

# WebGL

## CS425: Computer Graphics I

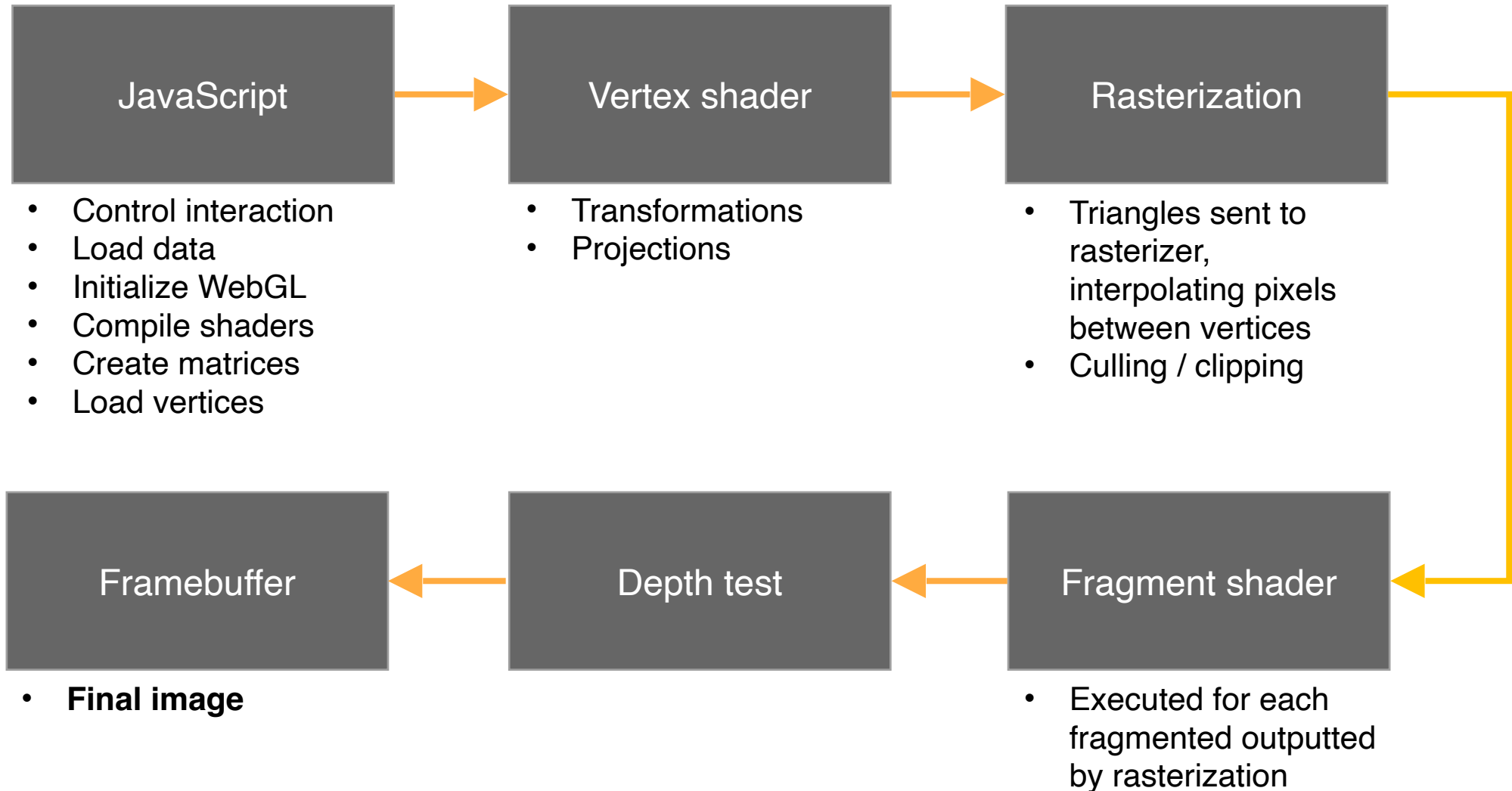
Khairi Reda

# Administrativa

---

- Sign up for Piazza if you have not! (Link/code posted on Blackboard)
- Still debugging issue with GitHub Classroom
- For now, please use direct link to repo
  - Please work locally for now — don't fork the above repo
  -

# WebGL: a bird's eye view



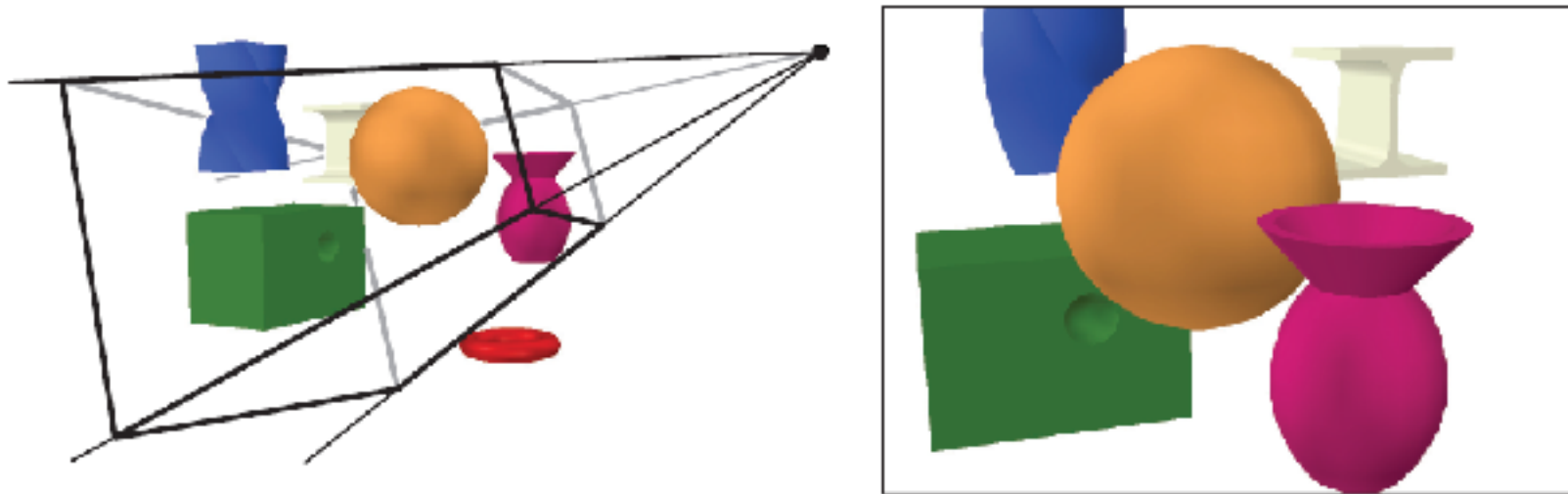
# WebGL

---

- API for rendering graphics within a web browser without plug-ins.
- Hardware accelerated.
- Shader based (no fixed-function API).
  - Fixed function pipeline: set of calls for matrix transformation, lighting.
  - Programmable pipeline: shaders for vertex and fragment processing.
- WebGL 2.0 based on OpenGL ES 3.0.

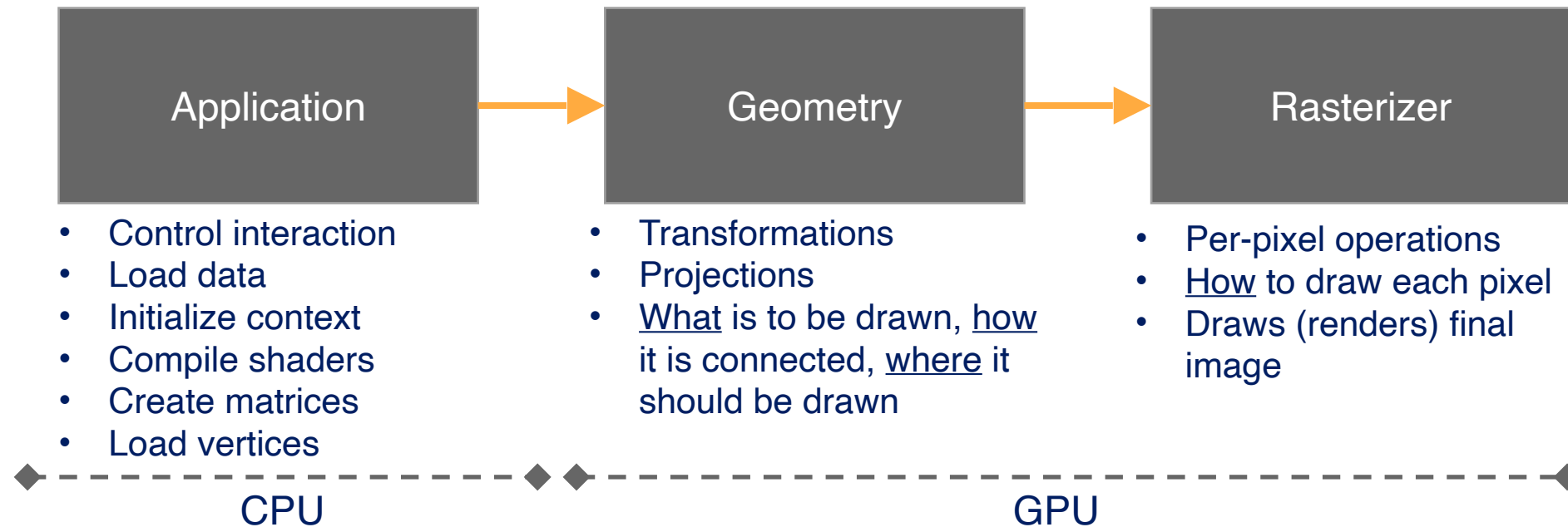
# Rendering pipeline

- Graphics system steps to render a scene to a 2D screen.



From: Real-Time Rendering 4<sup>th</sup> Ed

# Rendering pipeline



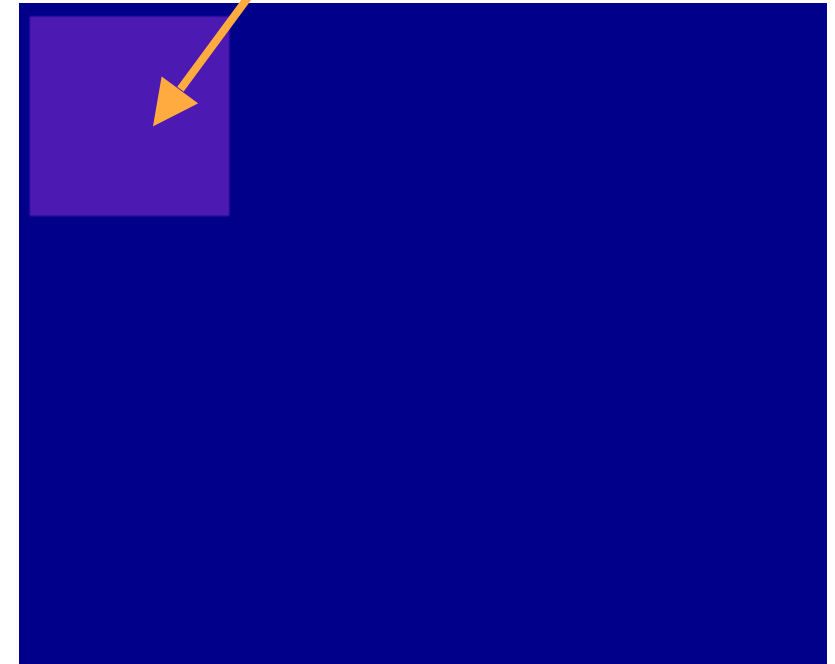
Slowest pipeline stage will determine *rendering speed* (in frames per second).

Simple example: bottleneck stage takes 20 ms to execute. Rendering speed:  $1/0.020 = 50$  fps.

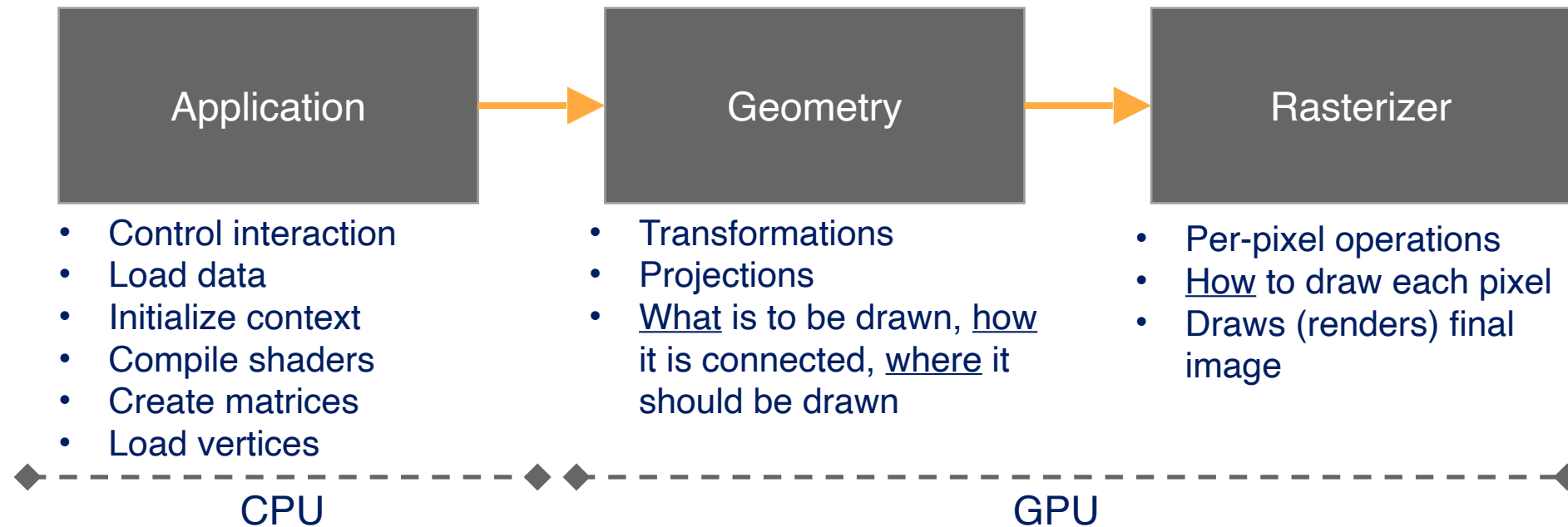
# WebGL: a bird's-eye view

```
<html lang="en">
<style>
  body { background-color: darkblue;}
  #glcanvas { width: 100px; height: 100px;}
</style>
<script type="text/javascript">
  function main() {
    var canvas = document.querySelector('#glcanvas
  ');
    var gl = canvas.getContext('webgl2');
    gl.clearColor(0.3, 0.1, 0.7, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);
  };
  window.onload = main;
</script>
<body>
  <canvas id="glcanvas"></canvas>
</body>
</html>
```

WebGL canvas



# Rendering pipeline

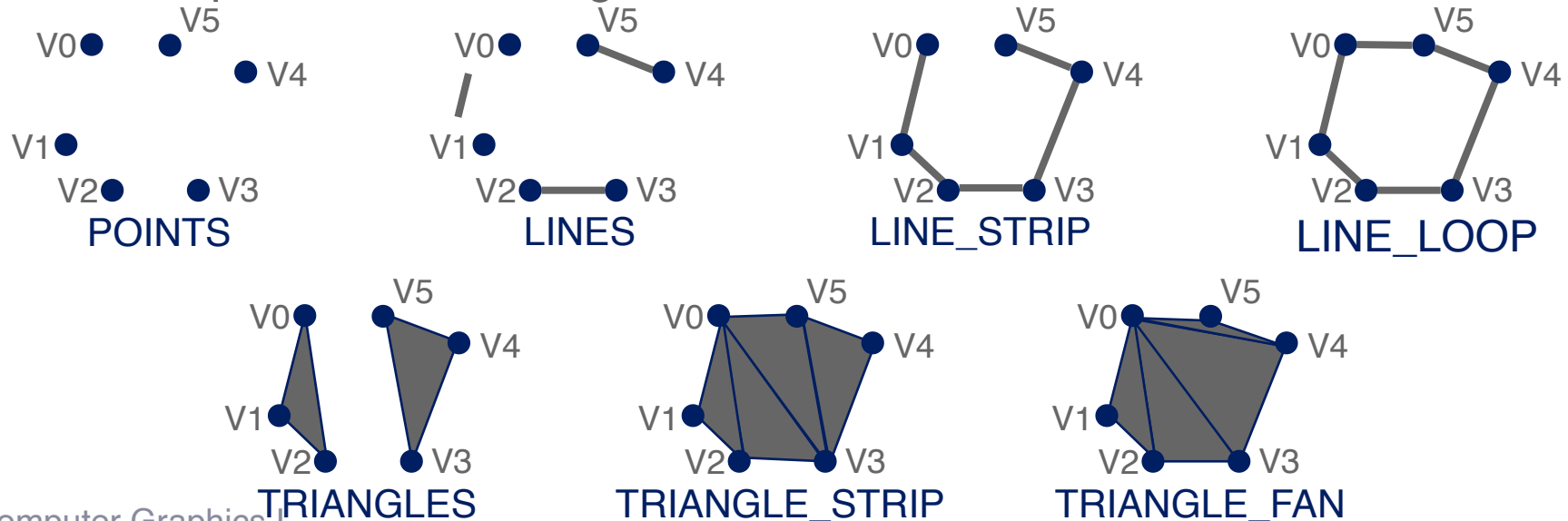


Slowest pipeline stage will determine *rendering speed* (in frames per second).

Simple example: bottleneck stage takes 20 ms to execute. Rendering speed:  $1/0.020 = 50$  fps.

# Application stage

- Executed on the CPU.
- Sends the geometry to be rendered to the geometry stage.
  - Primitives: points, lines, triangles.



# Application stage and WebGL

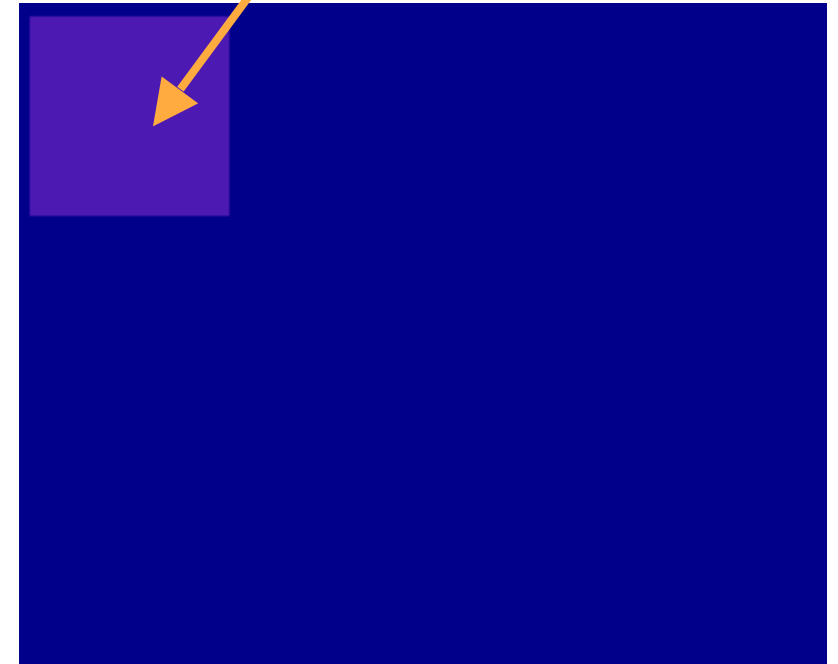
- Important: WebGL is a **state machine**, functions set or retrieve some state in WebGL. Information necessary for rendering is stored in the WebGL context.
- Get context from the canvas element (`canvas.getContext("webgl2")`)
- Load and initialize shaders (`WebGLShader`) and programs (`WebGLProgram`).
- Loads geometries (attributes and buffers).
- Handles rendering loop.

