A Review of Some Image Pixel Interpolation Algorithms

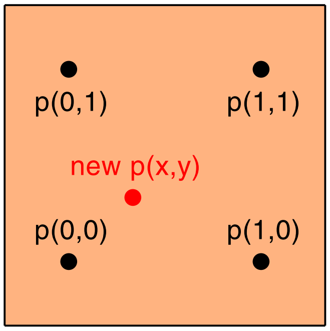
Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552
copyright c2007 as GuruGram #32
http://www.tinaja.com
don@tinaja.com
(928) 428-4073

It often may be necessary to resample a **bitmap** pixel image. Perhaps because it is to be resized, rotated, or have its perspective corrected, or get intentionally **distorted**, or have its image shape rectified. It turns out there are several newer approaches to this problem that differ markedly in image quality, in processing speed, complexity, language programmability, and ease of understanding.

What I thought I would do here is review some pixel interpolation methods that include **Nearest Neighbor**, **Bilineal Interpolation**, **Bilineal With Lookup**, a **3x3 Compromise**, and **Bicubic Interpolation**. I was unable to find any derivation of Bicubic Interpolation that I felt was reasonably complete and understandable, so we will expand upon it in detail here.

The Interpolation Problem

Typically, you will have a sampled data system representing your image. With a two dimensional array of samples usually linearly spaced in the **x** (horizontal) and **y** (vertical) directions. For one reason or another, you will require a group of new sample points intermediate to your input pixels. Typically, one new pixel will have four **nearest neighbors** on a rectangular grid...



It is often simplest to **assume a unit square between the four nearest pixels**. Our new and sought after pixel will then be some **x fraction** and **y fraction** into this unit square. Your interpolation problem then consists of finding suitable values for these fractions.

Your sought after new pixel position should be the result of some pixel-by-pixel calculation seeking to change the size or distortion of the image. Both the new **x** and **y** results will have an **integer** portion that decides **which** initial group of four pixels to use. Plus a **residue** or **fractional remainder** portion that positions you **within** the selected four sample square.

Your task is to then find a credible intensity value for that new location.

Typically, you will have to repeat your interpolation **at least three times**, once for each of the red, green, and blue pixel bitmap planes. Thus, algorithm speed can easily become a key limiting issue. Especially on megapixel images.

A crucial rule…

NO NEW DATA will be added by ANY interpolation scheme!

The best you can do is minimize interpolation artifacts.

Before sampling, your **image function can be assumed to be some continuous surface**. To make the surface smooth, **we do require a continuous function and continuous derivatives** at each sample point. The surface can be assumed to be bandwidth limited, and the existing samples can be assumed to be dense enough.

Nearest Neighbor

With the Nearest Neighbor scheme, you just **grab the nearest pixel and use it**. One simple way to do this is to round your **x** value and add it to a rounded and doubled y value. This will give you four integers **0**, **1**, **2**, and **3** that can use **table lookup** or **case** commands to read one of the four corner pixels.

We might start with this sample data set array and use it to compare the different interpolation methods…

```
[ [0.60 0.60 0.48 0.24 0.60 0.60]
  [0.60 0.60 0.48 0.24 0.60 0.60]
  [0.00 0.00 0.36 0.12 0.48 0.48]
  [0.24 0.24 0.48 0.60 0.12 0.12]
  [0.12 0.12 0.24 0.48 0.36 0.36]
  [0.12 0.12 0.24 0.48 0.36 0.36] ]
```

Although this is a **6x6** data grid, we will only concern ourselves with the middle **4x4** array representing **3 x 3 = 9** unit squares. The reasons for this will become apparent when we get to the Bicubic Interpolation method.

