The Graphics Pipeline

Ray Tracing: Why Slow?

Basic ray tracing: 1 ray/pixel

Ray Tracing: Why Slow?

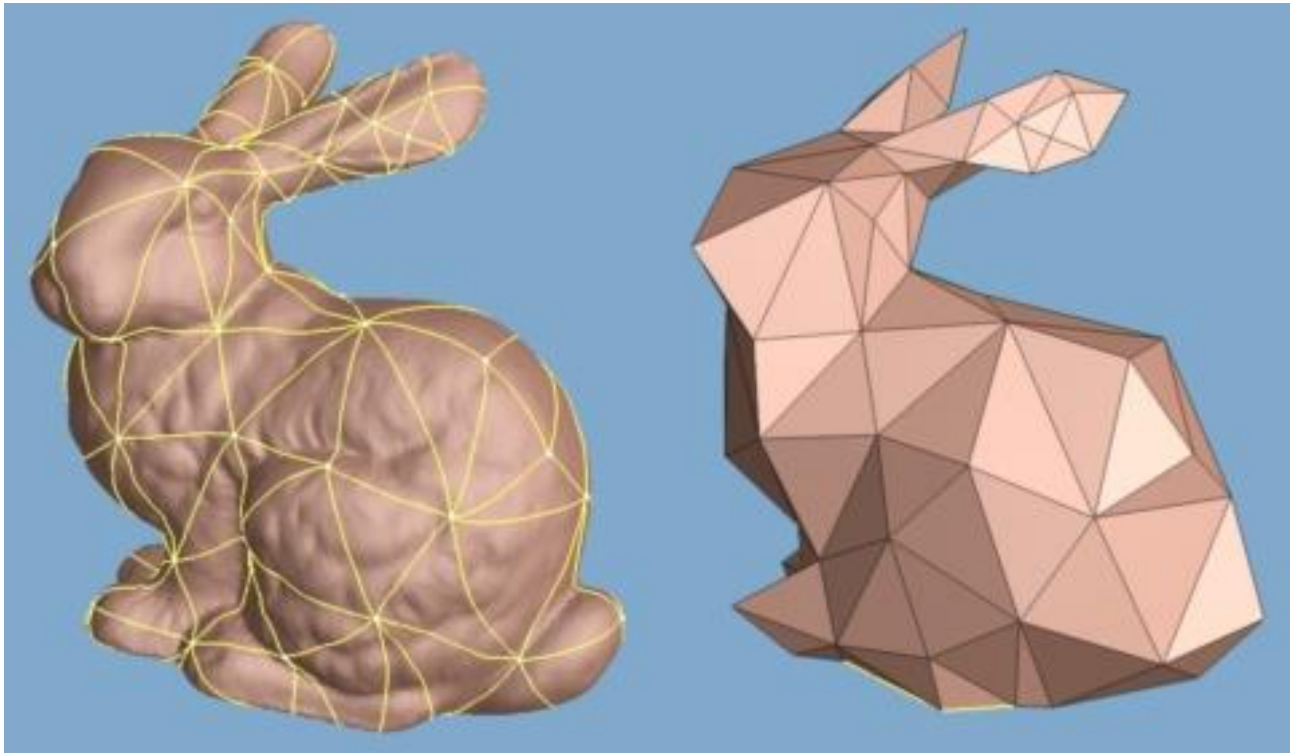
Basic ray tracing: 1 ray/pixel

But you **really** want shadows, reflections, global illumination, antialiasing…

- 100-1000 rays/pixel

Rendering: Rasterization

Tessellate objects into primitives

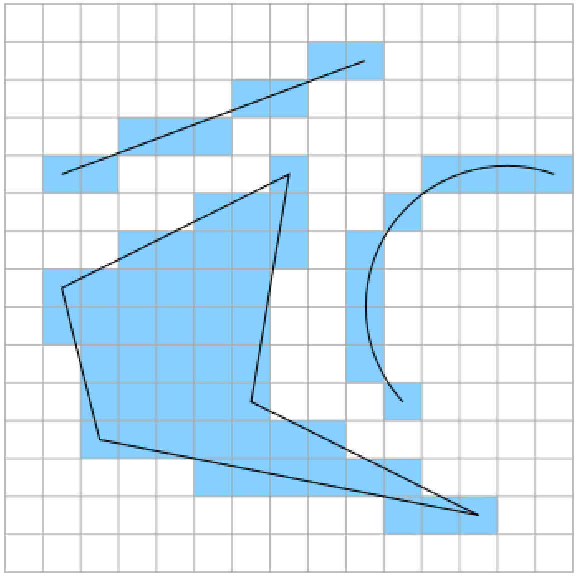


Rendering: Rasterization

Tessellate objects into primitives

Draw each separately:

- determine position and color
- draw pixels to screen

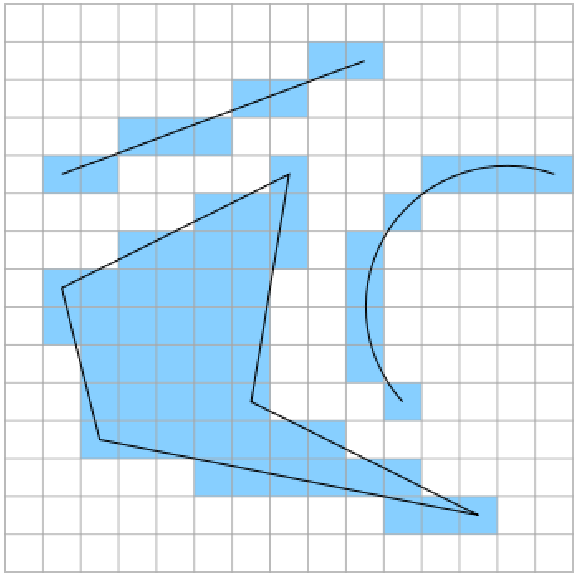


Rendering: Rasterization

Tessellate objects into primitives

Draw each separately:

- determine position and color
- draw pixels to screen

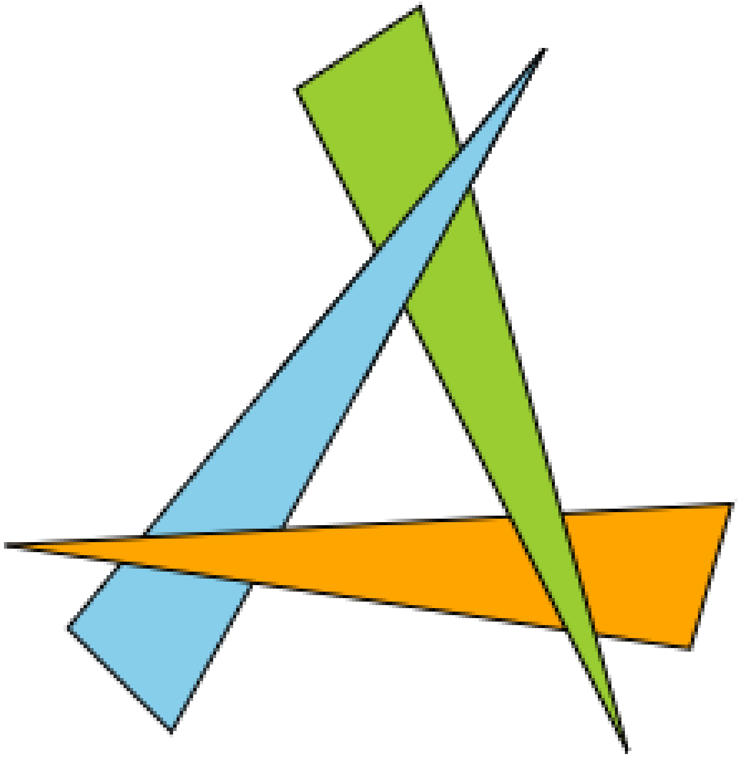


Embarrassingly parallel

Fast

Rendering: Rasterization

How to deal with overlaps?

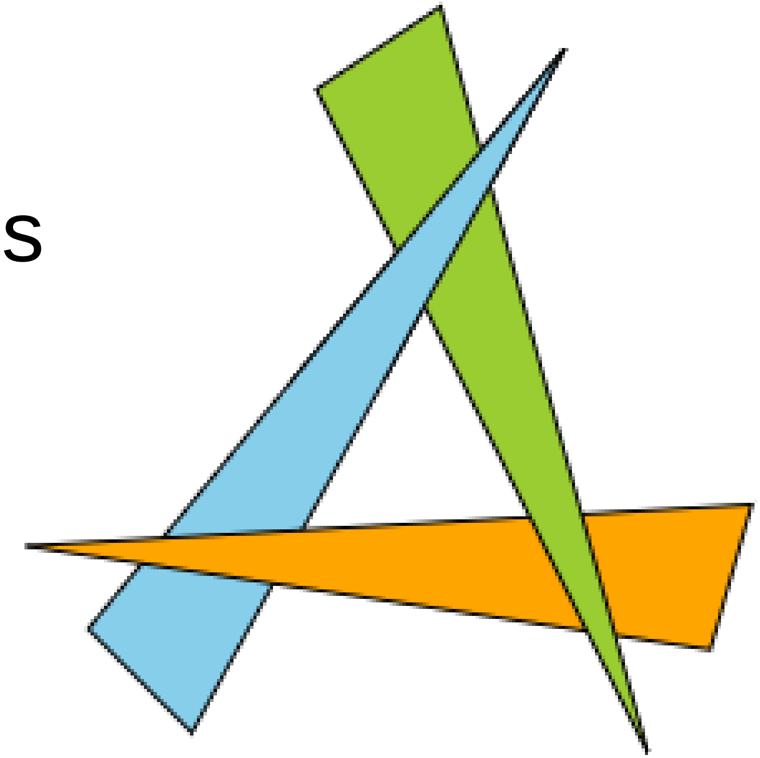


Rendering: Rasterization

How to deal with overlaps?

Keep track of **depth** of previously-drawn pixels

Depth image or depth buffer



Rendering: Rasterization

How to deal with overlaps?

- depth buffer

How to deal with shadows/reflections?

Rendering: Rasterization

How to deal with overlaps?

- depth buffer

How to deal with shadows/reflections?

- hmm…

Ray Tracing vs Rasterization

Ray Tracing	**Rasterization**
Loop over **pixels**	Loop over **triangles**
Light effects “easy” shadows, reflections, caustics, …	Light effects require hacks and tricks
Slow-ish	Blazingly fast
Used in movies	Used in games

