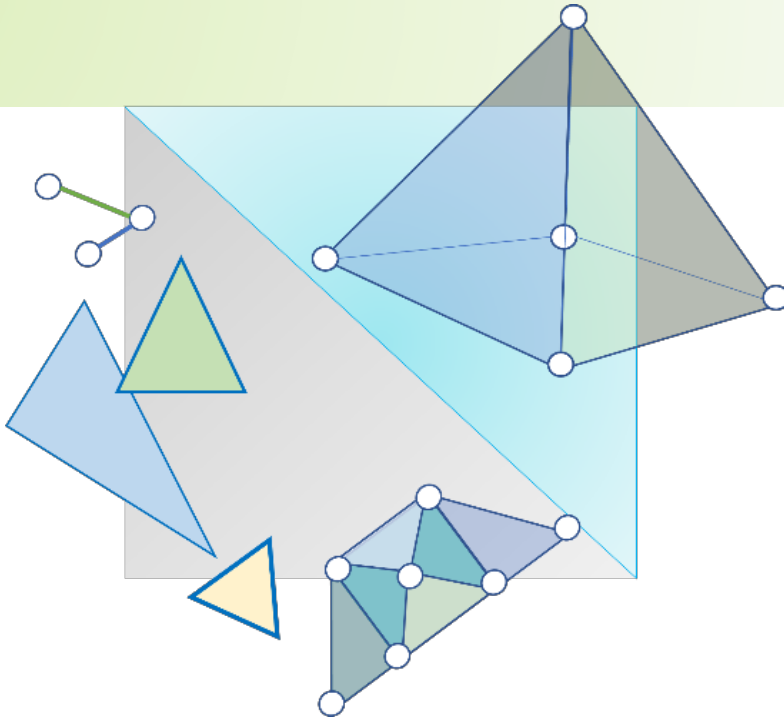
CITS3003 Graphics & Animation

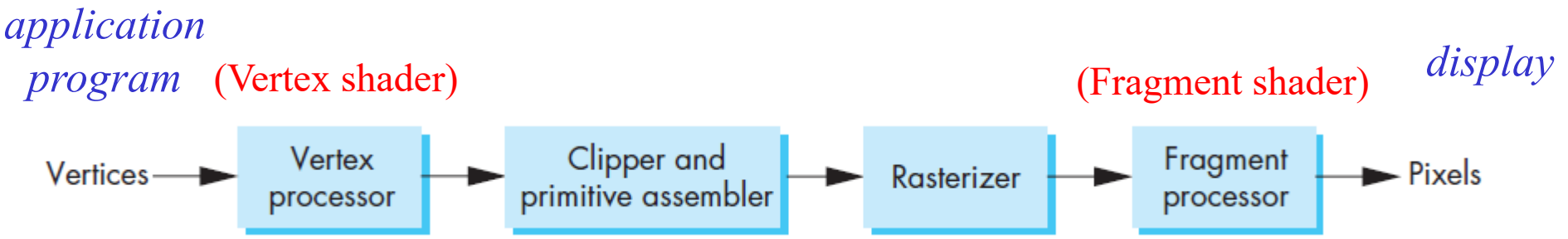


Lecture 6:
Vertex and Fragment Shaders-2

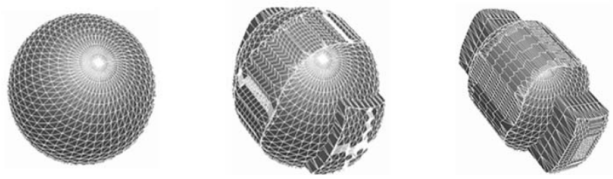
Content

- Vertex Shader
- Examples of Vertex Shader
- Fragment Shader
- Examples of Fragment Shader
- How the application program and vertex shader work together

