Intermediate Graphics & Animation Programming

GPR-300
Daniel S. Buckstein

Geometry Manipulation
Week 7

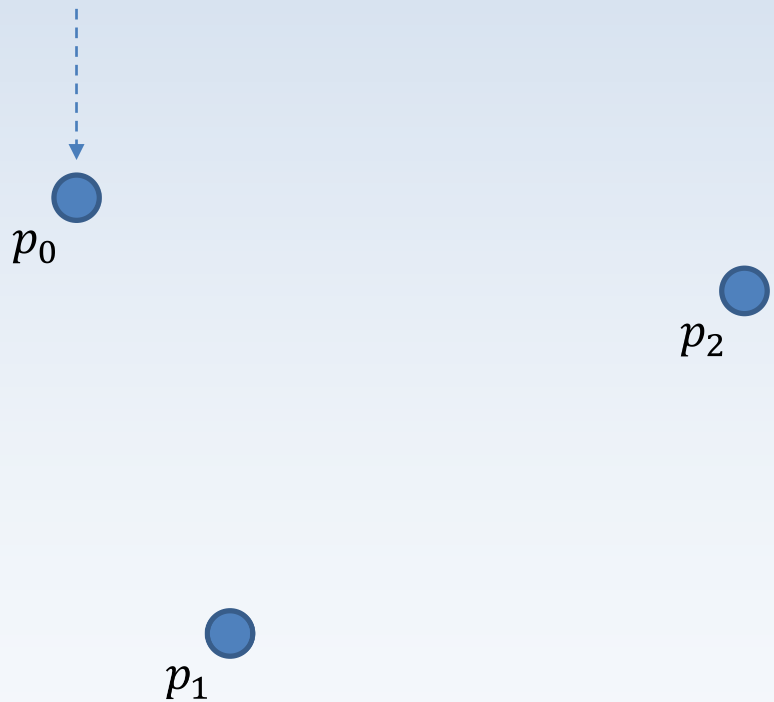
License

Geometry Manipulation

- Key graphics terminology
  - Vertex, attribute, primitive, fragment, pixel
  - Vertex Buffer Objects (VBOs)
  - Vertex Array Objects (VAOs)
  - Index/Element Buffer Objects (IBO/EBOs)
- Geometry shaders
  - Key terms & syntax
  - Examples
- Tessellation shaders

Key Terms

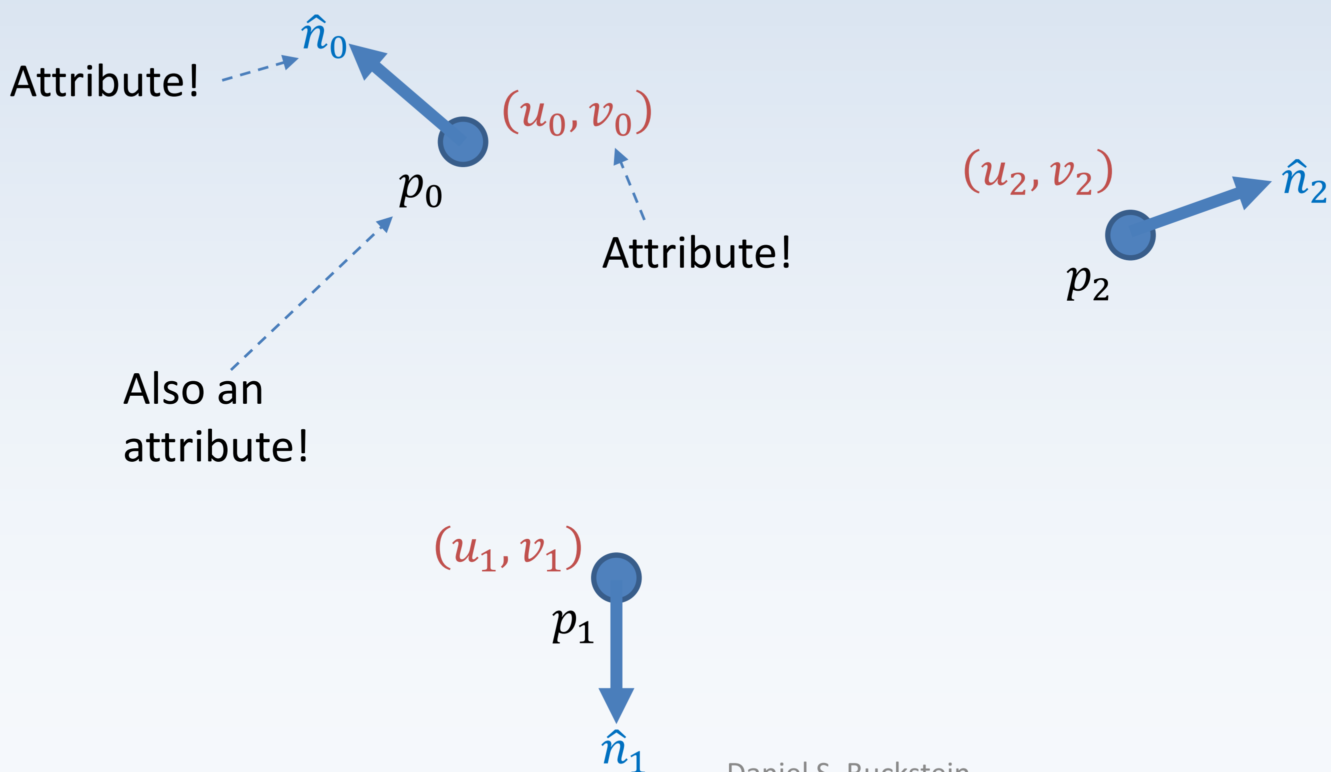
- Important definitions and where they come into play: ***<u>vertex</u>***



TL;DR:
They are just *points in 3D*.

