Chapter 2a
CSCI-396
Jeff Bush

# Rendering

# Image Formation



- Using the Synthetic Camera Model
- Objects are processed one at a time
  - In the order they are generated by the application
- Basic pipeline architecture:



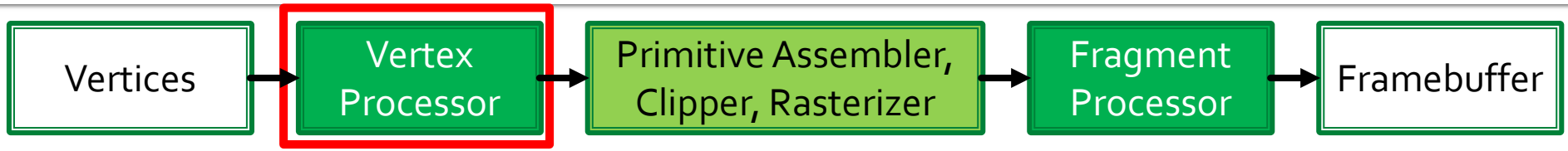



- All steps can be implemented on the GPU

- *Note that this is complicated but important – we will go through it 4 times right now!*

# Vertex Processing






- Input:
  - Vertex Buffer Objects / Attributes
- Outputs all properties of each vertex:
  - Required: position (in camera/eye coordinates)
  - Optional:
    - Color
    - Normal
    - Material
    - Texture coordinates
    - …

# Primitive Assembly



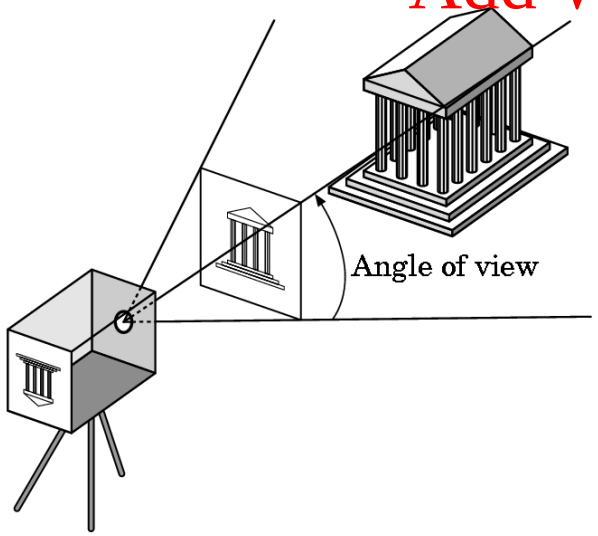

- Vertices are collected into geometric objects
  - Points (1 vertex)
  - Line segments (2 vertices)
  - Triangles (3 vertices)

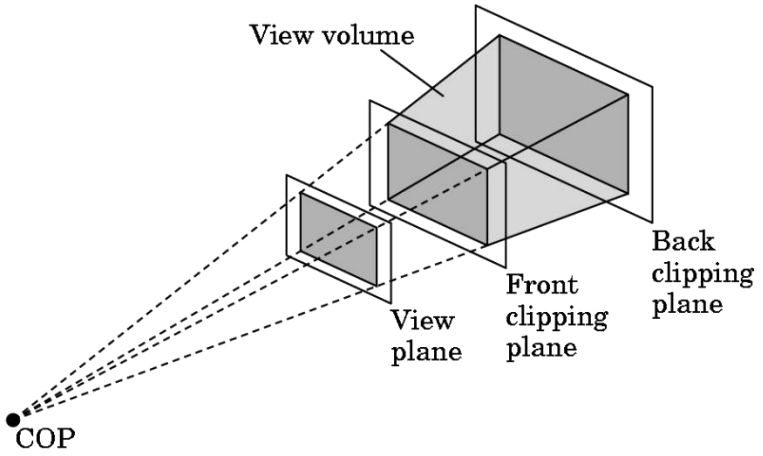
# Clipping




- A camera cannot see the whole world
- Objects outside of the visible world are *clipped* out of the scene for efficiency





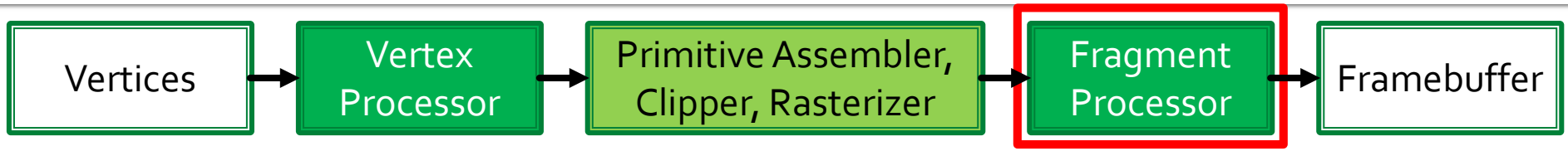
# Rasterization




- If an object is not clipped out, it may contribute to the appropriate pixels in the framebuffer
- Rasterizer produces a set of *fragments* for each object
  - Converts eye coordinates to window coordinates
- *Fragments* are "potential output pixels"
  - They have a pixel location and a depth
    - Depth being the distance from the camera
  - Any other vertex values are *interpolated*

# Fragment Processing

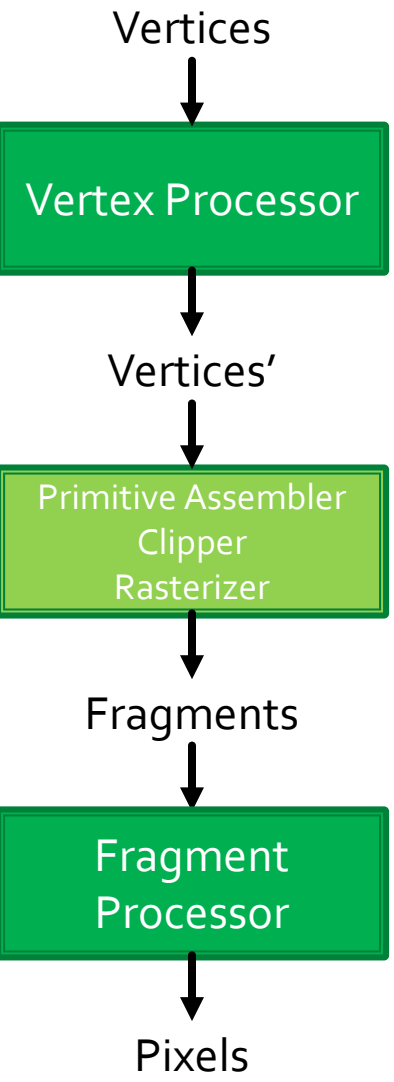





- Each fragment needs to be processed to determine the color of the corresponding pixel
- Colors are determined by computations, interpolation of vertex values, and/or texture mapping
- Fragments may be blocked (and thus hidden) by other fragments with a lower depth (closer to the camera)
  - But fragments can be partially/fully transparent so cannot throw out deeper fragments until after a color is assignment to every fragment

# Pipeline Overview






From the JS code, sent to the GPU through buffers. Each vertex has any number of ***attributes***: position, color, material, …

Behavior defined with ***vertex shader*** code.



Vertex processor generates camera/eye coordinates for each vertex.
It may generate additional values for each vertex.





Behavior built into OpenGL/GPU (can be overridden though).

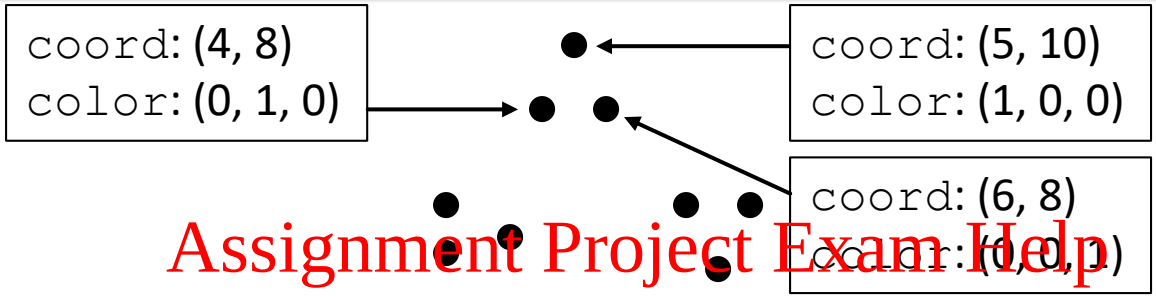
Rasterizer produces fragments with window location and depth.
Each fragment has interpolated vertex values called ***varyings***.

Behavior defined with ***fragment shader*** code.
OpenGL/GPU does final combination of fragments across depth.

The final output is colored pixels.

# Pipeline Example: Vertices







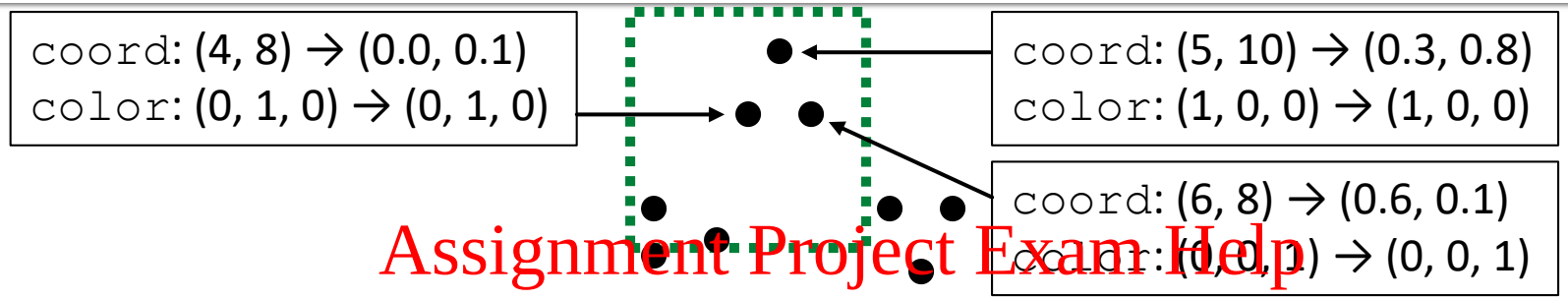

- ***Attributes*** for each vertex are stored in ***vertex buffer objects*** (***VBOs***)
  - VBOs are essentially arrays of primitive values (e.g. unsigned 8-bit ints or 32-bit floats) that are grouped in 1 to 4 components/values at a time
  - They are setup in JavaScript and then loaded into GPU RAM
- Example above would have 2 VBOs like:

```
coords = [5, 10, 4, 8, 6, 8, …]
colors = [1, 0, 0, 0, 1, 0, 0, 0, 1, …]
```

  - Coords are in groups of 2 (not always 2, we just have 2D data)
  - Colors are in groups of 3 here (not always 3, just that way for this data)
- When counting all 9 vertices, how long are each of the VBOs?

