Intermediate Graphics & Animation Programming

GPR-300
Daniel S. Buckstein

Intro to **GLSL**: The Open**GL** **S**hading **L**anguage
Weeks 1 – 2

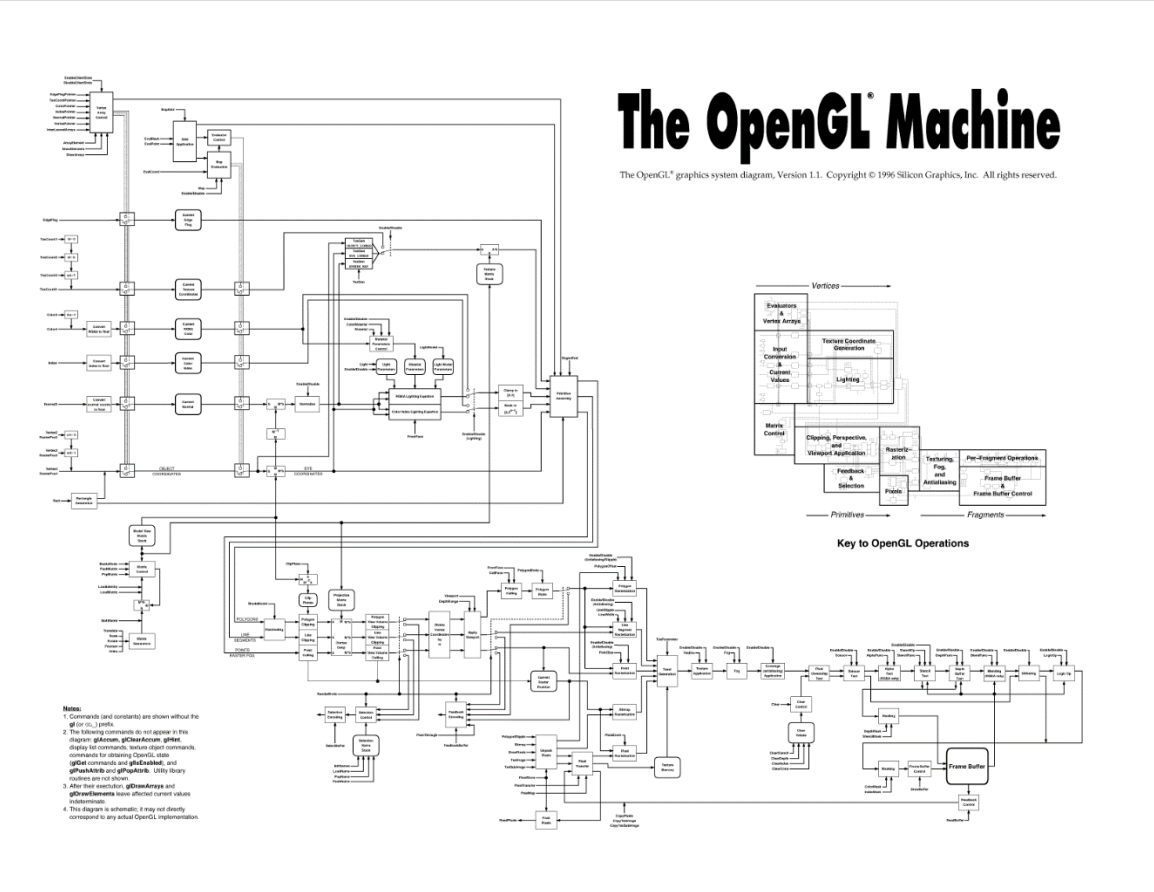
License

Modern Rendering

- Fixed-function vs. programmable pipeline
- Types of shaders, examples
- Key shader terminology
- Food for thought

