

Introduction to WebGL

What is WebGL?

WebGL is a JavaScript API that allows us to implement interactive 3D graphics, straight in the browser.

WebGL runs as a specific context for the HTML `<canvas>` element, which gives you access to hardware-accelerated (GPU) 3D rendering in JavaScript.

Run on many different devices, such as desktop computers, mobile phones and TVs.

What can WebGL do?

Real-time interactive 3D graphics in browser.

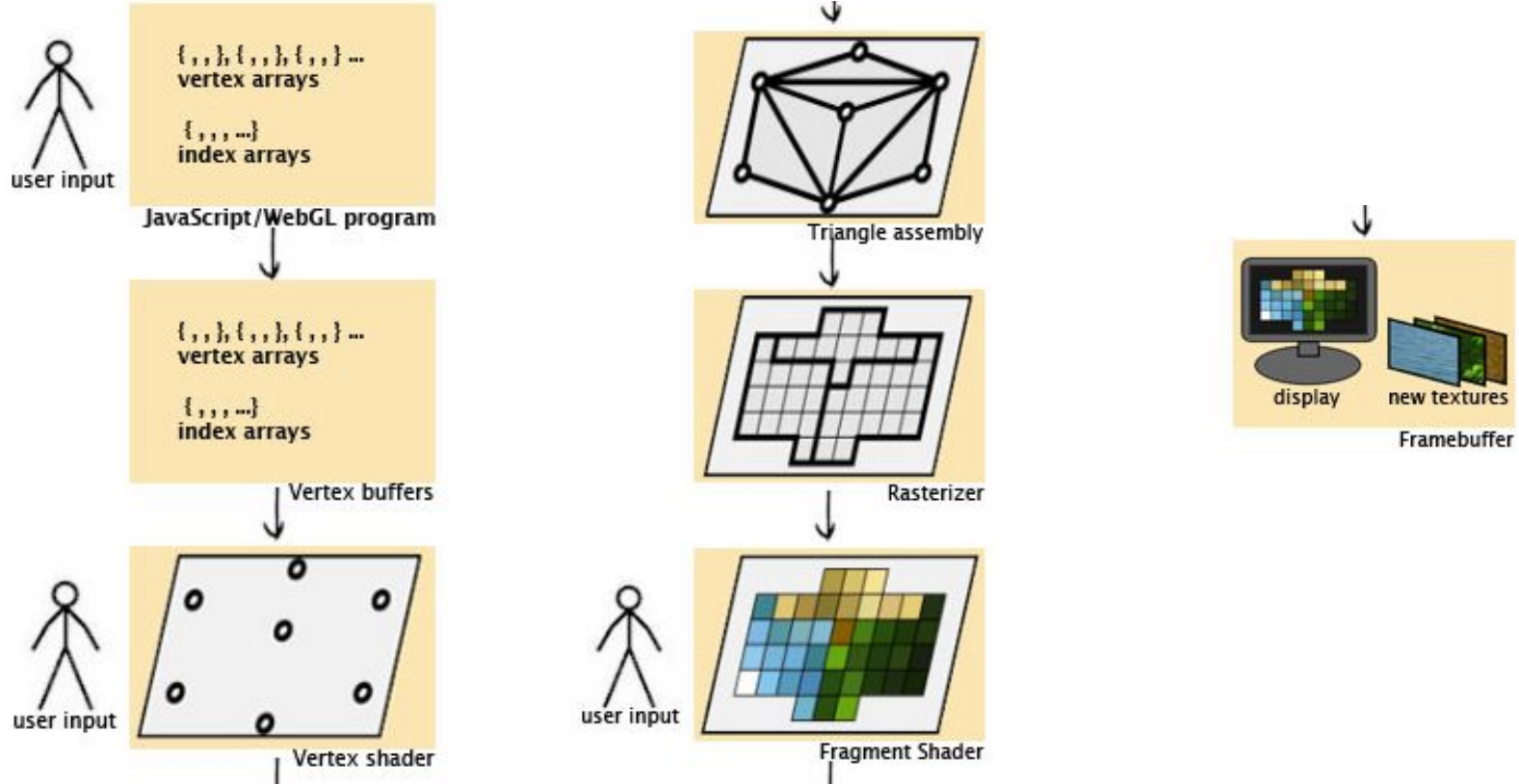
Interactive music videos, games, data visualization, art, 3D design environments, 3D modeling of space, plotting mathematical functions, physical simulations...