# Pipeline Example: Vertex Shader





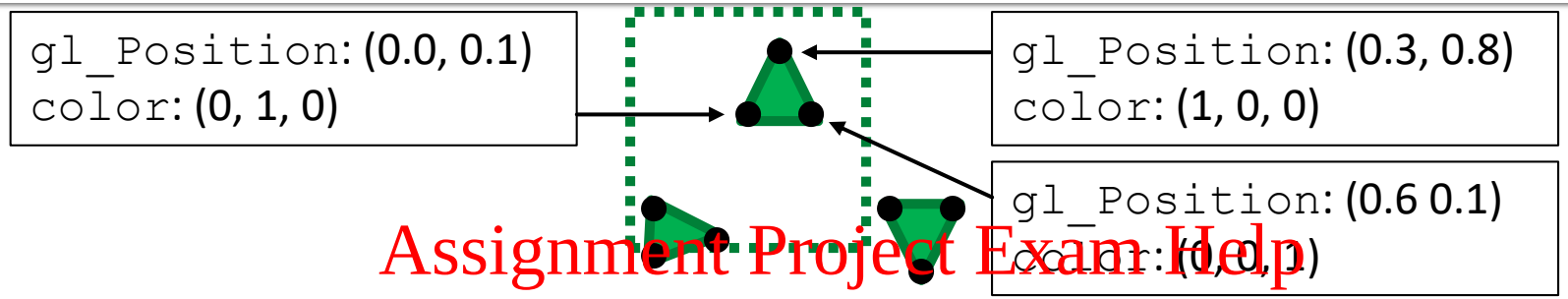





- ***Vertex shader*** takes all attributes as input and produces `gl_Position` and ***varyings*** as ouput
  - Required to set `gl_Position` (in ***clip coordinates***)
  - Other outputs are interpolated and passed to fragment shader
    - Once again these must be primitive values in groups of 1 to 4
- These vertices have 2 attributes: `coord` and `color`
  - In this simple example we compute `gl_Position` from `coord` and pass color along unchanged

# Pipeline Example: Primitive Assembly



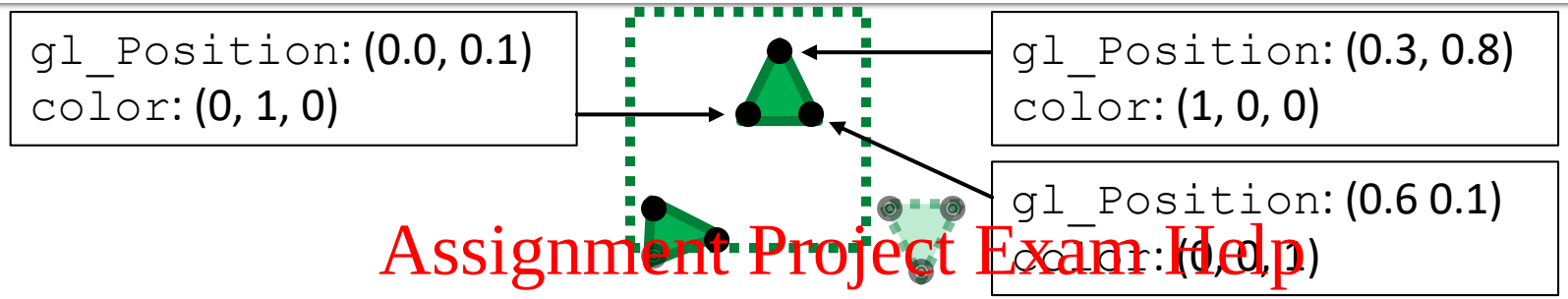





- The ***primitive assembler*** groups (i.e. assembles) vertices into basic shapes (i.e. primitives) based on how WebGL was told to render them
  - If we told WebGL to draw using `TRIANGLES` then each consecutive set of 3 vertices will be grouped into triangles
  - What do you think would happen with `LINES`?

# Pipeline Example: Clipper



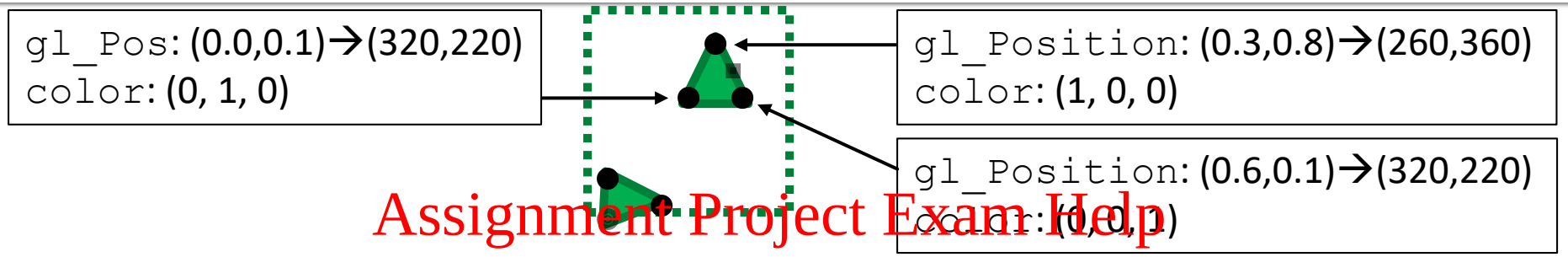





- ***Clipper*** removes all shapes outside of the ***clipping region*** (dashed line above)
  - Clipping means to remove things (like clipping out a coupon)
  - This greatly improves efficiency, in a large 3D world the vast majority of the shapes will be outside of the view
- WebGL's clipping region is from $-1$ to $1$ for $x$, $y$, and $z$
  - Center is at $(0, 0, 0)$, extends from $(-1, -1, -1)$ to $(1, 1, 1)$
- The vertex shader must convert coordinates properly so that everything that is visible lies within that region

# Pipeline Example: Rasterizer





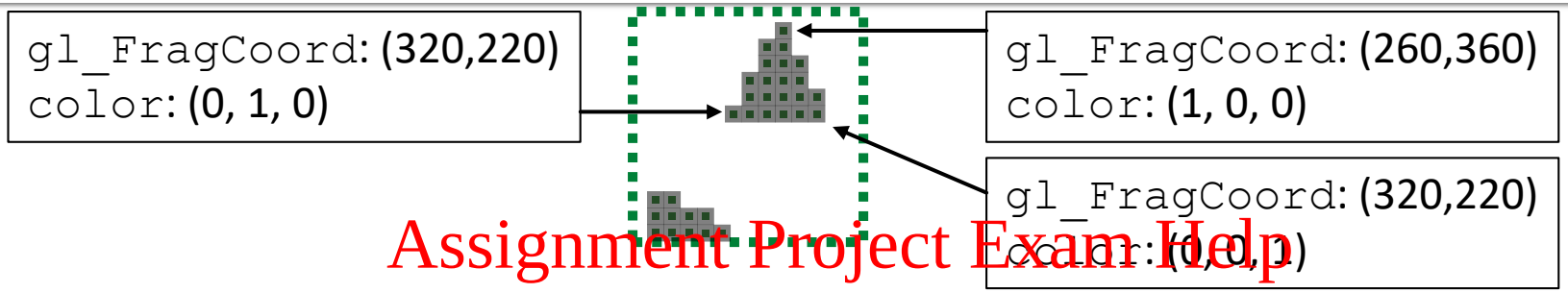



- ***Rasterizer*** creates the ***fragments***
  - Includes space between vertices that are part of shapes inside clipping region
- Each fragment has `gl_FragCoord` – position in canvas coords
  - Computations are shown above for a 400x400 Canvas – how?
- All varyings (vertex shader outputs) are ***linearly interpolated*** between vertices of the shape
  - Example: black square (single fragment) is halfway between two vertices and thus the `color` will be half of each of the colors of those vertices:

    $0.5 * (1,0,0) + 0.5 * (0,0,1) \Rightarrow (0.5,0,0.5)$ which is a dark purple
  - What about fragment one quarter of the way down? (closer to top)
  - What about fragment in the exact center of the triangle?

# Pipeline Example: Fragment Shader



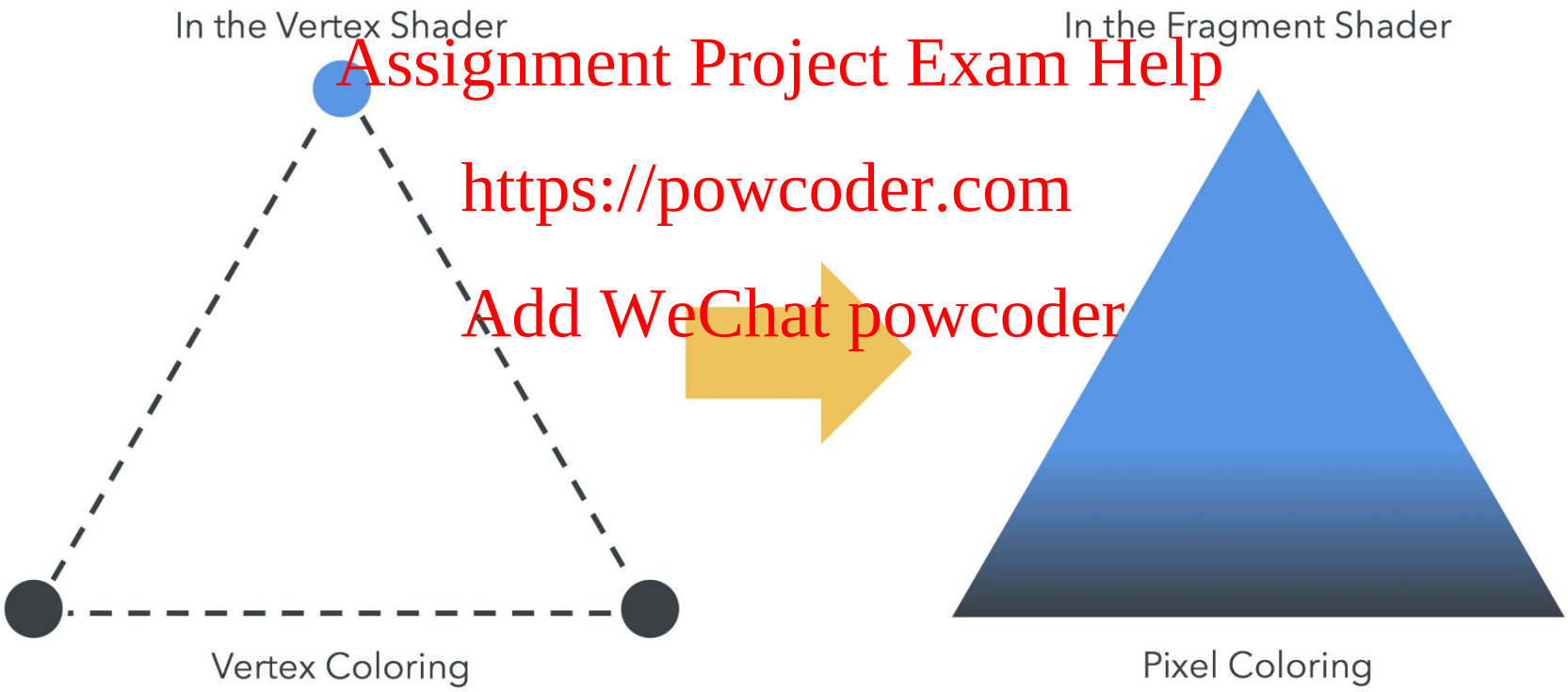





- ***Fragment shader*** computes the color and depth of each fragment
  - Input is `gl_FragCoord` and all interpolated vertex outputs
- Have to assign a color to the special `fragColor` variable
  - It's the entire purpose of the fragment shader
  - In the beginning we will just pass the incoming color as the `fragColor` but later we will apply lighting effects here
- Can optionally assign a depth to `gl_FragDepth`
  - Default is `z` value of `gl_FragCoord`

