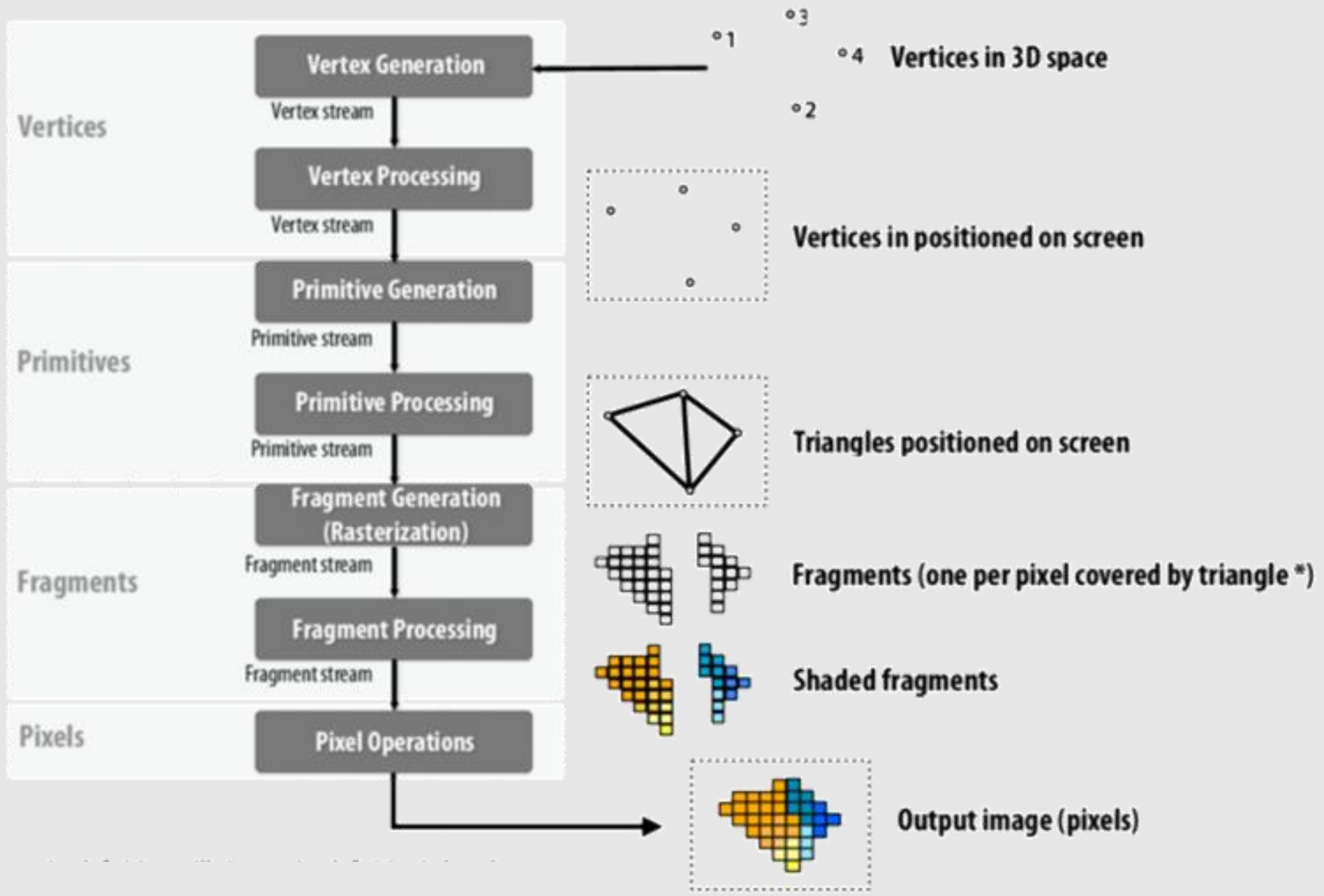
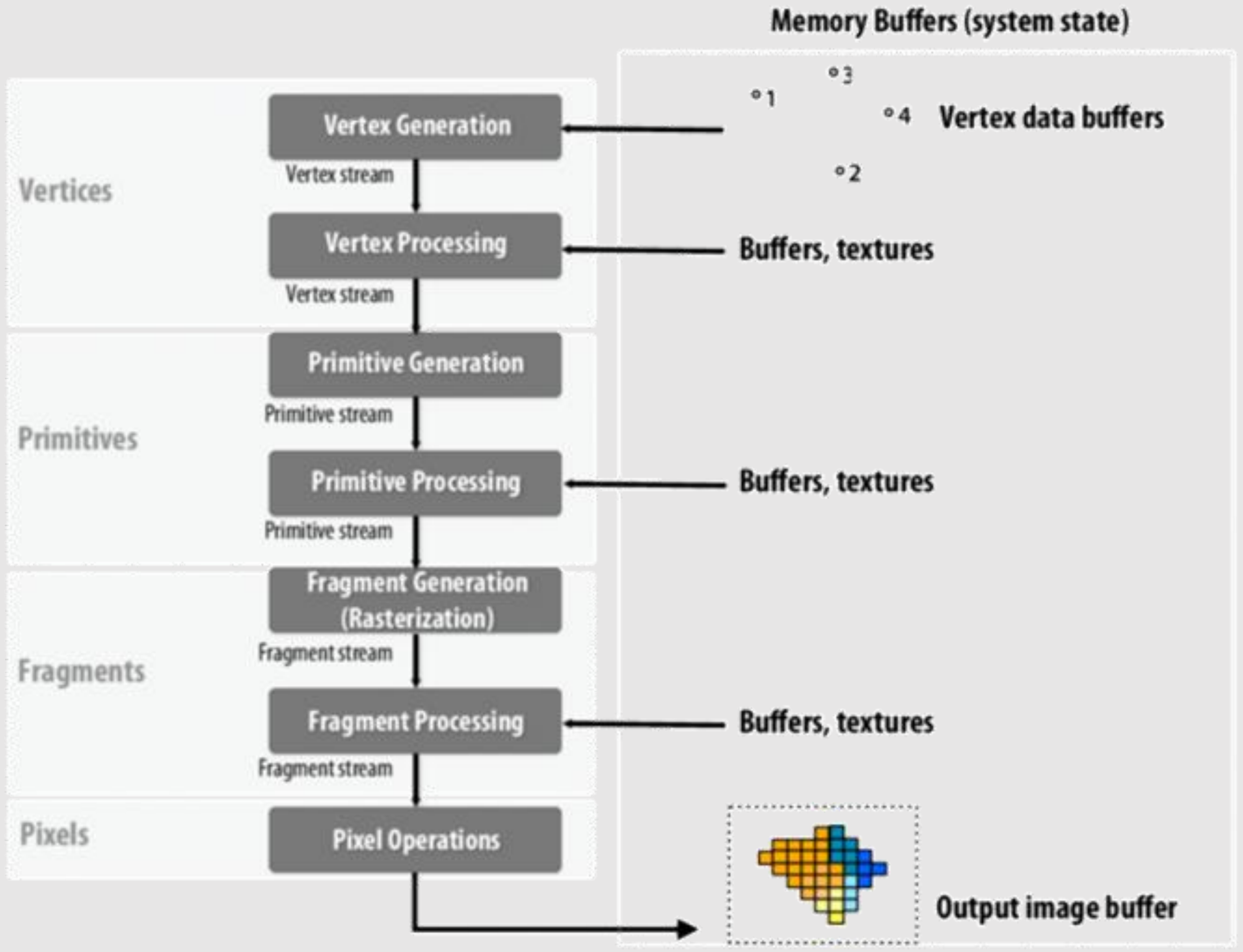
Special Topics in A1: Graphics APIs & Architecture

- The Graphics Pipeline
- Graphics APIs
- Deferred Shading
- Graphics Architecture

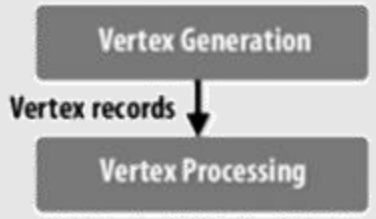
The Graphics Pipeline

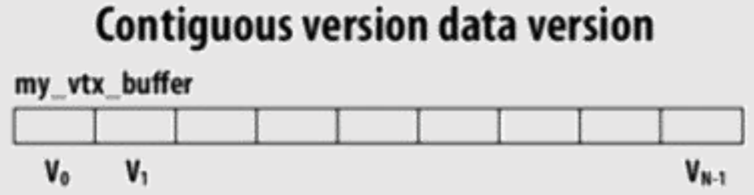


The Graphics Pipeline

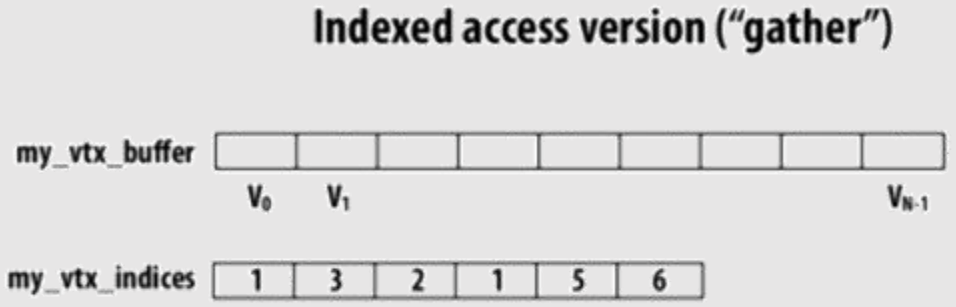


Assembling Vertices



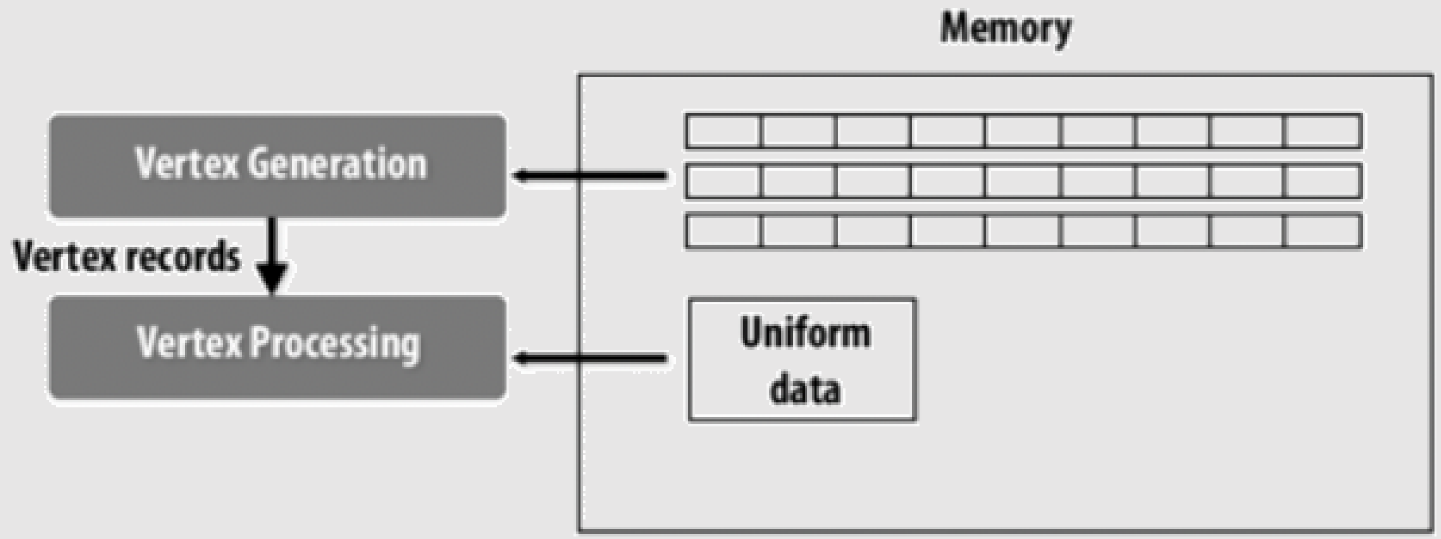


```
glBindBuffer(GL_ARRAY_BUFFER, my_vtx_buffer);
glDrawArrays(GL_TRIANGLES, 0, N);
```



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

Vertex Shader

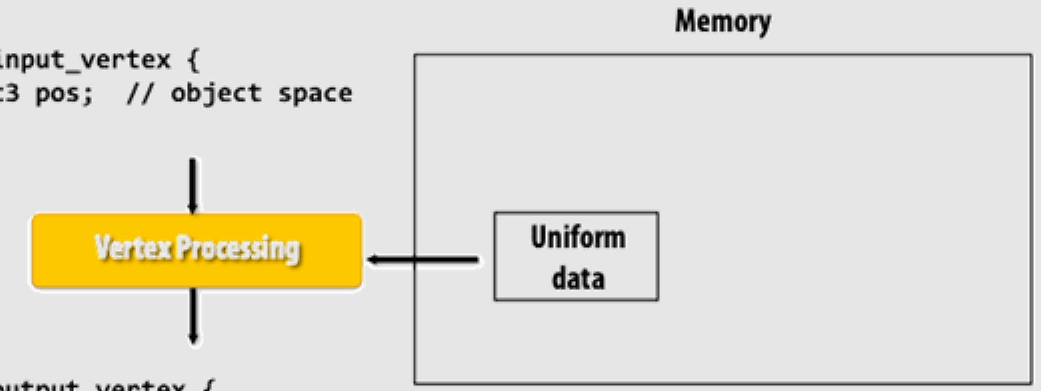


- **Uniform data:** constant read-only data provided as input to every instance of the vertex shader
  - Example: object-to-clip-space vertex transform matrix

- Vertex processing operates on a stream of vertex records + read-only “uniform” inputs
  - Easily amendable to parallelism

Vertex Shader

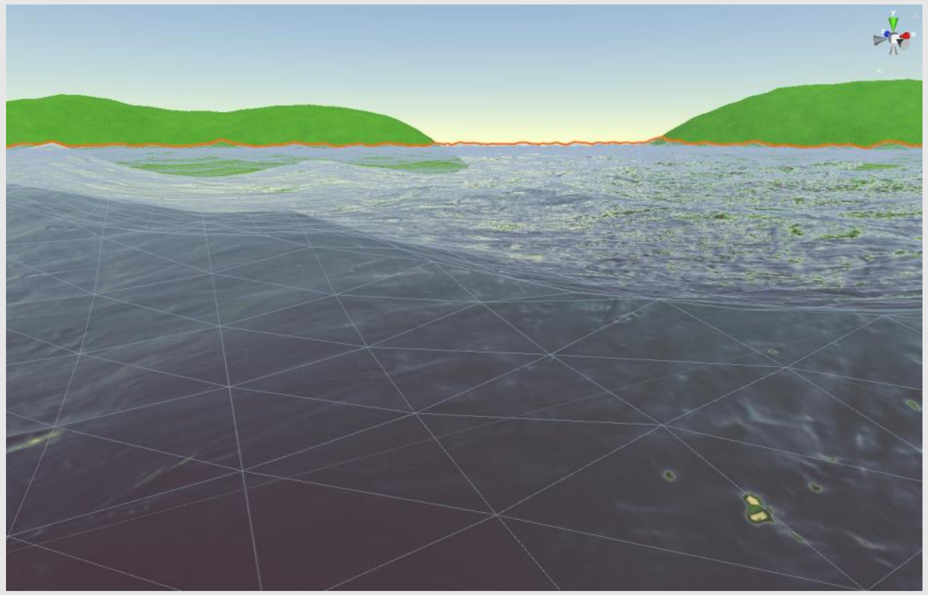
```
struct input_vertex {
   float3 pos;  // object space
};
```



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

1 input vertex $\longrightarrow$ 1 output vertex

independent processing of each vertex



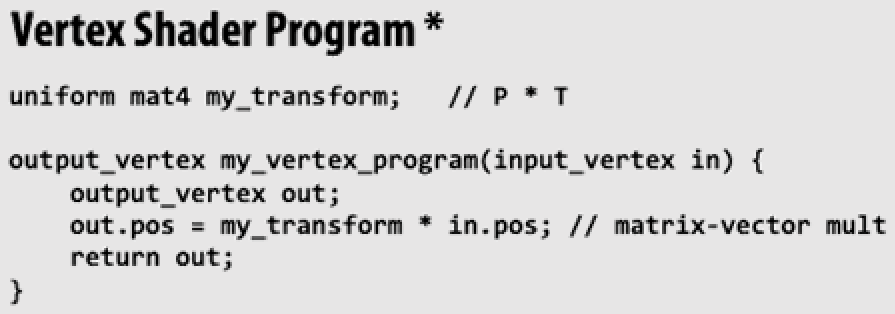
Vertex Shader Program *

```
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;
}
```

- Vertex shaders provide per-vertex operations that change attributes of vertices such as their positions and normal
  - Example: fluid simulation

