tutorial

Maxime Gamboni

COLLABORATORS			
	TITLE : tutorial		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Maxime Gamboni	April 14, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

tutorial

1.1 picFX tutorial

Welcome to the picFX tutorial guide.

[It has been written on fourteenth April ’99 with version 0.86a of picFX]

1.2 1.1: introduction

* This guide is designed to help you using picFX and going further than the boring "ripples", "sphere", "and so on" deformations.

* I will assume that you already know the use of picFX. If not, first read some of the "standard" guide.

* There is no specific order for the chapters, you can jump over a chapter that does not interest you and go later back to it.

* This tutorial is still very small, but do not hesitate to write me both for asking how to do this or that thing on a picture, and to suggest me some effect you have discovered.

* So I will update this tutorial every time there is something to add.

* There are currently only three pictures in the ’pictures’ directory but this number should increase.

Folium is the picture for an implicit function. Each component has some different k constants. See Chapter 2.3 .

Pool: I started with a MagicWB pattern, then added some xˆ2+yˆ2 like function to have the circles, and then made a complicated deformation (I forgot what ;-)

Ripples: I took a friend of mine and appiled a simple translation mask to it. Easy but impressive... (He allowed me to put his picture on the 'net ;-)

1.3 1.2: Basic math functions

Here's a little list of the supported math function and some explanation on their utility for picture manipulation/creation.

a+b; a-b; a*b; a/b: I guess you know what they do :-)

aˆb: Raise a to power b. Use sqrt(a) instead of aˆ0.5 for square root.

sin(a); cos(a): Trig functions. Use them e.g. to do some waves (do sin(a)*127+128 to have the full 0-255 range) (try sin(x/5+cos(y/5))*12

Take care, trig functions require a floating point mode; a is a radient value.

tan(a); asin(a); acos(a); atan(a): Other trig functions, these are less used than the above.

sinh(a); cosh(a); tanh(a) are the hyperbolic functions. The inverses (a...) are available but do not work yet.

min(a,b): this outputs the smallest value between a and b. Having a constant will impose a minimum value to b.

max(a,b): outputs the biggest. Having one constant will impose a lower bound to the other.

Doing min(a,max(b,c)) with a and b constant will force c to remain between b and a (b<c<a)

a%b: a modulo b (remainder of the division of a by b). You will usually have b constant.

r(c,x',y');g(c,x',y');b(c,x',y'): Inter-referencing. The possibilities of these functions are infinite and exposed in chapter three.

c is a project number, it must be a constant. (x';y') is the coordinate that will be read.

1.4 1.3: What can't be done with picFX

picFX, like any software, isn't perfect, and there are things that simply can't be done. You will have to use other programs to do these things:

1) Write text, like in paint programs. If you want to put some text in your picture, you can either save the picture and reload it elsewhere, but is might be better (so that you will beneficiate of picFX features while doing it) to write the text with some other software on a blank page, then save it and finally reload it in picFX in a new project. You can then insert the text in your picture.

2) Use freehand drawing. Unless you find an implicit equation defining your drawing, you can't do that. (This is why I said you would need some math skills ;-)

3) Filling an area by its borders. Found in most paint programs, this feature is unavailable in picFX, as it can't be described using equations. The only thing you can do is try to find some equations desribing the surface, or, again, use some other software.

4) Draw lines, rectangles (full and especially empty), circles, and other things that seem so easy in other software are possible but rather complicated to do in picFX.

Please note that I do not intend to put this features in picFX (it would break the 'philosophy' of my program). But I will maybe make it easier to communicate with other programs, for instance using clipboard.

1.5 2.1: Simple spreads

Each colour on your computer screen is composed by a certain amount of red, green and blue. These amounts vary from zero (no colour, black) upto 255 (full colour).

For example, mixing red and green makes some yellow, and yellow's definition is (255,255,0): full red and green and no blue.

You have probably already seen those red,green and blue sliders in paint software.

Here is a small list of colours and their definitions:

(0,0,0) is black and (255,255,255) is white.