# Interpolation of Varyings





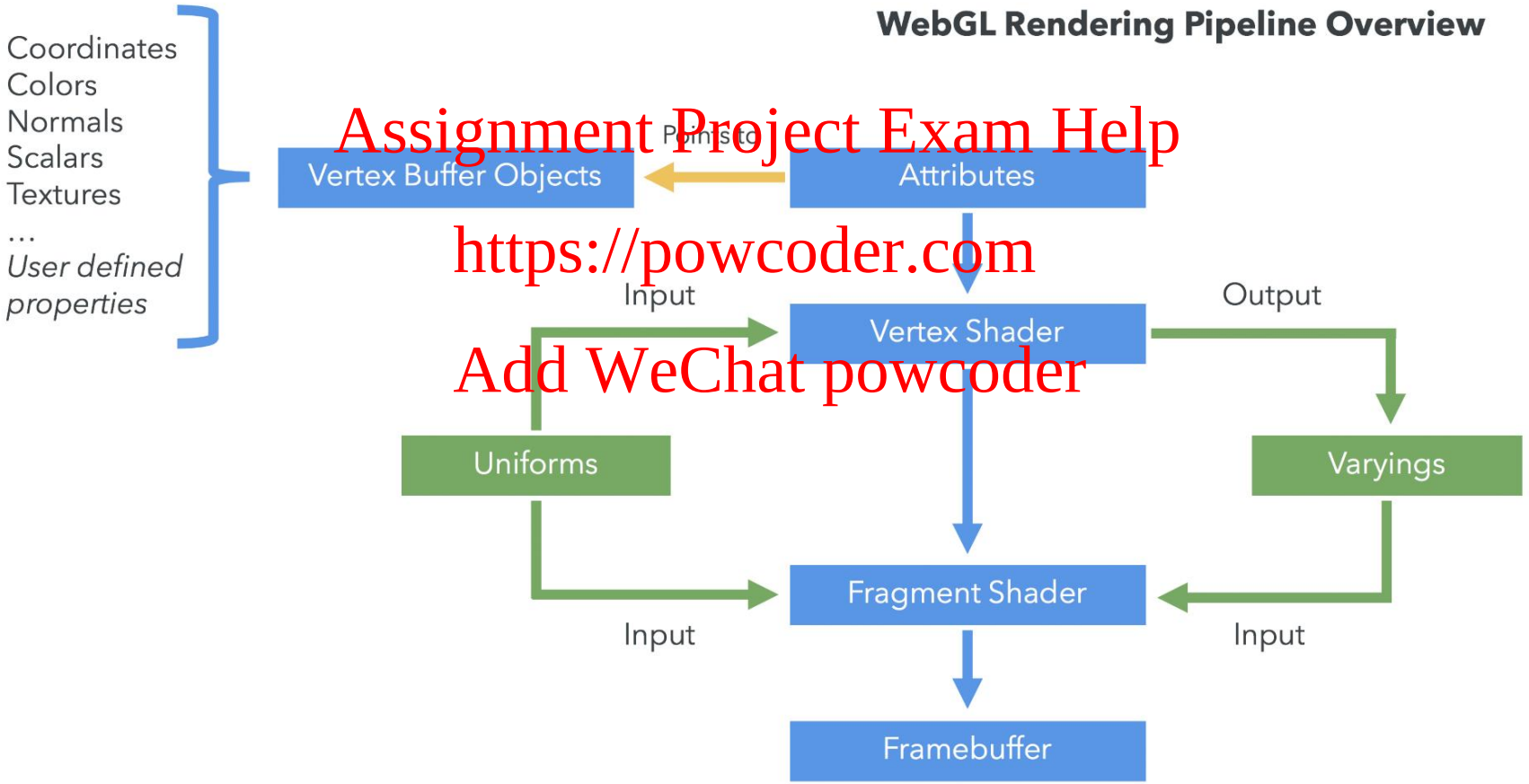
# Pipeline Example: Framebuffer



- The final processing is done after all of the fragments are completed
- The fragments for each pixel are sorted by their depth and ones that are deeper are discarded
  - Unless there are transparency effects
- The final color of each pixel is saved to the ***framebuffer*** which is displayed on the screen

# Rendering Pipeline Summary





# Coordinate Systems



- Object/world/model coordinates
  - The initial vertex coordinates of the models
  - Can have any scale/origin/orientation you want
  - In complex scenes each object will have its own set of coordinates, the object has a set of coordinates in the overall world, and the camera has its own set of coordinates
- Clip coordinates
  - The coordinates by the vertex shader
  - Everything visible must be the box from $(-1,-1,-1)$ to $(1,1,1)$
- Window coordinates
  - In pixels, coordinates relative to the canvas object in the HTML document – $(0,0)$ is the lower-left corner
  - Produced by the rasterizer and given to fragment shader

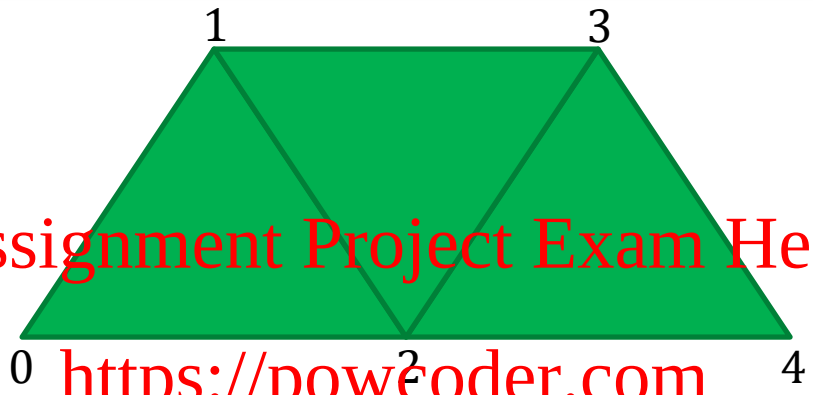
# Coordinate Systems



- Each change in coordinate system is equivalent to an ***affine transformation***
  - Similar to a ***linear transformation***
    - Matrix multiplication with a transformation matrix
- We will be constantly converting between systems
- For the time being we are going to place all of our objects in clip-space to begin with and we won't have to deal with any transformations
  - Everything should be from $(-1,-1,-1)$ to $(1,1,1)$

# Indices



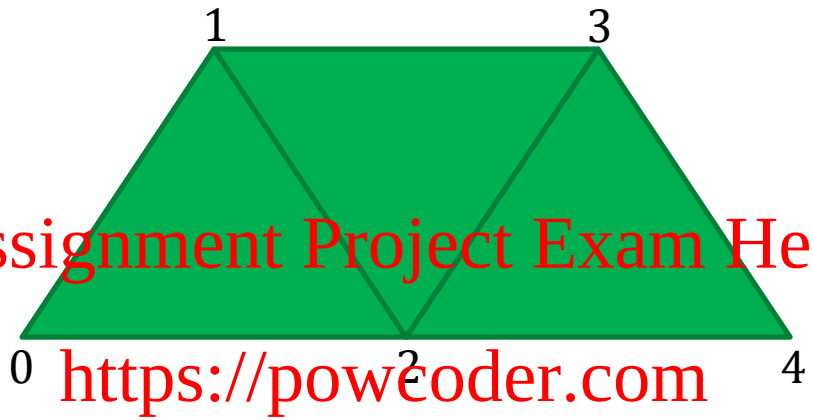





- Frequently shapes will share vertices with other shapes
- The naïve way would represent each vertex separately, even if shared
  - The above could be made with 9 vertices
- The smarter way would be to refer to the same vertex more than once
  - Only need 6 vertices then
  - We need a way to say which vertices to use for each triangle
    - The ***Index Buffer Object*** (***IBO***)

# Indices






| Vertex | Coord |
|---|---|
| 0 | $(-0.9, -0.9)$ |
| 1 | $(-0.5, 0.9)$ |
| 2 | $(0, -0.9)$ |
| 3 | $(0.5, 0.9)$ |
| 4 | $(0.9, -0.9)$ |



- Naïve VBO:

```
[-0.9, -0.9, 0, -0.9, -0.5, 0.9,
 -0.5, 0.9, 0, -0.9, 0.5, 0.9,
 0, -0.9, 0.9, -0.9, 0.5, 0.9]
```



- VBO + IBO:

```
[-0.9, -0.9, -0.5, 0.9, 0, -0.9, 0.5, 0.9, 0.9, -0.9]
[0, 2, 1, 1, 2, 3, 2, 4, 3]
```

- With larger objects and/or more attributes this saves tons of memory
- Since vertex shader is run for every vertex, this saves a lot of processing

# VBO and IBO Types



- VBOs can be groups of 1-4 values of the same primitive type
  - Floats of 32-bits or (u)ints of 8-, 16-, or 32-bits
  - Most common is `Float32`
- IBOs can only by unsigned ints of 8- or 16-bits
  - Some browsers may support 32-bits as well
  - Almost always `Uint16` since `Uint8` would only support a total of 256 vertices
    - Instead the limit is 65536 vertices

# Allocating the VBO and IBO



- First, we need to create the arrays in JavaScript:

```
let coords = [-0.9, -0.9, -0.5, 0.9, 0, -0.9, …];
let indices = [0, 2, 1, 1, 2, 3, 2, 4, 3];
```

  - If you won't change these then can make typed arrays now instead of later



- Then we need to create a new buffer on the GPU to store the data:

```
posBuffer = gl.createBuffer();
```



- Then say we are using that buffer for the next few commands:

```
gl.bindBuffer(gl.ARRAY_BUFFER, posBuffer);
```



- Then we copy the data from JavaScript to the GPU buffer:

```
gl.bufferData(gl.ARRAY_BUFFER, Float32Array.from(coords),
              gl.STATIC_DRAW);
```

- Finally, un-bind the buffer (not needed, but recommended):

```
gl.bindBuffer(gl.ARRAY_BUFFER, null);
```

- For IBOs: replace **`gl.ARRAY_BUFFER`** with **`gl.ELEMENT_ARRAY_BUFFER`**

*TODO: Complete the buffer allocations in the trapezoid program*

# Completely Cleaning Up



- If you are never going to use a buffer again you can free the GPU memory it occupies:

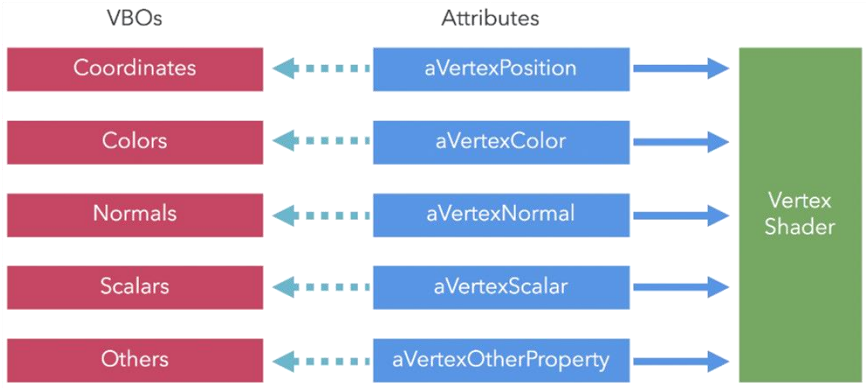
```
gl.deleteBuffer(bufferId)
```

# Associating Attributes to VBOs



- We now have GPU buffers full of data and a vertex shader with an "aPosition" attribute
  - We still have to associate these with each other so that when the vertex shader runs its inputs/attributes are loaded from the buffer
- To achieve this we need to:
  1. **Get the attribute index (can be cached)**
  2. Bind a VBO
  3. **Point attribute to the VBO**
  4. **Enable the attribute**
  5. Unbind





# Associating Attribute with VBO: Step #3



```
gl.vertexAttribPointer(index, size, type,
                       normalize, stride, offset)
```

- `index`: the attribute location (from step #1)
- `size`: the number of elements used for each vertex, examples:
  - 2D coords use 2
  - RGB colors use 3
- `type`: the primitive type, e.g. `gl.FLOAT`
- `normalize`: for now keep `false`
- `stride`: usually `0` meaning vertices are sequential
- `offset`: usually `0` meaning the buffer starts with the first vertex
- Typical call will be:

```
gl.vertexAttribPointer(aName, size, gl.FLOAT,
                       false, 0, 0);
```

  - Just need to change `Name` and specify `size`

# Attribute Naming Convention



- In this class we will use `aName` as the name of an attribute:
  - `aPosition` would be the position attribute
  - `aColor` would be the color attribute
- Attributes are defined in the vertex shader as `in` variables:
  - `in vec2 aPosition;`
  - `in vec3 aColor;`
- In JavaScript we will also have a global variable `aName` which refers to the attribute location in the program:
  - At the very top of the script:

    ```
    let aPosition;
    ```

  - Once the program is linked:

    ```
    aPosition = gl.getAttribLocation(program, 'aPosition');
    ```

# Associating an Attribute with a VBO



1. Get the attribute index (usually done in `init`)

```
aName = gl.getAttribLocation(program, 'aName');
```

2. Bind a VBO:
   - Same as when allocating
3. Point attribute to the VBO:

```
gl.vertexAttribPointer(aName, size, gl.FLOAT,
                       false, 0, 0);
```

4. Enable the attribute:

```
gl.enableVertexAttribArray(aName);
```

5. Unbind:
   - Same as when allocating

*TODO: in trapezoid program complete the attribute associations*

# Drawing the Vertices



- We now how GPU buffers full of data and those buffers are associated with attributes in the vertex shader – we are ready to render!





