CS 10:
Problem solving via Object Oriented Programming

Winter 2017

Tim Pierson
260 (255) Sudikoff

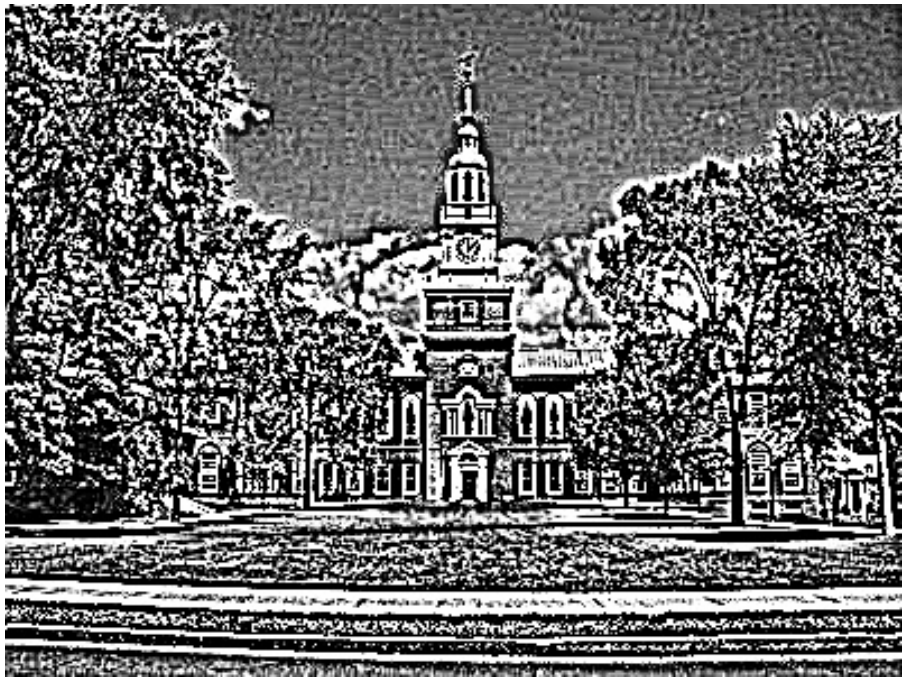
Agenda

1. Manipulating individual pixels
2. Accounting for geometry
3. Interaction
4. Puzzle

Today we will look at processing images as a step toward more sophisticated OOP

Turn this image…

… into this image



Starting with skeleton code

ImageProcessor0.java

- Stores an image with getter/setter methods
- Will enhance to include more sophisticated functionality

ImageProcessingGUI0.java

- Constructor sets up instance variable called "*proc*" to hold *ImageProcessor0* object
- *draw()* calls *proc.getImage()* to display *proc*'s image
- *handleKeyPress()* has option to save image to disk; calls *proc.getImage()* then *repaints*
- The big idea is that *ImageProcessor0* object *proc* will manipulate the image and GUI just uses it

Pixel colors are made up of Red, Green, and Blue components of varying intensity

RGB color values determine color displayed

Red	Green	Blue	Result
255	255	255	**White**
0	0	0	**Black**
255	0	0	**Bright red**
0	255	0	**Bright green**
0	0	255	**Bright blue**
128	0	0	**Not-as-bright-red**
0	128	0	**Not-as-bright green**
0	0	128	**Not-as-bright-blue**

Each pixel color is a 24-bit integer where bits:
16-23 = red component
8-15 = green component
0-7 = blue component

So each R,G, or B components has 8 bits to control color intensity

More colors:
http://www.cs.dartmouth.edu/~tjp/cs10/notes/4/colors.html

We can pick up the color of a pixel, modify it, and write it back to the image

Example: dim a pixel's color

```
//pick up color of pixel at x,y location
Color color = new Color(image.getRGB(x, y));

//extract red, green, blue components and dim them
int red = color.getRed() / 2; //divide by 2 dims intensity
int green = color.getGreen() / 2;
int blue = color.getBlue() / 2;

//write dimmed color back to image
Color newColor = new Color(red, green, blue);
image.setRGB(x, y, newColor.getRGB());
```

With a nested loop we can dim all pixels in an image

Example: dim all pixel colors

```
for (int y = 0; y < image.getHeight(); y++) { //loop over all y
  for (int x = 0; x < image.getWidth(); x++) { //loop over all x
    // Get current color; scale each channel; put new color
    Color color = new Color(image.getRGB(x, y));
    int red = color.getRed() / 2;  //first 8 bits
    int green = color.getGreen() / 2; //second 8 bits
    int blue = color.getBlue() / 2; //third 8 bits
    Color newColor = new Color(red, green, blue);
    image.setRGB(x, y, newColor.getRGB());
  }
}
```

