Graphics Programming

Ressources

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

A good reference for WebGLRenderingContext (all gl functionality)
https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext

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

Handin:

Only the results of exercise 3 and 4. Please include webgl-debug.js:

1. lab03-exercise03-quad.html
2. lab03-exercise04-circle.html
3. webgl-debug.js

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

- lab03-groupXX.zip (Where **XX** is the group number on Canvas)
    - lab03-groupXX
        - lab03-exercise03-quad.html
        - lab03-exercise04-circle.html
        - lab03-exercise05-objects.html
        - webgl-debug.js
        - webgl-lint.js

***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 lab03-groupXX** for this to be structured properly:*

- *[AnyFolder] ←- CHOOSE COMPRESS HERE, then name .zip appropriately*
    - *lab03-groupXX*
        - *ALL lab folders and files*

***Windows**: Select the respective folder lab03-groupXX.zip in the Explorer, right-click and choose Send To -> Compressed(zipped) Folder*

Any submission not following these guidelines automatically receives 0 points.

Setup

Create a new folder for the lab. Download the exercise files from Canvas (Lab 03) into it.

1: Basic setup from the class

Open “lab03-exercise01-basics.html”. Complete the file with all the missing functionality we covered in class.

Setting up the debugging environment for WebGL.

Since WebGL communicates with the GPU and it is very hard to get information from the GPU about what is wrong over there, we use two libraries that help us get good error information:

1. WebGL debug: The official WebGL debugging tool.
2. WebGL lint: Some guy made this because WebGL debug clearly wasn’t enough. (I literally just found this randomly on GitHub and it's AMAZING!

You also need to download the “webgl-debug.js” and “webgl-lint.js” files and place them in the same folder as your .html files.

Using WebGL debug

With “webgl-debug.js” in he same folder as your html file, two lines are needed (already included in the setup):

```
<script src="webgl-debug.js"></script> // this will include the file
```

and

```
gl = WebGLDebugUtils.makeDebugContext(gl); // this will activate it
```

Using WebGL lint

This is a bit easier, just put “webg-lint.js” in the same folder as your html files and include it with this line:

```
<script src="webgl-lint.js"></script> // this will include the file
```

There is no activation needed for this

This is the most simple webgl program there is. It covers all required steps to draw our very first own triangle. This exercise is extensively commented on in the code, please find all further instructions there.

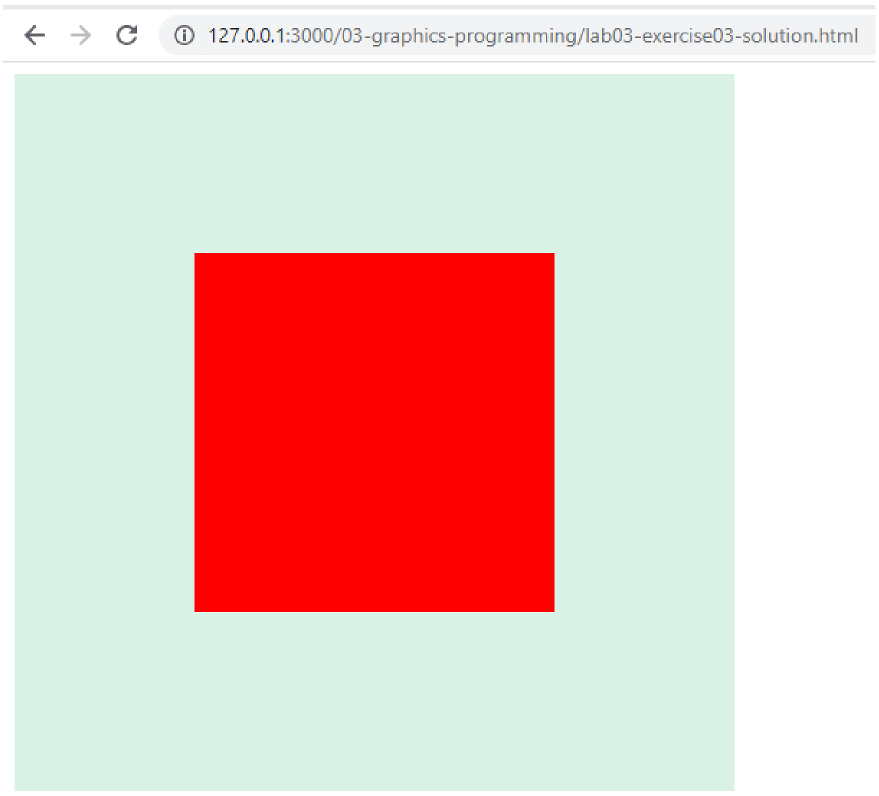
2: Cleanup and refactoring

1. Open “lab03-exercise02-refactoring.html”.
2. The goal for this exercise is to bring the code from 1. into a better more manageable layout. Instead of one giant script, we will split drawing and setup, to eventually be able to keep drawing and updating for a realtime application. Copy or rewrite the code from 1. into the respective functions:
    a. `getDomShaderSrc(elementID)`
    b. `createShaderProgram(vertexShaderScr, fragmentShaderSrc)`
    c. `createArrayBuffer(array)`
    d. `draw()`
3. The rest is already set up for you.