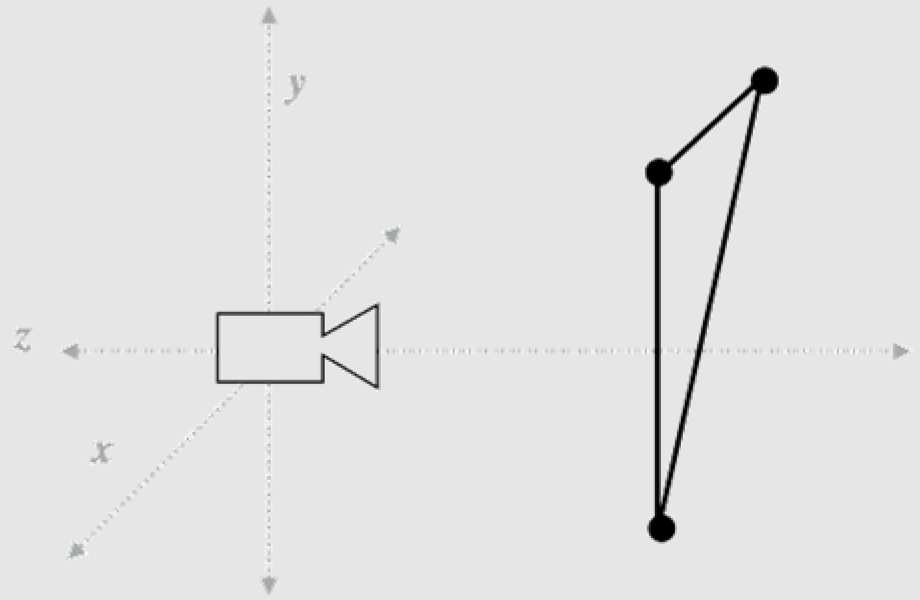
* (Note this is pseudocode, not GLSL syntax)

Vertex Shader



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;
}
```

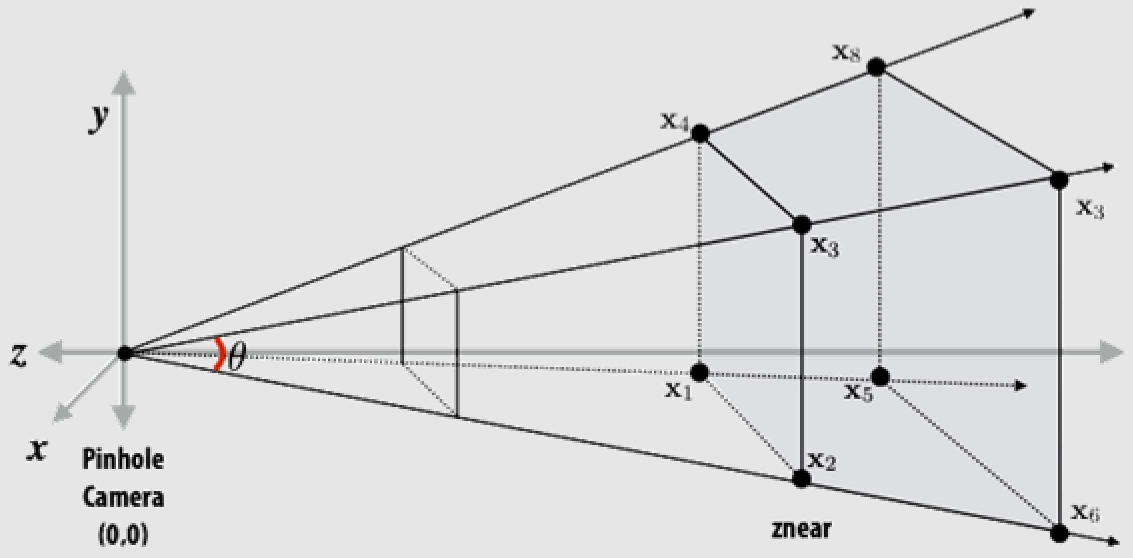


Transform triangle vertices from world-space coordinates into camera space

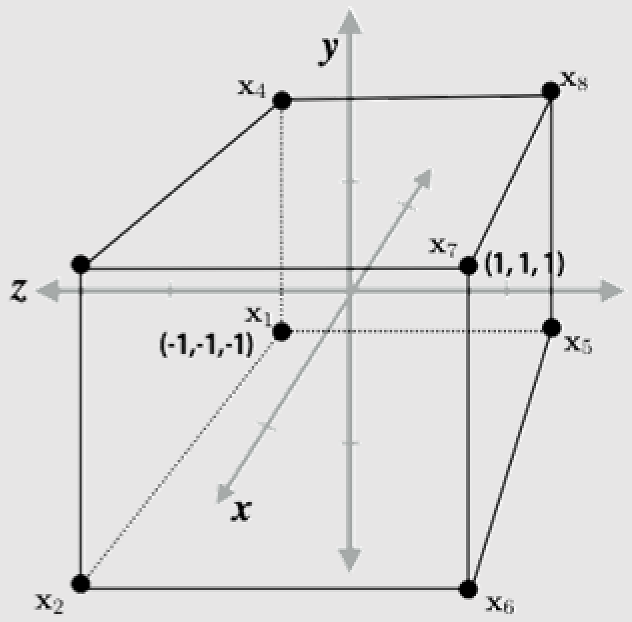
Vertex Shader

Camera Space

also known as camera view frustrum

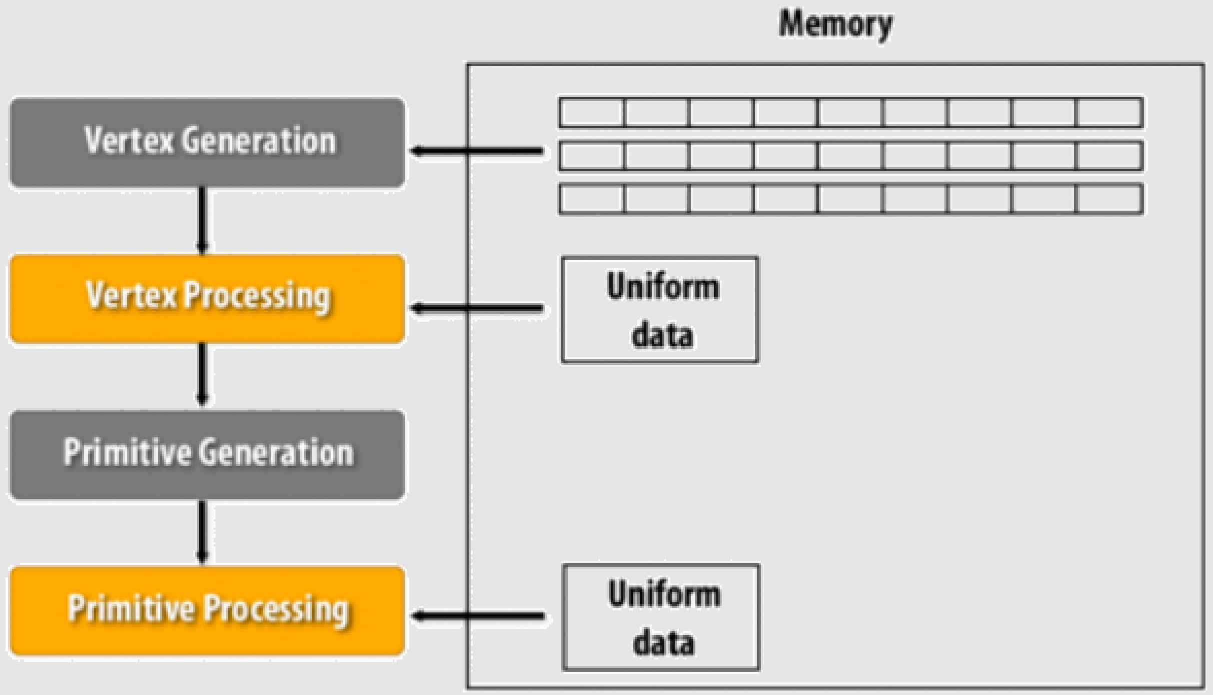


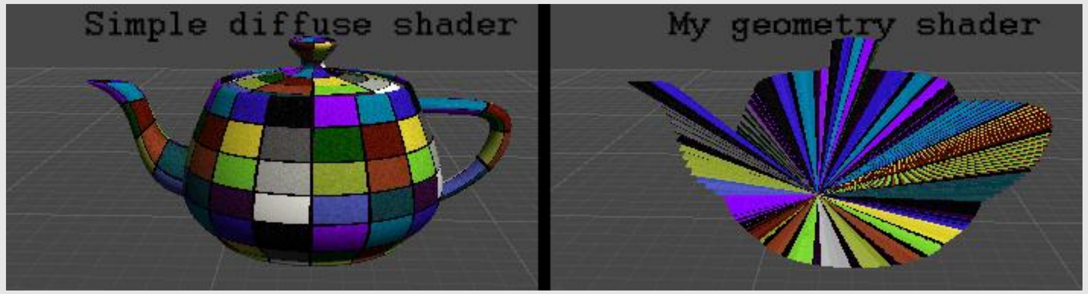
Normalized Space



Apply perspective projection transform to transform triangle vertices to normalized coordinate space

Geometry Shader





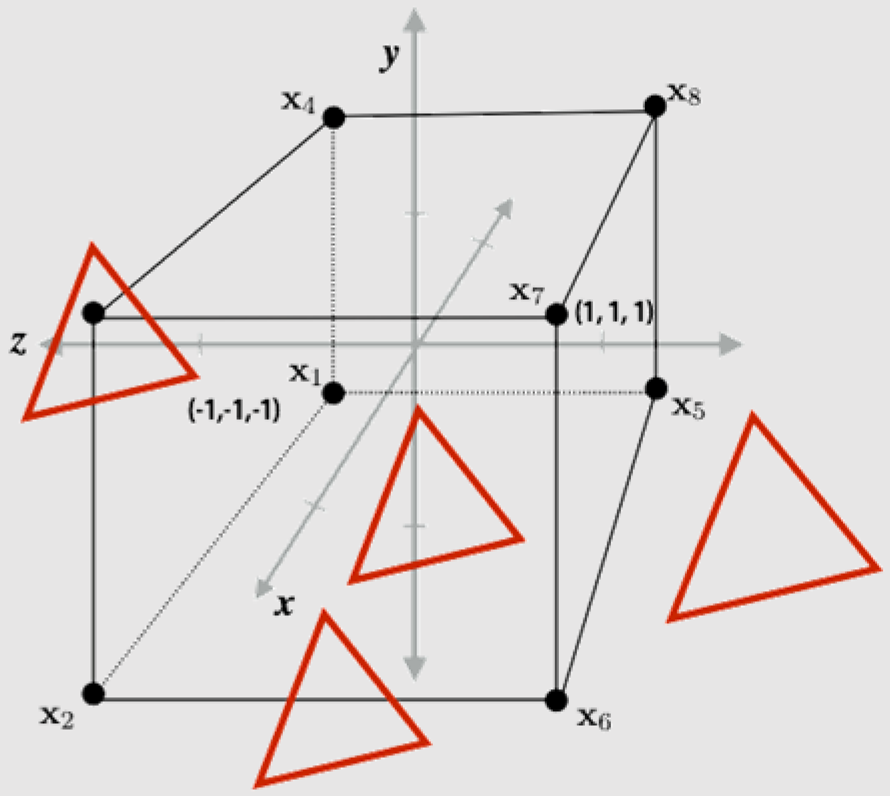
Some guy in Unity having a hard time with geometry shaders (2014)

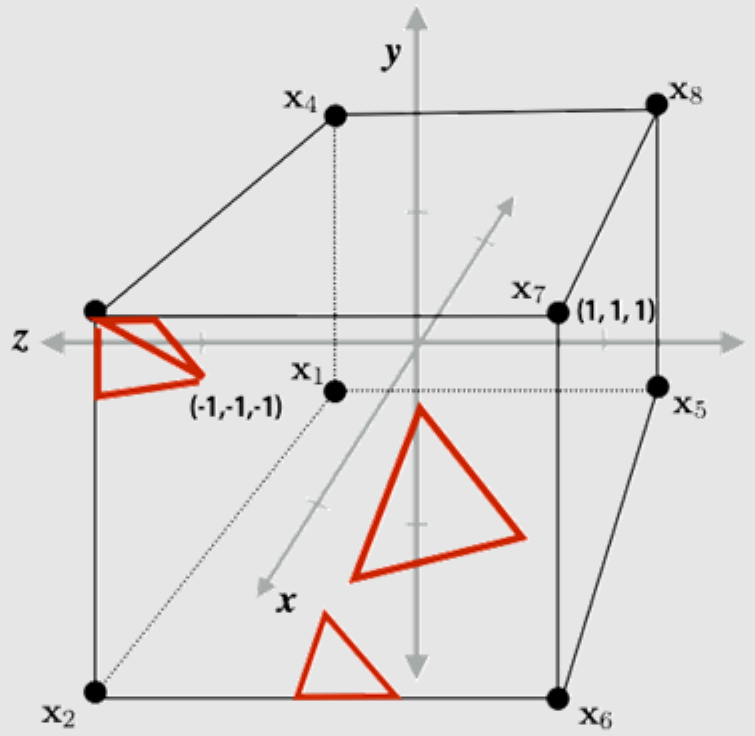
- Recently added to OpenGL in 2007
  - As such, not many people use it

- Allows user to retarget the connectivity of their geometry by specifying tessellation operations or adding in additional geometry
  - Example: computing vertex shader on coarse geometry and then subdividing a surface in the geometry shader

* "Geometry Shader" in OpenGL/Direct3D/Metal terminology

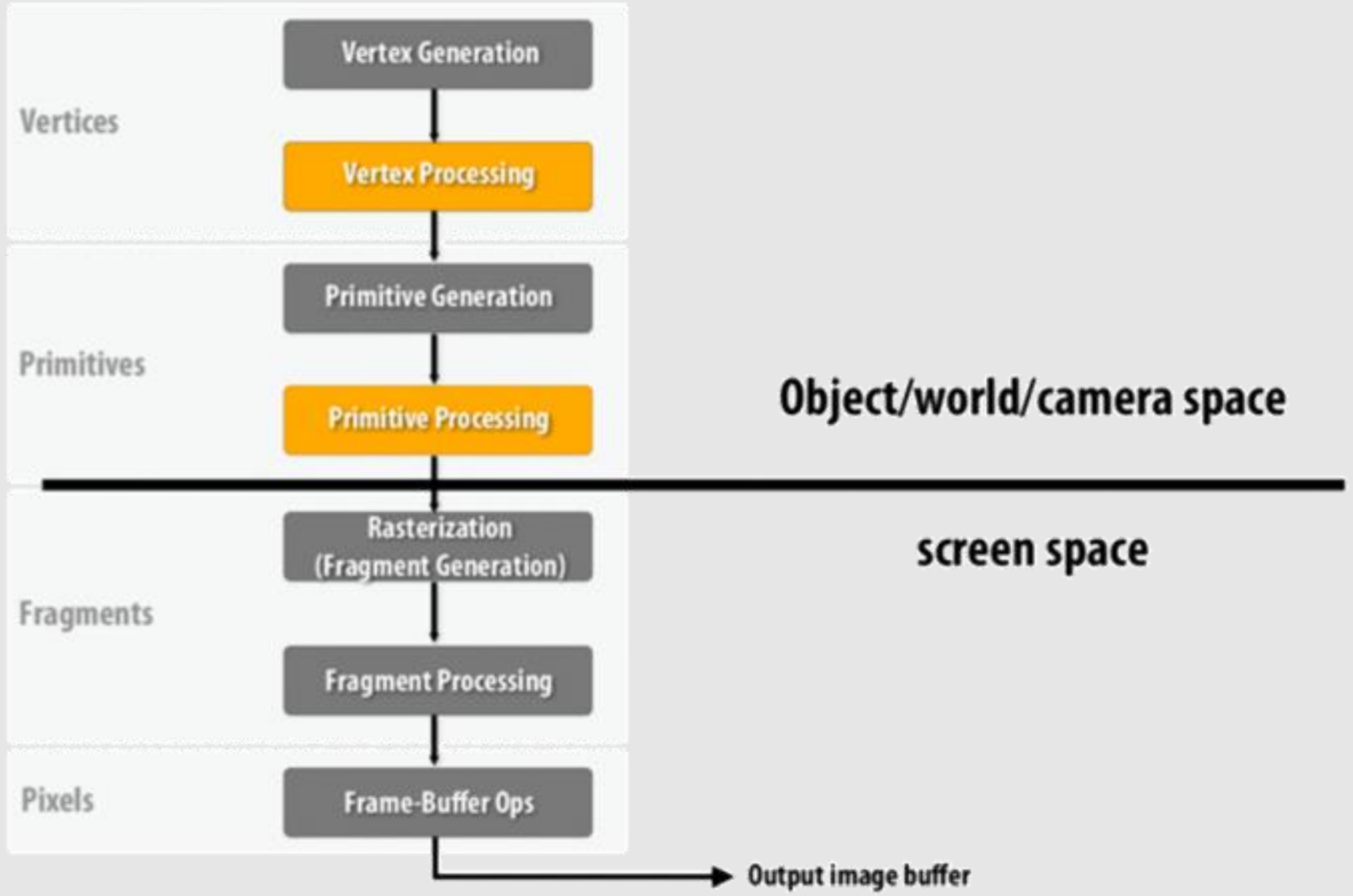
Geometry Shader (Clipping)



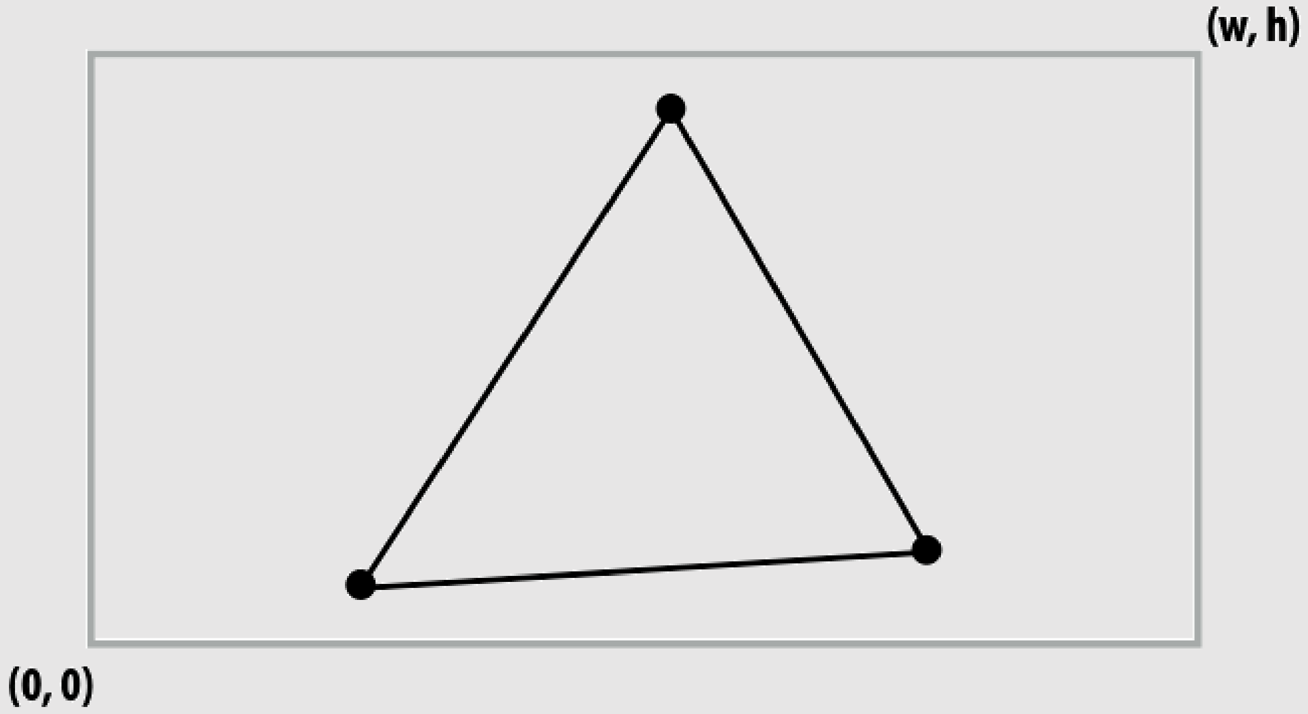


Discard triangles that lie complete outside the unit cube (culling). This may create additional triangles

Change Of Space

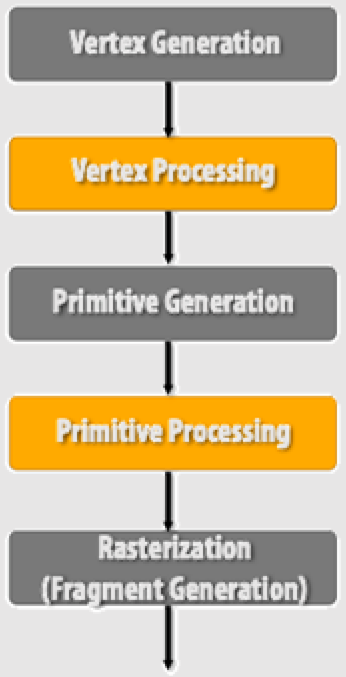


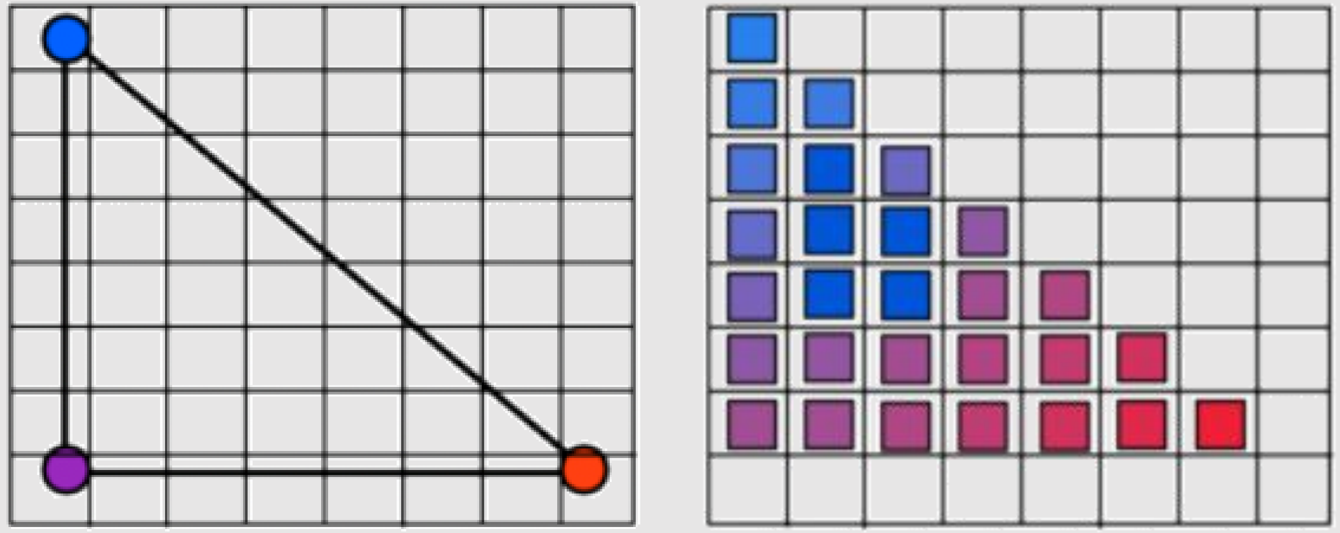
Transform To Screen Coordinates



Transform vertex (x,y) positions from normalized coordinates into screen coordinates (based on screen [w,h])

Fragment Shader

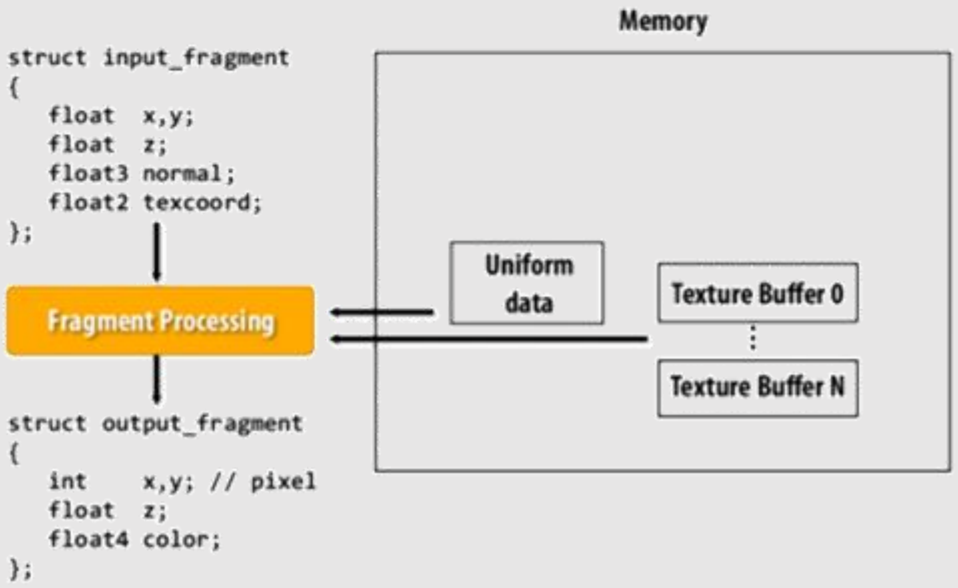




