FREE eBook

LEARNING webgl

Free unaffiliated eBook created from
Stack Overflow contributors.

#webgl

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: webgl

It is an unofficial and free webgl ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official webgl.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with webgl

Remarks

WebGL is a rasterization API that generally runs on your GPU giving you the ability to quickly draw 2D and 3D graphics. WebGL can also be used to do computations on arrays of data.

WebGL is a very low-level API. At a base level WebGL is an engine that runs 2 user supplied functions on the GPU. One function is called a *vertex shader*. A vertex shader's job is to compute vertex positions. Based on the positions the function outputs WebGL can then rasterize various kinds of primitives including points, lines, or triangles. When rasterizing these primitives it calls a second user supplied function called a *fragment shader*. A fragment shader's job is to compute a color for each pixel of the primitive currently being drawn.

These functions are written in a language called GLSL that is somewhat C/C++ like and strictly typed.

It's up to the programmer to supply those functions to make WebGL draw 2d, 3d or compute something. Nearly every piece of WebGL is about setting up those 2 functions and then supplying data to them.

Data can be supplied from 4 sources.

- Uniforms

Uniforms are inputs to shader functions very much like function parameters or global variables. They are set once before a shader is executed and remain constant during executing

- Attributes

Attributes supply data to vertex shaders only. Attributes define how to pull data out of buffers. For example you might put positions, normal, and texture coordinates into a buffer. Attributes let you tell WebGL how to pull that data out of your buffers and supply them to a vertex shader. A vertex shader is called a user specified number of times by calling `gl.drawArrays` or `gl.drawElements` and specifying a count. Each time the current vertex shader is called the next set of data will be pulled from the user specified buffers and put in the attributes

- Textures

Textures are 2D arrays of data up 4 channels. Most commonly those 4 channels are red, green, blue, and alpha from an image. WebGL doesn't care what the data is though. Unlike attributes and buffers, shaders can read values from textures with random access.

- Varyings

Varyings are a way for a vertex shader to pass data to a fragment shader. Varyings are interpolated between the values output by the vertex shader as a primitive is rasterized using a

fragment shader

Examples

Installation or Setup

WebGL is a browser technology so there isn't much to set up other than to have a browser. You can get started with WebGL on JSFiddle or Codepen or JSBIn or any number of other sites that let you edit HTML, CSS, and JavaScript online though there will be a few limitations (see below). You can also host open source files on github pages or similar services.

On the other hand at some point you're probably going to work locally. To do that it's recommended you run a simple web server. There are plenty to choose from that are simple to use and require very little setup.

