# CFG Report: Lichtenstein Art Generator

By Jonathan Flynn NCCA-CVA Level C 2014

## Background

Roy Lichtenstein was an American pop artist born 1923 and died in 1997. Lichtenstein had a unique style, favoring the comic strip style, producing hard-edged and precise compositions. His work was heavily influenced by both popular advertising and the comic book style. Some of his most famous pieces are *Whaam!* and *Drowning Girl.*

## Algorithms and Techniques

For the creation of a 'Roy Lichtenstein' style pop art piece of art can be broken down into three different stages:

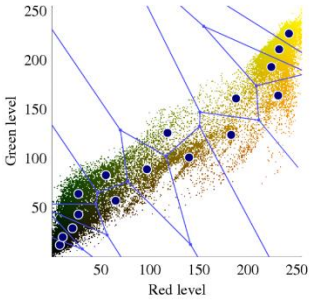
- Colour Quantizing
- Halftoning
- Edge Detection (Canny Edge Detect)

## Colour Quantizing

Colour quantizing, otherwise known as palette reduction, reduce an images colours to either a pre-defined set of colours or an adaptive selection closest to the colours already existing within the image. The process is critical for displaying images with many colors on devices that can only display a limited number of colors, usually due to memory limitations, and enables efficient compression of certain types of images. However, for the generation of this artwork we are doing this process for an artistic effect over memory limitations.

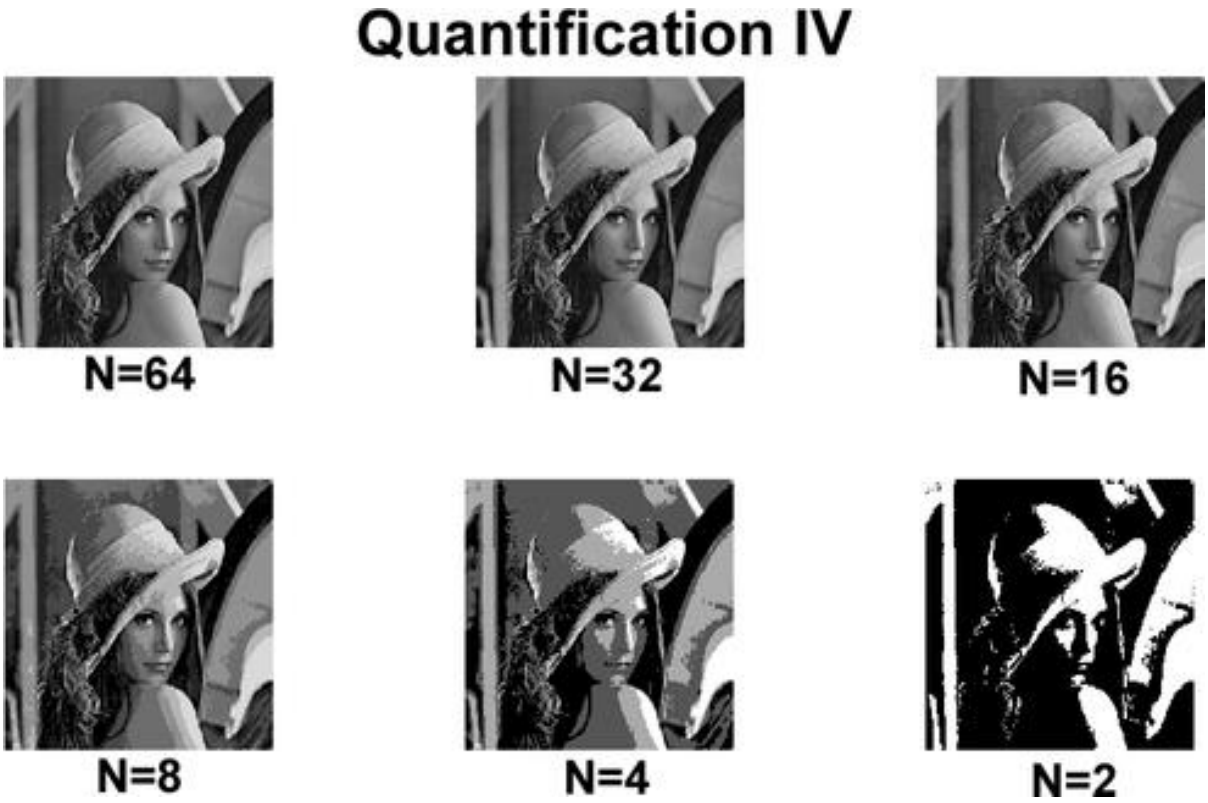
Most standard techniques treat color quantization as a problem of clustering points in three-dimensional space, where each of the points represent colors found in the original image and the three axes represent the three color channels. Almost any three-dimensional clustering algorithm can be applied to color quantization, and vice versa. After the clusters are located, typically the points in each cluster are averaged to obtain the representative color that all colors in that cluster are mapped to. The most often used color channels are usually red, green, and blue,

