GPU Computing Using WebGL -Day 1

Tuesday, March 19, 2019 8:09 AM

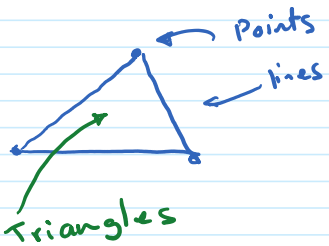
Go to the following link

http://www.chaos.gatech.edu/ccis2019/sc1/

Objectives:

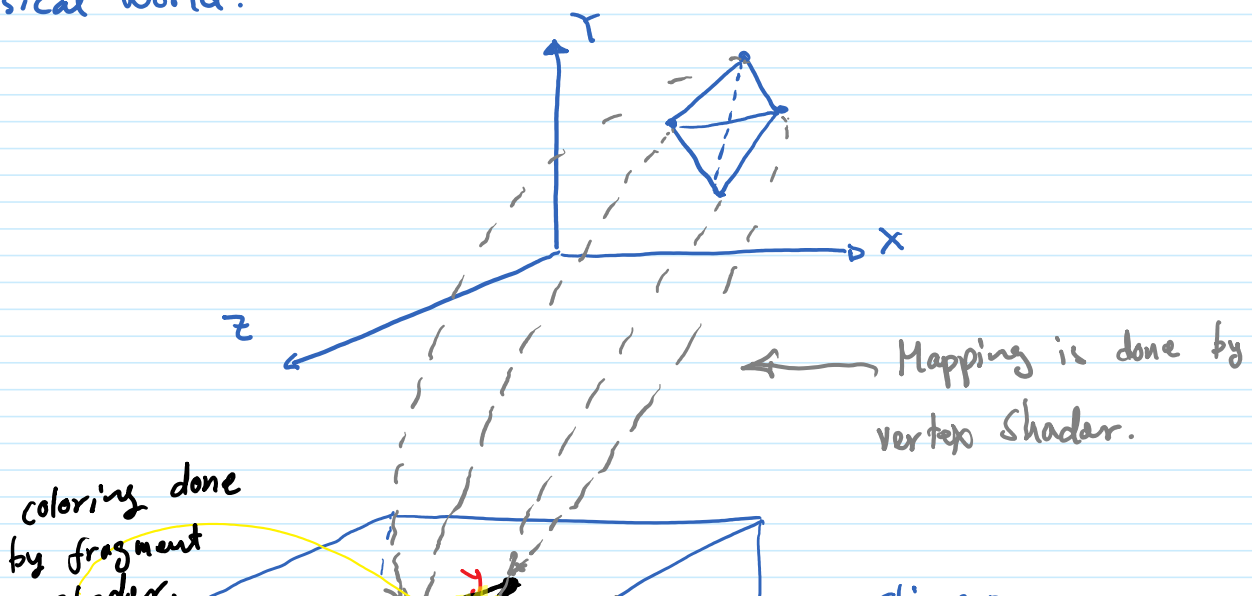
* Intro to graphical pipeline
* Intro to vertex & fragment shaders

