CPSC 478/578: Computer Graphics **Jan. 15, 2020**
Spring 2020, Assignment 2

Due: Wednesday, February 12, 2020 at 11:59 PM

Homework 2: Transformations and WebGL

For written portions of this assignment, turn in a PDF of your writeup. You can prepare the answer in Microsoft Word or LaTeX, but it is equally fine if the PDF just contains scans of your hand-written answers.

1 Transformations

Read Chapter 6 of Marschner and Shirley, and Chapter 4.1-4.5 of Ganovelli.

1.1 Scaling Matrices

In general, the 2D scaling matrix

$$\Sigma = \begin{bmatrix} \sigma_x & 0 \\ 0 & \sigma_y \end{bmatrix} \tag{1}$$

contains two eigenvalues, $\lambda_x = \sigma_x$ and $\lambda_y = \sigma_y$. It also have two eigenvectors:

$$\mathbf{q}_x = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad \text{(along the } x\text{-axis)}$$

$$\mathbf{q}_y = \begin{bmatrix} 0 \\ 1 \end{bmatrix}. \qquad \text{(along the } y\text{-axis)}$$

Thus, you can only really stretch things uniformly along the $x = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$ or $y = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ axes. What if you want to stretch things diagonally along the $\begin{bmatrix} 1 \\ 1 \end{bmatrix}$ axis?

Derive a general scaling matrix, parameterized by a single variable, σ_{diagonal}, that accomplishes this (2 points). Include a written description of the geometric reasoning you used to arrive at this expression. The explanation is more important than the final matrix expression (8 points).

1.2 Rotation Matrices

In general, any 3D rotation matrix only has a single real-valued eigenpair, i.e. an eigenvector and its corresponding eigenvalue $(\lambda, \mathbf{q})$. The eigenvalue is always 1.

Keeping in mind the definition of an eigenpair, $\mathbf{Aq} = \lambda\mathbf{q}$:

- Explain what the vector $\mathbf{q}$ represents geometrically. (3 points)
- Explain why it makes sense that $\mathbf{q}$ is the eigenvector. (3 points)
- Explain why if $\lambda \neq 1$, even if $\mathbf{q}$ remained exactly the same, the matrix would no longer match our intuitive notion of a "rotation". (4 points)

1.3 Rotation Matrices Again (578 only)

One way to check if some 2×2 matrix $\mathbf{A}$ *might* be a rotation matrix is to check if $\det(\mathbf{A}) = 1$. Unfortunately, this is a necessary, but not sufficient, condition.

Do the following:

- Using the Frobenius norm, $\|\mathbf{A}\|_F$, derive a second test that guarantees that the matrix is a rotation (1 point), and state your reasoning for this test (4 points).
- Are these two conditions sufficient in 3D? Show why this is or is not true. (5 points)

2 Geometry in WebGL

2.1 Draw Your Own Shape

Let's draw an octahedron (Fig. 1). The starter code provided is from Chapter 3 of the Ganovelli book, and already shows how to draw a cube, cone, cylinder, and ribbon.

Your shape should be centered about the origin, and the edge lengths of all the sides should be 2. Make a new file `octahedron.js` that draws this object. You will also need to modify `renderingprimitives.js` so that it knows which LookAt matrix to use. You should be able to re-use the same matrix as the Cube.

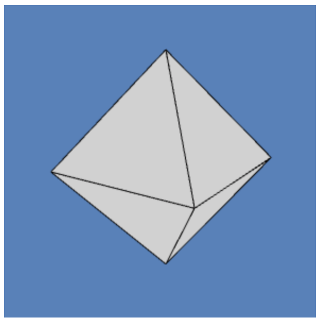


Figure 1: Octahedron in WebGL.

2.2 Set Your Own Colors

Modify the the files `primitives_cube.html`, `cube.js` and `renderingprimitives.js` files so that they instead draw on a black background, and each vertical edge has a different color: red, green, blue, and yellow. Three screenshots of the cube as it rotates is in Fig. 2.