- To draw something we need to call either `drawArrays` or `drawElements`



  - These two functions as very similar except:
    - `drawArrays` uses only a VBO to draw (vertices must be in proper order, duplicated vertices may exist)
    - `drawElements` uses an IBO to determine the order of vertices to draw from a VBO
    - Sadly, they change the order and units of arguments

# drawArrays()



```
gl.drawArrays(mode, first, count)
```

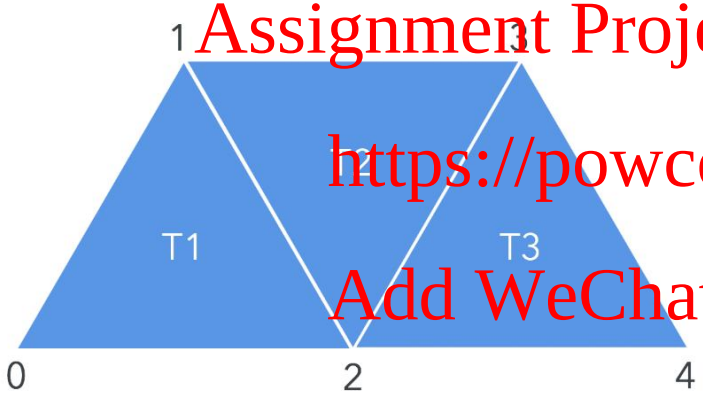


- `mode`: the type of primitive to draw and how the vertices of that primitive are organized:
  - `gl.POINTS`, `gl.LINES`, `gl.TRIANGLES`
    - Advanced: `gl.LINE_STRIP`, `gl.LINE_LOOP`, `gl.TRIANGLE_STRIP`, `gl.TRIANGLE_FAN`
- `first`: first vertex in the buffers to use for drawing
- `count`: number of vertices in the buffers to draw
  - Not the number of primitives to draw, number of vertices
  - `gl.drawArrays(gl.TRIANGLES, 0, 5)` draws 1 triangle

# drawArrays()



## Using drawArrays




| Index | Vertex Coordinates |
|---|---|
| 0 | (0, 0) |
| 1 | (10, 10) |
| 2 | (20, 0) |
| 3 | (30, 10) |
| 4 | (40, 0) |



Triangle 1 Triangle 2 Triangle 3

Vertex array = `[0, 0, 20, 0, 10, 10, 10, 10, 20, 0, 30, 10, 20, 0, 40, 0, 30, 10]`

`drawArrays` uses the vertex data in the order they are defined in the vertex array.

# drawElements()



```
gl.drawElements(mode, count, type, offset)
```

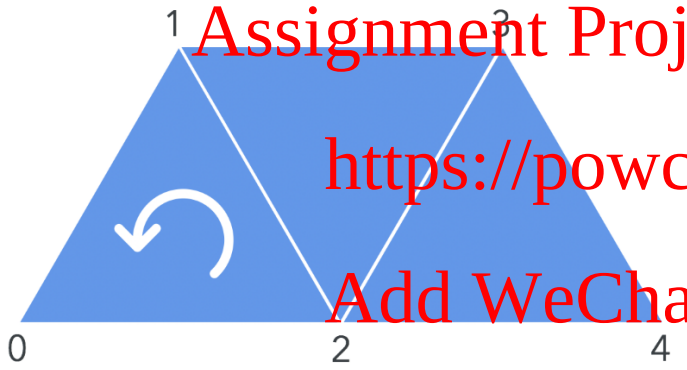
- `mode`: same as in `drawArrays`
- `count`: same as in `drawArrays`
  - but now second argument instead of third
- `type`: either `gl.UNSIGNED_BYTE` or `gl.UNSIGNED_SHORT`
  - Must match type of IBO, usually `gl.UNSIGNED_SHORT`
- `offset`: the number of *bytes* offset into the IBO for first vertex
  - Typically will do `offset*`*`ibo`*`.BYTES_PER_ELEMENT`

- Before calling you must make sure the appropriate IBO is bound

# drawElements()



## Vertices and Indices




| Index | Vertex Coordinates |
|---|---|
| 0 | (0, 0) |
| 1 | (10, 10) |
| 2 | (20, 0) |
| 3 | (30, 10) |
| 4 | (40, 0) |



coordinates

Vertex array = `[0, 0, 10, 10, 20, 0, 30, 10, 40, 0]` → Vertex Buffer

Index array = `[0, 2, 1, 1, 2, 3, 2, 4, 3]` → Index Buffer

triangles

Triangles in the index array are *usually*, but not necessarily, defined in counter-clockwise order.

# Trapezoid



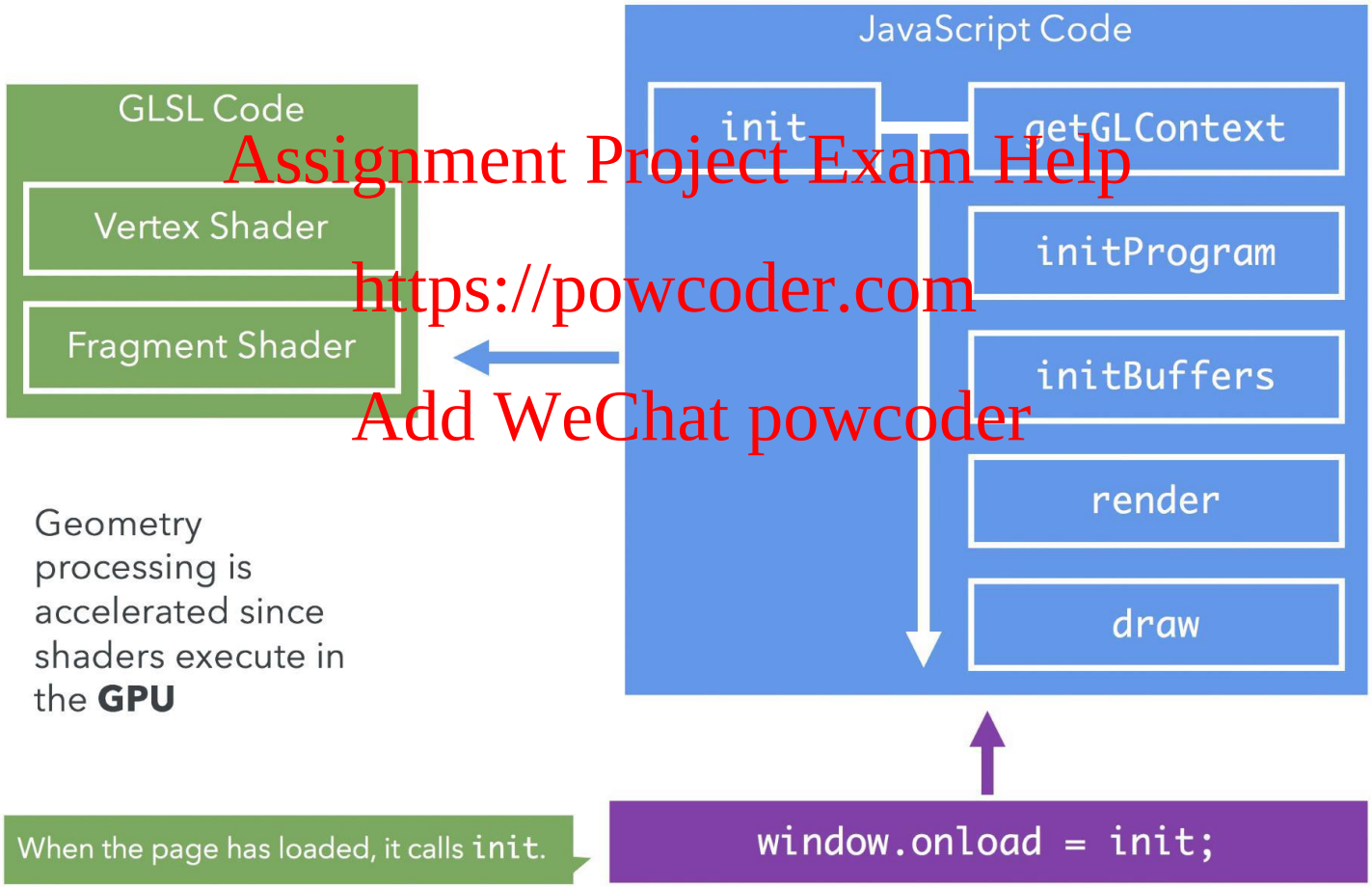
- Complete the trapezoid program
  - Need to do the last *TODO* about drawing
  - Make sure you use the correct function call(s) and correct arguments
- Once complete the output should be correct



- What happens if you give the wrong count?
- What happens if you call the wrong function?
- Can you rewrite it to use the other function?

# Overview





# Code Cleanup



- The `init()` function is getting quite long
  - It should be cleaned up, broken into a few separate functions
- Additionally, we should get rid of a few global variables…

# Vertex Array Objects (VAOs)



- Each object (using its own buffers) will require buffer binding and attribute associating/enabling repeated
  - Becomes longer for each attribute we have
    - Eventually 4+ attributes, that's 12 lines for each object!
- Besides annoying to maintain, it takes time
  - And it happens every render cycle
- Solution: ***Vertex Array Objects*** (***VAOs***)
  - While bound, VAOs record changes made by `bindBuffer()`, `vertexAttribPointer()`, and `enableVertexAttribArray()`
  - When re-bound later, restores the recorded global state
    - Mostly, on for use with `draw*()` functions

# Vertex Array Objects (VAOs)



**In `init()`:**

```
trapezoidVAO = gl.createVertexArray();
gl.bindVertexArray(trapezoidVAO);
// create, bind, and copy data to buffer
// enable and associate attributes
gl.bindVertexArray(null);
```



Need to save VAO "globally" but no longer need to save buffers globally

**In `render()`:**

```
gl.bindVertexArray(trapezoidVAO);
gl.draw*(…);
gl.bindVertexArray(null);
```

*Modify the trapezoid-vao code to use a VAO*

# Additional Vertex Attributes



- So far, we have only had one vertex attribute: the position (`aPosition`)
  - Another common attribute is color
- Assuming shaders are already setup to use it, what JavaScript code do we need to add/modify to use an additional attribute?