Key Terms

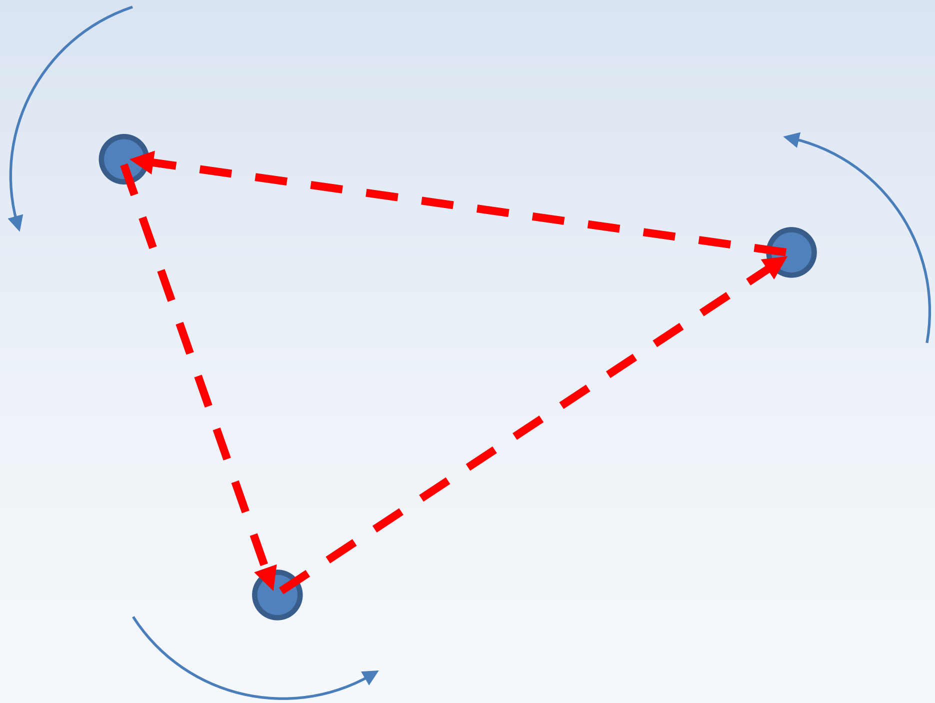
- Important definitions and where they come into play: ***attribute***



Anything pertinent to our *points in 3D*.

Key Terms

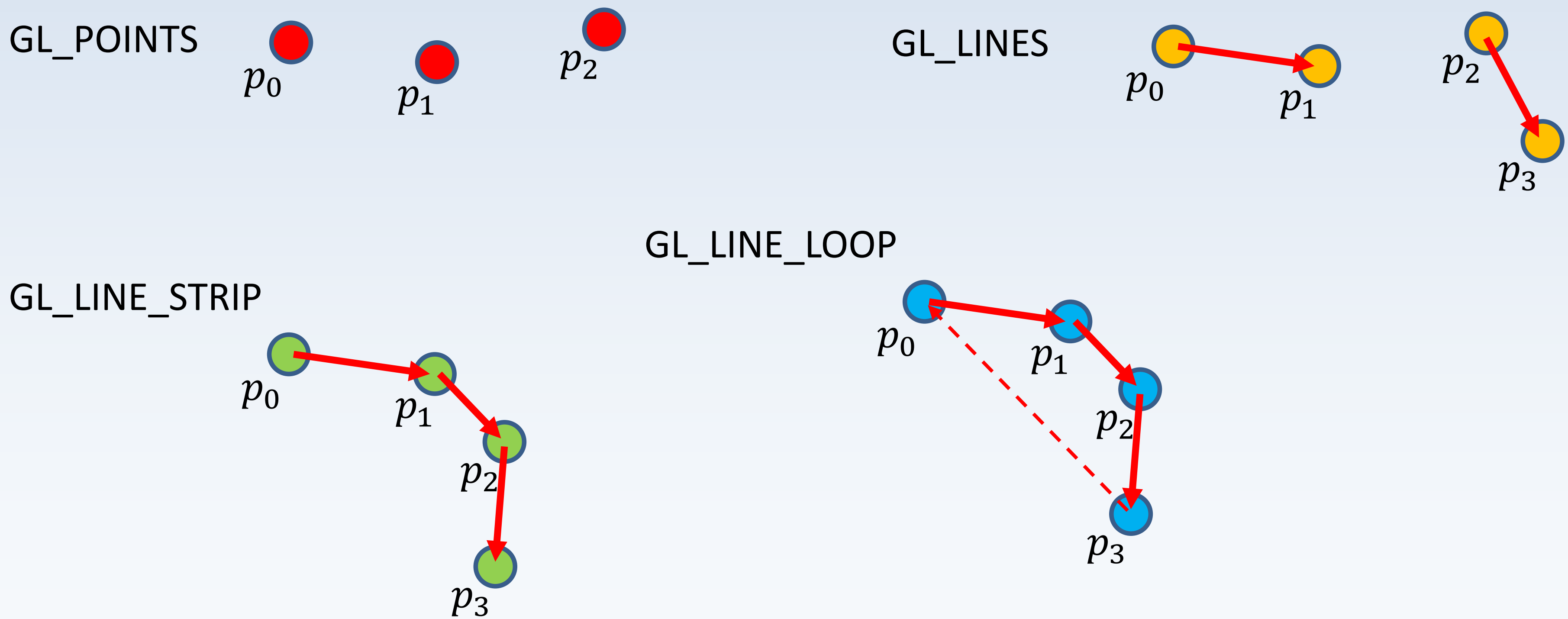
- Important definitions and where they come into play: ***primitive***