# WebGL canvas.getContext()

```
<html lang="en">
<style>
  body { background-color: darkblue;}
  #glcanvas { width: 100px; height: 100px;}
</style>
<script src="main.js" type="module"></script>
<body>
  <canvas id="glcanvas"></canvas>
</body>
</html>
```

```
function main() {
  var canvas = document.querySelector("#glcanvas");
  gl = canvas.getContext("webgl2");
  gl.clearColor(0.3, 0.1, 0.7, 1.0);
  gl.clear(gl.COLOR_BUFFER_BIT);
}
window.onload = main;
```

WebGL canvas



# Loading shaders

- Add GLSL shaders to external files and import them:

```
import vertexShaderSrc from './vertex.glsl.js';  
import fragmentShaderSrc from './fragment.glsl.js'
```

- Simple vertex shader and fragment shaders:

```
export default `#version 300 es  
  
in vec4 position;  
  
void main() {  
    gl_Position = position;  
}  
`;
```

```
export default `#version 300 es  
precision highp float;  
out vec4 outColor;  
  
void main() {  
    outColor = vec4(1, 0, 0, 1);  
}  
`;
```

# Loading shaders

- Vertex shader creation:

```
var vertexShader = gl.createShader(gl.VERTEX_SHADER);

gl.shaderSource(vertexShader, vertexShaderSrc);
gl.compileShader(vertexShader);

if (!
gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS) ) {
    var info = gl.getShaderInfoLog(vertexShader);
    console.log('Could not compile WebGL program:' + info)
;
}
```

# Loading shaders

- Create a program - combination of two compile WebGL shaders (vertex and fragment):

```
var program = gl.createProgram();

gl.attachShader(program, vertexShader);
gl.attachShader(program, fragmentShader);

gl.linkProgram(program);

if (!gl.getProgramParameter(program, gl.LINK_STATUS) ) {
    var info = gl.getProgramInfoLog(program);

    console.log('Could not compile WebGL program:' + info);
}
```

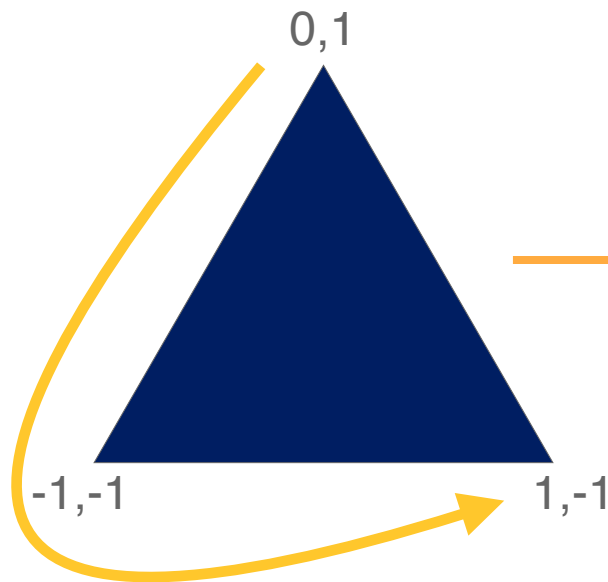
# Sending data to the GPU

- OpenGL objects are the structures responsible for *transmitting* data to and from the GPU.
- Several types of objects:
  - **Buffers**: arrays with data to send to the GPU.
    - Vertex positions, normals, indices, texture coordinates, colors.
  - **Vertex array object**: describes how vertex attributes are stored in the buffers.
  - Many others: uniforms, textures, varyings.

# Sending data to the GPU

- Initialization:
  1. Create vertex **buffer objects (VBO)** to store vertex data.
  2. Create a vertex **array object (VAO)** to specify how this data is formatted so it can be accessed properly by the vertex shader.
- Every rendering frame:
  1. Use shader programs.
  2. Bind buffers.
  3. Draw arrays.

# Buffer



```
var vertices2D = [  
    0.0, 1.0, 0.0,  
    -1.0, -1.0, 0.0,  
    1.0, -1.0, 0.0,  
];
```

Creating a buffer to store the triangle:

1. Create buffer object
2. Bind resource to binding point
3. Send strongly typed data to binded buffer

```
var positionBuffer = gl.createBuffer();  
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);  
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices2D), gl.STATIC_  
DRAW);
```

# Vertex array object

- VAO specifies layout of how data will be accessed by vertex shader.
- VAO typically contains multiple VBOs
  - One for vertex positions
  - A second for vertex colors
  - A third might be texture coordinates (more on this later)

# Vertex array object

pos[0]

x, y, z

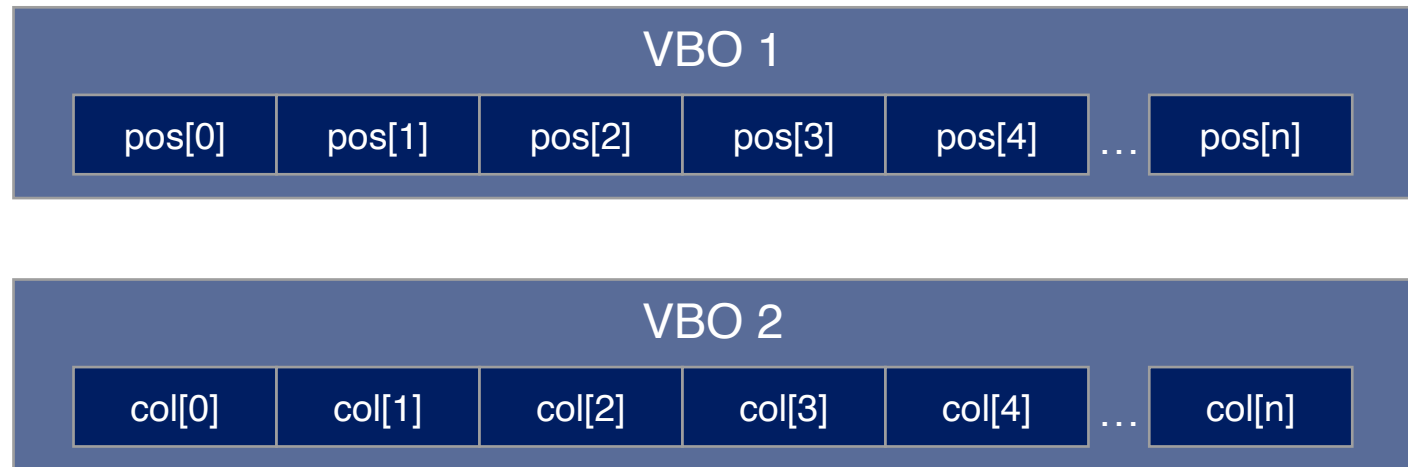
col[0]

r, g, b

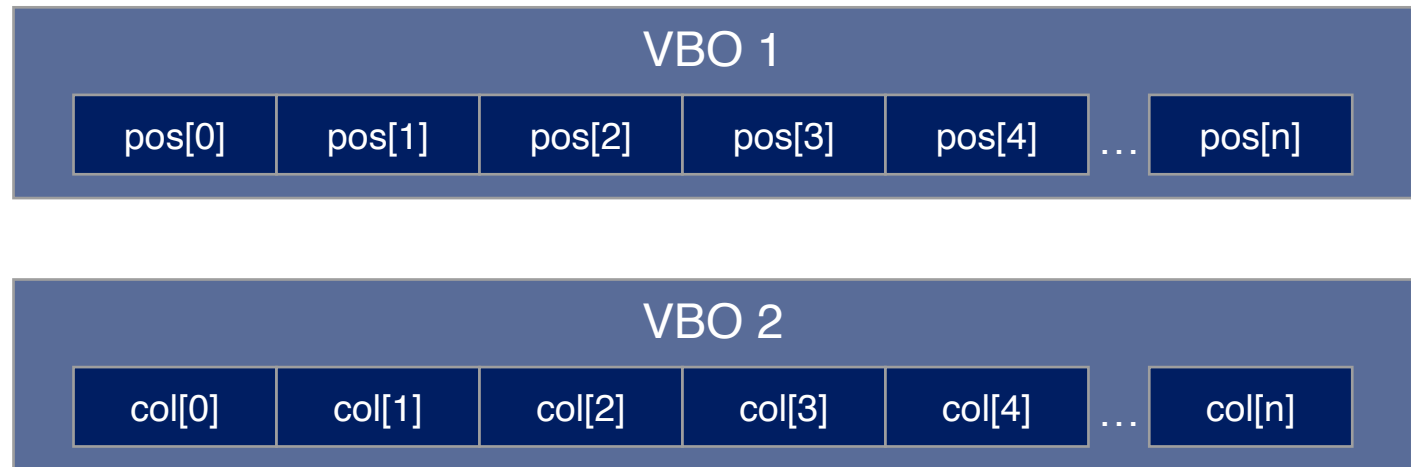
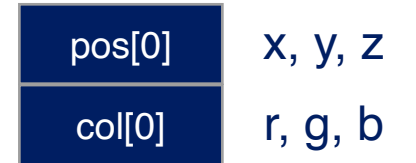
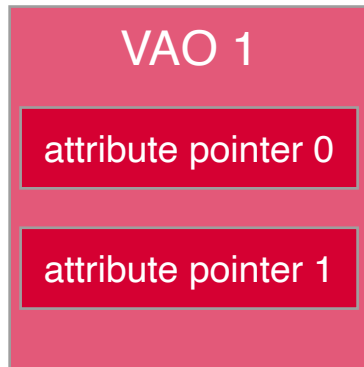
# Vertex array object

pos[0] x, y, z

col[0] r, g, b



# Vertex array object



# Vertex array object

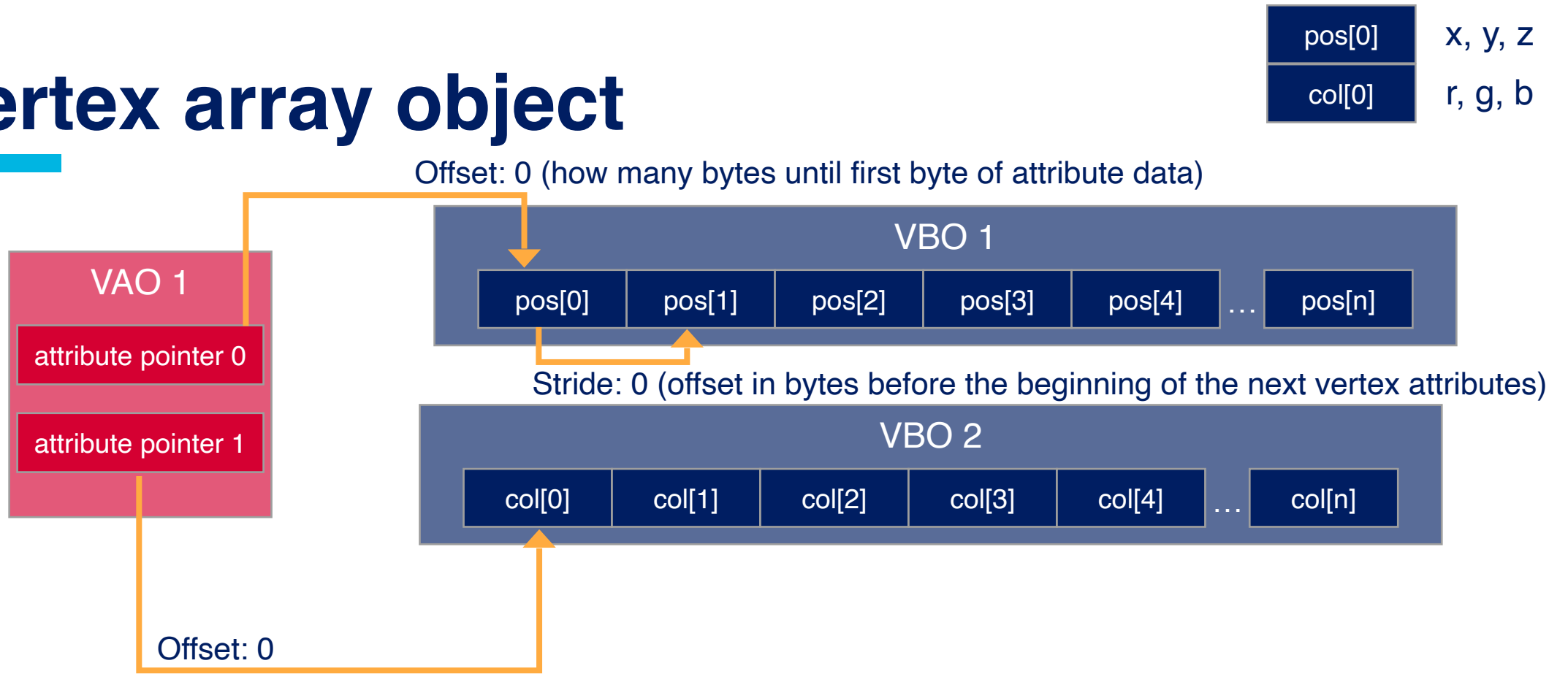
- VAO specifies layout of how data will be accessed by vertex shader.
- Location of the attribute in the program we created:

```
var posAttribLoc = gl.getAttribLocation(program, "position");
```

