Teaching GL

Dave Shreiner
Director, Graphics and GPU Computing, ARM
1 December 2012

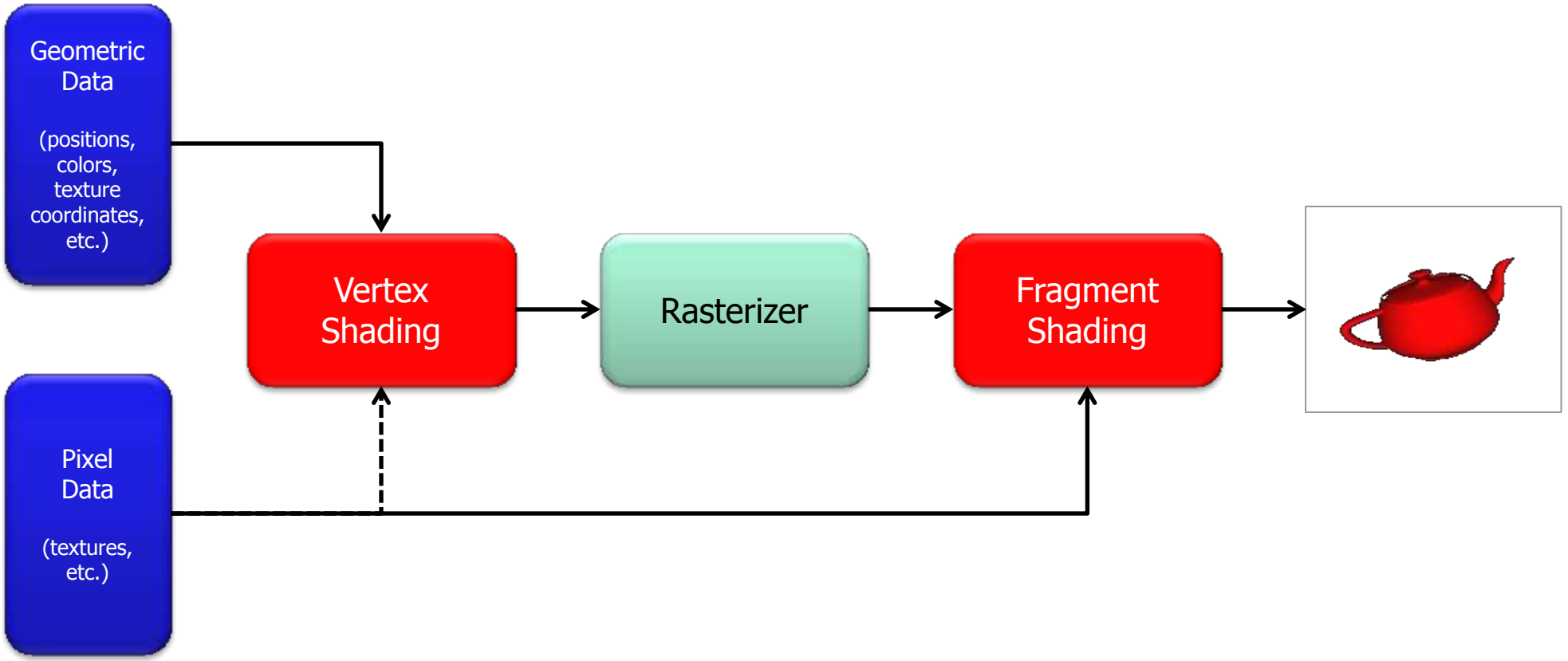
Agenda

- Overview of OpenGL family of APIs
- Comparison of rendering pipelines
- Debugging the most common OpenGL usage errors

