6.837 Lecture 3

Topics in Imaging

A wee bit of recursion

Filling Algorithms
Dithering
DCTs

Area Fill Algorithms?

- Used to fill regions *after* their boundary or contour has been determined.
- Handles Irregular boundaries
- Supports freehand drawings
- Deferred fills for speed (only fill visible parts)
- Allows us to recolor primitives
- Can be adapted for other uses (selecting regions)
- Unaware of any other primitives previously drawn.
- Later we'll discuss filling areas during rasterization

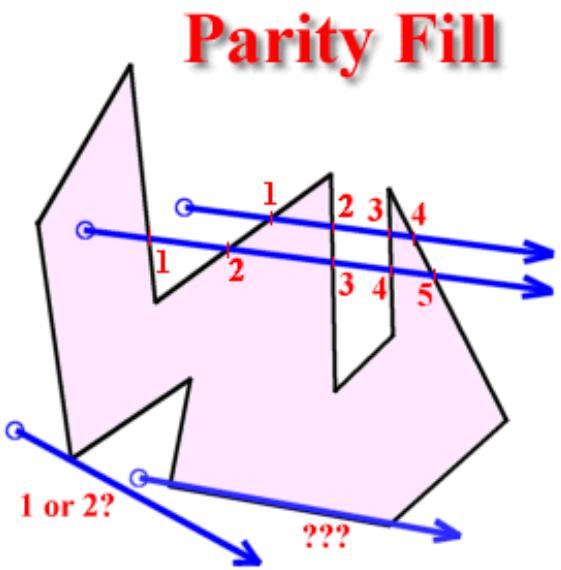
Parity Fill

Problem:

For each pixel determine if it is inside or outside of a given polygon.

Approach:

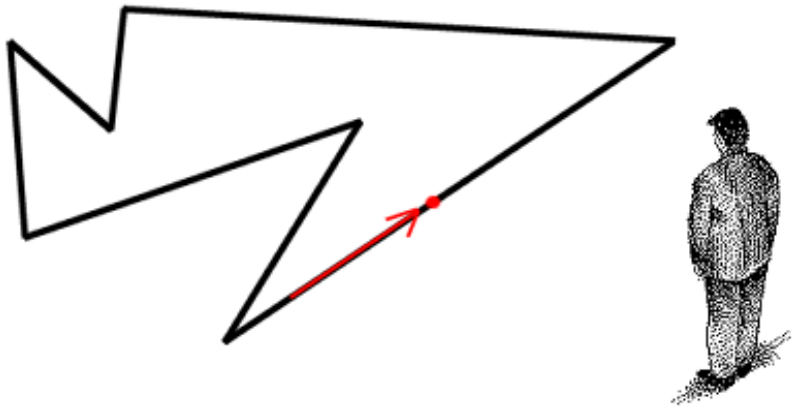
- from the point being tested cast a ray in an arbitary direction
- if the number of crossings is odd then the point is inside
- if the number of crossings is even then the point is outside



- Very fragile algorithm
- What if ray crosses a vertex?
- What if ray falls on an edge?
- Commonly used in ECAD
- Suitable for H/W

Winding Number

Imagine yourself watching a point traverse the boundary of the polygon in a counter-clockwise direction and pivoting so that you are always facing at it.



Your *winding number* is the number of full revolutions that you complete.

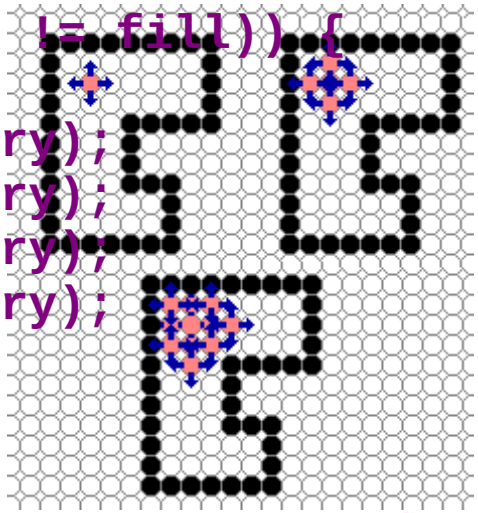
If you winding number is 0 then you are outside of the polygon, otherwise you are inside.

Boundary Fills

Boundary fills start from a point known to be inside of a region and fill the region until a boundry is found.

A simple recursive algorithm can be used:

```
    public void boundaryFill(int x, int y, int fill, int
boundary) {
        if ((x < 0) || (x >= width)) return;
        if ((y < 0) || (y >= height)) return;
        int current = raster.getPixel(x, y);
        if ((current != boundary) && (current != fill)) {
            raster.setPixel(fill, x, y);
            boundaryFill(x+1, y, fill, boundary);
            boundaryFill(x, y+1, fill, boundary);
            boundaryFill(x-1, y, fill, boundary);
            boundaryFill(x, y-1, fill, boundary);
        }
    }
```



