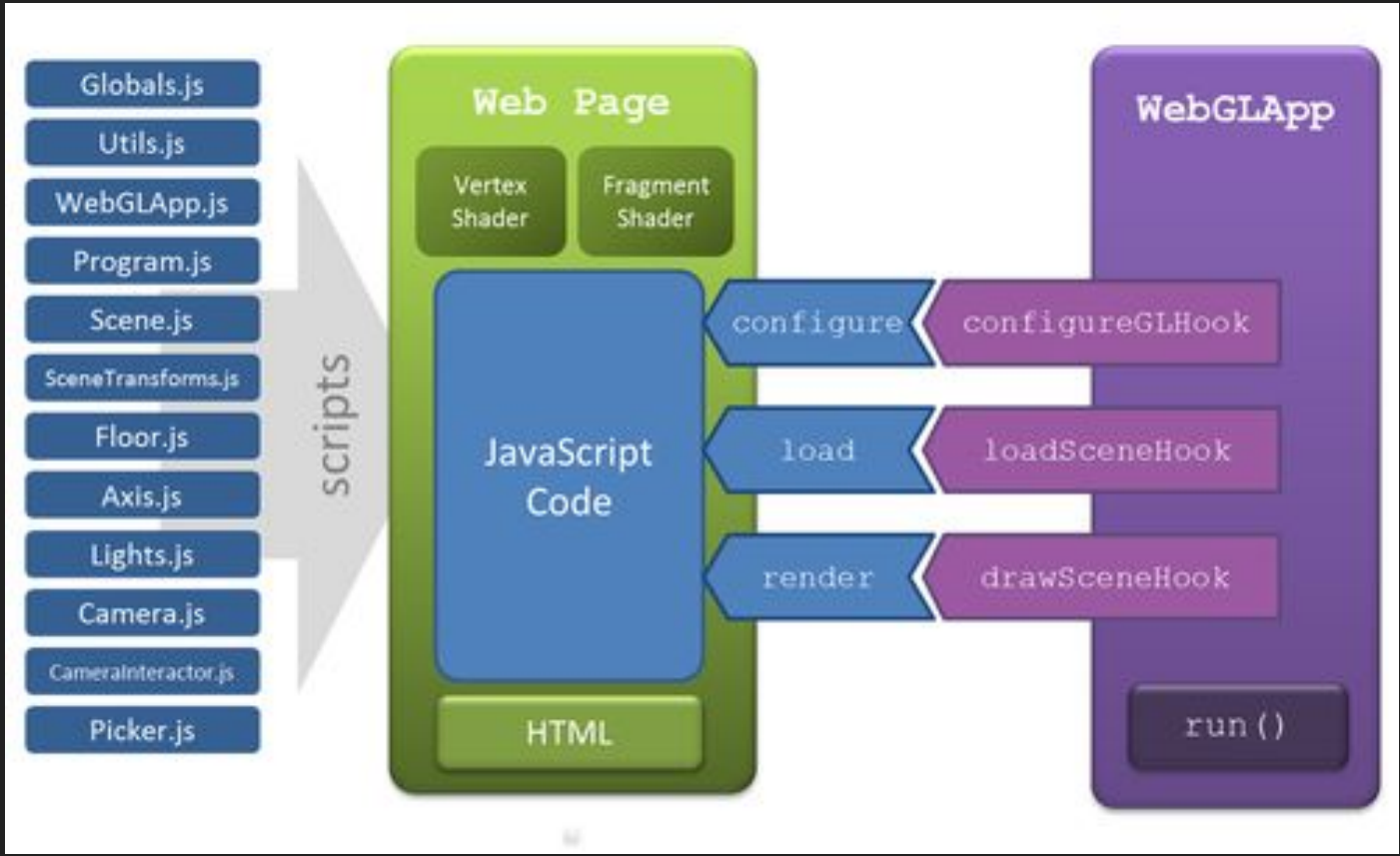
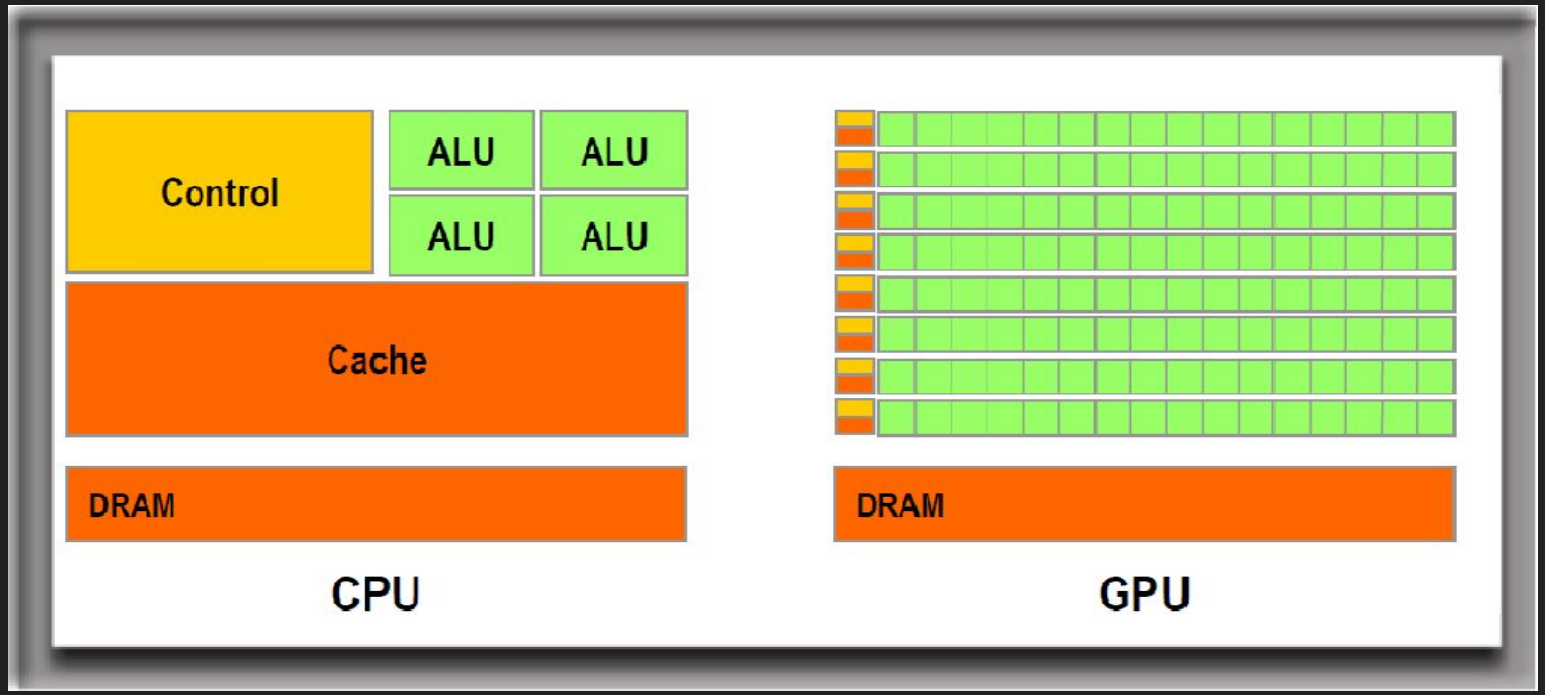
Computer Graphics Fundamentals

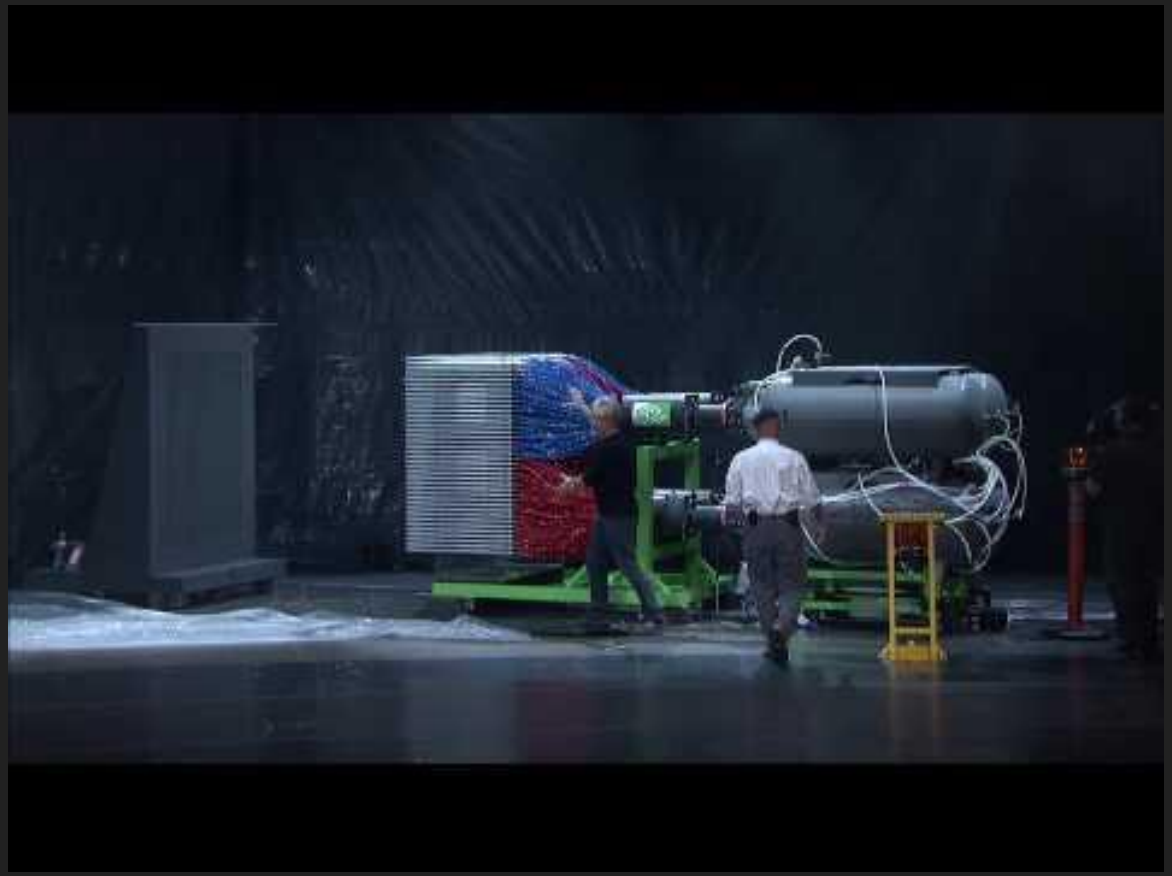
Graphics Programming with WebGL


Globals.js
Utils.js
WebGLApp.js
Program.js
Scene.js
SceneTransforms.js
Floor.js
Axis.js
Lights.js
Camera.js
CameraInteractor.js
Picker.js
scripts
Web Page
Vertex Shader
Fragment Shader
JavaScript Code
HTML
configure
configureGLHook
load
loadSceneHook
render
drawSceneHook
WebGLApp
run()