The OpenGL Family

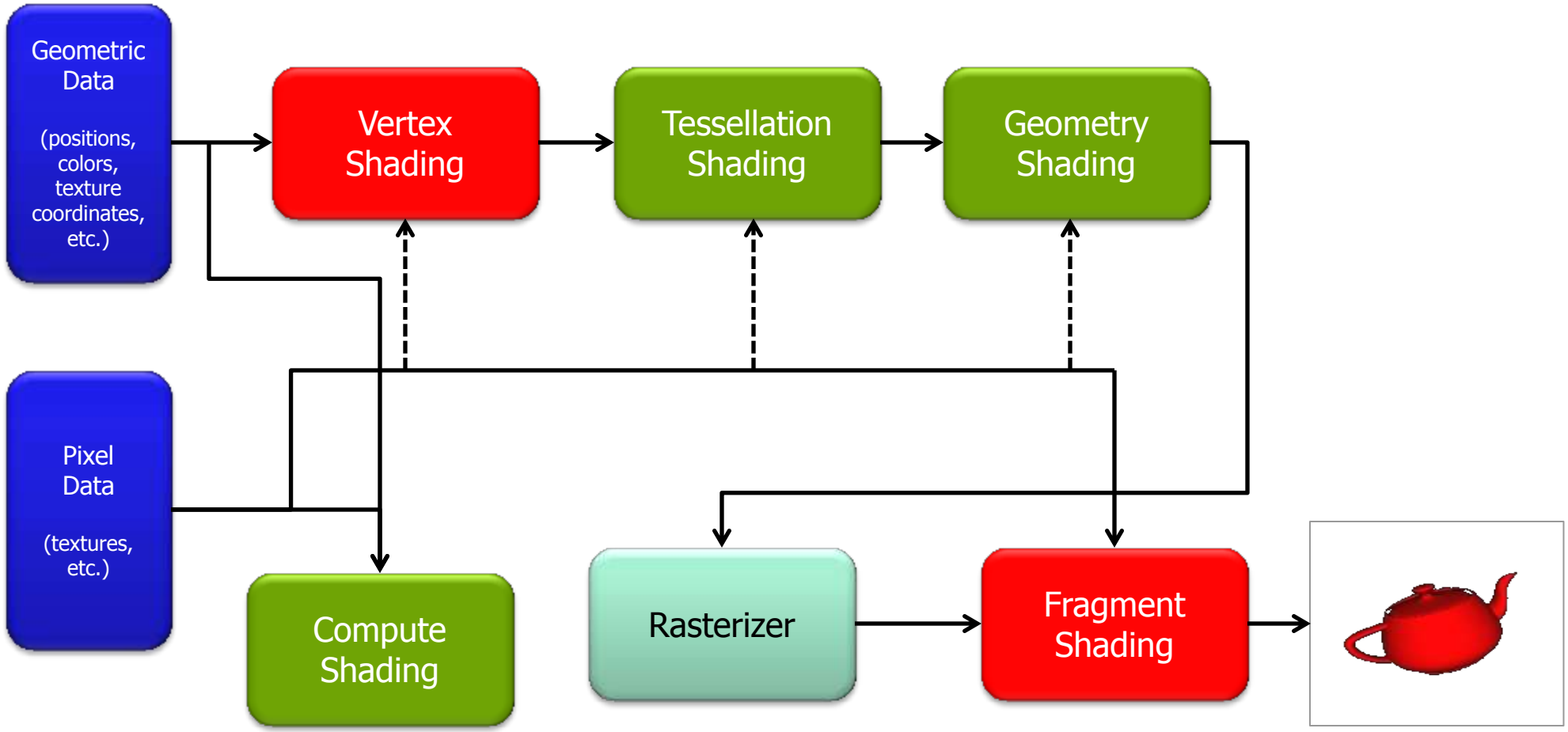
- **OpenGL**
  - The *cross-platform standard* for 3D Graphics
- **OpenGL ES**
  - The standard for *embedded* 3D Graphics
- **WebGL**
  - Bringing interactive 3D graphics to the Web through a JavaScript interface
- **OpenGL SC**
  - Verified 3D graphics API for *safety critical* applications
    - significantly different pipeline than the others
    - (not discussed in this session)

The Fundamental Rendering Pipeline



This is the pipeline that OpenGL ES and WebGL expose.
OpenGL extends the pipeline.

