Lecture 2:
The Real-Time Graphics Pipeline

Kayvon Fatahalian
CMU 15-869: Graphics and Imaging Architectures (Fall 2011)

Today

- The real-time graphics pipeline

- How the pipeline is used by applications (workload)

Issues to keep in mind

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

System stack

Application

(e.g, a computer game)

↓

Scene graph

(database representing/organizing the scene: objects, materials, lights, etc.)

↓

Graphics Pipeline

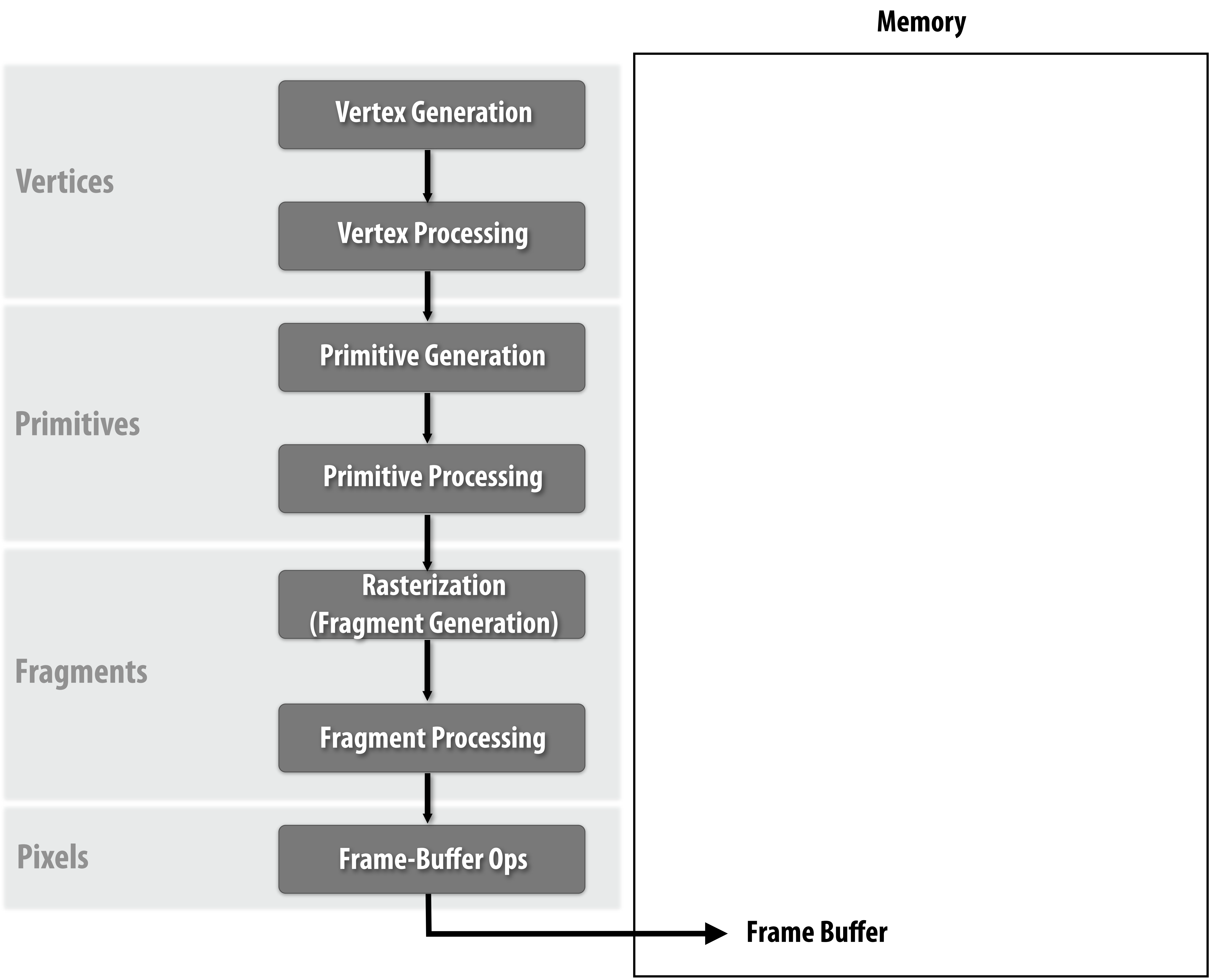
(OpenGL/Direct3D)

↓

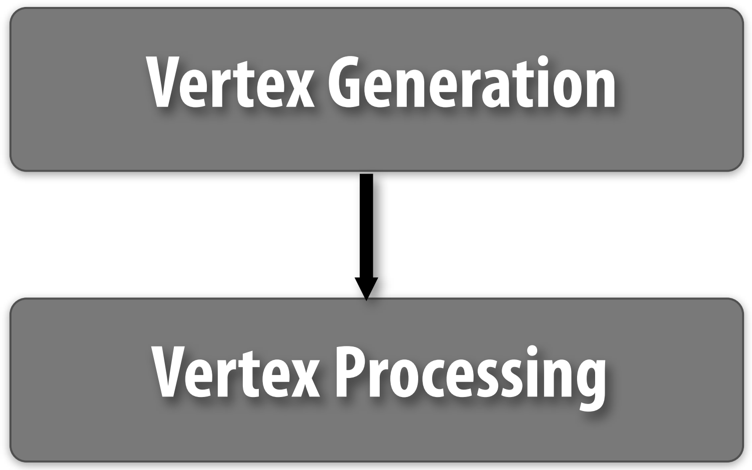
Graphics Pipeline Implementation

(software driver + GPU)

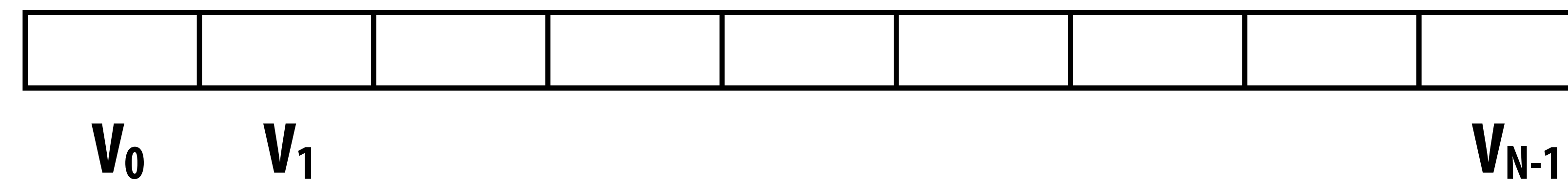
The graphics pipeline (from last time)



“Assembling vertices”

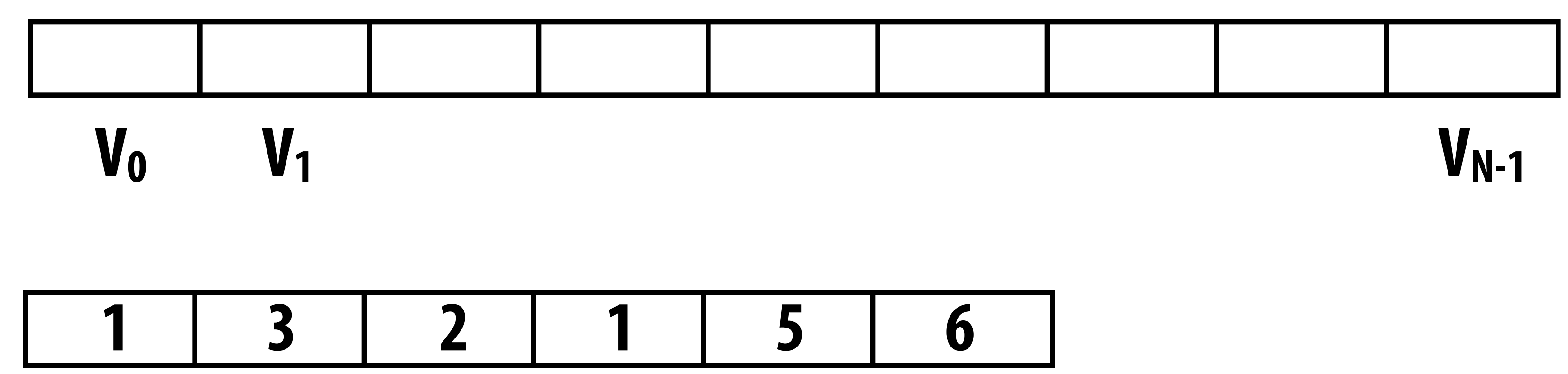


Contiguous Version



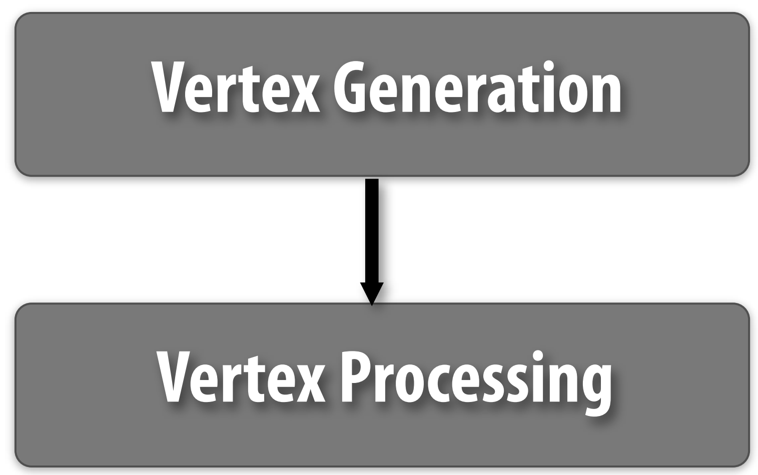

```
glBindBuffer(GL_ARRAY_BUFFER, my_vtx_buffer);
glDrawArrays(GL_TRIANGLES, 0, N);
```

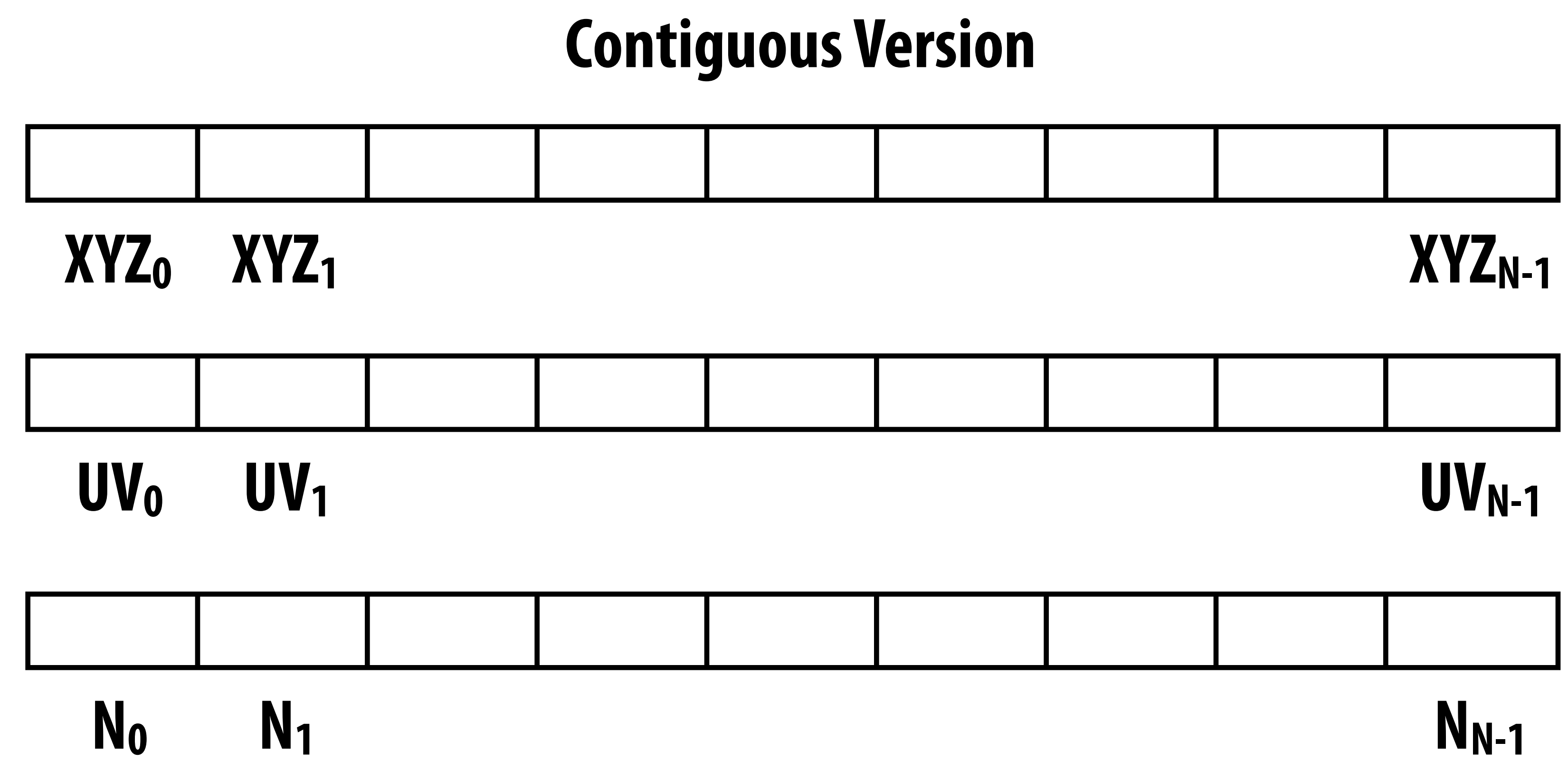
Indexed Version (gather)




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

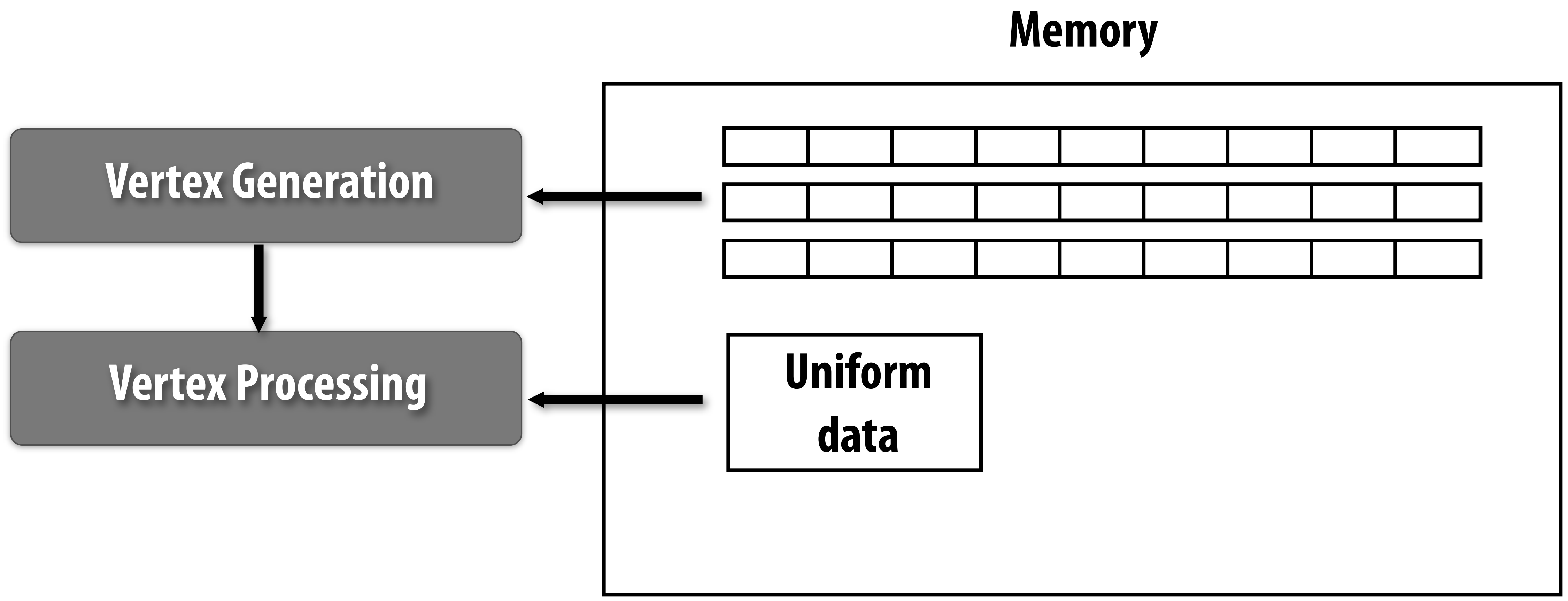
“Assembling vertices”





Current pipelines set limit of 16 float4 attributes per vertex.

