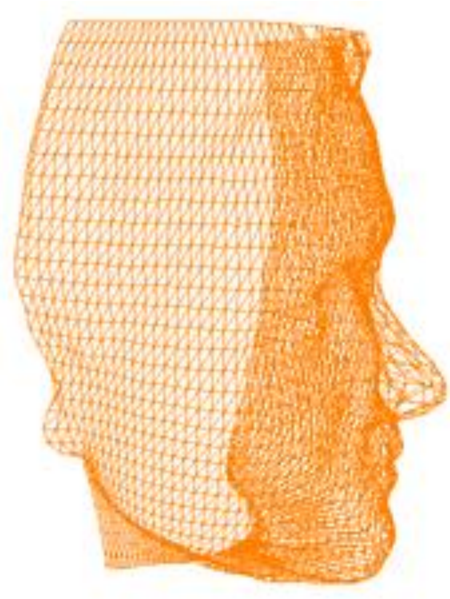
COSC422 Advanced Computer Graphics



1 OpenGL4 – A Brief Overview

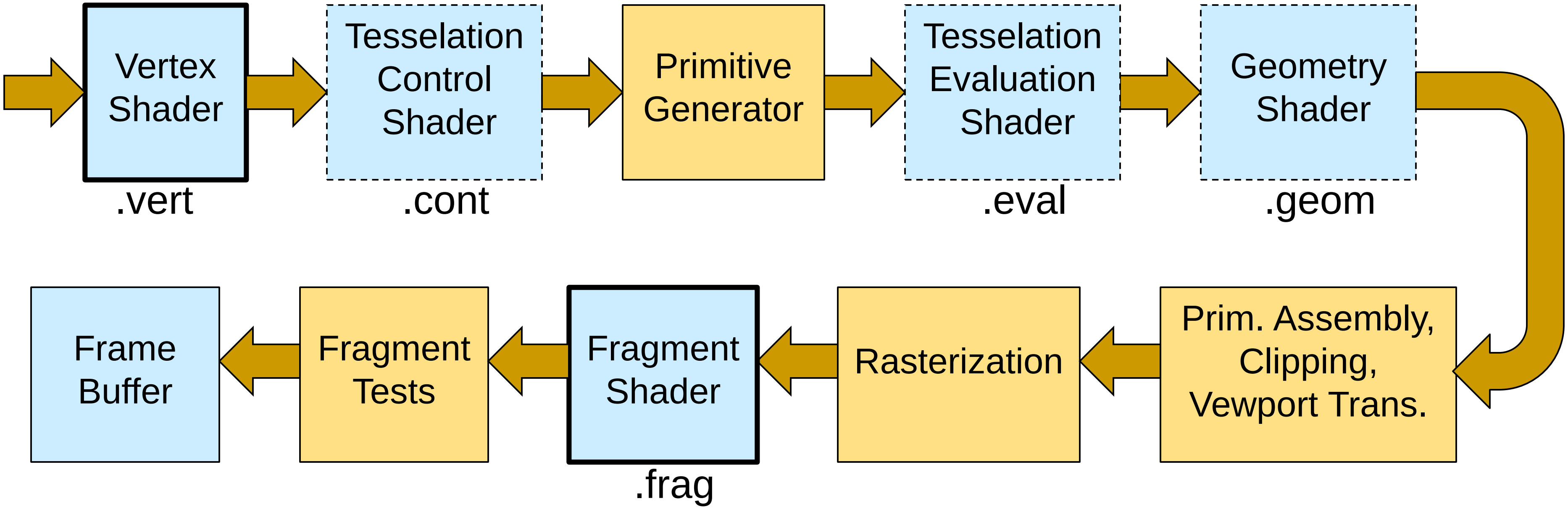
Semester 2
2021

R. Mukundan (mukundan@canterbury.ac.nz)
Department of Computer Science and Software Engineering
University of Canterbury, New Zealand.

