Project Diary

# Weeks 1 & 2 (11/1/2016 – 24/1/2016)

* Prototype code was refactored into MVC using Alex Netkachov’s (2015) template and old (non-MVC) code achieved into “prototype” directory
* Included a custom event class that allows listeners to subscribe to an event and receive notifications with data when the events fire: other implementations of events involved sending notifications through the DOM, which I thought was messy and not true to MVC
* MVC required me to start using JavaScript objects/class, which use a prototype model
* Original DFD diagram had to be changed:
  + All rendering is now done in the view, as it regards the presentation of data and should have no effect on the representation of data in the model
  + Following on from above, a custom colour class was created, allowing me to define colours how I want, independently of the rendering framework, which should be isolated to the view

# Week 3 (25/1/2016 – 31/1/2016)

* Prototype is now completely refactored into an MVC design
* Began looking into testing, allowing me to start TDD and unit tests for existing code
* Configured WebStorm to use Karma to run QUnit tests (JetBrains, 2015), this took a day and was a lot of faff but nothing important to conclude
* Tests written for some existing code, mostly easy to test functions such as the utils and model
* Some functions are noted as being untestable, mostly due to being unable to simulate file inputs by loading a file from local file space: I intend to either modify them to become testable or to define manual tests that can be used instead
* Colour quantisation function in the model is the first to be written using TDD (ish!)

# Week 4 (1/2/2016 – 7/2/2016)

* Ran into confusion with the values in my LUT and during the conversion between 24-bit and 16-bit
* Resolved confusion: multiply 16-bit value to get 24-bit value
* I initially thought it was a factor of 16 (16 \* 16 = 256) and while this maps 0 to 0, and 1 to 16, 15 (16-bit max value) maps to 240
* I knew the 24-bit range would be divided into 16 ranges (256 / 16 = 16) but I believed these ranges would have a size of 16
* Instead they have a range of 17: 255 / 17 = 15 divisions, plus 1 from 0 to 17, totalling 16 divisions with a range of 17
* Confusion resolved, new code written, and tested

# Week 5 (8/2/2016 – 14/2/2016)

* Noted that the small number of dominant colours were overpowering the majority, making them difficult to see
* Took natural logarithm of values to prevent smaller values from being dwarfed
* It worked, however when a bin had only 1 entry, the size of the plot would be 0 (ln(1) = 0) and the plot is drawn very big for some reason
* These 1 value colours will have to be either ignored or increased by 1. But they are so small that this will hardly be noticed
* I decided to use ln(x+1), which passes through the origin, ensuring that y is only 0 when the value is 0. It does skew values slightly, but I thought this was better than omitting all results with a value of 1
* If a sphere radius is below the minimum, do you omit it or set it to the minimum radius?

# Week 6 (15/2/2016 – 21/2/2016)

* Used simplified colour transformations (found at <http://www.easyrgb.com/index.php?X=MATH&H=07#text7>) to convert from RGB to Lab
* Refactored ColorRGB class to incorporate both RGB and Lab values into a single object
* New Color class also includes the code to convert between colours spaces
* Decided not to port code for colour quantisation into Color class too, because the colours should be stored in max resolution (24-bit) and only quantised when they are to be displayed
* Note: Look into those things that Richard mentioned in the graphics lecture for detecting which parts of code are taking the longest
* Wrote tests for the colour space transformations using test values obtained from same site as formulae
* Is this simple formula accurate enough compared to matrix transformations?
* The owners of the website assure me they are equivalent but I may conduct further tests to assure myself of this
* But for now I will trust their word and continue with development

# Week 7 (22/2/2016 – 28/2/2016)

* Implemented CIE-L\*a\*b\* to sRGB conversions
* More extensive tests are necessary for the conversions
* Began rendering Lab colour space cube, realised that fragment shaders are now necessary are colours are interpolated incorrectly by WebGL
* Rendered RGB cube using shaders (previously done with a series of JS functions) and is now much neater
* Lab cube also rendered using shaders
* However transformations had to be written in GLSL, which is nearly impossible to test, but the computational and structural advantages outweigh this disadvantage
* Changed RGB colour quantisation method to use floating point arithmetic rather than LUTs, which were overkill
* This is opens up the opportunity to perform quantisation within the Color class
* Colour quantisation inside Color class works well and removes complexity in model
* However lab values are not being plotted correctly
* Tests have been written for easily tested functions, but other things are harder to test with unit tests, namely the rendering
* Renderings still require to be tested by hand
* Found a silly bug that was colouring Lab colours incorrectly (while they were in the correct locations): I was using the rgb shaders, not the lab shaders
* Reformatted code so colour space changes work: included adding a static variable to Color
* Refactored Color class so that variables and methods can be private, giving the code more of a OO feel, which I am more familiar and comfortable with
* It also makes testing easier and helps assure that the class is being used correctly