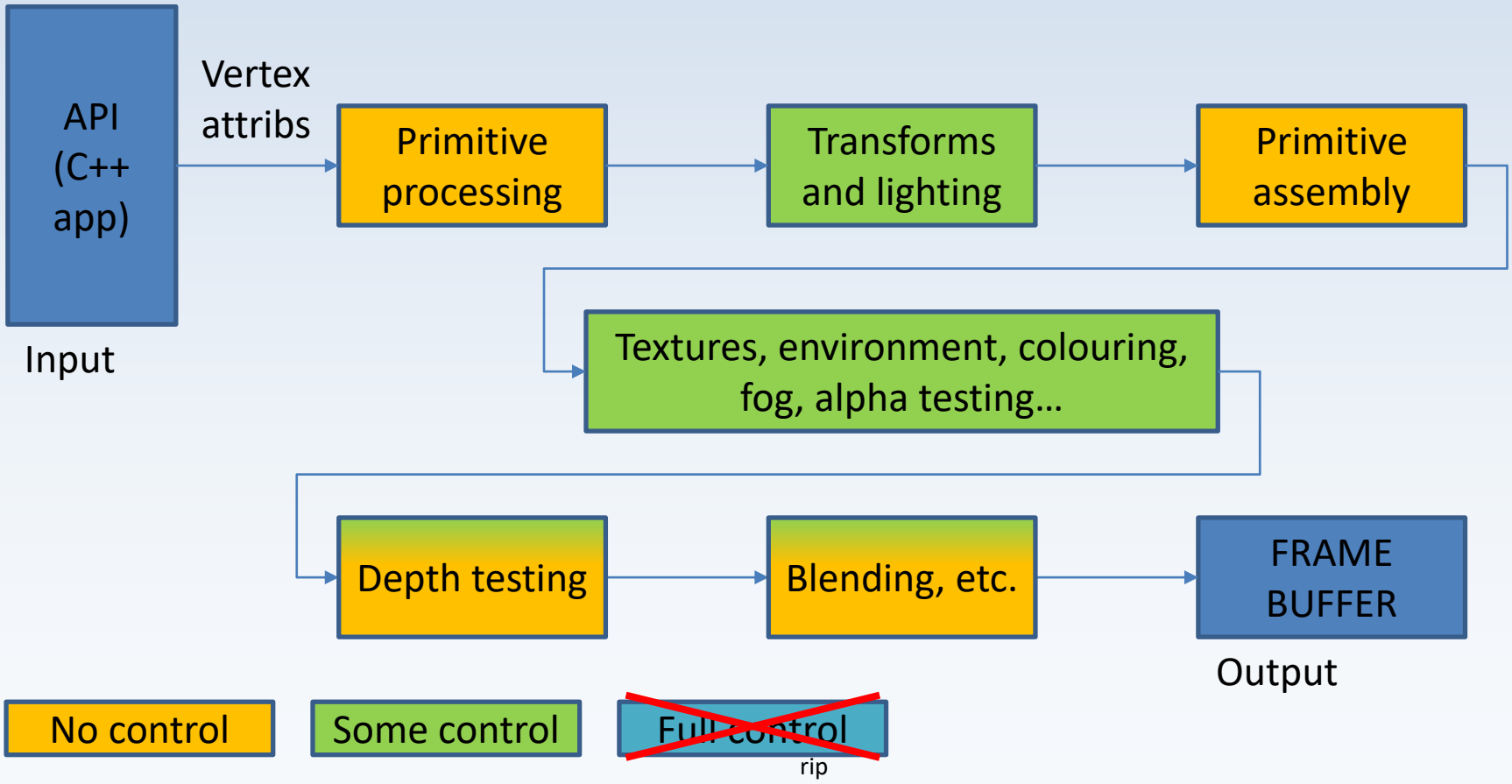
Fixed-Function vs. Programmable

- The OpenGL Machine:
  - https://www.opengl.org/documentation/specs/version1.1/state.pdf
- Fixed-function (version 1.1)



