Lecture 1:

Course Introduction +
Review of the Real-Time Graphics Pipeline

Visual Computing Systems
CMU 15-869, Fall 2014

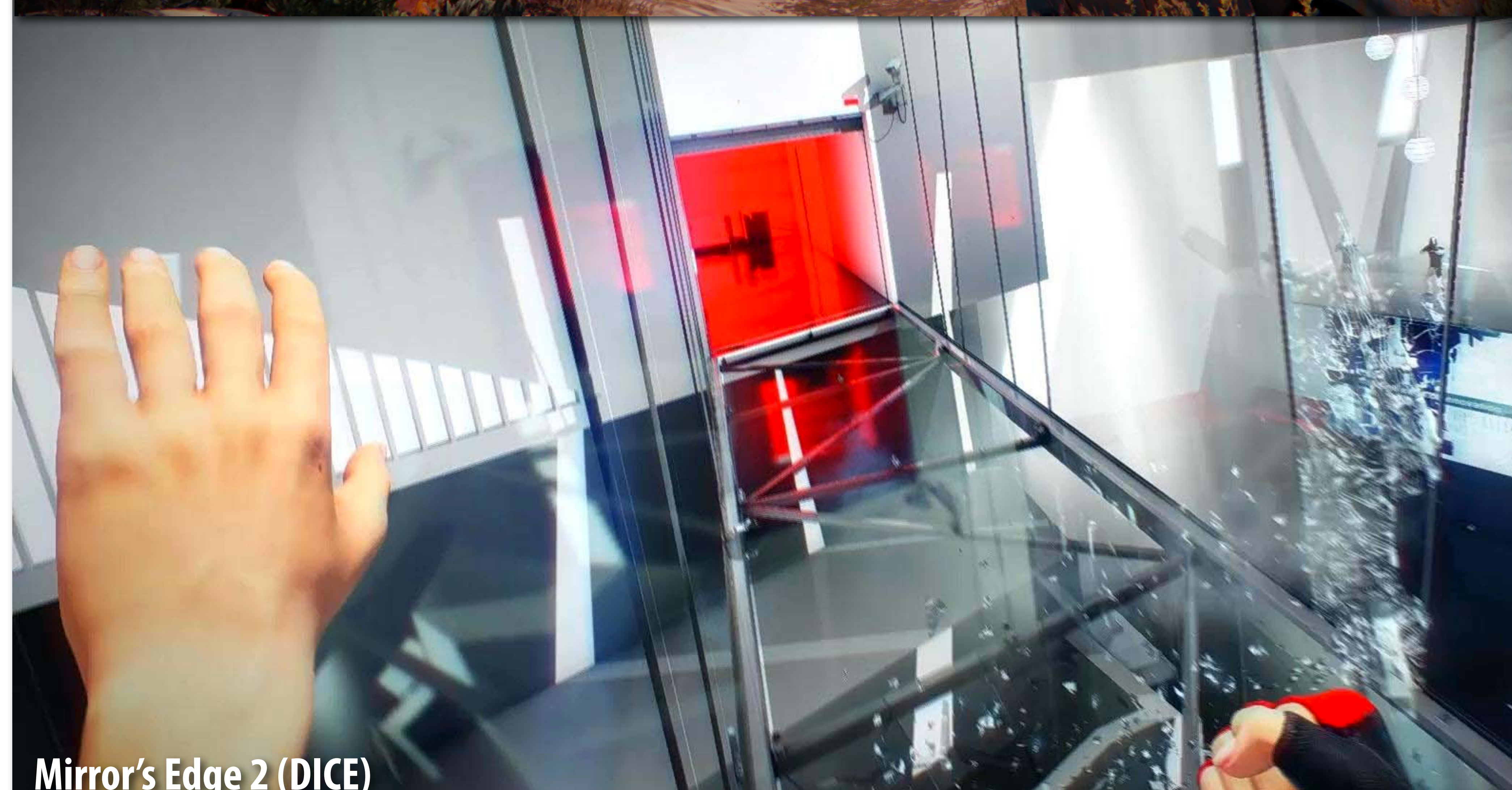
Why does this course exist?

Many applications that drive the need for high efficiency computing involve visual computing tasks

First thing that comes to mind: 3D "AAA" games
Efficiency gets you: more advanced graphics at 30 fps
Result: multi-TFLOP GPUs

Simple mobile games
Efficiency gets you: don't run down the battery
Result: different rendering algorithms

Destiny (Bungie)

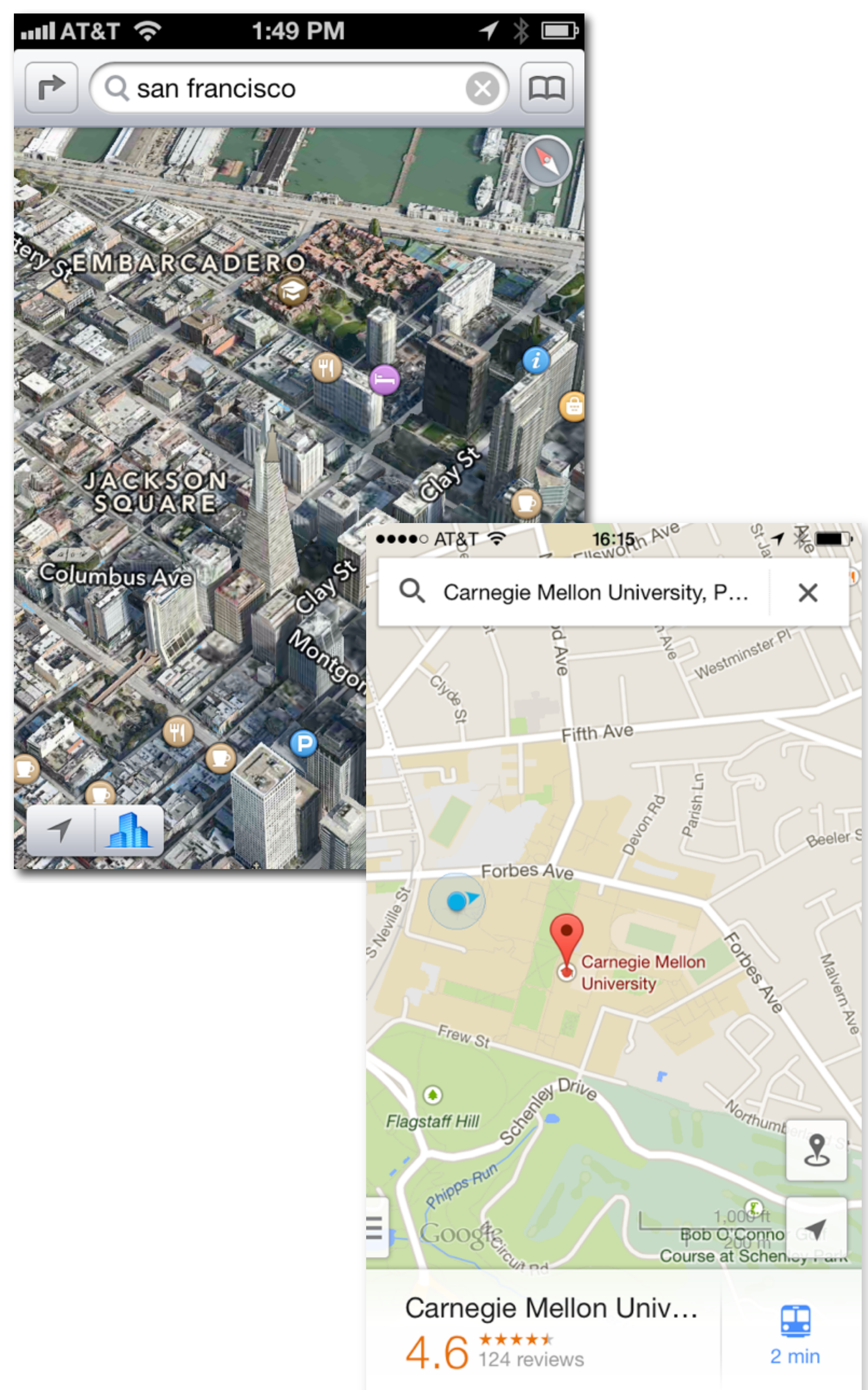

Mirror's Edge 2 (DICE)

Many applications that drive the need for high efficiency computing involve visual computing tasks

Record/play HD Video

2D rendering to "Retina" resolution displays*: maps, browsers, 60 fps touch UIs

*** Maps apps and web content have 3D rendering capabilities as well.**



High pixel count sensors and displays

Nokia Lumina smartphone camera:
41 megapixel (MP) sensor

Nexus 10 Tablet: 2560 x 1600 pixel display (~ 4MP)
(higher pixel count than 27” Apple display on my desk)

4K TV

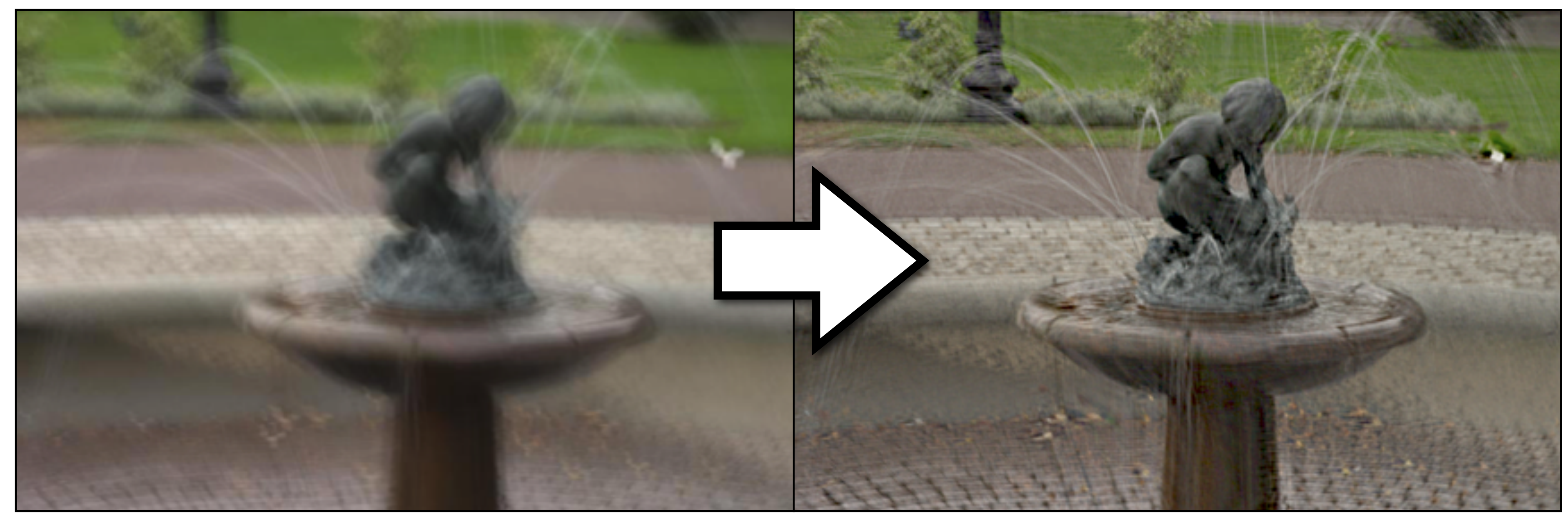
Rendering for VR and light-field displays: need for much higher pixel counts

Computational photography:

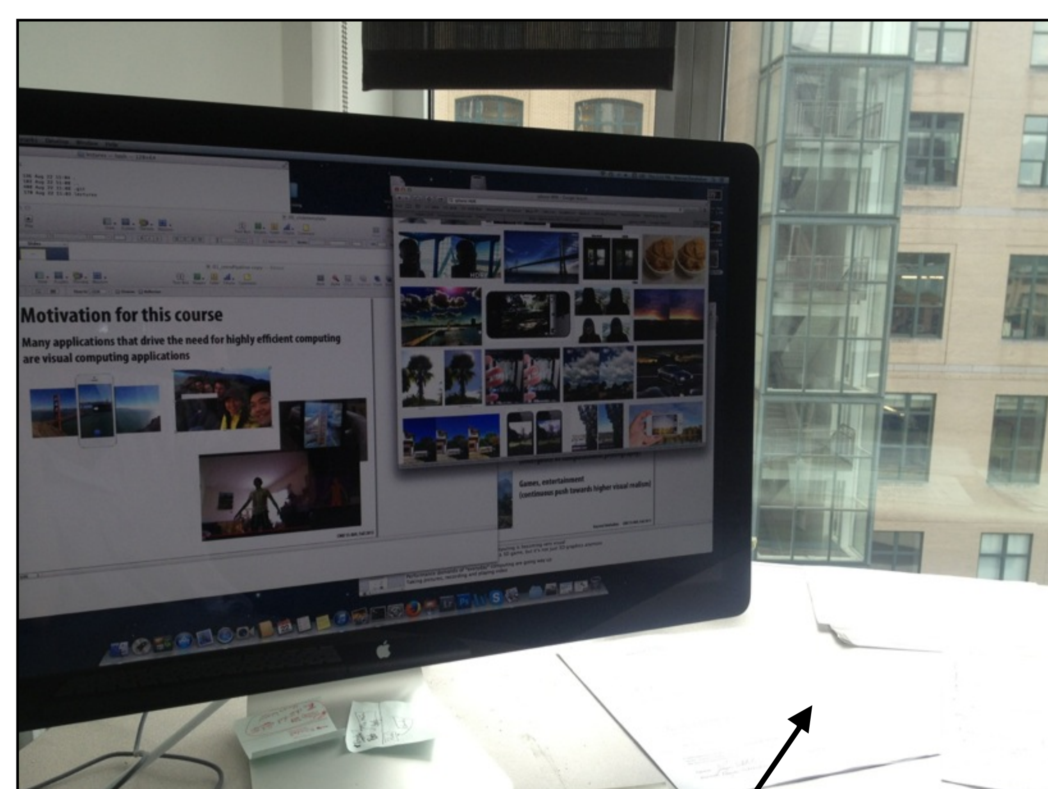
Current focus: achieve high-quality pictures with a lower-quality smart phone lenses/sensors through the use of image analysis and processing.

Automatic panorama:

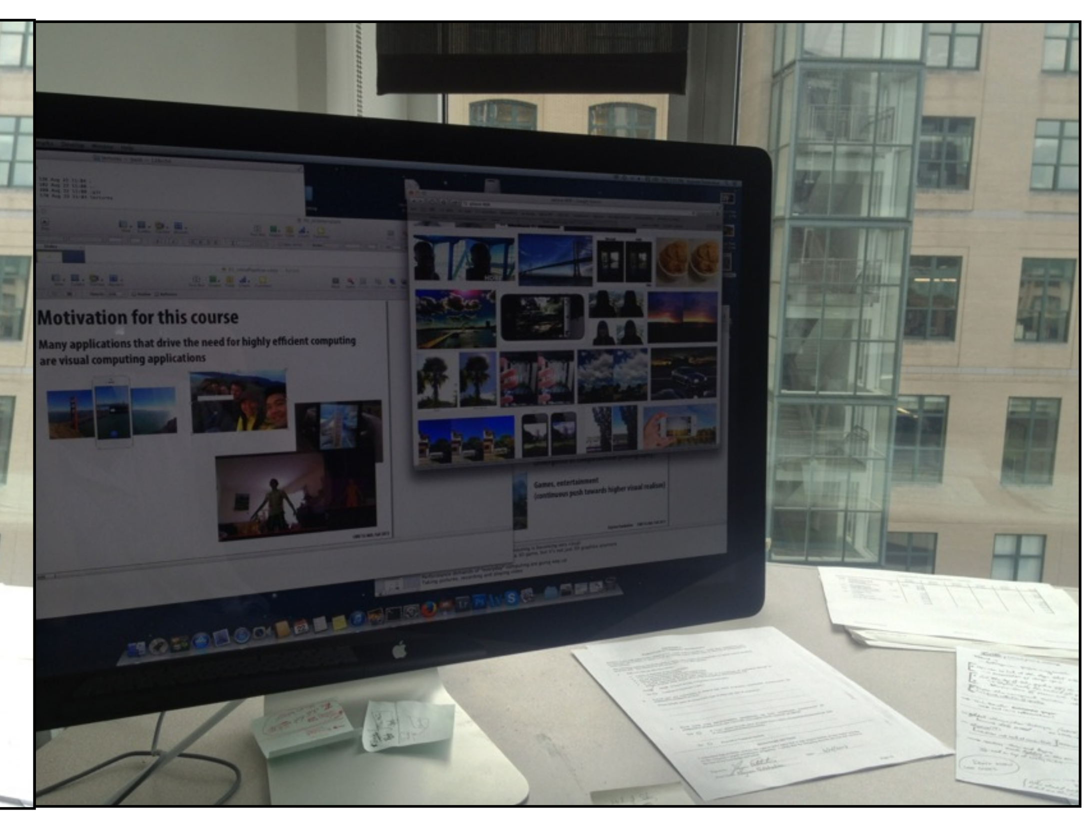
Remove camera shake:



High dynamic range (HDR) imaging:



Traditional photograph: part of image is saturated due to overexposure



HDR image: combine multiple exposures so image detail in both light and dark areas is preserved

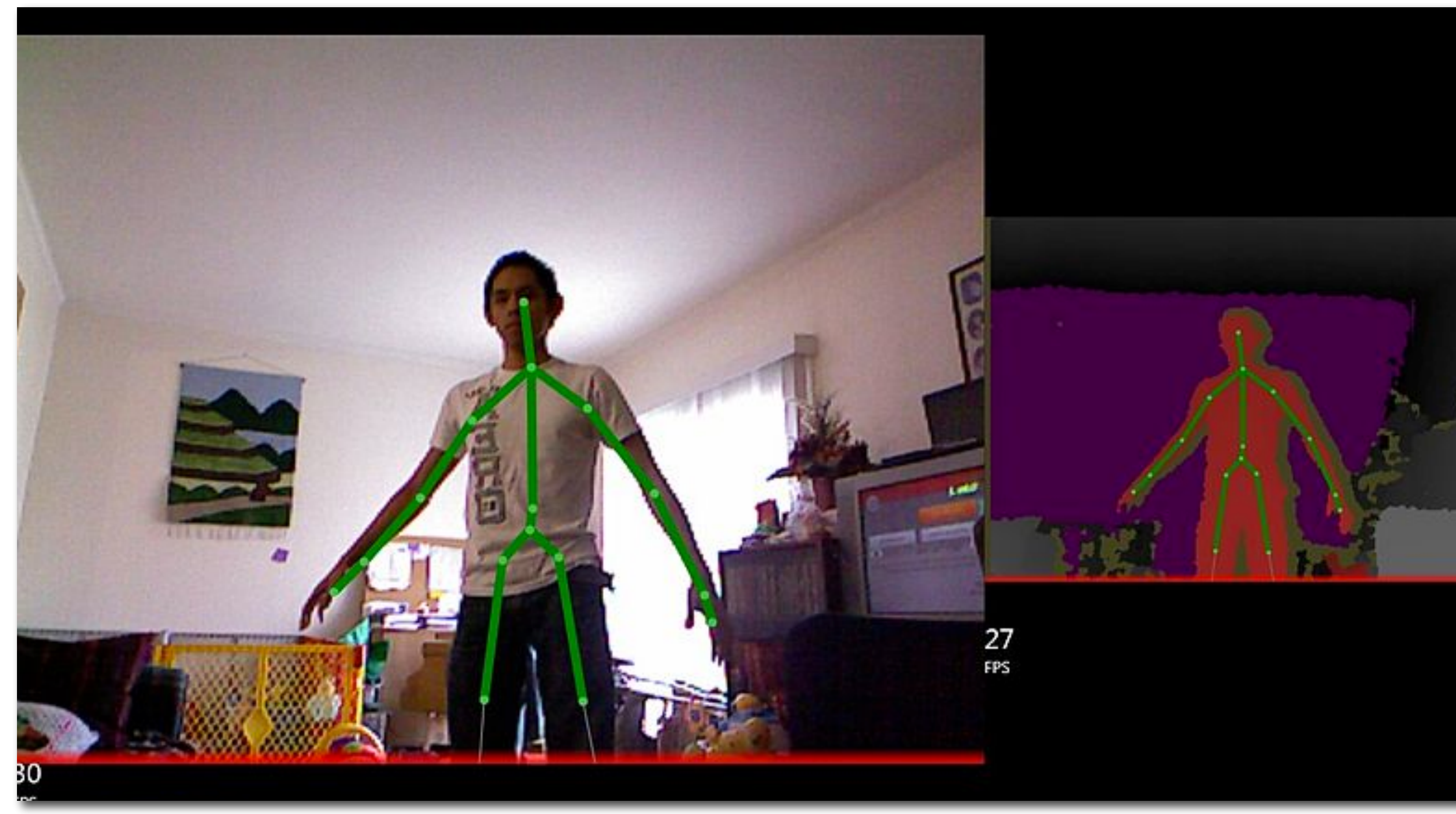
Image interpretation and understanding:

Extracting information from images recorded by ubiquitous image sensors

Big area of interest at both mobile device and data-center scales.

Auto-tagging, face (and smile) detection

Kinect: character pose estimation



Google Goggles: object identification search by image

Collision anticipation, obstacle detection

Enabling current and future visual computing applications requires focus on system efficiency

In this class we are going to think like architects. Which means we're going to talk a lot about a system's goals and about its constraints.