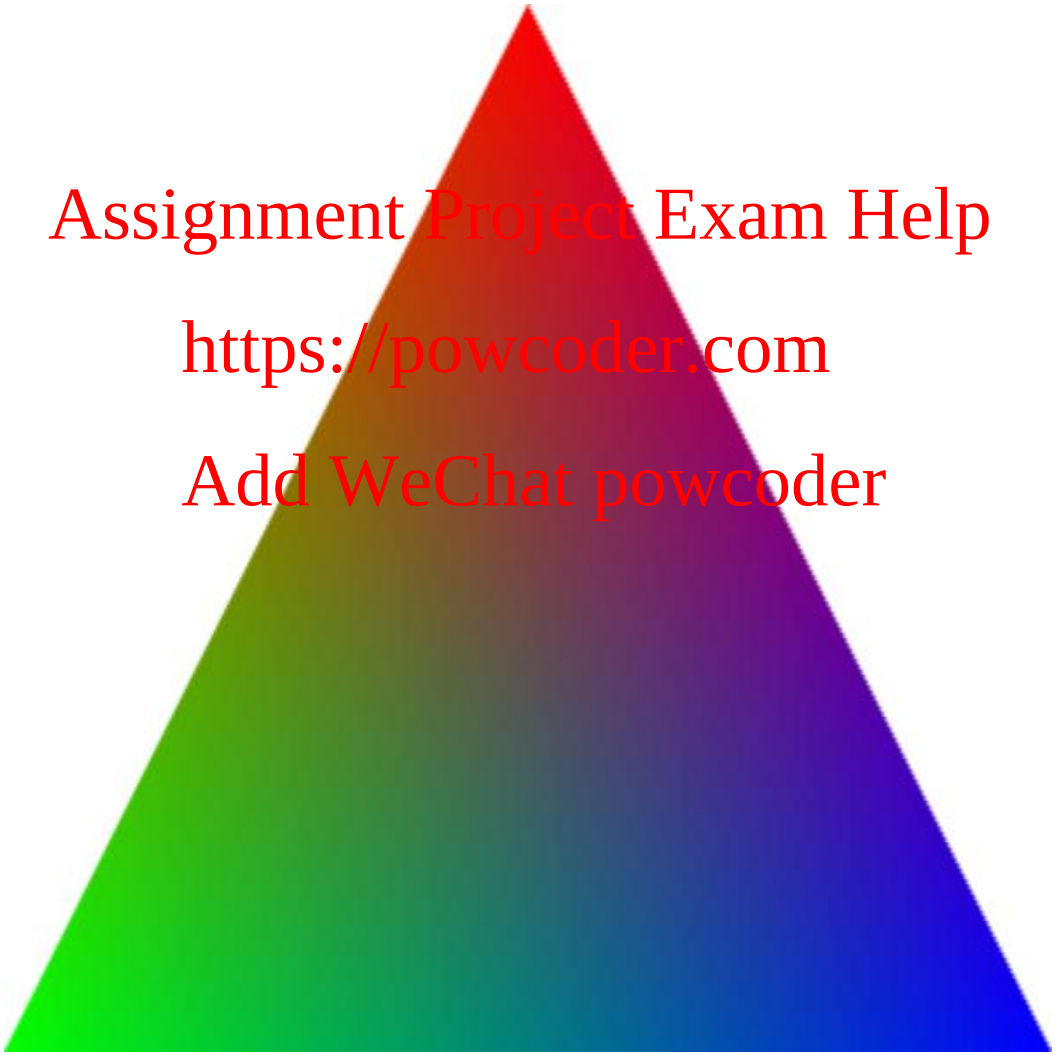
# Additional Vertex Attributes



- So far, we have only had one vertex attribute: the position (`aPosition`)
  - Another common attribute is color
- Assuming shaders are already setup to use it, what JavaScript code do we need to add/modify to use an additional attribute?
  - Get the attribute location
  - Create the JS array
  - Load the JS array into a VBO
  - Associate the VBO with the attribute while using the VAO
  - *No difference during rendering*

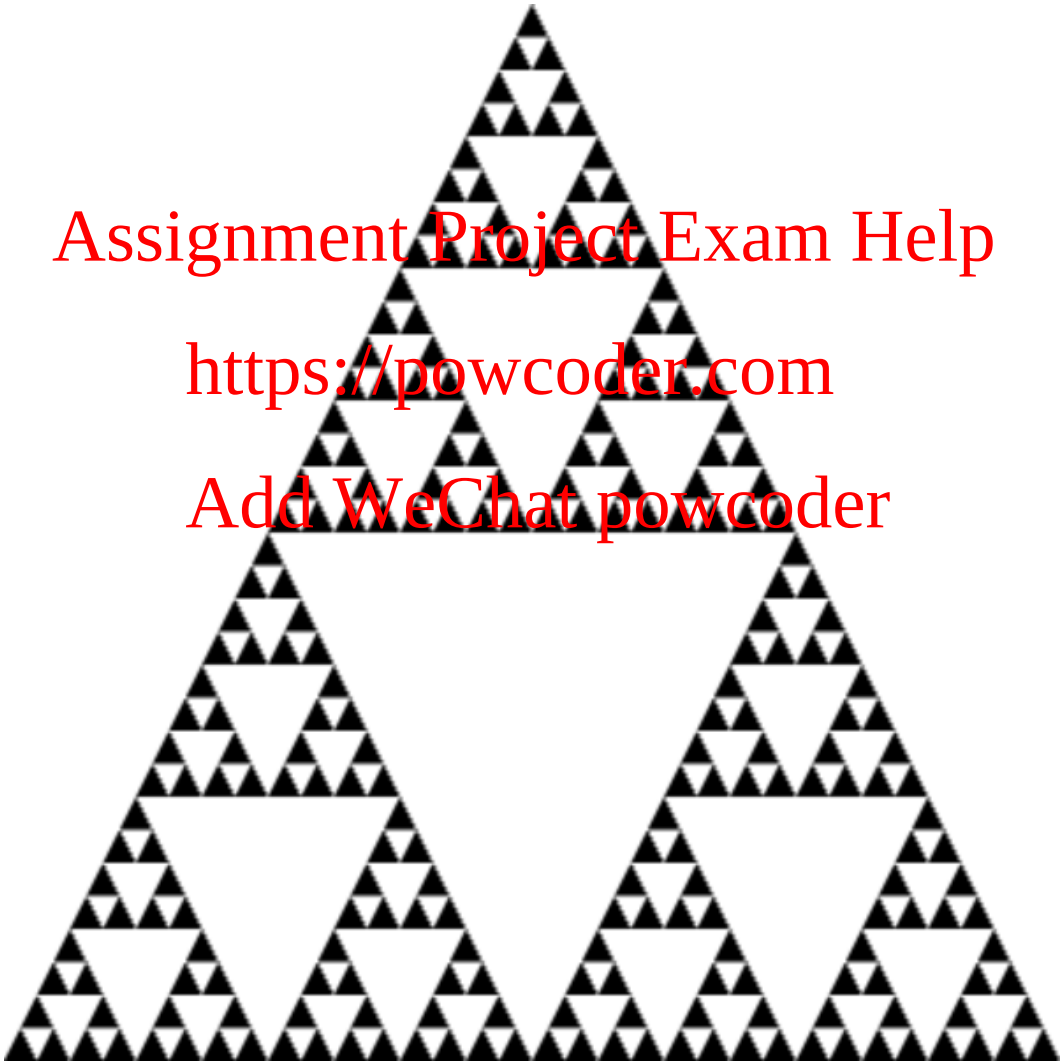
# Example: Maxwell's Triangle





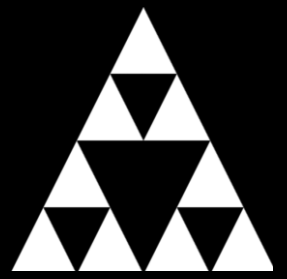
# Sierpinski's Triangle





# Sierpinski's Triangle





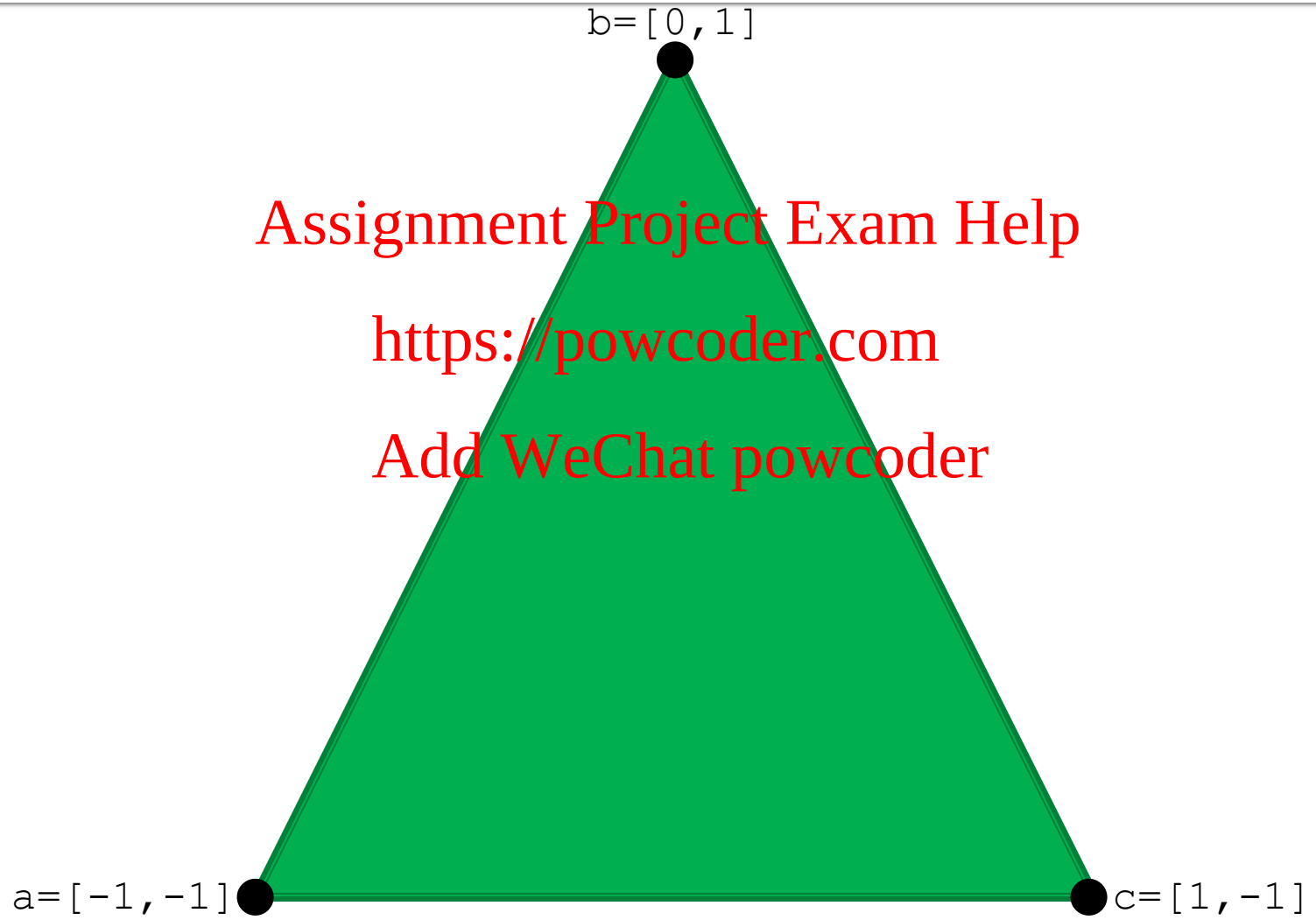
- Sierpinski's Triangle is a fractal that can be created recursively:
  - Start with a triangle
  - Create three triangles using the midpoints of each side of the original triangle
  - Repeat the last step for each of the new triangles
- The real fractal would repeat this forever but we want to repeat this only a certain number of times
  - In the last repetition we actually save the triangle to the list of vertices