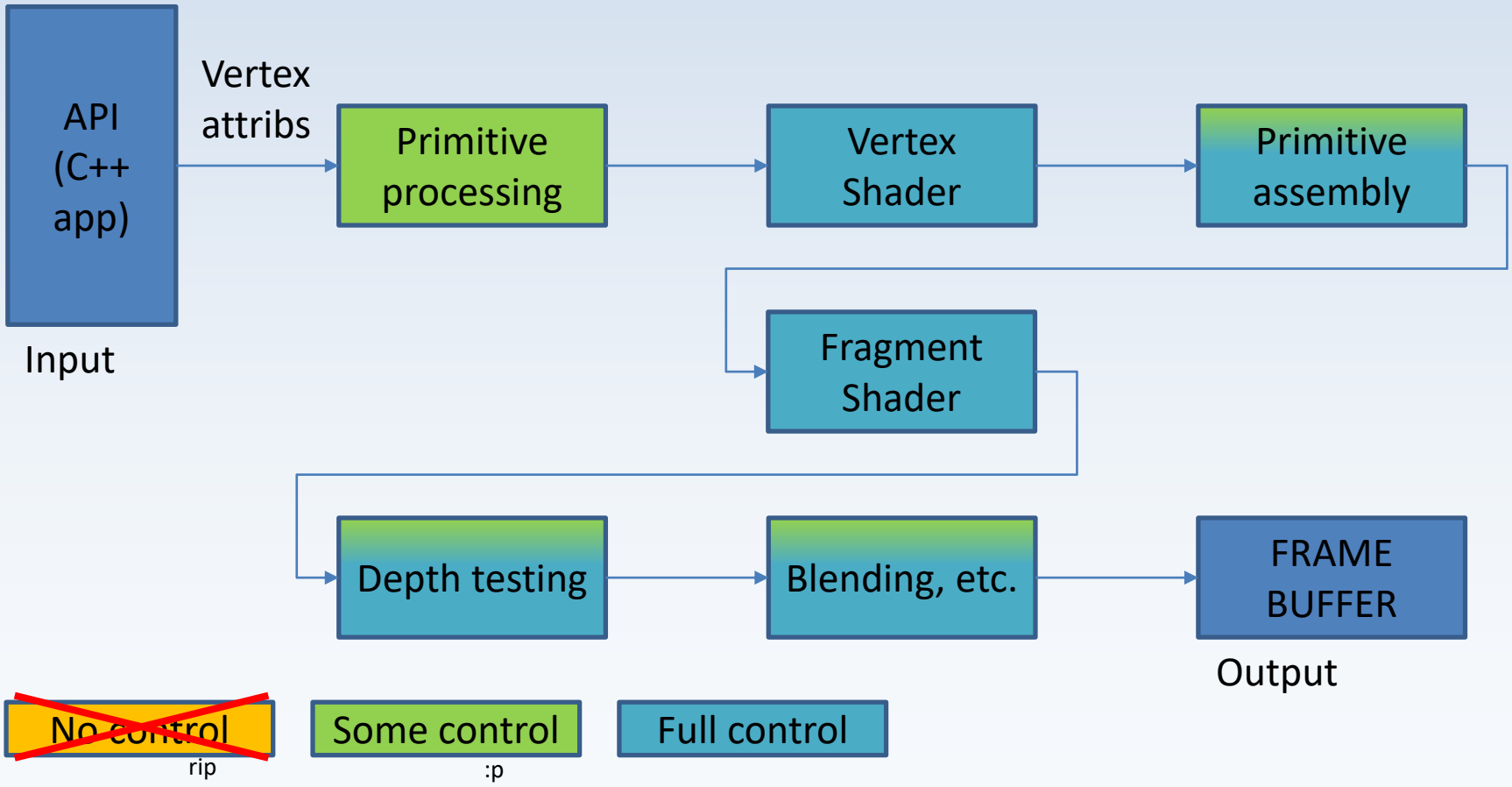
Fixed-Function vs. Programmable

- Fixed-function summary (OpenGL 1.1):