Using node.js as a server

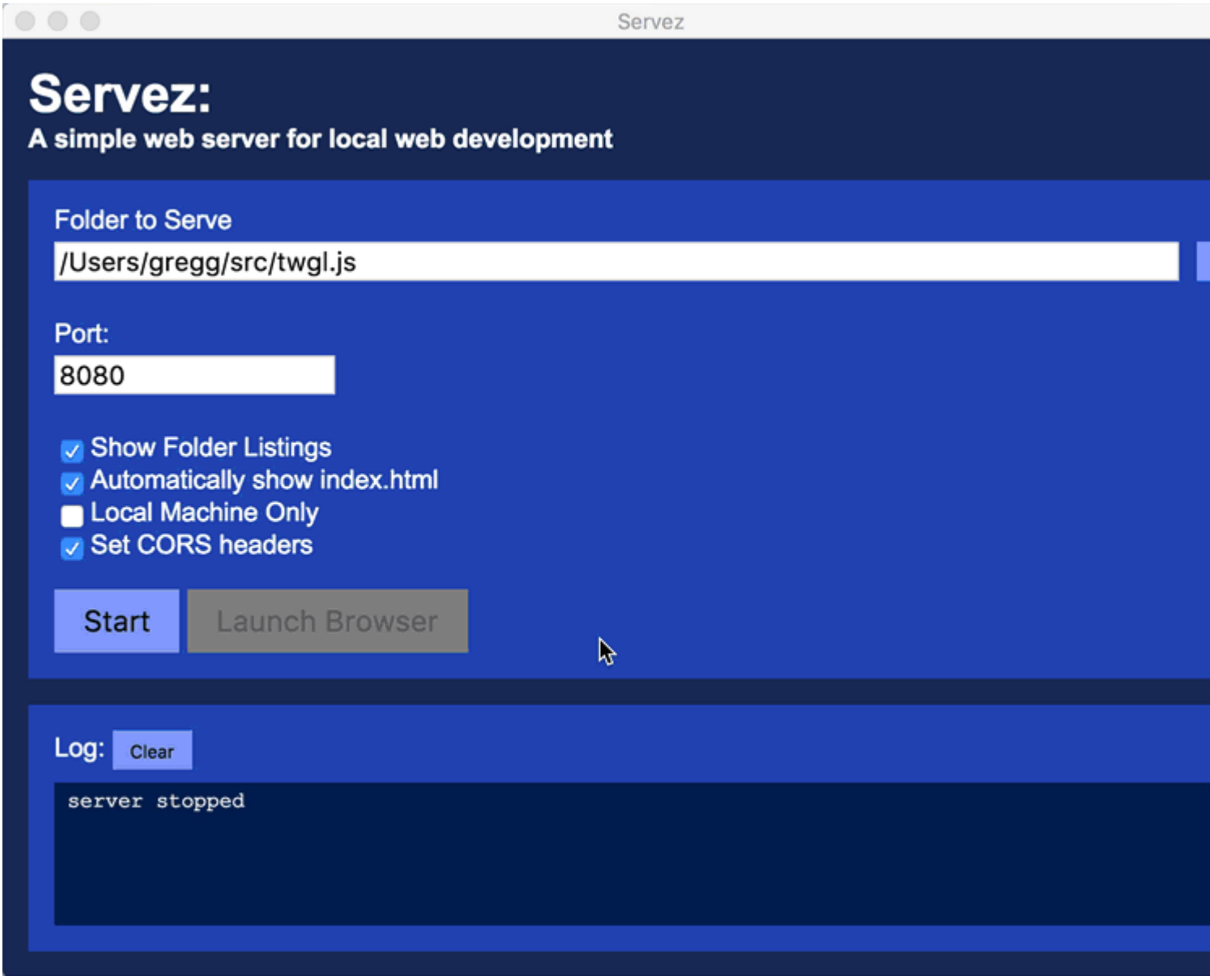
1. install node.js
2. Open a terminal or node command prompt and type `npm install -g http-server` (on OSX put `sudo` in front of that.
3. type `http-server` to start serving the files in the current folder OR `http-server path-to-folder` to server a different folder
4. Point your browser to `http://localhost:8080/name-of-file` to view your WebGL webpage

Using devd as a server

1. Download devd
2. Open a terminal and run devd with either `devd .` to server files from the current folder or `devd path-to-folder` to serve a different folder
3. Point your browser to `http://localhost:8000/name-of-file` to view your WebGL webpage

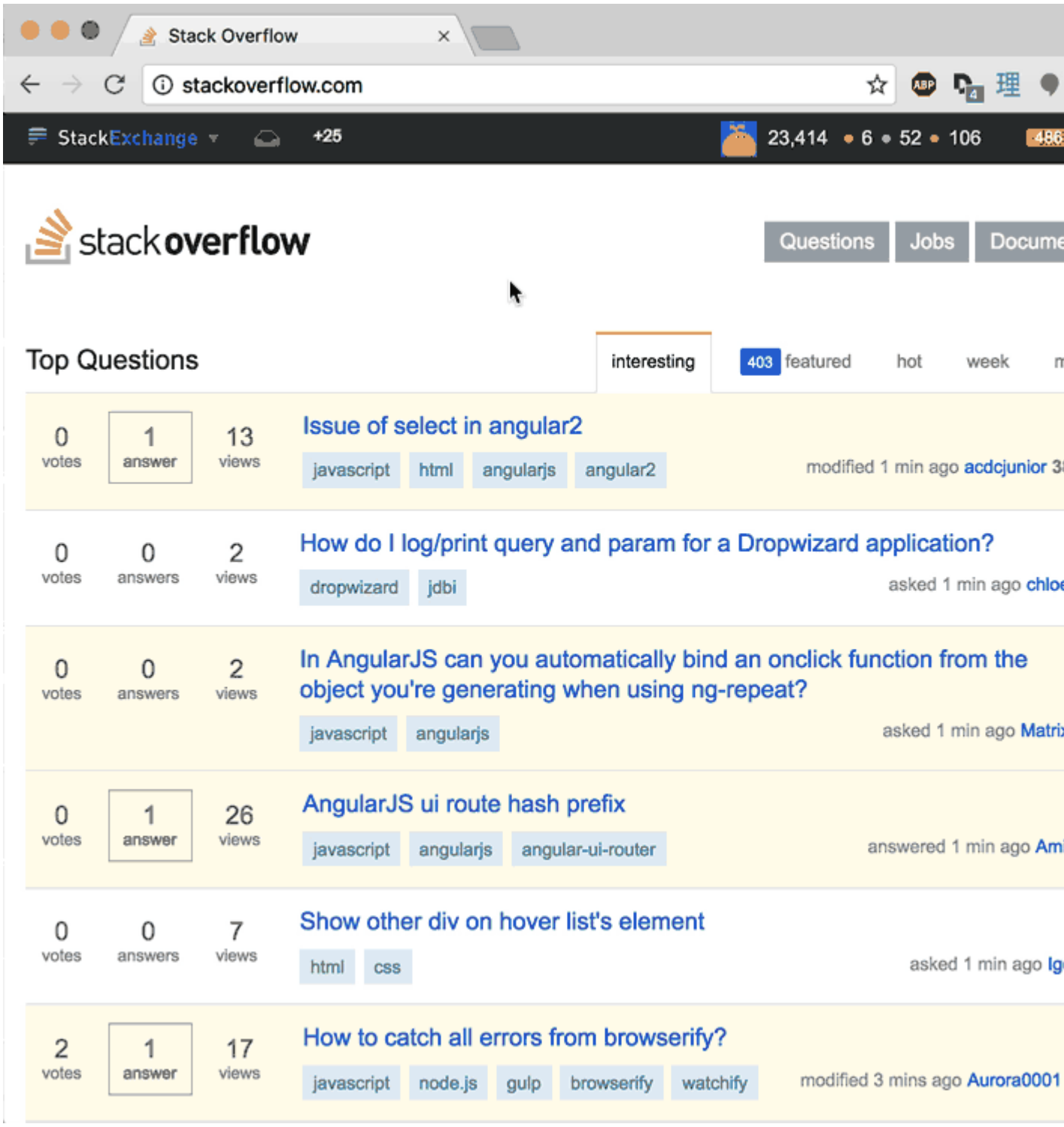
Using Servez as a server

1. Download Servez
2. Install It, Run it
3. Choose the folder to serve
4. Pick "Start"
5. Go to `http://localhost:8080` or pick "Launch Browser"

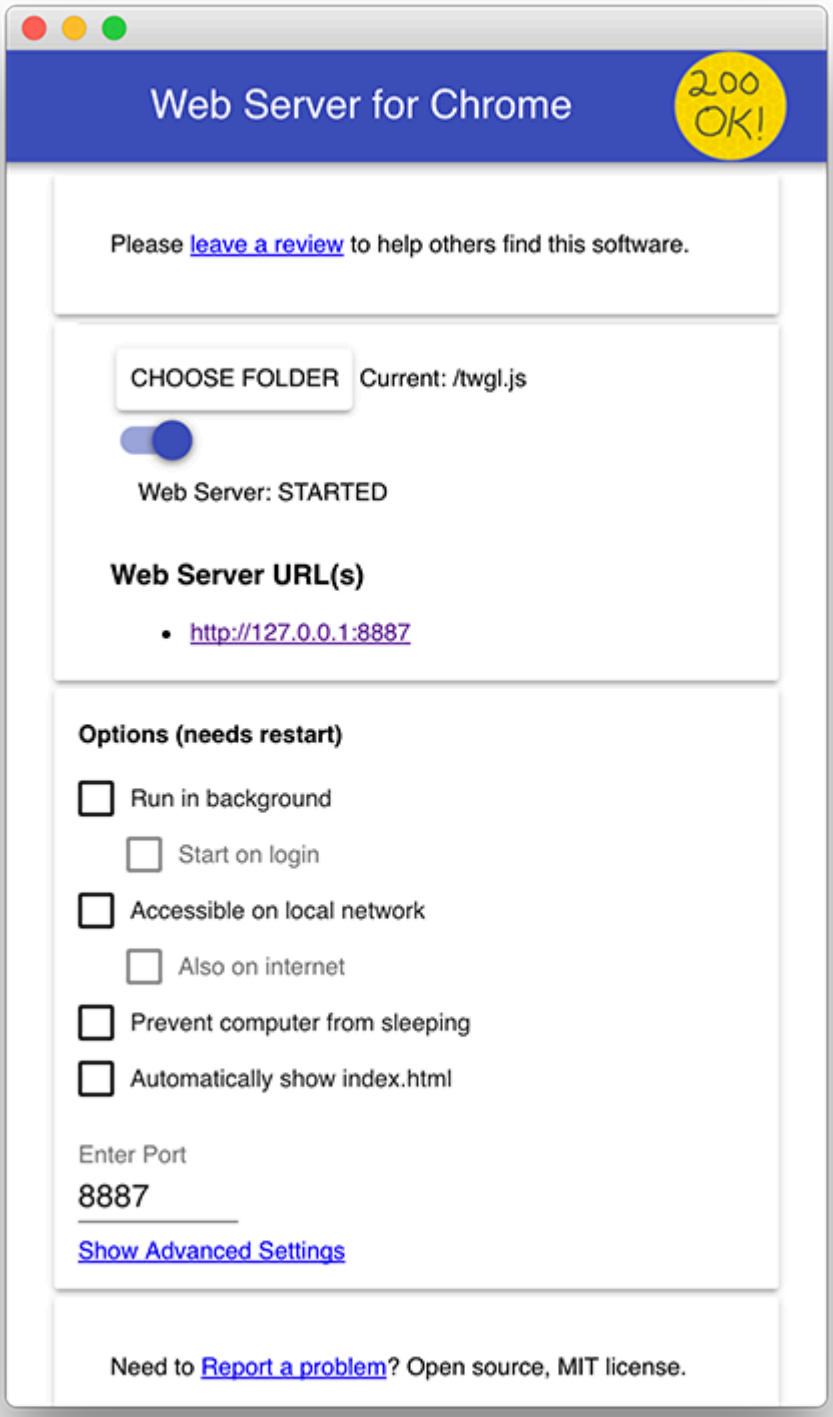


Using "*Web Server for Chrome*" Chrome Extension

1. Install the Web Server From Chrome
2. Launch it from the *Apps* icon on a new tab page.



3. Set the folder where your files are then click the `http://127.0.0.1:8787` link



Limitation of WebGL on Online Services

In WebGL it is very common to load images. In WebGL there are restrictions on how images can be used. Specifically WebGL can not use images from other domains without permission from the server hosting the images. Services that currently give permission to use images include imgur and flickr. See Loading Cross Domain Images. Otherwise you'll need to have the images on the same server as your webgl page or use other creative solutions like generating images with a

canvas tag

Hello World

Like it mentions in the *remarks* section we need to supply two functions. A vertex shader and a fragment shader

Let's start with a vertex shader

```
// an attribute will receive data from a buffer
attribute vec4 position;

// all shaders have a main function
void main() {

  // gl_Position is a special variable a vertex shader
  // is responsible for setting
  gl_Position = position;
}
```

If the entire thing was written in JavaScript instead of GLSL you could imagine it would be used like this

```
// *** PSUEDO CODE!! ***

var positionBuffer = [
  0, 0, 0, 0,
  0, 0.5, 0, 0,
  0.7, 0, 0, 0,
];
var attributes = {};
var gl_Position;

drawArrays(..., offset, count) {
  for (var i = 0; i < count; ++i) {
     // copy the next 4 values from positionBuffer to the position attribute
     attributes.position = positionBuffer.slice((offset + i) * 4, 4);
     runVertexShader();
     ...
     doSomethingWith_gl_Position();
}
```

Next we need a fragment shader

```
// fragment shaders don't have a default precision so we need
// to pick one. mediump, short for medium precision, is a good default.
precision mediump float;

void main() {
  // gl_FragColor is a special variable a fragment shader
  // is responsible for setting
  gl_FragColor = vec4(1, 0, 0.5, 1); // return redish-purple
}
```

Above we're setting `gl_FragColor` to `1, 0, 0.5, 1` which is 1 for red, 0 for green, 0.5 for blue, 1 for

alpha. Colors in WebGL go from 0 to 1.

