香港科技大學
THE HONG KONG
UNIVERSITY OF SCIENCE
AND TECHNOLOGY

CSIT5400 Computer Graphics

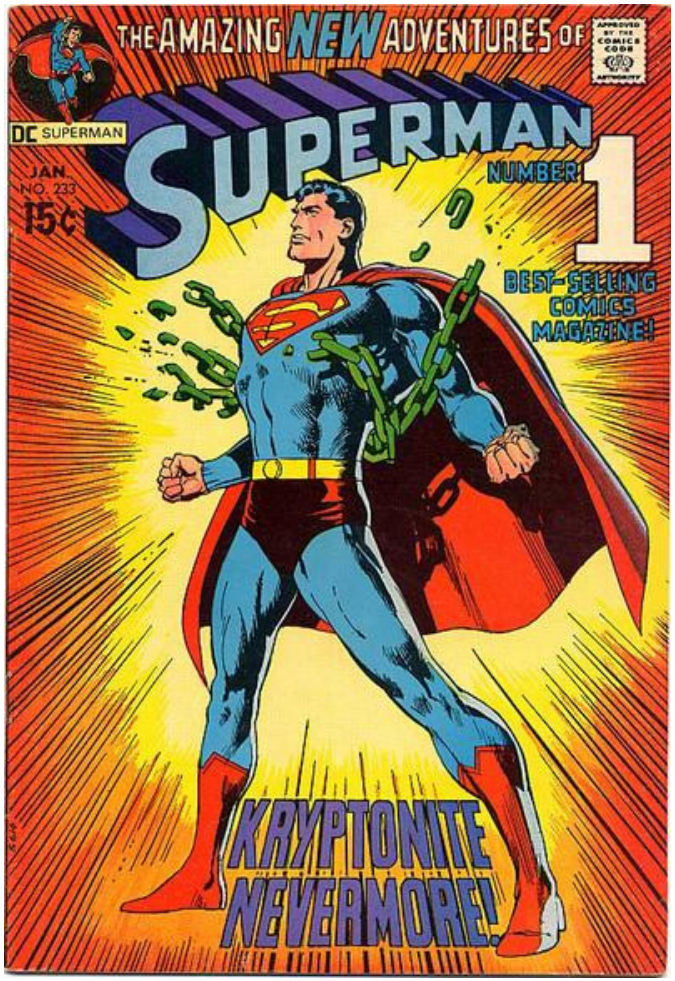
Assignment 1 – COMICification

COMICification

Assignment 1

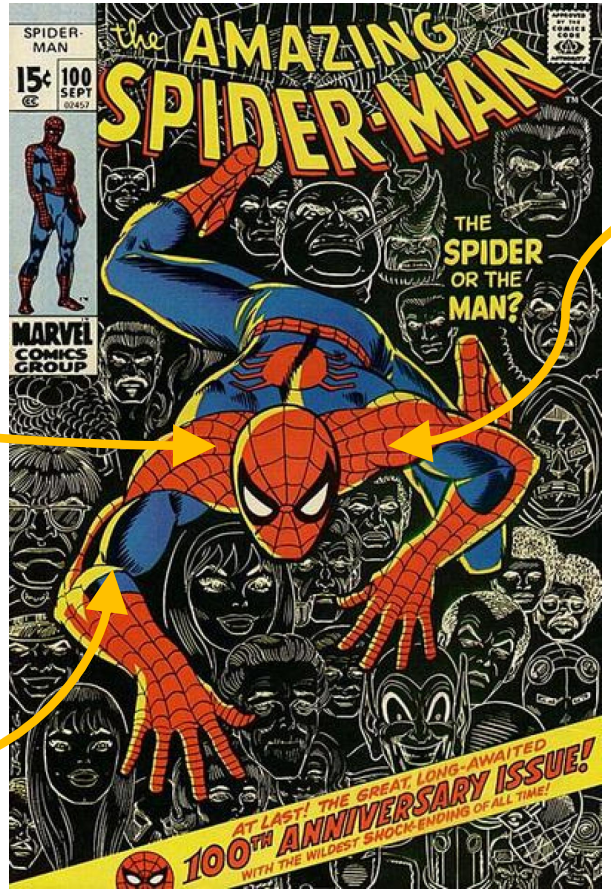
COMICification

- This assignment combines some image processing techniques to create interesting ‘1980s comic book’ style images
- The input to the system is an ordinary photo and the output is a comic-style image of that photo
- Most of the techniques you will use have already been discussed in the course



Some Observations

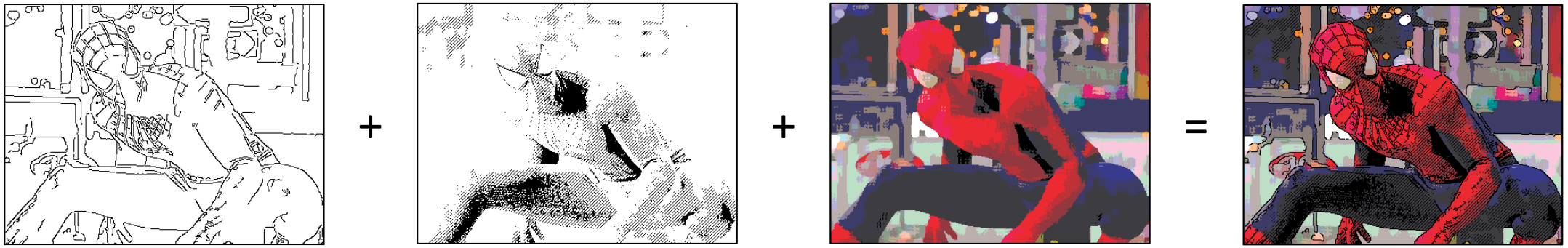
Outlines are emphasized using black lines



Colours are typically 'colourful' colours (not grayish/dull colours) and are coming from a fixed set of production colours

Dark shades are created by sketches of patterns using lines or checker

Three-Layered Processing

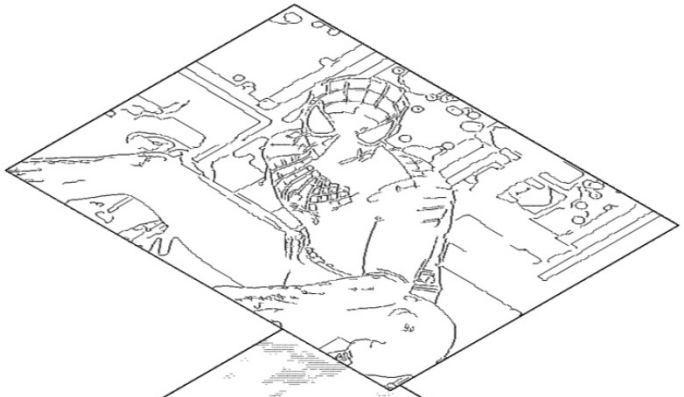


- Outlines (top layer)
  - Extracted using edge detection algorithms
- Shades (middle layer)
  - Created by overlaying a black and white pattern on the image
- Colours (bottom layer)
  - Produced by smoothing out colour regions, increasing the ‘colourful-ness’ of the image and mapping the new colours to a pre-defined output colour palette

