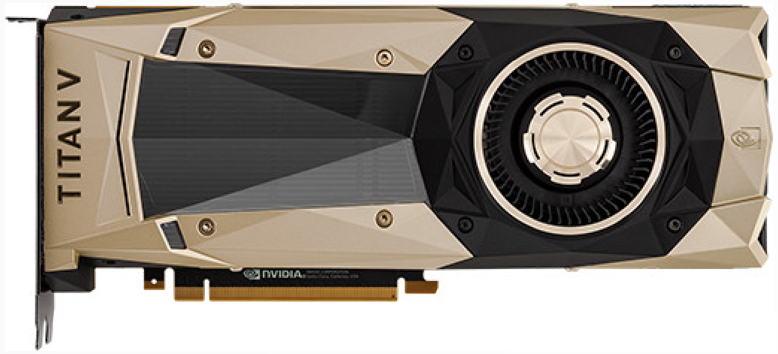
GPU Programming

Programming Models for Emerging Platforms



Titan V (2018)
5120 Cores
$3000

Titan V (2018)
2304 Cores
$600

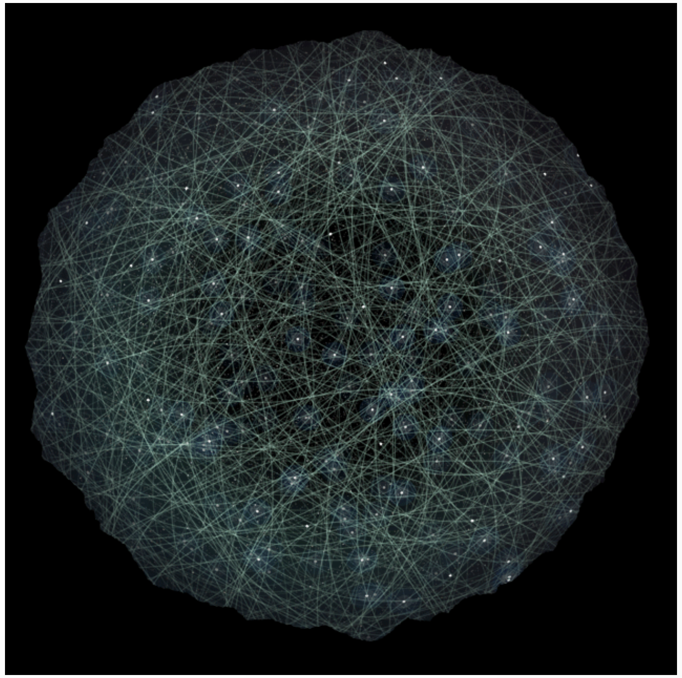

THE CELEBRATION OF A LIFETIME
Disney · PIXAR
COCO
NOVEMBER 22
IN 3D AND REAL D 3D


THE CELEBRATION OF A LIFETIME
Disney · PIXAR
COCO
NOVEMBER 22
IN 3D AND REAL D 3D


THE LEGEND OF
ZELDA
BREATH OF THE WILD


THE CELEBRATION OF A LIFETIME
Disney · PIXAR
COCO
NOVEMBER 22
IN 3D AND REAL D 3D


THE LEGEND OF
ZELDA
BREATH OF THE WILD



GPU Prograaming

- Good
  - Games and animations are fun (and lucrative) industries
  - Lots and lots of parallel programming cores for relatively cheap price

GPU Prograaming

- Good
  - Games and animations are fun (and lucrative) industries
  - Lots and lots of parallel programming cores for relatively cheap price
- Bad
  - GPUs are notoriously difficult to program, mainly due to the very distinctive and restrictive memory access patterns

GPU Prograaming

- Good
  - Games and animations are fun (and lucrative) industries
  - Lots and lots of parallel programming cores for relatively cheap price
- Bad
  - GPUs are notoriously difficult to program, mainly due to the very distinctive and restrictive memory access patterns
- Ugly
  - GPUs hardware and systems software change radically at a fast pace