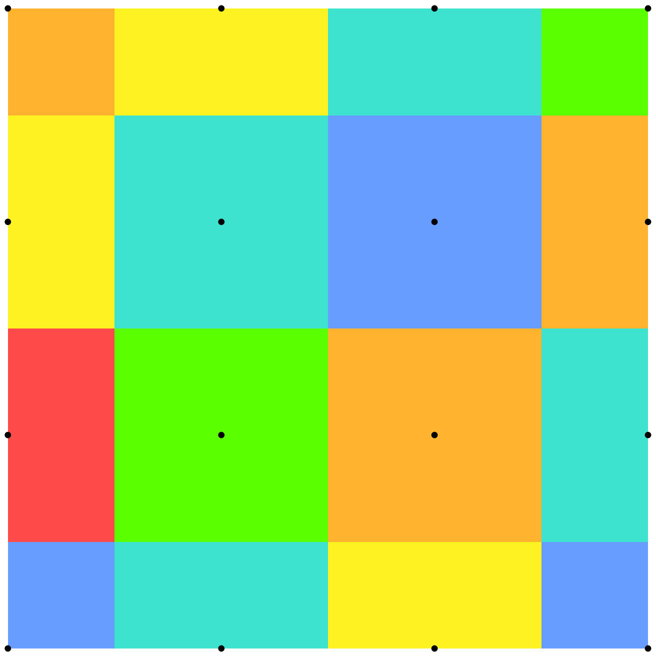
Here is some **PostScript** code...

```
dup floor cvi dup /yi exch store sub /yr exch store
dup floor cvi dup /xi exch store sub /xr exch store

xr round yr round 2 mul add cvi

   [ {data yi get xi get} {data yi get xi 1 add get}
     {data yi 1 add get xi get}{data yi 1 add get xi 1 add get}
   ] exch get exec
```

Our first lines convert the input position into box and position-in-box values. The middle line converts the position-in-box to an integer **0** to **3**. The final lines select the lower left, lower right, upper left, or upper right data point value. Here is what one of your bitmap planes will look like in false color...



As you can see, your results will mostly be in error and the artifacts may get downright ugly. Please note again that this is a **false color** plot of a **single** bitmap plane. In this case **red = 0** on up through **blue = 0.6**

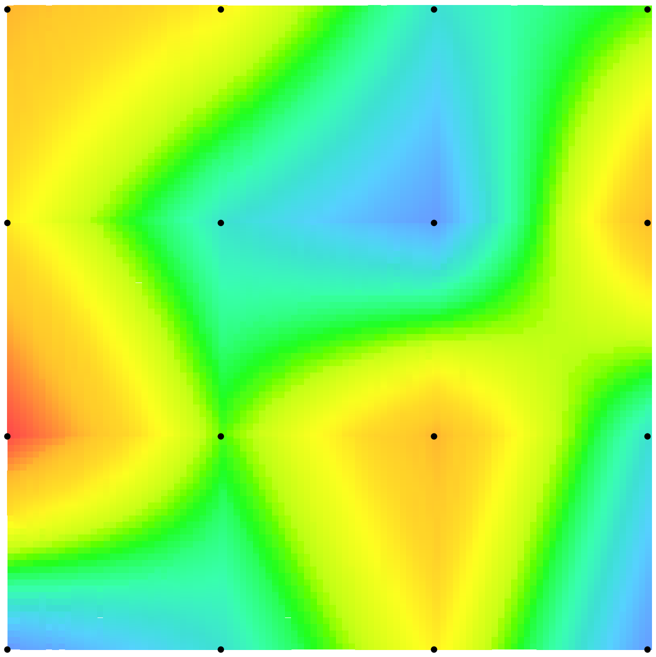
Nearest Neighbor is by far the fastest and by far the ugliest interpolation scheme. It can introduce major artifacts into your modified bitmap. As a relative speed indicator, on one unoptimized and older PostScript implementation, the time per pixel is just over **10 microseconds**. We can use this figure later for a relative baseline comparison of the other interpolation speeds.