The OpenGL 4.3 Rendering Pipeline



A Few Basics Will Go a Long Way …

- **All of the APIs are *shader based***
  - they use variants of the OpenGL Shading Language – GLSL
    - "up" porting (from ES or WebGL) to OpenGL shouldn't require any work
    - "down" porting may require a few changes

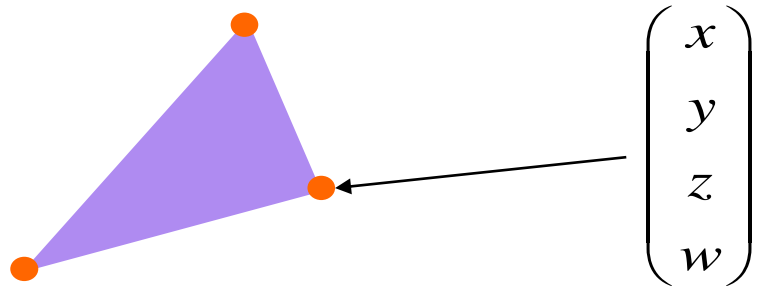
- **Always storing you data in *buffer objects* will guarantee portability and performance**
  - there are a few differences that you'll need to keep in mind

- **You'll also need to know a little *linear algebra***

- **Now, to introduce a few introductory concepts …**

Representing Geometric Objects

- **Geometric objects in OpenGL are represented using *vertices***
- **A vertex is a collection of generic attributes**
  - positional coordinates
  - colors
  - texture coordinates
  - any other data associated with that point in space
- **Positions are stored as 4-dimensional homogeneous coordinates**
- **Vertex data must be stored in vertex buffer objects (VBOs)**
- **VBOs must be stored in vertex array objects (VAOs)** (OpenGL only)

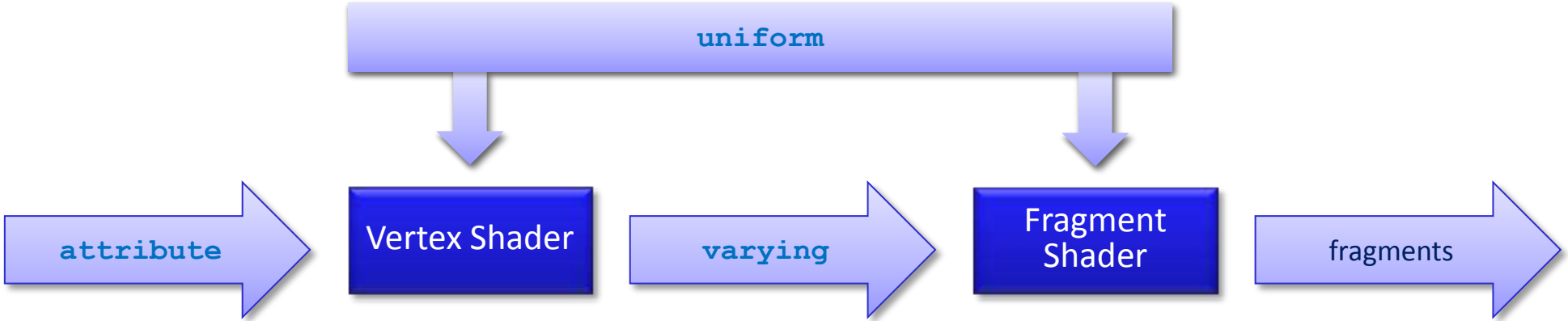


How Data Flows through GLSL Shaders

- ***attributes*** **(*in*) are the inputs into vertex shaders**
- ***varyings*** **(*out*) outputs of vertex shaders and inputs into fragment shaders**
- ***uniforms*** **are "constants" available to any shader stage**

API	GLSL Keyword	
	attribute	varying
OpenGL	`in`	`out`
ES / WebGL	`attribute`	`varying`