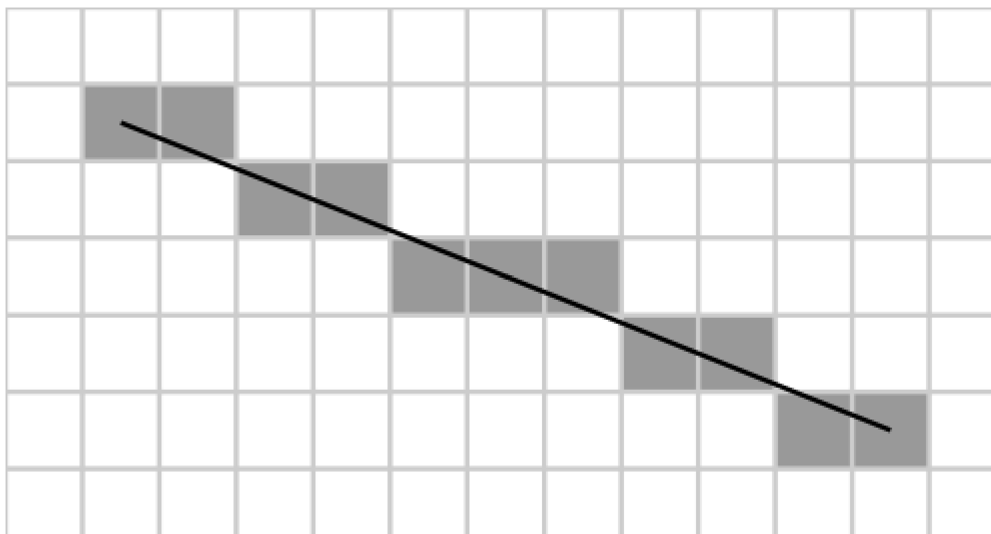
Ray Tracing vs Rasterization

Rasterized 3D graphics imagery

Ray traced 3D graphics imagery ©Siliconarts

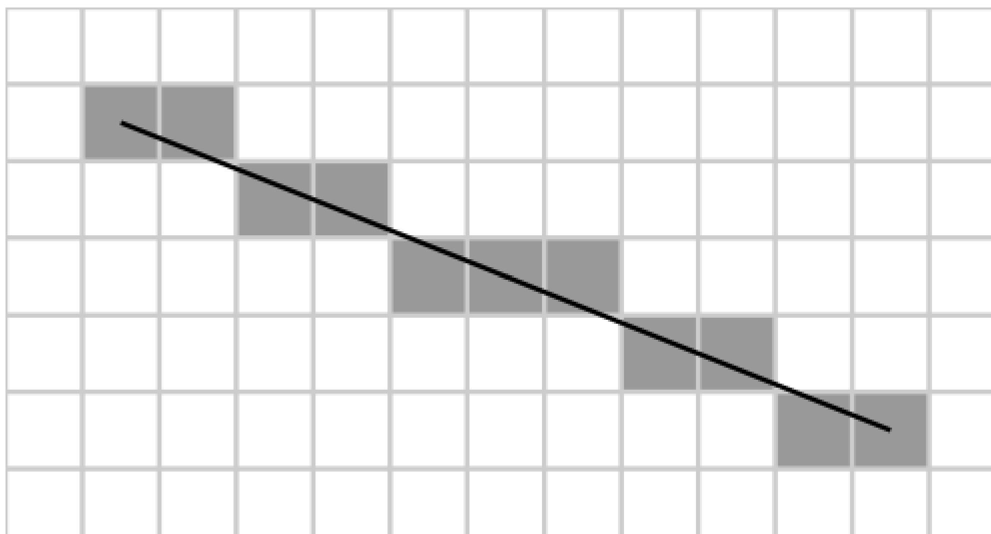
Rasterization Algorithms

Actually rasterizing objects not so easy…

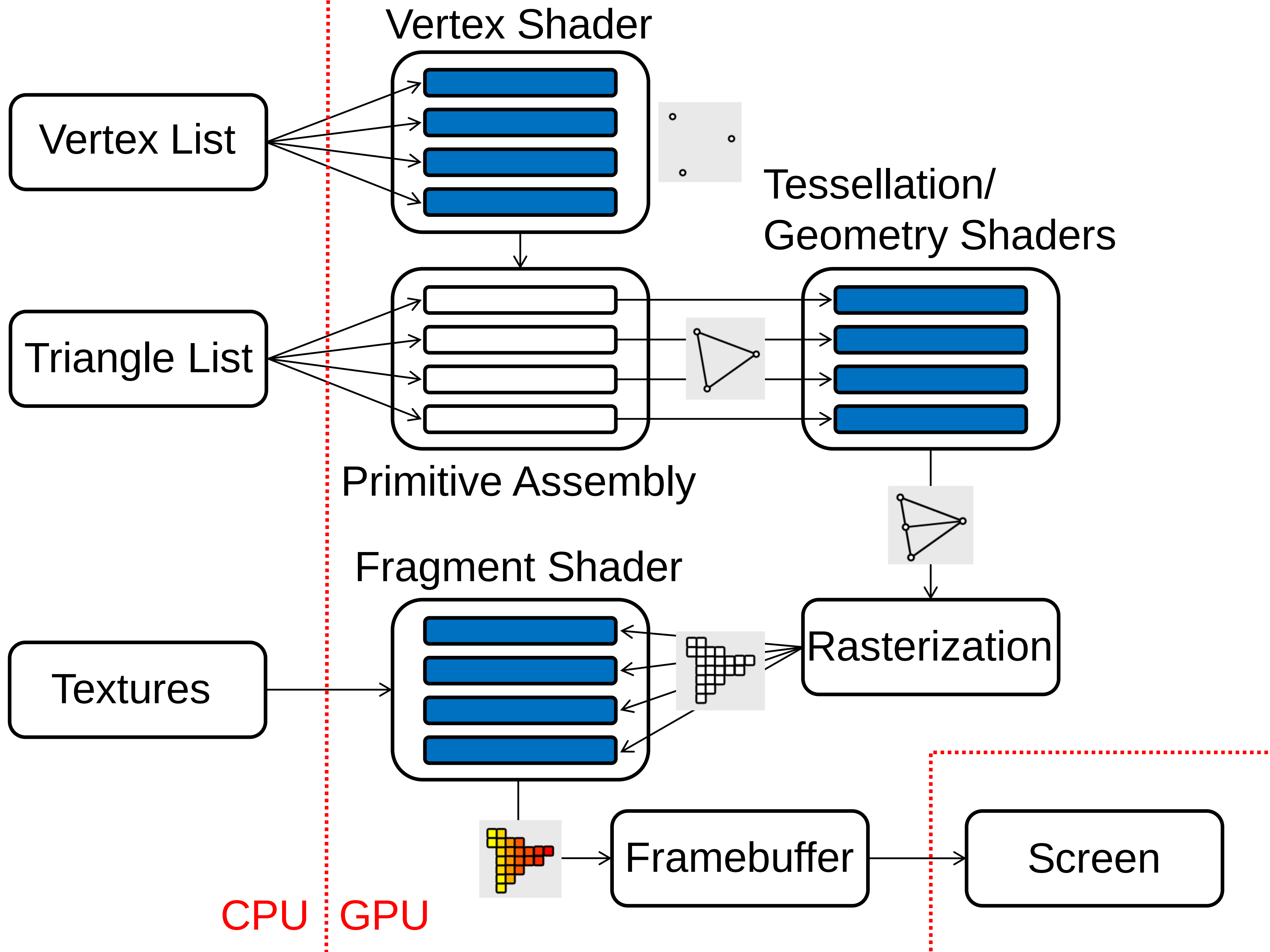


Rasterization Algorithms

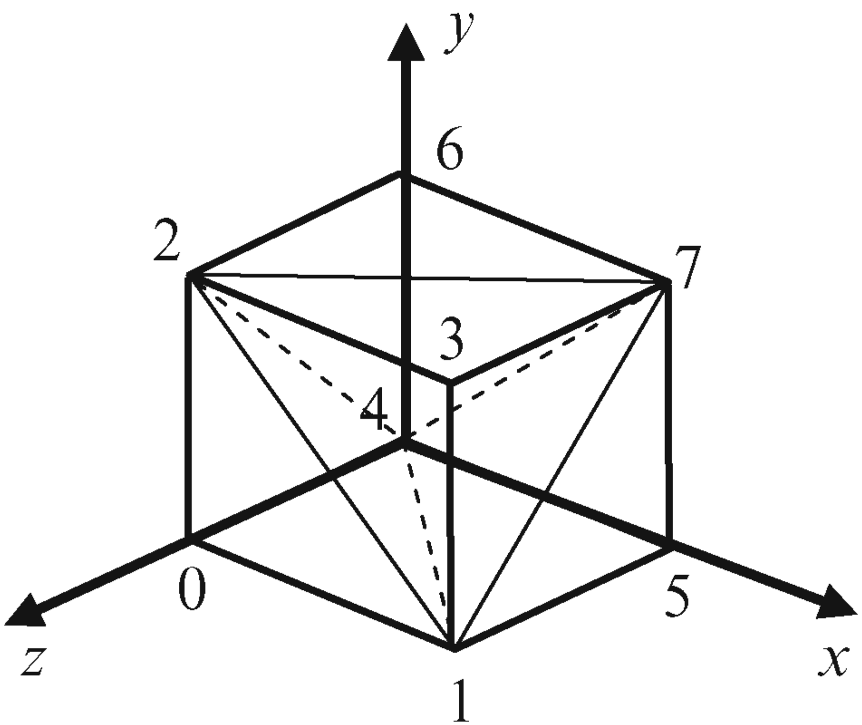
Actually rasterizing objects not so easy…



…so use specialized hardware to do it


Vertex Shader
Vertex List
Tessellation/
Geometry Shaders
Triangle List
Primitive Assembly
Fragment Shader
Rasterization
Textures
Framebuffer
Screen
CPU
GPU

Vertices and Triangles



Vertex List		
x	y	z
0.0	0.0	1.0
1.0	0.0	1.0
0.0	1.0	1.0
1.0	1.0	1.0
0.0	0.0	0.0
1.0	0.0	0.0
0.0	1.0	0.0
1.0	1.0	0.0

Triangle List		
i	j	k
0	1	2
1	3	2
2	3	7
2	7	6
1	7	3
1	5	7
6	7	4
7	5	4
0	4	1
1	4	5
2	6	4
0	2	4

Sending Data to the GPU

One vertex/triangle at a time: very slow

Vertex Buffer Objects: big arrays of data

- vertex positions
- vertex colors
- texture info
- etc

Shaders

Small arbitrary programs that run on GPU

Massively parallel

Shaders

Small arbitrary programs that run on GPU

Massively parallel

Four kinds: vertex, tessellation, geometry, fragment

Shaders

Small arbitrary programs that run on GPU

Massively parallel

Four kinds: vertex, tessellation, geometry, fragment

These days: used for many non-rendering applications (GPGPU)

Vertex Shader

Runs in parallel on every vertex

- no access to triangles or other verts

Vertex Shader

Runs in parallel on every vertex

- no access to triangles or other verts

Main job: transform vertex positions

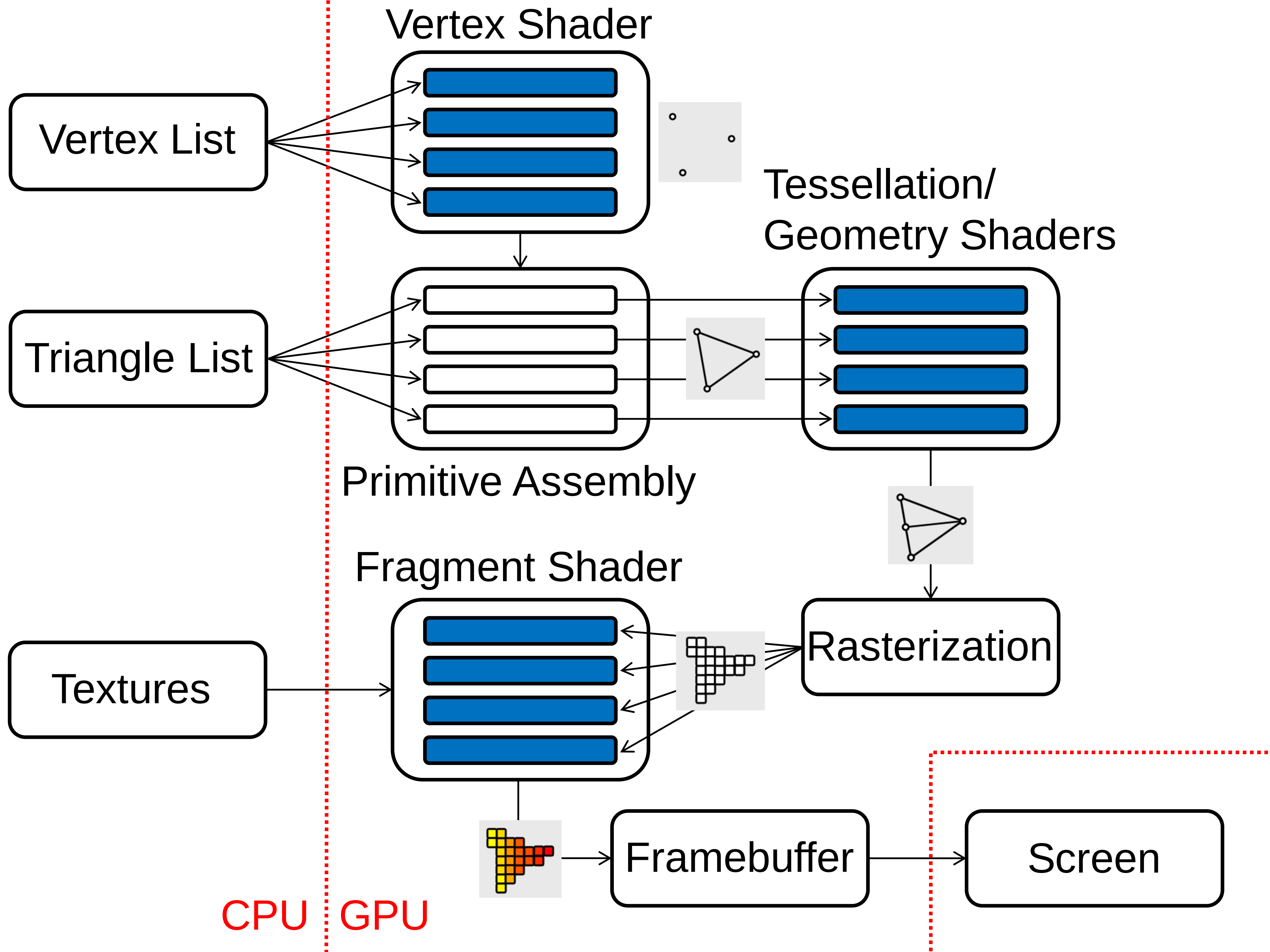
Vertex Shader

Runs in parallel on every vertex

- no access to triangles or other verts

Main job: transform vertex positions

Also used for shading



Vertex Shader
Vertex List
Tessellation/
Geometry Shaders
Triangle List
Primitive Assembly
Fragment Shader
Rasterization
Textures
Framebuffer
Screen
CPU
GPU

Processing Primitives

Assembly: group verts into polygons

Processing Primitives

Assembly: group verts into polygons

Tessellation shader: runs on each triangle

- can split triangles into subtriangles
  - increase level of detail near camera, etc

