Lab 04 - Shader Fundamentals

Ressources

On Buffers, Attributes and Uniforms: Find the file **Shader Fundamentals - Reference** on Canvas.

WebGL uniform setter syntax:
https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/uniform

Main WebGL reference
https://www.khronos.org/webgl/

A good reference for WebGL
https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

Javascript (ES6) documentation
https://developer.mozilla.org/en-US/docs/Web/JavaScript
http://es6-features.org

Handin:

All files of the completed exercise 3 with your own drawing. Should you not be able to complete the exercises all the way up to 3 just upload what you have of exercise 1 and 2 for up to 70% of points.

Guidelines
A ZIP file containing the following folders and files, **exactly structured like this:**

- lab04-groupXX.zip (Where **XX** is the group number on Canvas)
  - lab04-groupXX
    - **All files** from the exercises

***MAC**: To compress to Zip, select a folder in the Finder, right-click and choose "compress". You have to **Zip a folder containing the folder labXX-groupXX** for this to be structured properly:*

- *[AnyFolder] ←- CHOOSE COMPRESS HERE, then name .zip appropriately*
  - *labXX-groupXX*
    - *lab folders and files*

***Windows**: Select the respective folder labXX-groupXX in the Explorer, right-click and choose Send To -> Compressed(zipped) Folder*

Any submission not following these guidelines automatically receives 0 points.

Setup

Download lab04-shader-fundamentals.zip from Canvas, extract and open the folder in Atom.

1. File->Open Folder

Exercise 1: Triangles, Learning how to use Attributes and Uniforms. (20%)

Finding your way around the exercise

You can find all files for this exercise in the folder **lab04-exercise01-triangle**. A lot of the functionality in this exercise has been moved to other .js files. We still have a main .html file that opens and runs our project, but secondary things have moved around.

- `lab04-exercise1.html` This is the main file. It contains:
  - The html document.
  - The Vertex and Fragment shader code (scripts).
  - Import of other .js files.
  - `init()` and `setup()` functions to create the webgl context `gl`, the shader, all global objects, etc…
  - `draw()` functions that execute the Draw Calls.
- `triangle.js` This object file contains:
  - A class that defines a Triangle and its vertices and buffers for us. Usually we make a separate file for each class. We use a class to define the Triangle since we will need multiple Triangles with different properties.

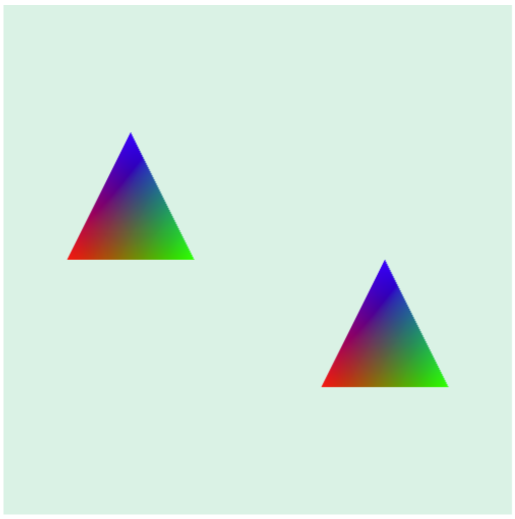
Two other files for this project are located in the `common` folder.
You can find the line `<script src="../common/gl-utils.js"></script>` in `Lab04-exercise1.html`. This means go UP one step (`..`) in the folder hierarchy and then to the `common` folder to find the `gl-utils.js` file.

- `gl-utils.js` This utility file contains:
  - A function to load the shader source code from the html document.
  - Setup of the shader program from the vertex and fragment shader source.
  - In future exercises we add more and more WebGLfunctions here.
- `webgl-debug.js` We always need to import this WebGLdebug file to get decent error output from WebGL.
- `webgl-lint.js` The second webgl debug library.

Goal

Fig1: Our goal for exercise 1 is to render these two triangles. We will practice using attributes by adding the coloring and uniforms by creating an offset from the original vertice positions.

The coloring for these triangles uses a separate color for each vertex. This technique is called **Vertex Colors**

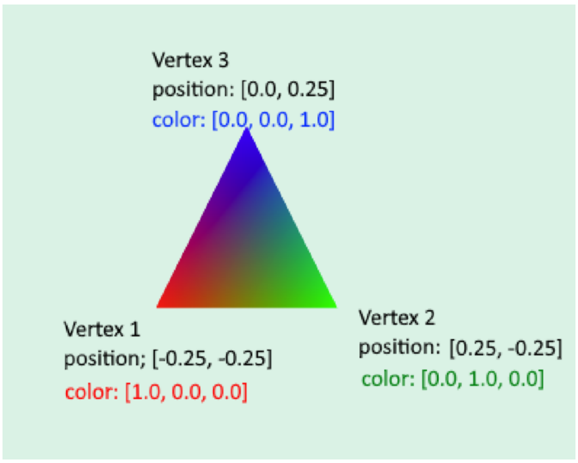


1.1 The first Triangle: Color Attribute, Vertex Colors

Let's start by just rendering one triangle in the middle and adding the color attribute to its vertices.

