Image Manipulation: Filters and Convolutions

Elements of Graphics
CS324e
Per-Pixel Manipulation

- Individual pixels do not influence neighboring pixels
- Possible modifications include shifts in:
  - Color
  - Brightness
  - Opacity
Grayscale

- RGB channels of pixel have the same value
- Content of image expressed through color value rather than hue or saturation
- How might we find a single value that captures the information of three color channels?
High Contrast

- Increase or decrease value of RGB channels based on pixel brightness
- Changes in value across image further emphasized
- How might we make some pixels darker and some pixels brighter?
HSV/HSB

- Hue-Saturation-Value commonly used in digital color pickers
- Hue: pure color
- Saturation: amount of color
- Value (Brightness): darkness or lightness of color
colorMode(model, range1, range2, range3)

Examples:
colorMode(RGB, 255, 255, 255);
colorMode(HSB, 360, 100, 100);
colorMode(RGB, 1.0, 1.0, 1.0);
colorMode(HSB, 100);
RGB Methods

- Extract red, green, and blue channels from a pixel:
  - red(color c)
  - green(color c)
  - blue(color c)
HSB Methods

- Extract hue, saturation and brightness from a pixel:
  - \texttt{hue}(\texttt{color c})
  - \texttt{saturation}(\texttt{color c})
  - \texttt{brightness}(\texttt{color c})
Consider...

colorMode(RGB, 255, 255, 255);
fill(50, 100, 100);
rect(0, 0, 50, 50); //Rect1
colorMode(HSB, 360, 100, 100);
fill(50, 100, 100);
rect(50, 50, 50, 50); //Rect2
Image Kernels

- Also called convolution matrix or mask
- Matrix used to **convolve** kernel values with image values
  - Square and small (3x3, 5x5 etc)
  - The larger the matrix, the more local information is lost
- Allows for "area" effects such as blur, sharpening and edge-detection
- Note: **not a matrix multiply**
Convolution

- Matrix convolution
  1. Multiplication of corresponding cells
  2. Summation of these values

\[
\begin{bmatrix}
39 & 33 & 35 & 36 & 31 \\
35 & 34 & 36 & 33 & 34 \\
34 & 33 & 36 & 34 & 32 \\
32 & 36 & 35 & 36 & 35 \\
33 & 31 & 34 & 31 & 32 \\
\end{bmatrix}
\times
\begin{bmatrix}
1 & 2 & 1 \\
2 & 4 & 2 \\
1 & 2 & 1 \\
\end{bmatrix}
= \begin{bmatrix}
1 \cdot 34 & 2 \cdot 36 & 1 \cdot 33 \\
2 \cdot 33 & 4 \cdot 36 & 2 \cdot 34 \\
1 \cdot 36 & 2 \cdot 35 & 1 \cdot 36 \\
\end{bmatrix}
= \{139 + 278 + 142\} = 559
\]
Kernel Application

- Each pixel has the convolution matrix applied to it
- Value is stored at corresponding location
What is the convolution output for the center cell of the highlighted 3x3 cells?

- 16
- 17
- 18
- 19
- 20
Hands-on: Understanding Convolutions

❖ Today’s activities:

1. Complete your tint method if it’s not finished (do not resubmit it)
2. Experiment with colorMode, switching between RGB and HSB
3. Use RGB and HSB methods to extract color’s information
4. Construct this kernel in Processing:

```
0   -1   0
-1   5   -1
0   -1   0
```
Applying Convolutions

Original Image

Sharpened Image
Kernel Traversal

- How can we traverse both the image pixels and the cells of the kernel?
Accessing pixel neighborhoods

Consider the call:

```java
int index = (x + i - 1) + img.width*(y + j - 1);
```

Provides an offset to the target pixel

Based on i and j values, offset reaches certain number of neighboring pixels in the x and y directions
Sharpen Example Code

```c
float[][[]] matrix = {{0, -1, 0}, {-1, 5, -1}, {0, -1, 0}};
/* Access individual pixel location (x, y) and initialize rgb floats to store new color channel values */
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        int index = (x + i - 1) + img.width*(y + j - 1);
        red += red(img.pixels[index]) * matrix[i][j];
        ... //Perform convolution on green and blue color channels
    }
}
red = constrain(abs(red), 0, 255);
... //Clamp green and blue values
```
Revisiting the Convolution Matrix