Processing Primitives

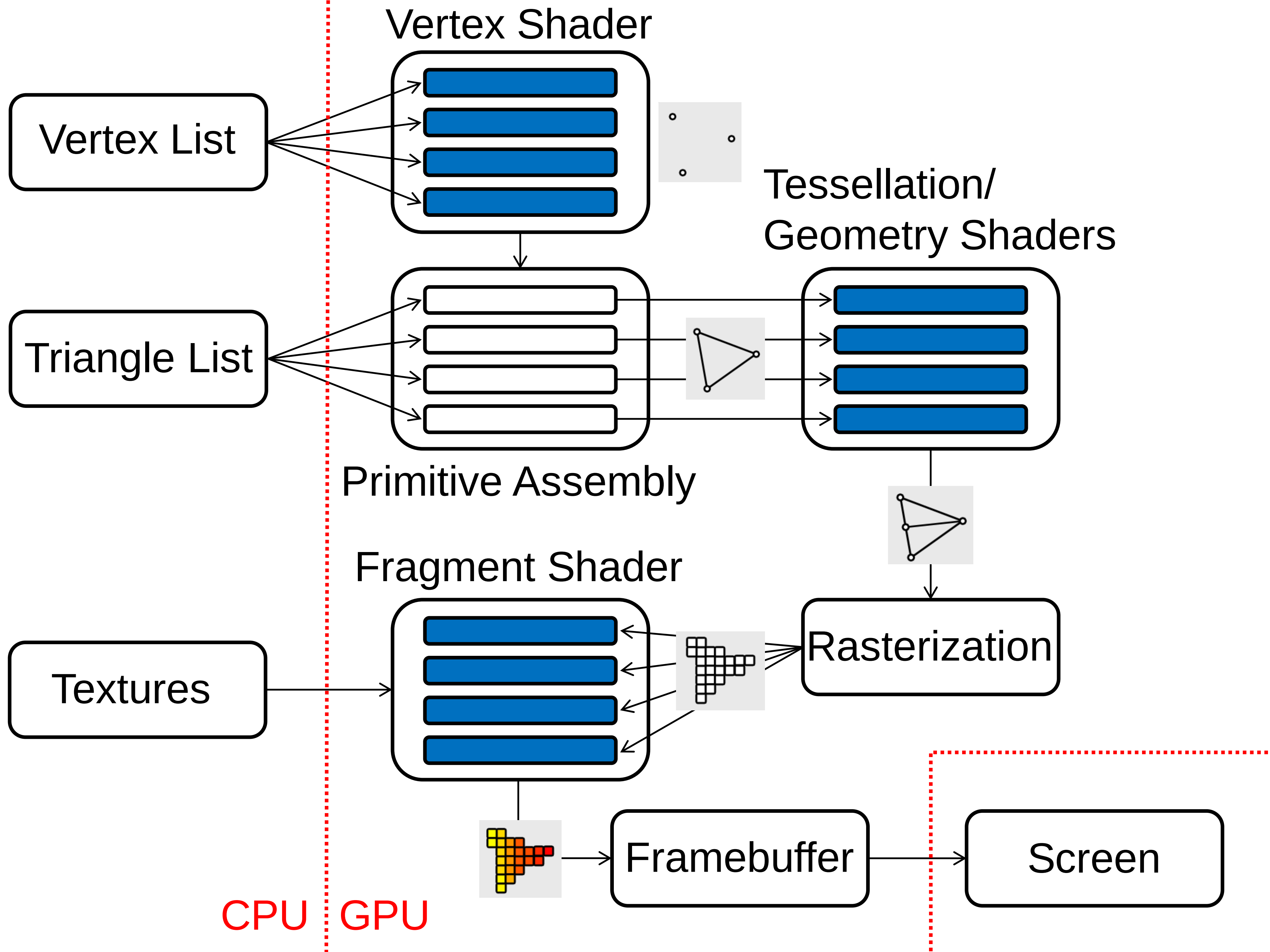
Assembly: group verts into polygons

Tessellation shader: runs on each triangle

- can split triangles into subtriangles
  - increase level of detail near camera, etc

Geometry shader: runs on each triangle

- can access verts and neighbors
- more general than tessellation, slower


Vertex Shader
Vertex List
Tessellation/
Geometry Shaders
Triangle List
Primitive Assembly
Fragment Shader
Rasterization
Textures
Framebuffer
Screen
CPU
GPU

Fragment Shader

Runs in parallel on each fragment (pixel)

- rasterization: one tri -> many fragments

Writes color and depth for one pixel (only)

Final texturing/coloring of the pixels

Fragment Shader

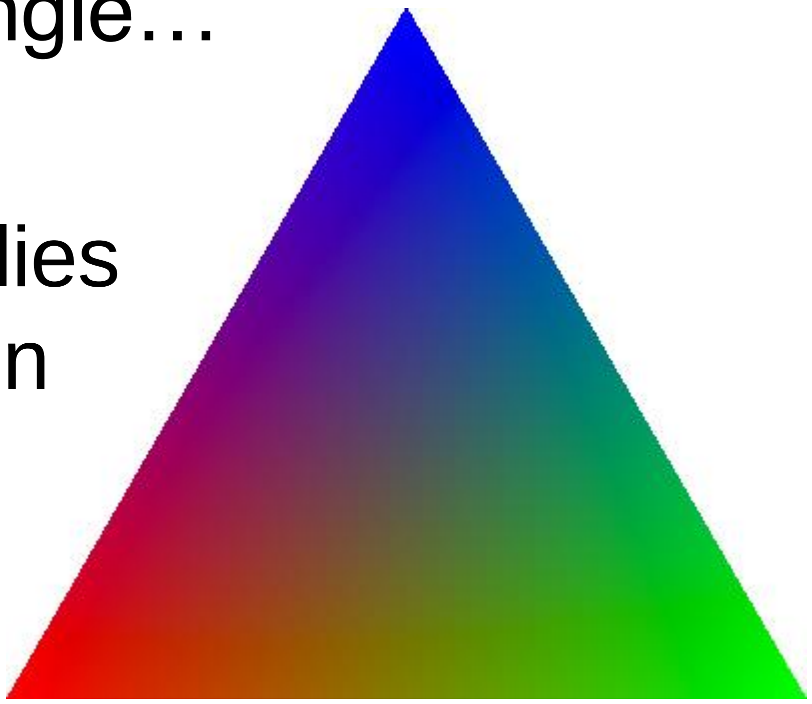
Many fragments per triangle…

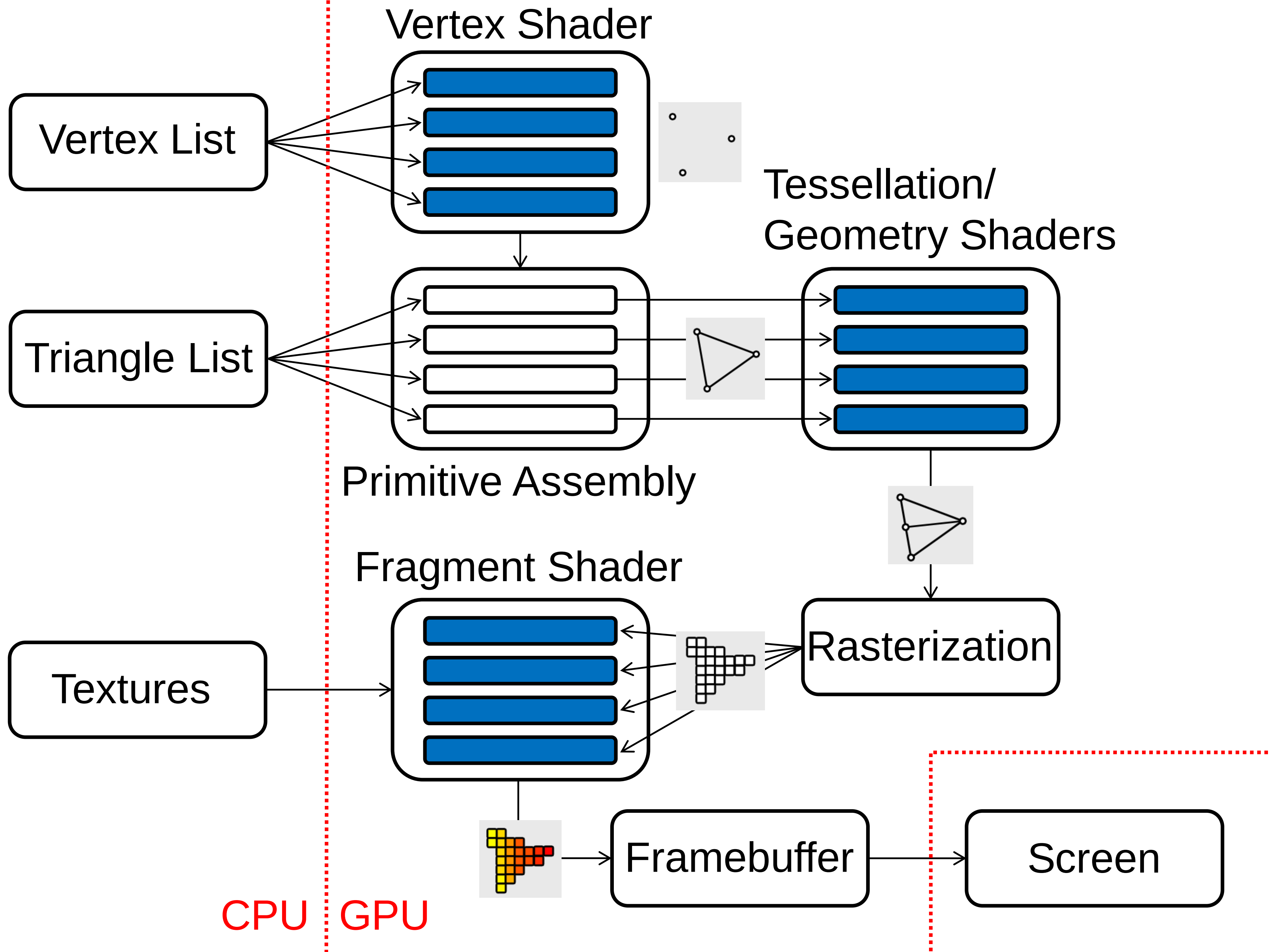
Fragment Shader

Many fragments per triangle…

GPU **automatically** applies barycentric interpolation

UV coords, normals, colors, …





Vertex Shader
Vertex List
Tessellation/
Geometry Shaders
Triangle List
Primitive Assembly
Fragment Shader
Textures
Rasterization
Framebuffer
Screen
CPU
GPU