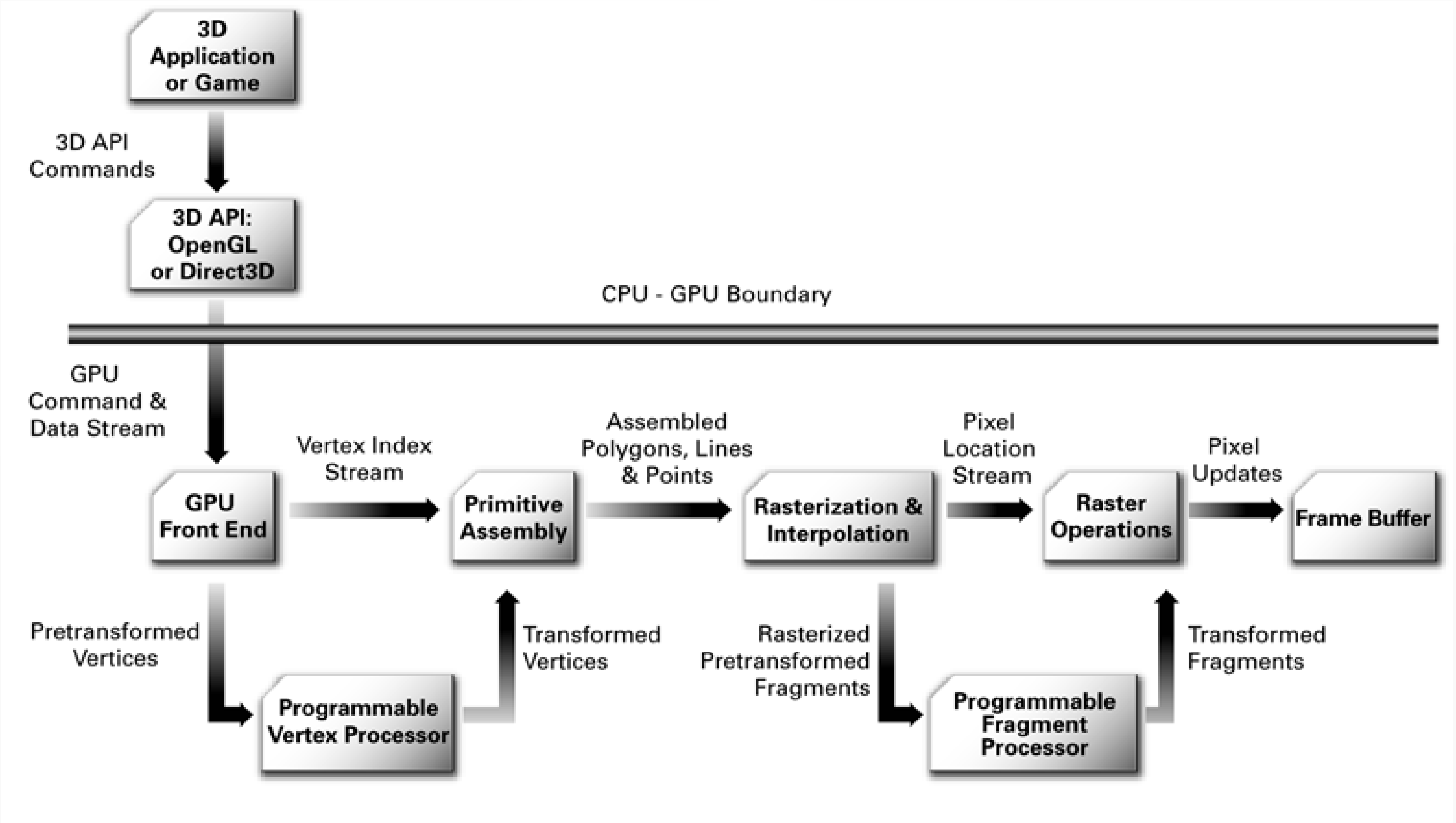
CPU Memory Access

- At any program point
  - Allocate/free local or global memory
  - Random memory access
    - Registers
      - Read/write
    - Local memory
      - Read/write to stack
    - Global memory
      - Read/write to heap
    - Disk
      - Read/write to disk

GPU Memory Access

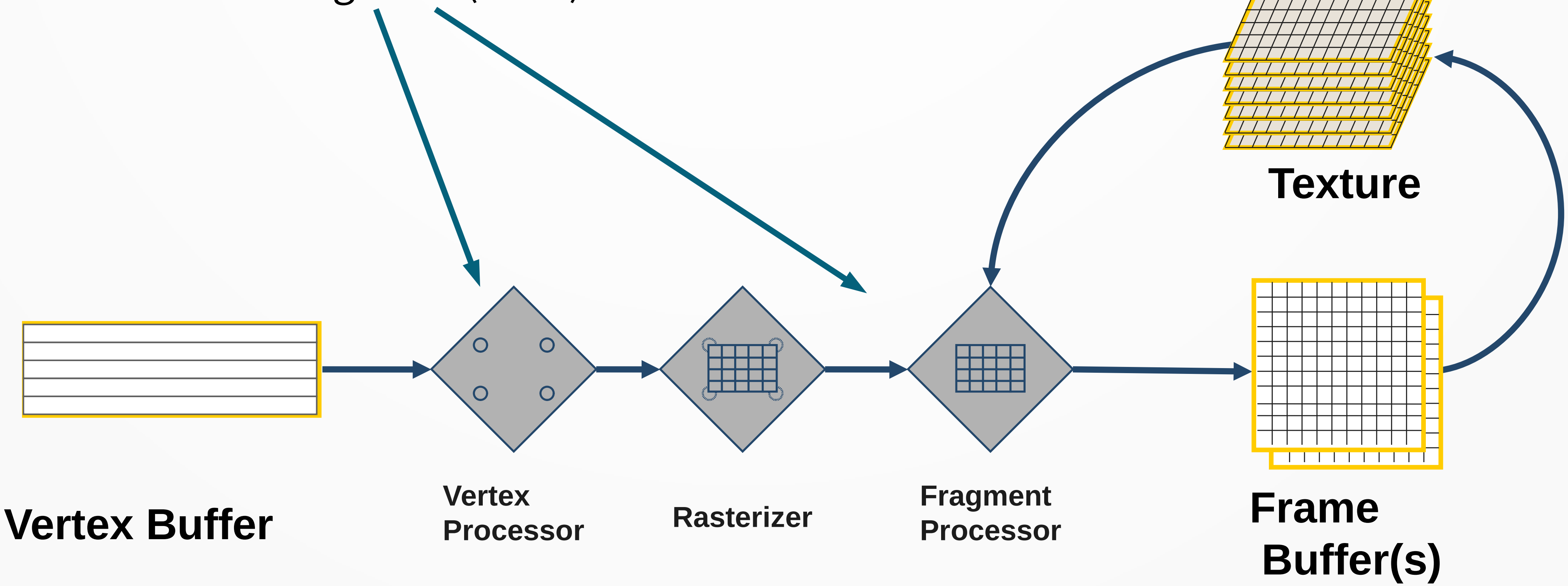
- Programs deployed on GPU cannot access CPU memory
- Restricted memory access *within* GPU memory
  - Allocate/free GPU memory only at the beginning and end of the computation unit
  - During computation
    - Access GPU registers
    - No stack-style memory on GPU
    - Restrictions on “global memory”
    - No disk access