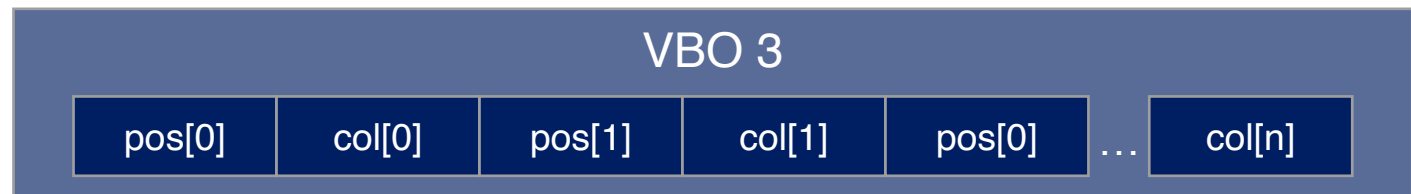
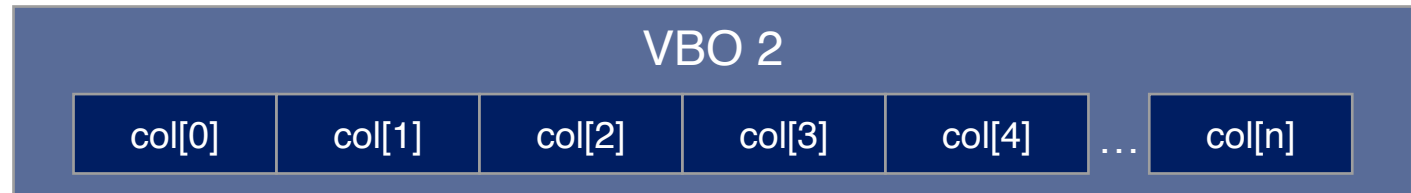
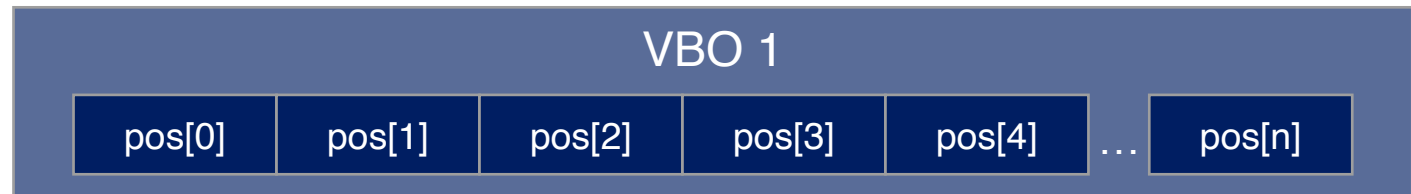
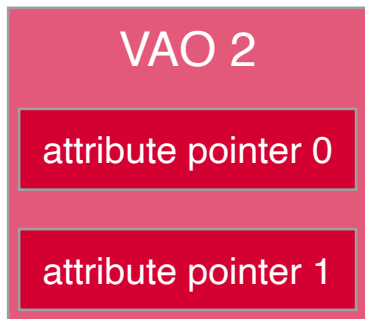
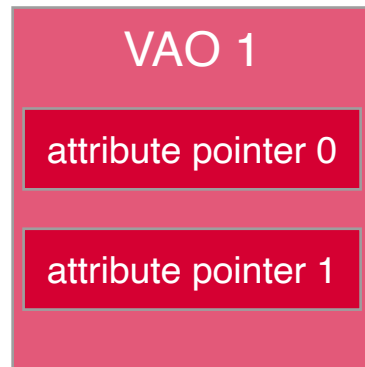
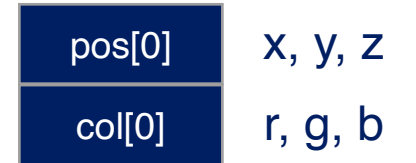
- Creating a vertex array object so shader can access buffer:

```
var vao = gl.createVertexArray();
gl.bindVertexArray(vao);
gl.enableVertexAttribArray(posAttribLoc);
var size = 3; // 3 components
var type = gl.FLOAT; // 32bit floats
var normalization = false;
var stride = 0; var offset = 0;
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
gl.vertexAttribPointer(posAttribLoc, size, type, normalized, stride, offset);
```

# Vertex array object

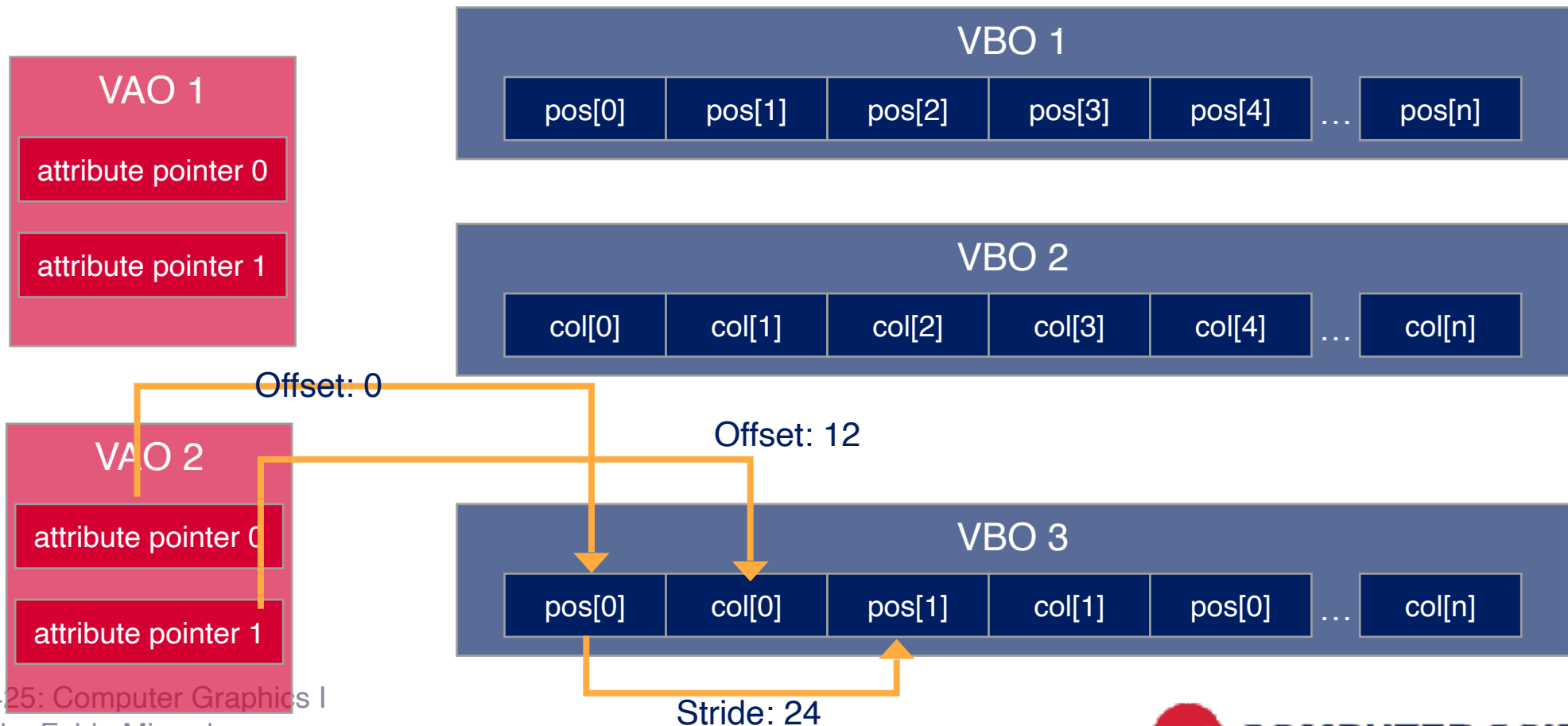


# Vertex array object



# Vertex array object

pos[0]	x, y, z
col[0]	r, g, b



# Take a breather



# Rendering frame

- This will be executed every time we want to render.
- Set viewport (maps clip space to screen space):

```
gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);
```