(255,0,0) is red, (0,255,0) is green and (0,0,255) is blue.

(0,255,255) is cyan, (255,0,255) is magenta and (255,255,0) is yellow.

Now open a new project window if none is opened yet (NEW button)

The three fields named r(0,x,y)=, g(0,x,y)= and b(0,x,y)= [zero may be another number, depending on the current project].

contains these definitions. So if you write:

r(0,x,y)=255

g(0,x,y)=255

b(0,x,y)=0

And click on "render", the current project window will fill with yellow. You may try to put other numbers.

Instead of just numbers, you may put functions of x and y. X is the number of pixels from the left of the window, beginning from zero, and y is the number of pixels from the top.

Try now setting:

r(0,x,y)=x

g(0,x,y)=y

b(0,x,y)=255

You will see a nice spread. In the top left of the window, x=y=0 and the colour components is (0,0,255), blue.

In the top right corner, x is 255, and the components will be (255,0,255), which is magenta.

In the bottom border, it goes from (0,255,255) (cyan) to (255,255,255) (white). Try clicking in some places of the project window and look what is written in the bottom part of the messages-window.

· The computer automatically applies a 256 modulo to avoid overflowing. Try the following to see it:

r(0,x,y)=2*x

g(0,x,y)=2*y

b(0,x,y)=255

Let's try some more complex operations:

The following are the default settings.

r(0,x,y)=x

g(0,x,y)=y

b(0,x,y)=(x+y)/2

Try clicking in some places of the project window after having rendered..

1.6 2.2: Monochrome graphs

When you have r=g=b (e.g. (153,153,153)) you will have a colour between black and white, i.e. a shade of grey.

In this part we will study monochrome graphs, [because it is easier to understand what happens].

I will only give you single functions. Write it as red component and set green and blue to copy the red one.

First example, a funny fractal-like picture:

If you do x*y/255, You will have a kind of diagonal spread, black in the top left corner and brighter in the bottom right one.

But because modulo 256 is used when it gets too big, you do not see anymore this spread but some other stange structure. Try it with x*y.

Try then to put multiply it by some constant, like three (x*y*3) or bigger ones (15,31,...). You will remark that when you are quite near from a power of two, the picture seems regular but is actually quite chaotic.

Second example, some circles to go on...

If you write x^2+y^2, you will get the square of the distance from the top left corner, which will quickly overcome 255. You will see some other (quite unexpected) circles appear.

These other circles are not actually "real", but are caused by the fact that the program does not calculate all possible places but only the one that correspond to whole values of x and y (I.e. it does for instance calculate the pixels (37;0) and (38;0) but nothing between, like (37.5;0).)

Replace now x by (128-x) and y by (128-y): (128-x)^2+(128-y)^2. You may wonder why I asked you to do that, since there is no visible change. Actually the only "real" circle is now at the center and the other are "fake".

If you divide by something, for example 4:((128-x)^2+(128-y)^2)/4, you will see what I mean. You can see some circles that seem to appear over the "real" ones. These are the "fakes" that seemed so "real" before...

Calculate now the square root of the previous expression: ((128-x)^2+(128-y)^2)^0.5 and switch to 32bits floating point (because of the square root).

Now the circles are gone and the colour of pixel is the distance to the center (Thanks to Pythagore ;-)

Let's have some fun with the trig functions. Keep the functions you had before, but put a sine on it: sin(((128-x)^2+(128-y)^2)^0.5). I guess you won't see much (Unless having eagle-eyes ;-). This is because the sine function varies between -1 and 1.

So try now to multiply it by 127 and add 128 (the [-1;1] range changes now to [1;255]). Now you can at last see the circles. But how tight they are, they cause a headache, don't they?

But it is possible to make the gap quite wider, dividing the contents of the sine function by a constant. Try the following:

sin(((128-x)^2+(128-y)^2)^0.5/5)*127+128. (Now the distance to the center varies slower than before and the rings are therefore more spaced.