For OpenGL, all shader stages use
- `in` for all inputs into a shader
- `out` for all outputs from a shader



Anatomy of a Simple Vertex Shader (OpenGL version)

```
in  vec4 vPosition;
in  vec4 vColor;
out vec4 color;

void main()
{
    color = vColor;
    gl_Position = vertex;
}
```

Anatomy of a Simple Vertex Shader (ES/WebGL version)

```
attribute vec4 vPosition;
attribute vec4 vColor;
varying   vec4 color;

void main()
{
    color = vColor;
    gl_Position = vertex;
}
```


Anatomy of a Simple Fragment Shader (OpenGL version)

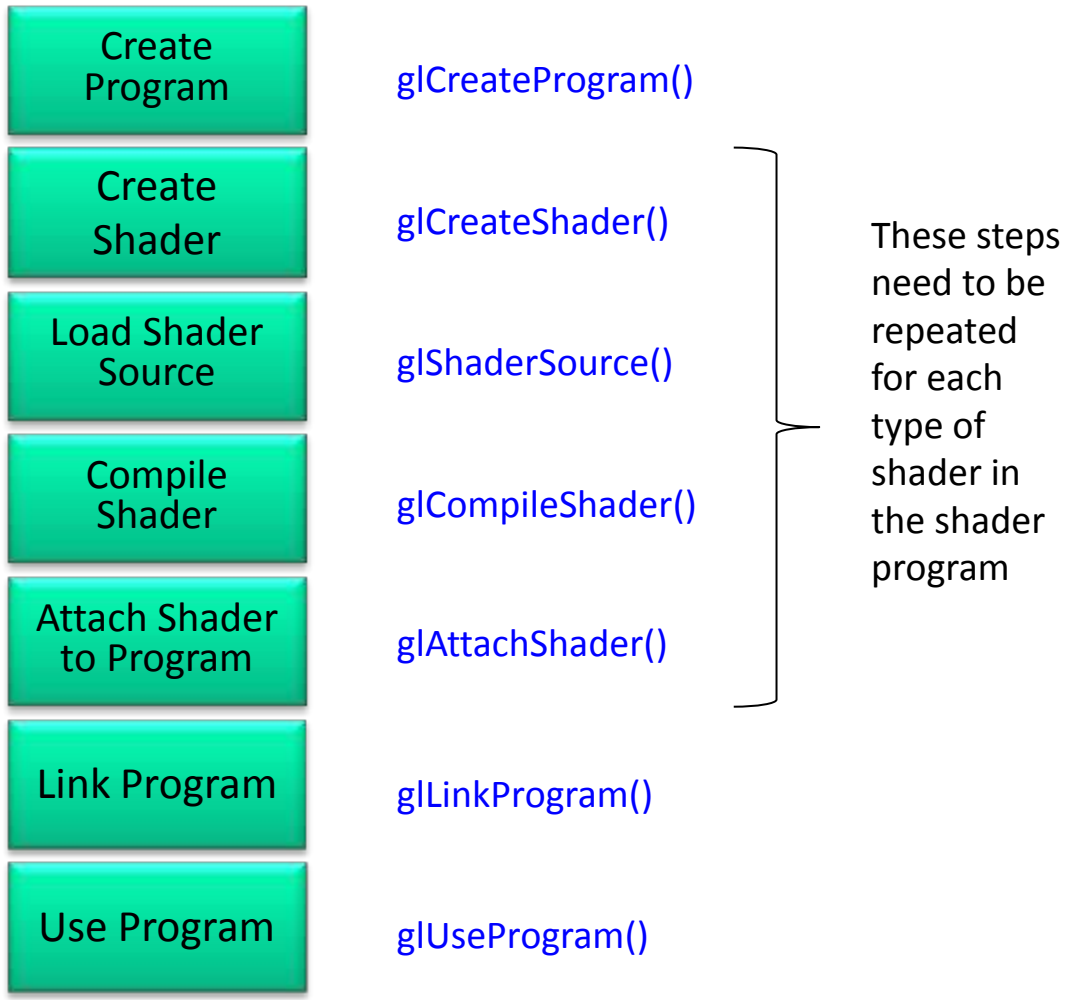
```
in  vec4 vColor;
out vec4 color;

void main()
{
    color = vColor;
}
```