A lot of this question is just slugging through API documentation to call the right arcane functions in the correct order (just like in real software development). The code from question 1.3 from Homework 1 is a good reference here, as are Chapter 2.3 from Ganovelli and this tutorial.

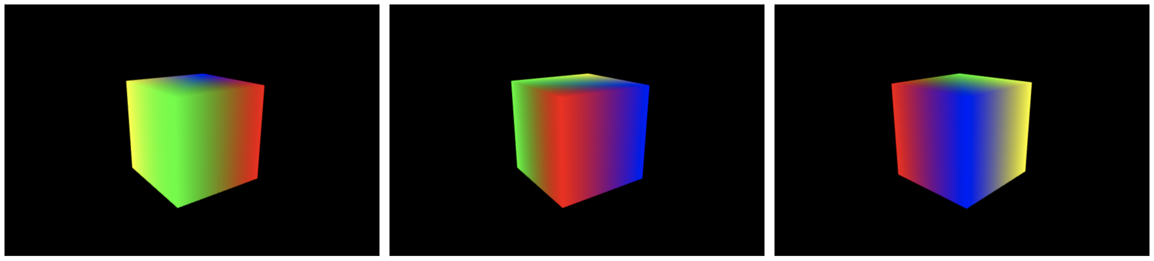


Figure 2: Recolored cube as it rotates.

2.3 Draw A Sphere

Modify your octahedron drawing routine to draw a sphere. Name your modified files primitives `sphere.html` and `sphere.js`. Obtain the vertices by projecting the vertices of the octahedron to a sphere. First, generate an octahedron whose triangular faces have been split n number of times, and then move these vertices radially until they lie on the surface of a sphere (Fig. 3). Your HTML script should take n as input.

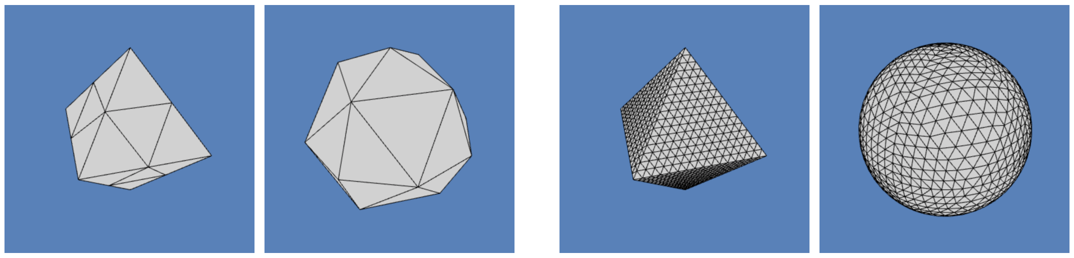


Figure 3: **Left pair:** An octahedron subdivided once, and then projected to a sphere. **Right pair:** An octahedron subdivided four times, and. then projected to a sphere.

2.4 Loop Subdivision (578 Only)

Read 3.7.5 in Ganovelli and implement the *Loop* subdivision scheme on your octahedron (see Fig. 4. The HTML should accept the number of desired subdivisions.