Rendering happens on the GPU

CPU vs GPU

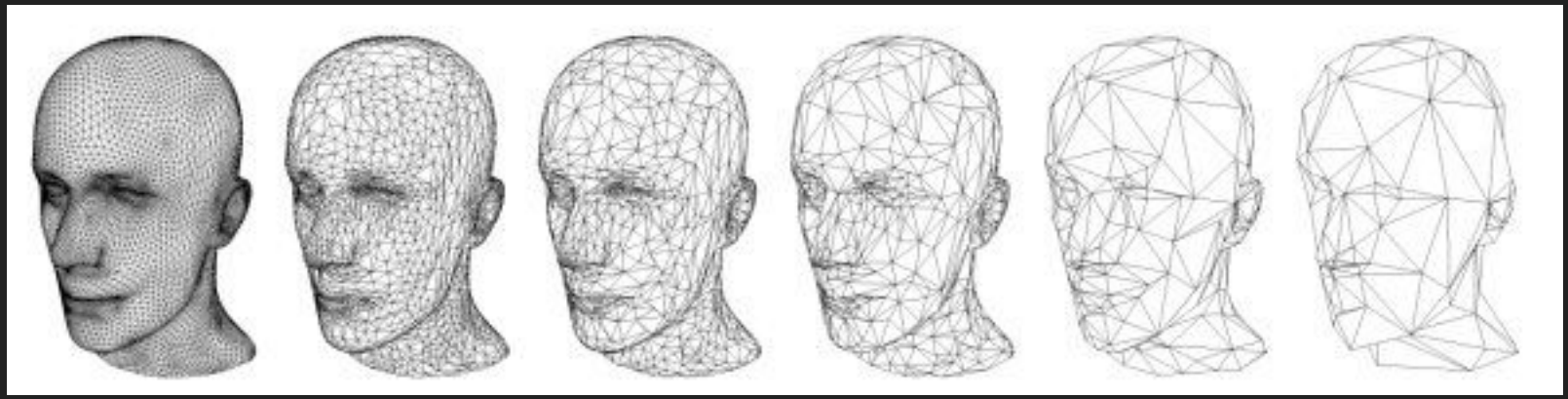




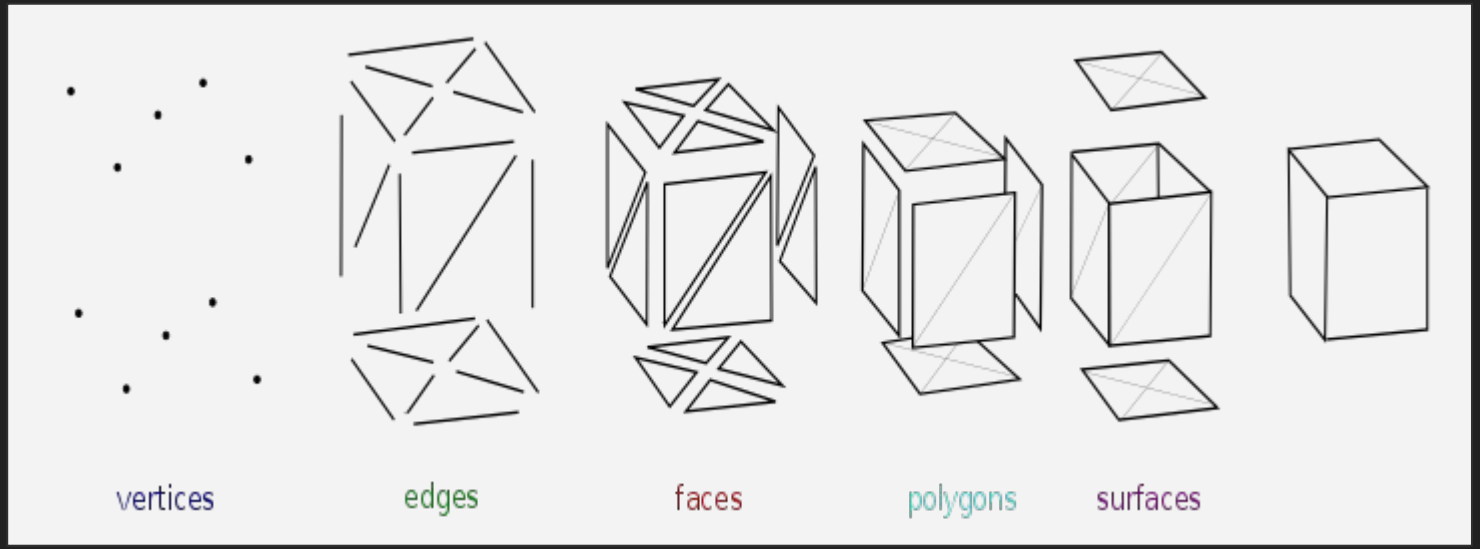
Mythbusters CPU vs GPU Drawing

https://www.youtube.com/watch?v=-P28LKWTzrI

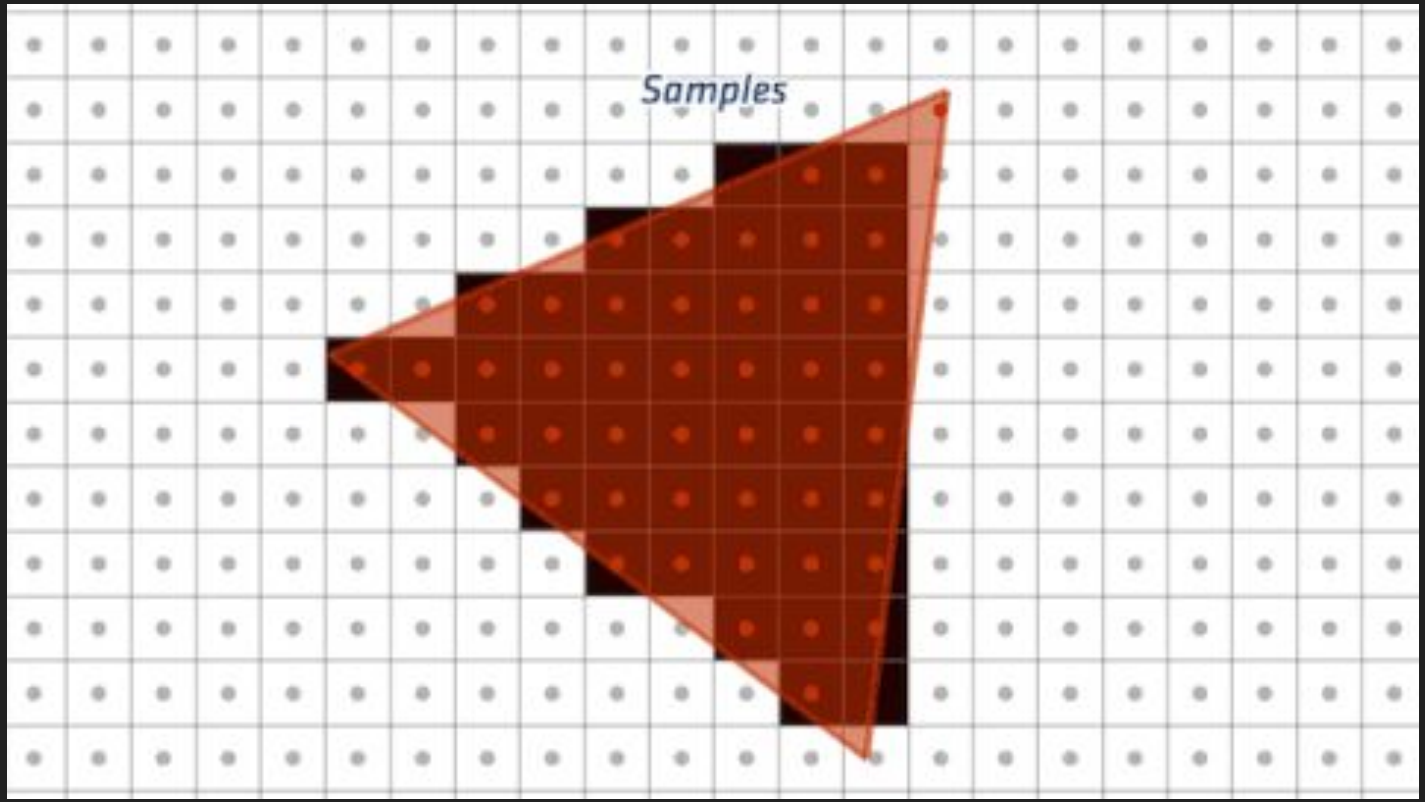
Drawing rasterized graphics:
Abstracting Shapes into triangles (Mesh)