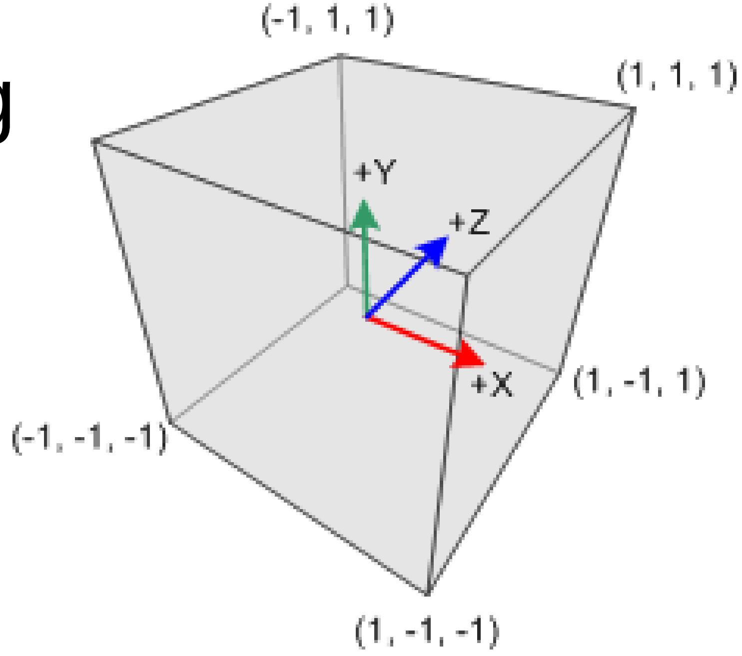
Normalized Device Coordinates

Before rasterization, must decide what geometry to show and where

Normalized Device Coordinates

Before rasterization, must decide what geometry to show and where

GPU draws everything in unit cube

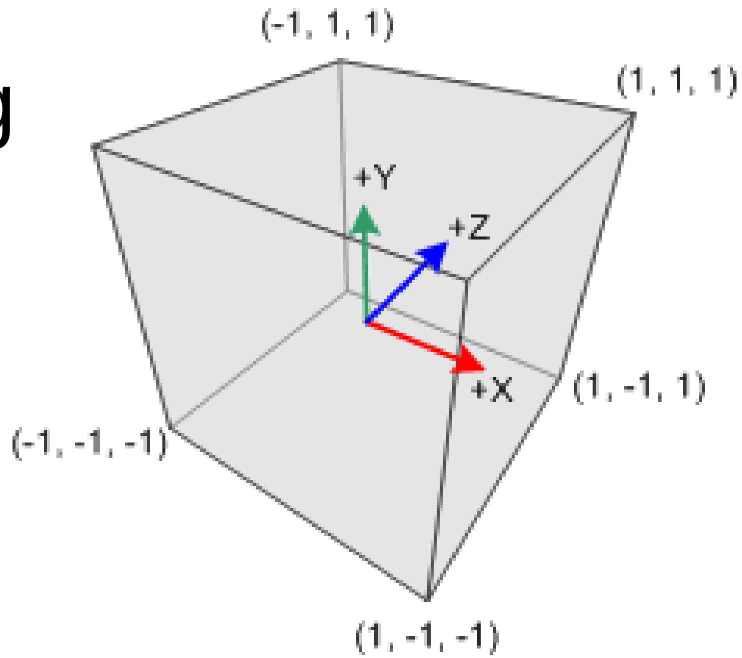


Normalized Device Coordinates

Before rasterization, must decide what geometry to show and where

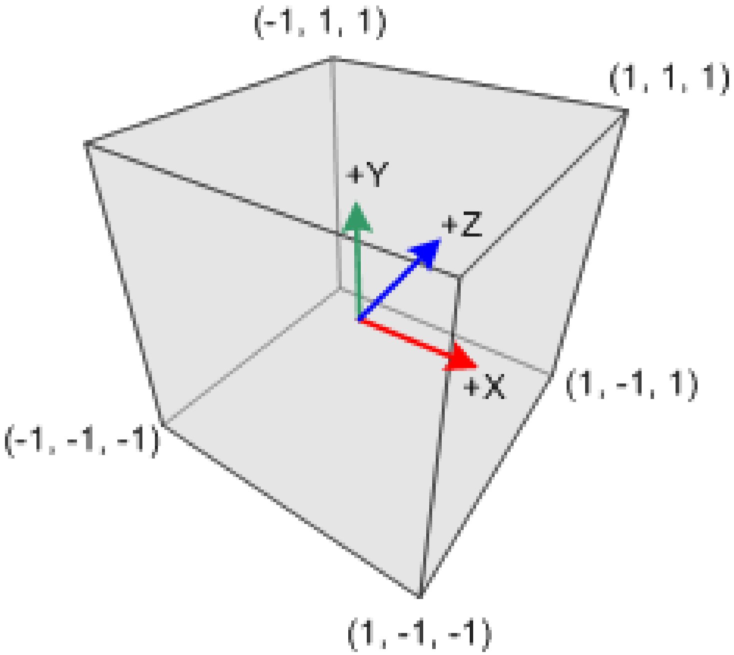
GPU draws everything in unit cube

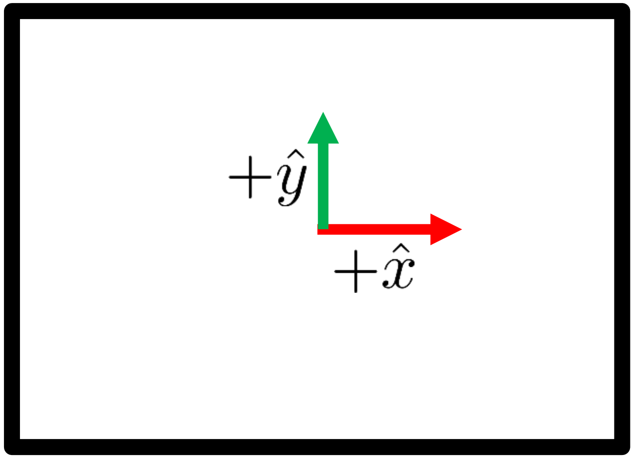
Everything **clipped**



Normalized Device Coordinates

X & Y axes map to screen width & height

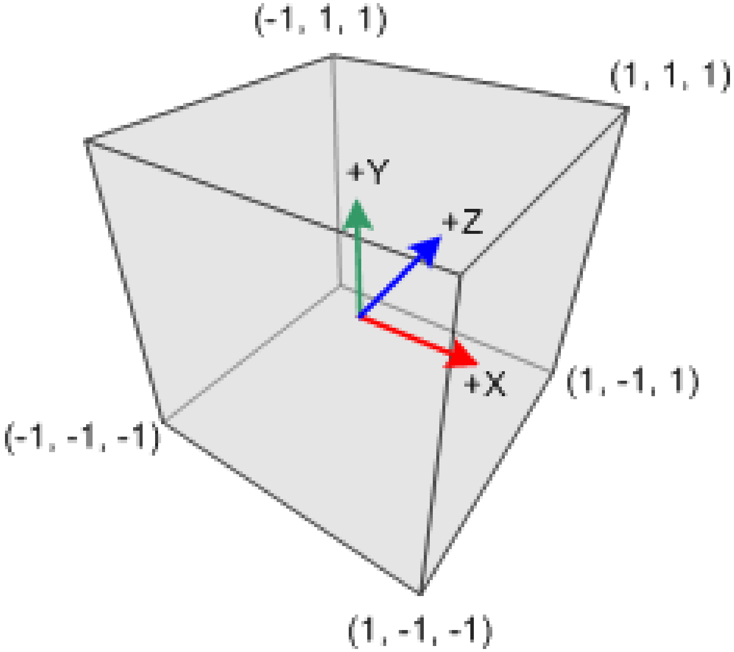


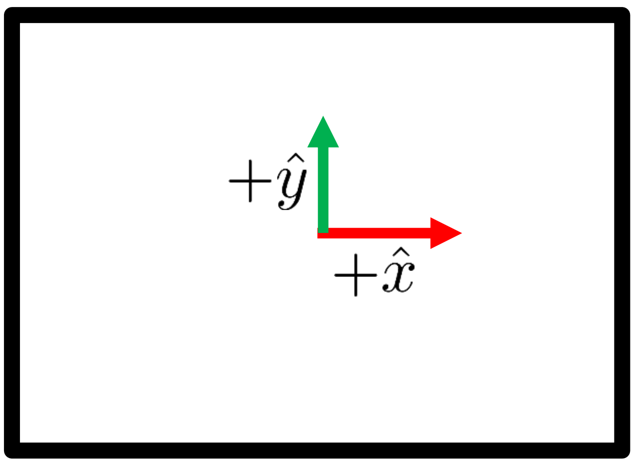


Normalized Device Coordinates

X & Y axes map to screen width & height

Z used for **depth**

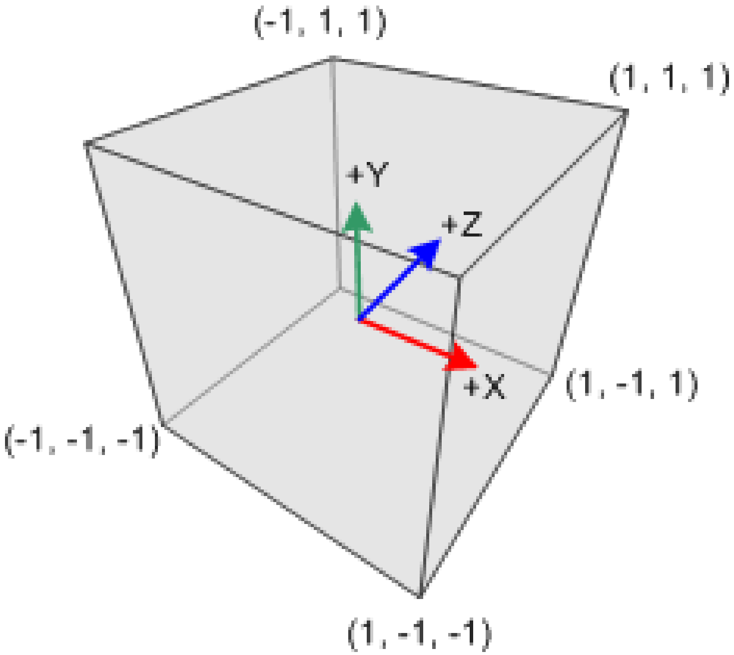




Normalized Device Coordinates

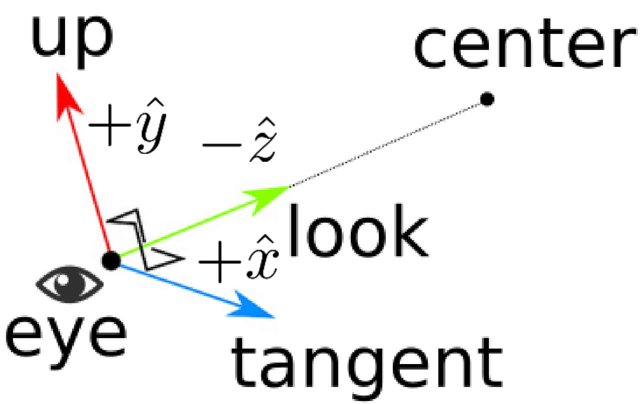
Notice: deeper points have **higher** z

(not right-handed)



Camera Coordinates

Notice: look down **negative** z direction

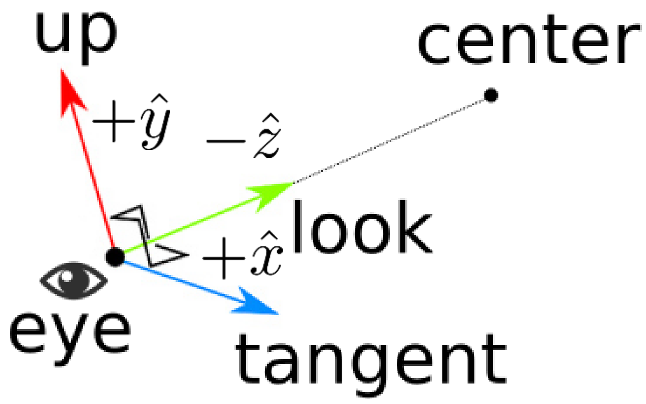


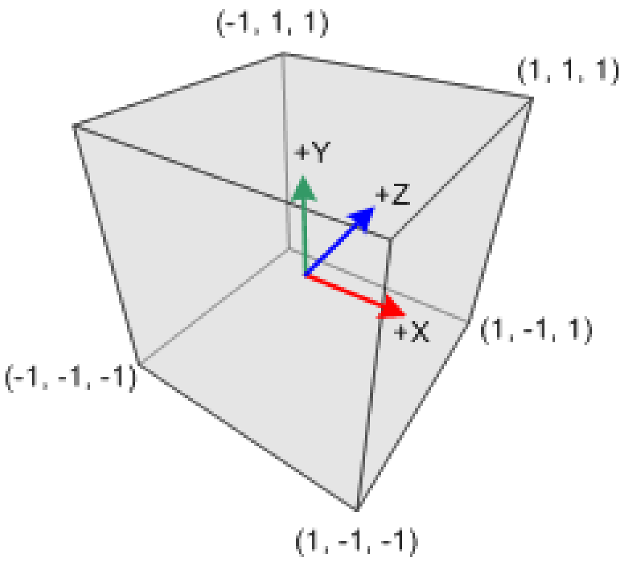
Camera Coordinates

Notice: look down **negative** z direction

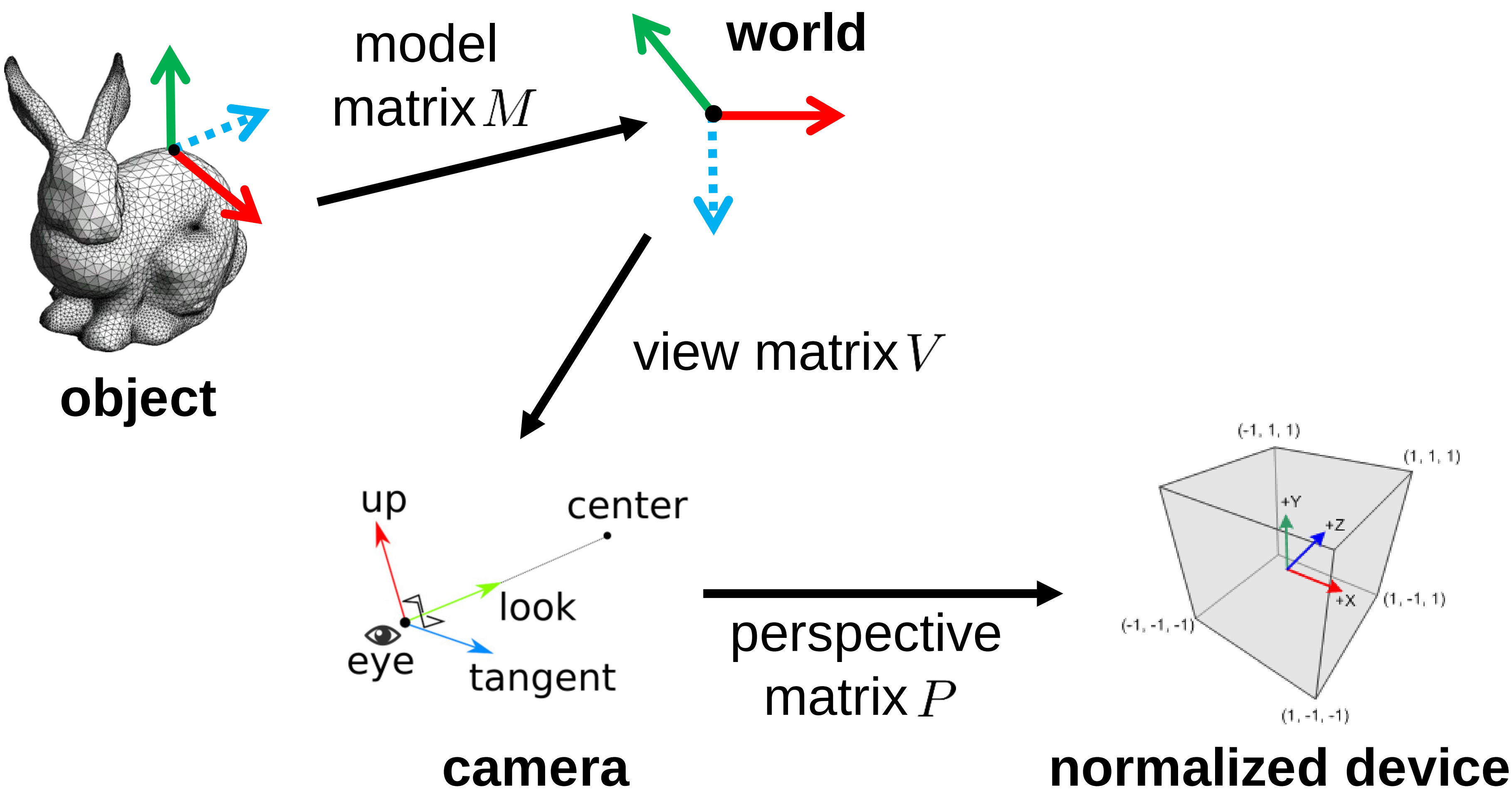
Projection: transform from camera to NDC