To have some fun, try now adding y to the whole function (replacing 128):

sin(((128-x)^2+(128-y)^2/5)^0.5)*127+y.

This is like if the whole rings had been a little bent, and overcoming 255 (or getting under zero) in some places.

Third example, looking in the water.

We'll start with almost the same function than before, the rings, but a lot tighter:

sin(((128-x)^2+(128-y)^2)^0.5*2)*127+128

Save this somewhere (e.g. create a new project) and try the following before we go on:

sin(x/5)*sin(y/5)*127+128

Once you have seen it, let's go back to the previous one, adding sin(x/5)*sin(y/5)*5 to the contents of the sine:

sin(((128-x)^2+(128-y)^2)^0.5*2+sin(x/5)*sin(y/5)*5)*127+128

You can now have fun changing some constants. E.g. change the *2 (right after the square root) by *3 and change sin(x/5)*sin(y/5) by sin(x/5)*sin(y/6)..

Last example, some abstract art.

Let's go back to our concentric rings:

cos(((x-128)^2+(y-128)^2)^0.5/5)*127+128

Replace +128 by +x+y. You will see the effect you already know.

Now add some little waves:

(cos(((x-128)^2+(y-128)^2)^0.5/5)+sin(x/5)*sin(y/5))*127+x+y

This makes some funny effect

Multiplying these little waves by (y-100)/100 breaks the symetry and makes the picture more interesting:

(cos(((x-128)^2+(y-128)^2)^0.5/5)+sin(x/5)*sin(y/5)*(y-100)/100)*127+x+y

1.7 Implicit functions

You have maybe already heard about implicit functions.

An implicit function is an equation containing both x and y variable, and every solution (for x and y) of this equation is a part of the graph.

For instance, the equation of a simple circle would be: x^2+y^2=1. All (x;y) points satisfying this equation are on the graph of this function, which is a circle centered at (0;0) and of radius one.

Putting the whole equation in the left part of the equation (at the left of the "=") makes a function f(x,y)=0.

You can very easily turn an explicit (y=f(x)) equation into implicit form: f(x)-y=0.

picFX lets you "light" the graph of any implicit equation. Look at the steps below:

1) First adapt your equation so that it suits your project (remember that x goes from zero to the width of the project, usually 255; y does the same)

Once you have your equation f(x,y)=0, we will work on the left expression and forget about the "=0".

2) Since f(x,y) can be either negative or positive, raise it to square (unnecessary if you're sure that it will never happen). You will then have f(x,y)^2

[You can try rendering your function here. It is usually some chaothic stuff, but you will sometimes get some interesting results]

3) Now add a constant [We'll name it k]. Depending how "steep" your function is near its zeros, that constant may have to be quite big (upto one million sometimes).

You will anyway have to change this constant once it is finished, so that the result suits your needs.

Now we are sure that the function is always equal or bigger than this constant.

f(x,y)^2+k

4) Let's do now the inverse of this function (It will be very small, except near the places where the original function is near zero, where it will get near 1/k).

1/(f(x,y)^2+k)

5) To use the full [0-255] range, multiply by k*255

k*255/(f(x,y)^2+k)

Ok. Now you know how to do it, you may either try it at once (If you can't wait ;-) or do what I suggest step by step.

Let's try a some conics for a start. I suggest to copy the equations below just in the red field:

1) A simple equation would be:

(128-x)^2+((128-y)/2)^2=1000

Substracting one thousand makes:

(128-x)^2+((128-y)/2)^2-1000=0

2) Take care here: We know that x^2+y^2/4 is always positive. However now we substracted the squared radius (1000), it may go upto -1000, so we will have to raise to square:

((128-x)^2+((128-y)/2)^2-1000)^2

You can try rendering it now. It will look completely random trash. However if you divide it by e.g. 100, you will see the implicit graph in black.