```
struct fragment      // note similarity to output_vertex from before
{
   float  x,y;       // screen pixel coordinates (sample point location)
   float  z;         // depth of triangle at sample point

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

}
```

- Separate triangles into fragments
- For each fragment, compute the output RGBA

Texturing



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

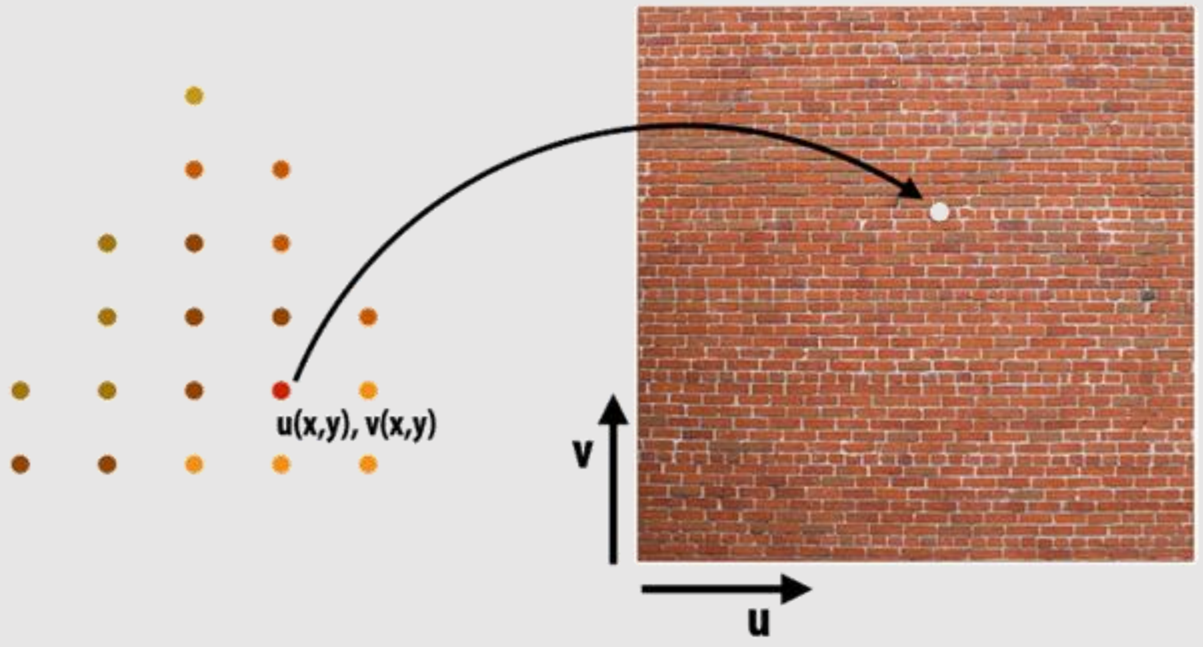
```
struct output_fragment
{
   int    x,y; // pixel
   float  z;
   float4 color;
};
```

```
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;
}
```

- Textures stored as images in memory
- Most expensive part of rasterization
    - Texture caching is a popular solution

Texturing

```
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;
}
```

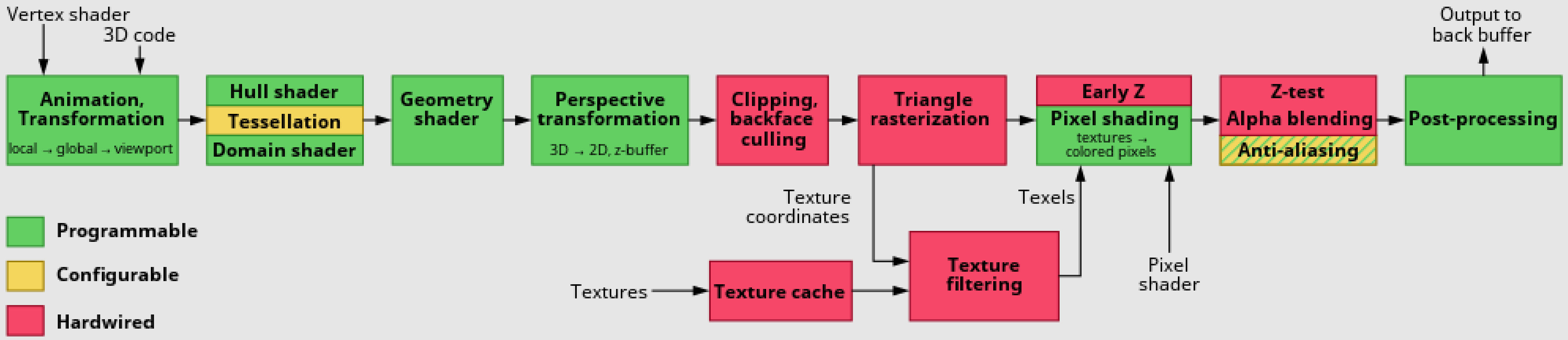


Texture is just a buffer of values. Map (x,y) coordinates to (u,v) texture coordinates and look up

The Texture Pipeline

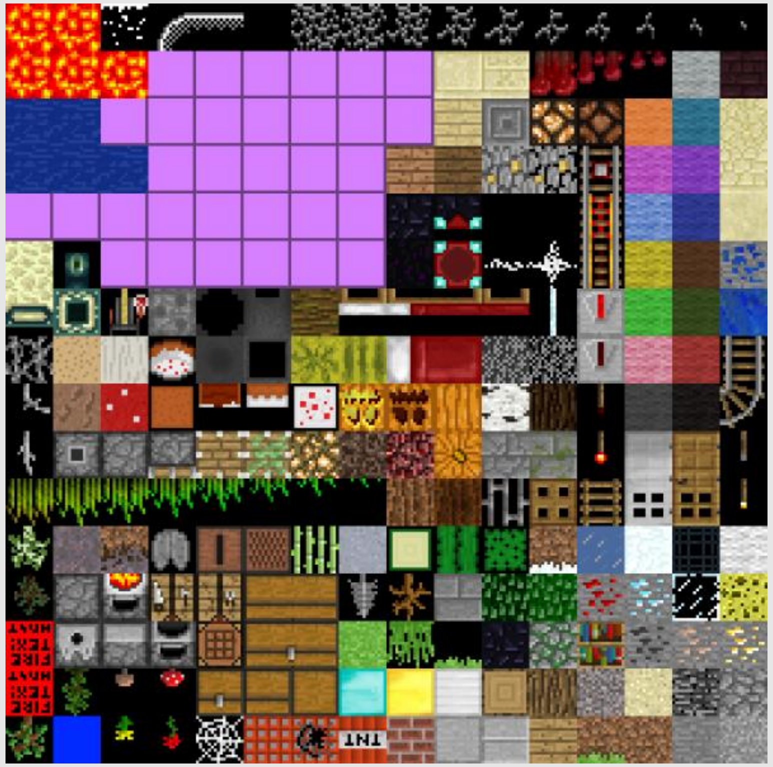
- During the rasterization stage, fragment shader has access to a texture interface
  - An I/O for textures, with its own texture cache!
    - Useful since neighboring fragments will request neighboring texels
  - Built-in texture filtering operations
  - Texture operations are fixed-function
    - Designed to be hardware optimized given that they can be called several times per fragment

Surface Maps via Adaptive Triangulations (2023) Schmidt et. al.

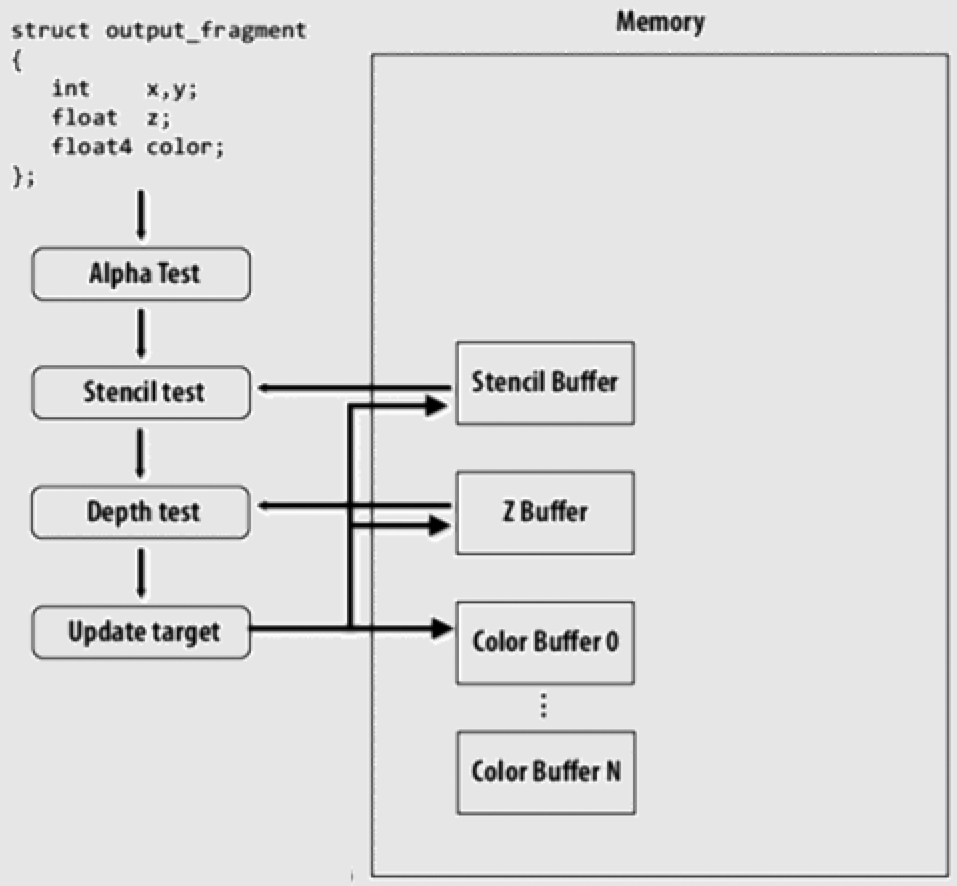


Texture Atlas

- In old school 2D video games, it was expensive to load different images
  - Different images meant different cache locations, which lead to more cache conflicts
  - IO was expensive
- To keep important texture data readily available in the cache, important assets were combined into one image
  - Frequently referenced → frequently in cache
- Modern video game architectures have better memory/caching/IO
  - No need for texture atlases
  - Still common to keep all assets pertaining to one character in same texture


Minecraft (2011) Mojang

Frame-Buffer Operations

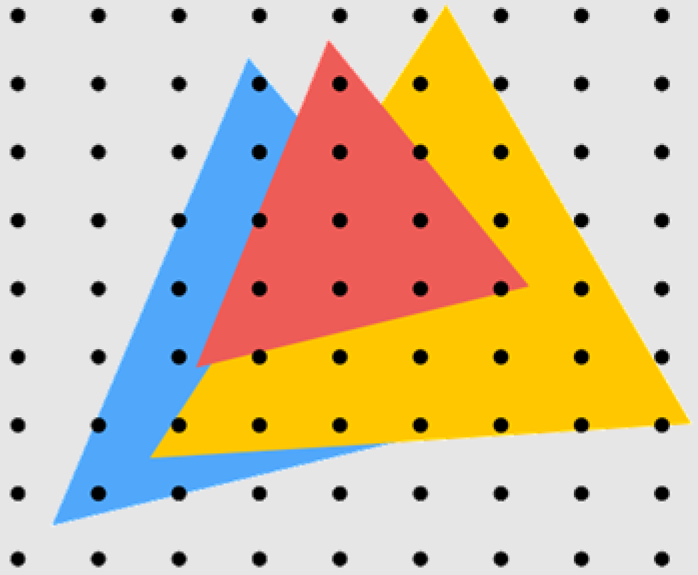


