



COMP2720: Automating Tools for New Media

Mirroring and drawing directly
into pictures

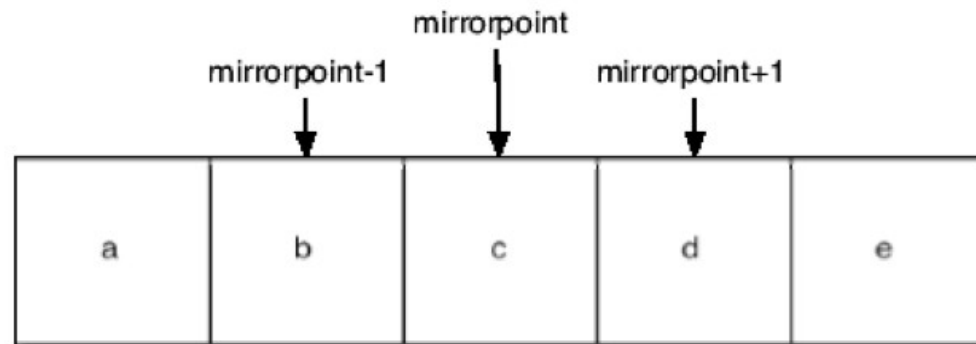


Learning objectives of this lecture

- How to mirror a picture
- Horizontal and vertical mirroring
- Drawing lines
- Learning more JES drawing functions
- Drawing a picture
- Vector- versus bitmap pictures
- Programmed graphics
- (extra-curricular) Image compression, JPEG

If you know where the pixels are: Mirroring

- Imagine a mirror horizontally across the picture, or vertically
- What would we see?
- How do generate that digitally?
 - We simply copy the colors of pixels from one place to another
- Cut a picture down the middle, stick a mirror on the slice
- Do it by using a loop to measure a *difference*
 - *The index variable is actually measuring distance from the mirrorpoint*
- *Then reference to either side of the mirrorpoint using the difference*



Program (recipe) for mirroring

```
def mirrorVertical(source):
    mirrorpoint = int(getWidth(source) / 2)
    for y in range(1, getHeight(source)+1):
        for xOffset in range(1, mirrorpoint):
            pright = getPixel(source, xOffset + mirrorpoint, y)
            pleft = getPixel(source, mirrorpoint - xOffset, y)
            c = getColor(pleft)
            setColor(pright, c)
```

How does it work?

- Compute the half-way horizontal index.
- The y value travels the height of the picture.
- The xOffset value is an *offset*.
 - It's not actually an index.
 - It's the amount to add or subtract.
- We copy the color at *mirrorpoint-offset* to *mirrorpoint+offset*.



```
def mirrorVertical(source):  
    mirrorpoint = int(getWidth(source) / 2)  
    for y in range(1, getHeight(source)+1):  
        for xOffset in range(1, mirrorpoint):  
            pright = getPixel(source, xOffset + mirrorpoint, y)  
            pleft = getPixel(source, mirrorpoint - xOffset, y)  
            c = getColor(pleft)  
            setColor(pright, c)
```

Can we do this with a horizontal mirror?

```
def mirrorHorizontal(source):  
    mirrorpoint = int(getHeight(source) / 2)  
    for yOffset in range(1, mirrorpoint):  
        for x in range(1, getWidth(source)+1):  
            pbottom = getPixel(source, x, yOffset + mirrorpoint)  
            ptop = getPixel(source, x, mirrorpoint - yOffset)  
            setColor(pbottom, getColor(ptop))
```

Of course!




What if we wanted to copy bottom to top?