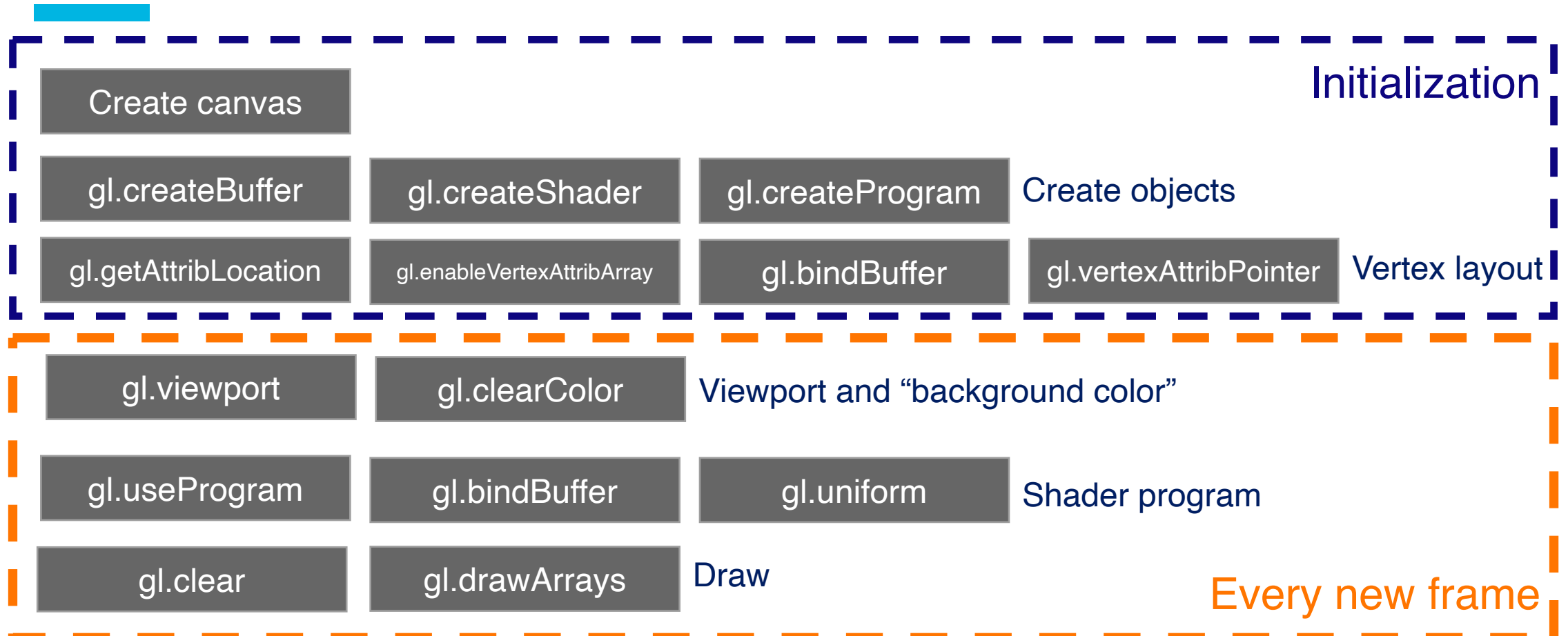
- Clear the canvas:

```
gl.clearColor(0, 0, 0, 0);  
gl.clear(gl.COLOR_BUFFER_BIT);
```

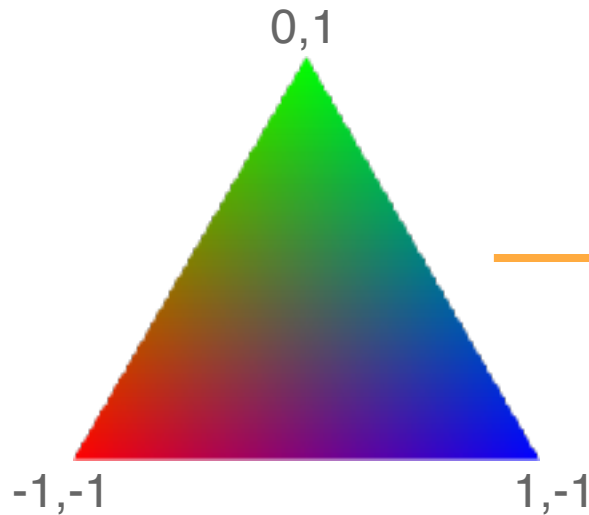
- Use program, bind VAO and draw arrays:

```
gl.useProgram(program);  
gl.bindVertexArray(vao);  
var primitiveType = gl.TRIANGLES;  
var count = 3;  
gl.drawArrays(primitiveType, 0, count);
```

# Overview



# Simple example



```
var vertices = [  
    0.0, 1.0, 0.0,  
    -1.0, -1.0, 0.0,  
    1.0, -1.0, 0.0  
];
```

```
var colors = [  
    0.0, 1.0, 0.0, 1.0,  
    1.0, 0.0, 0.0, 1.0,  
    0.0, 0.0, 1.0, 1.0,  
];
```

- Initialization:

1. Create shaders and program.
2. Create buffers.
3. Create VAO.

- Rendering:

1. Use program.
2. Bind VAO.
3. Draw arrays.

## Simple example: shaders

```
export default `#version 300 es

in vec4 position;
in vec4 color;

out vec4 vColor;

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

```
export default `#version 300 es

precision highp float;

in vec4 vColor;
out vec4 outColor;

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

## Simple example: Create shader

```
function createShader(type, source) {
  var shader = gl.createShader(type);

  gl.shaderSource(shader, source);
  gl.compileShader(shader);

  if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS) ) {
    var info = gl.getShaderInfoLog(shader);
    console.log('Could not compile WebGL program:' + info);
  }

  return shader;
}
```

## Simple example: Create program

```
function createProgram(vertexShader, fragmentShader) {
    var program = gl.createProgram();

    gl.attachShader(program, vertexShader);
    gl.attachShader(program, fragmentShader);
    gl.linkProgram(program);
    if (!gl.getProgramParameter(program, gl.LINK_STATUS) ) {
        var info = gl.getProgramInfoLog(program);
        console.log('Could not compile WebGL program:' + info);
    }

    return program;
}
```

## Simple example: Create buffer and VAO

```
function createBuffer(vertices) {
    var buffer= gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER,buffer);
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);

    return buffer;
}
```

## Simple example: Create buffer and VAO

```
function createVAO(posAttribLoc, colorAttribLoc) {
    var vao = gl.createVertexArray();

    gl.bindVertexArray(vao);
    gl.enableVertexAttribArray(posAttribLoc);
    var size = 3;
    var type = gl.FLOAT;
    gl.bindBuffer(gl.ARRAY_BUFFER, posBuffer);
    gl.vertexAttribPointer(posAttribLoc, size, type, false, 0, 0);

    gl.enableVertexAttribArray(colorAttribLoc);
    var size = 4;
    var type = gl.FLOAT;
    gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
    gl.vertexAttribPointer(colorAttribLoc, size, type, false, 0, 0);

    return vao;
}
```

## Simple example: Initialization

```
function main() {
  var canvas = document.querySelector("#glcanvas");
  canvas.width = canvas.clientWidth;
  canvas.height = canvas.clientHeight;
  gl = canvas.getContext("webgl2");

  var vertexShader = createShader(gl.VERTEX_SHADER, vertexShaderSrc);
  var fragmentShader = createShader(gl.FRAGMENT_SHADER, fragmentShaderSrc);
};

program = createProgram(vertexShader, fragmentShader);
posBuffer = createBuffer(vertices);
colorBuffer = createBuffer(colors);
var posAttribLoc = gl.getAttribLocation(program, "position");
var colorAttribLoc = gl.getAttribLocation(program, "color");
vao = createVAO(posAttribLoc, colorAttribLoc);
}

window.onload = main;
window.requestAnimationFrame(draw);
```

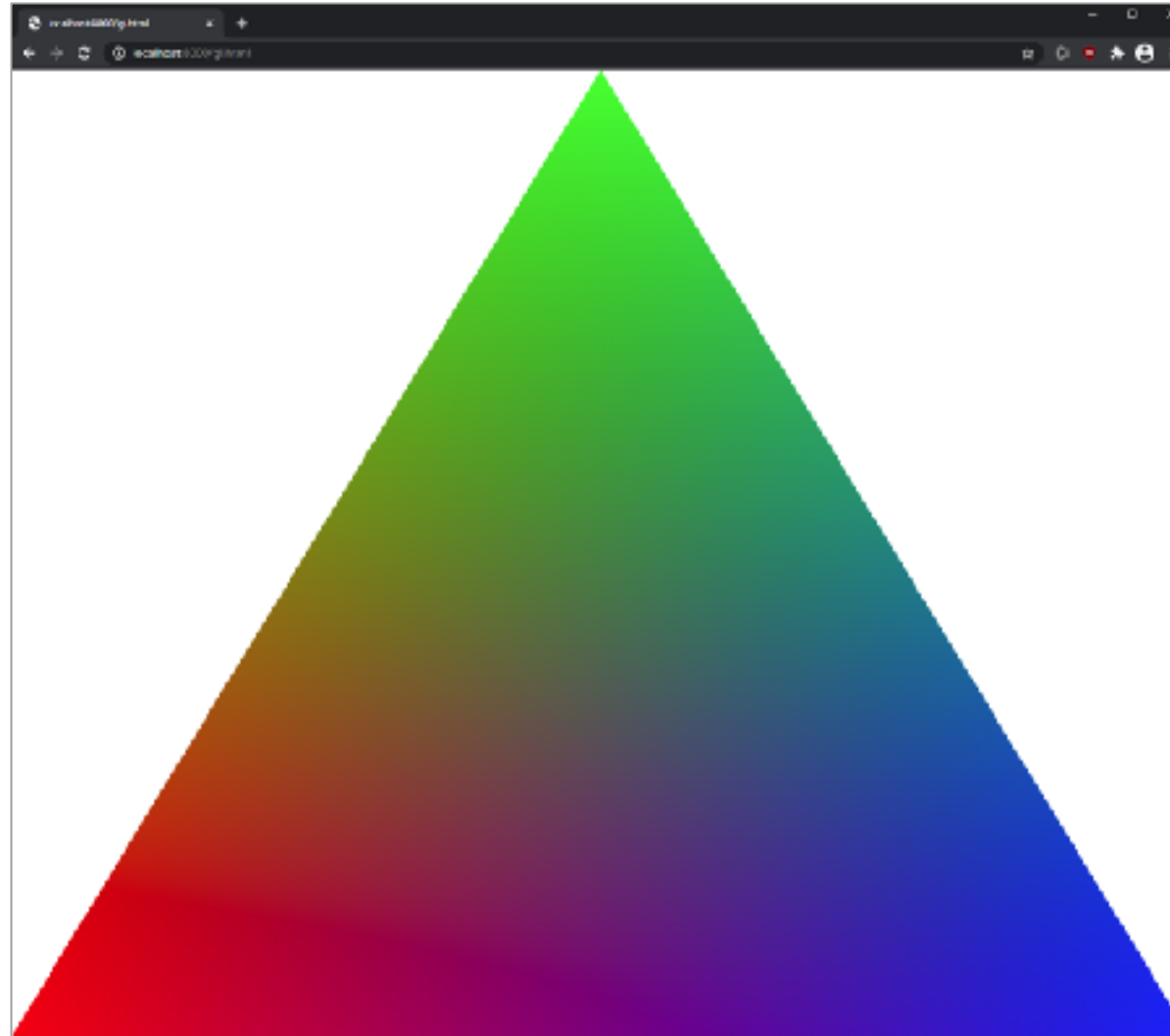
## Simple example: Render loop

```
function draw() {
    gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);

    gl.clearColor(1, 1, 1, 1);
    gl.clear(gl.COLOR_BUFFER_BIT);

    gl.useProgram(program);
    gl.bindVertexArray(vao);
    var primitiveType = gl.TRIANGLES;
    var count = 3;
    gl.drawArrays(primitiveType, 0, count);
}
```