Graphics Pipeline



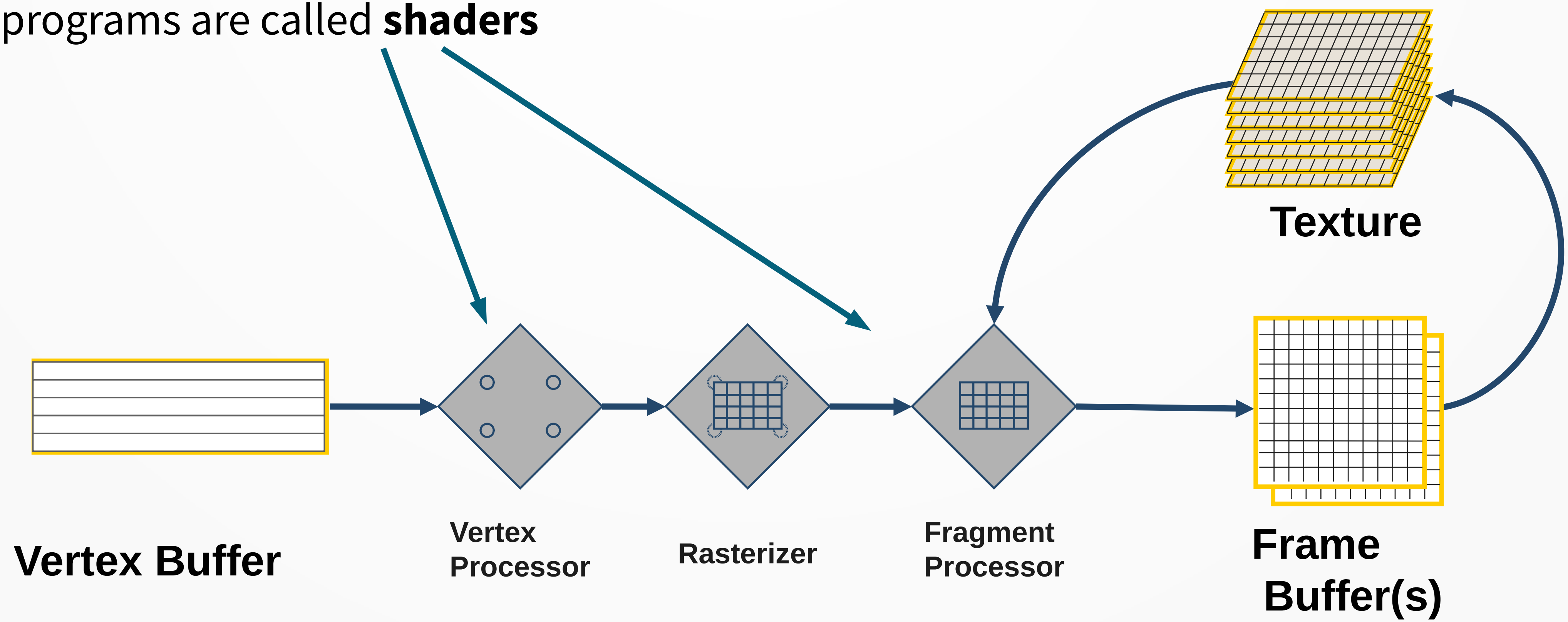
Graphics Pipeline

Key programmable processing units:
Vertex Processor and Fragment (Pixel) Processor



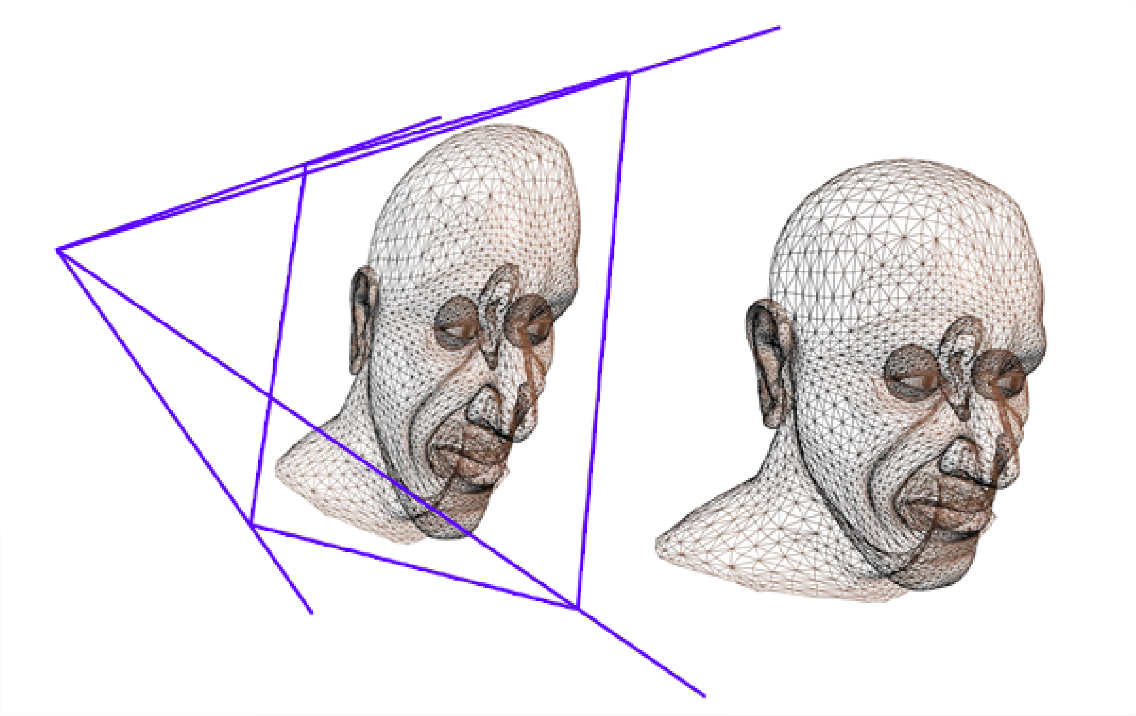
Graphics Pipeline

We can write programs that run on these processors.
These programs are called **shaders**



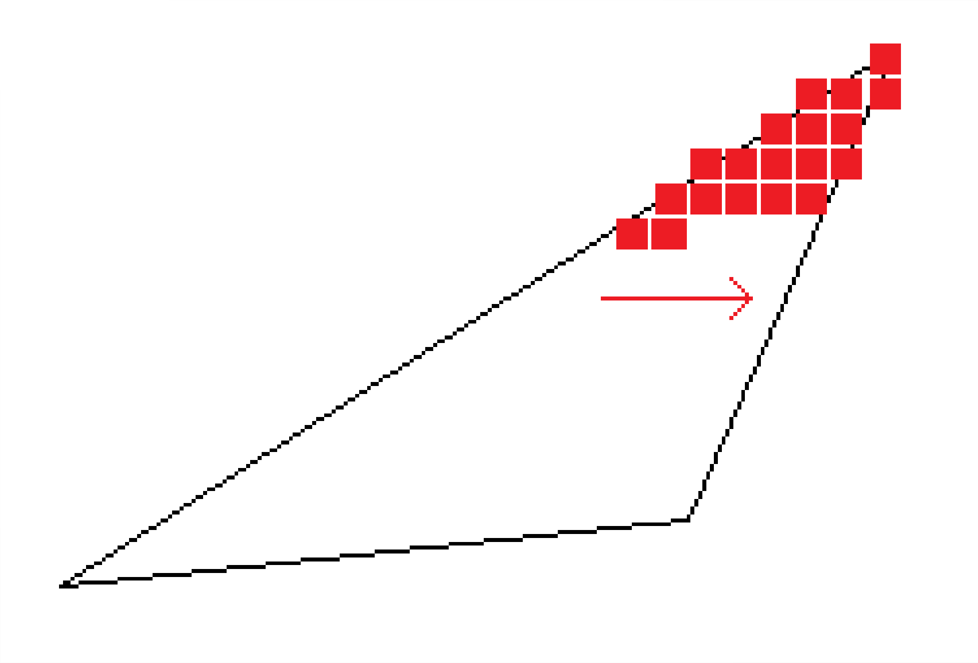
Vertex Shaders

- 3D object represented as vertices
- Vertex shader transforms vertices to vertices (performs a transformation for every vertex being rendered)
- Mapping from 3D space to 2D space