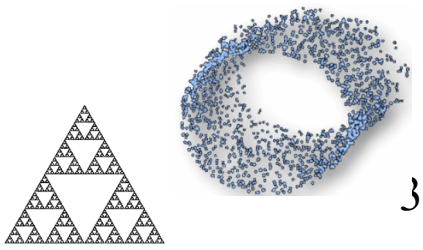
What vertex shader can do?
(Application perspective)



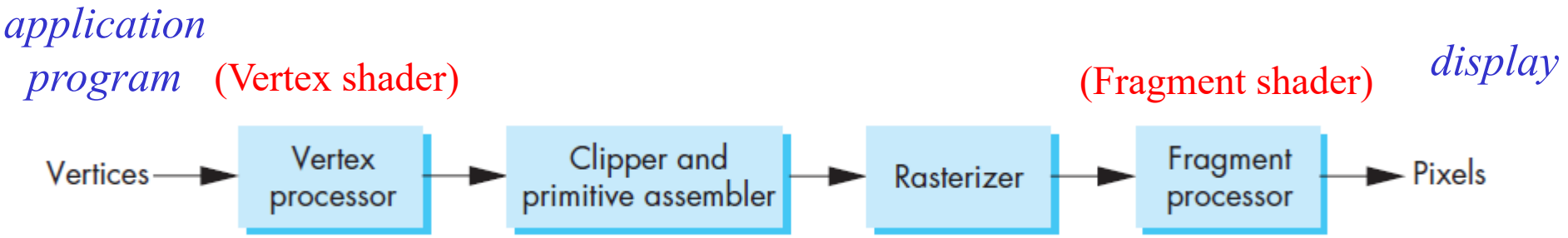
- Geometric transformations
  - Change location, rotation, scale of objects/camera
  - Apply 3D perspective transformation – make far objects smaller



- Moving vertices
  - Perform morphing
  - Compute wave motion & deformation due to physical forces
  - Simulate particle effects – for fire, smoke, rain, waterfalls, …
  - Compute fractals



What vertex shader can do?
(Application perspective)



- Lighting – vertex shader can also
  - Calculate shading color using light and surface properties
  - Calculate cartoon shading (for special effects)

Vertex Shader

- The vertex shader processes one vertex – it takes in one vertex from the vertex stream as input and generates the transformed vertex (optionally with attributes) to the output vertex stream.

- Multiple shader programs can be invoked and run in parallel to render complex scenes in real-time.

A Simple Vertex Shader

```
#version 150

in vec4 vPosition;

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

GLSL version 1.50

input from application

data type

must link to variable in application

built-in variable

OpenGL Version	GLSL Version
2.0	1.10
2.1	1.20
3.0	1.30
3.1	1.40
3.2	1.50

A more complex vertex shader

```
#version 150
in vec4 vPosition;
out vec4 color;
uniform vec3 theta;

void main()
{
    ….          // code omitted
    color = …..; // compute the out variable color
    gl_Position = vPosition; // may be a more complex expression
}
```

Vertex shader generates per-vertex data, which is typically passed on to the rasterizer, which interpolates vertex attributes over the surfaces of primitives to produce fragments. These fragments are then processed by the fragment shader.

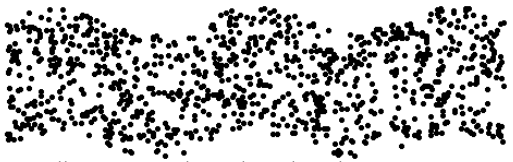
Can also have uniform variable

Vertex Shader – Example 1

Below is a ***wave motion*** *vertex shader* example:

```
in vec4 vPosition;
uniform float xs, zs; // frequencies
uniform float h; // height scale
uniform float time; //time

void main()
{
  vec4 t = vPosition;
  t.y = vPosition.y
        + h*sin(time + xs*vPosition.x)
        + h*sin(time + zs*vPosition.z);

 gl_Position = t;
}
```

Remember: Uniform variables cannot be modified in the shader

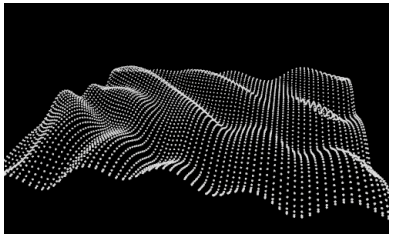


https://www.acs.psu.edu/drussell/demos/waves/wavemotion.html

Vertex Shader – Example 2

Below is a ***particle system*** example:

```
in vec3 vPosition;
uniform mat4 ModelViewProjectionMatrix;
uniform vec3 vel;
uniform float g, m, t;

void main() {
    vec3 object_pos;
    object_pos.x = vPosition.x + vel.x*t;
    object_pos.y = vPosition.y + vel.y*t + g/(2.0*m)*t*t;
    object_pos.z = vPosition.z + vel.z*t;

    gl_Position = ModelViewProjectionMatrix *
                  vec4(object_pos,1);
}
```



https://web.archive.org/web/20150930172450/http://www.easteal.org/pic/show.php?id=67612

Vertex Shader

```
attribute vec4 vPosition;
varying vec4 color;
uniform vec3 theta;

void main()
{
    ….          // code omitted
    color = …..;
    gl_Position = vPosition;
}
```

Older Version

```
in  vec4 vPosition;
out vec4 color;
uniform vec3 theta;

void main()
{
    ….          // code omitted
    color = …..;
    gl_Position = vPosition;
}
```

Opengl Version 3+

What can Fragment Shader do?
(Application perspective)

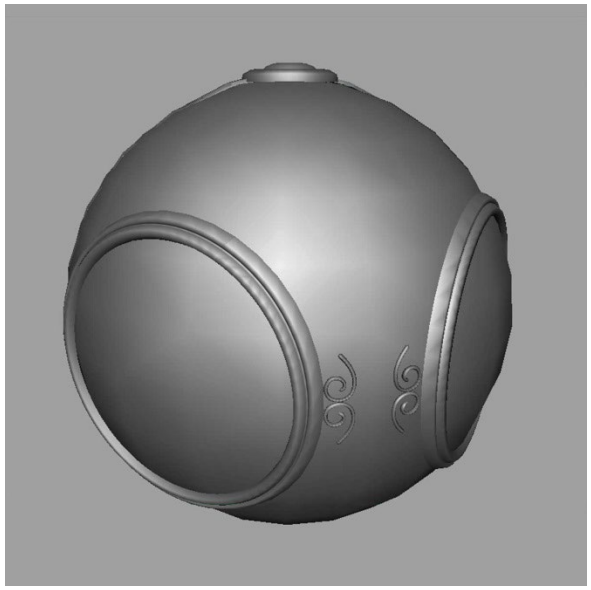
- Per fragment lighting calculations

  (recall that a *fragment* is a potential pixel that not only has location coordinates but also has colour, depth, and alpha values)

per vertex lighting

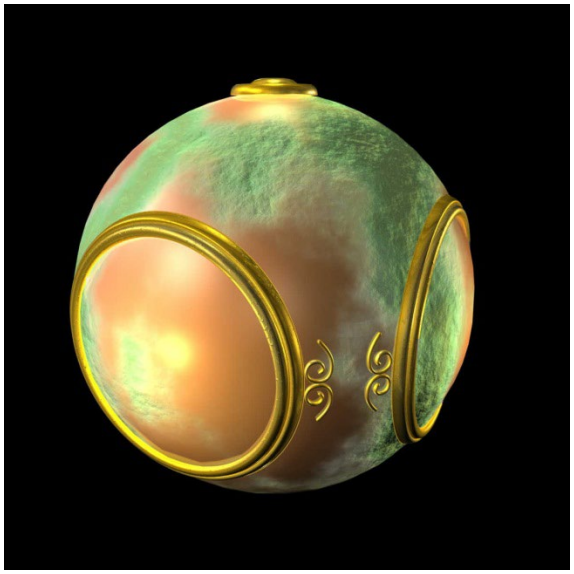
per fragment lighting

What can Fragment Shaders do?
(Application perspective)



smooth shading

environment mapping



bump mapping

What can Fragment Shader do?
(Application perspective)

- Lighting calculation
  - Per fragment lighting

- Texture mapping, including
  - Environment mapping
  - Bump mapping

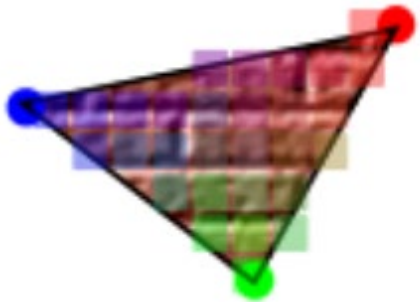


Image credits: Author: Romain Vergne link

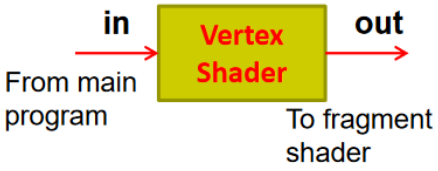
What can Fragment Shader do?
(Application perspective)