Now that we have written the 2 functions lets get started with WebGL

First we need an HTML canvas element

```
<canvas id="c"></canvas>
```

Then in JavaScript we can look that up

```
var canvas = document.getElementById("c");
```

Now we can create a WebGLRenderingContext

```
var gl = canvas.getContext("webgl");
if (!gl) {
   // no webgl for you!
   ...
```

Now we need to compile those shaders to put them on the GPU so first we need to get them into strings. You can get your strings any way you normal get strings. By concatenating, by using AJAX, by putting them in non-JavaScript typed script tags, or in this case by using multiline template literals

```
var vertexShaderSource = `
// an attribute will receive data from a buffer
attribute vec4 position;

// all shaders have a main function
void main() {

  // gl_Position is a special variable a vertex shader
  // is responsible for setting
  gl_Position = position;
}
`;

var fragmentShaderSource = `
// fragment shaders don't have a default precision so we need
// to pick one. mediump is a good default
precision mediump float;

void main() {
  // gl_FragColor is a special variable a fragment shader
  // is responsible for setting
  gl_FragColor = vec4(1, 0, 0.5, 1); // return redish-purple
}
`;
```

Then we need a function that will create a shader, upload the source and compile the shader

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

```
  gl.compileShader(shader);
  var success = gl.getShaderParameter(shader, gl.COMPILE_STATUS);
  if (success) {
    return shader;
  }

  console.log(gl.getShaderInfoLog(shader));
  gl.deleteShader(shader);
}
```

We can now call that function to create the 2 shaders

```
var vertexShader = createShader(gl, gl.VERTEX_SHADER, vertexShaderSource);
var fragmentShader = createShader(gl, gl.FRAGMENT_SHADER, fragmentShaderSource);
```

We then need to *link* those 2 shaders into a *program*

```
function createProgram(gl, vertexShader, fragmentShader) {
  var program = gl.createProgram();
  gl.attachShader(program, vertexShader);
  gl.attachShader(program, fragmentShader);
  gl.linkProgram(program);
  var sucesss = gl.getProgramParameter(program, gl.LINK_STATUS);
  if (success) {
    return program;
  }

  console.log(gl.getProgramInfoLog(program));
  gl.deleteProgram(program);
}
```

And call it

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

Now that we've created a GLSL program on the GPU we need to supply data to it. The majority of the WebGL API is about setting up state to supply data to our GLSL programs. In this case our only input to our GLSL program is `position` which is an attribute. The first thing we should do is look up the location of the attribute for the program we just created

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

Attributes get their data from buffers so we need to create a buffer

```
var positionBuffer = gl.createBuffer();
```

WebGL lets us manipulate many WebGL resources on global bind points. You can think of bind points as internal global variables inside WebGL. First you set the bind point to your resource. Then, all other functions refer to the resource through the bind point. So, let's bind the position buffer.

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

Now we can put data in that buffer by referencing it through the bind point

```
// three 2d points
var positions = [
  0, 0,
  0, 0.5,
  0.7, 0,
];
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);
```

There's a lot going on here. The first thing is we have `positions` which is a JavaScript array. WebGL on other hand needs strongly typed data so the line `new Float32Array(positions)` creates a new array of 32bit floating point numbers and copies the values from `positions`. `gl.bufferData` then copies that data to the `positionBuffer` on the GPU. It's using the position buffer because we bound it to the `ARRAY_BUFFER` bind point above.

The last argument, `gl.STATIC_DRAW` is a hint to WebGL about how we'll use the data. It can try to use that info to optimize certain things. `gl.STATIC_DRAW` tells WebGL we're not likely to change this data much.

