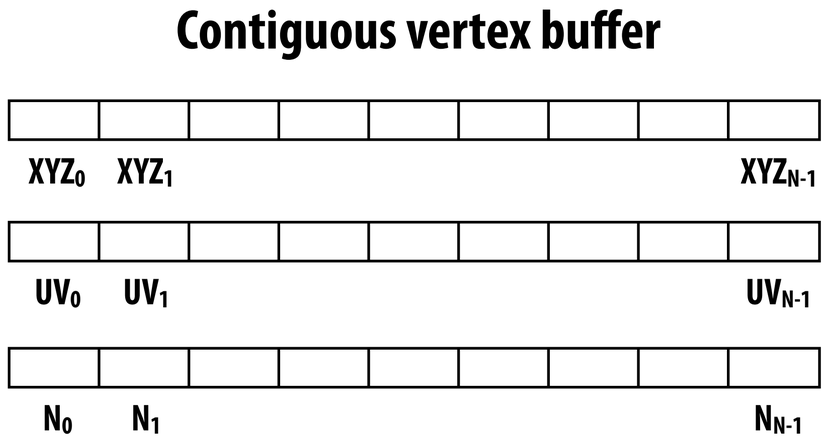
Indexed access version (“gather”)




```
glBindBuffer(GL_ARRAY_BUFFER, my_vtx_buffer);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT,
               my_vtx_indices);
```

Constructing (“assembling”) vertices



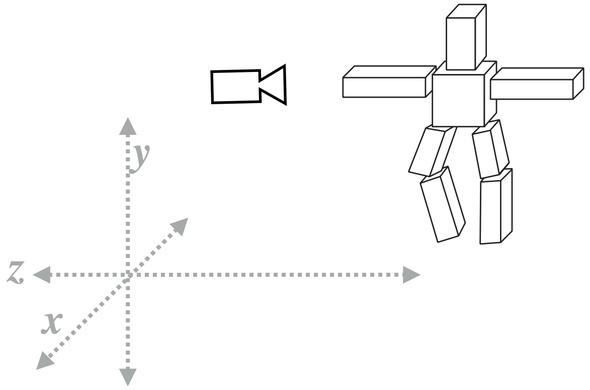


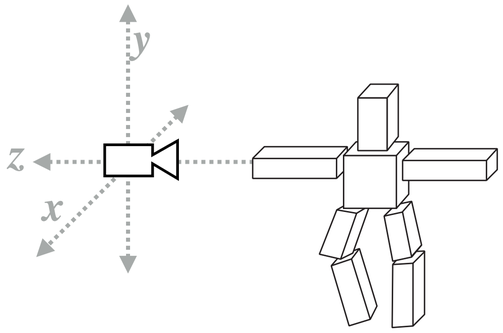
Output of vertex generation is a collection of vertex records.

What the vertex processing stage does

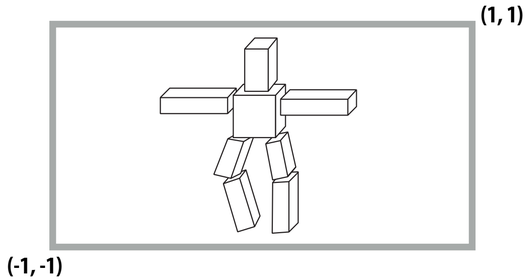
Objects and the camera in 3D world coordinates

Transform triangle vertices from their original coordinates into camera space coordinates



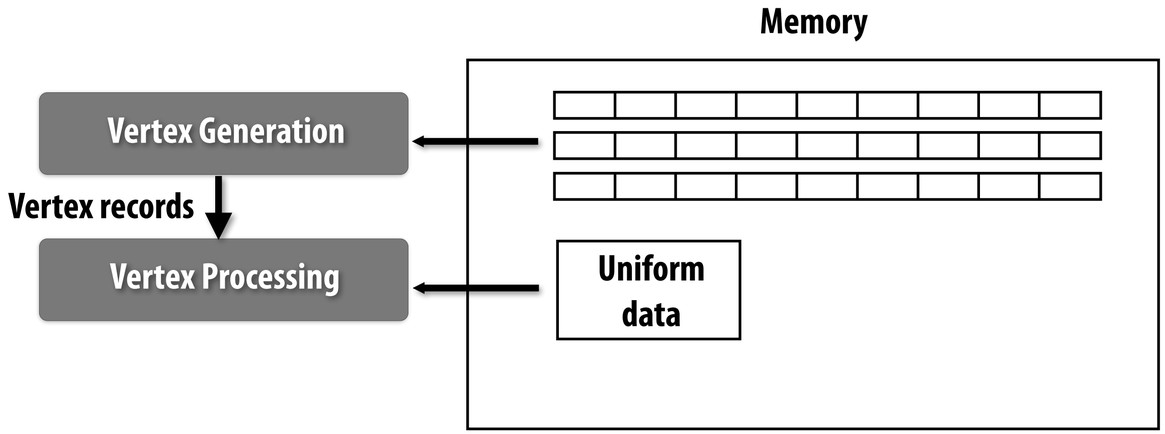


Position of objects is now relative to location of camera



Project objects onto normalized 2D screen coordinates

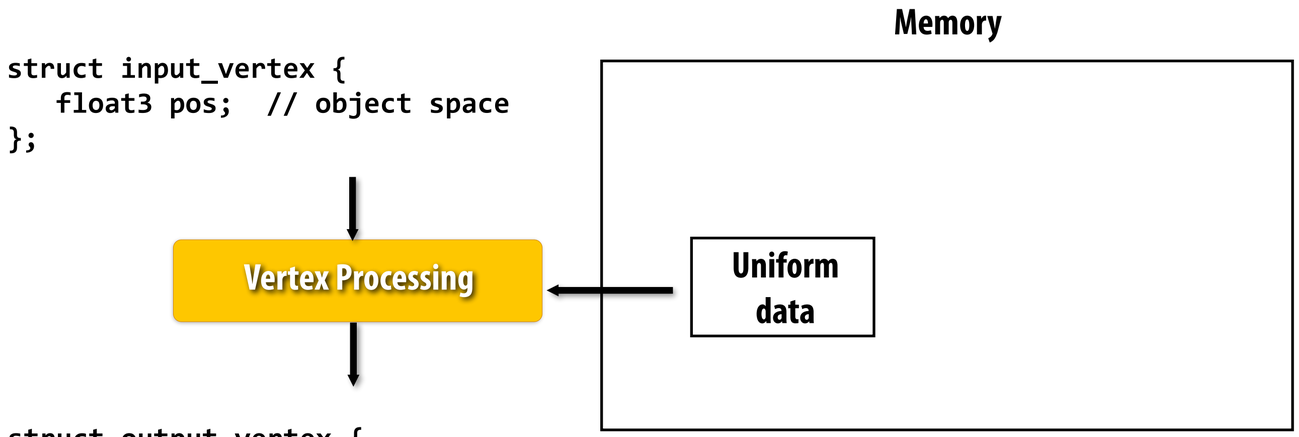
Vertex processing: inputs



Uniform data: constant read-only data provided as input to every instance of the vertex shader
e.g., object-to-clip-space vertex transform matrix

Vertex processing operates on a stream of vertex records + read-only "uniform" inputs.

Vertex processing: inputs and outputs



1 input vertex ⟶ 1 output vertex

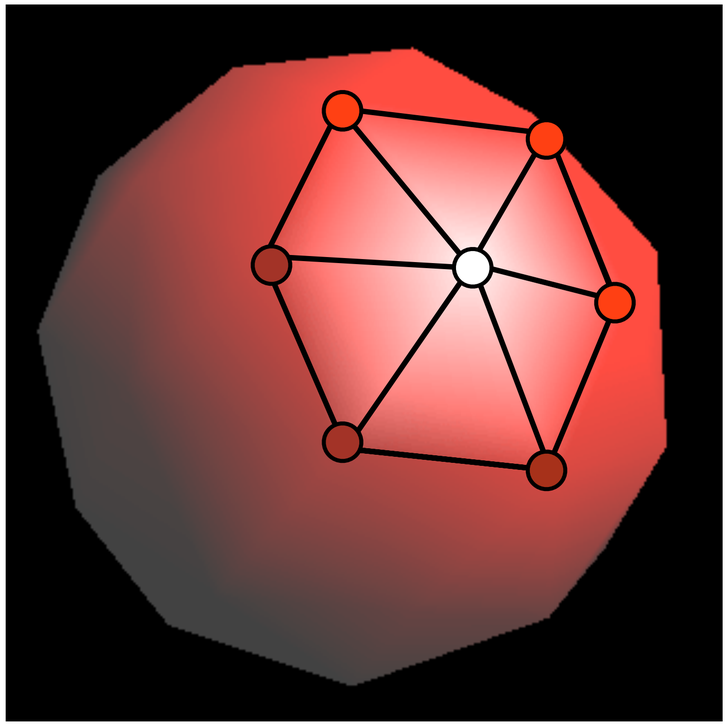
independent processing of each vertex

Vertex Shader Program *

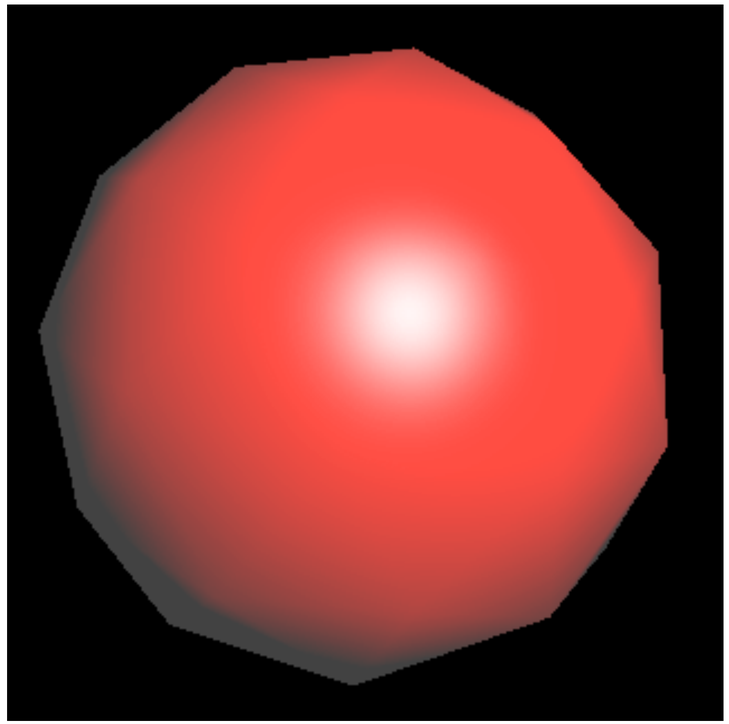
```
uniform mat4 my_transform;   // P * T

output_vertex my_vertex_program(input_vertex in) {
    output_vertex out;
    out.pos = my_transform * in.pos; // matrix-vector mult
    return out;
}
```

(* Note: this is pseudocode, not valid GLSL syntax)

Another per-vertex computation: lighting



Per-vertex lighting computation

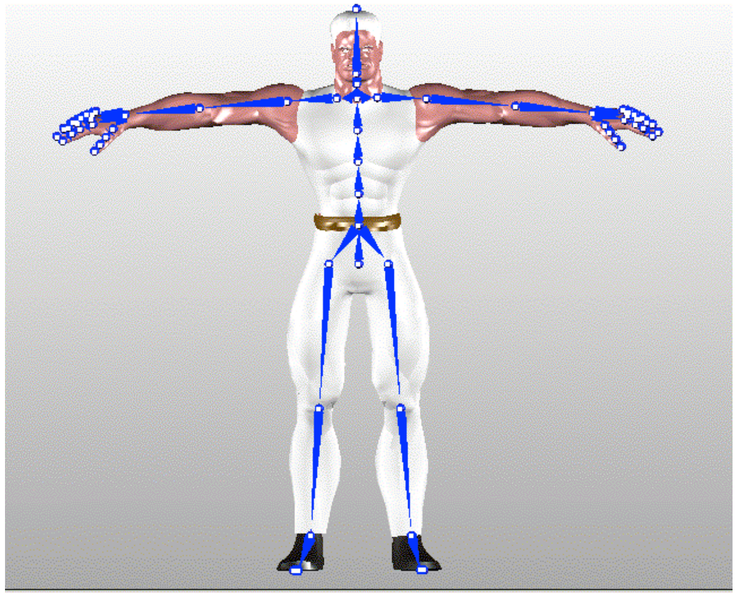


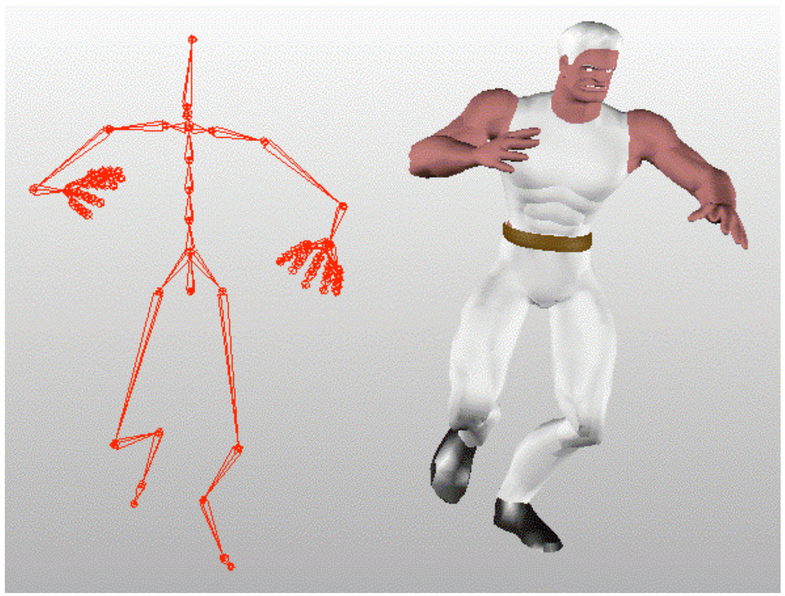
Per-vertex normal computation, per pixel lighting