*Complete the TODOs in sierpinski*

# Sierpinski's Triangle



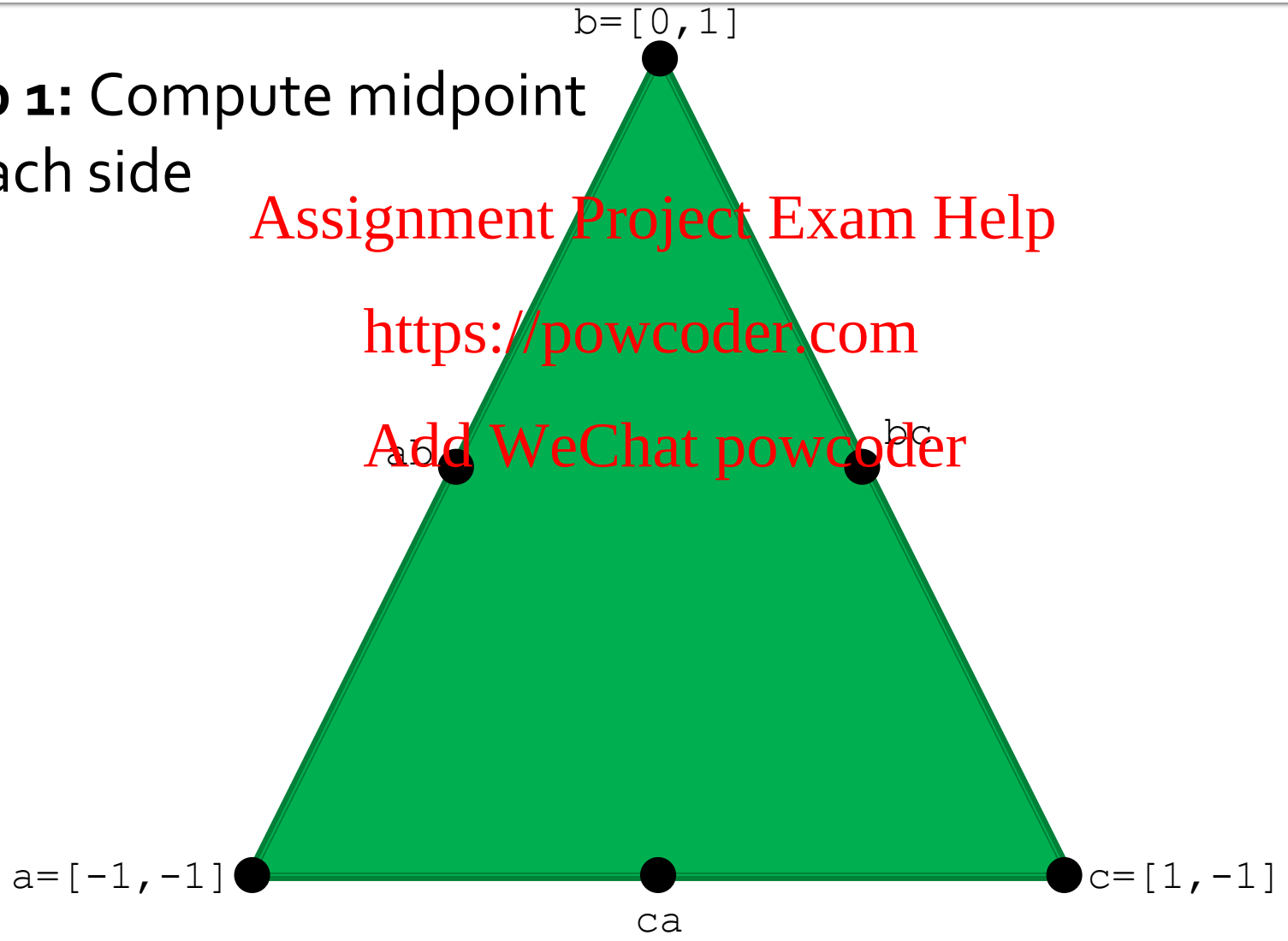


# Sierpinski's Triangle



**Step 1:** Compute midpoint of each side



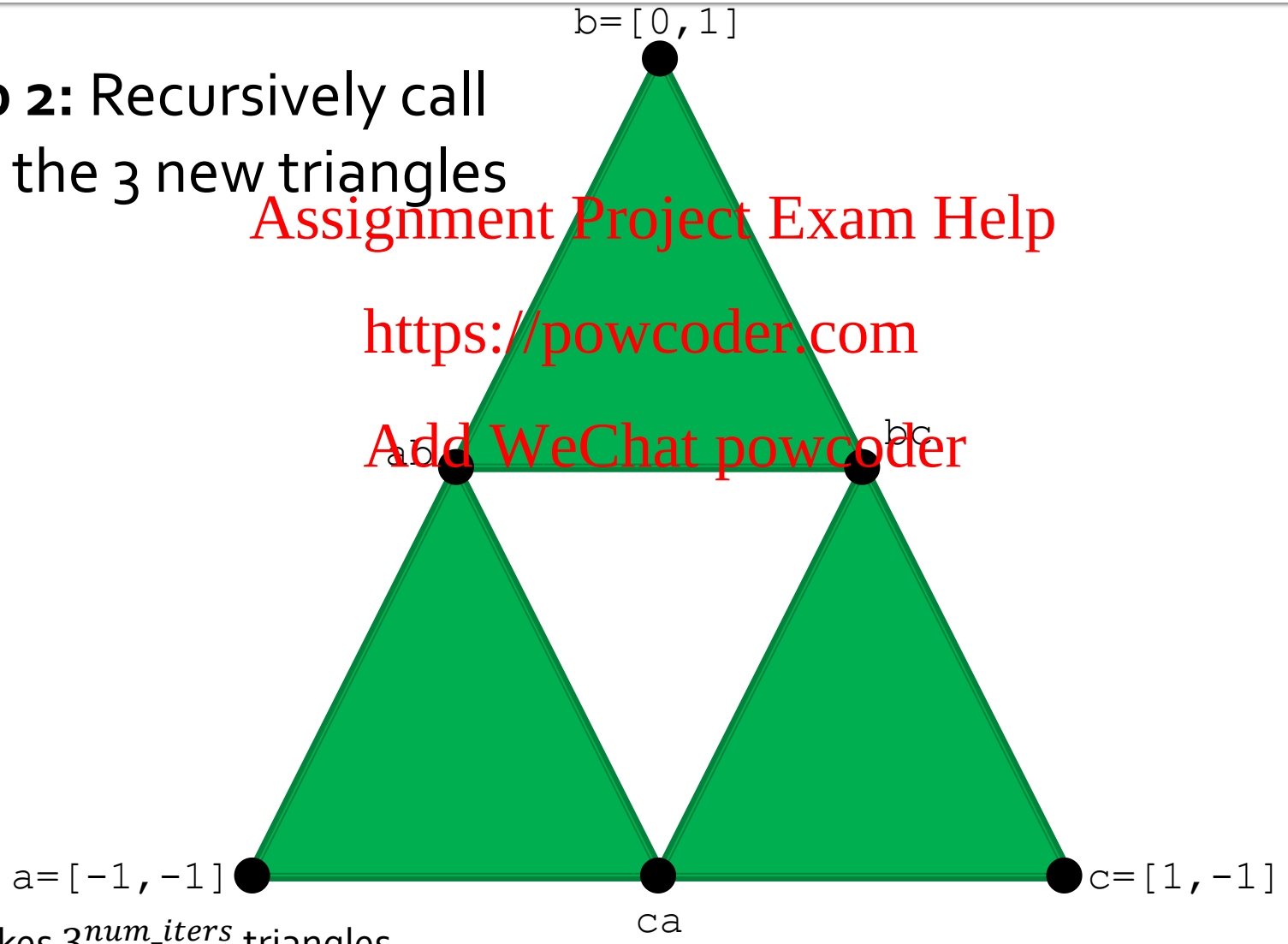


# Sierpinski's Triangle



**Step 2:** Recursively call with the 3 new triangles

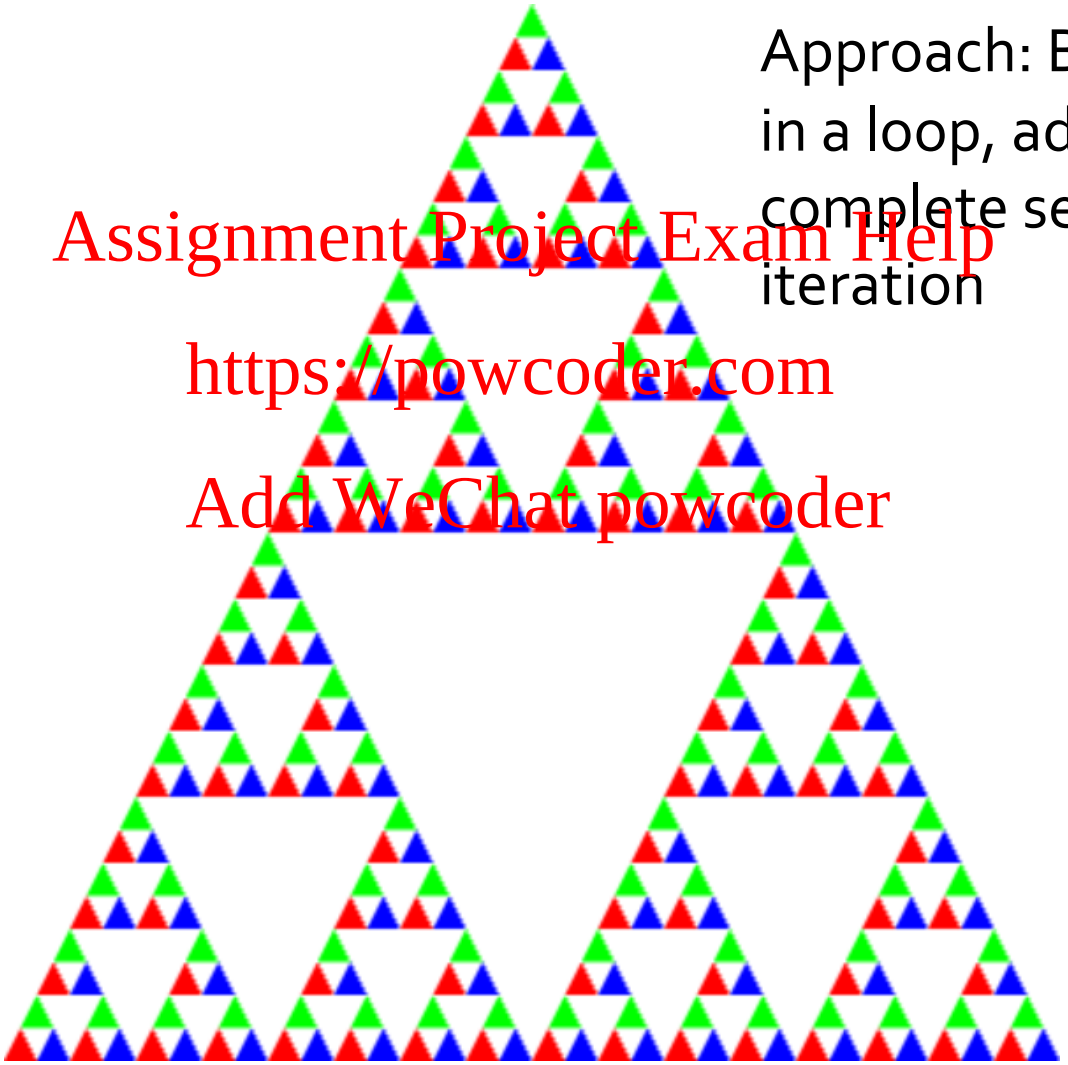





***Hint:*** Makes $3^{num\_iters}$ triangles

# Sierpinski's Triangle - Colored



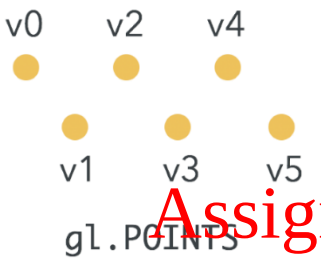




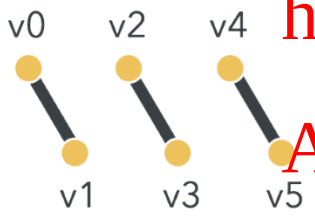
Approach: Build `colors` in a loop, adding each complete set during each iteration