Shapes put together from *points in 3D*.

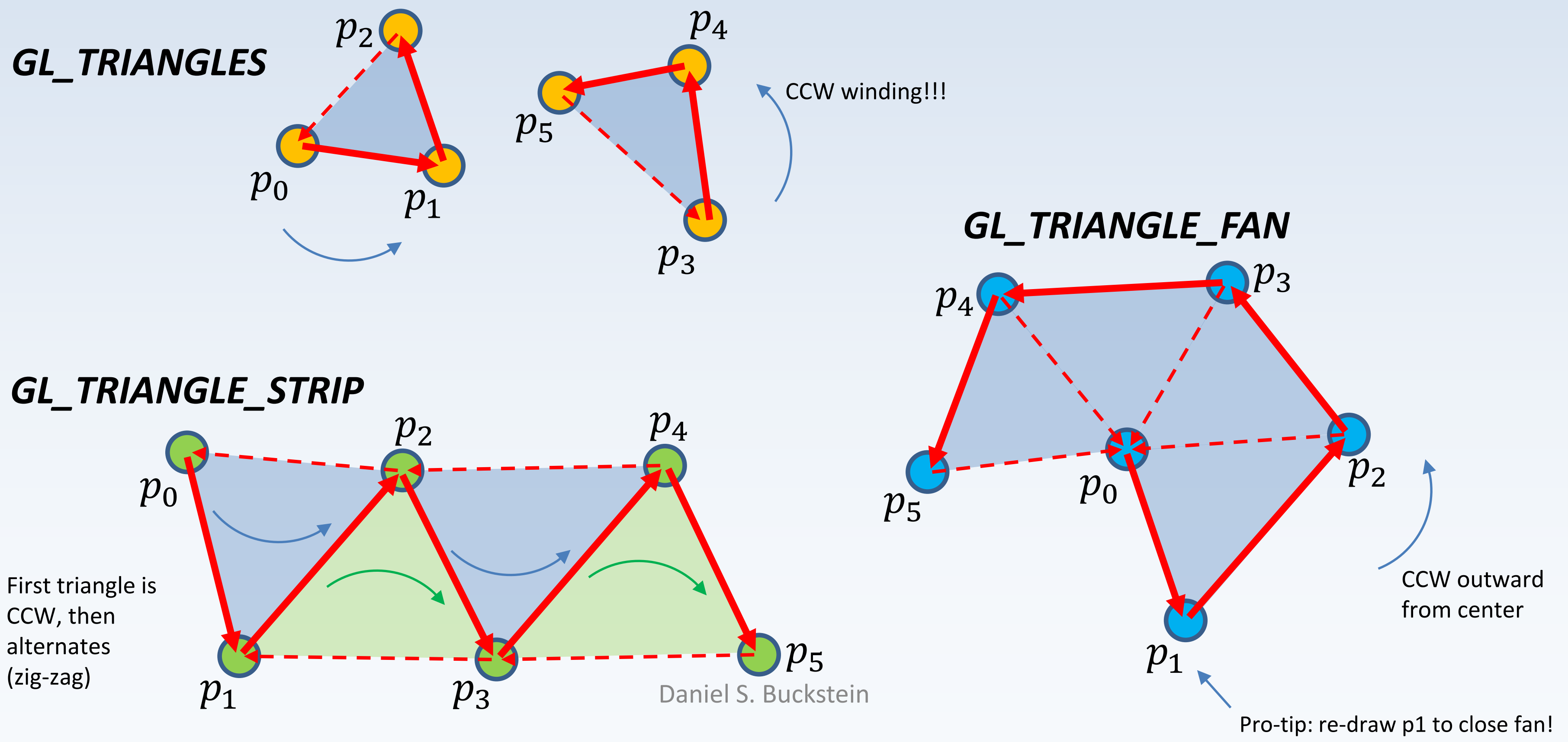
Key Terms

- Important definitions and where they come into play: ***primitive***



Key Terms

- Important definitions and where they come into play: ***primitive***

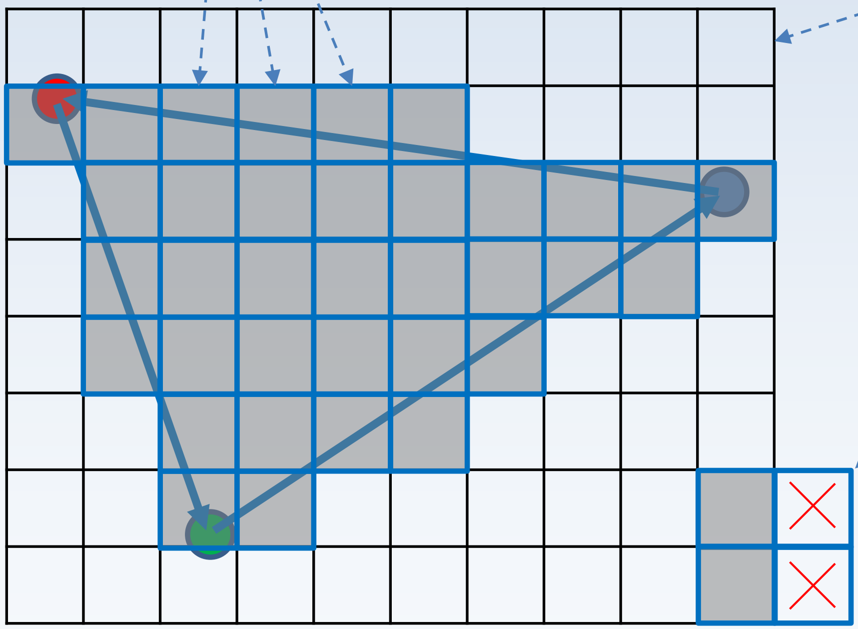


Daniel S. Buckstein

Key Terms

- Important definitions and where they come into play: ***fragment***

Clipping & rasterization:

