Intro to Shaders

Last Time

Processors generally have challenges dealing with two big issues:

- Data Dependence
- Branching
  - Loops
  - If-else
  - (Other stuff, as it turns out!)

CPUs solve this problem by investing a lot of time and performance into branch and data management.

GPUs solve this problem by being used for graphics applications, which rarely have to deal with these problems.

How do we program GPUs?

Programming Framework	**Used By**
OpenGL/Vulkan	Graphics Programmers
DirectX	Windows Graphics Programmers
CUDA/HIP	Machine Learning/Scientific Programmers
Writing raw PTX/SPIR-V	Crazy people

What's wrong with Vulkan?

Nothing. Vulkan is a really good API which opens the door for all sorts of interesting control over the GPU. DOOM 2016 saw between 5% and 35% improvement in performance when switching from OpenGL to Vulkan.

However, it's also incredibly complex, fiddly, and hard to get right, even for a graphics framework. That's quite a high bar to meet.

Is there a processing unit that's good at both CPU and GPU tasks?

You could probably make one for $100k a pop.

If the GPU is only good at parallel, non-dependent workloads, do I have to know which processor my code is better for before I deploy it?

Yes. "You" (or your runtime) have to choose which processor your code runs on, which means that you need to know what's better at what.

This is not a choice you have to make in Processing, since Processing splits the workload appropriately for you. If you're not sure in other languages, just choose the CPU.

Why not use P2D by default?

P2D has known issues having to do with image quality and missing features.

How will the project presentation be graded?

Based on the presentation you give. You'll need to hit all the points in the presentation requirement (in particular, giving enough of an overview of what you did that someone who didn't do your project can understand what you did).

If you upload materials, you can present using my laptop. You can also choose to bring your own laptop and present on that.

Tradeoff of risks: you can't test your code on my laptop (if you want to do a live demo) but you should check your laptop on the room's AV system if you want to use your own.

How does code get turned into zeros and ones?

The instructions we examined last class can actually be turned into binary instructions, almost in a 1:1 fashion.

```
1  X1 = ADD X1 X2
2  X2 = SUB X2 X3
3  X3 = MUL X3 X4
4  X4 = DIV X4 X5
5
6  X5 = LOAD [myObj]
7  STORE X5 TO [myObj]
8
9  BRANCH (X4 == 0) 4
10 X7 = CALL square(X2)
```

These instructions can be encoded by packing smaller numbers into a bigger number, in a similar manner to how `colors` work in Processing.

```
1  X1 = ADD X1 X2
2  X2 = SUB X2 X3
3  X3 = MUL X3 X4
4  X4 = DIV X4 X5
```

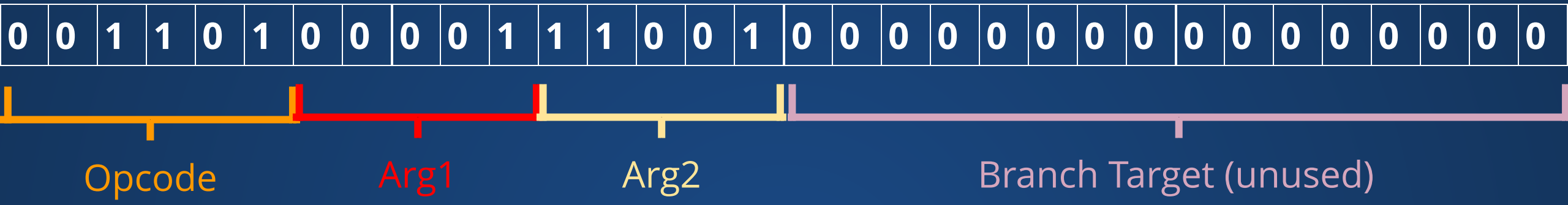
Operation	Opcode
ADD with 2 variables	0
ADD with 1 variable, 1 constant	1
ADD with 2 constants	2
SUB with 2 variables	3
SUB with 1 variable, 1 constant	4
etc....	...