```
#version 150

out vec4 fragcolor;

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

declare that *fragcolor* as an output variable of the shader

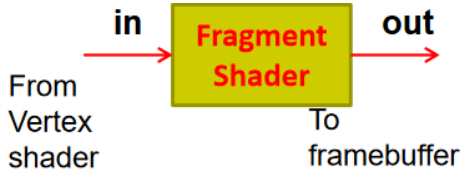
fragcolor must be computed and output

Example: Vertex and fragment shaders




```
#version 150

const vec4 red =
   vec4(1.0, 0.0, 0.0, 1.0);
in vec4 vPosition;
out vec4 color_out;

void main(void)
{
  gl_Position = vPosition;
  color_out = red;
}
```

Vertex shader




```
#version 150
in vec4 color_out;
out vec4 fragcolor;

void main(void) {
  fragcolor = color_out;
}
// in pre-OpenGL 3.2
// versions, use built-in:
// gl_FragColor = color_out;
```

Fragment shader

out variables declared in the vertex shader must be *in* variables in the fragment shader

Example: Vertex and fragment shaders

Older versions

```
const vec4 red =
   vec4(1.0, 0.0, 0.0, 1.0);
attribute vec4 vPosition;
varying vec4 color_out;

void main(void)
{
  gl_Position = vPosition;
  color_out = red;
}
```