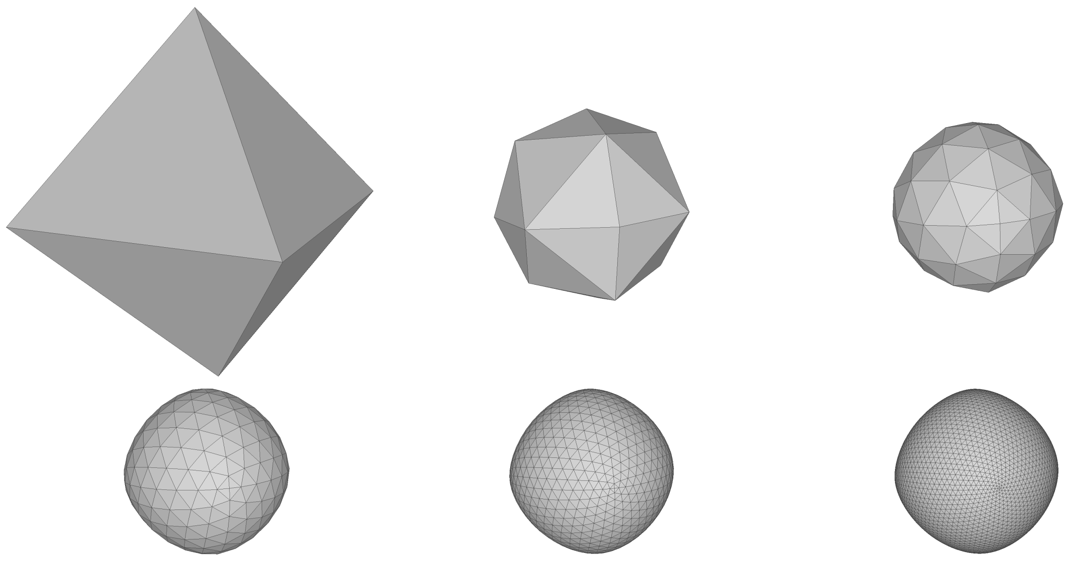


Figure 4: Starting from the upper left, an initial octahedron, successively Loop subdivided five times. Note that it shrinks quite a bit during the first few subdivisions.

3 Introduction to Fragment Shaders

3.1 The Mandelbrot Set

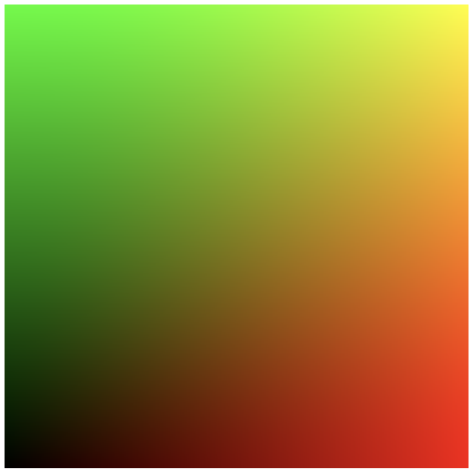
Implement the Mandelbrot Set, as described in class by modifying the fragment shader in `mandelbrot.html`. For your convenience, the Mandelbrot set computation is outlined in Figure 5. GLSL will fight you if you try to implement it exactly as listed in Fig. 5, so it is up to you to determine what GLSL wants.

The starter code generates Figure 6a, but when you are done, it should generate Figure 6b. This was generated with maxRadius= 2.0, maxIterations=100. The color inside the set was set to red, and the colors outside were set to $1 - \frac{1}{\text{iter}}$.

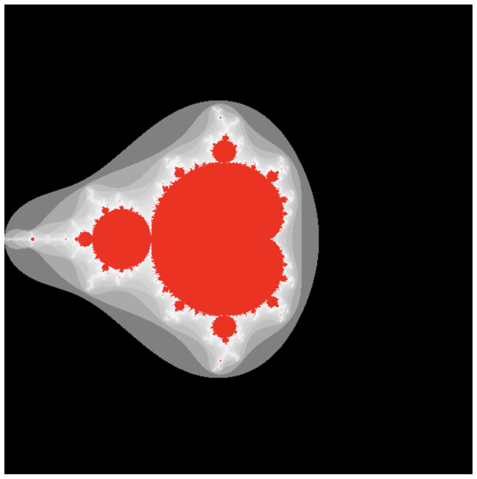
```
function INSIDEMANDELBROTSET($\mathbf{q}$)          Q is the point
    $\mathbf{c} = \mathbf{q}$
    while $|\mathbf{q}| \leq$ maxRadius and $i <$ maxIterations do
        $\mathbf{q} = \mathbf{q}^2 + \mathbf{c}$
        $i = i + 1;$
    end while
    if $i ==$ maxIterations then
        return False
    end if
    return True
end function
```