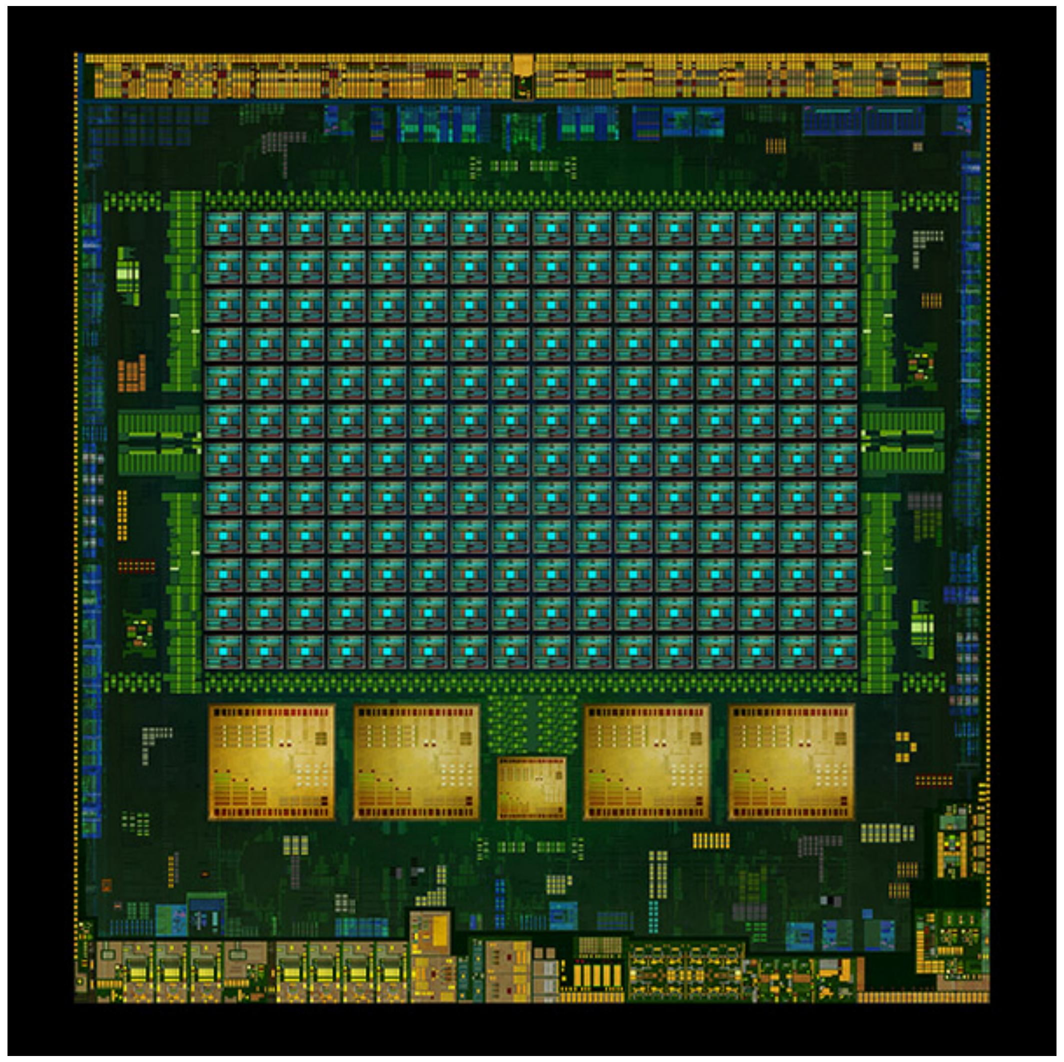
Example goals:

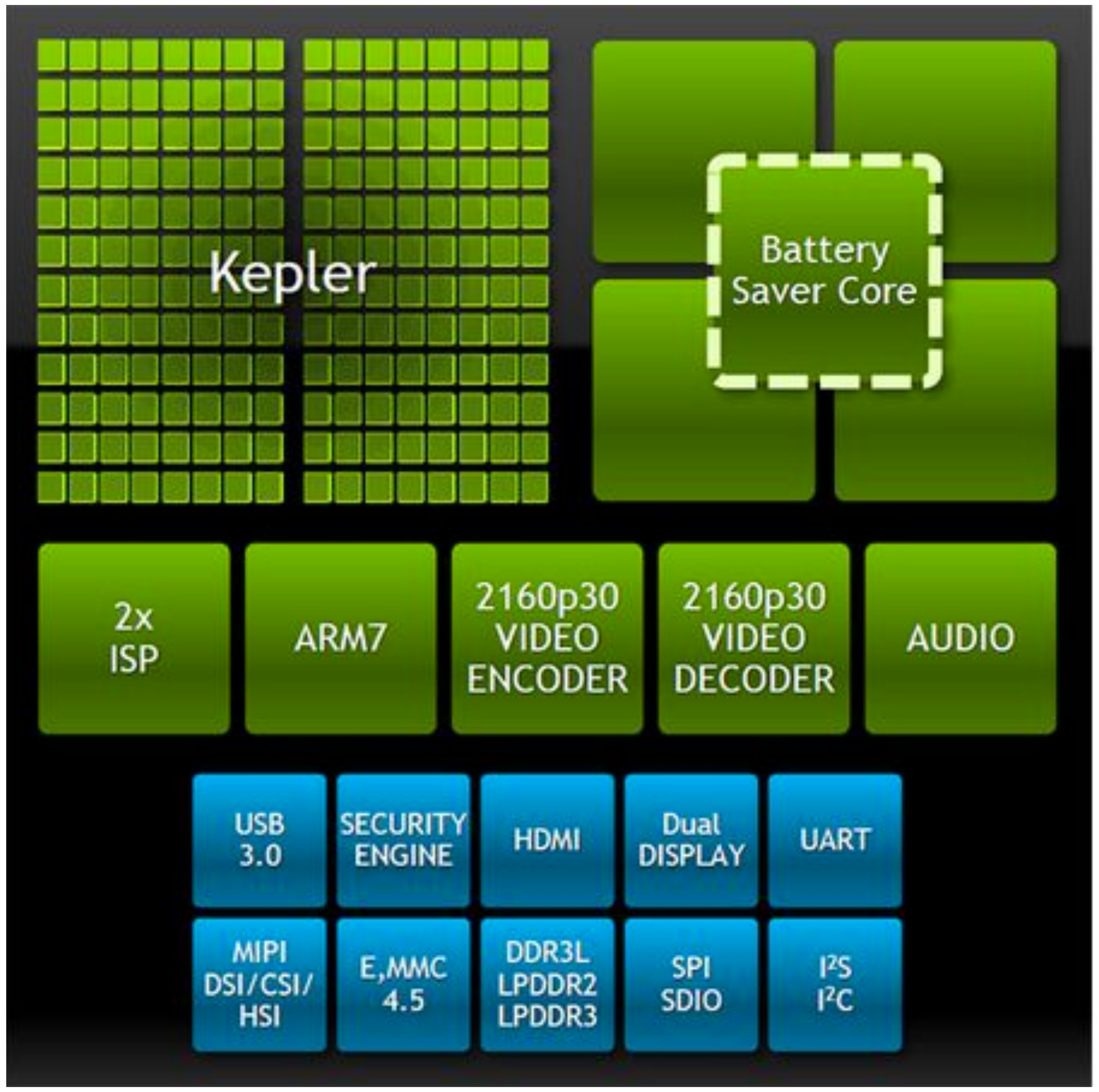
- Real-time rendering of a one-million polygon scene at 30 fps on a high-res display
- Provide interactive user feedback when acquiring a panorama
- 1080p video recording for one hour per phone charge

Example constraints:

- Chip die area (chip manufacturing cost)
- System design complexity
- Preserve easy application development effort
- Backward compatibility with existing software
- Power

Parallelism and specialization in HW design





Example: NVIDIA Tegra K1

Four high-performance ARM Cortex A15 CPU cores for applications

One low performance (low power) ARM CPU core

One Kepler SMX core (to run graphics shaders and CUDA programs)

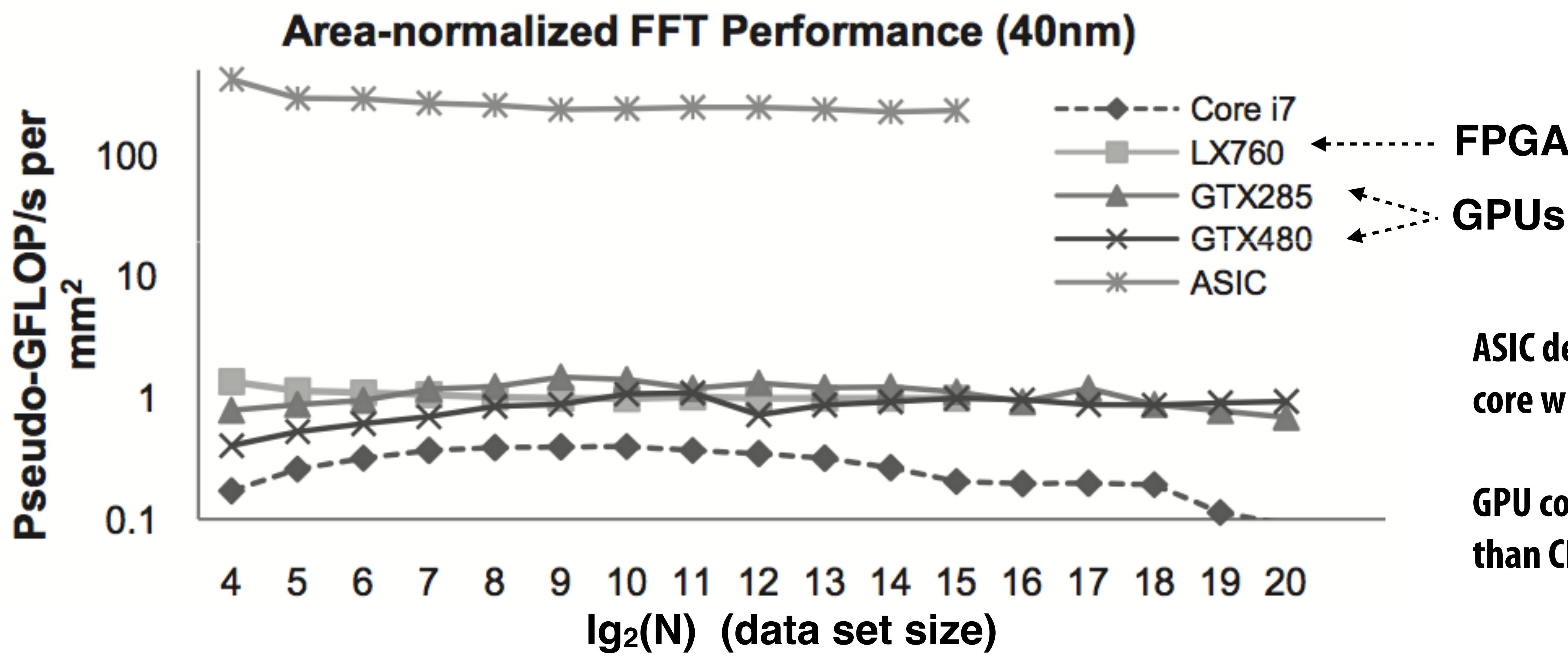
Fixed-function HW blocks for 3D graphics and image/video compression and camera image processing (image signal processor = ISP)

Design philosophy:
Run important workloads on the most efficient hardware for the job.

Other modern examples:

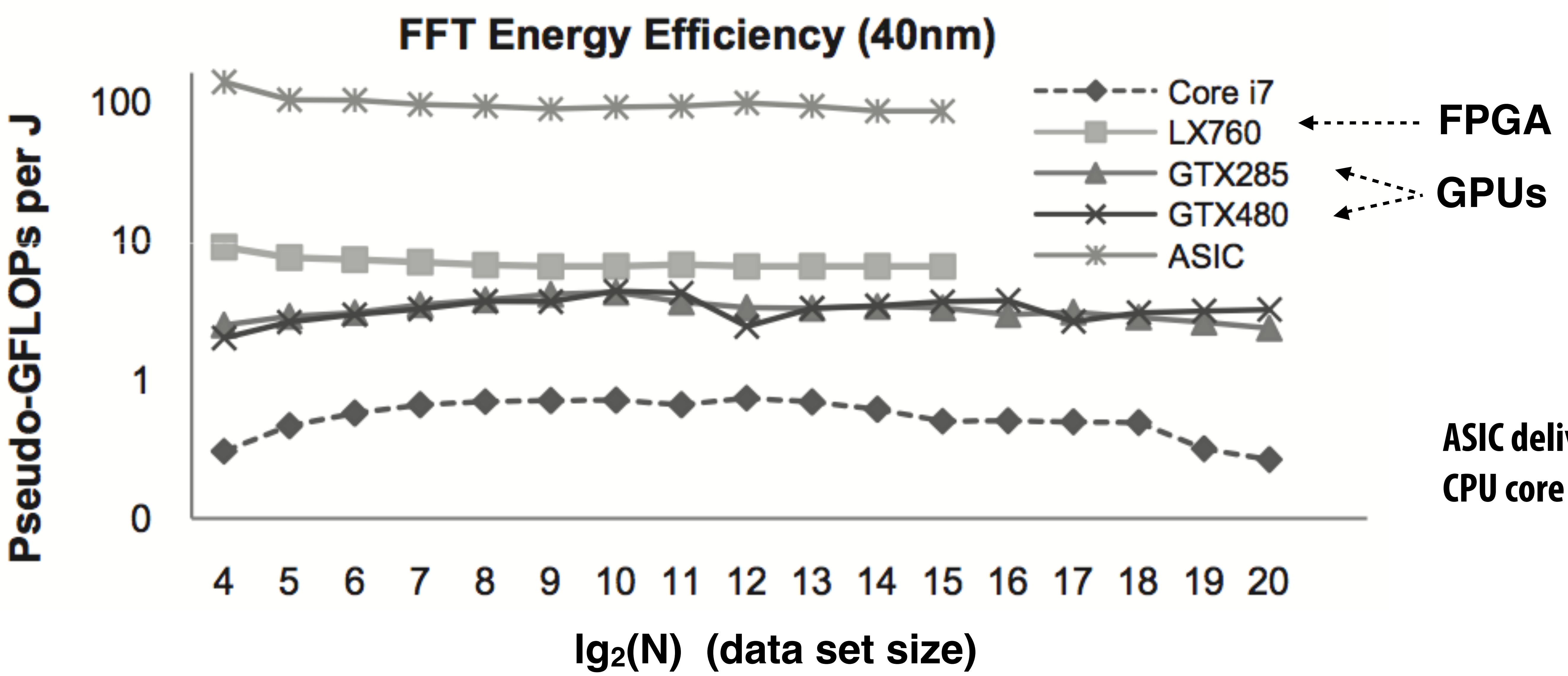
Apple A6X
Qualcomm Snapdragon

Specialized hardware is efficient!



ASIC delivers same performance as one CPU core with ~ 1/1000th the chip area.

GPU cores: ~ 5-7 times more area efficient than CPU cores.

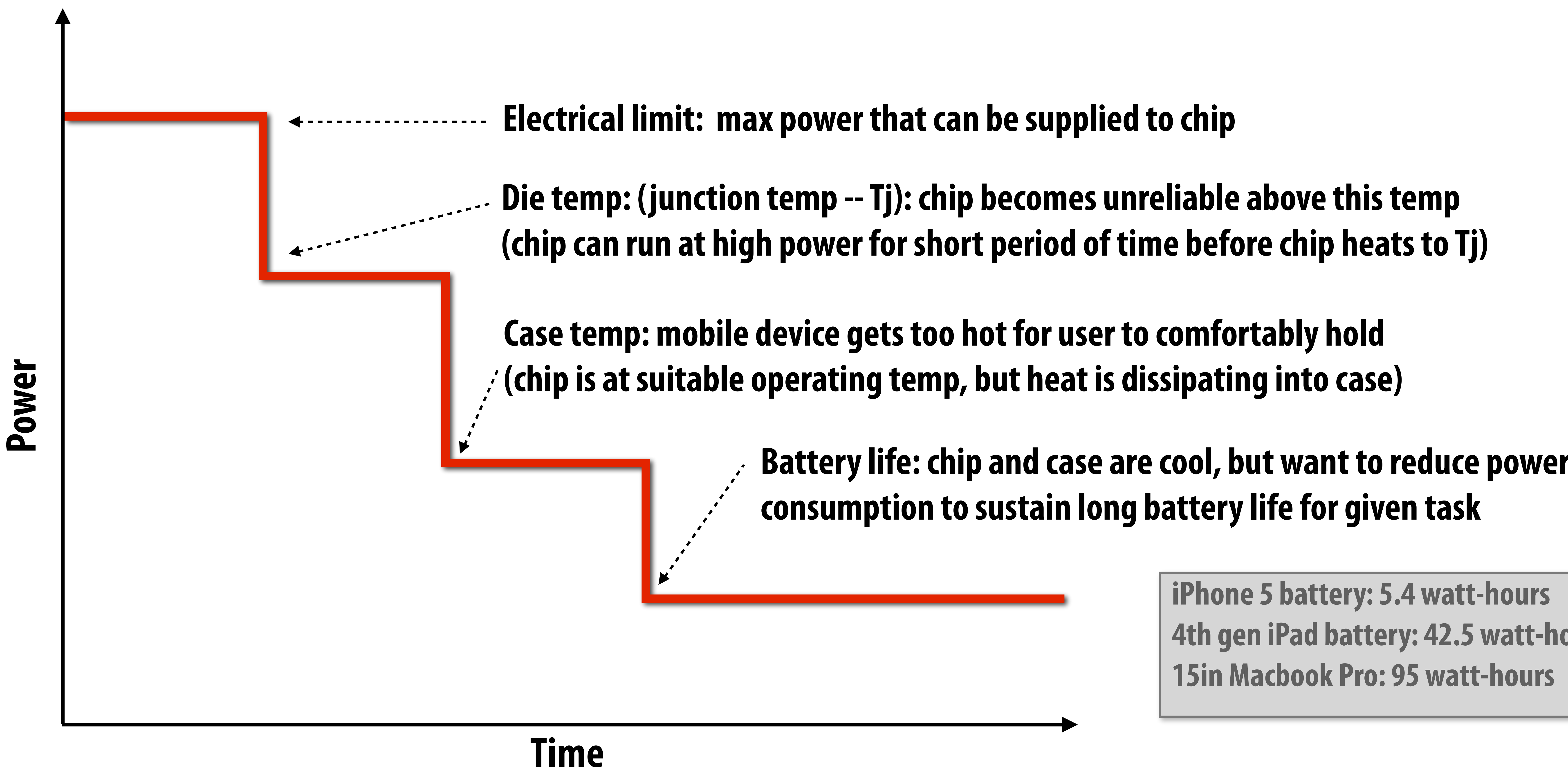


ASIC delivers same performance as one CPU core with only ~ 1/100th the power.