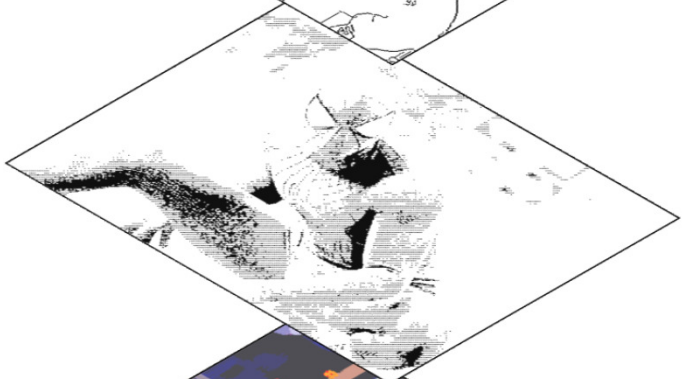
The COMICification Process

Input image

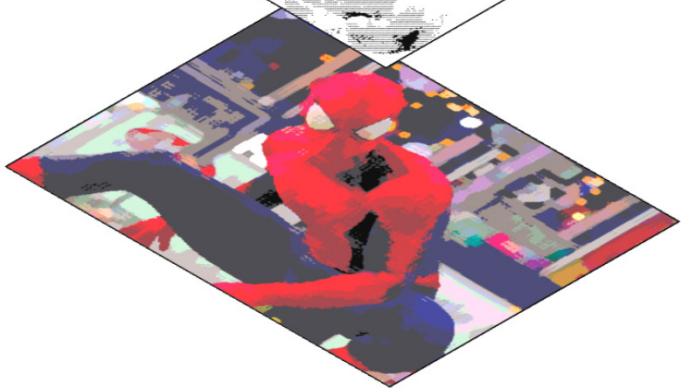
Outlines



Shades



Colours



Combined result

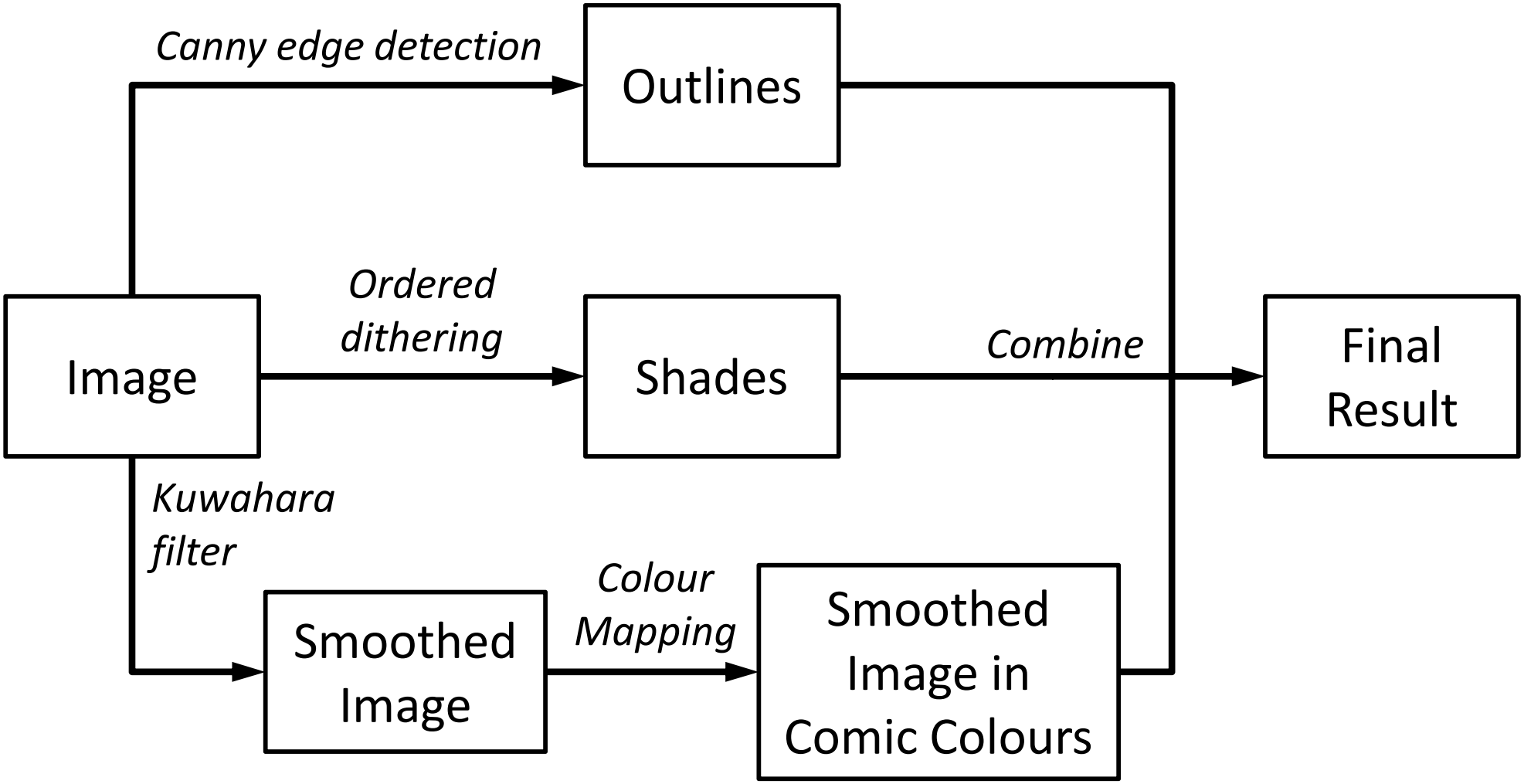
The Image Processing Techniques

- Top layer – outlines
  - Canny edge detection
- Middle layer – shades
  - Ordered dithering
- Bottom layer – colour
  - Kuwahara filter
  - Mapping colours to a comic colour palette

You have/will learned these in the course

You can do this using what you have learned about the colour systems

The Process Flow



The Assignment System

COMICification

Source Image: Superman ▾

Result Image:

Outlines: Canny Edge Detection ▾ Shades: Ordered Dithering ▾ Colours: Comic Colours ▾ COMICify!

Canny Edge Detection

Gaussian Size: 5

Strong Threshold: 100

Weak Threshold: 50

Ordered Dithering

Pattern: Lines ▾

Kuwahara Filter

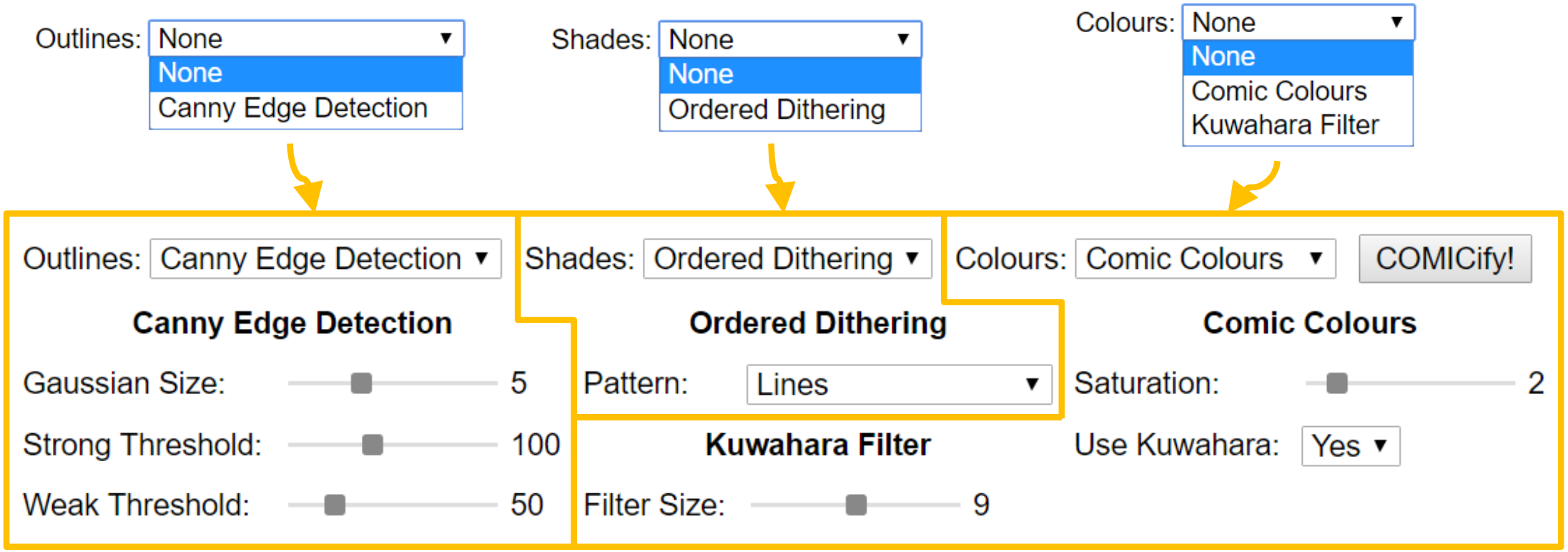
Filter Size: 9

Comic Colours

Saturation: 2

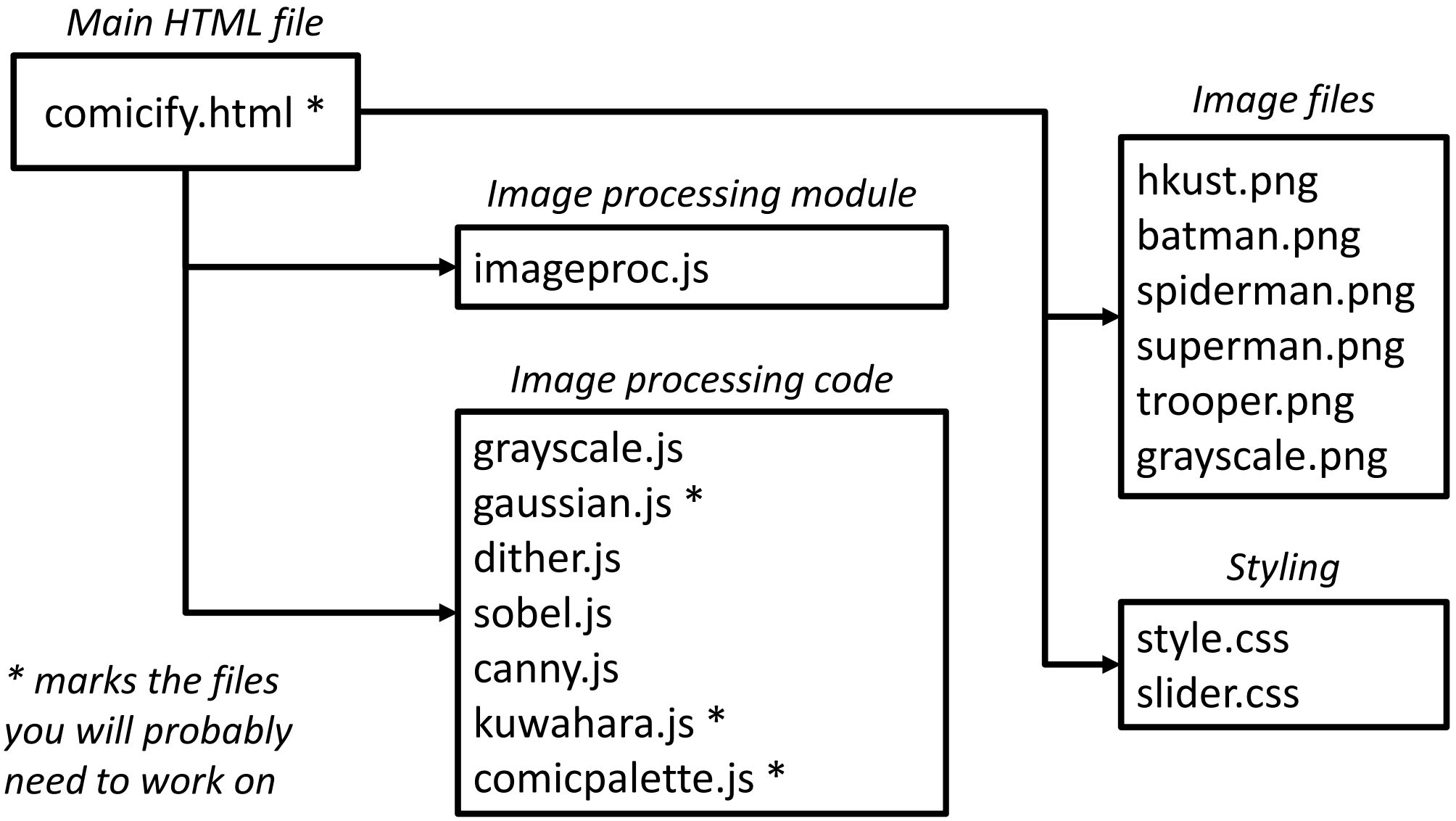
Use Kuwahara: Yes ▾

The Controls



- These controls can be used to set up and display the three layers separately

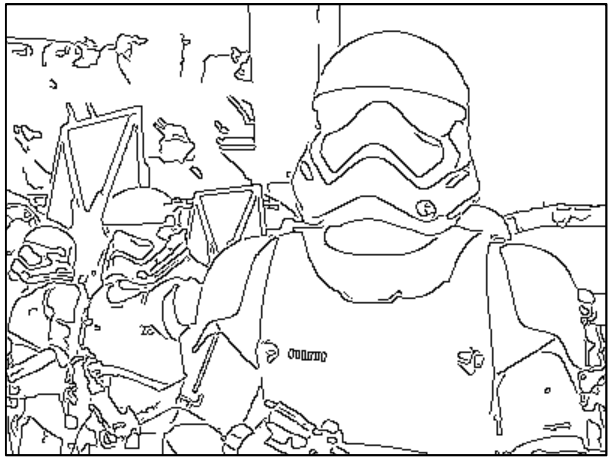
The File Structure



Top Layer – Outlines

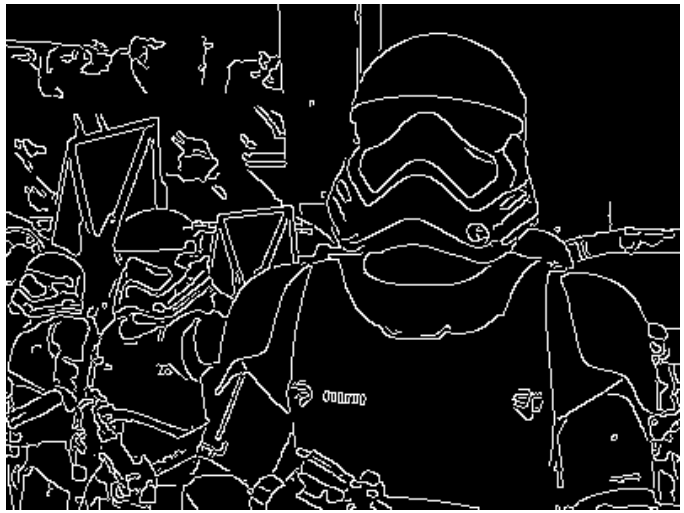
- The outlines of the input image can be extracted using edge detection algorithms
- We have learned a few of those and the better one among them is the Canny edge detection
- Therefore, in the assignment, you will use it to capture the outlines of the image

Canny edge detection



The Outlines in the Starting Code

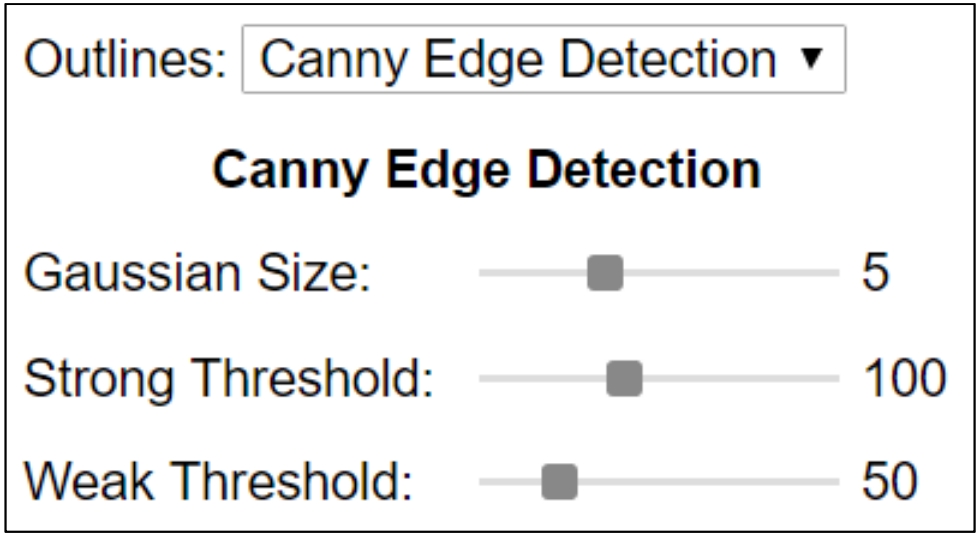
- The outlines displayed in the starting code are shown in white on a black background



- This is because the output of the Canny edge detection uses (255, 255, 255) to represent an edge pixel
- Later on when you blend the edge to the final result, the display of the edge should be inverted so that the edge will become black on white

Canny Edge Detection

- The following settings can be adjusted for the Canny edge detection in the assignment
  - Size of the Gaussian kernel
    - The available sizes are 3, 5, 7 and 9
  - Thresholds for strong and weak edges
    - The thresholds for tracing the edges at the last stage of the Canny edge detection



Canny Edge Detection – Your Task

- The Canny edge detection code has been given in the `cannyEdge` function in `canny.js`
- It is already working correctly given a strong and a weak threshold values
- However, although the function accepts a kernel size parameter, the Gaussian kernel used at the moment is a fixed 3x3 Gaussian kernel
- You need to extend the code so that the Gaussian kernel can have any given size (odd numbers only)

Generating the Gaussian Kernel

- The code for applying Gaussian blur is in the function `gaussianBlur` inside `gaussian.js`
- Rather than using a precomputed Gaussian kernel, you must generate a Gaussian kernel on the fly using the following steps:
  1. Generate a 1D Gaussian kernel based on the size
  2. Determine the 2D Gaussian kernel from the 1D one
  3. Compute the divisor of the 2D Gaussian kernel

Generating a 1D Gaussian Kernel

- The 1D Gaussian kernel can be computed based on the following equation we have shown before:

$$g(x,y) = \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{x^2}{2\sigma^2}\right) \cdot \frac{1}{\sqrt{2\pi}\sigma} exp\left(-\frac{y^2}{2\sigma^2}\right)$$

- Obviously you only need to generate a single 1D kernel because the x and y kernels are the same
- In the assignment, you need to set the standard deviation (σ) to be one-fourth of the kernel size

Finding the 2D Kernel

- Once you have generated the 1D kernel, the 2D kernel can be easily found
- You do not need to normalize the kernel values and the divisor of the kernel is simply the sum of all coefficients
- For example, a 3x3 kernel would be similar to the one shown on the right:
  (The numbers are rounded in the example but you need not do that)

1D Kernel:

0.219	0.532	0.219

2D Kernel:

0.048	0.116	0.048
0.116	0.283	0.116
0.048	0.116	0.048

Divisor: 0.94

Making the Kernel in the Code

- You need to use the following variables to store the values for the generation of your Gaussian kernel in `gaussian.js`:
  - `sigma` - the standard deviation of the Gaussian distribution
  - `rowMatrix` - the 1D Gaussian kernel
  - `kernel` - the 2D Gaussian kernel created from the 1D Gaussian kernel
  - `divisor` - the divisor for the 2D Gaussian kernel

The Code Showing the Kernel

- The following code shown on the right will be used to show the kernel in the JavaScript console
- You must not remove or change it or we will not be able to mark your work

```
/***** DO NOT REMOVE - for marking *****/
var line = "";
console.log("Row matrix:");
for (var i = 0; i < size; i++)
    line += rowMatrix[i] + " ";
console.log(line);

console.log("Kernel:");
for (var j = 0; j < size; j++) {
    line = "";
    for (var i = 0; i < size; i++) {
        line += kernel[j][i] + " ";
    }
    console.log(line);
}
console.log("Divisor: " + divisor);
/***** DO NOT REMOVE - for marking *****/
```

An Example Console Output

- Here is an example console output for the generation of a 3x3 2D kernel, using Chrome:

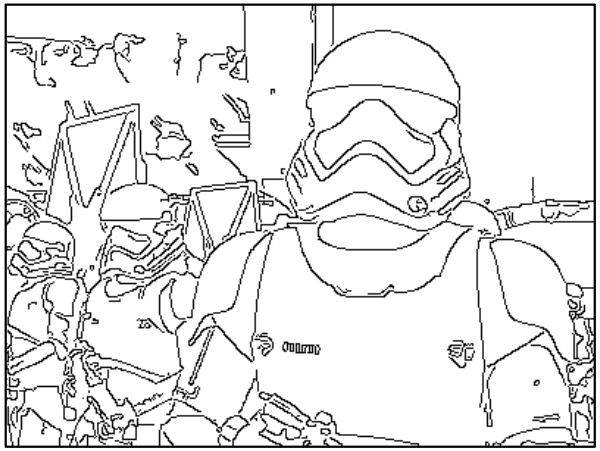
```
Row matrix:
0.2186800995679915 6 0.5319230405352436 0.2186800995679915 6
Kernel:
0.047820985947066 7 0.1163209834667558 9 0.047820985947066 7
0.1163209834667558 9 0.2829421210522584 7 0.1163209834667558 9
0.047820985947066 7 0.1163209834667558 9 0.047820985947066 7
Divisor: 0.9395099987075489
```

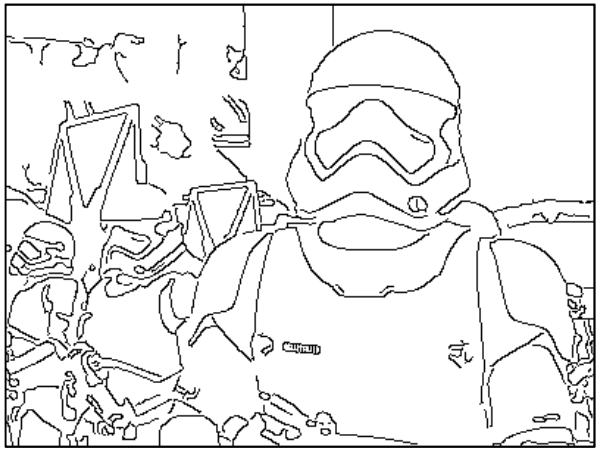
- Other browsers may have slightly different output values with a different format

Adjusting the Gaussian Kernel

- With the same thresholds, you should get less edges when the size of the Gaussian kernel increases

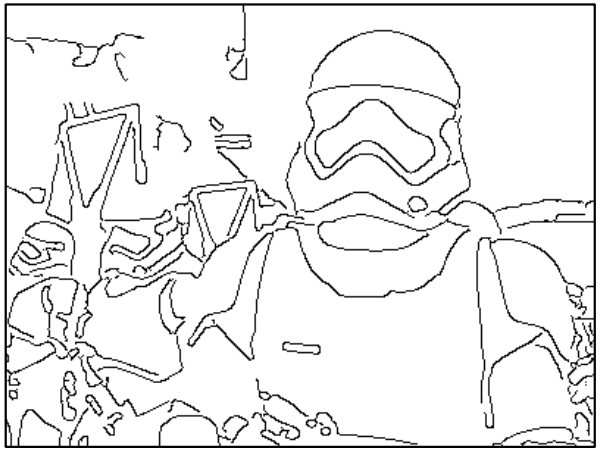
Kernel size = 3





Kernel size = 5

Kernel size = 7



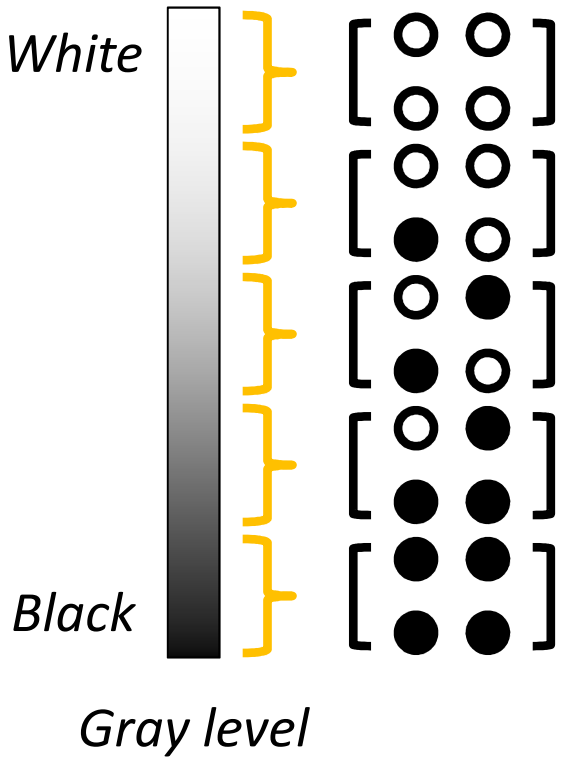
Kernel size = 9

Middle Layer – Shades

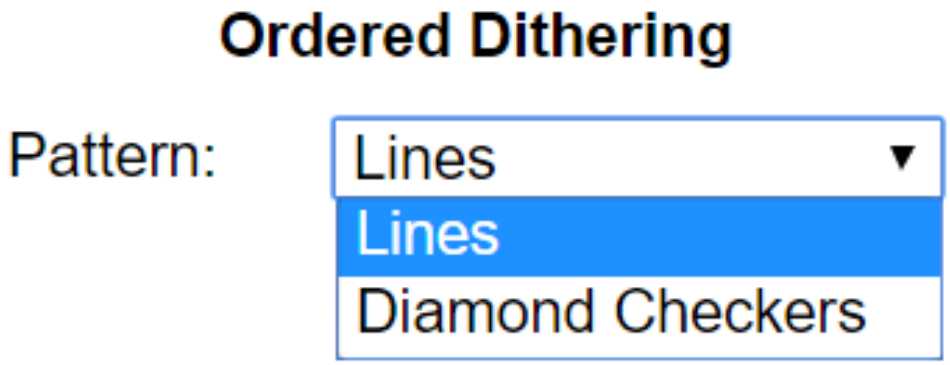
- The comic colours are coming from a set of production colours and they are typically too colourful and not dark enough for some areas in the image
- Shades can be added to the image so that the colours look darker than what they really are
- In comic books, shades are created by sketching black lines or patterns, e.g. checker, around the dark areas
- The assignment uses ordered dithering to add the shades in the image

Ordered Dithering

- Remember we used the Bayer's matrix as the pattern in ordered dithering so that an image can be represented by five different levels of gray patterns



- Rather than the Bayer's matrix, the assignment allows the use of two specific patterns:



Ordered Dithering – Your Task

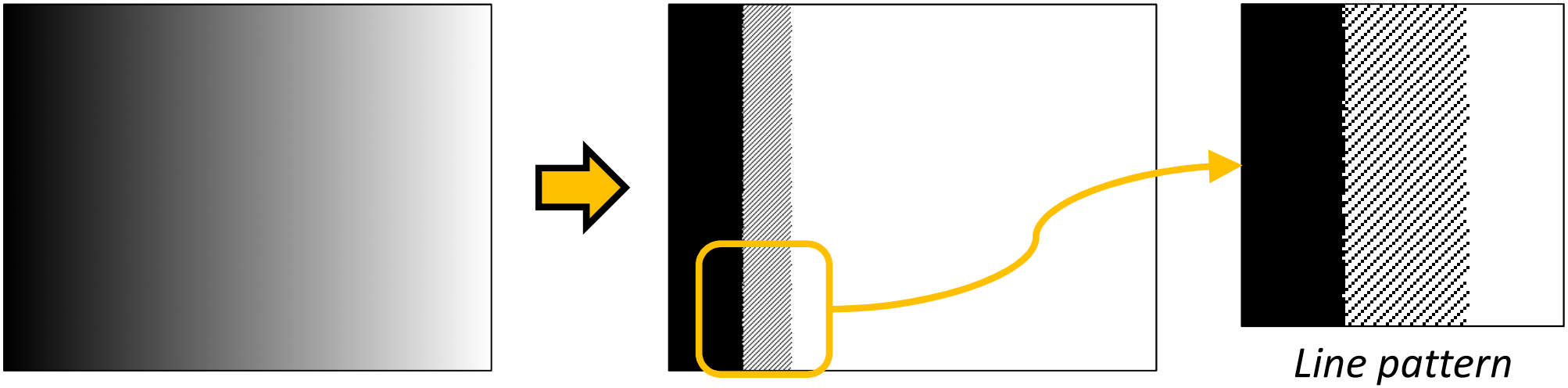
- At the moment, the starting code always uses a 2x2 Bayer's matrix to create the shades
- You need to create the line pattern and the checker pattern based on the selected pattern in the control

```
var matrix = [
    [1, 3],
    [4, 2]
];
var levels = 5;
```

- Either pattern is passed to the `dither` function in `dither.js` as a two dimensional array
- The number of gray levels is passed to the function using the `levels` parameter

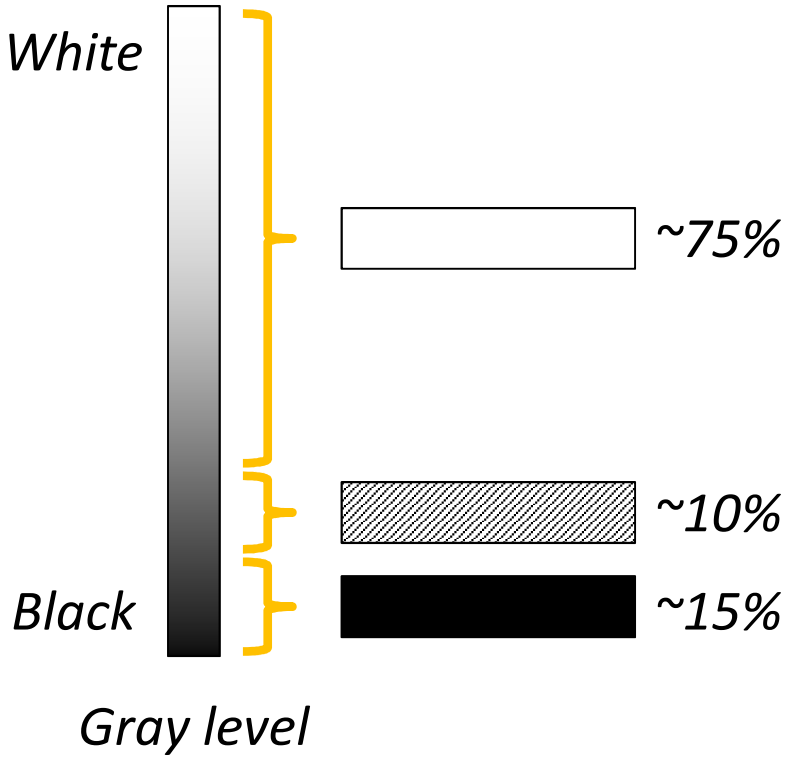
The Line Pattern

- The grayscale image provided in the starting code can be used to verify the correctness of the patterns
- For example, the line pattern should look like this after applying it on the grayscale image:



Gray Level of the Line Pattern

- From the image on the previous slide you can see that the line pattern, as well as the black region, do not distribute evenly in the grayscale image
- In the assignment, you must use the distribution shown on the right:
- As you can see, a large portion (75%) of the gray level is displayed as pure white

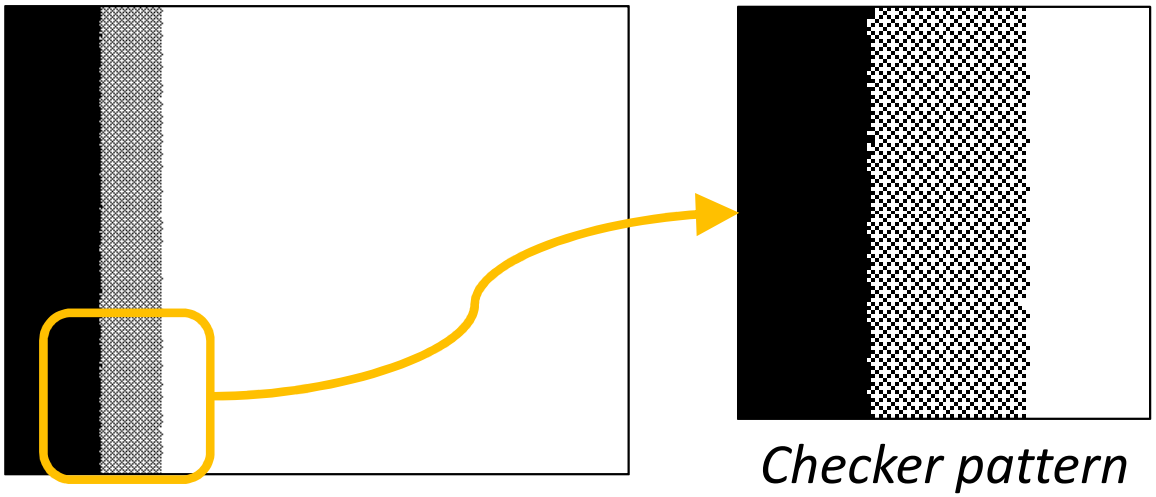


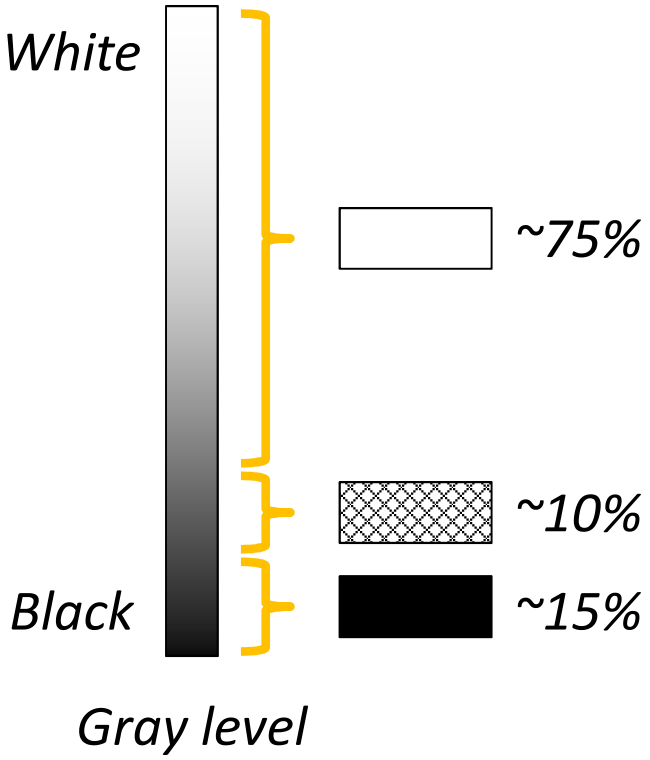
Using the Matrix and the Range

- To make the distribution shown on the previous slide, you will need to fully understand how ordered dithering works, i.e.:
  - The matrix stores the different grayscale patterns in an incremental way
  - Each coefficient decides the grayscale level at which the corresponding white pixel appears
  - The coefficients must be smaller than the number of gray levels representing a pixel

The Diamond Checker Pattern

- The way to make the checker pattern is very similar to the one to make the line pattern
- The grayscale image after applying the checker pattern and its distribution are shown below:





Using the Patterns on Images

Using the line pattern

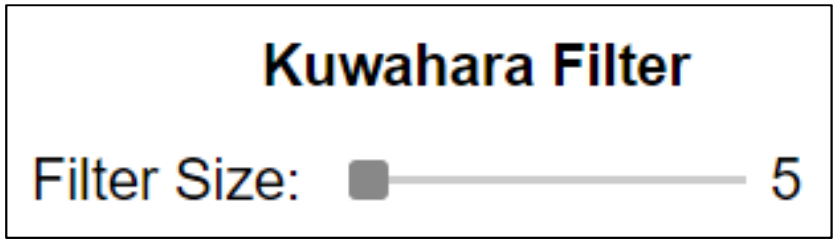
Using the diamond checker pattern

Bottom Layer – Colours

- At this layer, the colours of the image are adjusted based on two observations from comics:
  - The colours in a comic book are drawn using smooth and flat regions of colours
  - The colours are typically very colourful with very little gray/darkness
- The assignment attempts to simulate the above by first applying Kuwahara filter to the image and then mapping the colour to a different colour palette

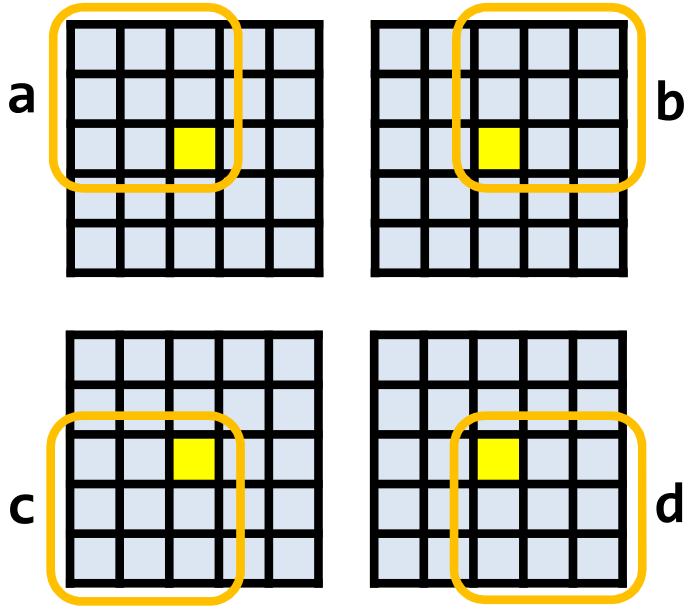
Colours – Your Task 1

- In the colours part, you need to process the pixels using three separate operations
- The first operation is to apply the Kuwahara filter
  - The Kuwahara filter given in the `kuwahara` function in `kuwahara.js` is working fine for a 5x5 Kuwahara filter
  - However, the assignment allows you to change the size of the Kuwahara filter from 5 to 9 or 13
  - You need to change the code so that the filter works with any of the given sizes



The Kuwahara Filter

- The Kuwahara filter uses four sub-regions to decide which colour to put at the centre
- A 5x5 Kuwahara filter has sub-regions of a size of 3x3
- When the size of the Kuwahara filter increases, the size of the sub-regions will also increase
- Since they only overlap at the centre pixel, you should be able to work out the size of the sub-regions

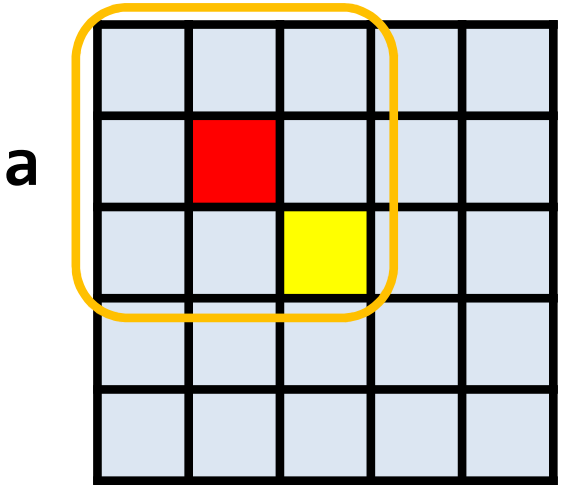


Changing the Kuwahara Code

- The `kuwahara` function in the starting code uses a fixed size for the sub-regions, i.e. 3x3
- You will need to go through the code, understand it and modify each line of code which is affected by a change of filter size
- This will also affects an internal function `regionStat` which computes the statistics of each region
- The size of the Kuwahara filter is given by the `size` parameter, which is not used in the code now

The regionStat Function

- The `regionStat` function calculates the statistics of a sub-region, given the centre position of the region
- For example, to use the function for region **a** as shown on the right, you will need to pass the location of the red pixel to the `regionStat` function



- Changing the size of the sub-regions will affect the positioning of the sub-regions

Using Different Kuwahara Filter Size

- The images below show the results of applying the Kuwahara filter with three different sizes:

Filter size = 5x5

Filter size = 9x9

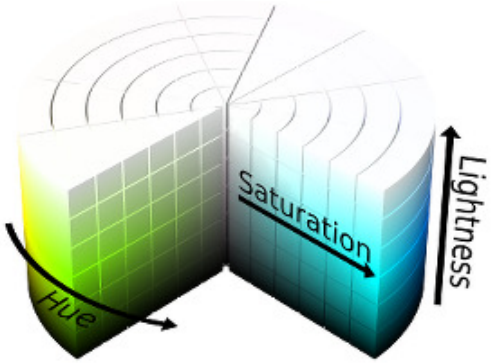
Filter size = 13x13

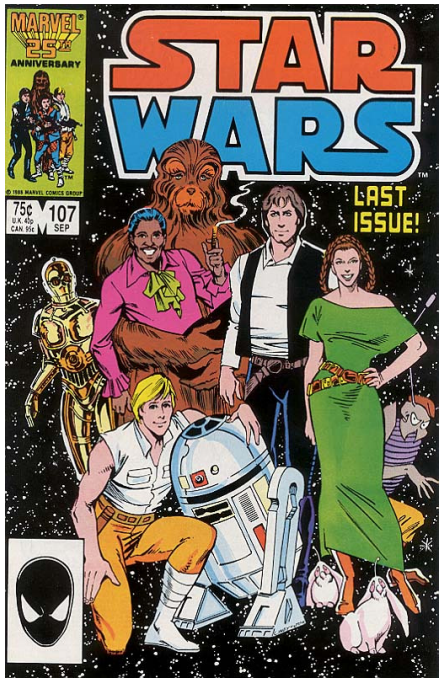
Colours – Your Task 2

- After using the Kuwahara filter, you will need to map the colours to a comic palette
- Before the mapping, you need to adjust the colours so that they become more colourful
  - You can do this by adjusting the saturation of the colours to increase their colourfulness
- The adjusted colour is then mapped to a colour in the colour palette of the 1980s comic books

Colours in Comic Books

- If we increase the saturation value of a colour in the HSL system, the colour will move away from gray and become more colourful





- Comics typically has colourful colours (apart from black) and therefore you can make an image look more like comics by increasing the saturation

Making Saturated Colours

- To make a more saturated colours, you need to:
  1. Convert the colours to HSL, for example:

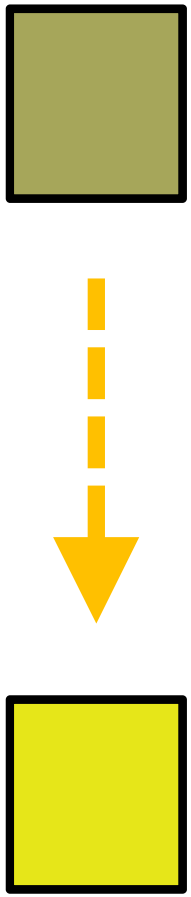
     RGB = 166, 166, 90 ⇨ HSL = 60, 30%, 50%

  2. Increase the S value, for example:

     HSL = 60, 30%, 50% ⇨ HSL = 60, 80%, 50%

  3. Convert the adjusted colours back to RGB, for example:

     HSL = 60, 80%, 50% ⇨ RGB = 230, 230, 25



Increasing the Saturation

- You control the adjustment using a factor from 1 to 10

- The saturation value of a colour is then adjusted by multiplying the saturation with the factor
- If the factor is 1, the colour will not change at all; if it is 2, the colour will have its saturation doubled (but remember the maximum saturation is 1)
- In the code, you can convert between colour systems using the colour conversion functions given in `imageproc.js`

Images with Different Saturation Adjustment

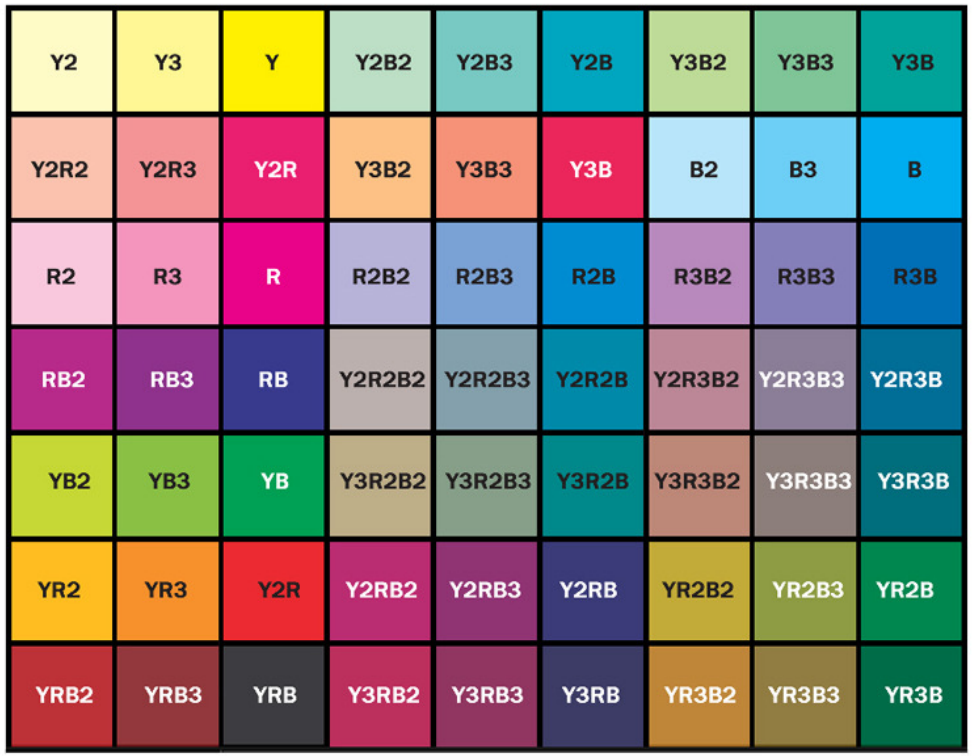
- Let's see an image with the saturation adjusted by a factor of 2, 5, and 8:

*Saturation * 2* *Saturation * 5* *Saturation * 8*

- After increasing the saturation, the 'dullness' of the colours has gone and replaced by 'colourfulness'

Colours – Your Task 3

- After adjusting the image colours, you can map the colours to the ones in a comic colour palette
- The colour palette the assignment used is shown on the right
- It contains the colours that can be produced by the comics printers around the 1980s



From http://facweb.cs.depaul.edu/sgrais/comics_color.htm

The Colour Palette

- The RGB values of the palette have been put in the `palette` array in `comicpalette.js`
- In addition to the colours shown on the previous slide, black and white are also appended to the array
- Given a pixel colour, you need to find the closest approximation of the colour from the comic colour palette

```
var palette = [
    [254, 251, 198],
    [255, 247, 149],
    [255, 240,   1],
            :
            :
            :
    [  0, 108,  72],
    [  0,   0,   0],
    [255, 255, 255],
];
```

Finding the Approximation of a Colour

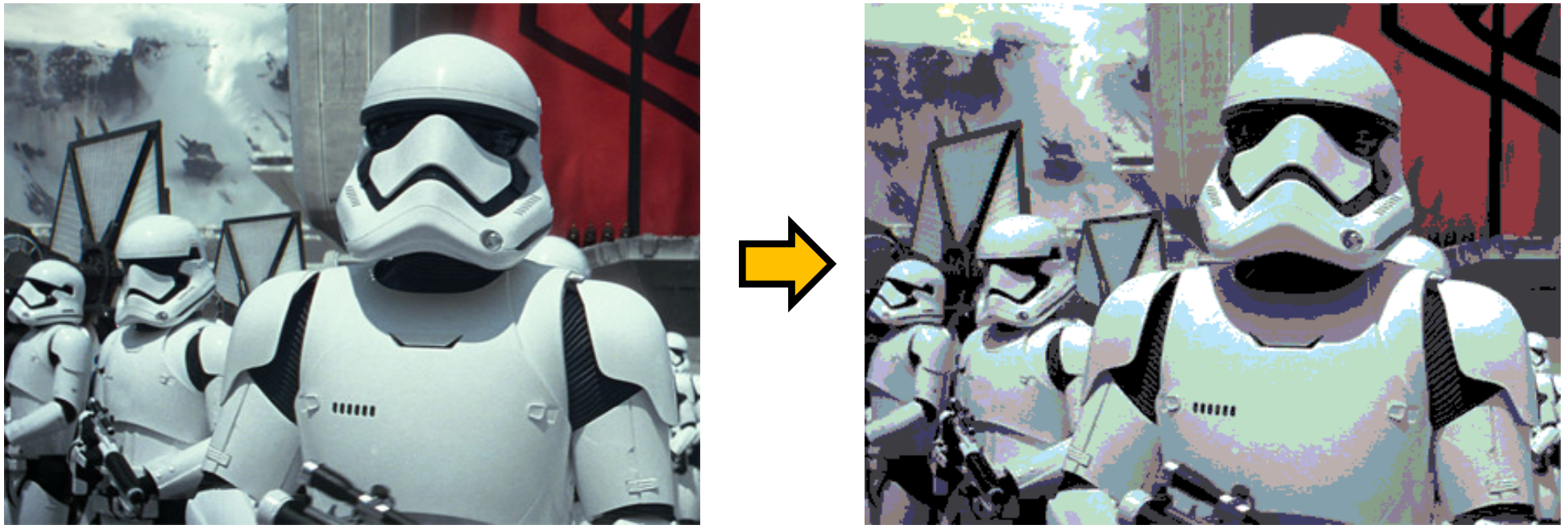
- You can find the approximation of a colour from the colour palette using ‘colour distance’
- The distance between two colours, A and B, is simply the Euclidean distance in the RGB space, i.e.

$$distance = \sqrt{(A_{red} - B_{red})^2 + (A_{green} - B_{green})^2 + (A_{blue} - B_{blue})^2}$$

- The approximation of a pixel colour is then the colour in the palette which is closest to the pixel colour, i.e. going through the palette to find the closest one

Using the Comic Palette

- Mapping the colours of an image to the comic palette will give you quantization error as expected



Note that Kuwahara filter and saturation adjustment have not been applied in the image above

Using Kuwahara and Colour Palettes

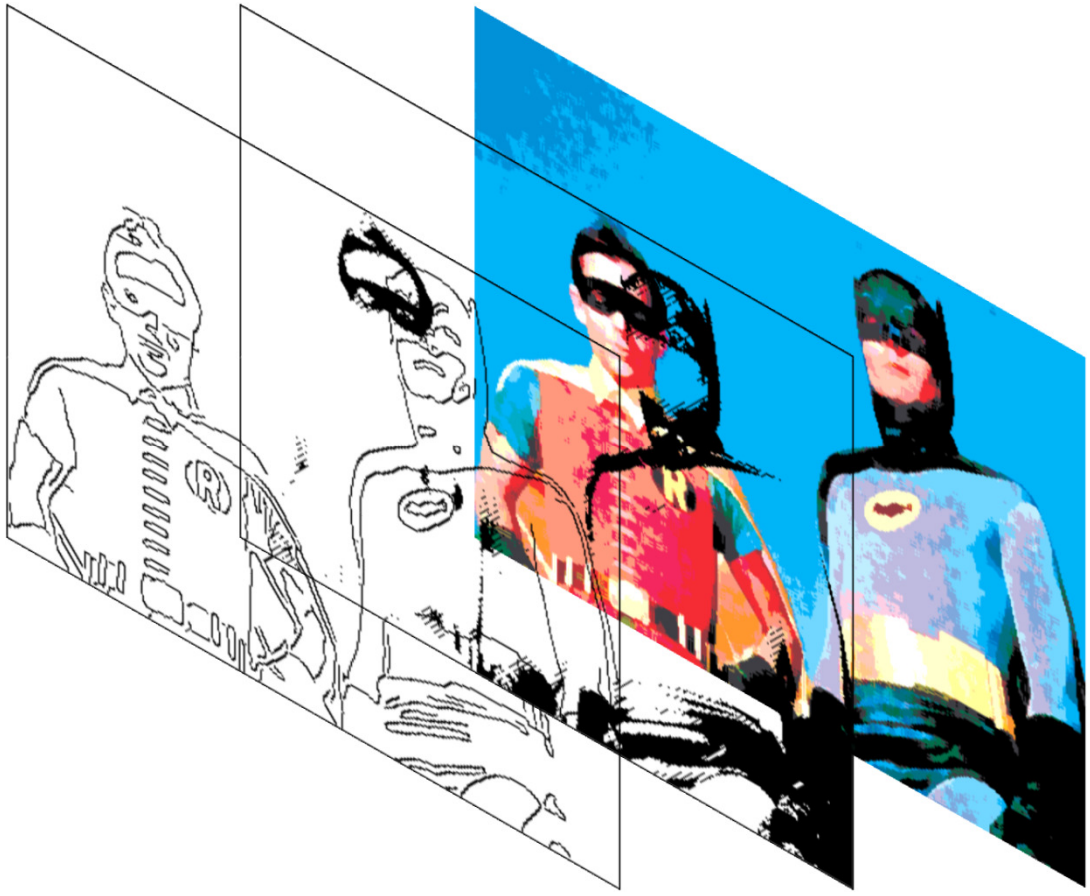
- The assignment system can apply Kuwahara filter to the image and then do the colour mapping, or they can be done separately
- The image on the right shows the image processed with both Kuwahara filter and colour palette mapping

