Lab 2: Different Data Types and Interactive Computer Graphics

Objectives:

1. Learn Different types of data used in GLSL such as attribute, uniform and varying
2. Implementing mouse interaction in WebGL.

Data Flow in the Graphics Pipeline:

The JavaScript side of the program sends values for attributes and uniform variables to theGPU and then issues a command to draw a primitive. The GPU executes the vertex shader once for each vertex. The vertex shader can use the values of attributes and uniforms. It assigns values to gl_Position and to any varying variables that exist in the shader. After clipping, rasterization, and interpolation, the GPU executes the fragment shader once for each pixel in the primitive. The fragment shader can use the values of varying variables, uniform variables, and gl_FragCoord. It computes a value for gl_FragColor. The figure 1 summarizes the flow of data:

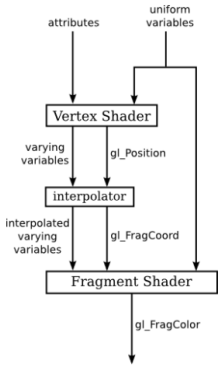


Figure 1 Graphics pipeline in dataflow

Attribute:

An attribute can take a different value for each vertex in a primitive. The basic idea is that the complete set of data for the attribute is copied in a single operation from a JavaScript array into memory that is accessible to the GPU. Unfortunately, setting things up to make that operation possible is non-trivial.

Uniform and Varying:

Global variable declarations in a vertex shader can be marked as attribute, uniform, or varying. A variable declaration with none of these modifiers defines a variable that is local to the vertex shader. Global variables in a fragment can optionally be modified with uniform or varying, or they can be declared without a modifier. A varying variable should be declared in both shaders, with the same name and type. This allows the GLSL compiler to

determine what attribute, uniform, and varying variables are used in a shader program. This qualifier forms a link between a vertex shader and fragment shader for interpolated data. It can be used with the following data types - *float, vec2, vec3, vec4*, *mat2, mat3, mat4*, or arrays. (see sample code 1 and 2). The summary on attribute, uniform and varying qualifiers are listed in the Figure 2.2

Sr.No.	Qualifier & Description
1	**attribute** This qualifier acts as a link between a vertex shader and OpenGL ES for per-vertex data. The value of this attribute changes for every execution of the vertex shader.
2	**uniform** This qualifier links shader programs and the WebGL application. Unlike attribute qualifier, the values of uniforms do not change. Uniforms are read-only; you can use them with any basic data types, to declare a variable. **Example** − uniform **vec4** lightPosition;
3	**varying** This qualifier forms a link between a vertex shader and fragment shader for interpolated data. It can be used with the following data types − float, vec2, vec3, vec4, mat2, mat3, mat4, or arrays. **Example** − varying **vec3** normal;

Figure 2: Attribute, uniform and varying qualifiers

Sample code 1

See sample1.txt from the folder

Sample code 2

See sample2.txt from the folder

Mouse Interaction:

The *mousedown* event is fired at an Element when a pointing device button is pressed while the pointer is inside the element. We can utilize this to call draw functions to render on the canvas based on clicking. The following code segment from code 2.2 is responsible for invoking draw function for mouse click.

GLSL Control Statements and Built-in functions

Control Statements:

GLS is a C-like language. We can use our basic concepts of control statements (e.g. if-else, switch-case, loop, etc). A helpful resource is: https://www.shaderific.com/glsl-statements. An example of IF-ELSE in vertex shader is provided below.

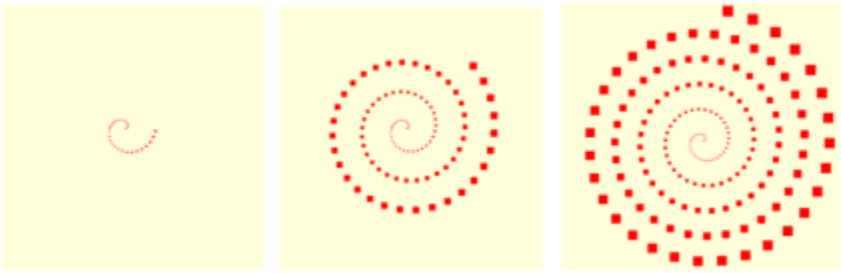
```
 attribute vec3 a_coords;
attribute vec3 a_colors;
uniform float u_shift;
varying vec3 v_color;

void main() {
    if (u_shift < 0.7)
        gl_Position = vec4(a_coords.x - u_shift,
                           a_coords.y,
                           a_coords.z,
                           1.0);
    else
        gl_Position = vec4(a_coords.x,
                           a_coords.y,
                           a_coords.z,
                           1.0);
    v_color = a_colors; }
```

Built-in Functions:
GLSL provides a significant number of built-in functions and you should be familiar with them. Please see this page (https://www.shaderific.com/glslfunctions) to get idea regarding some of these functions. An example of a vertex shader code segment that uses a built-in function *clamp()* is provided below.

```
void main() {
    gl_Position = vec4(clamp(a_coords.x - u_shift, -0.5, 1.0),
                             a_coords.y,
                             a_coords.z,
                             1.0);
```

Task

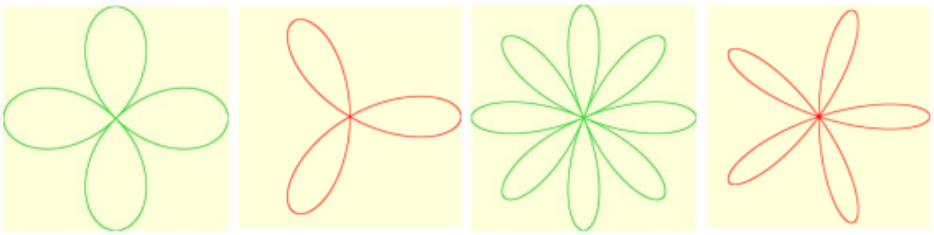
1. Create a 2D spiral. For each click, the spiral will keep increasing. The outer points will be bigger than the inner points depending on the distance from the center. Note that, you have to send 2D data to the GPU from CPU. The following figure shows different situation of the canvas after several mouse clicking (from left to right)



Hints

- Use *gl_PointSize* in the vertex shader to fix the size of the point. (Example: https://www.tutorialspoint.com/webgl/webgl_drawing_points.htm)
- Use GLSL *distance()* function to calculate the distance between a vertex and the center in the vertex shader.
- To generate the vertices for a 2D spiral in CPU, you can use JavaScript's Math library to apply the formula, e.g. *Math.cos()*. Use *push()* function to build up an array of vertices of the spiral using a loop. • Appy optimistically while using/ reusing vertex buffer.

2. Create a 2D flower based on *Rhodonea* curve. Use *GL_POINTS* in your draw call. For each click, the arrangement of the petals will keep changing. Also, the color of the points will be alternated between green and red for each click. Note that, you have to send 2D data to the shader from CPU



Hints

- Rose in math: https://en.wikipedia.org/wiki/Rose_(mathematics)
- You can track odd/ even clicking in the shader to alternate color to generate the vertices for a 2D spiral in CPU, you can use JavaScript's Math library to apply the formula, e.g. *Math.cos().* Use *push()* function to build up an array of vertices of the spiral using a loop.
- Appy optimistically while using/ reusing vertex buffer