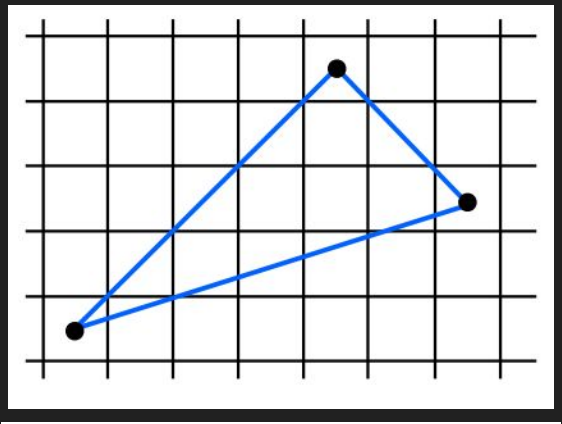
Drawing rasterized graphics
Terminology



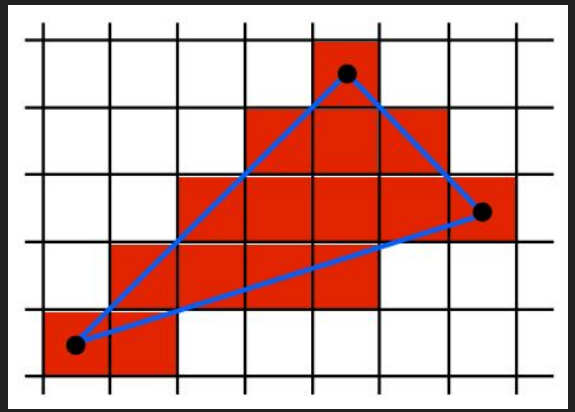
Rasterization



Drawing rasterized graphics: The idea



Triangle data send to the GPU.



Pixels the GPU will color on the output using normal rasterization.

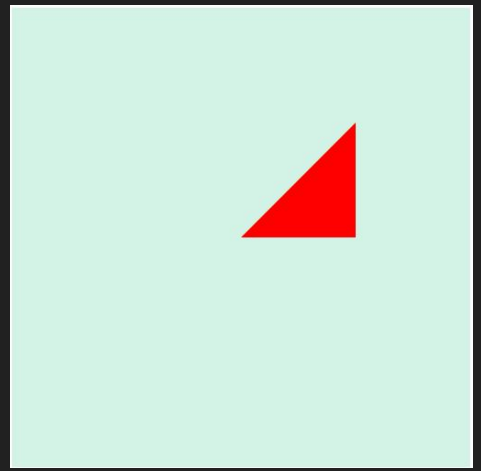
Drawing rasterized graphics: Using the GPU to draw for you

1: Give the GPU a bunch of points (vertex positions).

1.5: Give the GPU the data on how the points are connected.
(By default: just a list, every three points is a triangle.)

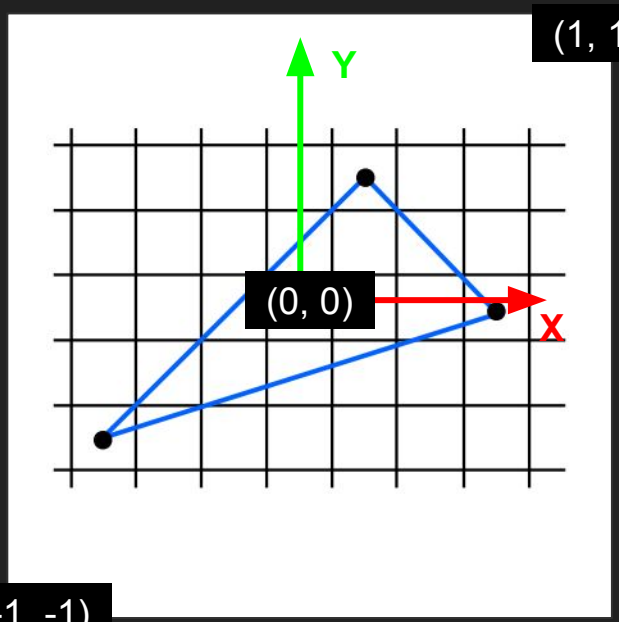
2: Let the GPU draw the triangles (mesh) to an output texture (usually screen).

```
var vertices = [
    0.0, 0.0, // first point
    0.5, 0.0, // second point
    0.5, 0.5, // third point
];
```



Drawing rasterized graphics: The coordinate system

Where did my points go?

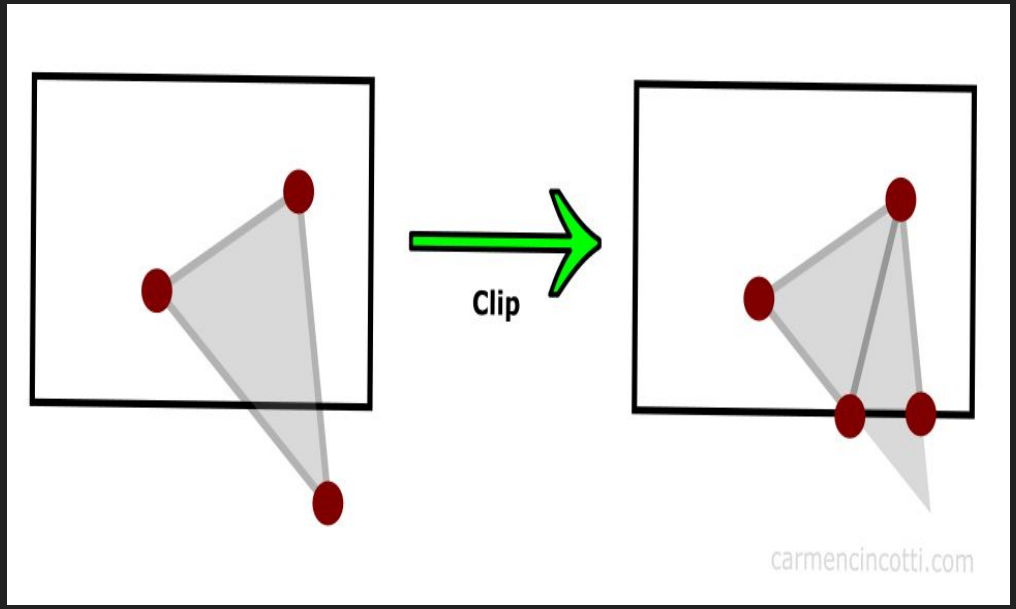


NDC: Normalized Device Coordinates

Standard Cartesian Coordinate System

Origin at the center

Edges stretching to 1 and -1 on the X and Y axis.