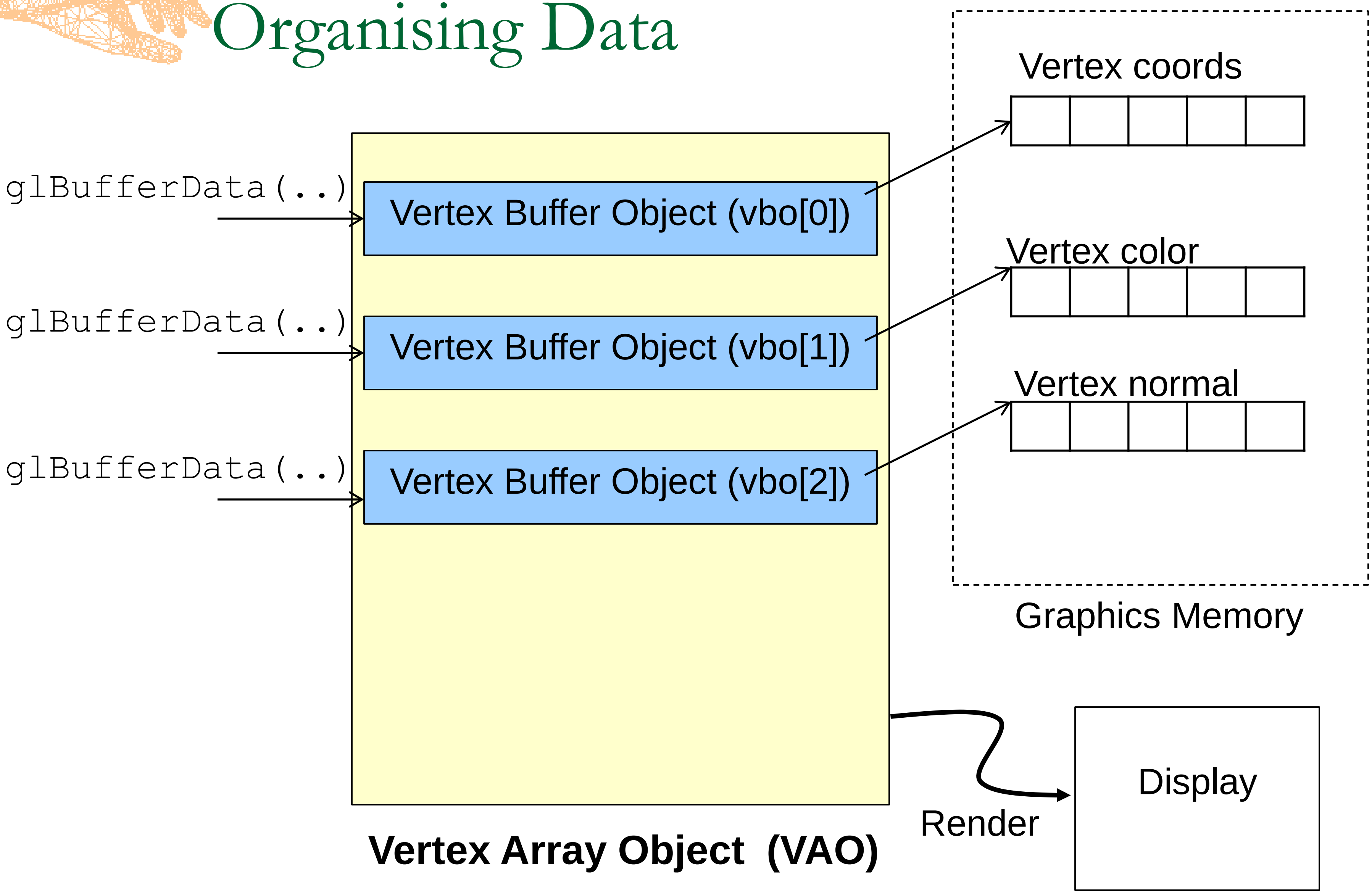
Lecture Outline

- Introduction to the Programmable Pipeline
- Organising Data
  - Vertex Buffer Objects
  - Vertex Array Objects
- Basic Computations using Vertex, Fragment Shaders
  - Transformations, Lighting
  - Texturing
- Tessellation Shader Stage
  - GL_PATCHES
  - Control and Evaluation Shaders
- Geometry Shader