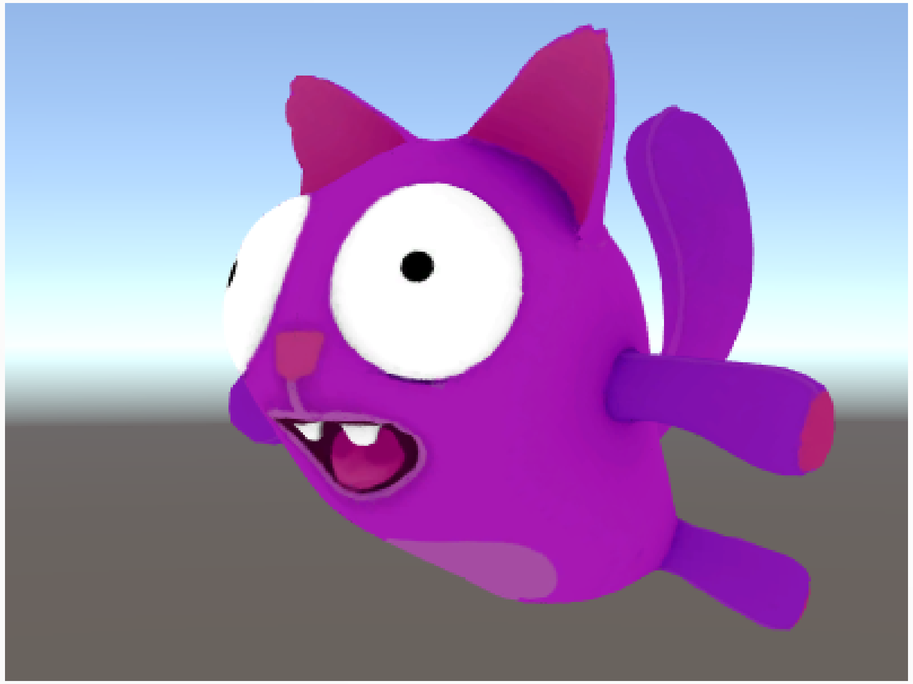


Pixel Shaders

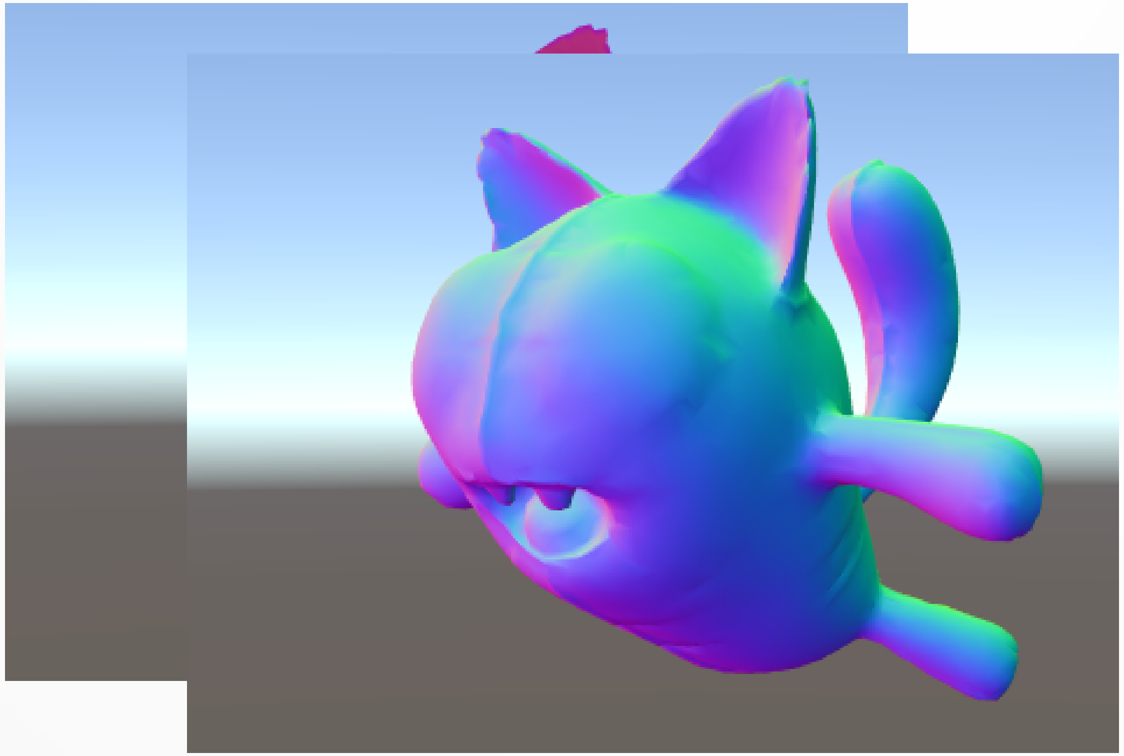
- Pixel shader transforms EVERY pixel in the pipeline
- Apply transformation to pixel properties (color, depth, etc)
- Output is directed to the frame buffer, ready for display



