Picture Encoding and Manipulation

We perceive light different from how it actually is

- Color is continuous
  - Visible light is wavelengths between 370 and 730 nm
    - That’s 0.00000037 and 0.00000073 meters
  - But we perceive light with color sensors that peak around 425 nm (blue), 550 nm (green), and 560 nm (red).
    - Our brain figures out which color is which by figuring out how much of each kind of sensor is responding
Luminance vs. Color

• We perceive borders of things, motion, depth via luminance
  – Luminance is not the amount of light, but our perception of the amount of light.
  – We see blue as “darker” than red, even if same amount of light.
• Much of our luminance perception is based on comparison to backgrounds, not raw values.

Luminance perception is color blind.
Different parts of the brain perceive color and luminance.

Digitizing pictures into dots

• We digitize pictures into many tiny dots
• Enough dots and it looks continuous to the eye
  – Our eye has limited resolution
  – Our background/depth acuity is particularly low
• Each picture element is a pixel
  i.e. “picture element”
Pixels in Python

• Pixels are *picture elements*
  – Each pixel in python is an object that “knows” its *color*  
    • E.g. given a pixel, a Python function can get the color of it.
  – It also “knows” where it is in the *picture*  
    • E.g. given a pixel and a picture, a Python function can find out where the pixel is located in the picture

A Picture is a *matrix* of pixels

• It’s not a continuous line of elements, that is, a 1-D *array*
• A picture has two dimensions: Width and Height
• We need a 2-dimensional array: a *matrix*

Just the upper left hand corner of a matrix.
Referencing a matrix

- We talk about positions in a matrix as \((x, y)\), or (horizontal, vertical)
- The origin \((1, 1)\) is in the upper left corner of the picture
- Element \((2, 1)\) in the matrix at left is the value 12
- Element \((1, 3)\) is 6

RGB

- In RGB, each color has three component colors:
  - Amount of red
  - Amount of green
  - Amount of blue
- In a CRT each appears as a dot and is blended by our eye.
- In most computer-based models of RGB, a single byte (8 bits) is used for each
  - So a complete RGB color is 24 bits, 8 bits of each
How much can we encode in 8 bits?

• Let’s walk through it.
  – If we have one bit, we can represent two patterns: 0 and 1.
  – If we have two bits, we can represent four patterns: 00, 01, 10, and 11.
  – If we have three bits, we can represent eight patterns: 000, 001, 010, 011, 100, 101, 110, 111
• The rule: In \( n \) bits, we can have \( 2^n \) patterns
  – In 8 bits, we can have \( 2^8 \) patterns, or 256
  – If we make one pattern 0, then the highest value we can represent is \( 2^8 - 1 \), or 255

Encoding RGB

• Each component color (red, green, and blue) is encoded as a single byte
• Colors go from (0, 0, 0) to (255, 255, 255)
  – If all three components are the same, the color is in grayscale
    • (50, 50, 50) at (2, 2)
  – (0, 0, 0) (at position (1, 2) in example) is black
  – (255, 255, 255) is white
Is that enough?

- We’re representing color in 24 (3 * 8) bits.
  - That’s 16,777,216 \( (2^{24}) \) possible colors
  - Our eye can discern millions of colors, so it is pretty close
  - But the real limitation is the physical devices: We don’t get 16 million colors out of a monitor

- Some graphics systems support 32 bits per pixel
  - May be more pixels for color
  - More useful is to use the additional 8 bits to represent not color but 256 levels of translucence

Media jargon: 4th byte per pixel is the “Alpha channel”

<table>
<thead>
<tr>
<th></th>
<th>320 x 240 image</th>
<th>640 x 480 image</th>
<th>1024 x 768 monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>24 bit color</strong></td>
<td>1,843,200 bits</td>
<td>7,372,800 bits</td>
<td>18,874,368 bits</td>
</tr>
<tr>
<td></td>
<td>230,400 bytes</td>
<td>921,600 bytes</td>
<td>2,359,296 bytes</td>
</tr>
<tr>
<td><strong>32 bit color</strong></td>
<td>2,457,600 bits</td>
<td>9,830,400 bits</td>
<td>25,165,824 bits</td>
</tr>
<tr>
<td></td>
<td>307,200 bytes</td>
<td>1,228,800 bytes</td>
<td>3,145,728 bytes</td>
</tr>
</tbody>
</table>
Reminder: Manipulating Pictures

```python
>>> file = pickAFile()
>>> print file
C:\Documents and Settings\Kenrick\My Documents\Class\CSA109\JES\content\MediaSources\ducks\ducks 010.jpg
>>> picture = makePicture(file)
>>> show(picture)
>>> print picture
Picture, filename C:\Documents and Settings\Kenrick\My Documents\Class\CSA109\JES\content\MediaSources\ducks\ducks 010.jpg
height 240 width 320
```

What is a “picture”?  