Bilineal Interpolation

With **Bilineal Interpolation**, you proportion your result to the **relative distance** between your four original points. You could first do an **x** interpolation and then a **y**. Or vice versa. Curiously, either way should lead you to this approximate but highly useful final formula...

$$f(x,y) = f(0,0)(1-x)(1-y) + f(1,0)(x)(1-y) + \\ f(0,1)(1-x)(y) + f(1,1)(x)(y)$$

This measures the distance to each of the old corner points and provides a weighted average. The results are far better looking than nearest neighbor...



We see that there a very few artifacts remaining, but that **bilineal interpolation should be useful for all but the most critical of applications**.

Here is some bilineal interpolation code done in **PostScript**...

```
dup floor cvi dup /yi exch store sub /yr exch store
dup floor cvi dup /xi exch store sub /xr exch store

data yi get xi get 1 xr sub  1 yr sub  mul mul
data yi get xi 1 add get xr 1 yr sub mul mul  add
data yi 1 add get xi get 1 xr sub yr  mul mul add
data yi 1 add get xi 1 add get xr mul yr mul add
```

We once again split our input position into a pair of integer positions and two fractional residue values. Each original data point is then converted into a weighted average using the calculations shown.

As a relative speed indicator, again on one unoptimized and older PostScript implementation, the execution time per pixel is just over **14 microseconds**. Thus, there is only a forty percent penalty for vastly superior results. The surprisingly low speed difference is apparently caused by **PostScript** being quite fast and adept at on-stack adds and multiplies, but somewhat slower on its proc lookups and executions.

Bilineal via Table Lookup

The speed of calculating any pixel interpolation is highly dependent both on the system speed and the language used. In general, interpreted languages will be slowest and compiled languages faster. Hand crafted machine language will be faster still. Fastest of all, of course, would be dedicated hardware in the form of a FPGA or whatever.

It is often faster to look up a value in a table than to calculate it. Which suggests that some implementations of Bilineal Interpolation may be significantly improved by replacing repeated multiply and adds with a quantized table lookup of some predetermined constants.

However, this is apparently **not** the case with **PostScript** and table lookup interpolation may not be beneficial enough with your language choice. Your results may vary.

The 3x3 Compromise

If extreme speeds are essential, the limiting minimum case of a **3x3** bilineal table lookup might be useful. The results would end up pretty bad, but should still be significantly better than nearest neighbor. Only one lookup is needed **4/9ths** of the time, a two lookup average **4/9ths** of a time, and the full four lookup average a mere **1/9th** of the time.

Bicubic Interpolation

This is the Godzilla of pixel interpolation algorithms. It gives absolutely superb results with negligible artifacts. But is very hard to understand and requires an extreme number of complex calculations.

Bicubic Interpolation attempts to reconstruct the **exact surface** between your four initial pixels. It does this by extracting **sixteen** pieces of information. Based on the **values** of the samples, the **x slopes** of those values, the **y slopes** of those values, and the **xy slope cross products** of those values.

It turns out that any point on a two dimensional unity normalized surface can be represented by a set of **sixteen** cubic polynomial equations.

The key bicubic equations are…

```
p(x,y) = a00*x^0^y^0 + a01*x^0^y^1 +
         a02*x^0^y^2 + a03*x^0^y^3 +

         a10*x^1^y^0 + a11*x^1^y^1 +
         a12*x^1^y^2 + a13*x^1^y^3 +

         a20*x^2^y^0 + a21*x^2^y^1 +
         a22*x^3^y^2 + a23*x^2^y^3 +

         a30*x^3^y^0 + a31*x^3^y^1 +
         a32*x^3^y^2 + a33*x^3^y^3
```

This expression can be simplified somewhat by noting that **x^0 = y^0 = 1** and **x^1 = x** and **y^1= y**. Our non-trivial problem is to find the sixteen constant coefficients **a00** through **a33** for an initial four unit square data points and their nearest eight neighbors.