The most popular algorithm used is the median cut algorithm, invented by Paul Heckbert in 1980. An even more modern method of this is clustering using octrees, which are tree data structures in which each internal node has exactly eight children, and using these essentially constructs a histogram of equal sized ranges and colours are assigned to these ranges containing the most points.




This image shown on the left shows a histogram of a two dimensional image with only the colours green and red. All of the regions shown by blue lines are what will be turned into the colour sampled at the single blue spot within that region. This creates 16 colour palette for the image. These blue lines are also known as a Veroni Diagram which is a way of dividing space into a number of different regions.

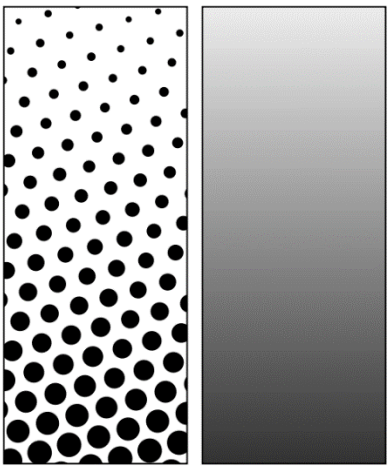
The main colours used by a Lichtenstein piece (or the image shown above) are yellow, red, blue, black and white. The colours will be used to represent the closest matching colour on the image. This means that my code will need to reduce the colours of an image using Adaptive colours or colours that closely match the ones in the image, and then compare the defined colours wanted by the user to replace. Then, the closest match between all of these colours will then be used and replace the ones in the image.




DIFFERENT LEVELS OF COLOUR QUANTIZE WITH DIFFERENT NUMBERS OF COLOURS

## Halftoning

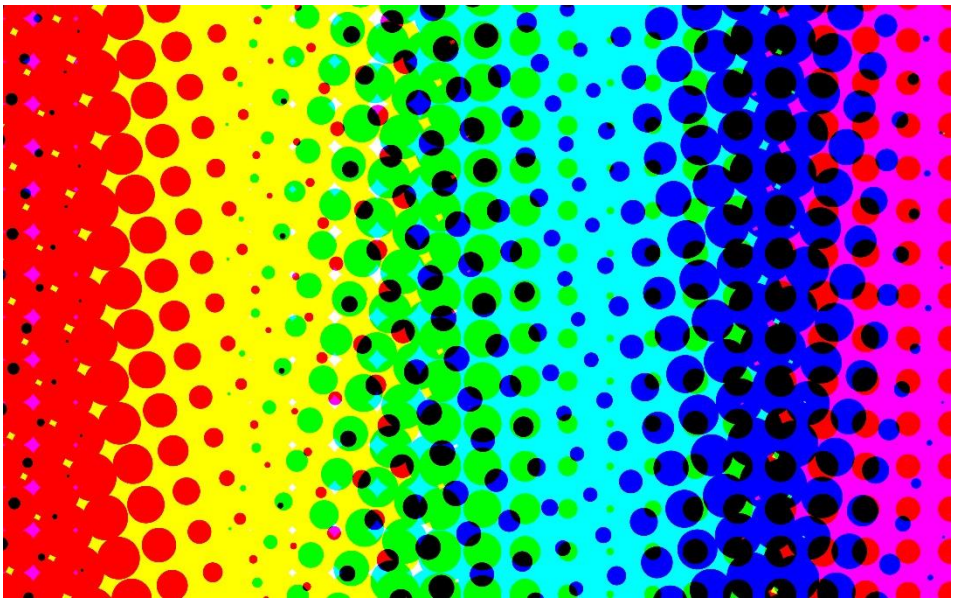
Halftoning is a technique used primarily in printing to simulate the visual of continuous tone imagery by tricking the eye using dots, varying in either size or shape, to represent an image.