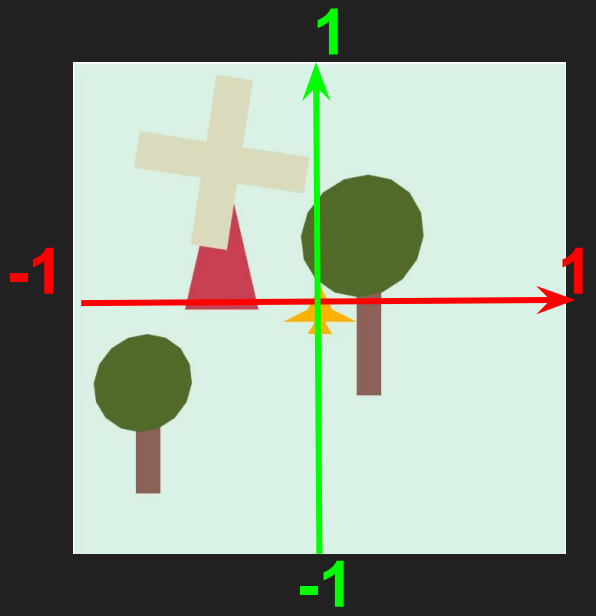
NDC: Normalized Device Coordinates

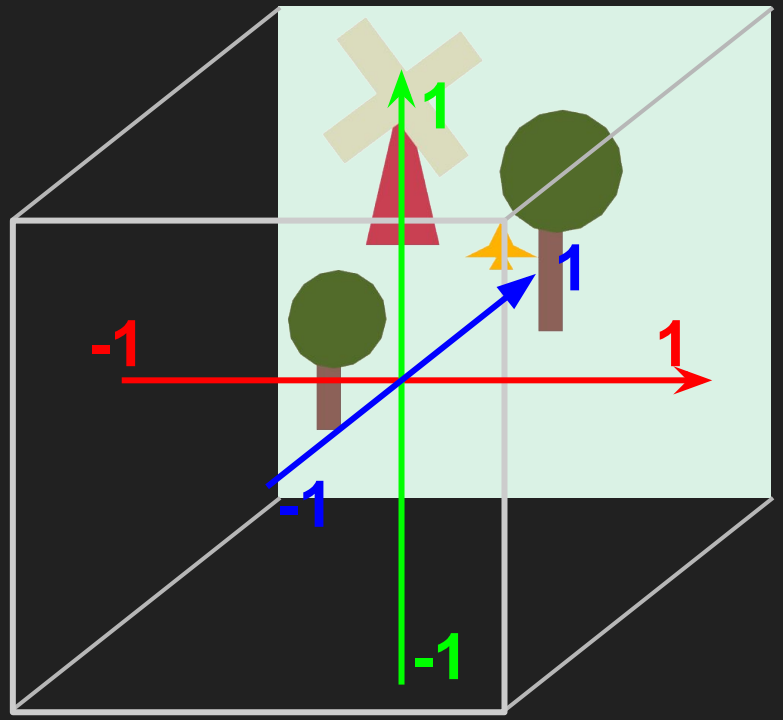
Or: CLIP SPACE

Removing geometry not visible on screen to save processing time

Clipping happens automatically on the GPU

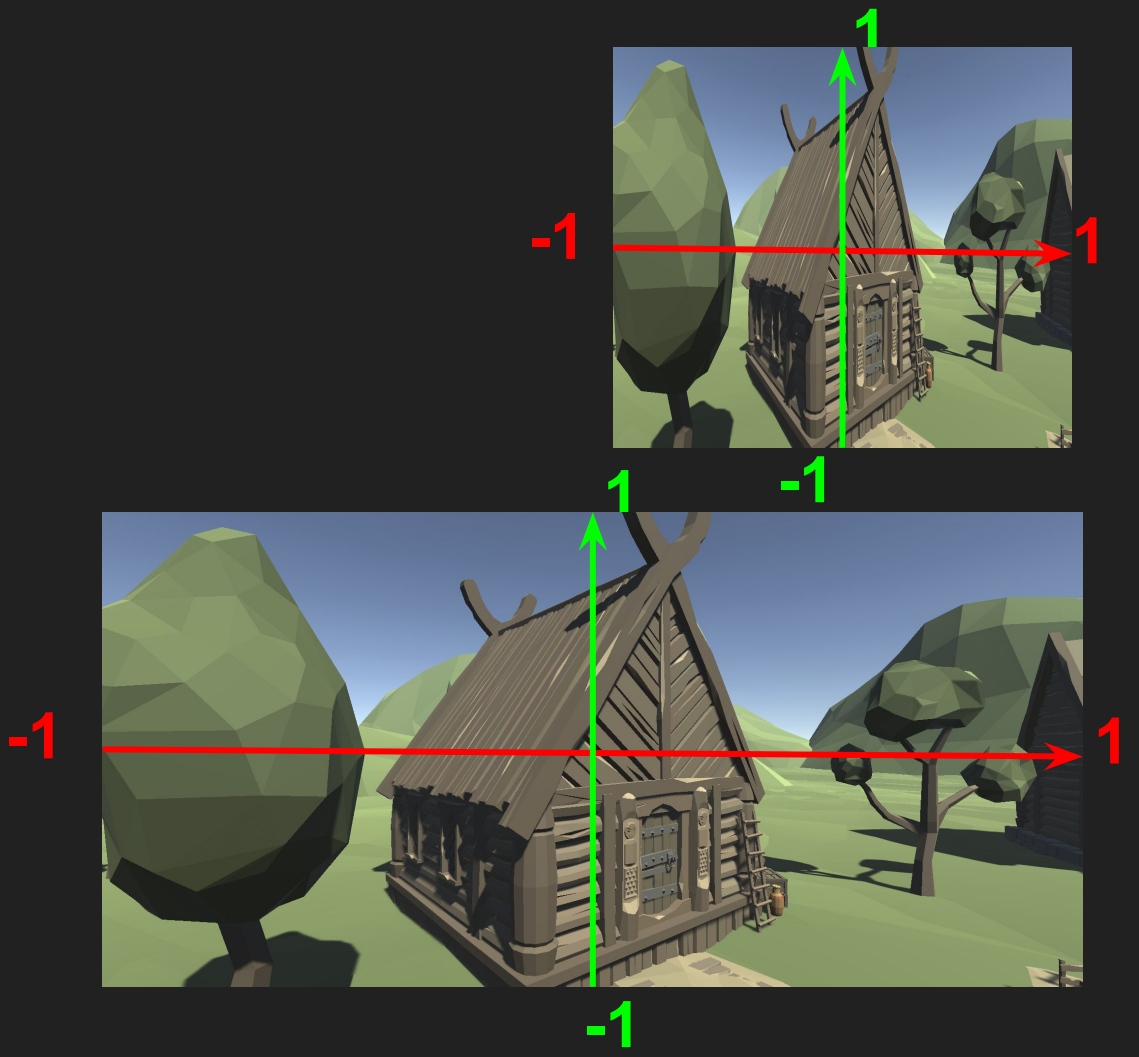
Clip Space is a cube





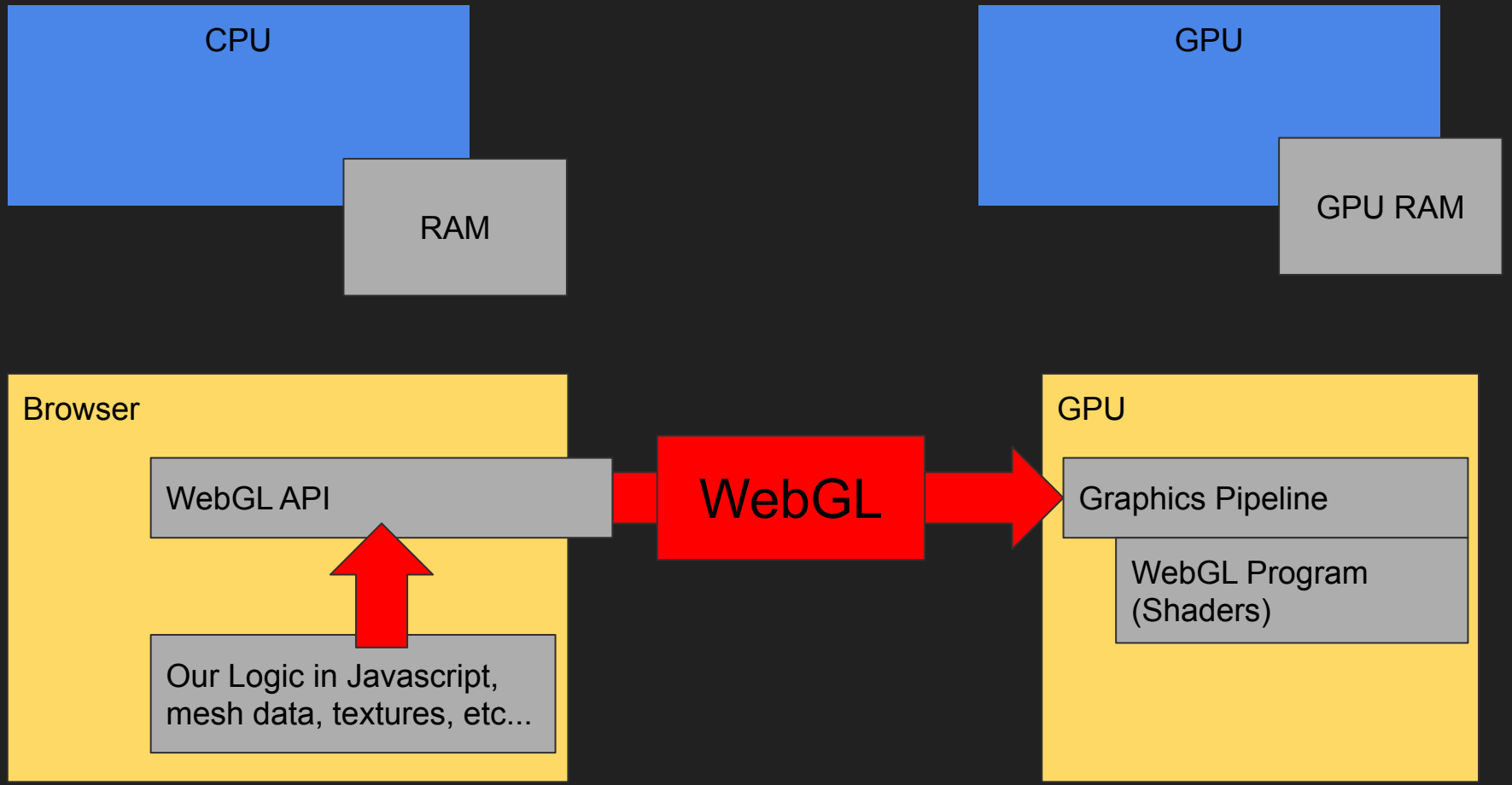
Output Image

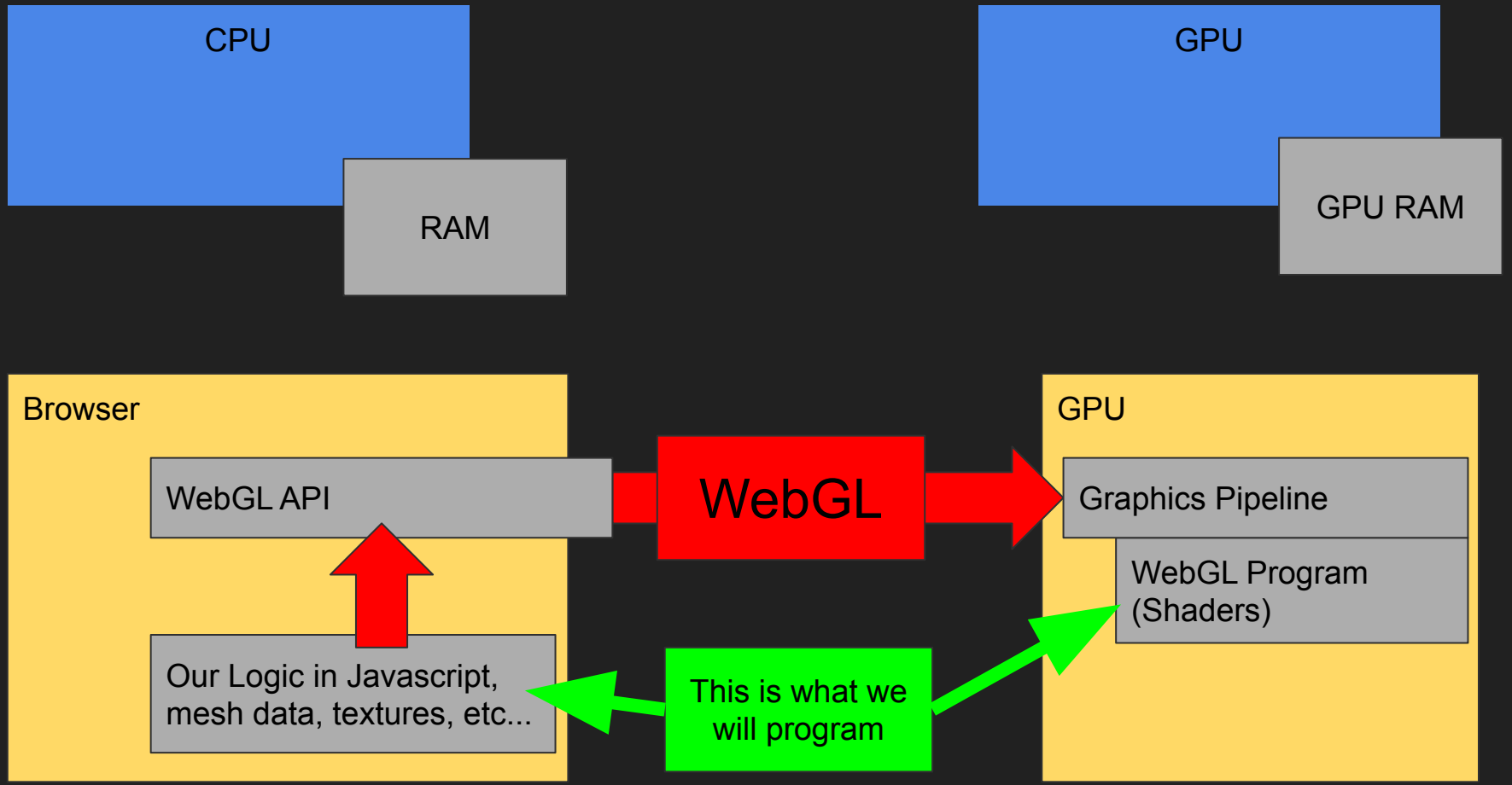
Clip Space is always -1 to 1, but Screen Space does not have to be a square image.



How to transfer data from the CPU to the GPU?

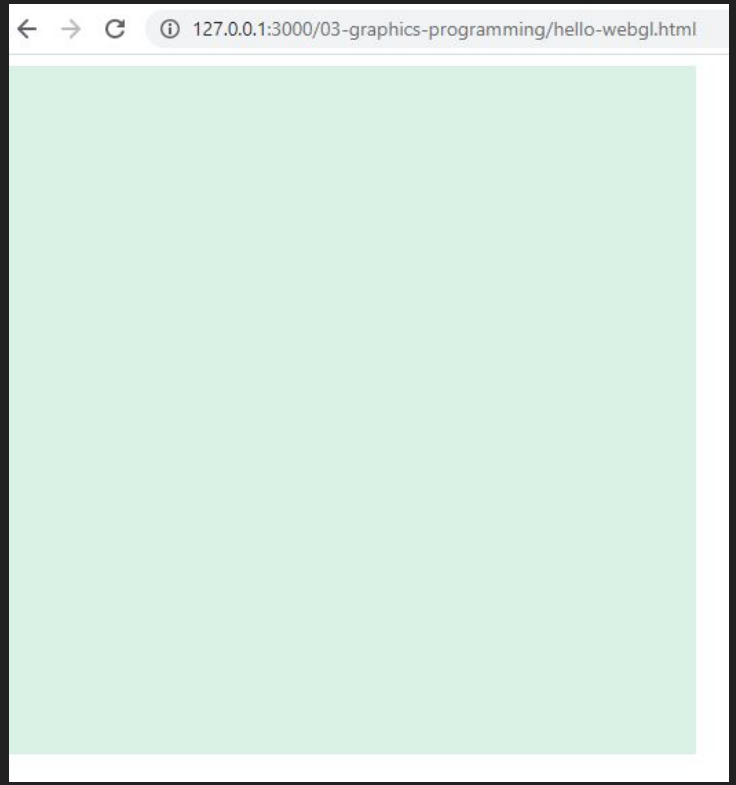
How to transfer data from the CPU to the GPU?


CPU
RAM
GPU
GPU RAM
Browser
WebGL API
Our Logic in Javascript,
mesh data, textures, etc...
WebGL
GPU
Graphics Pipeline
WebGL Program
(Shaders)


CPU
RAM
GPU
GPU RAM
Browser
WebGL API
WebGL
GPU
Graphics Pipeline
WebGL Program
(Shaders)
Our Logic in Javascript,
mesh data, textures, etc...
This is what we
will program

Drawing to the browser: The canvas

```html
<body>
    <canvas width="500" height="500" id="webgl-canvas"></canvas>
    <script type="text/javascript">
        "use strict"; // use strict javascript compiling

