# Homework 1: Math and JavaScript

CS 440 Computer Graphics
Habib University
Fall 2019

Due: 18h on Wednesday, 11 September

Each problem below specifies the names of the files you have to submit for it. Please make sure your submitted files have the indicated names. Any files in your GitHub repository with these names at the time of the deadline will be considered as your submission.

## 1. Geometry trumps JS

We will be using WebGL to render 3D geometry in the browser. Programming for WebGL is done in JavaScript which is not natively aware of geometric types, e.g. vector or matrix. The file, `MV.js`, in the `Common` folder in the `Code` section on the website of Module I's book defines abstractions that allow programs to be written in terms of geometric entities, e.g. *vectors*, instead of JavaScript arrays. It also defines a function to convert these abstract data types to the required JavaScript types when needed. Using this file, our code can be in terms of constructs that are more natural to our application domain

Go over the file `MV.js` in order to get familiar with the types that it provides and their related operations. You need not spend too much time understanding the body of the function, as long as you can infer, e.g. from the name, what each function does. For now, you may skip functions in the following sections in the file.

- Basic Transformation Matrix Generators,
- ModelView Matrix Generators, and
- Projection Matrix Generators.

Wherever possible in your code for subsequent problems, prefer using geometric primitives and operations from `MV.js` over those provided by JavaScript.

## 2. Taking it for a Spin

In this problem we will utilize some of the functionality provided by `MV.js`. Write a JavaScript script which asks the user for a dimension between 2 and 4 inclusive, and then asks the user to enter 2 vectors of that dimension. It then presents the user a menu with the following choices.

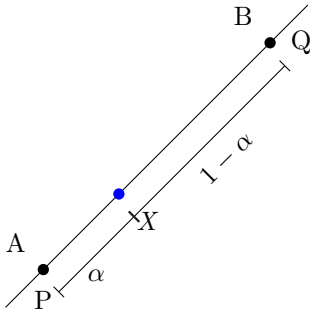
- Tell whether the vectors are equal
- Show the lengths of the vectors
- Show the normalized vectors
- Show the sum of the vectors
- Show the difference of the vectors
- Show the dot product of the vectors
- Show the cross product of the vectors
- Exit

The program should repeat until the user chooses to exit.

Use JavaScript and `MV.js` only. Do not use any higher level library. You may have to look up how to perform user I/O in JavaScript. Output typically uses `alert()` or `console.log()`. Make sure to find out how to invoke your browser's *console* as that will often be your only means to debug your program.

Write your program in 2 files–`operations.html` and `operations.js`–where the program logic resides in `operations.js` and the HTML file includes `operations.js` and `MV.js`. Opening the HTML file in a browser should launch the script.
Files: operations.html, operations.js

## 3. Mapping and Linear Interpolation




In the figure on the left, $X$ lies on the line segment $PQ$ such that

$$|PX| : |XQ| = \alpha : (1 - \alpha) \ , \ 0 \leq \alpha \leq 1,$$

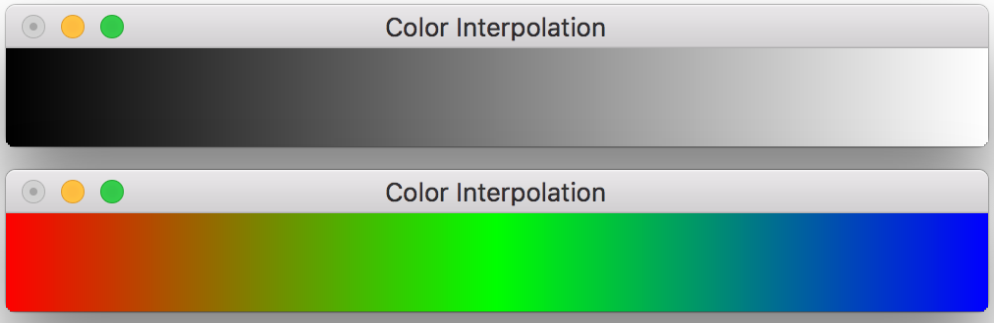
which leads to

$$X = \alpha Q + (1 - \alpha)P.$$

$X$ is thus said to be an *affine combination* of $P$ and $Q$.
Imagine a mapping from the range $PQ$ to a new range $AB$. Write a function `map_point` that takes $P$, $Q$, $A$, $B$, and $X$ as parameters and uses the `mix` function from the file `MV.js` to compute and return the mapping of $X$ to the range $AB$. $P$, $Q$, and $X$ are of the same type which may be a scalar or any of the vectors defined in `MV.js`. $A$ and $B$ are of the same type which may be a scalar or any of the vectors defined in `MV.js`. Your function should return a value of the same type as $A$ and $B$. This function will be useful in subsequent assignments including this one.