- Very simple: Swap the *order of pixels* in the bottom line

```
def mirrorHorizontal(source):  
    mirrorpoint = int(getHeight(source) / 2)  
    for yOffset in range(1, mirrorpoint):  
        for x in range(1, getWidth(source)+1):  
            pbottom = getPixel(source, x, yOffset + mirrorpoint)  
            ptop = getPixel(source, x, mirrorpoint - yOffset)  
            setColor(ptop, getColor(pbottom))
```

Set color this way, instead of this


`setColor(pbottom, getColor(ptop))`

Messing with Santa some more



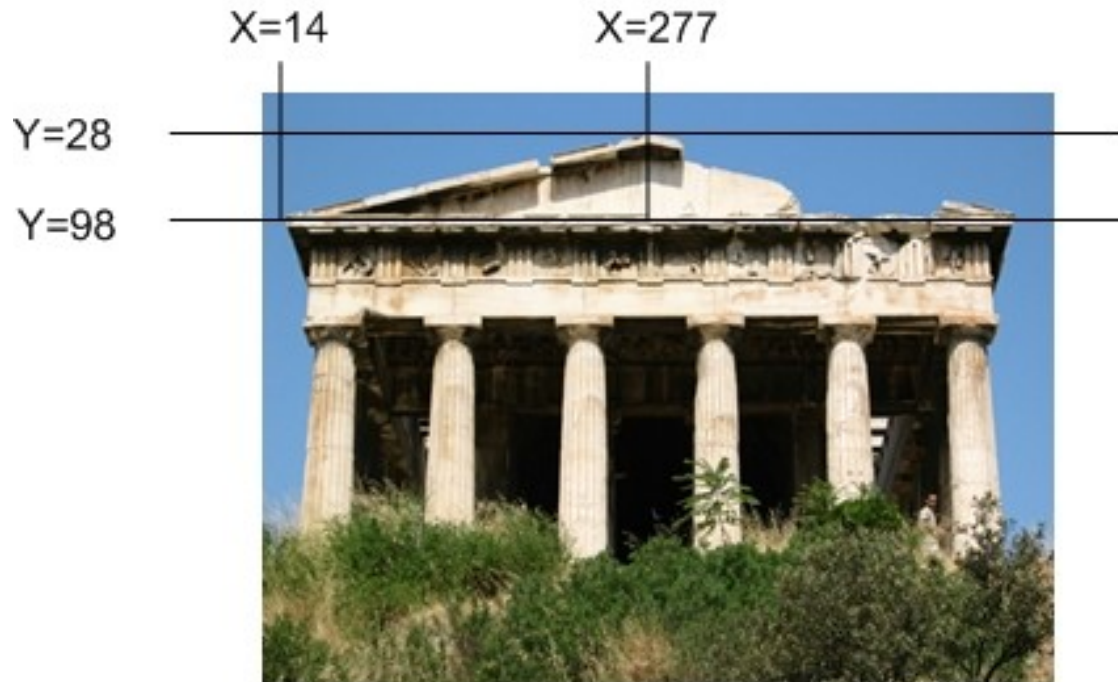
Doing something useful with mirroring

- Mirroring can be used to create interesting effects, but it can also be used to create realistic effects.
- Consider this image that M.G. took on a trip to Athens, Greece.
 - Can we “repair” the temple by mirroring the complete part onto the broken part?



Figuring out where to mirror

- Use **MediaTools** to find the mirror point and the range that we want to copy.



Program to mirror the temple

```
def mirrorTemple():
    source = makePicture(getMediaPath("temple.jpg"))
    mirrorpoint = 277
    lengthToCopy = mirrorpoint - 14
    for x in range(1, lengthToCopy):
        for y in range(28, 98):
            p1 = getPixel(source, mirrorpoint - x, y)
            p2 = getPixel(source, mirrorpoint + x, y)
            setColor(p2, getColor(p1))
    show(source)
    return source
```

Did it really work?

- It clearly did the mirroring, but that doesn't create a 100% realistic image.
- Check out the shadows: Which direction is the sun coming from?