        let canvas = document.getElementById("webgl-canvas");
        let gl = canvas.getContext("webgl2"); // WebGLRenderingContext (WebGL 2!)
        // post error if not supported
        if(!gl){ console.error("WebGL context is not available."); }

        gl.clearColor(0.85, 0.95, 0.9, 1); // set clear color (RGBA)
        gl.clear(gl.COLOR_BUFFER_BIT); // clear color buffers
    </script>
</body>
```



```
let canvas = document.getElementById("webgl-canvas");
let gl = canvas.getContext("webgl2"); // WebGLRenderingContext (WebGL 2!)
// post error if not supported
if(!gl){ console.error("WebGL context is not available."); }

gl.clearColor(0.85, 0.95, 0.9, 1); // set clear color (RGBA)
gl.clear(gl.COLOR_BUFFER_BIT); // clear color buffers
```

`<canvas/>` : Canvas element in the DOM. Here WebGL can draw on.

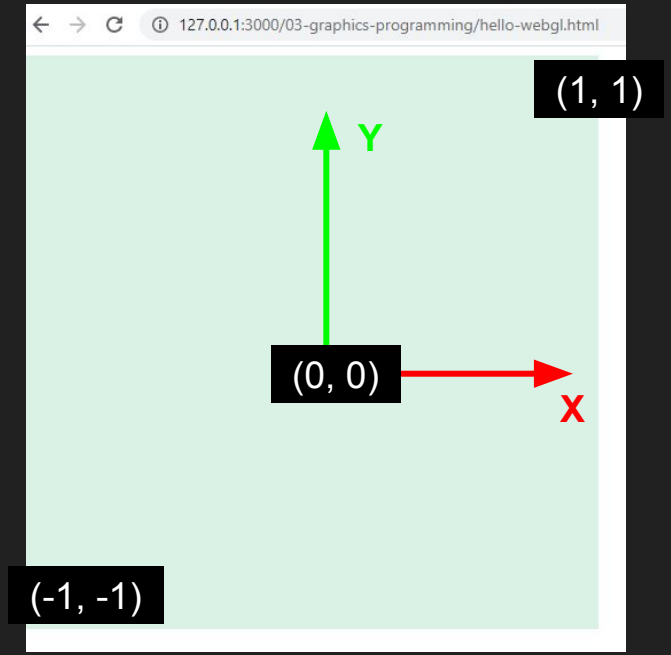
`var canvas` : Reference to the canvas DOM element.

`var gl` : WebGLRenderingContext, handles all basic WebGL

`gl.clearColor()`: Sets the color to clear the canvas.

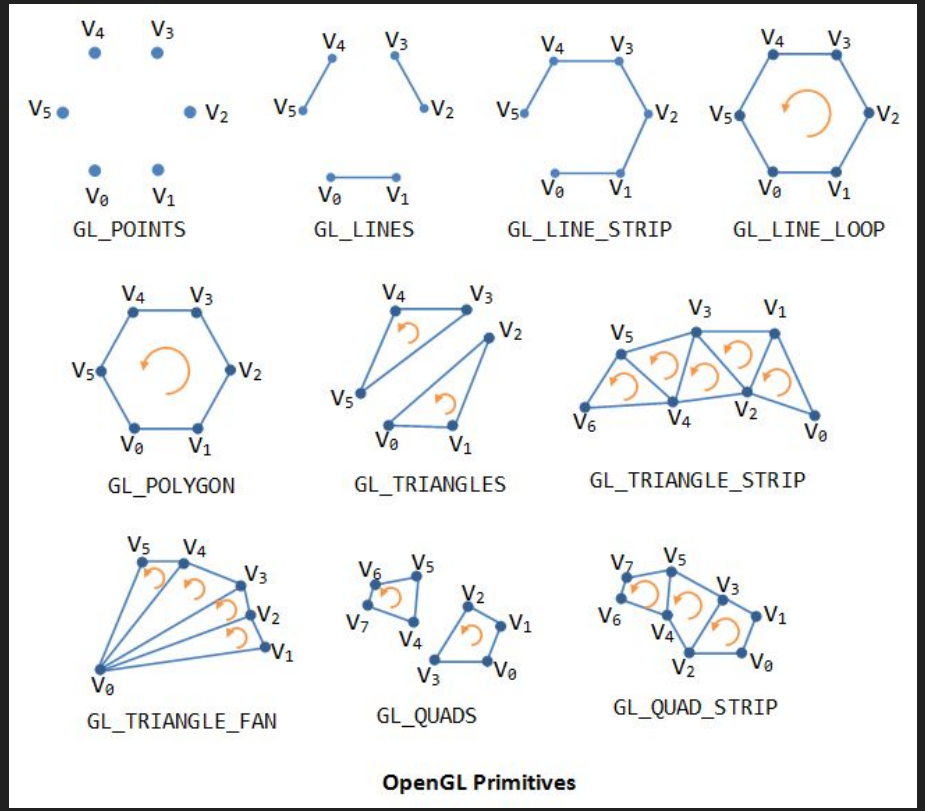
`gl.clear()`: Actually executes the clear action.