```
struct output_fragment
{
   int    x,y;
   float  z;
   float4 color;
};
```

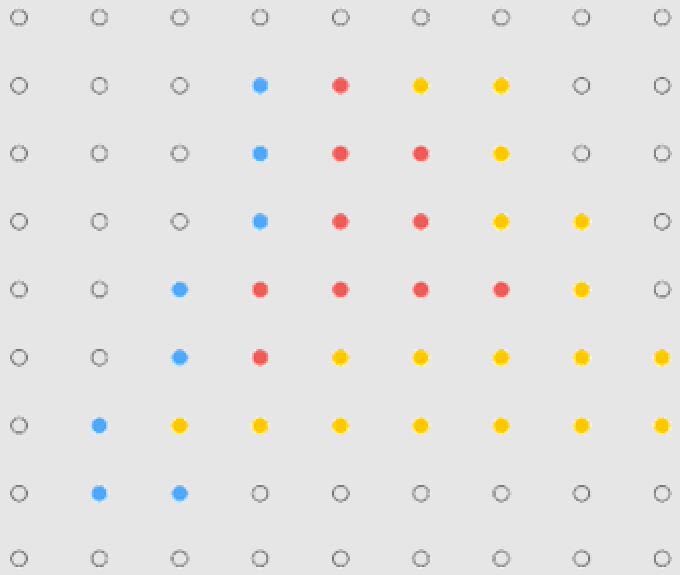
- **Alpha Test**
  - Allows fragments with alpha value greater/less than a constant specified by the user to pass
- **Stencil Test**
  - Allows fragments that pass a user-defined per-pixel function to pass
    - Stored in stencil buffer
    - Example: mattes + masking
- **Depth Test**
  - Allows fragments that are closest in depth to pass
    - Stored in Z-buffer
- **Update Target**
  - If pixel passes, modify stencil depth and color buffers
  - Reads can be done in parallel, writes require locking
    - We'll look at techniques later to accelerate this

Depth Test

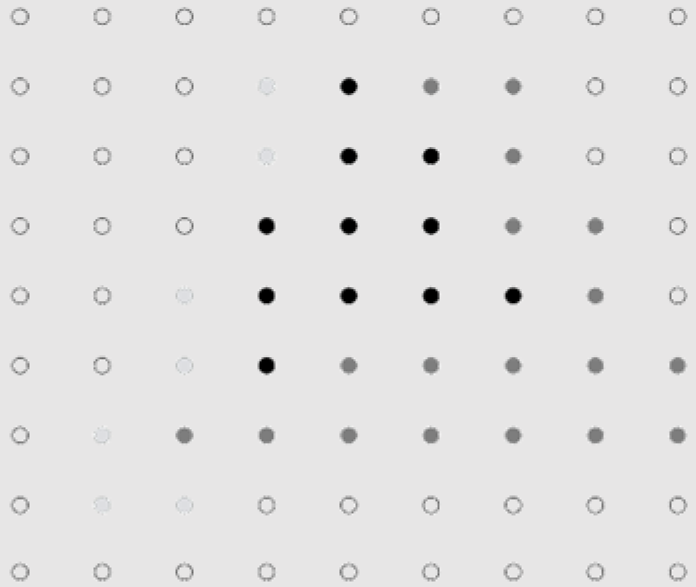
Primitives



Color Buffer



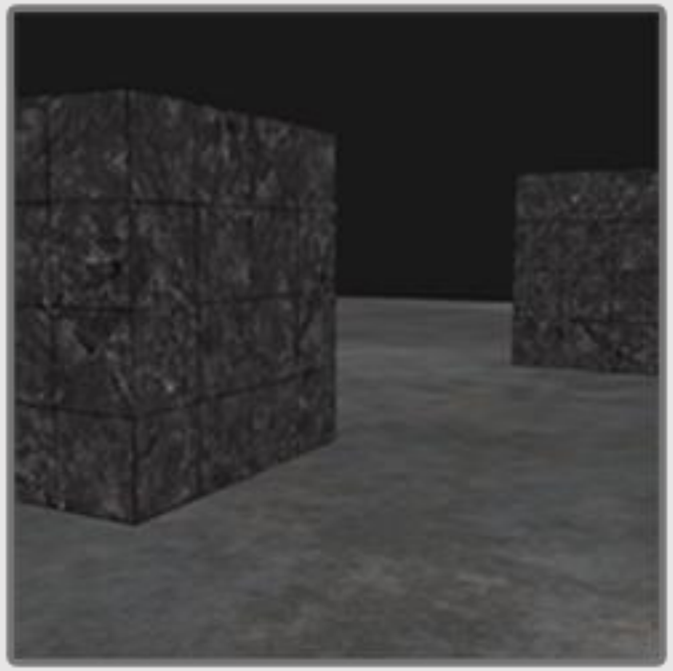
Depth Buffer



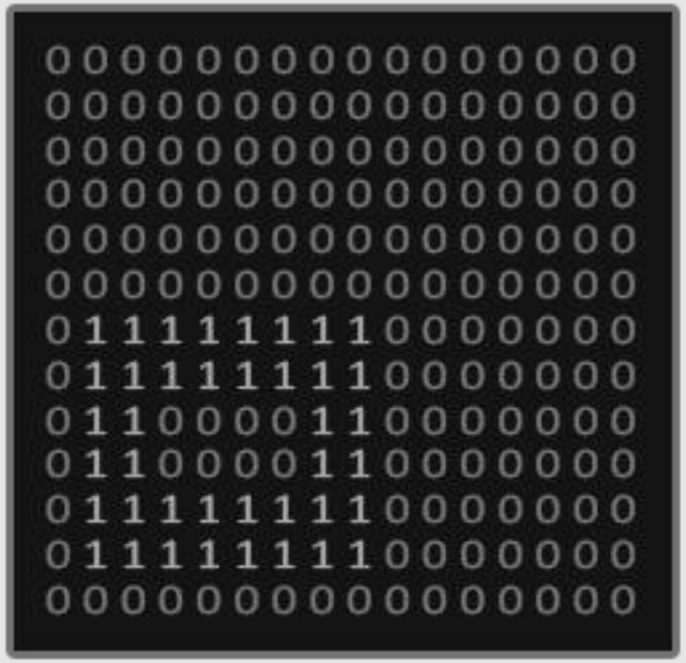
```
depth_test(tri_d, tri_color, x, y) {

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

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

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

Stencil Test

Color Buffer

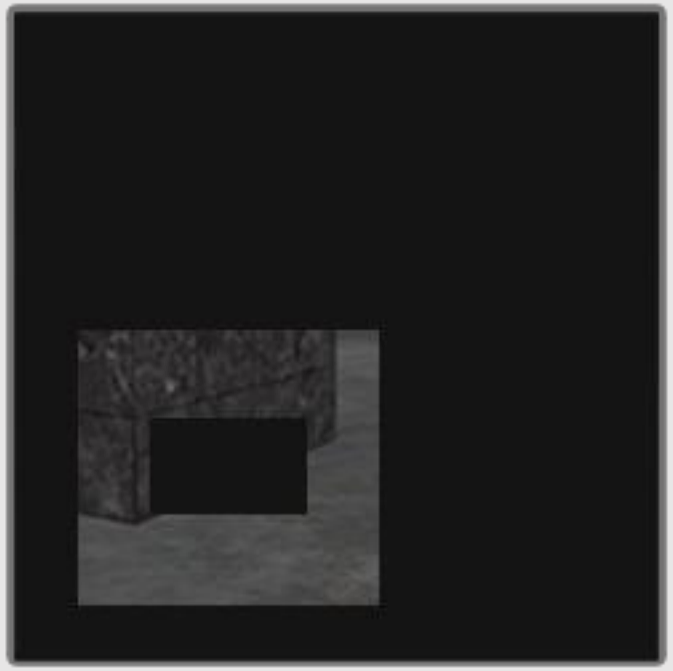


Stencil Buffer

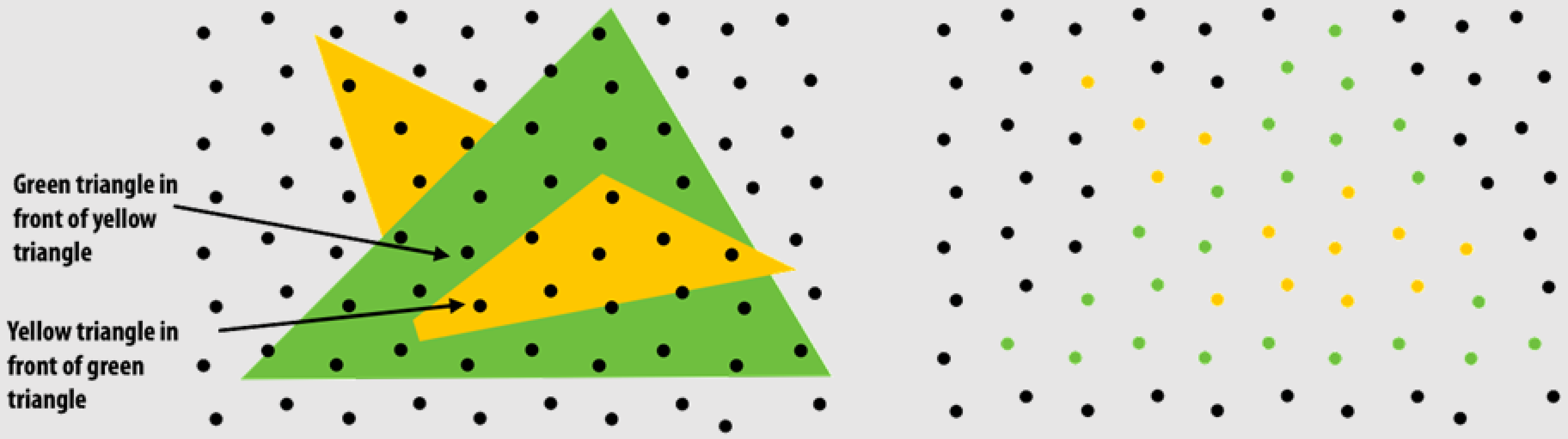


```
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0000000000000000
0111111110000000
0111111110000000
0110000110000000
0110000110000000
0111111110000000
0111111110000000
0000000000000000
```

Result

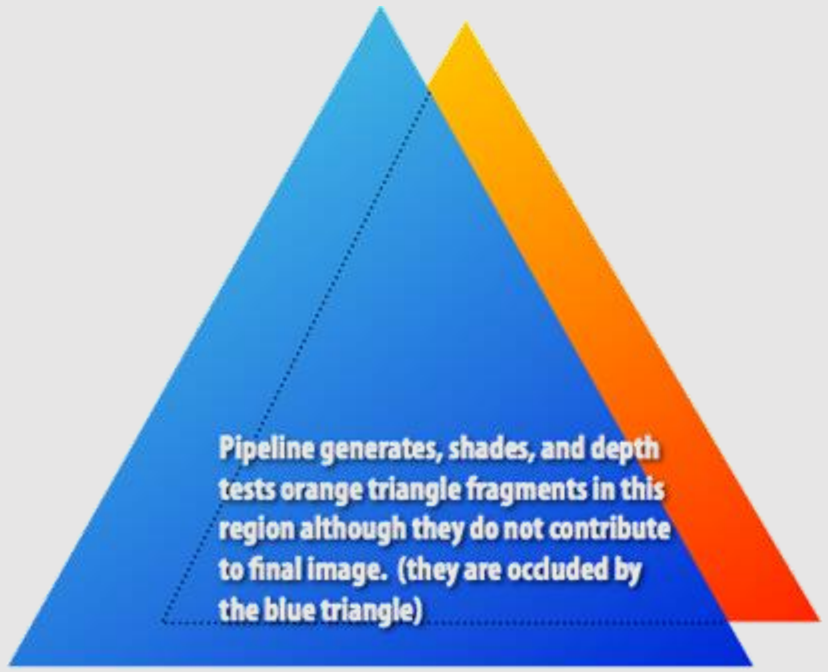


Depth Test


Green triangle in front of yellow triangle
Yellow triangle in front of green triangle

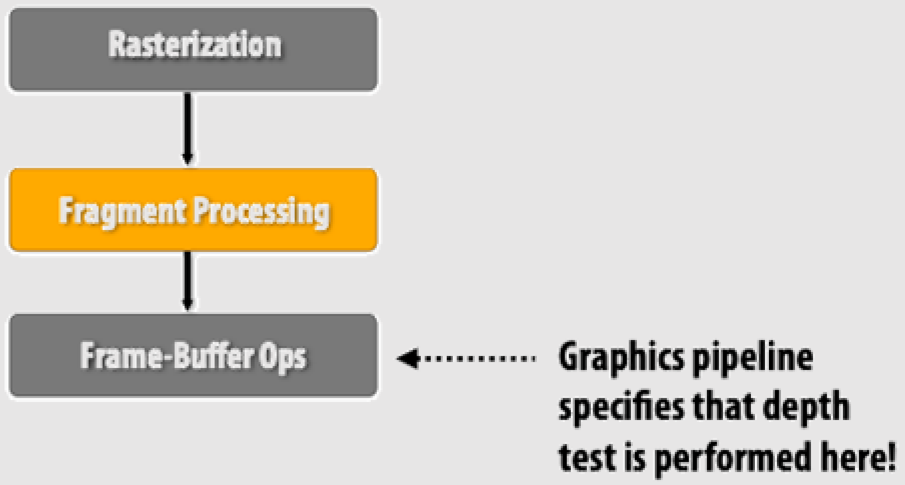
- Q: Does depth-buffer algorithm handle interpenetrating surfaces?
    - A: Of course! Occlusion test is based on depth of triangles at a given sample point. The relative depth of triangles may be different at different sample points