- A picture object in JES is an encoding that represents an image
  - Knows its height and width
    - i.e. it knows how many pixels it contains in both directions
  - Knows its filename
    - A picture isn’t a file, it’s what you get when you `makePicture()` a file...but it does “remember” the file it came from.
  - Knows its `window` if it’s opened (via `show` and repainted with `repaint`)
    - which we will need to do later....
Manipulating pixels

\texttt{getPixel(picture, x, y)} gets a single pixel.
\texttt{getPixels(picture)} gets all of them into an array.

```python
>>> pixel = getPixel(picture, 1, 1)
>>> print pixel
Pixel, color=color r=168 g=131 b=105
```  
```python
>>> pixels = getPixels(picture)
```  
```python
>>> print pixels[0]
Pixel, color=color r=168 g=131 b=105
```

\textbf{Square brackets: standard way to refer to an element in an array}—which we'll generally not use.

Close, but not quite

- The preceding slide is not quite true—there is a small bug in the way \texttt{getPixels} works.

get\texttt{Pixels}\texttt{[1]} returns back only the pixels in this area, skipping the top row and left column.
What can we do with a pixel?

- `getRed`, `getGreen`, and `getBlue` are functions that take a pixel as input and return a value between 0 and 255.
- `setRed`, `setGreen`, and `setBlue` are functions that take a pixel as input and a value between 0 and 255.

We can also get, set, and make Colors

- `getColor` takes a pixel as a parameter and returns a Color object from the pixel.
- `setColor` takes a pixel as a parameter and a Color, then sets the pixel to that color.
- `makeColor` takes red, green, and blue values (in that order) each between 0 and 255, and returns a Color object.
- `pickAColor` lets you use a color chooser and returns the chosen color.
- We also have functions that can `makeLighter` and `makeDarker` an input color.
How “close” are two colors?

• Sometimes you need to find the *distance* between two colors, e.g., when deciding if something is a “close enough” match

• How do we measure distance?
  – Pretend it’s Cartesian coordinate system
  – Distance between two points:

\[ \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \]

  – Distance between two colors:

\[ \sqrt{(\text{red}_1 - \text{red}_2)^2 + (\text{green}_1 - \text{green}_2)^2 + (\text{blue}_1 - \text{blue}_2)^2} \]

Demonstrating: Manipulating Colors

```python
>>> print getRed(pixel)
168
>>> setRed(pixel, 255)
>>> print getRed(pixel)
255
>>> color = getColor(pixel)
>>> print color
color r=255 g=131 b=105
>>> setColor(pixel, color)
>>> newColor = makeColor(0, 100, 0)
>>> print newColor
color r=0 g=100 b=0
>>> print distance(color, newcolor)
174.41330224498358
>>> print color
color r=168 g=131 b=105
>>> print makeDarker(color)
>>> print newColor
>>> newcolor = pickAColor()
>>> print newcolor
color r=255 g=51 b=51
```
We can change pixels directly…

```python
>>> pict = makePicture(file)
>>> show(pict)
>>> red = makeColor(255,0,0)
>>> setColor(getPixel(pict, 10, 100), red)
>>> setColor(getPixel(pict, 11, 100), red)
>>> setColor(getPixel(pict, 12, 100), red)
>>> setColor(getPixel(pict, 13, 100), red)
>>> repaint(pict)
```

But that’s really tedious… Manipulating pictures more cleverly is coming up next

How do you find out what RGB values you have? And where?

- Use a paint program – or use the MediaTools!
- Drag mediatoools.image onto squeakVM to run

(especially useful when testing and debugging…)
Better Pixel Manipulation - Use a loop!

```python
def decreaseRed(picture):
    for p in getPixels(picture):
        value = getRed(p)
        setRed(p, value * 0.5)
```

Used like this:
```python
>>> file = r"c:\mediasources\katie.jpg"
>>> picture = makePicture(file)
>>> show(picture)
>>> decreaseRed(picture)
>>> repaint(picture)
```

How loops are written

- **for** is the name of the command
- An *index variable* is used to hold each of the different values of a sequence
- The word **in**
- A function that generates a *sequence*
  - The index variable will be the name for one value in the sequence, each time through the loop
- A colon ("::")
- And a *block*
What happens when a loop is executed

• The *index variable* is set to an item in the *sequence*
• The block is executed
  – The variable is often used inside the block
• Then execution *loops* to the *for* statement, where the index variable gets set to the next item in the sequence
• Repeat until every value in the sequence was used.