We can make whatever we want on pictures already

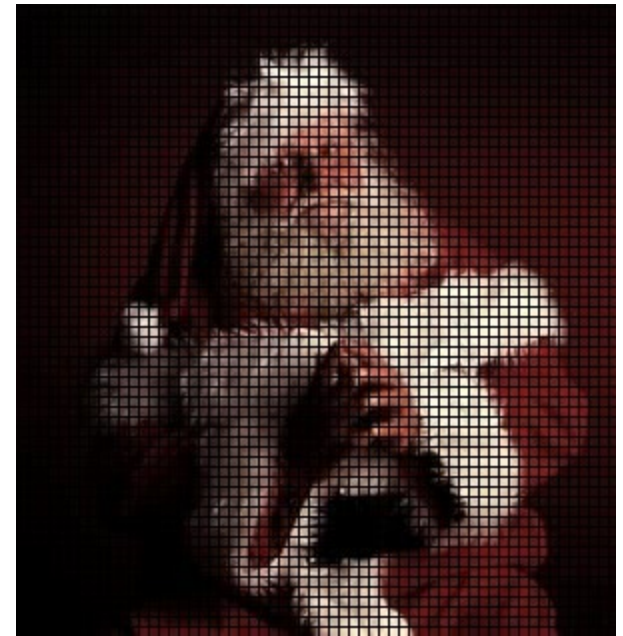
- *All* drawing on pictures comes down to changing pixel color values.
- By directly changing values to black (or whatever else we want), we can draw whatever we want.

Drawing lines on Santa

```
def lineExample():  
    pic = makePicture(pickAFile())  
    new = verticalLines(pic)  
    new2 = horizontalLines(new)  
    show(new2)  
    return new2
```

```
def horizontalLines(src):  
    for y in range(1,getHeight(src),5):  
        for x in range(1,getWidth(src)+1):  
            setColor(getPixel(src,x,y),black)  
    return src
```

```
def verticalLines(src):  
    for x in range(1,getWidth(src),5):  
        for y in range(1,getHeight(src)+1):  
            setColor(getPixel(src,x,y),black)  
    return src
```



Yes, some colors are already defined

Yes, some colors are already defined in JES

- Colors defined for you already: *black, white, blue, red, green, gray, lightGray, darkGray, yellow, orange, pink, magenta, and cyan.*
 - It's a good habit in large programs to use named constants, instead of "magic numbers" (see later).



But that's tedious

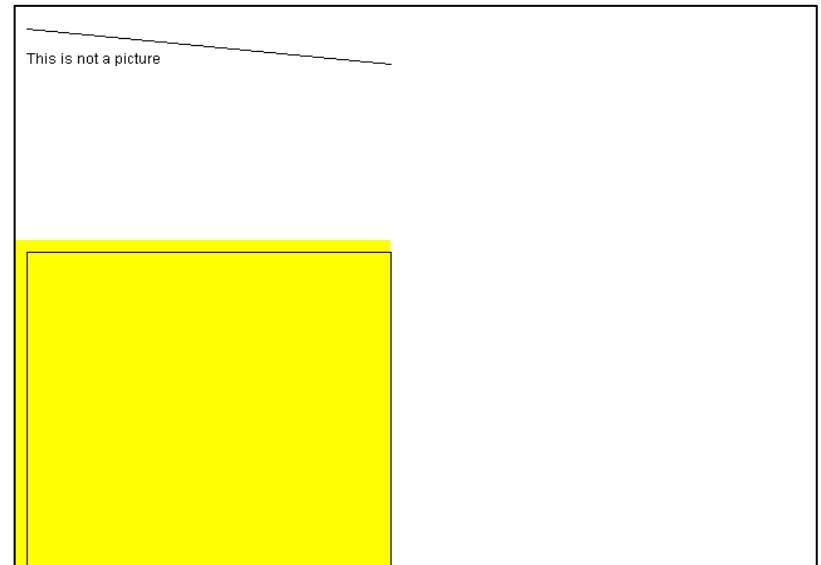
- It's slow and tedious to set every pixel you want.
 - Also, what if you wanted to redraw the line later? JES doesn't know it's a "line".
- What you really want to do is to think in terms of your desired effect (think about "requirements" and "design").

New functions

- `addText(picture,x,y,string)` puts the string starting at position (x,y) in the picture.
- `addLine(picture,x1,y1,x2,y2)` draws a line from position $(x1,y1)$ to $(x2,y2)$.
- `addRect(pict,x1,y1,w,h)` draws a black rectangle (unfilled) with the upper left hand corner of $(x1,y1)$ and a width of w and height of h .
- `addRectFilled(pict,x1,y1,w,h,color)` draws a rectangle filled with the color of your choice with the upper left hand corner of $(x1,y1)$ and a width of w and height of h .