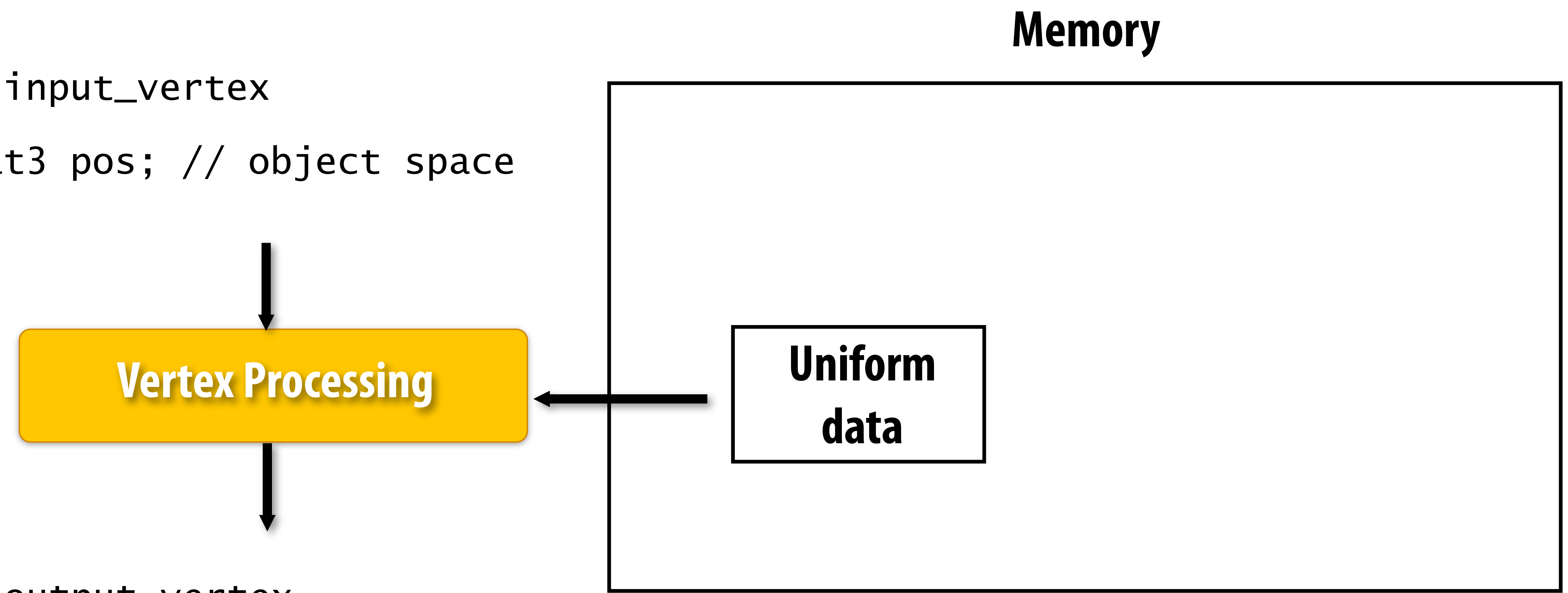
Vertex stage inputs



Uniform data: constant across vertices
e.g., vertex transform matrix

Vertex stage inputs

```
struct input_vertex
{
   float3 pos; // object space
}
```




```
struct output_vertex
{
   float3 pos; // NDC space
}
```