[Chung et al. MICRO 2010]

Limits on chip power consumption

- General rule: the longer a task runs the less power it can use
  - Processor's power consumption (think: performance) is limited by heat generated (efficiency is required for more than just maximizing battery life)



iPhone 5 battery: 5.4 watt-hours
4th gen iPad battery: 42.5 watt-hours
15in Macbook Pro: 95 watt-hours

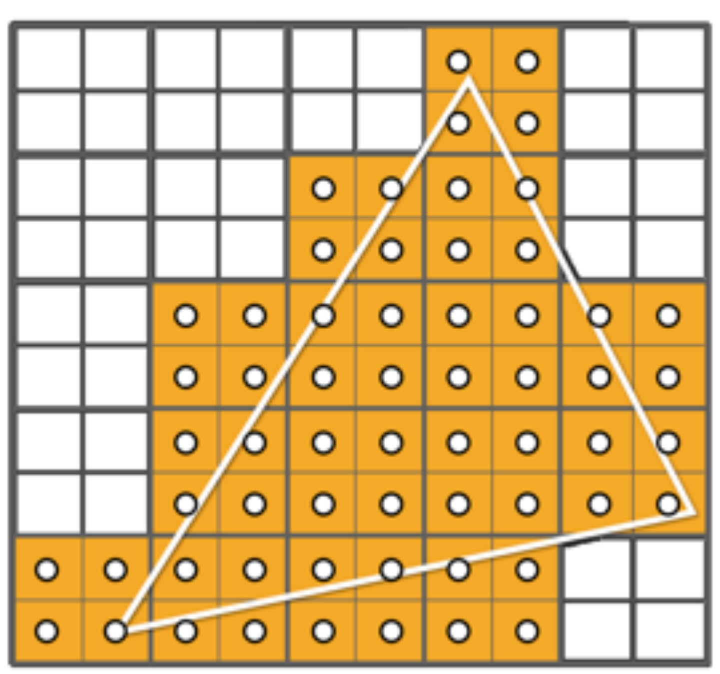
Credit: slide adopted by original slide from M. Shebanow

What this course is about

1. The characteristics/requirements of important visual computing workloads
2. Techniques used to achieve efficient system implementations

VISUAL COMPUTING WORKLOADS

Algorithms for 3D graphics, image processing, compression, etc.



mapping/scheduling

Parallelism
Exploiting locality
Minimizing communication

MACHINE ORGANIZATION

High-throughput hardware designs: Parallel and heterogeneous

DESIGN OF GOOD ABSTRACTIONS FOR VISUAL COMPUTING

choice of programming primitives
level of abstraction

In other words

It is about understanding the fundamental structure of problems in the visual computing domain…

To design better algorithms

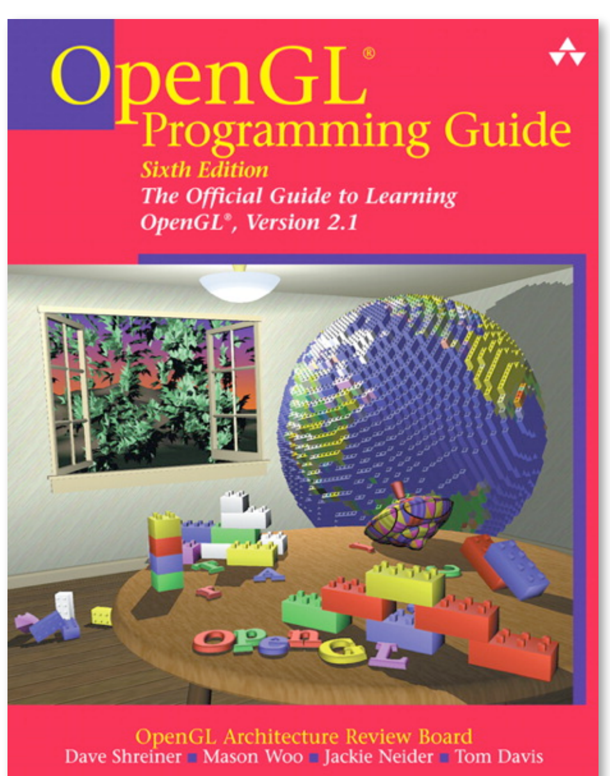
To build the most efficient hardware to run these applications

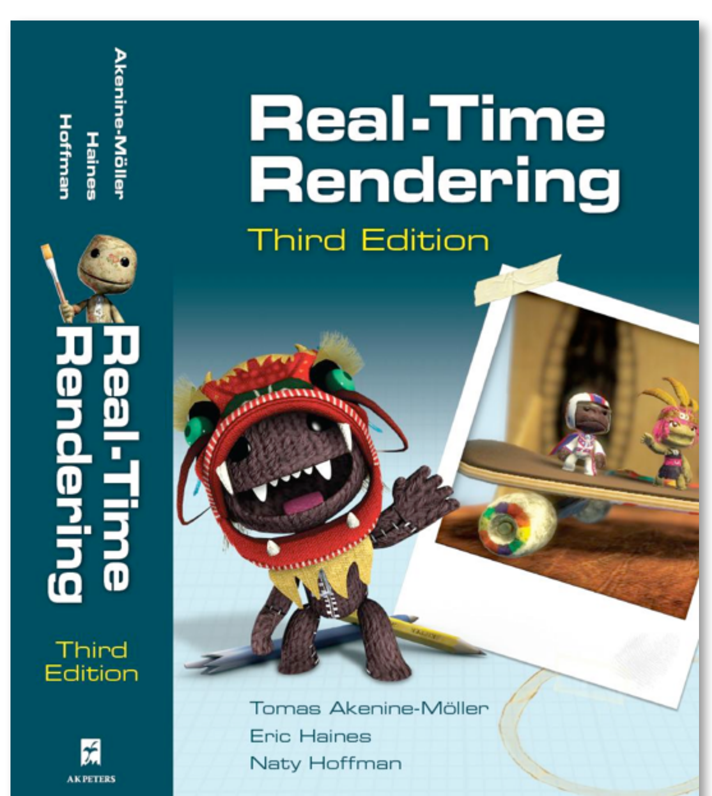
To design the right programming systems to make developing new applications simpler and also highly performant.

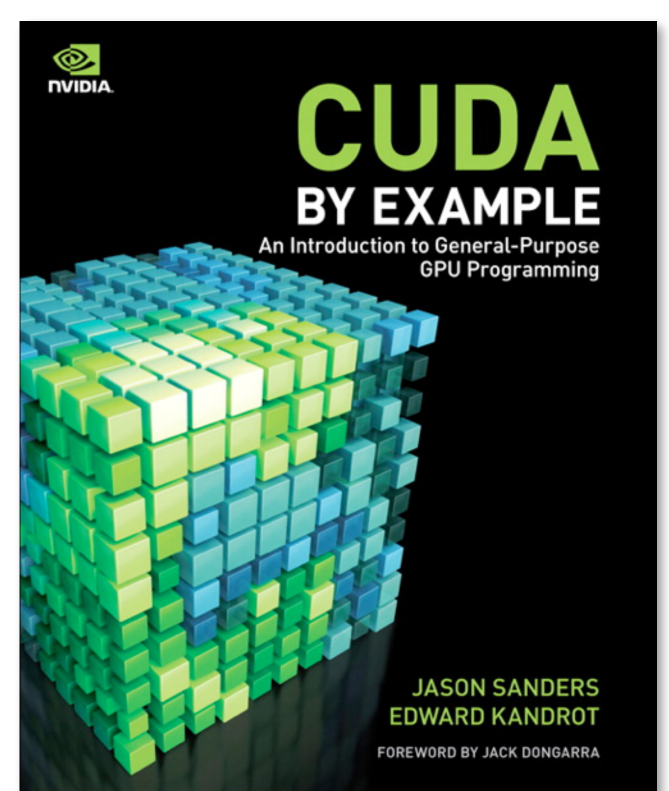
What this course is NOT about

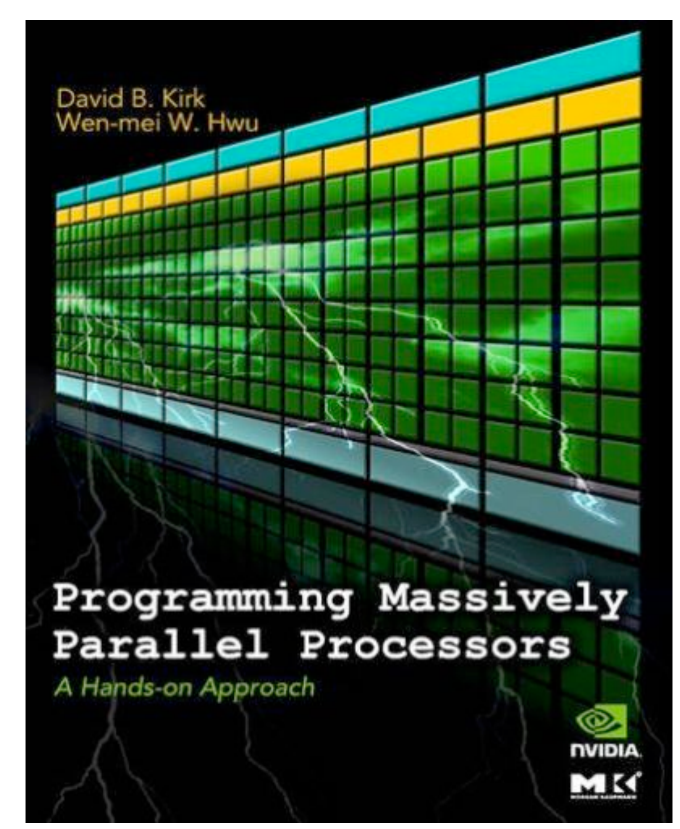
- This is not an [OpenGL, CUDA, OpenCL] programming course
  - But we will be analyzing and critiquing the design of these systems in detail
  - I expect you know these systems or pick them up as you go.

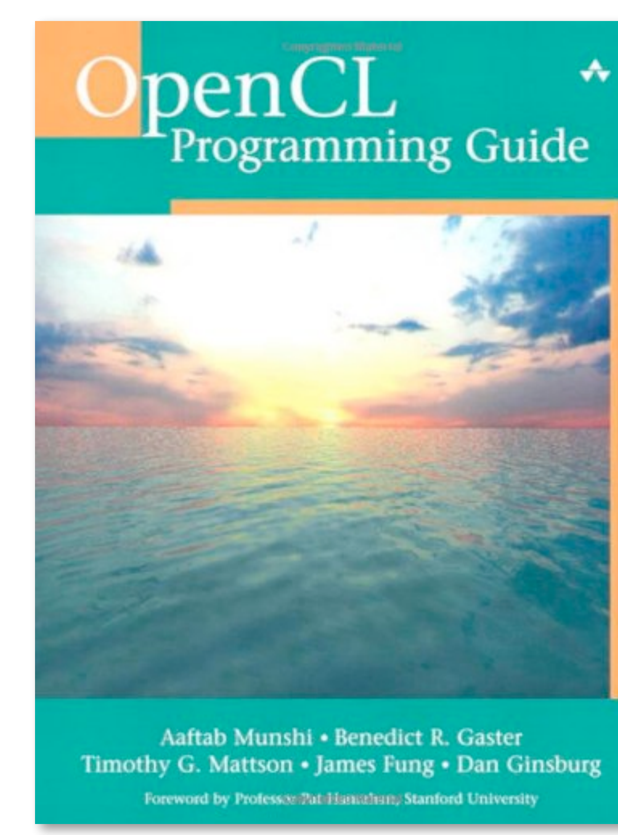
Many excellent references...











Course Logistics

Major course themes/topics

Schedule

Note, please consult the **suggested readings page** for a list of readings relevant to lecture topics.

Sep 8	Course Introduction + the Real-Time Graphics Pipeline Real-time rendering from a systems perspective
Sep 10	Graphics Pipeline Parallelization and Scheduling Characteristics of the pipeline workload, Molnar's scheduling taxonomy, trade-offs between parallelism, communication, and locality
Sep 15	Geometry Processing and the Scheduling Challenges of Data Amplification Clipping, tessellation, challenges of parallel scheduling)
Sep 17	Visibility: High-Performance Rasterization and Occlusion Visibility algorithms and their fixed-function implementation, occlusion culling, anti-aliasing, frame-buffer compression
Sep 22	Texturing Part I: Basic Algorithms and Cache-Efficient Data Layout Anti-aliasing using the mip-map, hardware texture unit implementation, prefetching and caching policies
Sep 24	Texturing Part II: Texture Compression Hardware-accelerated texture decompression techniques
Sep 29	Shading Language Design Level of abstraction decisions, mapping to GPU processing cores
Oct 1	Compute-Mode GPU Programming Models and Emerging "So-Called-Low-Level" APIs Motivation for alternative abstractions, interoperability issues with graphics pipeline, AMD's Mantle, Apple's Metal
Oct 6	Deferred Shading, and Why it's Now So Popular in Games Motivation for use, impact on global renderer scheduling decisions
Oct 8	The REYES Rendering Architecture The REYES rendering pipeline, and it's differences from OpenGL
Oct 13	High-Performance Ray Tracing and Emerging Ray Tracing Hardware Workload characteristics, coherence optimizations, potential for hardware acceleration
Oct 15	The Light Field and Image-Based Rendering Light field theory, how image-based rendering has found its legs in a world with ubiquitous cameras
Oct 20	The Digital Camera Image Processing Pipeline: Part I Sensor basics, noise sources, early stages of the image pipeline
Oct 22	The Digital Camera Image Processing Pipeline: Part II Gamma, JPG compression, auto-focus, auto-exposure
Oct 27	Scheduling Image Processing Pipelines Work efficiency, parallelism, and locality trade-offs using Halide
Oct 29	Image Processing Algorithm Grab Bag Case studies of workload characteristics and scheduling possibilities
Nov 3	Image Processing Architectures Design characteristics, requirements, and capabilities of emerging systems
Nov 5	H.264 Video Compression
Nov 10	Beyond RGB Pixels I: Light-Field Cameras Unique challenges of light-field photography
Nov 12	Beyond RGB Pixels II: Depth Cameras How the Microsoft Kinect system works, algorithms for computing depth and pose
Nov 17	A Systems View of Large-Scale 3D Reconstruction Modern work on 3D reconstruction from large image collections
Nov 19	TBD - Kayvon out of town
Nov 24	Challenges of processing billions of images
Nov 26	**No Class - Thanksgiving Holiday**
Dec 1	Active Research Topic: Adaptive-Rate and Low-Rate Shading
Dec 3	Active Research Topic: Rendering Challenges of VR

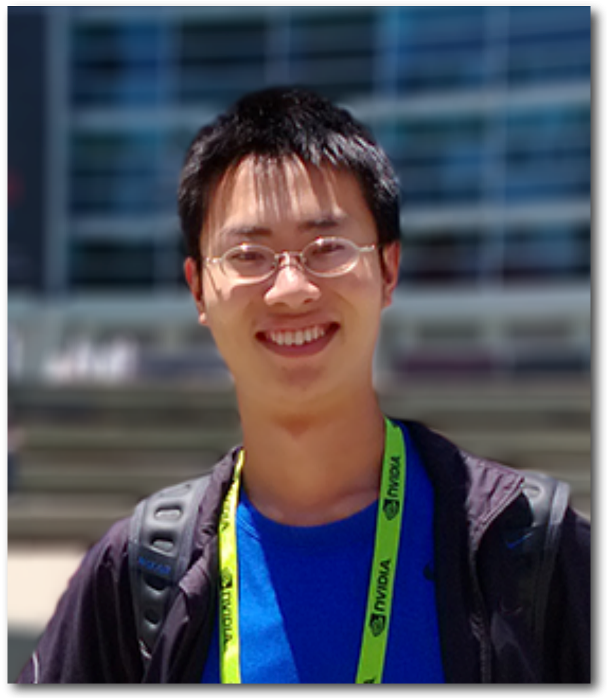
Rendering systems:
Primarily real-time 3D graphics as implemented by modern games

High-performance image processing
Camera image pipeline (for photography)
Image processing for computer vision at scale.

Miscellaneous topics (may change)

Logistics

- **Course web site:**
  - http://15869.courses.cs.cmu.edu
- **All announcements will go out via Piazza**
  - https://piazza.com/cmu/fall2014/15869/home
- **Kayvon's office hours: drop in or by appointment (EDSH 225)**
- **Your knowledgable TA: Yong He (GHC 7117)**



Expectations of you

- **30% participation**
  - There will be ~1-2 assigned paper readings per class
  - Everyone is expected to come to class and participate in discussions based on readings
  - You are encouraged discuss papers and or my lectures on the course discussion board.
  - If you form a weekly course reading/study group, I will buy Pizza.

- **25% mini-assignments (2-3 short programming assignments)**
  - Implement a basic parallel triangle renderer
  - Implement a RAW image processing pipeline

- **45% self-selected final project**
  - I suggest you start talking to me now (can be teams of up to two)

- **We have toys to play around with throughout the semester:**
  - You are encouraged to experiment with them and report what you learn back to the class
  - Two Oculus DK2s
  - Two NVIDIA Shields, one Jetson K1

Somewhat philosophical question:
What is an “architecture”?

An architecture is an abstraction

It defines:

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

Choice impacts system's expressiveness/scope and its suitability for efficient implementation.

The 3D rendering problem

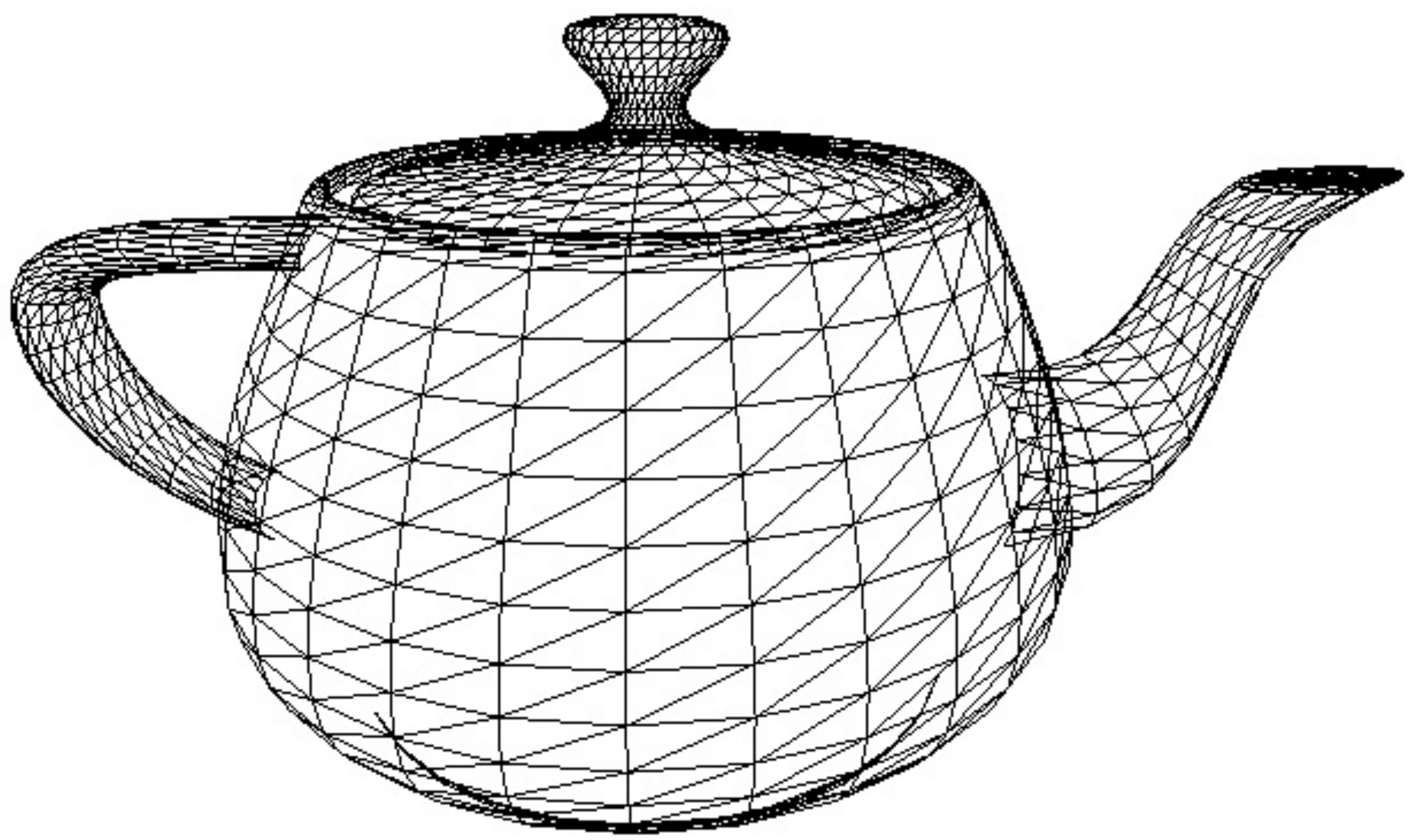


Image credit: Henrik Wann Jensen

Input: description of a scene

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

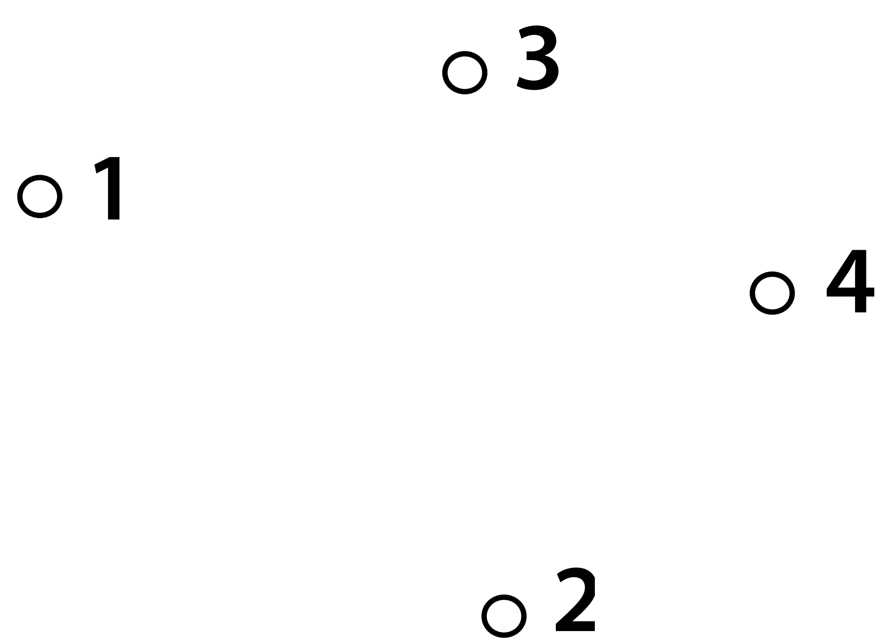
Output: image

Main problem statement: How does each geometric element contribute to the appearance of each output pixel in the image, given a description of surface properties and lighting conditions.