Let's Watch it in Action

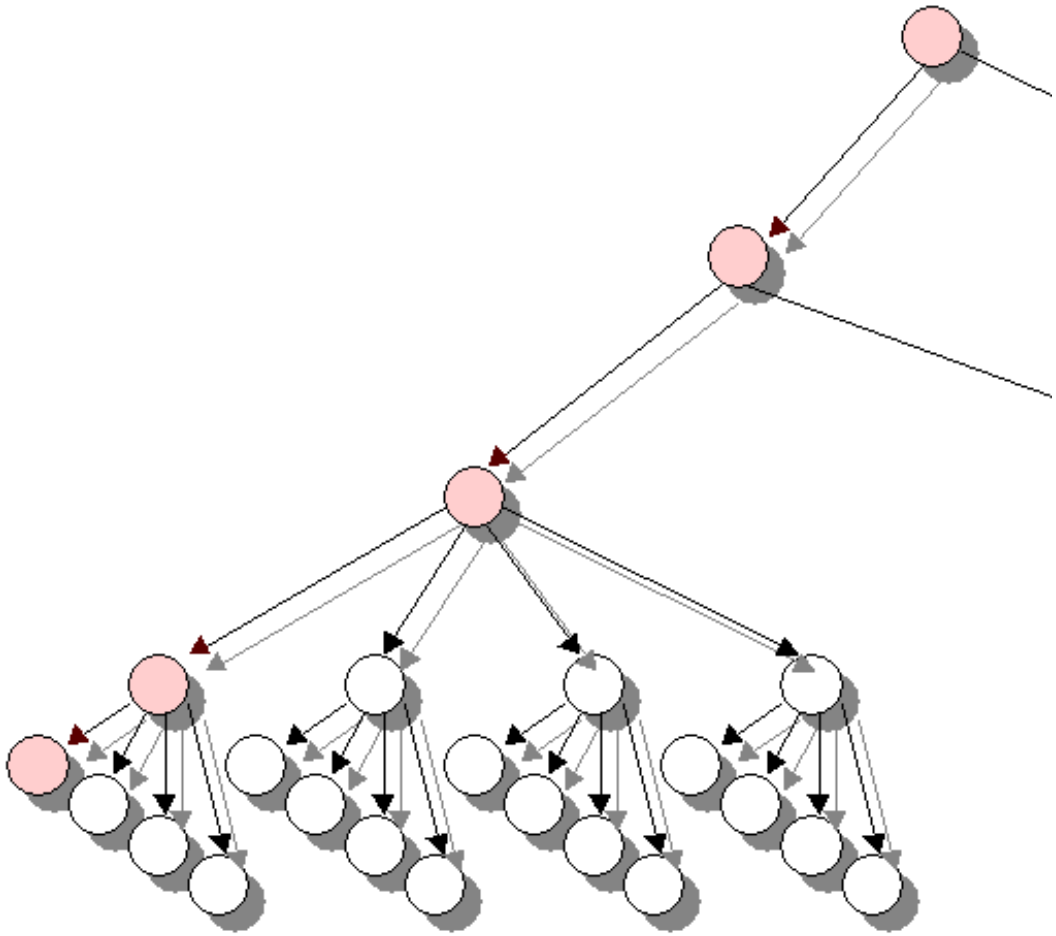
First, you should guess how you expect the algorithm to behave. Then select a point on the figure's interior. Did the algorithm act as you expected?

Serial Recursion is Depth-First

So the fill algorithm will continue in one direction until a boundary is reached.

It will then change directions momentarily and attempt to continue back in the original direction.

Will parallel execution of the algorithm behave the same way?



At Full Speed

To the right is the same algorithm operating at full speed.

Left-button click inside one of the regions to start the fill process. Click the right button to reset the image to its original state

A Flood-Fill

Sometimes we'd like a area fill algorithm that replaces all *connected* pixels of a selected color with a fill color. The *flood-fill algorithm* does exactly that.

```
            public void floodFill(int x, int y, int fill, int
old) {
                if ((x < 0) || (x >= width)) return;
                if ((y < 0) || (y >= height)) return;
                if (raster.getPixel(x, y) == old) {
                    raster.setPixel(fill, x, y);
                    floodFill(x+1, y, fill, old);
                    floodFill(x, y+1, fill, old);
                    floodFill(x-1, y, fill, old);
                    floodFill(x, y-1, fill, old);
                }
            }
```

Flood fill is a small variant on a boundary fill. It replaces *old* pixels with the *fill* color.

Flood-Fill in Action

It's a little awkward to kick off a flood fill algorithm, because it requires that the *old* color must be read before it is invoked. It is usually, established by the initial pixel *(x, y)* where a mouse is clicked.

Self-Starting Fast Flood-Fill Algorithm

The follow implementation self-starts, and is also somewhat faster.

```
            public void fillFast(int x, int y, int
fill)
            {
                if ((x < 0) || (x >= raster.width))
return;
                if ((y < 0) || (y >=
raster.height)) return;
                int old = raster.getPixel(x, y);
                if (old == fill) return;
                raster.setPixel(fill, x, y);
                fillEast(x+1, y, fill, old);
                fillSouth(x, y+1, fill, old);
                fillWest(x-1, y, fill, old);
                fillNorth(x, y-1, fill, old);
            }
```

Fill East

```
    private void fillEast(int x, int y, int fill,
int old) {
        if (x >= raster.width) return;
        if (raster.getPixel(x, y) == old) {
            raster.setPixel(fill, x, y);
            fillEast(x+1, y, fill, old);
            fillSouth(x, y+1, fill, old);
            fillNorth(x, y-1, fill, old);
        }
    }
```

Note:

There is only one clipping test, and only three subsequent calls.
Why?
How much faster do you expect this algorithm to be?

Working Algorithm

```
private void fillSouth(int x, int y, int fill, int old) {
    if (y >= raster.height) return;
    if (raster.getPixel(x, y) == old) {
        raster.setPixel(fill, x, y);
        fillEast(x+1, y, fill, old);
        fillSouth(x, y+1, fill, old);
        fillWest(x-1, y, fill, old);
    }
}
```

You can figure out the other routines yourself.

4-Way and 8-Way Connectedness

A final consideration when writing a area-fill algorithm is the size and connectivity of the neighborhood, around a given pixel.

An eight-connected neighborhood is able to get into knooks and crannies that an algorithm based on a four-connected neighborhood cannot.

Code for an 8-Way Connected Flood Fill

As you expected...
the code is a simple modification of the 4-way connected flood fill.

```
public void floodFill8(int x, int y, int fill, int old) {
    if ((x < 0) || (x >= raster.width)) return;
    if ((y < 0) || (y >= raster.height)) return;
    if (raster.getPixel(x, y) == old) {
        raster.setPixel(fill, x, y);
        floodFill8(x+1, y, fill, old);
        floodFill8(x, y+1, fill, old);
        floodFill8(x-1, y, fill, old);
        floodFill8(x, y-1, fill, old);
        floodFill8(x+1, y+1, fill, old);
        floodFill8(x-1, y+1, fill, old);
        floodFill8(x-1, y-1, fill, old);
        floodFill8(x+1, y-1, fill, old);
    }
}
```

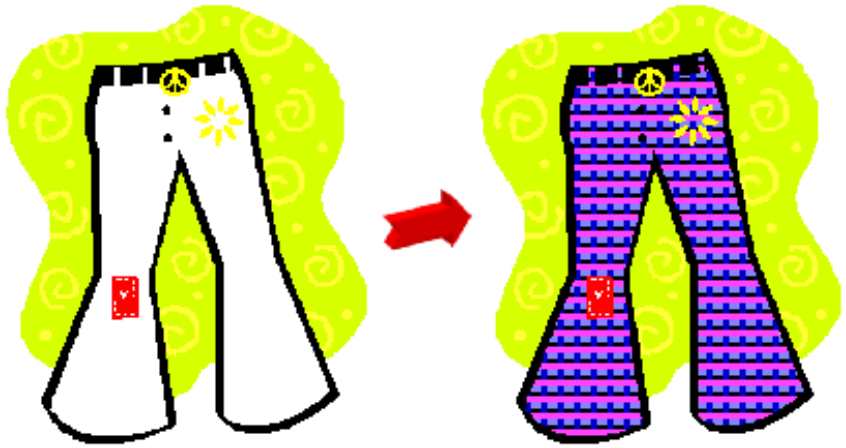
More 8-Way Connectedness

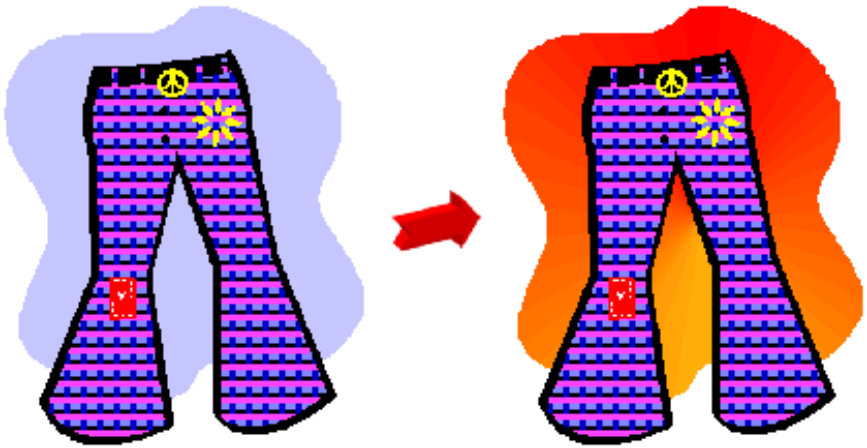
Sometimes 8-way connectedness can be too much.

Flood-Fill Embellishments

The flood-fill and boundary-fill algorithms can easily be modified for a wide variety of new uses:

1. Patterned fills
2. Approximate fills
3. Gradient fills
4. Region selects
5. Triangle Rendering!

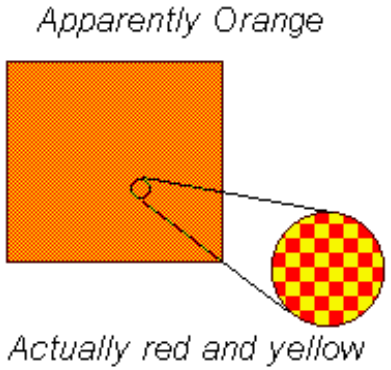




Monochrome and Color Dithering

Dithering and halftoning techniques are used to render images and graphics with more apparent colors than are actually displayable.

When our visual systems are confronted with large regions of high-frequency color changes they tend to blend the individual colors into uniform color field. Dithering and halftoning attempt to use this property of perception to represent colors that cannot be directly represented.