1 input vertex → 1 output vertex

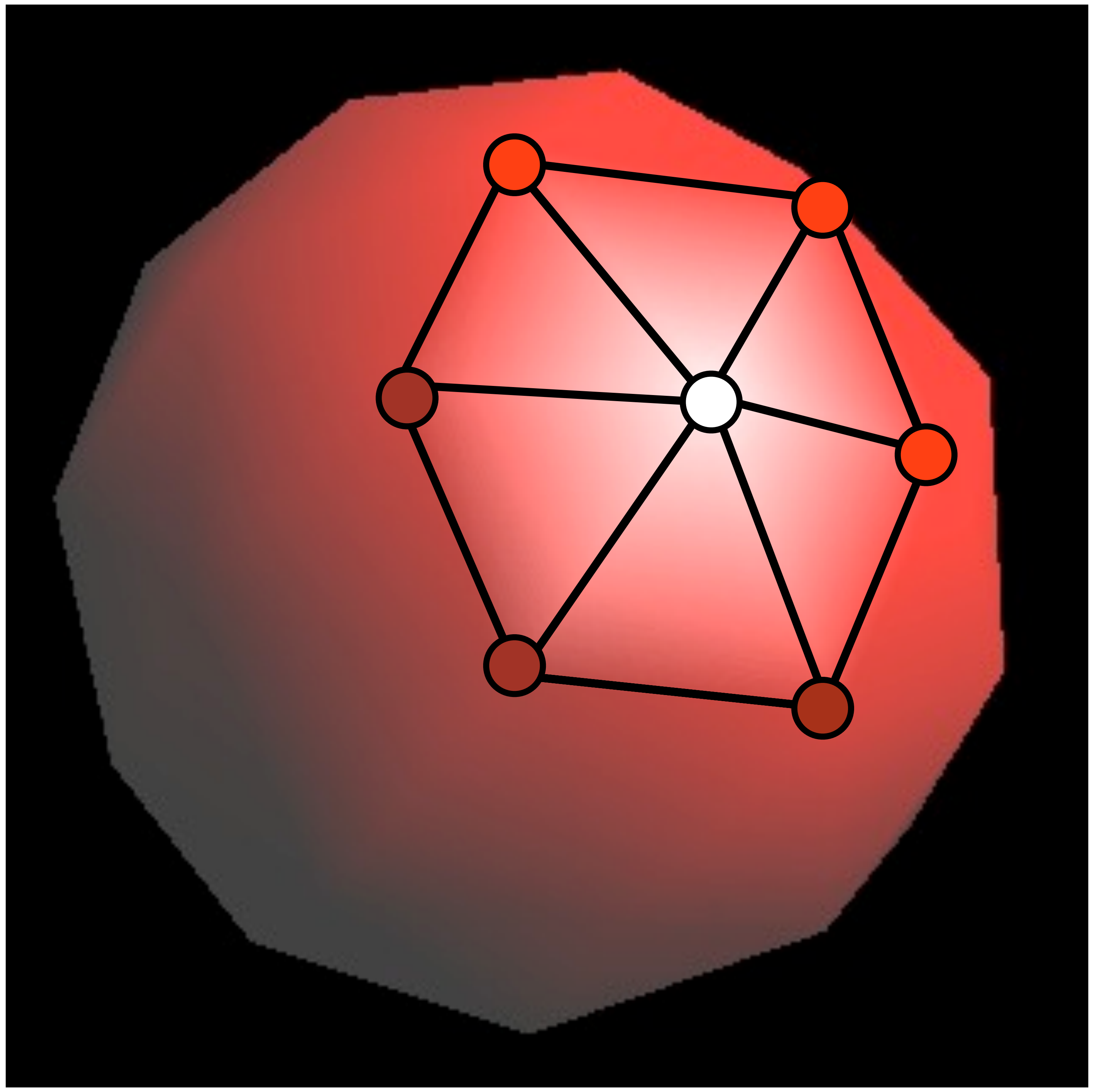
independent processing of each vertex

Program

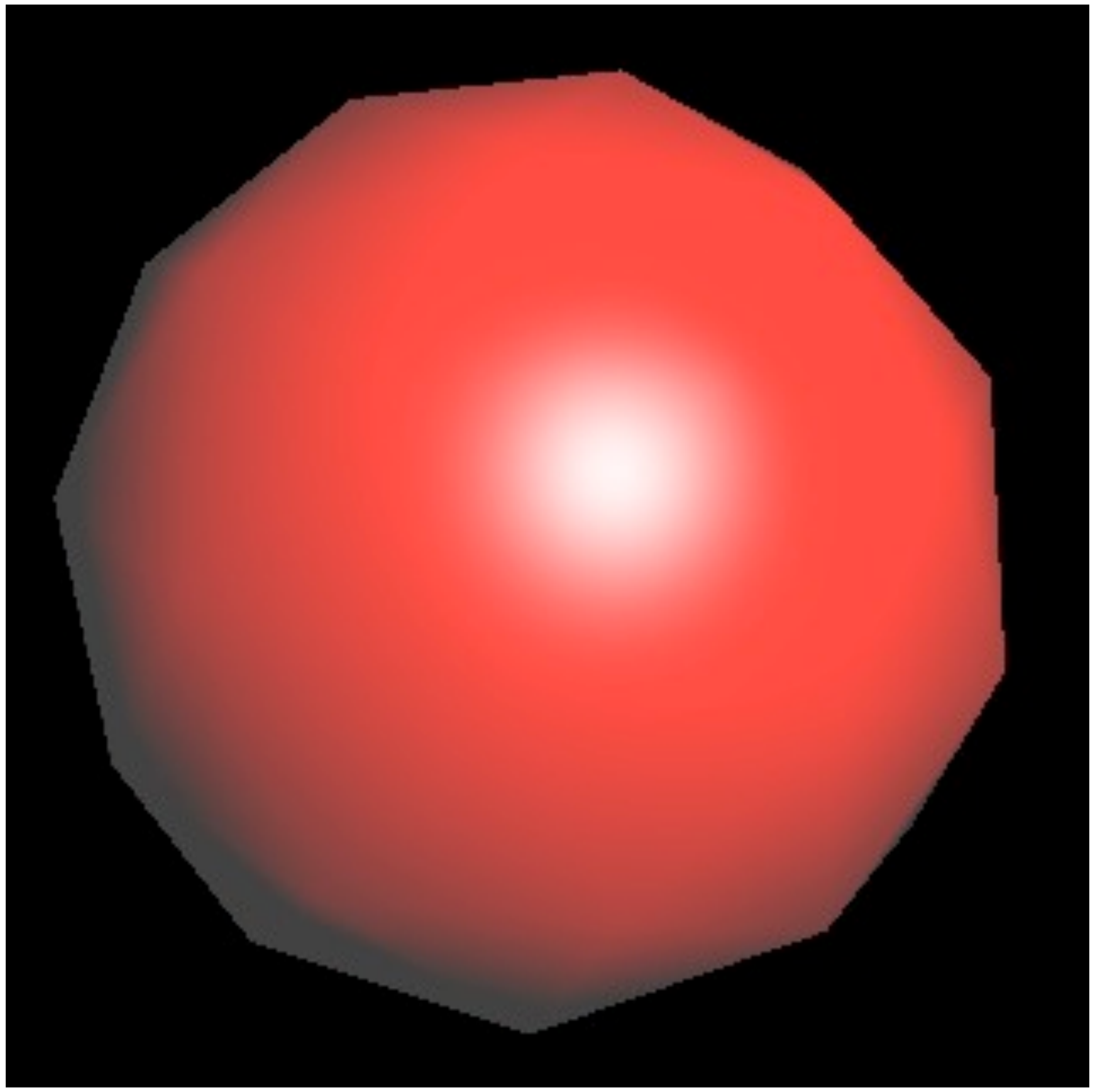
```
uniform mat4 my_transform;

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

(* Note: for clarity, this is not proper GLSL syntax)**

Vertex processing example: lighting



Per vertex lighting

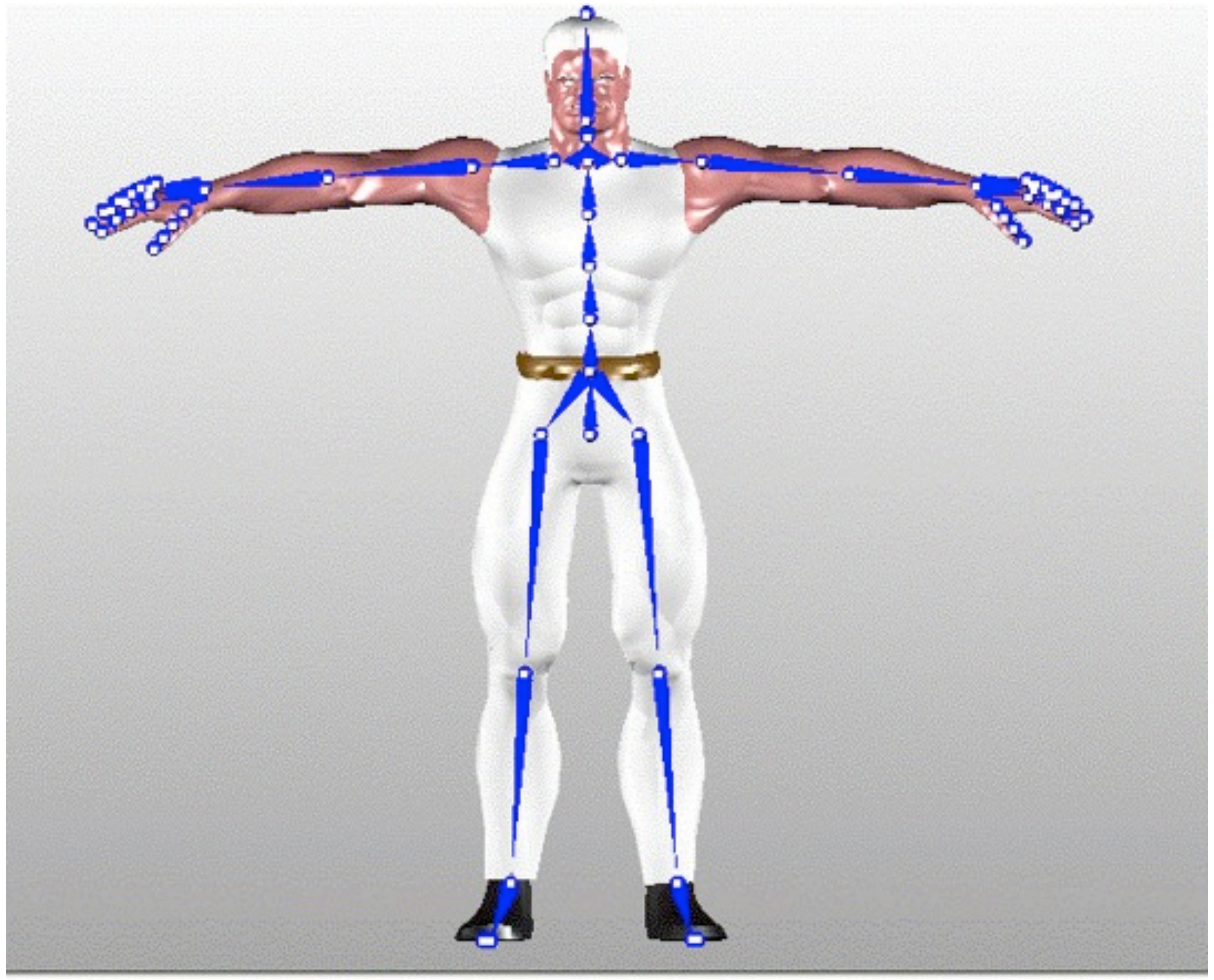


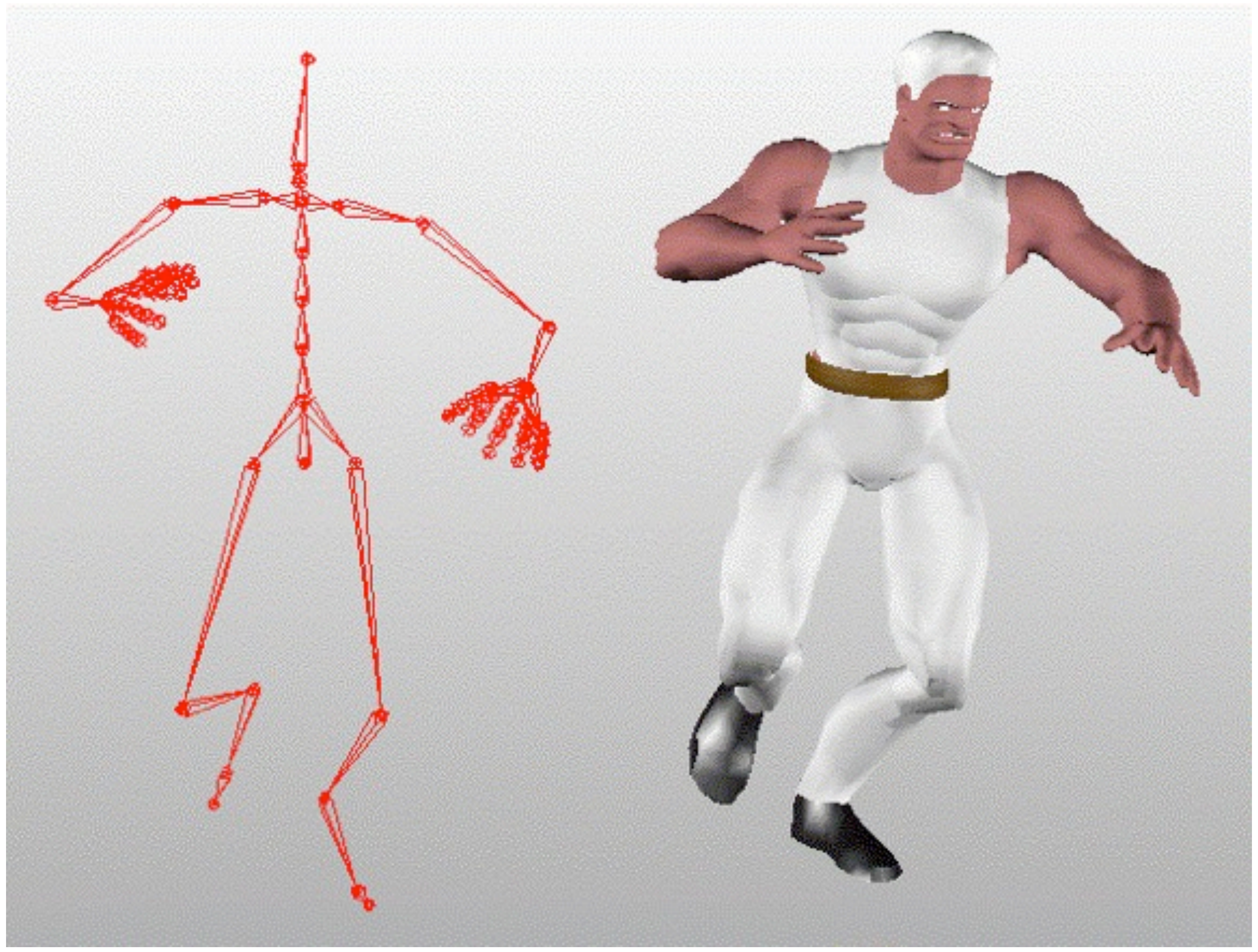
Per vertex normal, per pixel lighting

Per vertex data: surface normal, surface color

Uniform data: light direction, light color

Vertex processing example: skinning





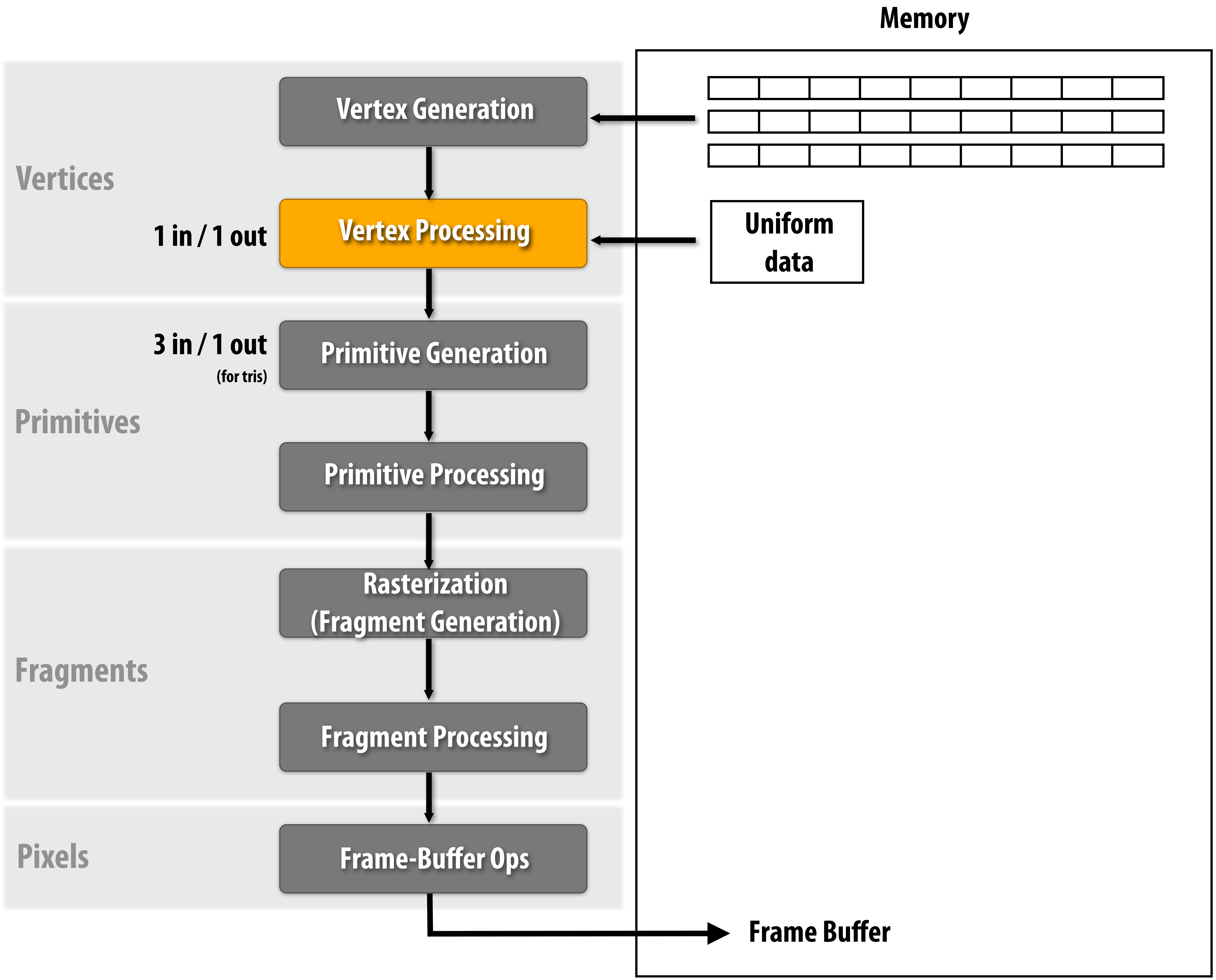
$$V_{skinned} = \sum_{b \in bones} w_b M_b V_{base}$$

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

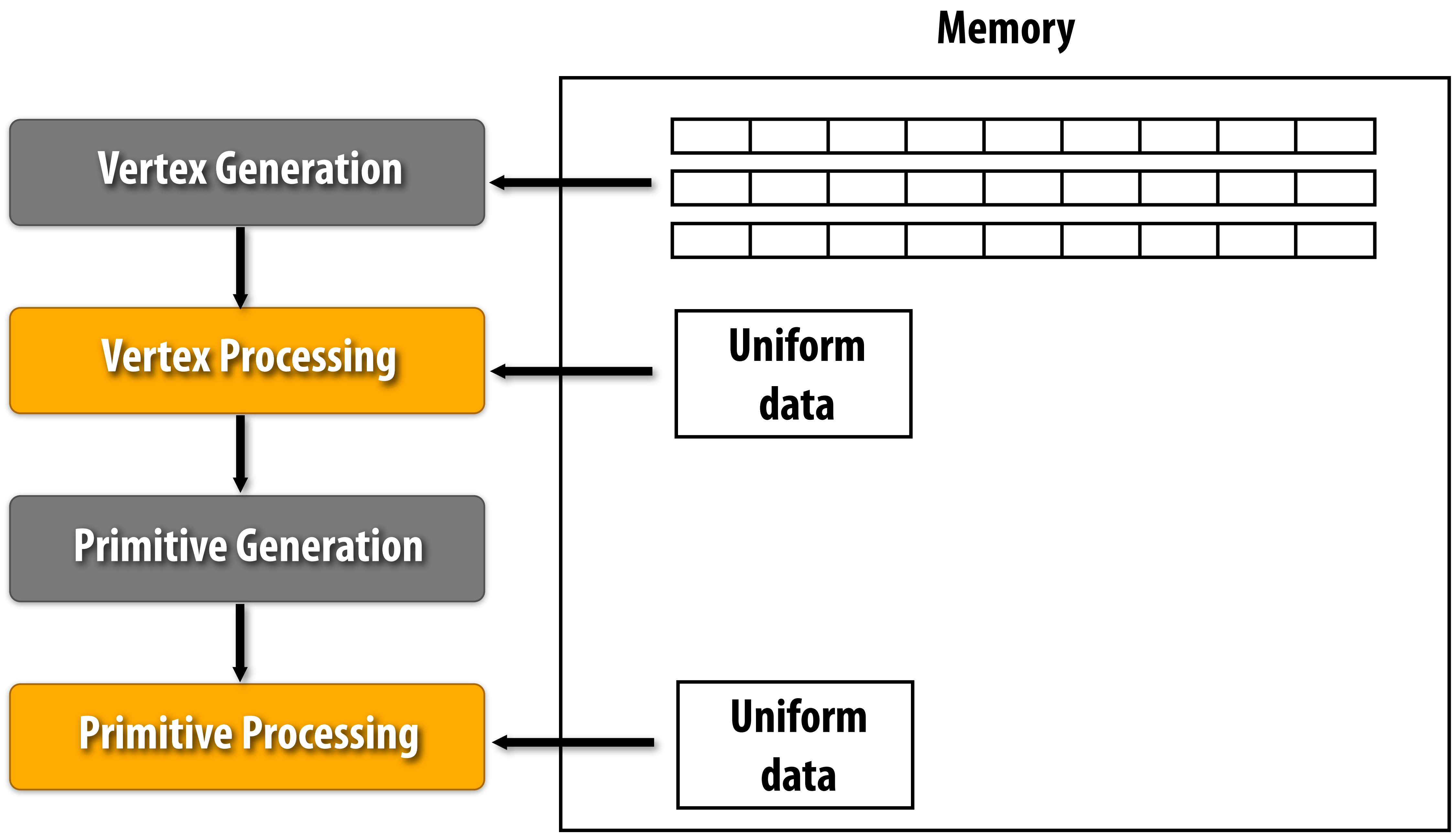
