计算机图形学实验7：GLSL

一、实验目的和要求

1. 掌握基本的着色器编程
2. 熟悉GLSL的语法
3. 理解着色器级别的纹理支持
4. 了解边缘检测算法的基本原理

二、实验内容和原理

在WebGL中使用着色器

在使用WebGL的任何操作之前，我们都需要得到WebGL的上下文`WebGLRenderingContext`的实例`gl`。以后的WebGL方法都通过`gl.method()`的形式给出。

我们要使用着色器，光有代码是不够的，我们还需要一个着色器对象来让其真正地工作起来：

1. 创建着色器对象：

   我们使用`gl.createShader()`函数来创建一个着色器对象。

   其定义如下：

   ```
   createShader(type: GLenum): WebGLShader | null;
   ```

   其中，`type`的可选值一般为`gl.VERTEX_SHADER`或者`gl.FRAGMENT_SHADER`，对应顶点着色器以及片段着色器。

2. 传入着色器代码：

   我们通过`gl.shaderSource()`函数来将着色器代码传入到指定的着色器中。

   其定义如下：

   ```
   shaderSource(shader: WebGLShader, source: string): void;
   ```

   其中，`shader`是指定的着色器对象，`source`是着色器代码。

3. 编译着色器：

   我们通过`gl.compileShader()`函数 来编译着色器代码，这样的着色器才能够正常地使用。

   其定义如下：

   ```
   compileShader(shader: WebGLShader): void;
   ```

我们通过一个函数来处理这些麻烦的过程，并提供简单的编译失败报错机制。

```
/**
 * Create and compile a shader
 * @param {WebGLRenderingContext} gl
 * @param {Number} type
 * @param {string} source
 */
```

```
function createShader(gl, type, source) {
    var shader = gl.createShader(type);
    gl.shaderSource(shader, source);
    gl.compileShader(shader);

    //得到编译状态参数，看看编译是否成功
    var success = gl.getShaderParameter(shader, gl.COMPILE_STATUS);
    if (success){
        return shader;
    }

    //如果编译失败，我们需要打印错误信息
    console.log(gl.getShaderInfoLog(shader));
    gl.deleteShader(shader);
}
```

现在我们有了着色器对象，我们还需要将着色器对象链接到我们使用的当前的着色程序中，才能使其真正发挥作用：

1. 创建着色程序：

   首先我们通过 `gl.createProgram()` 函数创建一个着色程序。

   其定义如下：

   ```
   createProgram(): WebGLProgram | null;
   ```

2. 将着色器绑定到着色程序：

   接着，我们使用 `gl.attachShader()` 函数将着色器对象绑定到着色器程序中。

   其定义如下：

   ```
   attachShader(program: WebGLProgram, shader: WebGLShader): void;
   ```

   注意到我们只需要传入着色器对象即可，程序会自动识别着色器对象对应的类型并完成正确的绑定。

3. 链接着色程序：

   最后，我们使用 `gl.linkProgram()` 函数来完成着色程序与着色器对象之间的链接。

   其定义如下：

   ```
   linkProgram(program: WebGLProgram): void;
   ```

同样的，我们也可以写一个函数来处理这些麻烦的过程，并提供简单的报错机制：

```
/**
 * Create a program and link its shaders
 * @param {WebGLRenderingContext} gl
 * @param {WebGLShader} vertexShader
 * @param {WebGLShader} fragmentShader
 */
function createProgram(gl, vertexShader, fragmentShader) {
    var program = gl.createProgram();
    gl.attachShader(program, vertexShader);
    gl.attachShader(program, fragmentShader);
    gl.linkProgram(program);
```

```
    var success = gl.getProgramParameter(program, gl.LINK_STATUS);
    if (success) {
        return program;
    }

    console.log(gl.getProgramInfoLog(program));
    gl.deleteProgram(program);
}
```