3) Since we do not know yet how many give to k, let's try e.g. ten:

((128-x)^2+((128-y)/2)^2-1000)^2+10

4 and 5)

2550/(((128-x)^2+((128-y)/2)^2-1000)^2+10)

Render this and you will see that there are only some lonesome pixels. This means that we will have to make k quite bigger:

2550000/(((128-x)^2+((128-y)/2)^2-1000)^2+10000)

Now it is quite nicer.

Replacing the "+" by a "-" [between the two squared x and y] makes an hyperbola:

Write it in the green field and let the red like before.

And if you try the opposite (y^2-x^2 instead of x^2-y^2), you will get the other one:

[I removed here the "/2" to have a more regular picture]

r=2550000/(((128-x)^2+(128-y)^2-1000)^2+10000)

g=2550000/(((128-x)^2-(128-y)^2-1000)^2+10000)

b=2550000/(((128-y)^2-(128-x)^2-1000)^2+10000)

Second example, Descartes' folium:

1) The generic equation is: x^3+y^3-3*a*x*y=0, with a a real number bigger than zero.

As usual, to have our center at (128;128), we shall replace x by (x-128) and y by (y-128):

(x-128)^3+(y-128)^3-3*a*(x-128)*(y-128)

As it require some fine tuning, I give you directely my solution:

51000/((((x-128)^3+(y-128)^3)/1000-(x-128)*(y-128)/5)^2+200)

You can have fun putting different k values for each colour component:

r=510000/((((x-128)^3+(y-128)^3)/1000-(x-128)*(y-128)/5)^2+2000)

g=255000/((((x-128)^3+(y-128)^3)/1000-(x-128)*(y-128)/5)^2+1000)

b=25500/((((x-128)^3+(y-128)^3)/1000-(x-128)*(y-128)/5)^2+100)

The resulting picture is Pictures/Folium

Last example, Bernouilli's Lemniscate:

The generic equation is (x^2+y^2)^2=a^2*(x^2-y^2)

Once adapted to our needs (I have set a=100 and k=1000000 and then divided the f(x,y) expression by 1000 to keep small enough numbers), this equation gets:

255000000/((((128-x)^2+(128-y)^2)^2/1000-10*((128-x)^2-(128-y)^2))^2+1000000)

1.8 Having fun with colours

We will now start studying inter-referencing, i.e. accessing another picture to render a project.

In this section we will only modify the colours of the original pictures.

Open any picture (if possible some colourful one, no black and white) with the change size mode.

Then create a new project, with same size than the original picture.

Set the following equations, just to copy the original picture..

Here each pixel of project one will read the same pixel of project zero.

r(1,x,y)=r(0,x,y)

g(1,x,y)=g(0,x,y)

b(1,x,y)=b(0,x,y)

We will now create a negative version of project one. As each component range from 0 to 255, just do 255-x to have the negative version:

r(1,x,y)=255-r(0,x,y)

g(1,x,y)=255-g(0,x,y)

b(1,x,y)=255-b(0,x,y)

Now, black and white. To have the intensity of a colour, make the mean value of all components (r+g+b)/3 and apply it to all components of the resulting picture to get black 'n' white result:

r(1,x,y)=(r(0,x,y)+g(0,x,y)+b(0,x,y))/3

g(1,x,y)=(r(0,x,y)+g(0,x,y)+b(0,x,y))/3

b(1,x,y)=(r(0,x,y)+g(0,x,y)+b(0,x,y))/3

Now, what about dividing the colour by its intensity? This sometimes results in strange effects, but it sometimes may look nice... Create a new project so that picFX will not have to recalculate the intensity (we will just reuse the result of previous rendering).

When you divide a colour component by its intensity (and multiply by 85 to have something to see ;-), the whole picture will have the same intensity (except black areas).:

r(2,x,y)=85*r(0,x,y)/r(1,x,y)

g(2,x,y)=85*g(0,x,y)/r(1,x,y)

b(2,x,y)=85*b(0,x,y)/r(1,x,y)

* I have always read the red component of project one because it is supposed to be black and white, but it doesn't matter, actually.