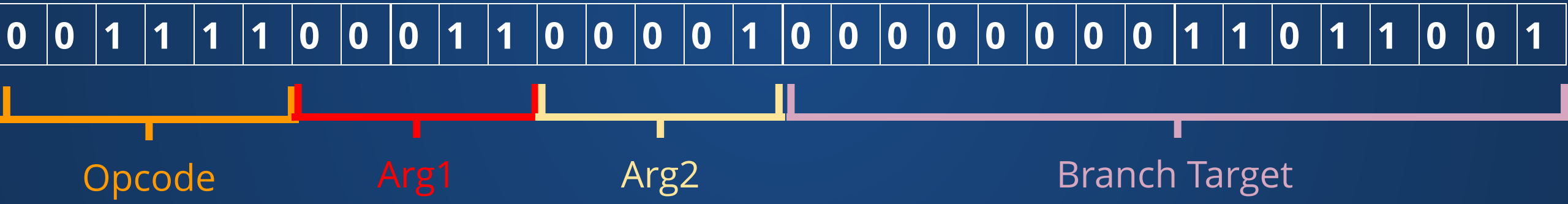
Instruction	Specification Tuple
ADD X1, X1	0, 1, 1, 0
ADD X1, 10	1, 1, 10, 0
SUB X3, X1	3, 3, 1, 0
ADD 0, 0	2, 0, 0, 0
SUB X3, 1	4, 3, 1, 0

Operation	Opcode
LOAD	13
STORE	14
BRANCH with 2 registers, less than	15
BRANCH with 2 registers, equal to	16
BRANCH with 1 register and 1 constant, equal to	17
etc...	...

Instruction	Specification Tuple
X1 = LOAD [25]	13, 1, 25, 0
STORE X32 to [117]	14, 32, 117, 0
BRANCH (X3 < X7), 217	15, 3, 1, 217
BRANCH (X24 == 17), 2023	17, 24, 17, 2023

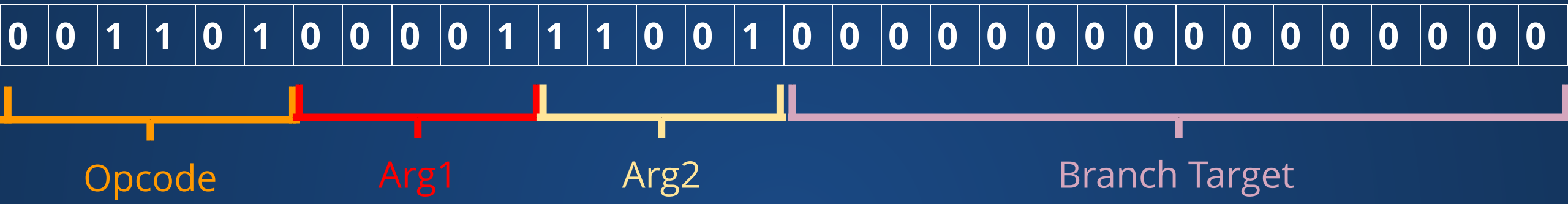
We can now take these instruction encodings and turn them into binary!





Instruction	Specification Triple
X1 = LOAD [25]	13, 1, 25, 0
BRANCH (X3 < X7), 217	15, 3, 1, 217

We can now take these instruction encodings and turn them into binary!



We used 32 bits for each instruction here, so this would be a 32-bit processor.

Similar processes happen for GPU code.

OpenGL

An open graphics programming system supported by almost all major vendors.

