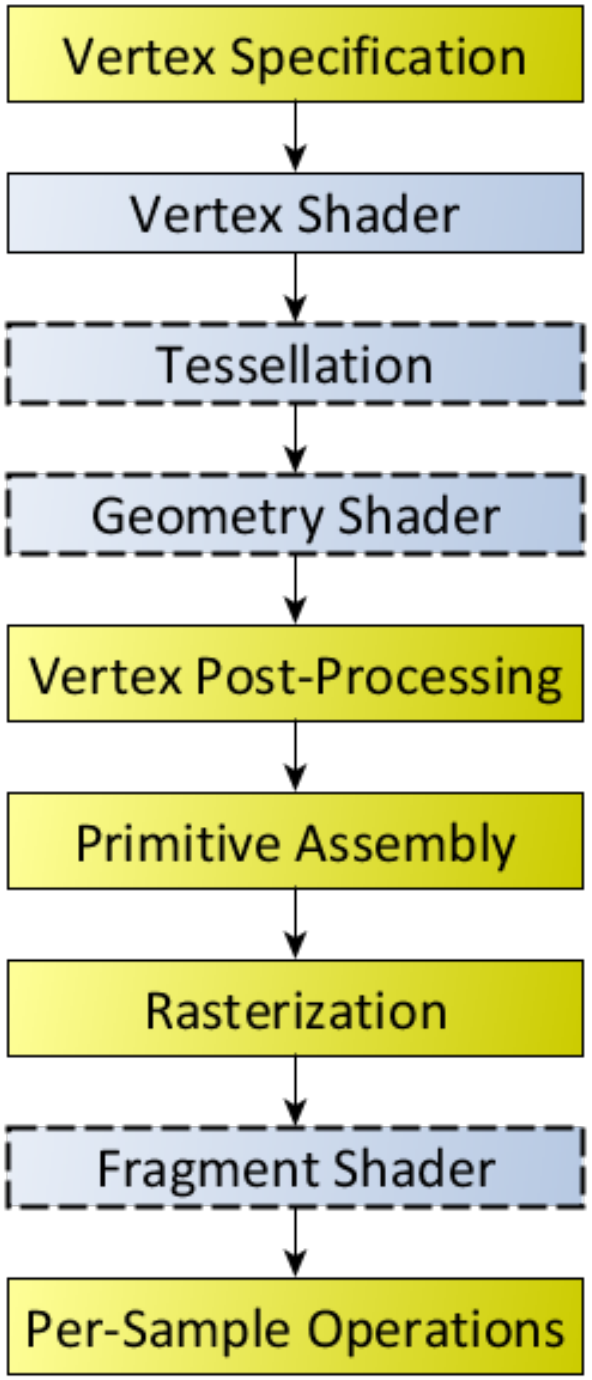
OpenGL with Shaders

OpenGL vs OpenGLES

- OpenGL built for desktop applications
  - Fewer power/heat concerns
- OpenGL**ES** created for embedded/ mobile applications
  - Concerns about power/heat
  - Has fewer features than OpenGL

Shaders

- Small arbitrary programs that run on GPU
- Massively parallel
- Four kinds: vertex, geometry, tessellation, fragment
- Now used for GPGPU calculations, but we're focusing on the graphics aspect!



Data Types

Allows for storage of:

- scalars: `bool`, `int`, `uint`, `float`, `double`
- vectors: `bvecn`, `ivecn`, `uvecn`, `vecn`, `dvecn`
- matrices: `matnxm`, `matn`

Note: matrices are always floating point

Functions

Users can define functions for greater flexibility

```
void method() { //code here }
```

Or use built-in functions:

`sqrt`, `pow`, `abs`, `sin`, `step`, `length`, `reflect`, etc

http://www.shaderific.com/glsl-functions/

Flow Control

All the usual suspects:

- If-else/Switch
- For/While/Do-while (avoid!)
- Break/Return/Continue
- Discard (only in fragment shader)

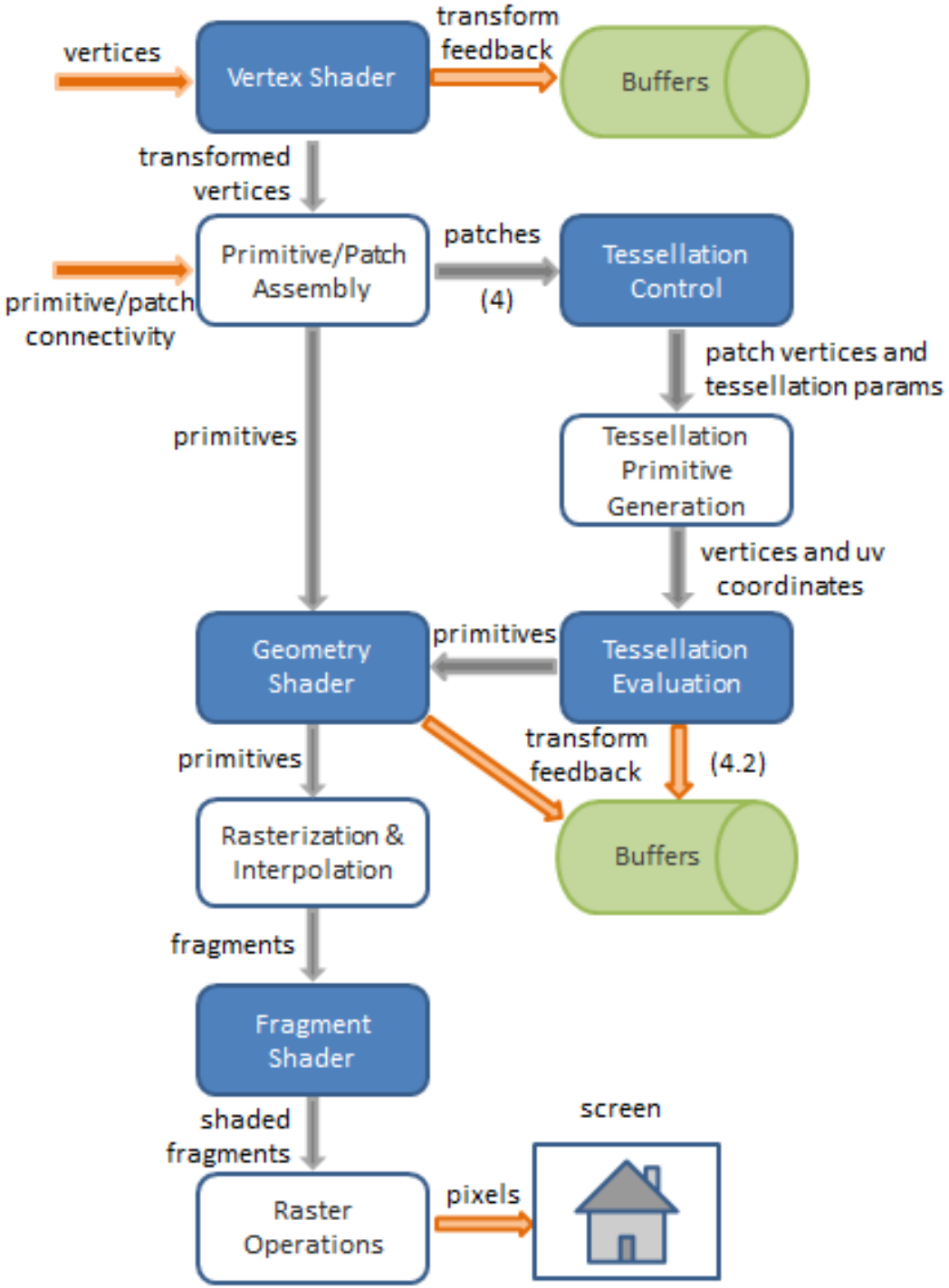
Swizzling

Access vector components individually

```
vec4 a_vector;
a_vector.x + a_vector.y;
```

Any combination allowed: `a_vector.xxyx;`

Syntactic sugar masks: xyzw, rgba, stpq


vertices
Vertex Shader
transform feedback
Buffers
transformed vertices
Primitive/Patch Assembly
primitive/patch connectivity
patches
(4)
Tessellation Control
patch vertices and tessellation params
Tessellation Primitive Generation
vertices and uv coordinates
primitives
Tessellation Evaluation
Geometry Shader
primitives
transform feedback
(4.2)
Buffers
primitives
Rasterization & Interpolation
fragments
Fragment Shader
shaded fragments
Raster Operations
pixels
screen

Vertex Shader

- Runs in parallel on every vertex
- No access to triangles or other vertices

What can we use the vertex shader for?

Vertex Shader Uses

Per-vertex lighting

- Apply toon-shading or other NPR techniques

Height-fields

- Adjust position of vertex based on function or input data

Compute transforms

- Perform matrix calculations in shader

Transforming Vertices

`gl_Position` variable must be assigned for each vertex

- vec4 (x, y, z, w)
- determines vertex transforms during shading

Vertex shader main method:

```
void main() {
 gl_Position = MVP * vertex_position;
}
```

Providing Input to Shaders

Remember `glVertexAttribPointer`?

VAO tracks data between CPU and GPU

- Notifies what data to use and how to use it
- Knows whether data is coming or going

Qualifiers modify storage or behavior of variables

Layout Qualifiers

Determines which buffer stores what values

Example:

`layout(location = attribute index)`
associates buffer to use with VAO index (defined earlier)

Overrides `glBindAttribLocation`

Note: Currently not available in ELSL

Storage Qualifiers

`in` or `out` determines if assignment is being inputted or outputted

`in` links into current shader

`out` links onto next shader stage

Expanded Layout Example

```
layout(location = 4) in vec3
  position;
void main() {
   gl_Position.xyz = position;
   gl_Position.w = 1.0;
}
```

What is this doing?

