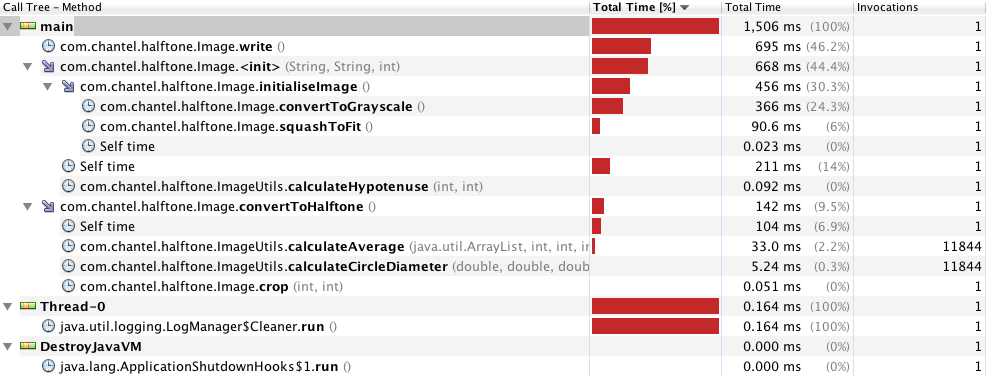
**How Code Performance Was Measured**

Report

Code performance was measured using the VisualVM profiling tool. This was achieved by simplifying the code originally written to perform half-toning on an image to run through Eclipse (the development environment) rather than command line. Each test was run with a hardcoded image and a halftone dot size of 10 (meaning that the image was drawn using dots of a maximum size of 10). In order to profile the code effectively, a breakpoint was placed on the start of the code (the first line as soon as the application began running), and then the application was run in debug mode. While the application’s execution was sitting on this line, VisualVM was launched and CPU profiling was turned on. The application was then resumed, ensuring that the application ran from start to finish through the profiling tool. At the end of the profiling process, a snapshot as seen below was obtained:

Each snapshot, as seen above, contained the total running time of each of the methods within the application and how they contributed to the overall running time of the application. Several different types of image files were input into the application and ran using the profiling tool in order to obtain a realistic measurement of code performance.

**Input Data for Profiling**

The input data used for profiling was a 3 : 2 aspect ratio 1.2 megapixel image, a 4 : 3 aspect ratio 1.2 megapixel image, and a 4 : 3 aspect ratio 13 megapixel image.

The two 1.2 megapixel images were selected for use in testing because the target device (Nexus 7) has a 1.2 megapixel front facing camera. The aspect ratios of 3 : 2 and 4 : 3 were chosen because these two aspect ratios are most commonly used for outputting images from any image capturing device. Thus these two images were selected in an attempt to emulate the “realistic” types of input that will be given to the application when run on the Nexus 7.

For good measure, an additional two 13 megapixel image with aspect ratio 4 : 3 was included as the highest megapixel camera out on the market attached to an Android smartphone is a 13 megapixel camera. It was decided to test this image because it is quite possible that another user may send a photo of themselves from their phone to a user who has the Ye Olde Times app, and there is no guarantee what type of phone that user might have. Thus, in order to cover the worst possible case in which the user sending the photo has a camera with the highest megapixel image output on the market, it was decided to include this image.

**The Changes Made and Why They Were Made**

It was decided to make changes to the convertToGrayscale() method in the Image class as it can be seen from the above snapshot that this method was always the method which took the longest (disregarding the write method as it did nothing more than use the Java ImageIO.write() method which is necessary to output the image), irrespective of the size and aspect ratio of image being fed into the application. It was decided to change the way in which the grayscale conversion was performed as in the original version of the code, the image passed in was being converted to grayscale by using complicated luminosity formula weightings which aim to get the “true” grey colour out of a single pixel. As multiplication takes more CPU clock cycles than addition, it was decided to change the way that this formula was being calculated to use no multiplication and instead, addition of the three colours along with a single division operation (to divide by the number of colours) which ideally should be more efficient. It was also decided to remove the hash map that was being used to store all of the grey colours to avoid having to create duplicate objects where there were many pixels with the same grey colour. It was decided to make this change because it was deduced that it took a large amount of time to create the hash map in memory and then perform reads and writes to memory every time a grey colour was created. The original code is provided below to demonstrate how grayscale conversion was performed prior to any modifications being made:

**public** **void** convertToGrayscale()

Report

{

// Loop over each pixel in the image

**for** (**int** i = 0; i < height; i++)

{

pixels.add(**new** ArrayList<Integer>());

**for** (**int** j = 0; j < width; j++)

{

// Obtain the pixel's red blue and green values

Color pixelRGB = **new** Color(image.getRGB(j,i));

// Obtain grey colour through luminocity formula weightings

**int** grey =

(**int**)((pixelRGB.getRed()\*0.2126)+(pixelRGB.getGreen()\*0.7152)+(pixelRGB.getBlue()\*0.0722));

/\* If the grey color has not been encountered before, put it into the

\* greyVarients hashmap so it can be reused if the same

\* grey is encountered again

\*/

Color greyColor;

**if**(greyVarients.get(grey)!= **null**)

greyColor = greyVarients.get(grey);

**else**

{

greyColor = **new** Color(grey, grey, grey);

greyVarients.put(grey, greyColor);

}

// Update image to greyscale

image.setRGB(j, i, greyColor.getRGB());

// Store pixel data so we can later process averages of cells without overlapping circles from the halftone effect interfering

pixels.get(i).add(grey);

}

}

}

It can be seen from the above code that the integer variable “grey” was being obtained by multiplying each red, green and blue component of a single pixel by separate weightings and then adding those values together in order to get the average grey colour. Additionally, there was a hash map being used in order to store all of the grey Color objects the first time that a certain type of grey was encountered such that there would be no need to recreate the grey colour every time the same grey value was encountered later on in the image.

The improved code is provided below, which shows modifications which perform a simpler method of averaging red, green and blue values to find the appropriate grey colour and the removal of the hash map for storing the grey Color objects.

**public** **void** convertToGrayscale()

{

// Loop over each pixel in the image

**for** (**int** i = 0; i < height; i++)

{

pixels.add(**new** ArrayList<Integer>());

**for** (**int** j = 0; j < width; j++)

{

// Obtain the pixel's red blue and green values

Color pixelRGB = **new** Color(image.getRGB(j,i));

// Obtain grey colour through luminocity formula weightings

**int** grey =

(**int**)(((pixelRGB.getRed())+(pixelRGB.getGreen())+(pixelRGB.getBlue()))/3);



Report

// Update image to greyscale

image.setRGB(j, i, **new** Color(grey, grey, grey).getRGB());

// Store pixel data so we can later process averages of cells without

overlapping circles from the halftone effect interfering

pixels.get(i).add(grey);

}

}

}

It can be seen in the above modified code that the average grey colour is now being evaluated by simply adding all of the red, green and blue values for the pixel and then dividing them by 3. Then, instead of creating a new instance of the grey colour inside of a variable and then evaluating whether the grey was in the hash map or not, the hash map has been completely removed. The variable holding the “Color” object for the grey colour was also removed as there was no longer any need to keep it once the hash map was removed. Instead, the Color object is created when it is passed into the setRGB method which also saves time.

**Implications of these Changes**

The implications of making these changes was that firstly, the change to the calculation of the grey colour for each pixel made the image look less well blended (the gradients of grey throughout the image showed sharper transitions than before). This was because the calculation of the grey colour was a lot less accurate when calculated as an addition of all three red green and blue values divided by the number of colours. Below are two images, one where half-toning was performed prior to making the changes to how the grey colour was calculated and the other where half-toning was performed after making the changes. It can be seen in the area encircled in green that there is a visible difference in the harshness of transition from one grey level to another in terms of the gradient in that portion of the image.

**Before Modification After Moficiation**

However an additional implication was that the application’s ability to efficiently perform garbage collection was now being relied upon because the Color objects were now being instantiated every single time the setRGB method was being called. However, these trade-offs were determined to be reasonable as the improvement in speed as a result of making these changes was relatively significant.

