Some Inverse Graphics Transforms

Don Lancaster
Synergetics, Box 809, Thatcher, AZ 85552

http://www.tinaja.com
don@tinaja.com
(928) 428-4073

Certain math processes called **Graphics Transformations** are normally used to change the shape, size, and distortion of an image or drawing to meet some specific new need. These transformations can be **linear** to handle such tasks as translation, rotation, or scaling. Or **nonlinear** to carry out fancier remappings such as starwars, perspective, spherical, tunacan, or countless others.

A review of both linear and nonlinear graphic transformations **appears here**. The usual difference between the two is that the calculations are the **same** multiplies and adds for each and every pixel with a linear transformation, but are possibly **different** and fancier calcs for each and every nonlinearly transformed pixel.

Linear or nonlinear, these transformations are a "**goes to**" sort of thing. In which new data points are found as a result of existing pixels.

But there are also times and places when you may want to create a new image or drawing where you instead want a "**comes from**" pixel calculation. For instance, you might want to create a new image **x** pixels wide and **y** pixels high that has been calculated from some existing data source. While creating each pixel **once and only once**. All the while gracefully handling out-of-range data.

Skilled mathematicians often will perform an **inverse matrix** to deal with these "comes from" **inverse graphics transformations**.

In this **GuruGram**, we will instead look at some simpler math that sometimes can give us the same results in a more understandable way. We will see that some inverse transforms are trivially simple. Others are subtle, might be harder to calculate, or may introduce ambiguity or insolvability.

The Linear Inverse Graphics Transform

The linear graphics transform is normally used for translation, scaling, or rotation. It is also often the starting point for fancier nonlinear transformations.

One way to show the linear graphics transform is…

$$new(x) = ax + by + c = g$$
$$new(y) = dx + ey + f = h$$

To find the inverse graphics transform, we need to find the original and unknown **x** and **y** in terms of our known **new(x)** and **new(y)**. Using plain old algebra, we can either subtract the first equation from the second or else we can solve the first equation for **x** and substitute it into the second. Either route should give us…

$$x = [\ e(g-c) - b(h-f)]\ /\ (ae-bd)$$
$$y = [\ a(h-f) - d(g-c)]\ /\ (ae-bd)$$

Note that once you have **x**, you can use a faster and simpler **back substitution** to find **y**. Rather than making the second full calculation.

We might use a fancier method or two to check our work. Naturally, **it is super important to get the fundamental math right before you actually use it**.

An older **determinants** method can actually be simpler…

Starting with a system determinant of...

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} g \\ h \\ 1 \end{bmatrix}$$

x can be found as...

$$\begin{bmatrix} g & b & c \\ h & e & f \\ 1 & 0 & 1 \end{bmatrix} \text{ divided by } \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix}$$

and y can be found as...

$$\begin{bmatrix} a & g & c \\ d & h & f \\ 0 & 1 & 1 \end{bmatrix} \text{ divided by } \begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix}$$

Using the standard determinant expansion rules found in the above link should give you exactly the same linear inverse graphics transformation equations.

We could also do this using **Gauss Jordan Elimination**…

Starting with the Gauss Jordan matrix...

$$\begin{bmatrix} a & b & c & g \\ d & e & f & h \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Normalizations and subtractions can create...

$$\begin{bmatrix} 1 & \sim & \sim & \sim \\ 0 & 1 & \sim & \sim \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Which, after evaluating the intermediate "~" terms gives y by inspection. x follows by back substitution.

Translation, Scaling, and Rotation

Translation is somewhat trivial. **a** and **c** are **1**, **b** and **d** are **0**. **c** is our **xshift** and **f** is our **yshift**. Substituting in our inverse linear transformation gives…

Simple translation:

$$new(x) = x - xshift$$
$$new(y) = y - yshift$$

Scaling involves **a = xmag**, **c = ymag**, **b** and **d** = 0, **c = xshift**, and **f = yshift** leads to these possibilities…

Scaling only:

$$new(x) = x / xmag$$
$$new(y) = y / ymag$$

Scaling and translation:

$$new(x) = (x - xshift) / xmag$$
$$new(y) = (y - yshift) / ymag$$

Rotation through an angle **(θ)** needs some tricky trig with **a = cos(θ)**, **b=sin(θ)**, **d=sin(θ)**, and **e = -cos(θ)**. The denominator becomes $cos(\theta)^2 + sin(\theta)^2$ which by trig identity is unity.

Leaving us with…

Rotating about lower left:

$$x = -newx*\cos(\theta) - newy*\sin(\theta)$$
$$y = +newy*\cos(\theta) - newx*\sin(\theta)$$

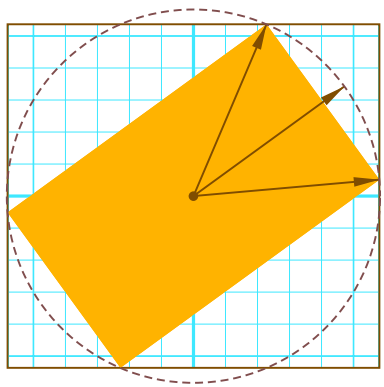
Rotating about the center:

$$x = -(newx-xoffset)\cos(\theta) - (newy-yoffset)\sin(\theta)$$
$$y = +(newy-yoffset)\cos(\theta) - (newx-xoffset)\sin(\theta)$$

Our **xshift** and **yshift** values should be **one half of their respective widths**. We could also get at these expressions by first noting that **$\cos(\theta) = \cos(-\theta)$**. And that **$\sin(\theta) = -\sin(-\theta)$** by trig identities.