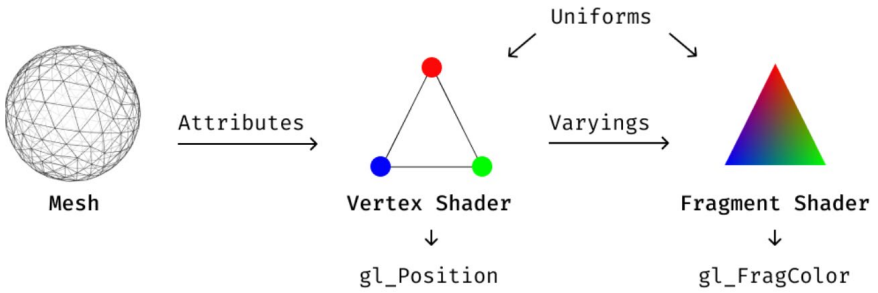
Vertex shader

```
varying vec4 color_out;

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

Built-in variable

Fragment shader

varying variables declared in the vertex shader must be *varying* variable in the fragment shader



Shaders and Application Program Must Work Together

- Using OpenGL to render 3D images generally involves sending data through the OpenGL shader pipeline. For example, to draw a simple 3D object such as a cube, you will need to at least send the following items:
  - the vertices for the cube model
  - some transformation matrices to control the appearance of the cube's orientation in 3D space

How to send data from Application to the Shader?

Sending data from Application to the Shader

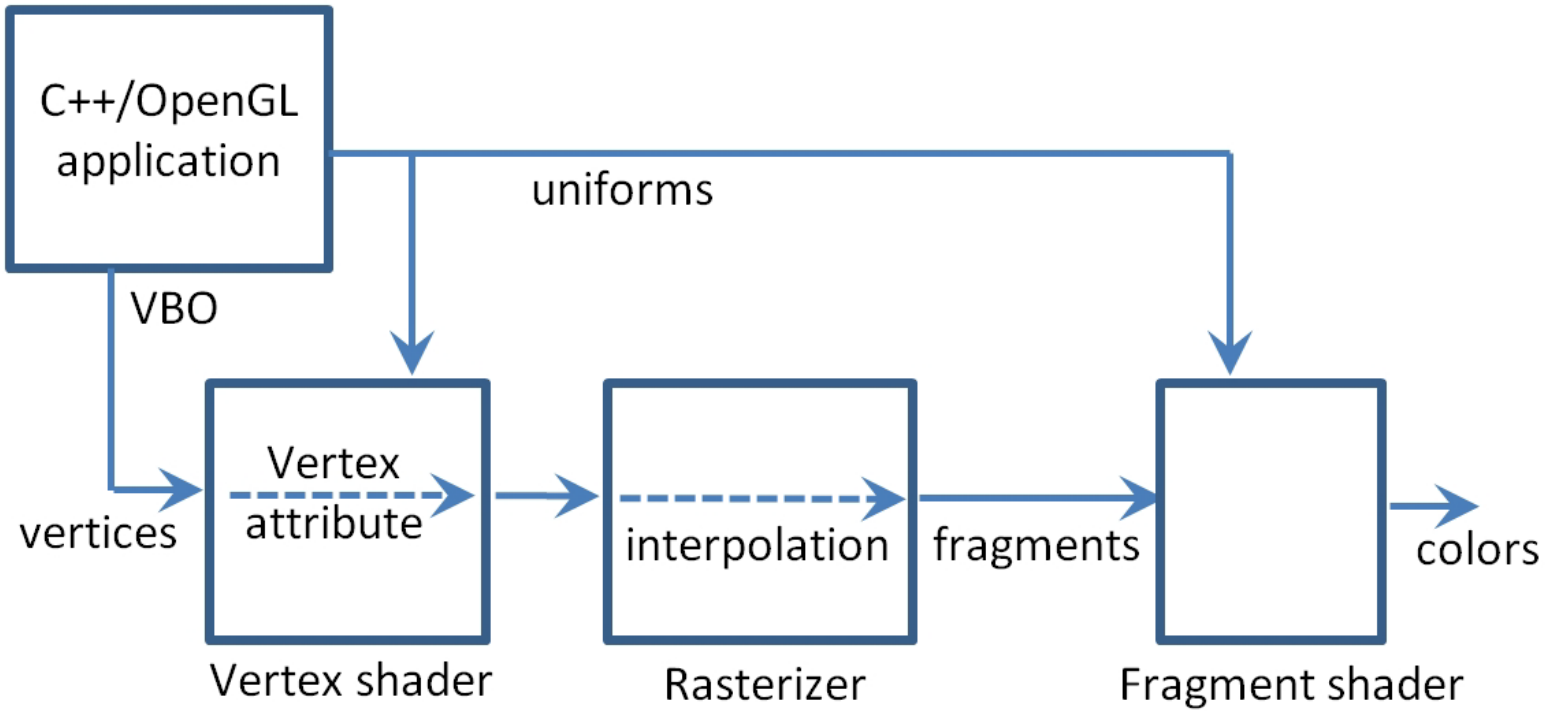
Two methods:

- through a *buffer* to a *vertex attribute*, or
- *uniform variable*

In OpenGL, a buffer is called a **VBO**
(**V**ertex **B**uffer **O**bject)

