Classic studio view cameras had a number of adjustments on them that allowed image corrections. One of these was a centered vertical tilt on the film plane back. Used properly, this tilt could make the sides of buildings (and telephone poles in particular) perfectly vertical. Such an intentional distortion is often called an Architect’s Perspective.

Today, the same effect can look quite good on eBay product photos...

Note how the vertical edges in the image are in fact vertical. I’ve long had a set of perspective correction utilities available for your use.
Such as KEYCOR01.PSL which we reviewed in the Image Keystone Correction of GuruGram #55. Or our earlier SWINGT01.PSL which we reviewed in the Digital Camera Swings and Tilts back in GuruGram #15.

Sadly, these utilities had a flaw in that they would introduce curvature distortion for higher tilt values. This was caused by attempting to work one pixel line at a time when rewriting the bitmaps.

With our latest AOSUTIL1.PSL utilities first described in our Bitmap to PS Array Conversions of GuruGram #84 and enhanced upon here, it is now possible to do true image remapping in both X and Y directions at the same time. While greatly reducing any distortions for higher amounts of correction.

A new ARCHPER1.PSL utility set is now available that does these improved Architect’s Perspective corrections for you. It takes an existing .BMP image, makes suitable tilt and keystone modifications, and then resaves as a new and corrected .BMP image. You use this utility by reading it as an ordinary ASCII textfile, modifying some data values in a textfile or editor, and then sending it to Acrobat Distiller.

By Using Acrobat Distiller as a PostScript Computer.

While the utility is written in PostScript and optionally uses my Gonzo Utilities, no knowledge of PostScript programming is required for routine use.

One Gotcha:

Acrobat Distiller versions newer than 8.1 default to preventing diskfile access.

The workaround from Windows is to run "Acrodist -F" from the command line

Solutions for other systems are found here.

If you think of your image as similar to that on a view camera’s ground glass, the tilt correction geometry you will need turns out to be remarkably similar to...

The Starwars Nonlinear Transform

An intro tutorial on nonlinear graphics transforms appears as NONLINGR.PDF. In general, a linear transform lets you move, magnify, rotate, or even anamorphically stretch an image.

Anything fancier (such as converting a trapezoid to a rectangle) will demand more exotic nonlinear techniques. As will Architect’s Perspective.

One of the more common tutorial examples was this Starwars Transform…
Select a tilt angle $\theta$ with $0^\circ = $ flat and $90^\circ = $ vertical.

Predefine a tilt factor geometric constant $k$...

$$ k = \text{fullheight} \times \tan \theta $$

The nonlinear transform is then...

$$ x' = \frac{xk}{k + y} $$
$$ y' = \frac{yk}{k + y} $$

The tilt factor $k$ is the distance to the vanishing point. Both nonlinear transforms follow by way of similar triangles.

We’ll need two modifications of the Starwars transform for Architect’s Perspective. These involve moving into a centered space that the nonlinear transforms will work around.

First, we will add a $y_{cen}$ that vertically centers our tilting action. In deference to its view camera heritage and to prevent any "top vs bottom" numeric problems, we will usually keep $y_{cen}$ at one half the total image height.

A second $x_{cen}$ mod will also be added. $x_{cen}$ will be the axis of zero horizontal correction. It is used to apportion (or balance) how much fix is to be applied to the right and left subject edges. In general, $x_{cen}$ will not be in the middle. And could even be far left or far right if both subject edges lean the same way.

The translations between our image space and centered space are...

$$ \text{centeredx} = \text{imagex} - x_{cen} $$
$$ \text{centeredy} = \text{imagey} - y_{cen} $$
$$ \text{imagex} = \text{centeredx} + x_{cen} $$
$$ \text{imagey} = \text{centeredy} + y_{cen} $$

It is super important to linearly transform between the centered and image spaces, while nonlinearly transforming only about the $0,0$ axis of the centered image space. The Architect’s Perspective forward nonlinear transforms are...
For high processing speeds, we absolutely must minimize all the individual new pixel-by-pixel calculations. Fortunately, ycnew needs only done once per line, not every pixel.

And our xcold requires only a simply scaling at pixel calc time. You can see that ycnew/ycold equals 1 at y=0. You will definitely want to trap this out to prevent a possible div0 hassle.

zzz here is now the distance from the center to the vanishing point and will be ycen times the tangent of the tilt angle. Our vanishing point will be infinite at 90 degrees and zero when "flat" at 0 degrees.

Normal nonlinear transforms are a "goes to" sort of thing. When pixel remapping, we will instead need reverse or "comes from" nonlinear transforms. As detailed in INVEGRAF.PDF of GuruGram #85.

Here are the Architect's Perspective reverse nonlinear transforms...

\[
\begin{align*}
ycold &= ycnew \times \frac{zzz}{zzz - ycnew} \\
xcold &= xcnew \times \frac{ycold}{ycnew}
\end{align*}
\]

We thus have two variables that control our Architect's Perspective nonlinear transform. The zzz (derived from tiltangle and ycen) decides exactly how much correction to make, and xcen decides how to balance that correction between the left and right subject edges. We might use these two values to guess the amount of perspective correction needed for a given image. And then refine our guess with a second or third pass.

But it would seem intuitively better to find a way of...

**Improving Data Input**

If we know two specific points in an image where an exact amount of correction is required, we can solve the above forward nonlinear transform equations for zzz and xcen. Letting us get an exact solution on our first attempt.

The geometry might end up looking something like this...
Key points on your image are easily selectable using the rectangle tools and
readouts in Paint, Imageview32, or similar graphics programs. You first select a
ycorrheight, which is the point at which your tilt corrections are to be made. And
a xcorrwidth which will be the new subject width after all corrections are made.