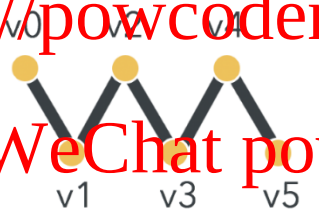
# WebGL Primitives

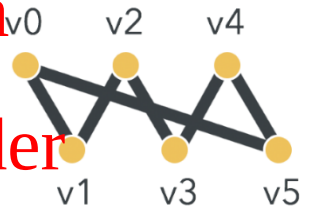


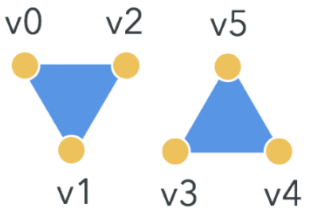


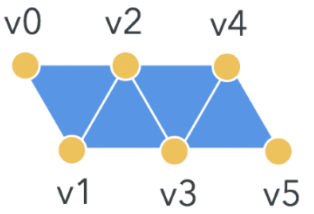


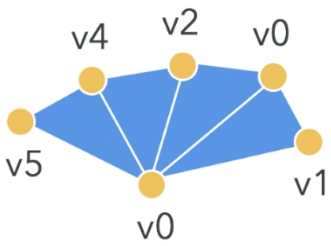



# Polygons?



- WebGL only deals with triangles which makes it efficient since they are always:

  - Simple: edges cannot cross

  - Convex: all points on line segments between any two vertices are also in the polygon

  - Flat: all vertices lie in the same plane
- No other polygon always has these properties
- Older versions of OpenGL assumed these rules for other polygons and when they weren't followed bad things happened

# Other Polygons



- Rectangle: just 2 triangles
- Circle: a whole bunch of triangles
- Other polygons need to *triangulated*
  - Converted into a bunch of triangles
  - Not just any set of triangles will do
    - Long and thin triangles render poorly
    - Equilateral triangles render well
      - Overall we want to maximize the minimum angle
      - Delaunay Triangulation

# Exercise: Draw a Rectangle



- There are several ways to draw a square
- In "2-Rendering/rectangle-basic" create a rectangle as 2 triangles using 6 vertices
  - Use drawArrays() and thus no IBO
  - Try to make it not touch the sides so you can see that it is definitely a square

# Vertex Repetition



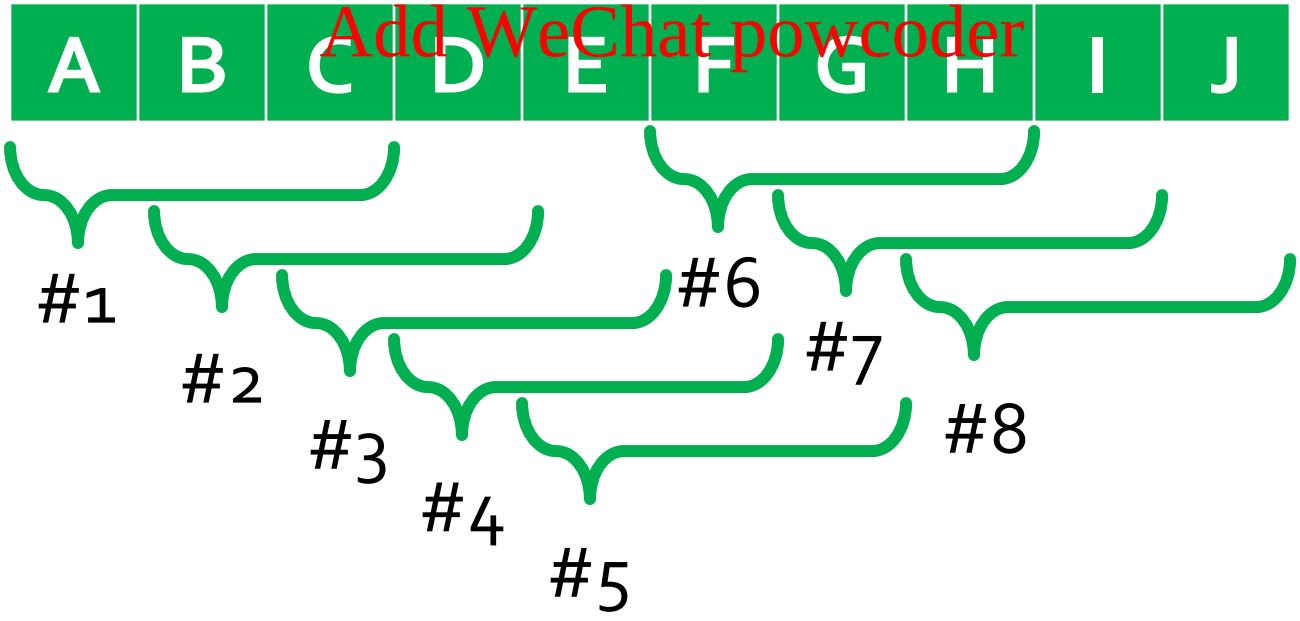
- A rectangle has 4 corners but we needed 6 vertices to show it!
- This has 50% more data sent to the GPU and 50% more executions of the vertex shader
  - In this case that is tiny, but in larger projects that could become serious
- We could reduce this using IBOs but for the moment let's explore other options
  - We can use a different drawing mode

# Triangle Strip



- We can avoid vertex repetition this time by using the `TRIANGLE_STRIP` drawing method
- This draws a series of triangles that share an edge with the previous triangle drawn



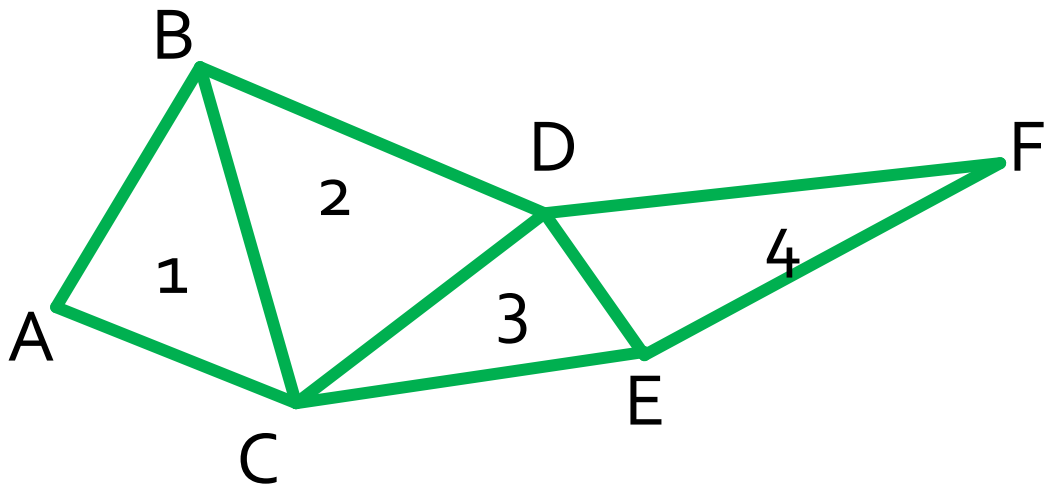


# Triangle Strip



- For $n$ triangles it takes $n + 2$ vertices
- The diagram below creates the triangles ABC, CBD, CDE, and EDF
  - The reversed letters are unimportant for the moment





# Example: Draw a Rectangle



- In "2-Rendering/rectangle-strip" create a rectangle as a triangles using a triangle strip with 4 vertices
- In `render` you will use:



```
gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
```

# Triangle Strip Limitations



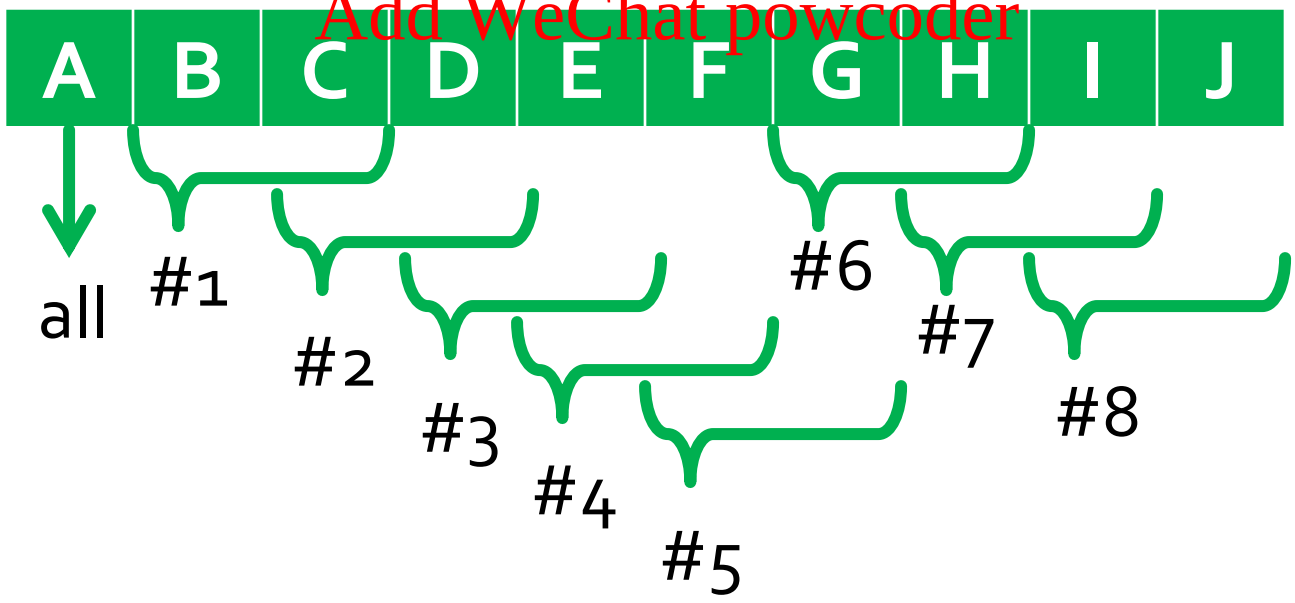
- Each triangle must share 2 vertices with the last triangle
  - However we can add two duplicate points to create 2 degenerate triangles to cause a jump
- Since attributes (like color) are assigned to vertices cannot easily create a series of solidly colored triangles or other shapes