OpenGL-4 Shader Stages

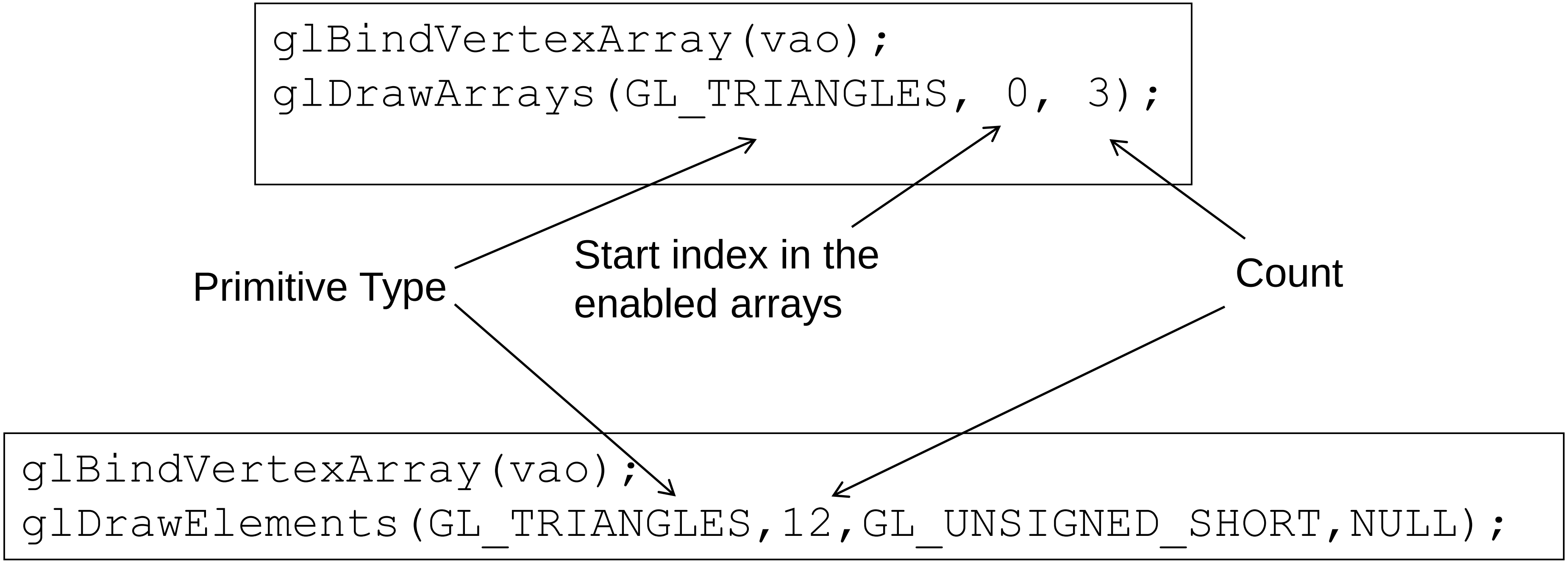


Organising Data



Rendering

- Bind the VAO representing the vertex data
- Render the collection of primitives using `glDrawArray()` or `glDrawElements()` command:



```
glBindVertexArray(vao);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

Primitive Type

Start index in the enabled arrays

Count

```
glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES,12,GL_UNSIGNED_SHORT,NULL);
```

Vertex Shader

- The vertex shader will execute once of every vertex.
- The position and any other attributes (normal, colour, texture coords etc) of the *current vertex*, if specified, will be available in the shader.
- Positions and attributes of other vertices are not available.

other pos. not avaliable

- A vertex shader *normally* outputs the clip coordinates of the current vertex, and also performs lighting calculations on the vertex.
- **`gl_Position`** is a built-in out variable for the vertex shader. A vertex shader *must* define its value.

Fragment Shader

- A fragment shader is executed for each fragment generated by the rasterizer.
- A fragment shader outputs the colour of a fragment and optionally the depth value.
- Several colour computations (texture mapping, colour sum etc.), and depth offsets can be performed inside a fragment shader.
- A fragment shader can also discard a fragment.
- A fragment shader has the built-in in variable `gl_FragCoord` and built-in out variables **`gl_FragColor`** and `gl_FragDepth`

(optional)

Defining Transformations

Vertex Shader

```
#version 330

layout (location = 0) in vec4 position;
uniform mat4 mvpMatrix;

