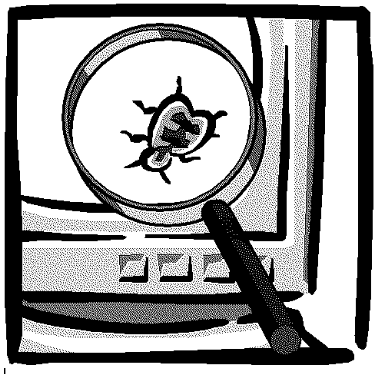
Antialiasing and Resampling

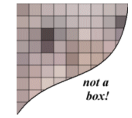
- What is a pixel?
- Signal Processing
- Image Resampling
- Image Dithering



What is a Pixel?

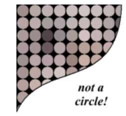
A pixel is not...

- a box
- a disk
- a teeny tiny little light



A pixel is a point...

- it has no dimension
- it occupies no area
- it cannot be seen
- it can have a coordinate



A pixel is *more* than just a point, it is a *sample*

More on Samples

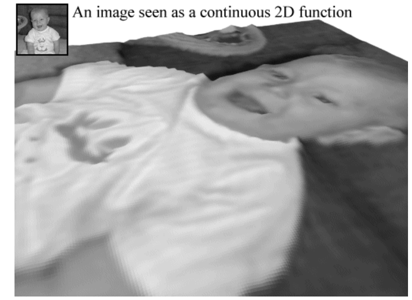
Most things in the real world are continuous
Everything in a computer is discrete
The process of mapping a continuous *function* to a discrete one is called *sampling*
The process of mapping a continuous *variable* to a discrete one is called *quantization*
When we represent or render an image using a computer we must both sample and quantize

Picturing an Image as a 2D Function

An *ideal* image can be viewed as a function, *I(x, y)*, that gives an intensity for any given coordinate *(x, y)*. We could plot this function as a height field. This plot would lowest at dark points in the image and highest at bright points.

Most of the functions that we encounter are *analytic* and, therefore, can be expressed as simple algebraic relations. An image, in general, cannot representaed in such a simple form. Instead, we represent images as tabulated functions.

So how do we fill this table?



An image seen as a continuous 2D function

Sampling Grid

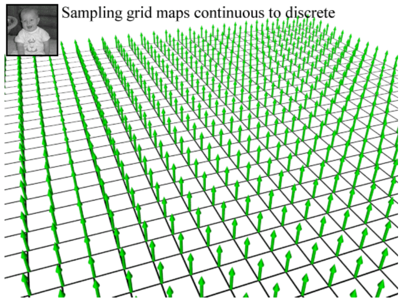
The most common (but not the only) way to generate the table values necessary to represent our function is to multiply the function by a *sampling grid*. A sampling grid is composed of periodically spaced Kronecker delta functions.

The definiton of the 2-D Kronecker delta is:

$$\delta(x,y)=\begin{cases}1, & (x,y)=(0,0)\\ 0, & \text{otherwise}\end{cases}$$

And a 2-D sampling grid:

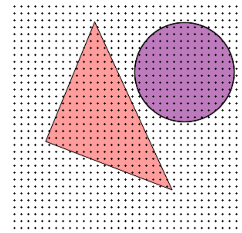
$$\sum_{j=0}^{h-1}\sum_{i=0}^{w-1}\delta(u-i,v-j)$$


Sampling grid maps continuous to discrete

Sampling an Image

When a continuous image is multiplied by a sampling grid a discrete set of points are generated. These points are called samples. These samples are pixels. We store them in memory as arrays of numbers representing the intensity of the underlying function.

The same analysis can be applied to geometric objects:



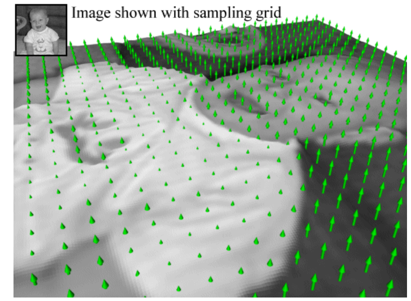


Image shown with sampling grid

The Big Question

How densely must we sample an image in order to capture its essence?

Since our sampling grid is *periodic* we can appeal to Fourier analysis for an answer. Fourier analysis states that all periodic signals can be represented as a summation of sinusiodal waves. Thus every image function that we understand as a height field in the *spatial domain*, has a dual representaion in the *frequency domain*.