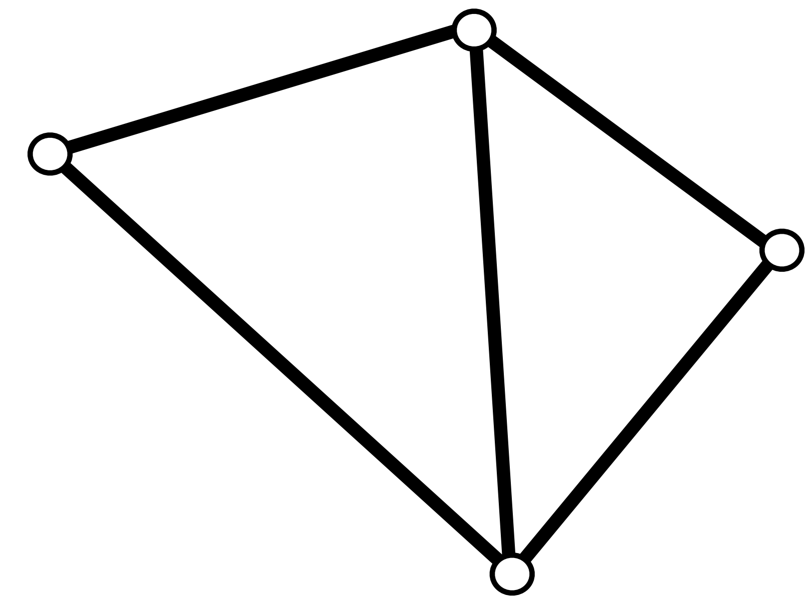
The real-time graphics pipeline architecture

(A review of the GPU-accelerated OpenGL/D3D graphics pipeline, from a systems perspective)

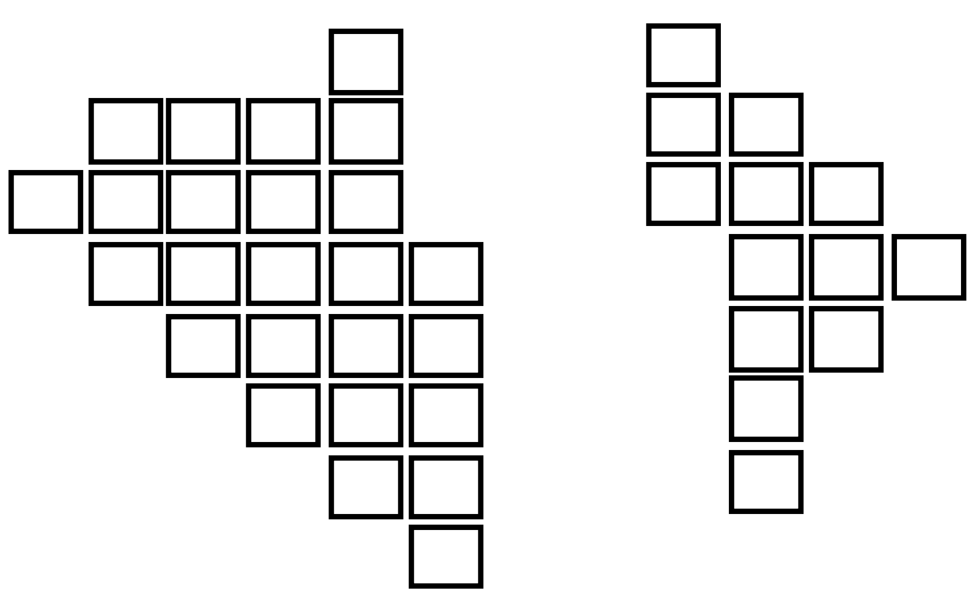
Real-time graphics pipeline entities



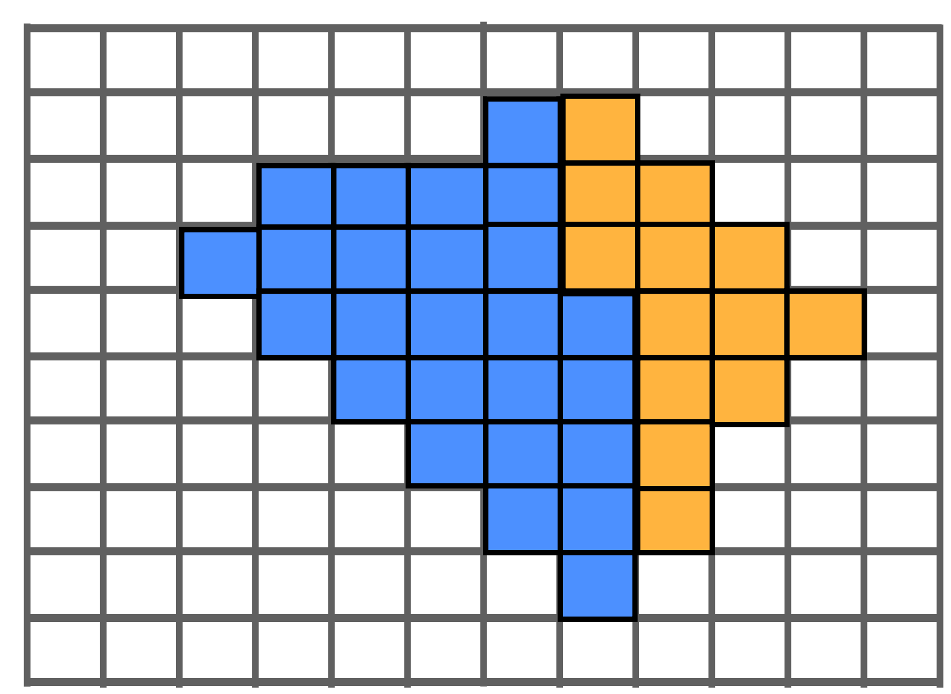
Vertices



Primitives (triangles, points, lines)

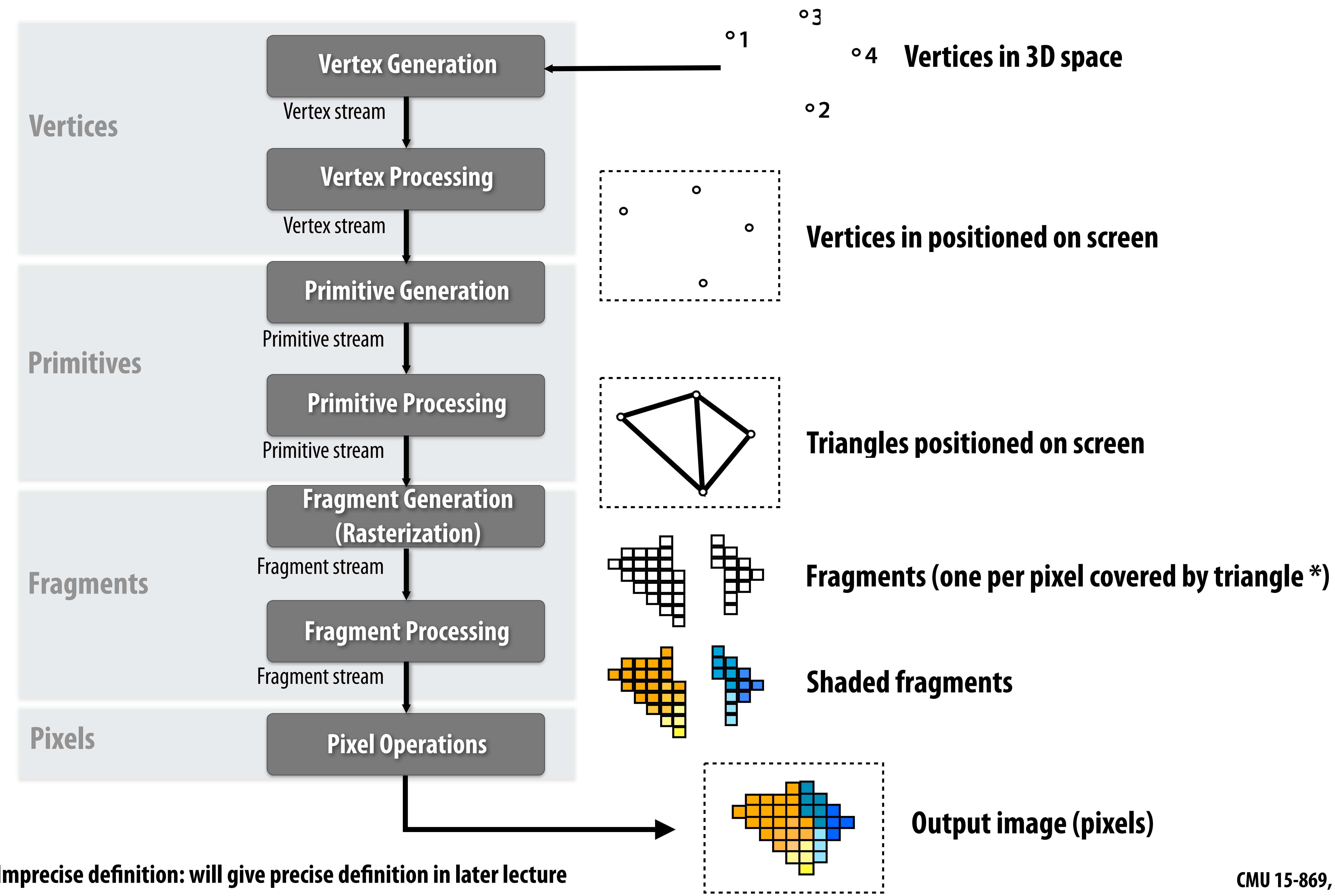


Fragments



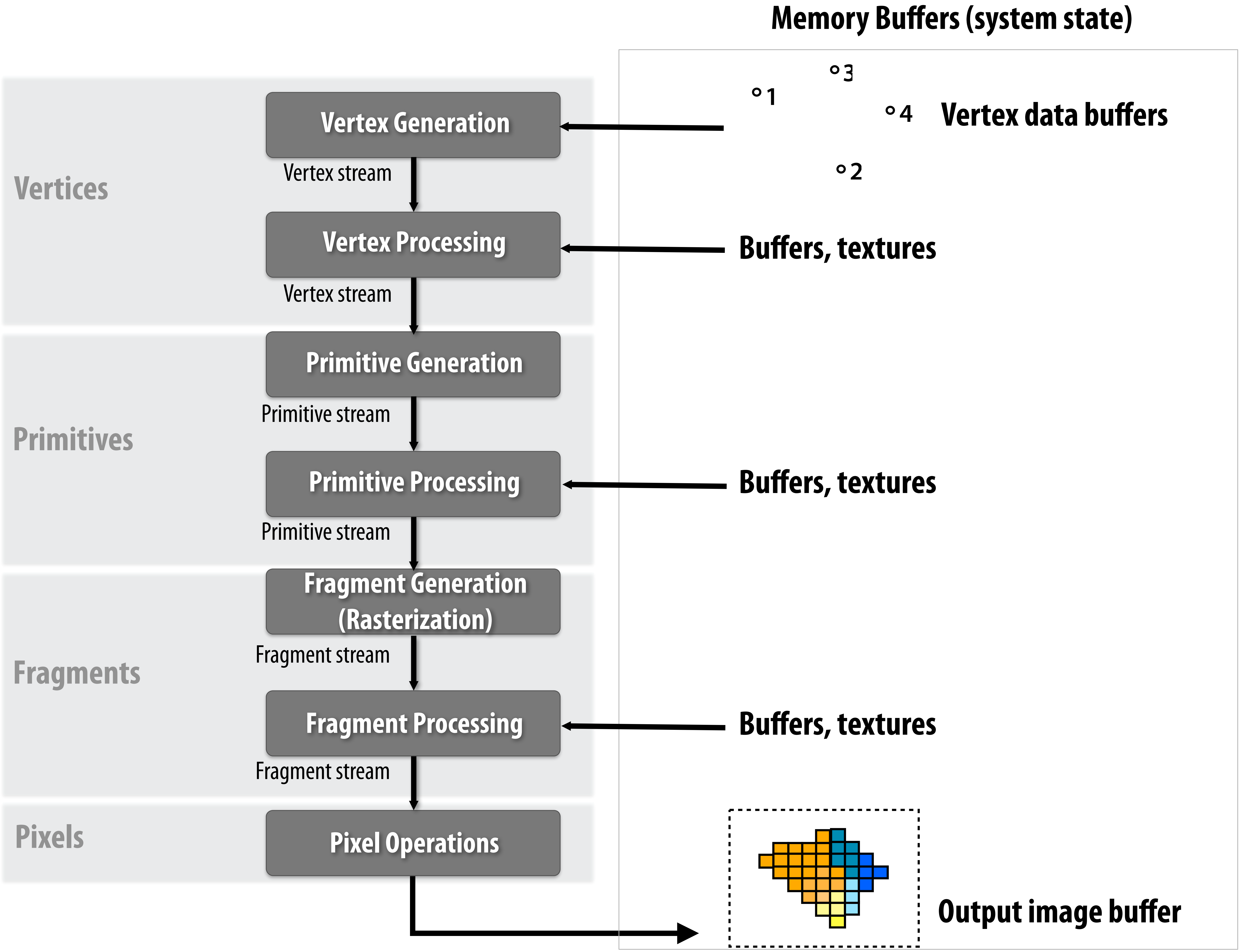
Pixels

Real-time graphics pipeline operations



* Imprecise definition: will give precise definition in later lecture

Real-time graphics pipeline state



3D graphics system stack

Application

(e.g, a computer game)

Scene graph

(application's database representing the scene: geometry, materials, lights, etc.)

Graphics pipeline

(OpenGL/Direct3D)

Graphics pipeline implementation

(software driver + GPU)

clients to the system (use the abstraction)

the abstraction we are discussing now

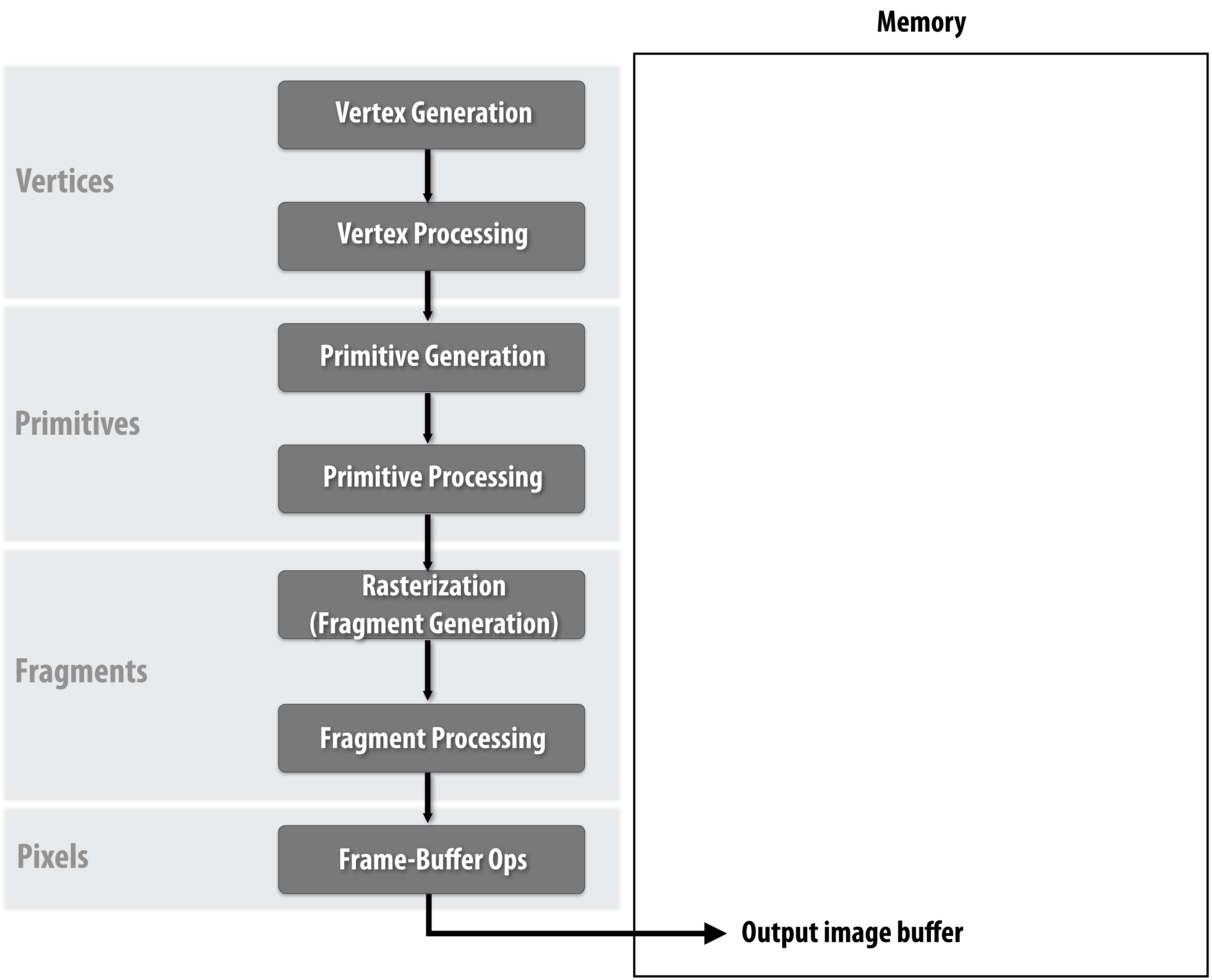
implements the abstraction

Issues to keep in mind during this review *

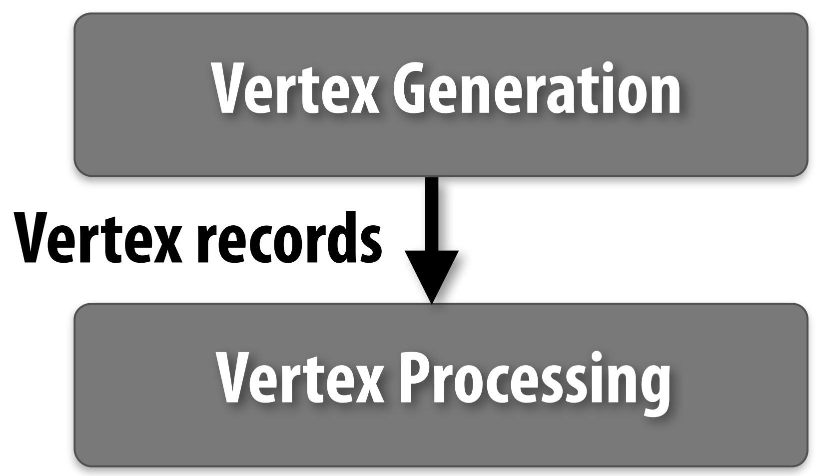
- Level of abstraction
- Orthogonality of abstractions
- How is pipeline designed for performance/scalability?
- What the pipeline does and <u>DOES NOT</u> do

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

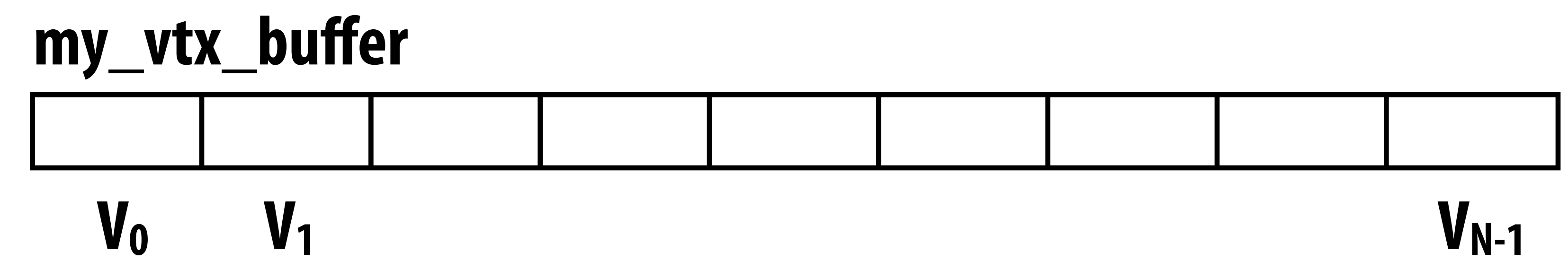
The graphics pipeline



“Assembling” vertices

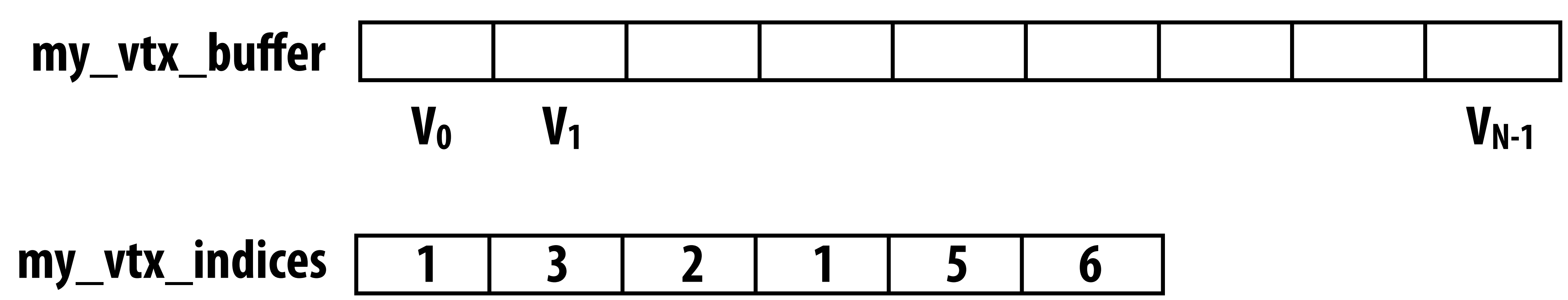


Contiguous version data version



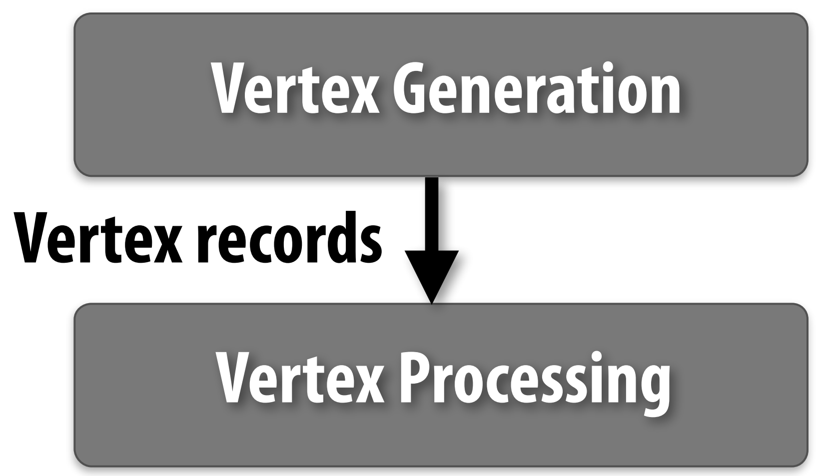

```
glBindBuffer(GL_ARRAY_BUFFER, my_vtx_buffer);
glDrawArrays(GL_TRIANGLES, 0, N);
```