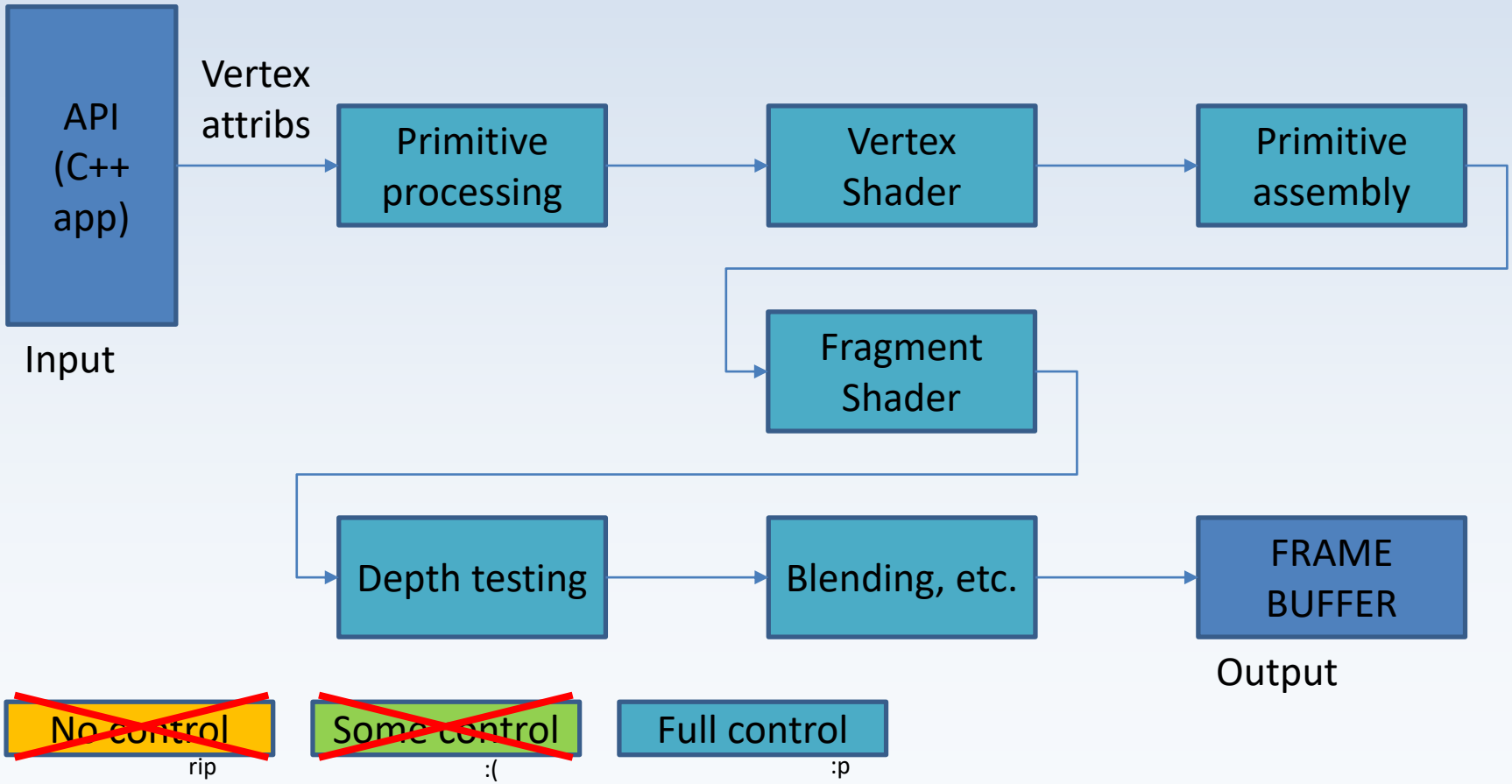
Fixed-Function vs. Programmable

- The move towards programmable pipeline:



Fixed-Function vs. Programmable

- Fun fact: there's this new thing called *Vulkan*…



Fixed-Function vs. Programmable

- Fixed-function pipeline:

- OpenGL 1
- All lighting and shading done automatically…
- …on a ***per-vertex basis***
- Not much control over anything