Anatomy of a Simple Fragment Shader (ES/WebGL version)

```
varying   vec4 color;

void main()
{
    gl_FragColor = color;
}
```

Getting Your Shaders into OpenGL

- **Shaders need to be compiled and linked to form an executable shader program**
- **OpenGL provides the compiler and linker**
  - you access them by making function calls to the API
- **A program must contain**
  - one vertex shader
  - one fragment shader
  - other shader stages are optional (OpenGL only)



OpenGL Shader Plumbing

- **You need to make several connections to get data into the pipeline**
  - only need to do this attributes – varyings take care of themselves

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void
main()
{
  gl_Position =
vPosition;
  color = vColor;
}
```

Shader Compilation

Variable Name	Attribute Index
vPosition	12
vColor	2

OpenGL Shader Plumbing

- **You need to make several connections to get data into the pipeline**
  - only need to do this attributes – varyings take care of themselves

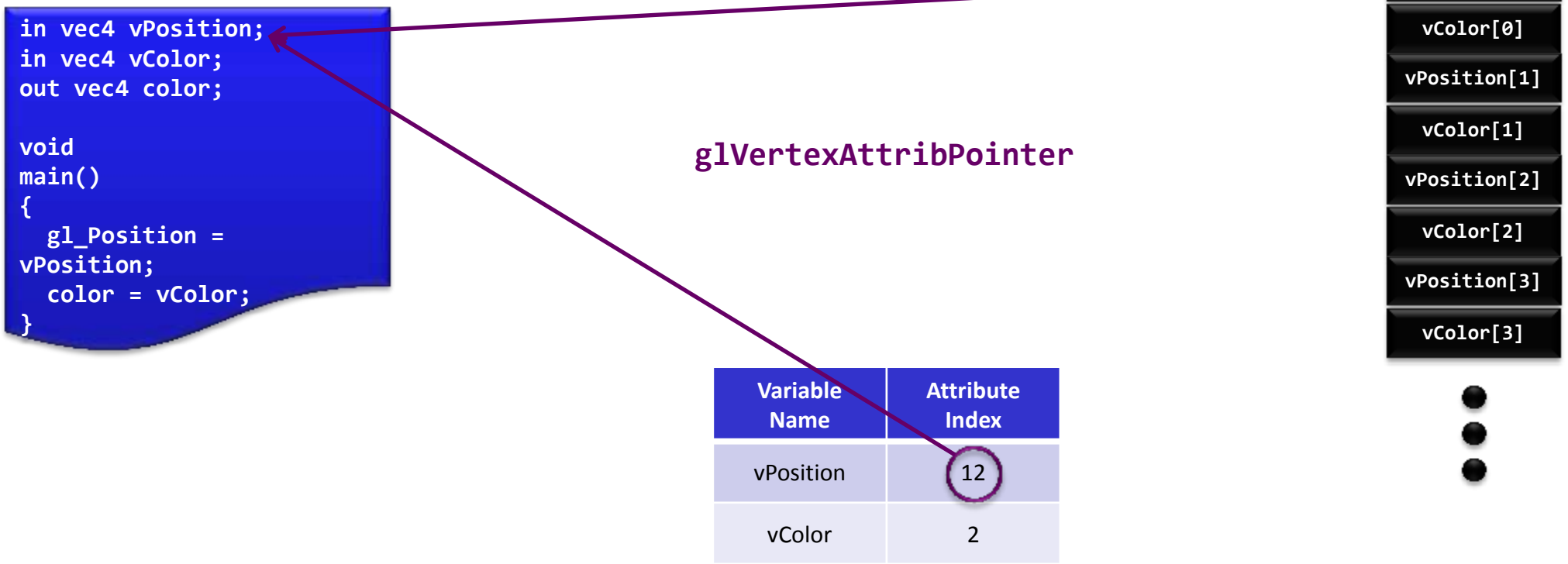
```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void
main()
{
  gl_Position =
vPosition;
  color = vColor;
}
```

glGetAttribLocation

Variable Name	Attribute Index
vPosition	12
vColor	2

OpenGL Shader Plumbing

- **You need to make several connections to get data into the pipeline**
  - only need to do this attributes – varyings take care of themselves



Debugging the Most Common Problems

The Bind-to-Edit Model

- **OpenGL uses *objects* for storing data**
  - it's loosely modeled on object-oriented programming, but not quite …
- **Objects are *implicitly* specified by a binding operation**
  - glBind*() calls
- **Objects are usually bound twice:**
  1. object creation and initialization
  2. specification as a data source for OpenGL rendering
- **This most often causes problems for novice programmers**
  - they forget the second binding (to use the data), or
  - they're bound to the wrong object

"Where are my objects?"

- **There are many steps that are required to have geometry show in a window**

- **The following must all occur correctly:**

1. specifying data into vertex arrays
2. compiling and linking shaders
3. associating vertex attributes with shader variables
4. specifying the *viewport*
5. loading uniform variables for shader use
6. transforming and projecting geometry into the viewport
7. passing the transformed vertices out of the vertex shader

- **Where should you start looking?**

Verifying Your Data's Getting Into OpenGL

- **You can use *Normalized Device Coordinates* (NDCs) to make sure data's flowing correctly**
  - x and y need to be between -1.0 and 1.0
  - z needs to be between 0.0 and 1.0

