CSc 471 - Fall 2019
Fundamentals of Computer Rendering
Assignment 2
University of Victoria

Due: October 20, 2019 at 23:55. **Late assignments will not be accepted.**

1 Overview

Edge detection is an image processing technique that identifies regions where there is a significant change in brightness of the image. It provides with a way of detecting changes in the boundaries of objects and their topologies. This technique has applications in computer vision, image processing, image analysis, and image pattern recognition. In fact, a relatively straight-forward way of doing face detection involves edge detection.

One of the simplest ways of doing edge detection is by using the *Sobel operator*, which is designed to approximate the gradient of the image intensity at each pixel. The Sobel operator is implemented using a *convolution filter*, or *convolution kernel*. The idea behind applying a convolution kernel on an image is to replace the value of a pixel by using a matrix of weights and applying them to the nearby neighbours of the pixel. So for the case of a 3×3 matrix, we would select the 8 neighbouring pixels to the one we are trying to replace and then multiply each one (including the one we are trying to replace) by their corresponding weights in the matrix. The resulting value can then be computed by adding all the products together.

The convolution kernels for the Sobel operator are the following: 3×3 kernels:

$$\mathbf{S}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad \mathbf{S}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Let the result of applying $\mathbf{S}_x, \mathbf{S}_y$ be s_x, s_y respectively. Then we can approximate the gradient with the following equation:

$$g = \sqrt{s_x^2 + s_y^2}$$

The edge is then determined based on the value of g. If it is above a certain threshold, we highlight it in the resulting image. In Section 4 there are some ideas on how to implement this in OpenGL.

Your task is to write a single C++ program that does the following:

- Loads the provided mesh and renders it to the screen,

- Applies the Sobel operator to the rendered image, and
- Displays the results to the screen.

2 Specification

For full marks, your submissions **must** have the behaviour specified below. Failure to implement this functionality will result in marks being deducted.

- Load the provided mesh and render it. This can be done with the code shown in the labs.
- Apply a two-pass rendering scheme, where the Sobel operator is applied on the second pass.
- Provide a way to toggle between the original render and the Sobel operator.

You will be provided with an assignment bundle that will contain the following:

1. A full CMake setup for the assignment including code to generate the headers like the one seen in the lab,
2. an OBJ file for a torus knot,
3. a `main.cpp` file with a function to create the quad vertices with texture coordinates and an empty `main`,
4. a blank `assignment.hpp` header, and
5. a blank `UniformLocations.glsl` file.

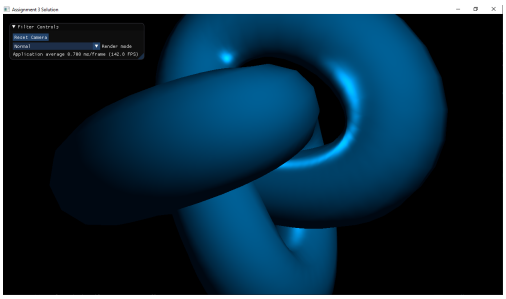
Your submission must contain the following:

1. One `main.cpp` containing all the appropriate C++ code to perform the specified tasks.
2. One `assignment.hpp` containing the definitions of any auxiliary data structures, functions, etc that you may require. Note that you may leave this file blank.
3. Any vertex and fragment shaders that you require, along with the `UniformLocations.glsl` file (you may leave this blank if you wish). Note that these *must* observe the file extension convention specified in the labs.
4. In the submission comments, document how to switch between the operator and the original render.

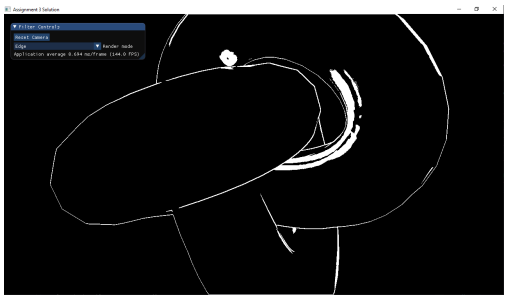
Please note that the full assignment bundle does not need to be provided in your submission, only the individual files listed above.

3 Evaluation

Your code must compile and run correctly in ECS 354. If your code does not compile or crashes at runtime, you will receive a mark of zero. This assignment is worth 8% of your final grade and will be marked out of 10.



(a) The shaded model



(b) The Sobel operator with threshold of 0.01

Figure 1: Sample output

4 Recommendations

The process for doing the two pass render is identical to the one shown in the labs. The render buffer can be set in the same way, just ensure that you use `GL_NEAREST` for the min and mag filters as we do not want any interpolation for this.

The sampling of the texture for the kernels can be done by using the position of the fragment as opposed to the actual texture coordinate. Since the texture that we bound to the framebuffer is the same size as the screen we are rendering to, the coordinates of the fragments map precisely with the coordinates of the pixels on the screen. This means that the vertex shader need not forward any kind of data to the fragment shader. The position of the fragment can be obtained with `gl_FragCoord`. Remember that the fragments have 3 coordinates, where the first two represent the position of the fragment and the third is the depth.

Once you have the position of the fragment, you can then use the `texelFetchOffset` function from GLSL to offset the position of the fragment and retrieve the individual pixels. Alternatively you can do the offset calculation by hand and retrieve the colour yourself. Once this is done, the final step is to compute the luma value for that particular pixel. The luma value can be computed with the following formula:

$$l = c \cdot (0.2126, 0.7152, 0.0722)$$

Where $\cdot$ is the dot product. With the luma values for each pixel, you can apply the previously discussed formulas and obtain g. Example output can be seen in Figure 1.