* If you want to avoid the ugly gray you obtain were it was black (zero divided by zero outputs some random value), replace r(1,x,y) by max(1,r(1,x,y)). (or r(1,x,y)+1).

* You may wonder why I chose 85 as factor? this is because the a component may be upto three times bigger than the mean intensity, this is why I multiplyed by 255/3. Raise it a little if you do not have any plain (x,0,0) colour.

Colour Extraction

Suppose you have a picture where you would like to apply an effect only at some precise places, being e.g. blue.

You would have to use what I call a mask, i.e. a picture of intensity 255 or as high as possiblewhere we want the effect to act, and lower elsewhere.

We will first try to get one precise colour, i.e. only that exact colour will be in the mask and everything else will be out. This is quite easy.

You can for example take a magicWB backdrop (I took MarbleSpecky, for instance). I did not include it in the distribution for copyright reasons.

I suggest to first look for colours that are higher than the one we want and then the ones that are lower. We will then add these two masks.

If you write min(constant,colourcomponent), all colours smaller than the constant will be "highered" to this value, and when the result is different from constant, the original value was bigger.

And max(cte+1,min(cte,rgb)) will be cte+1 if rgb is bigger than cte, and cte if it isn't. Substract then cte: max(cte+1,min(cte,rgb))-cte and you will have either zero or one.

Doing the same for an upper bound, just do it the other way round: min(cte-1,max(cte,rgb))-cte.

If we add these two values, we will get:

if rgb < cte, 0 +(-1)=-1 (which will get 255 when rendered!)

if rgb = cte, 0 + 0 = 0

if rgb > cte, 1 + 0 = 1.

We can then do this for each colour component, and the mask should be where the resulting colour is black (i.e. 0,0,0):

r(1,x,y)=max(r+1,min(r,r(0,x,y)))+min(r-1,max(r,r(0,x,y)))-2*r

g(1,x,y)=max(g+1,min(g,g(0,x,y)))+min(g-1,max(g,g(0,x,y)))-2*g

b(1,x,y)=max(b+1,min(b,b(0,x,y)))+min(b-1,max(b,b(0,x,y)))-2*b

Of course you have to replace r, g and b by the components you want (click in the window of project zero to read the colour under the pixel!)

Then create a new project (number two), which will contain the actual mask:

r(2,x,y)=min(1,r(1,x,y)+g(1,x,y)+b(1,x,y)))

* multiply by 255 if you want to see what your mask looks like, but do not forget to divide again when you want to combine them!

* only one component is required, so you can either let the classic "x", "y" in the two others or copy the whole expression in all three fields (but it will be thrice slower!)

You will see later how to combine project, but if you're eager to know it, try the following:

r(1,x,y)=r(3,x,y)*r(2,x,y)+r(0,x,y)*(1-r(2,x,y))

g(1,x,y)=g(3,x,y)*r(2,x,y)+g(0,x,y)*(1-r(2,x,y))

b(1,x,y)=b(3,x,y)*r(2,x,y)+b(0,x,y)*(1-r(2,x,y))

project zero is still our source picture;

project one was used to create the mask. Since it is not used anymore, we can put the resulting picture there (please recycle ;-)

project two is the mask (I assumed it was only zero/one here, i.e. that you didn't multiply it by 255. If you did, just divide the whole expression by 255 (or whatever you put)).

project three is the picture you want to put where the (r,g,b) colour of project one was.

Now some hints if you want to do some more complicated stuff, extracting colour having some precise features (e.g. having red component between 100 and 150 and so on).

Note that masks are not required to be full/none, there can be intermediary values in this mask (i.e. this mask works a little like an alpha channel).

* You have already seen how to get a single colour, you can use it to take colours between wider bounds:

c(1,x,y)=max(l+1,min(l,c(0,x,y)))+min(h-1,max(h,c(0,x,y)))-l-h

c is the component you are working on (r,g or b) and l,h are the bounds between which (bounds included) the colour must be.

The disadvantage of this way is that the mask is full and then suddenly zero when we get though a bound. It would be better to have a slighter slope.