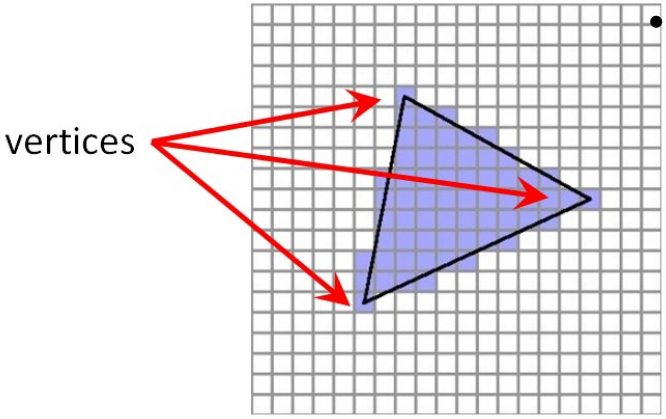
VBOs are declared and instantiated in the C++/OpenGL application

Overview of data flow for a C++/OpenGL application, through pipeline shaders *(vertex attributes and uniform variables)*



Which one should I use?

- Use a **uniform variable** for values that are constant for the entire object being drawn (such as transformation matrices)
  - data that may change relatively infrequently compared to per-vertex attributes
- Use a **vertex attribute** for per vertex data or when you want the values to be interpolated by the rasterizer.

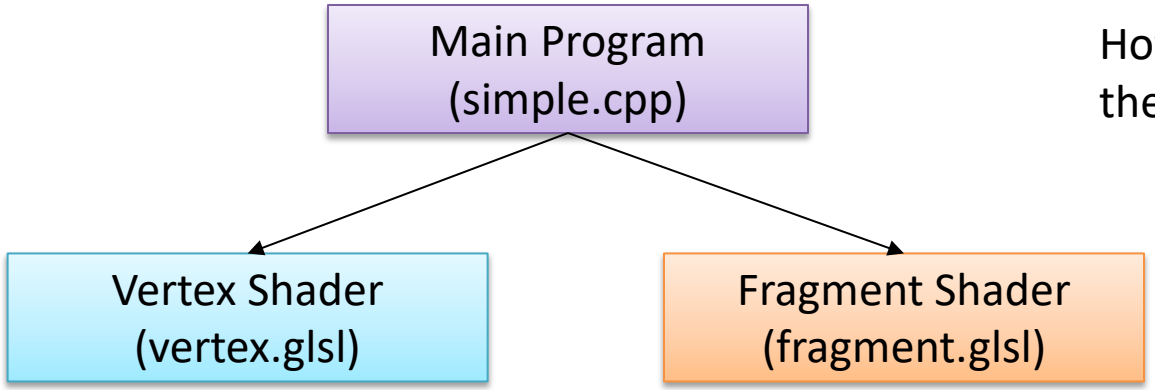


rasterization of vertices

- Rasterization *linearly interpolates vertex attribute* values so that the displayed pixels seamlessly connect the modeled surfaces.

Shaders and Application Program Must Work Together

- For each variable declared using the qualifier attribute (or in) in the vertex shader, the application needs to know how to link to it.