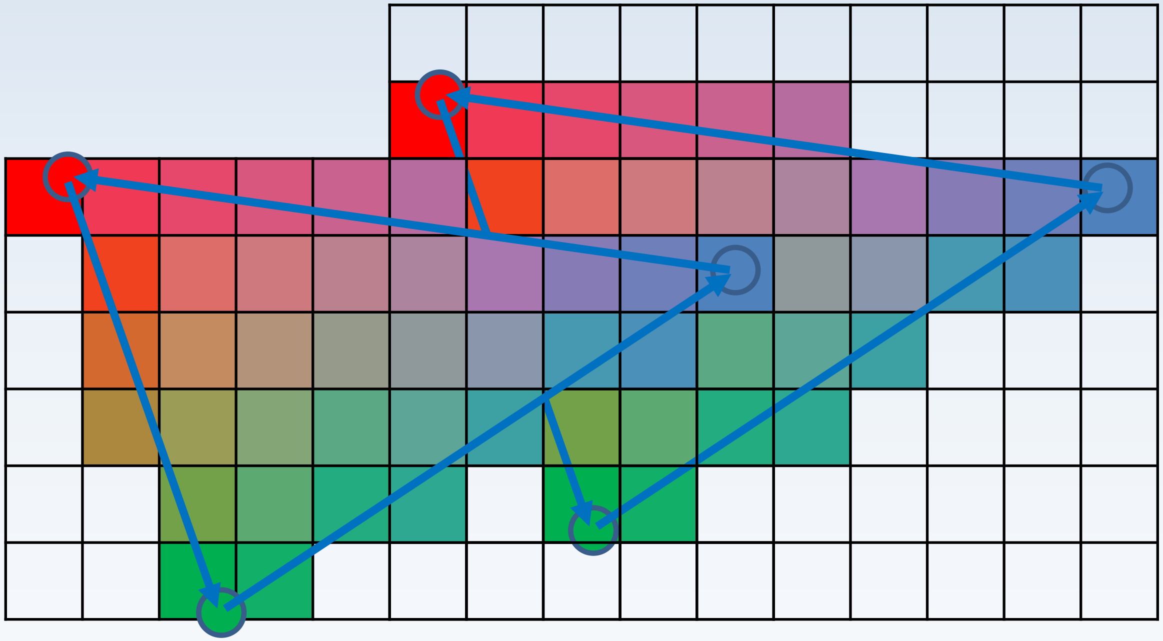


This *3D grid* is called the *raster*

No geometry outside raster will generate fragments… called *clipping*

Key Terms

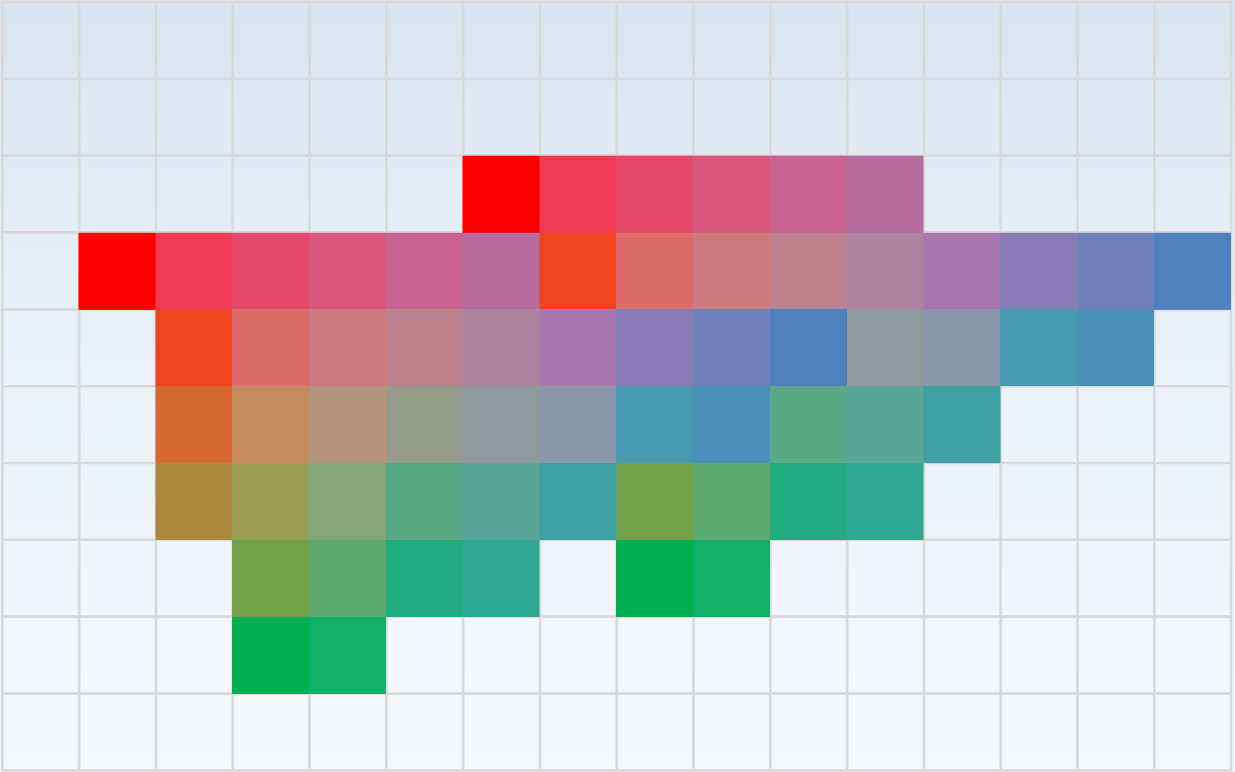
- Important definitions and where they come into play: ***fragment***:
    - Values in-between vertices are ***interpolated***



Fragments are ***STILL 3D***. Think of a frag as the smallest visible 3D data on the screen.

Key Terms

- Important definitions and where they come into play: ***pixel***