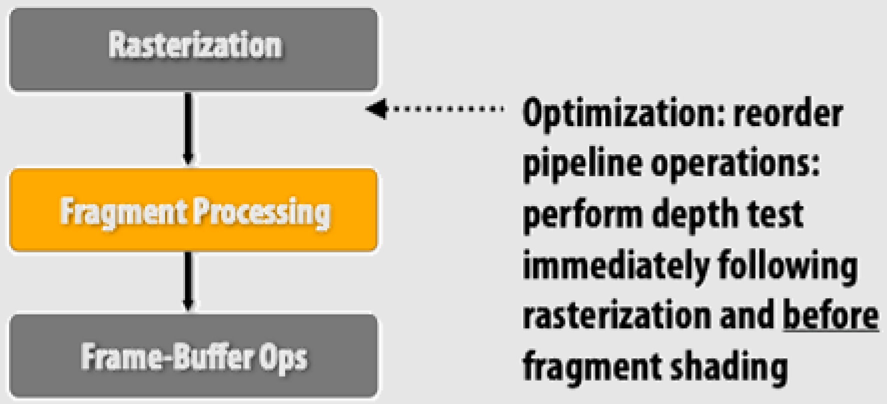
Early occlusion-culling ("early Z")


Pipeline generates, shades, and depth tests orange triangle fragments in this region although they do not contribute to final image. (they are occluded by the blue triangle)

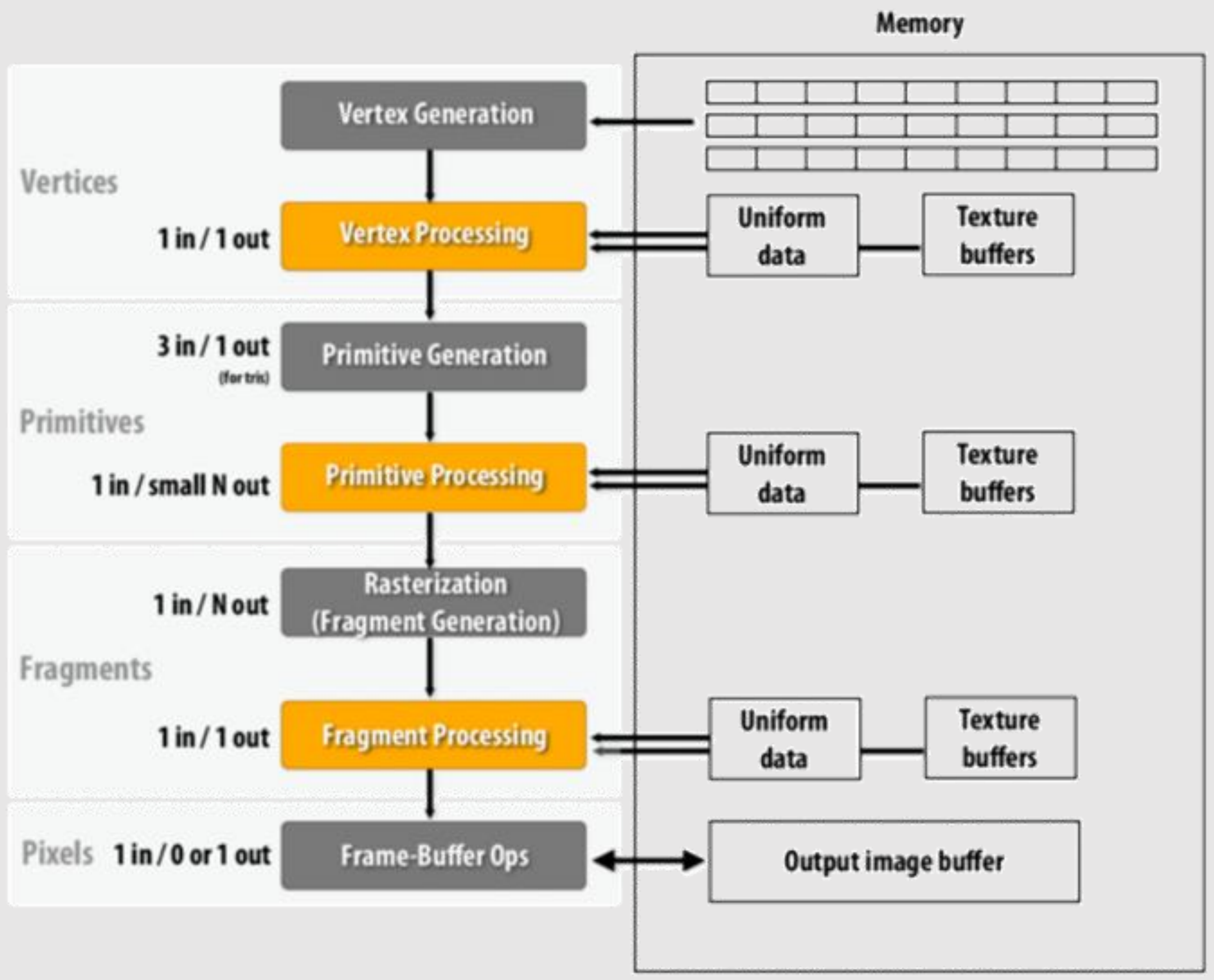
Early occlusion-culling ("early Z")

- A GPU implementation detail: not reflected in the graphics pipeline abstraction
  - But it's still there
- Key assumption: occlusion results do not depend on fragment shading
  - Note: early Z only provides benefit if closer triangle is rendered by application first!
  - Application developers are encouraged to submit geometry in front-to-back order


Rasterization
Fragment Processing
Frame-Buffer Ops
Graphics pipeline specifies that depth test is performed here!


Rasterization
Fragment Processing
Frame-Buffer Ops
Optimization: reorder pipeline operations: perform depth test immediately following rasterization and before fragment shading

The 3D Graphics Pipeline



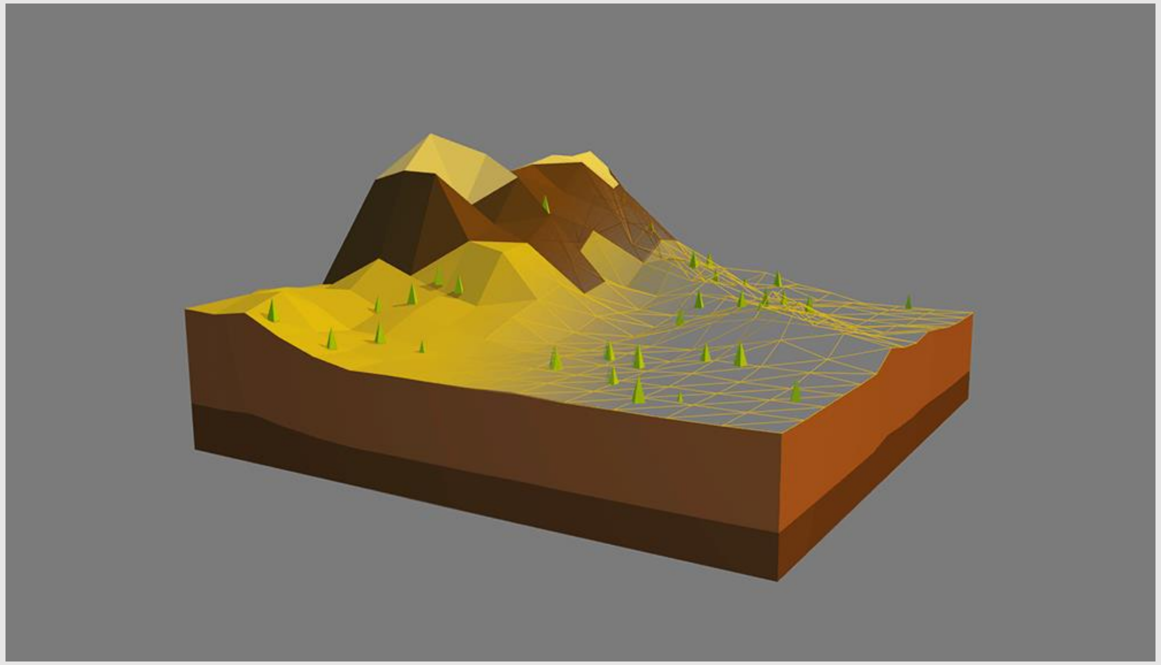
- ~~The Graphics Pipeline~~
- Graphics APIs
- Deferred Shading
- Graphics Architecture

Graphics APIs

- OpenGL
  - Shaders are written in GLSL
  - Cross-Platform support
- Vulkan
  - The cooler OpenGL
  - Provides more control & performance on GPU
- DirectX
  - Microsoft graphics package
  - Direct3D is the suite-specific tool for 3D graphics rendering
- Metal
  - Apple graphics package
  - Written in Objective-C and Swift
  - Dedicated Metal Shading Language
    - Integrated Metal debugger in Xcode

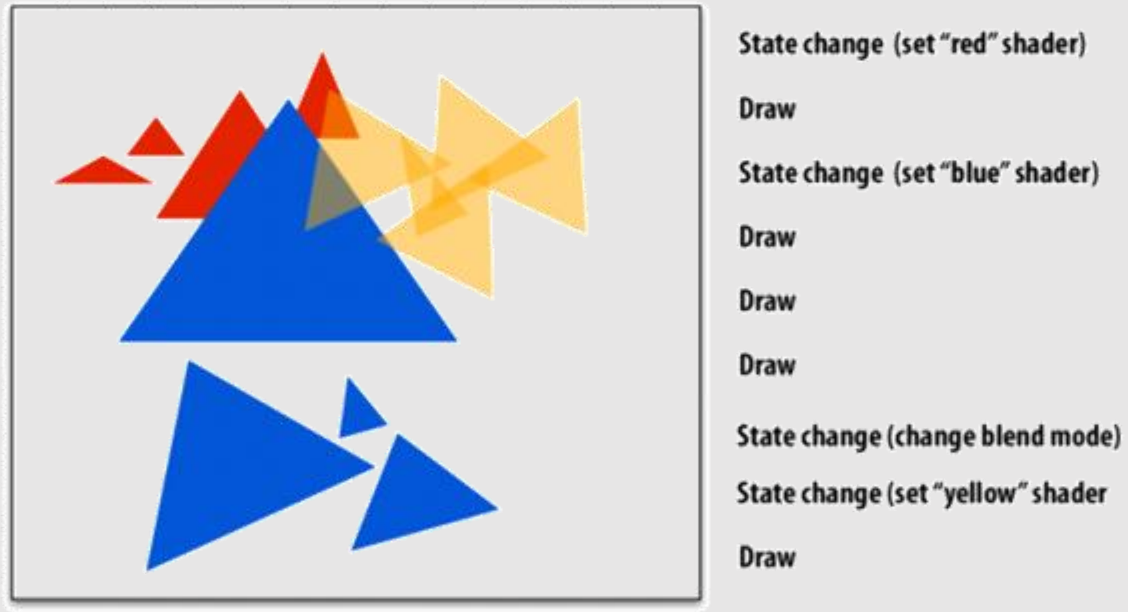
Graphics APIs on the Web

- WebGL
  - JavaScript API for running GLSL code in browser
    - Web port of OpenGL
  - Still runs on GPU
  - Easier means for sharing code/applications
- WebGPU
  - Run GPU commands in browser
  - Successor to WebGL
- Three.JS
  - Higher level graphics framework
  - Uses WebGL under the hood


Three.JS

Programming the Graphics Pipeline

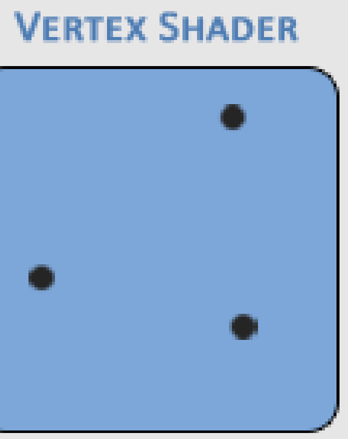
- Graphics programming is referred to as a state machine
  - Every draw **operation** changes the active output's **state**
- Efficiently managing state changes is a major challenge in implementations
  - Not all state changes are the same cost
- Expensive state changes:
  - **Changing targets (texture)**
    - Requires texture cache flush, which severely hurts texture bandwidth
  - **Changing shaders**
    - Forces the pipeline to a stop and flushes contents
    - Trashes any prefetching data as a result of change in access patterns


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

Binding Vertex Shaders

```
// shader code is just a string
// made to look like c++ code
const char *vertexShaderSource = "#version 330 core\n"
   "layout (location = 0) in vec3 aPos;\n"
   "void main()\n"
   "{\n"
   ”   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
   "}\0";
}
```

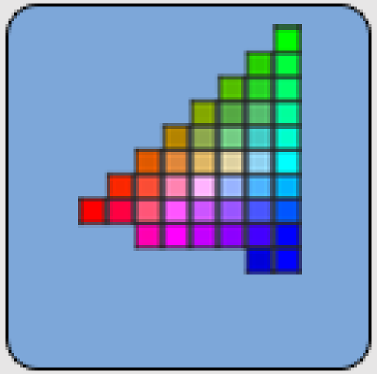
```
// create vertex shader reference
unsigned int vertexShader;
vertexShader = glCreateShader(GL_VERTEX_SHADER);

// bind vertex shader to reference
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);

// compile shader
glCompileShader(vertexShader);
```



Binding Fragment Shaders

```
// shader code is just a string
// made to look like c++ code
const char *fragmentShaderSource = "#version 330 core\n"
   ” out vec4 FragColor; \n"
   "void main()\n"
   "{\n"
   ” FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
   "}\0";
}
```

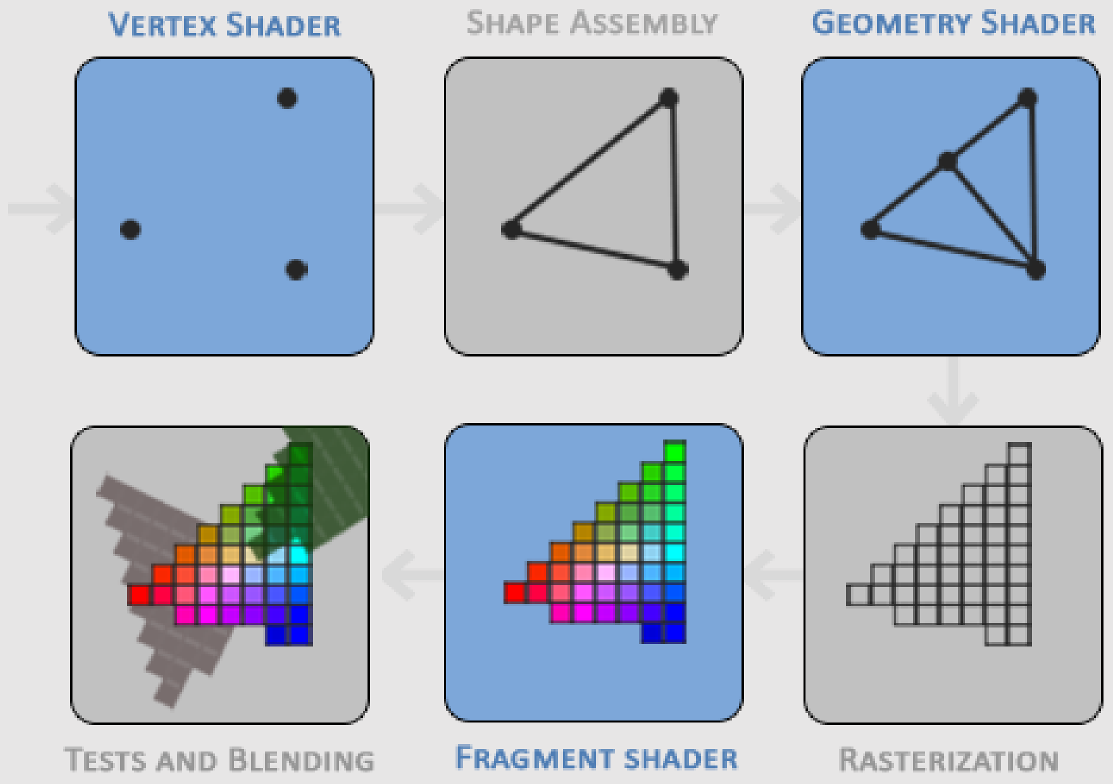
```
// create fragment shader reference
unsigned int fragmentShader;
fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);

// bind fragment shader to reference
glShaderSource(fragmentShader, 1, & fragmentShaderSource, NULL);

// compile shader
glCompileShader(fragmentShader);
```