Demos:

<http://helloracer.com/webgl/>

<http://arodic.github.io/p/jellyfish/>

Rendering pipeline



Libraries?

Provide common functions, ready-made models, shaders

Faster development

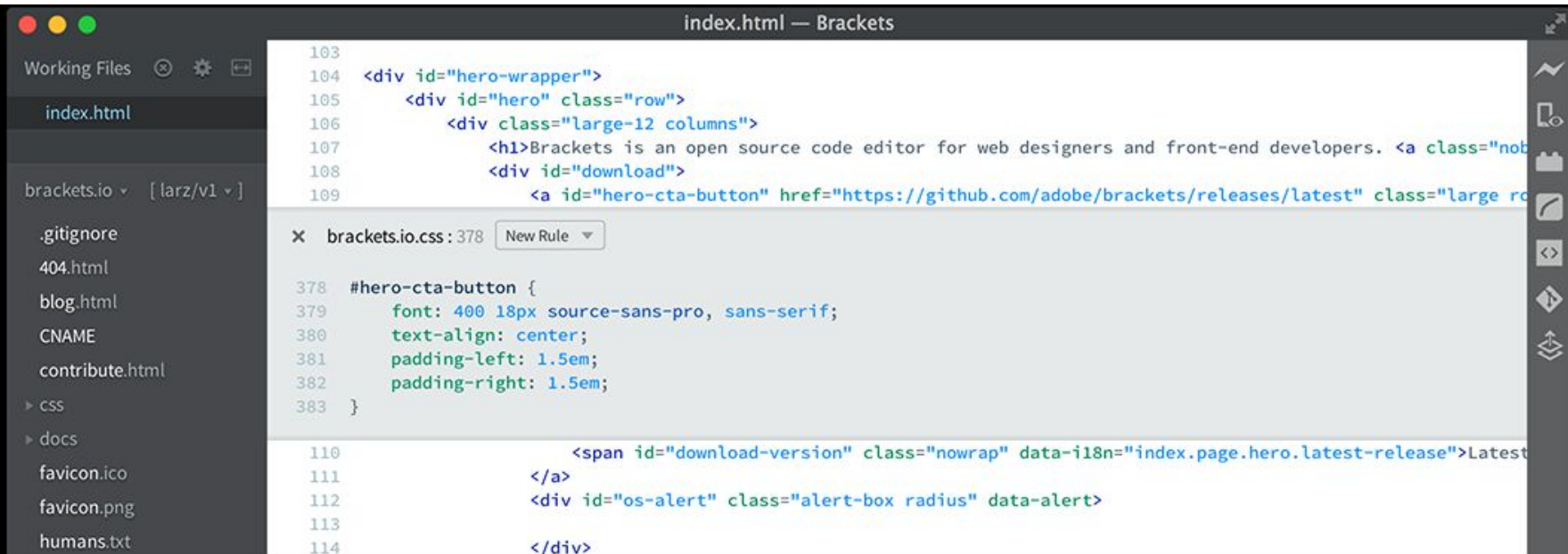
Robust

Popular ones: Three.js, PhiloGL, GLGE, J3D

In our class, we will directly work on WebGL for learning purpose.

You need a text editor

Brackets, Sublime Text....



The image shows a screenshot of the Brackets text editor. The title bar reads "index.html — Brackets". The left sidebar shows a file explorer with "index.html" selected. The main editor area displays HTML code for a hero section, including a "download" button. A CSS rule for "#hero-cta-button" is highlighted in the editor. The code is as follows:

```
103
104 <div id="hero-wrapper">
105   <div id="hero" class="row">
106     <div class="large-12 columns">
107       <h1>Brackets is an open source code editor for web designers and front-end developers. <a class="not
108       <div id="download">
109         <a id="hero-cta-button" href="https://github.com/adobe/brackets/releases/latest" class="large ro

X brackets.io.css: 378 New Rule ▾

378 #hero-cta-button {
379   font: 400 18px source-sans-pro, sans-serif;
380   text-align: center;
381   padding-left: 1.5em;
382   padding-right: 1.5em;
383 }

110     <span id="download-version" class="nowrap" data-i18n="index.page.hero.latest-release">Latest
111     </a>
112     <div id="os-alert" class="alert-box radius" data-alert>
113
114     </div>
```