We can transform signals from one domain to the other using the Fourier transform.

$$F(u,v)=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)\,e^{-i2\pi(ux+vy)}dxdy$$

$$f(x,y)=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u,v)\,e^{\,i2\pi(ux+vy)}dudv$$

Convolution

In order to simplify our analysis we will consider 1-D signals for the moment. It will be straightforward to extend the result to 2-D.

Some operations that are diffcult to compute in the spatial domain are simplified when the function is tranformed to its dual representation in the frequency domain. One such function is *convolution*.

Convolution describes how a system with impulse response, *h(x)*, reacts to a signal, *f(x)*.

$$f(x)*h(x)=\int_{-\infty}^{\infty} f(\lambda)h(x-\lambda)d\lambda$$

This integral evaluation is equivalent to multiplication in the frequency domain

$$f(x)*h(x)\rightarrow F(u)H(u)$$

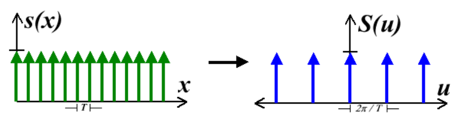
The converse is also true

$$F(u)*H(u)\rightarrow f(x)h(x)$$

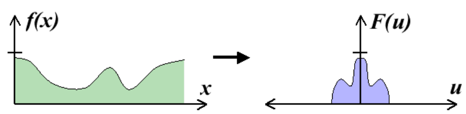
Sampling in the Frequency Domain

Image sampling was defined as multiplying a periodic series of delta functions by the continuous image. This is the same as convolution in the frequency domain.

Consider the sampling grid:

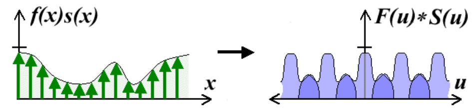


And the function being sampled

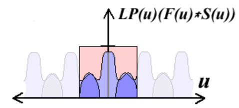


Reconstruction

This amounts to accumulating copies of the function's spectrum centered at the delta functions of the sampling grid.



Remember the goal of a sampled represention is to faithfully represent the underlying function. Ideally we would apply a *low-pass filter* to our sampled representation to *reconstruct* our original function. We will call this processing a *reconstruction filter*.

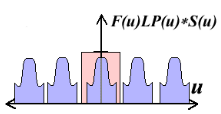


In this example we mixed together copies of our function (as seen in the darker overlap regions). In this case subsequent processing does not allow us to separate out a representative copy of our function.

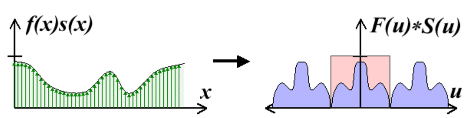
Aliasing

This mixing of spectrums is called *aliasing*. It has the effect of introducing high-frequencies into our original function. There are two ways of dealing with aliasing.

The first is to low pass filter our signal before we sample it.



The second is to increase the sampling frequency



Sampling Theorem

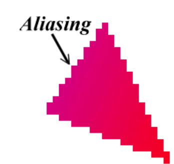
In order to have any hope of accurately reconstructing a function from a periodically sampled version of it, two conditions must be satisfied:

1. The function must be bandlimited.
2. The sampling frequency, f_s, must be at least twice the maximum frequency, f_{max}, of the function.

Satisfying these conditions will eliminate aliasing.

In practice:

- "Jaggies" are aliasing
-
- Both of the techniques discussed are used
  1. Super-sampling (more samples than pixels)
  2. Low-pass prefiltering (averaging of super-samples)



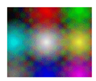
Reconstruction Revisited

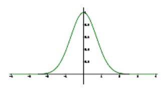
Once we've satified the sampling theorem we can theoretically reconstruct the function. This requires low-pass filtering the samples that are used to represent the image.

Remember the samples that we store in memory are *points*; they have no area only positions. It is this *reconstruction filter* that gives a pixel area.

Question: If what I say is true how can we just stick pixel values in our frame buffer and see a picture? If they were just points (like I claim) we would see anything.

The fact is that a CRT has a built in reconstruction filter. Most real world devices do because they do not have an infinite frequency response.



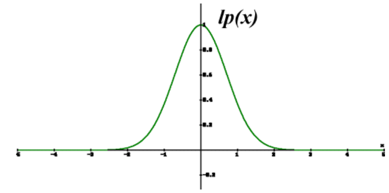


Each pixel illuminates a spot on the screen whose intensity fall off is well approximated by a gaussian.

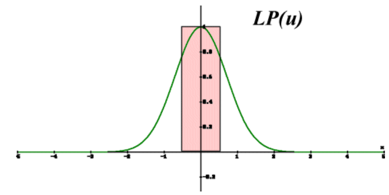
Previous Lecture 11 Slide 14 6.837 Fall '98 Next

Gaussian Reconstruction

So how does a Gaussian do as a reconstruction filter? In the spatial domain:



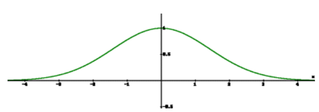
In the frequency domain:

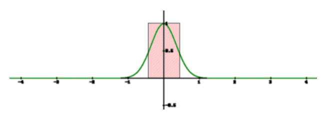


Previous Lecture 11 Slide 15 6.837 Fall '98 Next

Gaussian Verdict

- Wider in spatial domain corresponds to narrower in the frequency domain





- We are forced to live with either *low-frequency attenuation* of *high-frequency leakage*
- Live with blur
- Infinite extent
- CRT resolution
- Only reconstruction kernel that is both *linearly separable* and *radially symmetric*

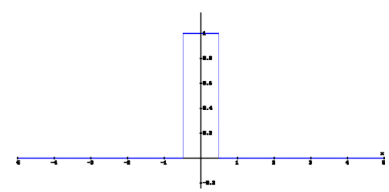
For a CRT images we basically have to live with the reconstruction filters that we are given. However, when we write an magnification program we are in control of the magnification.

Previous Lecture 11 Slide 16 6.837 Fall '98 Next

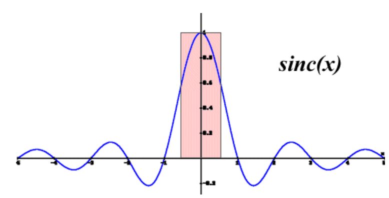
Nearest-Neighbor Reconstruction

One choice of a reconstruction filter has the shape of a simple box in the spatial domain. **Note:** This is not the shape of the pixel but rather the shape of the reconstruction filter!

In the spatial domain:



In the frequency domain:



Previous Lecture 11 Slide 17 6.837 Fall '98 Next

Nearest-Neighbor Example

Blockiness is due to high-frequency leakage

Not the same as aliasing!

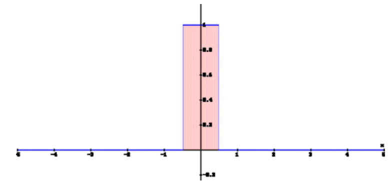
Even properly sampled images can be rendered as blocky

This is a rahter poor reconstruction

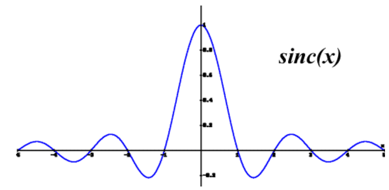
Is there an Ideal Reconstruction Filter?

Theoretically yes, practically no

We can work backwards from our desired frequency domain response:

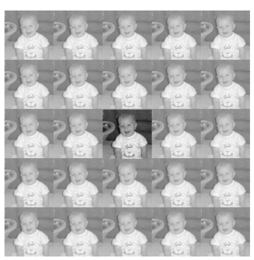


To compute the ideal reconstruction filter in the spatial domain:



Problems with a Sinc Reconstruction Filter

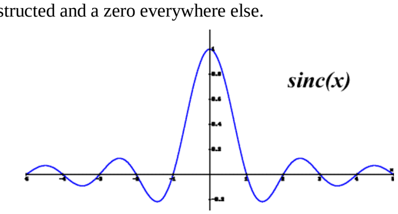
- It has infinite spatial extent. Every pixel contributes to every point that we interpolate.
- Hard to compute in the spatial domain.
- Assumes that the samples repeat periodically



- Infinite sum
- Gibbs phenomenon

Lessons from the Sinc Function

Notice the values of the sinc function at the sample points (integer values). It has a value of one at the point being reconstructed and a zero everywhere else.



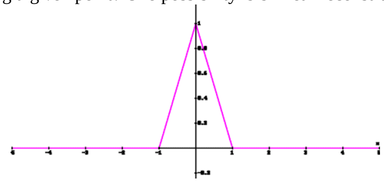
This means that a sinc will exactly reconstruct the sample values at the sample points.