(typically in vertex shader)



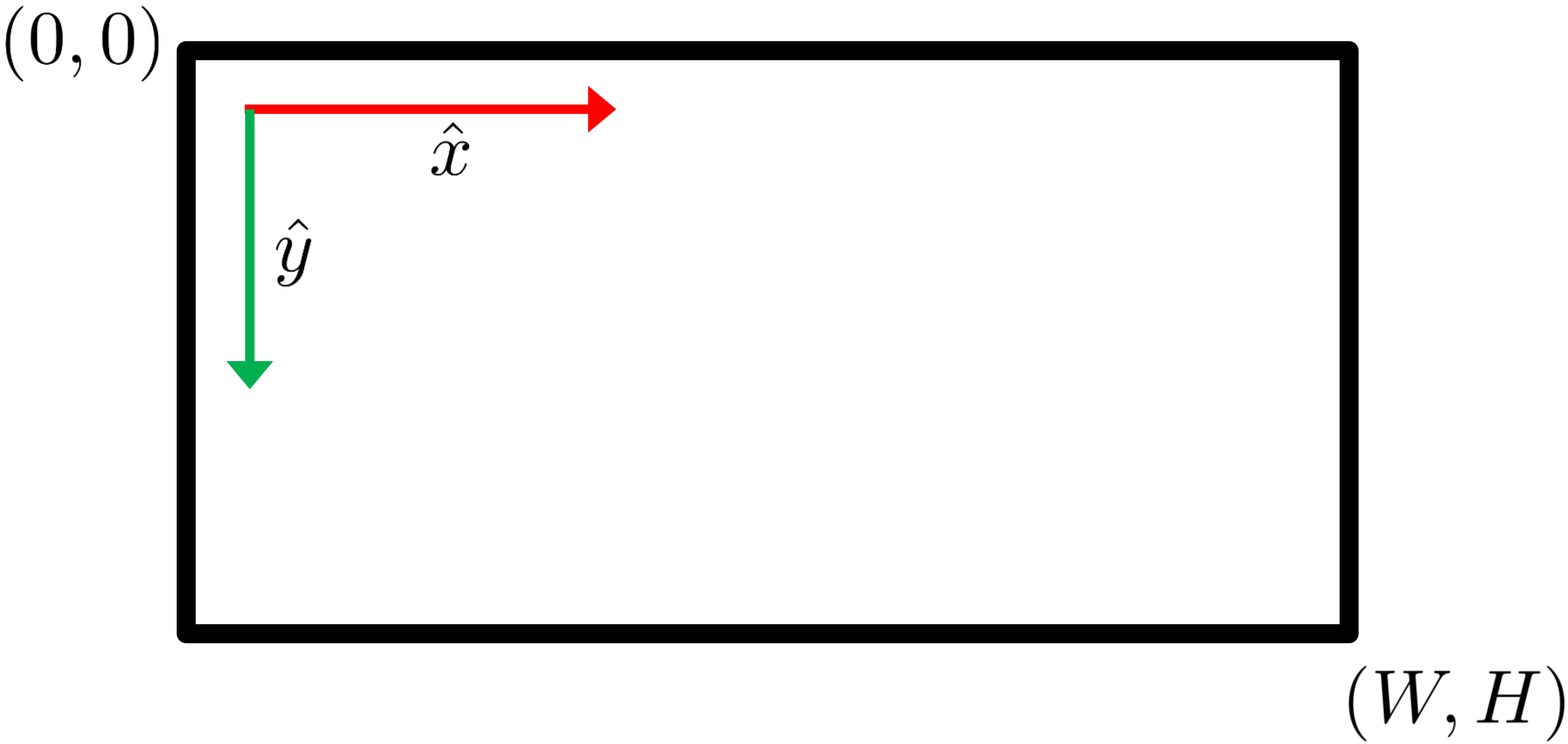


Coordinate Systems in Graphics



For Extra Confusion

Screen coordinates



Framebuffer

Memory region containing pixel data

The old days: mapped to RAM with DMA

- CPU could write to it directly

Framebuffer

Memory region containing pixel data

The old days: mapped to RAM with DMA

- CPU could write to it directly

Now: GPU controls it

Framebuffer

Several layers:

- Color buffer: RGB of each pixel

Framebuffer

Several layers:

- Color buffer: RGB of each pixel
- Depth buffer

Framebuffer

Several layers:

- Color buffer: RGB of each pixel
- Depth buffer
- Stencil buffer, etc

Framebuffer

Several layers:

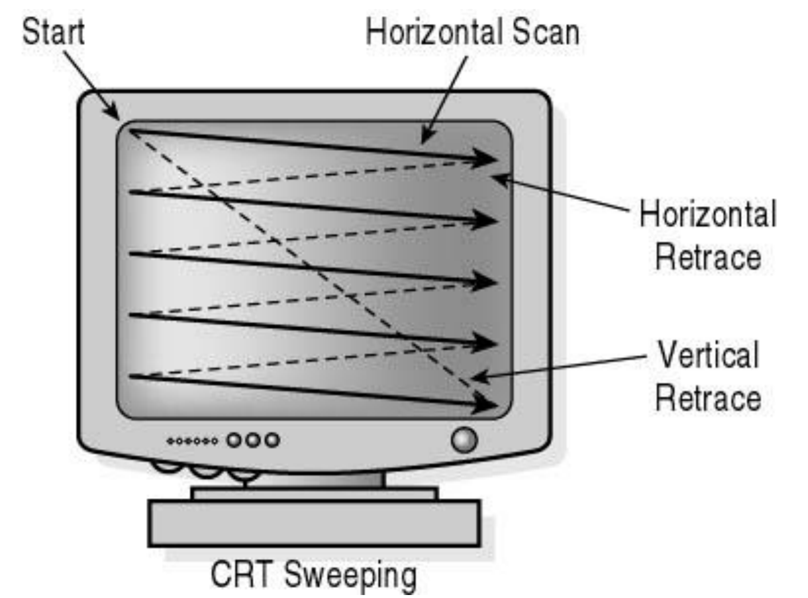
- Color buffer: RGB of each pixel
- Depth buffer
- Stencil buffer, etc

Can be saved to file, to texture, to screen

Displaying the Framebuffer

CRTs: beam sweeps across screen drawing pixels

- one pass: 1/60 secs

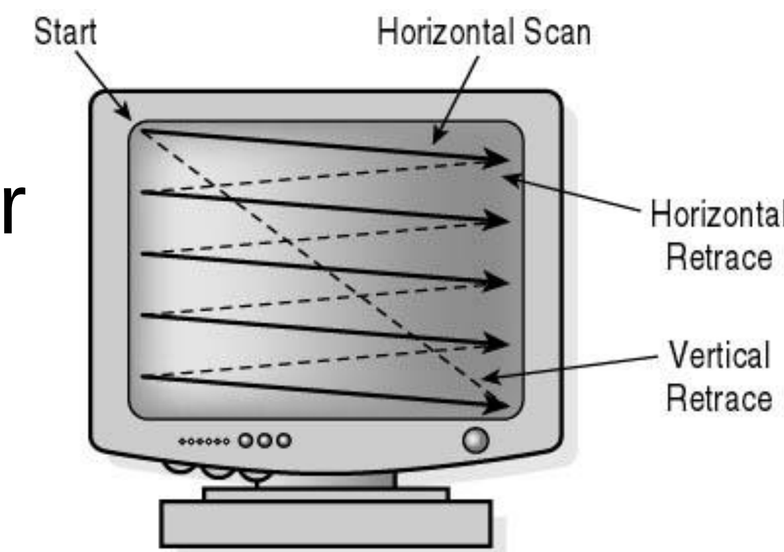


Displaying the Framebuffer

CRTs: beam sweeps across screen drawing pixels

- one pass: 1/60 secs

LCDs: grabs framebuffer every 1/60 secs



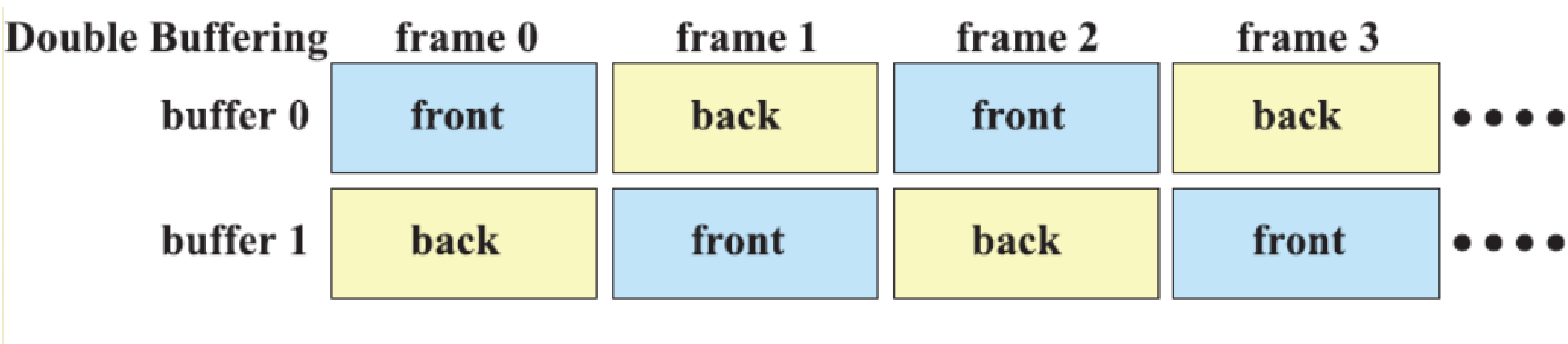
CRT Sweeping

Flickering and Tearing

Framebuffer changes while monitor draws

Double-Buffering to Stop Tearing

Use two framebuffers

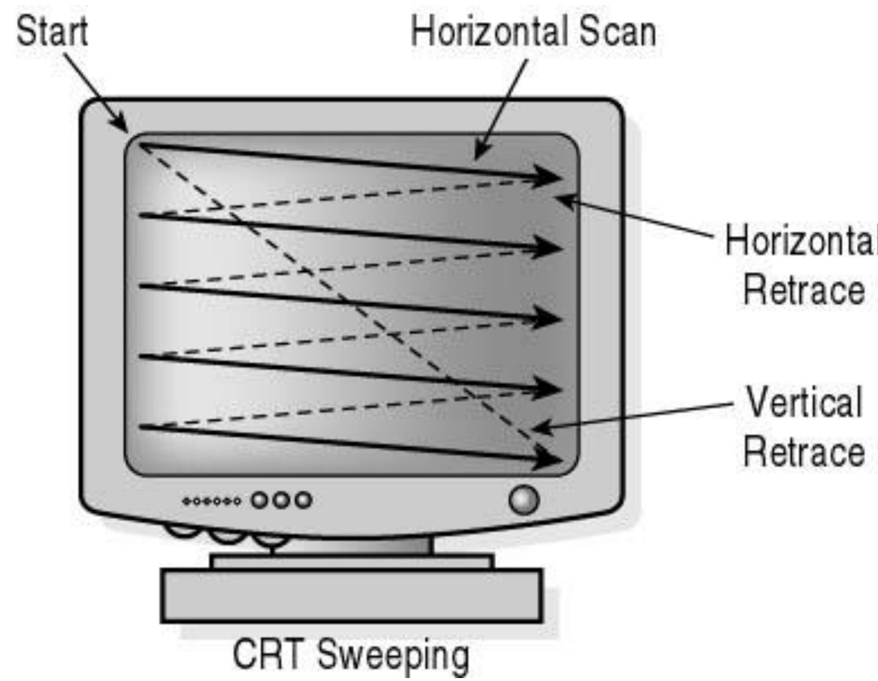


Render to **back buffer** while showing **front buffer**

Then swap

Double-Buffering to Stop Tearing

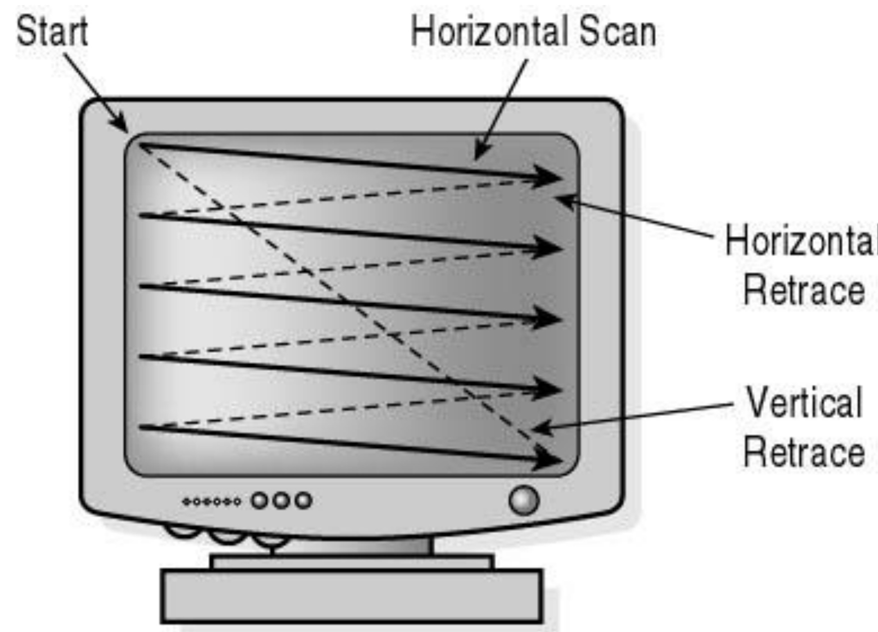
On CRTs: must wait for **vertical retrace** to swap