Pixels are ***2D***

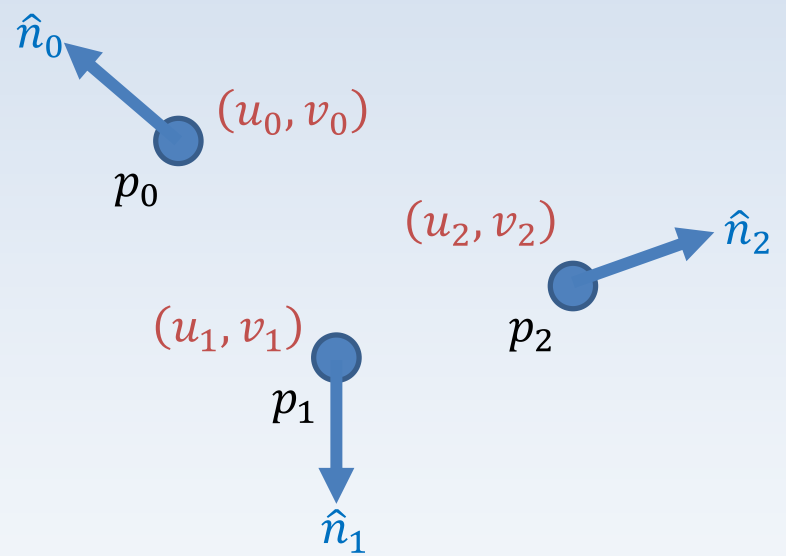
(…finally)

Buffers & Data Flow

- Immediate mode (bad): all attributes defined per-vertex *when* the vertex is to be processed
- Send data to GPU *immediately*
- Discarded *immediately* after use ☹
- Vertex Buffer Object (VBO): stores vertex attributes!
- “Retained mode”: prepare first, send to GPU for *holding* (good)

Buffers & Data Flow

- How many VBOs can you have for a *single renderable object*???