Indexed access version (“gather”)

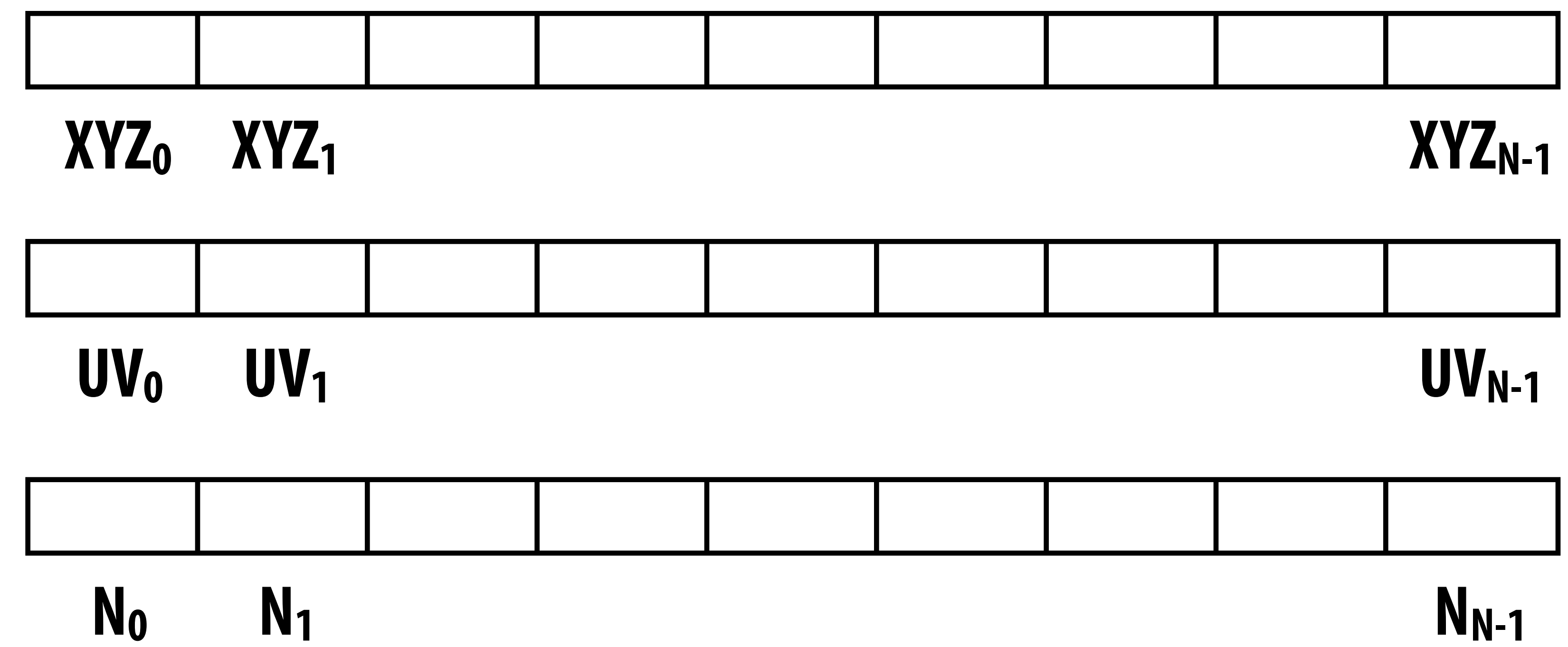



```
glBindBuffer(GL_ARRAY_BUFFER, my_vtx_buffer);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT,
               my_vtx_indices);
```

“Assembling” vertices



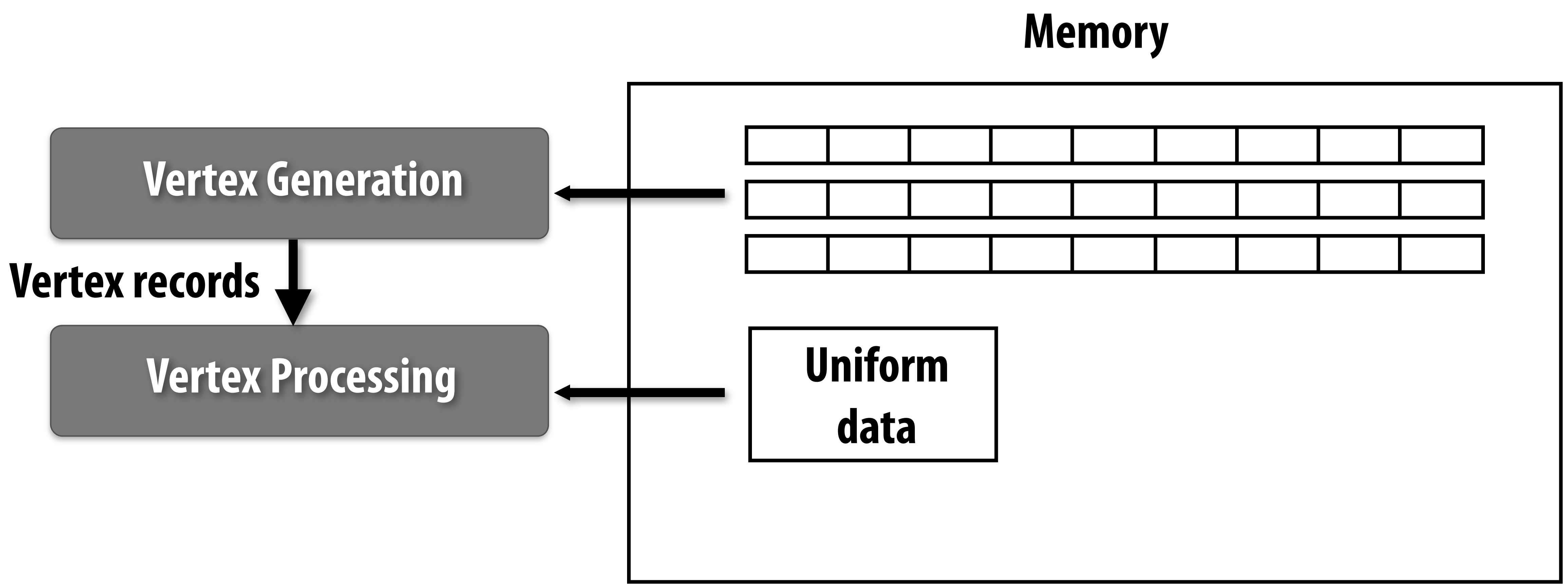
Contiguous vertex buffer



Output of vertex generation is a collection of vertex records.

Current pipelines set a limit of 32 float4 attributes per vertex. (512 bytes)
Why? (to be answered in a later lecture)

Vertex processing inputs

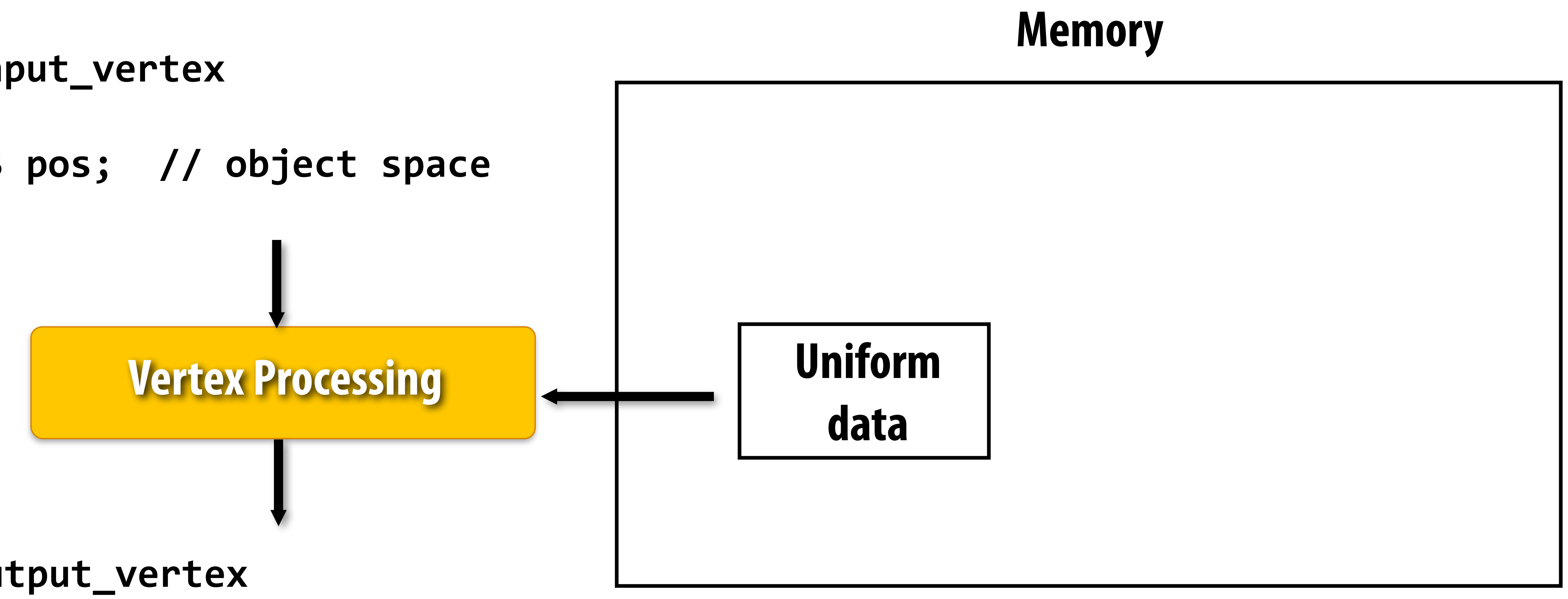


Uniform data: constant read-only data provided as input to every instance of the vertex shader
e.g., object-to-clip-space vertex transform matrix