Uniforms

Global GLSL variables

- Constant within the shader
- Same value for all verts/fragments
- Cannot be passed to `in` or `out`

Why might this be useful?

Specifying Uniforms

Can specify a variety (and number) of scalars, vectors, and matrices:

`glUniform3f`

`glUniform2i`

`glUniform4ui`

`glUniform3fv`

`glUniformMatrix4fv`

Using Uniforms

```
uniform type variable_name;
```

Uniforms can be set using default initialization:

```
uniform float scale = 2.0;
```

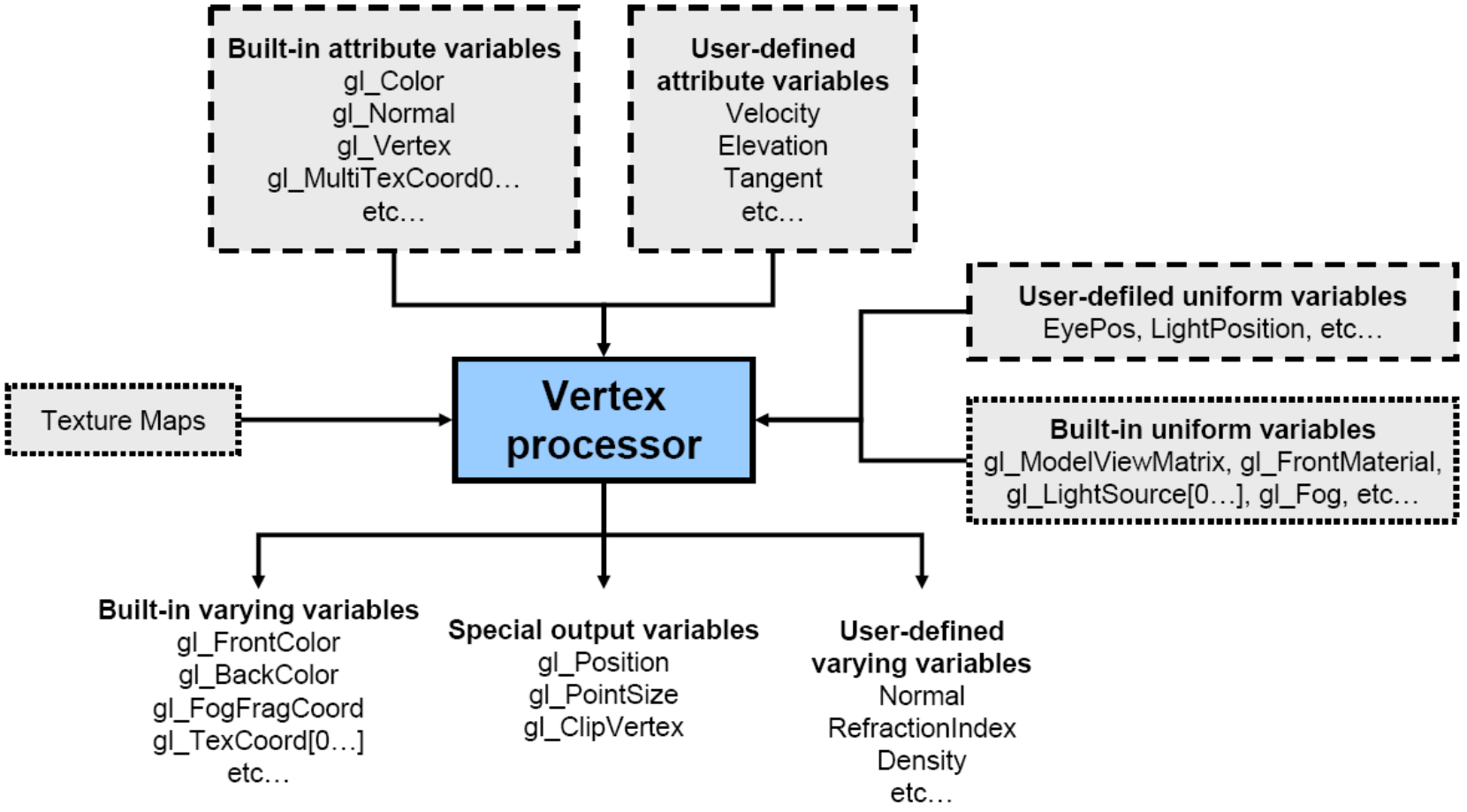
Or pass in uniforms using `glUniform`:

```
GLint scale_location =
  glGetUniformLocation(program_id, “scale”);
glUniform1f(scale_location, 2.0);
```

Expanded Uniform Example

```
layout(location = 0) in vec4
  vertex_position;
uniform mat4 MVP;
void main() {
   gl_Position = MVP *
     vertex_position;
}
```

Vertex Shader Inputs and Outputs



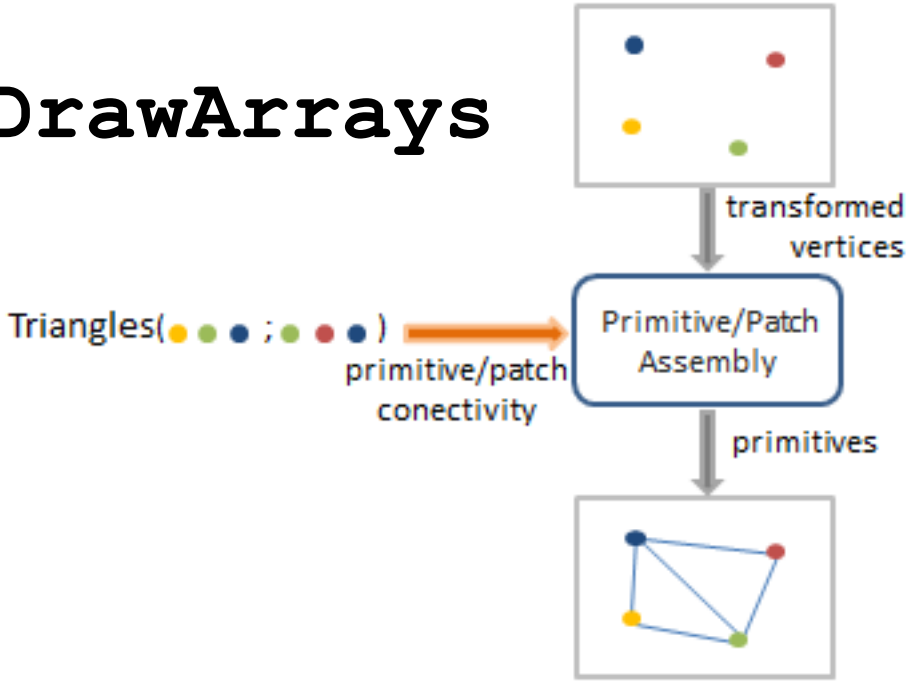
Built-in attributes from old OpenGL (but still useful conceptually)

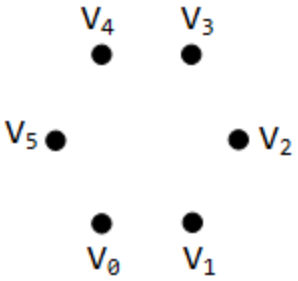
Processing Vertices

Must assemble a group of verts into a polygon

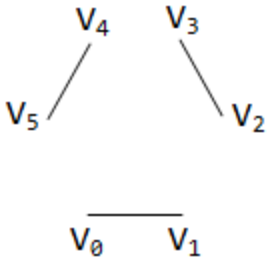
Primitives can be: points, lines, triangles, patches

Assembly defined by **`glDrawArrays`**

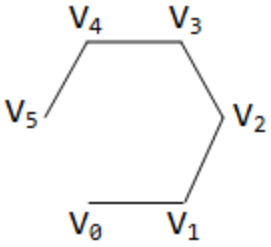




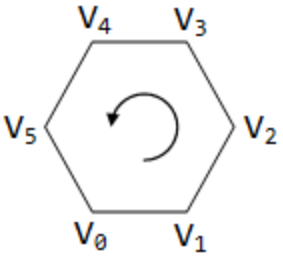
GL_POINTS



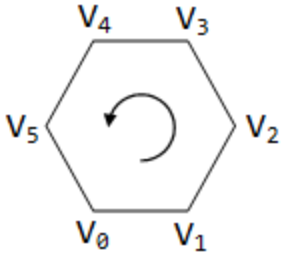
GL_LINES



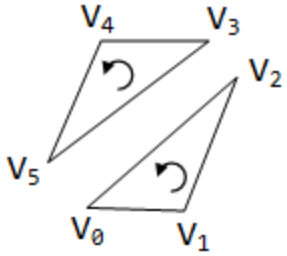
GL_LINE_STRIP



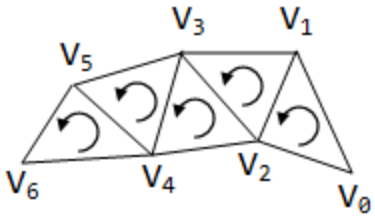
GL_LINE_LOOP



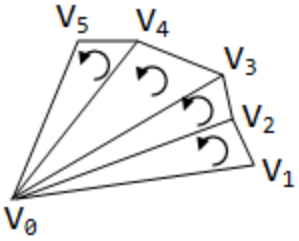
GL_POLYGON



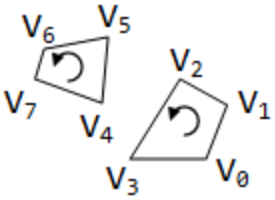
GL_TRIANGLES



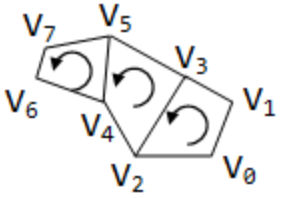
GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN

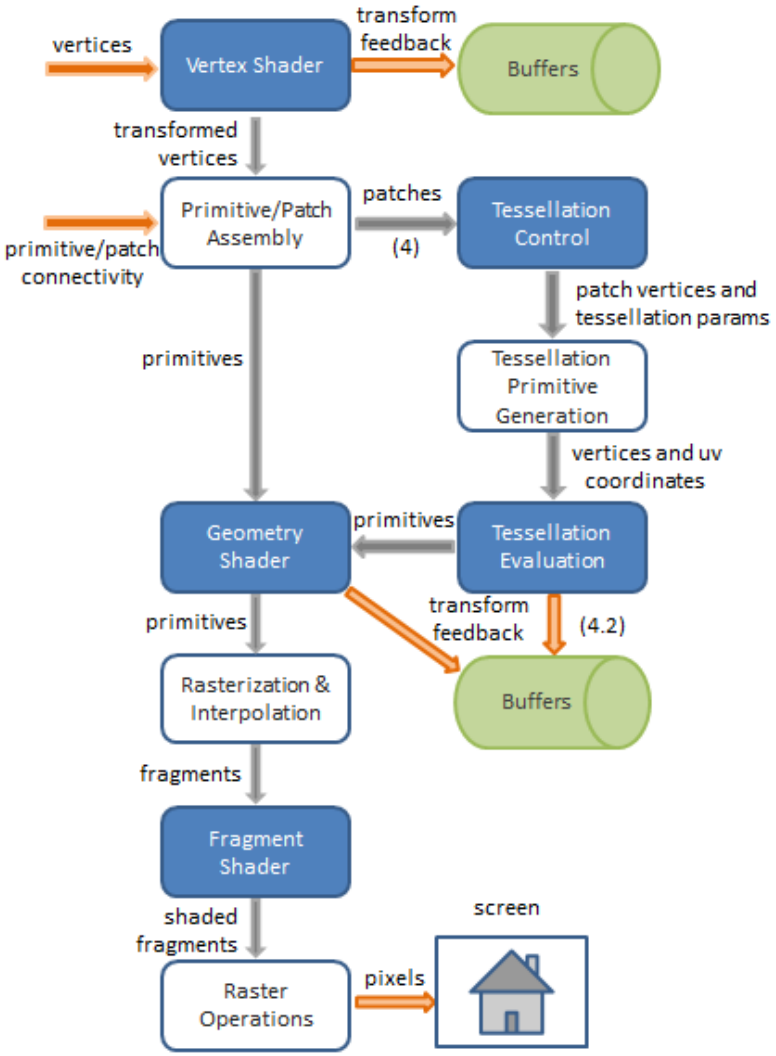


GL_QUADS



GL_QUAD_STRIP

Okay, Back to the Pipeline…

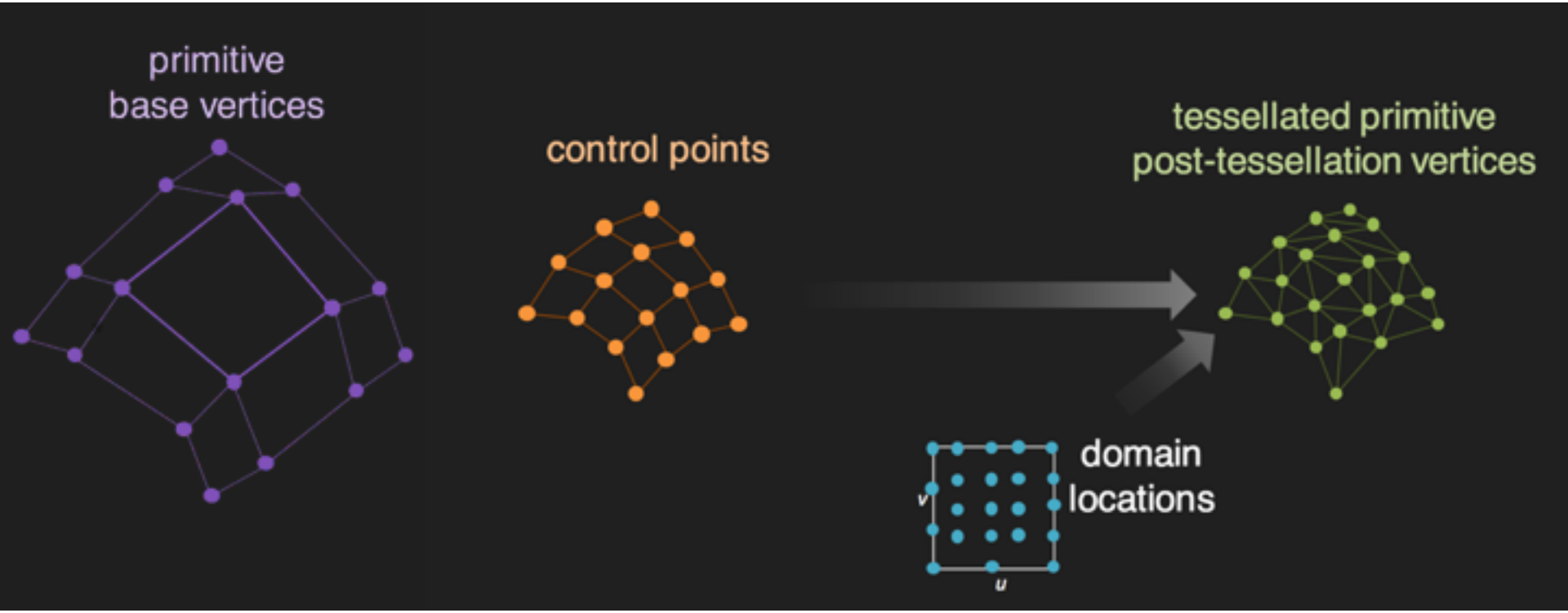


Tessellation Shader

Controls amount of tessellation per patch

- Lower poly models can be subdivided into higher resolution models
- Values calculated for generated vertices
- Optional

Three Parts of Tessellation



Tessellation Uses

- GPU-based subdivision of geometry
  - Can also perform smoothing algorithms
- Level of detail (LOD) controllable within the shader pipeline

Geometry Shader

Takes primitives and outputs multiple primitives

- Not optimized for subdivision (tessellation shader's job)
- Ability to work on entire primitive
- Optional

Geometry Input Primitives

Primitives input:

- points
- lines
- lines_adjacency
- triangles
- triangles_adjacency

Input type set in `layout`

Geometry Output Primitives

Zero or more primitives output:

- points
- line_strip
- triangle_strip

`emitVertex()` adds vertex to outputted primitive

`EndPrimitive()` generates primitive

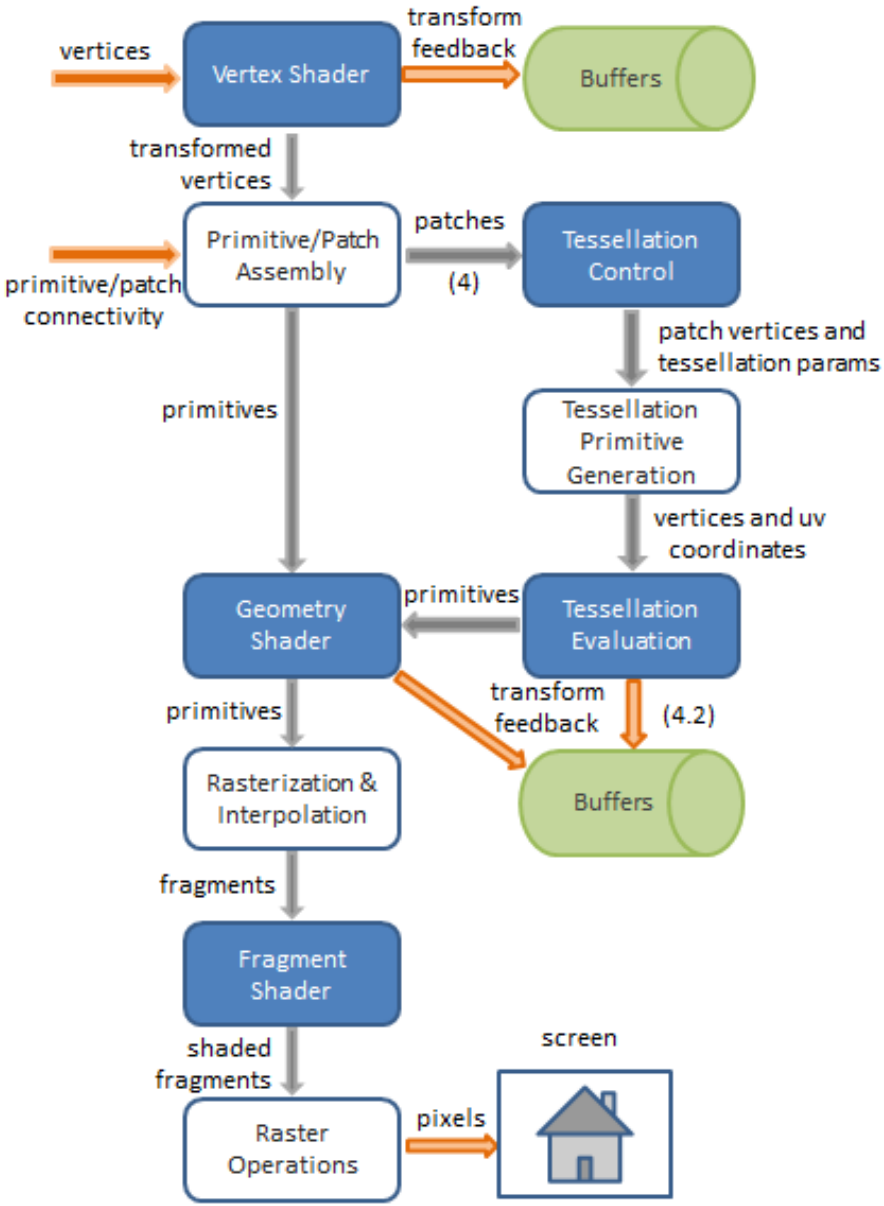
Geometry Shader Uses

Shader can be invoked multiple times for multiple passes within geometry shader

- Layered rendering (dynamic cubemaps, etc)