Per vertex data: blend coefficients (depend on current animation frame)

Uniform data: "bone" matrices

The graphics pipeline



Primitive processing

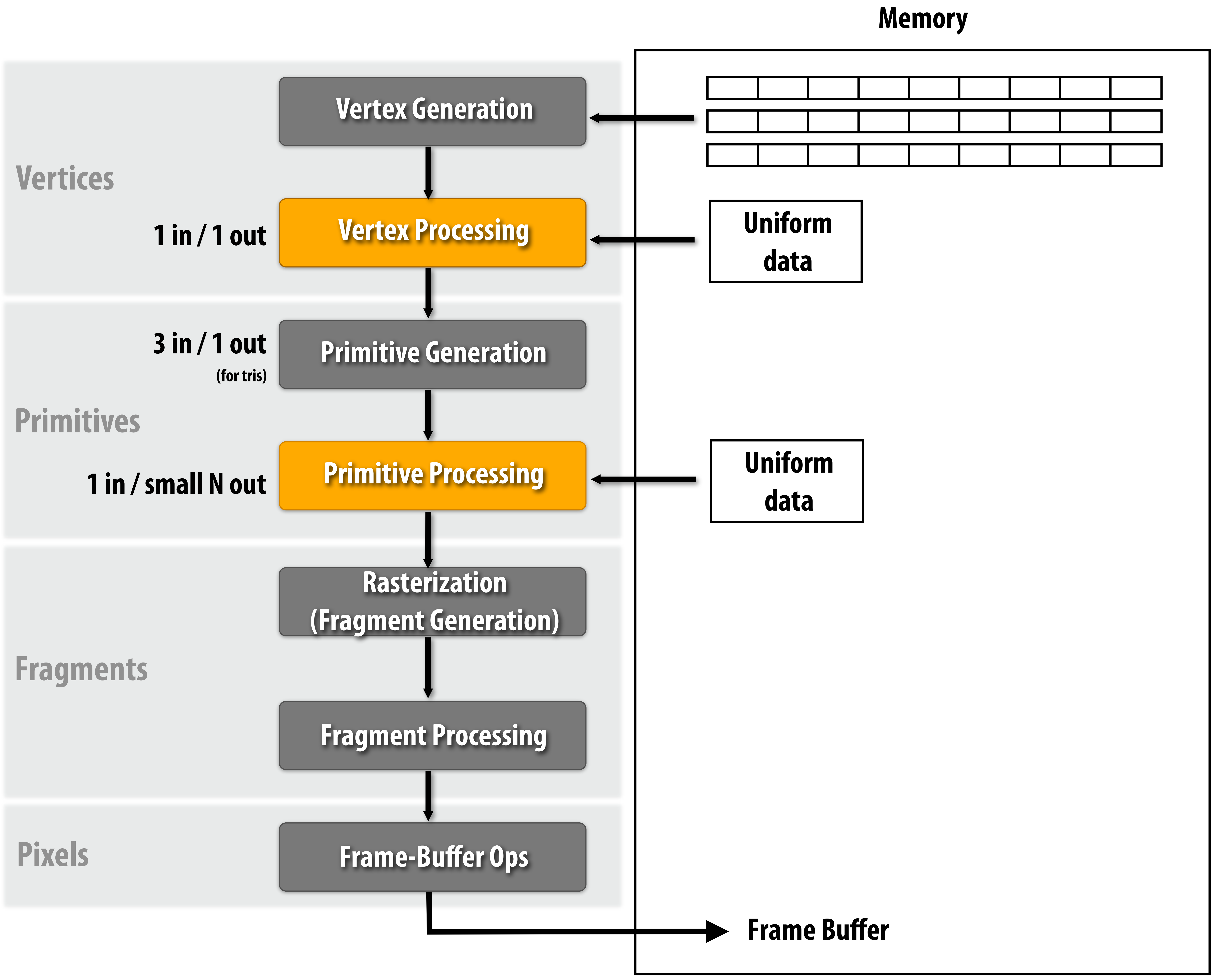


input vertices for 1 prim → output vertices for N prims**

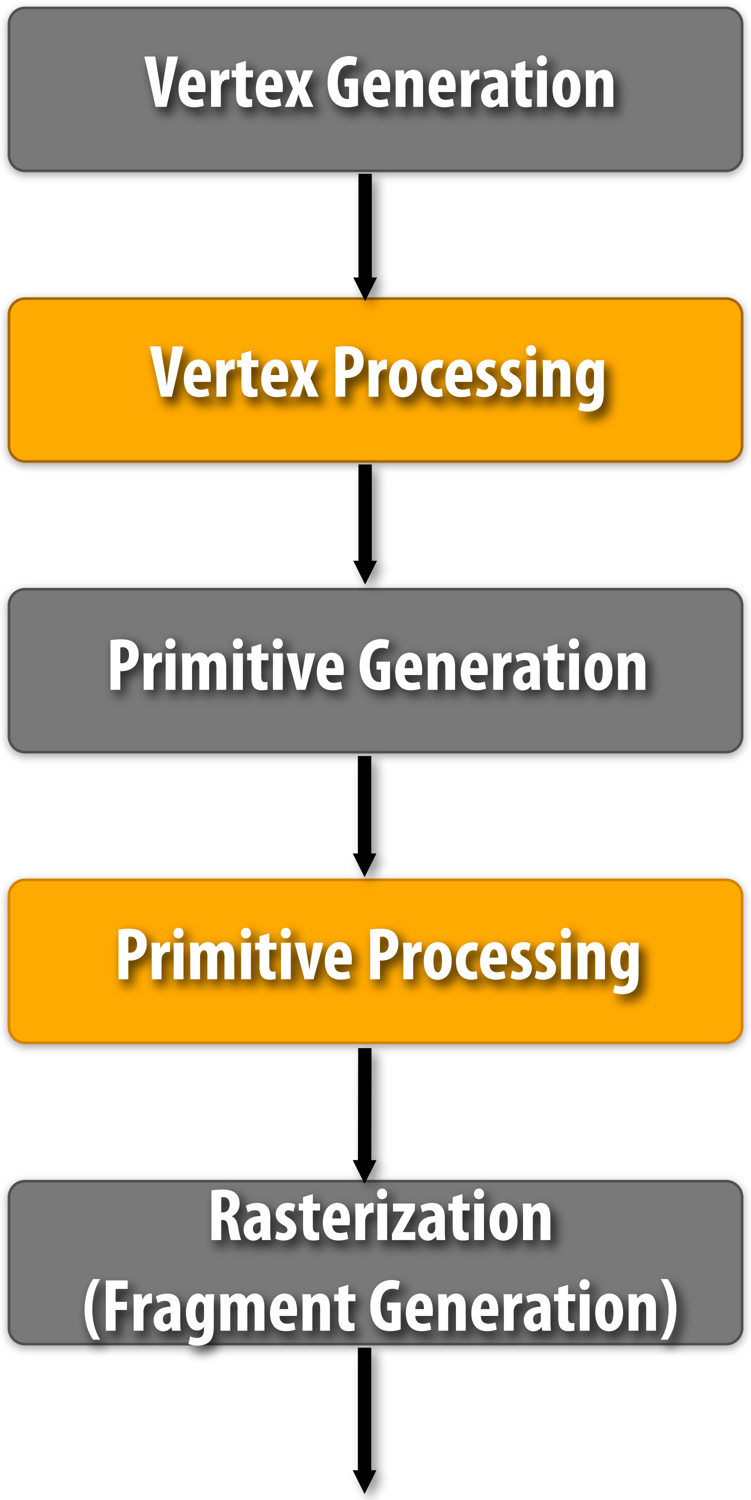
independent processing of each INPUT primitive

** caps output at 1024 floats of output

The graphics pipeline

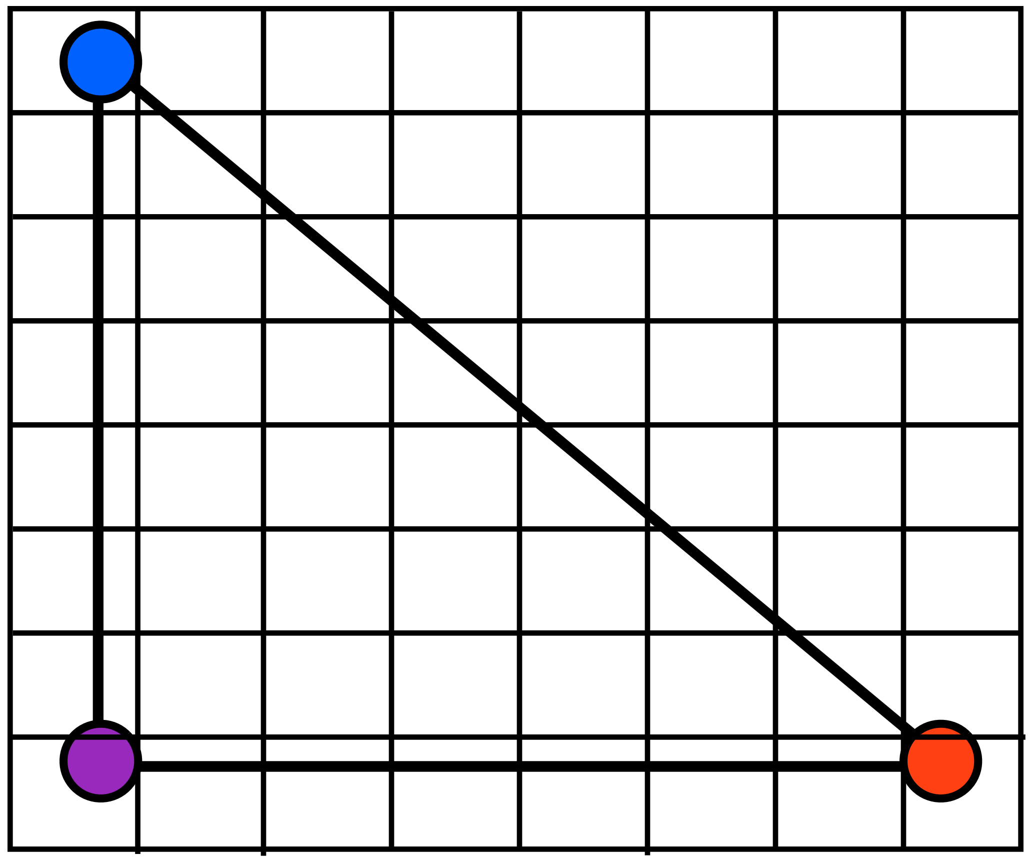


Rasterization



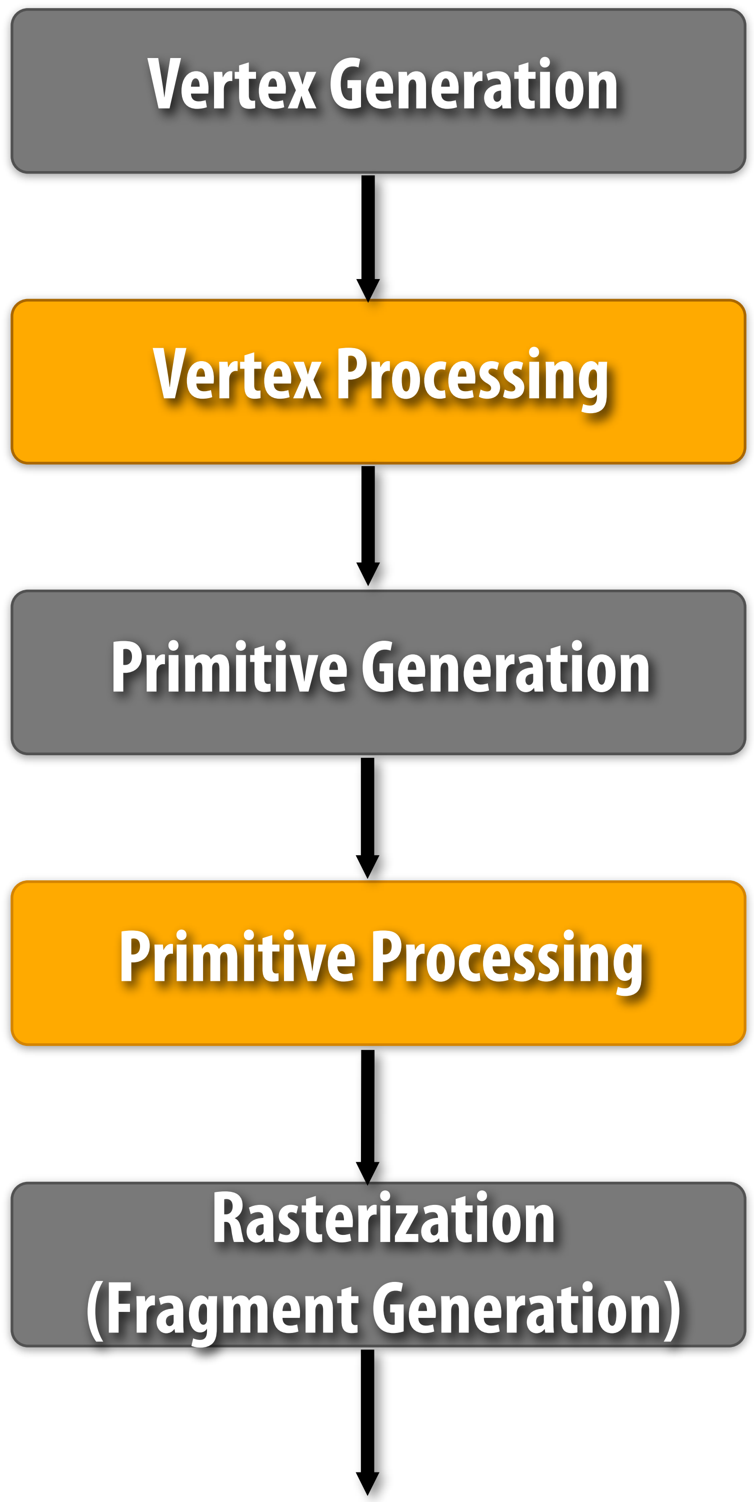
1 input prim ⟶ N output fragments

N is unbounded
(size of triangles varies greatly)



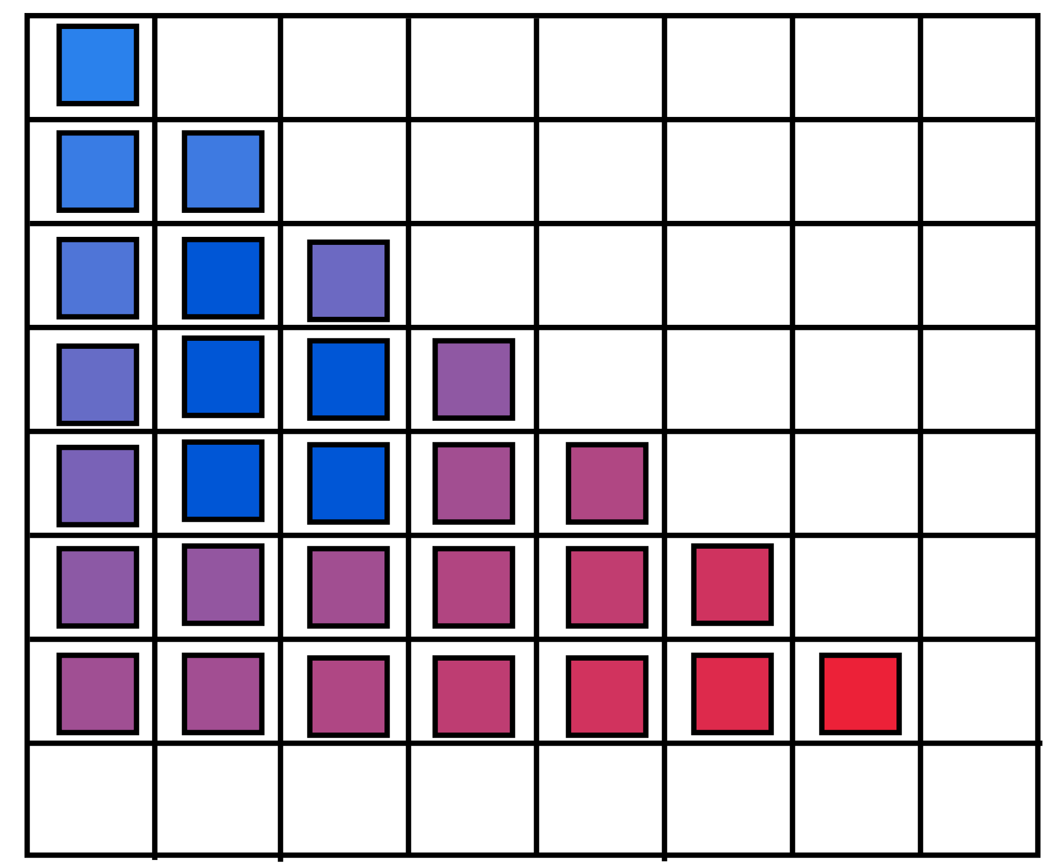
```
struct fragment // note similarity to output_vertex from before
{
   float  x,y;  // screen pixel coordinates
   float  z;    // depth of triangle at this pixel