void main()
{
       gl_Position = mvpMatrix * position;
}
```

Output in **clip coordinates**

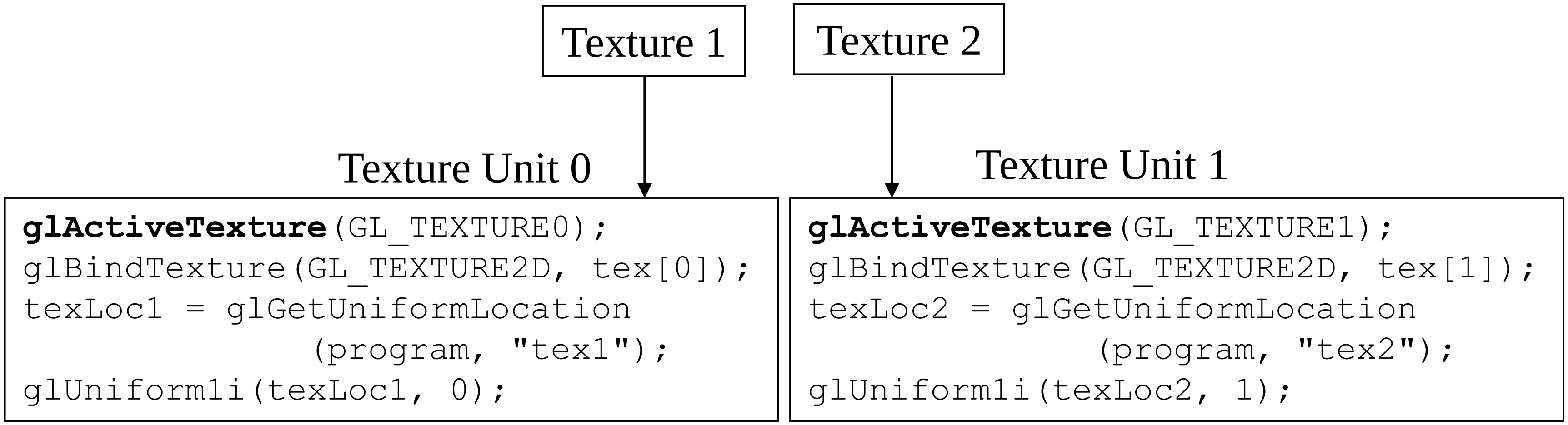
Input in world coordinates

Lighting Calculations (Vertex Shader)

for more complex Lighting
- can compute this in th ~~fragment~~ geom. Shader

```
    vec4 posnEye = mvMatrix * position;     //point in eye coords
    vec4 normalEye = norMatrix * vec4(normal, 0);
    vec4 lgtVec = normalize(lightPos - posnEye);
    vec4 viewVec = normalize(vec4(-posnEye.xyz, 0));
    vec4 halfVec = normalize(lgtVec + viewVec);
    vec4 material = vec4(0.0, 1.0, 1.0, 1.0);  //cyan
    vec4 ambOut = grey * material;
    float shininess = 100.0;
    float diffTerm = max(dot(lgtVec, normalEye), 0);
    vec4 diffOut = material * diffTerm;
    float specTerm = max(dot(halfVec, normalEye), 0);
    vec4 specOut = white *  pow(specTerm, shininess);

    gl_Position = mvpMatrix * position;
    theColour = ambOut + diffOut + specOut;
}
```

Multi-Texturing



```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE2D, tex[0]);
texLoc1 = glGetUniformLocation
               (program, "tex1");
glUniform1i(texLoc1, 0);
```

```
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE2D, tex[1]);
texLoc2 = glGetUniformLocation
               (program, "tex2");
glUniform1i(texLoc2, 1);
```

Texture Coordinates