```
// Create a vertex array object
GLuint vao;
glGenVertexArrays( 1, &vao );
glBindVertexArray( vao );

// Create and initialize a buffer object
GLuint buffer;
glGenBuffers( 1, &buffer );
glBindBuffer( GL_ARRAY_BUFFER, buffer );
glBufferData( GL_ARRAY_BUFFER, sizeof(points), points, GL_STATIC_DRAW );

// Load shaders and use the resulting shader program
GLuint program = InitShader( "vshader.glsl", "fshader.glsl" );
glUseProgram( program );

// enable the vertex attribute and associate the attribute with the buffer
GLuint loc = glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( loc );
glVertexAttribPointer( loc, 2, GL_FLOAT, GL_FALSE, 0,
                       BUFFER_OFFSET(0) );
```

Shaders and Application Program Must Work Together

initShader( )

Connects main program to shader files

- Compiles and links vertex, fragment shaders
- Linking involves
  - making the connections between the input variables from one shader to the output variables of another.
  - making the connections between the other input/output variables of a shader to appropriate locations in the OpenGL environment.

```
// Load shaders and use the resulting shader program
   GLuint program = InitShader( "vertex.glsl", "fragment.glsl" );
   glUseProgram( program );
```

Shaders and Application Program Must Work Together

initShader( )

Connects main program to shader files

- Compiles and links vertex, fragment shaders
- Linking involves
  - making the connections between the input variables from one shader to the output variables of another.
  - making the connections between the other input/output variables of a shader to appropriate locations in the OpenGL environment.

Vertex shader file

```
in vec4 vPosition
```

Location of vPosition →

Variable
Variable 1
vPosition
.....
Variable N

Compiler puts all variables declared in shader into a table

Shaders and Application Program Must Work Together

- All attribute/uniform variable names are stored in a table.
- The application program can get an **index** for each attribute variable from the table.

Shaders and Application Program Must Work Together

Vertex shader file

```
in vec4 vPosition
```

Location of vPosition →

Variable
Variable 1
vPosition
.....
Variable N

1-Get index/location of vertex attribute **vPosition**

```
GLuint vPos = glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( vPos );
```

Shader program handle returned by InitShader() (slide25)

2-Enable vertex array attribute

3-Describe the form of data in the vertex array

Address in the buffer, where data begins

```
glVertexAttribPointer( vPos, 3, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0) );
```

Location of vPosition

3 (x,y,z) floats per vertex

Data no normalized (0-1 range)

Stride/Data is contiguous

Reference to *attribute* (or *in*) variables – An Example

- **In application program (in function init()):**

The application program can refer to the vertex attribute via this index

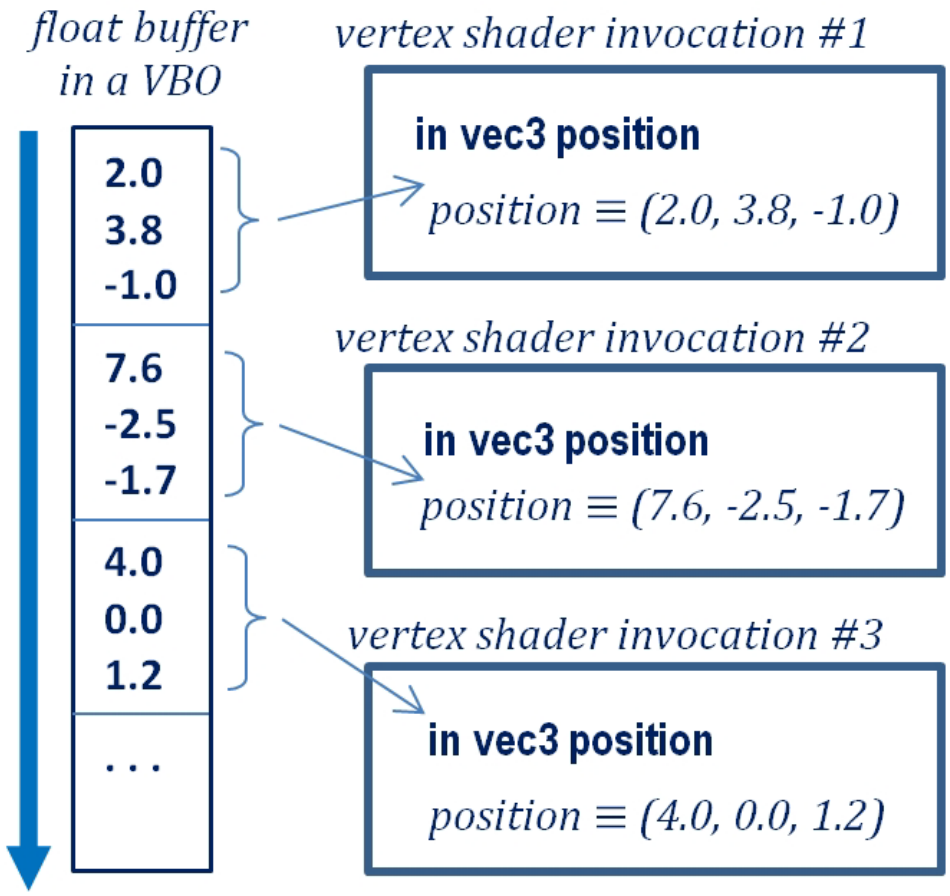
```
GLuint loc = glGetAttribLocation( program, “position" );
glEnableVertexAttribArray( loc );
glVertexAttribPointer( loc, 3, GL_FLOAT, GL_FALSE, 0,
                       BUFFER_OFFSET(0) );
```

- **In vertex shader:**

Must be the same