**The Results**

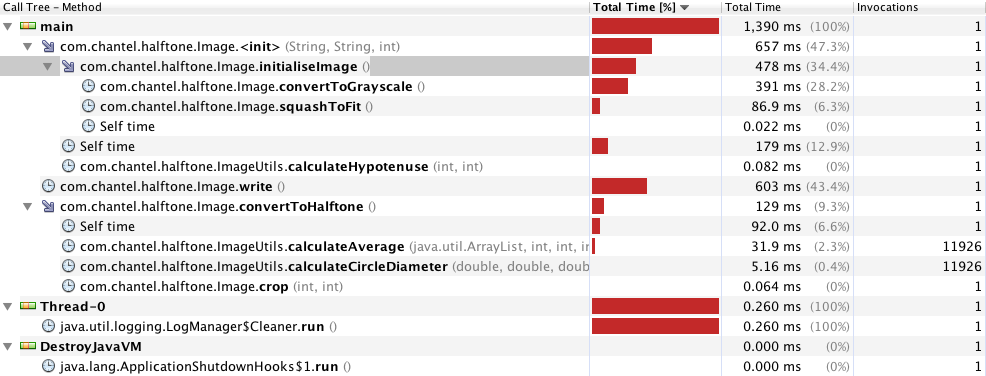
**Before**

CARMEN TODO:

**3 : 2 aspect ratio 1.2 megapixel image**

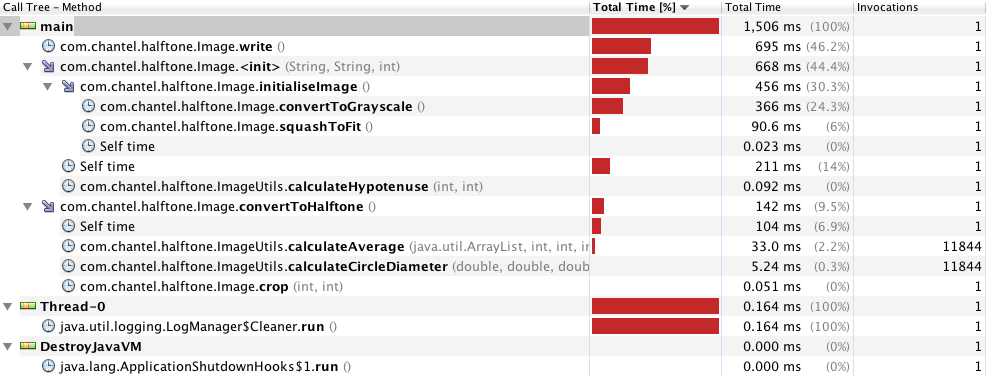


Report



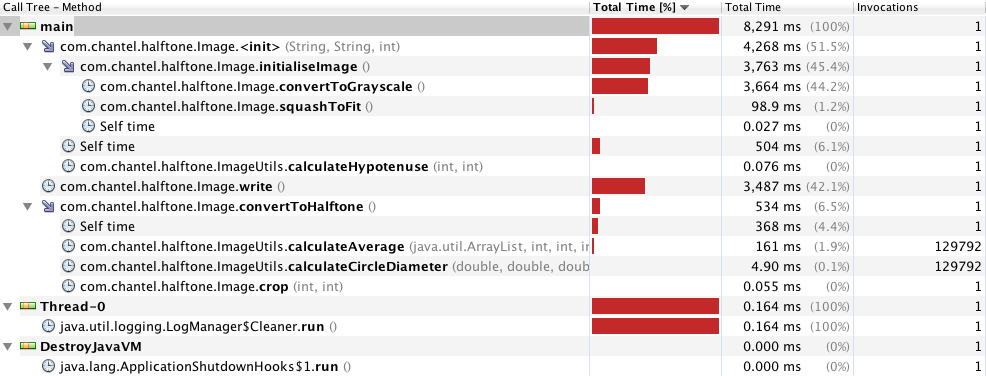
CARMEN TODO:

**4 : 3 aspect ratio 1.2 megapixel image**



CARMEN TODO:

**4 : 3 aspect ratio 13 megapixel image**



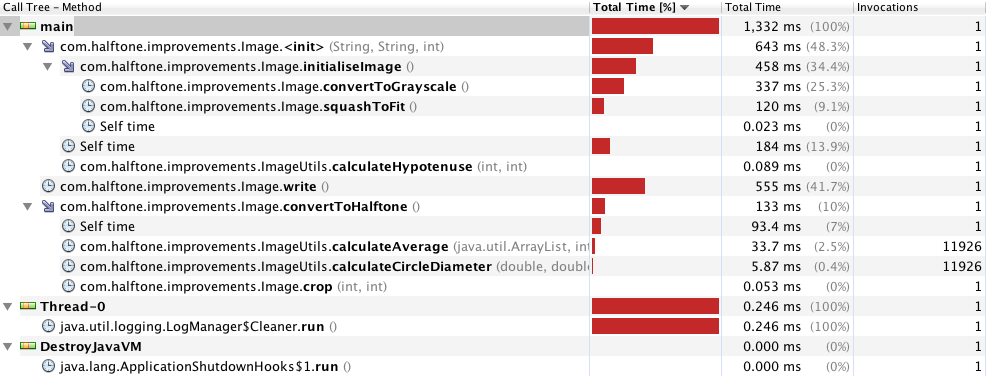
Report

CARMEN TODO:

**After**

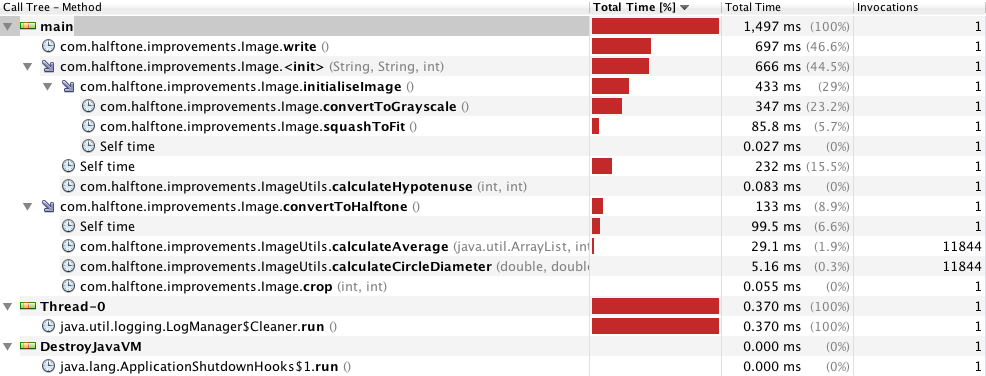
CARMEN TODO:

**3 : 2 aspect ratio 1.2 megapixel image**



CARMEN TODO:

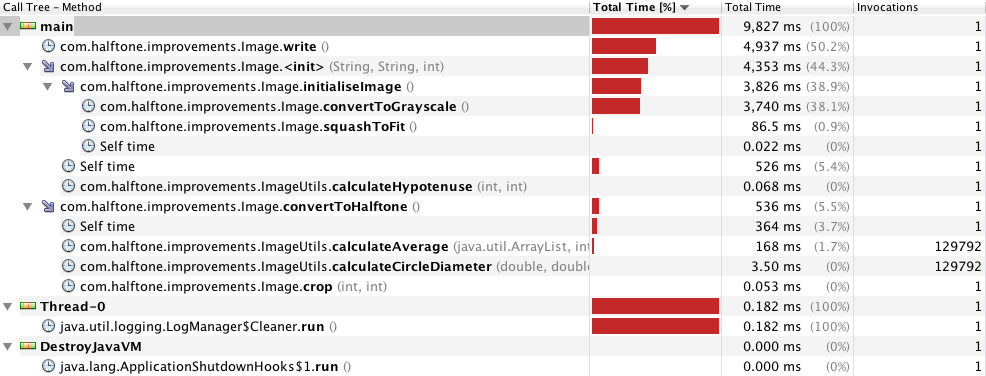
**4 : 3 aspect ratio 1.2 megapixel image**



Report

CARMEN TODO:

**4 : 3 aspect ratio 13 megapixel image**



CARMEN TODO:

**Conclusions and Recommendations for the Future**

CARMEN TODO: