*Le radial blur*

Bon, alors, avant de rentrer dans l'explication du radial blur, on va déja s'échauffer avec le blur vertical qui utilise le même principe de base.

Et oui, avant de se lancer il faut bien se dire que si l'on veut un bon rendu avec une bonne rapidité, il faut oublier les multizoom (zoom du zoom du zoom...) et les moyennages le long d'un vecteur (comme fait photoshop avec son radial blur) qui fait apparaitre 5 images transparentes si l'on moyenne sur 5 points le long d'un vecteur donc, ces trucs là, poubelle, beurk.

Le truc, c'est qu'il faut faire un "larsen" dans l'image, ce qu'on obtient en ecrivant dans l'image qui sert de source. Et comme tout larsen, le calcul des coefficients et la résolution (8/16/24 bits..) sont très importants. Déja, pour pas se prendre la tête, on travaille en  8 bits par composantes.   
 

|  |
| --- |
| ***Le blur vertical*** |

La, c'est facile, on peut ecrire l'effet comme suit :

for(y = 0;y < tx;y++)   
 for( x = 0 ; x < tx ; x++)   
  image[x][y] = a\*t[x][y] + b\*image[x][y-1]

<1> a+b = environ 1   
<2> b assez petit

ca revient, pour tout pixel, à prendre un peu de la couleur du pixel du dessus et de l'ajouter à son ancienne couleur. <2> permet de doser l'étalage des couleurs, et <1> permet de ne pas saturer! (en bref : a\*255+b\*255 = 255 donc, ca part pas en couille comme une merde:)

Pour vous éviter la galère pour trouver b de facon à régler le taux de blur de manière à peu près linèaire voila ce que j'utilise:

avec cof compris entre 0 et 1 :

cof = 1-cof\*.5;   
cof\*=cof;   
cof\*=cof;   
cof\*=cof;   
a=cof\*65535;        // précision 16bit   
b=(1-cof)\*65535;

pour les fénéants, voila le bout de code qui blurise une image rvb32 :   
(notez le petit décalage des coefs de 16 bit vers la droite pour éviter trop de manip) (oui oui, vous pouvez faire ca en 2 instructions en mmx, alors pas la peine de m'incendier:)

void BlurY(unsigned long\*rgb,float cof)   
{   
 int x,y;   
 unsigned long r,g,b;   
 unsigned long s,d;   
 unsigned long co1,co2;

 if(cof<0)   
  cof=0;   
 if(cof>1)   
  cof=1;   
 cof = 1-cof\*.5;

 cof\*=cof;   
 cof\*=cof;   
 cof\*=cof;

 co2=cof\*65535;   
 co1=(1-cof)\*65535;

 for(x=0;x<512;x++)   
  rgb[x]=0;

 for(y=1;y<256;y++)   
 {   
  for(x=0;x<512;x++)   
  {   
   s=rgb[x+(y<<9)-512];   
   d=rgb[x+(y<<9)];

   b=( co1\* (s&0xff)   + co2\*(d&0xff)   )>>16;   
   if (b>255) b=255;   
   g=( co1\* ((s&0xff00)>>8)  + co2\*((d&0xff00)>>8)  )>>16;   
   if (g>0xff) g=0xff;   
   r=( co1\* ((s&0xff0000)>>16)  + co2\*((d&0xff0000)>>16) );   
   if (r>0xff0000) r=0xff0000;

   rgb[x+(y<<9)]=(r&0xff0000)+(g<<8)+b;   
  }   
 }   
}

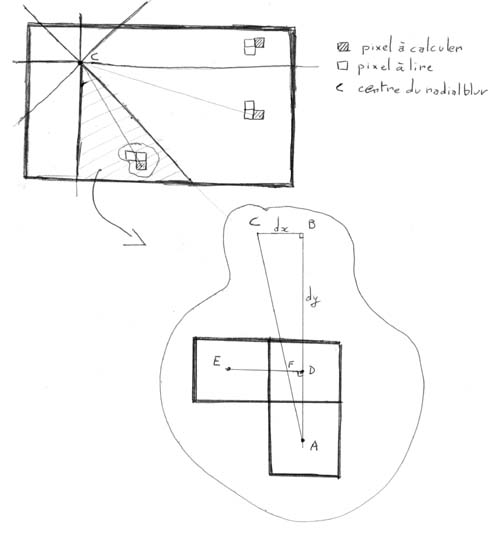
Pour ceux qui ont vu MyWorld, les transition de la demo, c'est le même principe, mais je fais varier ces coefs en fonction de x et du temps... (3 h de réglage! sans rigoler!)