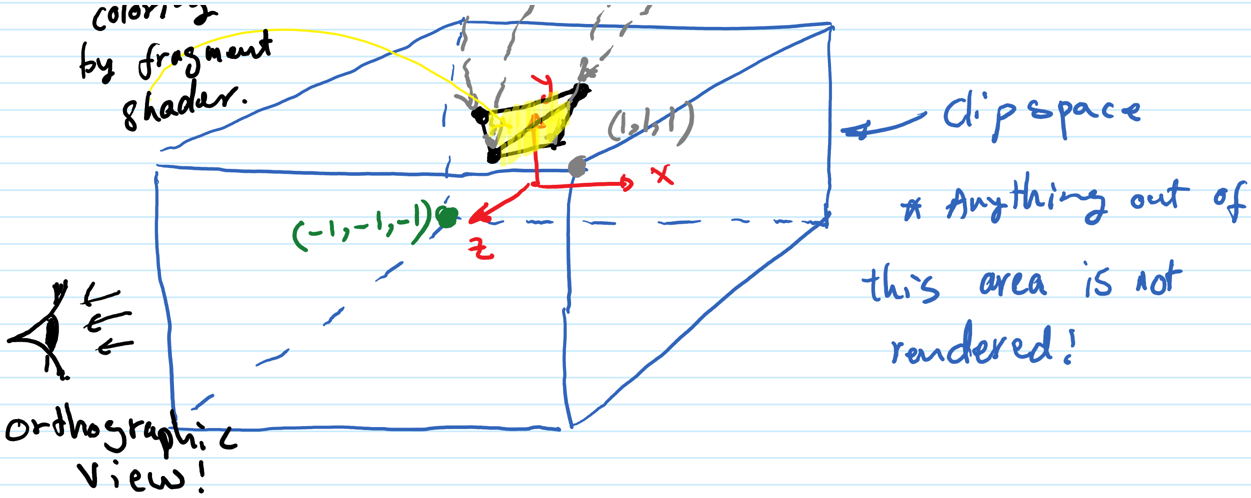
Geometrical Primitives

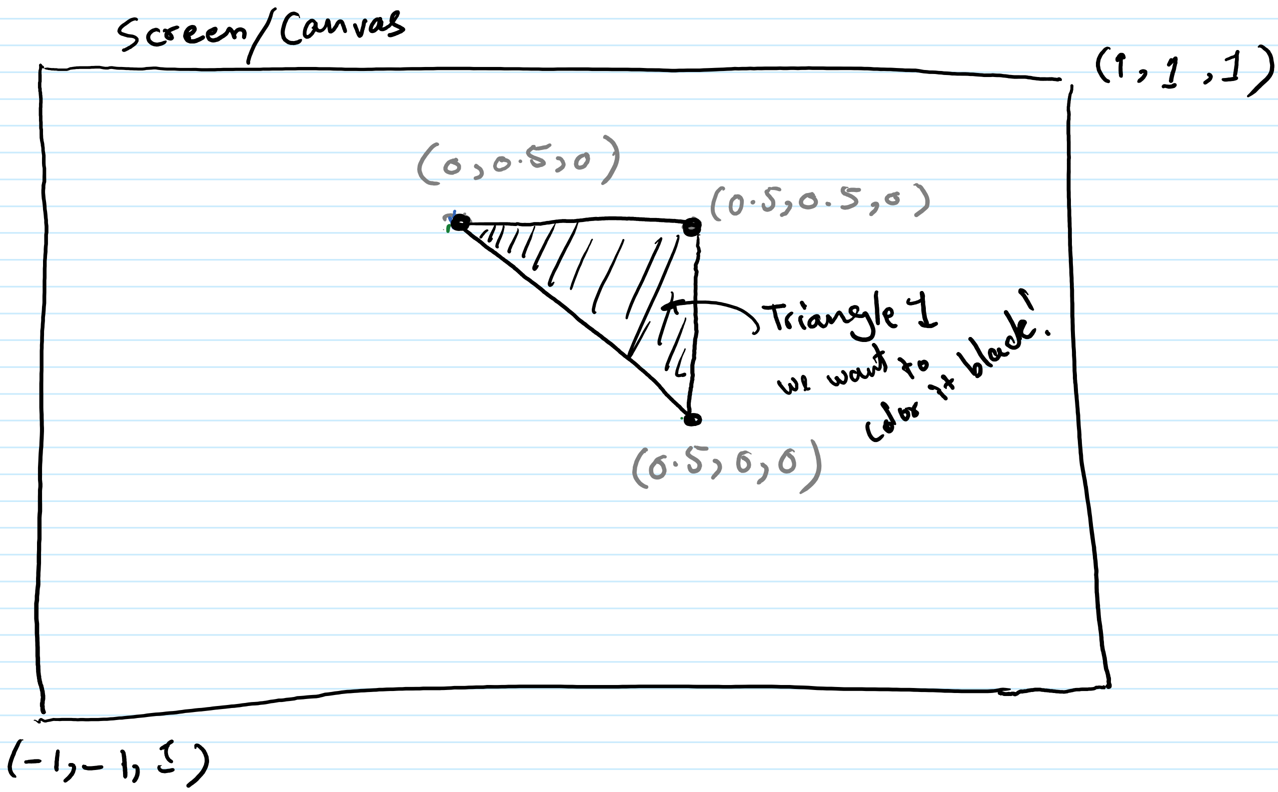


* Points
* Lines
* Triangles

Physical World:







We start by entering project 01:

Let's start from the index.html file. To run each program you need to open this file using FireFox. Today, we are not going to change it. But, let's have a look at it ;-)

```
<!DOCTYPE html>
<html>
<head>
    <title>Triangle</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1">

    <script src='config.js'></script>
    <script data-main="app/main" src="libs/require.js"></script>
</head>
<!--@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@-->
<!--                          MAIN BODY OF THE HTML                                      -->
<!--@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@-->
<body id='main-body' >
```

↑ The program will look for main.js under directory "app"

```
</head>
<!--@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@-->
<!--                         MAIN BODY OF THE HTML                            -->
<!--@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@-->
<body id='main-body' >
    <canvas id="canvas_1"
            style="border:1px solid #000000;" >
        <!-- This message is displayed if canvas is not available -->
        Your browser does not support the HTML5 canvas tag.
    </canvas>
<body>
</html>
```

The program will look for main.js under directory

This is where we draw on the screen!

Now, let's have a look at the main.js file:

```
define([    'shader!vertShader.vert',
            'shader!compShader.frag',
            'Abubu/Abubu'
            ],
function(   vertShader,
            compShader,
            Abubu
            ){
"use strict" ;

/*========================================================================
 * Global Parameters
 *========================================================================
 */
var log = console.log ;
var env = {} ;
window.env = env ;
```

Loads "vertShader.frag" file into the variable "vertShader"

```
/*========================================================================
 * Initialization of the GPU and Container
 *========================================================================
 */
function loadWebGL()
{
    env.width = 512 ;
    env.height = 512 ;

/*------------------------------------------------------------------------
 * Access and setup canvas
 *------------------------------------------------------------------------
 */
    var canvas_1 = document.getElementById("canvas_1") ;
    canvas_1.width = env.width ;
    canvas_1.height = env.height ;

    /* Setup geometry */
    env.geometry = {} ;
    env.geometry.vertices = [
            0.5,    0.5,    0.,
            0.0,    0.5,    0.,
            0.5,    0.0,    0.,
        ] ;
    env.geometry.noVertices= 3 ; // No of vertices
    env.geometry.noCoords  = 3 ; // No of coordinates
    env.geometry.premitive = 'triangle_strip' ;

    /* Setup a solver */
    var renderer = new Abubu.Solver( {
        fragmentShader  : compShader.value,
        vertexShader    : vertShader.value,
        geometry : env.geometry ,
        canvas : canvas_1,
    } ) ;

    renderer.render() ;
}/*  End of loadWebGL  */
```

we define this function to run our webgl program

Bring in canvas with "canvas_1" id for drawing!

set its width & height

x-coordinates, first point

we define our pipeline here!

where drawing needs to happen!

Ordering the program to draw the geometry using our pipeline on the canvas!

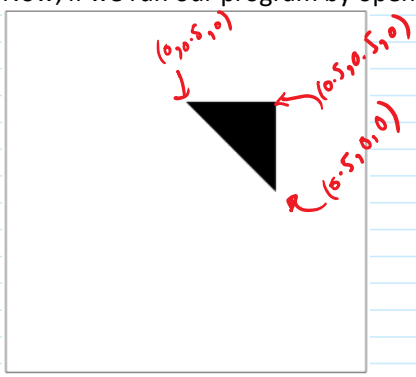
Now, let's edit the vertex shader (vertShader.vert). Since coordinates of our triangle fit on the screen, we can do a one to one map:

```
#version 300 es
precision highp float ;
precision highp int ;

in vec4 position;

void main() {
    gl_Position = vec4(position.x, position.y,position.z,1.0);
}
```