More functional ImageProcessor

ImageProcessor.java

- *dim()* implements code from last slide
- *brighten()* does the opposite of dim, but must check max color value
- *scaleColor()* allows each RGB component to scale individually, must cast doubles to ints with (int)
- *noise()*
  - adds random noise to each color channel
  - *random()* returns number [0,1)
  - multiply random() * 2 then -1 to get range -1..1
  - multiply that -1..1 number by scaling factor to increase range as desired

constrain() method check values to ensure they do not exceed min/max bounds

***constrain()* function**

```
private static double constrain(double val, double min, double max) {
        if (val < min) {
            return min;
        }
        else if (val > max) {
            return max;
        }
        return val;
}
```

Comments

- Will be called often, so to avoid duplicating same bounds checks, create a helper method and call it where needed

constrain() method is of type static

***constrain()* function**

```
private static double constrain(double val, double min, double max) {
        if (val < min) {
            return min;
        }
        else if (val > max) {
            return max;
        }
        return val;
}
```

Comments

- *static* means method is same one for all objects created of this class
- exists outside each specific object
- called "*class variable*", not instance variable
- call with ClassName.method() example Math.random(), also main()

Agenda

1. Manipulating individual pixels
2. Accounting for geometry
3. Interaction
4. Puzzle

Flipping an image requires track where we are and where we want to write

ImageProcessor.flip()

- Create a new blank image "*result*" with *createBlankResult()*
- Nested loop over each row (y) and each column (x)
- Account for geometry where original row written to different row in new image (e.g., when y =0 then original row 0 written to image.getHeight()-0-1)
- Update pixel in "*result*" image
- When loops finish, set object's image variable to new image (original image will be garbage collected)
- What would happen if we did not create a new image?

We can also alter pixels based on neighboring pixels

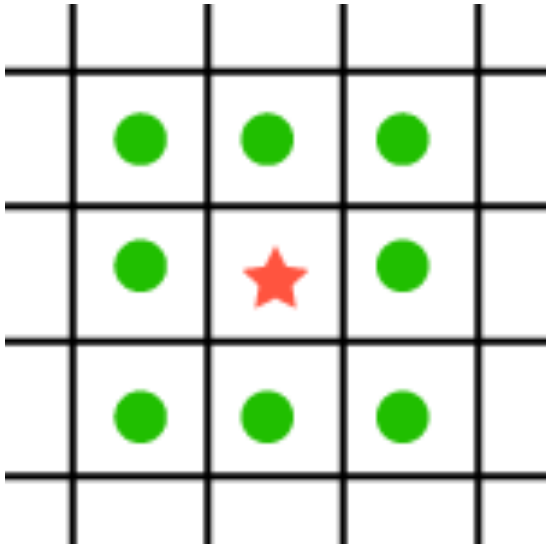
ImageProcessor.scramble()

- Create a new blank image "*result*" with *createBlankResult()*
- Nested loop over each row (y) and each column (x)
- Account for geometry where we pick a random pixel +/- 1 pixel from current location (but not off screen)
- Update pixel in "*result*" image
- When loops finish, set object's image variable to new image (original image will be garbage collected)

Sometimes we want to operate on a pixel's neighbors

Blur image by averaging around each pixel's neighbors

Pixel and neighbors



Averaging can smooth outliers

10	12	13
12	34	11
10	13	11

Average = (10+12+13+12+34+11 +10+13+11)/9 = 14

Replace all values in new image with average of all neighbors

Average() examines neighbors to smooth (blur) and image

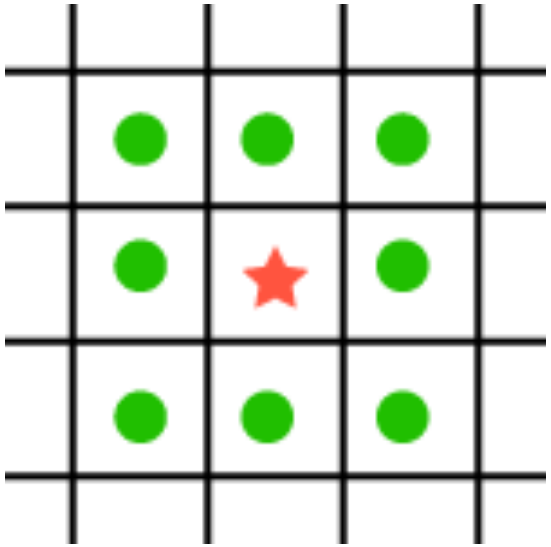
ImageProcessor.java

- *average()* implements code from last slide
- Create a new image to hold result
- Loop over all pixels (nested loop)
- Loop over all neighbors of each pixel ("*radius*" away above, same level, below) NOTE: whitespace
- Make sure not to go off screen, use *constrain()*
- Calculate average for all color components
- Write average to pixel at (x,y) location in new image
- Set image to resulting image
- Do not make radius too big or you'll have a wait!
- What would happen if we did not use a new image to store results, but instead used the original?