# Simple example



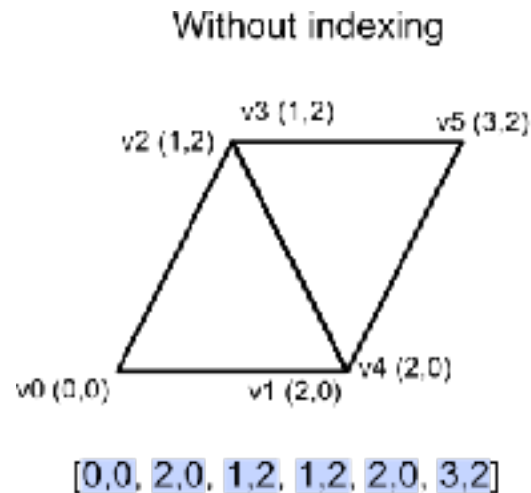
# Lab

---

- Create a web application to render triangles. This application should be composed of two main elements: a configuration panel, and a WebGL canvas.
- Randomly create 100 triangles, whose position should be random but should still be within the canvas
- One slider to specify the number of triangles to draw (between 0 — 100)
- Add 3 sliders to specify an RGB color that will control the color of the top vertex

# VBO indexing

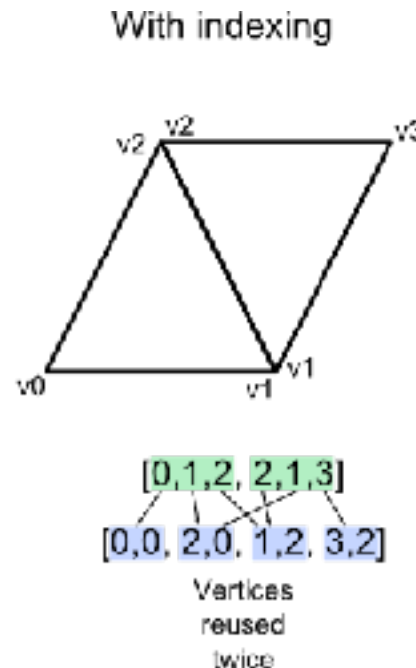
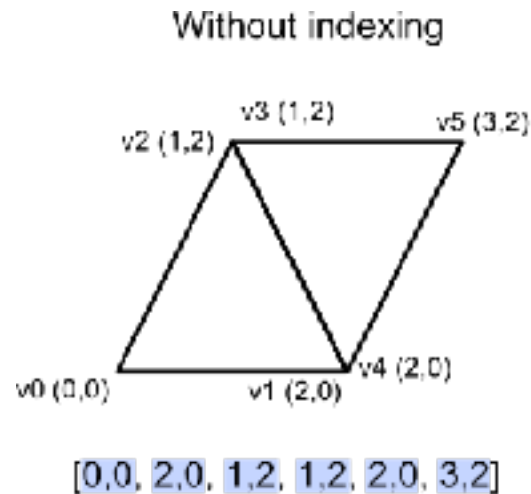
- In order to avoid duplicating our vertices whenever two triangles share an edge, we can use indexing.



```
gl.bindVertexArray(vao);  
gl.drawArrays(gl.TRIANGLES, 0, count  
);
```

# VBO indexing

- In order to avoid duplicating our vertices whenever two triangles share an edge, we can use indexing.

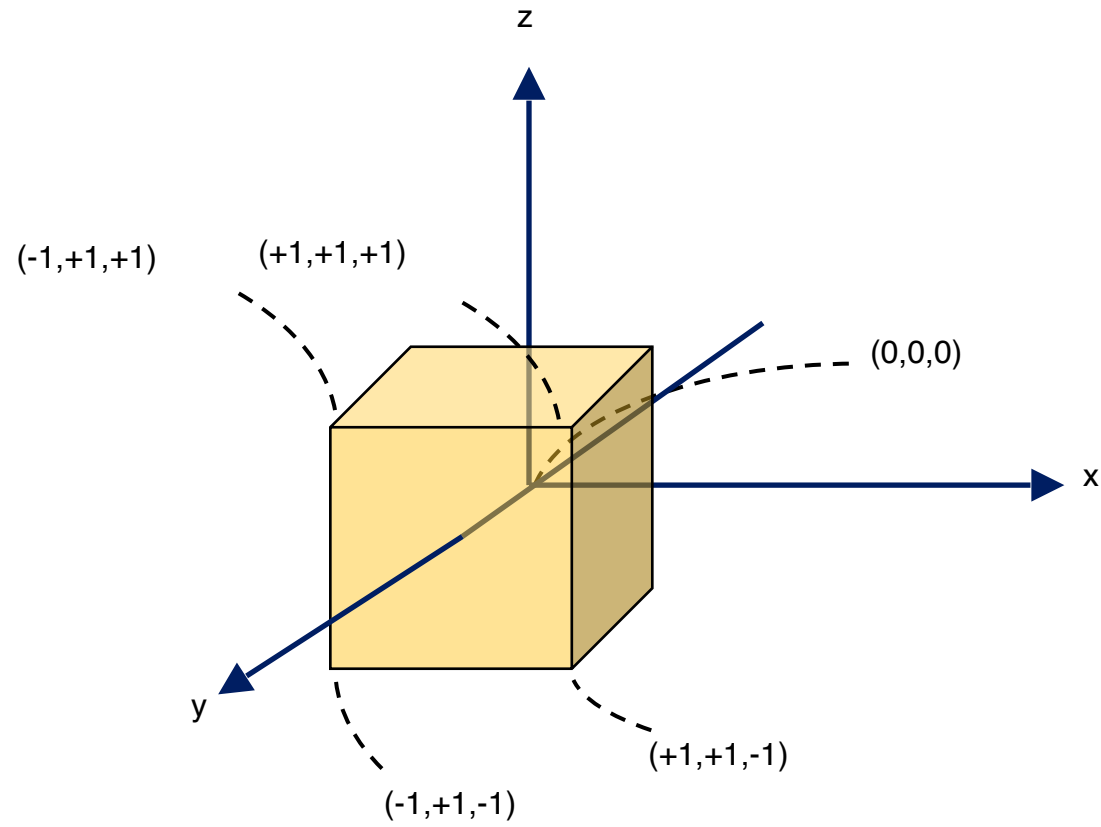


```
gl.bindVertexArray(vao);  
gl.drawArrays(gl.TRIANGLES, 0, count  
);
```

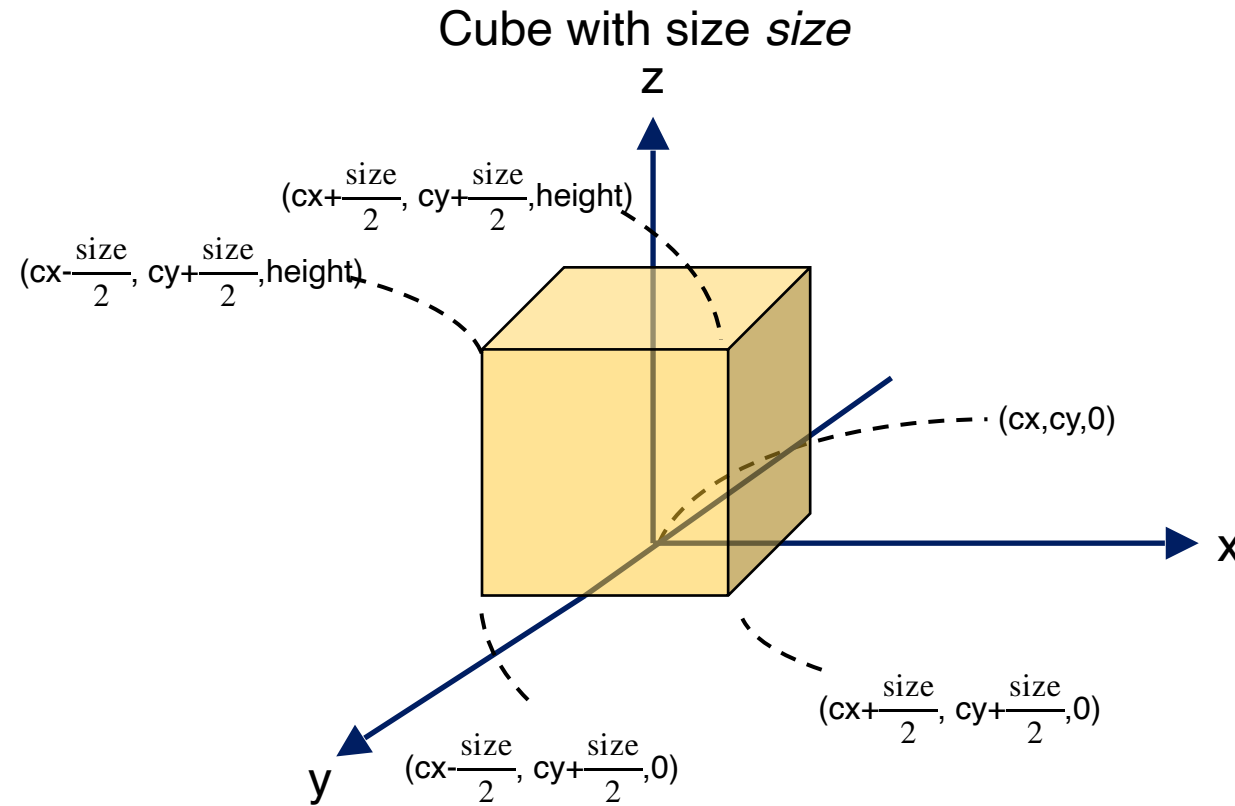
```
gl.bindVertexArray(vao);  
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER,  
buffers.elements);  
gl.drawElements(gl.TRIANGLES, count,  
gl.UNSIGNED_SHORT, 0);
```