Then, let's design our fragment shader (compShader.frag) to color our geometry:

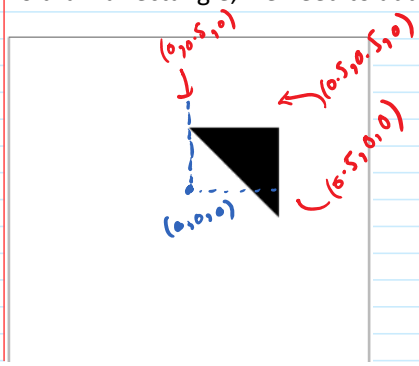
```
#version 300 es
precision highp float ;
precision highp int ;

out vec4 outcolor ; // output color of the shader

void main() {
    outcolor = vec4(0.,0.,0.,1.) ;
    return ;
}
```

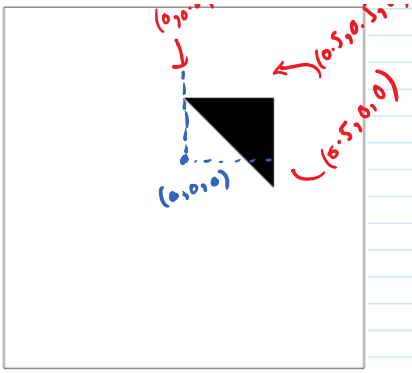
Now, if we run our program by opening it in FireFox we should see the following:



Now, let's copy the content of project 01-traingle, to 02-rectangle and continue…

To draw a rectangle, we need to add one more triangle to our previous example:

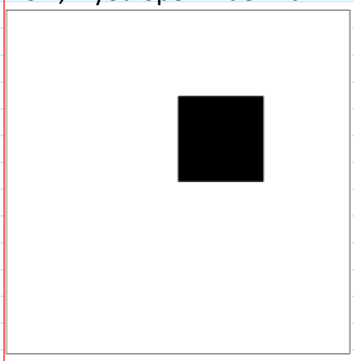




So, let's add it to the list of our vertices in main.js and edit the file accordingly.

```
    /* Setup geometry */
    env.geometry = {} ;
    env.geometry.vertices =  [
            0.5,    0.5,    0.,
            0.0,    0.5,    0.,
            0.5,    0.0,    0.,
            0.0,    0.0,    0.,
        ] ;
    env.geometry.noVertices= 4 ; // No of vertices
    env.geometry.noCoords  = 3 ; // No of coordinates
    env.geometry.premitive = 'triangle_strip' ;

    /* Setup a solver */
    var renderer = new Abubu.Solver( {
        fragmentShader  : compShader.value,
        vertexShader    : vertShader.value,
        geometry : env.geometry ,
        canvas : canvas_1,
    } ) ;
```

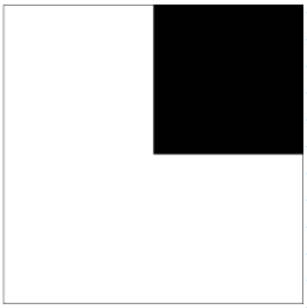
Now, if you open index.html in your FireFox browser you should see:



Let's now modify the rectangle to a unit rectangle which can be more useful. Let's go back to main.js and edit accordingly:

```
/* Setup geometry */
env.geometry = {} ;
env.geometry.vertices =  [
        1.0,    1.0,    0.,
        0.0,    1.0,    0.,
        1.0,    0.0,    0.,
        0.0,    0.0,    0.,
    ] ;
env.geometry.noVertices= 4 ; // No of vertices
env.geometry.noCoords  = 3 ; // No of coordinates
env.geometry.premitive = 'triangle_strip' ;
```

And now the program will plot:



Now, we can do a simple mapping in our vertex shader (vertShader.vert) to map our geometry to (-1,-1,0) on the bottom left corner and (1,1,0) on the top right corner:

```
#version 300 es
precision highp float ;
precision highp int ;

in vec4 position;

void main() {
    gl_Position = vec4(position.xy*2.-1., position.z,1.0);
}
```

This carries our mapping!