Shader can generate a lot of primitive data output (and variety) from limited input

- Reduces CPU to GPU bandwidth

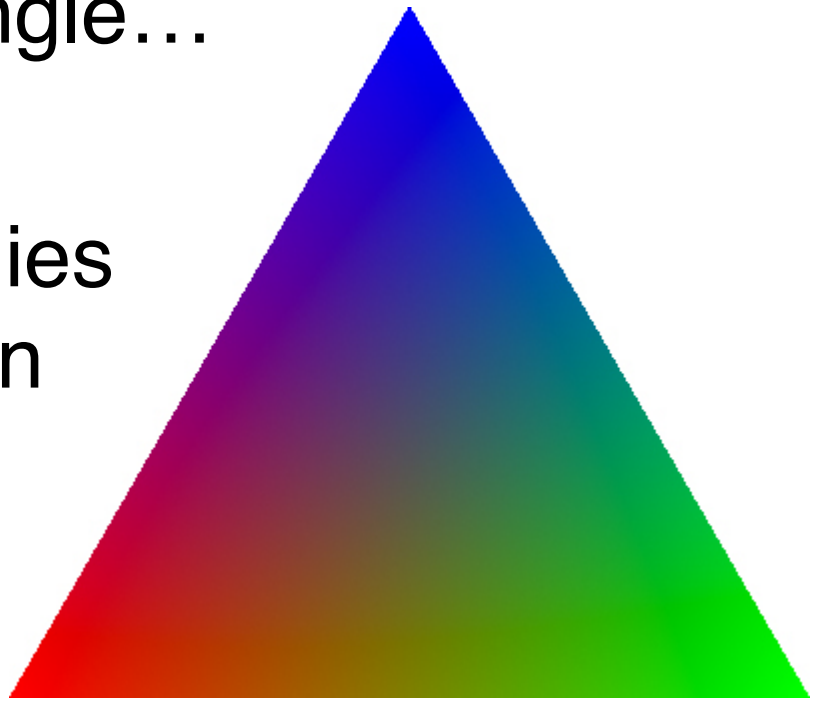

vertices
Vertex Shader
transform feedback
Buffers
transformed vertices
primitive/patch connectivity
Primitive/Patch Assembly
patches
(4)
Tessellation Control
patch vertices and tessellation params
primitives
Tessellation Primitive Generation
vertices and uv coordinates
Geometry Shader
primitives
Tessellation Evaluation
transform feedback
(4.2)
primitives
Rasterization & Interpolation
Buffers
fragments
Fragment Shader
shaded fragments
screen
Raster Operations
pixels

Fragment Shader

Many fragments per triangle…

GPU **automatically** applies barycentric interpolation

UV coords, normals, colors, …



Fragment Shader

Runs in parallel on each fragment (pixel)

- rasterization: one triangle -> many fragments

Writes color and depth for one pixel (only)

Final texturing/coloring of the pixels

Fragment Shader Example

```
out vec4 frag_color;
void main() {
  frag_color = vec4(1.0, 0.0,
    0.0, 1.0);
}
```