* You can either use the way I showed you with implicit equations, an equation of the form k/((x-c)^2+l) (k is a factor, x is the variable, here something like r(0,x,y); c is the value where we want the mask to be the higher and l lets you adjust the slope.

(The maximum value comes when x gets to c and is k/l)

* You can also use something like max(0,l-k*(x-c)^2): here l is the maximum value you want it to have (usually 255!), k is a factor, the bigger it is the quicker you will get to zero. X and c are as before.

This one has the advantage to have really no mask when you get far enough from c, unlike the previous.

You can work with each components separately, like I did before, but you can also combine them (e.g. to get the intensity).

If all this is not enough to let you make the mask you want, you can try to use the x and y vars. (e.g. if the mask you want is only at a given distance from a point, make a second mask that contains just this circle and multiply both to get what you want.

You can also use another paint package to draw your mask, if really necessary.

1.9 Basic transformations

We will in this section work on two projects: one source-image and a destination project. I will reference them as projects zero and one.

All three components will always have the same formulas (just replacing r by g and then b ;-), so I will only tell you the red one.

Choose a colourful and varied picture for the first project, so that you can easily see where all parts have gone, and make the second picture the same size as the first one.

* Let's begin with the classic Blur effect. Although it may seem simple, it requires unfortunately quite long formulas with picFX.

We will first calculate the mean values of the same pixel and the ones on the right and below:

r(1,x,y)=(r(0,x,y)+r(0,x+1,y)+r(0,x,y+1))/3

If you want some better effect, you can also add the lower-right pixel: r(0,x+1,y+1).

And if you want to give some more weight to a pixel, multiply it by two or even three. Do not forget to adjust the divider at the end (which must be the sum of all factors).

(See below)

You could for example give some horizontal blur (to give a movement effect):

r(1,x,y)=(3*r(0,x,y)+2*r(0,x+1,y)+r(0,x,y))/6

* Now to the (also classic) relief effect. Note that it may be better to first turn the picture to black and white before doing it. Just try and see.

There isn't much things to do here, it is almost always the same formula.

The idea is to calculate the difference between the current pixel and the ones that are near it (I add 128 to have grey when we are at the middle, instead of black):

r(1,x,y)=r(0,x,y)-r(0,x+1,y)+128 is horizontal relief.

r(1,x,y)=r(0,x,y)-r(0,x,y+1)+128 is vertical. Simply add these two to have bi-dimensional relief:

r(1,x,y)=2*r(0,x,y)-r(0,x,y+1)-r(0,x+1,y)+128

I let you play with these values (e.g. try multiplying the expression (before +128) by a constant to increase/decrease the effect.

Just take care to set a floating point mode when doing this, and avoid overcoming 256 or going under zero, it wouldn't look nice...

(even if you can use the min()/max() functions, but in these case the relief will be somehow "wrong".

The three above transformations are actually three convolution effects, once with the matrix

3 2 1 (dividing by 6)

the second:

1 1

1 1 (÷4), which produces that blur effect

and the last has matrix

2 -1

-1 0 (I added 128, because the sum of coefficients here is zero)

* Linear transformations.

These are of the form r(1,x,y)=r(0,ax+by+c,a'x+b'y+c'), with a,b,c,a',b',c' constants.

Some simple examples:

r(0,y,x) is a symetry through the diagonal (y=x) line.

r(0,x/2,y/2) doubles the size of the image (you will only see the top left quarter)

r(0,x+y/2,y-x/2) rotates (and reduces) the picture a little, 'round the top-left corner.

I can't really give you some general formulas for these, I will let you experiment yourself. Just note that you may replace x by (x-t) or y by (y-u), or both, to move the "center" of the transformation.

And: r(1,x,y)=r(0,ax+by+c,ay-bx+d) does always a regular rotation (no other linear distortion, like (x,y+x/2) would do).

* Non-linear transformation. (These are all transformation that do not match a linear transformation ;-)

They are of the form r(1,x,y)=r(0,f(x,y),g(x,y)), where you put what you want as f and g.