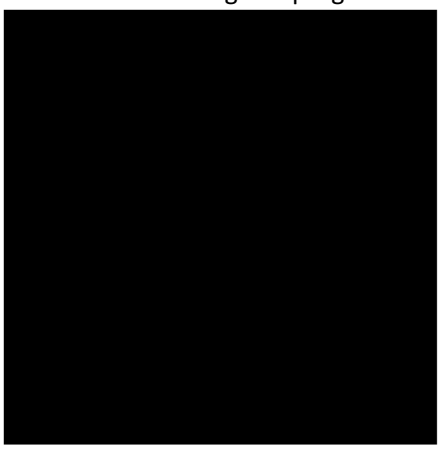
We can also write the above shader in a compact form:

```
#version 300 es
precision highp float ;
precision highp int ;

in vec4 position;

void main() {
    gl_Position = vec4(position.xy*2.-vec2(1.),position.z,1.0);
}
```

vec2(1.) ≡ vec2(1.,1.)

we can do simple vector & matrix operations in shaders!

The result of running this program is drawing the following shape:



** NOTE **

The default geometry in Abubu.js is the unit rectangle that we just defined and

used. Hence, if we don't provide any geometry to the solver, the solver will assume a unit rectangle (two triangles) geometry.

Now, let's copy the content of project 02 into the directory starting with 03 to start a new project.

Other than mapping vertex positions, vertex shader can calculate interpolated value for each pixel on the screen. For example if we can calculate the interpolated the position of each pixel in our physical world. Let's edit the vertex shader (vertShader.vert).

```
#version 300 es
precision highp float ;
precision highp int ;

in vec4 position;

out vec2 pixPos ;

void main() {
    pixPos = position.xy ;
    gl_Position = vec4(position.xy*2.-vec2(1.),position.z,1.0);
}
```

Now, let's bring our interpolated values into our fragment shader (compShader.frag) and use it:

```
#version 300 es
precision highp float ;
precision highp int ;

out vec4 outcolor ; // output color of the shader

in  vec2 pixPos ;

void main() {
    outcolor = vec4(pixPos.x,0.,0.,1.) ;
    return ;
}
```

Now, if we run the program we will get the following:

Now, let's copy the content of project 03 into the 04 project. From, now on we will just use the vertex shader that we have just developed and keep reusing it.

All the magic will happen in the fragment shader. So, let's make another change in our fragment shader (compShader.frag).

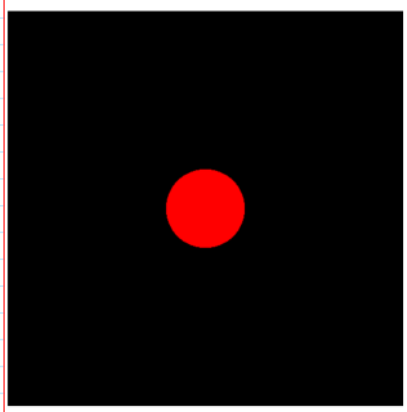
```
#version 300 es
precision highp float ;
precision highp int ;

out vec4 outcolor ; // output color of the shader

in  vec2 pixPos ;

void main() {
    if ( length(pixPos-vec2(0.5,0.5))<0.1){
        outcolor = vec4(1.0,0.,0.,1.) ;
    }else{
        outcolor = vec4(0.,0.,0.,1.) ;
    }
    return ;
}
```

Check if distance of the pixel from (0.5,0.5) is less than 0.1

return color red

return black!

Now, if we run the program (by now you should know by opening index.html in FireFox) we should get the following result:



Quick question: is the graphics card drawing a circle?

.
.
.
.
.
.

Answer: NO! The graphics card is drawing two rectangles and our coloring recipe in the fragment shader colors the pixel in a way that a circle is formed!