Input per-vertex data: surface normal, surface color

Input uniform data: light direction, light color

Another per-vertex computation: skeletal animation via "skinning"





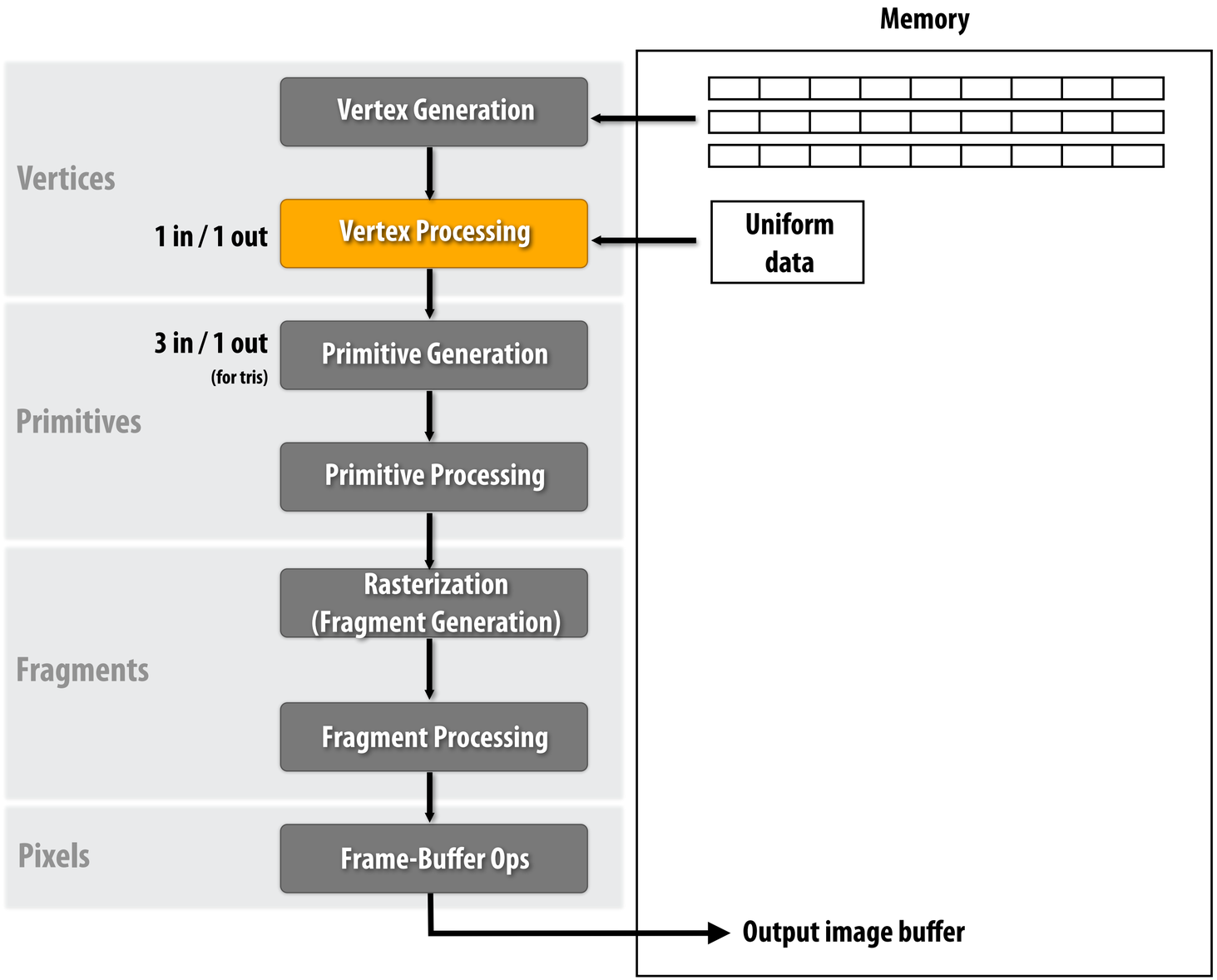
$$V_{skinned} = \sum_{b \in bones} w_b M_b V_{base}$$

Input per-vertex data: base vertex position (V_{base}) + blend coefficients (w_b)

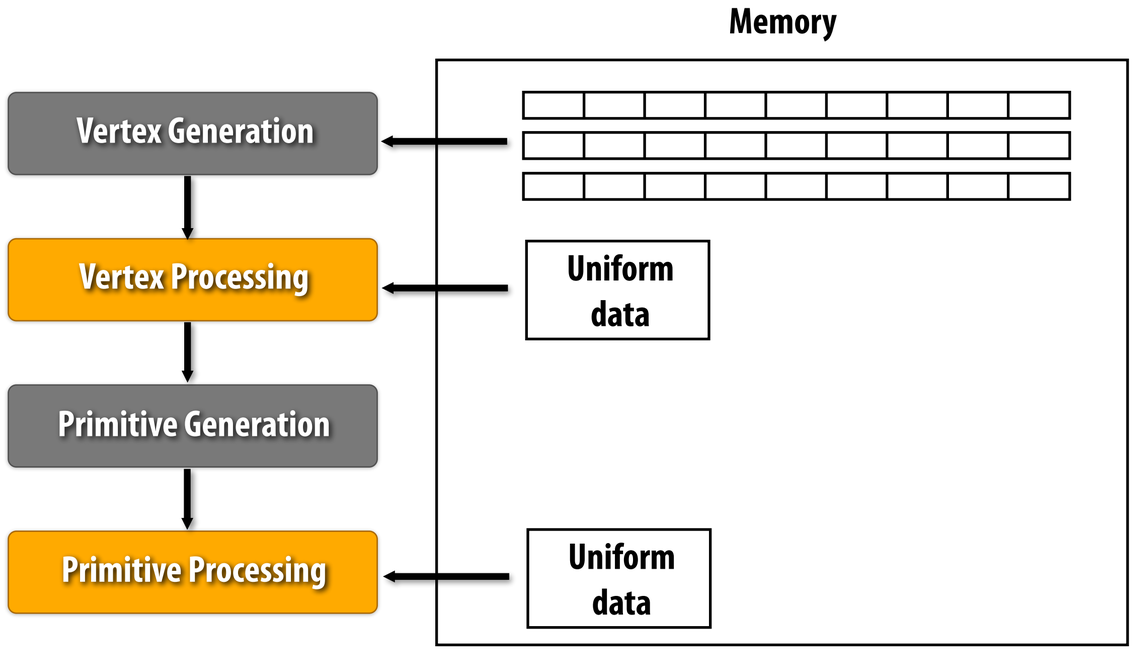
Input: uniform data: "bone" matrices (M_b) for current animation frame

Image credit: http://www.okino.com/conv/skinning.htm

Primitive generation: group vertices into primitives



Programmable primitive processing *



input vertices for 1 prim ⟶ output vertices for N prims **

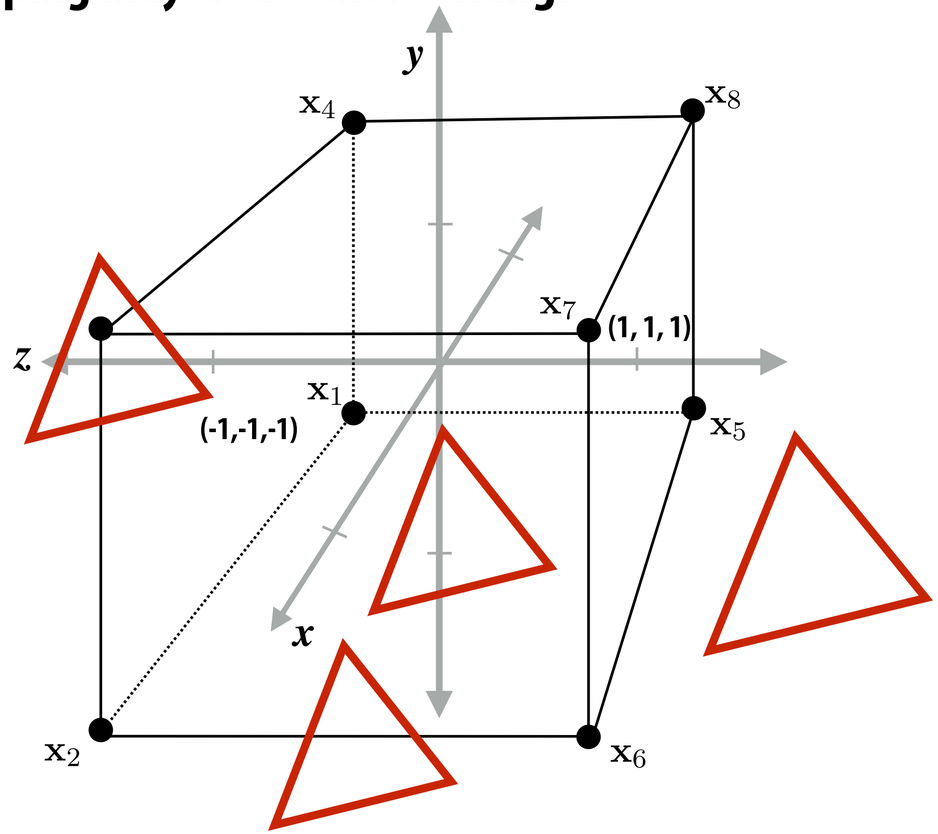
independent processing of each INPUT primitive

* "Geometry shader" in OpenGL/Direct3D terminology

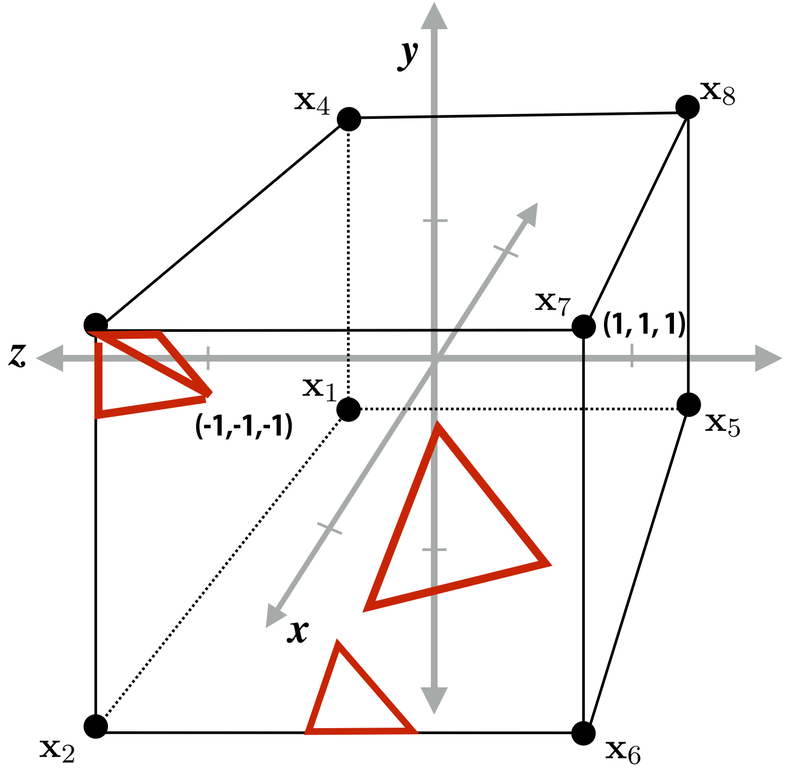
** Pipeline caps output at 1024 floats of output