Example picture

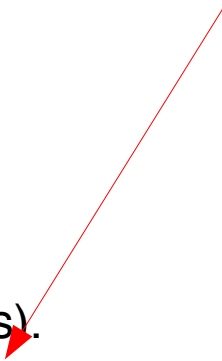
```
def littlePicture():  
    canvas=makeEmptyPicture(640, 480)  
    addText(canvas,10,50,"This is not a picture")  
    addLine(canvas,10,20,300,50)  
    addRectFilled(canvas,0,200,300,500,yellow)  
    addRect(canvas,10,210,290,490)  
    return canvas
```



A *thought* experiment

Isn't this just
“programming”?

Yes, but jargon is:
“vector graphics”

- Look at that previous slide: Which is smaller (i.e. fewer bytes)?
 - The program that drew the picture.
 - The pixels in the picture itself.
 - It's a no-brainer
 - The program is less than 100 characters (100 bytes).
 - The picture is stored on disk at about 15,000 bytes.
 - So, if we could tell JES to draw a line, we'd need less memory than if we drew it ourselves by telling JES to paint pixels. (And JES could be cleverer than we are.)
- 

Vector-based vs. bitmap graphical representations

- Vector-based graphical representations are basically executable programs that generate the picture on demand.
 - *Postscript*, *Flash*, and *AutoCAD* use vector-based representations.
- Bitmap graphical representations (like *JPEG*, *BMP*, *GIF*) store individual pixels or representations of those pixels.
 - *JPEG* and *GIF* are actually compressed representations.



Vector-based representations can be smaller

- Vector-based representations can be much smaller than bit-mapped representations.
 - Smaller means faster transmission (*Flash* and *Postscript*).
 - If you want all the detail of a complex picture, no, it's not.



But vector-based has more value than that

- Imagine that you're editing a picture with lines on it.
 - If you edit a bitmap image and extend a line, it's just more bits.
 - There is no way to really realise that you have extended or shrunk the line.
 - If you edit a vector-based image, it's possible to just change the specification.
 - Change the numbers saying where the line is.
 - Then it really is the same line.
- That's important when the picture drives the creation of the product, like in automatic cutting machines.

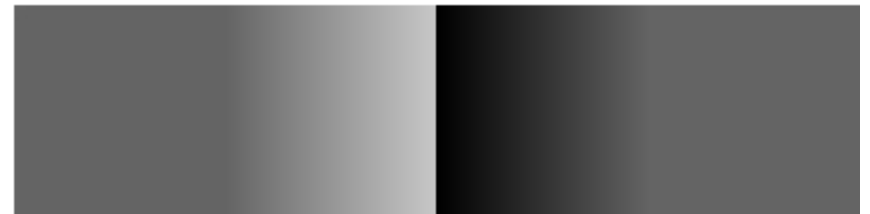


But how could we change the specification (the program)?

- How could we reach in and change the actual program?
- That's called *string manipulation*.
 - The program is just a string of characters (text).
 - We want to manipulate those characters, in order to manipulate the program.
 - The vector graphics program *interprets* the text representation of a graphic.
- We do that in a couple of weeks...