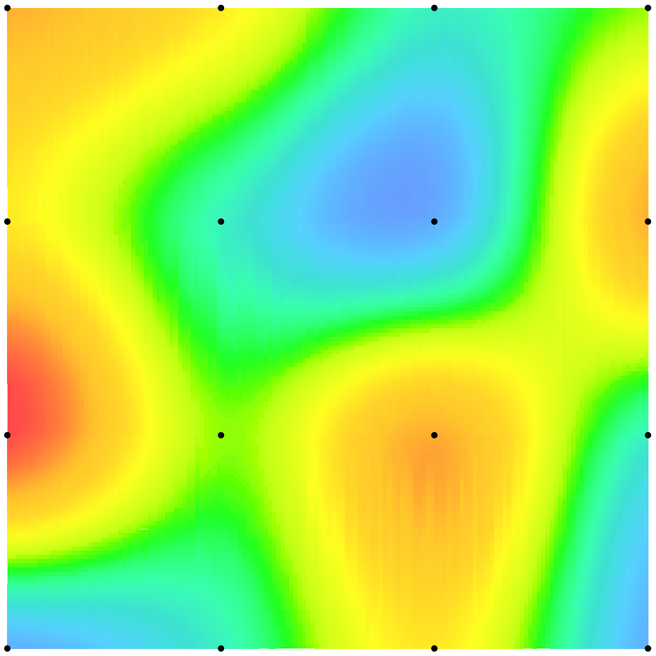
We can start by substituting some variables to simplify the notation. So, let **w0 = f(0,0)** or the pixel value at our lower left point. Let **w1 = (f0,1)** or the pixel value at our lower right point. Let **w2 = f(0,0)** or the pixel value at our upper left point. And let **w3 = (f1,1)** or the pixel value at our upper right point.

Similarly, we will let **x0** through **x3** represent the **xslope** (or partial derivative with respect to x) at each point. Normally, we do not know the exact x slope value. So, **we can approximate it by taking the average of the changes between your previous and next data point**.

We can also let **y0** through **y3** represent the **yslope** (or partial derivative with respect to y) at each point. Finally, to handle the behavior near the middle of our four data point set, we can let **z0** through **z3** represent the **slope product** or (partial derivative cross product) at each point.

We actually end up using **16 data points** for calculation, with the "extras" being needed to find our x and y slopes. This is not a problem **interior** to your pixel map. But you will have to do something at the edges to either ignore or fake the slopes. Which is why our sample data has "extra" values around its outside edge.

The results can be quite impressive...



We see an artifact free surface that we can pick any point off of. The surface is continuous and has continuous slopes at every point.

Bicubic calculations are often done using **matrix techniques**. Since these can be hard to understand, we will instead use ordinary algebra here. Helped along with a partial differential equation or two. All you will need to know about partial differentials here is that (A) A partial differential is the **slope** in **one** variable direction, and (B) The differential (or its limit derivative) of **f(x) = b0 + b1x + b2x^2 + b3x^3** is its **slope** of **b1 + 2b2x + 3b3x^2**.

Finding a new bicubic interpolated pixel value inside a unit square starts off by finding **w0** through **z3**. These are already known from our available pixel data, or can be easily calculated from it. You then use these **w0** through **z3** to evaluate expressions for the various **ax** coefficients.

Sadly, we now have **w0** through **z3** as a function of **a00** through **a33**. We instead need **a00** through **a33** as a function of **w0** through **z3**. So, we have to solve 16 linear equations in 16 unknowns to isolate the **axx** values.

Finally, from our newly calculated **a00** through **a33** values, we can find the value of our new pixel using our bicubic formula shown above.

Here is how **w0** through **z3** are related to the **a00** through **a33** coefficients...

```
w0 = f(0,0) = a00
w1 = f(1,0) = a00 + a10 + a20 + a30
w2 = f(0,1) = a00 + a01 + a02 + a03
w3 = f(1,1) = a00 + a10 + a20 + a30 +
              a01 + a11 + a21 + a31 +
              a02 + a12 + a22 + a32 +
              a03 + a13 + a23 + a33

x0 = fx(0,0) = a10
x1 = fx(1,0) = a10 + 2a20 + 3a30
x2 = fx(0,1) = a10 + a11 + a12 + a13
x3 = fx(1,1) = 1*(a10 + a11 + a12 + a13) +
               2*(a20 + a21 + a22 + a23) +
               3*(a30 + a31 + a32 + a33)

y0 = fy(0,0) = a01
y1 = fy(1,0) = a01 + a11 + a21 + a31
y2 = fy(0,1) = a01 +2Aa02 + 3a03
y3 = fy(1,1) = 1*(a01+a11+a21+a31) +
               2*(a02+a12+a22+a32) +
               3*(a03+a13+a23+a33)

z0 = fxy(0,0) = a11
z1 = fxy(1,0) = a11 + 2a21 + 3a31
z2 = fxy(0,1) = a11 + 2a12 + 3a13
z3 = fxy(1,1) = 1*a11 + 2*a12 + 3*a13 +
                2*a21 + 4*a22 + 6*a23 +
                3*a31 + 6*a32 + 9*a33
```

In general, these equations are found by substituting **1** and **0** values for **x** and **y** and their slope derivatives. Let's look at four typical examples...

- **w2:** Finds value at x=0, y=1. x=0 drops out all equations with powers of x=1, x=2, and x=3. Leaving a01*y + a02*y + a03*y. Which at y=1 becomes w2 = a01 + a02 + a03.

- **x1:** Finds xslope at x=1, y=0. y=0 drops out all equations with powers of y=1, y=2, and y=3. Leaving a10*x + a20*x^2 + a30*x^3. Whose partial derivative is a10 + 2a20*x + 3a30*x^2.

- **y3:** All equations are initially active. x=1 everywhere. y slopes will be 0 + a01y + ... + a02y + ... + 3a03y^2 + ... Which at y=1 will become the result shown.
- **z3:** Cross products of the slopes will be a11(1x*1y) + a12(1x*2y) + a13(1x*3y^2) + a21(2x^2*y) + a22(2x^2*2y^2) + ... Which at x=1 and y=1 becomes 1a11 + 2a12 + 3a13 + 2a21 + 4a22 + ...

Finding the Coefficients

We do now have **w0** through **z3** as functions of **a00** through **a33**. To solve for individual **axx** values, we have to solve **16** linear equations in **16** unknowns. This can be done manually, or by inserting the values into any of several math equation programs. The results should be…

```
a00 =  w0,
a01 =  y0,
a02 = -3w0 + 3w2 -2y0 - y2
a03 =  2w0 - 2w2 + y0 + y2
a10 =  x0
a11 =  z0
a12 = -3x0 + 3x2 - 2z0 - z2
a13 =  2x0 - 2x2 +  z0 + z2
a20 = -3w0 + 3w1 - 2x0 - x1
a21 = -3y0 + 3y1 - 2z0 - z1
a22 =  9w0 - 9w1 - 9w2 + 9w3 + 6x0 + 3x1 +
      -6x2 - 3x3 + 6y0 - 6y1 + 3y2 - 3y3 +
       4z0 + 2z1 + 2z2 + z3
a23 = -6w0 + 6w1 + 6w2 - 6w3 -4x0 - 2x1 +
       4x2 + 2x3 -3y0 + 3y1 - 3y2 + 3y3 +
      -2z0 -  z1 - 2z2 -  z3
a30 =  2w0 - 2w1 +  x0 +  x1
a31 =  2y0 - 2y1 +  z0 +  z1
a32 = -6w0 + 6w1 + 6w2 -6 w3 -3x0 - 3x1 +
       3x2 + 3x3 -4y0 + 4y1 - 2y2 + 2y3 +
      -2z0 - 2z1 -  z2 -  z3
a33 =  4w0 - 4w1 - 4w2 + 4w3 + 2x0 + 2x1 +
      -2x2 - 2x3 + 2y0 - 2y1 + 2y2 - 2y3 +
        z0 +  z1 +  z2 +  z3
```

It is tedious and highly error prone to try and do this by hand. There are some obvious simplifications. Four values are available by inspection. Eight of these can be solved with pairs of equations. Leaving four equations in four unknowns.

