CUSTOM POST-PROCESSING WEBGL EFFECTS

DOCUMENTATION

GOAL

The goal of this project is to learn more about shader programming for the GPU via WebGL and Three.js. The focus will be on post-processing effects which are shader programs that are run in passes on the initial rendering of a scene. Thus, the effects themselves are not applied to the objects in the scene, but rather, applied to the 2D output image of the rendered scene.

PROGRAM STRUCTURE

The structure is broken up into separate files for easier modularity and development:

The **project.html** file contains the base code that sets up the webpage. All the Javascript code that is necessary to run the program is imported in this html file, and a simple *init* function utilizes these imported files to initialize a *MainWindow* and *GUI* class.

The **gui.js** file contains all the code necessary for setting up dat.gui for enabling/disabling certain custom shaders and changing various settings inside them. An *effectController* contains the settings for each of the custom shaders, as well as an “is active” flag that tells the effect composer whether to use that specific shader in the next rendering pass. Most of the settings have an *onChange* callback function that changes the uniform values that are sent to the shader based on the current settings chosen through the GUI.

The **mainWindow.js** file contains most of the Javascript code necessary for the project. It handles the initialization of the Three.js scene as well as handling the Three.js *EffectComposer* for rendering. Threads are created during the initialization that handle simple animations by sending new uniform values to specified shaders every *n* seconds via Javascript’s *setInterval* function. Event handlers are also created in this file to handle window resizing and camera changes.

The **effects** directory contains separate Javascript files for each of the custom shaders implemented. Inside each of these files, there is only a Javascript JSON object containing definitions for the shader’s uniforms, vertexShader, and fragmentShader.

EFFECT COMPOSER

For handling the rendering of the scene, the Three.js *EffectComposer* is utilized. With the *EffectComposer*, rendering can be broken up into multiple passes. When working with shaders in this framework, the output of the previous pass is automatically sent to the shader as a uniform 2D texture called *tDiffuse*. Since this project is going to be working with post-processing effects, utilizing the 2D texture of the previous pass is necessary.

In this project, the first pass of the *EffectComposer* is always a basic *RenderPass*. This renders the scene normally, without any post-processing effects. However, the output of this pass is not sent to the screen; it is instead sent as a 2D texture to the next rendering pass.

The next set of passes are the various custom shaders that were implemented. In this project, they are set in a certain order, however you can toggle each one to be included or not included in the *EffectComposer* via the GUI. The *EffectComposer* automatically sends the final pass output to the screen.

BLOOM SHADER

A bloom effect causes areas of bright light to extend beyond their natural borders. This can cause the lights to appear brighter than they would otherwise be. A simple way to simulate a bloom effect is to blur an image, and then add a fraction of the blurred image's color values to the original image's color values. The values produced are then clipped to the range (0.0, 1.0).

In this shader, a basic vertex shader is used. Most of the work comes from the fragment shader. To produce a blur effect in WebGL, a fast Gaussian blur technique is used, adapted from the code available at https://github.com/Jam3/glsl-fast-gaussian-blur. This provides the blurred color value for the image at each pixel. This value is then added to the color value of each pixel in the original image, which is obtained through the *tDiffuse* uniform. The result is then clipped to the range (0.0, 1.0) to produce a final RGB color value for the fragment.

The settings for this shader allow us to send uniforms controlling the blur size and effect strength. The blur size allows us to choose between using 5, 9, or 13 neighboring pixels for use in determining the blurred image's color values at each fragment. The effect strength controls the fraction of the blurred image's values to add to the original image's values.

CRT SHADER

The goal of this shader was to produce an effect similar to old-style videos. Notable features of old-school CRT displays are strong scanlines, where black horizontal stripes separate colored stripes, and slight chromatic aberration, which slightly displaces the RGB color channels from their normal positions. A glitching animation effect is also introduced that causes horizontal sections of the original image to slide to the left or right away from their normal positions for a moment.

Several modifications happen in the fragment shader. To begin with, a method for determining pseudo-randomly which sections of the image should glitch during animation had to be formulated. For this purpose, a random value is produced in Javascript every *n* seconds (via the *setInterval* function discussed above) and passed to the shader as a uniform float value. Then, the y coordinate of the image is modded by this random value, before being normalized to the range (0.0, 1.0). In this way, the random value produced via the Javascript code represents the "inverse frequency" of the glitch effect, since the lower the value is, the more sections ranging from 0.0 to 1.0 are produced in the given image.

Since this value represents frequency, the frequency setting in the GUI controls how the random value is produced in Javascript. The value is simply a random number in the range (0.0, 1.0) multiplied by the frequency value, with an arbitrary bias (in this case, 200) being added to it as well.