Q = a + bi
Q^2 = a^2 + 2abi + b^2i^2
Q^2 = a^2 - b^2 + 2abi
Q^2 + C = a^2 - b^2 + 2abi + a + bi
Q^2 + C = (a^2 - b^2 + a, 2ab + b)

Figure 5: Pseudo-code for Mandelbrot Set computation



(a) Mandelbrot Set, Starter Code



(b) Mandelbrot Set, All Working

3.2 Factored Polynomials

While the Mandelbrot set uses the polynomial $f(\mathbf{q}) = \mathbf{q} = \mathbf{q}^2 + \mathbf{c}$, you can set this equation to *whatever you want*, and get all sorts of wild shapes Let's explore other possibilities by setting it to a factored 5th-order polynomial,

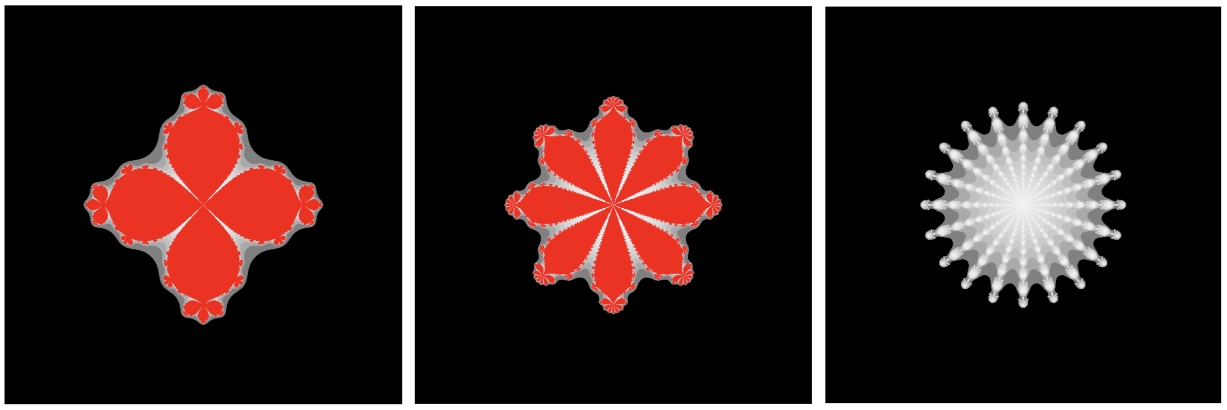
$$f(\mathbf{q}) = \mathbf{q}(\mathbf{q} - \mathbf{r}_0)(\mathbf{q} - \mathbf{r}_1)(\mathbf{q} - \mathbf{r}_2)(\mathbf{q} - \mathbf{r}_3). \tag{2}$$

The first root is always set to zero (the leading $\mathbf{q}$ in $f(\mathbf{q})$), and the real and imaginary components of each following kth root location is set to:

$$\mathrm{Re}(\mathbf{r}_k) = 1.01 \cos\left(2\pi \frac{k}{n}\right) \tag{3}$$

$$\mathrm{Im}(\mathbf{r}_k) = 1.01 \sin\left(2\pi \frac{k}{n}\right). \tag{4}$$

Since we are looking at a 5th order polynomial, $n = 4$ in this case. Get this factored polynomial working for arbitrary n. The fractals for several values of n are shown in Fig. 7.



(a) $n = 4$ (b) $n = 8$ (c) $n = 20$

Figure 7: Julia sets of several factored polynomials.