## 4. Color Interpolation

.




WebGL renders in a *canvas* on an HTML page. Individual pixels in the canvas have *pixel coordinates* which start at 0 on the left and increase rightward. Pixel coordinates are always whole numbers. WebGL imposes a coordinate system on the canvas in which the origin is at the middle of the canvas, the left and right extremes of the canvas have $x$ values of -1 and 1 respectively, and the bottom and top extremes of the canvas have $y$ values of -1 and 1 respectively.

Imagine interpolating colors as shown above using WebGL. The width of the canvas is equal to $W$ pixels. In the figure on the top, the left extreme of the canvas corresponds to black (*vec3(0,0,0)*) and the rightmost to white (*vec3(1,1,1)*). The pixels between the horizontal extremes are different shades of gray.

In the figure on the bottom, the left extreme of the canvas corresponds to red (*vec3(1,0,0)*), the center to green (*vec3(0,1,0)*), and the right extreme to blue (*vec3(0,0,1)*). The pixels between the left and the center have a color value interpolated between red and green. The pixels between the center and the right have a color value interpolated between green and blue.

Write a script that inputs $W$, the width of the canvas, and a valid pixel coordinate, i.e. $\in [0, W)$, and uses your function from Problem 3 to output

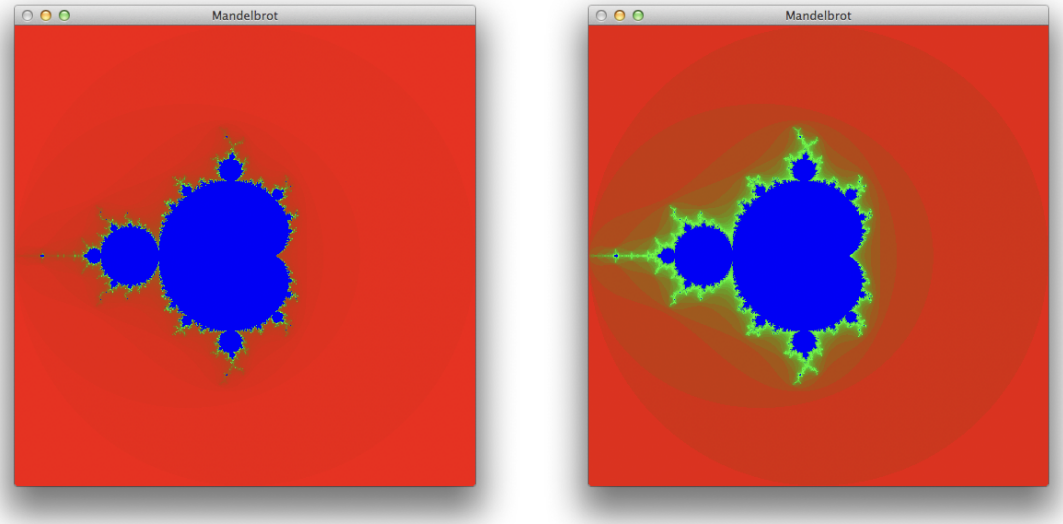
(a) the corresponding WebGL coordinate (assume $y$ to be 0),

(b) the value of the corresponding shade of gray, and

(c) the value of the corresponding color.

Files: interpolate.html, interpolate.js

## 5. The Mandelbrot Set




Imagine visualizing the Mandelbrot set as shown above. The set consists of complex numbers lying on the circle of radius 2 at the origin in the complex plane. The visualizations above show the relevant portion of the complex plane; the top left corner corresponds to $-2+2i$ and the bottom right to $2-2i$. Each pixel in the visualization corresponds to a complex number. The colors assigned to the pixels in the visualizations are based on certain properties of the corresponding complex numbers but they can be ignored for this problem. The canvas in both cases has the same width and height, $L$. As always, WebGL imposes its own coordinate system on the canvas as described in Problem 4.

Write a script that inputs $L$ and a pixel position defined by its row and column number. In the context of the canvas the origin is at the top left, the column number increases from left to right and the row number increases from top to bottom. Your script should then use your function from Problem 3 to output

(a) the complex number corresponding to the input pixel, and

(b) the corresponding WebGL coordinate.

Files: mandelbrot.html, mandelbrot.js