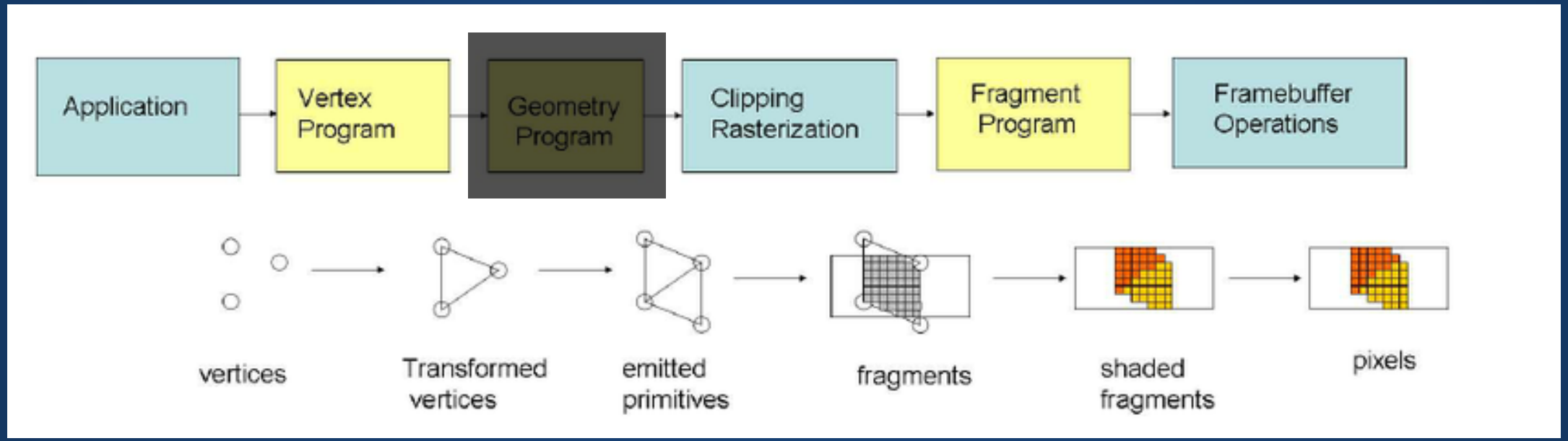
Provides a *programmable pipeline* for graphics applications.

Requires support from multiple parties:

- Hardware needs to support the draw commands used
- Operating system needs to provide the appropriate libraries, software, and interfaces to support OpenGL usage.
- Application needs to provide the programs to run.

The Graphics Pipeline

Some of the GPU's functions are fixed in the hardware, and there's nothing we can do to change them.



Shaders

What are Shaders?

Shaders are programs that are run on the GPU.

Written in a special *shading language*. In our case, this will be the OpenGL Shading Language (GLSL), but can also be others, e.g. HLSL, RenderManSL, etc.

GLSL

A language which is most similar to C (but pretty similar to Java).

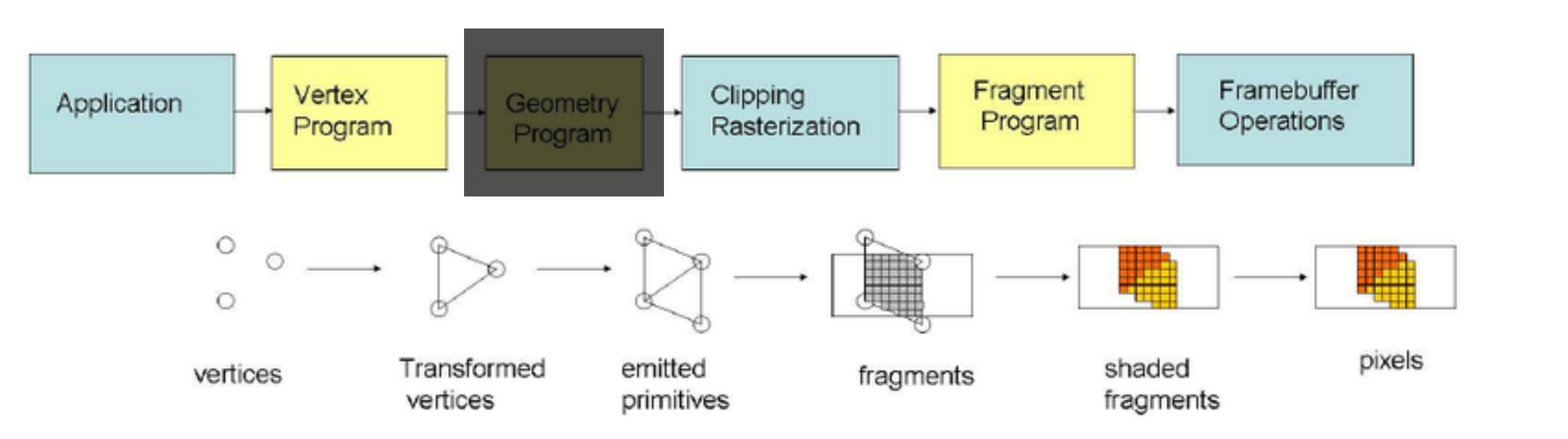
Because it runs on the GPU, don't have access to things like print, or the ability to read files.