Note that the blur and relief are not non-linear transformation, because they access the inter-referencing functions several times.

Here it is even worse than the linear transformation. Although I can give you some tricks, there are no general formula to make you get a precise thing. You have to guess :-).

Compute the distance to a specific point (X;Y) (remember that you can read the coordinates of a given point clicking on a project window!)

1.10 4.1: masks to select an area

We have always applied the effects to the whole bitmap. It will sometimes be useful to use an intermediary project to "filter" this effect:

Example: You have two projects 0 and 1, a mask in project 2 and want to draw the result in project 3:

0:AA 1:BB 2:01 will result in 3:AB

This works a bit like an alpha channel, project two holding that alpha value (of course projects one and two may be two completely different loaded pictures).

Let's say that a value of zero in project 2 results in outputing the value of project 0 and a value of 255 outputs project 1.

To read from zero, you put r(0,x,y)*(255-r(2,x,y))/255, and reading from one is

r(1,x,y)*r(2,x,y)/255

Note that if you only use 0 or 255 (i.e. none or full alpha), you may make your mask have only values zero and one, so that you aren't required to divide by 255 everytime.

Note well that r(r(2,x,y),x,y) is not allowed, as the project number must be a constant value! (Because of freezing problems)

Well, now add the two values, to obtain: (r(0,x,y)*(255-r(2,x,y))+r(1,x,y)*r(2,x,y))/255

In all components, you obtain:

r(3,x,y)=(r(0,x,y)*(255-r(2,x,y))+r(1,x,y)*r(2,x,y))/255

g(3,x,y)=(g(0,x,y)*(255-r(2,x,y))+g(1,x,y)*r(2,x,y))/255

b(3,x,y)=(b(0,x,y)*(255-r(2,x,y))+b(1,x,y)*r(2,x,y))/255.

You may also read the different components from the mask, if you want, instead of having one mask for the whole picture.

To obtain these masks, you can use colour extraction (chapter 3.1)

Some mathematic formula (e.g. implicit graphes, chapter 2.3 , but anything else is ok :-))

You can combine these two ways. If for example you did some colour extraction and got some superfluous areas, you can try to find a function that covers everything but these areas and multiply both.

If you want to add masks (a bit like a logic OR), use the max() function between them. Another way to do it is to multiply their inverses, something like r(2,x,y)=255-(255-r(0,x,y))*(255-r(1,x,y)

And to apply a mask to a transformation, you can 'multiply the changes' by c(k,x,y)/255, replace c and k accordingly.

This is mostly useful for instance with blur and other convolution effects, and mayve with colour effects.

1.11 4.2: translation masks

This method consists in having a "translation project", which contains how much a pixel has to move in the source project. (Actually it works the other way round, it indicates how far you have to go to fetch the pixel data).

You can then do effects that would either be very difficult or awfully slow to do directely.

In all these examples, project zero is the source project, one is the mask and two is the destination.

If you want a linear translation, only one component of the mask will be used, and multiplied by different constanty horizontally and vertically:

r(2,x,y)=r(0,x+u*r(1,x,y)/255,y+v*r(1,x,y)/255)

g(2,x,y)=g(0,x+u*r(1,x,y)/255,y+v*r(1,x,y)/255)

b(2,x,y)=b(0,x+u*r(1,x,y)/255,y+v*r(1,x,y)/255)

The translation will always be done in the direction of vector (u,v). (Of course, I let you replace u/255 by a single value; you may also add a constant to x and y)

Two dimensional translation is easy, just read two different components of the mask (here, r and g):

r(2,x,y)=r(0,x+k*r(1,x,y),y+k*g(1,x,y))

g(2,x,y)=g(0,x+k*r(1,x,y),y+k*g(1,x,y))

b(2,x,y)=b(0,x+k*r(1,x,y),y+k*g(1,x,y))

It is possible to load pictures as a mask, pictures you might have done with other paint or picture manipulation programs.