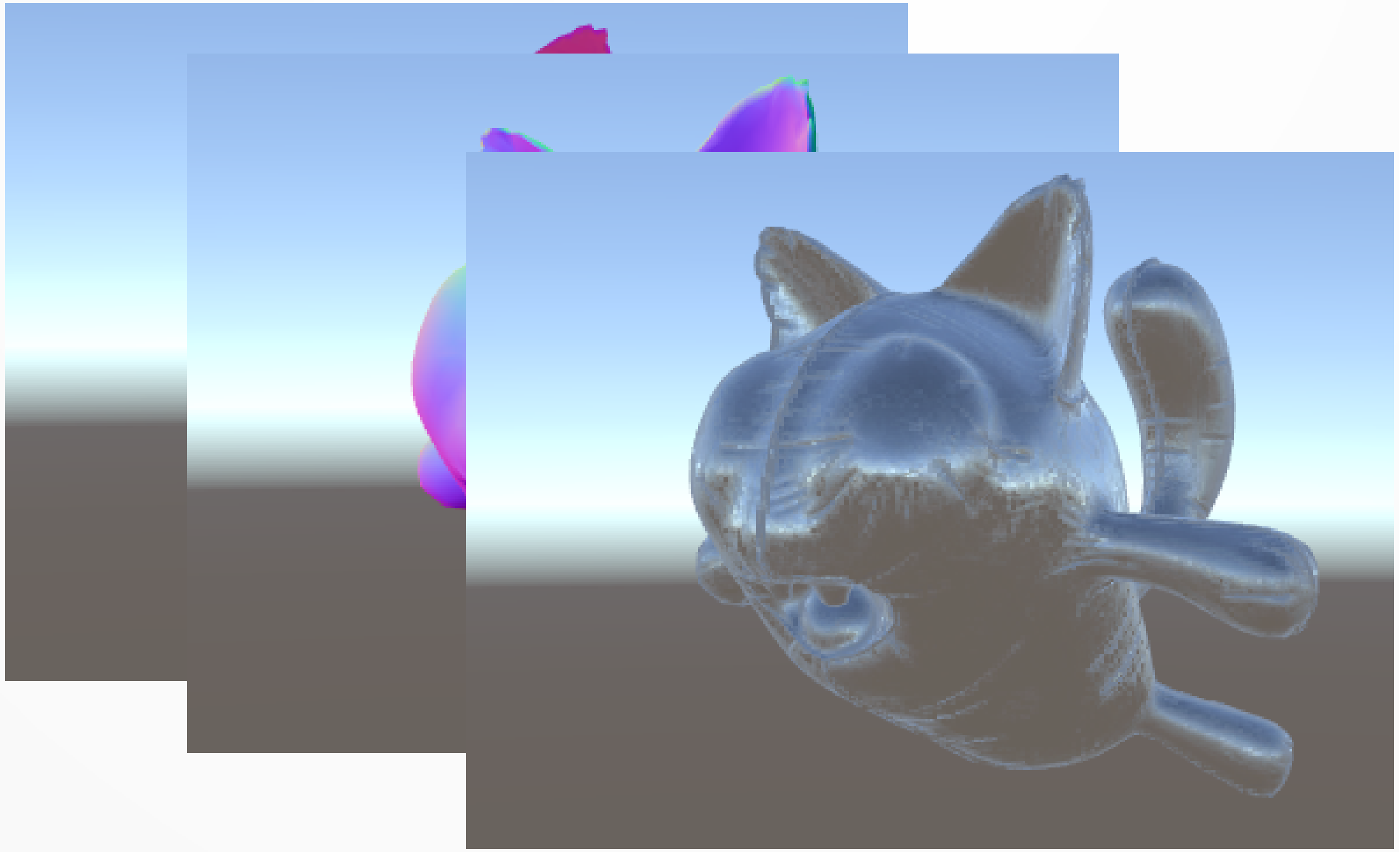
Shaders Combined



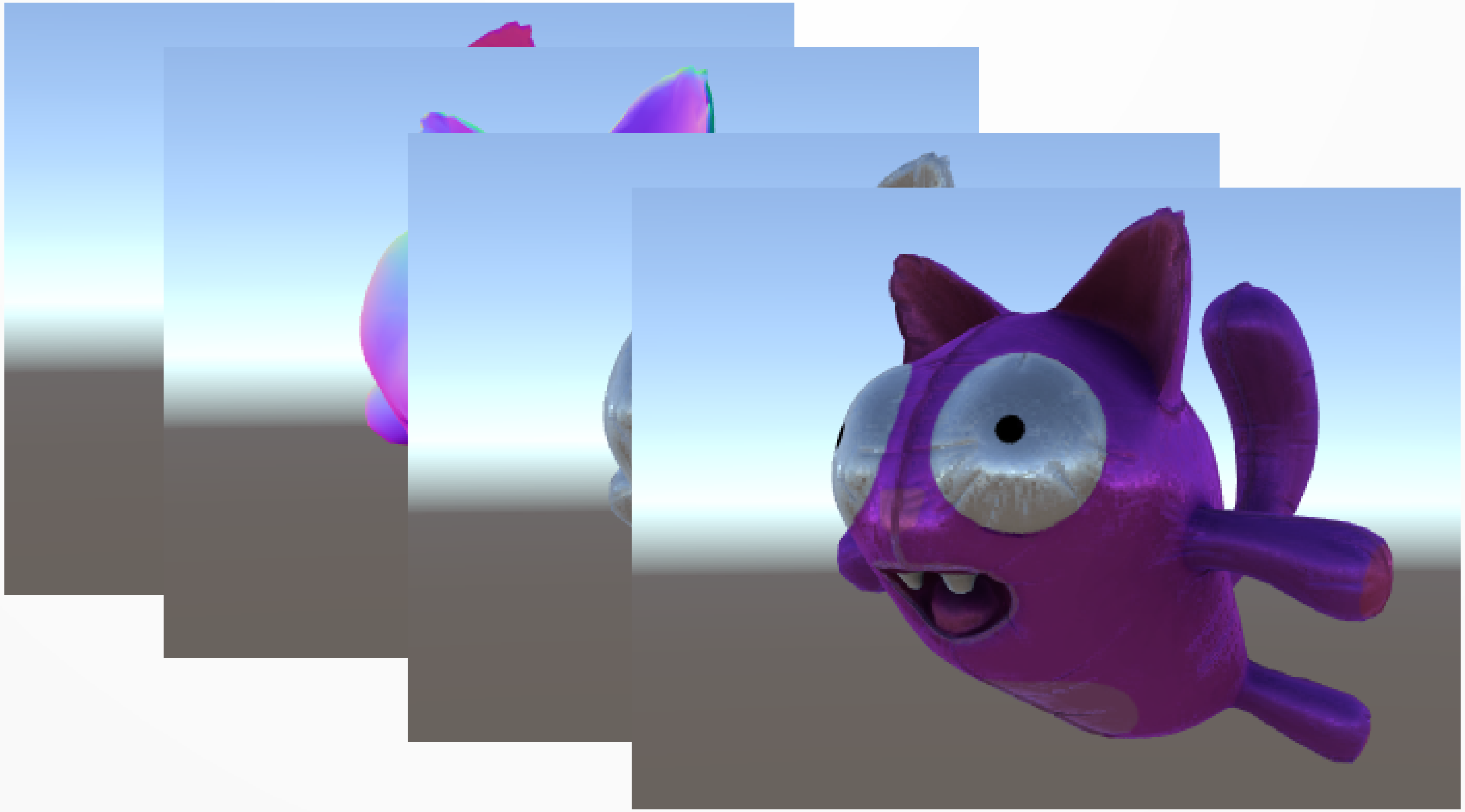
Shaders Combined



Shaders Combined