sharpen() works similarly to *average()*, but subtracts neighbors weights

Sharpen image by subtracting each pixel's neighbors

Pixel and neighbors



Subtract neighbor weights

-1	-1	-1
-1	9	-1
-1	-1	-1

Result= pixel * 9 – sum(neighbors)

- **Replace all values in new image with computed value**
- **This is called convolution**
- **Used in deep learning and signal processing**

Agenda

1. Manipulating individual pixels
2. Accounting for geometry
3. Interaction
4. Puzzle

Adding some interactivity by handling key and mouse presses

ImageProcessorGUI.handleKeyPress()

- Get key pressed
- Call appropriate function on processor (named *proc*)
- Can control radius for *average()* and *sharpen()*
- *repaint() at end*

ImageProcessorGUI.handleMousePress()

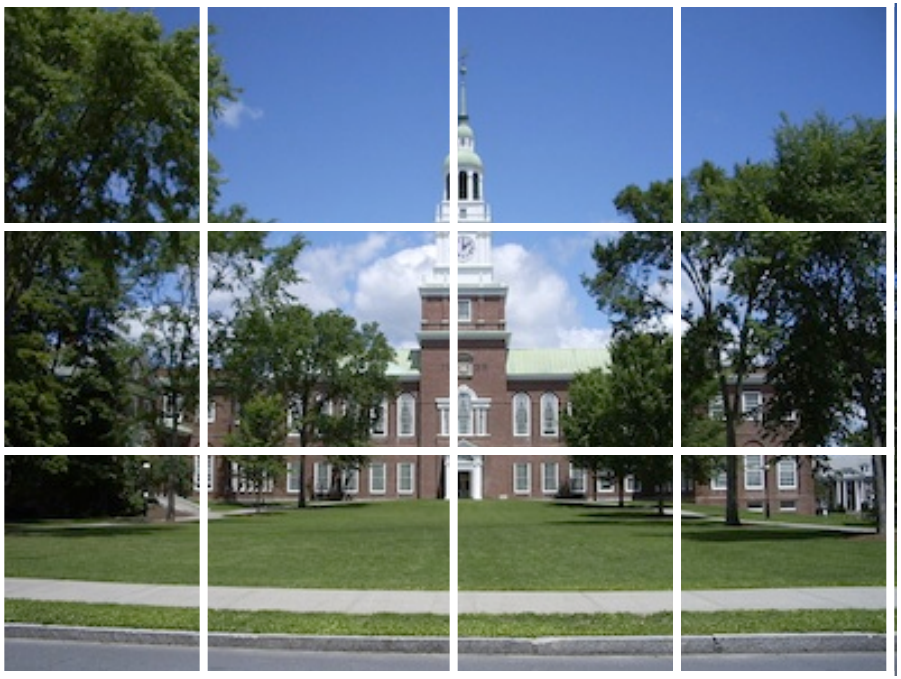
- Add ability to pick up the color at mouse location x,y when press "p" key, and then press mouse store in color in "pickedColor" instance variable
- Add ability to *drawSquare()* of *pickedColor* and *radius* when press "q" then press mouse at location (x,y)
- *repaint() at end*

Agenda

1. Manipulating individual pixels
2. Accounting for geometry
3. Interaction
4. Puzzle

Puzzle breaks an image into multiple pieces and stores pieces in an ArrayList

Original image



4 x 3 puzzle pieces

Piece 0	Piece 1	Piece 2	Piece 3
Piece 4	Piece 5	Piece 6	Piece 7
Piece 8	Piece 9	Piece 10	Piece 11

Puzzle.java

Puzzle.java

- Creates pieces from original image and stores them in an ArrayList
- *getPiece(r,c)* calculates index into ArrayList for given row and column, returns that image piece
- *createPieces()* splits original image into pieces
- *getSubImage()* creates new BufferedImage of pixels from original image
- *swapPieces()* swaps piece in ArrayList at r1,c1 with piece at r2,c2, using temp variable
- *shufflePieces()* loops over each row and column and swaps with a random (possibly same) piece