Correction values are then measured as oldxleft, leftxshift and rightxshift. Our
"missing" values can be treated as dependent variables, noting that newxleft =
oldxleft + leftxshift. And that newxright = newxleft + xcorrwidth. And that our
oldxright = newxright + rightxshift.

From these values we can now calculate...

\[ \text{xcen} = \text{xcorrwidth} \times \left[ \frac{\text{leftxshift}}{(\text{leftxshift} - \text{rightxshift})} \right] + \text{oldxleft} + \text{leftxshift} \]

\[ \text{zzz} = \left[ \text{oldxleft} + \text{xcorrwidth} - \text{xcen} \right] \times \left[ \text{ycorrheight} - \text{ycen} \right] / \text{leftxshift} \]

The xcen calculation uses plain old similar triangles and \( y = mx + b \).

xcen and zzz are precalculated and only their results reused. Thus preventing
unneeded and time wasting repeated math.

We can optionally find our tilt angle...
tiltangle = atan(zzz/ycen)

If xcen and zzz are calculated from actual image data, **tiltangle** will not normally be needed for the nonlinear transforms. **tiltangle** ends up simply an optional visualization aide.

**Staying on the Same Page**

Certain reverse nonlinear transforms from xcnEW and ycnEW may try to reach out-of-range data that is above, below, left, or right of the original .BMP bitmap.

Error testing each individual new pixel one-on-one would likely be highly time prohibitive. Instead, **limits should be precalculated**. These precalculations need be done only once per project or once per line at a ridiculously lower total time penalty.

Here is one way to calculate limits to prevent off bitmap access attempts...

```
IF yfract = [ zzz /(zzz + ycen) ] > 1
    THEN ymin = floor {-ycen * yfract } + 1
    ELSE ymin = -ycen + 1

IF yfract = [ zzz /(zzz + ycen) ] < 1
    THEN ymax = floor { ycen / yfract } + 1
    ELSE ymax = ycen - 1

IF xfract = [ zzz/(zzz + ycnew)] < 1
    THEN xmin = int [floor {-xcen * xfract}]+ 1
    ELSE xmin = -xcen

IF xfract = [ zzz/(zzz + ycnew)] < 1
    THEN xmax = int [floor {(xwidth - xcen) * xfract}] - 1
    ELSE xmax = xwidth - xcen - 1
```

For stronger tilt corrections, some recentering or y axis scaling may be needed. These linear transformations are easiest provided in further post processing.

**A Working Utility**

You can explore Architect’s Perspective corrections using our ARCHPER1.PSL utility. What follows here is best understood by having ARCHPER1.PSL up in a separate textfile window.

The utility operates by capturing the original bitmap to three PostScript arrays of red, green, and blue strings.
As detailed in our BMP2PSA.PDF of GuruGram #84. A new three arrays of strings are separately created as redAOS1, greenAOS1, and blueAOS1. Reverse nonlinear transformations combined with bilinear interpolation then get done on a pixel by pixel basis to do the transformation. Finally, the new arrays of strings are converted into a final new and corrected bitmap.

The utility is in four parts, consisting of the basic array-of-strings manipulation procs, some globally exportable routines that may also be of use elsewhere, the actual Architect’s perspective code specifics, and a final example area.

Specifically, our main code loop in ARCHPER1.PSL is fixilt. This first captures our input bitmap to red, green, and blue arrays of strings as redAOS, greenAOS, and blueAOS. Data overflow limits are then calculated for later use. Each new pixel is then processed by way of a combination of a nonlinear reverse transform and a bilinear interpolation.

Effectively moving each pixel to its corrected position. Finally, the corrected arrays of strings are rewritten to a new output .BMP bitmap.

Going into more detail, the high level pixel processing is handled by procpixels. This mid level code sets up a loop within a loop that grabs the output pixel positions one line at a time and then one pixel at a time. Each pixel is then processed by procrgbpers.

procrgbpers in turn does the reverse nonlinear transforms and writes the repositioned old data to appropriate new pixels.

In general, the old x and y values needed for the current pixel location will be fractional. One of three bilinear interpolators named doredbilin, dogreenbilin, and dobluebilin are called. Each writes its interpolated old pixel result to the appropriate array of strings. Limits are precalculated to prevent out-of-bounds pixel errors.

**Speed Issues**

On a faster PC, ARCHPER1.PSL presently takes around five seconds to correct a 512x512 pixel bitmap. The processing time goes up roughly with the square of the resolution. A guideline...

> Perspective corrections are best done at DOUBLE the final image size.
> Otherwise, crop as tightly as possible and use the lowest possible resolution.

Speed has been partially optimized on ARCHPER1.PSL. Besides all of the usual PostScript Speedup Tricks, we might be able to make further improvements.
Further speedups should also be possible by optimizing the nonlinear transforms and the actual bilinear interpolations. But might make the code more obtuse and difficult to understand. And finally, being very careful when making correction measurements can eliminate the need for any second pass adjustments.

**For More Help**

The basic full two dimensional .BMP bitmap to PS Array of Strings tutorial appears as [BMP2PSA.PDF](#) with its actual PS utility at [PIXINTP1.PSL](#). A recent addition was [AIRBRUSH.PDF](#) whose capabilities coincidentally appear in our first image above. Additional .BMP manipulation enhancements and expansions are planned.

News about the latest updates and addons should first appear in [WHTNU08.ASP](#) or later blog entries.

Similar tutorials and additional support materials are found on our PostScript and our GuruGram library pages. As always, Custom Consulting is available on a cash and carry or contract basis. As are seminars and workshops. For details, you can email don@tinaja.com. Or call (928) 428-4073.