In the example of a greyscale image, with an infinite range of greys, the halftone process reduces this image so that only one colour of ink is used to represent all the colours from white to black. This is done primarily by changing the size of the circles, with the smaller the circle radius representing whiter colours and bigger radius representing blacker colours. The process relies on an optical illusion to give the result.

Halftoning essentially takes an area of pixels in the shape of a rectangle, samples the luminosity of each of these pixels and gets the average from all the pixels selected. Then, a circles is drawn in the centre of this rectangle with the radius determined by the overall luminosity and size of the area of pixels.

The most effective method uses dots that are spaced diagonally on the image, so that when these circles are the max radius they fill all the image, leaving no white or background behind these dots.



The technique shown in the image above uses multiple angles for the colours CMYK and draws the dots so that when they overlap, it creates an additive effect to the colours: this means that more colours that overlap each other, the darker it becomes. This is one of the most common way used to represent CMYK images for printing.

For my project, I will need to create halfoning for RGB images, allowing for a set colours to be used for the circles drawn and also allowing for any choice of colour for the background.

## Edge Detect (Canny Edge Detection)

Canny Edge Detection is the most favoured technique used for detection the edge for images. It uses a multi stage algorithim to detect a wide range of edges.

There are four stages to the Canny algorithim:

1. Noise Reduction and Greyscale
2. Gradient Intensity of the image
3. Non-maximum suppression (edge-thinning)
4. Edge tracing using hysterias thresholding

### Noise Reduction and Greyscale

This stage involved the application of a 5x5 Gaussian blur with sigma = 1.4. This value can be vary depending on the amount noise in image. The result is a slightly blurred version of the original image. The image is then turned into a greyscale image either before or after this stage.

The result of this is slightly blurred, greyscale version of the image which is not affected by a single noisy pixel to any significant degree.

### Gradient Intensity of the image

Since the edges of an image point in different directions, we need a way to get the estimate gradient of each pixel. A search is carried out to determine the gradient of a pixel using a Sobel operator to detect the horizontal, vertical and diagonal edges. It essentially calculates the intensity of the image at each point. Edges of an image are places with a sudden jump in intensity. What Sobel does is calculates changes between pixels as well as the direction of the change. The sobel operator uses two masks (also known as kernals): one for the Y direction and one for X. If A is the image and Gx and Gy are two images which at each point contain the horizontal and vertical derivative approximations, the computations are as follows:

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{A}$$

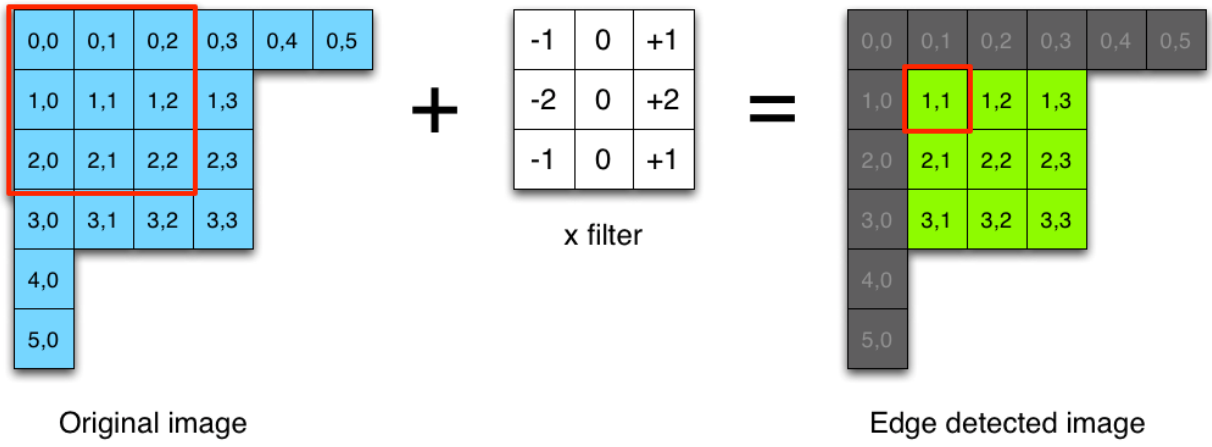
Then, the magnitude is calculated using the hypotenuse formula:

$$\mathbf{G} = \sqrt{{\mathbf{G}_x}^2 + {\mathbf{G}_y}^2}$$

And the angle is calculated using the following:

$$\mathbf{\Theta} = \operatorname{atan2}(\mathbf{G}_y, \mathbf{G}_x)$$

When using these functions, the outer edge pixels can't be used since sliding occurs on each pixel. See the diagram below for a visual representation of this:




Each of the edge detected angles is then rounded to the angles 0, 45, 90 and 135 degrees. Since the result will give negative degrees, these values will need to be converted to positive using 360 – the degrees.

### Non-maximum suppression (edge-thinning)

What this stage essentially does is take the direction of each pixel along with the magnitude and checks if the magnitude is less than its diagonals, vertical or horizontal pixels dependent on the angle. Then, we mark that pixel as not an edge if the value is less. For example, if the angle = 0 at a pixel, the magnitude at that pixel will be compared to the pixel above and below it. If the pixel is less than either of the pixels above or below it, the pixel isn't an edge.

What this does in more complex terms, is assumes a local magnitude in the set gradient direction of 0,45,90 or 135 degrees. At every pixel, it suppresses the edge strength of the center pixel (setting the magnitude to 0) if its magnitude is not greater than the magnitude of the two neighbors in the gradient direction. The following example has been taken from http://en.wikipedia.org/wiki/Canny_edge_detector :

- if the rounded gradient angle is zero degrees (i.e. the edge is in the north–south direction) the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the **east and west** directions,
- if the rounded gradient angle is 90 degrees (i.e. the edge is in the east–west direction) the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the **north and south** directions,
- if the rounded gradient angle is 135 degrees (i.e. the edge is in the northeast–southwest direction) the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the north west and **south east** directions,
- if the rounded gradient angle is 45 degrees (i.e. the edge is in the north west–south east direction) the point will be considered to be on the edge if its gradient magnitude is greater than the magnitudes at pixels in the north east and **south west** directions.

### Edge tracing using hysterias thresholding

Large intensity gradients mean a higher chance of an edge then small intensity gradients. Canny Edge Detection uses thresholding with hysteresis to help specify when an edge switches into not being an edge.

This process requires two thresholds – one high and one low. What we do with these values is assume that important edges should be along continuous curves in the image which allows us to follow a faint section of a given line and to discard a few noisy pixels which have produced large gradients as a result. The higher threshold ensures that all the edges marked we can be fairly sure are genuine. With the direction of the pixels previously, we can begin to travel along the pixels and trace the lines present. As we do this, we compare the lower threshold values as well, allowing for us to trace faint sections of edges adjacent to the assured edge from the higher threshold.

What we essentially need to do is traverse all the pixels within an image in the higher thresholded image, mark all the edges in this image and traverse all the adjacent pixels to that pixel, checking if the adjacent pixel isn't in the higher threshold but is in the lower threshold. If this is the case, this is a part of an edge but with lower intensity. Once all the pixels have been traversed, the image is created.