3: The Quad (rectangle), (40%)

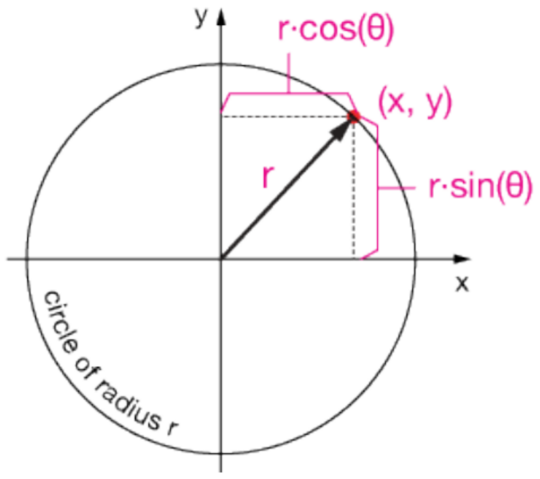
1. Create a copy of your completed "lab03-exercise02-refactoring.html" and name it "lab03-exercise03-quad.html".
2. Modify the contents of the position buffer and the number of drawn vertices to draw a quad consisting of two separate triangles (6 vertices). The quad should be in the center with an edge length of 1 (width and height of 1). Since the NDC (Normalized Device Coordinates, or Clip Space) spans -1 to 1 on X- and Y-axis, the total width of the visible space is 2. So your quad with edge length 1 will look like this:

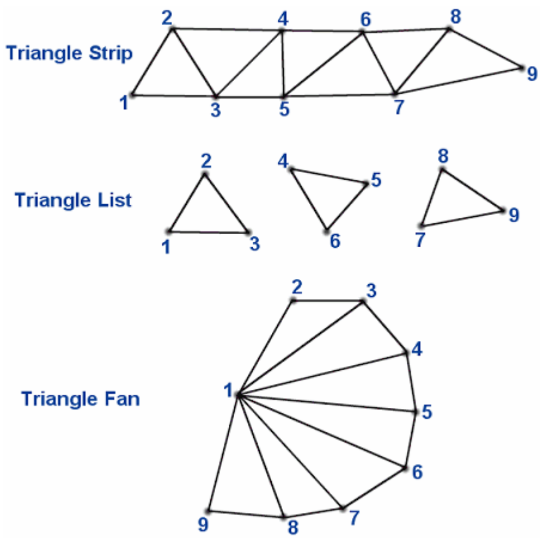
Example output:


127.0.0.1:3000/03-graphics-programming/lab03-exercise03-solution.html

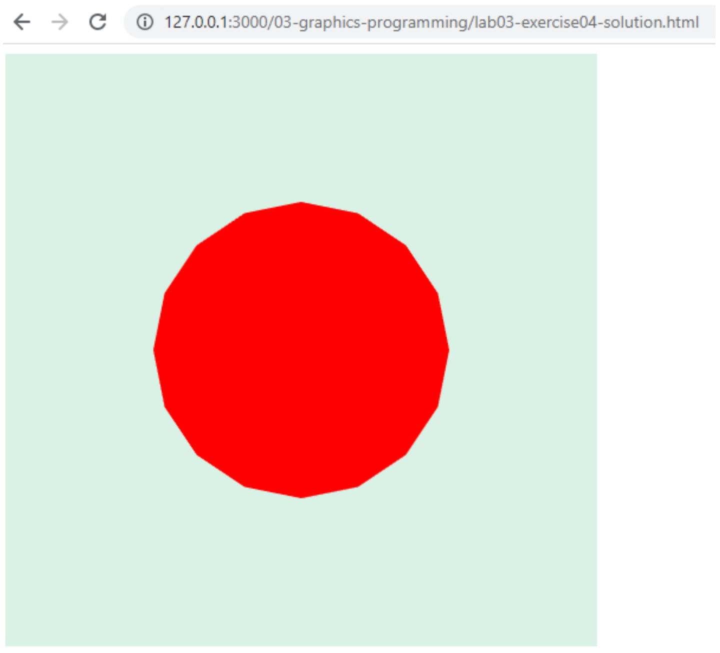
4. Drawing a circle (50%)

1. Create a copy of your completed "lab03-exercise03-quad.html" and name it "lab03-exercise04-circle.html".
2. Modify the position data to draw a circle instead of a quad.
   a. The circle should be in the center with a radius of 0.5.
   b. Use `gl.TRIANGLE_FAN` instead of `gl.TRIANGLES` when drawing with `gl.drawArrays()`, to save on the number of vertices required.
   c. Calculate the vertex x and y position data using trigonometry. `Math.sin(angle)` and/or `Math.cos(angle)` and `Math.PI` will help you (all in Radians!). Keep in mind how a `gl.TRIANGLE_FAN` assembles triangles from the vertex data in your `positionBuffer`.





Example output:


127.0.0.1:3000/03-graphics-programming/lab03-exercise04-solution.html

5. Multiple objects (10% or more)

Create a copy of your completed "lab03-exercise03-quad.html" or "lab03-exercise04-circle.html" and name it "lab03-exercise05-objects.html".

See if you can modify the code so that a circle and a cube (or even more different shapes) will be drawn together using multiple draw calls (multiple calls of `gl.drawArrays()`). Make sure each of the objects is visible and not hidden behind others by modifying the vertex data accordingly.

You can use the same WebGL shader program for all draw calls, but you will have to create separate attribute buffers (vertex data buffers).

You can either just duplicate everything, or better: Try to organize your code, maybe create new functions and avoid writing too much duplicated code.