Custom Post-Processing WebGL Effects

Project Proposal

# 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.

# What I Would Like To Demonstrate

I would like to demonstrate various post-processing effects that I have personally coded up in WebGL, with settings that can be controlled via a GUI. The end results should have great performance due to the utilization of the GPU when programming in WebGL.

Some custom post-processing effects which I would like to demo are the Sobel operation (edge detection algorithm), a glitchy animated CRT-like effect, and chromatic aberration effect – all of which can be combined and customized via a GUI. More effects could be added and demoed as I find time to implement them.

# Resources I Plan to Use

I plan to only utilize Three.js’s built-in WebGL support with the post-processing effects pipeline being constructed from Three.js’s composer and passes objects. The shader programs will be coded up by me in WebGL. The dat.gui library will be utilized for GUI functionality in changing various settings about the shader effects.

All libraries which will be utilized will be included via script tags in the main html file with sources as external GitHub repositories (downloading the libraries manually will not be required). My own shader code will be provided as separate files, which are included via script tags in the main html file.

# Outline

The code will be structured such that the code that handles the GUI and the code that handles the setup and rendering of the Three.js scene are found in separate files as Javascript classes.

The GUI will be implemented using the dat.gui library. The GUI will have checkboxes to turn on any of the post-processing effects, however they will be passed through the composer in a fixed order; the user will not be able to specify running one effect before another, but multiple effects can be turned on at the same time. The GUI will also have various sliders and other settings controlling uniforms in each of the various effects’ shader programs which control how they operate.

Each effect that is implemented in the final project will be found in the “effects” directory as individual Javascript files. These Javascript files will simply contain a Javascript object with “uniforms”, “vertexShader”, and “fragmentShader” properties. The WebGL code for each effect will be found in the “vertexShader” and “fragmentShader” properties as an array of strings which are joined together by newlines. Passing these Javascript objects to the Three.js’s ShaderPass class will create a pass that can be sent to Three.js’s EffectComposer class and added into the rendering pipeline.

For effects that can be animated by changing the shader’s uniforms every *N* seconds, there will be a function that initializes shader update threads via Javascript’s **setInterval** function. These shader update threads will simply change the specified ShaderPass’s uniform values every *N* seconds.

Shader effects will be capable of being toggled on or off via the GUI. In these instances, the **initComposer** function will be called again which will re-create the rendering pipeline with the appropriate ShaderPass’s being added into the pipeline at each of their hard-coded entry points. The toggle simply defines whether the pass will be added at its specified insertion point or not. The computational cost of the **initComposer** function should not be visible to the user; toggles should be nearly seamless.

The basic scene which all of these post-processing effects will be demonstrated on will consist of 3 objects set inside a spherical panorama background. One object will be a basic sphere with a Phong material, one will be a Teapot model with a Lambert material, and one will be a trefoil torus knot with a Phong material. This scene setup may be subject to change. I might try to implement a few object-based shader effects while learning more about WebGL.

The post-processing effects guaranteed to be in the final demo will consist of a Bloom, CRT, Chromatic Aberration (will be labeled as Glitch), and Sobel effects.

The Bloom effect will utilize a fast Gaussian blur via linear sampling (modified from <https://github.com/Jam3/glsl-fast-gaussian-blur>) layered on top of the original image to produce a simple bloom effect.

The CRT effect will consist of displaying scan lines (with a specified lineWidth uniform) across the entire image utilizing the modulo operation in WebGL. There will also be simple red and green blurs in the x-direction to mimic old CRT display aberrations. The effect will also be able to be animated in which every *N* seconds, random scanlines are shifted by some random value the left or right of their original position. The animation will have a variety of settings to adjust how it operates.

The Chromatic Aberration (labeled as Glitch) effect will consist of shifting the red, green, and blue color channels of the input image pass in 3 different directions, scaled by a changeable uniform. This effect will be capable of being animated by choosing a random scaling amount within a specified range every *N* seconds.

The Sobel effect will consist of a fast Sobel convolutions to find the x and y gradients in the image. The final magnitude of these two gradients will be equivalent to the edge strength in the final output image. These edges will be displayed according to their magnitude values.

More post-processing effects and possibly object-based shader programs will be implemented and controllable via the GUI as I find time to add more, however, the post-processing effects listed above should provide plenty of opportunities for me to learn how to develop interesting WebGL shaders by themselves.