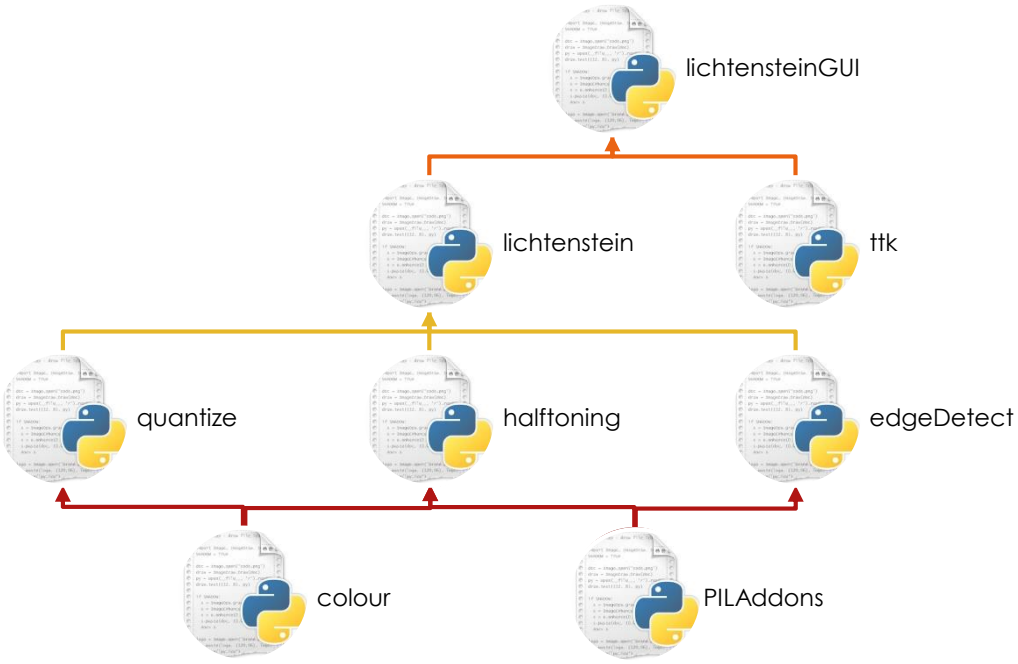
## Implementation

I decided to create the program in Python due to having more experience with the language and being more confident in it. I created and handled all the images using the Python Image Library (PIL) and also decided to create the GUI using Tkinter.

The structure of my program consists of 8 different modules:

- colour.py
- PILAddons.py
- quantize.py
- halftoning.py
- edgeDetect.py
- lichtenstein.py
- lichtensteinGUI.py
- ttk.py

Here is a flow chart showing the structure of my program:




**Points to consider:**

1. The use of a Gaussian Filter was available with the version of PIL I was working with, but found out too late that the version in the university labs uses a different version without this feature. This means that the code blurs using only a set amount from a PIL ImageFilter. The commented lines can be seen in the comments of the code. This means that all references to sigma for Gaussian blur and de-noise amount in the GUI are set to fixed amount and won't impact the final result
2. The use of a preview image is possible with a certain module in the Tkinter library which has been implemented, however this module again is not available in the unveristy labs so sadly this feature has been disabled. The screenshots represent how this would look if the module was available (except in the how to use manual), and if you wish to test the code with this feature working, you can test this on

## colour.py

This module contains feature to use for dealing with RGB colours. There is nothing much of note here involving the implementation of algorithms in the program. Check the pydoc documentation for more information on this module.

## PILAddons.py

This module contains features to use with the PIL and adds new features to the module, such as a Palette object and the creation of circles from a centre point with a specified radius. It does contain some useful features for traversing images such as finding adjacent pixels and iterating through all the x,y coordinates of an image. Again, there is nothing much of note here involving the implementation of algorithms in the program. Check the pydoc documentation for more information on this module.