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

}
```

Rasterization



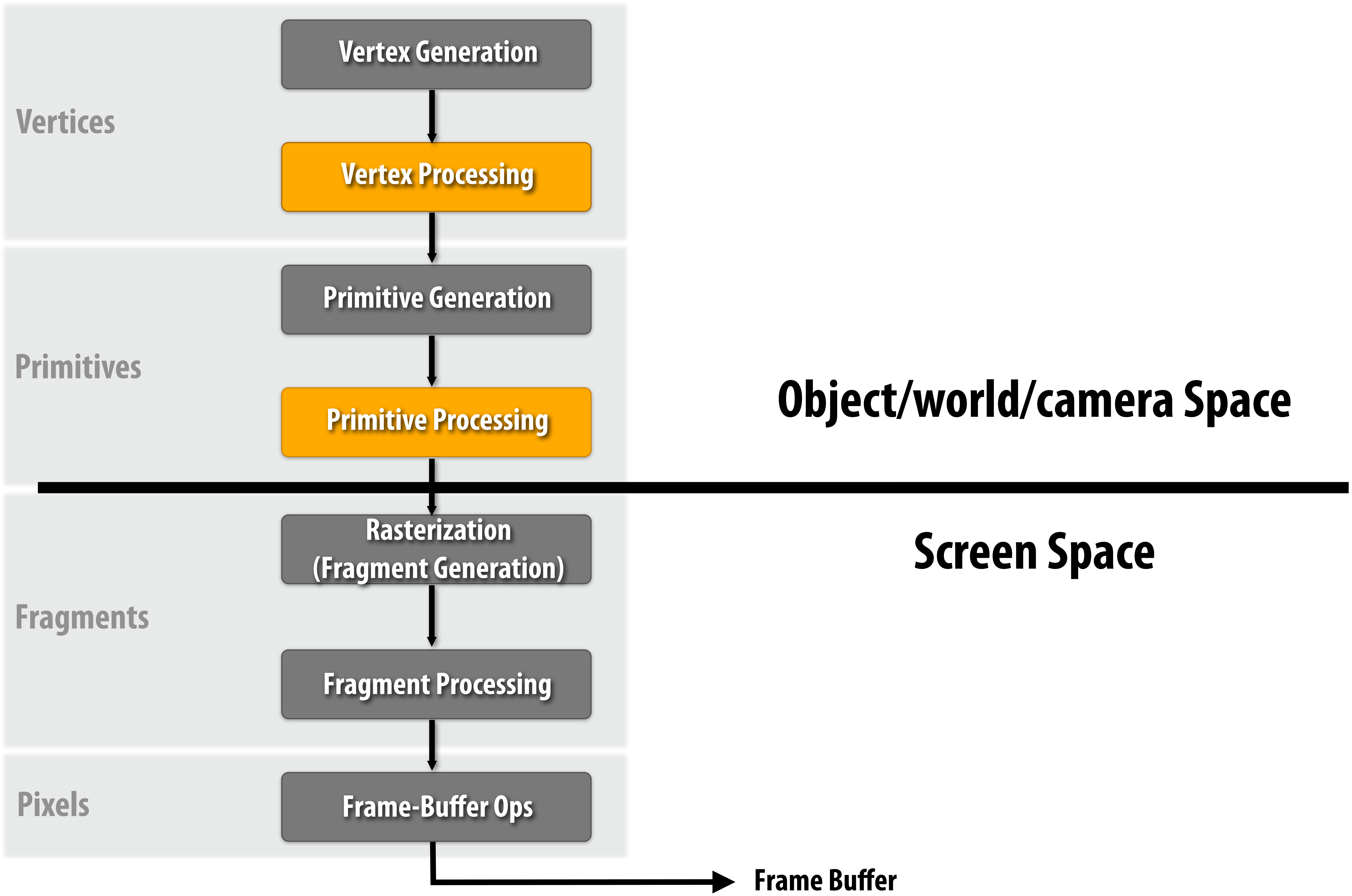
Compute covered pixels

Sample vertex attributes once per covered pixel

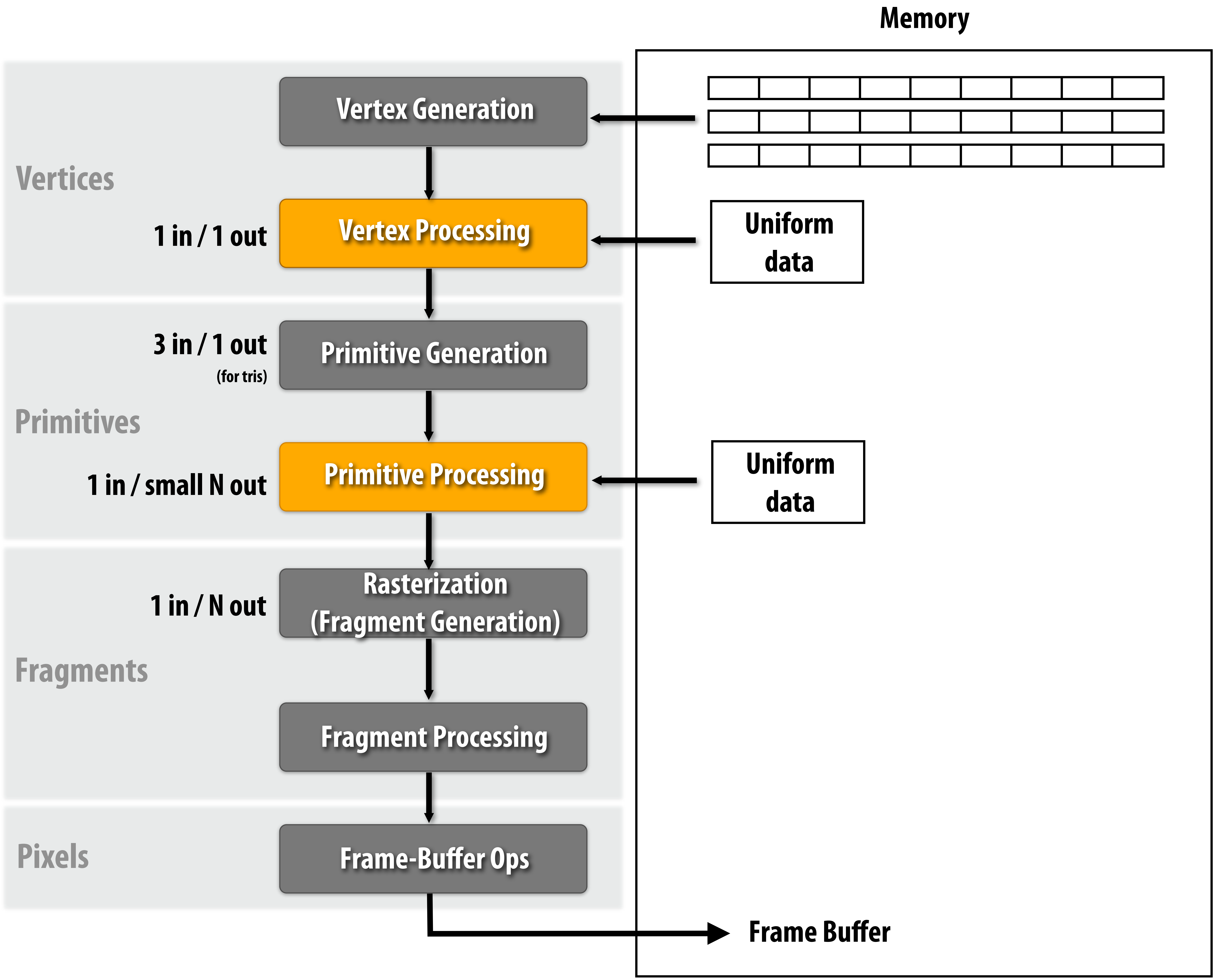


```
struct fragment // note similarity to output_vertex from before
{
   float  x,y;  // screen pixel coordinates (sample point location)
   float  z;    // depth of triangle at sample point

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

}
```

The graphics pipeline

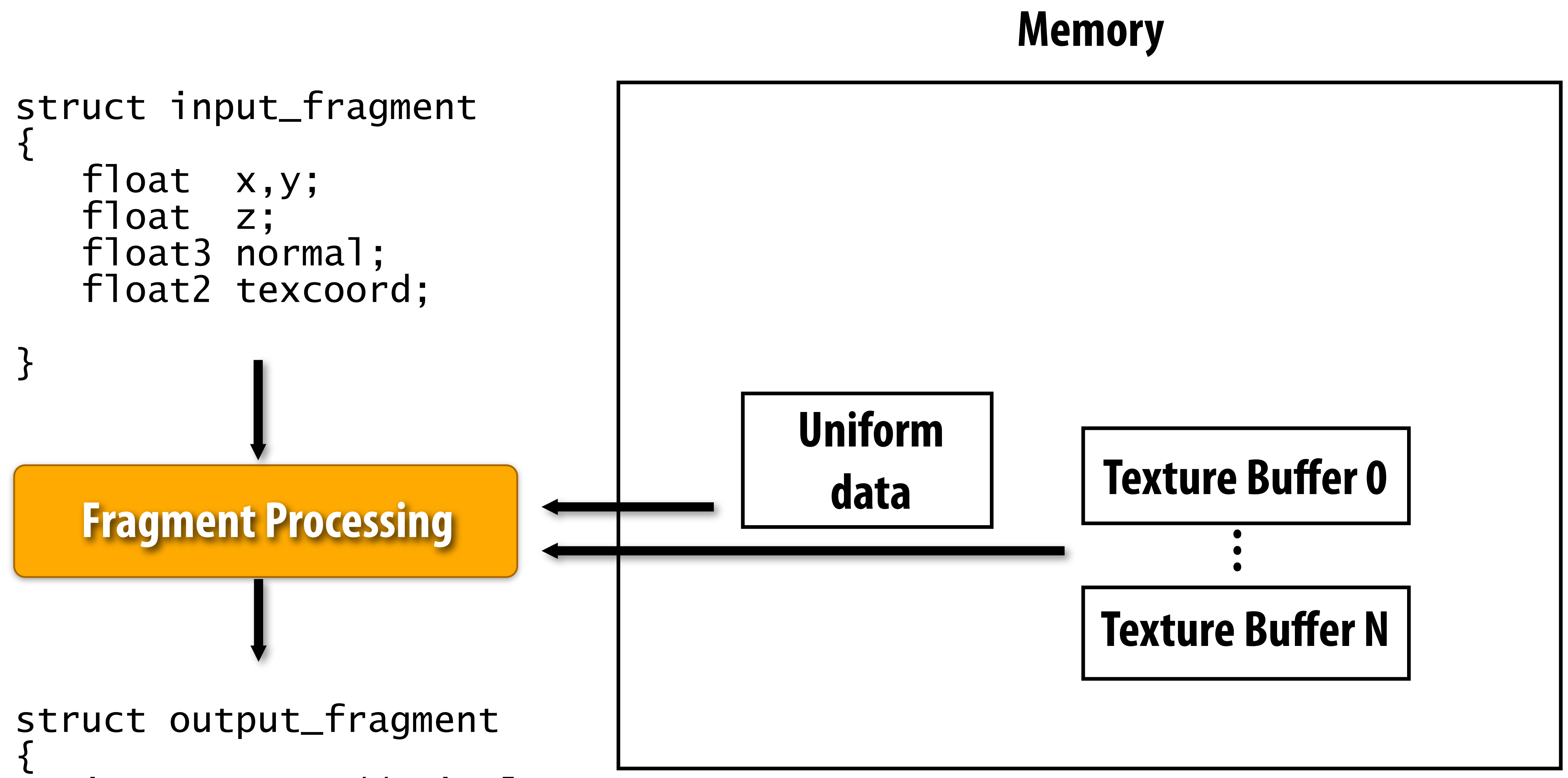


The graphics pipeline



Fragment processing

```
struct input_fragment
{
   float  x,y;
   float  z;
   float3 normal;
   float2 texcoord;

}
```




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

```
texture my_texture;

output_vertex my_vertex_program(input_vertex input)
{
    output_fragment out;

    float4 material_color = sample(my_texture, input.texcoord);

    for (all lights in scene)
    {
        out.color += // compute light reflectance towards camera
    }