Primitive processing: clipping

- Discard triangles that lie complete outside the unit cube (culling)
  - They are off screen, don't bother processing them further
- Clip triangles that extend beyond the unit cube to the cube
  - Note: clipping may create more triangles



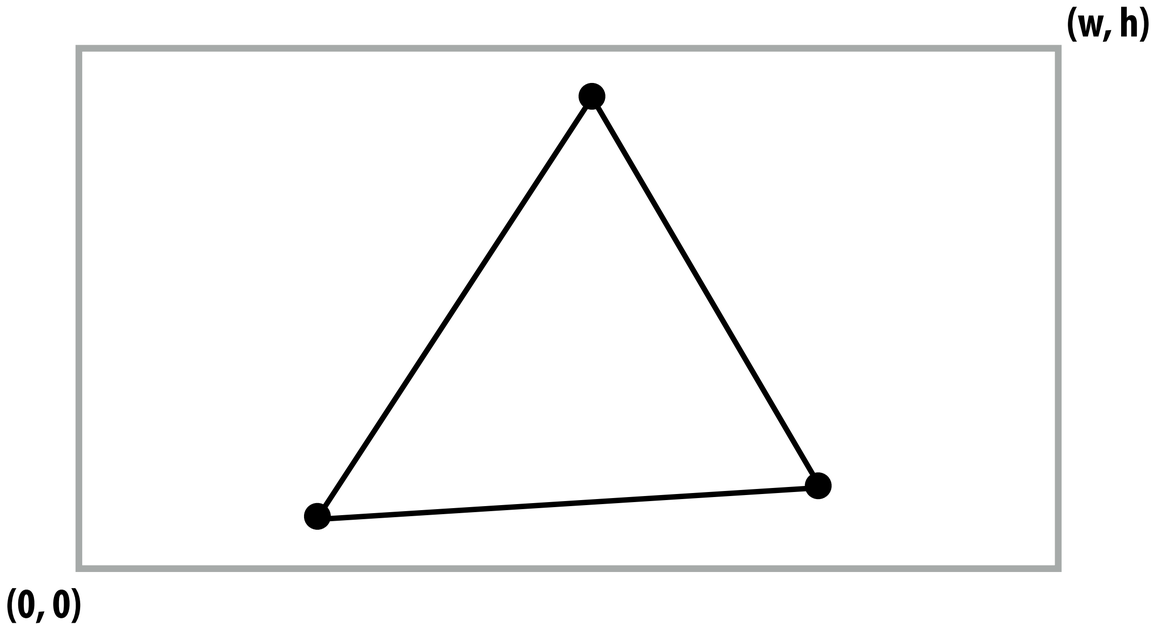
Triangles before clipping



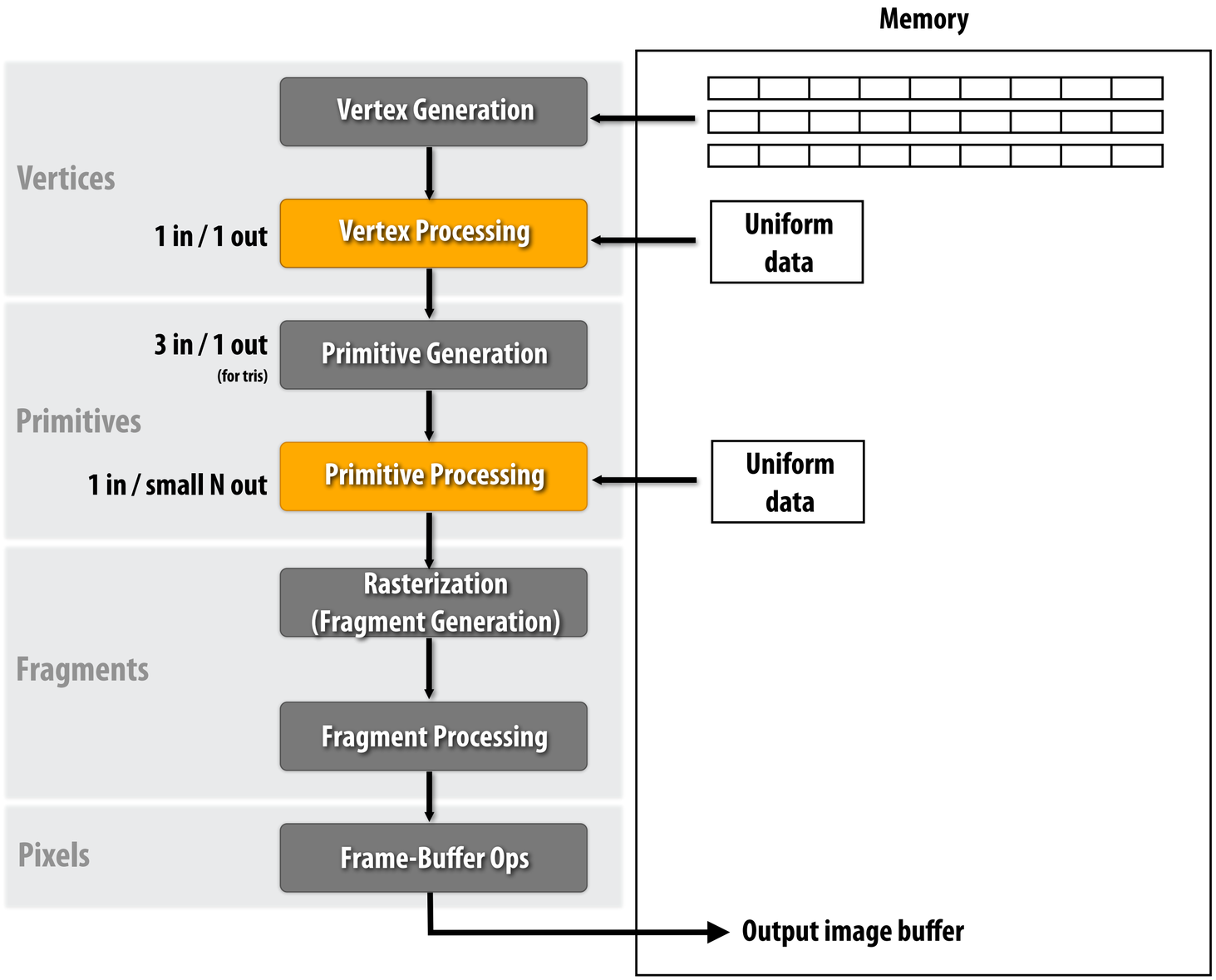
Triangles after clipping

Transform to screen coordinates

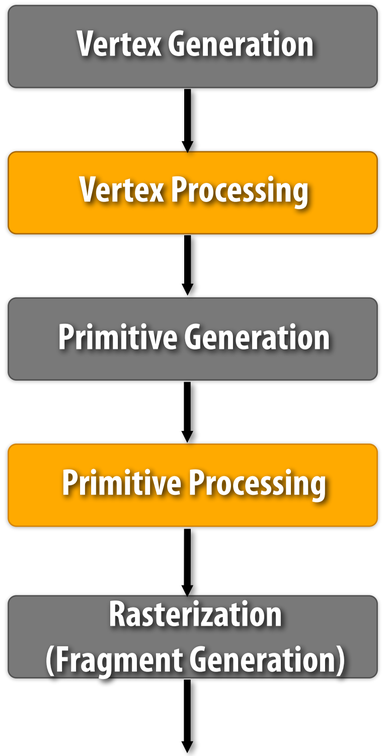
Transform vertex xy positions from normalized coordinates into screen coordinates (based on screen w,h)



The graphics pipeline

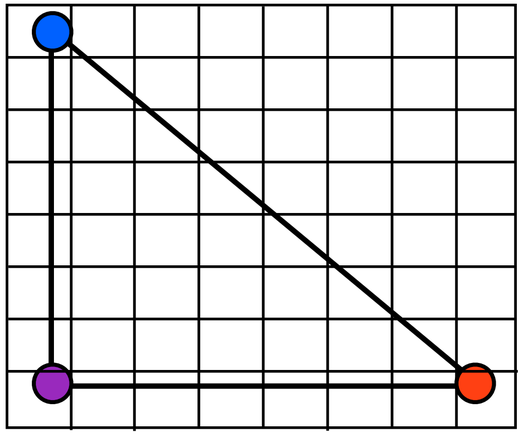


Rasterization (fragment generation)



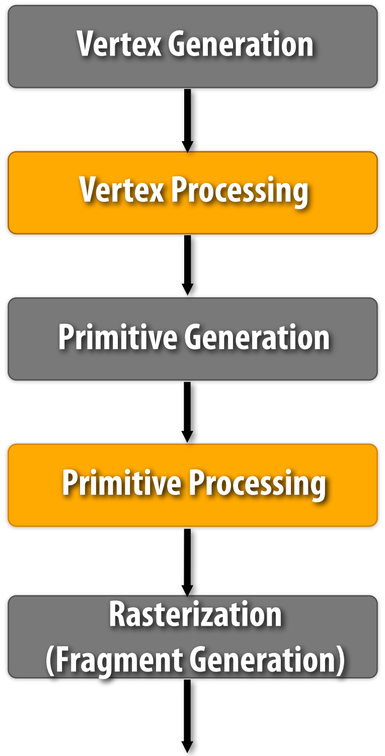
1 input prim ⟶ N output fragments

N is unbounded
(size of triangles varies greatly)



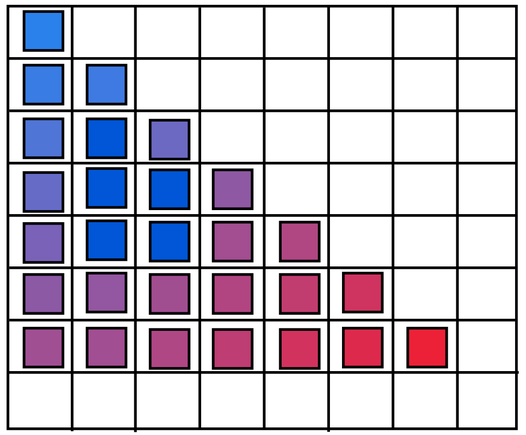
```
struct fragment      // note similarity to output_vertex from before
{
   float  x,y;       // screen pixel coordinates (sample point location)
   float  z;         // depth of triangle at sample point

   float3 normal;    // interpolated application-defined attribs
   float2 texcoord;  // (e.g., texture coordinates, surface normal)
};
```

Rasterization (fragment generation)