NDC on the Canvas



WebGL geometry

When given an array of position data, WebGL can draw it in different ways:

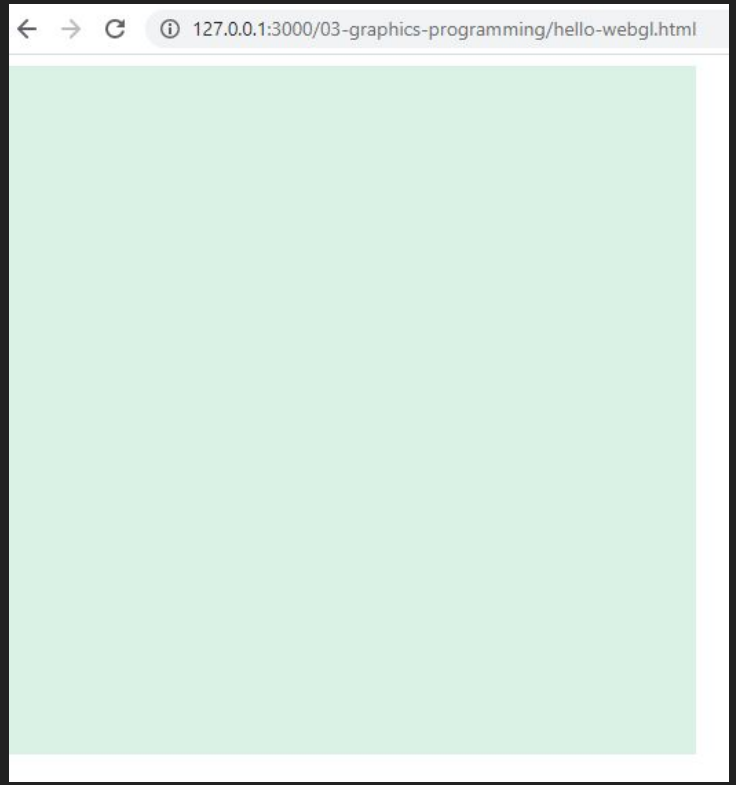


Our first triangle: Task List

1. Create Canvas and get WebGL context.
2. Create array of position data.
3. Create a WebGL-shader-program for the GPU (performs actual drawing)
4. Compile shader-program and push it to the GPU.
5. Push our position data to the GPU.
6. Tell WebGL which shader-program to use for the data pushed.
7. Tell WebGL to now draw using the current pushed data and the current active shader-program.

1. Create Canvas and get WebGL context

```html
<body>
    <canvas width="500" height="500" id="webgl-canvas"></canvas>
    <script type="text/javascript">
        "use strict"; // use strict javascript compiling

        let canvas = document.getElementById("webgl-canvas");
        let gl = canvas.getContext("webgl2"); // WebGLRenderingContext (WebGL 2!)
        // post error if not supported
        if(!gl){ console.error("WebGL context is not available."); }

        gl.clearColor(0.85, 0.95, 0.9, 1); // set clear color (RGBA)
        gl.clear(gl.COLOR_BUFFER_BIT); // clear color buffers
    </script>
</body>
```



1(.5). Debugging help

The Khronos Group: **`WebGLDebugUtils`**

Will be included in the exercises as file “webgl-debug.js”

Include in html file like this: `<script src="webgl-debug.js"></script>`
BEFORE the actual script!

Add this to setup to create debug context: (rest happens automatically)

```
let canvas = document.getElementById("webgl-canvas");
let gl = canvas.getContext("webgl2"); // WebGLRenderingContext (WebGL 2!)
// post error if not supported
if(!gl){ console.error("WebGL context is not available."); }

gl = WebGLDebugUtils.makeDebugContext(gl); // enable debugging
```

2. Create array of position data

```
let positionArray = new Float32Array([
    0.0, 0.0, // first point
    0.5, 0.0, // second point
    0.5, 0.5, // third point
]);
```

WebGL needs the data as a 32 bit floating point value, so we can’t use the normal Javascript array.

We want to draw a 2D triangle, so that's three points of each a X and a Y position.

Our first triangle: Task List

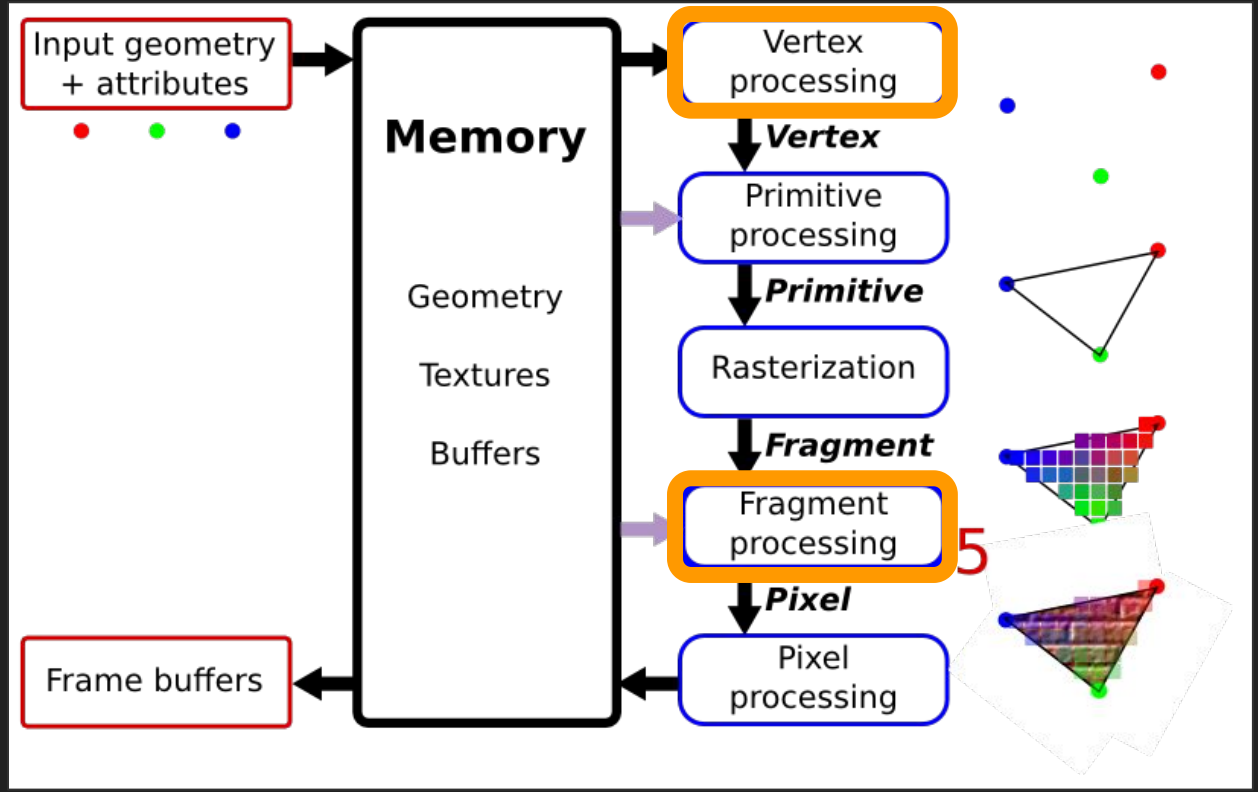
1. Create Canvas and get WebGL context. ✔
2. Create array of position data. ✔
3. Create a WebGL-shader-program for the GPU
4. Compile shader-program and push it to the GPU.
5. Push our position data to the GPU.
6. Tell WebGL which shader-program to use for the data pushed.
7. Tell WebGL to now draw using the current pushed data and the current active shader-program.

3. Create a WebGL-shader-program for the GPU

A quick intro to GPU programming:

The GPU Pipeline:

Programmable steps



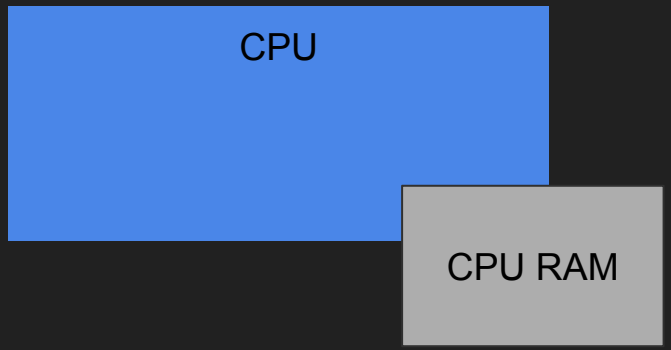
3. a) Vertex Shaders

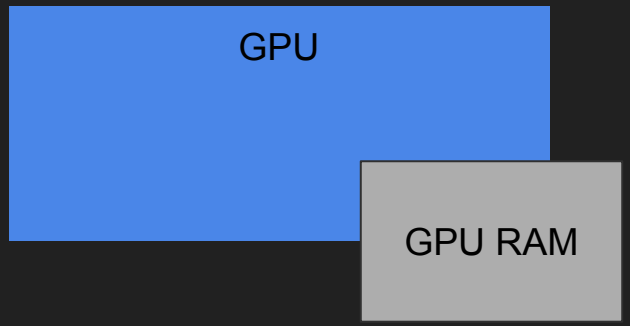
- Included in a script DOM element for now for easy loading.
- Accepts all vertex based attributes (Like position, but there are many others)
- **`attribute`** signifies a variable containing data we pushed per vertex to the GPU

```
<script type="vertex-shader" id="vertexShader">
    attribute vec2 a_position;

    void main() {
        gl_Position = vec4(a_position.x, a_position.y, 0, 1);
    }
</script>
```

- Runs for every vertex.
- **`gl_Position`** is a predefined output variable setting the vertex position.