}
```

Many uses for textures

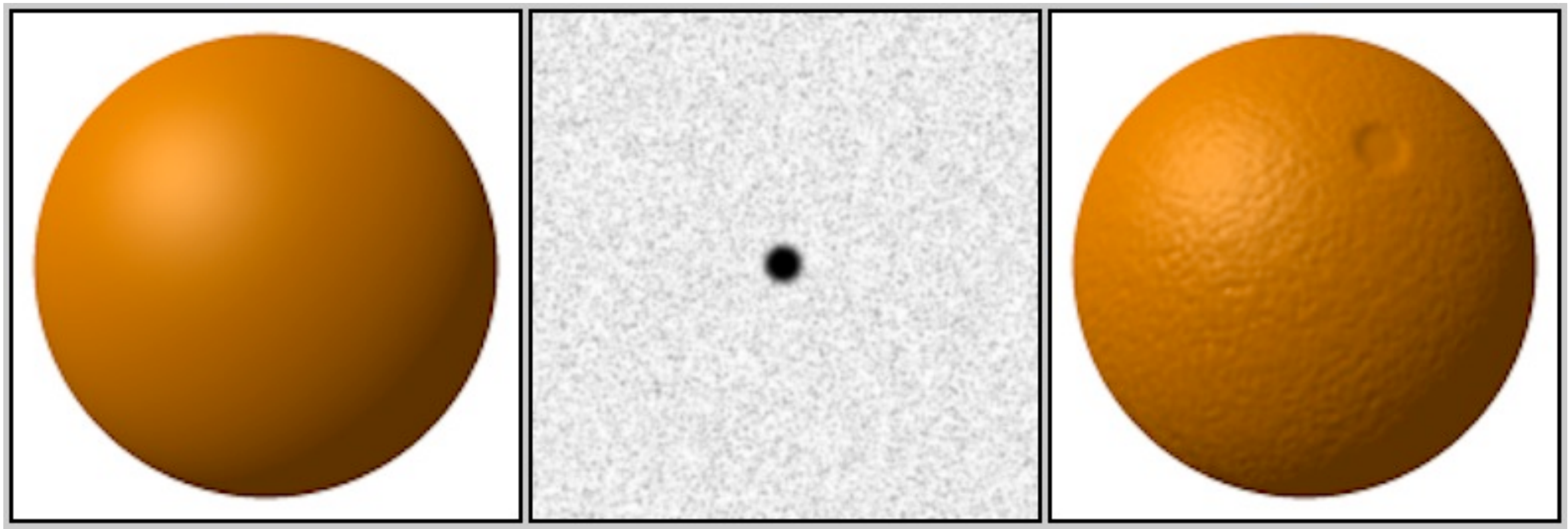
Provide surface color/reflectance

Tom Porter's Bowling Pin

Source: RenderMan Companion, Pls. 12 & 13

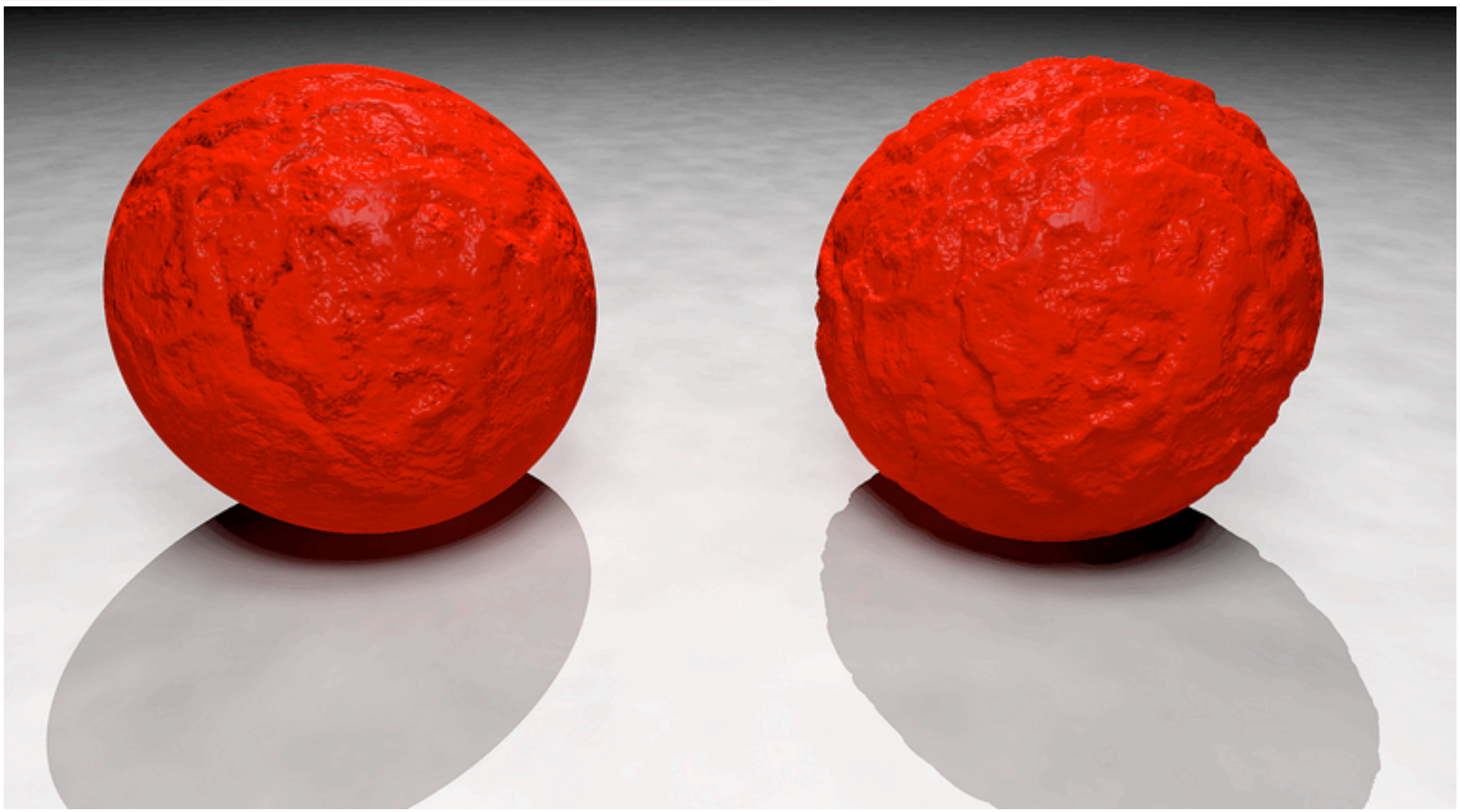
Slide credit: Pat Hanrahan

Bump mapping



Bump mapping:
Displace surface in direction of normal (for lighting calculations)

[Image credit: Wikipedia]



Normal mapping

Modulate interpolated surface normal

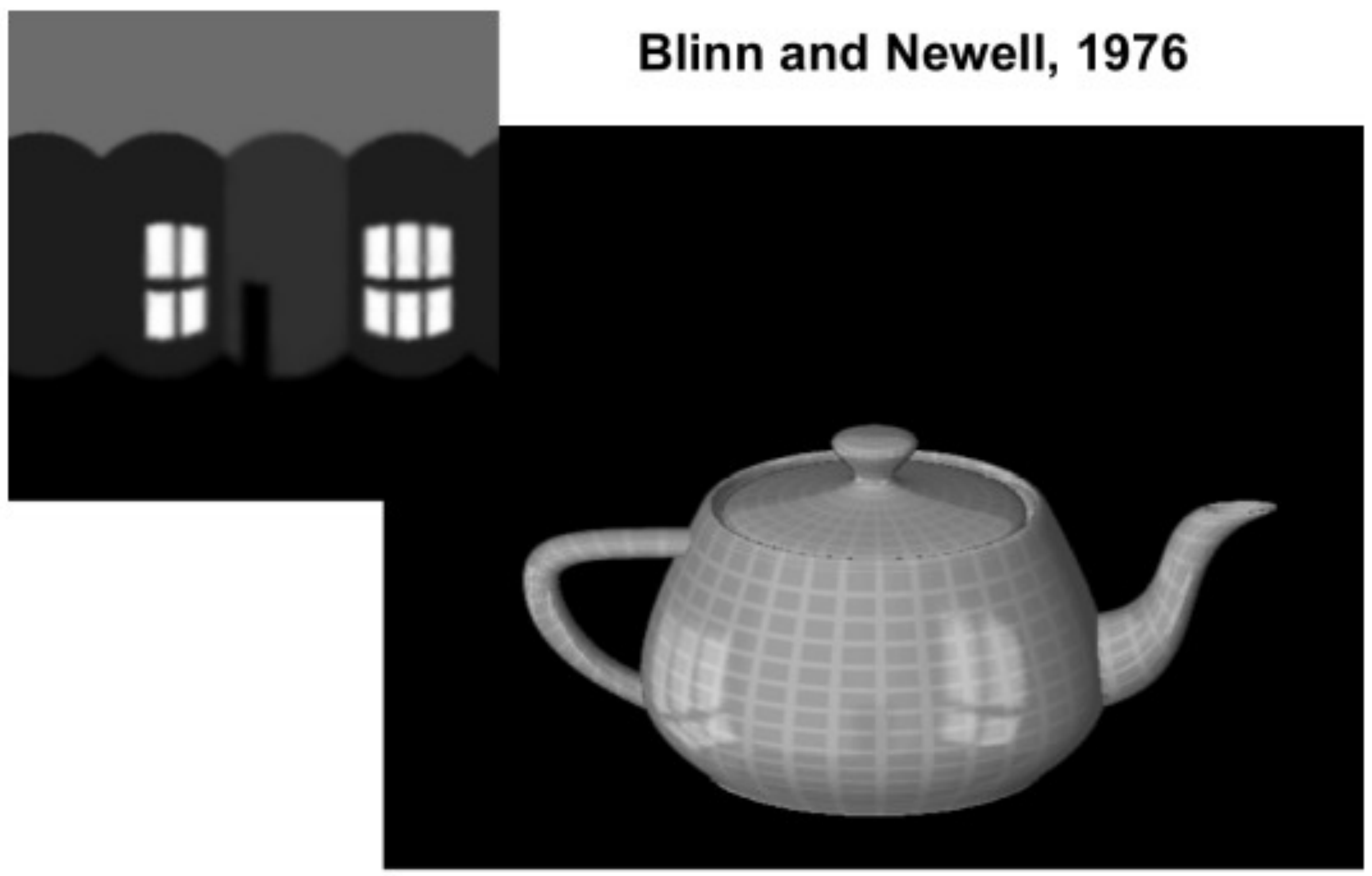
(nx,ny,nz) = (r,g,b)

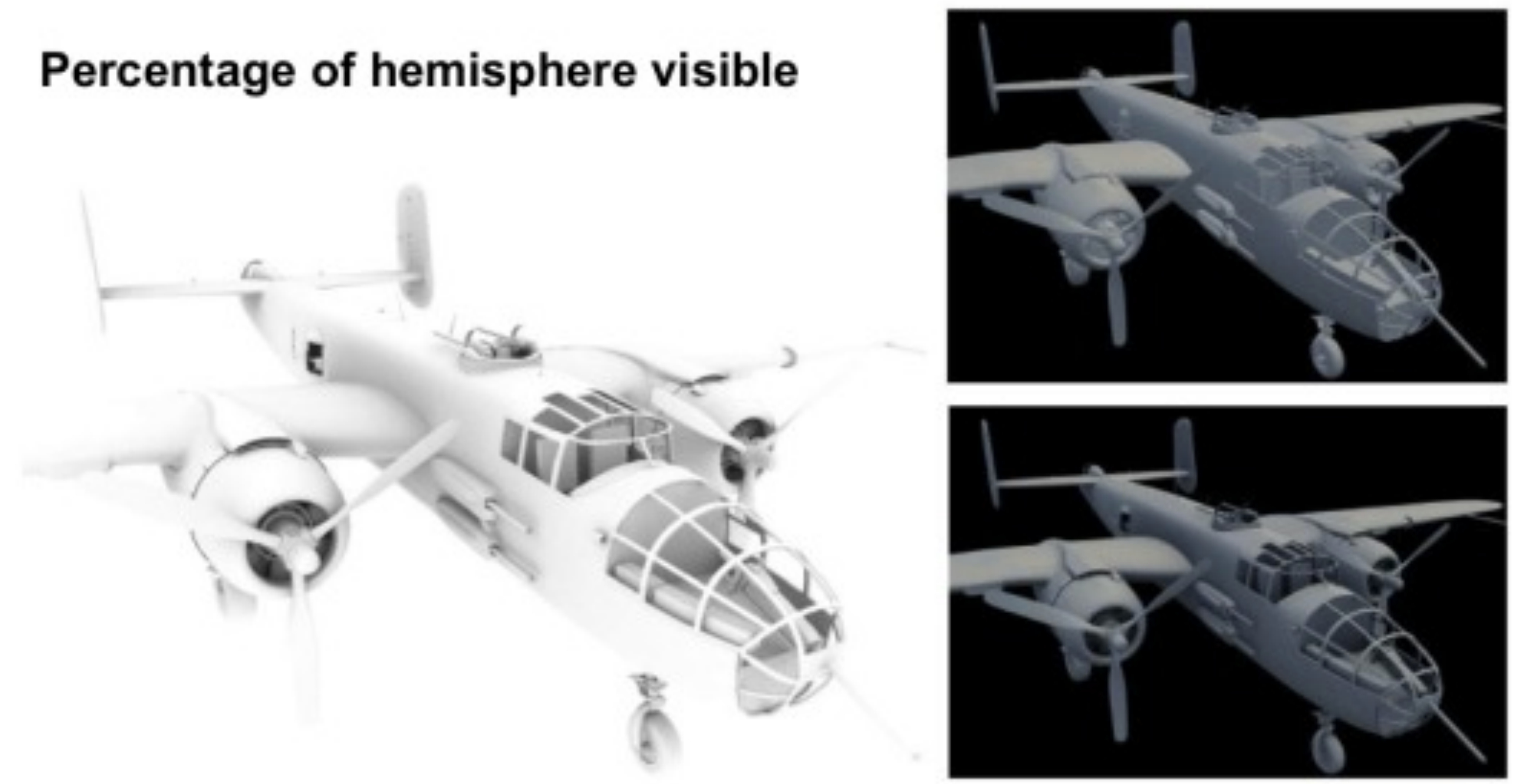
Slide credit: Pat Hanrahan

Many uses for textures

Store precomputed lighting

Blinn and Newell, 1976

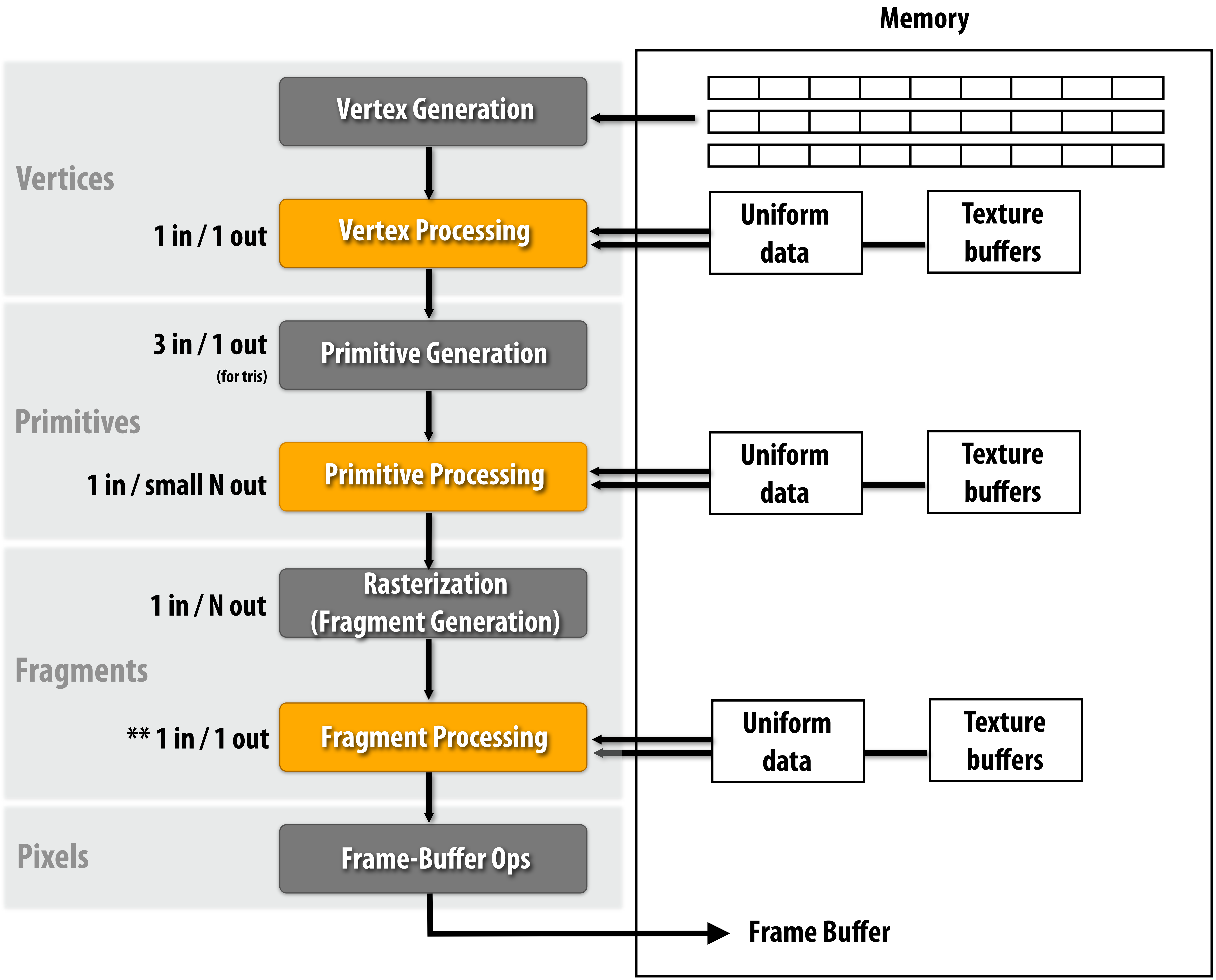




From Production ready global illumination, Hayden Landis, ILM

Slide credit: Pat Hanrahan

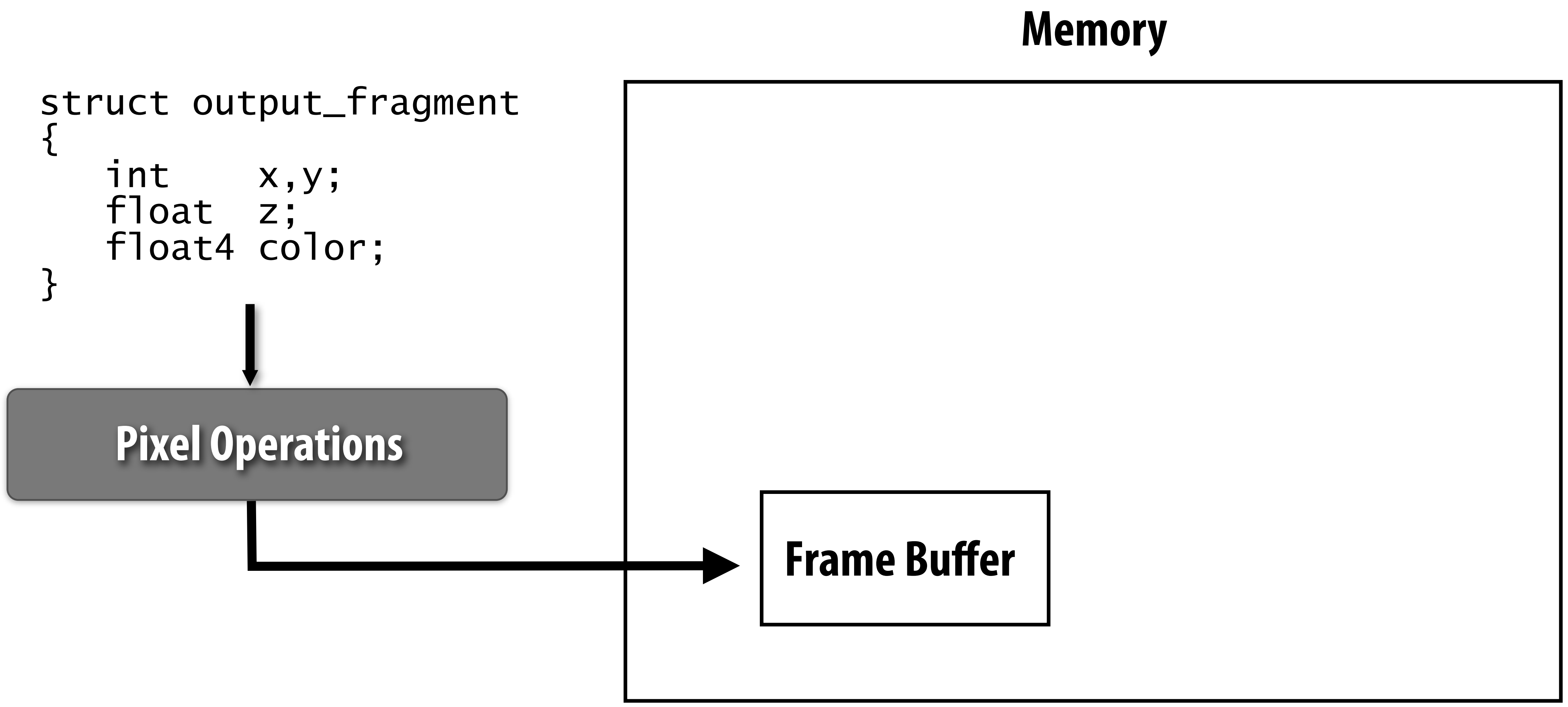
The graphics pipeline



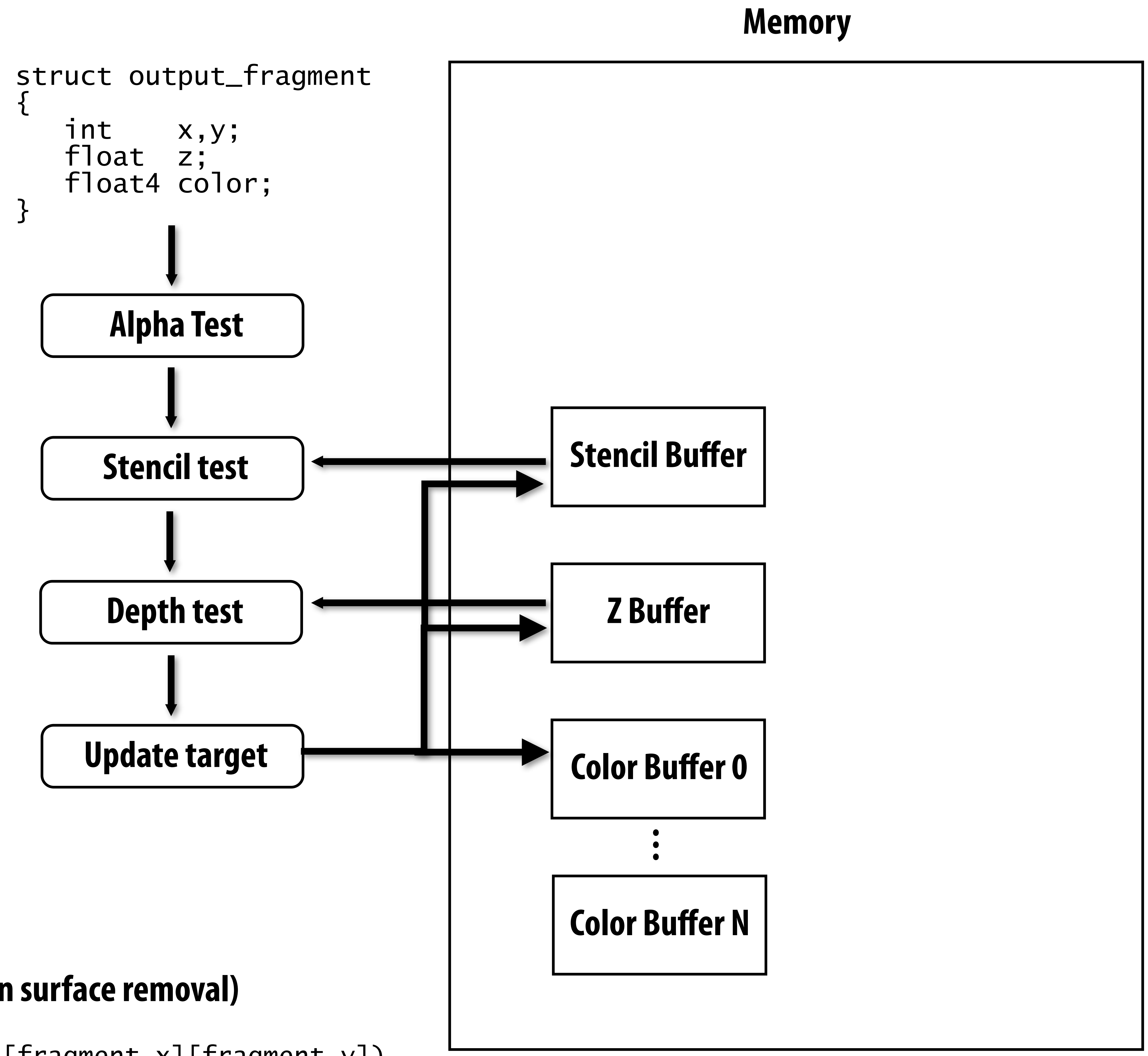
** can be 0 out

Frame-buffer operations

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



Frame-buffer operations



Depth test (hidden surface removal)

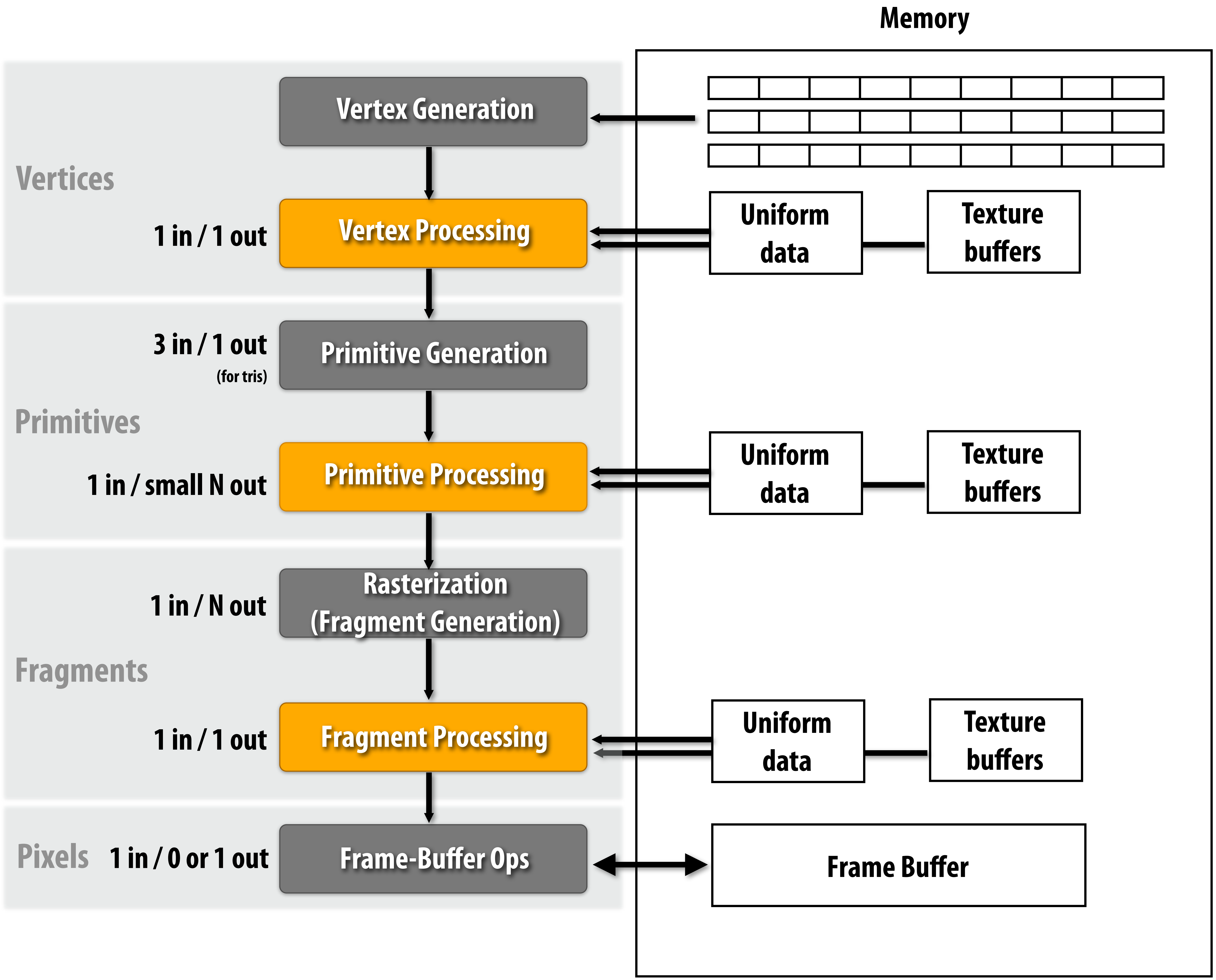
```
if (fragment.z < zbuffer[fragment.x][fragment.y])
{
    zbuffer[fragment.x][fragment.y] = fragment.z;
    colorbuffer[fragment.x][fragment.y] =
         blend(colorbuffer[fragment.x][fragment.y], fragment.color);
}
```

Frame-buffer operations

Depth test (hidden surface removal)

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

The graphics pipeline



Programming the pipeline

- Issue draw commands ⟶ frame-buffer 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