Now that we've put data in the a buffer we need to tell the attribute how to get data out of it. First off we need to turn the attribute on

```
gl.enableVertexAttribArray(positionAttributeLocation);
```

Then we need to specify how to pull the data out

```
var size = 2;          // 2 components per iteration
var type = gl.FLOAT;   // the data is 32bit floats
var normalize = false; // use the data as is
var stride = 0;        // 0 = move size * sizeof(type) each iteration
var offset = 0;        // start at the beginning of the buffer
gl.vertexAttribPointer(
   positionAttributeLocation, size, type, normalize, stride, offset)
```

A hidden part of `gl.vertexAttribPointer` is that it binds the current `ARRAY_BUFFER` to the attribute. In other words now that this attribute is bound to `positionBuffer` we're free to bind something else to the `ARRAY_BUFFER` bind point.

note that from the point of view of our GLSL vertex shader the position attribute was a `vec4`

```
attribute vec4 position;
```

`vec4` is a 4 float value. In JavaScript you could think of it something like `position = {x: 0, y: 0, z: 0, w: 0}`. Above we set `size = 2`. Attributes default to `0, 0, 0, 1` so this attribute will get its first 2 values (x and y) from our buffer. The z, and w will be the default 0 and 1 respectively.

After all that we can finally ask WebGL to execute are GLSL program.

```
var primitiveType = gl.TRIANGLES;
var offset = 0;
var count = 3;
gl.drawArrays(primitiveType, offset, count);
```

This will execute our vertex shader 3 times. The first time `position.x` and `position.y` in our vertex shader will be set to the first 2 values from the positionBuffer. The 2nd time `position.xy` will be set to the 2nd 2 values. The last time it will be set to the last 2 values.

Because we set `primitiveType` to `gl.TRIANGLES`, each time our vertex shader is run 3 times WebGL will draw a triangle based on the 3 values we set `gl_Position` to. No matter what size our canvas is those values are in clip space coordinates that go from -1 to 1 in each direction.

Because our vertex shader is simply copying our positionBuffer values to `gl_Position` the triangle will be drawn at clip space coordinates

```
  0, 0,
  0, 0.5,
  0.7, 0,
```

How those values translate to pixels depends on the `gl.viewport` setting. `gl.viewport` defaults to the initial size of the canvas. Since we didn't set a size for our canvas it's the default size of 300x150. Converting from clip space to pixels (often called screen space in WebGL and OpenGL literature) WebGL is going to draw a triangle at

```
clip space      screen space
  0, 0       ->   150, 75
  0, 0.5     ->   150, 112.5
0.7, 0       ->   255, 75
```

WebGL will now render that triangle. For every pixel it is about to draw it will call our fragment shader. Our fragment shader just sets `gl_FragColor` to `1, 0, 0.5, 1`. Since the Canvas is an 8bit per channel canvas that means WebGL is going to write the values `[255, 0, 127, 255]` into the canvas.

There are 3 major things we still haven't covered from the *remarks*. Textures, varyings, and uniforms. Each of those requires it's own topic.

Read Getting started with webgl online: https://riptutorial.com/webgl/topic/2605/getting-started-with-webgl

Chapter 2: State

Examples

Attributes

Attributes are global state (*). If they were implemented in JavaScript they would look something like this

```
// pseudo code
gl = {
  ARRAY_BUFFER: null,
  vertexArray: {
    attributes: [
      { enable: ?, type: ?, size: ?, normalize: ?, stride: ?, offset: ?, buffer: ? },
      { enable: ?, type: ?, size: ?, normalize: ?, stride: ?, offset: ?, buffer: ? },
      { enable: ?, type: ?, size: ?, normalize: ?, stride: ?, offset: ?, buffer: ? },
      { enable: ?, type: ?, size: ?, normalize: ?, stride: ?, offset: ?, buffer: ? },
      { enable: ?, type: ?, size: ?, normalize: ?, stride: ?, offset: ?, buffer: ? },
      { enable: ?, type: ?, size: ?, normalize: ?, stride: ?, offset: ?, buffer: ? },
      { enable: ?, type: ?, size: ?, normalize: ?, stride: ?, offset: ?, buffer: ? },
      { enable: ?, type: ?, size: ?, normalize: ?, stride: ?, offset: ?, buffer: ? },
    ],
    ELEMENT_ARRAY_BUFFER: null,
  },
}
```

As you can see above there are `8` attributes and they are global state.