3. a) Vertex Shaders

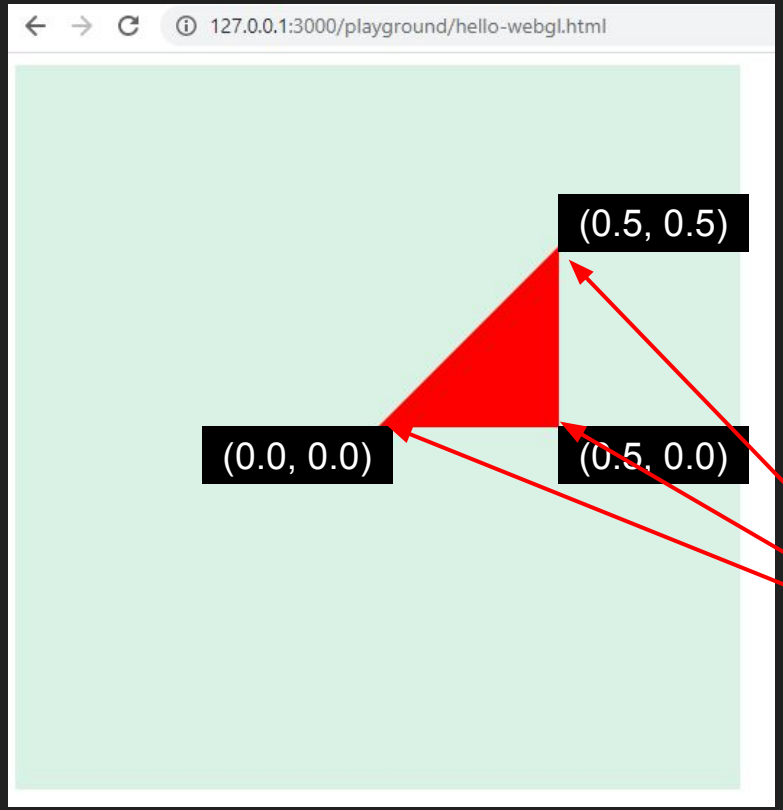





```
let positionArray = new Float32Array([
    0.0, 0.0, // first point
    0.5, 0.0, // second point
    0.5, 0.5, // third point
]);
```

```
<script type="vertex-shader" id="vertexShader">
    attribute vec2 a_position;

    void main() {
        gl_Position = vec4(a_position.x, a_position.y, 0, 1);
    }
</script>
```

3. a) Vertex Shaders




```
let positionArray = new Float32Array([
    0.0, 0.0, // first point
    0.5, 0.0, // second point
    0.5, 0.5, // third point
]);
```

```
<script type="vertex-shader" id="vertexShader">
    attribute vec2 a_position;
    void main() {
        gl_Position = vec4(a_position, 0, 1);
    }
</script>
```

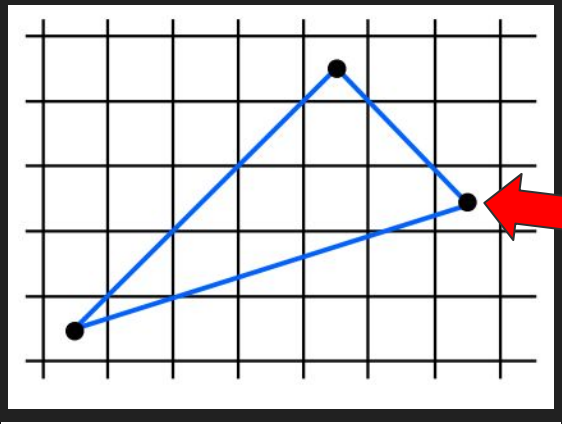
3. b) Fragment Shaders

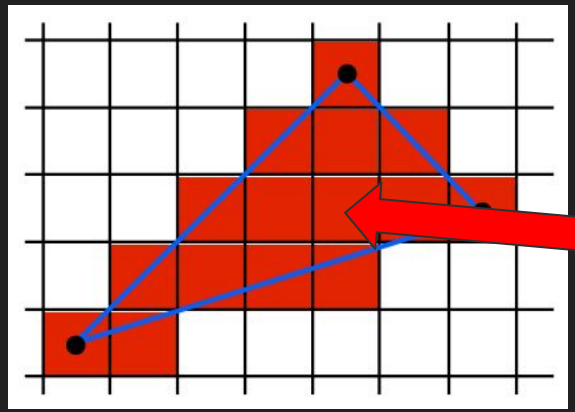
Run for every pixel drawn on the screen covered by a triangle.

Setting the final output color: **`gl_FragColor`**

```
<script type="fragment-shader" id="fragmentShader">
    precision highp float; //float precision settings

    void main()
    {
        gl_FragColor = vec4(1,0,0,1); // rbga
    }
</script>
```

Vertex and Fragment Shader





```
<script type="vertex-shader" id="vertexShader">
    attribute vec2 a_position;

    void main() {
        gl_Position = vec4(a_position.x, a_position.y, 0, 1);
    }
</script>
```

```
<script type="fragment-shader" id="fragmentShader">
    precision highp float; //float precision settings

    void main()
    {
        gl_FragColor = vec4(1,0,0,1); // rbga
    }
</script>
```

Our first triangle: Task List

1. Create Canvas and get WebGL context. ✔
2. Create array of position data. ✔
3. Create a WebGL-shader-program for the GPU ✔
4. Compile shader-program and push it to the GPU.
5. Push our position data to the GPU.
6. Tell WebGL which shader-program to use for the data pushed.
7. Tell WebGL to now draw using the current pushed data and the current active shader-program.

4. Compile shader-program and push it to the GPU

a) Fetch shader source text from DOM.
b) Create and compile Vertex and Fragment shaders.
c) Create WebGL Shader Program (with Vertex and Fragment Shader).
d) Link the Shader Program to the GPU (push it) and validate.

4. a) Fetch shader source text from DOM.

```
// fetch shader program text from DOM
let vertexShaderElement = document.getElementById("vertexShader");
let fragmentShaderElement = document.getElementById("fragmentShader");
if ( !vertexShaderElement ) {
    alert( "Unable to load vertex shader " + vertexShaderId );
}
if ( !fragmentShaderElement ) {
    alert( "Unable to load fragment shader " + fragmentShaderId );
}
```

4. b) Create and compile Vertex and Fragment shaders.

```
// create shaders, set source code and compile them
let vertexShader = gl.createShader(gl.VERTEX_SHADER);
let fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(vertexShader, vertexShaderElement.text);
gl.shaderSource(fragmentShader, fragmentShaderElement.text);
gl.compileShader(vertexShader);
gl.compileShader(fragmentShader);
// check for compiler errors in vertex and fragment shader
if(!gl.getShaderParameter(vertexShader, gl.COMPILE_STATUS)){
    console.error('ERROR could not compile vertex shader.', gl.getShaderInfoLog(vertexShader));
}
if(!gl.getShaderParameter(fragmentShader, gl.COMPILE_STATUS)){
    console.error('ERROR could not compile fragment shader.', gl.getShaderInfoLog(fragmentShader));
}
```

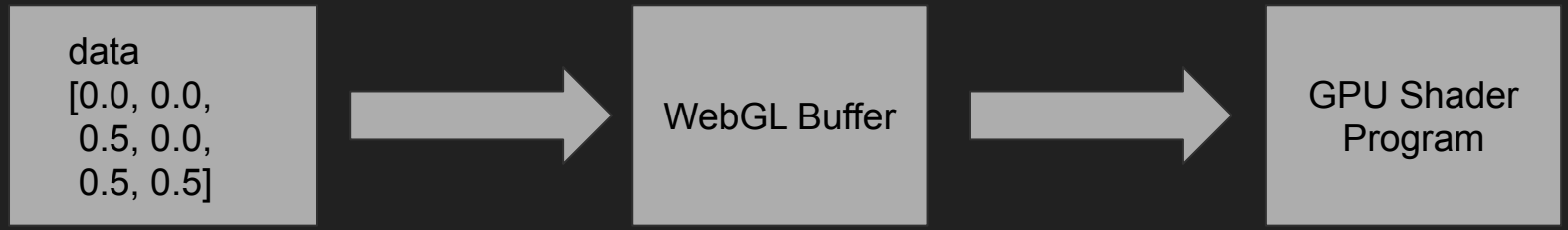
4. c) Create WebGL Shader Program (with Vertex and Fragment Shader).

```
// create shader program and attach vertex and fragment shader
let shaderProgram = gl.createProgram();
gl.attachShader(shaderProgram, vertexShader);
gl.attachShader(shaderProgram, fragmentShader);
// Link Program, completing its preparation and uploading to the GPU
gl.linkProgram(shaderProgram);
if(!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)){
    console.error('ERROR linking program!', gl.getProgramInfoLog(shaderProgram));
}
// Validate that everything worked and the program is now ready to run on the GPU
gl.validateProgram(shaderProgram);
if(!gl.getProgramParameter(shaderProgram, gl.VALIDATE_STATUS)){
    console.error('ERROR validating program!', gl.getProgramInfoLog(shaderProgram));
}
```

Our first triangle: Task List

1. Create Canvas and get WebGL context. ✔
2. Create array of position data. ✔
3. Create a WebGL-shader-program for the GPU ✔
4. Compile shader-program and push it to the GPU. ✔
5. Push our position data to the GPU.
6. Tell WebGL which shader-program to use for the data pushed.
7. Tell WebGL to now draw using the current pushed data and the current active shader-program.