Fragment Shader Uses

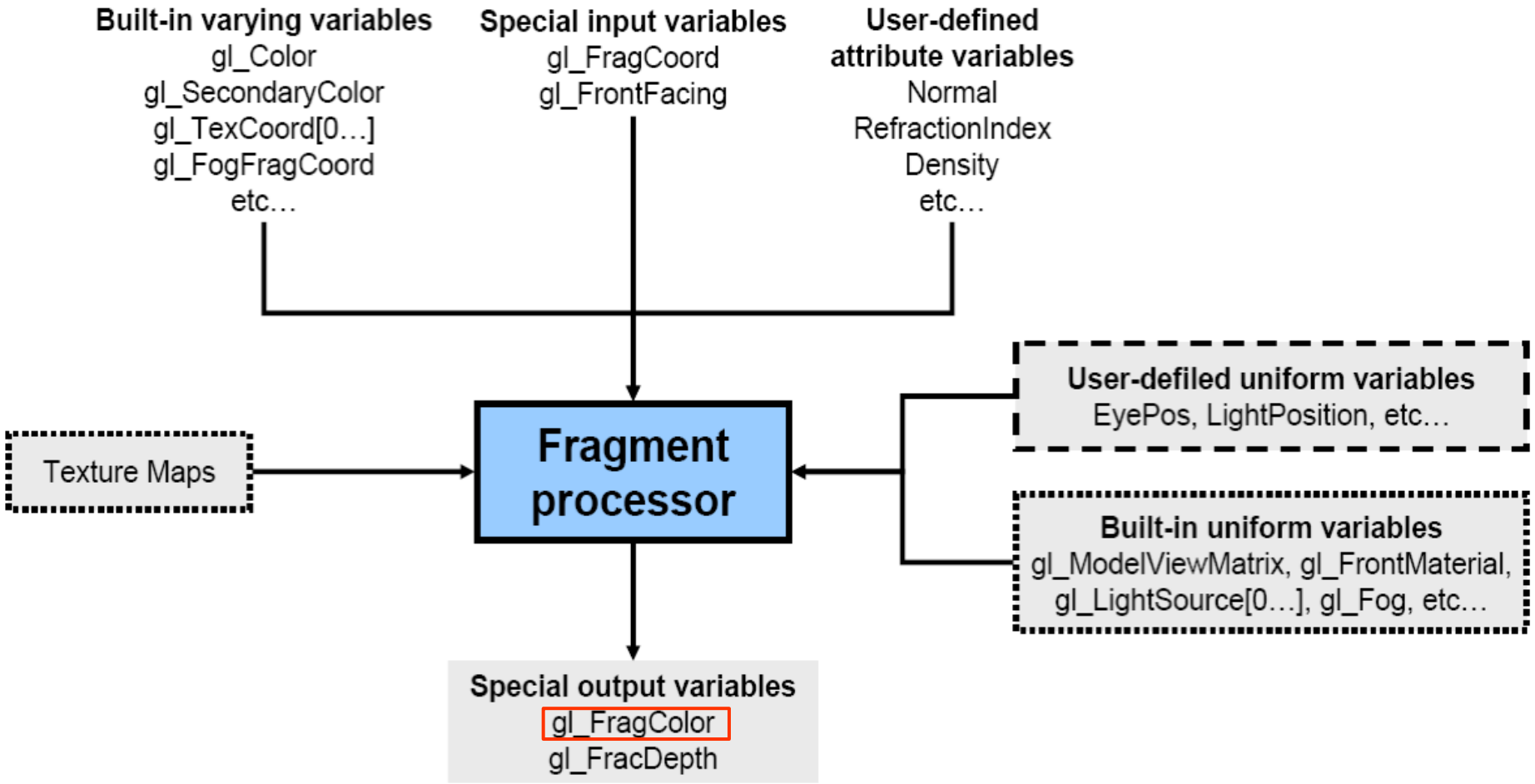
Per-fragment lighting

- Compute lighting on each fragment rather than each vertex

Bump-mapping

Environment-mapping

Fragment Shader In and Outs



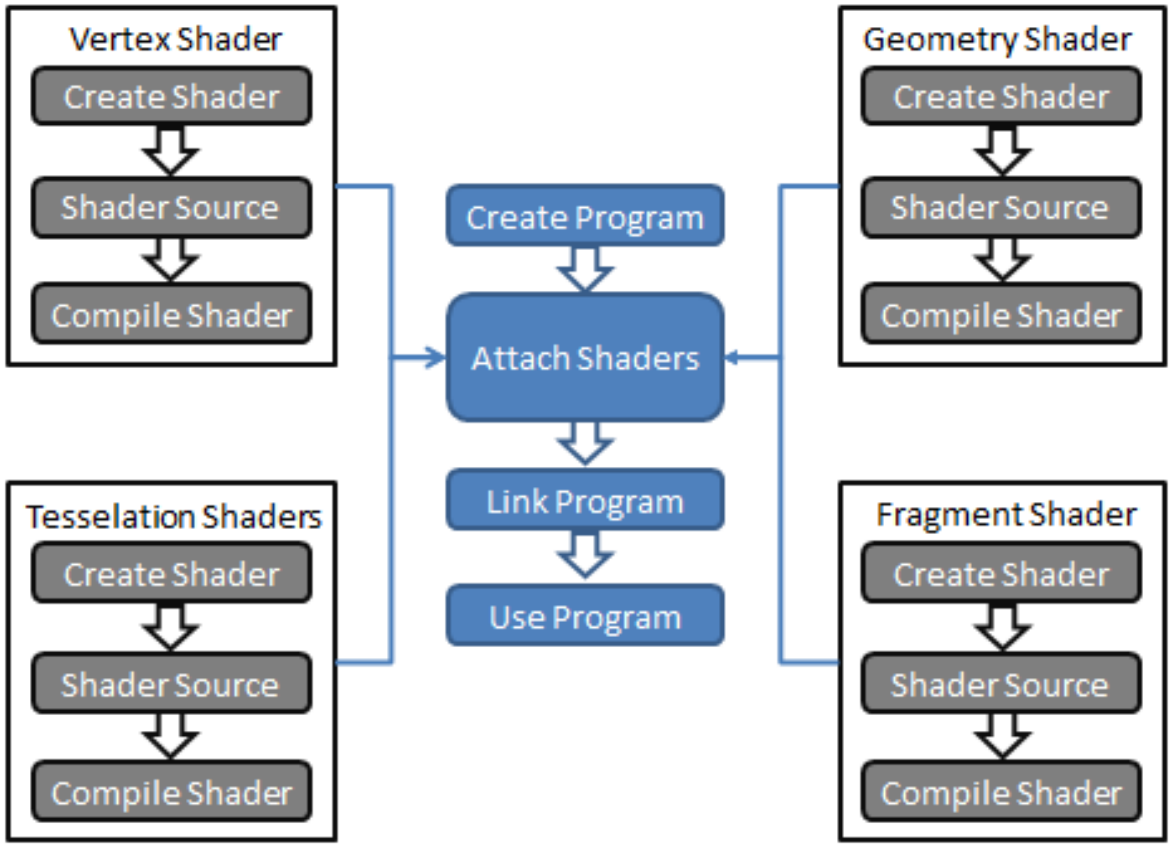
Open GL Tutorial

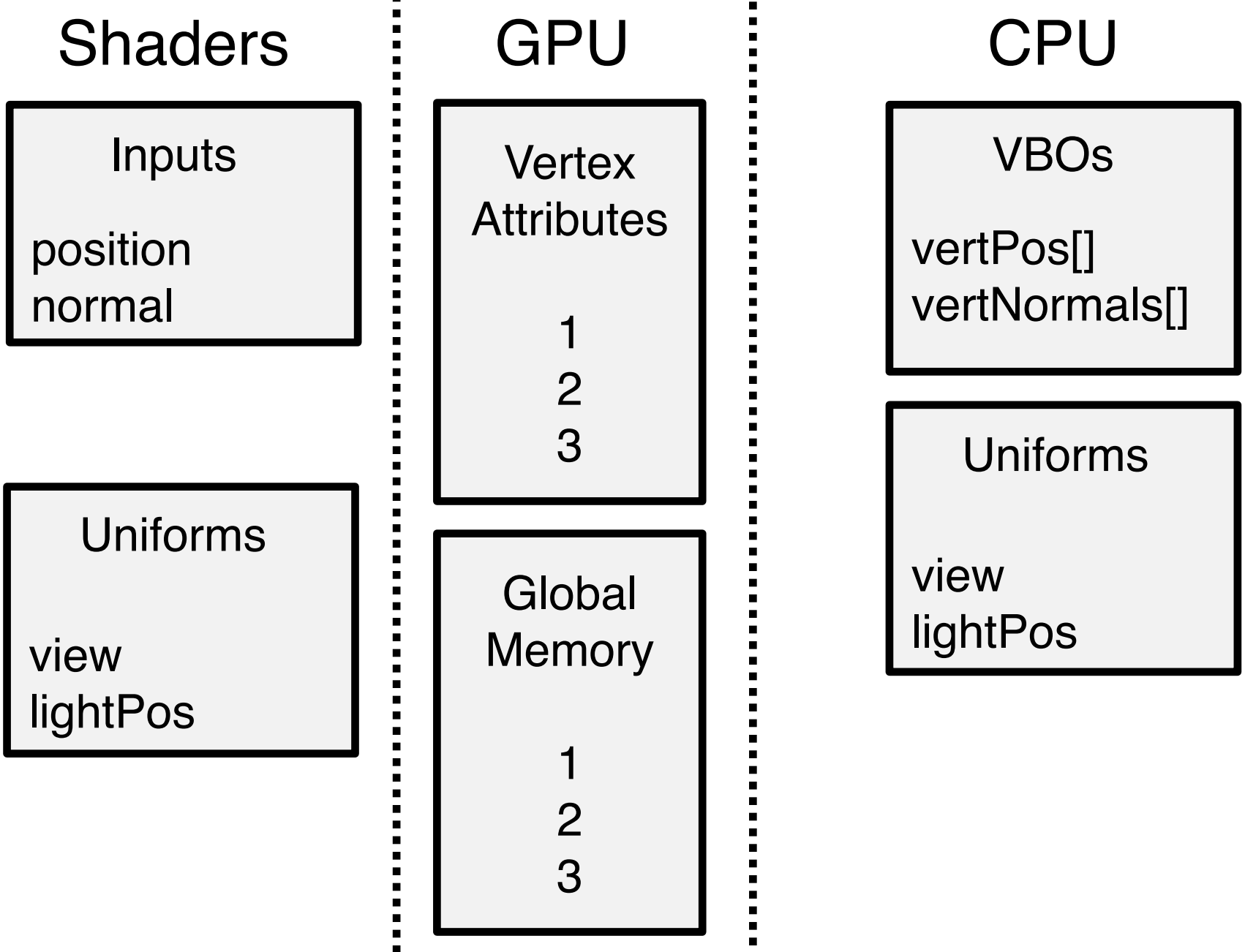
http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/

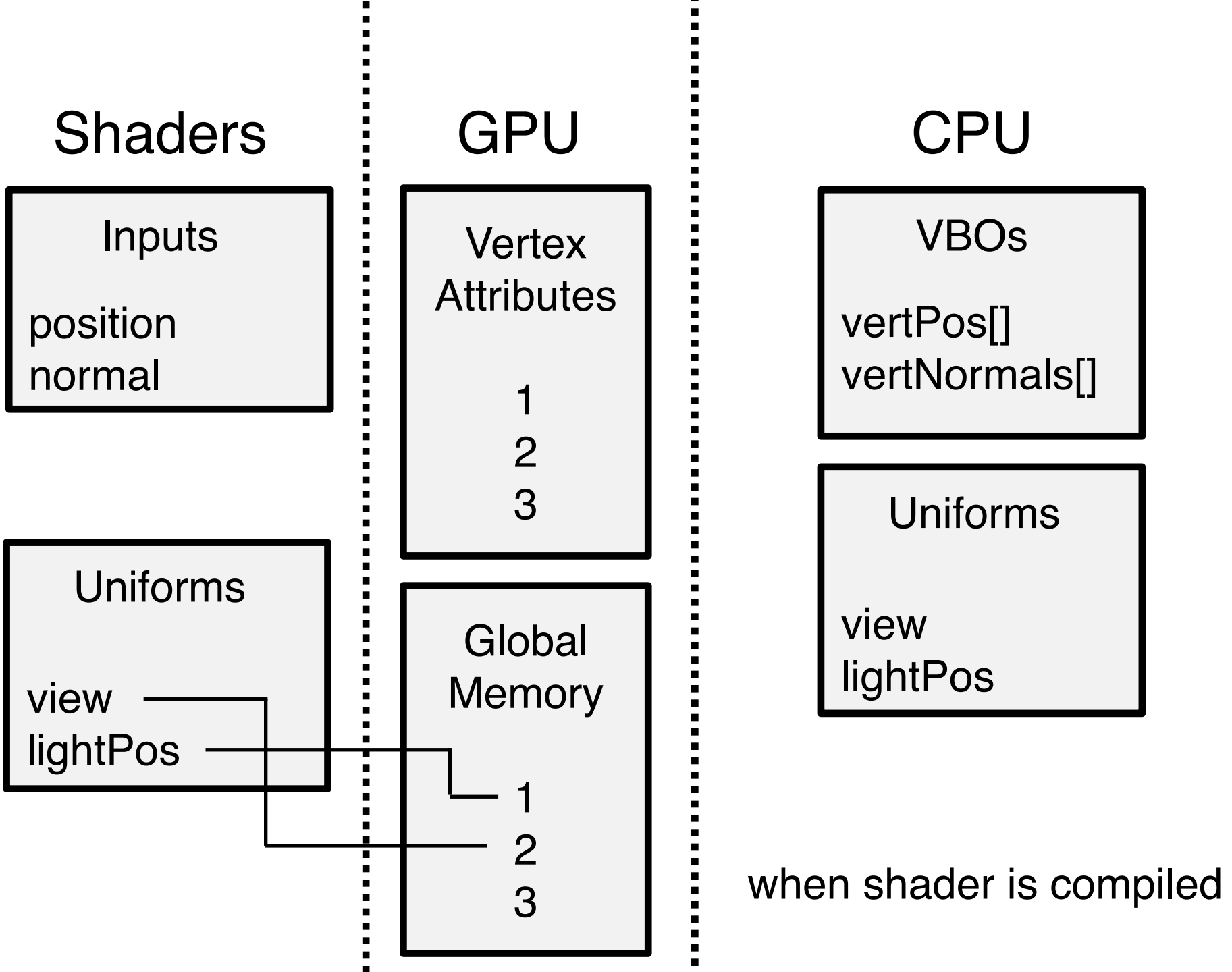
http://www.opengl-tutorial.org/beginners-tutorials/tutorial-4-a-colored-cube/

