In this lecture...

- Recap: how digital images are stored
- Image Processing Part 1: Pixel Operations
- Very brief look at Classes (how to use one)
- Processing Specific:
  - getting the color of a pixels
  - setting the color of a pixel
  - Transforming images
Operations on Images

Recall Vector vs. Raster Images

Vector graphic images constructed from vector primitives. Easier to work with within the program. Processing has support of SVG.

Raster Images made up of pixels. Processing has an inbuilt class called PImage. The topic of this lecture.

Raster Image

- An image made up of many small regularly placed cells called pixels (picture elements)
- Stored as an array of numerical values commonly called a pixelmap, pixmap or bitmap

Image Files

- An image is an array of pixels
- Each pixel is a variable which has an RGB value.

- Images Files usually have
  - header: consisting atleast of the width and height of the image, also some other specific details
  - body: consisting pixel array that is just a simple list of colours

Example: PPM

- PPM (Portable PixelMap) is a very simple file format

Header consists of
- "P3" symbol indicating this is a PPM file (version 3)
- Width and height of image
- Colour depth

Body consists of bytes.
Each number is red green then blue.
Image Resolution

- Different resolution images obviously yield better quality images particularly when resized
- Must find a balance: low-res images look blocky BUT high-res images are expensive on memory, transmission time...

Practical Example

![Practical Example Image]

Images in Processing

- **PImage**
  
  PImage b;
  
  b = loadImage("teapot.jpg");
  
  image(b, 0, 0);

Brief aside classes

- Teapot is a CLASS of things that has...
  - A handle, a spout, a lid
  - A colour, a size, some contents
- Teapot’s can be...
  - Filled, poured
- My_Teapot is an OBJECT of the teapot class
  - My teapot’s colour is purple
  - My teapot is large
  - My teapot is full (of tea)

Brief Aside: Classes

- Classes are complex structures usually associated with some data, and some functions specific to the class
  - Data a.k.a. members, fields, variables, attributes
  - Functions a.k.a. methods

```java
class cross {
  float centerx;
  float centery;
  float c_size;
  color c_colour;
  void draw() {
    //set the stroke colour
    stroke(c_colour);
    //draw a couple of lines
    line(centerx - c_size, centerx - c_size, centerx + c_size, centery + c_size);
    line(centerx + c_size, centerx - c_size, centerx - c_size, centery + c_size);
  }
}

cross mycross;
mycross = new cross();
mycross.centers = 100;
mycross.centary = 100;
mycross.c_size = 30;
mycross.c_colour = color(255, 0, 0);
mycross.draw();
```
Takeaway Messages

- Classes are a very powerful programming mechanism
  - Almost all modern programming uses classes or something like them
  - Developing classes effectively can take some experience
- ... on the other hand USING Classes is relatively easy (by design)
  - Only a little bit different from using functions and variables

The main differences is that we prefix the class' function and variable names by the name of the object

PImage variables

- width : Image width
- height : Image height
- pixels[] : array containing the colour of the pixels

Loading and drawing images

- Declare a new Image using the following code:
  - PImage img;
  - img = loadImage ("picture.gif");
  - image (img, 0, 0);

- Other image related functions:
  - img.set(10, 10, color(255, 0, 0));
  - color c = img.get(20, 20);

- Setting the colour of a pixel in an Image:
  - img.set(10, 10, color(255, 0, 0));
  - color c = (img).get(20, 20);

- Setting the colour of a pixel in the viewport (just leave out the image name)
  - set(10, 10, color(255, 0, 0));
  - color c = get(20, 20);

Easier to follow (go to this version)

One way to do it (quicker to write)

PImage im;

im = loadImage("laDefense.jpg");

image(im, 0, 0);

for (int i = 0; i < im.width * im.height; i++)
  im.pixels[i] = color(255, 0, 0);

Working with PImage

- Although we can deal with the image as the one-dimensional array (which it is), a few helper functions can make a program easier to read and work with. The above two pieces of code should achieve the same thing (essentially turn all the pixels of the image red)

The Viewport is also an Image

- Slightly confusing bit: These functions also apply to the program window (i.e. the drawing area or canvas) which behaves like an image

- Setting the colour of a pixel in an Image:
  - img.set(10, 10, color(255, 0, 0));
  - color c = (img).get(20, 20);

- Setting the colour of a pixel in the viewport (just leave out the image name)
  - set(10, 10, color(255, 0, 0));
  - color c = get(20, 20);

Note the difference (circled in red)
Sample program to iterate through an image

```cpp
color newcolor;
newcolor = color (255, 0, 0);
for (int x=0; x<im.width; x++)
for (int y=0; y<im.height; y++)
{
  im.set (x,y, newcolor);
}
```

N.B. the drawing window itself can be considered an image (remove `im` in the above code)

Sample Code

• The following sample code is used for the examples in the rest of the lecture.

• What we essentially want to do in image processing is calculate a new colour `cc` for each pixel.

• This is based on the original colour `c` and some pixel operations which we will define.

```cpp
PImage im;
void setup()
{
im = loadImage("Image8.jpg");
size(600, 600);
noLoop();
}
void draw()
{
color oldcolor, newcolor;
for (int x=0; x<im.width; x++)
for (int y=0; y<im.height; y++)
{
  oldcolor = im.get (x,y);
  //newcolor = ? Provide operation here
  set (x, y, newcolor);
  //set pixel on the framebuffer
}
}
```

In the following slides, we will provide additional code that you can plug into this sample program to get the desired effect.