**getPixels** returns a sequence of pixels

• Each pixel knows its color and place in the original picture
• Change the pixel, you change the picture
• So the loop below assigns the index variable *p* to each pixel in the picture *picture*, one at a time.

```python
def decreaseRed(picture):
    for p in getPixels(picture):
        originalRed = getRed(p)
        setRed(p, originalRed * 0.5)
```
Do we need the variable *originalRed*?

- Not really: Remember that we can swap names for data and function calls that are equivalent.

```python
def decreaseRed(picture):
    for p in getPixels(picture):
        originalRed = getRed(p)
        setRed(p, originalRed * 0.5)
```

Let's walk that through slowly…

```python
def decreaseRed(picture):
    for p in getPixels(picture):
        originalRed = getRed(p)
        setRed(p, originalRed * 0.5)
```

Here we take a picture object in as a parameter to the function and call it *picture*.
Now, get the pixels

```python
def decreaseRed(picture):
    for p in getPixels(picture):
        originalRed = getRed(p)
        setRed(p, originalRed * 0.5)
```

We get all the pixels from the `picture`, then make `p` be the name of each one one at a time

<table>
<thead>
<tr>
<th>Pixel, color</th>
<th>Pixel, color</th>
<th>Pixel, color</th>
</tr>
</thead>
<tbody>
<tr>
<td>r=135</td>
<td>r=133</td>
<td>r=134</td>
</tr>
<tr>
<td>g=116</td>
<td>g=114</td>
<td>g=114</td>
</tr>
<tr>
<td>b=48</td>
<td>b=46</td>
<td>b=45</td>
</tr>
</tbody>
</table>

Get the red value from pixel

```python
def decreaseRed(picture):
    for p in getPixels(picture):
        originalRed = getRed(p)
        setRed(p, originalRed * 0.5)
```

We get the red value of pixel `p` and name it `originalRed`

<table>
<thead>
<tr>
<th>Pixel, color</th>
<th>Pixel, color</th>
<th>Pixel, color</th>
</tr>
</thead>
<tbody>
<tr>
<td>r=135</td>
<td>r=133</td>
<td>r=134</td>
</tr>
<tr>
<td>g=116</td>
<td>g=114</td>
<td>g=114</td>
</tr>
<tr>
<td>b=48</td>
<td>b=46</td>
<td>b=45</td>
</tr>
</tbody>
</table>

originalRed = 135
Now change the pixel

```python
def decreaseRed(picture):
    for p in getPixels(picture):
        originalRed = getRed(p)
        setRed(p, originalRed * 0.5)
```

Set the red value of pixel `p` to 0.5 (50%) of `originalRed`

Then move on to the next pixel

```python
def decreaseRed(picture):
    for p in getPixels(picture):
        originalRed = getRed(p)
        setRed(p, originalRed * 0.5)
```

Move on to the next pixel and name it `p`
Get its red value

```python
def decreaseRed(picture):
    for p in getPixels(picture):
        originalRed = getRed(p)
        setRed(p, originalRed * 0.5)
```

Set `originalRed` to the red value at the new `p`, then change the red at that new pixel.

And change *this* red value

```python
def decreaseRed(picture):
    for p in getPixels(picture):
        originalRed = getRed(p)
        setRed(p, originalRed * 0.5)
```

Change the red value at pixel `p` to 50% of value.
And eventually, we do all pixels

- We go from this… to this!

“Tracing/Stepping/Walking through” the program

- What we just did is called “stepping” or “walking through” the program
  - You consider each step of the program, in the order that the computer would execute it
  - You consider what exactly would happen
  - You write down what values each variable (name) has at each point.
- It’s one of the most important debugging skills you can have.
  - And everyone has to do a lot of debugging, especially at first.
Did that really work?  
How can we be sure?

• Sure, the picture *looks* different, but did we actually decrease the amount of red? By as much as we thought?
• Let’s check it!

```python
>>> file = pickAFile()
>>> print file
C:\Documents and Settings\Kenrick Mock\My Documents\mediasources\barbara.jpg
>>> pict = makePicture(file)
>>> pixel = getPixel(pict, 2, 2)
>>> print pixel
Pixel, color=color r=168 g=131 b=105
>>> decreaseRed(pict)
>>> newPixel = getPixel(pict, 2, 2)
>>> print newPixel
Pixel, color=color r=84 g=131 b=105
>>> print 168 * 0.5
84.0
```
Want to save the new picture?

- **writePictureTo**(picture, "filename.jpg")
- Writes the picture out as a JPEG
- **Be sure to end your filename as “.jpg”!**
- If you don’t specify a full path, will be saved in the same directory as JES.

Checking it in the MediaTools