Feedback loop

- Issue draw commands ⟶ frame-buffer contents change

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

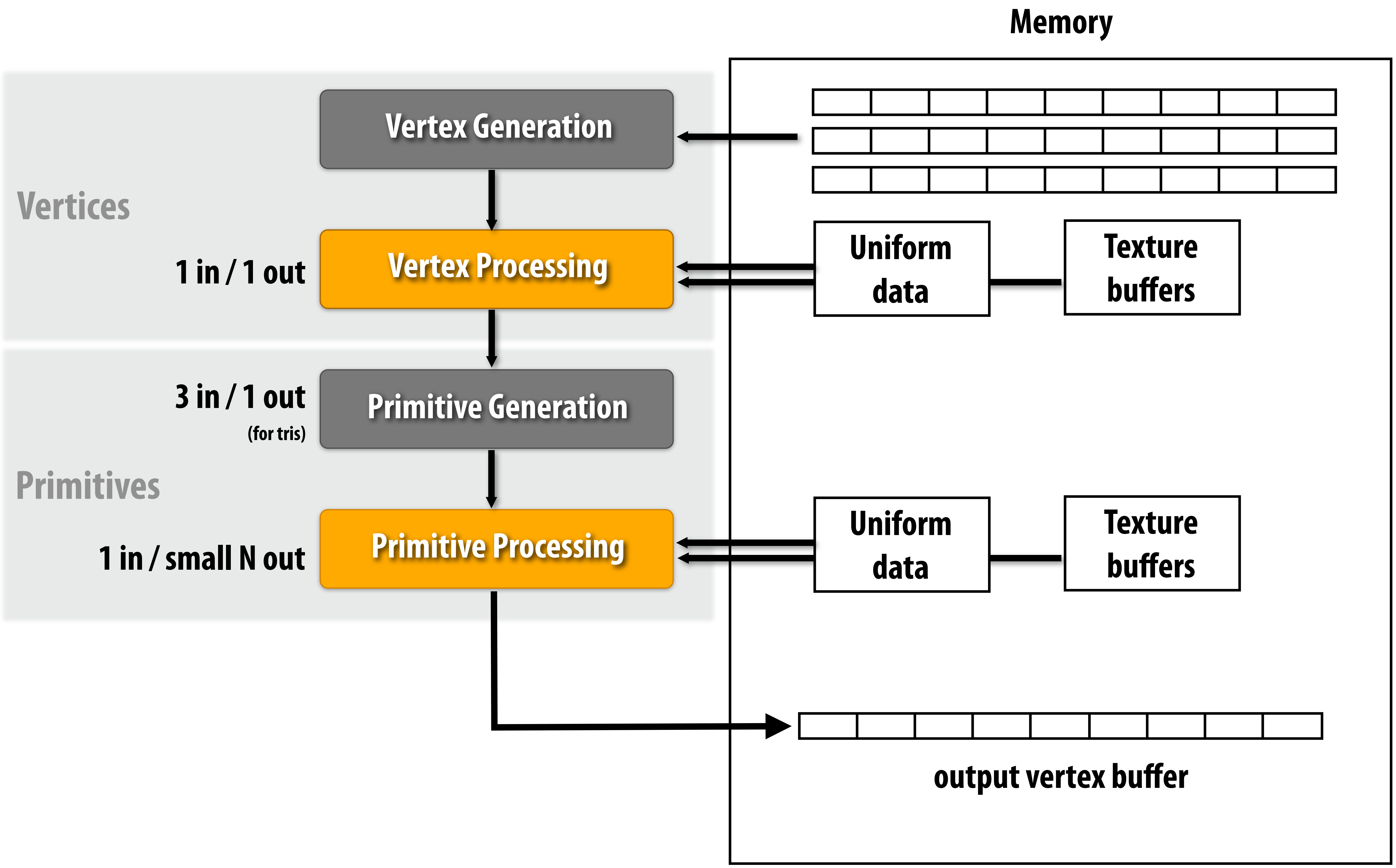
Key idea for:
shadows
environment mapping
post-processing effects

1000-1500 draw calls per frame

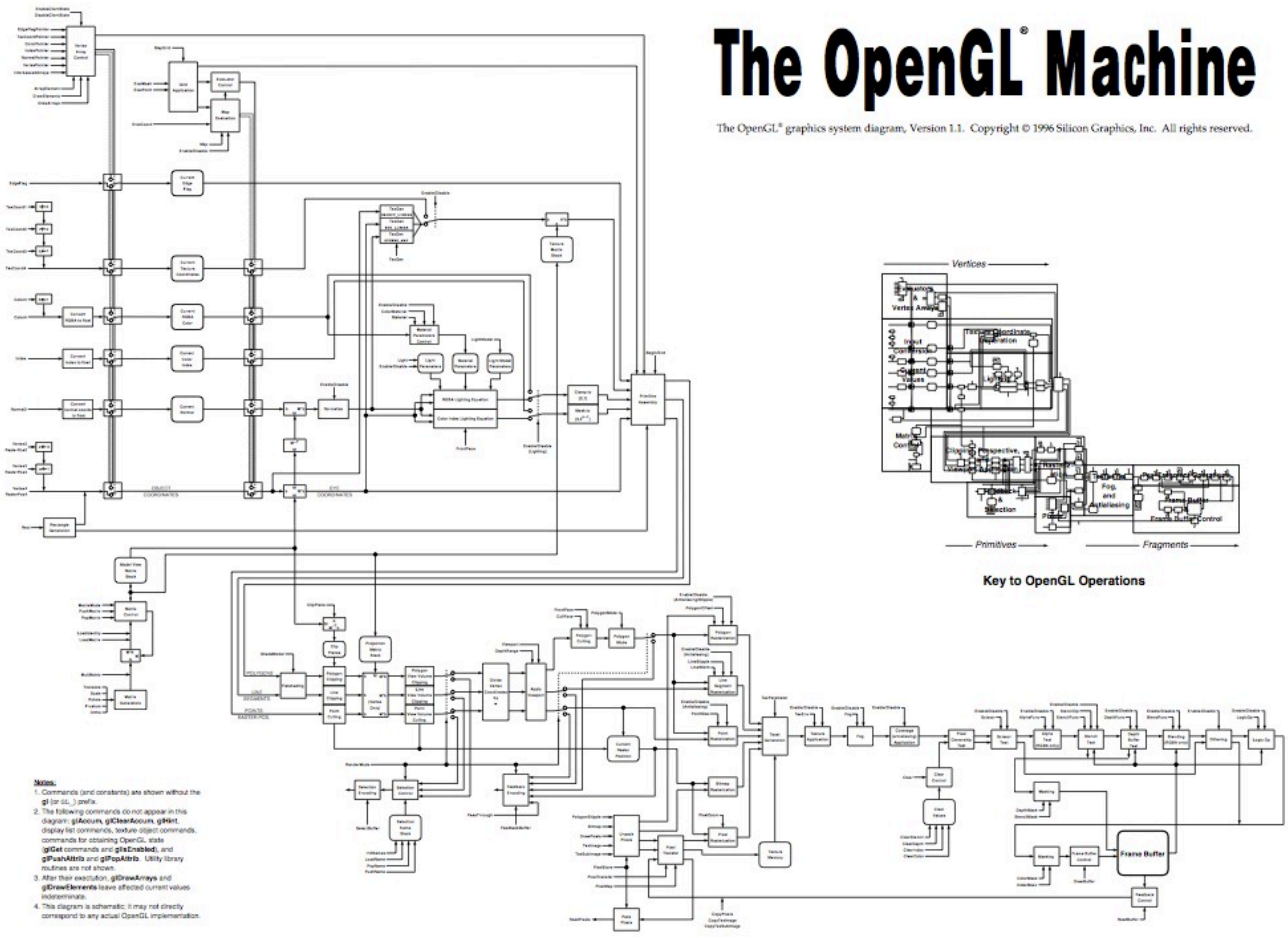
(source: Johan Andersson, DICE -- circa 1998)

Feedback loop 2

- Issue draw commands ⟶ save intermediate geometry

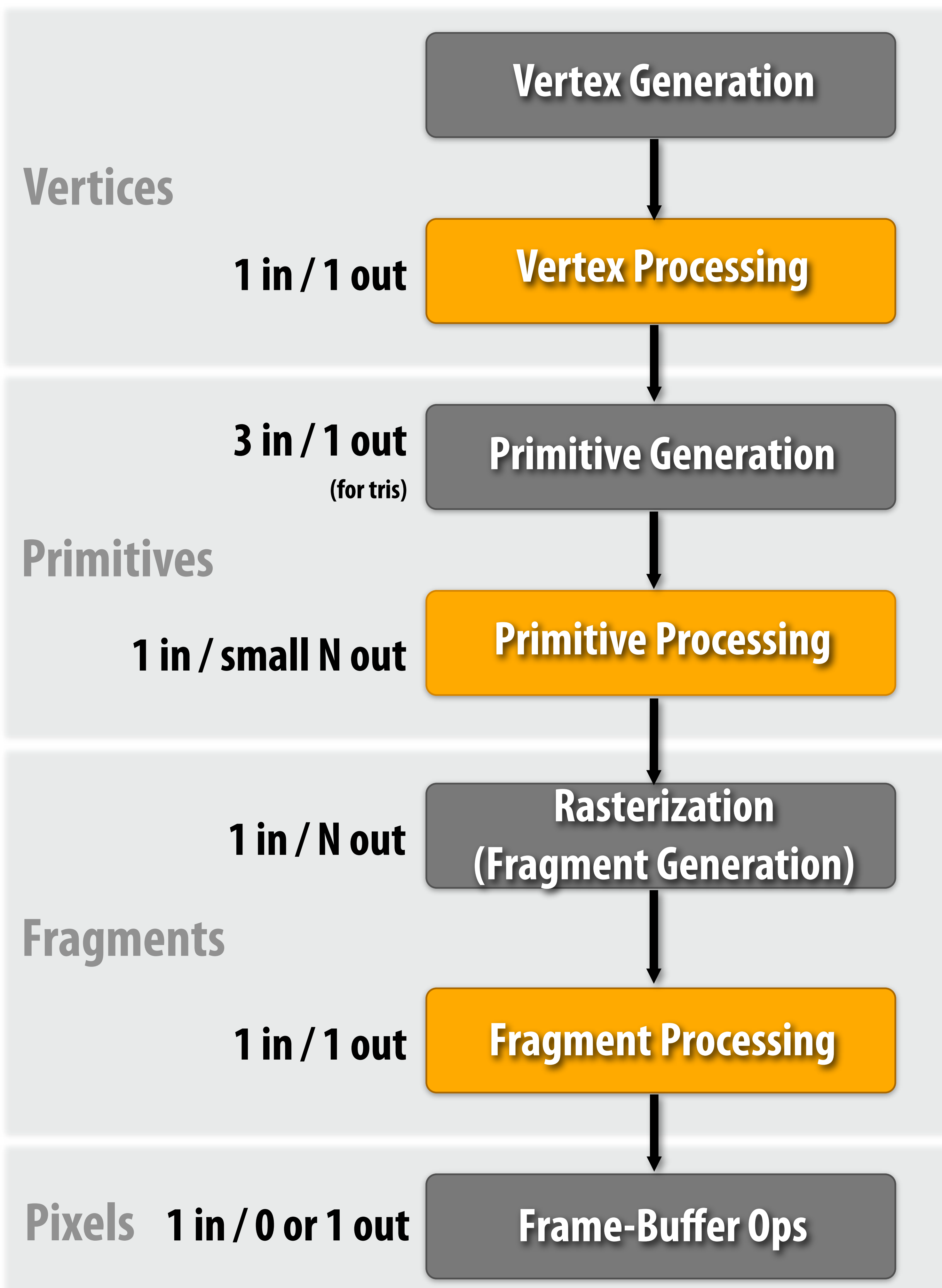


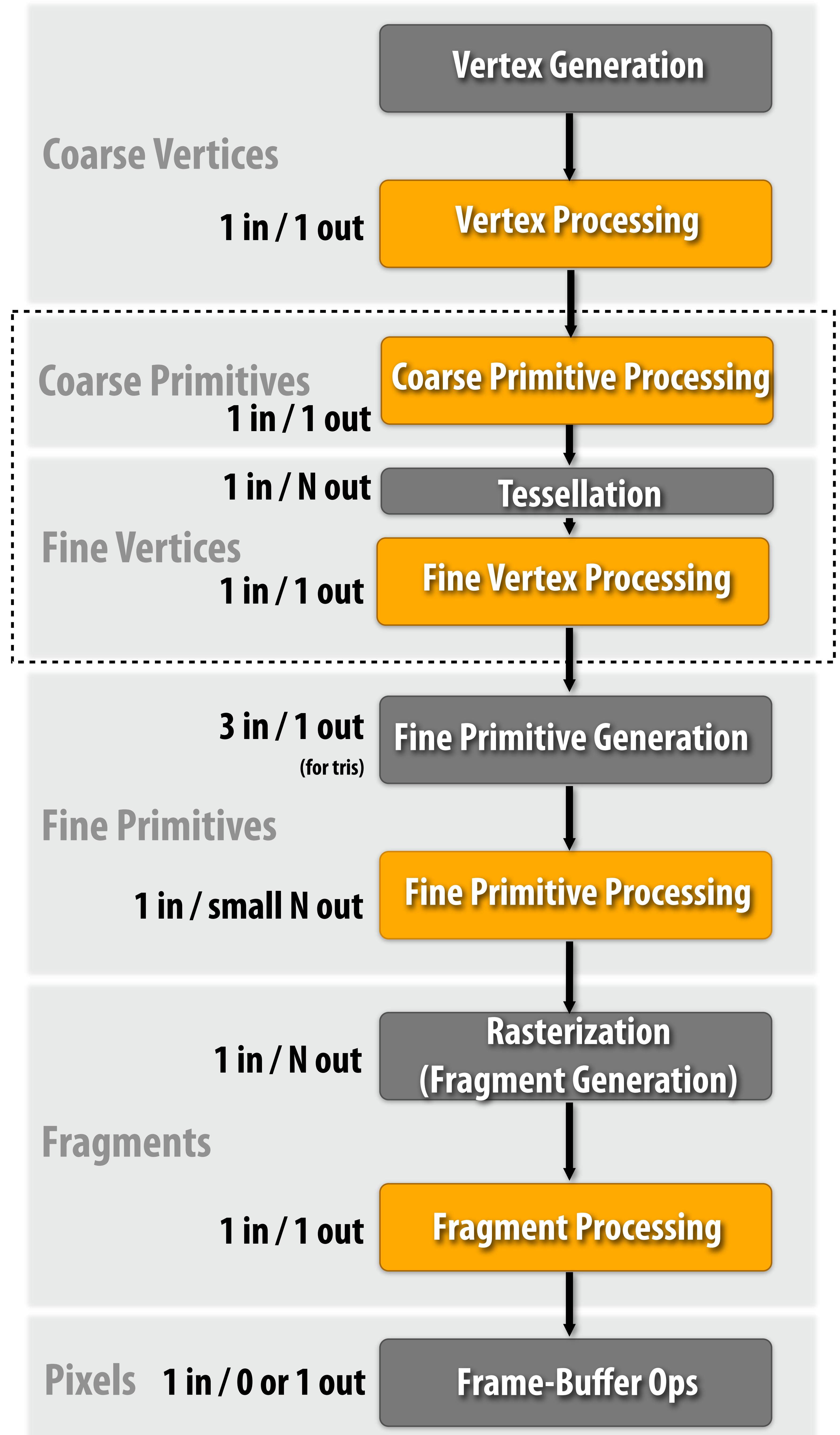
OpenGL state diagram (OGL 1.1)



Graphics pipeline with tessellation

(OpenGL 4, Direct3D 11)





Graphics pipeline characteristics

- **Level of abstraction**
  - Declarative, not imperative

    ("Draw a triangle, using this fragment program, with depth testing on" vs. "draw a cow made of marble on a sunny day")
  - Programmable stages give large amount of application flexibility
  - Configurable: Turn stages on and off, feedback loops
  - Low enough to allow application to implement many techniques, high enough to abstract over radically different implementations

Graphics pipeline characteristics

- **Orthogonality of abstractions**
  - All vertices treated the same
    - Vertex programs work for all primitive types
  - All primitives turned into fragments
    - Fragment programs oblivious to primitive type
    - Hidden surface remove via z-buffering: oblivious to primitive type
    - Same is true for anti-aliasing (will be discussed later)

Graphics pipeline characteristics