Fig2: On the right you can see the triangle with its 3 Vertices and their 2 Vertex Attributes: position and color.

Vertex Color: Defining the color of the object by assigning it to each vertex with color attribute is called vertex coloring.



To do this you will need to complete several TODOs in `lab04-exercise1.html` and `triangle.js`. I recommend to do that in this order:

1.1.a First, we just draw a triangle without using the color attribute yet.

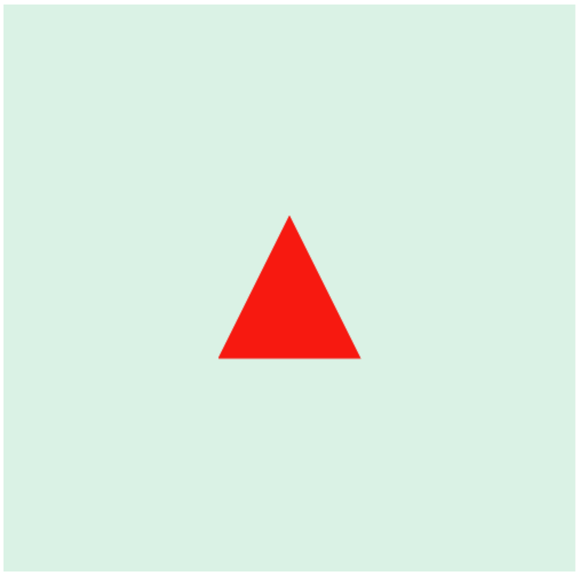
1. The shader in `lab04-exercise1.html` can draw already by just coloring everying red:
   ```
   gl_FragColor = vec4(1,0,0,1); // rbga -> [1,0,0,1] = red
   ```
   To do that we need to complete some TODOs in `triangle.js` and the `draw()` function in `lab04-exercise1.html`.
2. In `triangle.js` you need to create the array with the position data `positionData` and fill it with the correct values shown in Fig2.
3. You then create the `positionBuffer` and fill it with the data from `positionData`. **Tipp:** check your solutions from lab03 if you don't remember how

to do this. **Tipp:** Have a look at the **Appendix** to see how to work inside of a Javascript **class** (`Triangle`) and how to use the `this` keyword.

4. Ignore any TODOs that refer to the `color` attribute for now.
5. In `lab04-exercise1.html.` complete `drawTriangle()`
    a. Here you need to use the shader program, connect the buffer to the attribute in the vertex shader (set pointer) and do the Draw Call.
    b. **Tipp:** check your solutions from lab03 if you don't remember how to do this.

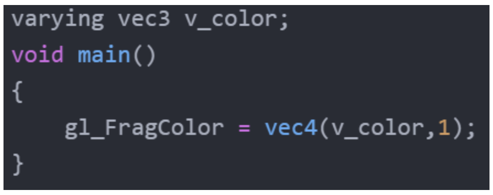
Fig3: You should now see something like this:

If nothing shows up, make sure to check the browser console for errors (right click website and select "inspect") and/or use the debugger of your browser to step through the code.



1.1.b Now we add the color attribute.

1. In `lab04-exercise1.html` add the respective attribute `vec3 a_color` to the vertex shader and the varying `vec3 v_color` to both vertex- (as output) and fragment- (as input) shader. Make sure to set `gl_FragColor` to the new varying (alpha remains 1):



```
varying vec3 v_color;
void main()
{
    gl_FragColor = vec4(v_color,1);
}
```

2. In `triangle.js` now add the properties for the `colorData` (fill with values according to Fig2) and `colorBuffer`. Create the buffer and fill it with the color data. `positionData` should have a total of 6 values in the array (2 values per position and 3 vertices: 2*3=6. `colorData` should have 9 values, since each color has 3 values: r,g,b.
3. In `lab04-exercise1.html` you need to add the attribute pointer for `a_color` and the new `colorBuffer` in `drawTriangle()`.