Ferinstance, if you want to do a manual solution, you can grab **a00**, **a01**, **a10**, and **a11** by inspection. **w1** and **x1** can then be simultaneously solved for **a20** and **a30**. Then **x2** and **z2** can be simultaneously solved for **a12** and **a13**. And **w2** and **y2** can be simultaneously solved for **a02** and **a03**. And **y1** and **z1** can in turn get simultaneously solved for **a21** and **a31**.

If you save them for last, the remaining variables can end up as four equations in four unknowns. With all previous calculated values being reducible down into four constants, one for each equation. Two of these equations can then relate **a22** and **a33**, while another two can relate **a32** and **a33**. And substituted back to solve for **a33**. The final three variables follow by simple arithmetic.

A Summary

An important warning: **Bicubic calculations might rarely end up slightly above unity or slightly below zero**. The rest of your code must be able to clip these values or otherwise deal with them.

Again on one unoptimized and older **PostScript** implementation, the execution time per pixel for a bicubic interpolation was just over **100 microseconds**. Thus bicubic speed in this instance is **seven times worse** than bilineal and is **ten times worse** than nearest neighbor.

I suspect that when further optimized on a newer Distiller and a faster machine, the results will end up something like **three seconds per megabyte per color plane** for bilineal and **twenty seconds per megabyte per color plane** for bicubic. Thus, bicubic can most certainly be used for typical **PostScript** bitmap image manipulations but will likely forever remain somewhat on the slow side.

Your results will vary with your language and implementation choices. Again, compiled languages will usually beat out interpreted ones. Hand crafted machine code will often be much faster still, and custom hardware may even be faster.

Reducing bilineal to somewhat coarser table lookups may or may not prove time effective or worthwhile on some implementations.

In some cases, doubling your sample rate can significantly reduce interpolation problems. But only with **4X** speed and **4x** storage penalties that may not end up being cost effective.

I originally thought a **2X** sampled **3x3** compromise interpolation solution would be useful for such things as **eBay images** . But it appears that a genuine bicubic interpolation can end up nearly speed competitive.

With far better results.

Some Improvements

Finding "real" derivative slopes at each data point could possibly further improve bicubic. This would tend to sharpen the transitions. But would take some sort of more **content specific** and wider area processing.

Once any method and language is selected, second or third passes can be made to tweak and improve your code's operating speed. Some general techniques for **PostScript** speedups appear **here**.

This slightly obtuse rework of our bilinear code is **34** percent faster…

```
dup cvi dup /yi exch store sub /yr exch store
dup cvi dup /xi exch store sub /xr exch store

data yi get dup xi get 1 xr sub  mul exch xi 1 add
get xr mul add 1 yr sub mul

data yi 1 add get dup xi get 1 xr sub mul exch xi
1 add get xr mul add yr mul add
```

For More Help

Complete bicubic **PostScript** code detailed examples can be extracted from the **sourcecode** for this **GuruGram**.

Working PostScript code for bilineal and bicubic has also been extracted to **this utility** and **this demo**.

Much of our analysis and review done here was based on the two **Wikipedia** entries found **here** and **here**. Additional insight into single dimensional cubic spline image interpolation appears **here**.

Similar tutorials and additional support materials are found on our **Cubic Spline**, our **PostScript** and our **GurGram** library pages. As always, **Custom Consulting** is available on a cash and carry or contract basis. As are seminars.

For details, you can email **don@tinaja.com**. Or call **(928) 428-4073**.