One VBO *per-attribute*:

p_{0_x}	p_{0_y}	p_{0_z}	p_{1_x}	p_{1_y}	p_{1_z}	p_{2_x}	p_{2_y}	p_{2_z}
$\hat{n}_{0_x}$	$\hat{n}_{0_y}$	$\hat{n}_{0_z}$	$\hat{n}_{1_x}$	$\hat{n}_{1_y}$	$\hat{n}_{1_z}$	$\hat{n}_{2_x}$	$\hat{n}_{2_y}$	$\hat{n}_{2_z}$
u_0	v_0	u_1	v_1	u_2	v_2			

One VBO for *all attributes*:

p_{0_x}	p_{0_y}	p_{0_z}	p_{1_x}	p_{1_y}	p_{1_z}	p_{2_x}	p_{2_y}	p_{2_z}	$\hat{n}_{0_x}$	$\hat{n}_{0_y}$	$\hat{n}_{0_z}$	$\hat{n}_{1_x}$	$\hat{n}_{1_y}$	$\hat{n}_{1_z}$	$\hat{n}_{2_x}$

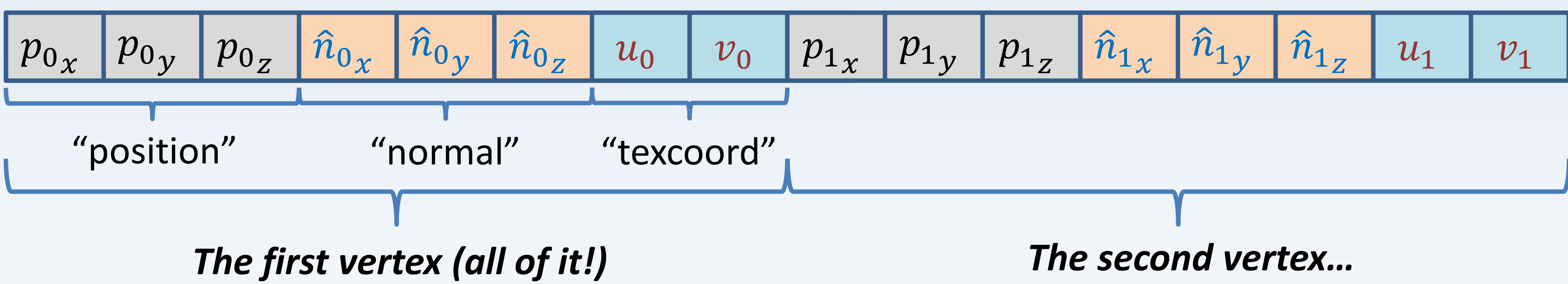
One or more *interleaved* VBOs:

p_{0_x}	p_{0_y}	p_{0_z}	$\hat{n}_{0_x}$	$\hat{n}_{0_y}$	$\hat{n}_{0_z}$	u_0	v_0	p_{1_x}	p_{1_y}	p_{1_z}	$\hat{n}_{1_x}$	$\hat{n}_{1_y}$	$\hat{n}_{1_z}$	u_1	v_1

Buffers & Data Flow

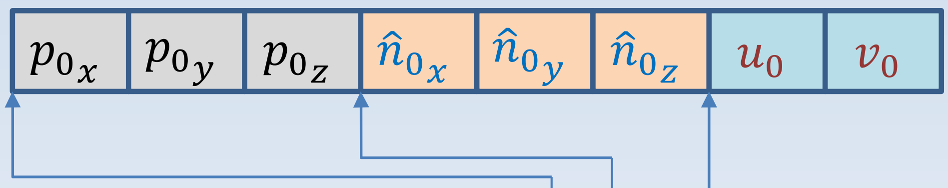
- A shader program eats raw data:
- API defines geometry, stores it in ***vertex buffer***
- Vertex shader reads this ***only as you tell it to!***

Data in *interleaved* VBO:



Buffers & Data Flow

- *Vertex shaders*: responsible for *reading* the vertex data:




```
layout (location = 0) in vec3 pos;
layout (location = 2) in vec3 nrm;
layout (location = 8) in vec2 uv;
```

- VAO defines locations and attribute sizes
  - (i.e. the data format)

Buffers & Data Flow

- Setting up a VBO:

```
// generate a VBO
glGenBuffers( 1, &bufferHandle );
// bind generated buffer to make changes to it
glBindBuffer( GL_ARRAY_BUFFER, bufferHandle );
// place raw data in buffer
glBufferData( GL_ARRAY_BUFFER,
    bufferSize, dataPtr, GL_STATIC_DRAW );
// disable ALL attribute buffers
glBindBuffer( GL_ARRAY_BUFFER, 0 );
```

Buffers & Data Flow

- How are VBOs used?
- ***States***: retained attributes are *enabled* and *disabled*
- Attributes for a single vertex are linked together using *states*
- States can be *saved* instead of enabling and disabling before and after every usage…
- ***Vertex Array Object (VAO)***

Buffers & Data Flow

```
// generate a VAO ("state machine")
glGenVertexArrays( 1, &arrayHandle );
// bind generated VAO to make changes to it
glBindVertexArray( arrayHandle );


// next slide: how to associate VBO with VAO,
assuming the above two lines were last called
```