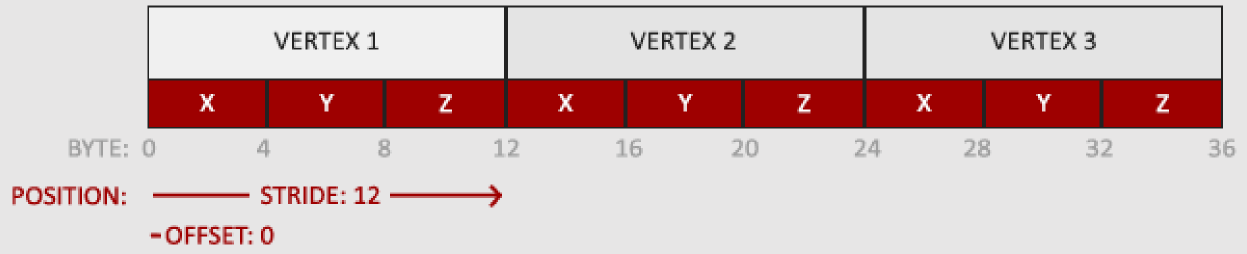
FRAGMENT SHADER

Creating A Graphics Program

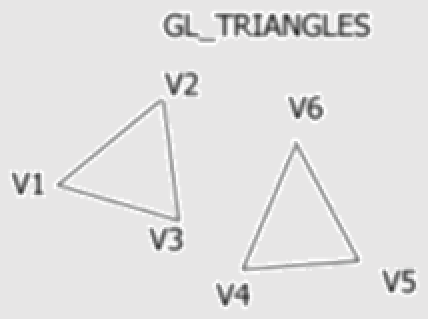
```
// create shader program reference
unsigned int shaderProgram;
shaderProgram = glCreateProgram();

// bind vertex and fragment shaders
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
```

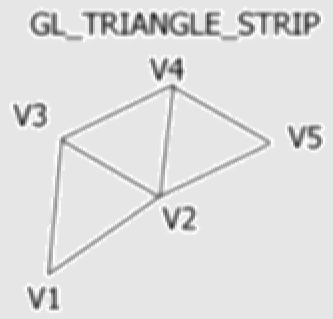


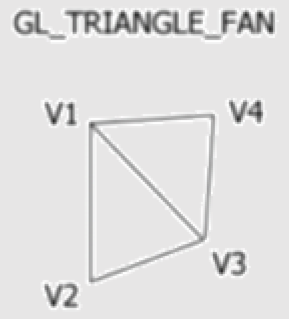
Binding Shaders



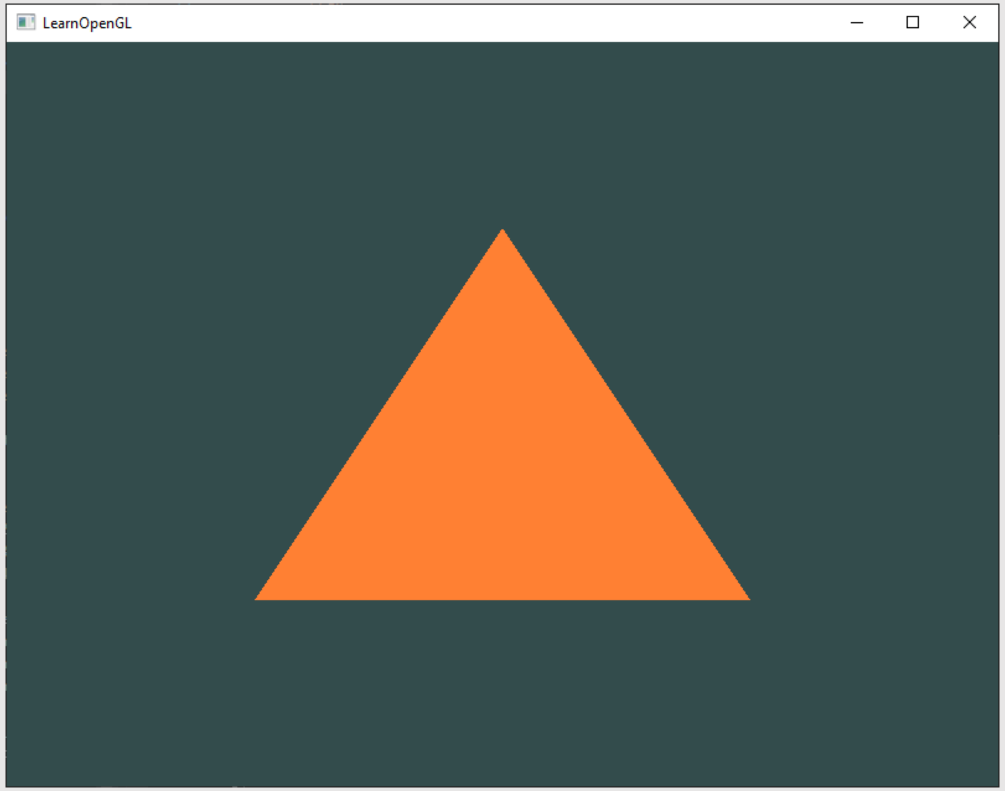
```
// create a Vertex Array Object VAO for multiple VBOs
glBindVertexArray(VAO);

// create a Vertex Buffer Object VBO
// bind vertices array to VBO
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

// specify how vertices will be parsed
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);

// enable shader program and vertex buffer
glUseProgram(shaderProgram);
glBindVertexArray(VAO);

// specify draw method
glDrawArrays(GL_TRIANGLES, 0, 3);
```







The ‘Hello World’ Of Graphics APIs



Hello Triangle, OpenGL

- Instead of printing, we want to draw
  - Easiest thing to draw: **a triangle!**
- Hardest part of Graphics APIs: getting the bindings correct
- If you can draw a triangle, you have your pipeline in order
  - Simply swap out shaders and data, and go from there!

Tutorial: https://learnopengl.com/Getting-started/Hello-Triangle

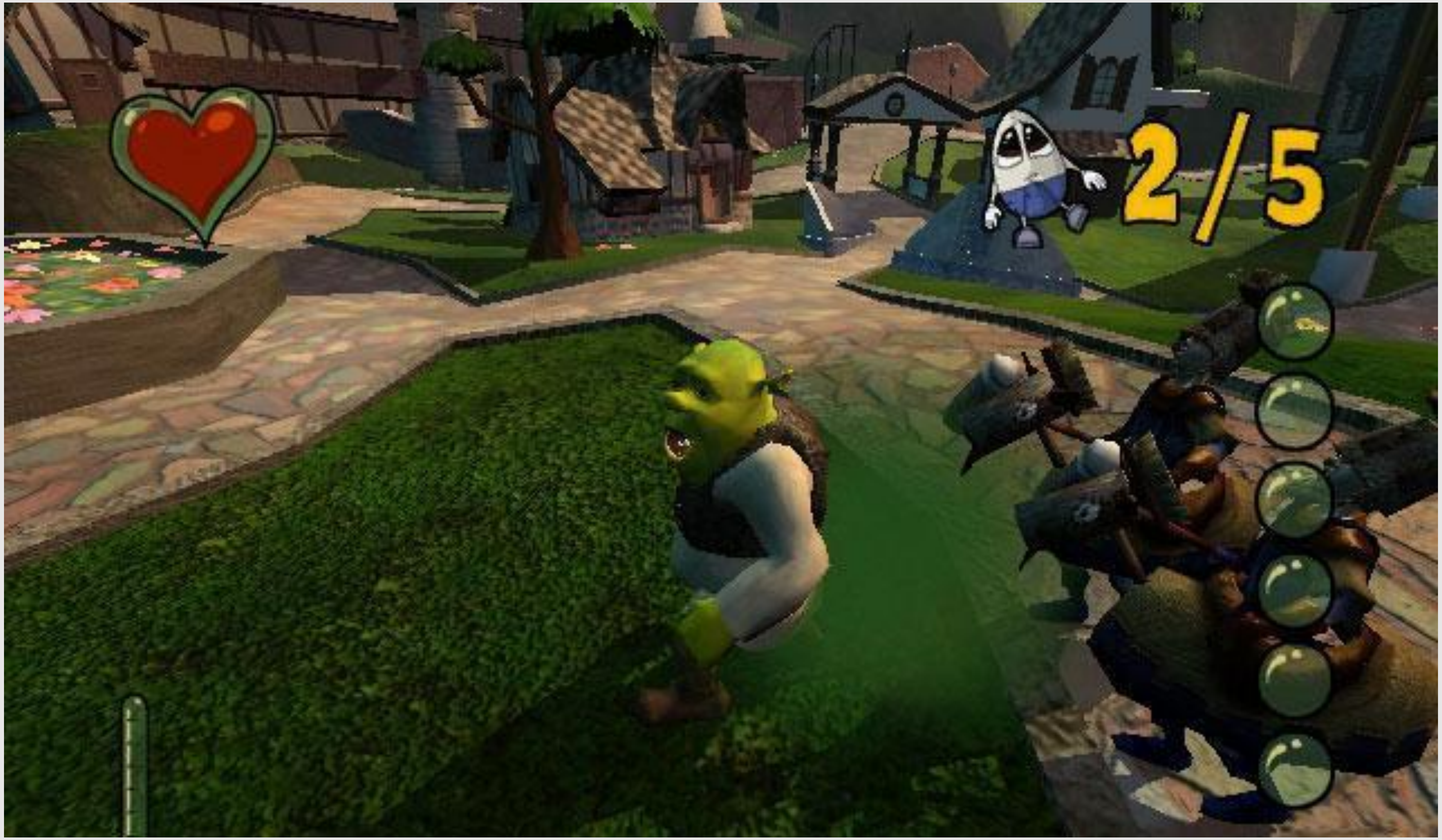
- ~~The Graphics Pipeline~~
- ~~Graphics APIs~~
- Deferred Shading
- Graphics Architecture

Deferred Shading

- Popular algorithm for rendering in modern games
- **Idea:** restructure the rendering pipeline to perform shading **after** all occlusions have been resolved
- Not a new idea. Implemented in several classic graphics systems, but not directly supported in most high-end GPUs
  - But modern graphics pipeline provides mechanism to allow applications to implement deferred shading efficiently

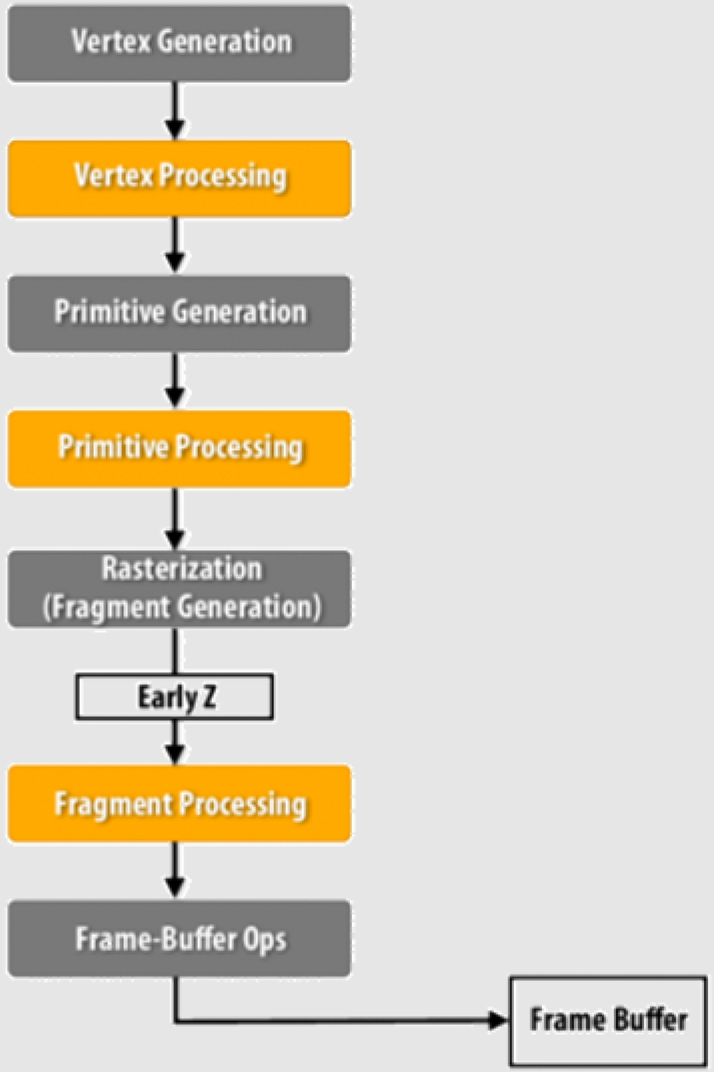
Assassin's Creed III (2012) Ubisoft

What was the first video game to use deferred shading?



Shrek (2001) Digital Illusions Canada

Feed Forward Rendering

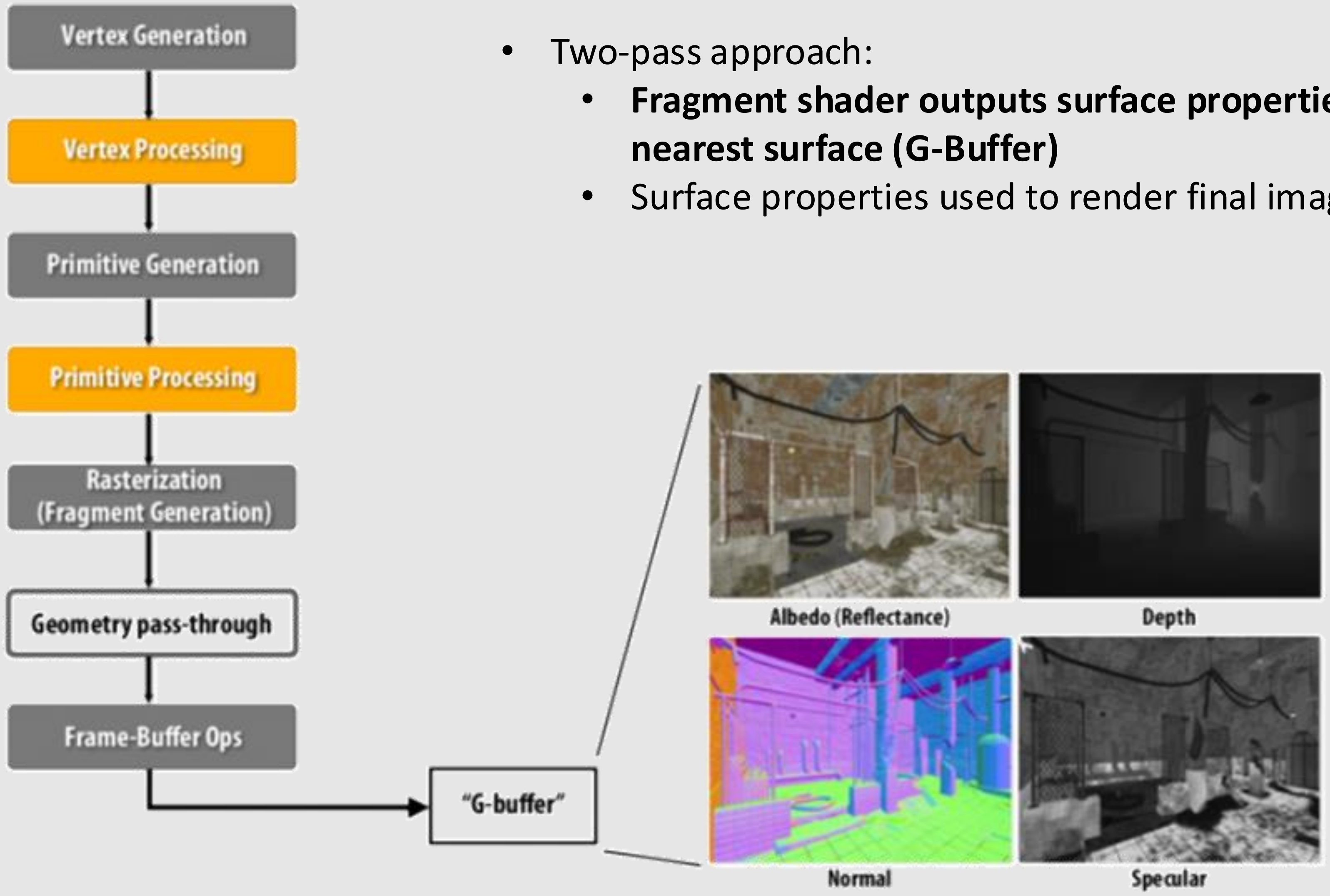


- Occlusion testing done after fragments have been generated

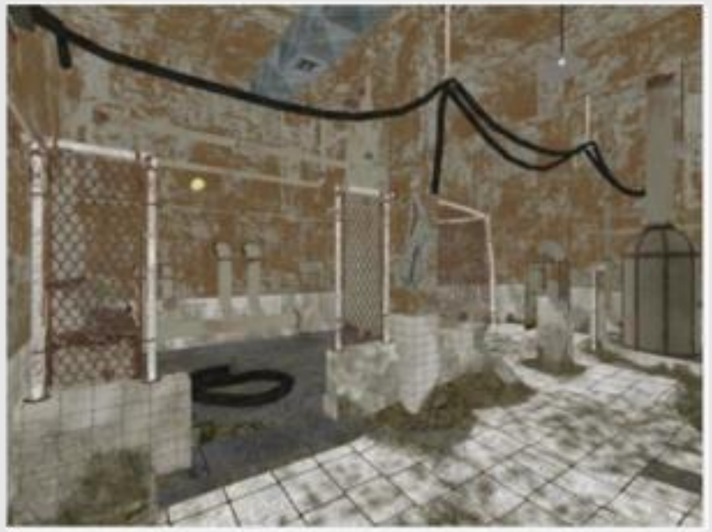
- **Best-case:** fragments provided near-to-far
- **Worst case:** fragments provided far-to-near
    - Even Early-Z cannot keep every fragment from being rasterized