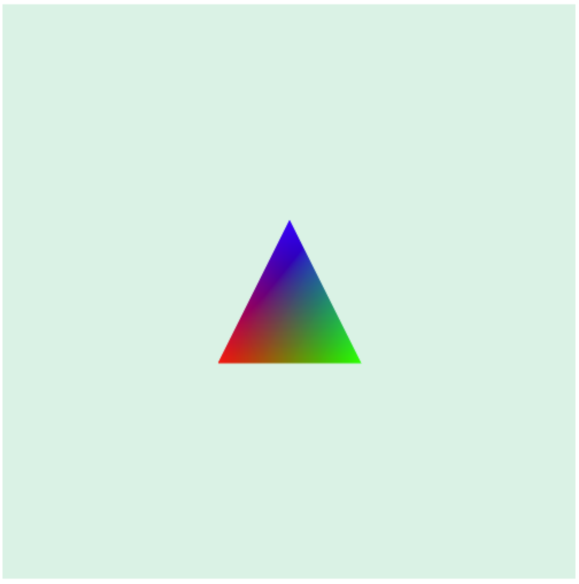


Fig4: With the color attribute used, we now get a nice rainbow triangle. See how the color gets interpolated over the surface of the triangle between the 3 colors that you set:
[1.0,0.0,0.0] : red
[0.0,1.0,0.0] : green
[0.0,0.0,1.0] : blue
This happens because `v_color` is a Varying and Varyings interpolate over the surface of the triangle between the output of each of their connected vertices.

1.2 Working with Uniforms: Triangle offset.

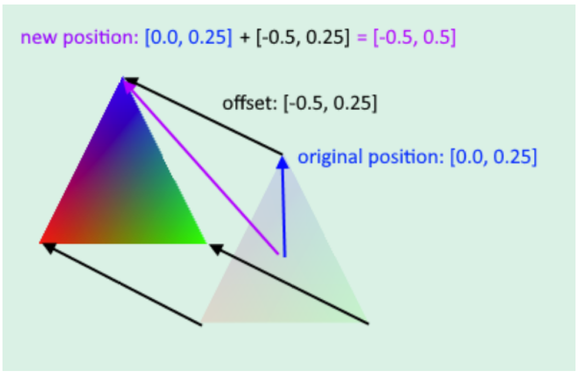
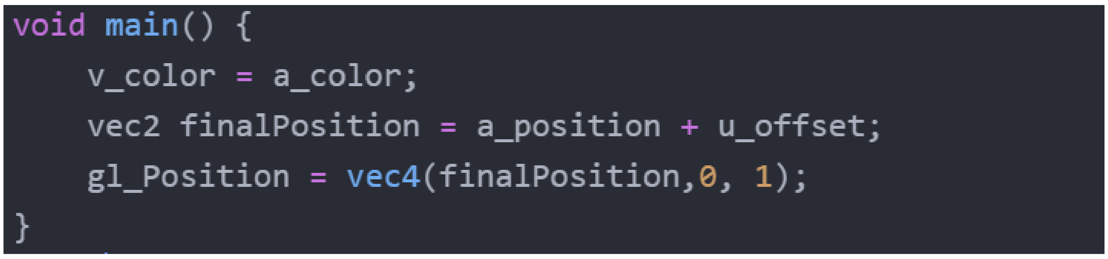

new position: [0.0, 0.25] + [-0.5, 0.25] = [-0.5, 0.5]
offset: [-0.5, 0.25]
original position: [0.0, 0.25]

Fig5: The idea here is to displace each vertex in the vertex shader by a given vector called `u_offset`. Since the vertex shader is running for each vertex, we can displace each vertex position as shown here on the example of Vertex 3. As each vertex is displaced by the exact same vector **offset**, `u_offset` needs to be a **uniform** (the value stays the **same for the entire Draw Call**, unlike an **attribute** such as `a_position`, whose value changes for each vertex).
We just need to add the **uniform** `u_offset` to `a_position` before setting the vertex shader output of `gl_Position`.

Tipp: To learn more about **uniforms** and their use, check out the pdf **Shader Fundamentals - Reference** on Canvas.

1. Add the uniform `vec2 u_offset` to the vertex shader in `lab04-exercise1.html`. Now add a_position and u_offset together before you set gl_Position.



```
void main() {
    v_color = a_color;
    vec2 finalPosition = a_position + u_offset;
    gl_Position = vec4(finalPosition,0, 1);
}
```

2. With the vertex shader updated, we now need to set the uniform data. In `triangle.js` you can see that the Triangles already get `offsetX` and `offsetY` passed as parameters in the constructors. The values are then saved in the properties with the same name.
3. Uniforms are much simpler to handle than attributes, **they do not need a buffer**. So all we have to do is set the wanted values for `u_offset`, before executing the Draw Call in `drawTriangle()`:

```
let offsetLocation = gl.getUniformLocation(shaderProgram, "u_offset");
gl.uniform2fv(offsetLocation, [triangle.offsetX, triangle.offsetY]);
```

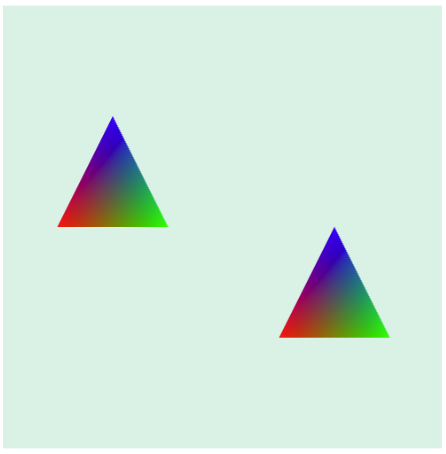
   Note how we use the function `gl.uniform2fv()`, the `2fv` stands for
   - **2: 2** values (u_offset is a vec2).
   - **f**: of type **`gl.FLOAT`** (floating point numbers)
   - **v**: supplied to the function as a **vector** (array) : `[triangle.offsetX, triangle.offsetY]`
   - You could for example also use (without making an array first):

```
gl.uniform2f(offsetLocation, triangle.offsetX, triangle.offsetY);
```

     I prefer to use the vector (array) variant for consistency, since in the future assignments, we have all vector and matrix data stored in arrays.
   - Other functions for other uniform data types can be found here https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/uniform
4. Now all you need to do is add a second triangle with a different offset:

```
g_triangle1 = new Triangle(-0.5,0.25);
g_triangle2 = new Triangle(0.5,-0.25);
```

   And add it to the `draw()` function.



Exercise 2: Quads, Learning how to use the Index Buffer (Element Array Buffer) (20%)

Finding your way around the exercise

The setup is very similar to exercise 1. You find all the files in the folder **lab04-exercise02-quad**. A lot of the functionality in this exercise has been moved to other .js files. We still have a main .html file that opens and runs our project, but secondary things have moved around.

- `lab04-exercise2.html` This is the main file. It contains:
  - The html document.
  - The Vertex and Fragment shader code (scripts).
    - These are different from the triangle shaders! They do not use an attribute `a_color`. But there is a new uniform called `u_tint`.
  - Import of other .js files.
  - `init()` and `setup()` functions to create the webgl context `gl`, the shader, all global objects, etc…
  - `draw()` functions that execute the Draw Calls.
- `quad.js` This object file contains:
  - A class that defines a Quad and its vertices and buffers for us. Usually we make a separate file for each class. We use a class to define the Quad since we will need multiple Quads with different properties.

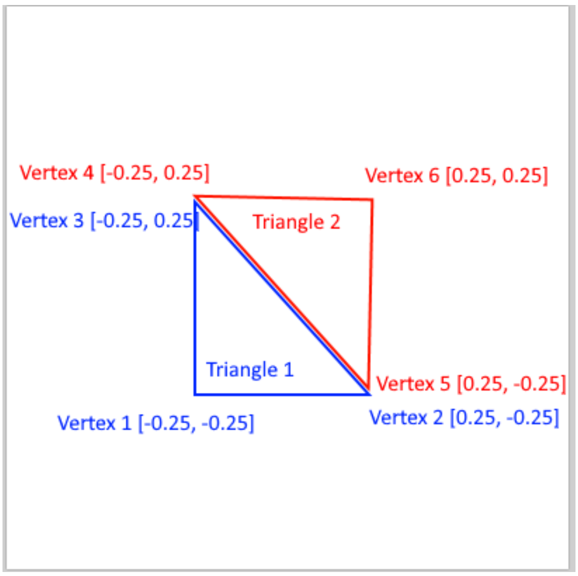
Two other files for this project are located in the `common` folder.
You can find the line `<script src="../common/gl-utils.js"></script>` in `Lab04-exercise1.html`. This means go UP one step (`..`) in the folder hierarchy and then to the `common` folder to find the `gl-utils.js` file.

- `gl-utils.js` This utility file contains:
  - A function to load the shader source code from the html document.
  - Setup of the shader program from the vertex and fragment shader source.
  - In future exercises we add more and more WebGLfunctions here.
- `webgl-debug.js` We always need to import this WebGLdebug file to get decent error output from WebGL.
- `Webgl-lint.js` The other webgl debugging library.

Quads

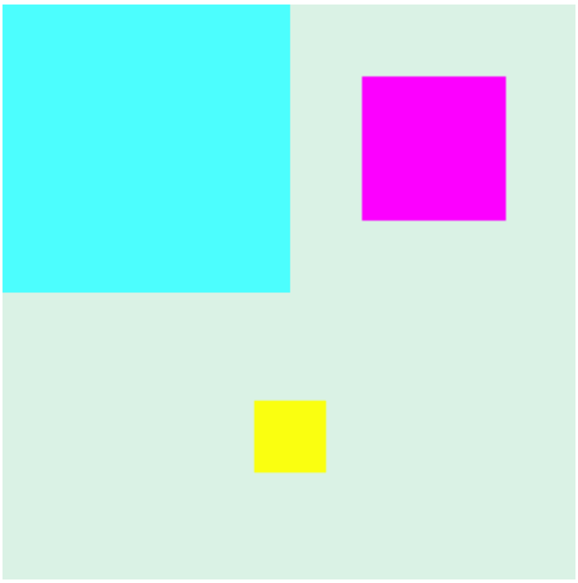
Fig6: Quads are essentially just 2 Triangles connected to each other as shown here. Since we have 2 Triangles within the same object, we need 6 vertices to draw this. That means the position buffer would hold 12 values (6 vertices with a position each, consisting of 2 values [x,y] => 6*2=12.

But you can also see that Vertex 3 and 4 as well as Vertex 2 and 5 are **duplicates**, which is the issue we will tackle in this exercise using the **Index Buffer** (or Element Array Buffer).
Obviously we won't achieve any speed increase by eliminating 2 duplicates, but a Quad is a good and simple object of study to learn the technique.



Goal

Fig7: The goal is to draw 3 Quads in the colors [r,g,b]:
Cyan: [0.0, 1.0, 1.0],
Yellow: [1.0, 1.0, 0.0] and
Magenta: [1.0, 0.0, 1.0]
Arranged something like this (exact placement is not required).



Tint Color: As you can see the quads all have a solid **color**, we call that **tint**. Since **tint** is the same across the whole object, we can make it a uniform in the shader: `u_tint`. `U_tint` needs to be in the fragment shader, since it defines coloring. **Uniforms** can be defined in **both the vertex or fragment shader.**

Scale: They feature different sizes, we will introduce a new uniform `u_scale` to the vertex shader to **scale** them.
Offset: The quads also have an **offset**, same as the triangles, so we need another uniform `u_offset` in the vertex shader.

Furthermore we will make use of the **Element Array Buffer** to eliminate duplicate vertices (not visible).

2.1 Completing the setup

Before dealing with the **Element Array Buffer**, let's finish the setup. Complete all TODOs that **do not** concern the **Element Array Buffer**. I recommend this order:

2.1.a First, complete the quad vertex shader

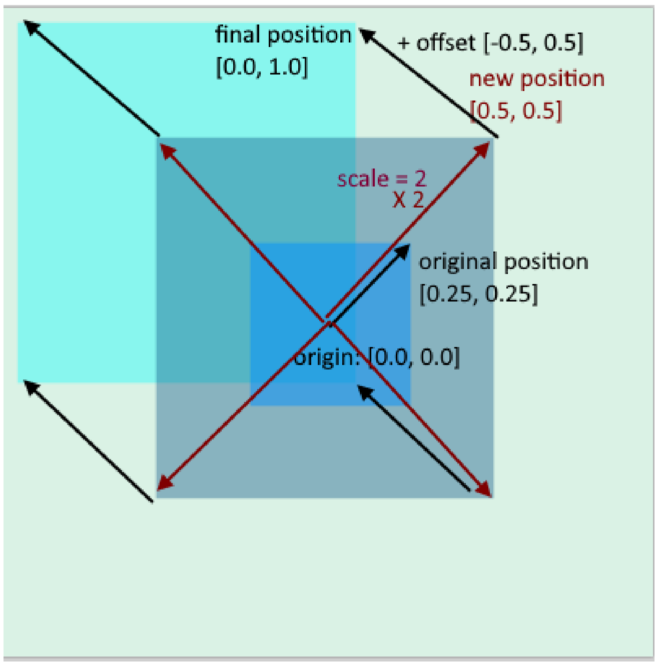
1. Add the **offset** `u_offset` as a uniform and `gl_Position` calculation to the vertex shader same as the Triangles. We do not need an attribute `a_color`, because the quads do not use vertex coloring! But a uniform called u_tint to set the color instead.
2. The quads have different **scales**, so we need another uniform called `u_scale` in the vertex shader. To scale a quad, we can make use of the fact that all vertex positions of the quad are arranged around the center. So multiplying the vertex position by the scale, we can move it further away or close to the center. Since `u_scale` is just a number, it needs to be of type float.

```
attribute vec2 a_position;
uniform vec2 u_offset;
uniform float u_scale;
void main() {
    vec2 finalPosition = a_position * u_scale;
    finalPosition += u_offset;
    gl_Position = vec4(finalPosition, 0, 1);
}
```

2.1.b Scaling objects using scalar multiplication (multiplying vectors and numbers)

Fig8: Here you can see how the scaling operation works using scalar multiplication (multiply a vector with a number). It's important we scale first, to get an equal displacement of all vertices around the origin. After scaling we can apply the offset and move the quad.