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

Vertex processing inputs and outputs

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




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

1 input vertex → 1 output vertex

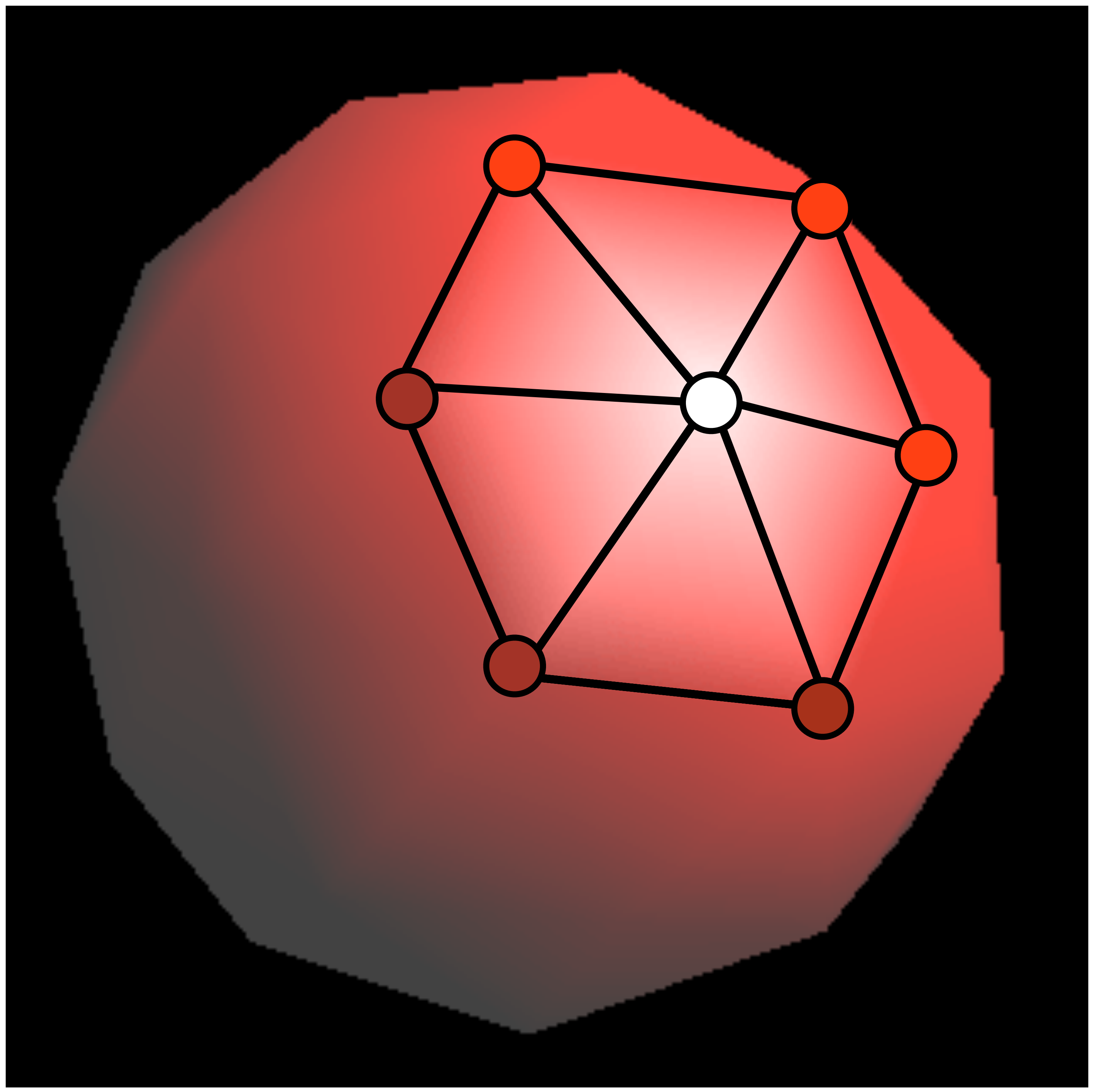
independent processing of each vertex

Vertex Shader Program *

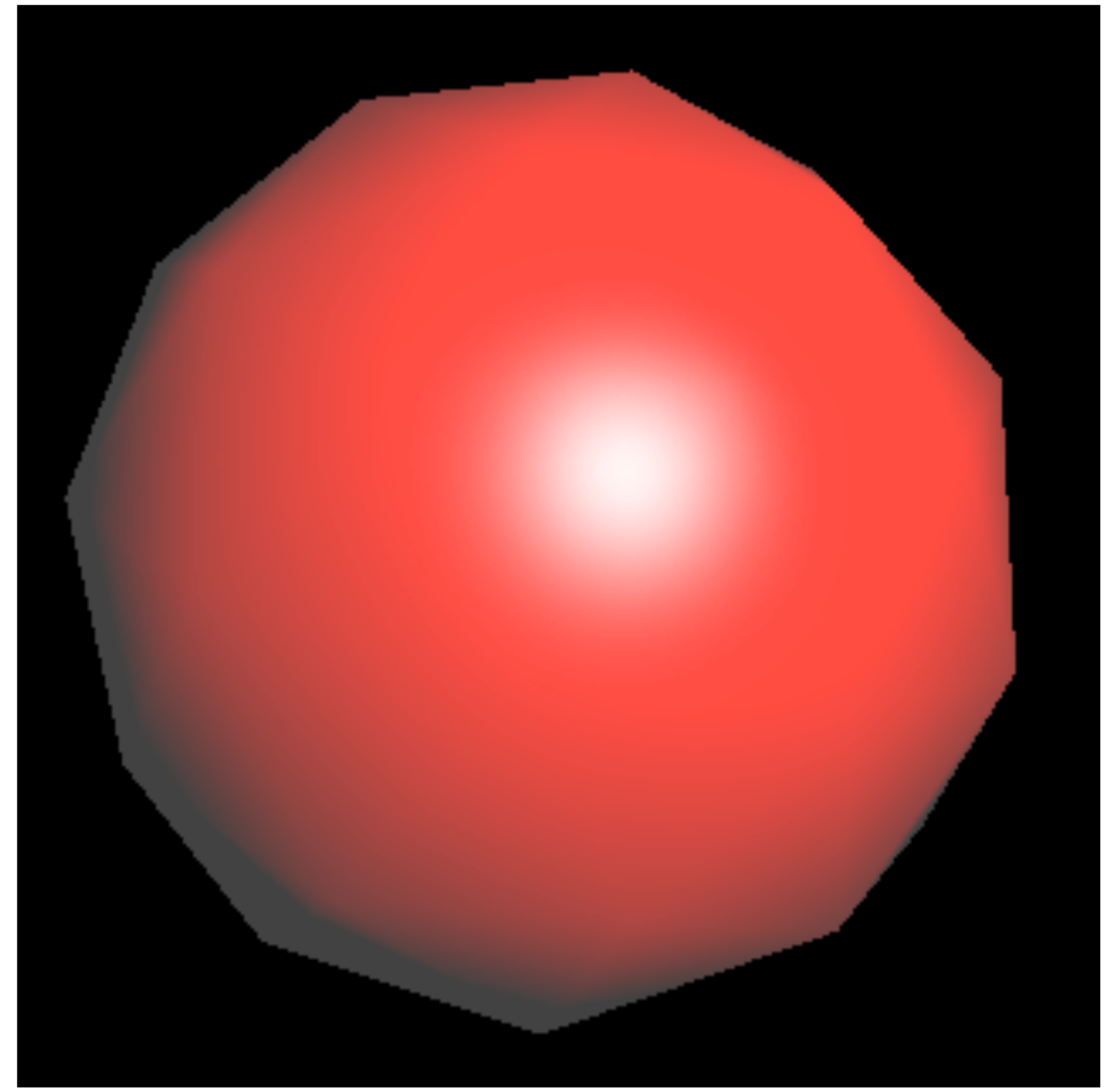
```
uniform mat4 my_transform;

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)

Example: per-vertex lighting



Per-vertex lighting computation

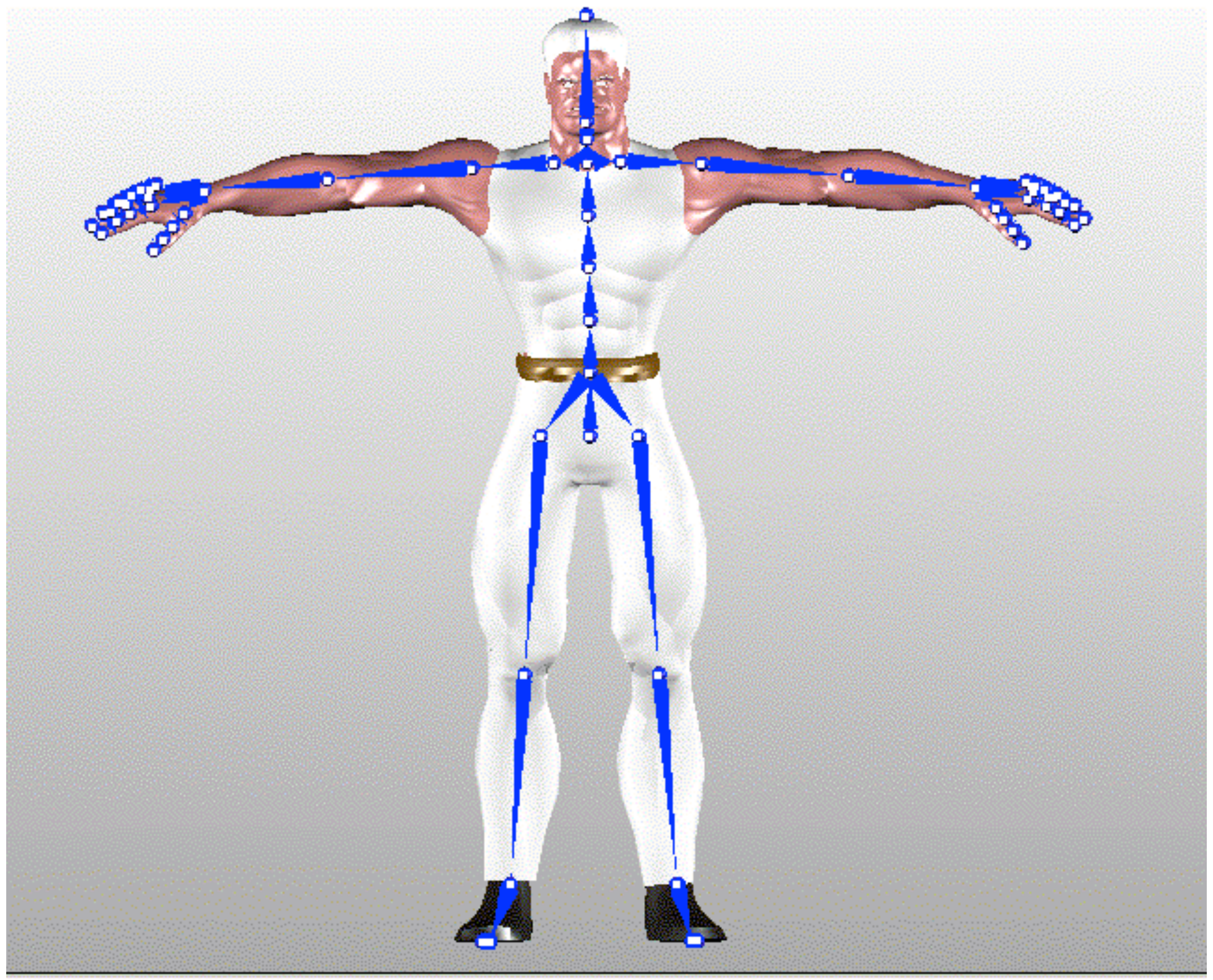


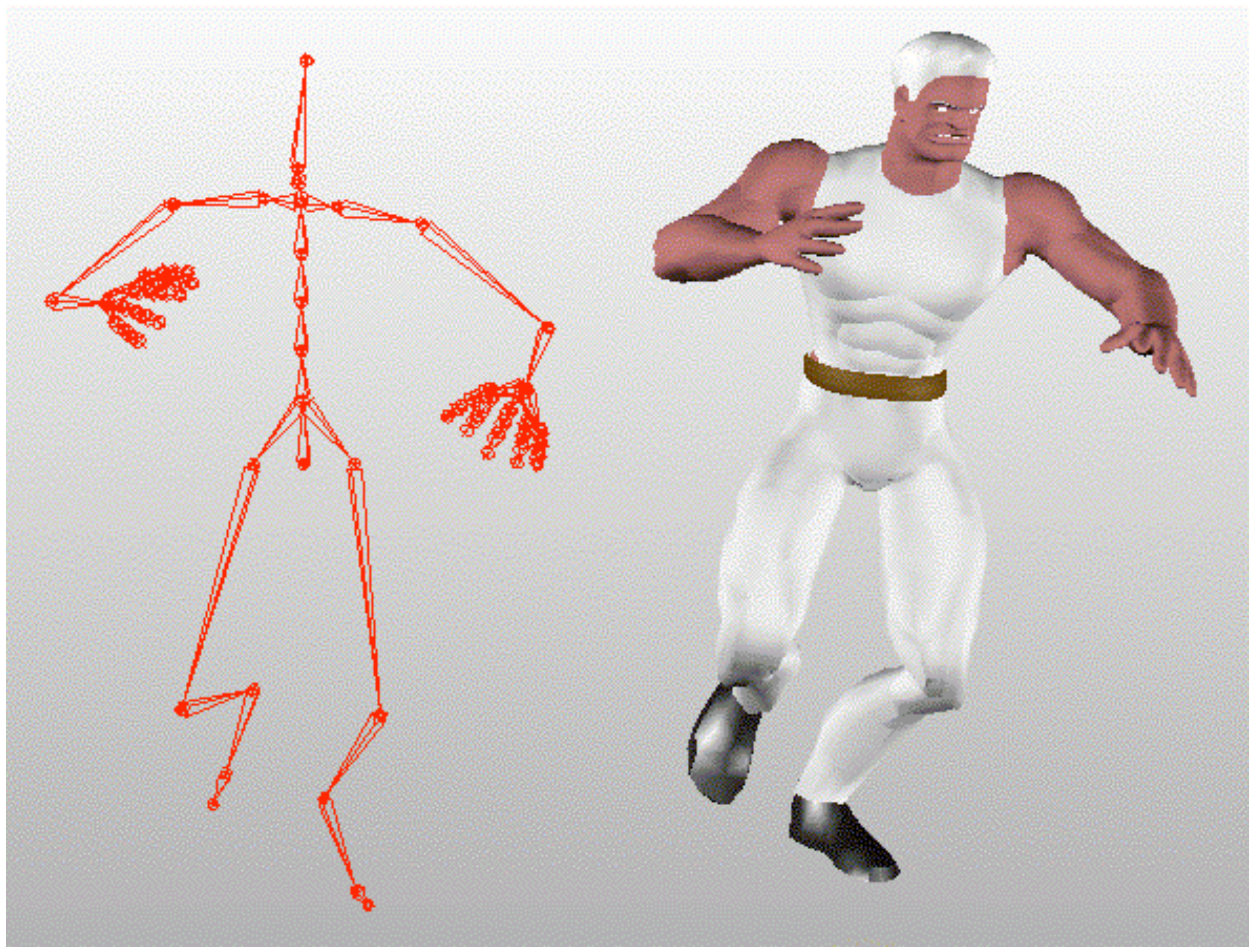
Per-vertex normal computation, per pixel lighting

Per-vertex data: surface normal, surface color

Uniform data: light direction, light color

Example: vertex skinning





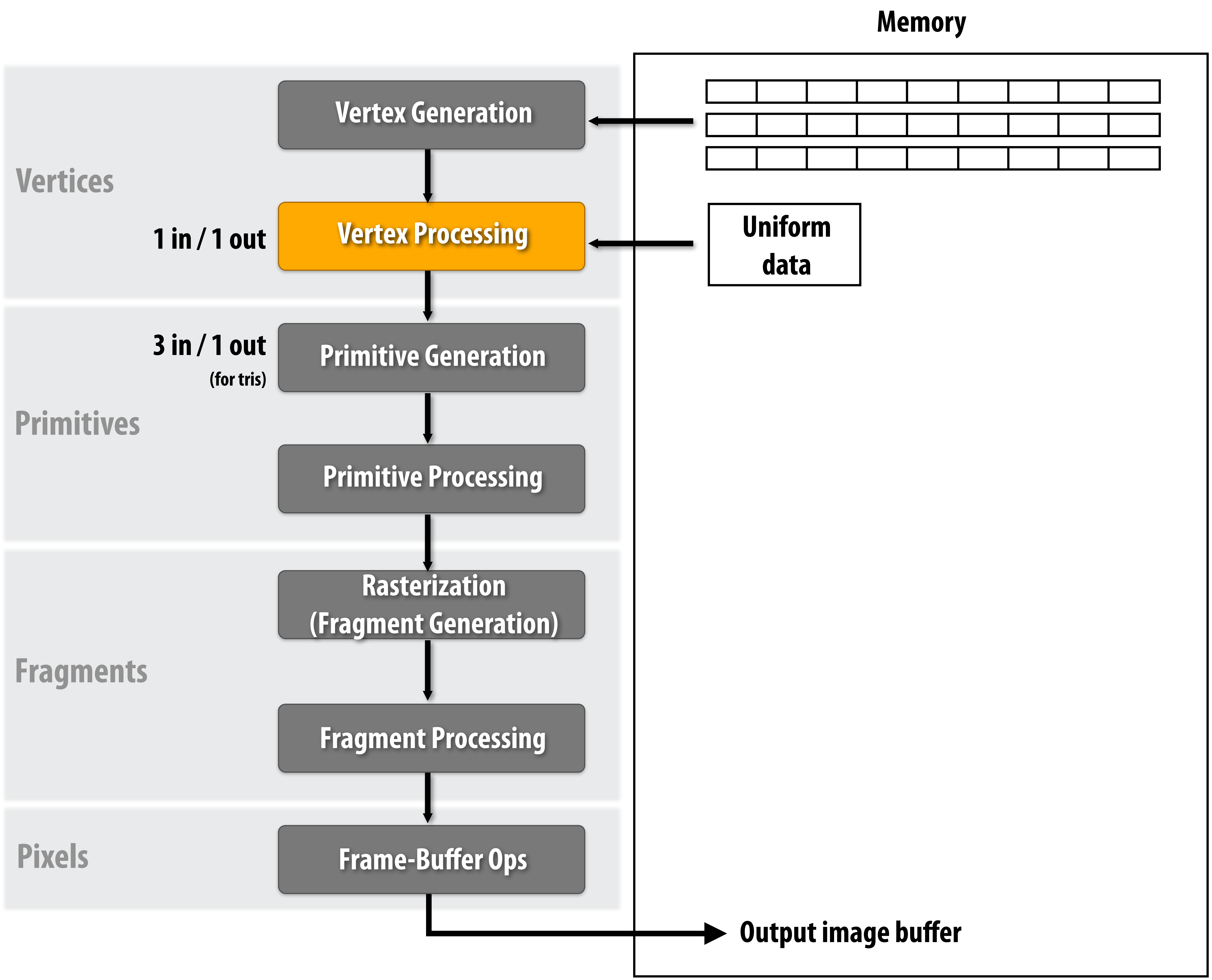
$$V_{skinned} = \sum_{b \in bones} w_b M_b V_{base}$$

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

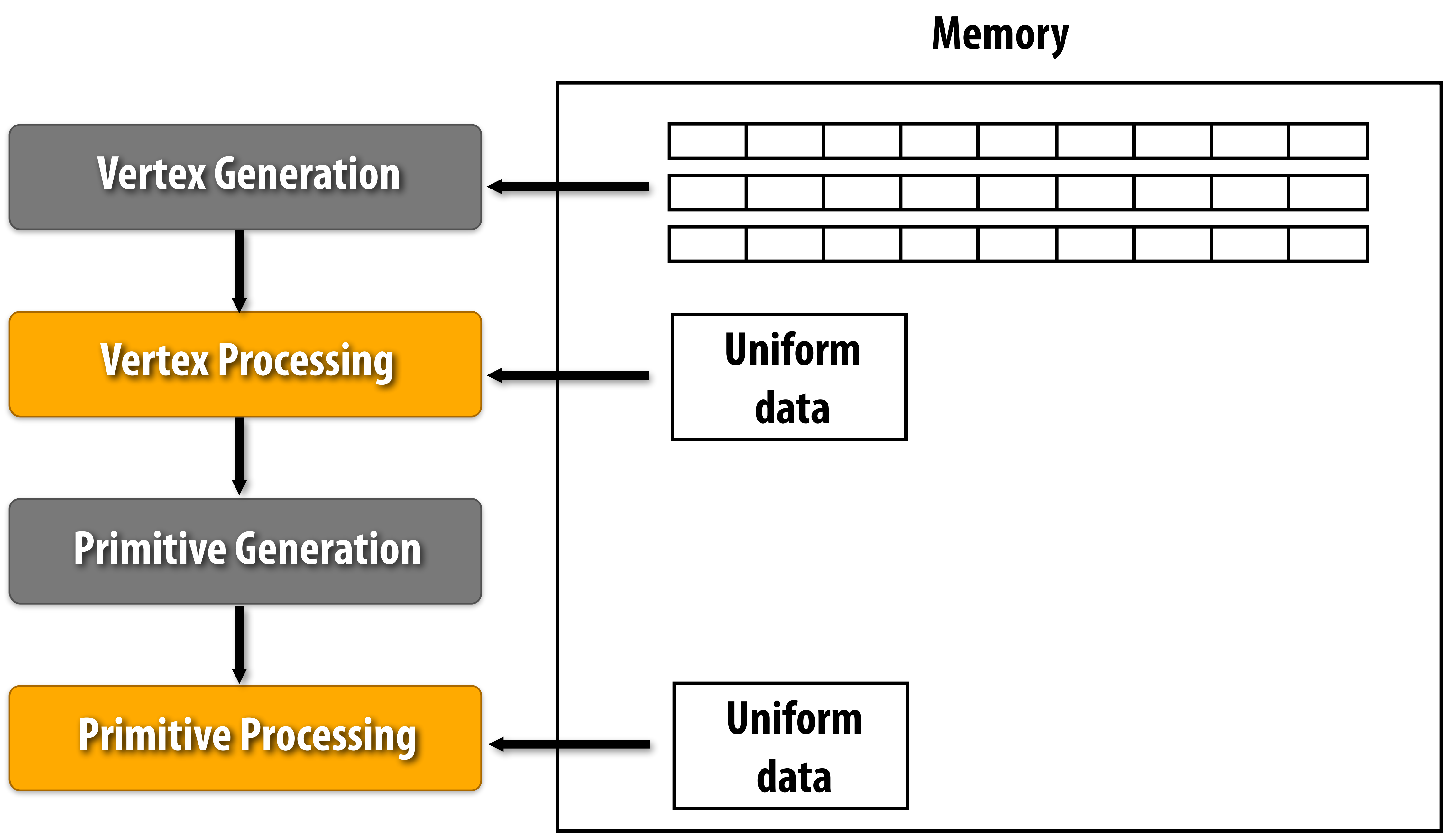
Uniform data: "bone" matrices (M_b) for current animation frame

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

The graphics pipeline



Primitive processing *



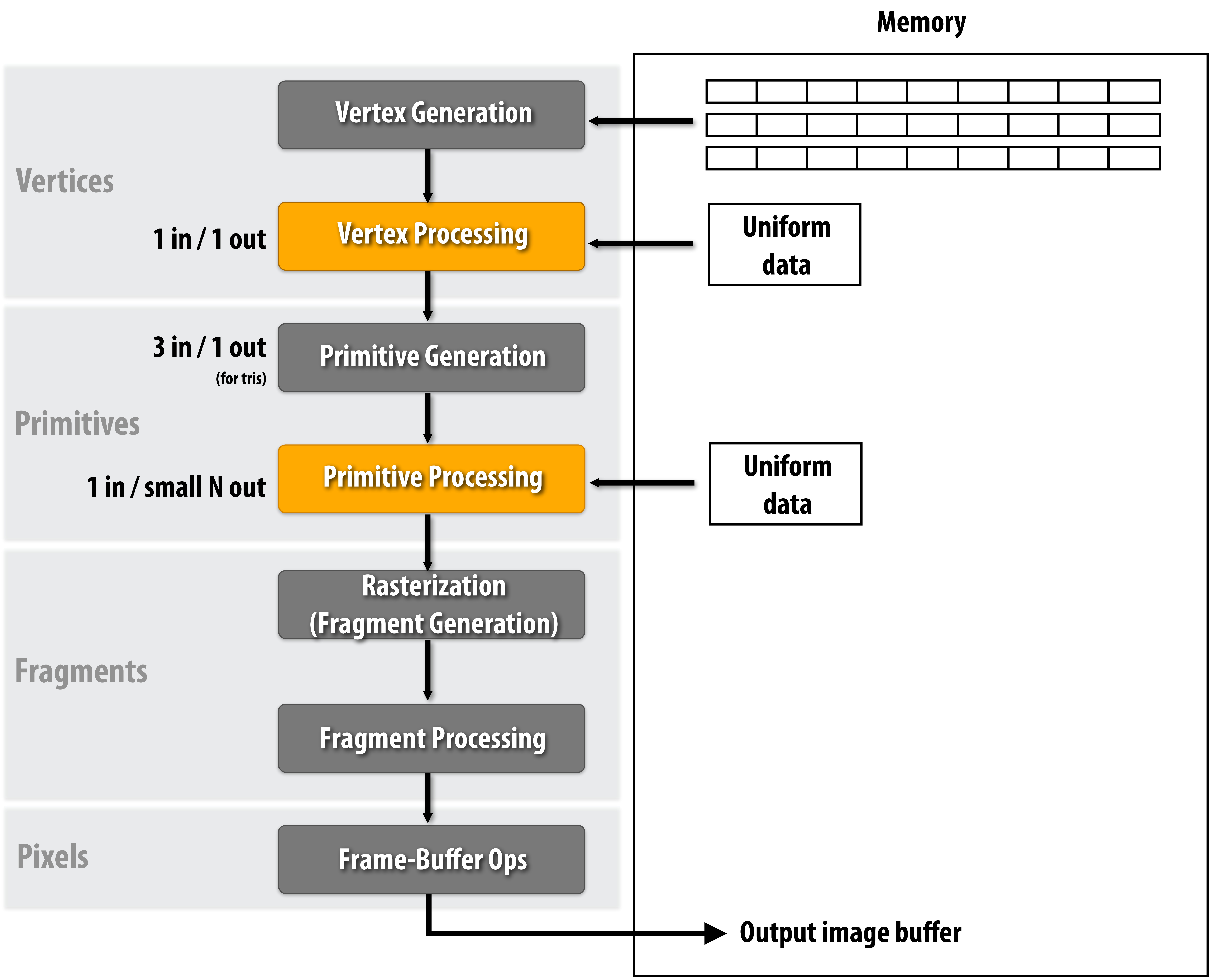
input vertices for 1 prim ⟶ output vertices for N prims **

independent processing of each INPUT primitive

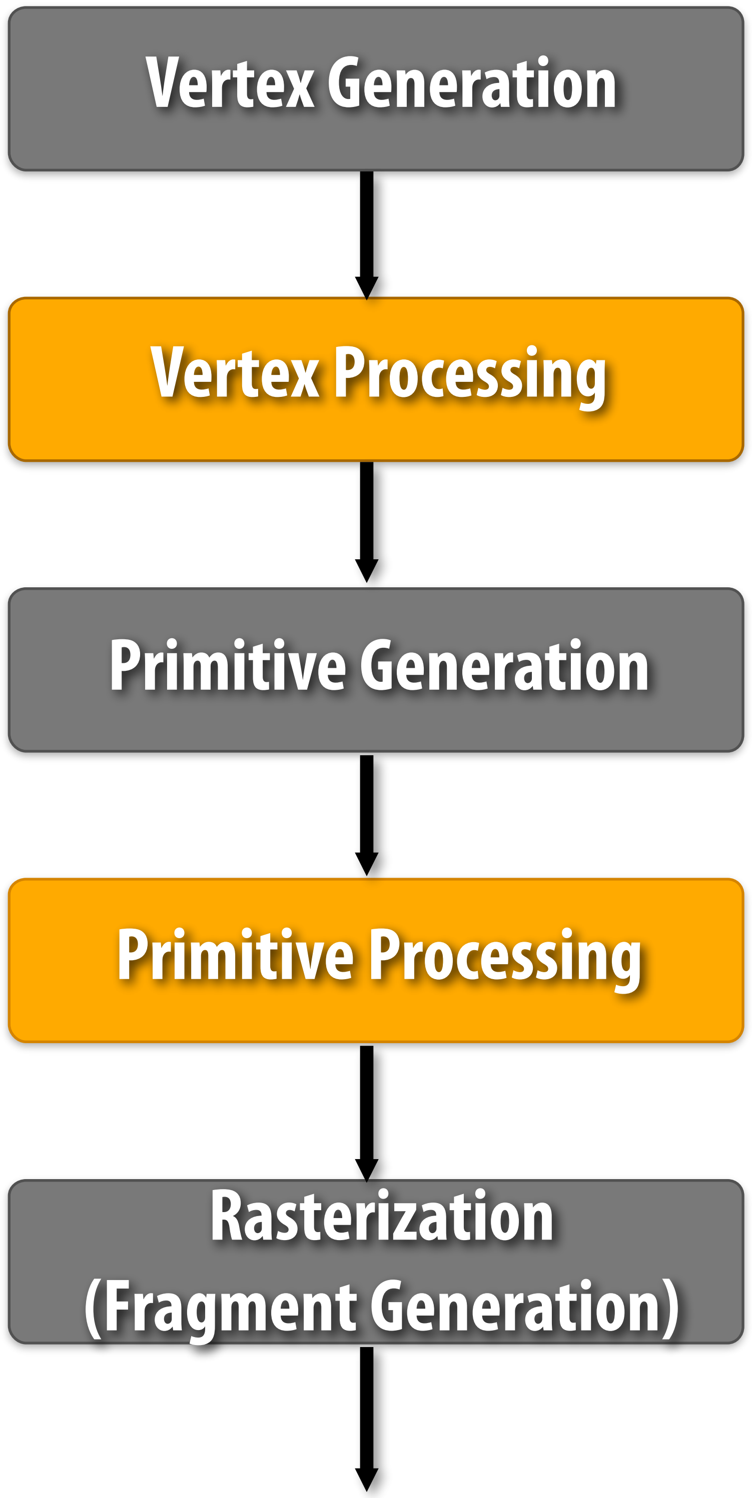
* "Geometry shader" in OpenGL/Direct3D terminology

** Pipeline caps output at 1024 floats of output

The graphics pipeline

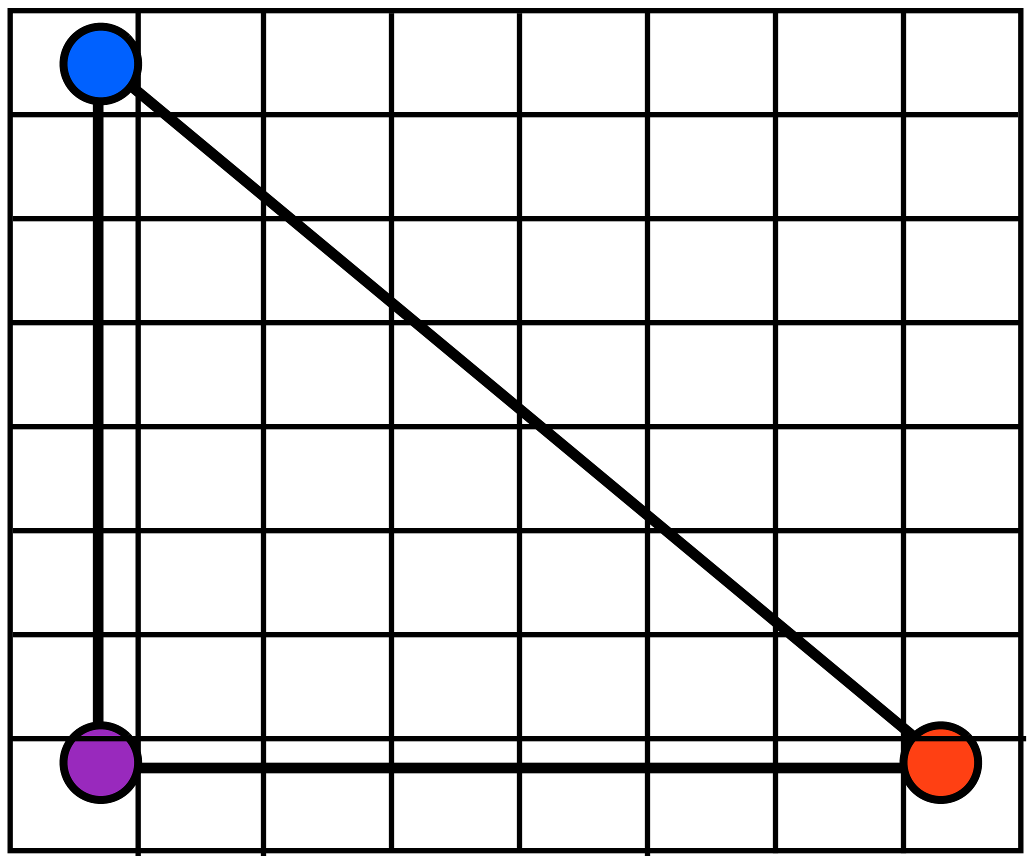


Rasterization



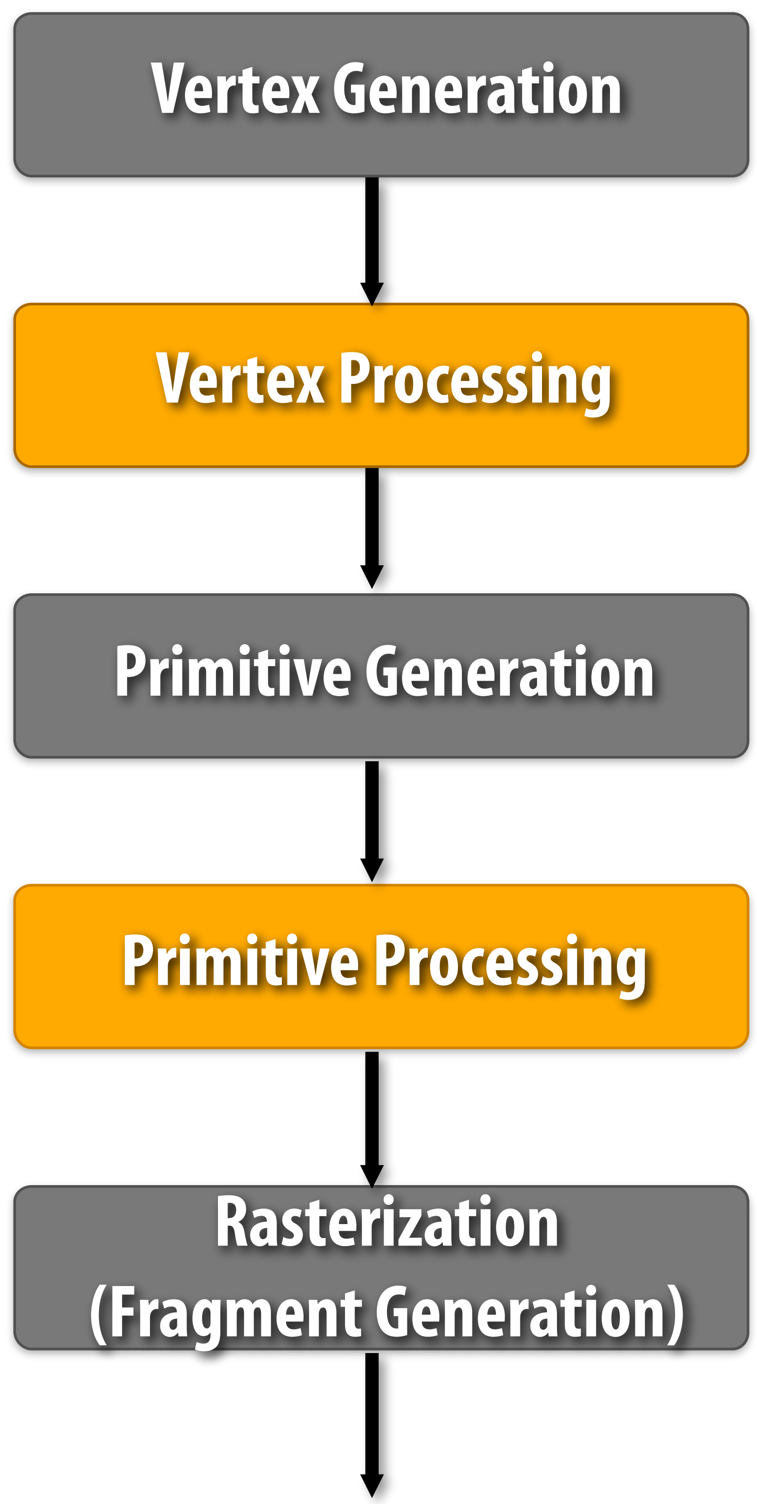
1 input prim ⟶ N output fragments

N is unbounded
(size of triangles varies greatly)



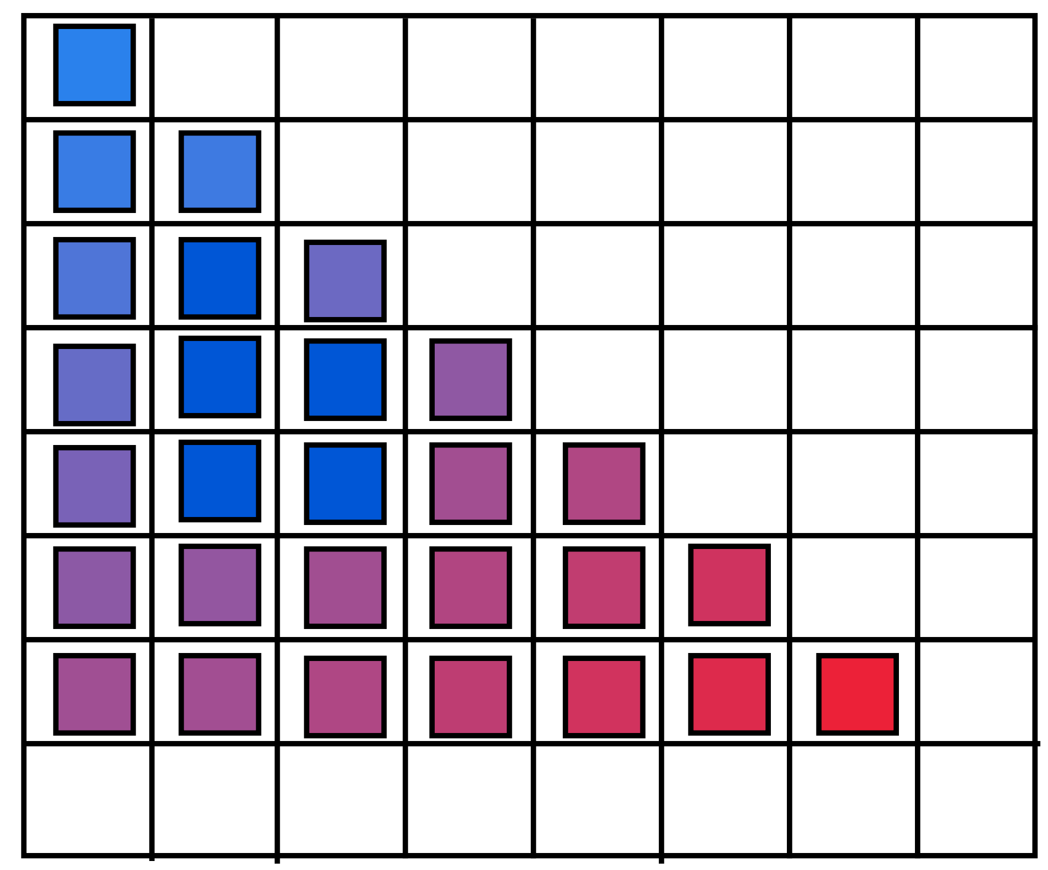
```
struct fragment // note similarity to output_vertex from before
{
   float  x,y;  // screen pixel coordinates (sample point location)
   float  z;    // depth of triangle at sample point