既然我们已经有了这两个函数了，我们可以进一步简化我们的着色程序初始化过程：

```
/**
 * Initialize the program
 * @param {WebGLRenderingContext} gl
 * @param {string} src_vs The source of vertex shader
 * @param {string} src_fs The source of fragment shader
 * @returns {WebGLProgram}
 */
function programInit(gl, src_vs, src_fs) {
    var vertexShaderSource = getShaderSource(src_vs);
    var fragmentShaderSource = getShaderSource(src_fs);

    var vertexShader = createShader(gl, gl.VERTEX_SHADER, vertexShaderSource);
    var fragmentShader = createShader(gl, gl.FRAGMENT_SHADER, fragmentShaderSource);

    var program = createProgram(gl, vertexShader, fragmentShader);

    return program;
}
```

这样一来，我们只需要考虑如何写着色器程序，并不用再在意着色程序是怎样使用我们写的程序的了。

WebGL中的GLSL

GLSL是一种类C语言，一个典型的GLSL程序包括一系列的变量声明以及若干个函数和方法，包含一个返回值为`void`的主函数`main()`。在顶点着色器中，我们的主要任务是为`gl_Position`赋值；在片段着色器中，我们的主要任务是为`gl_FragColor`赋值。

着色器不能自动地从缓冲处获得数据，当我们的着色器面临着各种各样的外部数据输入需求时，我们需要显式地定义数据的获取方式。GLSL的数据获取包括以下四种：

1. `attribute`：属性和缓冲

   缓冲是发送到GPU的二进制数据序列，我们需要定义对应的属性来告诉GPU我们希望读入的是什么样的数据。譬如`attribute vec2 a_position`就是告诉GPU我们每次从缓冲中读入一个二维向量。其含义是一个节点的位置。

2. `unifrom`：全局变量

   全局变量是在着色器运行前通过API提供的接口预先赋值的变量，并在渲染过程中全局有效。

3. `texture`：纹理

   纹理是一个数据序列，可以在着色程序运行中随意读取其中的数据。 大多数情况存放的是图像数据。

4. `varying`：可变量

   可变量是一类特殊的变量，我们要在顶点着色器和片段着色器中同时定义这个变量，以方便顶点着色器向片段着色器传值。在现在的OpenGL使用的GLSL中，统一使用in/out关键字来完成传值过程。但是WebGL显然保留了更加原始的OpenGL特性。

对于`attribute`以及`uniform`，我们需要在外部给予着色器输入：

- `attribute`：我们要将属性绑定到一个缓冲上，并提供给该属性足够多的数据以使其正确地读取和使用缓冲中的数据：

  1. 得到属性的地址

     我们使用`gl.getAttribLocation()`函数来得到我们要使用的属性的地址。

     其定义如下：

     ```
     getAttribLocation(program: WebGLProgram, name: string): GLint;
     ```

     其中，`name`是我们在顶点着色器中定义的属性名。

  2. 启用属性

     我们使用`gl.enableVertexAttribArray()`函数来启用对应的属性。

     其定义如下：

     ```
     enableVertexAttribArray(index: GLuint): void;
     ```

     其中，使用的`index`就是我们之前得到的属性的地址。

  3. 将缓冲绑定到绑定点上

     我们使用`gl.bindBuffer`来将缓冲绑定到对应的绑定点上（一般为`gl.ARRAY_BUFFER`）。

     其定义如下：

     ```
     bindBuffer(target: GLenum, buffer: WebGLBuffer | null): void;
     ```

     其中，`target`就是我们的绑定点，如`Buffer`为空则释放该绑定点。

  4. 指定属性读取缓冲的方式

     在最后，光有缓冲是不够的，着色器依然不知道应该如何正确地读取缓冲。有时候，一个顶点会使用<<顶点坐标>,<纹理坐标>,<法向量>,<颜色>>这样的元组来表示，这样一来一个条目中会包含有各种属性的信息，而我们只需要绑定一个缓冲，并使各个属性都能按照规则正确地读入数据。

     我们调用`gl.vertexAttribPointer()`函数来告知属性它应当如何读取数据。同时，将属性通过绑定点与缓冲相绑定，之后，绑定点可以绑定到其它的缓冲对象，而不影响现在的结果。