- Each pixel has the convolution matrix applied to it
- Value is stored at corresponding location

What happens if we store values in existing image?
Intermediate Buffer

- Array of pixels that matches the size of the image
- Provides “safe” location for storing image data
- Allows program to preserve original image data if necessary
- Buffering is also a common trick to increase speed of rendering (aka double buffering)
Creating a Buffer

- Can create a duplicate image:
  - `loadImage(image_file);` //load twice

- Or can create a blank image:
  - `createImage(width, height, ARGB);`

- Then copy pixel values from one buffer to another
  - `copy(img, x, y, width, height, x, y, width, height);`
Copying an Image

❖ Shallow copy:

PImage img1;

PImage img2 = img1;

❖ Deep copy*:

    img2.copy(img1, 0, 0, img1.width, img1.height, 0, 0, img2.width, img2.height);

* Note that img2 must be initialized (either loaded from image or created as a blank image) before a deep copy will work!
Box Blur

- Pixel value is based on average of its neighborhood:

\[
\frac{1}{9} \times \begin{bmatrix}
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1 \\
\end{bmatrix},
\begin{bmatrix}
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1 \\
\end{bmatrix},
\begin{bmatrix}
1 & 1 & 1 \\
1 & 1 & 1 \\
1 & 1 & 1 \\
\end{bmatrix}
\]

or approximately:

\[
\begin{bmatrix}
0.11 & 0.11 & 0.11 \\
0.11 & 0.11 & 0.11 \\
0.11 & 0.11 & 0.11 \\
\end{bmatrix},
\begin{bmatrix}
0.11 & 0.11 & 0.11 \\
0.11 & 0.11 & 0.11 \\
0.11 & 0.11 & 0.11 \\
\end{bmatrix},
\begin{bmatrix}
0.11 & 0.11 & 0.11 \\
0.11 & 0.11 & 0.11 \\
0.11 & 0.11 & 0.11 \\
\end{bmatrix}
\]
Gaussian Blur

- Use of Gaussian function for convolution:

\[ G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2 + y^2}{2\sigma^2}} \]

- Low-pass filter that reduces high frequency features including noise

- Weighted average better preserves features
Approximate Gaussian Blur

- Same idea as a Gaussian blur but now discretized
- Apply weights to neighbors in kernel based on distance from the center
- Total weight must still equal 1

\[
K = \frac{1}{256} \begin{bmatrix}
1 & 4 & 6 & 4 & 1 \\
4 & 16 & 24 & 16 & 4 \\
6 & 24 & 36 & 24 & 6 \\
4 & 16 & 24 & 16 & 4 \\
1 & 4 & 6 & 4 & 1
\end{bmatrix}
\]
Edge Detection

- Determines sharp discontinuities in value (i.e. edges)
- Provides information about scene:
  - Depth
  - Illumination
  - Material
- Important filter for computer vision/feature extraction
Sobel Operator

- Two 3x3 kernels that approximate horizontal and vertical derivatives (i.e. changes in light intensity)

\[
G_x = \begin{bmatrix}
-1 & 0 & +1 \\
-2 & 0 & +2 \\
-1 & 0 & +1
\end{bmatrix} \ast A \quad \text{and} \quad G_y = \begin{bmatrix}
-1 & -2 & -1 \\
0 & 0 & 0 \\
+1 & +2 & +1
\end{bmatrix} \ast A
\]

- Horizontal and vertical convolutions performed independently

- Gradient magnitude (i.e. rate of change in both directions) calculated from results
Edge Cases

❖ What happens when we try to convolve the edge pixels of our image?
❖ How can we handle this “missing” data?
  ❖ Leave edges untouched (easiest)
  ❖ Fill in missing pixels with 0 or 255
  ❖ Wrap missing pixels (from the other side of the image)
  ❖ Mirror missing pixels (from the other side of the kernel)
❖ How do these choices affect the image appearance?
I call `PImage buffer = originalImage;` to copy my original image to an intermediate buffer. True or false: this allows me to store convolutions safely on the image.
Hands-on: Using Convolutions

Today’s activities:

1. Take your “sharpen” kernel and place it in a 3x3 2D array in Processing
2. Create an image buffer to store the final, convolved image data
3. Apply the sharpen kernel to an image and store the convolved data into your secondary image buffer (this should display to the screen)