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

Rasterization



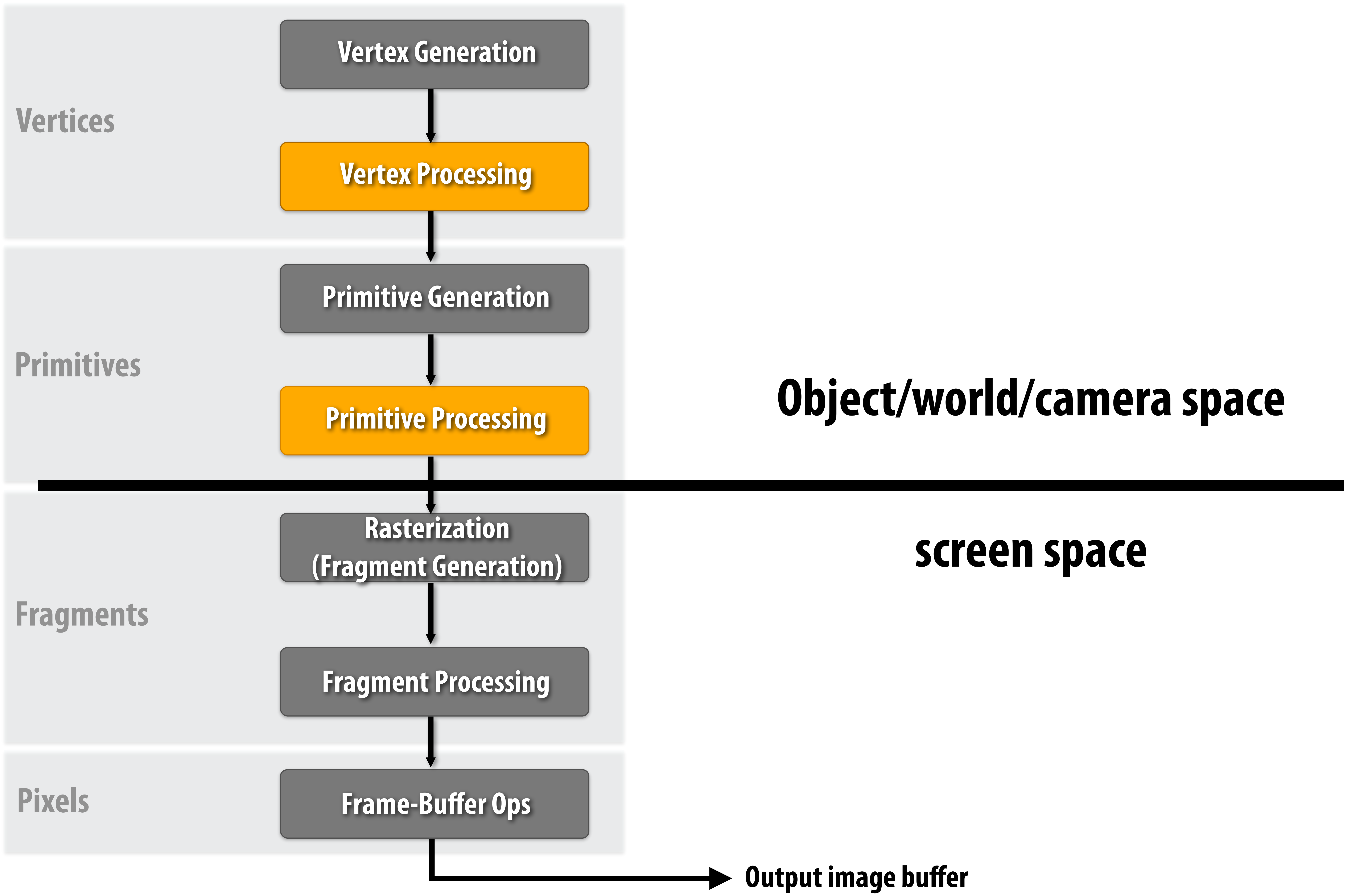
Compute covered pixels

Sample vertex attributes once per covered pixel

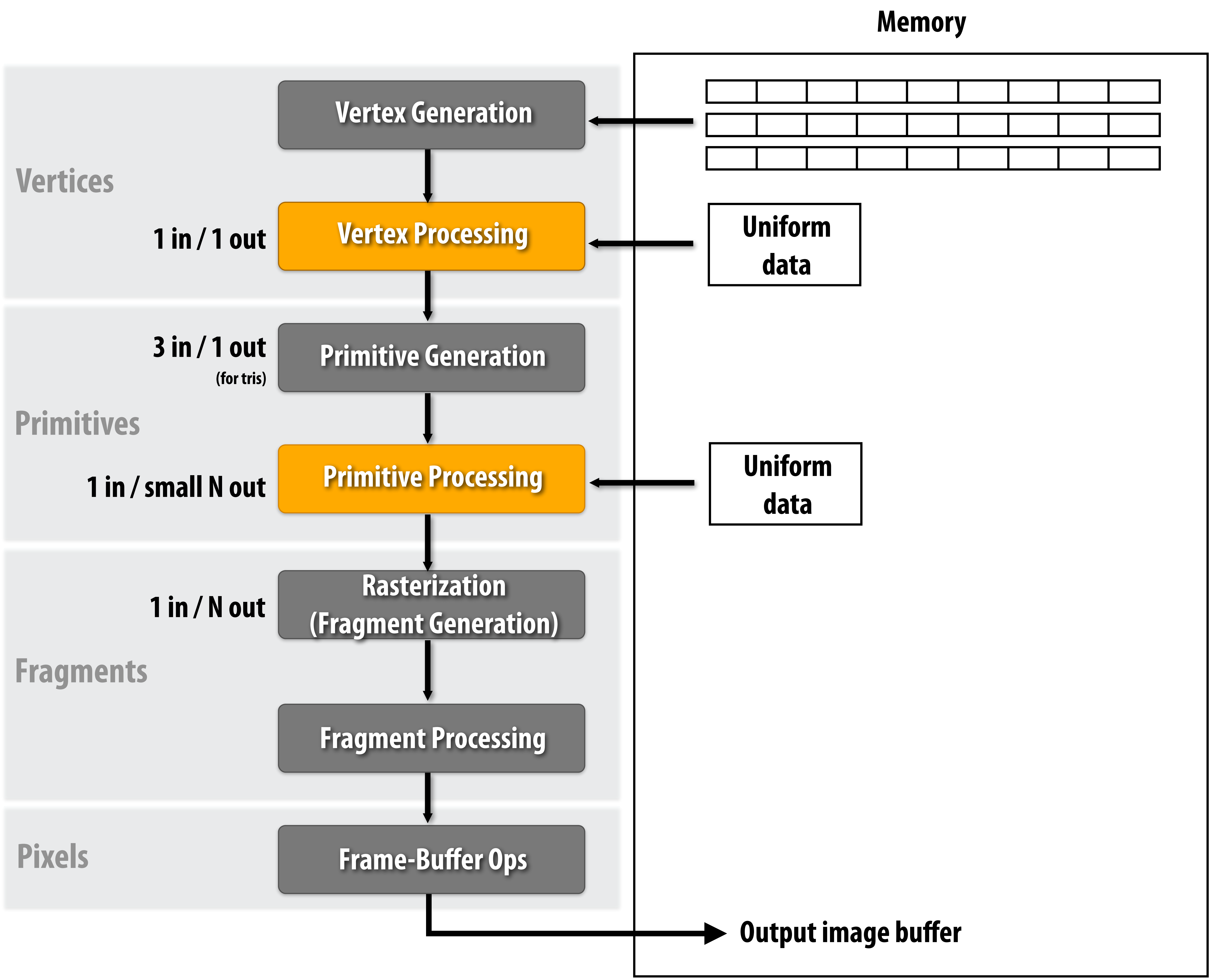


```
struct fragment // note similarity to output_vertex from before
{
   float  x,y;  // screen pixel coordinates (sample point location)
   float  z;    // depth of triangle at sample point

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

}
```

The graphics pipeline

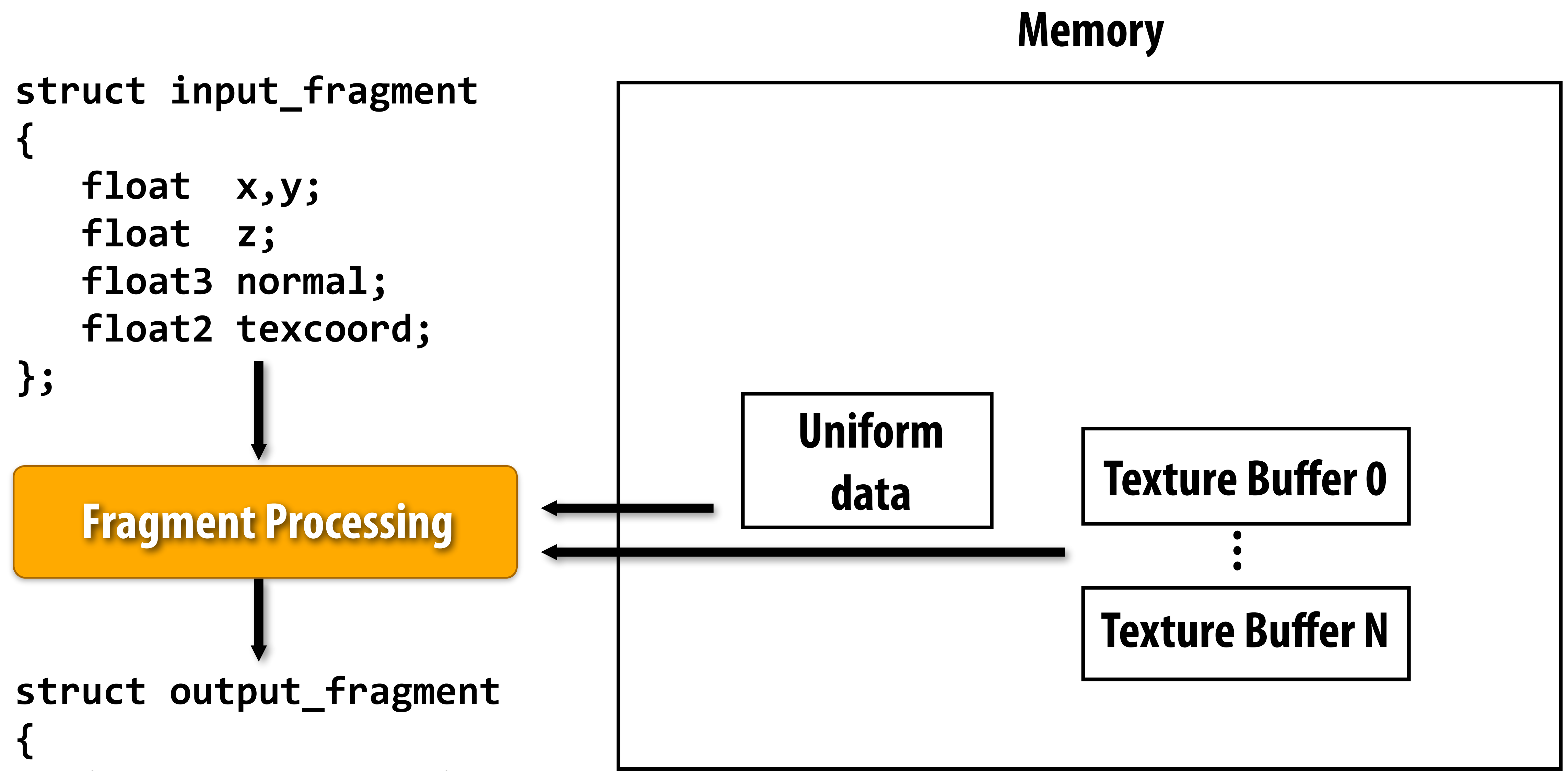


The graphics pipeline



Fragment processing

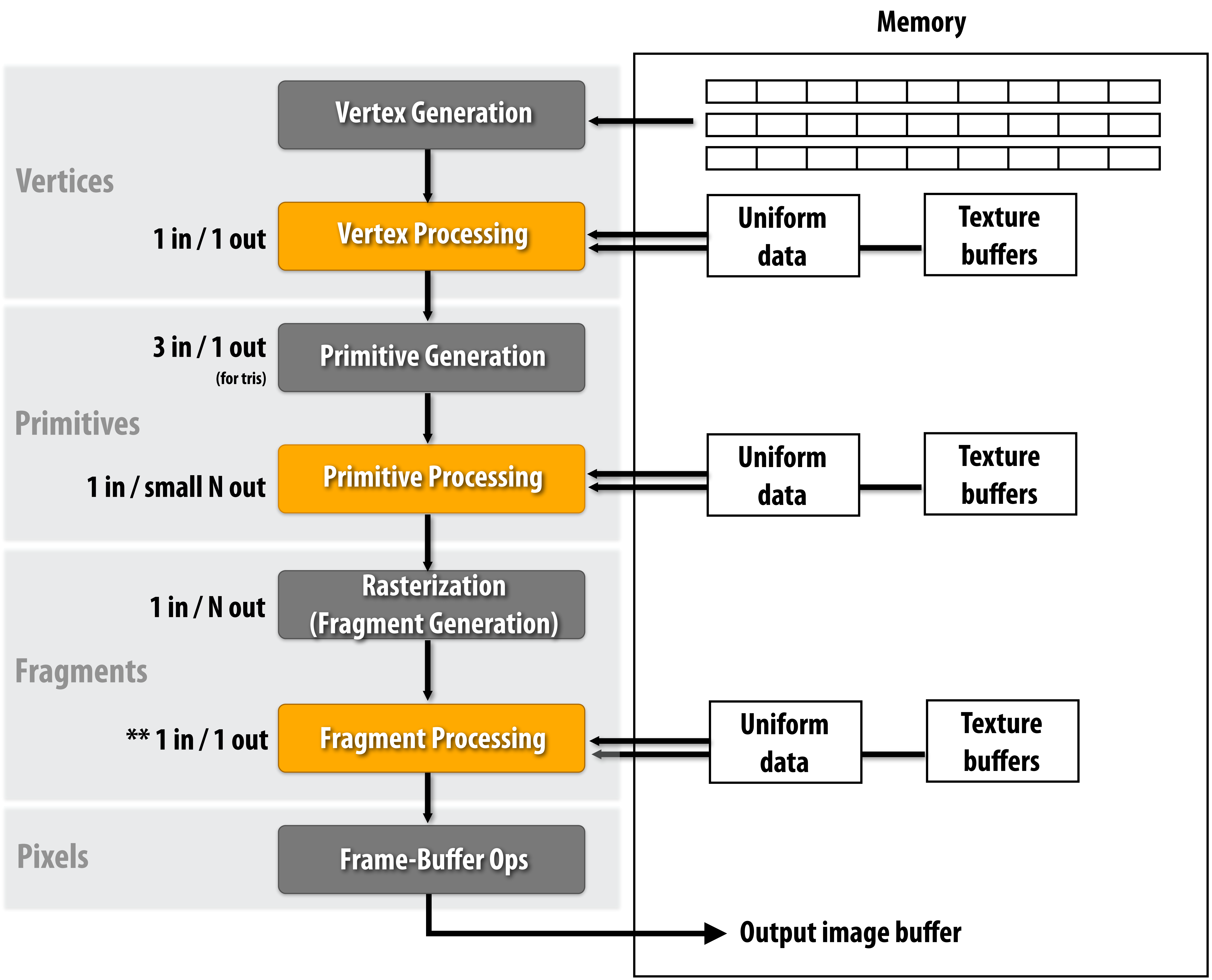
```
struct input_fragment
{
   float  x,y;
   float  z;
   float3 normal;
   float2 texcoord;
};
```