Double-Buffering to Stop Tearing

On CRTs: must wait for **vertical retrace** to swap

- “vsync”
- occurs 1/60 sec



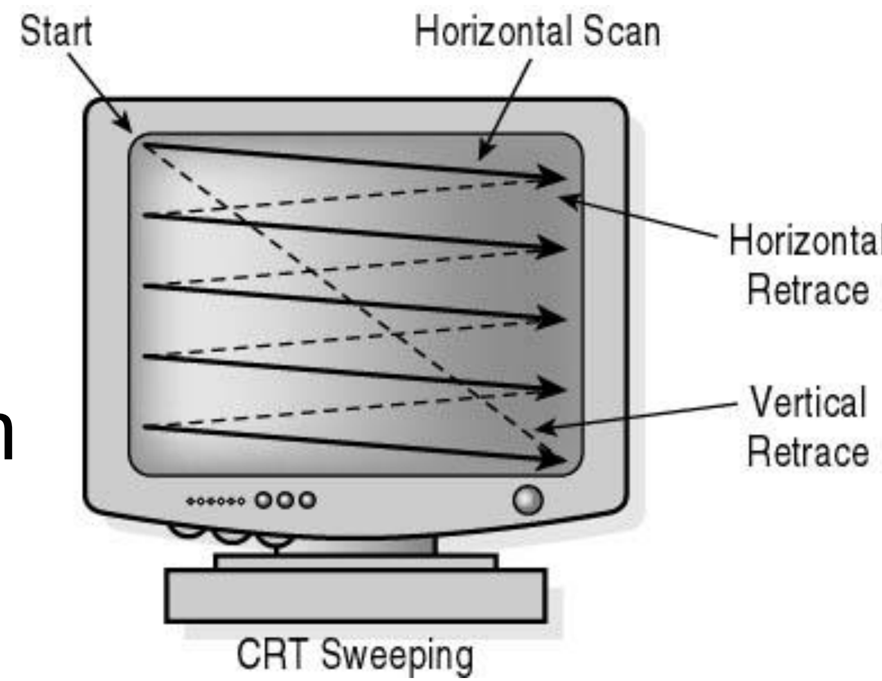
CRT Sweeping

Double-Buffering to Stop Tearing

On CRTs: must wait for **vertical retrace** to swap

- "vsync"
- occurs 1/60 sec

On LCDs: swap when not reading



Communicating with GPU

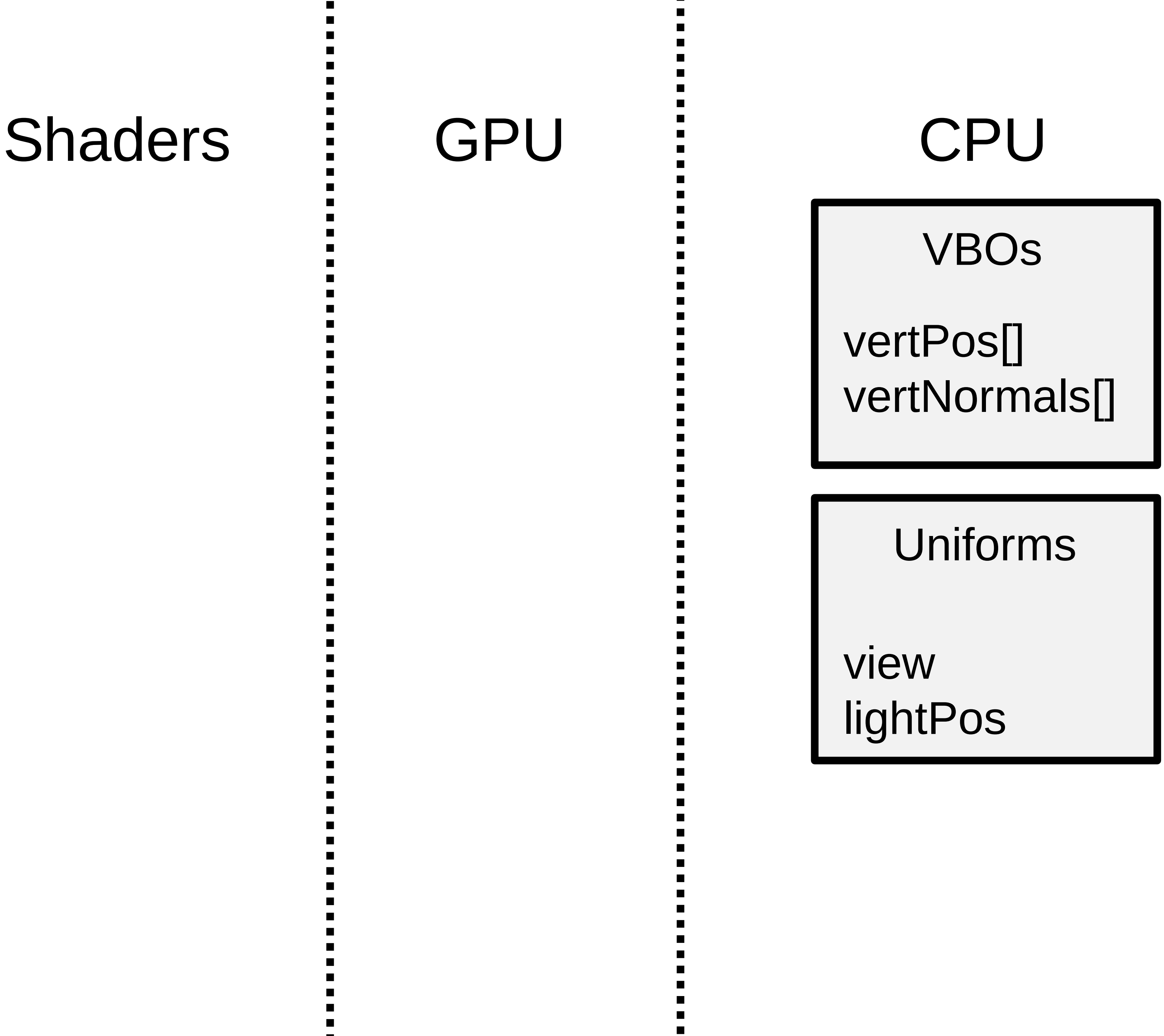
Very low level / awkward

Communicating with GPU

Very low level / awkward

Two types of data:

- vertex attributes in VBOs
- global variables (“uniforms”)


Shaders
GPU
CPU
VBOs
vertPos[]
vertNormals[]
Uniforms
view
lightPos

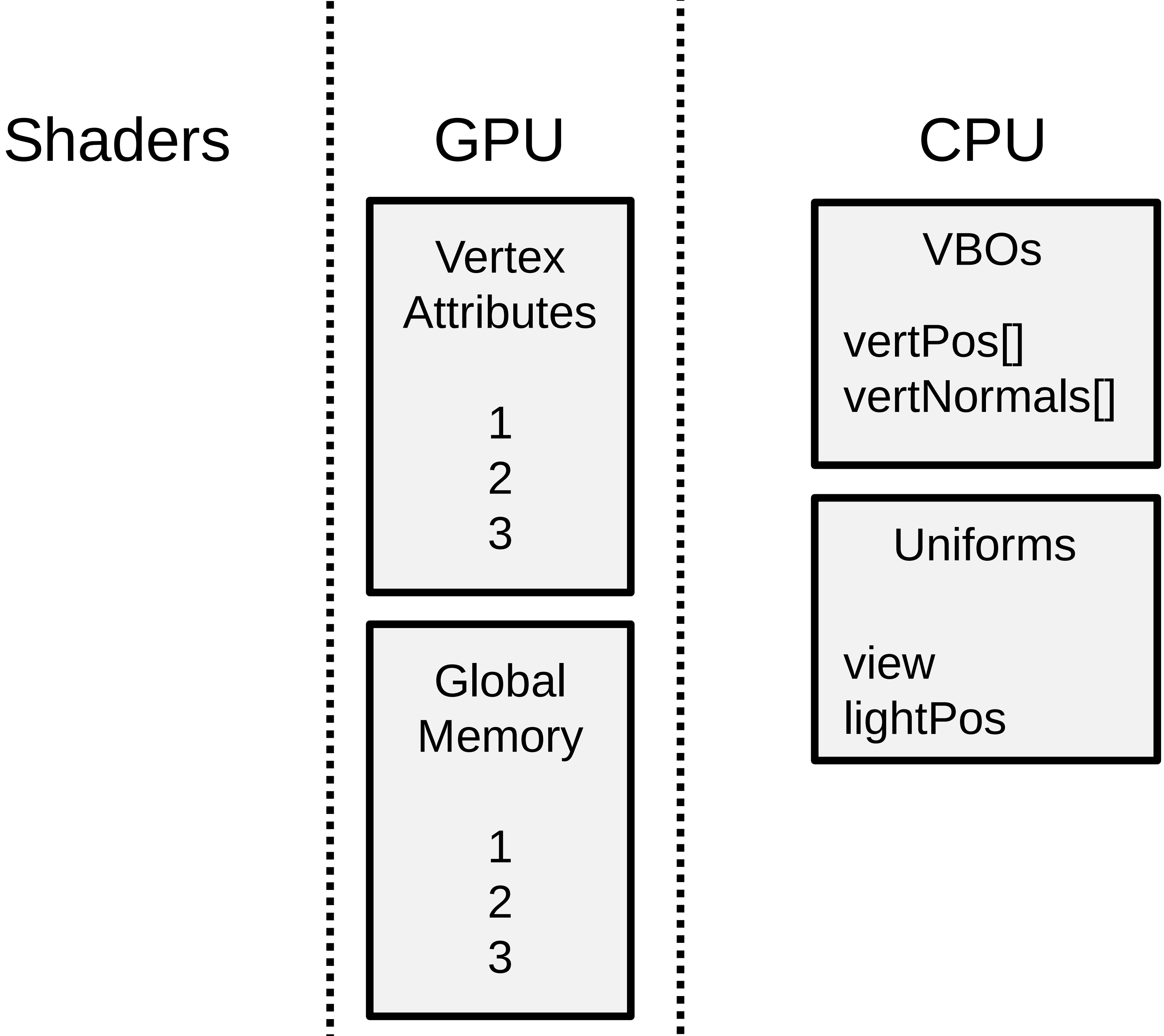
Communicating with GPU

Very low level / awkward

Two types of data:

- vertex attributes in VBOs
- global variables (“uniforms”)

GPU stores no variable names – just location numbers


Shaders
GPU
CPU
Vertex Attributes
1
2
3
Global Memory
1
2
3
VBOs
vertPos[]
vertNormals[]
Uniforms
view
lightPos

Communicating with GPU

Very low level / awkward

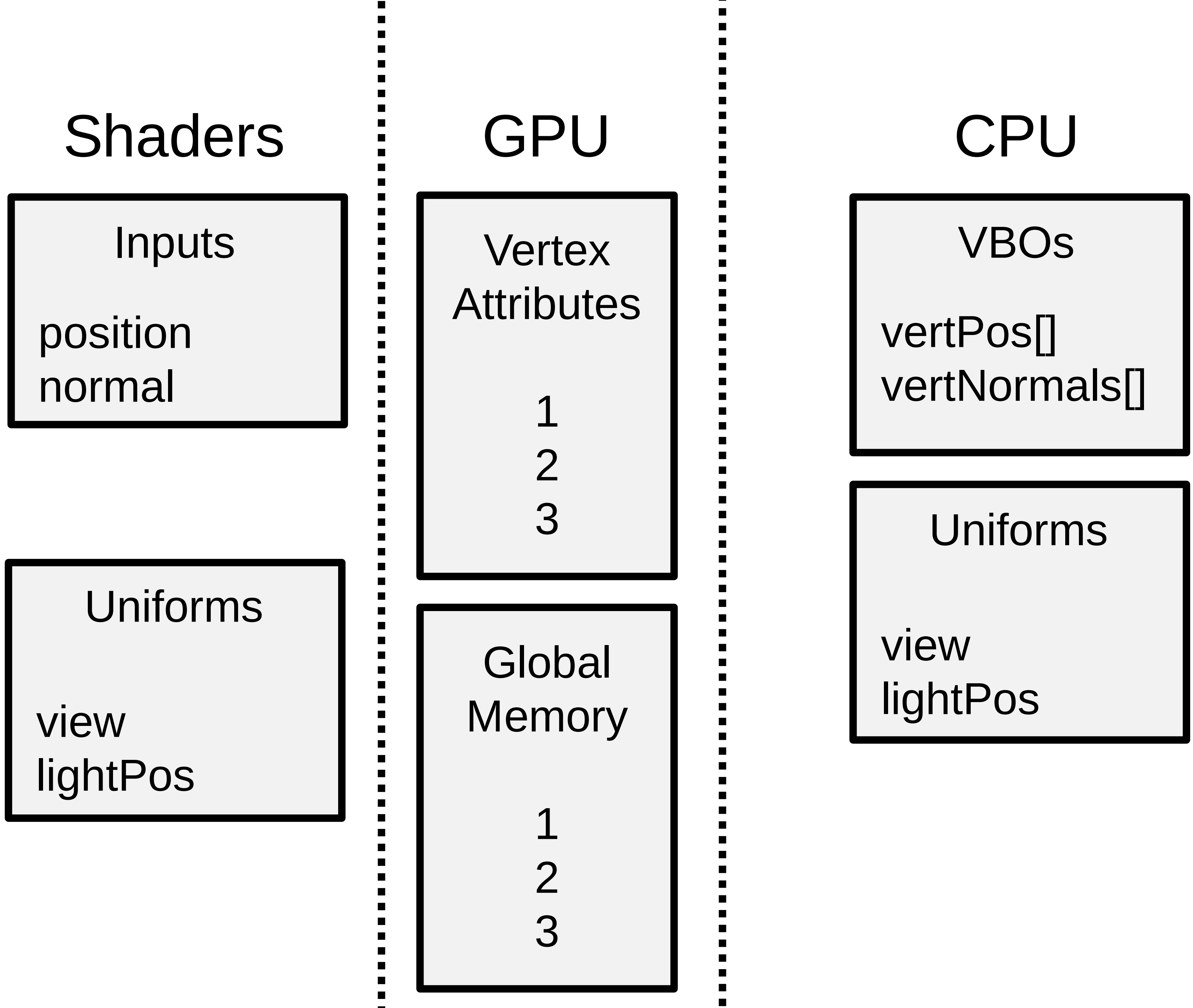
Two types of data:

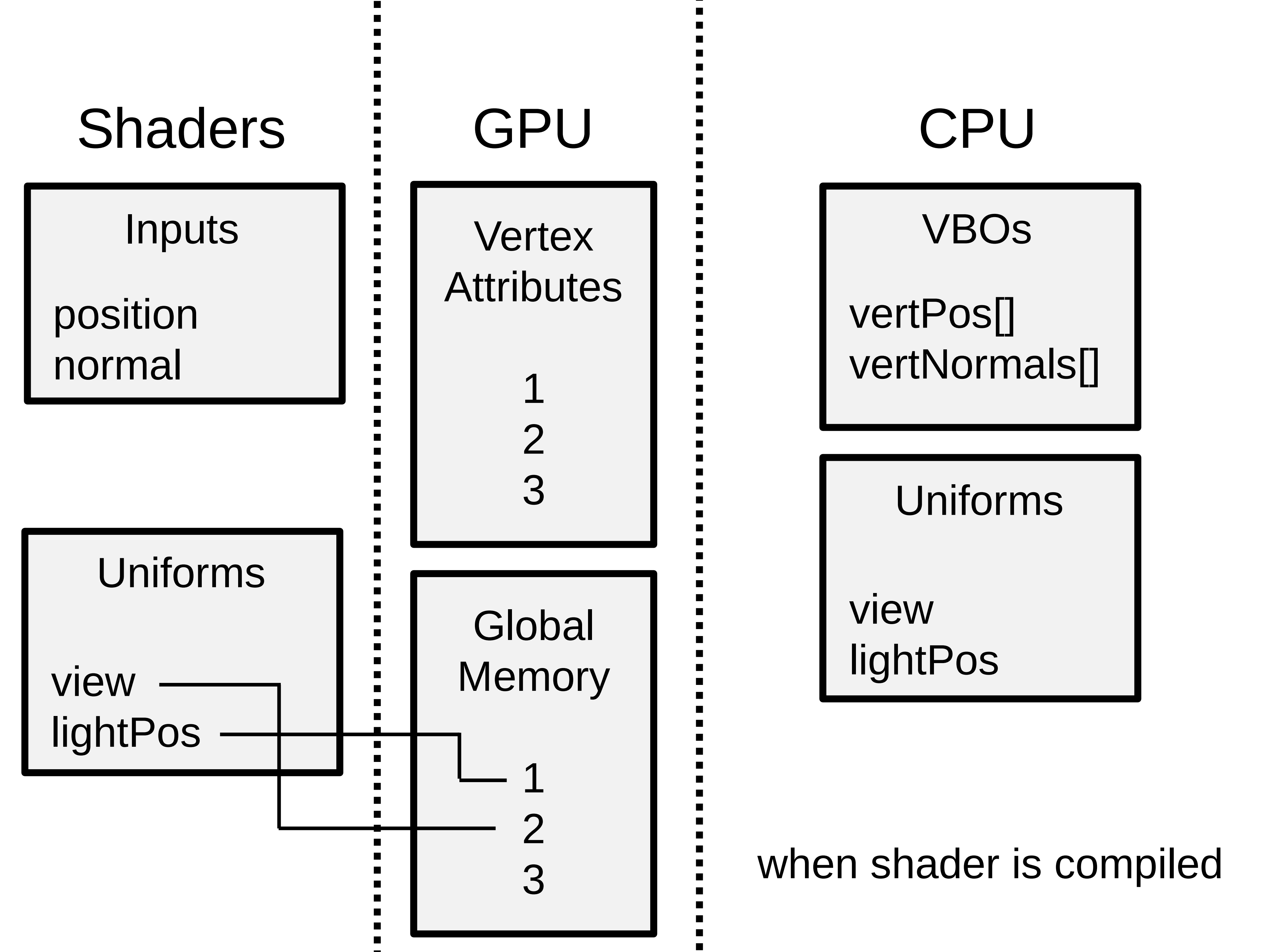
- vertex attributes in VBOs
- global variables (“uniforms”)

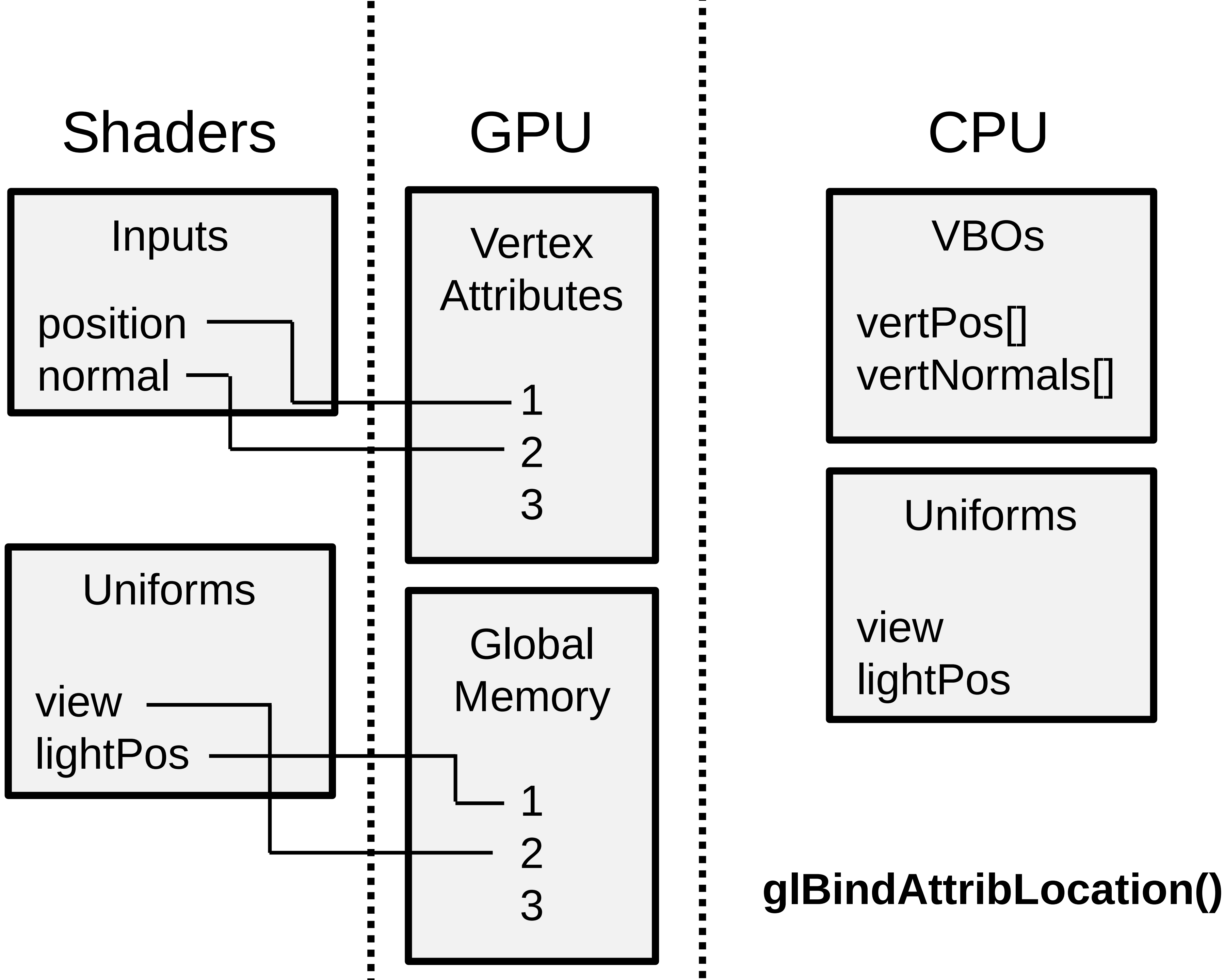
GPU stores no variable names – just location numbers

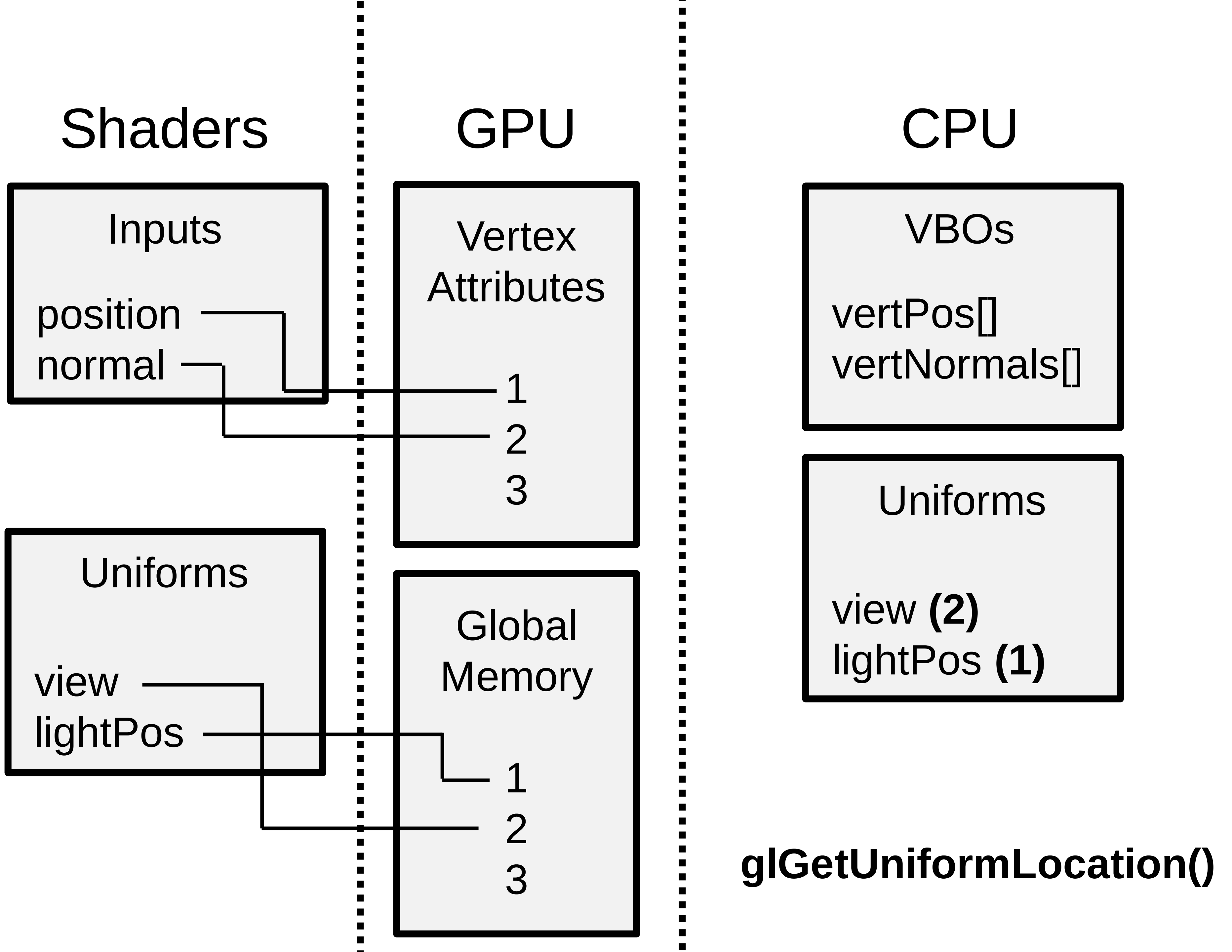
GPU programming is lots of “plumbing”