Fixed-Function vs. Programmable

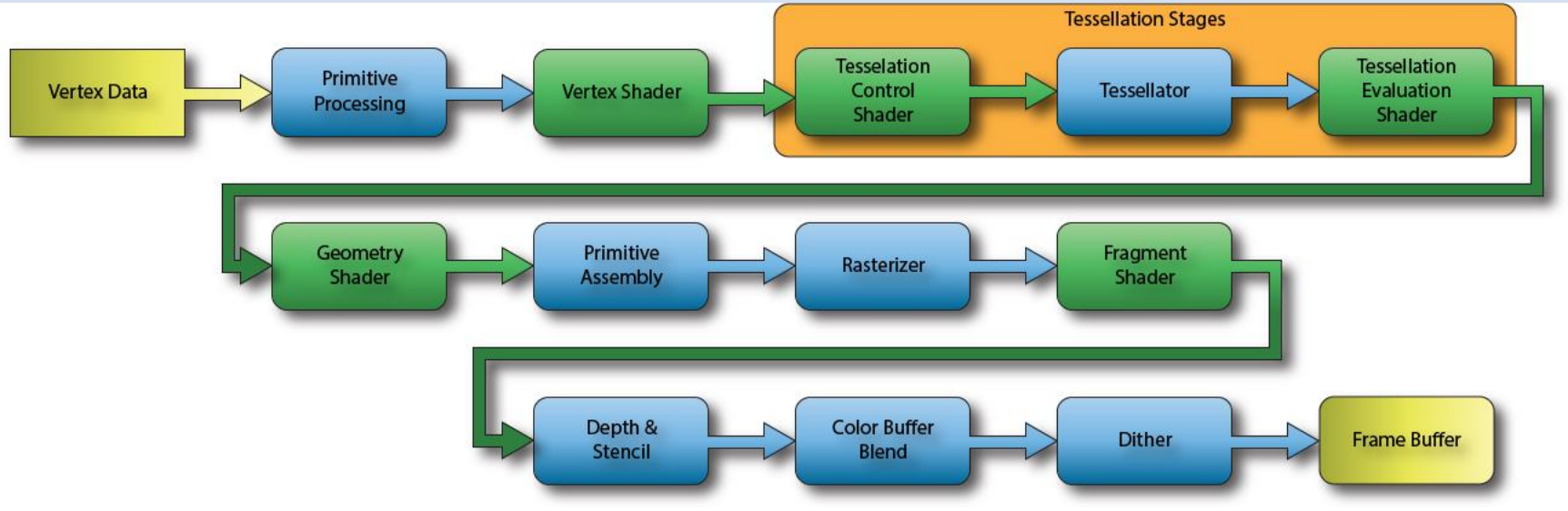
- Programmable pipeline:

- Current version of OpenGL is 4.5 (Aug. 2014)
- Full control over vertex and fragment ops

- …what's a vertex?
- …what's a fragment?

Fixed-Function vs. Programmable

- Programmable pipeline:



Data input/output

OpenGL-controlled process

Shader: programmer-controlled process

http://www.3dgep.com/wp-content/uploads/2014/02/OpenGL-4.0-Pipeline.png

Fixed-Function vs. Programmable

- So what is a ***shader***, then???
- A program that runs on the GPU
- Most languages are similar to C
- GLSL, HLSL, Cg…
- ***A small piece of code that executes on the GPU and replaces some component of the fixed-function pipeline.***
    - (hence, “programmable pipeline”)

Types of Shaders

- There are ***six*** (6) types of shaders that we use in modern OpenGL:

1. **Vertex shader (GLSL 1.1 [OpenGL 2.0]):**
   → process a single *vertex* and its attributes
2. Tessellation control shader (GLSL 4.0):
   → determine rules for subdividing primitives
3. Tessellation evaluation shader (GLSL 4.0):
   → perform fast, recursive subdivision

Types of Shaders

- There are ***six*** (6) types of shaders that we use in modern OpenGL (cont'd):

4. Geometry shader (GLSL 3.2):
   → process a single *primitive* as a set of verts
5. **Fragment shader (GLSL 1.1 [OpenGL 2.0])**
   → process a single *fragment*, output color
6. Compute shader (GLSL 4.3):
   → independent shader used for GPGPU

Types of Shaders

- GLSL ***program***: a series of *shaders* used to construct a ***custom rendering pipeline***

- A *program* or *series of programs*: "***algorithm***"
- Example: post-processing, "bloom"

- Minimum number of shaders per program???

Types of Shaders

- ***Vertex Shader***
- A vertex is your ***input data***: "***attributes***"
- Describe a single vertex in space
- TWO primary uses:

1) ***Required****: *set built-in variable* "`gl_Position`"
    - This is the vertex position in *clip space*, OpenGL uses it

2) Optional: *pass data* down the pipeline

*This step is required *unless* using a *geometry shader* that sets `gl_Position` (see below)

Types of Shaders

- ***Vertex Shader***
- Example vertex shader in GLSL 4.x: “pass-thru”

```
#version 450
layout (location = 0) in vec4 position;
void main()
{
    gl_Position = position;
}
```

Types of Shaders

- ***Fragment Shader***
- A fragment is your ***output data***: ***COLOR***
- Long story short: paint a single pixel
- Two main jobs* again:
- 1) Receive data from previous pipeline stages
- 2) Output ***data*** to drawing canvas (the end)
    - …doesn't have to be just color… color is *just data*…