Two Side Trips

Before we get into some nonlinear inverse graphics transformations, let's briefly look at two side issues. The first of these is that **any center rotated image has to get bigger if the corners are not to be clipped**…



For openers, note that **the rotated object will always fit inside a circle whose diameter equals the original diagonal**. This diameter will range from unity to 1.41, becoming a maximum for a square image rotated by 45 degrees.

It is useful to think of two vectors that will **lead** and will **lag** the rotation angle by **atan(height/width)**. These vectors will "point at" two corners of your image or bitmap.

The larger **absolute cosine** of lead or lag scaled by the diagonal will equal the new **width**. The larger **absolute sine** of lead or lag times the diagonal will equal the new **height**.

Like so…

To find the new size of a rotated image:

1. **Calculate the circle diameter as the square root of the sum of the squares of the height and the width.**
2. **Find the lead and lag angle as atan(width/height).**
3. **Find the new width as the LARGER ABSOLUTE COSINE of the rotation PLUS the lead angle and the rotation MINUS the lag angle. TIMES the diagonal.**
3. **Find the new height as the LARGER ABSOLUTE SINE of the rotation PLUS the lead angle and the rotation MINUS the lag angle. TIMES the diagonal.**

Here is some possible **PostScript** code…

```
PS utility finds the new size of a rotated image...
/findsize {
   /diag width dup mul height dup mul add sqrt store
   /laglead height width atan store
   /newwide rot laglead add cos abs rot laglead sub cos abs
    2 copy le{exch} if pop diag mul store
   /newhigh rot laglead add sin abs rot laglead sub sin abs
    2 copy le{exch} if pop diag mul store
         } store
```

These procs accept input variables of **width** and **height** and return new variables of **newidth** and **newheight**. You'll find details on using Acrobat Distiller as a PostScript Interpreter **here**, and much more on PostScript in general **here**.

Some background color (typically black or white) will have to be inserted into those pixels not covered by the rotated image. Similar but fancier math can be used if the rotation is to be anywhere else but centered.

Pixel Interpolation

A second side issue is that **images are normally quantized to integers**. Your **x** and **y** calculations are almost certain to demand fractional pixel values. Thus, some sort of **pixel interpolation** likely will be needed.

A pixel interpolation tutorial **appears here**. Three popular choices are the **nearest neighbor**, which is often too crude; **bilineal interpolation** which is often a usable choice; or **bicubic interpolation** which can give superb results, but is quite obtuse and computationally intensive.

Here is a partially optimized **PostScript** bilineal interpolation proc…

```
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
```

This assumes your data will be in the form of an array-of-arrays or (preferably) an **array-of-strings**. Typically, each internal array element will represent the **x** values of an image row, while the vertical position will represent the **y** values of that row in the total image. Per **this tutorial**.

As with a rotation, most any nonlinear transform may end up requesting some "offscreen" values for certain **x** and **y** pixels. Which must be substituted.

Going Nonlinear I — Starwars and Architect's Perspective

The "starwars" and "architects perspective" (or "view camera tilt") nonlinear graphics transformations are quite similar.

In the starwars case, you purposely want the top of your image to shrink down into infinity. Giving the illusion of a long text crawl. In the architects perspective case, the top of your building will be too narrow, and you want to **expand** your image to create perfectly vertical building edge lines.

Or, more typically, the top of the item you are photographing for **eBay** is "too large" because it is closer to your camera. And you will want to **contract** the top of the image. This time to create vertical product lines.

In general, finding the inverse of a nonlinear transformation works about the same way as a linear one. Instead of finding unknowns **new(x)** and **new(y)** as a function of known **x** and **y**, you instead find unknowns **(x)** and **(y)** as a function of knowns **new(x)** and **new(y)**. By using any of our previous techniques of simple algebra, determinants, or Gauss Jordan reduction.

Many nonlinear graphics transformations can be found in **this tutorial**.

Here are the starwars transforms and their inverses…

Centered starwars/architect transformation:

Let k = fullheight tan (-θ) where 0 degrees = flat and 90 degrees = vertical.

The forward nonlinear transforms will be

new(x) = xk / (k+y)
new(y) = yk / (k+y)

Whose inverse nonlinear transform solves as...

x = new(x) * (k / (k - new(y)))
y = new(y) * (k / (k - new(y)))

Off-center inverse transform shifts by...

x = new(x) * (k / (k - new(y))) - xshift
y = new(y) * (k / (k - new(y)))

Some of our earlier tilt correction routines introduced distortion for tilt angle values above fifteen degrees. The above algorithms should correct this defect.

Going Nonlinear II — Perspective and Scanner Pasteins

Our above transformations can be rotated by **90** degrees to create the equivalent of a view camera "swing". This can be used for linear to perspective conversion. It can also be used to paste in an originally flat scanned image. Which can provide superb lettering with an infinite depth of field. More details on this in our **Bitmap Typewriter** tutorial.

We can simply swap **x** for **y** in our previous inverse transform…

Centered perspective/scanner pastein inverse transformation:

x = new(x) * (k / (k - new(x)))
y = new(y) * (k / (k - new(x))) - yshift

For More Help

Our intent is to add additional nonlinear inverse graphics transforms here as the need arises. 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. For details, you can email **don@tinaja.com**. Or call **(928) 428-4073**.