## quantize.py

This module contains the implementation of the Colour Quantize algorithems needed for the creation of the image.

Since the PIL already comes implemented with the feature to reduce the palette of an image to an adaptive, defined number of colours it allowed for this part of the code to be implemented easily. However, I had to

implement my own technique for comparing colours to ones present in an Adaptively colour image. The code below shows a portion of the **'colour_switch'** function used to transfer these colours:

```
    cCloseness = {}
    for cCol in curC:
        cCloseness[cCol] = {}
        for nCol in newC:
            cCloseness[cCol][nCol] = sum([abs(cCol[i]-nCol[i]) for i in range(3)])

    finalPalette = curC[:]
    for newColour in newC:
        curCol, close = None, 766
        for currentColour, comparisons in cCloseness.items():
            if comparisons[newColour] < close:
                curCol, close = currentColour, comparisons[newColour]

        finalPalette[finalPalette.index(curCol)] = newColour
        del cCloseness[curCol]

    return finalPalette
```

This piece of code uses a dictionary, 'cCloseness' to store the overall closeness of each new colour in 'nCol' to the current colours of the Adaptively coloured image in 'cCol'. The calculation to find the closeness of each colour finds the difference between each of the Red, Green and Blue channels to one of the current colours and adds all of these values together. This is on line 5. Once this is done, the loweset closeness (or most close) colour is found and replaced in the palette. Then the final palette is returned with all the colours which have been replaced and the ones that haven't. For information on the rest of the code, check the pydoc.

## halftoning.py

This module contains the implementation of the Halftoning algorithm needed for the creation of the image.

To halftone the image, I had to iterate through the entire image in chuncks, essentially drawing a grid on the image and sampling each box of pixels to get it's luminosity. The code below shows this:

```
for i,x in enumerate(xrange(box/-2, img.size[0], box)):
        col = 0 if i % 2 == 0 else box/2
        for y in xrange(box/-2, img.size[1], box):
            pixelColours = []
            for v in xrange(x, x+box):
                for n in xrange(y,y+box+col):
                    try:
                        pixelColours.append(imgPix[v,n])
                    except IndexError:
                        pass
```

This code travels throughout the width of the image with the 'x' coordinate being that position, but also having this xrange staring at outside the area of the image, so that any pixels that don't fit perfectly into the size of box are considered as well. The value 'i' is used to create diagonally positioned circles by checking if the current iteration is either odd or even and either adding a 'col'umn offset dependent on this. Then the box area is iterated through with the 'v' and 'n' values being each pixel inside the box area and the colour of the pixel stored into a list 'pixelColours' if the pixel at coordinate [v,n] exists. Then the rest of code deals with the average luminosity of the area and draws a circle with that information.

## edgeDetect.py

This module contains the implementation of the Edge Detect algorithm needed for the creation of the image.

The process for creating Canny Edge detection was handled in the same way as the theory stated in 'Algorithms and Techniques'. There are a large amount of areas that would warrant mentioning in this module and so I have tried to comment the code as much as possible to ensure each step is clear. The following piece of code shows the Non-maximum suppression section of the code:

```
for x,y in pila.pixel_generator(width, height, 1,1):
    if sobelOutDir[x][y]==0:
        if (sobelOutMag[x][y]<=sobelOutMag[x+1][y]) or \
           (sobelOutMag[x][y]<=sobelOutMag[x-1][y]):
            magSup[x][y]=0
    elif sobelOutDir[x][y]==45:
        if (sobelOutMag[x][y]<=sobelOutMag[x+1][y+1]) or \
           (sobelOutMag[x][y]<=sobelOutMag[x-1][y-1]):
            magSup[x][y]=0
    elif sobelOutDir[x][y]==90:
        if (sobelOutMag[x][y]<=sobelOutMag[x][y+1]) or \
           (sobelOutMag[x][y]<=sobelOutMag[x][y-1]):
            magSup[x][y]=0
    else:
        if (sobelOutMag[x][y]<=sobelOutMag[x-1][y+1]) or \
           (sobelOutMag[x][y]<=sobelOutMag[x+1][y-1]):
            magSup[x][y]=0
```

This piece of code iterates through every pixel in the image (except for the outer edges) checking the direction of that pixel from the sobel operation. If that value is 0, the pixels horizontally adjacent are checked for a higher intensity. If this is True, then the intensity at that pixel is reduced to 0. This is then repeated for each angle, respectively checking the pixels adjacent in the angle they are pointing. Check the pydoc for more information.