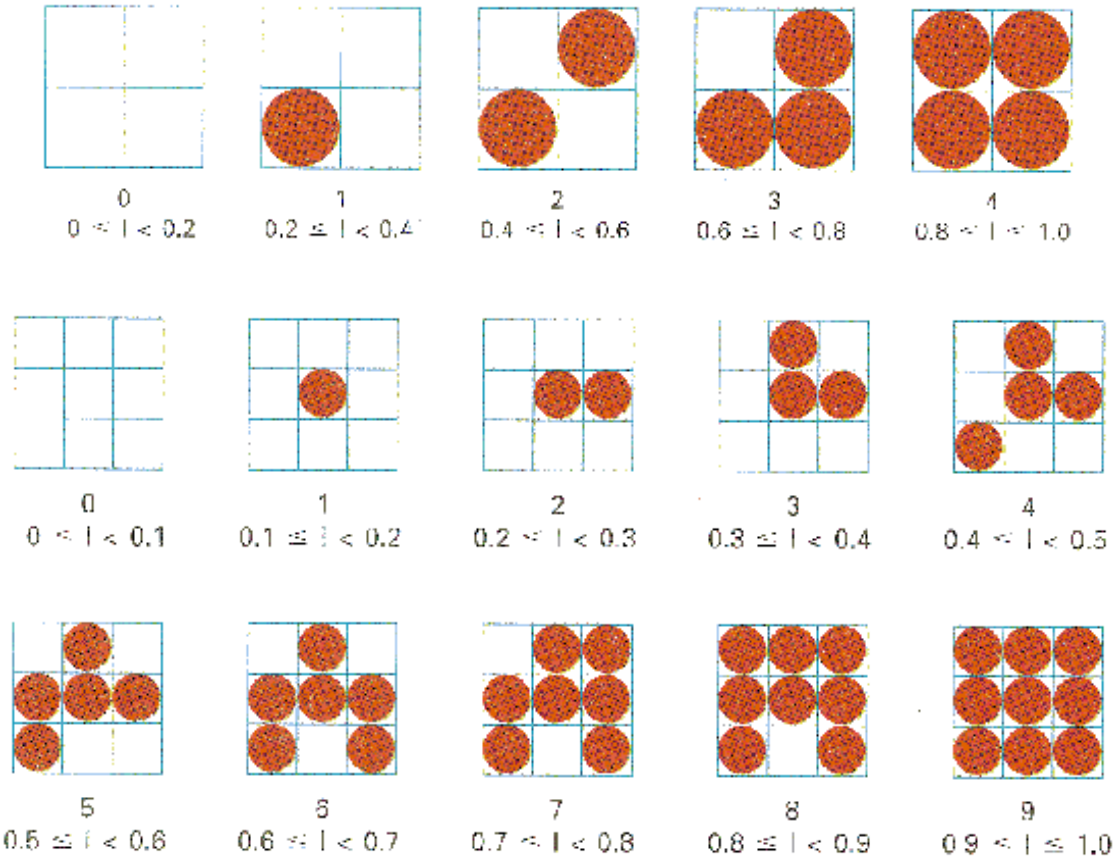


Classical Halftoning

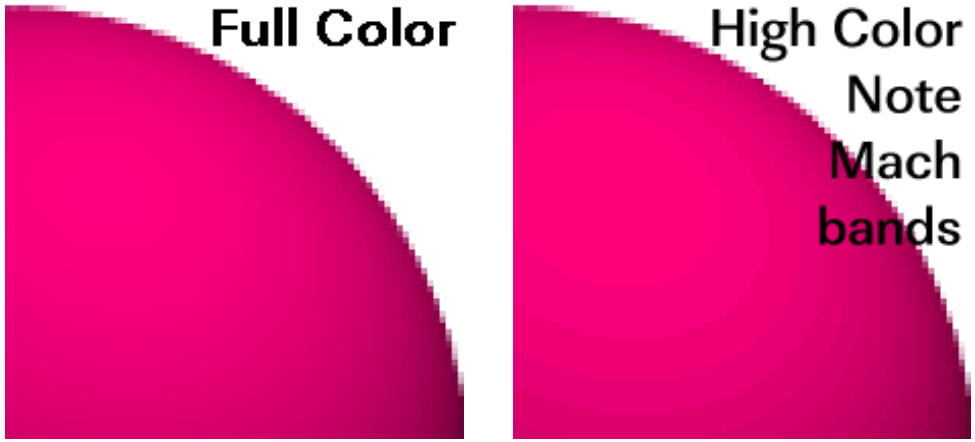
Use dots of various sizes to represent intensity, used in newspapers and magazines.

Halftoning with Pixel Patterns

Trading spatial resolution to improve intensity range.



When do we need dithering?



- We can discern approximately 100 brightness levels (depends on hue and ambient lighting)
- True-color displays are usually adequate under normal indoor lighting (when the nonlinearities of the display are properly compensated for).
- High-color displays provide only 32 shades of each primary. Without dithering you will see contours.
- Made worse by Mach-banding
- Worse on indexed displays
- Largest use of dithering is in printed media (newsprint, laser printers)

When do we need dithering?

Under fixed lighting conditions the typical person can discern approximately 100 different brightness levels. This number varies slightly with hue (for instance we can see more distinct shades of green than blue). Which particular set of 100 levels also changes as a function of the ambient lighting conditions.

The 256 colors available for each primary in a true color display are usually adequate for representing these 100 levels under normal indoor lighting (when the nonlinearities of the display are properly compensated for). Thus, there is usually no need to dither a true color display.

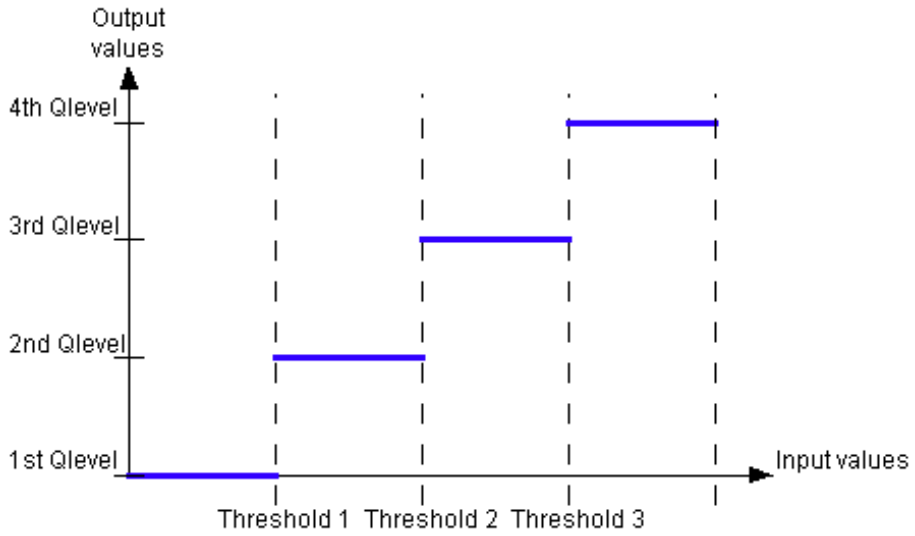
A high-color display, however, only allows 32 shades of a given primary, and without dithering you will usually be able to detect visible contours between two colors that vary by only one level. Our visual system happens to be particularly sensitive to this, and it even amplifies the variation so that it is more pronounced than the small intensity difference would suggest. This apparent amplification of contours is called Mach-banding, and it is named for the psycho physical researcher who first described it.

On index displays dithering is frequently used to represent color images. Given a 256 entry color map you can only represent approximately 6 colors per red, green, and blue primary (6x6x6=216). However, if just one or two hues are used it is possible to allocate enough color table entries (~50 per hue) so that dithering can be avoided.

By far the largest customer of dithering is in printed media. You probably seen dithering yourselves on newsprint or in the printing on continuous-tone images on a laser printer. In most printing technologies there is very little control of the shade of ink that can be deposited at a particular point. Instead only the density of ink is controlled.

Quanitization and Thresholding

The process of representing a continuous function with discrete values is called *quantization*. It can best be visualized with a drawing:

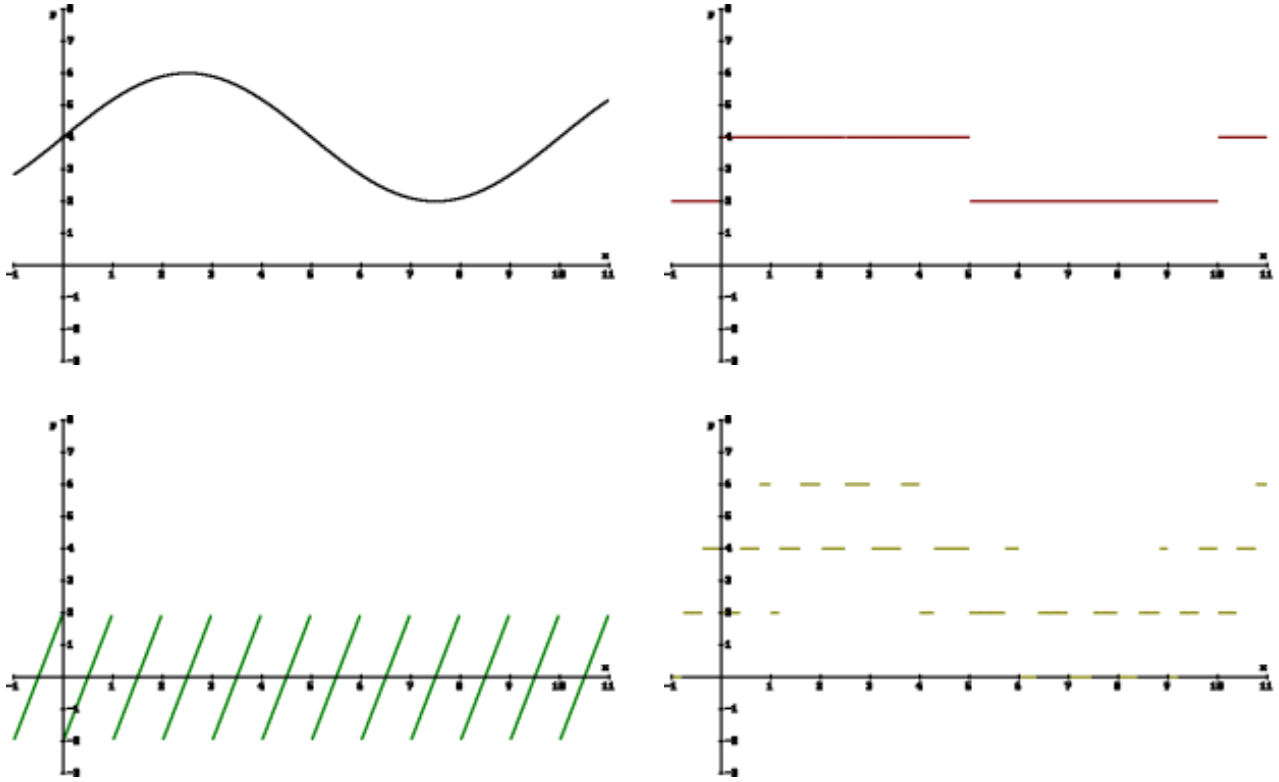


The input values that cause the quantization levels to change output values are called *thresholds*. The example above has 4 quantization levels and 3 thresholds. When the spacing between the thresholds and the quantization levels is constant the process is called *uniform quantization*.

The Secret... Noise

Dithering requires the addition of *spatial noise* to the original signal (color intensity). This noise can be regular (a repeated signal that is independent of either the input or output), correlated (a signal that is related to the input), random, or some combination.

Dithering can be neatly summarized as a quantization process where noise has has been introduced to the input. The character of the dither is determined entirely by the structure of the noise. Note: Dithering decreases the SNR yet improves the percieved quality of the output.



Dither Noise Patterns

Let's consider some dither noise patterns.

One simple type of noise is called *uniform* or *white noise*. White noise generates values within an interval such that all outcomes are equally likely. Here we add *zero-mean* white noise to our image. The term zero-mean indicates that the average value of the noise is zero. This will assure that our dithering does not change the apparent brightness of our image.

The only thing left to specify is the range of noise values, this is called the noise's *amplitude*. The amplitude that makes the most sense is the spacing between thresholds. This is not, however a requirement. It is rare to specify a larger amplitude (Why?), but frequently slightly smaller amplitudes are used. Let's look back at our example to see what random noise dithering looks like.

The result is not a good as expected. The noise pattern tends to clump in different regions of the image. The unsettling aspect of this clumping is that it is unrelated to the image. Thus the dithering process adds apparent detail to the image that are not really in the image.

Ordered Dither

The next noise function uses a regular spatial pattern. This technique is called *ordered dithering*. Ordered dithering adds a noise pattern with specific amplitudes.

2 by 2 Ordered dither noise pattern

3/8	-1/8
-3/8	1/8

Units are the fraction of the difference between quantization levels.

4 by 4 Ordered dither noise pattern

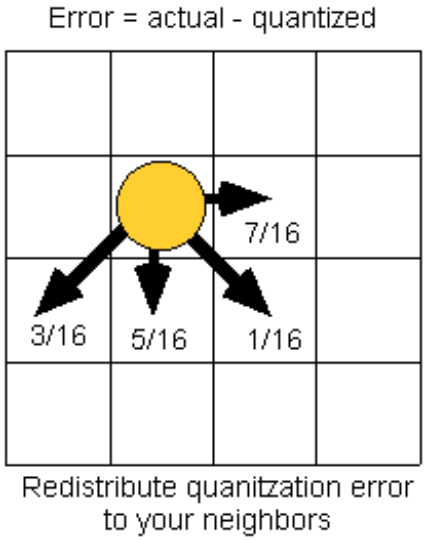
15/32	-1/32	11/32	-5/32
-9/32	7/32	-13/32	3/32
9/32	-7/32	13/32	-3/32
-15/32	1/32	-11/32	5/32

Ordered dither gives a very regular screen-like pattern remincent of newsprint.

Error Diffusion

Error diffusion is a *correlated* dithering technique, meaning that the noise value added to a given pixel is a function of the image.

Error diffusion attempts to transfer the residual error due to quantization to a region of surrounding pixels. Most common techniues are designed so that the pixels can be processed in an ordered single pass. For example the following pattern assumes that each scanline is processed from left to right as the image is scaned from top to bottom.



Error diffusion is generally, the preferred method for the display of images on a computer screen. The most common artifacts are vissible worm-like patterns, and the leakage of noise into uniform regions.

Lossy Image Compression

	GIF	JPEG
Colors	256	2^{24}
Compression	Dictionary-based (LZW)	Transform Based
Processing Order	Quantize, Compress	Transform, Quantize
Primary Use	Graphics, Line Art	Photos
Special Features	Transparency, Interleaving	Progressive

The topic of image compression is a very involved and decribes a wide range of methods. Our goal here is to provide a sufficient background for understanding the most common compression techniques used. In particular those used on the WWW.

There are generally two types of images that are widely used on the WWW, gifs and jpegs. As you are probably aware, each method has its own strengths and weaknesses. Both methods also use some form of image compression.

Some of the highlights of these compression methods are summarized in the table shown on the right.

LZW Compression

LZW compression is an acronym for Lemple-Ziv-Welch compression. This is a classical, lossless dictionary-based compression system.

Prior, to compression any GIF image must be reduced to 256 colors. This can be done using quantization followed by dithering as discussed earlier. This is the where the *loss* occurs in GIF compression.

After the image is quantized then LZW compression is applied to reduce the image size. LZW compression starts with a dictionary that contains all possible symbols (in our case numbers from 0 to 255). For the sake of illustration, I will use characters in this explanation.

Basically, build up a dictionary of longer and longer strings based on the input stream. The encoder compares the incoming characters for the longest possible match and returns the index for the that dictionary entry. It then then adds a new dictionary entry with the current character concatenated to the longest match.

LZW Compression Algorithm:

```
 w = NIL;
 while ( read a character k )
{
  if wk exists in the dictionary
   w = wk;
  else
    add wk to the dictionary;
    output the code for w;
    w = k;
}
```

LZW Compression

Input Sring =
"*WED*WE*WEE*WEB*WET"

LZW Compression Algorithm:

```
w = NIL;
while ( read a character k )
{
  if wk exists in the dictionary
   w = wk;
  else
    add wk to the dictionary;
    output the code for w;
    w = k;
}
```

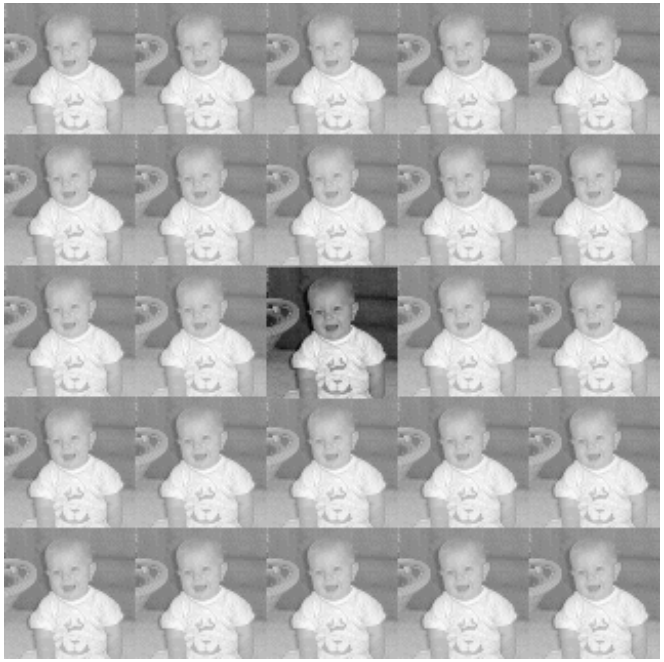
w	k	Output	Index	Symbol
NIL	*			
*	W	*	256	*W
W	E	W	257	WE
E	D	E	258	ED
D	*	D	259	D*
*	W			
*W	E	256	260	*WE
E	*	E	261	E*
*	W			
*W	E			
*WE	E	260	262	*WEE
E	*			
E*	W	261	263	E*W
W	E			
WE	B	257	264	WEB
B	*	B	265	B*
*	W			
*W	E			
*WE	T	260	266	*WET
T	EOF	T		

Transform Coding

Transform coding removes the redundancies (correlation) in an images by changing coordinate systems.

We can think of a cluster of pixels, for instance those in an 8 by 8 block, as a vector in some high-dimensional space (64 dimension in this case). If we transform this matrix appropriately we can discover that a otherwise random collection of numbers is, in fact, highly correlated.

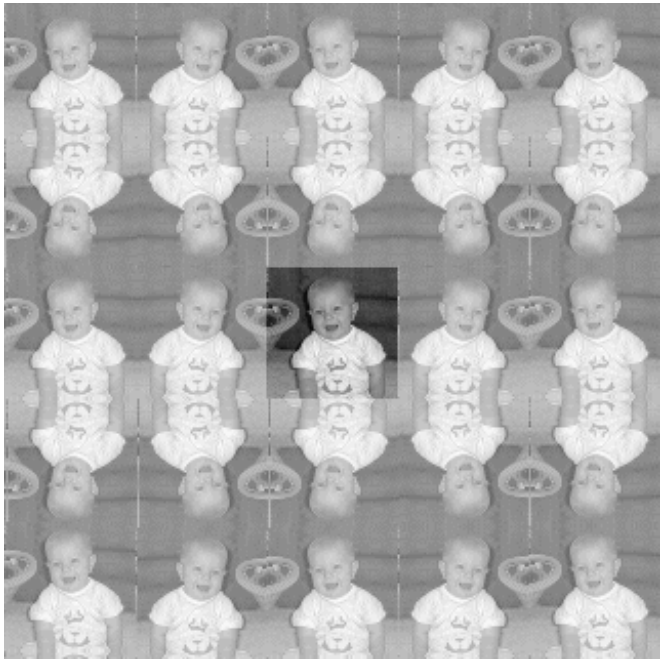
One common transform basis function that you have seen if the Fourier Basis (FFT). However the Fourier basis makes some assumptions that are not optimal for images.



First, it assumes that the image tiles an infinite plane. This leads to a transform that contains both even (cosine-like) and odd (sine-ike) components.

If we make copies of the images with various flips we can end up with a function that has only even (cosine-like) components.

The resulting FFT will only have real parts. This transform is called the *Cosine Transform*.



DCT example

By clicking on the image below you can transform it to and from it's Discrete Cosine Representation.

The DCT like the FFT is a *separable* transfrom. This means that a 2-D transfrom can be realized by first transfroming in the x direction, followed by a transforms in the y direction.

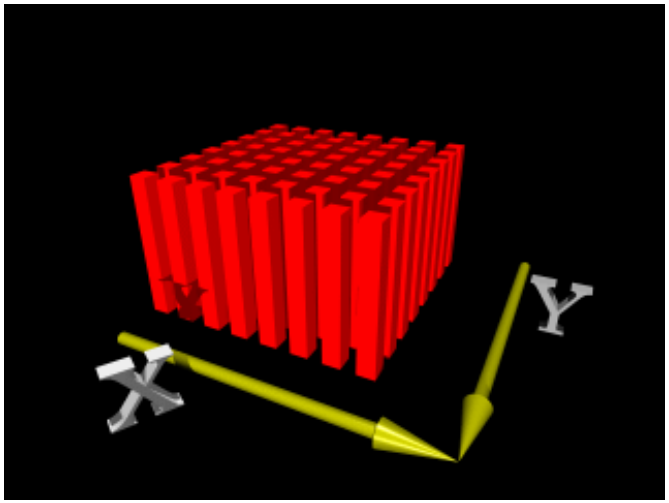
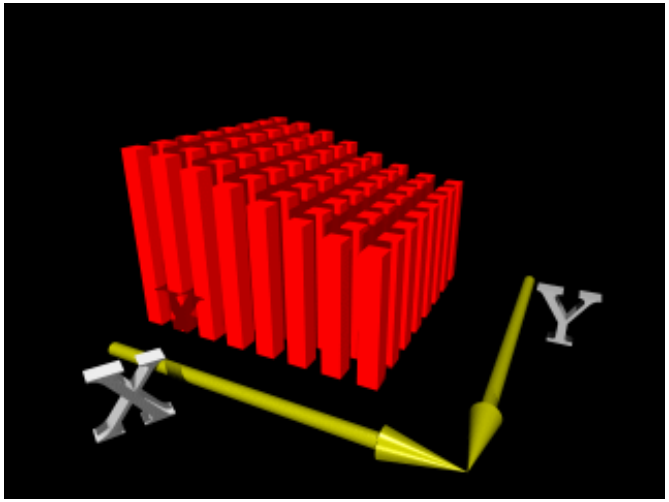
The blocksize shown here is 8 by 8 pixels.

Notice how the transformed image is a uniform gray color. This is indicative of a correlated image. In fact, most of the coefficients in a DCT transform are typically very close to zero (shown as gray here because these coefficients are signed values). Both JPEG and MPEG take advantage of this property by applying a quantization to these coefficents, which causes most values to be zero.

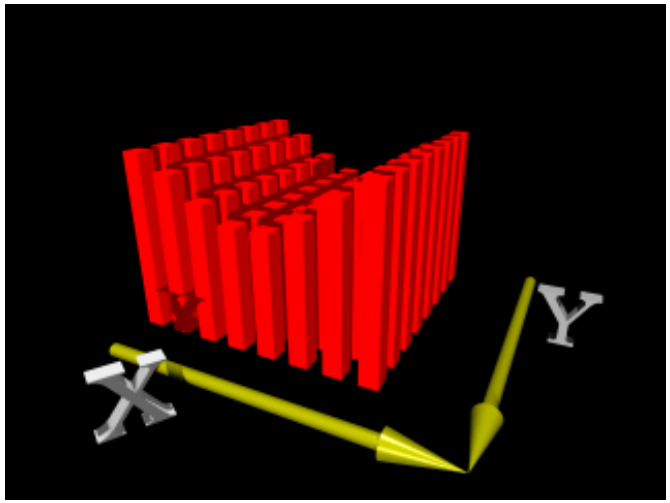
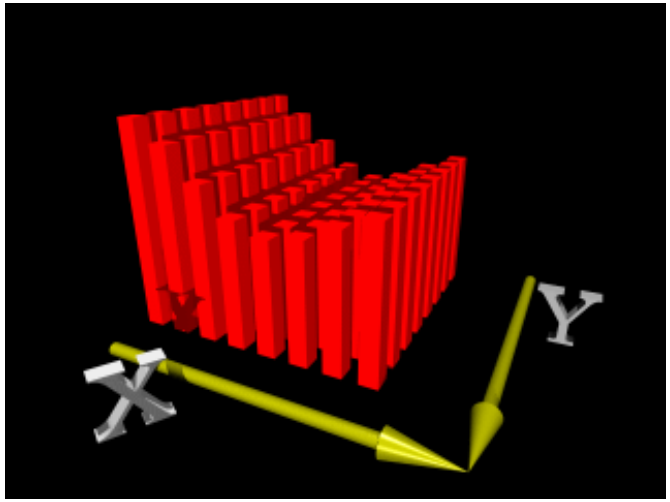
If we ignore floating point trucation errors, then the DCT transformation is an entirely lossless transform. The loss incurred in JPEG and MPEG types of compression is due to the quantization that is applied to each of the cofficients after the DCT transform.

See Description of MPEG including DCT tutorial

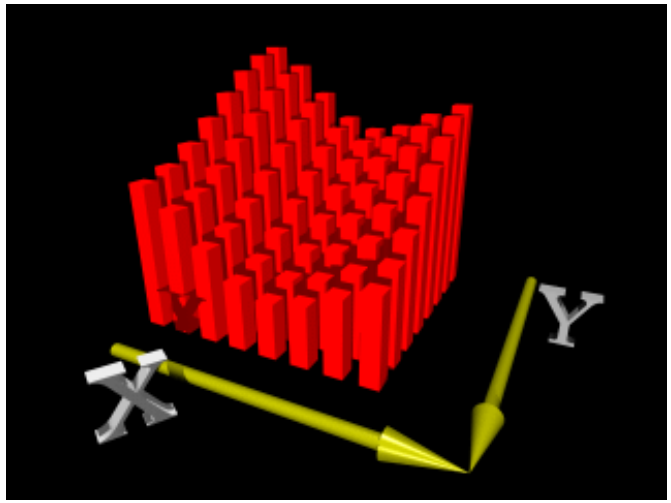
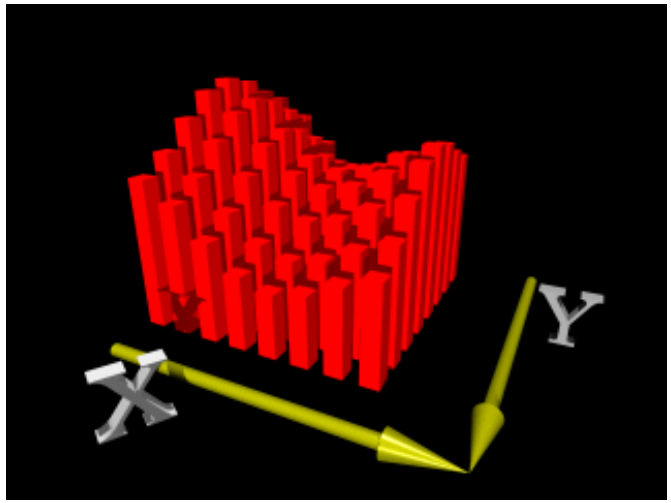
DCT examples