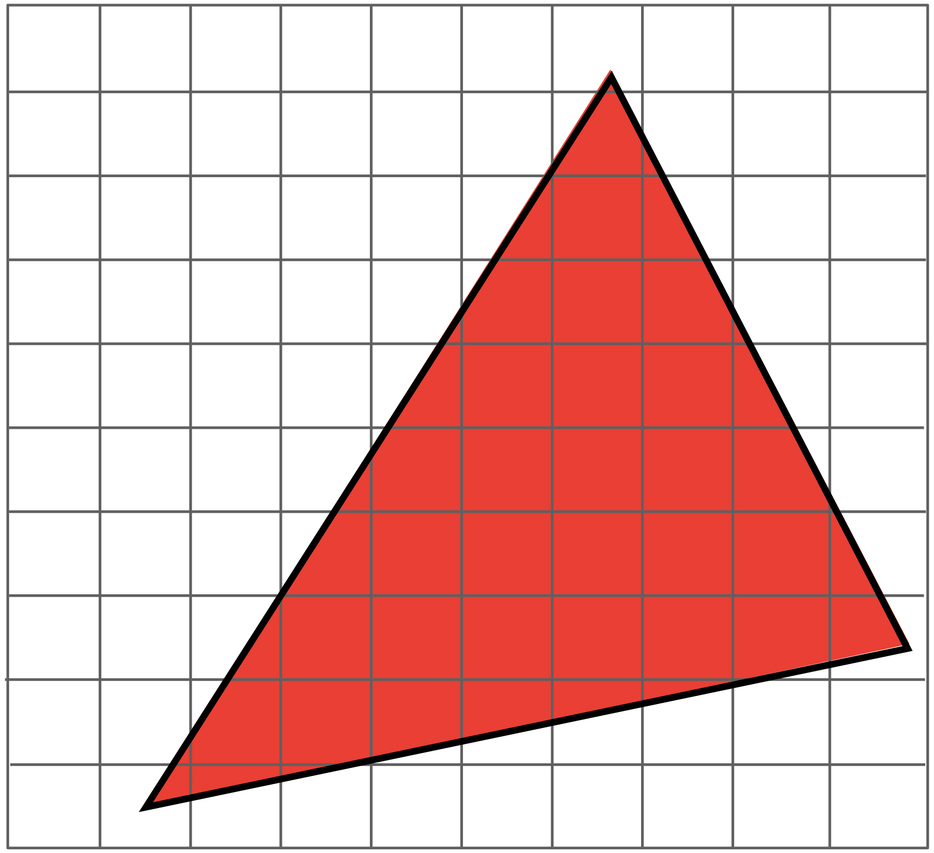
Compute covered pixels

Sample vertex attributes once per covered pixel

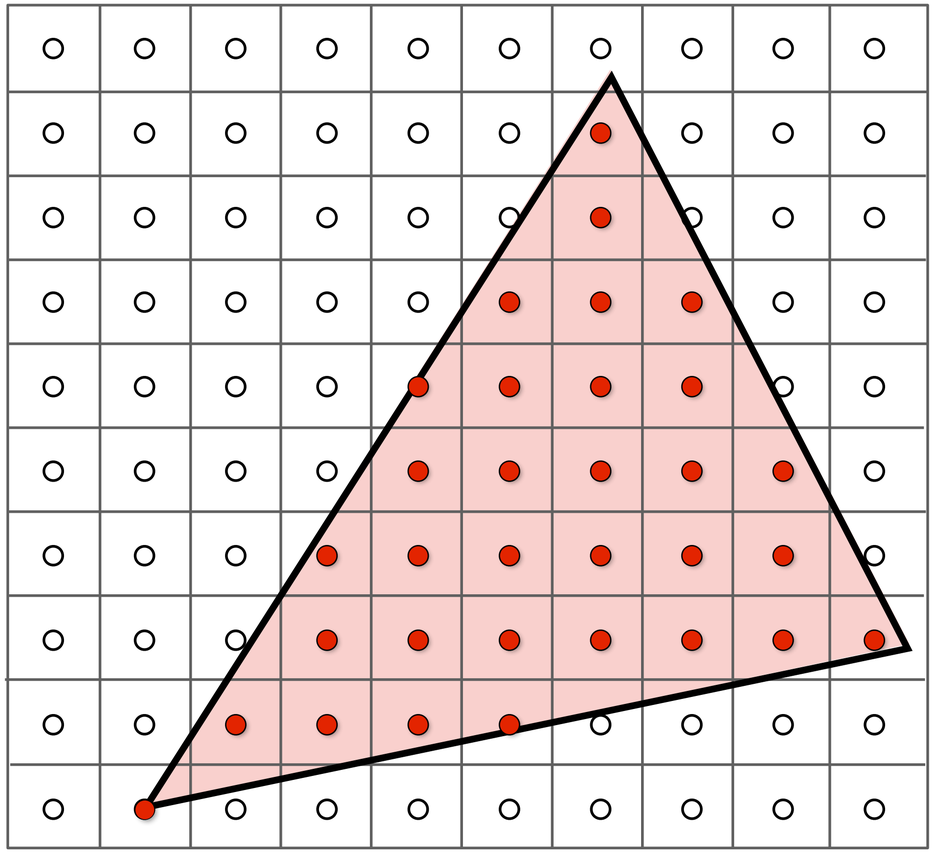


```
struct fragment      // note similarity to output_vertex from before
{
   float  x,y;       // screen pixel coordinates (sample point location)
   float  z;         // depth of triangle at sample point

   float3 normal;    // interpolated application-defined attribs
   float2 texcoord;  // (e.g., texture coordinates, surface normal)

}
```