## lichtenstein.py

This module implements all features of the previous modules and combines them to create the final result.

The only real noteworthy code for this module is as follows:

```
for x,y in pila.pixel_generator(*img.size):
    if quantPix[x,y] in qtNewCols:
        halfMaskPix[x,y] = 1
    else:
        halfMaskPix[x,y] = 0

compQuHt = Image.composite(quantImg, halfImg, halfMask)
finalImg = Image.composite(compQuHt, edgeImg, edgeMask)
```

What this code does is first of create a mask from all the new colours specified for the Colour Quantize feature of the code. The result checks each pixel colour in the result 'quantImg' using the pixel load object for that image. If the colour is one of the 'qtNewCols', then it gets a black value or '1' which means that it will cause the halftone image 'halfImg' to be transparent at that point. If not, it will be visible with a value '0'. Then the images are combined together, with the 'quantImg' being overlayed with 'halfImg' and having the 'halfMask' affecting what pixels are visible and then having the 'finalImg' created from this result and the 'edgeImg' being applied on top.

## lichtensteinGUI.py and ttk.py

The lichtensteinGUI.py is the Graphical User Interface for the generator and the ttk.py module is a module for additional Tkinter widgets which aren't installed on the system in the labs. Neither of these programs have features involving the implementation of the algorithms.

# Results

Here are each of the results created from each module and the final result:

## Colour Quantize:

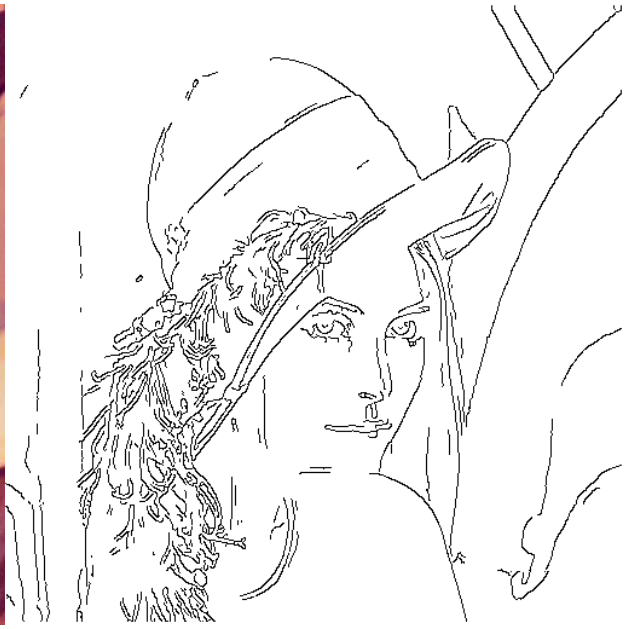
Using the colours: [(0,0,0), (255,255,255), (190,0,0), (0,16,115), (248,196,0)]

## Halftoning

Using Average Colours for circles with radius 8 and white background

## Edge Detect

Using sigma of 1.4, a high threshold of 0.2 and low threshold of 0.1 and line colour of black



## Lichtenstein (Final Result)

Combining all the above results.

## Sources of references and information:


http://en.wikipedia.org/wiki/Color_quantization
http://en.wikipedia.org/wiki/Halftone
http://en.wikipedia.org/wiki/Edge_detection
http://en.wikipedia.org/wiki/Canny_edge_detector
http://en.wikipedia.org/wiki/Sobel_operator
http://pythongeek.blogspot.co.uk/2012/06/canny-edge-detection.html