- **How is it designed for performance/scalability?**
  - [Reasonable low level]: low abstraction distance
  - Constraints on pipeline structure
    - Constrained data-flows between stages
    - Fixed-function stages
    - Independent processing of each data element (enables parallelism)
  - Different frequencies of computation (per vertex, per primitive, per fragment)
    - Only perform work at the rate required
  - Keep it simple
    - Common intermediate representations
      - Triangles, points, lines
      - Fragments, pixels
    - Z-buffer algorithm
  - "Immediate mode system": processes primitives as it receives them (as opposed to buffering the entire scene)
    - Leave global optimization of <u>how</u> to render scene to application (scene graph)

Graphics pipeline characteristics

- **What it DOES NOT do**
  - Modern OpenGL has no concept of lights, materials, modeling transforms
    - Only vertices, primitives, fragments, pixels, and STATE: buffers and shaders
  - No concept of scene
    - No global effects (must be implemented using multiple draw calls by application: e.g, shadow maps)
  - No I/O, window management

Perspective from Kurt Akeley

- Does the system meet original design goals, and then do much more than was originally imagined?
  - Simple, orthogonal concepts
  - Amplifier effect

- Often you've done a good job if no one is happy ;-)

  (you still have to meet design goals)

Readings

- M. Segal and K. Akeley. The Design of the OpenGL Graphics Interface
- D. Blythe. The Direct10 System. SIGGRAPH 2006