Image Processing Operations

Digital Image Processing

“Operations on Pixels” e.g. Photoshop / Paintshop Pro – filters – image adjustments – image effects

Image Processing Motivations

- Process image for the human
  - Improve for human interpretation
    - Clarification, intensification
    - Abstraction
    - Aesthetics
- Process image for the machine
  - Storage and transmission
  - Representation for machine perception

Typical Image Processing Operations

- Geometric Transforms
- Colour operations
- Digital Compositing
- Interpolation, mosaicing / demosaicing
- Image editing
- Image registration, differencing and morphing
- Image segmentation, object recognition
- Tone mapping, dynamic range extension

Compositing

Images © Hongkiat Lim (http://www.hongkiat.com/blog/30-nicest-photoshop-photo-effects-part-ii/)

http://www.cs.northwestern.edu/~holger/Research/VideoAbstraction/?page=trailer

Image Processing
Postprocessing

Real-time Postprocessing Shader © 2013, Shock. [Video URL]

Image Processing
- Sub field of Signal processing / Digital Signal Processing (DSP)
- Also related to video processing

This lecture we’ll look at some relatively simple operations involving
- Moving pixels
- Per pixel operations
  - Based on previous pixel value
  - Based on multiple pixels values (kernel)
- Some combo operations
- Image processing in Processing

Image data
- Images are made up of data in the form of a value that changes depending on the position in 2D space
  - An image is essentially a table with rows (x) and columns (y)

  ![Image Data Diagram]  
  - Think of it as a mechanism that returns something (a value) for every x,y position you give it
  - Change x and y and you’ll get a new value for this function
  - The value is an intensity score (for colour images: 3 scores for red, green and blue)

Working with image data
- Typical operations
  - Get value of pixel(x, y)
  - Calculate values based on pixel(x, y)
  - Assign a value for pixel(x, y)
  - Change the value of pixel(x, y)
  - ... Or a combination of these

Pixel functions
- 1 to 1 operations for the same pixel
  - e.g. Gray scale, threshold, brightness, contrast, colorize
Pixel functions

1 to 1 operations with change in x, y: rotate, mirror, flip

1 to (multiple images)
   to 1: blend, mask, average

1 to many functions: scale

Many to 1 (kernel functions): blur, smooth, edge enhance

A recurring type of operation is one that takes a neighbourhood (or kernel) of a certain size to affect the colour of a single pixel.

Computer Vision

"There is a human face in this picture."
"That is a girl face."
"The face is in position x' y'"

Computer Graphics

Create pixels to represent an object.

Some Operations From Last Week

In Brief
Simple bit-copy functions

- Simply juggling pixels around can achieve some useful effects e.g.

  - Rotating an image by 90 degrees
    
    \[
    \text{set}(x, y, \text{img.get}(y, x));
    \]
  - Mirroring an image
    
    \[
    \text{set}(x, y, \text{img.get}(\text{width-x, } y));
    \]
  - Flipping an image
    
    \[
    \text{set}(x, y, \text{img.get}(x, \text{height-y)});
    \]
  - Drawing an image twice as big
    
    \[
    \text{set}(x*2, y*2, \text{img.get}(x, y));
    \text{set}(x*2+1, y*2, \text{img.get}(x, y));
    \text{set}(x*2, y*2+1, \text{img.get}(x, y));
    \text{set}(x*2+1, y*2+1, \text{img.get}(x, y));
    \]

Operations on pixels: greyscale

- An average or weighted average of the RGB intensities

  \[
  \begin{align*}
  &\text{Unweighted average:} \\
  &\quad CC = (R+G+B)/3 \\
  &\text{Commonly used weighting:} \\
  &\quad CC = 0.3*R + 0.59*G + 0.11*B
  \end{align*}
  \]

  \[
  C = \text{im1.get}(x, y);
  \text{float r} = 0.3 \times \text{red}(C);
  \text{float g} = 0.59 \times \text{green}(C);
  \text{float b} = 0.11 \times \text{blue}(C);
  \text{cc = color}(r, g, b);
  \]

Negative image

- If 255 is full intensity then the negative value of any intensity \( c \) is \( 255 - c \)

  \[
  \begin{align*}
  &\text{e.g. the opposite of “red” (240, 0, 0) is (255-240, 255-0, 255-0)} \\
  &\text{i.e. (15, 255, 255) cyan}
  \end{align*}
  \]

Thresholding

- Traditionally describes the operation of reducing an image to a 2 colour bitmap;

  - All pixels less than a certain threshold level of brightness are drawn as black. The rest are drawn white

  \[
  \begin{align*}
  &\text{float g} = 0.3 \times \text{red}(c) + 0.59 \times \text{green}(c) + 0.11 \times \text{blue}(c); \\
  &\text{if (g>128) cc = 255;} \\
  &\text{else} \\
  &\text{cc = 0};
  \end{align*}
  \]

Threshold

- Thresholding: pick a certain brightness threshold value, \( T \). If brightness of current pixel is above \( T \) set brightness to 255 (white) Otherwise, set brightness to 0 (black)

Thresholding

- We can increase/decrease the threshold value to highlight areas of bright contrast at the expense of variations in dark areas, or vice versa.
Thresholding

- Alternatively: Set several discrete limits
  - e.g. 0, 50, 100, 150, 200
- All intensities are rounded down to the closest of these thresholds
- Some useful applications
  - E.g. mosaic effects, colour depth reduction

\[
\text{float } g = 0.3 \times \text{red}(c) + 0.59 \times \text{green}(c) + 0.11 \times \text{blue}(c);
\]
\[
\text{cc} = 55;
\]
\[
\text{if } (g > 50) \text{ cc} = 100;
\]
\[
\text{if } (g > 100) \text{ cc} = 150;
\]
\[
\text{if } (g > 150) \text{ cc} = 200;
\]
\[
\text{if } (g > 200) \text{ cc} = 255
\]

Multi-Thresholds

- Threshold levels – pick various values T1, T2, T3, T4, T5. Round brightness to the nearest Threshold value.

More Advanced Operations

- Multi-Thresholds
  - Threshold levels
  - Pick various values T1, T2, T3, T4, T5. Round brightness to the nearest Threshold value.

Halftoning

- A method for simulating a greater range of colours using patterns of ink spots of varying concentration.
- Originally used in print media.
- Works because our visual system picks up localised averaged intensities of colour.

Digital Halftones

- Simple half tone patterns replace thresholded pixels.

```
// sample code for just level 2:
if ((g > 100) && (g < 150))
{
  set(x*2, y*2, color(255));
  set(x*2+1, y*2, color(0));
  set(x*2, y*2+1, color(0);
  set(x*2+1, y*2+1, color(255));
}
```

- We can also apply a 3x3 pattern to support 9 different threshold values.

Full code provided at: [https://www.scss.tcd.ie/John.Dingliana/Processing/halftone/](https://www.scss.tcd.ie/John.Dingliana/Processing/halftone/)
Error Diffusion

- As we saw before, a pixel is thresholded (by rounding down its intensity to the nearest threshold level).
- The difference in actual intensity and threshold intensity is the error in the pixel.
  - I.e. it is actually rendered less bright than the original
  - Some brightness is lost in the overall image
- To compensate for this error we can distribute it over its neighbouring pixels.

Error Diffusion Example

- Instead of just rounding down, we note down how much we have rounded down by (this is called the error) and disperse the error amongst neighbouring pixels.

Dithering Patterns

- A more elegant way of halftoning
- On the left the dither pattern is twice the width and height of the original image
- Each original pixel is compared to a 2x2 block of pixels with the following values
  - 1 1 0 0
  - 0 0 1 1
- If the original value is higher than the value in the dither pixel, then it is enabled.

Halftoning vs. Error Diffusion

- Error = current pixel value minus thresholded value
- Change the pixel values of neighbouring pixels in the original image before they are processed.
  - a = a – (7/16) x Error
  - b = b – (3/16) x Error
  - c = c – (5/16) x Error
  - d = d – (1/16) x Error

Image is drawn from left to right so a common method is to diffuse the error to the right and below the current pixel.

Mosaics

- Threshold values can be used to select patterns, or even other images to replace small areas of the original image.

Comparision with halftoning: https://www.scss.tcd.ie/John.Dingliana/Processing/diffusion/
Edge detection

- How different is a pixel to its neighbours.
  - e.g. 5-point laplacian operator essentially compares a pixel to the average of its 4 adjacent neighbours

  \[
  \text{if } c(x, y) \text{ is the color of pixel at } x, y
  \]

  Then we calculate:

  \[
  4 \times c(x, y) - (c(x+1, y) + c(x-1, y) + c(x, y+1) + c(x, y-1))
  \]

  i.e. 4 x current colour − sum of neighbours

- If the pixel colour is the same as its neighbours the result is 0. Otherwise the value indicates the steepness of the change.

Smoothing / Blur

- Find the average colour of a current pixel and its neighbours
- Assign this colour to the pixel

Linear Smoothing / Blur

- Uniform Filter For each pixel \( (x, y) \), look at “nearby” pixels in a certain radius and average the values.

- Non-uniform filters take what is essentially a weighted average of nearby pixels
Edge Preserving Smooth

- The Kuwahara filter is one example of an edge preserving smooth
- For each pixel \((x,y)\) look at immediate adjacent regions
  - Find the variance i.e. difference between min and max intensity
  - For the region with the minimum variance
    Get its MEAN (average) colour and apply to \((x, y)\)

Image Arithmetic

- For each pixel, the RGB colours from one image can be added, subtracted, multiplied, averaged etc.. with RGB from the other
- Useful, amongst other things for combining some effects

Image Difference

- If \(a\) and \(b\) are pixels colors in imageA and imageB respectively
- And \(c\) is the color of the blended images
- then
  \[ c = aT + b(1-T) \]
- Different values of the variable \(T\) (between 0 and 1) will increase the emphasis on imageA or imageB
- We can do a smooth fade from imageA to imageB by gradually increasing \(T\)
Blending Images

```java
/**
   * blends between two images
   */
PImage img1;
PImage img2;
float t = 0;
void setup()
{
  size(800, 800);
  img1 = loadImage("image.jpg");
  img2 = loadImage("image2.jpg");
}
void draw()
{
  color c1, c2, cc;
  float r1, g1, b1, r2, g2, b2, rr, gg, bb;
  t = t+0.01; if (t>1) t = 0;
  for (int i=0; i<img1.width; i++)
    for (int j=0; j<img1.height; j++)
    {
      c1 = img1.get(i, j);
      c2 = img2.get(i, j);
      r1 = red(c1); r2=red(c2);
      b1= blue(c1); b2= blue(c2);
      g1 = green(c1); g2 = green(c2);
      rr = t*r1  +  (1-t)*r2;
      gg = t*b1  +  (1-t)*b2;
      bb = t*g1  +  (1-t)*g2;
      cc = color(rr, gg, bb);
      set(i, j, cc);
    }
}
```

Processing Utilities

**filter()**

- The filter function provides an easy way to apply filters to the image
- The filter can be applied to what's drawn on screen
  - Draw img to screen
  - Then Blur the screen
- Or it can be applied to an image
  - Blur the image stored in img
  - Then draw img

```java
PImage img = loadImage("apples.jpg");
image(img, 0, 0);
filter(BLUR, 6);
```

**Arithmetic between an image and the frame-buffer:**

```java
blend()
```

- The blend() function in Processing allows you to do arithmetic (amongst other things)
- Arithmetic between an image and the frame-buffer:

```java
blend(img1, 0, 0, img2, 0, 0, 100, 100, BLEND_RGB_ADD);
```

**Edge Detection**

- The difference between an image and a blurred version of that image is an edge image (why?)

```java
filter(INVERT);
filter(BLUR, 5);
filter(INVERT);
```

- If a second parameter is provided it is a threshold value which should be between 0 and 1 (effectively controls the brightness)
- Reduces number of colours to a certain amount, the second parameter specifies how many colours are used

```java
filter(THRESHOLD, 0.995);
```

- Turns an image black and white.
  - If a second parameter is provided it is a threshold value which should be between 0 and 1 (effectively controls the brightness)
  - Reduces number of colours to a certain amount, the second parameter specifies how many colours are used
Image Transformations

- void rotate(float angle);
- void translate(float tx, float ty);
- void scale(float sx, float sy);

rotate( radians(30) );
translate( -200, -200 );
image( img1, 0, 0 );

Processing expects angles to be specified in radians. But you can convert degrees to radians using the radians() function.

Not actually that straightforward underneath:


A Rasterization Issue: to be discussed in more detail at a later lecture.

Lab 17: Image Operations #2

This lab is worth 3% of the CS7029 module
You must submit your solution on mymodule.tcd.ie by 23:59 Monday 13th March (you should submit even if incomplete)
Late submissions will incur a penalty (mark capped at 40%)

Goals of this Lab

- In this lab we will look at
  - Some image operations using the filter function in processing
  - Operations that combine multiple images
  - Using the above to create some advanced effect. Either one of the following:
    OPTION A: Dynamically transition between two images
    OPTION B: Enhance a static image by edge detection

Please Note

- From this lab on, we won’t give you direct instructions for each step of the solution. Instead you are expected to ...
  - Read the notes and documentation
  - Figure out the answer based on some high-level instructions

The intention is to get you to eventually understand the code and tools well enough to be able to plan and create novel effects from scratch

Optional HELPER NOTES are provided in grey slides (such as this) but you should try solving the problem without them if you can
- If you do get stuck, do not hesitate to ask for help in the lab or by email.

STEP 1: Built in Image Filters

- Load an image (we’ll call it img1 here) and apply some of the following operations one after another
  - Blur an image
  - Threshold an image to black and white
  - Erode an image
  - Dilate an image
  - Greyscale then Posterize an Image

- you should try them all but in your submission you just need to show that you can do at least two filter operations one after another on the same image
REMINDER: Basic set up

- Start a new sketch with at least a setup and draw function
- Find an image file, save it to your sketch folder
- Load the image
- If you don't already know how to do this, read the documentation on loadImage [link] OR see previous labs

```java
PImage img;
void setup() {
  img = loadImage("download.jpg");
  size(800, 800);
}
void draw() {
  image(img, 0, 0);
}
```

HELPERS NOTES

- Read the documentation on the filter function in Processing
  - https://processing.org/reference/Processing
- Note: you can EITHER
  - draw the image to the viewport and filter the viewport image
    ```java
    image(img1, 0, 0);
    filter(BLUR, 2);
    ```
  - OR filter the image itself and then draw it
    ```java
    img1.filter(BLUR, 2);
    image(img1, 0, 0);
    ```
- The first technique is simpler but the second is useful if you want to do further operations on the image.
- Your task is to apply more than one filter to the same image.

**filter()**

- The filter function provides an easy way to apply filters to the image
- The filter can be applied to what's drawn on screen
  - Draw img to screen
  - Then Blur the screen
- OR it can be applied to an image
  - Blur the image stored in img
  - Then draw img

```
filter(INVERT);
filter(ERODE);
filter(TRESHOLD);
filter(DILATE);
filter(POSTERIZE, 4);
```

HELPERS NOTES

- Increases the dark areas of an image
- Increases the light areas of an image
- Reduces number of colours to a certain amount, the second parameter specifies how many colours are used

**STEP 2: Image Arithmetic**

- Load a second image (lets call it img2) and Attempt one of the following:
  - OPTION B: find the difference between img1 and img2 by subtracting one from the other
- OPTIONAL: Find the difference between the following two images
  - https://www.scss.tcd.ie/John.Dingliana/cs7029/Proc/disney1.png
  - https://www.scss.tcd.ie/John.Dingliana/cs7029/Proc/disney2.png

HELPERS NOTES

- You can either use the blend function provided in processing or use two nested for loops to do this on a pixel-by-pixel basis
- Using blend will require less lines of code, but you may be able to understand better exactly what's going on if you do it on a pixel-by-pixel basis (see next slide)
- Using blend():
  - Read the function description: https://processing.org/reference/blend.html
  - Blend can be applied to the image drawn on the screen or directly to an image stored in memory e.g:
    ```java
    name of destination image.blend (name of source image, top left corner, width, height, DIFFERENCE);
    ```
The `blend()` function in Processing allows you to do arithmetic (amongst other things).

Arithmetic between an image and the frame-buffer:

```java
blend(image, 0, 0, 100, 100, DIFFERENCE)
```

- Name of a `PImage`
- Top left corner x, y, width, height of rectangle on image
- Top left corner x, y, width, height of rectangle in frame buffer image
- Blend mode

**STEP 3: Edge detection**

Use a combination of filters to create an edge detection filter (this should resemble a line drawing).

Suggested steps:

A. Load the same image twice, storing it in `img1` and `img2`
B. Apply a blur to `img1`
C. Subtract `img1` from `img2`, store it in `img2`
D. Invert `img2`
E. Threshold `img2`
F. Draw `img2`

**Further Improvements**

You will need to adjust the filter parameters for the BLUR and THRESHOLD filters in order to get the best effect:

- The best values are dependent on each individual image
- But I suggest a BLUR value less than 2
  - e.g. `img2.filter(BLUR, 1.8)`
- And a high THRESHOLD value
  - e.g. `img2.filter(THRESHOLD, 0.99)`

**OPTIONAL:** If all works, see if you can combine (add) the edge image with a "POSTERIZED" version of `img1`

This should give it roughly a cartoon drawing look.

**Submission**

- Submit your sketch and any required images
- In addition submit a screenshot of the finished edge-detection image