Types of Shaders

- ***Fragment Shader***
- Example fragment shader in ***GLSL 4.x***
  - Works with example vertex shader above…

```
#version 450
layout (location = 0) out vec4 fragColor;
void main()
{
    fragColor = vec4 (1.0, 0.5, 0.0, 1.0);
}
```

Types of Shaders

- ***Fragment Shader***
- Another example… what does this do?

```
#version 450
void main()
{
    // y u no do anything  :(  ???
}
```

Key Shader Terms

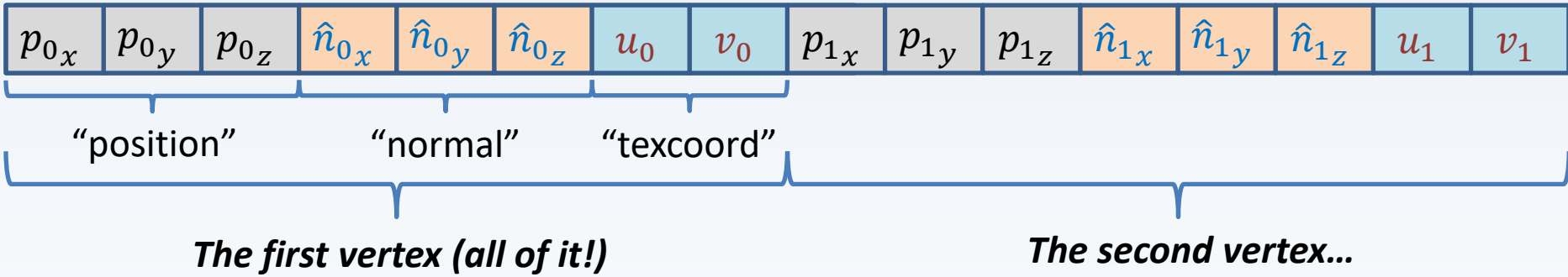
- GLSL looks a lot like C… but has its differences
- Key terms:

- ***Attribute***
- ***Uniform***
- ***Varying***
- ***Fragment color***
  - (more on this one later)

Key Shader Terms

- ***Attribute***: *data associated with a* ***vertex!!!***
- *Raw vertex data* processed in ***vertex shader!!!***
- *This is the raw data stored in our VBO!*
- Programmer defined… the building blocks of rendering!

Data in interleaved VBO:



Key Shader Terms

- ***Attribute***: ***vertex shader*** input

- Declaring an attribute in GLSL 1.2 (old):

```
attribute vec4 position;
```

Data input keyword — Data type — Attribute name

- Declaring an attribute in ***GLSL 4.x***:

```
layout (location = 0) in vec4 position;
```

Layout qualifier: which *vertex array* slot does this attribute occupy? — Input keyword — Data type (xyzw → vec4) — Attribute name

Key Shader Terms

- ***Uniform***: This one often confuses beginners…
- Attributes are ***different for each vertex***
- Uniforms are ***variables*** sent from CPU
- ***The same for all vertices in a single draw call!***
- Value is the same for the first vertex and the 1000th vertex during a single `glDrawArrays` call
- *May* change for each ***object*** (draw call)

Key Shader Terms

- ***Uniform***: ***variables*** sent from ***CPU*** (all shaders)

- GLSL 1.2 – 4.2:

```
uniform float myUnif;
```

Uniform keyword | Data type | Uniform name

- ***GLSL 4.3+***:

```
layout (location = 0) uniform float myUnif;
```

Layout qualifier: where is this variable? (*similar* to attribute slots, but ***not the same***) | Uniform keyword | Data type | Uniform name

Key Shader Terms

- ***Uniform***: ***variables*** sent from ***CPU***
- How to send uniform into GLSL (in C/C++):

- Get uniform handle (after linking program):

```
int myUniformHandle =
    glGetUniformLocation (program, "myUnif");
```

- Send variable (before draw call):

```
glUniform1f (myUniformHandle, 0.5f);
```

How many values | Data type | Uniform location | Value(s)