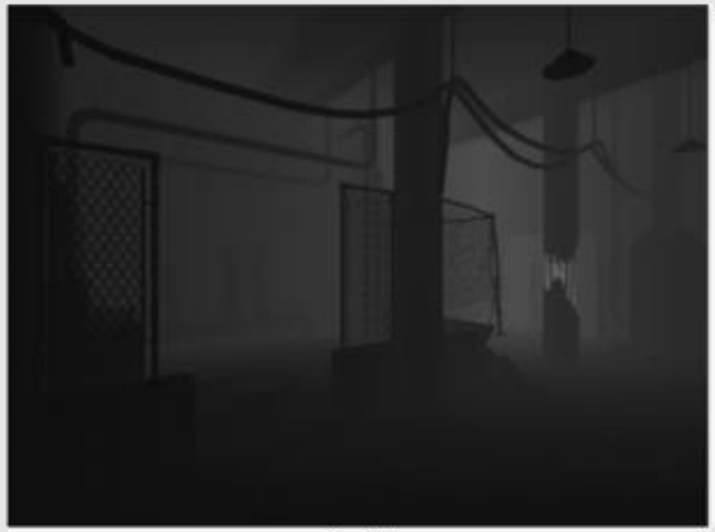
Deferred Shading

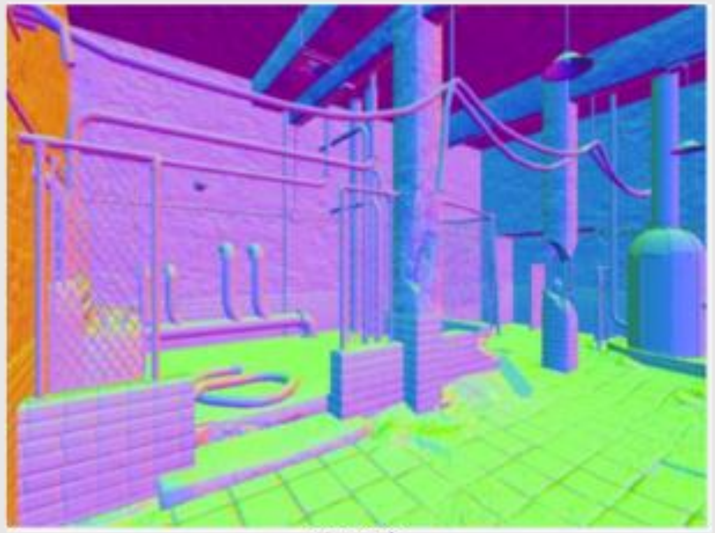
- Two-pass approach:
  - **Fragment shader outputs surface properties of nearest surface (G-Buffer)**
  - Surface properties used to render final image

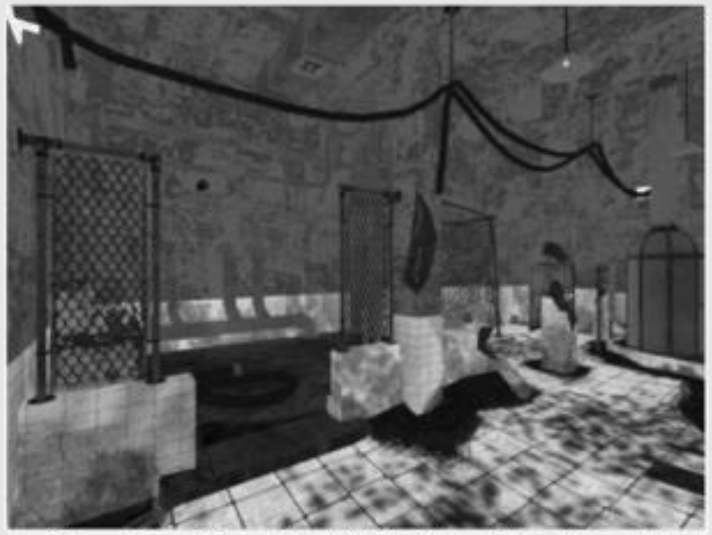


Geometry Buffer

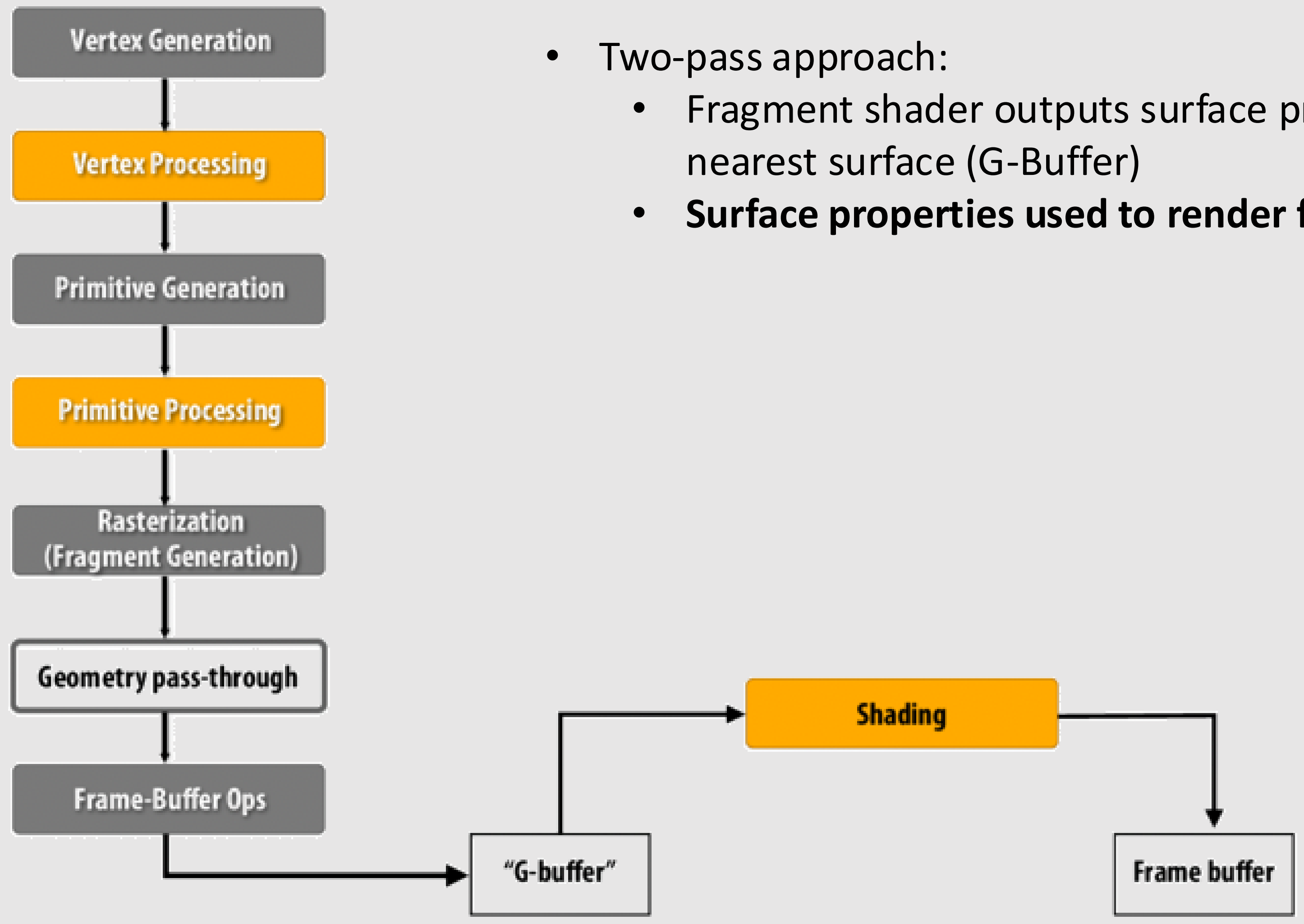

Albedo (Reflectance)


Depth


Normal


Specular

Deferred Shading

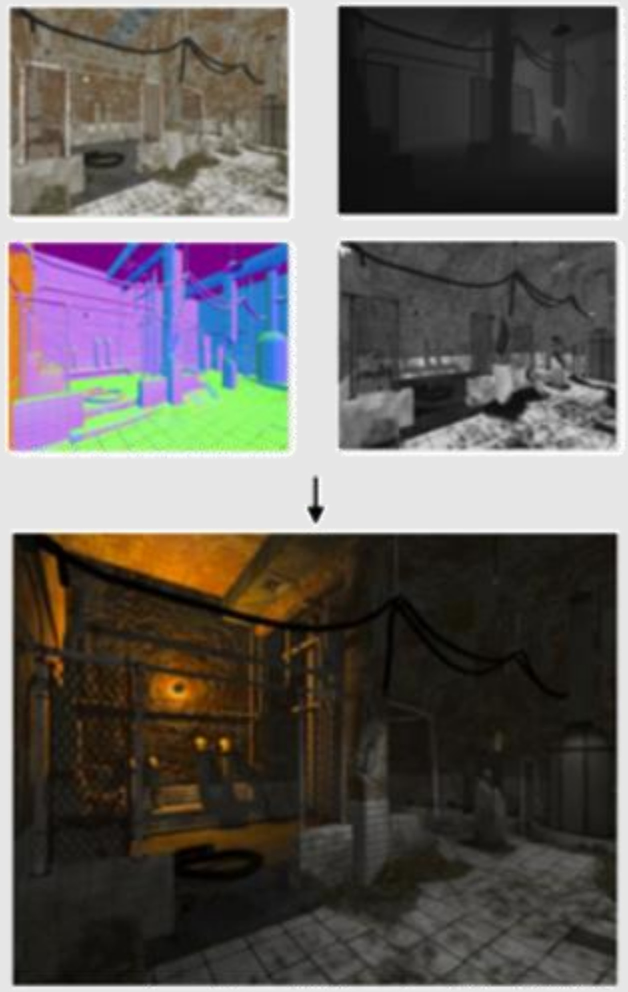


- Two-pass approach:
  - Fragment shader outputs surface properties of nearest surface (G-Buffer)
  - **Surface properties used to render final image**

Two-Pass Algorithm

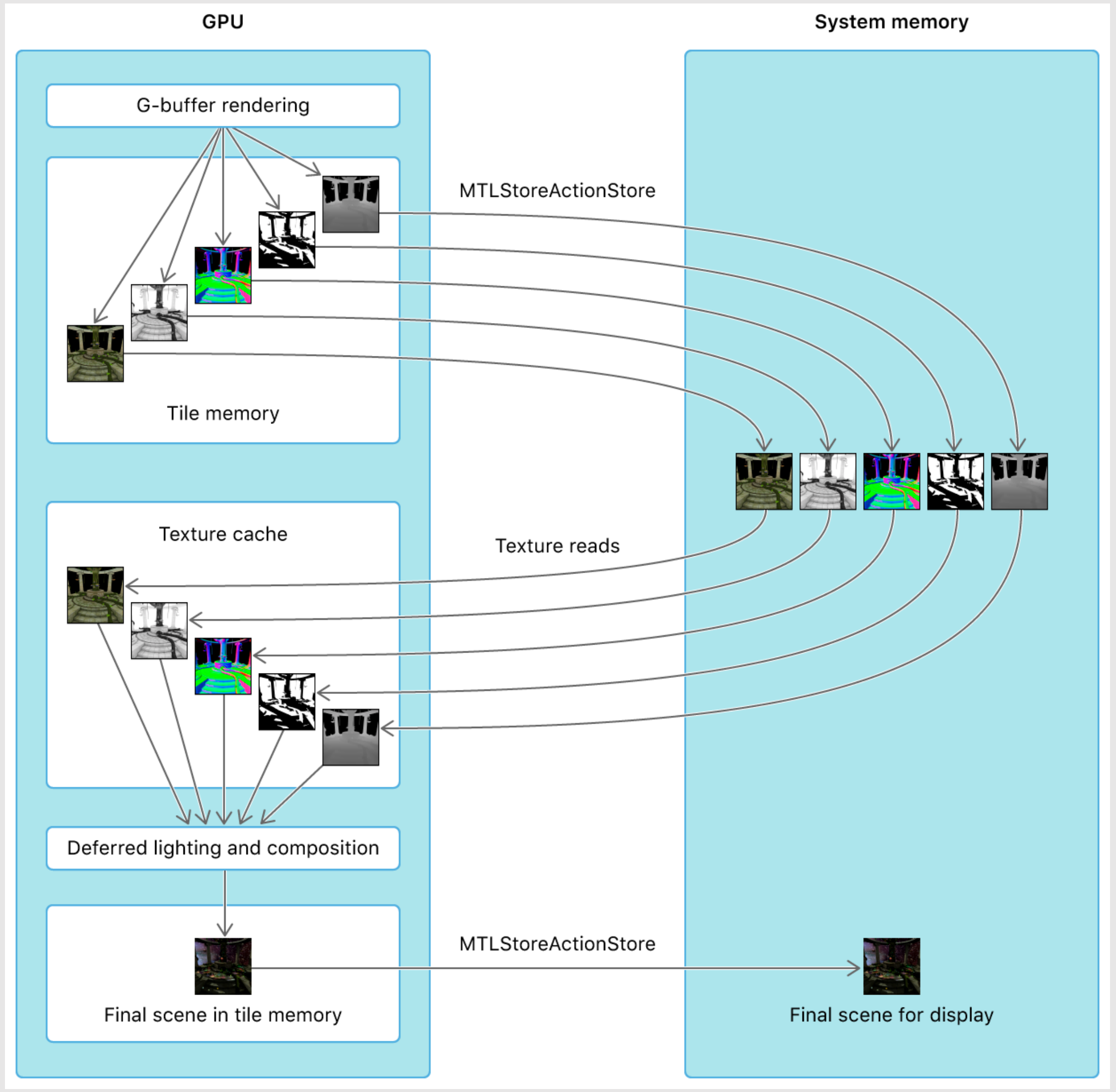
- **Pass 1: Geometry Pass**
  - Render scene geometry using traditional pipeline
  - Write visible geometry information to G-Buffer

- **Pass 2: Shading Pass**
  - For each G-Buffer sample, compute shading
  - Read G-Buffer data for current sample
  - Accumulate contribution of all lights
  - Output final surface color for sample



Leadwerks Engine

Deferred Rendering on Apple Chips



Metal Developer Guide (2020) Apple

- Threads work to compute G-Buffer together
  - Contents too large to remain in cache, spill to memory
- Go to shading once G-Buffer fully computed
  - Requires reads back into cache

- ~~The Graphics Pipeline~~
- ~~Graphics APIs~~
- ~~Deferred Shading~~
- Graphics Architecture

Graphics Processing Unit

- Developed to efficiently scale the 3D Graphics Pipeline
  - Every step was fixed function

- General Purpose Graphics Processing Unit (GPGPU)
  - Allowed for customizable nodes known as **shaders**
  - What shaders do we have?
      - **Vertex shader**
      - **Geometry shader**
      - **Fragment shader**
    - Other shaders
      - **Compute shader**
      - **Ray tracing shaders**
    - Are frame-buffer operations not considered shaders?
      - These are toggle-able stages of the pipeline
      - You don't write code for them