It has even symmetry (like a cosine function)

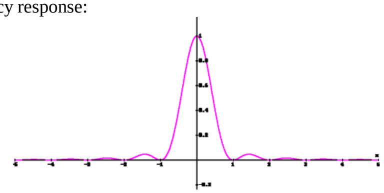
It has a slope of 0 at the sample point.

Approximations with Finite Extent

What sort of reconstruction we can do if we limit ourselves to considering only the pixels immediately surrounding a given point? One possibility is bilinear reconstruction:



Which has the frequency response:



Bilinear Example

Sometimes called a
tent filter

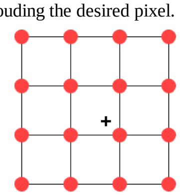
High-frequencies are significantly attenuated

Easy to implement
(just linearly interpolate between samples)

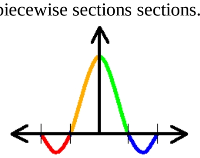
Still can produce artifacts

Larger Extents

We can use our knowledge about splines to design filters with arbitrary extents. Here we will consider a 4 by 4 neighborhood surrounding the desired pixel.



We can constrain our design usin the knowledge that we have gleaned from the ideal **sinc** reconstruction filter.
First, we divide our spline into four piecewise sections sections.



Spline Constraints

The piecewise cubic spline should satisfy the following:

1. The function must be symmetric
2. $\mathbf{C}_0$ and $\mathbf{C}_1$ continuity between spline segments
3. In order to assure that a uniform array of samples is reconstructed as a uniform continuos function it should satisfy the following convex relationship:

$$\sum_{n=-\infty}^{\infty} k(x-n) = 1$$

We can reformulate our general cubic such that it is symmetric as follows:

$$k(x) = A|x|^3 + Bx^2 + C|x| + D$$

This also allows us to consider only 2 segments instead of 4.

Bicubic Derivation

Starting with the equations of our two spline segments and their derivitives we can't construct the two desired spline segments.

The resulting family of reconstruction filters is:

$$k(x) = \frac{1}{6}\begin{cases} (12-9B-6C)|x|^3 + (-18+12B+6C)x^2 + (6-2B) & \text{if } |x|<1 \\ (-B-6C)|x|^3 + (6B+30C)x^2 + (-12B-48C)|x| + (8B+24C) & \text{if } 1 \le |x| \le 2 \\ 0 & \textit{otherwise} \end{cases}$$

Bicubic Example

Mitchell's 2-parameter spline family

Very low leakage

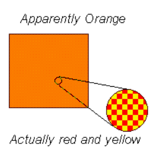
Includes all of Cardinal splines
(including Catmull-Rom)

Includes bicubic B-splines

Monochrome and Color Dithering

Dithering techniques are used to render images and graphics with more *apparent* colors than are actually available on a display. Dithering is sometimes called *digital half-toning*.

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 attempts to uses this property of perception to represent colors that cannot be directly represented.



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 names for the psycho physical researcher who first described it.

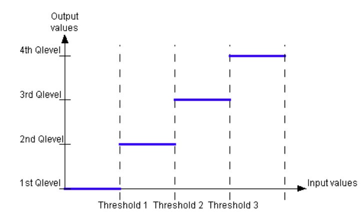
On index displays dithering is frequently used to represent color images. Given a 256 entry color map you can only represent approximately 6 colors per red, green, and blue primary (6x6x6=216). However, if just one or two hues are used it is possible to allocate enough color table entries (~50 per hue)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

Terminology

There are three key ideas that come to play when converting continuous images to dithered images. They are *quantization*, *thresholding*, and *noise*.

The process of representing a continuous function with discrete values is called quantization (Where have we seen this before?). 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 four quantization levels and three thresholds. When the spacing between the thresholds and the quantization levels is constant the process is called *uniform quantization*. Today, I will only discuss dithering in the context of uniform quantization. However, the concepts are easily adapted to any quantization scheme.

Let's look at some examples:

When a signal, or color, is modified from it original value by the addition of some other value, it is said that *noise* has been added. This variation can be regular (a repeated signal that is independent of either the input or output), correlated (a signal that is relate to the input), random, or some combination.

Dithering can be neatly summarized as a quantization process in which noise has has been introduced to the input. The character of the dither is determined entirely by the structure of the noise.

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 some interval such that all values are equally likely. We will *zero-mean* white noise to our image. The term zero-mean indicates that the average value of the noise is zero, therefore, the interval of values must be symmetric about the value of zero so that there will be a noise value of opposite sign to cancel each generated value. Since all values are equally likely, his assures that the result has an average value of zero.