     我们一般使用下面的代码块来完成这个操作：

```
{
    let size = 2;           //一个条目包含几个元素
    let type = gl.FLOAT;    //每个元素的数据类型
    let normalize = false; //恒为false
    let stride = 0;         //每次读取完属性之后，要移动多少单位以到达下一个条目
    let offset = 0;         //条目开始位置的偏移
    gl.vertexAttribPointer(
        positionLocation, size, type, normalize, stride, offset
    );
}
```

- `uniform`：我们要在着色程序开始前对这些全局变量进行传值：

    1. 获得全局变量的地址：

        我们使用函数`gl.getUniformLocation()`函数来获得其对应的全局变量的地址。

        其定义如下：

        ```
        getUniformLocation(program: WebGLProgram, name: string): WebGLUniformLocation | null;
        ```

        其中，`name`是全局变量的变量名。

    2. 通过这个地址为全局变量赋值：

        我们使用函数`gl.Uniform*()`函数来为全局变量直接赋值。一般而言，`Unifrom*()`的第一个参数全局变量的地址，之后的参数就是我们要传入的值。如果我们要对一个数组赋值的话，我们应该传入该数组的第一个元素（实际上定义中只要求传入一个可迭代对象）。

Sobel算子与边缘检测

我们如何定义一个图形的边缘？一般认为，一个像素，如果其周围的亮度变化剧烈，那么我们可以认为这个像素处在图形的边缘处（以人眼的分辨能力作为指标）。那么，要如何较为准确地表示亮度变化的剧烈程度？一个简单的办法就是差分。我们都知道，导数$\frac{dy}{dx}$表示的几何含义就是在某点上，y沿x方向的变化的剧烈程度。由于像素是一个个离散的点，那么dx可看作是均一的，只要简单地计算dy就能够表示类似的含义。

因此，我们给出简单的一个卷积核：

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$$

这样的一个卷积核计算的是中心像素周围所有像素沿水平方向的差分g_x，当然是不够的。我们还要计算垂直方向的差分g_y：

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

最后，我们用$G = \sqrt{g_x^2 + g_y^2}$近似地表达该像素附近的亮度变化的剧烈程度。当G大于我们设定给的阈值时，我们就将该像素高亮，认为其在边缘上。

Sobel算子是对上述方法的一个小改进，这个方法认为离得越远的像素对亮度变化的贡献越小，因此各行不再是均一的，我们得到水平方向的卷积核：

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

以及垂直方向的：

$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

三、操作系统与环境

- 操作系统：Windows 10
- 开发平台：Visual Studio Code， 使用Node.js的Servez架设简单Web Server
- 编程环境：JavaScript + WebGL

四、操作方法与实验步骤

顶点着色器代码

顶点着色器不需要执行什么特别的功能。只要将纹理坐标传递给片段着色器即可。

```
attribute vec2 a_position;
attribute vec2 a_texCoord;

uniform vec2 u_resolution;

varying vec2 v_texCoord;

void main(){
    //计算裁剪空间坐标
    vec2 zeroToOne = a_position / u_resolution;
    vec2 zeroToTwo = zeroToOne * 2.0;
    vec2 clipSpace = zeroToTwo - 1.0;
    gl_Position = vec4(clipSpace * vec2(1, -1), 0, 1);

    //传递纹理坐标
    v_texCoord = a_texCoord;
}
```