Time to write some HTML

We will keep everything in a single HTML file for example HelloTriangle

For larger programs, we will separate the HTML and JavaScript, like we are doing in our programming assignment

Using WebGL entails writing a bunch of startup code

Complexity comes from the flexibility of the API

will enable you to do really sophisticated stuff later on

Eventually we will use a helper library for the startup code

Example code: graphics.ics.uci.edu/CS112/discussion_week1_examples.zip

The HTML

```
<!DOCTYPE HTML>
<html lang="en">
<head>
<title>Hello Triangle</title>
<meta charset="utf-8">
</head>
<body onload="startup();">
<canvas id="myGLCanvas"
width="500" height="500">

</canvas>
</body>
</html>
```

We create an HTML page

Notice:

We create an HTML5 **<canvas>**
That is 500 x 500 pixels which we
will draw into.

We give it an id so we can refer to
it in the javascript that we will
write.

onload specifies an entry point
into the JavaScript we will
write...a function named **startup()**
will be called on a page load

Adding JavaScript

```
<script type="text/javascript">
var gl;
var canvas;
var shaderProgram;
var vertexBuffer;

function startup() {
  canvas =
  document.getElementById("myGLCanvas");

  gl = createGLContext(canvas);
  setupShaders();
  setupBuffers();
  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  draw();
}
</script>
```

JavaScript is included inside **<script>** tags

We have some global variables...
...and our initial function calls some other functions.

Bolded functions are the ones we will write.

clearColor is a WebGL function that sets the initial color of the pixels in the raster

getElementById is a Document Object Model (DOM) function that gets us a reference to the canvas created in the HTML document

Getting a WebGL Context

```
function createContext(canvas) {
  var names = ["webgl", "experimental-webgl"];
  var context = null;
  for (var i=0; i < names.length; i++) {
    try {
      context = canvas.getContext(names[i]);
    } catch(e) {}
    if (context) {
      break;
    }
  }
  if (context) {
    context.viewportWidth = canvas.width;
    context.viewportHeight = canvas.height;
  } else {
    alert("Failed to create WebGL context!");
  }
  return context;
}
```

We need to make sure the browser supports WebGL...so we try to get a reference to a WebGL context using the two names under which it might exist

If we get a context, we set the viewport dimensions of the context to match the size of the canvas.

You can choose to use less than the full canvas.

Creating Vertex Shader

```
var vertexShaderSource =  
  "attribute vec3 aVertexPosition;      \n" +  
  "void main() {                        \n" +  
  "  gl_Position = vec4(aVertexPosition, 1.0); \n" +  
  "}"
```

We'll talk more about shaders later but for now you should know:

We need to create a vertex shader program written in GLSL

We will use a JavaScript string to hold the source code for the vertex shader. **We'll see a better way to do this later.**

The shader must assign a value to `gl_Position`

Our shader basically just takes the position of an incoming vertex and assigns that position to `gl_Position`.

It actually does one thing to the incoming position...do you know what that is?

Creating Fragment Shader

```
var fragmentShaderSource =  
    "precision mediump float;           \n"+  
    "void main() {                       \n"+  
    "  gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0); \n"+  
    "}                                     \n";
```

Like the vertex shader program, the fragment shader code is written in GLSL and held in a string.

You can think of fragments as being almost pixels...they are produced by the WebGL rasterizer and have a screen space position and some other data related to them.

Our shader simply assigns each fragment the same color.

Again, we'll talk more about what the shaders do later...