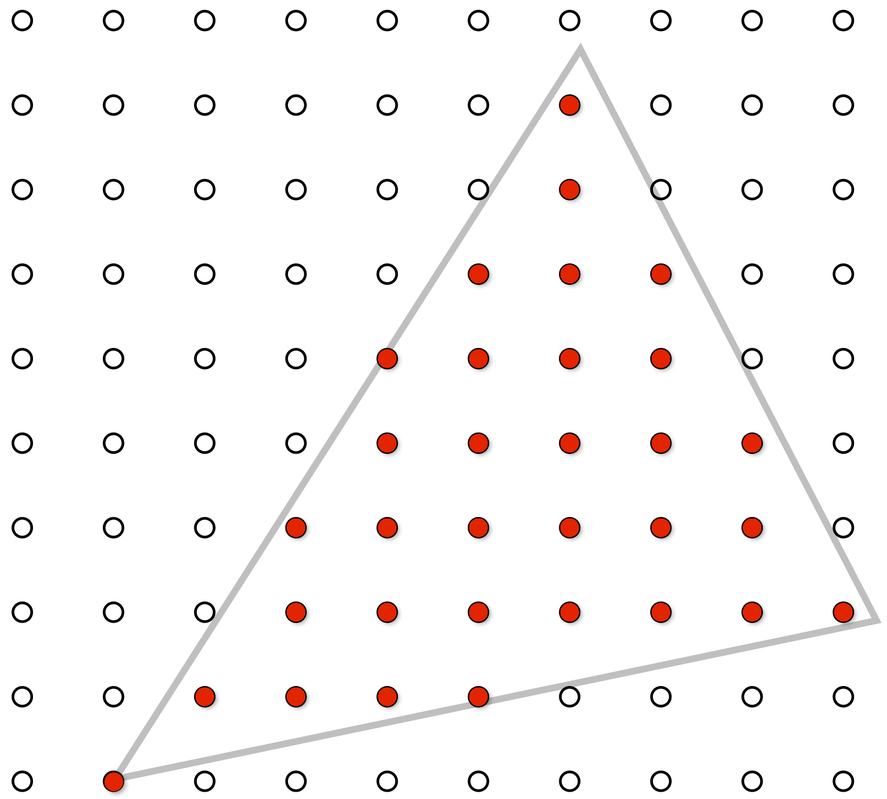
Drawing a triangle by 2D sampling



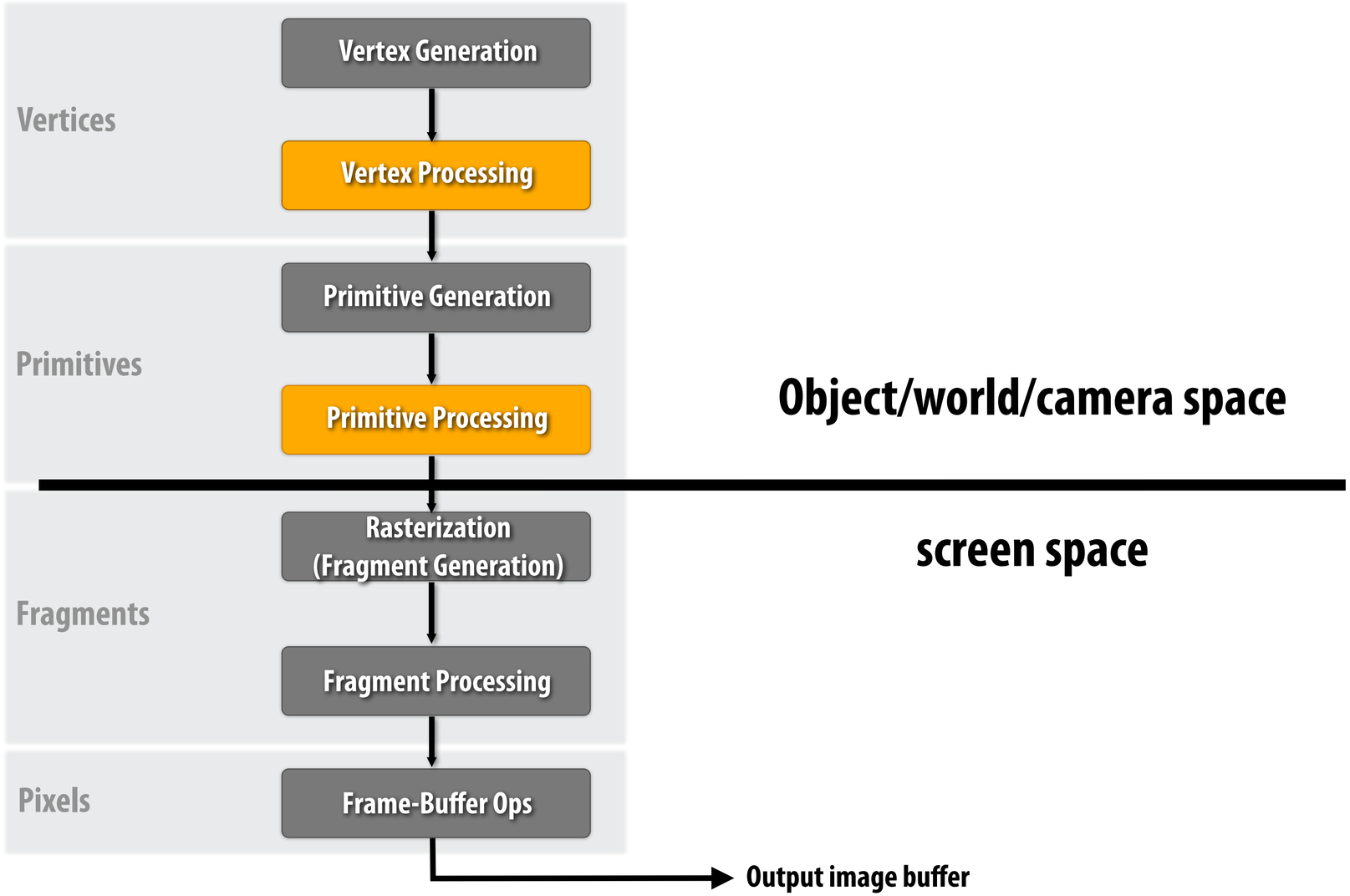
Sample coverage at pixel centers



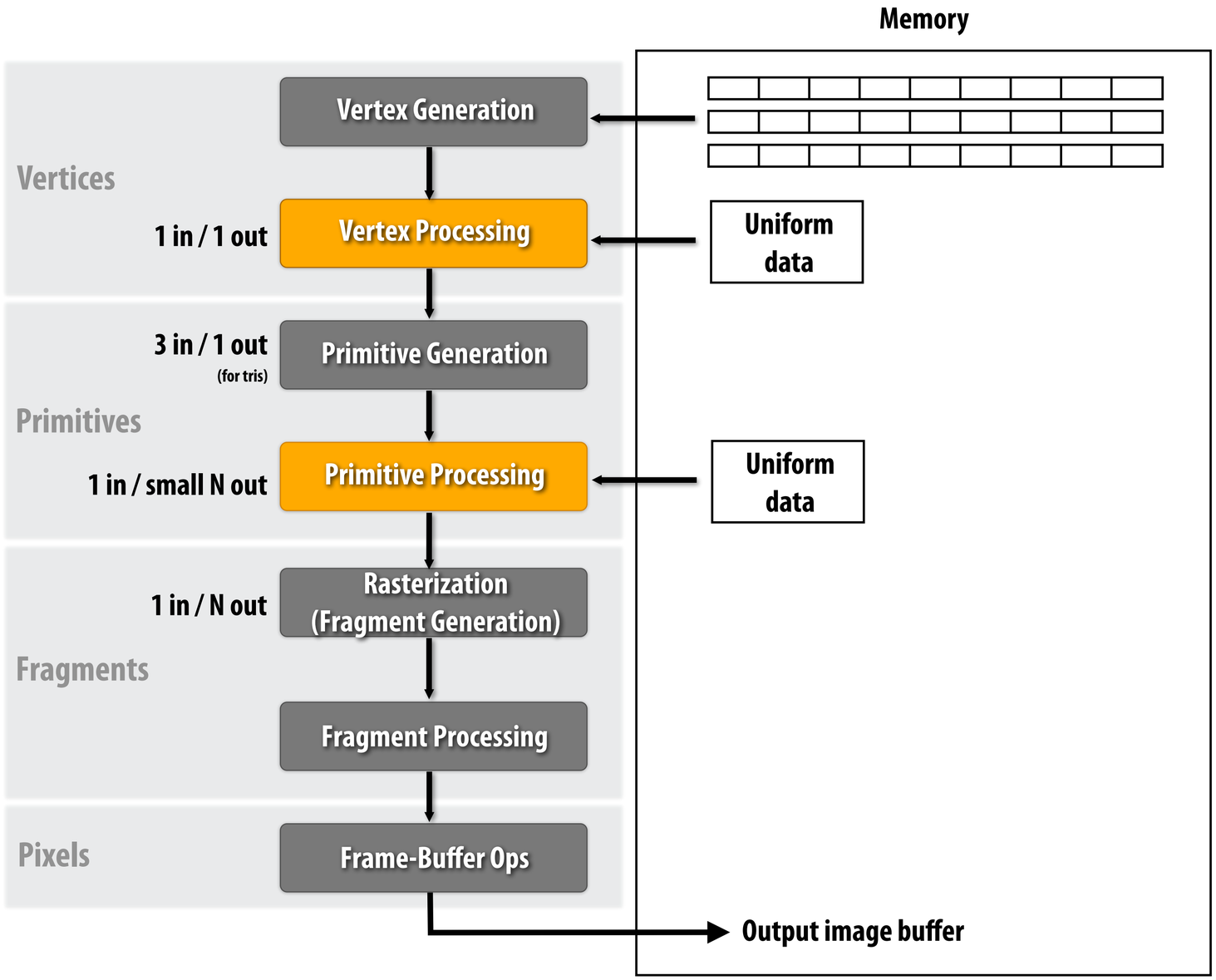
Sample coverage at pixel centers



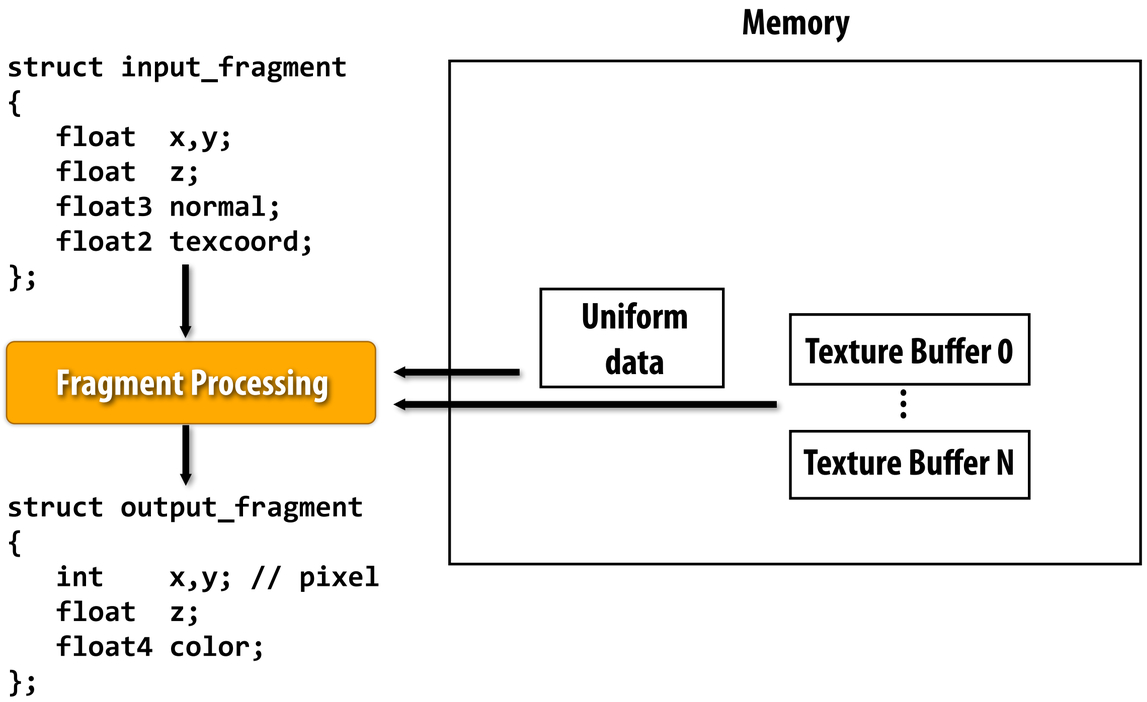
The graphics pipeline



The graphics pipeline



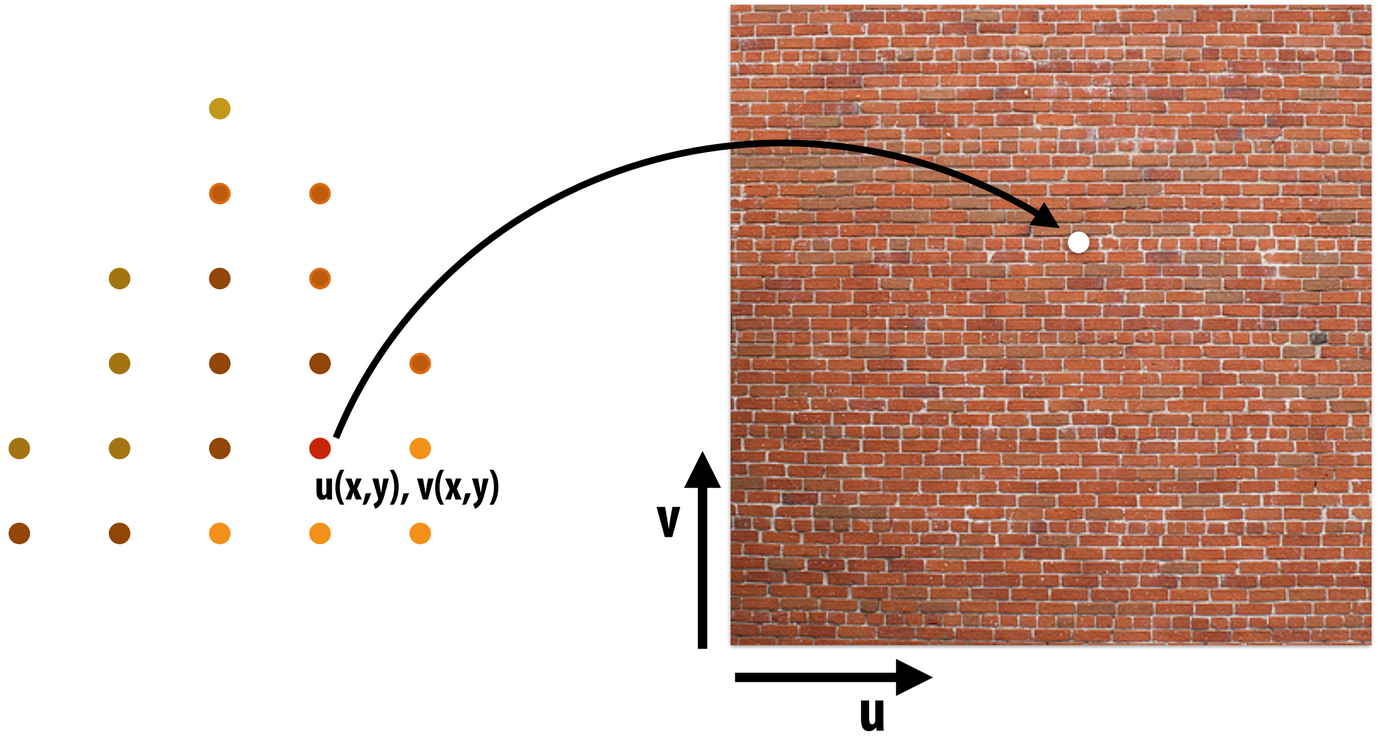
Fragment processing



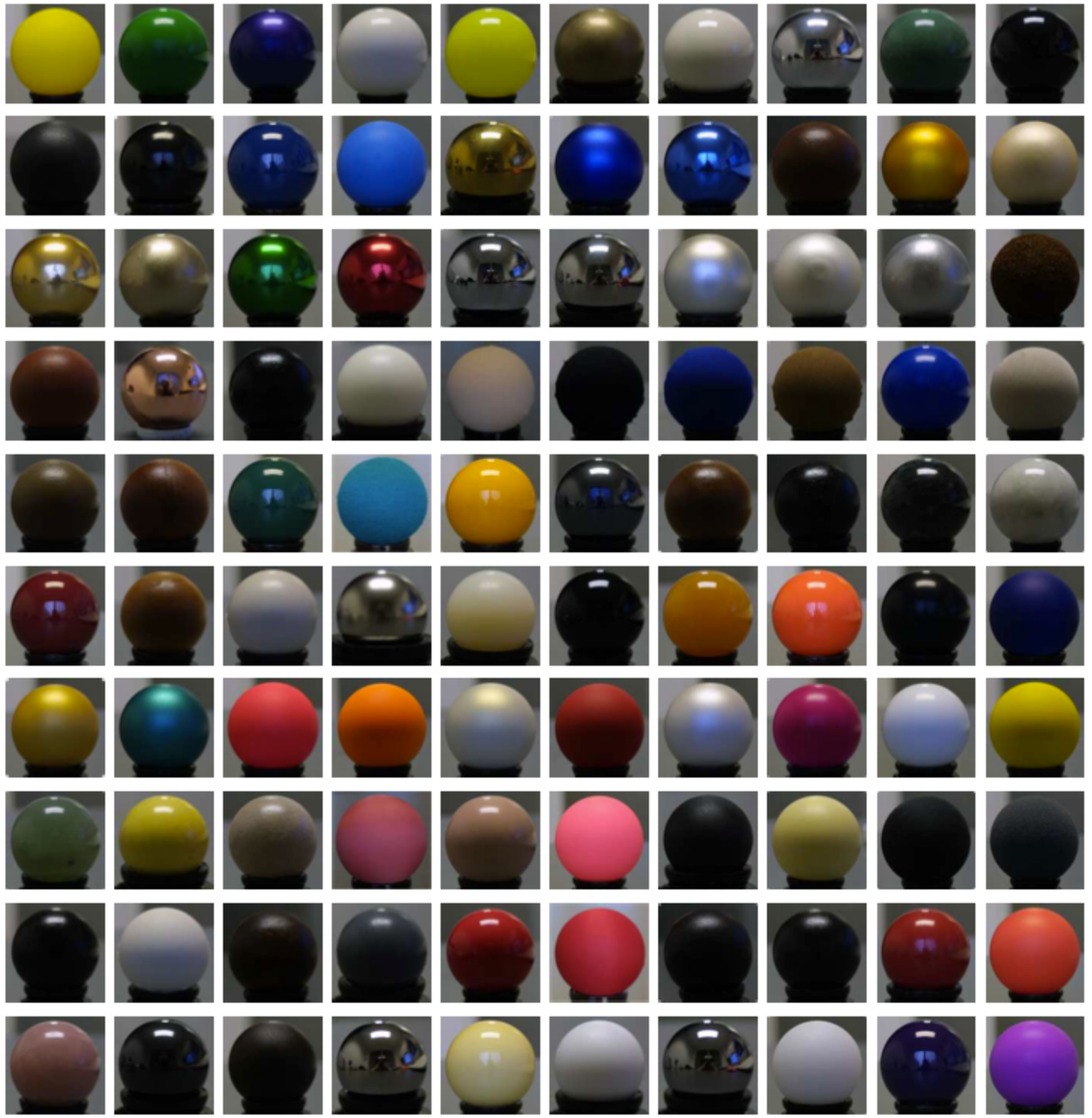

```
texture my_texture;

output_fragment my_fragment_program(input_fragment in)
{
    output_fragment out;
    float4 material_color = sample(my_texture, in.texcoord);

    for (each light L in scene)
    {
        out.color += shade(L) // compute reflectance towards camera due to L
    }
    return out;
}
```

Example per-fragment operation: computing fragment color

e.g., sample texture map



Many different materials in the world



Tabulated BRDFs

Images from Matusik et al. SIGGRAPH 2003

Materials

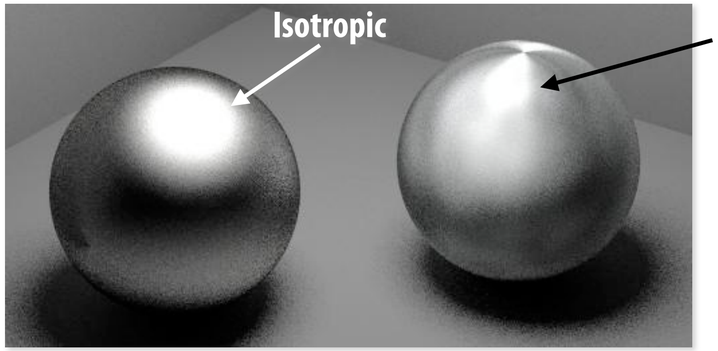
[Image credit: Jakob et al. 2014]

More complex materials

[Images from Lafortune et al. 97]