When you call `gl.enableVertexAttribArray(location)` or `gl.disableVertexAttribArray` you can think of it like this

```
// pseudo code
gl.enableVertexAttribArray = function(location) {
  gl.vertexArray.attributes[location].enable = true;
};

gl.disableVertexAttribArray = function(location) {
  gl.vertexArray.attributes[location].enable = false;
};
```

In other words `location` directly refers to the index of an attribute.

Similarly `gl.vertexAttribPointer` would be implemented something like this

```
// pseudo code
gl.vertexAttribPointer = function(location, size, type, normalize, stride, offset) {
  var attrib = gl.vertexArray.attributes[location];
  attrib.size = size;
  attrib.type = type;
  attrib.normalize = normalize;
  attrib.stride = stride ? stride : sizeof(type) * size;
```

```
  attrib.offset = offset;
  attrib.buffer = gl.ARRAY_BUFFER;  // !!!! <-----
};
```

Notice that `attrib.buffer` is set to whatever the current `gl.ARRAY_BUFFER` is set to. `gl.ARRAY_BUFFER` is set by calling `gl.bindBuffer(gl.ARRAY_BUFFER, someBuffer)`.

So, next up we have vertex shaders. In vertex shader you declare attributes. Example

```
attribute vec4 position;
attribute vec2 texcoord;
attribute vec3 normal;

...

void main() {
  ...
}
```

When you link a vertex shader with a fragment shader by calling `gl.linkProgram(someProgram)` WebGL (the driver/GPU/browser) decide on their own which index/location to use for each attribute. You have no idea which ones they're going to pick. It's up the the browser/driver/GPU. So, you have to ask it *which attribute did you use for* `position`, `texcoord` *and* `normal`*?*. You do this by calling `gl.getAttribLocation`

```
var positionLoc = gl.getAttribLocation(program, "position");
var texcoordLoc = gl.getAttribLocation(program, "texcoord");
var normalLoc = gl.getAttribLocation(program, "normal");
```

Let's say `positionLoc = 5`. That means when the vertex shader executes (when you call `gl.drawArrays` or `gl.drawElements`) the vertex shader expects you to have setup attribute 5 with the correct `type`, `size`, `offset`, `stride`, `buffer` etc.

Note that **BEFORE** you link the program you can choose the locations by calling `gl.bindAttribLoction(program, location, nameOfAttribute)`. Example:

```
// Tell `gl.linkProgram` to assign `position` to use attribute #7
gl.bindAttribLocation(program, 7, "position");
```

full attribute state

Missing from the description above is that each attribute also has a default value. It is left out above because it is uncommon to use it.

```
attributes: [
   { enable: ?, type: ?, size: ?, normalize: ?, stride: ?, offset: ?, buffer: ?
     value: [0, 0, 0, 1], },
   { enable: ?, type: ?, size: ?, normalize: ?, stride: ?, offset: ?, buffer: ?
     value: [0, 0, 0, 1], },
   ..
```

You can set the value with the various `gl.vertexAttribXXX` functions. The `value` is used when `enable` is `false`. When `enable` is true data for the attribute is pulled from the assigned `buffer`.

Vertex Array Objects

WebGL has an extension, [OES_vertex_array_object]

In the diagram above `OES_vertex_array_object` lets you create and replace the `vertexArray`. In other words

```
var vao = ext.createVertexArrayOES();
```

creates the object you see attached to `gl.vertexArray` in the pseudo code above. Calling `ext.bindVertexArrayOES(vao)` assign your created vertex array object as the current vertex array.

```
// pseudo code
ext.bindVertexArrayOES = function(vao) {
  gl.vertexArray = vao;
}
```

This lets you set all of the attributes and `ELEMENT_ARRAY_BUFFER` in the current VAO so that when you want to draw it's one call to `ext.bindVertexArrayOES` where as without the extension it would be up to one call to both `gl.bindBuffer` `gl.vertexAttribPointer` (and possibly `gl.enableVertexAttribArray`) per attribute.