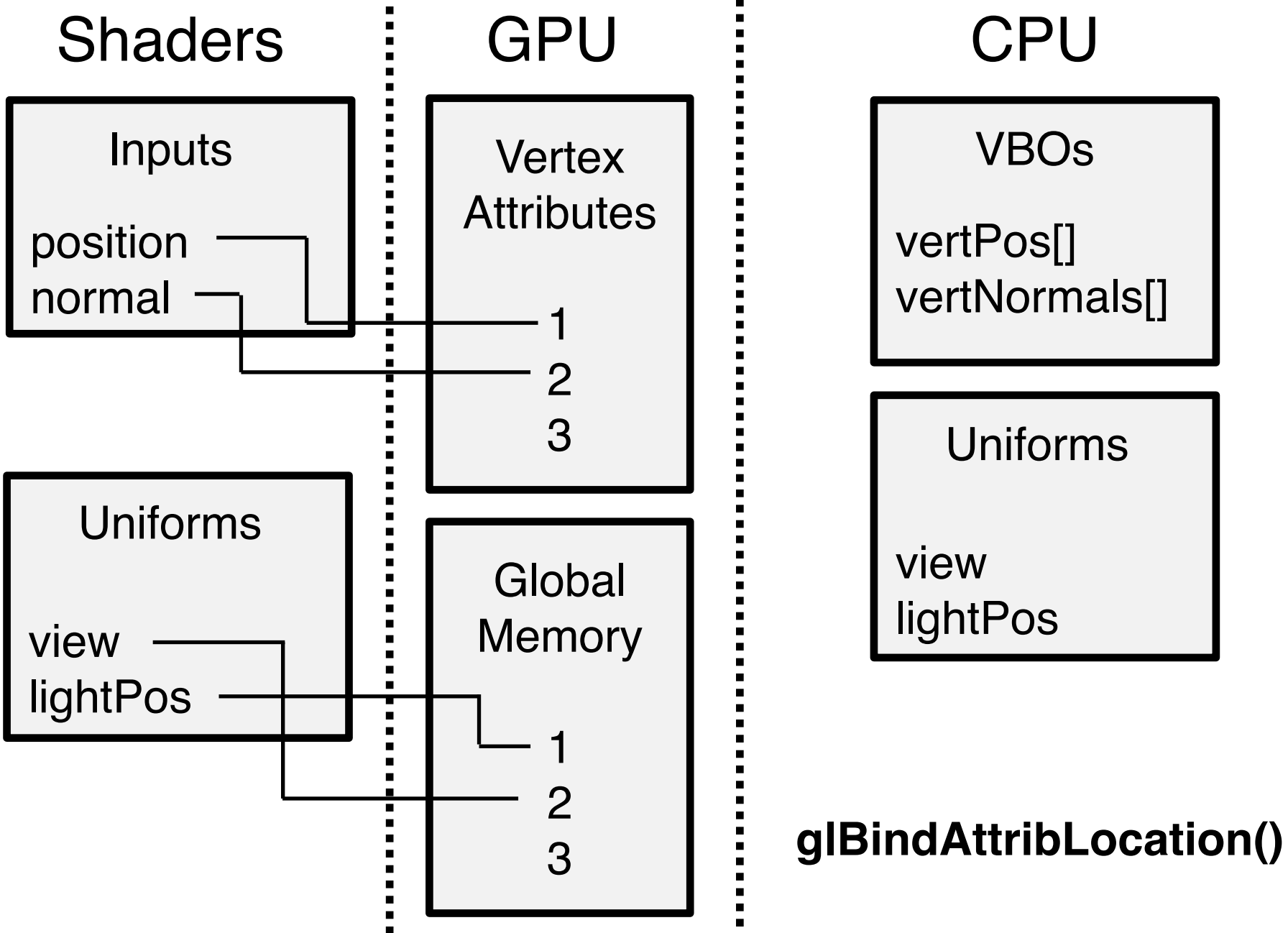
Using Shaders

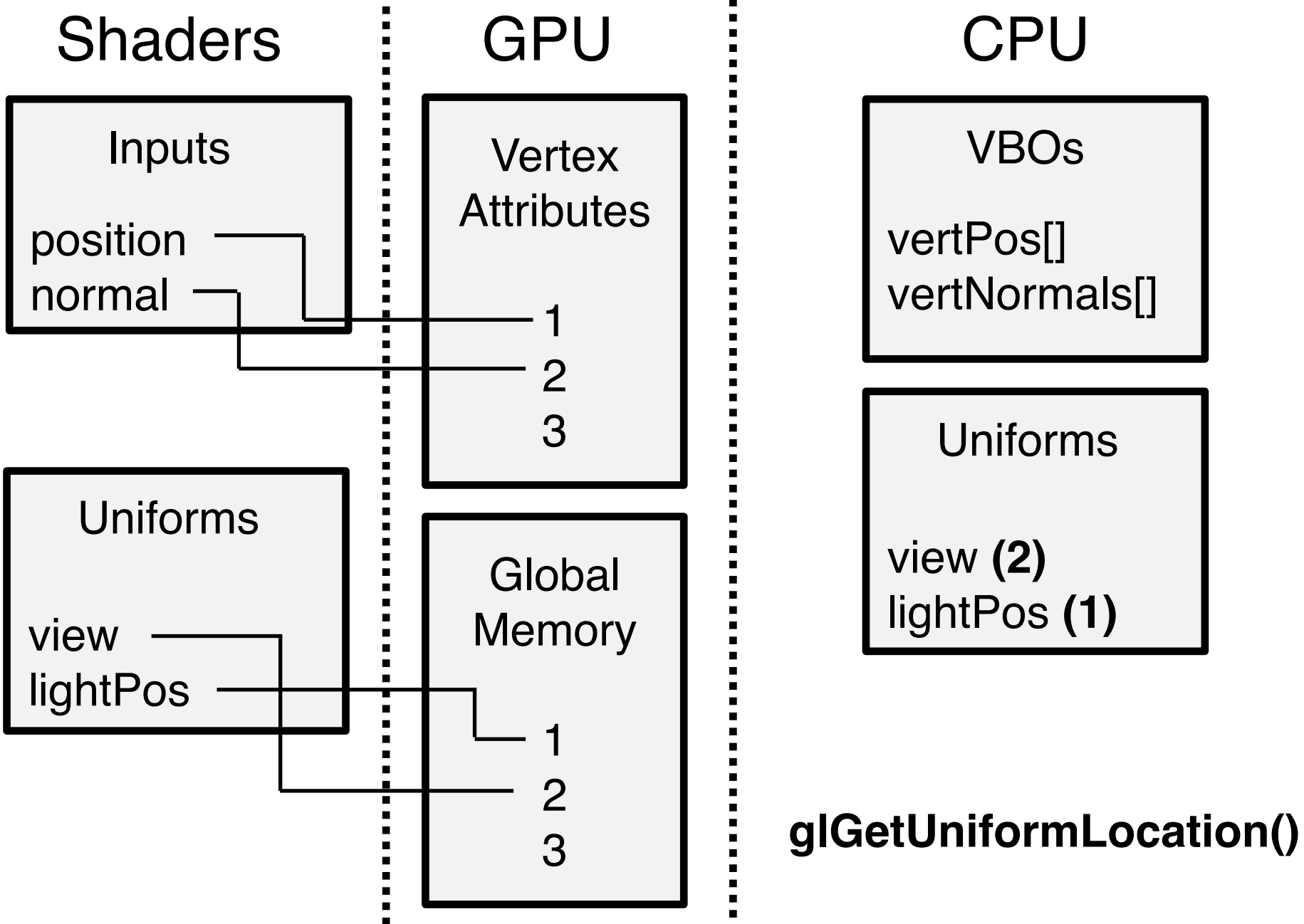
Must compile and link shaders to make executable:

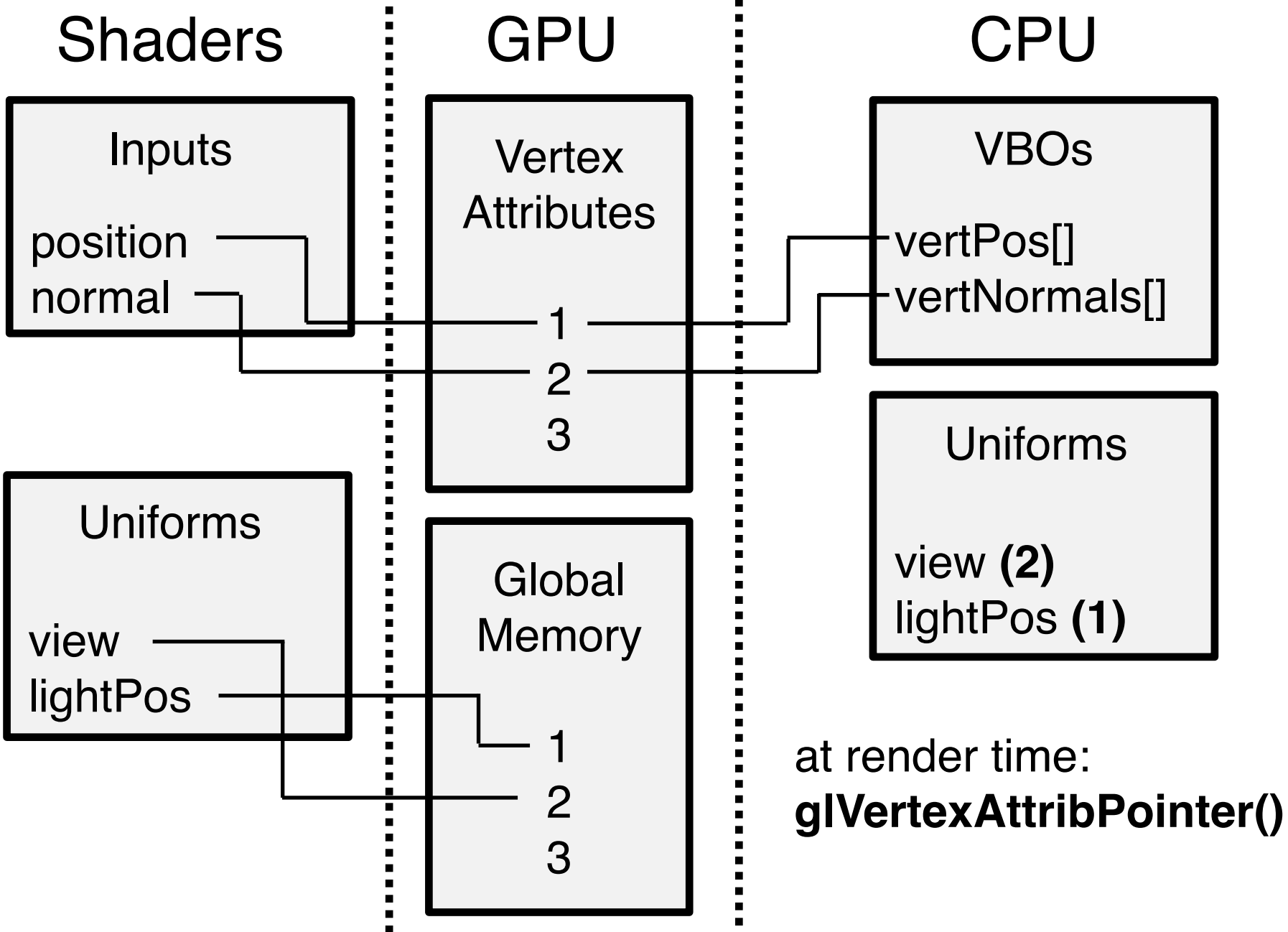


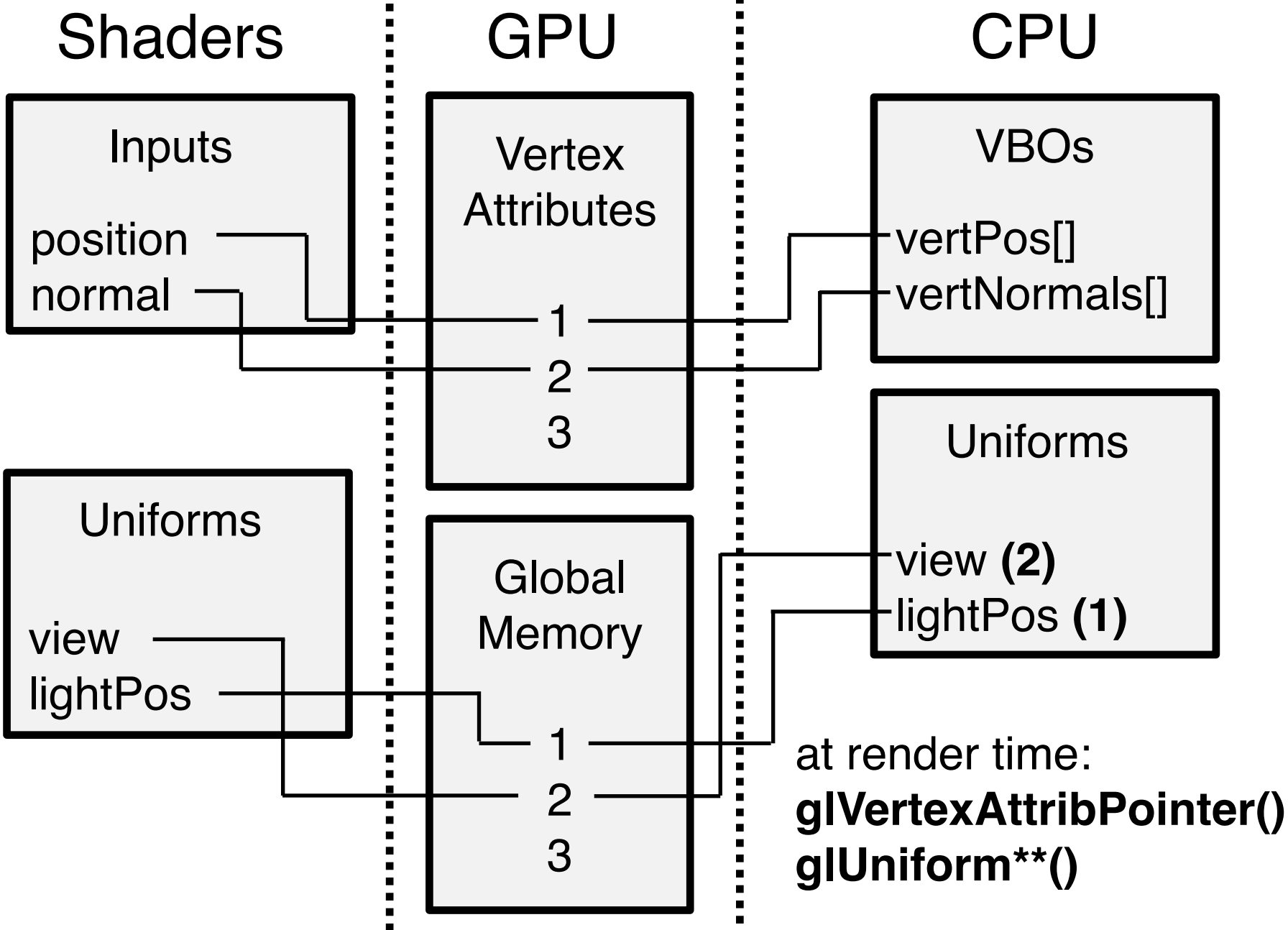

Shaders
GPU
CPU
Inputs
position
normal
Uniforms
view
lightPos
Vertex
Attributes
1
2
3
Global
Memory
1
2
3
VBOs
vertPos[]
vertNormals[]
Uniforms
view
lightPos

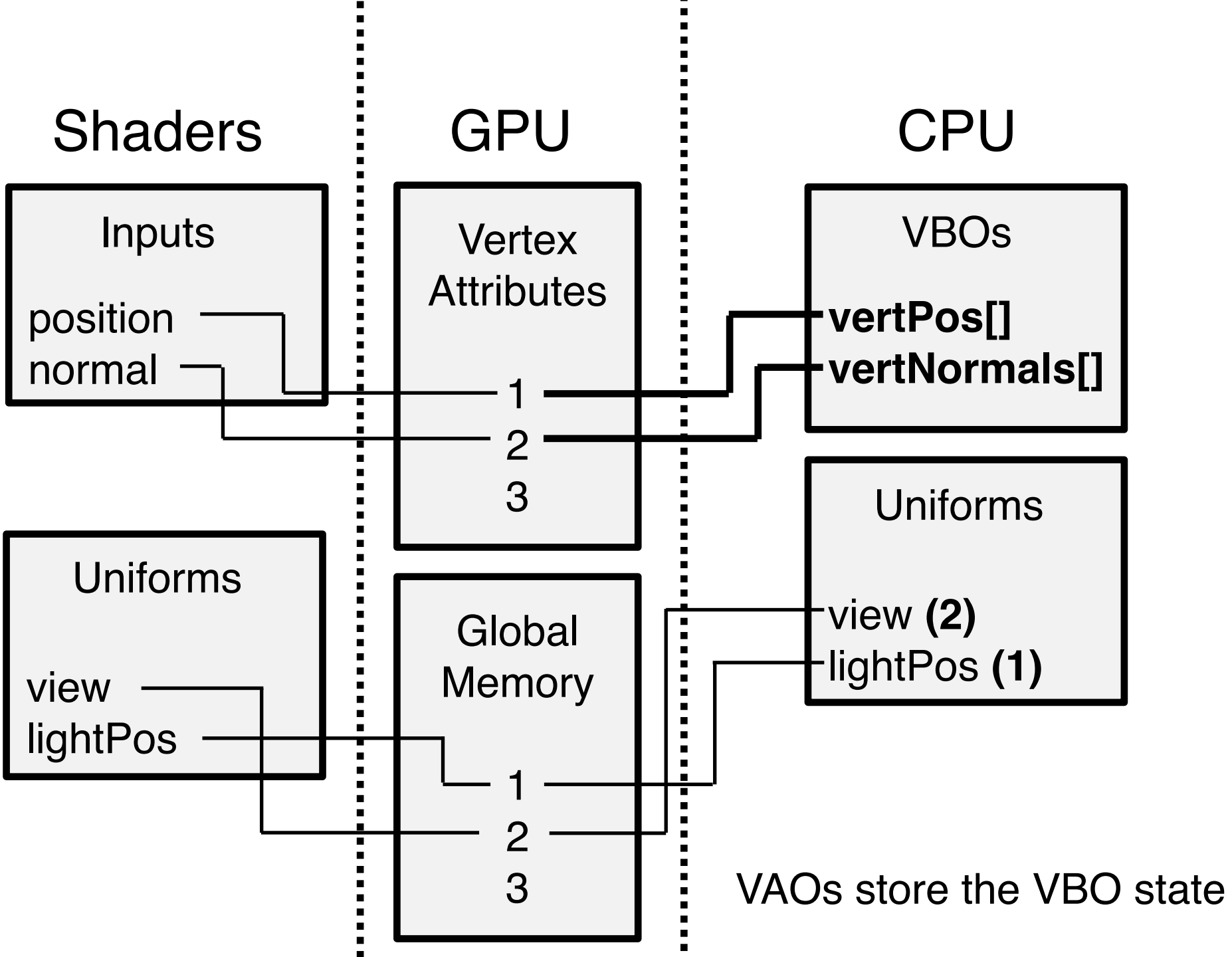

Shaders
GPU
CPU
Inputs
position
normal
Vertex Attributes
1
2
3
VBOs
vertPos[]
vertNormals[]
Uniforms
view
lightPos
Uniforms
view
lightPos
Global Memory
1
2
3
when shader is compiled