```
glBindBuffer(GL_ARRAY_BUFFER, vboID[2]);
glBufferData(GL_ARRAY_BUFFER, num* sizeof(float), texC, GL_STATIC_DRAW);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 0, NULL);
glEnableVertexAttribArray(2);
```

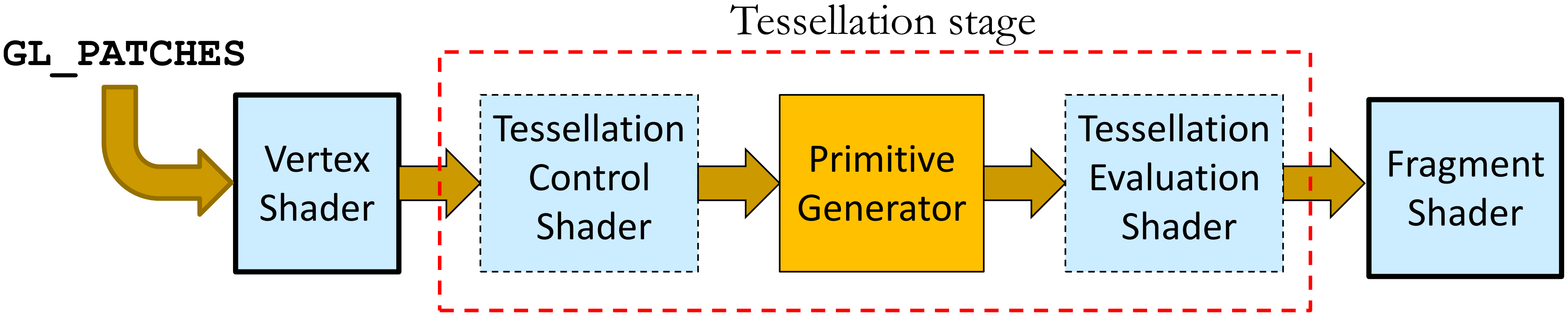
Multi-Texturing

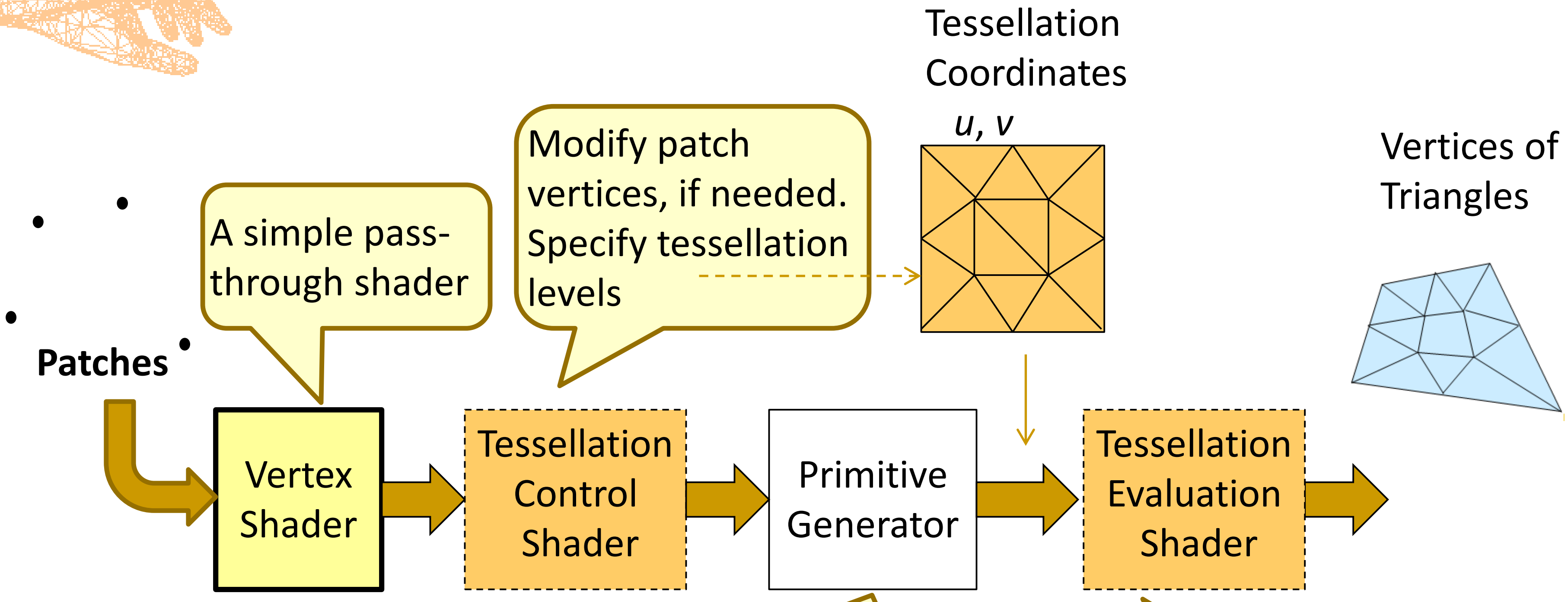
Fragment Shader:

```
uniform sampler2D tex1;
uniform sampler2D tex2;

in vec4 diffRefl;
in vec2 TexCoord;

void main()
{    textures
      vec4 tColor1 = texture(tex1, TexCoord);
      vec4 tColor2 = texture(tex2, TexCoord);

      gl_FragColor =  diffRefl*(0.8*tColor1+ 0.2*tColor2);
}
                output
                 color
```

Tessellation

- The tessellation stage of the OpenGL-4 pipeline can be used to generate a **mesh of triangles** based on vertices of a **patch** (a new geometric primitive). patch
- There are two shading stages used in tessellation:
  - Tessellation controller (optional): Sets tessellation parameters and any additional patch vertices.
  - Tessellation evaluator: Positions the vertices of the generated mesh on the patch using mapping equations defined by user.





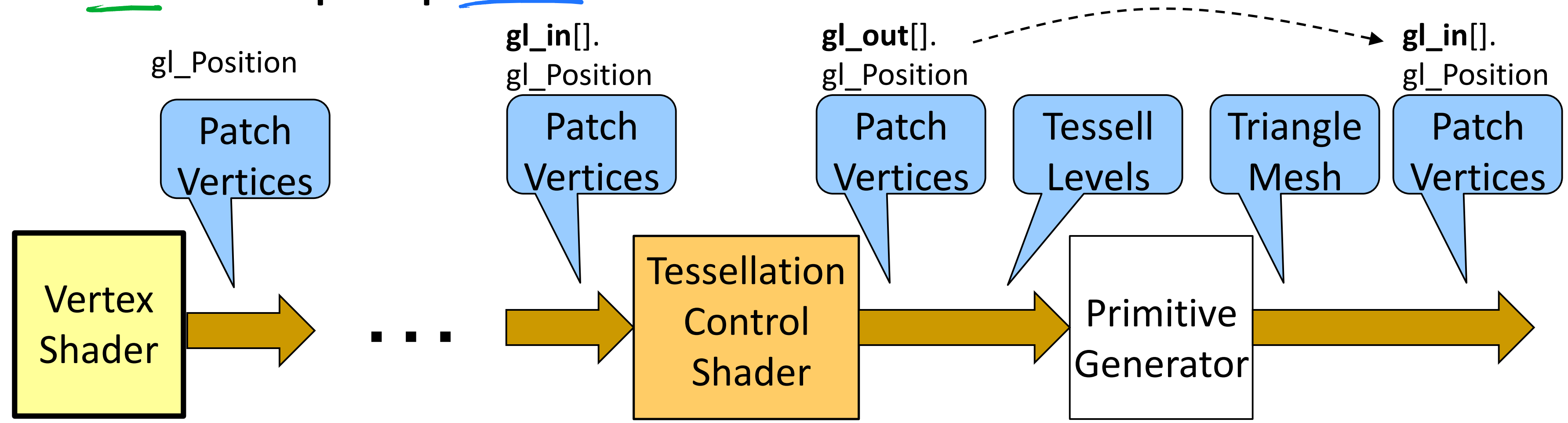
Tessellation Coordinates
u, *v*
Vertices of Triangles
Patches
A simple pass-through shader
Modify patch vertices, if needed. Specify tessellation levels
Vertex Shader
Tessellation Control Shader
Primitive Generator
Tessellation Evaluation Shader

- The primitive generator can output only **triangles**.
- The triangles form a tessellation of either a square or a triangle domain.
- The vertices of every triangle in the tessellation will have normalized coordinates.

The evaluation shader converts the primitive vertices (*u*, *v*) to 3D points using use-defined functions, and outputs them in clip coordinate space.

Tessellation Control Shader (TCS)

- The tessellation control shader is commonly used to set the inner and outer tessellation levels.
- Optionally, the shader can also create new or remove existing patch vertices. *All* patch vertices are available inside the shader in an array.
- The tessellation control shader will execute once for each output patch vertex.



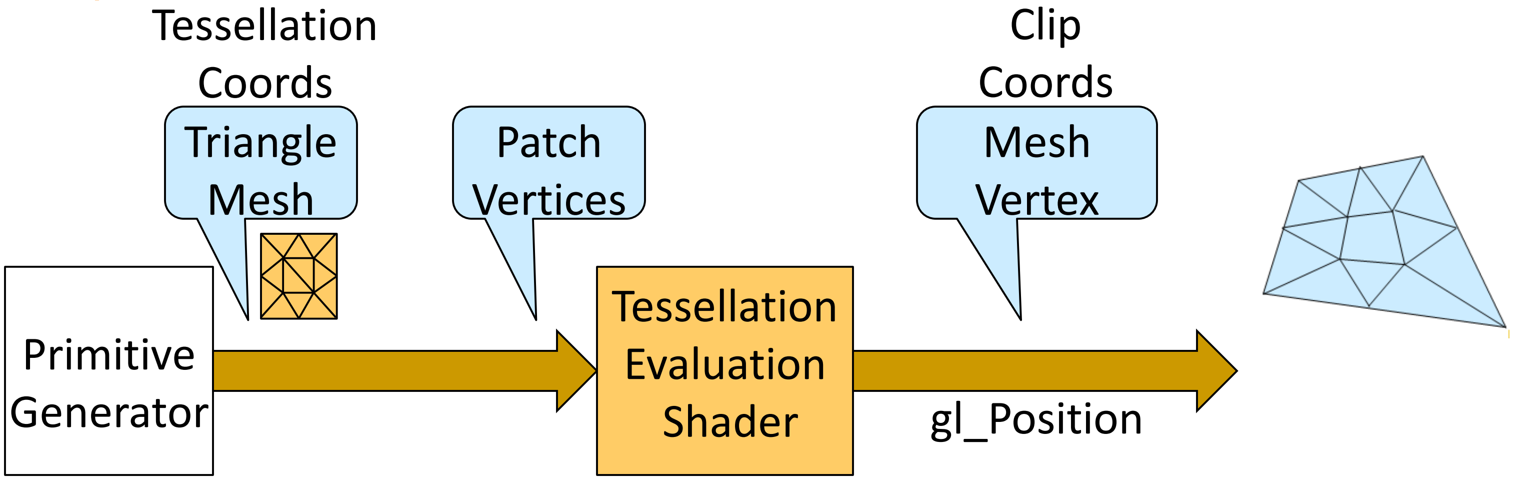
Tessellation Control Shader: Example

output patch vertices

Index of the current out vertex