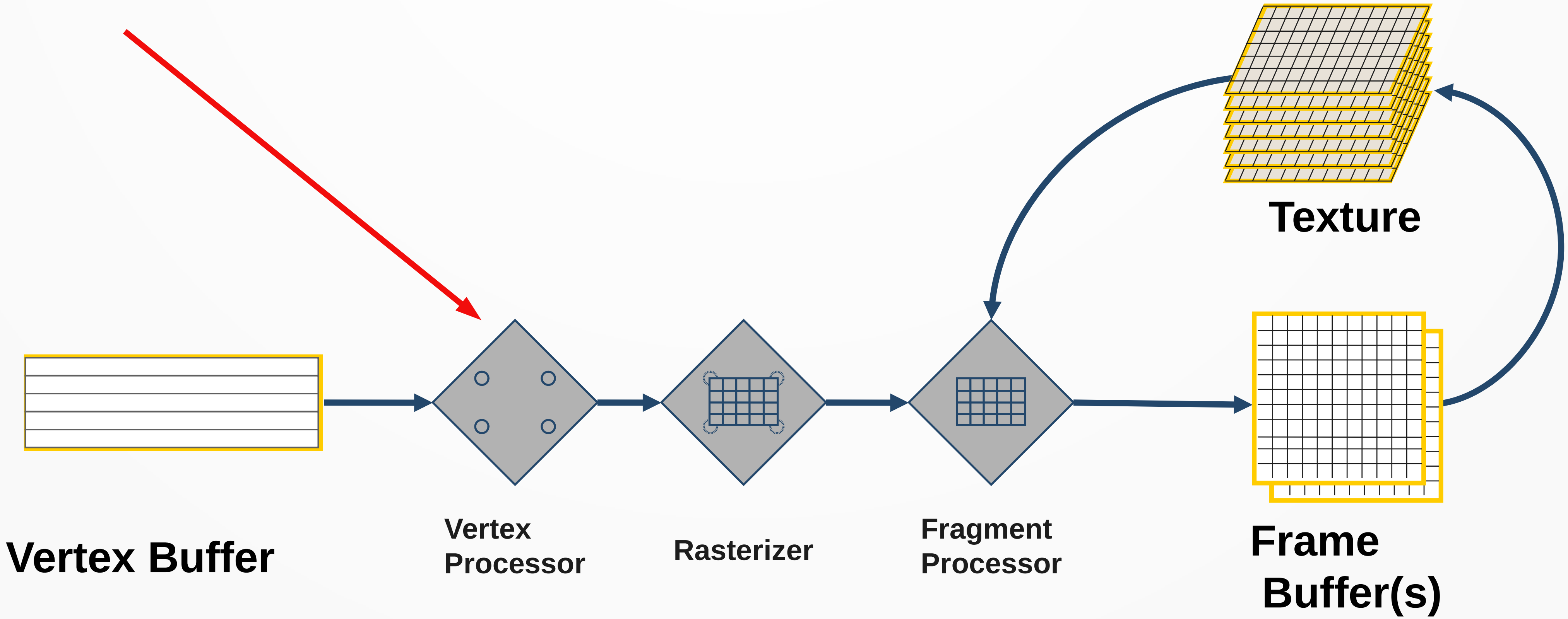


Shaders Combined



Graphics Pipeline

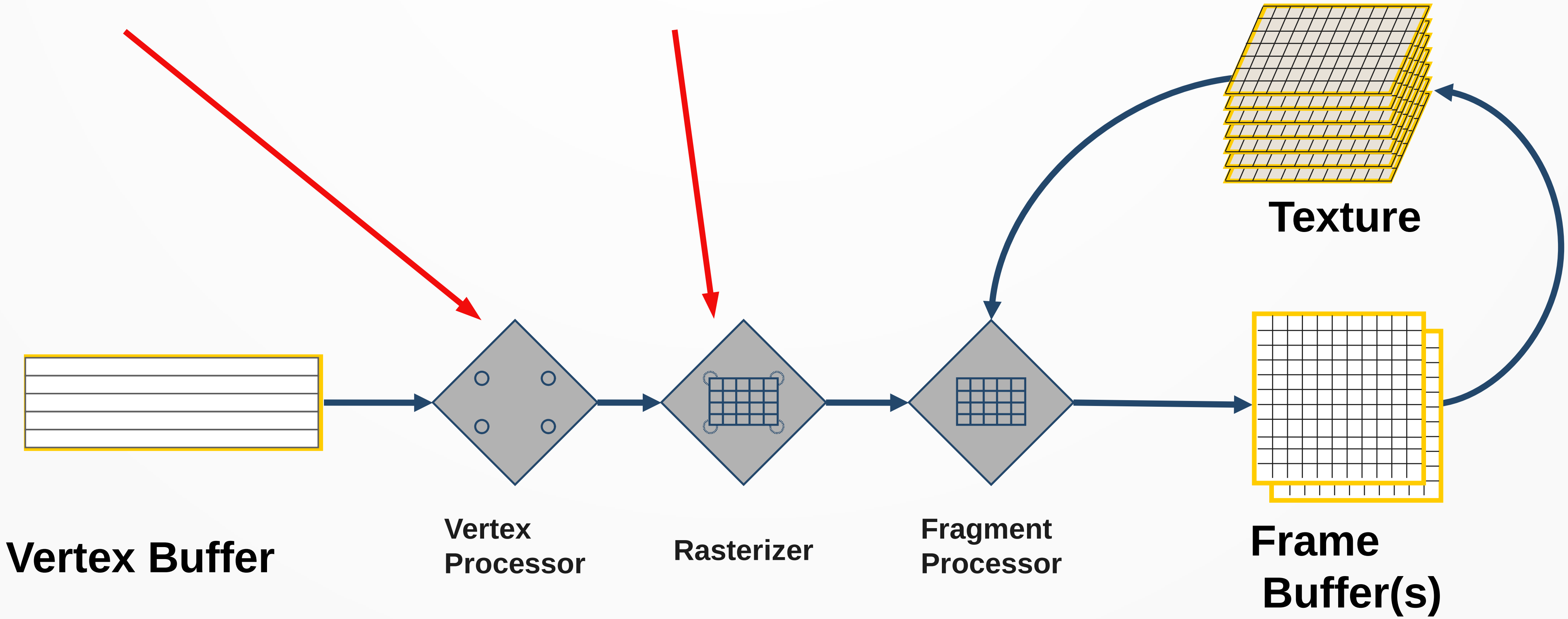
Input: Verticies,
Output: Vertices
Process Single Vertex