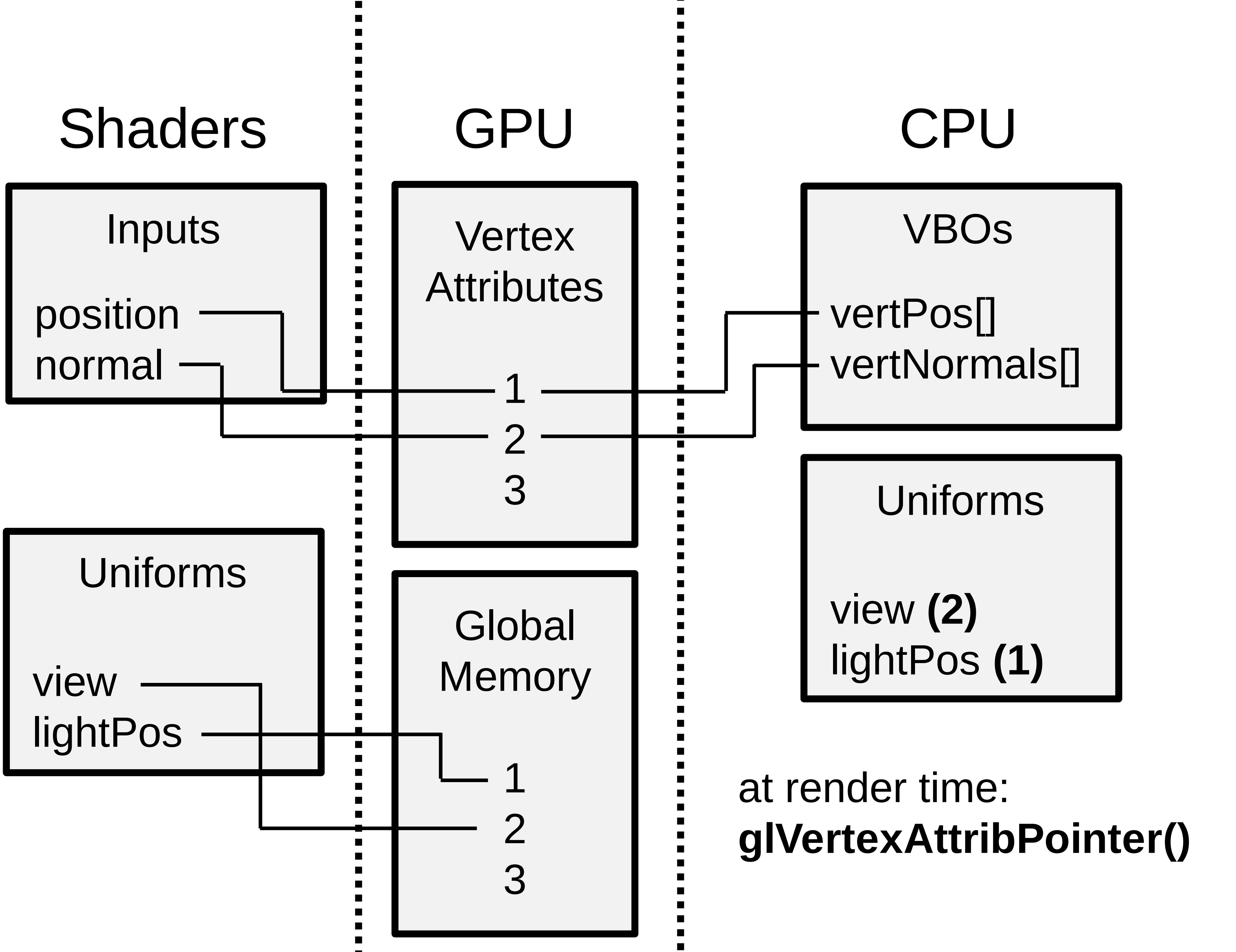
- binding inputs and outputs correctly

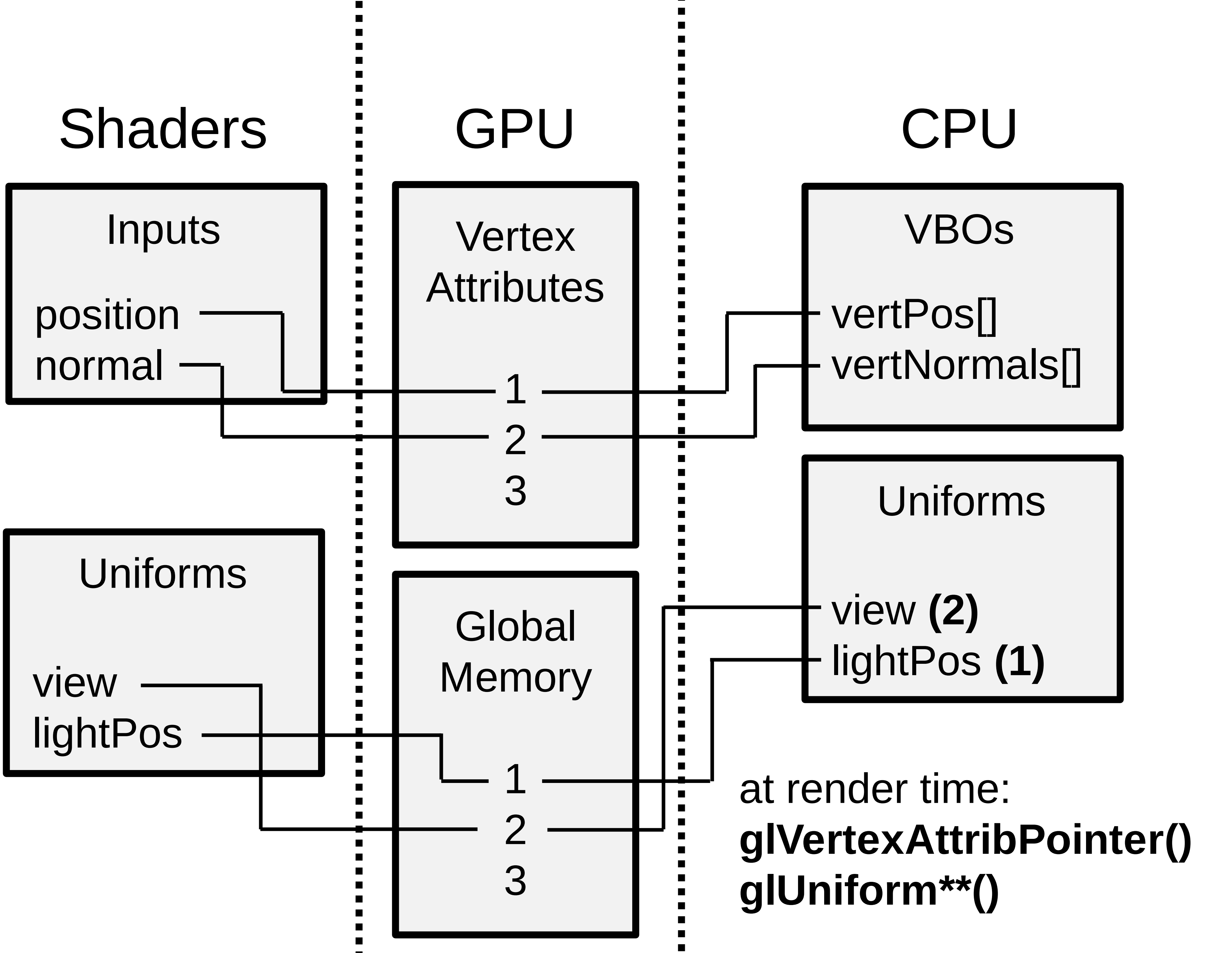

Shaders
GPU
CPU
Inputs
position
normal
Uniforms
view
lightPos
Vertex Attributes
1
2
3
Global Memory
1
2
3
VBOs
vertPos[]
vertNormals[]
Uniforms
view
lightPos

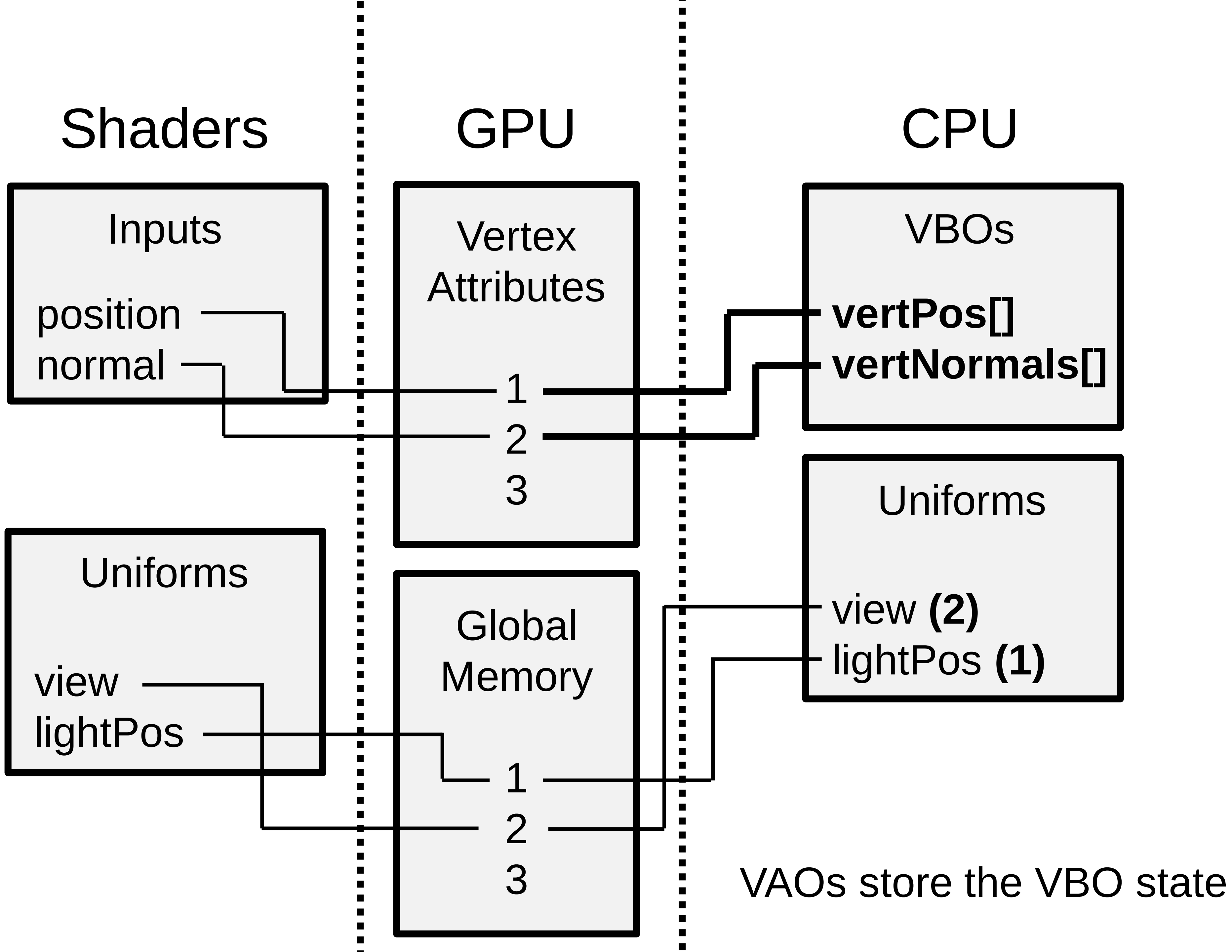

Shaders
GPU
CPU
Inputs
position
normal
Vertex
Attributes
1
2
3
VBOs
vertPos[]
vertNormals[]
Uniforms
view
lightPos
Global
Memory
1
2
3
Uniforms
view
lightPos
when shader is compiled


Shaders
GPU
CPU
Inputs
position
normal
Vertex
Attributes
1
2
3
VBOs
vertPos[]
vertNormals[]
Uniforms
view
lightPos
Global
Memory
1
2
3
Uniforms
view
lightPos
glBindAttribLocation()


Shaders
GPU
CPU
Inputs
position
normal
Vertex
Attributes
1
2
3
VBOs
vertPos[]
vertNormals[]
Uniforms
view
lightPos
Global
Memory
1
2
3
Uniforms
view (2)
lightPos (1)
glGetUniformLocation()


Shaders
GPU
CPU
Inputs
position
normal
Vertex
Attributes
1
2
3
VBOs
vertPos[]
vertNormals[]
Uniforms
view
lightPos
Global
Memory
1
2
3
Uniforms
view **(2)**
lightPos **(1)**
at render time:
glVertexAttribPointer()


Shaders
GPU
CPU
Inputs
position
normal
Vertex Attributes
1
2
3
VBOs
vertPos[]
vertNormals[]
Uniforms
view
lightPos
Global Memory
1
2
3
Uniforms
view **(2)**
lightPos **(1)**
at render time:
glVertexAttribPointer()
glUniform()**


Shaders
GPU
CPU
Inputs
position
normal
Vertex
Attributes
1
2
3
VBOs
vertPos[]
vertNormals[]
Uniforms
view
lightPos
Global
Memory
1
2
3
Uniforms
view (2)
lightPos (1)
VAOs store the VBO state

Ray Tracing: Why Slow? Reprise

Basic ray tracing: 1 ray/pixel

But you **really** want shadows, reflections, global illumination, antialiasing…

- 100-1000 rays/pixel

Much less hardware support

- inhomogeneous / unpredictable work