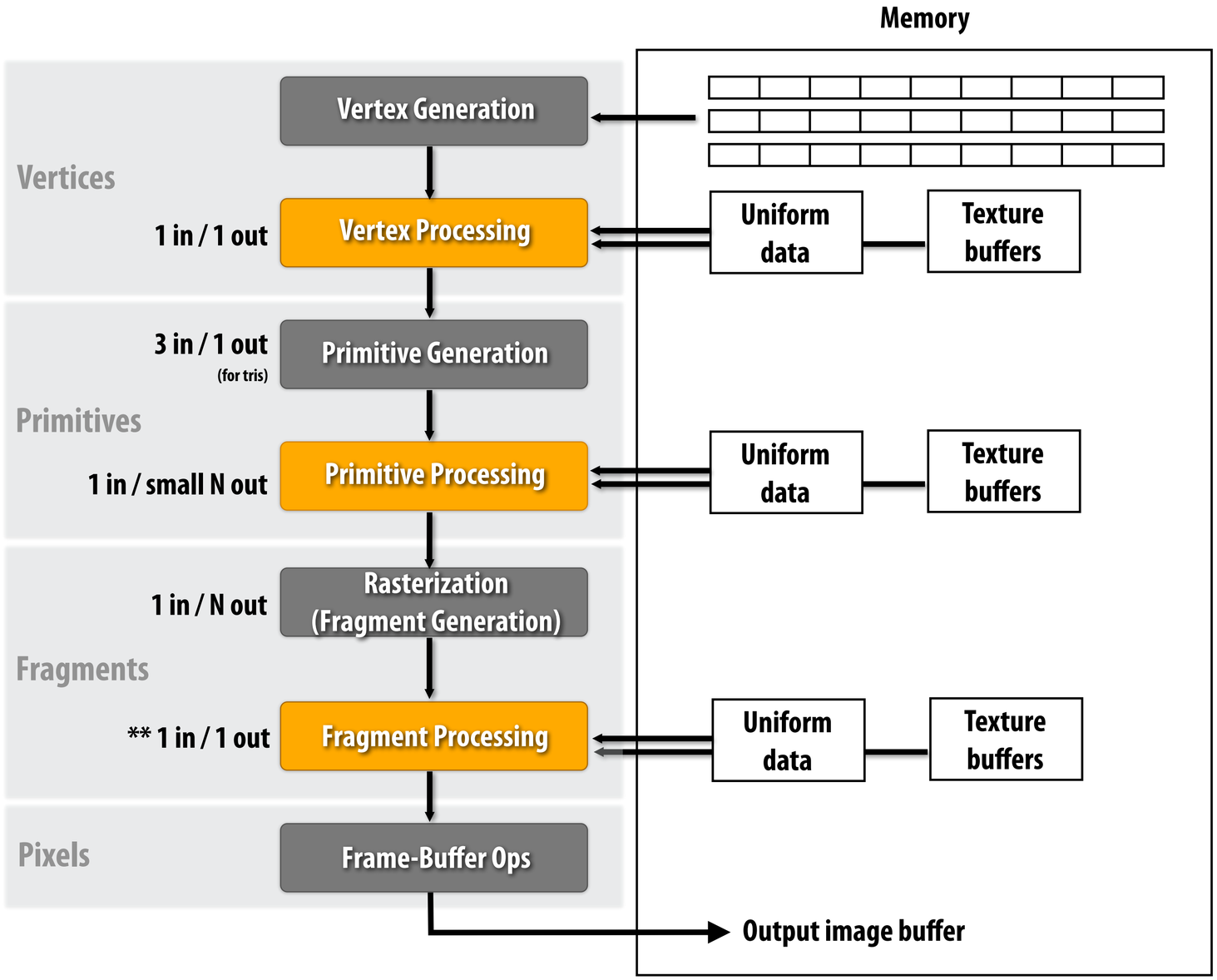
Fresnel reflection: reflectance is a function of viewing angle (notice higher reflectance near grazing angles)

[Images from Westin et al. 92]



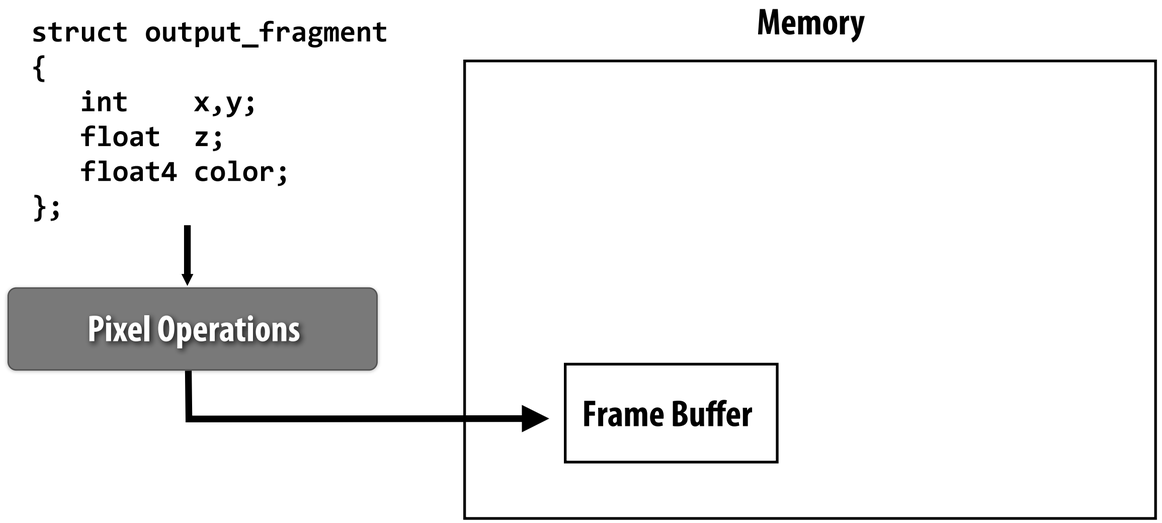
Anisotropic reflection: reflectance depends on azimuthal angle (e.g., oriented microfacets in brushed steel)

The graphics pipeline



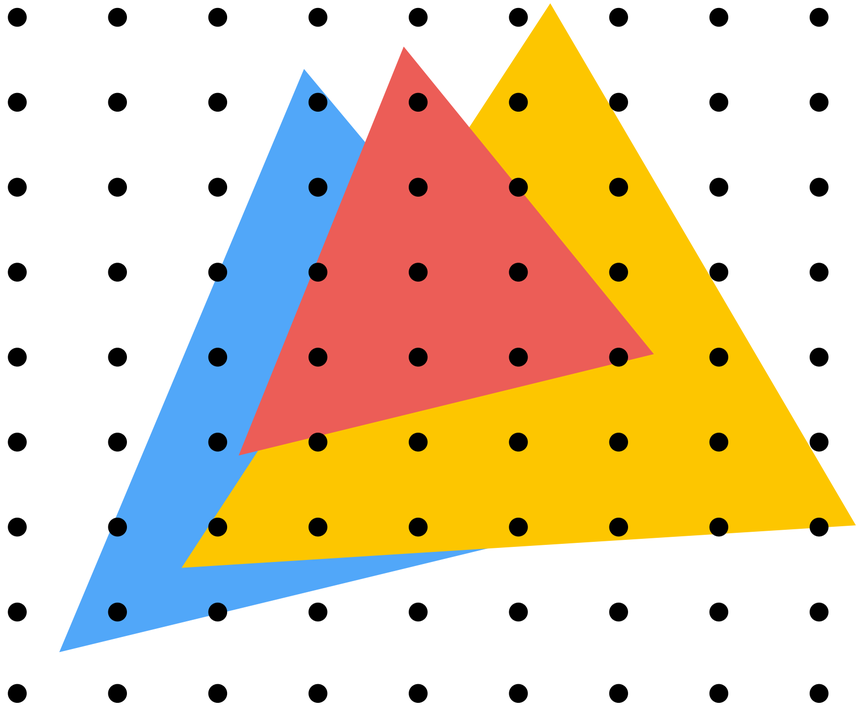
** can be 0 out

Frame-buffer operations



- Key responsibilities:
  - Accumulate/blend fragment color into frame buffer based on “depth test”

Occlusion: which triangle is visible at each covered sample point?

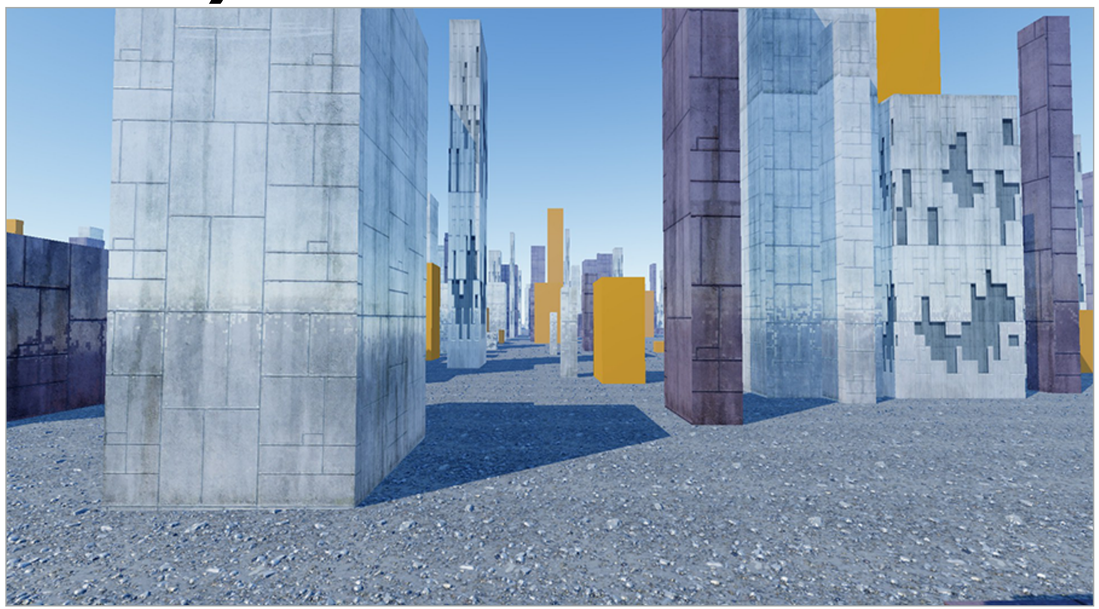


Opaque Triangles

Depth buffer (aka “Z buffer”)

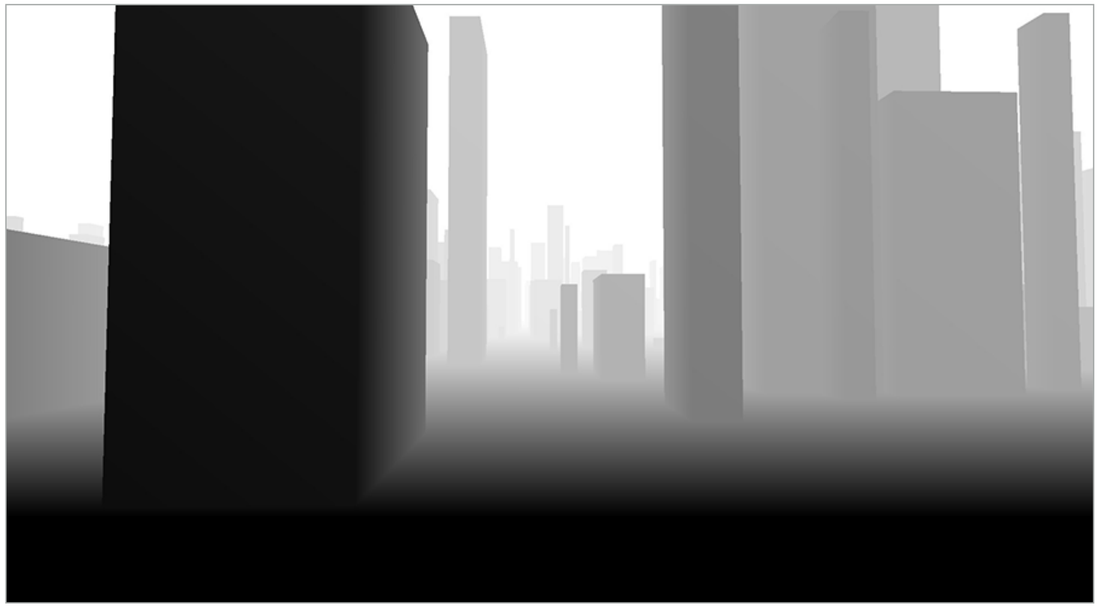
Color buffer:

(stores color per sample…
e.g., RGB)



Depth buffer:

(stores depth per sample)