Sample Code

• We change an image by applying a Procedure to some or all of its pixels.

• Examples:
  • Greyscale the Image
  • Clear Image
  • Threshold Image
  • Mirror the image
  • Line Drawing
  • Anti-aliasing
  • Filters
    Edge detect
    Blur/Smooth

Operations on pixels: greyscale

• An average or weighted average of the RGB intensities

• Just make sure the final result has values in the range from 255 to 255

  • Simple way to do this is just average the 3 components:
    ```cpp
    float grey = (R+G+B) / 3;
    ```

  • Commonly used weighting (optional)
    ```cpp
    grey = 0.3*R + 0.59*G + 0.11*B
    ```

```cpp
oldcolor = im.get (x, y);
float R = red (oldcolor);
float G = green (oldcolor);
float B = blue (oldcolor);
float grey = (R+G+B) / 3;
newcolor = color (grey, grey, grey);
```

Negative image

• If 255 is full intensity then the inverse of any intensity `c` is 255 – `c`

  • e.g. the opposite of “red” (240, 0, 0) is (255-240, 255-0, 255-0) i.e. (15, 255, 255) cyan

```cpp
oldcolor = get (x, y);
float r = 255 - red (oldcolor);
float g = 255 - green (oldcolor);
float b = 255 - blue (oldcolor);
newcolor = color (r, g, b);
```
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.

\[
\text{float } g = \text{red(oldcolor)} + \text{green(oldcolor)} + \text{blue(oldcolor)};
\]

\[
\text{float } t;
\]

\[
\text{if } (g > 128) \text{ t } = 255; \\
\text{else} \\
\text{t } = 0;
\]

\[
\text{cc = color(t, t, t)};
\]

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

**Input**  
**Output**

- Low threshold: All these levels of brightness are converted to white.
- A “halfway” threshold.
- High threshold: all but the brightest areas are rendered as black.

Thresholding Several Levels of Grey

- Alternatively: Set several discrete limits
- E.g. 0, 50, 100, 150, 200
- All intensities are rounded down to the closest of these thresholds

\[
\text{float } g = (\text{red(oldcolor)} + \text{green(oldcolor)} + \text{blue(oldcolor)}) / 3;
\]

\[
\text{newcolor } = \text{color(55, 55, 55)};
\]

\[
\text{if } (g > 50) \text{ newcolor } = \text{color(100, 100, 100)};
\]

\[
\text{if } (g > 100) \text{ newcolor } = \text{color(150, 150, 150)};
\]

\[
\text{if } (g > 150) \text{ newcolor } = \text{color(200, 200, 200)};
\]

\[
\text{if } (g > 200) \text{ newcolor } = \text{color(255, 255, 255)};
\]

Simple bit-copy functions

- Simply juggling pixels around can achieve some useful effects e.g.
  - Rotating an Image by 90 degrees
  - Mirroring an Image
  - Flipping an Image
  - Drawing an image twice as big

\[
\text{set(x, y, im.get(y, x))};
\]

\[
\text{set(x, y, im.get(width-x, y))};
\]

\[
\text{set(x, y, im.get(x, height-y))};
\]

\[
\text{set(x*2, y*2, im.get(x, y))};
\]

\[
\text{set(x*2+1, y*2, im.get(x, y))};
\]

\[
\text{set(x*2, y*2+1, im.get(x, y))};
\]

\[
\text{set(x*2+1, y*2+1, im.get(x, y))};
\]

(repeat these for all pixels in the image by placing this in the nested for loop)

Sample Code

```cpp
PImage im;
void setup()
{
  im = loadImage("Image8.jpg");
  size(im.width, im.height);
  noLoop();
}
void draw()
{
  color c, cc;
  for(int x=0; x<im.width; x++)
    for(int y=0; y<im.height; y++)
      {
        c = im.get(x, y);
        set(x, y, c);
        set(x+1, y, c);
        set(x, y+1, c);
        set(x+1, y+1, c);
      }
}
```

Sample Code

```cpp
PImage im;
void setup()
{
  im = loadImage("Image8.jpg");
  size(im.width*2, im.height*2);
  noLoop();
}
void draw()
{
  color c;
  for(int x=0; x<im.width; x++)
    for(int y=0; y<im.height; y++)
      {
        c = im.get(x, y);
        set(x, y, c);
        set(x+1, y, c);
        set(x, y+1, c);
        set(x+1, y+1, c);
      }
}
```

(repeat these for all pixels in the image by placing this in the nested for loop)

Sample Code

```cpp
PImage im;
void setup()
{
  im = loadImage("Image8.jpg");
  size(im.width, im.height);
  noLoop();
}
void draw()
{
  color c;
  for(int x=0; x<im.width; x++)
    for(int y=0; y<im.height; y++)
      {
        c = im.get(x, y);
        set(x, y, c);
      }
  set(x, y, c);
  set(x+1, y, c);
  set(x, y+1, c);
  set(x+1, y+1, c);
}
```

(repeat these for all pixels in the image by placing this in the nested for loop)