Multiplying by a scale smaller than 1 will shrink the quad.



2.1.c Second, complete the quad fragment shader

Instead of the varying v_color, the quads will get their color information as a uniform. Add a uniform vec3 u_color to the fragment shader and use that to set gl_FragColor:

```
gl_FragColor = vec4(u_tint,1);
```

2.1.d Complete Quad class.

The Quad class needs to have all information the quads need stored in properties and create the relevant buffers. For now this means offsetX, offsetY, scale and tint as well as the positionData and positionBuffer.

offsetX, offsetY, scale and tint are all set from the parameters of the constructor, so there is nothing else we need to do here. Note that tint is an array of length 3, since it stores the color information as [r,g,b].

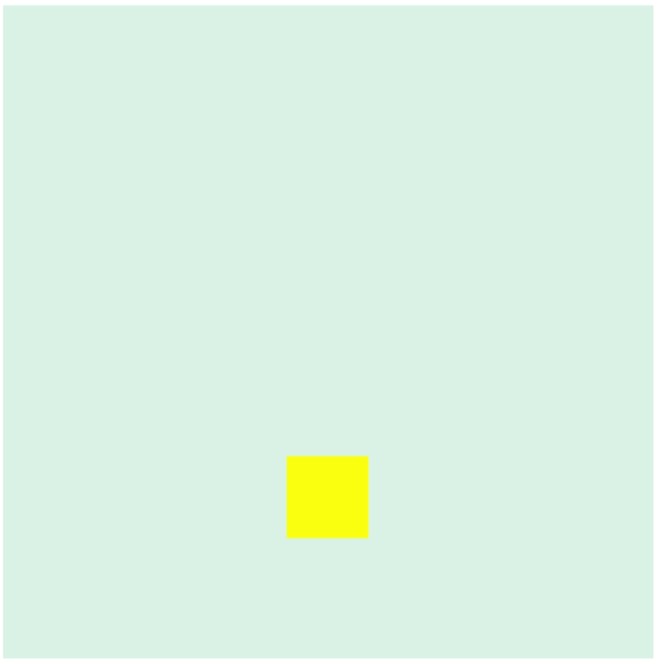
1. Same as for the triangles, enter the position values positionData and create the buffer positionBuffer. You can get the position data from Fig6. Mind that positionData will need to have all 12 position values in it (6 Vertices, with 2 values each), before we make use of the Element Array Buffer. Just enter first all values for Triangle 1, then all values for Triangle 2.
2. See that vertexCount is set to 6, since a Quad consists of 6 vertices.

2.1.e Draw a Quad

To be able to draw a simple quad we need to complete drawQuad() in lab04-exercise2.html.

1. Complete the connection of the quad's positionBuffer to the attribute a_position using gl.vertexAttribPointer() just as you did before with the triangles.
2. Now set all the uniform data. This works the same for all uniforms, no matter if they are in the vertex or fragment shader. Mind the uniform type! You need to use the correct method from gl.uniform[1234][fi][v]() (see https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/uniform) according to the data type of the uniform.
    a. u_offset is a vec2, so you need to use gl.uniform2fv() or gl.uniform2f()
    b. u_scale is a float, so you need to use gl.uniform1f()
    c. u_tint is a vec3, so you need to use gl.uniform3fv() or gl.uniform3f().
3. For now leave the Draw Call at using gl.drawArrays().

Fig9: The result for now.
I recommend that you also look at what happens if you do not set the scale and offset, or set it to different values. Just to try out what your code really does here and get a feeling for it.



2.1.f Draw ALL THE QUADS!

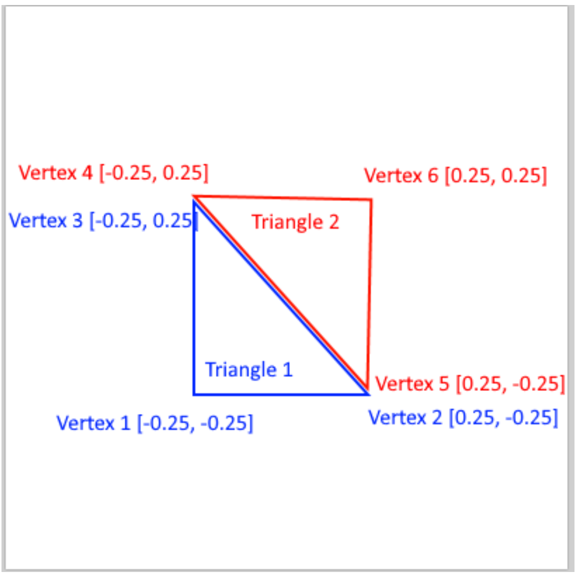
This again works the same as the triangles.