Input Image	Input Data	Input Graph	DCT Coefficients
Constant Gray	87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87 87		700 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 0 0 0 0 0 0 0 0 0
	105 102 97 91 84 78 73 70 105 102 97 91 84 78 73 70 105 102 97 91 84 78 73 70 105 102 97 91 84 78 73 70 105 102 97 91 84 78 73 70 105 102 97 91 84 78 73 70 105 102 97 91 84 78 73 70 105 102 97 91 84 78 73 70		700 100 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 0 0 0 0 0 0 0 0

DCT examples

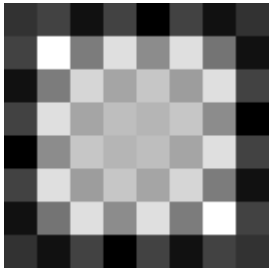
Input Image	Input Data	Input Graph	DCT Coefficients
	104 94 81 71 71 81 94 104 104 94 81 71 71 81 94 104 104 94 81 71 71 81 94 104 104 94 81 71 71 81 94 104 104 94 81 71 71 81 94 104 104 94 81 71 71 81 94 104 104 94 81 71 71 81 94 104 104 94 81 71 71 81 94 104		700 0 100 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 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 0 0 0 0 0 0 0
	121 109 91 75 68 71 80 86 121 109 91 75 68 71 80 86 121 109 91 75 68 71 80 86 121 109 91 75 68 71 80 86 121 109 91 75 68 71 80 86 121 109 91 75 68 71 80 86 121 109 91 75 68 71 80 86 121 109 91 75 68 71 80 86		700 100 100 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 0 0 0 0 0 0 0

DCT examples

Input Image	Input Data	Input Graph	DCT Coefficients
	156 144 125 109 102 106 114 121 151 138 120 104 97 100 109 116 141 129 110 94 87 91 99 106 128 116 97 82 75 78 86 93 114 102 84 68 61 64 73 80 102 89 71 55 48 51 60 67 92 80 61 45 38 42 50 57 86 74 56 40 33 36 45 52		700 100 100 0 0 0 0 0 200 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 0
	120 108 90 75 69 73 82 89 127 115 97 81 75 79 88 95 134 122 105 89 83 87 96 103 137 125 107 92 86 90 99 106 131 119 101 86 80 83 93 100 117 105 87 72 65 69 78 85 100 88 70 55 49 53 62 69 89 77 59 44 38 42 51 58		700 90 100 0 0 0 0 0 90 0 0 0 0 0 0 0 -89 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

DCT examples

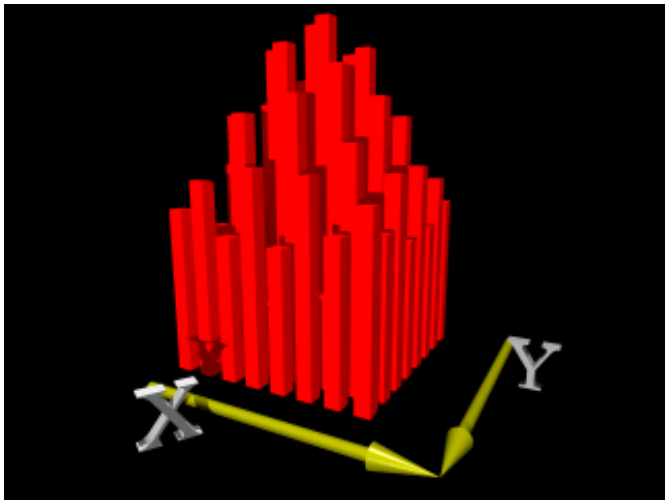
Input Image



Input Data

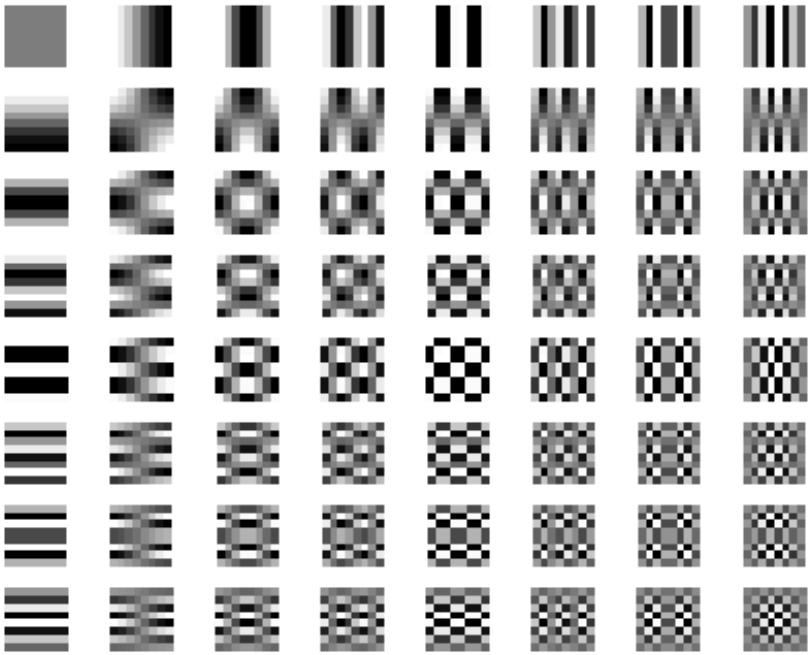
124	105	139	95	143	98	132	114
105	157	61	187	51	176	80	132
139	61	205	17	221	32	176	98
95	187	17	239	0	221	51	143
143	51	221	0	239	17	187	95
98	176	32	221	17	205	61	139
132	80	176	51	187	61	157	105
114	132	98	143	95	139	105	124

Input Graph



DCT Coefficients

950	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	0
0	0	0	0	0	0	0	500



Next Time

Drawing Lines