Example programmed graphic

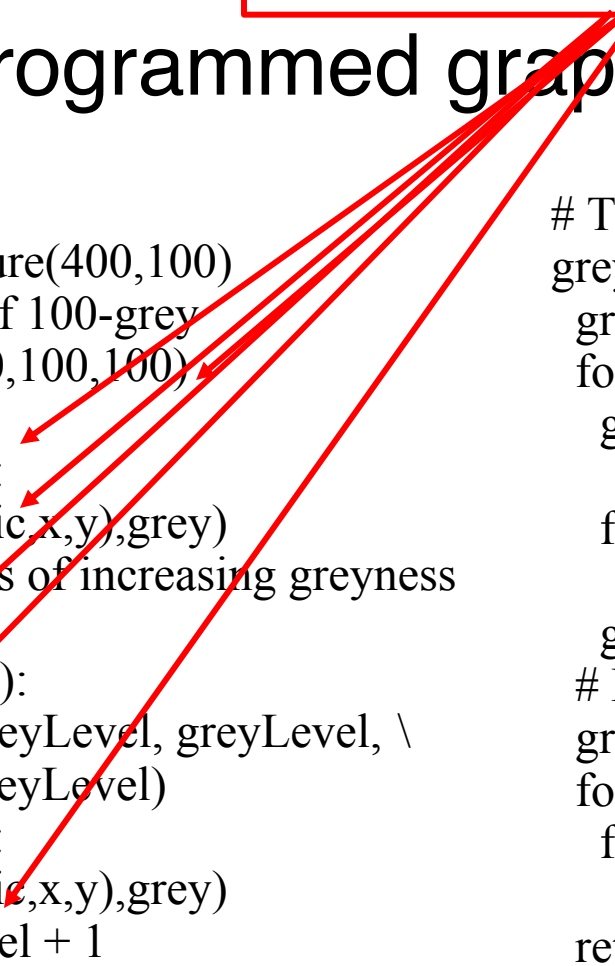
- If we did this right, we perceive the left half as lighter than the right half.
- In reality, the end quarters are actually the same colors.



Danger! “Magic numbers” Why “100”?

Building a programmed graphic

```
def greyEffect():
    pic = makeEmptyPicture(400,100)
    # First, 100 columns of 100-grey
    grey = makeColor(100,100,100)
    for x in range(1,100):
        for y in range(1,100):
            setColor(getPixel(pic,x,y),grey)
    # Second, 100 columns of increasing greyness
    greyLevel = 100
    for x in range(100,200):
        grey = makeColor(greyLevel, greyLevel, \
                        greyLevel)
        for y in range(1,100):
            setColor(getPixel(pic,x,y),grey)
        greyLevel = greyLevel + 1
    # Third, 100 columns of increasing
    greyness, from 0
    greyLevel = 0
    for x in range(200,300):
        grey = makeColor(greyLevel, \
                        greyLevel, greyLevel)
        for y in range(1,100):
            setColor(getPixel(pic,x,y),grey)
        greyLevel = greyLevel + 1
    # Finally, 100 columns of 100-grey
    grey = makeColor(100,100,100)
    for x in range(300,400):
        for y in range(1,100):
            setColor(getPixel(pic,x,y),grey)
    return pic
```



Removing the magic numbers

```
def greyEffect():
    ...

    # First, 'width' columns of grey
    greyLevel = 100
    grey = makeColor(greyLevel, greyLevel, greyLevel)
    width = 100
    height = 100
    for x in range(1, width):
        for y in range(1, height):
            setColor(getPixel(pic,x,y), grey)
    # Second, "width" columns of increasing greyness
    for x in range(width, width+width):
        for y in range(1,height):
            setColor(getPixel(pic,x,y),grey)
            greyLevel = greyLevel + 1
```

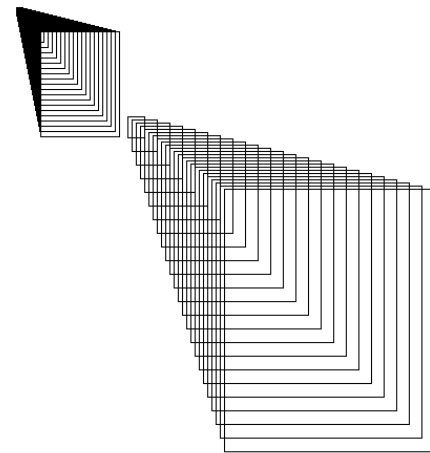
Another Programmed Graphic

```
def coolpic():  
    canvas=makeEmptyPicture(250, 250)  
    for index in range(25,1,-1):  
        color = makeColor(index*10,index*5,index)  
        addRectFilled(canvas,1,1,index*10,index*10,color)  
    show(canvas)  
    return canvas
```



And another

```
def coolpic2():  
    canvas=makeEmptyPicture(450, 450)  
    for index in range(25,1,-1):  
        addRect(canvas,index,index,index*3,index*4)  
        addRect(canvas,100+index*4,100+index*3,index*8,index*10)  
    show(canvas)  
    return canvas
```



Why do we write programs?