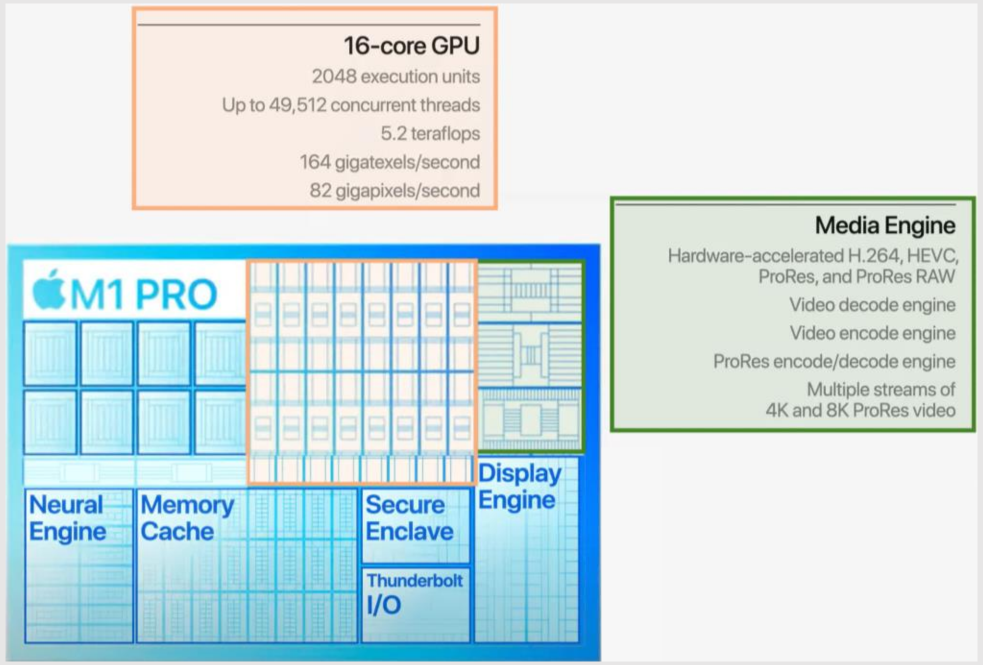
- Many applications outside graphics
  - Data scientists used GPUs to write their calculations to intermediate buffers and threw out the resulting render

Nvidia Geforce 256 (1999)

Unified GPUs

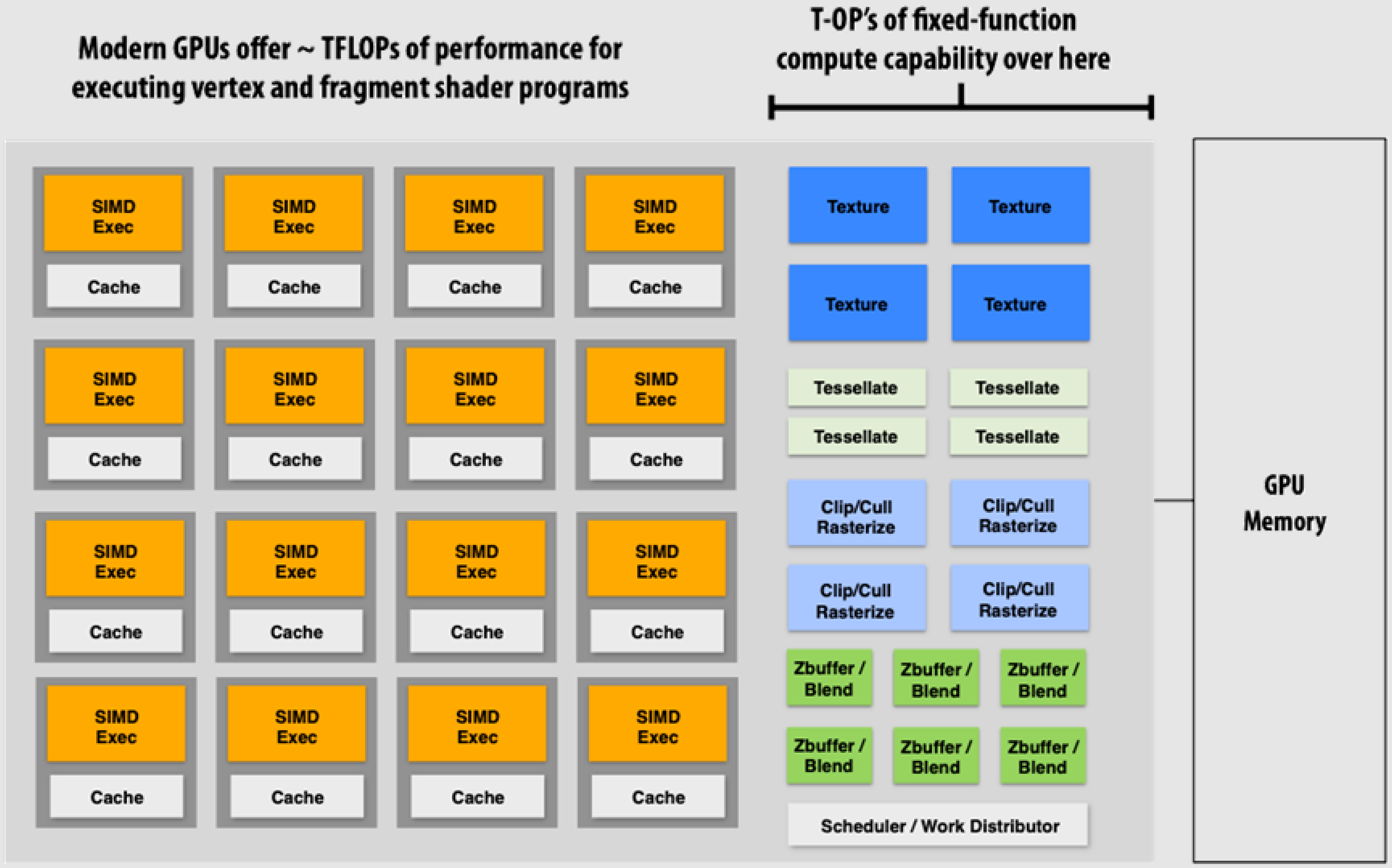
- Example: Apple M1 Pro
  - 8 high performance cores
  - 2 high efficiency (low power) cores
  - 16 core GPU
  - Fixed-function media engine for video encoding/decoding
    - Frees up CPU + GPU during rendering
    - Allows for more expensive video formats
  - Neural engine for more efficient machine learning tasks
    - Supports most deep learning layers
    - Push for CoreML on Mac applications

- **Design Philosophy:** run important workloads on the most efficient hardware for the job
  - Fixed-function provides efficiency and speed while freeing up general purpose resources high in demand

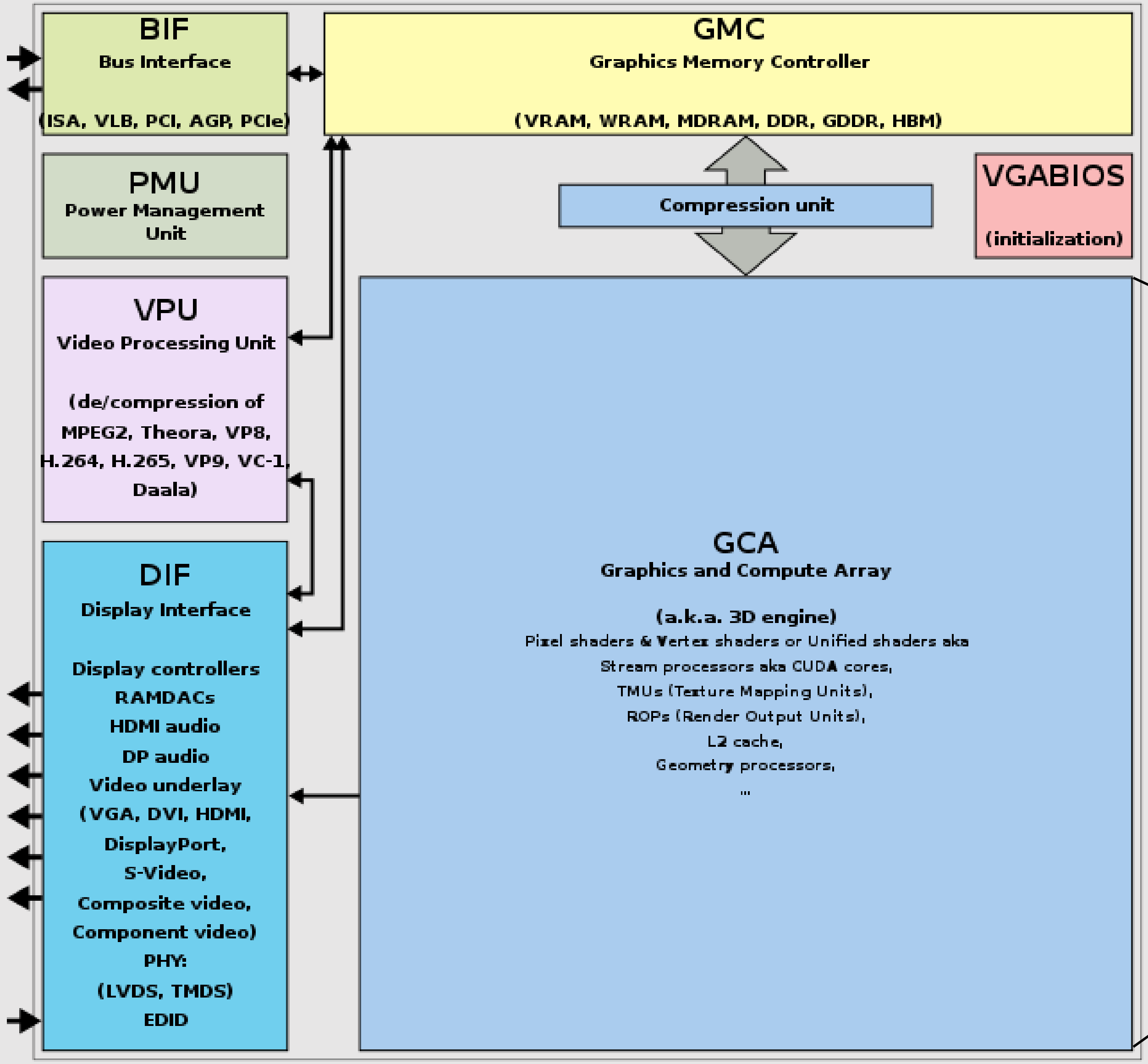


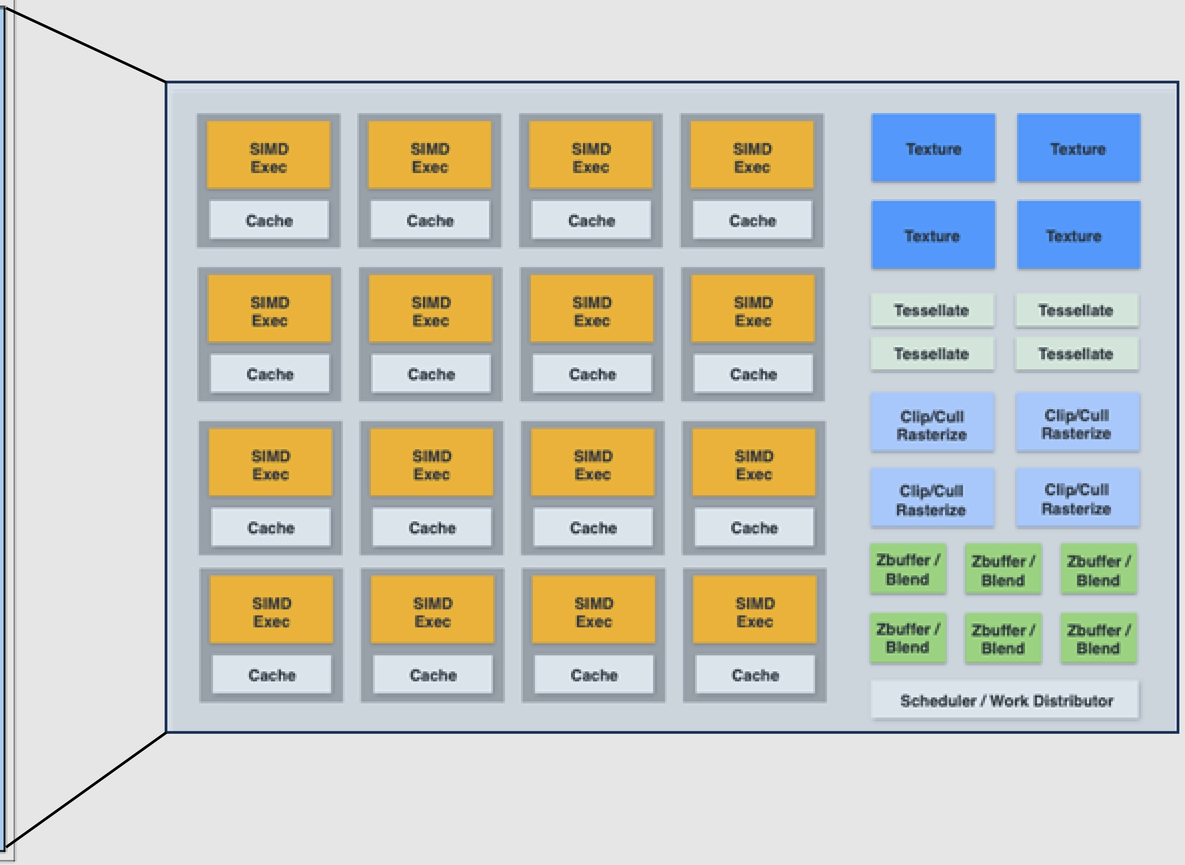
M1 Pro (2021) Apple

GPU Architecture


Modern GPUs offer ~ TFLOPs of performance for executing vertex and fragment shader programs
T-OP's of fixed-function compute capability over here
SIMD Exec
Cache
Texture
Tessellate
Clip/Cull Rasterize
Zbuffer / Blend
Scheduler / Work Distributor
GPU Memory

GPU Architecture


BIF
Bus Interface
(ISA, VLB, PCI, AGP, PCIe)
GMC
Graphics Memory Controller
(VRAM, WRAM, MDRAM, DDR, GDDR, HBM)
PMU
Power Management Unit
Compression unit
VGABIOS
(initialization)
VPU
Video Processing Unit
(de/compression of MPEG2, Theora, VP8, H.264, H.265, VP9, VC-1, Daala)
GCA
Graphics and Compute Array
(a.k.a. 3D engine)
Pixel shaders & Vertex shaders or Unified shaders aka
Stream processors aka CUDA cores,
TMUs (Texture Mapping Units),
ROPs (Render Output Units),
L2 cache,
Geometry processors,
...
DIF
Display Interface
Display controllers
RAMDACs
HDMI audio
DP audio
Video underlay
(VGA, DVI, HDMI, DisplayPort, S-Video, Composite video, Component video)
PHY:
(LVDS, TMDS)
EDID



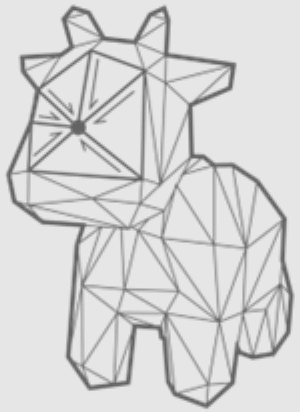
GPU Production

“In 2009, **Intel**, **Nvidia**, and **AMD/ATI** were the market share leaders, with **49.4%, 27.8%, and 20.6% market share respectively**. However, those numbers include Intel's integrated graphics solutions as GPUs.

Not counting those, **Nvidia and AMD control nearly 100%** of the market as of 2018. Their respective **market shares are 66% and 33%**”

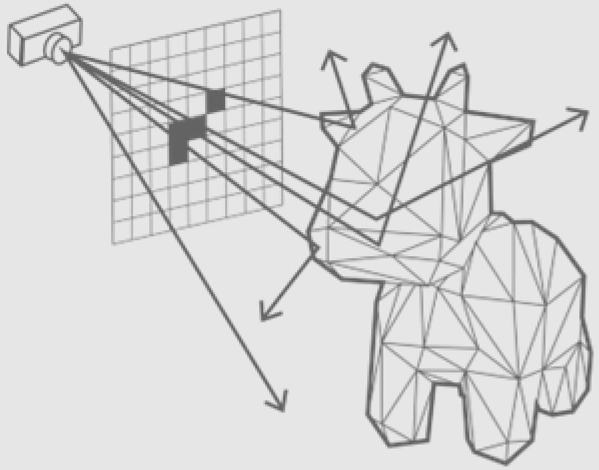
Course Roadmap

[A1: Rasterization]

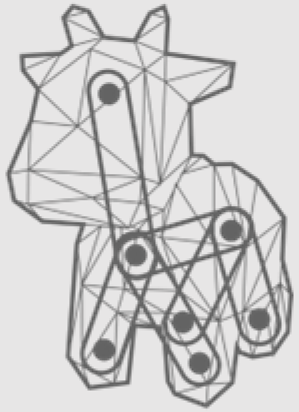


[A2: MeshEdit]

Next Time



[A3: PathTracer]



[A4: Animation]