5. Push our position data to the GPU: Buffers



a) Have data in typed Array. (Float32Array).
b) Create a Buffer.
c) Bind the Buffer and push the data into it (upload to GPU).
d) Tell the GPU how the buffer data has to be used (define the Vertex Attribute Layout)

5. a) Have data in typed Array. (Float32Array).

```
let positionArray = new Float32Array([
    0.0, 0.0, // first point
    0.5, 0.0, // second point
    0.5, 0.5, // third point
]);
```

5. b) Create a Buffer.
5. c) Bind the Buffer and push the data (upload to GPU)

```
let positionBuffer = gl.createBuffer();
// set id to the current active array buffer (only one can be active)
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
// upload buffer data
gl.bufferData(gl.ARRAY_BUFFER, positionArray, gl.STATIC_DRAW);
```

How does WebGL know which buffer to use?

```
let positionBuffer = gl.createBuffer();
// set id to the current active array buffer (only one can be active)
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
// upload buffer data
gl.bufferData(gl.ARRAY_BUFFER, positionArray, gl.STATIC_DRAW);
```

`gl.bufferData()` will push data to the buffer LAST bound!

`gl.ARRAY_BUFFER` means a buffer that uses array data.

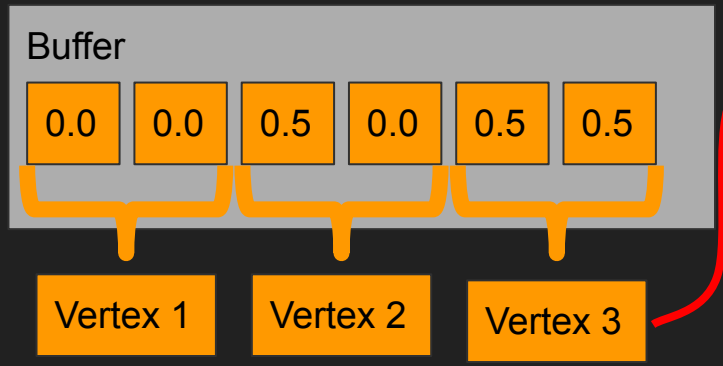
`gl.STATIC_DRAW` means that this buffers data will never be modified.

Buffer data transfer

```
let positionArray = new Float32Array([
    0.0, 0.0, // first point
    0.5, 0.0, // second point
    0.5, 0.5, // third point
]);
```

```
let positionBuffer = gl.createBuffer();
// set id to the current active array buffer (only one can be active)
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
// upload buffer data
gl.bufferData(gl.ARRAY_BUFFER, positionArray, gl.STATIC_DRAW);
```

Buffer data transfer: Attribute stream



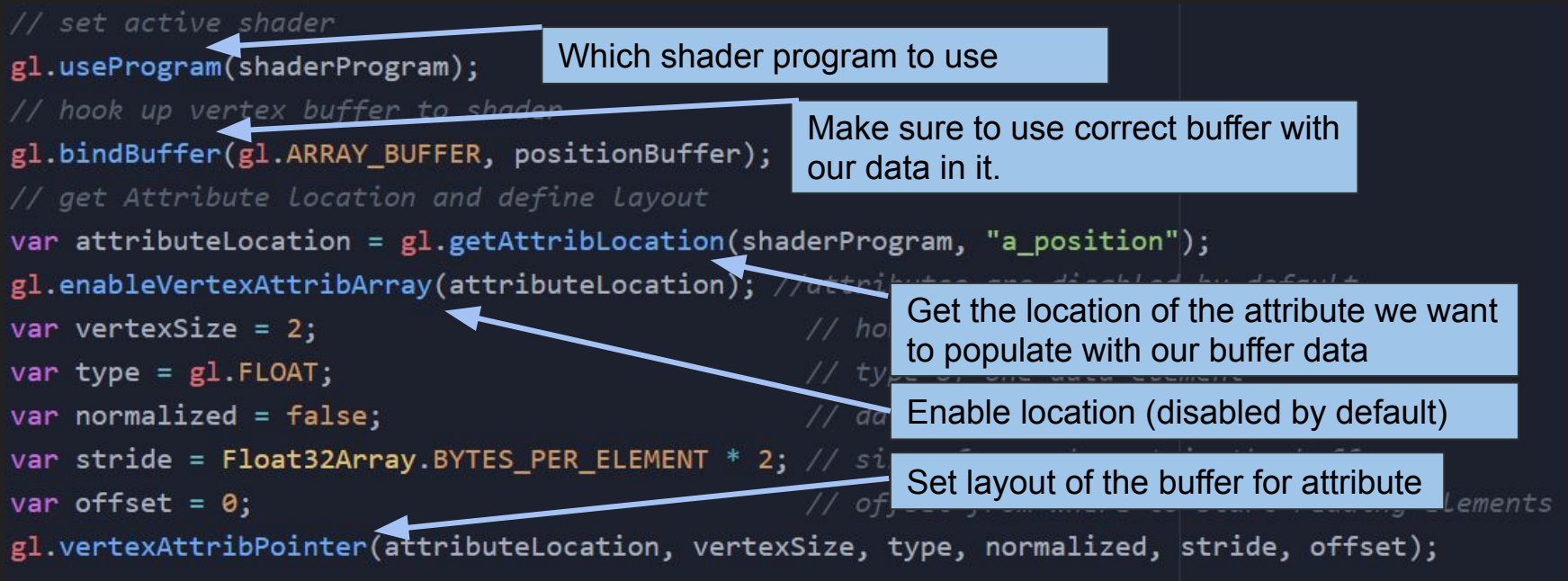

```
<script type="vertex-shader" id="vertexShader">
    attribute vec2 a_position;

    void main() {
        gl_Position = vec4(a_position.x, a_position.y, 0, 1);
    }
</script>
```

Buffer values need to go in chunks of two into `attribute vec2 a_position` of our vertex shader.

5. d) Tell the GPU how the buffer data has to be used (define the Vertex Attribute Pointer)

```
// set active shader
gl.useProgram(shaderProgram);
// hook up vertex buffer to shader
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
// get Attribute location and define layout
var attributeLocation = gl.getAttribLocation(shaderProgram, "a_position");
gl.enableVertexAttribArray(attributeLocation); //attributes are disabled by default
var vertexSize = 2;                            // how many elements per attribute
var type = gl.FLOAT;                           // type of one data element
var normalized = false;                        // data needs to be normalized?
var stride = Float32Array.BYTES_PER_ELEMENT * 2; // size of one element in the buffer
var offset = 0;                                // offset from where to start reading elements
gl.vertexAttribPointer(attributeLocation, vertexSize, type, normalized, stride, offset);
```

5. d) Tell the GPU how the buffer data has to be used (define the Vertex Attribute Pointer)



5. d) The short version

```
// set active shader
gl.useProgram(shaderProgram);
// hook up position buffer to shader
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
// get Attribute location and define Pointer
let positionLocation = gl.getAttribLocation(shaderProgram, "a_position");
gl.enableVertexAttribArray(positionLocation); //attributes are disabled by default
let vertexSize = 2;                           //how many elements per attribute
gl.vertexAttribPointer(positionLocation, vertexSize, gl.FLOAT, false, 0, 0);
```

Which Attribute

How many numbers per Vertex (position:2 [x,y])

What Data Type

Just put `false, 0, 0`

Our first triangle: Task List

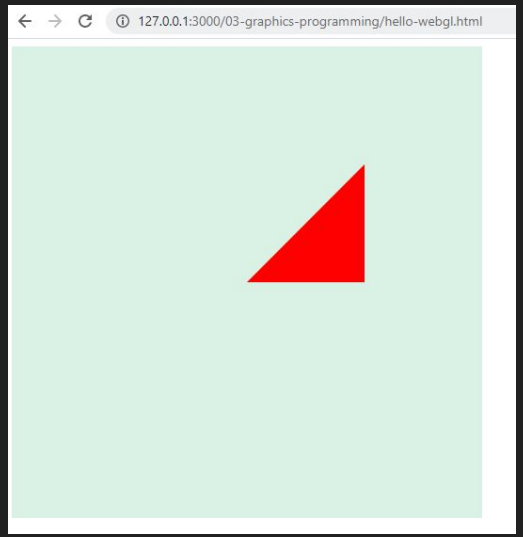
1. Create Canvas and get WebGL context. ✔
2. Create array of position data. ✔
3. Create a WebGL-shader-program for the GPU ✔
4. Compile shader-program and push it to the GPU. ✔
5. Push our position data to the GPU. ✔
6. Tell WebGL which shader-program to use for the data pushed. ✔
7. Tell WebGL to now draw using the current pushed data and the current active shader-program.

7. Tell WebGL to now draw using the current pushed data and the current active shader-program.

```
// draw geometry
let numVertices = positionArray.length/2; // how many vertices to draw
gl.drawArrays(gl.TRIANGLES, 0, numVertices);
```

7. Tell WebGL to now draw using the current pushed data and the current active shader-program.

```
// draw geometry
let numVertices = positionArray.length/2; // how many vertices to draw
gl.drawArrays(gl.TRIANGLES, 0, numVertices);
```



7. Tell WebGL to now draw using the current pushed data and the current active shader-program.

```
// draw geometry
let numVertices = positionArray.length/2; // how many vertices to draw
gl.drawArrays(gl.TRIANGLES, 0, numVertices);
```