Graphics Pipeline

Input: Verticies,
Output: Vertices
Process Single Vertex

Input: Verticies
Output: Pixels
Not programmable

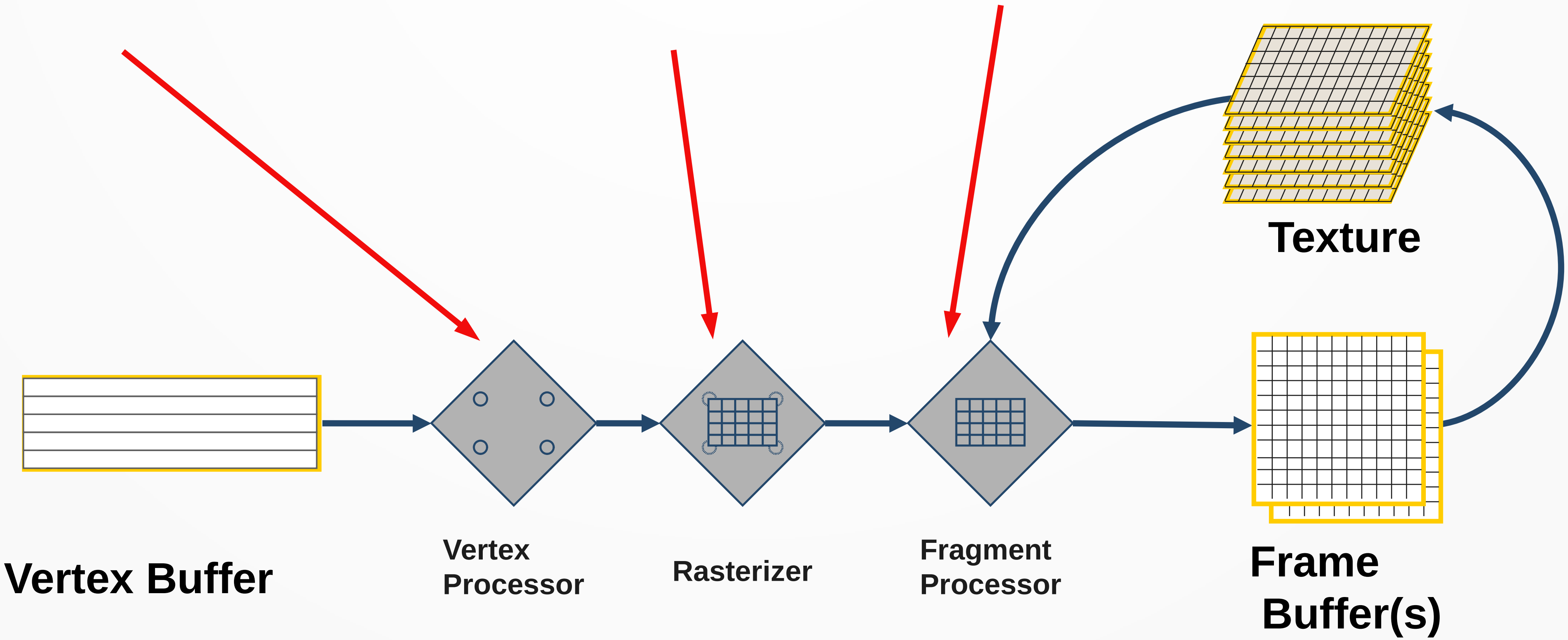


Graphics Pipeline

Input: Verticies,
Output: Vertices
Process Single Vertex

Input: Verticies
Output: Pixels
Not programmable

Input: Pixels
Output: Pixels
Random access to Texture



Writing Shader Programs

- Shaders originally were written in assembly-like language
- We know this is error prone:
  - Assembly is tedious at best
  - Graphics cards have a memory model different from CPUs
  - Makes for a very foreign programming environment

Writing Shader Programs

- For CPU programming, C gives us assembly-like control and abstracts some of the more tedious parts of assembly
- For GPU programming, **Cg** serves a similar purpose
  - Very crude abstractions layer, but there are a lot of new concepts for graphics programming
  - Cg lets us inspect some of them

A Vertex Shader

```
struct Output {
  float4 p : SV_POSITION;
  float4 color    : COLOR;
};

Output tis_green(float4 position : POSITION)
{
  Output o;
  o.p = UnityObjectToClipPos(position);
  o.color = float4(0,1,0,0);
  return o;
}
```

Program run on stream of vertices
(function invoked per vertex)

A Vertex Shader

```
struct Output {
  float4 p : SV_POSITION;
  float4 color     : COLOR;
};

Output tis_green(float4 position : POSITION)
{
  Output o;
  o.p = UnityObjectToClipPos(position);
  o.color = float4(0,1,0,0);
  return o;
}
```

Program run on stream of vertices (function invoked per vertex)

Transformed data structure

Input Vertex

Output Vertex

A Pixel Shader

```
float4 My_Fragment_Output tis_green_alt():
SV_Target
{
  return float4(0,1,0,0);
}
```

Program run on stream of pixels
(function invoked per pixel)

A Pixel Shader

```
float4 My_Fragment_Output tis_green_alt():
SV_Target
{
  return float4(0,1,0,0);
}
```

Program run on stream of pixels
(function invoked per pixel)

Outputs green
for all pixels

Sometimes functionality can be
done using vertex or pixel shader

Uniform Parameters

```
float4 tis_texture(float2 position : POSITION,
   uniform sampler2D image) {
…
}
```

Uniform parameter is the same value ACROSS all function invocations

1. View it as global memory for the shader

2. Set externally by the CPU

3. Some implementations allow mutation of this global memory

Semantic Bindings

```
struct Output {
  float4 p : SV_POSITION;
  float4 color     : COLOR;
};

Output tis_green(float4 position : POSITION)
{
  Output o;
  o.p = UnityObjectToClipPos(position);
  o.color = float4(0,1,0,0);
  return o;
}
```

How do we move data along hardware boundaries?