# Week 8 (29/2/2016 – 6/3/2016)

* Antony suggested loosing the cube for the CIE-Lab rendering as the axes are adequate, appropriate, and the cube misleading
* He also suggested the idea of having two independent renderings, side by side, one in RGB and the other Lab
* Limited the OrbitControls’ controls to the div containing the canvas
  + This firstly allows the user to use slide controls (i.e. brightness etc.) without moving the rendering
  + And secondly allows two renderings to be shown side by side
* Implemented UI controls for colour tools in the form of sliders (note: these will not work with IE9 and below)
* Added an HSL field to the Color object, allowing which is to be used to adjust the brightness and saturation of colours
* Brightness can now be changed using the sliders and the effects seen in the histogram
* Results take a number of seconds to calculate, so efficiency and optimisation will certainly need to be addressed
* This website (<http://colorizer.org/>) shows that changing L in Lab changes L in HSL pretty independently, but changing L in HSL changes both a and b in Lab as well as L: this is unexpected and also observed in my visualisation… worth investigation of the transitivity of these transformations!
* I am currently guessing that the range of L in HSL is 0 to 1, and this assumption gives expected results

# Week 9 (7/3/2016 – 13/3/2016)

* Little progress made
* Discussed how to continue for the last few weeks of development
* Prioritise usability and being able to see results and make conclusions from the visualisation
* Make it so that the image’s colours change as those in the histogram do
* Also implement last two colour controls
* Finally, if time, make the histogram quicker
* Secondly, adjusting brightness results in strange artefacts (extreme colours) to appear
* Are these a result of a poor brightness algorithm? Find out by changing the colour of the image
* Can I deal with clippings better?

# Week 10 (14/3/2016 – 20/3/2016)

* The image now changes colour with the histogram
* Increasing brightness almost looks like increasing saturation: very bright colours emerge/colours disappear into black
* Saturation adjustment now also works, using HSL in the same way to brightness
* Noticed that I wasn’t ensuring the S and L values were inside the 0 to 1 range when applying the adjustment: clipping to these ranges appears to give more sensible colour changes
* Contrast adjustments added using the algorithm found here: <http://www.dfstudios.co.uk/articles/programming/image-programming-algorithms/image-processing-algorithms-part-5-contrast-adjustment/>
* All colour controls now added: only question now is how to handle clipping
* Another useful thing to do would be to plot all the colours in a colour space to view it’s shape
* I decided it would be easier to simple find an image that contained all RGB colours
* To achieve this I had to fix my webpage so that it didn’t clip the input images to 200x200 pixels
* This worked well and I am able to see the shape of sRGB in Lab, and it is very interesting: not a double-ended cone but more of a squashed and off-centre rhombus
* However in the process of doing this I discovered that images larger than 200x200 took a long time to extract colours from, considerably longer than I had expected
* It is therefore even more important that I now look at how I can make this more efficient

# Week 11 (21/3/2016 – 27/3/2016)

* Began using Google Chrome’s profiler to determine which parts of my program are taking the longest
* I started by trying to optimise the initial uploading of an image and extracting colours
* I soon discovered that it was the getImageData that was taking up the most processor time: 24,000 to 25,000 ms for a 350x235 image
* I tried sampling a quarter of all pixels rather than every one: while this brought the computation down to about 6,500ms it wasn’t ideal and resulted in blocky images
* After a little Googling I realised that it was the getImageData call on the DOM object that was taking the longest, so I decided to get all the pixels in one getImageData call and iterate over the resulting array
* This brought the computation down to a blinding 66.1ms
* Hopefully I can do a similar operation for the putImageData method call
* I will also look into what processes are the most expensive when adjusting an image’s colour