由于我们可以加载不同的图片，为了方便，所有的顶点坐标都使用了屏幕像素坐标。但是，为了能够在着色器中运行，我们还是要把它映射成裁剪空间坐标。

片段着色器代码

```
precision mediump float;

uniform sampler2D u_image;
uniform vec2 u_textureSize;
uniform float u_Sobel1[9];
uniform float u_Sobel2[9];

varying vec2 v_texCoord;
```

```
9
10 void main(){
11     //一个像素的在水平和垂直方向上的偏移大小
12     vec2 onePixel = vec2(1.0, 1.0) / u_textureSize;
13
14     //在水平方向上进行差分
15     vec4 colorSumx =
16         texture2D(u_image, v_texCoord + onePixel * vec2(-1, -1)) *
   u_Sobel1[0] +
17         texture2D(u_image, v_texCoord + onePixel * vec2( 0, -1)) *
   u_Sobel1[1] +
18         texture2D(u_image, v_texCoord + onePixel * vec2( 1, -1)) *
   u_Sobel1[2] +
19         texture2D(u_image, v_texCoord + onePixel * vec2(-1,  0)) *
   u_Sobel1[3] +
20         texture2D(u_image, v_texCoord + onePixel * vec2( 0,  0)) *
   u_Sobel1[4] +
21         texture2D(u_image, v_texCoord + onePixel * vec2( 1,  0)) *
   u_Sobel1[5] +
22         texture2D(u_image, v_texCoord + onePixel * vec2(-1,  1)) *
   u_Sobel1[6] +
23         texture2D(u_image, v_texCoord + onePixel * vec2( 0,  1)) *
   u_Sobel1[7] +
24         texture2D(u_image, v_texCoord + onePixel * vec2( 1,  1)) *
   u_Sobel1[8];
25     //利用经验公式计算明度，这是305910公式的（据说更准确的）一个版本。
26     float luma_x = 0.2126 * colorSumx.r + 0.7152 * colorSumx.g + 0.0722 *
   colorSumx.b;
27
28     //在垂直方向上进行差分
29     vec4 colorSumy =
30         texture2D(u_image, v_texCoord + onePixel * vec2(-1, -1)) *
   u_Sobel2[0] +
31         texture2D(u_image, v_texCoord + onePixel * vec2( 0, -1)) *
   u_Sobel2[1] +
32         texture2D(u_image, v_texCoord + onePixel * vec2( 1, -1)) *
   u_Sobel2[2] +
33         texture2D(u_image, v_texCoord + onePixel * vec2(-1,  0)) *
   u_Sobel2[3] +
34         texture2D(u_image, v_texCoord + onePixel * vec2( 0,  0)) *
   u_Sobel2[4] +
35         texture2D(u_image, v_texCoord + onePixel * vec2( 1,  0)) *
   u_Sobel2[5] +
36         texture2D(u_image, v_texCoord + onePixel * vec2(-1,  1)) *
   u_Sobel2[6] +
37         texture2D(u_image, v_texCoord + onePixel * vec2( 0,  1)) *
   u_Sobel2[7] +
38         texture2D(u_image, v_texCoord + onePixel * vec2( 1,  1)) *
   u_Sobel2[8];
39     float luma_y = 0.2126 * colorSumy.r + 0.7152 * colorSumy.g + 0.0722 *
   colorSumy.b;
40
41     float luma = luma_x * luma_x + luma_y * luma_y;
42
43     //如果其在边缘上，就染白；不然染黑
44     if (luma > 0.2) gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);
45     else gl_FragColor = vec4(0.0, 0.0, 0.0, 1.0);
46 }
```

在片段着色器中，我们遇到的第一个问题是如何得到一个像素的周围八个像素的坐标。我们需要分别计算出一个像素在纹理坐标系下的水平长度和垂直长度是多少，即取图片的长度和宽度的倒数。

当我们利用卷积核计算完毕以后，我们得到的实际上是一个四维向量。我们依然不知道这个像素点上的亮度对应的是多少。此时，我选择利用Gray公式的一个据说比较准确的版本来计算亮度。