# Cube example

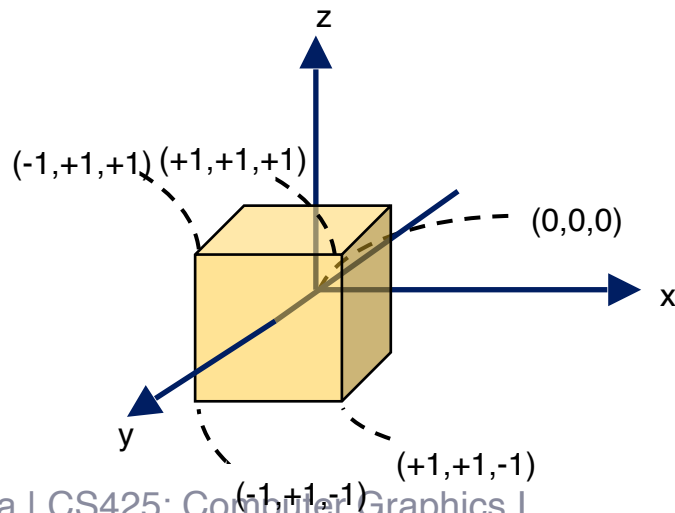
Cube with size 2



# Cube example

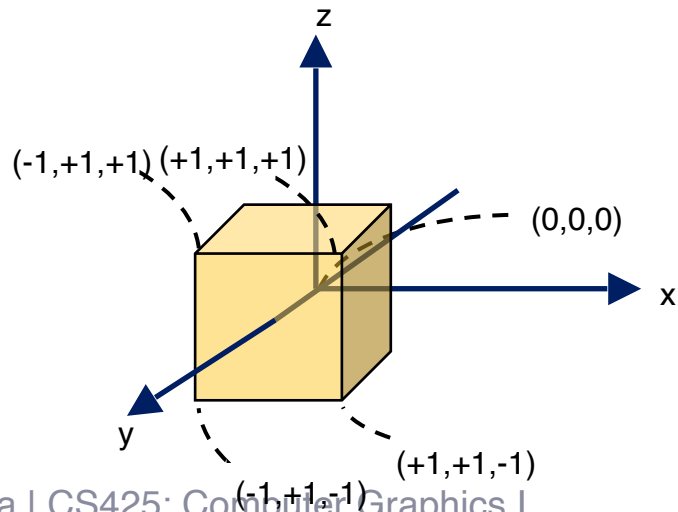


Cube with size 2

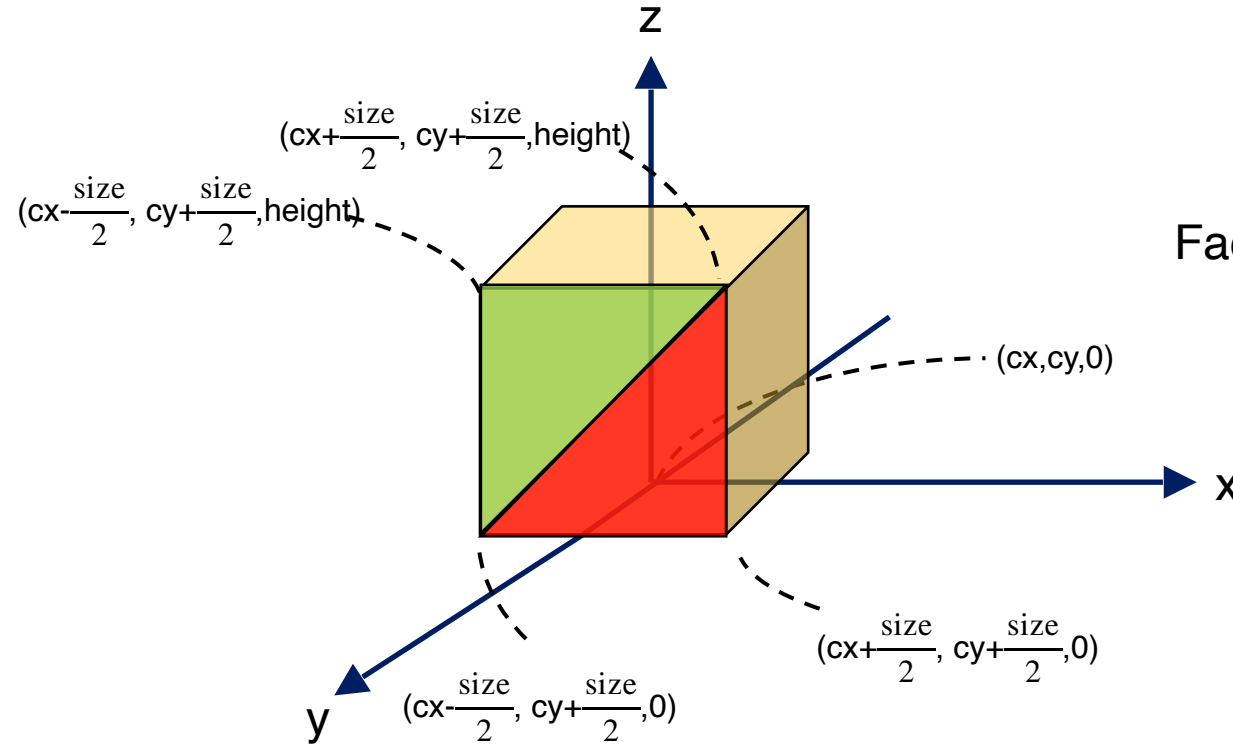


# Cube example

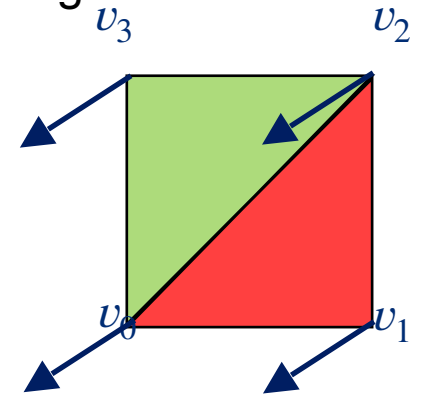
Cube with size 2



Cube with size *size*

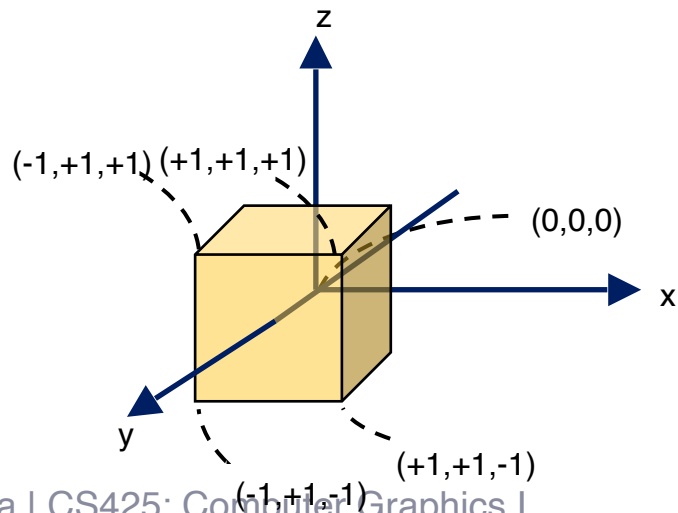


Face triangles with vertex normals

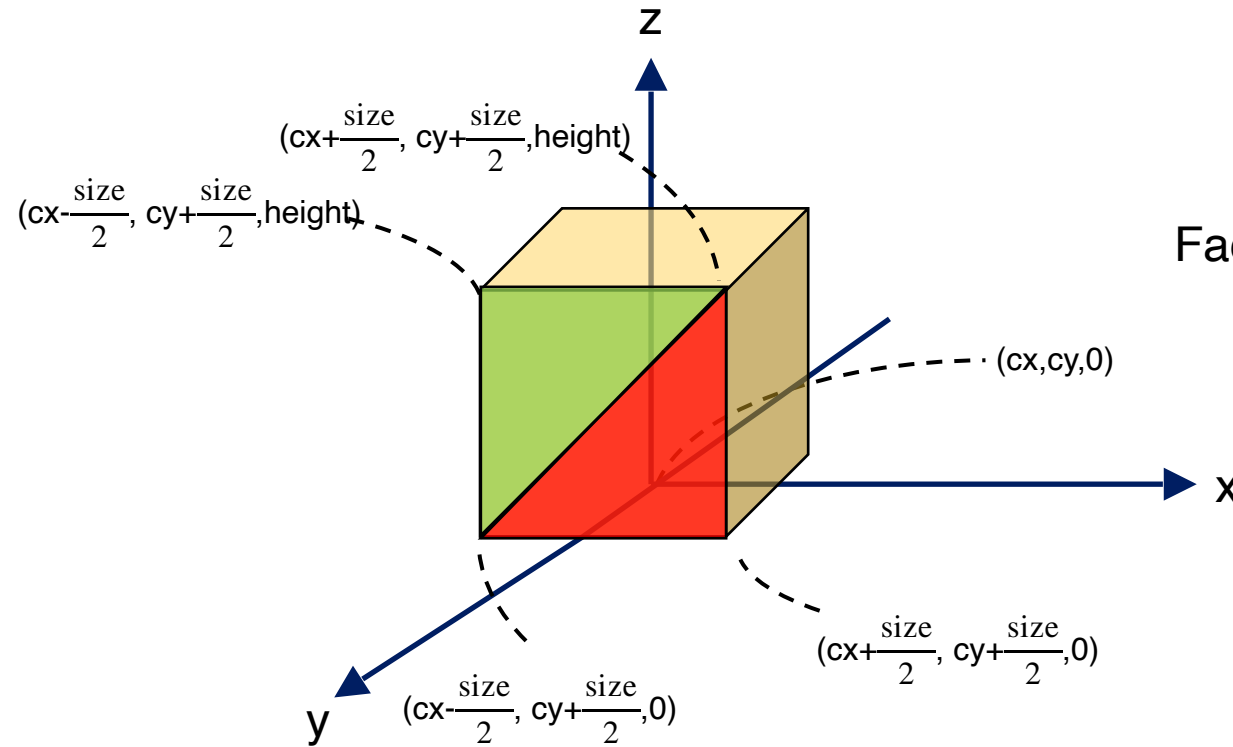


# Cube example

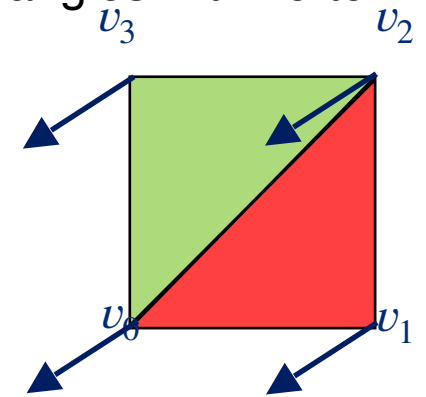
Cube with size 2



Cube with size *size*



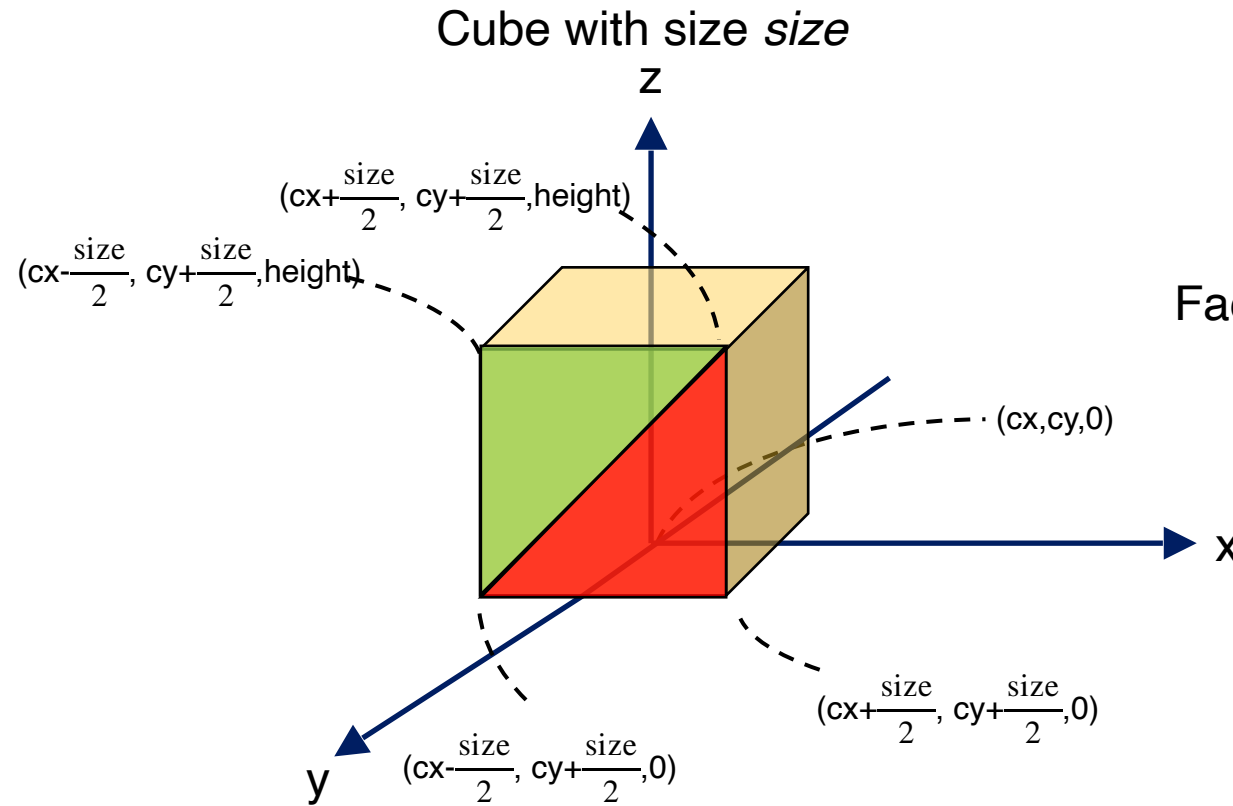
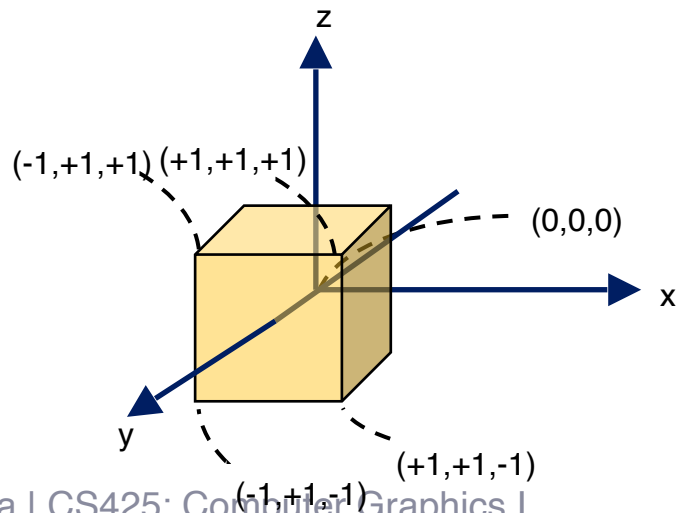
Face triangles with vertex normals



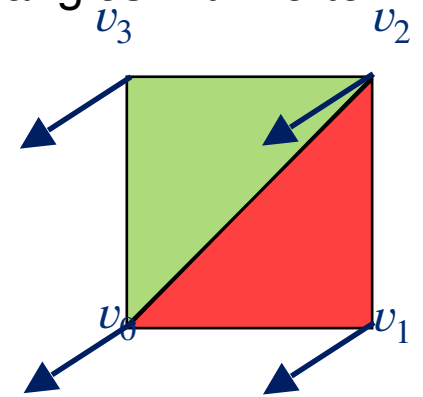
Vertex coordinates:  $[v_0, v_1, v_2, v_3]$   
 Normals:  $[0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0]$

# Cube example

Cube with size 2

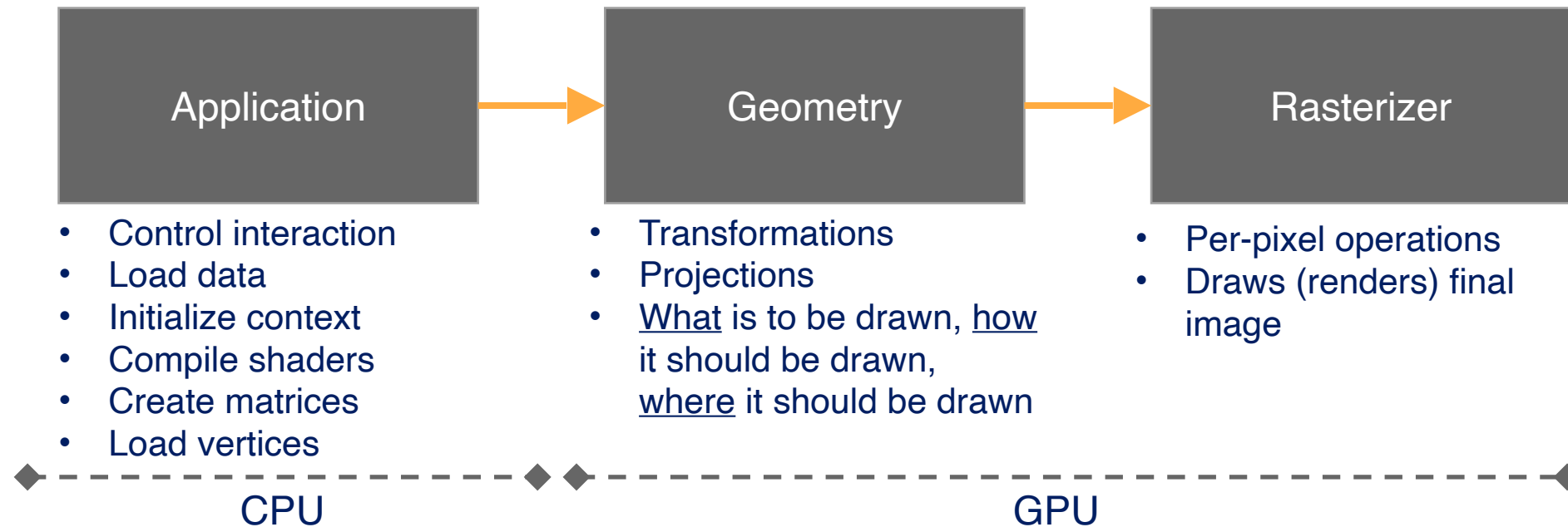


Face triangles with vertex normals



Vertex coordinates:  $[v_0, v_1, v_2, v_3]$   
 Normals:  $[0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0]$   
 Indices:  $[0, 1, 2, 0, 2, 3]$

# Rendering pipeline



Slowest pipeline stage will determine *rendering speed* (in frames per second).

Simple example: bottleneck stage takes 20 ms to execute. Rendering speed:  $1/0.020 = 50$  fps.

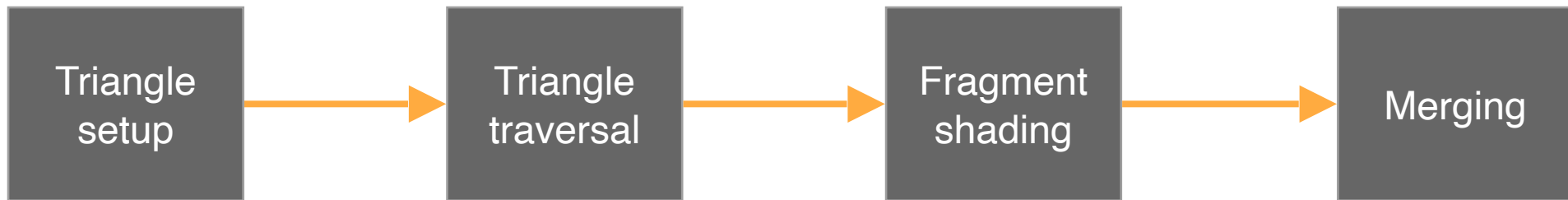
# Geometry stage

- Responsible for most of the per-polygon and per-vertex operations.



# Rasterizer stage

- Responsible for the computation of fragment and pixel colors.



# Lab

---