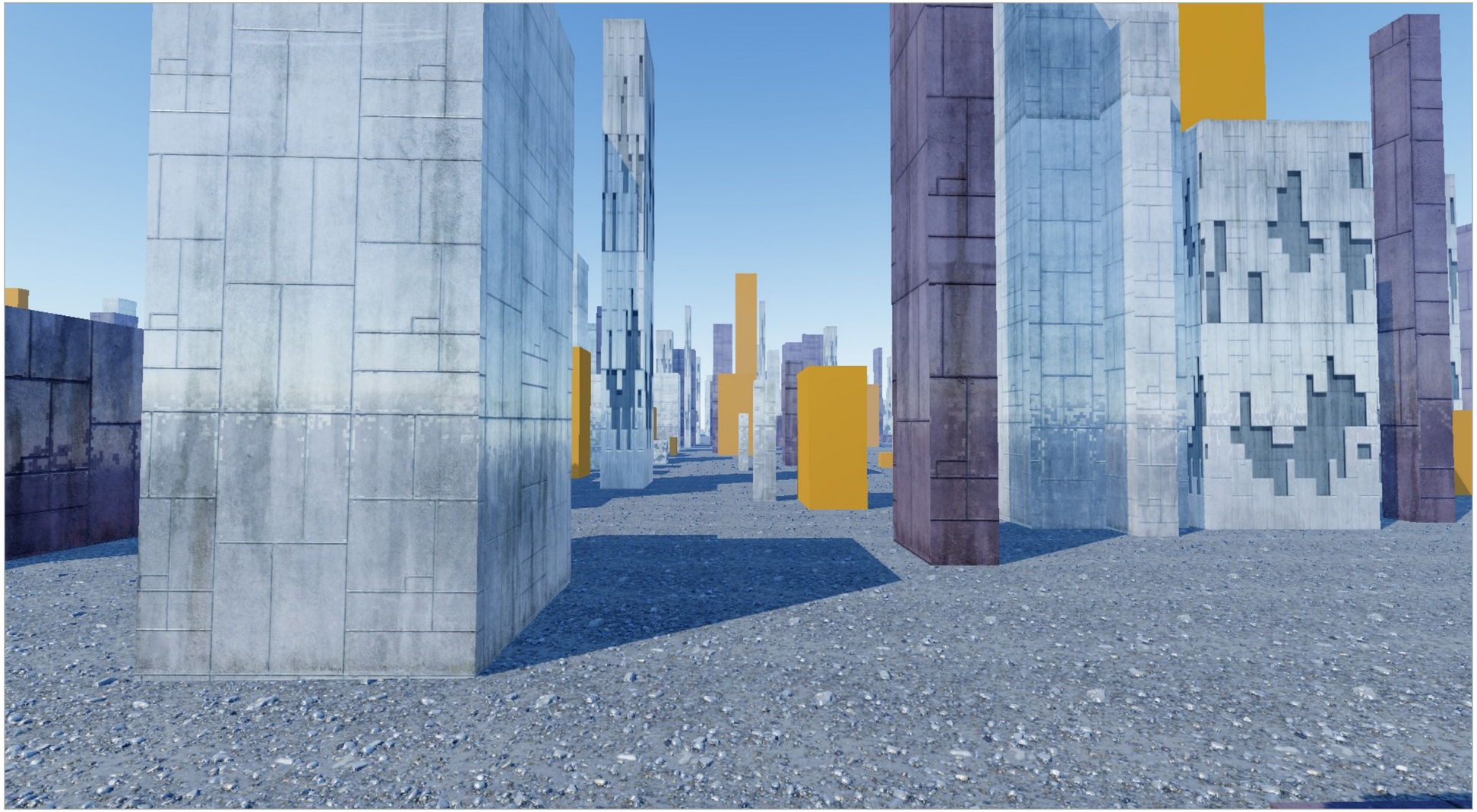


Stores depth of closest surface drawn so far

black = close depth

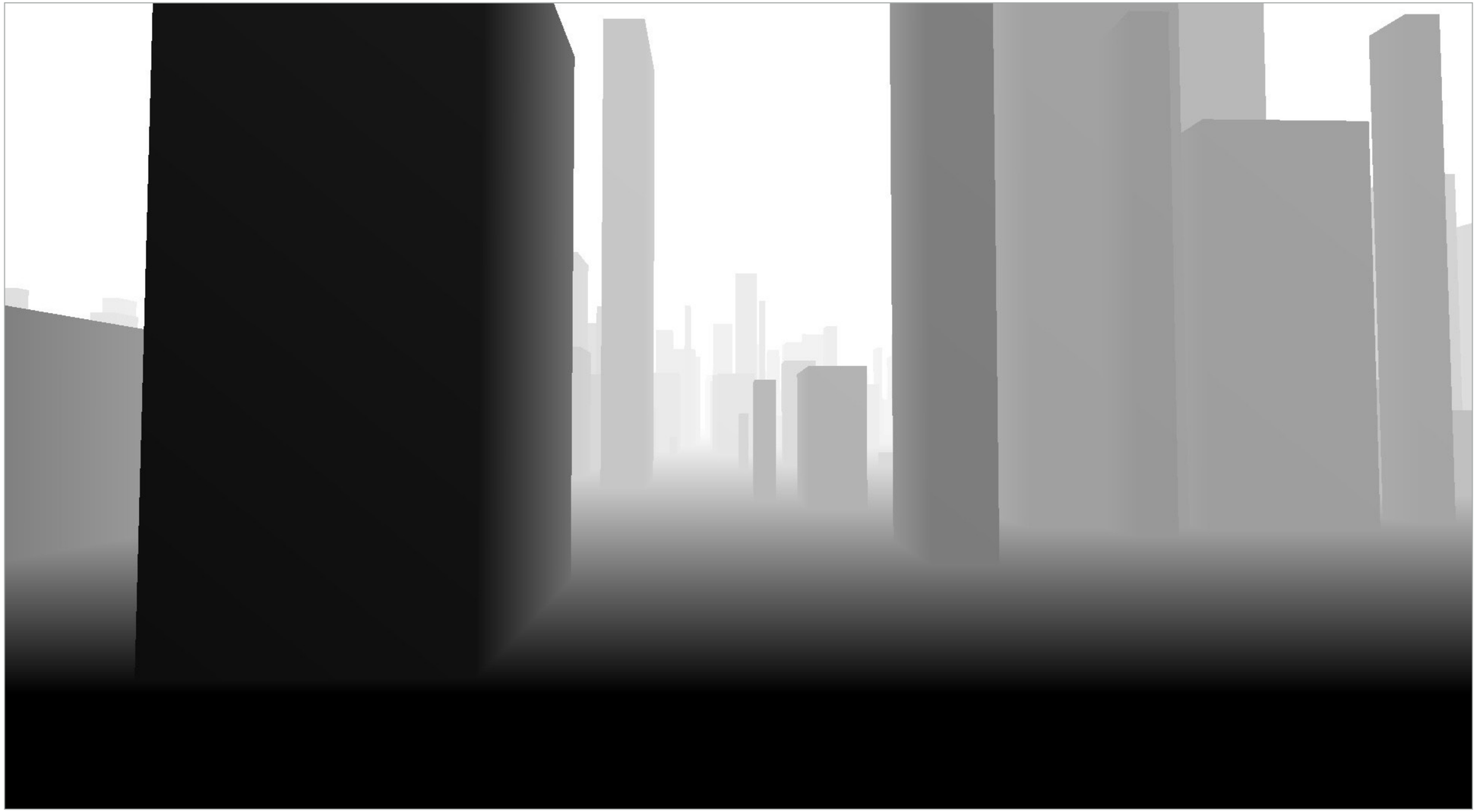
white = far depth

Depth buffer (a better look)



Color buffer

Depth buffer (a better look)



Corresponding depth buffer (after rendering all triangles)

Occlusion using the depth buffer (opaque surfaces)

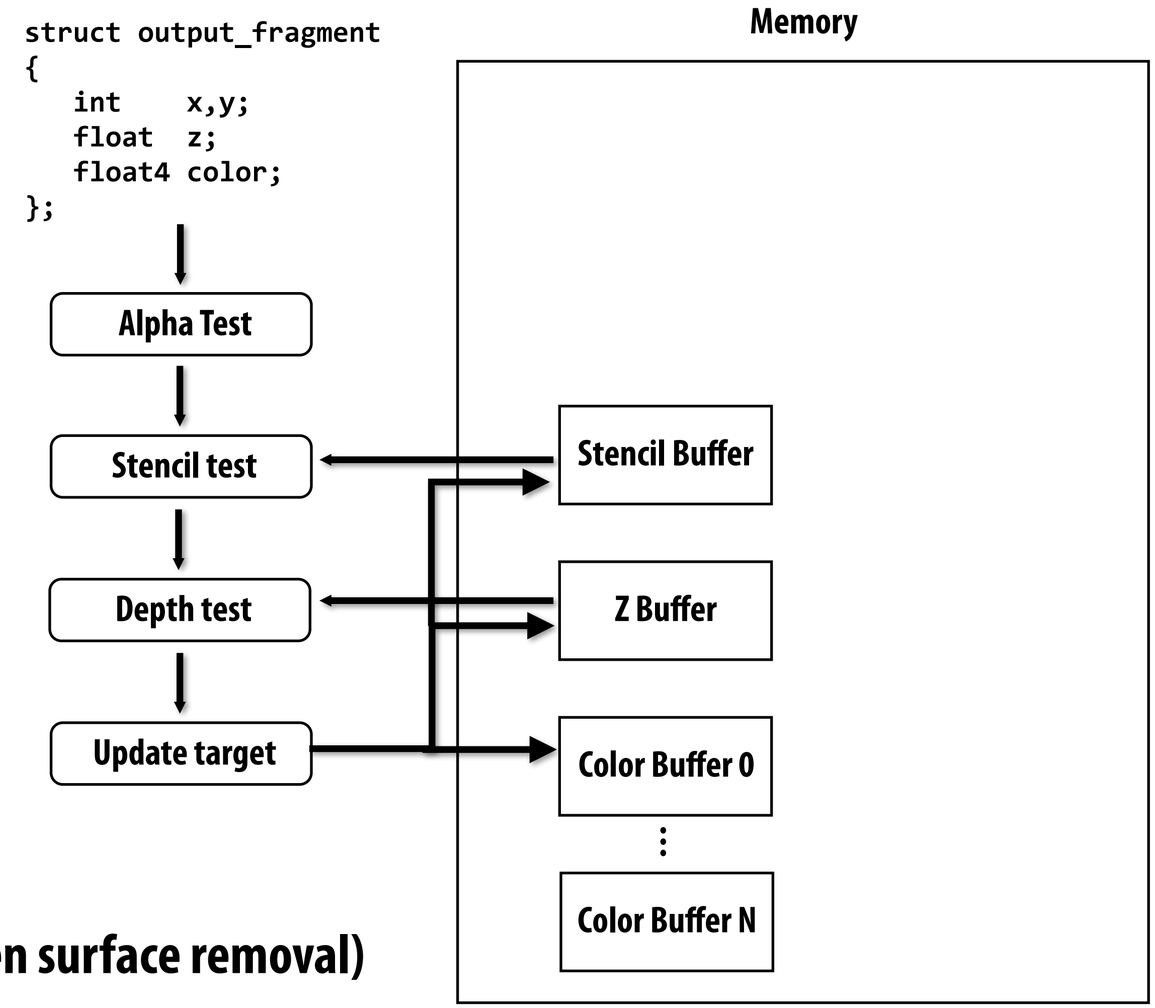
```
bool pass_depth_test(d1, d2) {
   return d1 < d2;
}


depth_test(tri_d, tri_color, x, y) {

  if (pass_depth_test(tri_d, depth_buffer[x][y]) {

    // triangle is closest object seen so far at this
    // sample point. Update depth and color buffers.

    depth_buffer[x][y] = tri_d;   // update depth_buffer
    color[x][y] = tri_color;      // update color buffer
  }
}
```

Summary: occlusion using a depth buffer

- Store one depth value per coverage sample (not per pixel!)
- Constant space per sample
  - Implication: constant space for depth buffer
- Constant time occlusion test per covered sample
  - Read-modify write of depth buffer if "pass" depth test
  - Just a read if "fail"
- Not specific to triangles: only requires that surface depth can be evaluated at a screen sample point

Frame-buffer operations (full view)