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

```
texture my_texture;

output_vertex my_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;
}
```

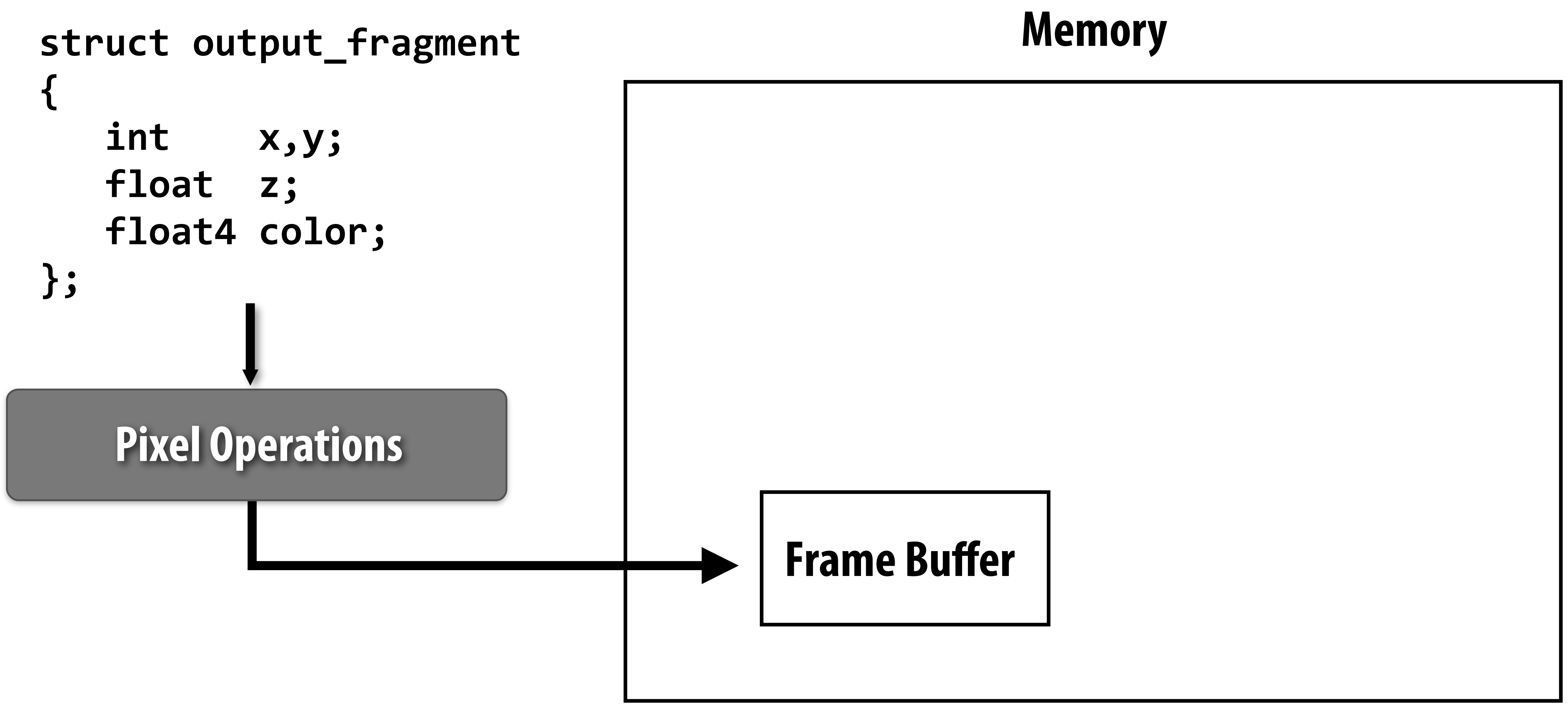
The graphics pipeline



** can be 0 out

Frame-buffer operations

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



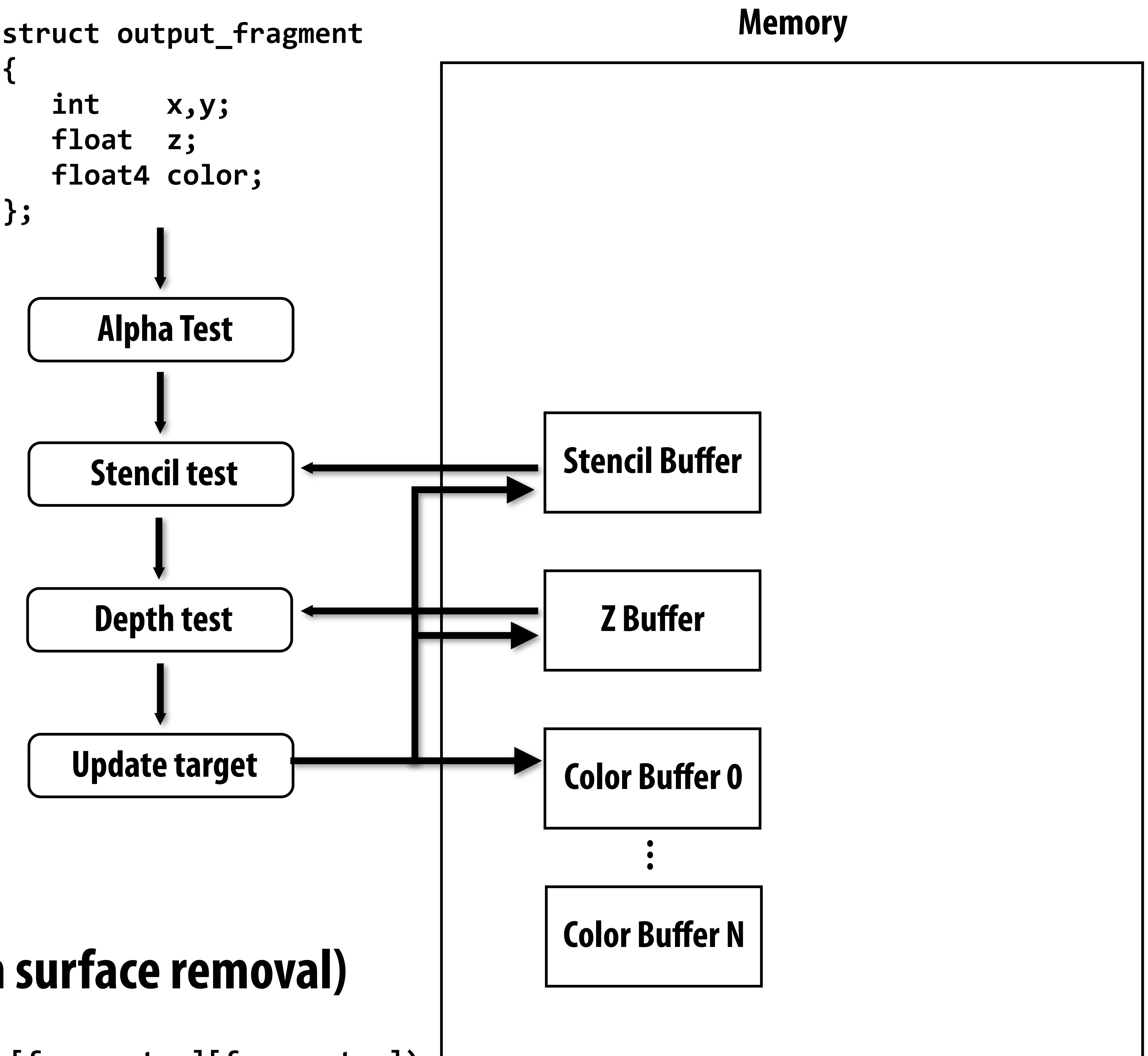
Frame-buffer operations

Depth test (hidden surface removal)

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

Frame-buffer operations (full view)

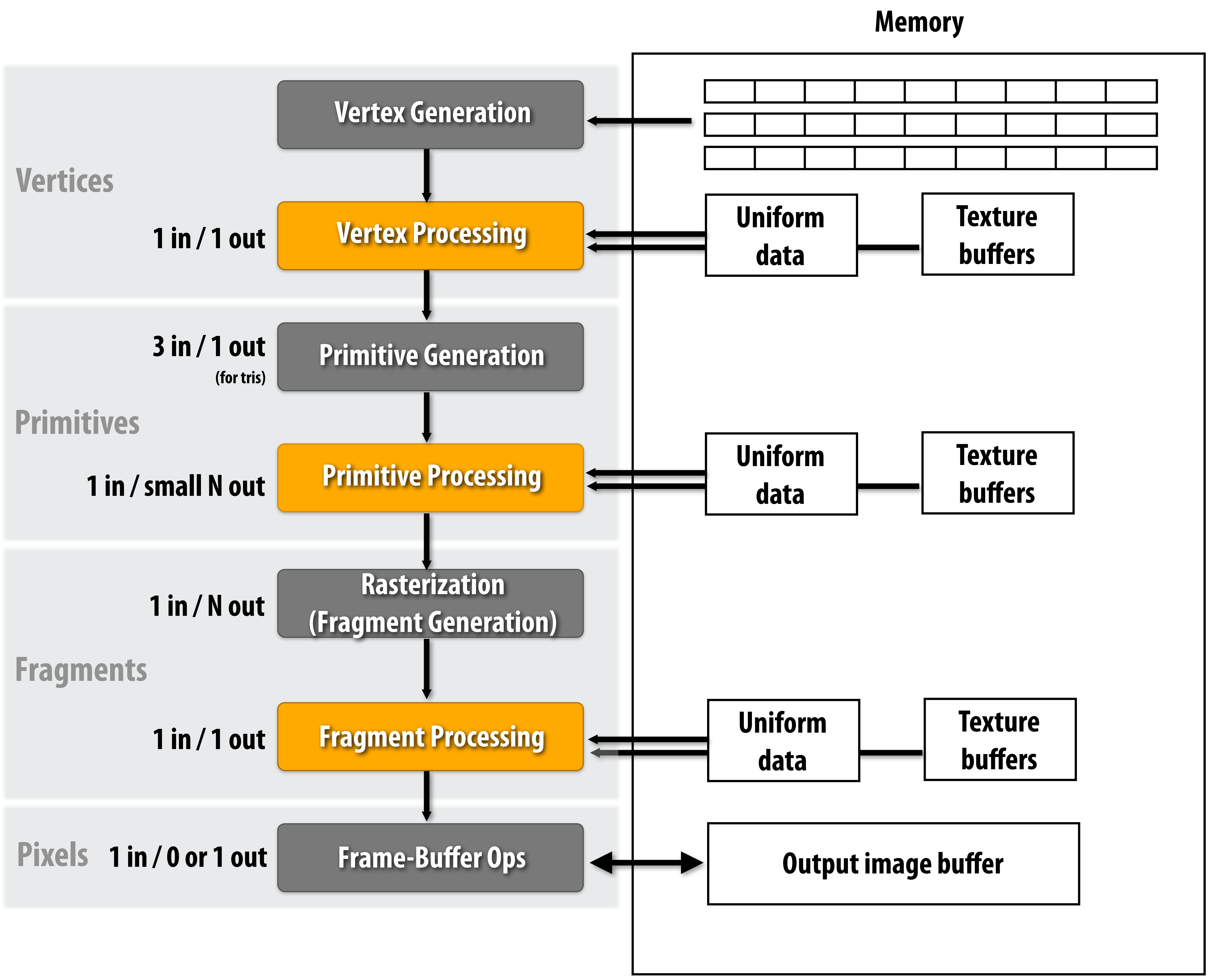
```
struct output_fragment
{
   int    x,y;
   float  z;
   float4 color;
};
```



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

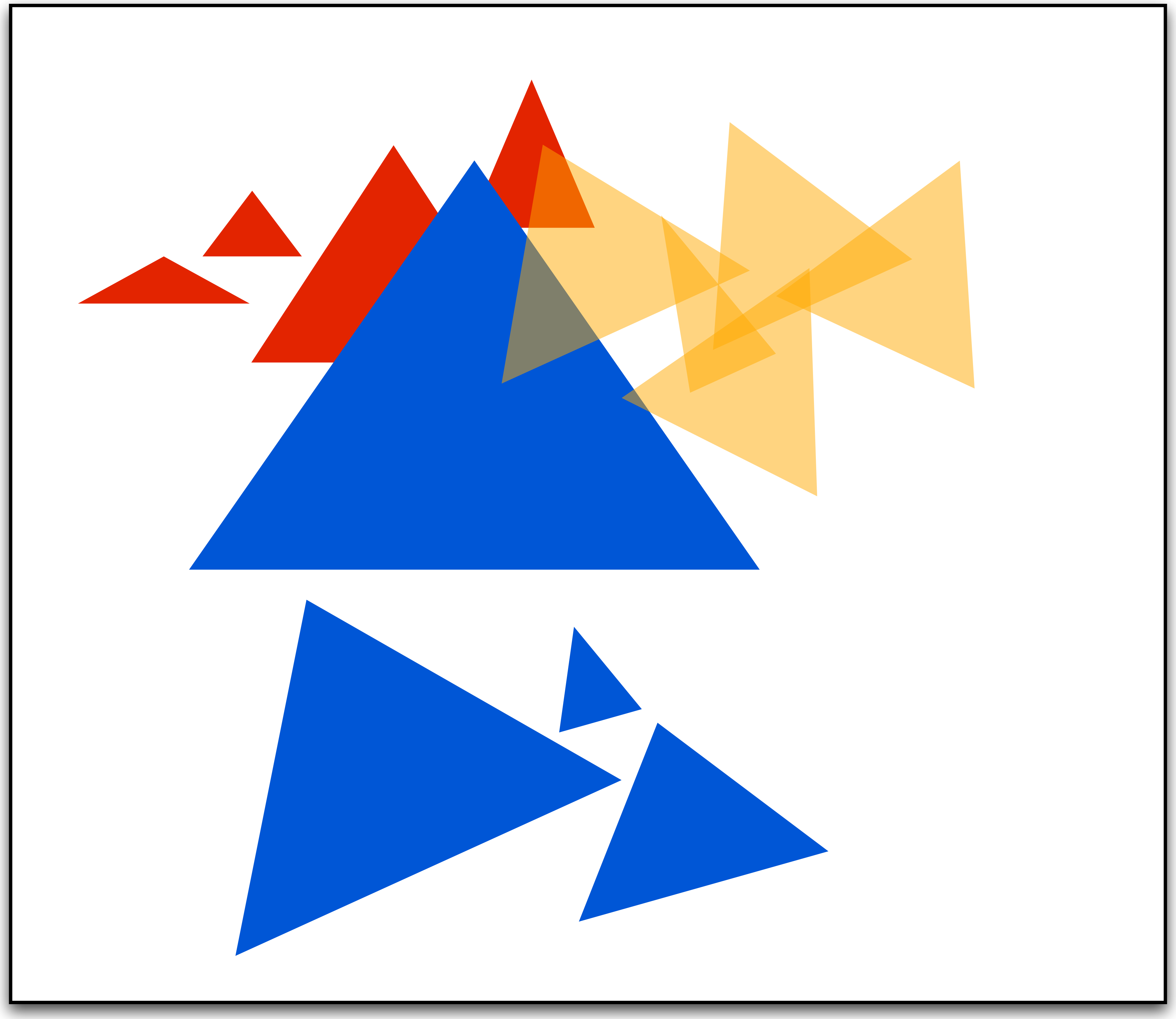
Draw

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

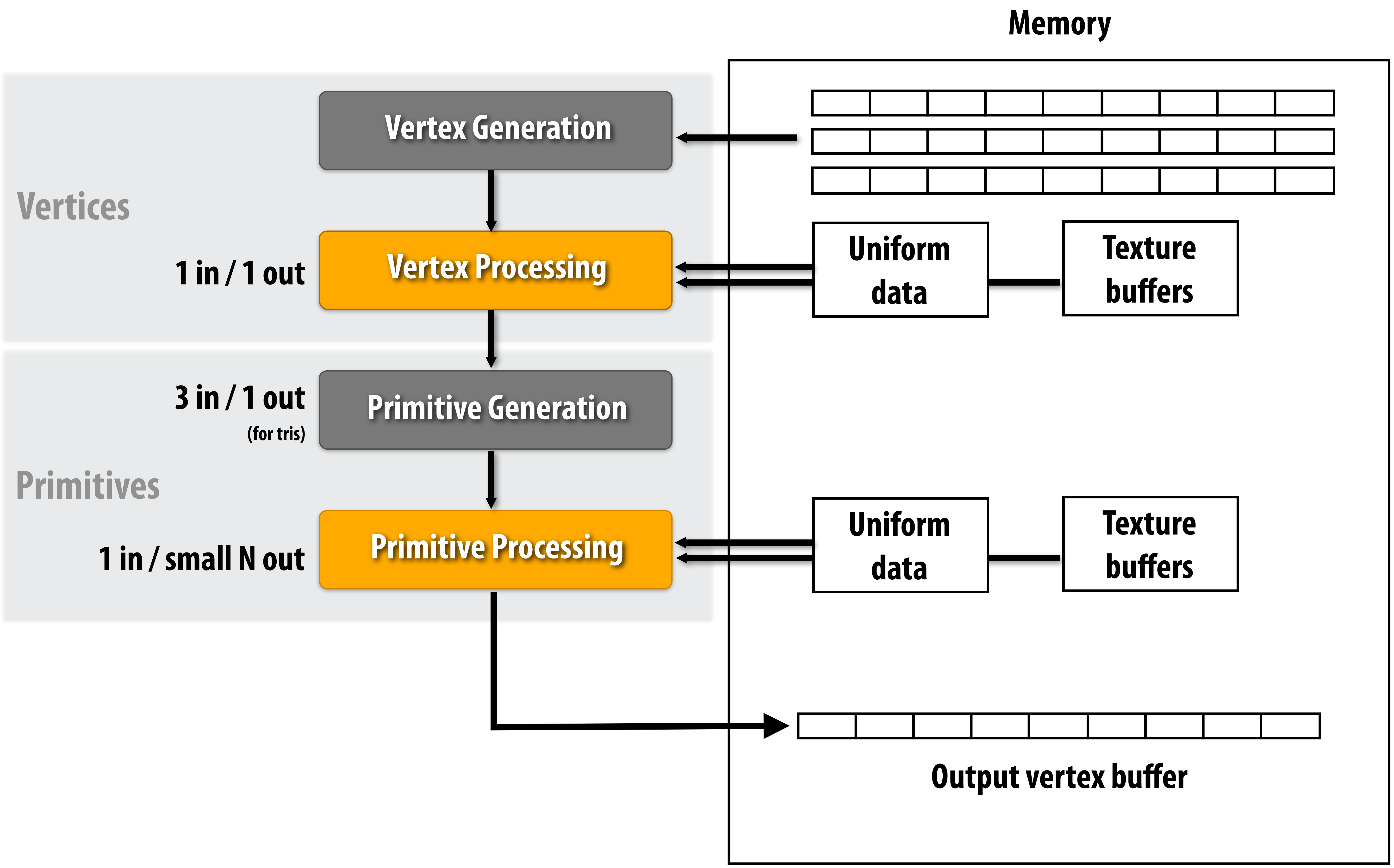
Command Type	Command
Draw	Draw using vertex buffer for object 5
Draw	Draw using vertex buffer for object 6
State change	Bind contents of output image as texture 1
Draw	Draw using vertex buffer for object 5
Draw	Draw using vertex buffer for object 6
	⋮

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 ⟶ save intermediate geometry



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 <u>DOES NOT</u> 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"
- <u>Programmable</u> stages provide application large amount of flexibility (e.g., to implement wide variety of materials and lighting techniques)
- <u>Configurable</u> (but not programmable) pipeline structure: 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

Orthogonality of abstractions

- All vertices treated the same regardless of primitive type
  - Result: vertex programs 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, modeling transforms**
  - Only vertices, primitives, fragments, pixels, and STATE (state = buffers, shaders, and configuration parameters)
  - Applications use these basic abstractions to implement lights, materials, etc.

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

- **No I/O or OS window management**

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 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 <u>how</u> to render scene to the application

Homework exercise: describe one example of a graphics pipeline design decision that enables high-performance implementations.

Perspective from 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 amplifier effect

Readings

- **Required**
  - D. Blythe. The Direct10 System. SIGGRAPH 2006

- **Suggested:**
  - Chapter 2 and 3 of Real-Time Rendering, Third Edition (see link on course site)
  - D. Blythe, Rise of the Graphics Processor. Proceedings of the IEEE, 2008
  - M. Segal and K. Akeley. The Design of the OpenGL Graphics Interface