```
#version 400
layout(vertices = 9) out;

void main()
{
    gl_out[gl_InvocationID].gl_Position
          = gl_in[gl_InvocationID].gl_Position;
    gl_TessLevelOuter[0] = 6;
    gl_TessLevelOuter[1] = 6;
    gl_TessLevelOuter[2] = 6;
    gl_TessLevelOuter[3] = 6;
    gl_TessLevelInner[0] = 5;
    gl_TessLevelInner[1] = 5;
}
```

Tessellation Evaluation Shader



Tessellation Coords

Triangle Mesh

Patch Vertices

Clip Coords

Mesh Vertex

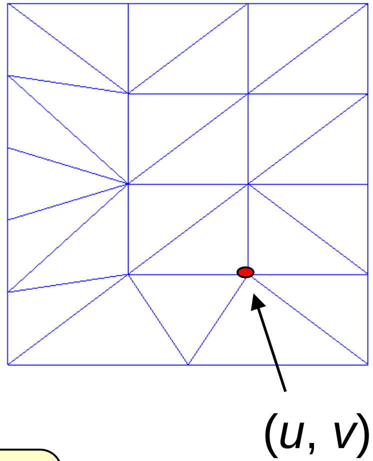
Primitive Generator

Tessellation Evaluation Shader

gl_Position

- The primitive generator emits a triangle mesh with vertices defined in a normalized domain. These coordinates are referred to as tessellation coords.
- The tessellation evaluator repositions each mesh vertex using patch vertices, and outputs them in clip coordinates.
- The evaluation shader executes once for each mesh vertex.

Tessellation Evaluation Shader



Domain

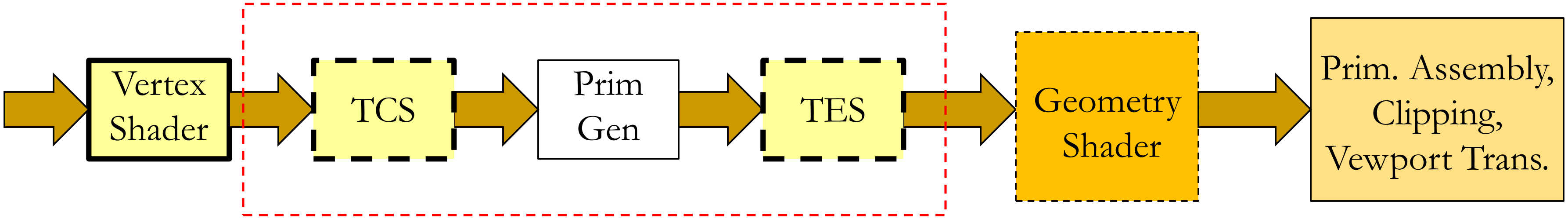
```
#version 400
layout(quads, equal_spacing, ccw) in;
uniform mat4 mvpMatrix;
vec4 posn;