```
in vec4 vPos;

void main()
{
  gl_Position = vPos;
}
```

- **Verify using very simple vertex and fragment shaders**
  - often called a "pass-thru" shader

```
out vec4 color;

void main()
{
  color = vec4(1);
}
```

- **If your test geometry shows up, then you know:**
  - data is correctly specified to OpenGL
  - attributes are correctly specified and enabled
  - shaders compiled and linked successfully
  - viewport is correctly specified

OpenGL Matrices for C Programmers

- **Matrices in OpenGL are *column-major***
  - this is the exact opposite of how C programmers think

- **There are many places were a matrix's transpose can be introduced**
  - in the application (most straightforward)
  - when specifying the matrix as a uniform – glUniformMatrix(…, GL_TRUE, …);
  - in the shader
    - column_major qualifier in GLSL
    - transpose() method in GLSL

$$\mathbf{M} = \begin{matrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{matrix}$$

Matrix Multiplication Is Not Commutative

- **Vertices are usually transformed by 4×4 matrices**
  - modeling and viewing transformations
  - projection transformations

- **Vertices should be multiplied on right of a matrix** $v' = \mathbf{M}\vec{v}$

- **Matrices should be accumulated on the right as well** $C(BA)$

"Where Are My Textures?"

- **Loading an image to use as a texture in OpenGL is easy**
  - use the glTexImage*D call
  - loads a single *texture level*
    - levels are fundamental to a technique called *mipmapping*

- ***Sampling* controls how colors are retrieved from a texture map**
  - including whether mipmaps are to be used

- **However, OpenGL's default sampling mode require mipmaps**

- **Three potential solutions:**
  - change the sampling mode:

    glTexParameteri(…, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  - load more mipmap levels
  - automatically build mipmaps: glGenerateMipmaps()

Why Do My Textures Looked Skewed?

- **glTexImage2D transfers pixels form an application's memory into OpenGL**
  - various parameters are used to control which addresses are read from
- **All of those parameters are controlled by the glPixelStrore function**
- **The default mode is to assume every pixel starts on a *four-byte* boundary**
  - makes sense for RGBA 8-bit per component images
  - but, a lot of images are RGB only (and have a *three-byte* stride)
- **Adjust the GL_UNPACK_ALIGNMENT to "1"**
  - default is "4"
  - "1" specifies byte alignment

Thanks!