- Could we do this in *PhotoShop*? Maybe
 - I'm sure that you can, but you need to know how...
 - Could I teach you to do this in *PhotoShop*? Maybe, but...
 - Might take a lot of demonstration.
- But this program is an *exact definition* of the process of generating this picture.
 - It works for anyone who can run the program, without knowing *PhotoShop*.
 - The lines can be exact, without “jaggies”.
 - The drawing process can be optimized under the hood, etc.



We write programs to *encapsulate* and *communicate* processes

def howtodoit:

if you can do something by hand:

do it by hand

if you need to teach someone else to do it:

consider a program

if you need to explain to lots of people how to do it:

definitely use a program.

if you want lots of people to do it without having to teach them something first:

definitely use a program

else:

ask an expert...

Image compression: JPEG format

(the following discussion is based on D. Austin feature from “What is...?” column in *Notices of the AMS*, Feb. 2008)

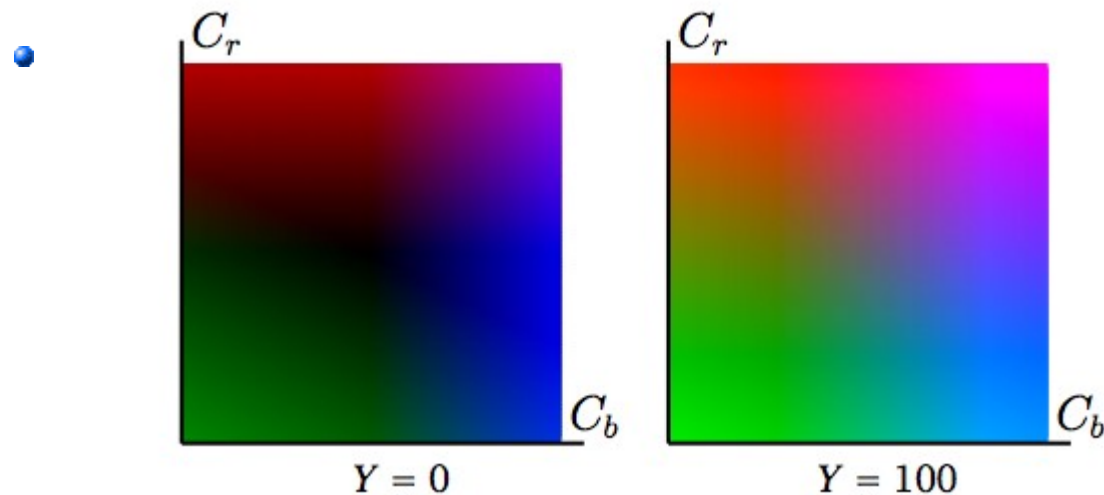
Question: suppose an image composed of 3,871,488 RGB colour pixels. Its size, if calculated bluntly, should be 11,614,464 bytes (confirm this!) Yet, the actual size of the file is only 734,268 bytes, almost 16 times smaller. How?



One of the most commonly used algorithm is due to *Joint Photograph Expert Group (JPEG)*

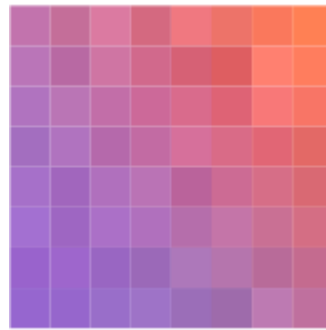
JPEG Algorithm, 1

- Instead of colour primaries (R, G, B) use luminance, red chrominance and blue chrominance (Y, C_b, C_r)
- Knowing (Y, C_b, C_r) one can recover (R, G, B) and v-v

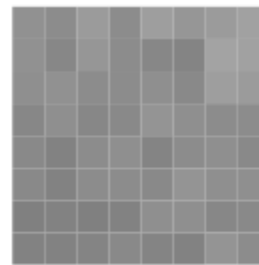


JPEG Algorithm, 2

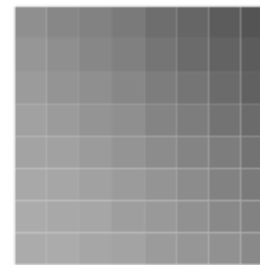
- Next, the pixel matrix is broken into 8x8 blocks, each processed independently
- Let's imagine one chosen 8x8 block looks like this