# Triangle Fan



- We can also avoid vertex repetition this time by using the `TRIANGLE_FAN` drawing method
- All triangles must share the initial vertex along with one other vertex of the previous triangle
  - Thus sharing an edge, but a different edge than was shared with `TRIANGLE_STRIP`



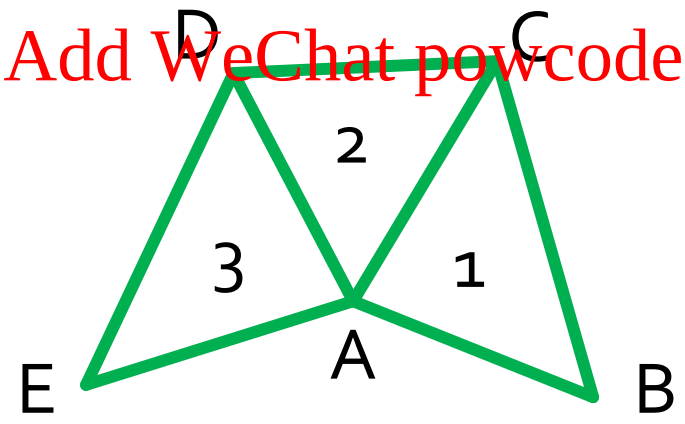


# Triangle Fan



- For $n$ triangles it takes $n + 2$ vertices
- In the diagram below creates the triangles ABC, ACD, and ADE





# Example: Draw a Rectangle



- In "2-Rendering/rectangle-fan" create a rectangle as a triangles using a triangle fan with 4 vertices
- In `render` you will use:



```
gl.drawArrays(gl.TRIANGLE_FAN, 0, 4);
```

# Triangle Fan Limitations



- Each triangle must share initial vertex and a vertex of the previous triangle
  - However any convex polygon can be drawn as a single fan
- Since attributes (like color) are assigned to vertices cannot easily create a series of solidly colored triangles or other shapes

# Example: Draw a Circle



- Circles are usually defined as a center point and a radius but everything in WebGL is defined using vertices/triangles
  - Must approximate using a high number of sides/vertices/triangles
- In "2-Rendering/circle-basic" complete the circle function that takes a center vertex, radius, number of sides, and an array of vertices that will be appended to

# Example: Draw a Circle



- Each triangle will have one vertex at the center of the circle and 2 along the edge of the circle
  - Each neighboring triangle will share the center vertex and a vertex along the edge
  - Make sure the circle is completed by having the last triangle share a vertex with the first triangle
- `function circle(c,r,n,verts)`
- How many sides before it looks like a circle?

# Example: Draw a Circle



```
let theta = 2*Math.PI/n;
let a = vec2(c[0]+r, c[1]);
for (let i = 1; i <= n; ++i) {
    let b = vec2(
        c[0]+Math.cos(i*theta)*r,
        c[1]+Math.sin(i*theta)*r);
    verts.push(c, a, b);
    a = b;
}
```

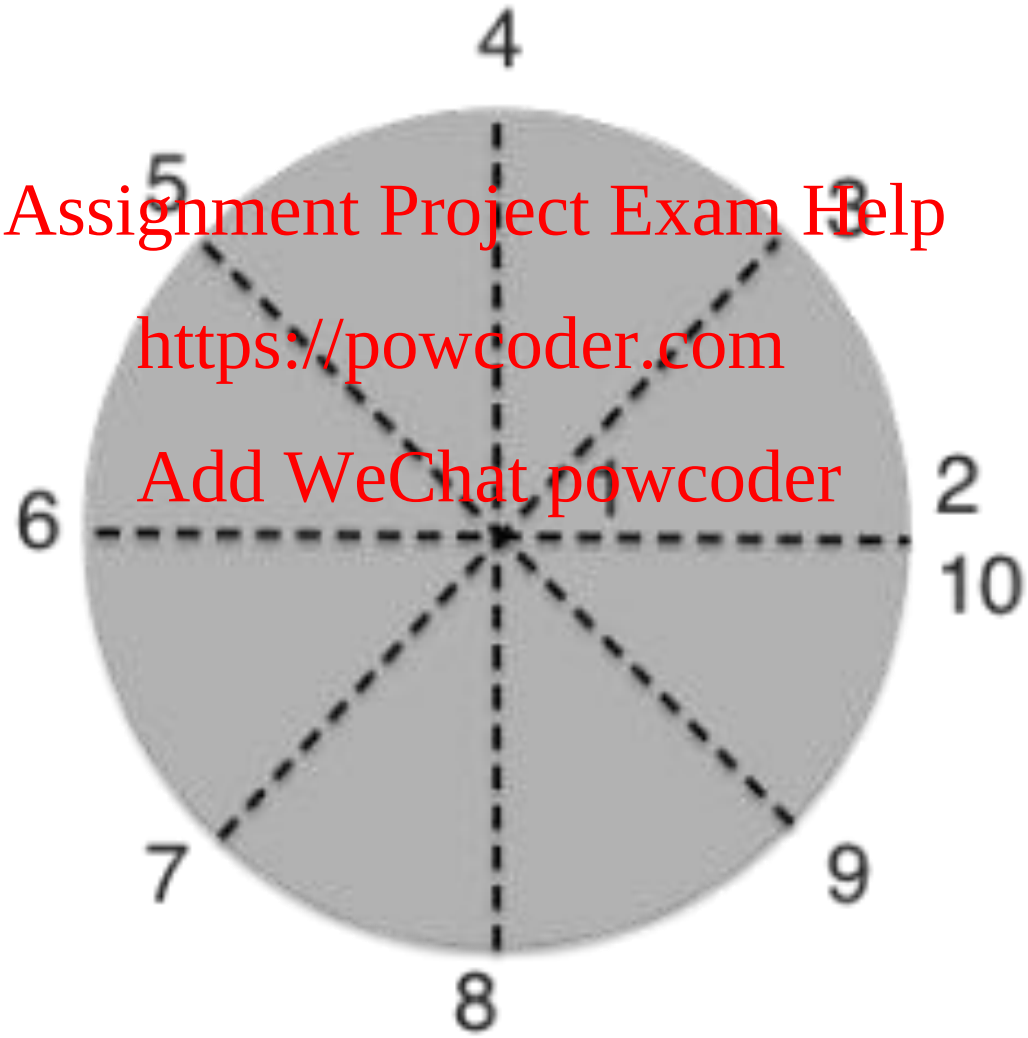
# Exercise: Circle w/o Redundancy



- To draw a circle approximated with $n$ sides we needed $3n$ vertices!
  - That is 200% more than we should be using!
- To reduce the number of vertices would you choose to draw a circle with a triangle strip or triangle fan? Why?
- Implement it in "2-Rendering/circle-better"
  - Is it easier or harder than the implementation with redundant vertices?
  - Are there any redundant vertices left?

# Circle using Triangle Fan





# User Interaction with JavaScript



- User interaction through webpage causes JS events to "fire"
- Examples of events:
  - `load` – page has finished loading (we use this now)
  - `click` – mouse is clicked
  - `mousedown/up` – mouse is pressed/released
  - `mousemove` – mouse is moved
  - `wheel` – mouse wheel is rotated
  - `keypress` – keyboard key is pressed (down and up)
  - `resize` – when the webpage changes size
  - `change` – when an input (e.g. drop-down or textbox) changes value
  - And many more…
- When an event fires it calls a function you specify:
  - When called, `this` refers to the element with the event registered
  - First parameter is an `Event` object with details about the event

# JavaScript: Listening for an event



```
elem.addEventListener('event', func);
```

- Where:
  - `elem` – HTML element that may have the event occur (e.g. the <canvas> element or the overall <html> (JavaScript calls this the `window`)
  - `'event'` – name of the event (e.g. `'load'` or `'click'`)
  - `func` – name of function to call (or definition of an anonymous function)
    - Our loading example uses an anonymous function
    - Our next example will refer to a function by name
- Can also stop listening:
  `elem.removeEventListener('event', func);`

# JavaScript: Event Callbacks



- In the *circle-resize* example listen for click events on the canvas element (line 29), calling the `onClick` function (defined at the end)
- In the `onClick` function, log out to the console the x and y positions of the click (relative to the canvas), the width and height of the canvas, and the event object `e`
  - Ignore the other TODOs for now
  - Play around with clicking in various spots in and out of the canvas
  - What coordinate system are the `x` and `y` values in within the click handler?

# Convert Clip Coordinates



- In the click handler, complete the next TODO: convert the `x` and `y` values to clip coordinates
  - Log them out and confirm that you are getting the right values



- Once you are done with that go onto the next TODO: computing the new radius
  - What values should you get at the middle? Near the left/right/top/bottom edges? Near the corners?
  - Log it out and confirm





- Working ahead? Complete TODOs in `initBuffers`, `render`, and `circle`
  - Don't do the last 3 in `onClick` though

# Buffer Information



- Sometimes we need to get information about the current bound buffers
  - Can be used to avoid passing buffers as arguments or to automatically figure out how many elements are to be drawn
- `gl.getParameter(param)`
  - Get currently bound buffer
  - Can be used to avoid passing buffer as argument
  - **VBO:** `gl.ARRAY_BUFFER_BINDING`
  - **IBO:** `gl.ELEMENT_ARRAY_BUFFER_BINDING`
  - Also used to get other WebGL parameters, such as `gl.VERTEX_ARRAY_BINDING` or `gl.VERSION`

# Buffer Information



- `gl.getBufferParameter(target, param)`
  - Gets a parameter of the currently bound buffer type
  - `target` is which currently bound buffer:
    - **VAO**: `gl.ARRAY_BUFFER`
    - **IAO**: `gl.ELEMENT_ARRAY_BUFFER`
  - `param` says which parameter to get, most useful:
    - `gl.BUFFER_SIZE` gets the number of ***bytes*** in the buffer
- How to get the number of vertices?
  - *Hint*: can get the number of bytes a certain type takes:
    - `Float32Array.BYTES_PER_ELEMENT` (VAO)
    - `Uin16Array.BYTES_PER_ELEMENT` (IAO)
- *TODO*: update `render()` to use `getBufferParameter()` instead of `NUM_SIDES` to determine number of vertices to draw
  - Will also need to make a minor update in `initBuffers()`

# Updating Data on the GPU



- We won't always have all data we need at beginning
  - For example, generated from user input
- During `init`, we can call `gl.bufferData()` with a size instead of an array to reserve space
  - Also specify usage as `gl.DYNAMIC_DRAW` (i.e. changing) instead of static (i.e. staying the same)
- Later you need to use `gl.bufferSubData()` (after binding) to update data in the buffer
  - You can update all data in the buffer, a single vertex, or add more vertices (if there is reserved space)

# gl.bufferSubData()



`gl.bufferSubData(target, offset, data)`

- Copies `data` into `target` (i.e. `gl.ARRAY_BUFFER`), starting at the provided ***byte*** `offset`



- Example: (I use `F32A` instead of `Float32Array`)



  - Copy 1,2,3 to the beginning:

```
gl.bufferSubData(gl.ARRAY_BUFFER, 0,
                 F32A.from([1, 2, 3]));
```



  - After that put in 4,5,6:

```
offset = 3*F32A.BYTES_PER_ELEMENTS;
gl.bufferSubData(gl.ARRAY_BUFFER, offset,
                 F32A.from([5, 6, 7]));
```

*TODO: Update the final 3 TODOs in `onClick`*