```
in vec3 position;
```

Data transmission between a VBO and a vertex attribute



The vertex shader runs once per vertex

Reference to *attribute* (or *in*) variables – Another Example

- **In application program (in function init()):**

```
GLuint loc, loc2;

loc = glGetAttribLocation(program, "vPosition");
glEnableVertexAttribArray(loc);
glVertexAttribPointer(loc, 3, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(0));

loc2 = glGetAttribLocation(program, "vColor");
glEnableVertexAttribArray(loc2);
glVertexAttribPointer(loc2, 3, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(sizeofpoints));
```

// vPosition and vColor are *in* variables in the vertex shader

- **In vertex shader:**

```
in vec3 vPosition;
in vec3 vColor;
```

Reference to *uniform* Variables – An Example

Sometimes we want to connect variables in OpenGL application to uniform variable in shader

- **In application program (init()):**

```
/* my_angle set in application */
GLfloat my_angle;
my_angle = 5.0 /* or some other value */
```

- Declare a variable in the application program
- Assign it a value

```
GLuint angleParam;
angleParam = glGetUniformLocation(myProgObj,"angle");
```

- find location of shader "**angle**" variable in linker table

```
glUniform1f(angleParam, my_angle);
```

- Connect: location of shader variable shader "**angle**" to application variable "my_angle"

The data type must be consistent

- **In vertex shader:**

```
uniform float angle;
```

- Declare a uniform variable in the shader

Reference to *uniform* Variables – An Example

Sometimes we want to connect variables in OpenGL application to uniform variable in shader

- **In application program (init()):**

```
/* my_angle set in application */
GLfloat my_angle;
my_angle = 5.0 /* or some other value */

GLuint angleParam;
angleParam = glGetUniformLocation(myProgObj,"angle");

glUniform1f(angleParam, my_angle);
```

- **In vertex shader:**

```
uniform float angle;
```

This line appears in the **display** callback function also, as the new value of *my_angle* computed in the application program for every frame needs to be copied to the vertex shader.

slido

Which of the following statements are correct:

ⓘ Start presenting to display the poll results on this slide.

Further Reading

- "Interactive Computer Graphics – A Top-Down Approach with Shader-Based OpenGL" by Edward Angel and Dave Shreiner, 6th Ed, 2012
  - Sec2. 2.8.2-2.8.5
    The Vertex Shader …The InitShader Function
  - Sec 3.12.2 Uniform Variables

- David Wolff - OpenGL 4.0 Shading Language Cookbook
- OpenGL Programming Guide 9th Edition-Kessenich
- Computer Graphics programming in OpenGL with C++- Scott Gordon

- A good reference on OpenGL shaders:
  http://antongerdelan.net/opengl/shaders.html