A few oddities like swizzling and `varying`/`uniform` variables.

```
1  varying vec4 thing;
2  uniform mat4 matt;
3
4  void main(){
5    vec3 v1 = vec3(1.0, 2.0, 3.0);
6    vec4 v2;
7
8    // These are not members!
9    v2.xyz = v1.xxy;
10   v2.w = v1.x;
11 }
```

Vertex Shader

Fragment Shader



Vertex Shader

Takes in vertices and applies an operation to every vertex.

Examples:

- Transform each vertex to a new position
- Compute lighting/color data for each vertex
- Can displace vertices to make a bumpy/textured surface (though in classic OpenGL, this is more commonly done in the tessellation shader).

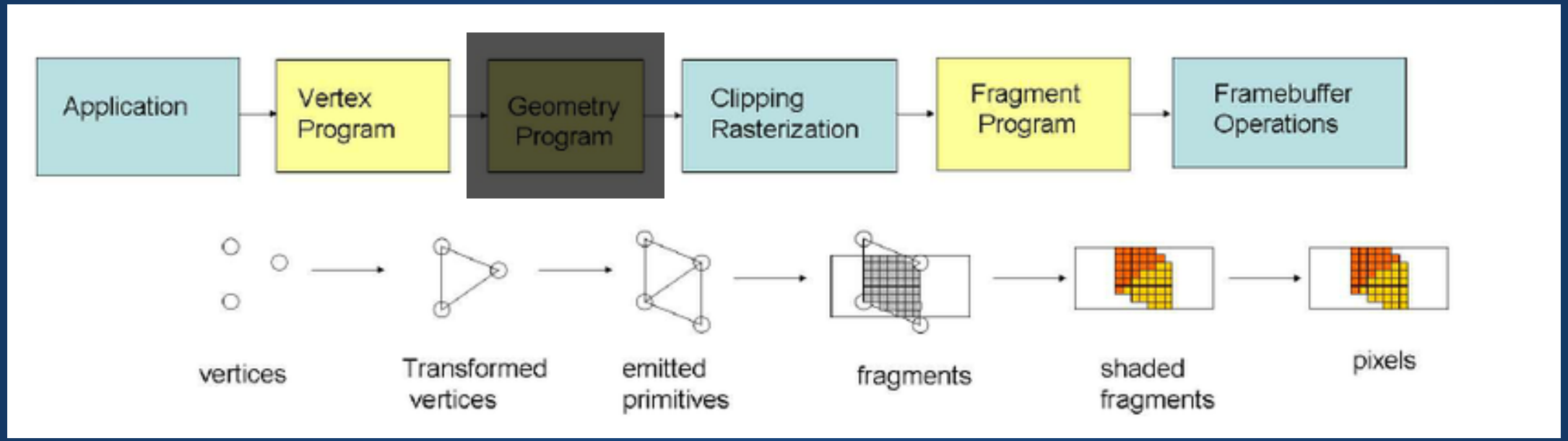
Vertex Shader

```
1 uniform mat4 transform;
2 attribute vec4 position;
3 void main() {
4   //Must set gl_Position in vertex shader
5   //for each vertex processed
6   gl_Position = transform * position;
7 }
```

Shaders run `main()` for each input they get! Inputs are passed in by shader-global variables.