The only thing left to specify is the range of noise values, this is called the noise's *amplitude*. **The noise amplitude that makes the most sense when uniform quantization is used 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.

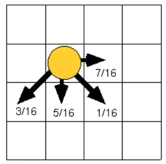
The next noise function uses a regular spatial pattern. This technique is called *ordered dithering*. Ordered dithering adds a noise pattern with specific amplitudes.

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

The final technique, called *error diffusion*, uses correlated noise based on the input image.

Error = actual - quantized



Redistribute quanitzation error
to your neighbors

Implementation

The following is an abstract class that all of my dithering demos are derived from

```
import java.awt.*;
import Raster;

public abstract class DitherRaster extends Raster {
    protected int quantize[];
    protected int threshold;

    public DitherRaster(Image img)
    {
        super(img);
    }

    public DitherRaster(DitherRaster in)
    {
```



```
        width = in.width;
        height = in.height;
        pixel = in.pixel;
        quantize = in.quantize;
        threshold = in.threshold;
    }

    public void setQuantLevels(int levels)
    {
        if (quantize == null)
            quantize = new int[256];

        threshold = (510 + levels) / (2*levels);
        for (int i = 0; i < 256; i++) {
            int q = Math.round(255*Math.round(i*(levels-1)/255.0)/(levels-1));
            quantize[i] = q;
        }
    }

    public abstract int getNoiseyPixel(int x, int y);

    public void draw(Raster bgnd)
    {
        int x = (bgnd.width - width) / 2;
        int y = (bgnd.height - height) / 2;
        for (int j = 0; j = 0) && (y+j < bgnd.height)) {
                for (int i = 0; i = 0) && (x+i < bgnd.width)) {
                        int pix = getNoiseyPixel(i, j);
                        if ((pix & 0xff000000) != 0)
                            bgnd.setPixel(pix, x+i, y+j);
                    }
                }
            }
        }
    }
}
```

The following implements the no dithering case:

```
import java.awt.*;
import DitherRaster;

public class NoDitherRaster extends DitherRaster {

    public NoDitherRaster(Image img)
    {
        super(img);
    }

    public NoDitherRaster(DitherRaster in)
    {
        super(in);
```



```
    }

    public int getNoiseyPixel(int x, int y) {
        return getPixel(x,y);
    }
}
```

Then the random dithering raster.

```
import java.awt.*;
import java.util.*;
import DitherRaster;

public class RandomRaster extends DitherRaster {
    Random whiteNoise;

    public RandomRaster(Image img)
    {
        super(img);
        whiteNoise = new Random();
    }

    public RandomRaster(DitherRaster in)
    {
        super(in);
        whiteNoise = new Random();
    }

    private int random;
    private int rcount;

    public int getNoiseyPixel(int x, int y) {
        int pix = getPixel(x,y);
        int a = pix & 0xff000000;
        int r = (pix >> 16) & 255;
        int g = (pix >> 8) & 255;
        int b = pix & 255;

        if (rcount == 0) {
            random = whiteNoise.nextInt();
            rcount = 4;
        }

        int noise = (random & 255) - 128;
        noise = (threshold * noise) >> 8;
        random >>= 8;
        rcount -= 1;

        r += noise;
        g += noise;
        b += noise;
```



```
        r = ((r & ~255) == 0) ? r : ((r < 0) ? 0 : 255);
        g = ((g & ~255) == 0) ? g : ((g < 0) ? 0 : 255);
        b = ((b & ~255) == 0) ? b : ((b < 0) ? 0 : 255);

        r = quantize[r] << 16;
        g = quantize[g] << 8;
        b = quantize[b];
        return (a|r|g|b);
    }
}
```