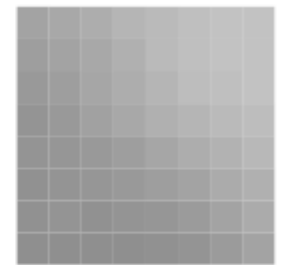
- and (Y, C_r, C_b) look like this



Y



C_b



C_r

JPEG Algorithm, 3

- As we know, the human eye is the most sensitive to the changes in Y , therefore the image transformation can be most efficiently encoded if the luminance Y is given the most bits at the expense of the chrominancies $C_{r,b}$
- Inside the chosen 8×8 block the luminance values $Y_{x,y}$ are replaced on the so called 2 dimensional *Discrete Cosine Transform (DCT)* coefficients (never mind mathematical details)
$$Y_{x,y} \rightarrow F_{u,v}$$
- Essentially, one set of 64 values is replaced on another set of 64 values.

JPEG Algorithm, 4

- These $C_{x,y}$ values do not change rapidly within the block, and anyway the human eye cannot perceive these changes. For the coefficients $F_{u,v}$ it means that their values which correspond to large values of (u,v) are very small. Therefore, these large (u,v) -value components can be neglected.
- The possibility to drop the large (u,v) -value components allows to *quantize* the F values, store them as integers, thus reducing the amount of bits and the file size.
- The quantization involves two ingredients: the α parameter which controls the degree of compression and hence the image quality.

JPEG Algorithm, 5

- The second ingredient is 8x8 matrix $\hat{Q} = [Q_{ab}]$
- In the matrix F when we move down or to the right, we encounter coefficients corresponding to sharp changes, and if the latter is absent (true almost always, and for every image with moderate resolution), these coefficients are very small.
- Storing F coefficients as integers amounts to rounding $F_{u,v}$, **but** the actual compression is achieved by rounding

$$\left\| \frac{F_{u,v}}{(\alpha Q_{u,v})} \right\|$$

The greater the compression (α) the more F coefficients are turned out 0 after the rounding, the more space is saved

JPEG Algorithm, 6

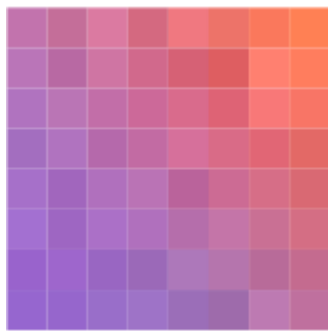
- The matrix Q (actually, three matrices, one for luminance and two for chrominances) is chosen *empirically*

$$Q = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

The elements are chosen to emphasize slow variation in pixels colour

JPEG Algorithm, 7

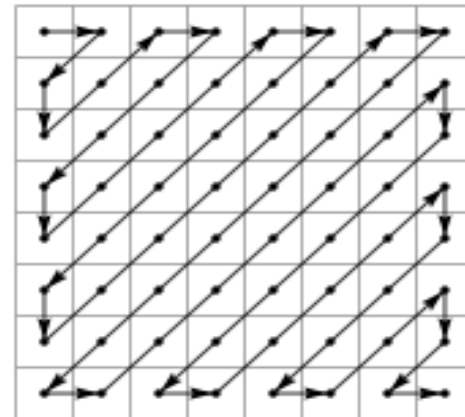
- The rounding turns the original 8x8 sub-image into \widehat{F}_α




	u							
	7	-2	1	0	0	0	0	0
v	4	0	1	0	0	0	0	0
	1	-1	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0

- The final step is to record the elements of \widehat{F}_α

7, -2, 4, 1, 0, 1, 0, 1, -1, 0, 0, 0, ... (35 zeros)
Instead of storing all 0s, we only record their #





What to do now

- Drop in labs this and next week (check the forum announcements for times): can discuss lab1/lab2 exercises and homework1 exercises
- Work on assignment 1
- Start working on portfolio (put lab 1 and lab 2 workings into portfolio)
- Next topic (on Friday): Sounds!