1. Add new global variables for the other 2 quads.
2. Create new quad objects, set their offset, scale and tint.
3. Draw the other quad objects
4. You don't have to exactly reproduce Fig7. But try if you can get close.

1.2 The Element Array Buffer

Now that the setup is completed, we can optimize the Quad drawing using the Element Array Buffer.

We want to eliminate the vertex duplicates 3/4 and 2/5. Since they share the same position attribute data and do not differ in any other attribute (in fact they have none, but what matters is that there are vertices that share the same data in all their attributes).



1. First you need to eliminate the duplicates from the `positionData` array in `quad.js`

```
this.positionData = [
    -0.25,-0.25, // Vertex 1
     0.25,-0.25, // Vertex 2 (previous also Vertex 5)
    -0.25, 0.25, // Vertex 3 (previous also Vertex 4)
     0.25, 0.25 // Vertex 4 (previous Vertex 6)
];
```

2. Next you need to set up the `indexData` property of the `Quad` class. We need to define 6 indices here, one for each Vertex we want to draw.
   a. We still draw 6 vertices! We just eliminate data from the buffers, to make it faster to upload the data to the graphics card.
   b. The 2 triangles have the indices (indices always start at 0!):
      i. Triangle 1: 0, 1 and 2
      ii. Triangle 2: 2, 1 and 3

```
this.indexData = [
  0, 1, 2,   // first triangle
  2, 1, 3   // second triangle
];
```

3.

4. Now you need to create the Element Array Buffer. This is just a regular buffer, except that you have to bind to `gl.ELEMENT_ARRAY_BUFFER`, instead of `gl.ARRAY_BUFFER` and have to make a `UINT16ARRAY` for the data. Indices are always whole positive numbers, or unsigned 16 bit integers (UINT16).

```
// create buffer id for index buffer
this.indexBuffer = gl.createBuffer();
// set id to the current active array buffer (only one can be active)
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, this.indexBuffer);
// upload buffer data
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
                new Uint16Array(this.indexData), gl.STATIC_DRAW);
```

5. Note that you still have to leave `vertexCount` at 6, since we still draw 6 vertices. What matters is the number of indices in the `indexBuffer`, not the amount of unique positions in the `positionBuffer`.
6. Lastly, we need to actually make use of the `indexBuffer` in `drawQuad()` in `lab04-exercise2.html`. Here we need to first
   a. Bind the index buffer to be the currently active `gl.ELEMENT_ARRAY_BUFFER`
   b. Use `gl.drawElements()` in order to draw index based.

```
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, quad.indexBuffer);
// draw geometry
gl.drawElements(gl.TRIANGLES, quad.vertexCount, gl.UNSIGNED_SHORT, 0);
```

7.

You won't notice any difference in the actual drawing. But you are now using the Element Array Buffer!

It does not really make sense to do this with 6 vertices. But we need to learn the technique, since later in the class we will load 3D models with 100.000 or more vertices to draw in the engine and saving on 10.000 or more duplicated data points makes a big difference here.

Exercise 3: Combine and draw something nice (50%)

This exercise has the pure purpose to combine all what you learned above into one big nice drawing with Triangles and Quads.

3.1 Setup

In the folder **lab04-exercise3-combine**, you find `lab04-exercise3.html` that already imports `gl-utils.js` and `webgl-debug.js` but does not do everything else we need for drawing quads and triangles.

1. Copy the files `quad.js` and `triangle.js` from your previous exercise into this folder.
2. Import those files in `lab04-exercise3.html`, maybe see `lab04-exercise1.html` or `lab04-exercise2.html` on how to do that.
3. We need both shaders for quads and triangles, since they use different kinds of attributes and uniforms (they have **different layouts**). We can't use the triangles shader to draw quads and vice-versa.
    a. Copy over the scripts for the vertex and fragment shaders to the marked spot (see TODO html comment) in `lab04-exercise3.html`
4. Add the two global variables for the 2 shader programs `g_triangle_shaderProgram` and `g_quad_shaderProgram`
5. Create both shader programs in `init()`. You can copy the code from `lab04-exercise1.html` and `lab04-exercise2.html` to do that.
6. Now that we have the 2 shader programs setup, we need the respective `drawTriangle()` and `drawQuad()` functions. Copy them over from `lab04-exercise1.html` and `lab04-exercise2.html`
7. You are now ready to go and draw something nice!