Compiling the Shaders

```
function setupShaders() {  
  
    var vertexShaderSource = ...  
    var fragmentShaderSource = ...  
  
    var vertexShader = loadShader(gl.VERTX_SHADER,  
    vertexShaderSource);  
  
    var fragmentShader = loadShader(gl.FRAGMENT_SHADER,  
    fragmentShaderSource);  
  
    ...  
}
```

```
function loadShader(type, shaderSource) {  
  
    var shader = gl.createShader(type);  
    gl.shaderSource(shader, shaderSource);  
    gl.compileShader(shader);  
  
    if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {  
        alert("Error compiling shader" +  
            gl.getShaderInfoLog(shader));  
        gl.deleteShader(shader);  
        return null;  
    }  
    return shader;  
}
```

We have a homemade helper function that compiles the shader and checks if there were compilation errors.

If there was an error, a JavaScript alert is issued and the shader object deleted.

Otherwise the compiled shader is returned.

Creating Program Object and Linking Shaders

```
function setupShaders() {  
    ...  
    shaderProgram = gl.createProgram();  
    gl.attachShader(shaderProgram, vertexShader);  
    gl.attachShader(shaderProgram, fragmentShader);  
    gl.linkProgram(shaderProgram);  
  
    if (!gl.getProgramParameter(shaderProgram,  
        gl.LINK_STATUS)) {  
        alert("Failed to setup shaders");  
    }  
  
    gl.useProgram(shaderProgram);  
  
    shaderProgram.vertexPositionAttribute =  
        gl.getAttribLocation(shaderProgram,  
            "aVertexPosition");  
}
```

We create a program object and attach the compiled shaders and link. At this point, we have a complete shader program that WebGL can use.

attributes are user-defined variables that contain data specific to a vertex.

The **attributes** used in the vertex shader are bound to an index (basically a number given to a slot). Our code needs to know the index associated with the attributes we use in the shader so that our draw function can feed the data correctly.

`vertexPositionAttribute` is a user-defined property in which we remember the index value

Setting up the Buffers

```
function setupBuffers() {  
  vertexBuffer = gl.createBuffer();  
  gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);  
  var triangleVertices = [  
    0.0, 0.5, 0.0,  
    -0.5, -0.5, 0.0,  
    0.5, -0.5, 0.0  
  ];  
  gl.bufferData(gl.ARRAY_BUFFER,  
    new Float32Array(triangleVertices),  
    gl.STATIC_DRAW);  
  vertexBuffer.itemSize = 3;  
  vertexBuffer.numberOfItems = 3;  
}
```

We next need to create a buffer that will hold the vertex data...this is the geometric data of the shapes we wish to render.

We create a WebGL buffer object and bind it so that WebGL knows it is the current buffer to work with.

`triangleVertices` is a user-defined JavaScript array containing the 3D coordinates of a single triangle.

We call a magic function to copy the vertex positions into the current WebGL buffer.

Two user-defined properties are used to remember how many vertices we have and how many coordinates per vertex.

Drawing the Scene

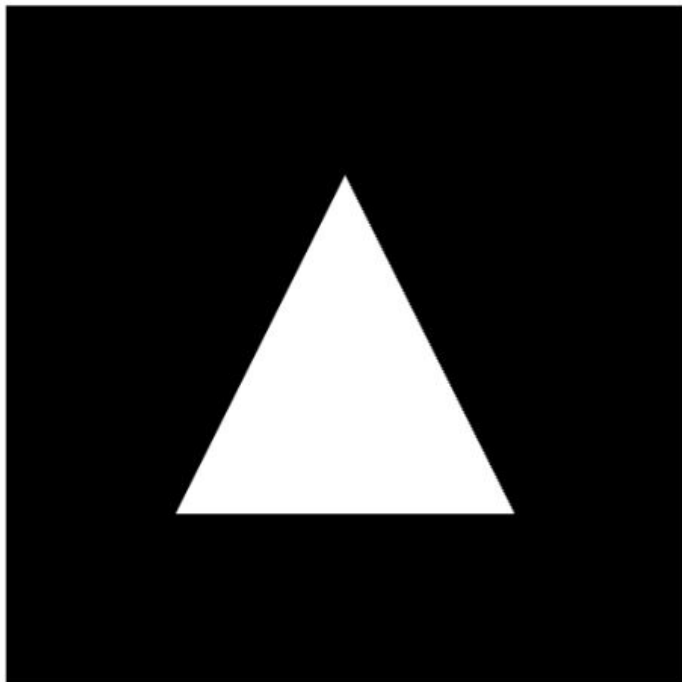
```
function draw() {  
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);  
  
  gl.clear(gl.COLOR_BUFFER_BIT);  
  
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,  
                        vertexBuffer.itemSize, gl.FLOAT, false, 0, 0);  
  
  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);  
  
  gl.drawArrays(gl.TRIANGLES, 0, vertexBuffer.numberOfItems);  
}
```