Now in each of these horizontal slices of the image which have a range of values from 0.0 to 1.0, a subsection of a specified width should be displaced to the left or right by a randomized value constricted to a specified maximum value. The width and maximum shift value are specified as uniforms in the GUI. The randomized shift value itself is produced in Javascript via the *setInterval* function and passed in as a uniform as well. If the value produced from the modulo expression falls within the range (0.5 – shiftWidth, 0.5 + shiftWidth), that section of the image gets displaced to the left or right by the random shift amount.

The angular setting in the GUI can cause the shift to take on a more triangular appearance by making the shift value greater towards the center of this subsection and lesser towards the edges.

Moving away from the glitch animation component of the shader, the colors of the CRT shader needed to be adjusted to better match the old-school display style. In these displays, green is much more pronounced, along with red being slightly more pronounced than blue. These colors are also slightly blurred along the x-axis, with green being much more blurred than the red channel.

In WebGL, this x-axis blurring is accomplished by simply looking to the left and right of the current fragments position in the *tDiffuse* texture and taking the average color value for a color channel. In this project, green is blurred to the left by 7 pixels and to the right by 12 pixels. Red is blurred only to the right by 5 pixels. Blue is left untouched.

Scanlines are simulated by finding an “opacity” for the base color of the image found after blurring and multiplying each color channel by this opacity. To obtain an opacity for this shader, we simply take the y-coordinate modulo a specified line width sent in as a uniform float value and chosen via the GUI. This is normalized to the range (0.0, 1.0). This is re-scaled and utilized such that values at the edges of this modulo operation are darker and appear as horizontal black bars, thus simulating scan lines.

DITHER SHADER

Dithering is an application of intentionally adding noise to an image to reduce quantization error. As an example, if you want to display an image in purely black and white, but you want to keep the level of detail somewhat consistent, you can apply noise in a structured way such that those 2 colors can give off the appearance of representing a whole gradient of colors.

In this project, ordered dithering is utilized by using a 4x4 Bayer matrix. This matrix is pre-computed as:

[	1.0	9.0	3.0	11.0	]
[	13.0	5.0	15.0	7.0	]
[	4.0	12.0	2.0	10.0	]
[	16.0	8.0	14.0	6.0	]

First, we obtain the average value between all 3 color channels for each fragment in WebGL representing the grayscale image. We then scale this value from the range (0.0, 1.0) to the new range (0.0, 16.0). From here, we determine the index of the Bayer matrix to compare the value to. This is computed as the x-coordinate modulo 4 and y-coordinate modulo 4. Then, if the scaled color value is < bayerMatrix[i][j], we set that fragment's color to black. Otherwise, we set that fragment's color to white.

As a note, this shader requires a WebGL 2 enabled browser. Otherwise, errors will be produced because there are int casts in the fragment shader code.

There are no settings to be altered by the GUI and, by extension, no uniforms sent in manually to the shader code.

GLITCH SHADER

This glitch shader is different from the type of glitch found in the CRT shader. A more appropriate name for this shader is chromatic aberration. This shader simply aims to displace the red, green, and blue color channels along different axes by a specified amount. When the animated flag is set for this shader, the amount each channel shifts by is randomly changed to be between 0.0 and the shift value set by the GUI and sent in as a uniform value.

The fragment shader code for this is simple. For each color channel, we shift our viewing texture coordinate along a different axis by the uniform float *shiftScale* amount. For our output fragment color, we then set each channel to the value viewed in the respective shifted texture. In the GUI, the *shiftScale* value is labeled as "Aberration."

SOBEL SHADER

This shader provides an example of the edge detection operation computed using the Sobel method. In the Sobel method of edge detection, an x-gradient kernel and y-gradient kernel are used in performing convolutions along the image. These gradients are:

$$\text{x-gradient} \quad \begin{bmatrix} -1.0 & 0.0 & 1.0 \\ -2.0 & 0.0 & 2.0 \\ -1.0 & 0.0 & 1.0 \end{bmatrix}$$

$$\text{y-gradient} \quad \begin{bmatrix} -1.0 & -2.0 & -1.0 \\ 0.0 & 0.0 & 0.0 \\ 1.0 & 2.0 & 1.0 \end{bmatrix}$$

In WebGL, convolutions can be performed by taking a weighted sum of the image with the kernel centered at the fragment coordinate. Padding values of 0.0 are used where there are no texture coordinates.

Once these convolutions are performed, a vector for each fragment is obtained consisting of an x-gradient and y-gradient. The magnitude of this vector is found via sqrt(pow(Gx, 2) + pow(Gy, 2)), which represents the magnitude of the edge with respect to both x and y gradients. The fragment's color is then set to (magnitude, magnitude, magnitude, 1.0), producing a black and white image displaying the magnitude of edges in the image.

There are no settings to be altered by the GUI and, by extension, no uniforms sent in manually to the shader code.