3.1 Draw something nice

Here you can be creative, feel free to change the vertex colors of the triangles or add the `u_scale` uniform to their shader if you like.
Note that all triangles need to have the same coloring, since it is set in their constructor! But you can add a new parameter for the `colorData` to the `Triangle` class to make differently colored triangles if you like.

Here is an example of a tree, done by drawing one quad and three triangles and changing the color data of the triangles a bit. You can draw what you want!

Exercise 4: Make a circle class (10% up to 20%)

Using the circle geometry from lab03, create a new `Circle` class to draw circles.

You don't need a new shader program, but can use the `tintcolor` shader program if you want circles that are uniformly colored, or the `vertexcolor` shader program if you like vertex colors for the circle. All you need is make sure to set up the `Circle` class with the appropriate properties, data and buffers. Maybe add a sun or make clouds or bushes out of multiple circles?

For additional extra points (up to 110% in total):
With `requestAnimationFrame(draw)` you can repeatedly call the `draw()` function to create an animation. (more here). `requestAnimationFrame` calls the callback function (`draw`) when the Canvas is ready to draw the next frame, usually about 30 to 60 a second in a modern browser. Calling `requestAnimationFrame(draw)` at the end of `draw()` will ready the next frame once the current one is done. For example: Add a movement animation by changing the `offset` gradually every `draw()` execution.

Appendix

Javascript Classes

```
class Triangle{
    constructor(offsetX, offsetY){
        this.offsetX = offsetX;
        this.offsetY = offsetY;

        this.positionData = [
            -0.25, -0.25,
             0.25, -0.25,
              0.0,  0.25
        ];

        this.positionBuffer = gl.createBuffer();
        gl.bindBuffer(gl.ARRAY_BUFFER, this.positionBuffer);
        gl.bufferData(gl.ARRAY_BUFFER,
                      new Float32Array(this.positionData), gl.DYNAMIC_DRAW);
        this.vertexCount = 3;
    }
}
```

The above code defines a **class** `Triangle` that has the following **properties**:

- `offsetX`: The Triangles offset from the data in the position buffer on the x Axis.
- `offsetY`: The Triangles offset from the data in the position buffer on the y Axis.
- `positionData`: An Array containing the positions [x,z] for each of the Triangles 3 vertices.
- `positionBuffer`: The WebGL Attribute Array Buffer containing the data for the vertex positions.
- `vertexCount`: How many vertices the Triangle has.

The `this` keyword

Let's say we have a `Triangle` object called `myTriangle`:

```
let myTriangle = new Triangle(0.5,0);
```

Inside the class definition (inside the `{ }` of the class definition) the `this` keyword now refers to `myTriangle`, whenever used. For example `myTriangle.offsetX` would be `0.5`, since we gave `0.5` as the first parameter in the `constructor`, which is then assigned to `this.offsetX` in said `constructor`.

Properties

We commonly define the properties of a class by creating in the `constructor`. A **property** is created by simply assigning a value to it in the `constructor` using the `this` keyword:

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

Any value that defines and belongs to an object should be a **property**. Therefore we assign the data and buffer for the Triangles to the `Triangle` class as **properties**.
We can now have multiple Triangles, with different offsets.

Tips on setting the Vertex Attribute Pointer

Please see **Shader Fundamentals - Reference** on Canvas for details on how to set up and work with buffers and attributes. If you got the basics down, here are some tips on how to work faster:

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

The above code is the standard list of calls to connect a buffer to an attribute in the vertex shader when drawing. When drawing, for each buffer you need to:

1. **Bind** the buffer.
2. Get the **location** (name) of the attribute you want to connect to the buffer.
3. **Enable** the Vertex Attribute Array position for this location. (ask me if you want to understand better why we have to do this).
4. Set the **Attribute Pointer** of this attribute.

When setting the Attribute Pointer, there are quite a few of parameters, but we almost always only care about:

1. **What is the name** of the attribute (or its **location** in the shader): eg. `positionLocation`.
2. **How many values** in the buffer we need for the attribute (eg. 2 for a `vec2` such as `a_position`, or 3 for a `vec3` such as `a_color`).

The other parameters `type, normalize, stride` and `offset`, can be set as follows directly:

- `type` always `gl.Float` (almost all attributes are 32bit floating point numbers).
- `normalize` always `false` (we don't want to automatically normalize vectors).
- `stride` always `0` (we always use default stride (size of `type * vertexSize`).
- `offset` always `0` (we always want to start at the beginning of the buffer).

There are of course situations where we would want to change this, but not for now.