The **viewport** method lets us tell WebGL how to convert from *clip space* in which coordinates range from -1 to 1 back into pixel coordinates. Here we use our two user-defined properties to set it to the full size of the canvas.

clear initializes the color buffer to the color set with **clearColor**.

We then tell WebGL to take values for aVertexPosition from the buffer currently bound to gl.ARRAY_BUFFER....and then we draw.

Result

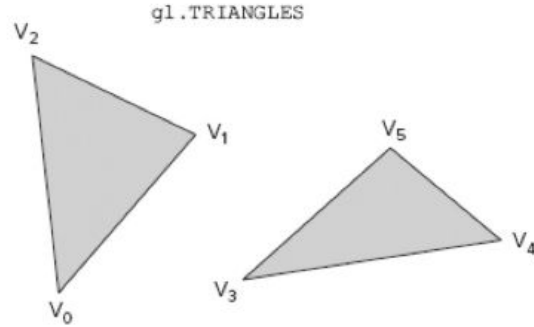


Can you?

- ❑ Change the triangle color?
- ❑ Change the background color?
- ❑ Change the triangle shape?
- ❑ Draw multiple triangles?
- ❑ Make the triangle look smaller without changing the vertex data?

Programming Assignment 0

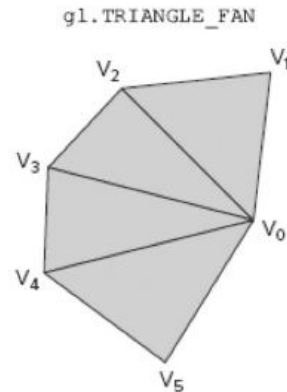
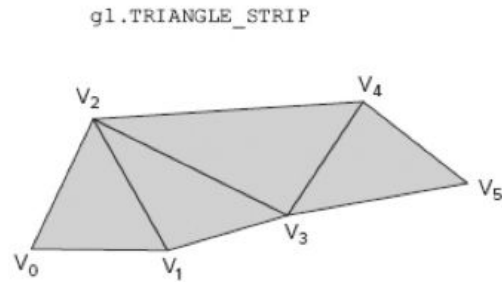
Geometric Primitives in WebGL



WebGL supports 3 basic geometric primitives:

1. Triangles
2. Lines
3. Point Sprites

We've already seen one way to send triangles into the pipeline.

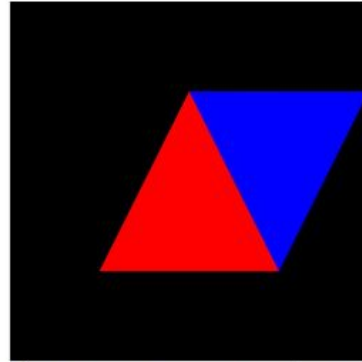


There are three different triangle drawing modes depending on how you specify the connectivity:

`gl.TRIANGLES`
`gl.TRIANGLE_STRIP`
`gl.TRIANGLE_FAN`

gl.TRIANGLES

```
vertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexPositionBuffer);
var triangleVertices = [
    0.0, 0.5, 0.0,
    -0.5, -0.5, 0.0,
    0.5, -0.5, 0.0,
    0.0, 0.5, 0.0,
    1.0, 0.5, 0.0,
    0.5, -0.5, 0.0
];
gl.bufferData(gl.ARRAY_BUFFER, new
    Float32Array(triangleVertices), gl.STATIC_DRAW);
vertexPositionBuffer.itemSize = 3;
vertexPositionBuffer.numberOfItems = 6;
...
gl.drawArrays(gl.TRIANGLES, 0,
    vertexPositionBuffer.numberOfItems);
```