void main()
{
    float u = gl_TessCoord.x;
    float v = gl_TessCoord.y;
    posn = (1-u)* (1-v) * gl_in[0].gl_Position
         + u * (1-v) *    gl_in[1].gl_Position
         + u * v *        gl_in[2].gl_Position
         + (1-u) * v *    gl_in[3].gl_Position;
    gl_Position = mvpMatrix * posn;
}
```

Tessellation coords

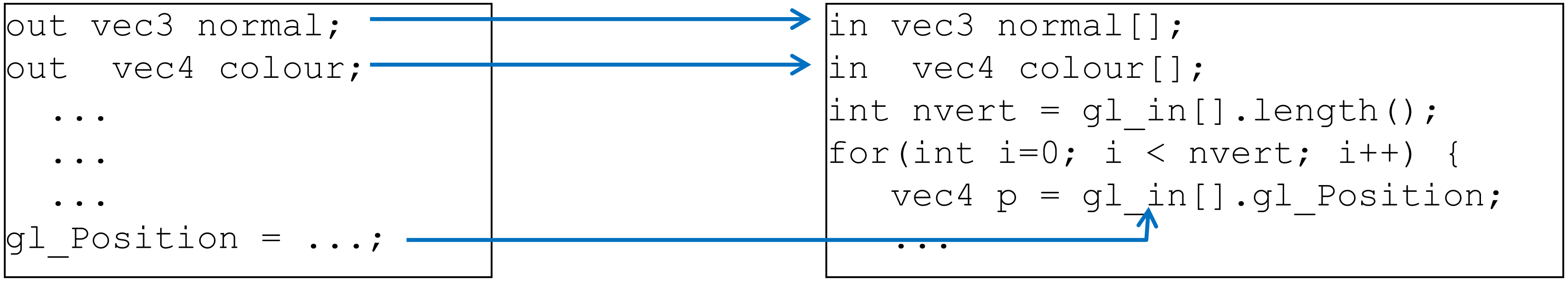
Patch vertices

Clip Coords

OpenGL Geometry Shader



- The geometry shader receives inputs from either TES or the vertex shader (if tessellation is not active).
- The geometry shader <u>receives inputs in arrays</u> with values corresponding to one whole primitive.
- A geometry shader can thus process an entire primitive.



```
out vec3 normal;
out  vec4 colour;
  ...
  ...
  ...
gl_Position = ...;
```

Myshader.vert

```
in vec3 normal[];
in  vec4 colour[];
int nvert = gl_in[].length();
for(int i=0; i < nvert; i++) {
   vec4 p = gl_in[].gl_Position;
   ...
```

Myshader.geom

Geometry Shader

- A geometry shader runs once per input primitive.
- A geometry shader can change the type of the input primitive and the number of vertices.
- It can also be used to discard primitives.
- A geometry shader can be made to execute a number of times for the same input primitive (instancing).
- The geometric shader is the last shader stage before clipping and rasterization. It must emit each vertex in the clip coordinate space using the built-in variable gl_Position.

Geometry Shader

- Input primitives:
  - points: GL_POINTS
  - lines: GL_LINES, GL_LINE_STRIP, GL_LINES_ADJACENCY
  - triangles: GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_TRAIANGLES_ADJACENCY

Application

```
glDrawArrays(GL_LINES,...);
```

- Output primitives:
  - points
  - line_strip
  - **triangle_strip**

Geometry shader

```
layout(triangle_strip,..)out;
```

Geometry Shader

- A geometry shader has two built-in functions `EmitVertex()` and `EndPrimitive()`.
- In one execution of the geometry shader, it can produce multiple vertices and multiple primitives (multiple triangles as part of a triangle strip).
- Each call to `EmitVertex()` appends one vertex at the end of the current primitive.
- After calling `EmitVertex()`, the values of all out variables become undefined.
- If the geometry shader exits without calling `EmitVertex()`, then it does not produce any output primitive.

Geometry Shader Output Example

Producing a triangle strip:

```
gl_Position = ...;  //Point 1
oColor = ...;
EmitVertex();
gl_Position = ...;  //Point 2
oColor = ...;
EmitVertex();
gl_Position = ...;  //Point 3
oColor = ...;
EmitVertex();
gl_Position = ...;  //Point 4
oColor = ...;
EmitVertex();
gl_Position = ...;  //Point 5
oColor = ...;
EmitVertex();
gl_Position = ...;  //Point 6
oColor = ...;
EmitVertex();
EndPrimitive();
```