五、实验数据记录与处理

实验的主程序如下：

```
//主函数，用于加载图片，图片的onload方法会调用我们的渲染函数。
function main() {
    let img = new Image();

    img.src = "wallpaper.jpg";
    img.onload = function() {
        render(img);
    }
}

/**
 *
 * @param {HTMLImageElement} img
 */
function render(img) {
    //初始化WebGL上下文
    let gl = contexInit();
    if (!gl) {
        alert("Failed to initialize WebGL contex!");
        return;
    }

    //初始化着色程序
    let program = programInit(gl, "./Sobel.vs", "./Sobel.fs");

    //得到属性地址
    let positionLocation = gl.getAttribLocation(program, "a_position");
    let texcoordLocation = gl.getAttribLocation(program, "a_texCoord");

    //设置顶点坐标缓冲数据以及纹理坐标缓冲数据
    let positionBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
    setRectangle(gl, 0, 0, img.width, img.height);

    let texcoordBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, texcoordBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([
        0.0, 0.0,
        1.0, 0.0,
        0.0, 1.0,
        0.0, 1.0,
        1.0, 0.0,
        1.0, 1.0,
    ]), gl.STATIC_DRAW);

    //创建并配置纹理对象
```

```
    let texture = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, texture);

    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);

    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE,
img);

    //获取全局变量地址（采样器不需要这么麻烦）
    let resolutionLocation = gl.getUniformLocation(program,
"u_resolution");
    let textureSizeLocation = gl.getUniformLocation(program,
"u_textureSize");
    let SobelxLocation = gl.getUniformLocation(program, "u_Sobel1[0]");
    let SobelyLocation = gl.getUniformLocation(program, "u_Sobel2[0]");

    //将画布大小调整为浏览器窗口大小
    resizeCanvasToDisplaySize(gl.canvas);

    gl.viewport(0, 0, gl.canvas.clientWidth, gl.canvas.clientHeight);

    gl.clearColor(0, 0, 0, 0);
    gl.clear(gl.COLOR_BUFFER_BIT);

    gl.useProgram(program);

    //绑定传冲和属性
    gl.enableVertexAttribArray(positionLocation);
    gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);

    {
        let size = 2;
        let type = gl.FLOAT;
        let normalize = false;
        let stride = 0;
        let offset = 0;
        gl.vertexAttribPointer(
            positionLocation, size, type, normalize, stride, offset
        );
    }

    gl.enableVertexAttribArray(texcoordLocation);
    gl.bindBuffer(gl.ARRAY_BUFFER, texcoordBuffer);

    {
        let size = 2;
        let type = gl.FLOAT;
        let normalize = false;
        let stride = 0;
        let offset = 0;
        gl.vertexAttribPointer(
            texcoordLocation, size, type, normalize, stride, offset
        );
    }

```

```
    //传入全局变量的值
    gl.uniform2f(resolutionLocation, gl.canvas.width, gl.canvas.height);
    gl.uniform2f(textureSizeLocation, img.width, img.height);

    let Sobelx = [
         1,  2,  1,
         0,  0,  0,
        -1, -2, -1,
    ];
    gl.uniform1fv(SobelxLocation, Sobelx);

    let Sobely = [
        1, 0, -1,
        2, 0, -2,
        1, 0, -1,
    ];
    gl.uniform1fv(SobelyLocation, Sobely);

    //绘制图形
    {
        let primitiveType = gl.TRIANGLES;
        let offset = 0;
        let count = 6;
        gl.drawArrays(primitiveType, offset, count);
    }
}

/**
 * 这个函数知识用来方便地生成矩形的顶点缓冲数据
 * @param {WebGLRenderingContext} gl
 * @param {Number} x
 * @param {Number} y
 * @param {Number} width
 * @param {Number} height
 */
function setRectangle(gl, x, y, width, height)
{
    let x0 = x;
    let y0 = y;
    let x1 = x + width;
    let y1 = y + height;

    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([
        x0, y0,
        x1, y0,
        x0, y1,
        x0, y1,
        x1, y0,
        x1, y1,
    ]), gl.STATIC_DRAW);
}
```

六、实验结果与分析

一个实例已经部署在我的个人网页上：http://www.mckpersonal.cn/MyWebGL/CG_Lab_7/index.html

实验源代码托管在GitHub：https://github.com/Sigmoni/MyWebGL/tree/main/CG_Lab_7

七、讨论与心得

本次实验使用了简单但是实用的边缘检测技术，让我见识到了图形学的妙用，进一步感受到了图形学的深奥与广阔。