For the vertex shader:

- `uniform` variables are the same across all inputs.
- `attribute` variables may change per-input, but are readonly.
- `varying` variables may change per-input and are writeable.


Application
Vertex Program
Geometry Program
Clipping Rasterization
Fragment Program
Framebuffer Operations
vertices
Transformed vertices
emitted primitives
fragments
shaded fragments
pixels

Fragment Shader

Takes in *fragments* (pieces of shapes that the pipeline has rendered).

Also takes in any outputs the fragment shader may have created.

Outputs the color to use on the fragment.

```
1 void main() {
2   // Color components are in [0.0, 1.0]
3   // instead of [0, 255]
4   gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
5 }
```

Moving Data

In our example vertex shader, we took a transform and position as input, and computed the transformed position of the vertex.

```
1 uniform mat4 transform;
2 attribute vec4 position;
3 void main() {
4   //Must set gl_Position in vertex shader
5   //for each vertex processed
6   gl_Position = transform * position;
7 }
```

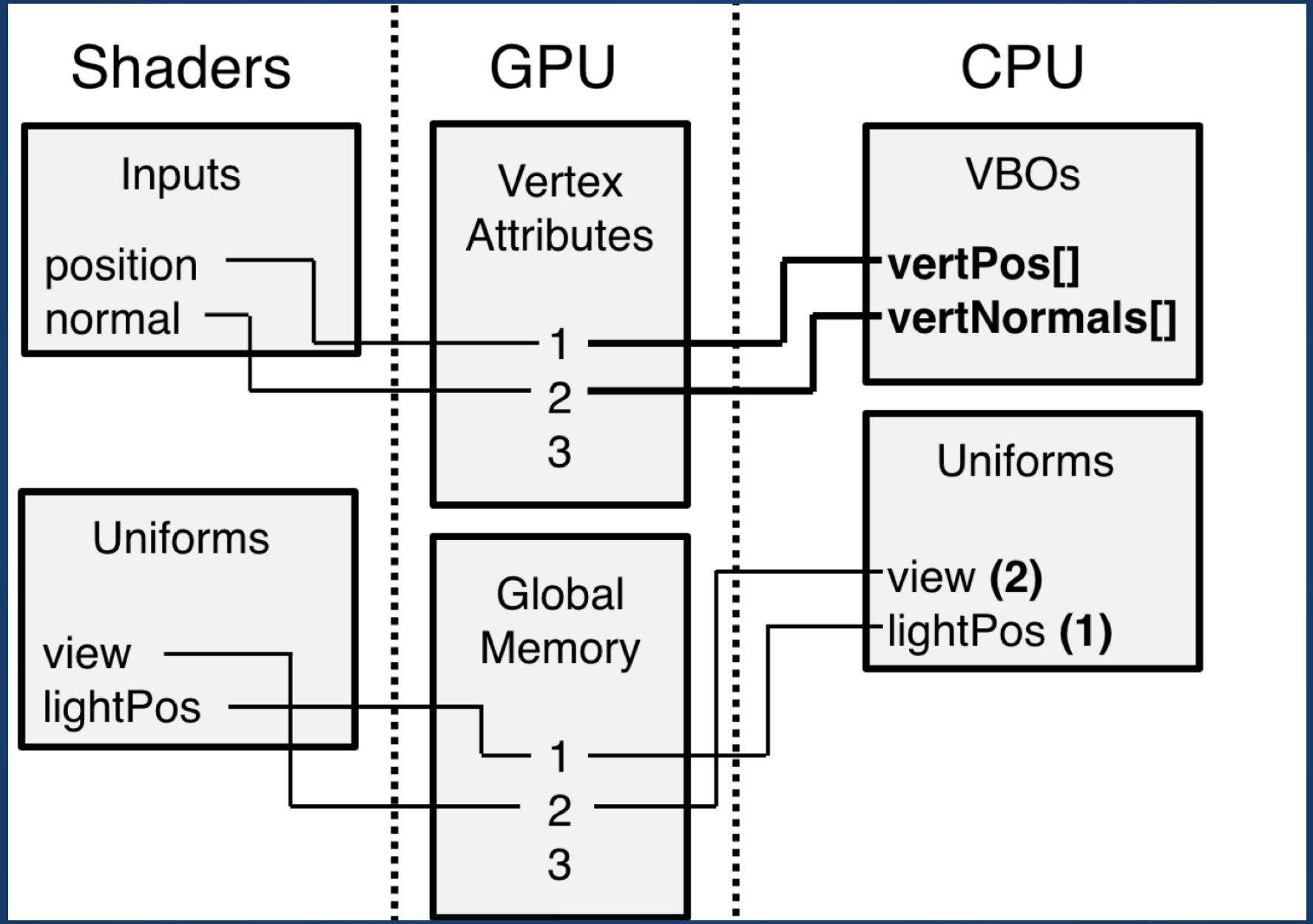
Question: Where does this data come from?