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

Switching VAOs, Shaders, VBOs

Possible to switch between VAOs, shaders and VBOs at any time

1. Set up all resources at beginning of program
2. Bind or unbind as necessary within the main loop (batch as much as possible)
3. Only update uniforms if values have changed

GLEW

OpenGL Extension Wrangler

Library for loading pointers at runtime into OpenGL core functions and extensions

Should work on all platforms

GLFW

OpenGL Framework

Handles windows, contexts, and receives input and events

Should work on all platforms

Textures

Create and bind texture within application using `glGenTextures` and `glBindTexture`

Take UV coordinates:

```
in vec2 uvcoords;
```

Return associated RGBA value:

```
texture(sampler, uvcoords);
```

Samplers

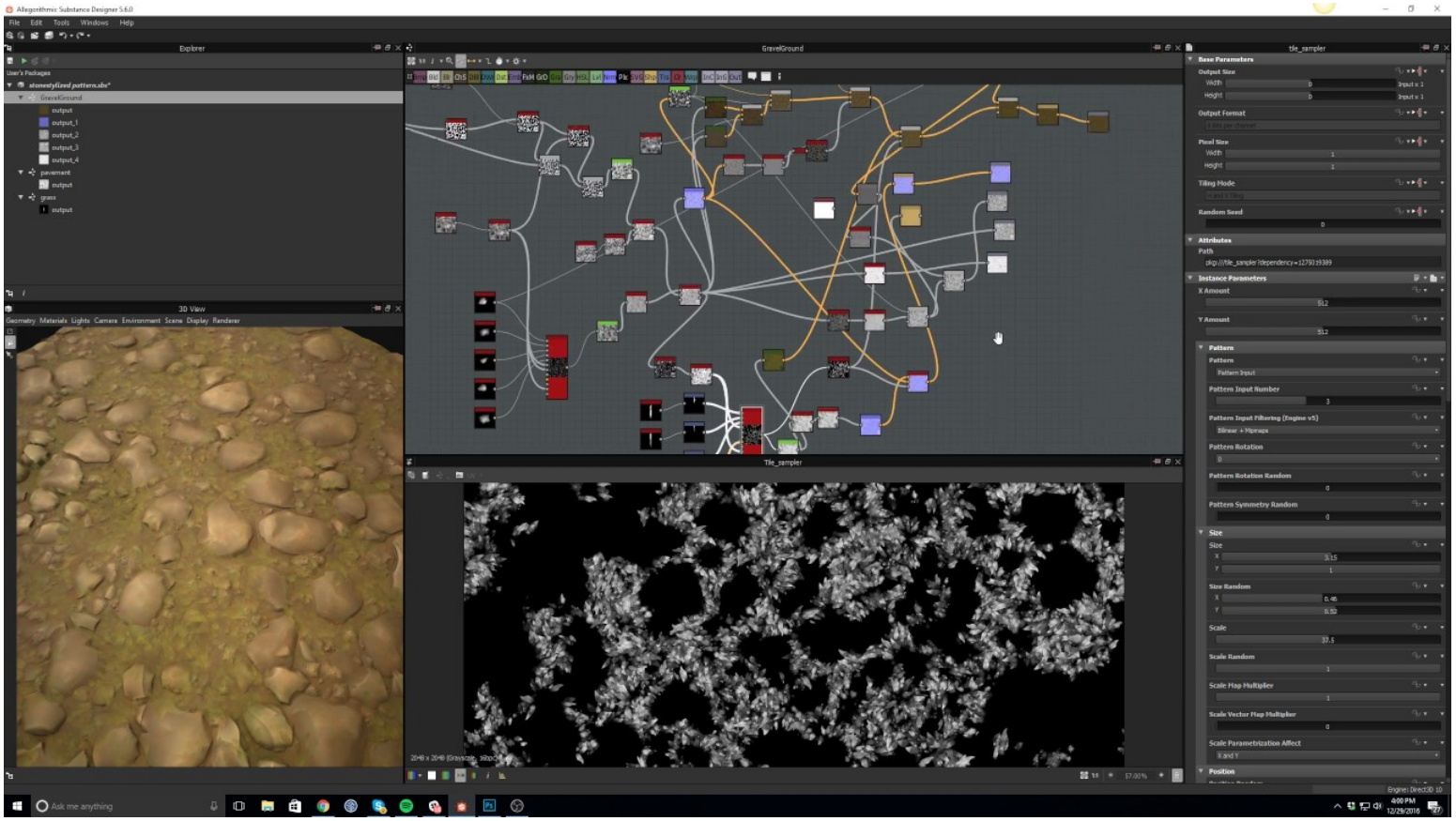
Allows reading from a particular texture and fetching texels

Textures have type depending on purpose:

- sampler1D, sampler2D, samplerCube, etc

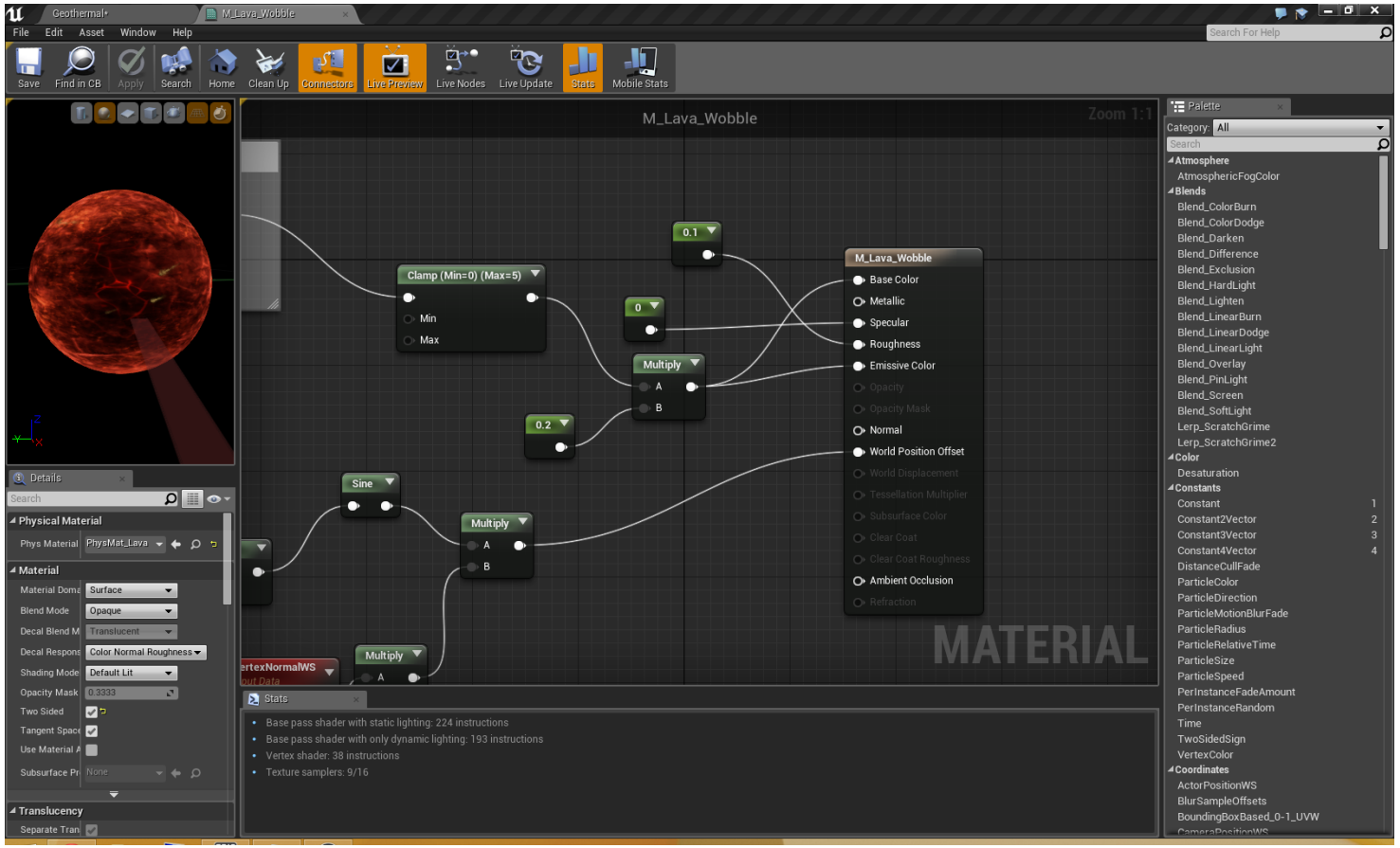
Sampling parameters determine how texture is accessed

Shaders in Art Pipelines



Substance Designer

Shaders in Art Pipelines



Unreal Engine 4 Materials

Additional Resources

http://www.opengl-tutorial.org/beginners-tutorials/tutorial-5-a-textured-cube/

https://www.shadertoy.com/