Uniforms

Uniforms are *per program* state. Every shader program has its own uniform state and its own locations.

Textures

Texture units are global state. If they were implemented in JavaScript they would look something like this

```
// pseudo code
gl = {
  activeTextureUnit: 0,
  textureUnits: [
    { TEXTURE_2D: ?, TEXTURE_CUBE_MAP: ? },
    { TEXTURE_2D: ?, TEXTURE_CUBE_MAP: ? },
    { TEXTURE_2D: ?, TEXTURE_CUBE_MAP: ? },
    { TEXTURE_2D: ?, TEXTURE_CUBE_MAP: ? },
    { TEXTURE_2D: ?, TEXTURE_CUBE_MAP: ? },
    { TEXTURE_2D: ?, TEXTURE_CUBE_MAP: ? },
    { TEXTURE_2D: ?, TEXTURE_CUBE_MAP: ? },
    { TEXTURE_2D: ?, TEXTURE_CUBE_MAP: ? },
    { TEXTURE_2D: ?, TEXTURE_CUBE_MAP: ? },
  ],
};
```

You can choose which unit to index with `gl.activeTexture`.

```
// pseudo code
gl.activeTexture = function(textureUnit) {
  gl.activeTextureUnit = textureUnit - gl.TEXTURE0;
};
```

Calling `gl.bindTexture` binds a texture to the active texture unit like this

```
// pseudo code
gl.bindTexture = function(target, texture) {
  var textureUnit = gl.textureUnits[gl.activeTextureUnit];
  textureUnit[target] = texture;
}
```

When you have a shader program that uses textures you have to tell that shader program which texture units you bound the textures to. For example if you have a shader like this

```
uniform sampler2D diffuse;
uniform sampler2D normalMap;
uniform samplerCube environmentMap;

...
```

For you need to query the uniform locations

```
var diffuseUniformLocation = gl.getUniformLocation(someProgram, "diffuse");
var normalMapUniformLocation = gl.getUniformLocation(someProgram, "normalMap");
var environmmentMapUniformLocation = gl.getUniformLocation(someProgram,
                                                           "environmentMap");
```

Then, after you've made your shader program the *current program*

```
gl.useProgram(someProgram);
```

You then need to tell the shader which texture units you did/will put the textures on. For example

```
var diffuseTextureUnit = 3;
var normalMapTextureUnit = 5;
var environmentMapTextureUnit = 2;

gl.uniform1i(diffuseUniformLocation, diffuseTextureUnit);
gl.uniform1i(normalMapUniformLocation, normalMapTextureUnit);
gl.uniform1i(environmentMapUniformLocation, environmentMapTextureUnit);
```

Now you told the shader which units you did/will use. Which texture units you decide to use is entirely up to you.

To actually bind textures to texture units you'd do something like this

```
gl.activeTexture(gl.TEXTURE0 + diffuseTextureUnit);
gl.bindTexture(gl.TEXTURE_2D, diffuseTexture);
```

```
gl.activeTexture(gl.TEXTURE0 + normalMapTextureUnit);
gl.bindTexture(gl.TEXTURE_2D, normalMapTexture);
gl.activeTexture(gl.TEXTURE0 + environmentMapTextureUnit);
gl.bindTexture(gl.TEXTURE_CUBE_MAP, environmentMapTexture);
```

For very simple WebGL examples that only use 1 texture it's common to never call `gl.activeTexture` since it defaults to texture unit #0. It's also common not to call `gl.uniform1i` because uniforms default to 0 so the shader program, will by default, use texture unit #0 for all textures.

All other texture functions also work off the active texture and texture unit targets. For example `gl.texImage2D` might look something like this

```
gl.texImage2D = function(target, level, internalFormat, width, height,
                         border, format, type, data) {
   var textureUnit = gl.textureUnits[gl.activeTextureUnit];
   var texture = textureUnit[target];

   // Now that we've looked up the texture form the activeTextureUnit and
   // the target we can effect a specific texture
   ...
};
```

Read State online: https://riptutorial.com/webgl/topic/4818/state

Credits

S. No	Chapters	Contributors
1	Getting started with webgl	Community, gman
2	State	gman, Nikola Lukic