Question: How does this data get there?

Manually moving data between the CPU and GPU is error prone. Let OpenGL handle it instead.

```
vertexPositions = [[1,2,3,1], [1,3,4,1], [1,4,7,1], ...]
transformMat = Mat4.identity()
# When shader runs on vertex 1, vertexPositions[1]
# is availble as the variable "positions"
glBindAttribute("position", vertexPositions)
# transformMat will be availble to all shader invocations
# under the variable name "transform"
glBindUniform("transform", transformMat)
```

```
uniform mat4 transform;
attribute vec4 position;
void main() {
  //Must set gl_Position in vertex shader
  //for each vertex processed
  gl_Position = transform * position;
}
```


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)

Moving Data Between Shaders

Shaders run in a fixed order. In our system, vertex runs first, then fragment.

To pass a variable between shaders, declare it in the first shader and assign a value to it. It will be available in the second shader.

```
1 attribute vec3 position;
2 attribute float size;
3 varying vec3 vert_Output;
4 void main() {
5   //Must set gl_Position in vertex shader
6   //for each vertex processed
7   gl_Position = vec4(position, 1.0);
8   vert_Output = vec3(gl_Position.xyz)
9 }
```

```
1 varying vec3 vert_Output;
2 void main() {
3   // We have access to vert_Output
4   // in the fragment shader!
5 }
```

Overall Setup Steps

1. Compile shader programs (e.g. vertex, fragment, etc.)
2. Link shader programs together to make one big shader
3. Tell the GPU to use our shader program
4. Specify data location on the CPU so that the GPU knows how to find it
5. Set output data location
6. Draw stuff!

Shaders in Processing

Shaders in Processing are Moderately Cursed

- No way to control varying variables fed into the shader
- Have to load vertex and fragment shaders together
- Basically completely undocumented---only way to know what happens is to read the Processing source code.
- Shaders are basically un-debuggable.

For Processing, stick to modifying in-class examples or existing shading code that works.

Processing API

```
PShader blur = loadShader("blur.glsl");
```

Just like most other `load_` functions. First filename is always a fragment shader, pass optional second filename to load a vertex shader as well.