- Create a web application to render triangles. This application should be composed of two main elements: a configuration panel, and a WebGL canvas.
- The configuration panel should be composed of:
  1. Three sliders to set the current RGB color of triangles.
  2. One slider with the number of triangles to be rendered in the WebGL canvas (minimum of 1 triangle, maximum of 100 triangles). Position and size of triangles should be randomly selected, making sure that *all* triangles are rendered inside canvas.
  3. Start and stop buttons. Once start is pressed, current RGB color should randomly change, *smoothly* updating the RGB sliders and color of rendered triangles.  
**CAREFUL NOT TO CHANGE COLORS TOO FAST!**
- WebGL canvas should render triangles according to configuration panel.

# Rendering frame

- This will be executed every time we want to render.
- Set viewport (maps clip space to screen space):

```
gl.viewport(0, 0, gl.canvas.width, gl.canvas.height);
```

- Clear the canvas:

```
gl.clearColor(0, 0, 0, 0);  
gl.clear(gl.COLOR_BUFFER_BIT);
```

- Use program, bind VAO and draw arrays:

```
gl.useProgram(program);  
gl.bindVertexArray(vao);  
var primitiveType = gl.TRIANGLES;  
var count = 3;  
gl.drawArrays(primitiveType, 0, count);
```

# Useful links

---

[https://developer.mozilla.org/en-US/docs/Web/API/WebGL\\_API](https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API)

<https://www.khronos.org/registry/webgl/specs/latest/2.0/>

<https://www.khronos.org/files/webgl20-reference-guide.pdf>