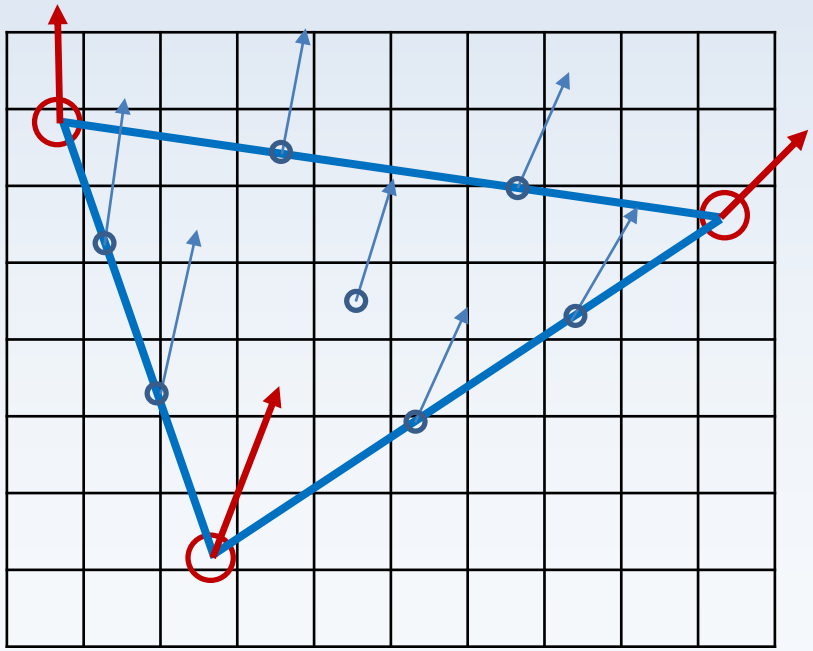
Key Shader Terms

- ***Varying***: values passed *from one shader to the next* in the same program
- Interpolated during *rasterization*, as fragments are generated:

Example:

Attributes (per vertex):
positions and normals

Varying (per fragment):
interpolated
positions and normals



Key Shader Terms

- ***Varying***: data passed through pipeline (all)
- GLSL 1.2: passing ***outbound*** varying data

```
varying vec3 varNormal;
```

Varying keyword — Data type — Name

- GLSL 1.2: receiving ***inbound*** varying data

```
varying vec3 varNormal;
```

Exactly the same as in vertex shader!

Key Shader Terms

- ***Varying***: values passed through the pipeline
- ***GLSL 4.x***: passing ***outbound*** varying data

```
out vec3 varNormal;
```

Data is being passed ***out***

Data type

Name

- ***GLSL 4.x***: receiving ***inbound*** varying data

```
in vec3 varNormal;
```

Data is being passed ***in***

Name and type must be the same as they appear in vertex shader!

Key Shader Terms

- ***Fragment color***: color to be converted to pixel
- Just as the vertex shader is the beginning of the pipeline and processes input…
- …fragment shader is at the end and decides the final output!
- We'll talk about this in detail when we get into framebuffers…
- …for now just know what you're looking at!

Key Shader Terms

- ***Fragment color***: ***fragment shader*** output
- GLSL 1.2: built-in, set an array value called `gl_FragData[]`
- ***GLSL 4.x***:

```
layout (location = 0) out vec4 fragColor;
```

Layout qualifier: which *render target* will the output become a pixel in?

Output keyword

Data type (vec4 → rgba)

Output name (totally arbitrary)

Additional Information

- Food for thought: ***Geometry Shader***
- Vertex shader: ***one*** vertex in, ***one*** vertex out
- Geometry shader: after VS, additional vertex processing: ***multiple*** vertices in, ***multiple*** out!
- Serves same purposes as VS (e.g. can set `gl_Position`), but for *multiple* vertices!
- Example: GPU particles: point in, quad out ☺
- Example: visualize vertex normal… how? ☺

Additional Information

- Food for thought: ***Uniform Buffer Object***
- Just as *attributes* live in a *VBO*, you may want your *uniforms* to live in a *UBO*
- Uniforms sent in a “block”
- Can get and set block data just like regular uniforms!
- Try it out and feel the optimization in your *bones* ;)

…terrible joke, you’ll appreciate it later

Additional Information

- Food for thought: ***Varying Structure***
- GLSL 4: try grouping your varyings together:

```
out myVaryingData {
    vec4 normal;
    vec4 position;
} passData; // outbound varying data!


in myVaryingData { ... // inbound: SAME NAMES!
```

The end.

- Questions? Comments? Concerns?