Then the ordered dithering raster.

```
import java.awt.*;
import java.util.*;
import DitherRaster;

public class OrderedDither extends DitherRaster {
    protected int pattern[];

    public OrderedDither(Image img)
    {
        super(img);
        pattern = new int[16];
        initPattern();
    }

    public OrderedDither(DitherRaster in)
    {
        super(in);
        pattern = new int[16];
        initPattern();
    }

    public void setQuantLevels(int levels)
    {
        super.setQuantLevels(levels);
        initPattern();
    }

    public void initPattern()
    {
        pattern[0]  = (15*threshold)/32;
        pattern[1]  = (-1*threshold)/32;
        pattern[2]  = (11*threshold)/32;
        pattern[3]  = (-5*threshold)/32;
        pattern[4]  = (-9*threshold)/32;
        pattern[5]  = (7*threshold)/32;
        pattern[6]  = (-13*threshold)/32;
        pattern[7]  = (3*threshold)/32;
        pattern[8]  = (9*threshold)/32;
        pattern[9]  = (-7*threshold)/32;
```

```
        pattern[10] = (13*threshold)/32;
        pattern[11] = (-3*threshold)/32;
        pattern[12] = (-15*threshold)/32;
        pattern[13] = (1*threshold)/32;
        pattern[14] = (-11*threshold)/32;
        pattern[15] = (5*threshold)/32;
    }

    public int getNoiseyPixel(int x, int y) {
        int pix = getPixel(x, y);
        int a = pix & 0xff000000;
        int r = (pix >> 16) & 255;
        int g = (pix >> 8) & 255;
        int b = pix & 255;

        int i = 4*(y & 3) + (x & 3);
        r = r + pattern[i];
        g = g + pattern[i];
        b = b + pattern[i];

        r = ((r & ~255) == 0) ? r : ((r < 0) ? 0 : 255);
        g = ((g & ~255) == 0) ? g : ((g < 0) ? 0 : 255);
        b = ((b & ~255) == 0) ? b : ((b < 0) ? 0 : 255);

        r = quantize[r] << 16;
        g = quantize[g] << 8;
        b = quantize[b];
        return (a|r|g|b);
    }
}
```

Then the error-diffusion dithering raster.

```
import java.awt.*;
import java.util.*;
import DitherRaster;

public class ErrorDiffusion extends DitherRaster {
    protected int rerror[][];
    protected int gerror[][];
    protected int berror[][];

    public ErrorDiffusion(Image img)
    {
        super(img);
        rerror = new int[2][width];
        gerror = new int[2][width];
        berror = new int[2][width];
        zeroError();
    }

    public ErrorDiffusion(DitherRaster in)
```



```
    {
        super(in);
        rerror = new int[2][width];
        gerror = new int[2][width];
        berror = new int[2][width];
        zeroError();
    }

    public void setQuantLevels(int levels)
    {
        super.setQuantLevels(levels);
        zeroError();
    }

    private void zeroError()
    {
        for (int i = 0; i < width; i++) {
            rerror[0][i] = 0;   rerror[1][i] = 0;
            gerror[0][i] = 0;   gerror[1][i] = 0;
            berror[0][i] = 0;   berror[1][i] = 0;
        }
    }

    public int getNoiseyPixel(int x, int y) {
        int pix = getPixel(x,y);
        int a = pix & 0xff000000;
        int r = (pix >> 16) & 255;
        int g = (pix >> 8) & 255;
        int b = pix & 255;

        int thisrow = y & 1;
        int nextrow = (y+1) & 1;

        r = r + rerror[thisrow][x];
        g = g + gerror[thisrow][x];
        b = b + berror[thisrow][x];

        r = ((r & ~255) == 0) ? r : ((r < 0) ? 0 : 255);
        g = ((g & ~255) == 0) ? g : ((g < 0) ? 0 : 255);
        b = ((b & ~255) == 0) ? b : ((b < 0) ? 0 : 255);

        int qr = quantize[r];
        int qg = quantize[g];
        int qb = quantize[b];

        rerror[thisrow][x] = 0;
        gerror[thisrow][x] = 0;
        berror[thisrow][x] = 0;

        r -= qr;
        g -= qg;
        b -= qb;
        rerror[nextrow][x] += (5*r + 8) >> 4;
```



```
        gerror[nextrow][x] += (5*g + 8) >> 4;
        berror[nextrow][x] += (5*b + 8) >> 4;

        if (x-1 >= 0) {
            rerror[nextrow][x-1] += (3*r + 8) >> 4;
            gerror[nextrow][x-1] += (3*g + 8) >> 4;
            berror[nextrow][x-1] += (3*b + 8) >> 4;
        }

        if (x+1 > 4;
            rerror[nextrow][x+1] += (r + 8) >> 4;
            gerror[thisrow][x+1] += (7*g + 8) >> 4;
            gerror[nextrow][x+1] += (g + 8) >> 4;
            berror[thisrow][x+1] += (7*b + 8) >> 4;
            berror[nextrow][x+1] += (b + 8) >> 4;
        }

        r = qr << 16;
        g = qg << 8;
        b = qb;
        return (a|r|g|b);
    }
}
```

This page last updated on Monday, October 07, 1996