Combining the Three Layers

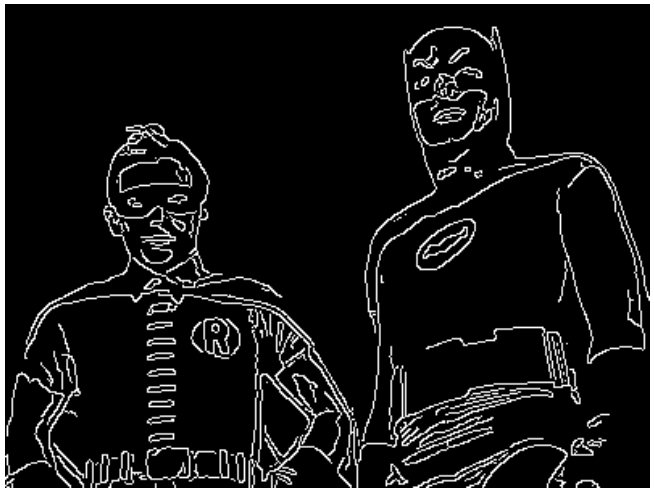
- We have only looked at the effects of each layer so far
- The COMICification process combines all three layers to form one single image
- However, you cannot simply add the three layers together because you only need the edge pixels and the shade pixels from the first two layers
- In addition to that, the edges and shades are always displayed using black colour

Combining the Layers – Your Task

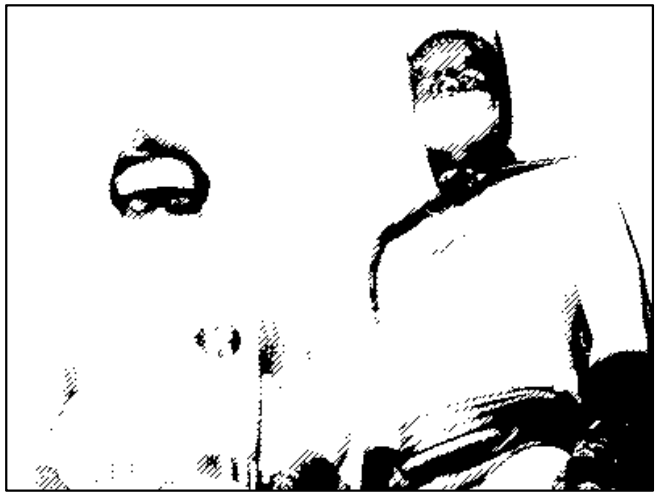
- The starting code only displays the result of each layer as it is
- You have to change the way the top two layers (outlines and shades) are blended into the bottom layer
- You will show only the black edges and shades



Showing the Edges and Shades



- For the edges:
  - The edges are stored as white in the result of the Canny edge detection
  - Therefore, you draw the white pixels as black but ignore the rest

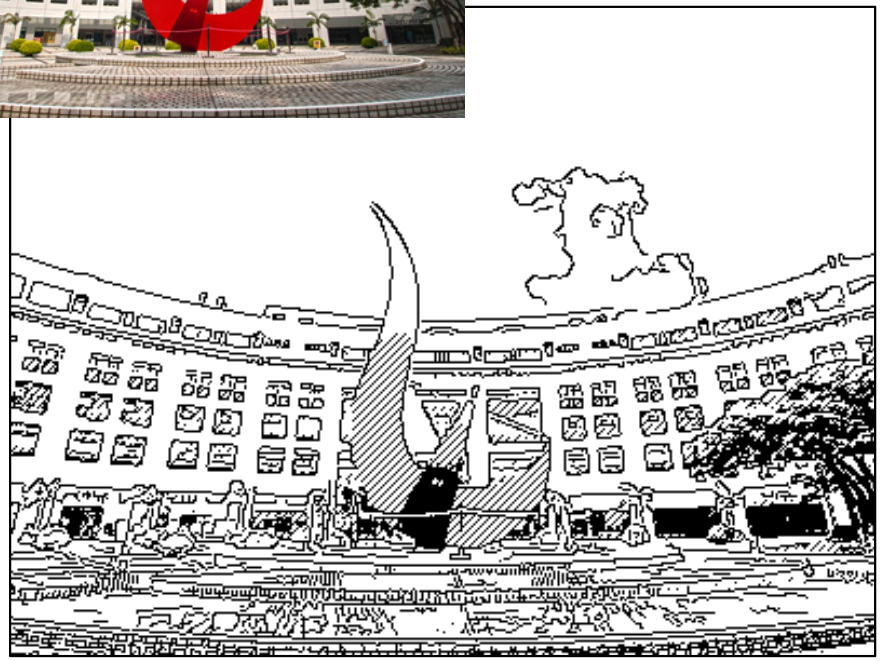


- For the shades:
  - The dithering result uses black to show completely black regions and the patterns for dark regions
  - Therefore, you draw the black pixels but ignore the rest

Final Results

- Here are some results:

Outlines + Shades



Outlines + Shades + Kuwahara

Final Results

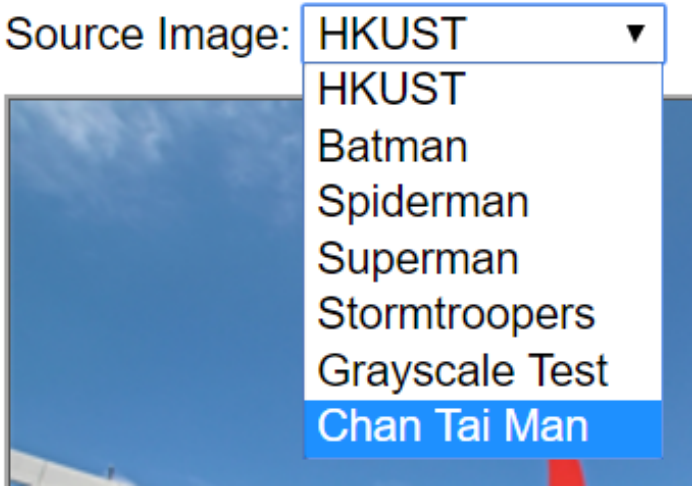
- More results:

Outlines + Comic Colours

Outlines + Shades + Comic Colours

Your Own Image

- You need to add an image of yourself to the list of source images that can be used by the system
- Here are the requirements of your image:
  - It must clearly show your face in the middle
  - It has a dimension of 400x300
  - The image file is called `me.png`
  - The image is shown using your name in the list; for example, if you are Chan Tai Man, your image will be listed like this:



Marking Scheme Overview

- Top layer – Outlines
  - Gaussian kernel generation 15%
  - Using Canny edge detection with different kernel sizes 5%
- Middle layer – Shades
  - Using line pattern 10%
  - Using diamond checker pattern 10%
- Bottom layer – Colours
  - Using Kuwahara filter with different filter sizes 15%
  - Making saturated colours 10%
  - Mapping colours to the comic colour palette 15%
- Combination of the layers 15%
- Your Image 5%

Submission

- The deadline of the assignment is:

8pm, Saturday, 23 Mar 2019

- To submit your assignment:
  - You need to put **everything** (HTML file, CSS files, JavaScript files and image files) into a zip file called `<your ITSC account>_a1.zip`
  - For example, if your ITSC account is *johnc*, you will put your files into `johnc_a1.zip`
  - You can then submit the zip file through canvas