- Assuming you are using `gl.drawArrays()`:
 - Each triangle requires you specify three new vertices
 - i.e. you can't reference vertex data already in the buffer
 - Number of triangles = number vertices/3

gl.TRIANGLE_STRIP

```
vertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexPositionBuffer);
var triangleVertices = [
    -0.5, -0.5, 0.0,
    0.5, -0.5, 0.0,
    0.0, 0.5, 0.0,
    1.0, 0.5, 0.0,
];
gl.bufferData(gl.ARRAY_BUFFER, new
    Float32Array(triangleVertices), gl.STATIC_DRAW);
vertexPositionBuffer.itemSize = 3;
vertexPositionBuffer.numberOfItems = 4;
....
gl.drawArrays(gl.TRIANGLE_STRIP, 0,
    vertexPositionBuffer.numberOfItems);
```



- Allows you to reuse vertices when drawing triangles that share vertices.
- Number of triangles = what?
- Notice that per-triangle color is not easy to achieve
- Order of the vertices is important

gl.TRIANGLE_FAN

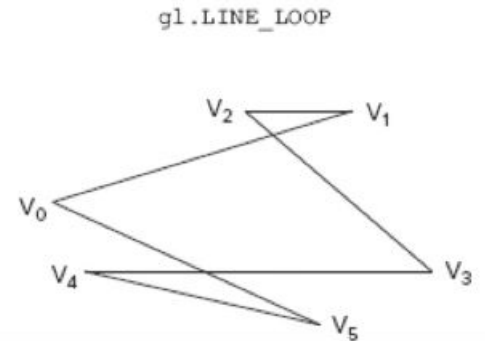
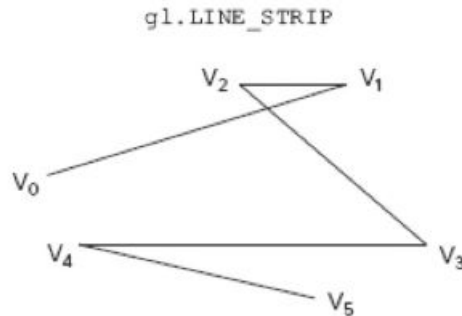
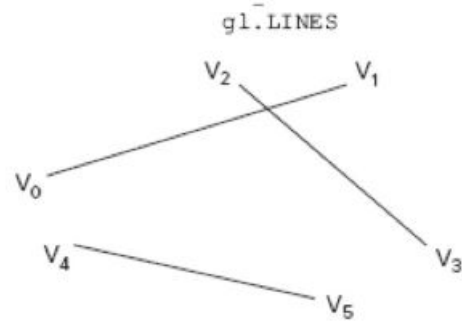
```
vertexPositionBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vertexPositionBuffer);
var triangleVertices = [
    0.5, -0.5, 0.0,
    1.0, 0.5, 0.0,
    0.0, 0.5, 0.0,
    -0.5, -0.5, 0.0,
];
gl.bufferData(gl.ARRAY_BUFFER, new
    Float32Array(triangleVertices), gl.STATIC_DRAW);
vertexPositionBuffer.itemSize = 3;
vertexPositionBuffer.numberOfItems = 4;
....
gl.drawArrays(gl.TRIANGLE_FAN, 0,
    vertexPositionBuffer.numberOfItems);
```



- ❑ First vertex is the fan center
- ❑ Next two vertices specify the first triangle
- ❑ Each succeeding vertex forms a triangle with the center and previous vertex
- ❑ How many triangles for a given number of vertices?
- ❑ Are fans and strips equivalent?

gl.LINES

- gl.LINES draws independent lines
(v0,v1), (v2,v3), (v4,v5)
- gl.LINE_STRIP draws a polyline
(v0,v1),(v1,v2),(v2,v3),(v3,v4),(v4,v5)
- gl.LINE_LOOP draws a line strip with a
line connecting the first and final vertex



gl.POINTS

- ❑ Specified with `gl.POINTS` mode
- ❑ Renders one point per vertex in the buffer
- ❑ using `N` pixels in the point is specified using `glPointSize(N)`