Buffers & Data Flow

```
// bind VBO to be associated
glBindBuffer( GL_ARRAY_BUFFER, bufferHandle );
// enable and configure each attribute!!!
glEnableVertexAttribArray( 2 ); // e.g. normals
glVertexAttribPointer( 2, // ^same as enabled
    3, // number of elements for this attribute
    GL_FLOAT, // core data type of attribute
    GL_FALSE, // GPU normalization, KEEP FALSE!
    0,        // STRIDE  ****VERY IMPORTANT****
    (char *)(slot0size+slot1size) ); // offset
```

Buffers & Data Flow

```
// how to draw using a prepared VAO: 2 LINES!!!

// STEP 1: enable VAO, *VBO IS ALREADY BOUND!*
glBindVertexArray( arrayHandle );
// STEP 2: draw!
glDrawArrays(   GL_TRIANGLES, // primitive
                0,            // start vertex
                numVerts );   // how many verts

// pssst: don't forget to disable when all done
glBindVertexArray( 0 );
```

Buffers & Data Flow

- Further optimization: IBO/EBO
- Index Buffer Object/Element Buffer Object
- Useful for geometry with ***repeated vertices***
- Reduces the size of your VBO: just use a set of *indices* to tell OpenGL the *order* in which it should process the vertices in a VBO! ☺
- Your VAO will remember an IBO just like it remembers a VBO!

Buffers & Data Flow

- IBO generation is the same, but it is bound to the target `GL_ELEMENT_ARRAY_BUFFER` instead of `GL_ARRAY_BUFFER`
- Example of how to fill IBO with raw data:

```
// raw data
const int indices[] = { 0, 1, 2, 3, 2, 1 };
// fill currently-bound IBO:
glBufferData( GL_ELEMENT_ARRAY_BUFFER,
    sizeof(indices), indices, GL_STATIC_DRAW );
```

Buffers & Data Flow

```
// how to draw using a prepared VAO with an
//  index buffer attached:
// STEP 1: enable VAO, VBO & IBO already bound!
glBindVertexArray( arrayHandle );
// STEP 2: draw!
glDrawElements( GL_TRIANGLES, // primitive
                numVerts,     // how many
                GL_INT,       // index type
                0 ); // offset to first element
```

Buffers & Data Flow

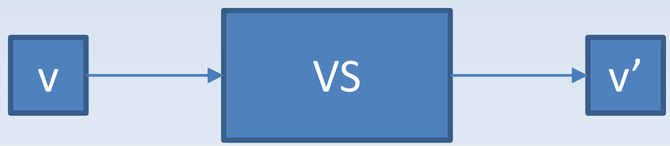
- *Even further* optimization: ***instanced*** draw call
- If you have a batch of renderables that are identical, use these functions to draw:
  – (the parameters are identical but there is an extra one at the end: how many time to draw the VAO)

```
glDrawArraysInstanced(..., primCount);

glDrawElementsInstanced(..., primCount);
```

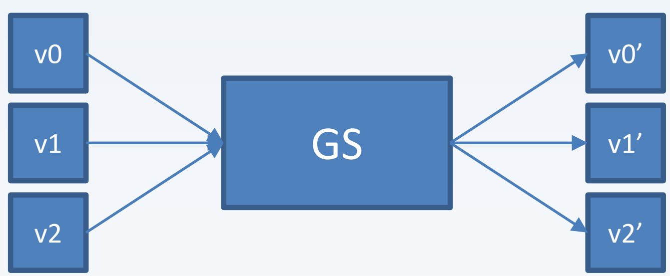
Geometry Shaders

- Vertex shaders receive and process data for *one vertex at a time*:



General rule: Vertex in Vertex out

- Geometry shaders receive and process data for *multiple vertices*, depending on *primitive*:



Example usage: Triangle in Triangle out

Geometry Shaders

- Geometry shader processes *primitives*
- Can be used to create, modify or remove geometry
- Can even convert from one primitive type to another!
  - E.g. triangle → line strip
  - E.g. point → triangle strip
  - E.g. lines → points

Geometry Shaders

- Geometry shader example: *pass-through triangles* (just copy clip position)

```
layout (triangles) in;
layout (triangle_strip, max_vertices = 3) out;
void main() {
    for (int i = 0; i < 3; ++i) {
        gl_Position = gl_in[i].gl_Position;
        EmitVertex();
    }
    EndPrimitive();
}
```

Geometry Shaders

- Geometry shader requirements:
- If VS does not set `gl_Position`, it must be done by the end of GS
- Emit enough vertices to produce the output primitive (e.g. triangle = 3 vertices)
- The *abridged* vertex pipeline:
- VS → GS → clipping → rasterization → FS