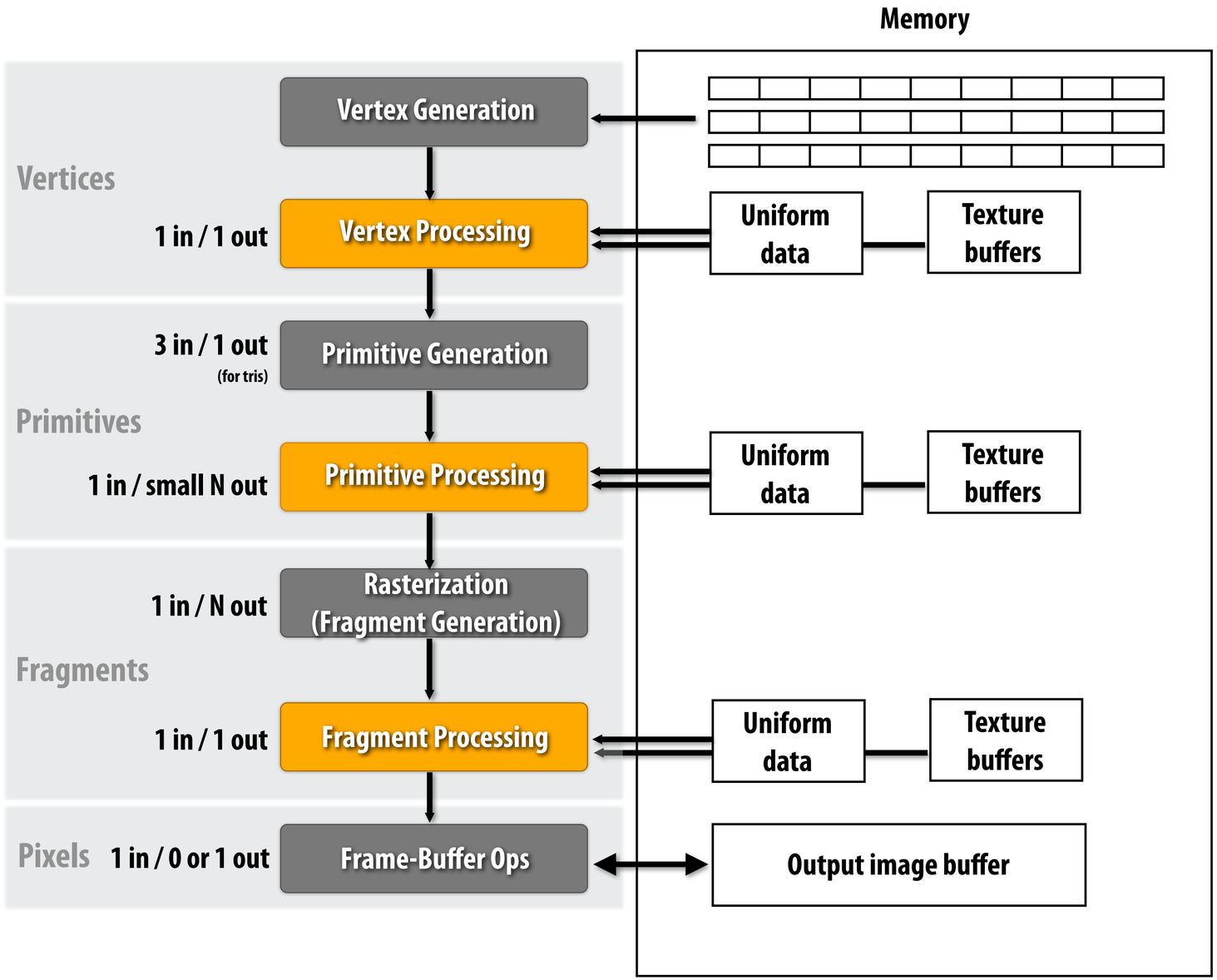
```
struct output_fragment
{
   int    x,y;
   float  z;
   float4 color;
};
```



Depth test (hidden surface removal)

```
if (fragment.z < zbuffer[fragment.x][fragment.y])
{
    zbuffer[fragment.x][fragment.y] = fragment.z;
    color_buffer[fragment.x][fragment.y] = blend(color_buffer[fragment.x][fragment.y], fragment.color);
}
```

The graphics pipeline



Programming the graphics pipeline

- Issue draw commands → output image contents change

Command Type	Command
State change	Bind shaders, textures, uniforms
Draw	Draw using vertex buffer for object 1
State change	Bind new uniforms
Draw	Draw using vertex buffer for object 2
State change	Bind new shader
Draw	Draw using vertex buffer for object 3
State change	Change depth test function
State change	Bind new shader
Draw	Draw using vertex buffer for object 4

Note: efficiently managing stage changes is a major challenge in implementations

A series of graphics pipeline commands

State change (set "red" shader)

Draw

State change (set "blue" shader)

Draw

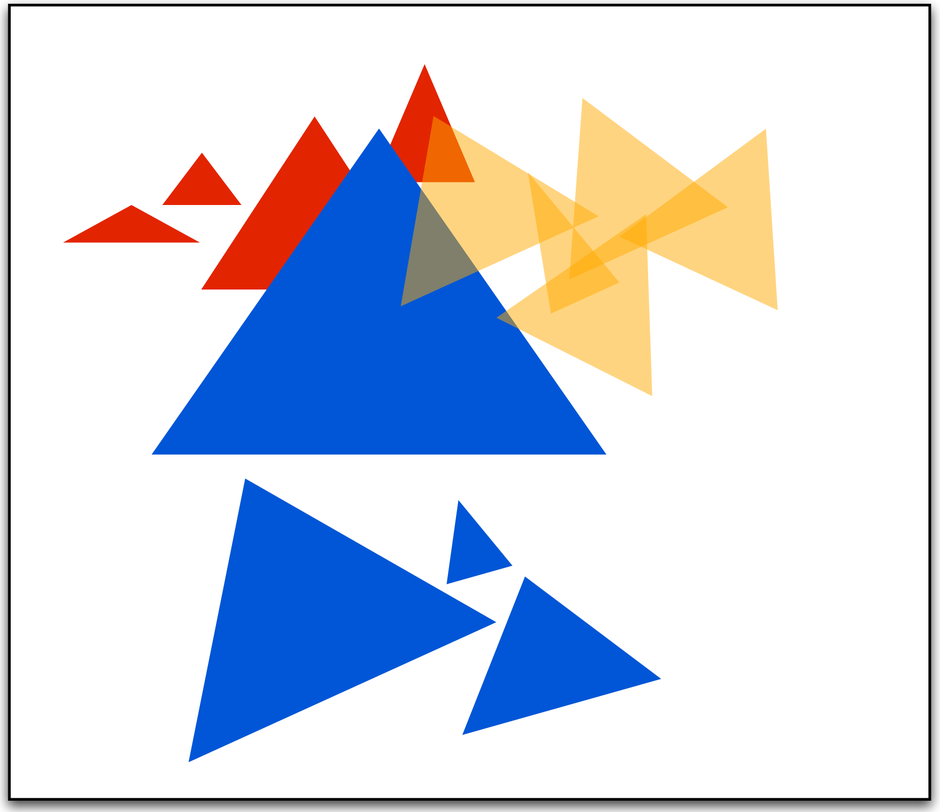
Draw

Draw

State change (change blend mode)

State change (set "yellow" shader

Draw



Feedback loop 1: use output image as input texture in later draw command

- Issue draw commands → output image contents change

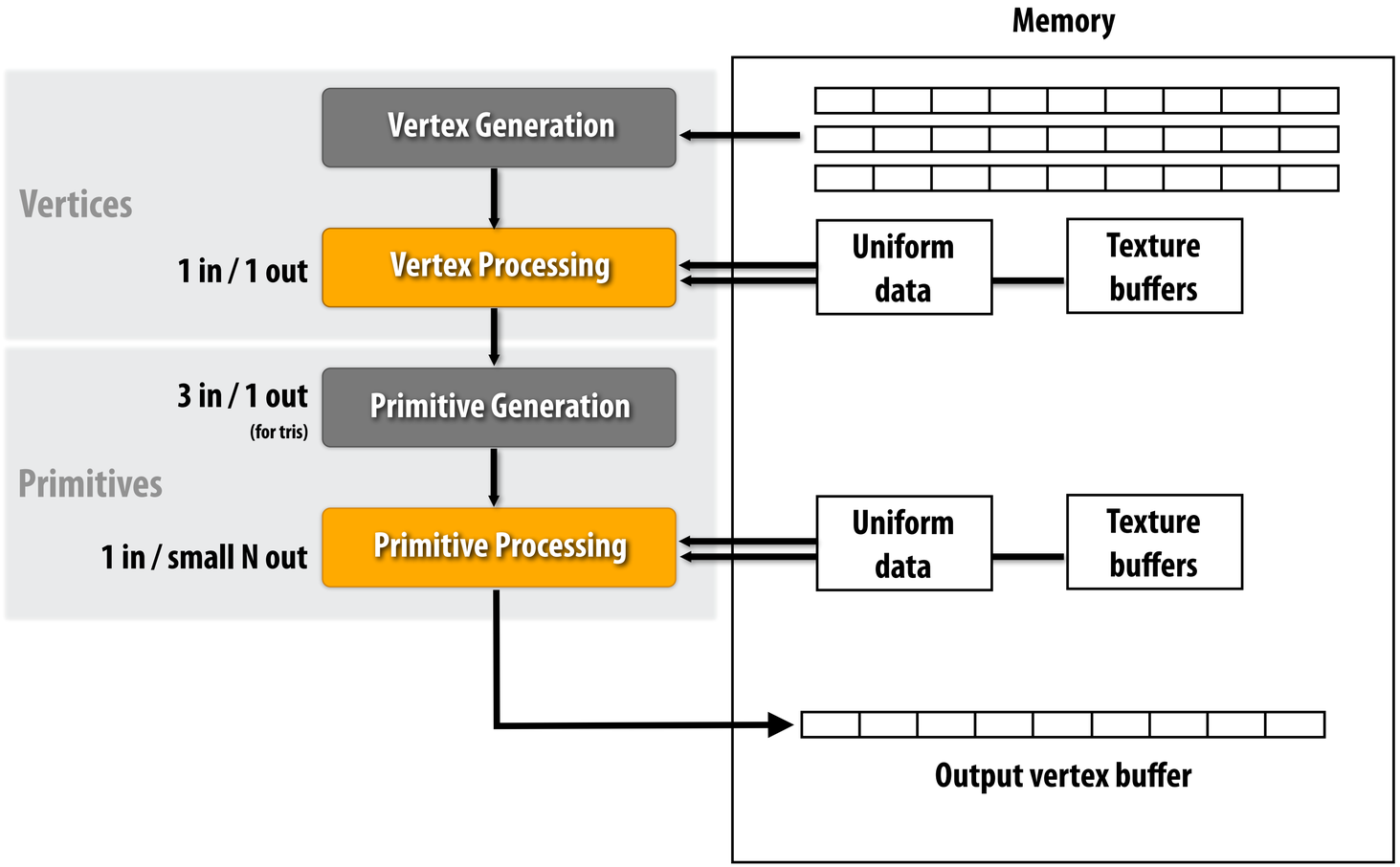
Command Type	Command
Draw	Draw using vertex buffer for object 5
Draw	Draw using vertex buffer for object 6
State change	Bind contents of output image as texture 1
Draw	Draw using vertex buffer for object 5
Draw	Draw using vertex buffer for object 6
	⋮

Rendering to textures for later use is key technique when implementing:

- Shadows
- Environment mapping
- Post-processing effects

Feedback loop 2: output intermediate geometry for use in later draw command

- Issue draw commands ⟶ emit geometry buffers



Analyzing the design of the graphics pipeline

- Level of abstraction
- Orthogonality of abstractions
- How is pipeline designed for performance/scalability?
- What the pipeline does and DOES NOT do

* These are great questions to ask yourself about any system we discuss in this course

Level of abstraction

- Imperative abstraction, not declarative
  - Application code specifies: "draw these triangles, using this fragment shader, with depth testing on".
  - It does not specify: "draw a cow made of marble on a sunny day"

- Programmable stages provide application large amount of flexibility
  (e.g., to implement wide variety of materials and lighting techniques)

- Configurable (but not programmable) pipeline structure: application can turn stages on and off, create feedback loops

- Abstraction is low enough to allow application to implement many techniques, but high enough to abstract over radically different GPU implementations (NVIDIA, AMD, Intel GPUs, mobile GPUs, etc.)

Orthogonality of abstractions

- **All vertices treated the same regardless of primitive type**
  - Result: vertex programs are oblivious to primitive types
  - The same vertex program works for triangles and lines

- **All primitives are converted into fragments for per-pixel shading and frame-buffer operations**
  - Fragment programs are oblivious to source primitive type and the behavior of the vertex program *
  - Z-buffer is a common representation used to perform occlusion for any primitive that can be converted into fragments

* Almost oblivious. Vertex shader must make sure it passes along all inputs required by the fragment shader

What the pipeline DOES NOT do (non-goals)

- **Modern graphics pipeline has no concept of lights, materials, geometric modeling transforms**
  - Only streams of records processed by application defined kernels: vertices, primitives, fragments, pixels
  - And pipeline state (input/output buffers, "shaders", and fixed-function configuration parameters)
  - Applications implement lights, materials, etc. using these basic abstractions

- **The graphics pipeline has no concept of a scene**

- **It is just a virtual machine that executes pipeline state change and primitive drawing commands**

Pipeline design facilitates performance/scalability

- [Reasonably] low level: low abstraction distance to implementation
- Constraints on pipeline structure:
  - Constrained data flow between stages
  - Fixed-function stages for common and difficult to parallelize tasks
  - Shaders: independent processing of each data element (enables data parallelism)
- Provide frequencies of computation (per vertex, per primitive, per fragment)
  - Application can choose to perform work at the rate required
- Keep it simple:
  - Only a few common intermediate representations
    - Triangles, points, lines
    - Fragments, pixels
  - Z-buffer algorithm computes visibility for any primitive type
- "Immediate-mode system": pipeline processes primitives as it receives them
  (as opposed to buffering the entire scene)
  - Leave global optimization of how to render scene to the application

Perspective from OpenGL designer Kurt Akeley

- Does the system meet original design goals, and then do much more than was originally imagined?

- If so, the design is a good one!
  - Simple, orthogonal concepts often produce this amplifier effect