CPU => Vertex Processor

Vertex Processor => Rasterizer

Rasterizer => Pixel Processor

Semantic Bindings

```
struct Output {
  float4 p : SV_POSITION;
  float4 color     : COLOR;
};

Output tis_green(float4 position : POSITION)
{
  Output o;
  o.p = UnityObjectToClipPos(position);
  o.color = float4(0,1,0,0);
  return o;
}
```

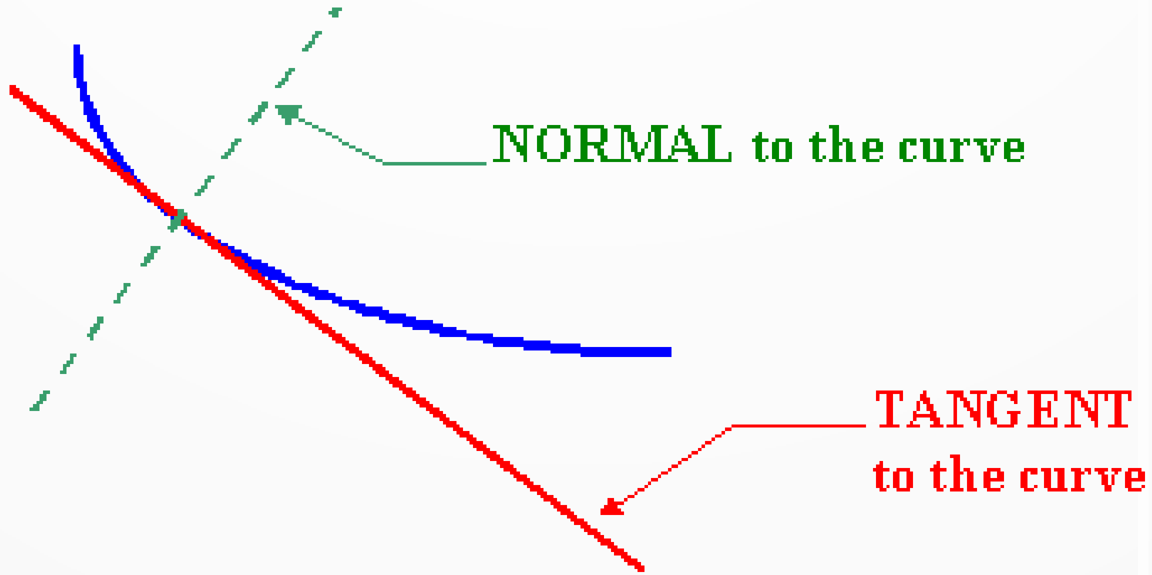
Semantic Bindings provide semantics on what data actually is

POSITION indicates this is a vertex

COLOR indicates this is a color

Semantic Bindings

- POSITION (in the type of float3 or float4)
- COLOR (in the type of float4)
- NORMAL (in type float3) and TANGENT (in type float4)



Semantic Bindings

- TEXCOORD0-TEXCOORD7 are used as indices/cooordinates in the texture array (in the types of float2/float3/float4).
- Texture memory allows random access
  - Used as additional data passed from CPU to GPU (as the input of the vertex shader
  - “temporary” shared storage passing from vertex shader to pixel shader
- The TEXCOORD outputs from vertex still subject to rasterization, so only output something that makes sense.

Semantic Bindings

- For convenience, Unity groups several semantic bindings together with built-in type support:
  - appdata_base: position, normal and one texture coordinate.
  - appdata_tan: position, tangent, normal and one texture coordinate.
  - appdata_full: position, tangent, normal, four texture coordinates and color.
- If you declare a parameter “x” to have “appdata_base” type, then “x.vertex” gives back POSITION data, “x.normal” gives back NORMAL data, “x.textcoord” gives back TEXTCOORD0 data. (Check Unity API for variations.

Building Up An Example

The Vertex Shader

```
struct Output {
    float4 pos : SV_POSITION;
    float2 leftTexCoord  : TEXCOORD0;
    float2 rightTexCoord : TEXCOORD1;
}

Output v_twoTextures(
                   float2 position : POSITION,
                   float2 texCoord : TEXCOORD0,
               uniform float2 leftSeparation,
                  uniform float2 rightSeparation)

{
  Output o;
  o.pos = UnityObjectToClipPos(position);
  o.leftTexCoord  = texCoord + leftSeparation;
  o.rightTexCoord = texCoord + rightSeparation;
  return o;
}
```

The Pixel Shader

```
float4 f_twoTextures(float2 leftTexCoord  : TEXCOORD0,
                     float2 rightTexCoord : TEXCOORD1,
                     uniform sampler2D decal): SV_Target

{
  float4 leftColor  = tex2D(decal, leftTexCoord);
  float4 rightColor = tex2D(decal, rightTexCoord);
  return lerp(leftColor, rightColor, 0.5);

}
```

A Word on Cg

- You may use Cg if you work with the Unity game engine
  - Hence, it's not a useless language. If you work in game dev, very likely you would use Unity
  - https://docs.unity3d.com/Manual/SL-VertexFragmentShaderExamples.html
- For general purpose gpu programming (GPGPU), Cg is too crude (and meant for graphics)
- We will explore GPGPU next lecture (CUDA)

Acknowledgments

- Yu David Liu, CS476/CS576 Slides
- Images
  - aandtech.com
  - imdb.com
  - wikipedia.org
  - http://www.engineering.ucsb.edu/~mbudisic/cs240/hw0.htm
  - https://docs.unity3d.com/Manual/SL-VertexFragmentShaderExamples.html