Tessellation Shaders

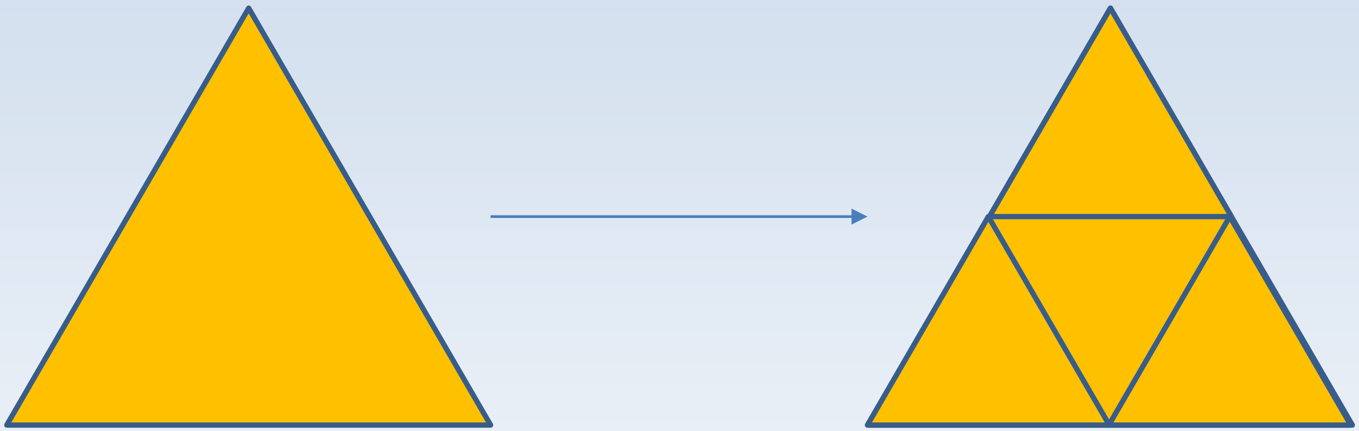
- The *complete* vertex pipeline:
- VBO → VS → TCS → TES → GS → clipping → rasterization → FS → FBO
- VBO: “it’s just data” (attribute storage)
- VS: vertex shader interprets attribute data
- TCS: tessellation ctrl. shader generates “control points”
- TES: tessellation eval. shader… tessellates

Tessellation Shaders

- The *complete* vertex pipeline:
    - (cont'd)
- GS: geometry shader generates and/or manipulates primitives
- Clipping: reform "out-of-bounds" geometry
- Raster: discretize geometry into fragments
- FS: fragment shader colors each fragment
- FBO: store fragments as pixels in images

Tessellation Shaders

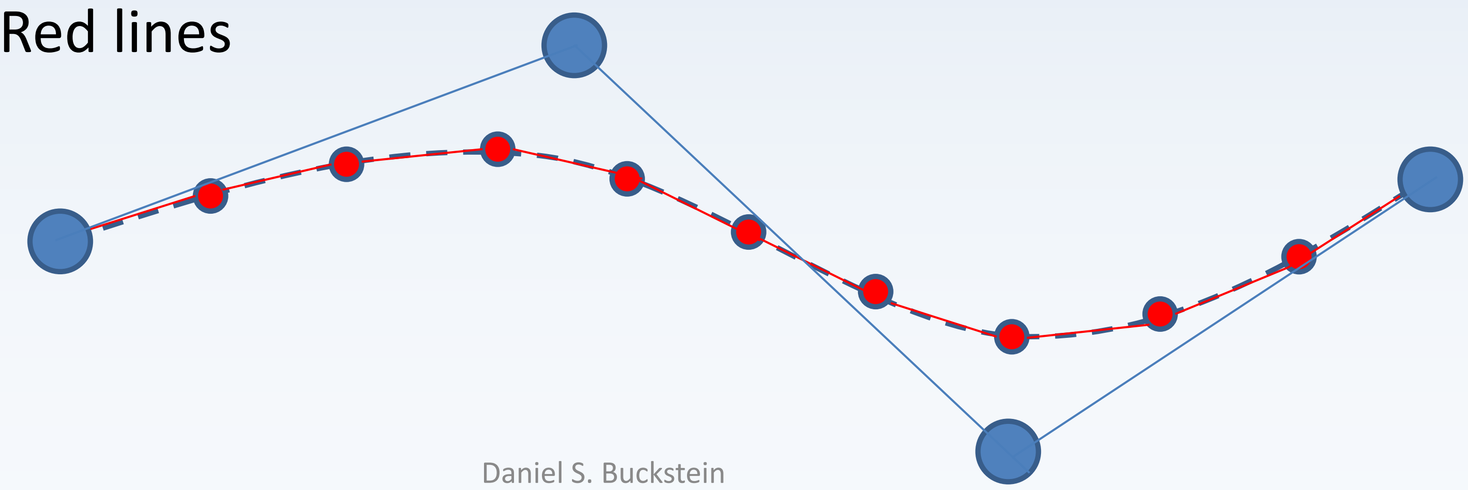
- Subdivision:



- Can be used for higher-definition geometry, curves, NURBS, LOD, etc.
- Geometry shader is final stop for geometry

Tessellation Shaders

- TCS e.g.: generate control values for a curve
  - Blue dots
- TES e.g.: generate vertices to form curve
  - Red dots
- GS e.g.: process resulting line segments
  - Red lines



The end.

- Questions? Comments? Concerns?