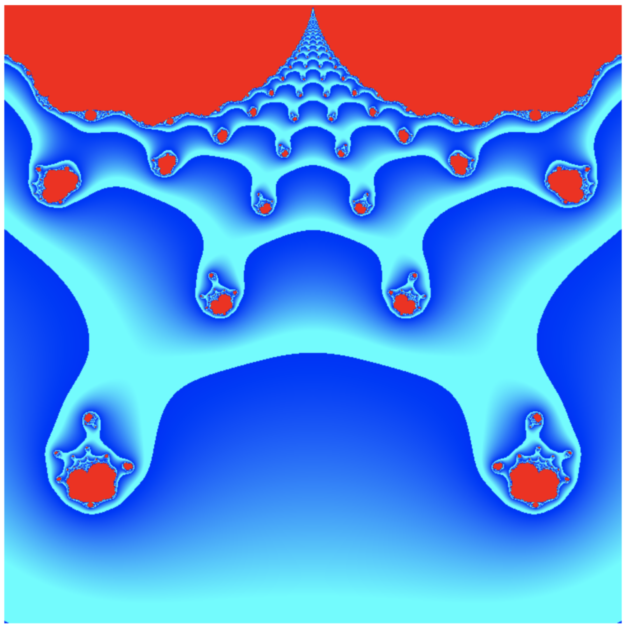


Figure 8: Professor Kim's wild shape. Make something wilder.

3.3 Make A Wild Shape

You can put the roots of the polynomial *anywhere* and get *all sorts of wild shapes.* Modify your shader to make something great. A few ideas for generating a wild shape:

- Manipulate the root locations based on something other than a circle.
- Modify the function you are computing. What happens when you sum multiple polynomials? What about a rational? What happens when you start adding arbitrary constants?
- Zoom in to different parts of the fractal to find interesting structures.
- Change the color scheme.
- Instead of using the number of iterations to set the color, use the accumulated $\mathbf{q}$, i.e. keep a running sum of $\|\mathbf{q}\|_2$ each iteration.

Back in the 1980s and 1990s, people came up with *all sorts* of different ways to make shapes. Now it's your turn. Take a look at Fig. 8 to see the shape I came up with after about 20 minutes of fussing with the shader; I'm sure you can find something much wilder. Be careful about adding too many roots to your polynomial though. The computation can *easily* get heavyweight enough to bring down your browser.

Turn in your final shader, and share a screenshot of your shape with the rest of the class by posting it to the designated discussion forum on Canvas.

4 Formatting Instructions

- Turn in a ZIP file named `LastName_FirstName_Assignment_2.zip`.
- Unzipping produces subdirectories for each question: `q1.1`, `q1.2`, `q1.3`, etc. For written questions, the subdirectory should contain the PDF.
- If necessary, each subdirectory can contain a `readme.txt` that answers any necessary questions, or explains how some pieces of your answer work while others do not.
- Follow problem-specific formatting instructions described above (input parameters, output names, etc.).

5 Points Breakdown

The assignment will be graded *on a Zoo lab machine*. Make sure your JavaScript runs fine on Google Chrome there. There are 90 possible points for 478 and 110 possible points for 578.

- **(10 points)** Diagonal scaling matrix question.
- **(10 points)** Rotation matrix question.
- **(10 points)** Second rotation matrix question (578 only).
- **(10 points)** Get octahedron working.
- **(10 points)** Get recolored cube working.
- **(10 points)** Get sphere rendering working.
- **(10 points)** Implement Loop subdivision for an octahedron (578 only).
- **(10 points)** Get the Mandelbrot set fragment shader working.
- **(10 points)** Get the factored polynomial fragment shader working. The version you turn in should be set to $n = 10$.
- **(10 points)** Turn in the code for your "wild shape" Julia set. You must also post to the discussion forum in order to get credit.
- **(8 points)** Follow the formatting instructions correctly.
- **(2 points)** At the top level directory, include a `HOURS.txt` that lists approximately how many hours you spent on this assignment. There is no right answer; we will only be analyzing this data anonymously.