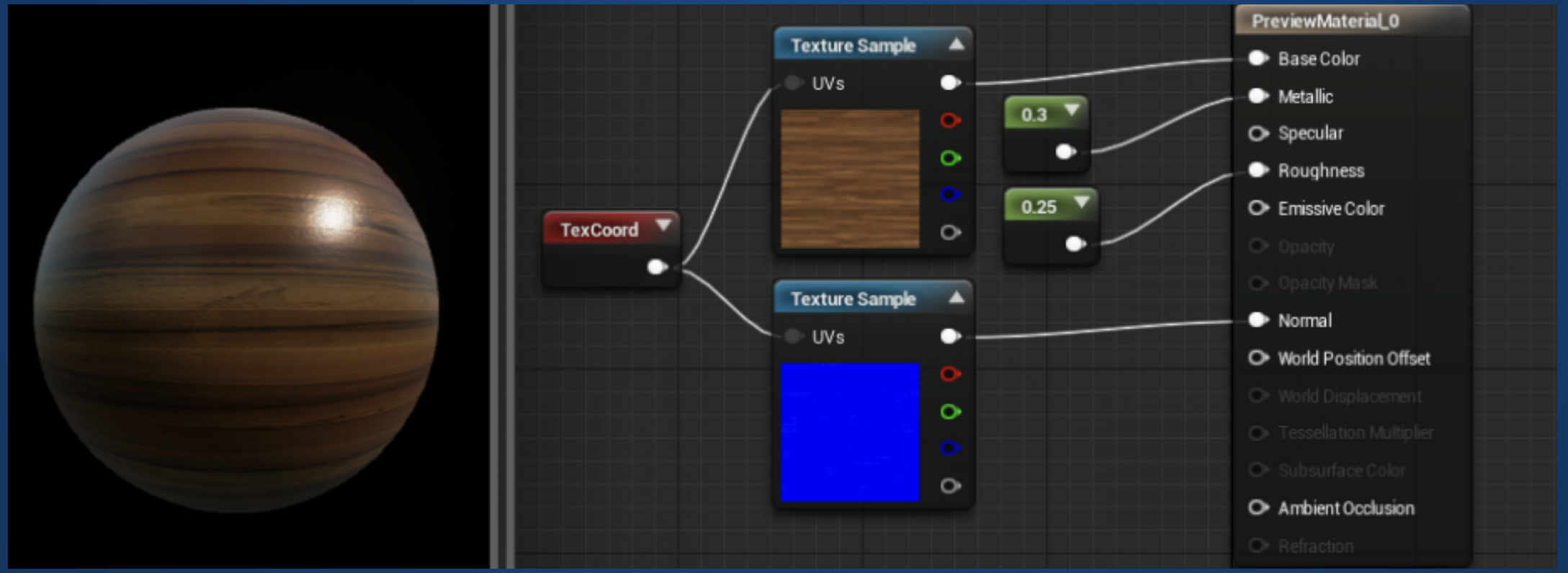
We can use the `.set()` method of PShader to set uniforms within the shader.

Shaders are hard!

Visual scripting languages can make it much easier to program shaders. Example: Substance allows us to program shaders for materials in a visual manner.

https://www.youtube.com/embed/y8q6-tgQjZc?enablejsapi=1

Engines like Unreal 4 also allow us to visually script shaders, along with providing things like previews of the material results



https://www.youtube.com/embed/TEmsqez2YQI?enablejsapi=1

Demo: Shaders

Hands-On: Shaders

1. Download the hands-on skeleton from Canvas. This contains enough skeleton code to do a very basic rendering of the famous Utah teapot.
2. Modify the code so that the ambient light and directional light work, by following the hints in the shader files.
3. OPTIONAL: Try adding another directional light, either with the same color or a different color.

Index Cards!

1. Your name and EID.

2. One thing that you learned from class today. You are allowed to say "nothing" if you didn't learn anything.

3. One question you have about something covered in class today. You *may not* respond "nothing".

4. (Optional) Any other comments/questions/thoughts about today's class.