Bon, passons aux choses plus sérieuses...

|  |
| --- |
| ***Le radial blur*** |

C'est un peu pareil que le blur vertical, on travaille avec les pixels voisins et, en fonction du vecteur centre->pixel, on calcules les coefficients.   
Les coefficients sont proportionnels à la coupe de ce rayon avec les pixel voisin.   
Le plus facile c'est de travailler par octant.

Etudions le cas de l'octant en bas droite ci-dessous :



Physiquement, plus le rayon CA passe près de D, plus la couleur de D sera prise en compte et le rayon CA passe près de E, plus la couleur de E sera prise en compte. Et si le rayon passe pile poil par D, et bien, on revient au cas du blur verticale.

On pourrait écrire le mix sous cette forme :

A = a \* A + b \* ( c \* D + (1-c) \* E )

alors, là, tout de suite, je vous vois venir : comment calculer c rapidement pour tous les pixels de l'écran.

Surtout, oubliez les cosinus, Thalès viens nous aider :

c = c/1 = CB/BA = dx / dy

Ce qui nous fait seulement 1 division et 2 accès à l'image par pixel.   
On pourrait tabuler la division et faire translater (virtuellement) la table qd on bouge le centre du radial, mais bof, 1 lecture mémoire vaut vraiment mieux qu'une division à l'heure actuelle? (à vous de tester) (moi, étant un gros fénéant, j'ai laissé la division)

Je vous laisse faire tous les octants un par un, le dessin suffit je pense.

***Les cas limites!***

Hé oui, pour éviter les traits entre les octants (beurk), faut se taper les cas limites avant de démarer le replissage des octants. Les cas limites, ce sont les lignes qui séparent les octants (cas simples).

***The TIP of the day***

alors voila, vous venez de coder votre radial blur, mais beurk, c'est tout flou!!! La, dépité, vous vous dites je suis pas assez précis, je passe en floatant, et ca marche c beau, je vois des beaux rayons.

Alors là, vous avez tout faux :

-> en float ou double, vous etes moins précis qu'en entier   
-> Les rayons viens du fait qu'on est moins précis

Et oui, après un long moment d'interrogation, voila mon explication :

Pour voir des LONGS rayons, il faut augmenter le contraste entre les rayons et pour augmenter le contraste, il faut "seuiller" d'une certaine manière la propagation des couleurs. Il faut stratifier la propagation.

En clair, en écrivant ce qui suit, on obtient des beaux rayons :   
(code qui remplir l'octant étudié)

for(yy=centrey+1;yy<256;yy++)   
{   
  finx=min(512,centrex+h);   
  for(xx=centrex+1;xx<finx;xx++) //octant 0   
  {   
    p=source+xx+(yy<<9);   
    tdest=\*p;   
    r=col\*(tdest>>16);    // col = define   
    g=col\*((tdest>>8)&0xff);   
    b=col\*(tdest&0xff) ;

    tol=((xx-centrex)<<16)/(yy-centrey);  // vx/vy   
    tdest=\*(p-513);     // source[xx-513+yy\*512];   
    r+=(tol\*(tdest>>16))&0xffff0000;   
    g+=(tol\*((tdest>>8)&0xff))&0xffff0000;   
    b+=(tol\*(tdest&0xff))&0xffff0000 ;   
    tdest=\*(p-512);     // source[xx-1+yy\*512];

    tol=65536-tol;   
    r+=(tol\*(tdest>>16))&0xffff0000 ;   
    g+=(tol\*((tdest>>8)&0xff))&0xffff0000;   
    b+=(tol\*(tdest&0xff))&0xffff0000;

    if(r>0xff0000) r=0xff0000;   
     g=g>>8;   
    if(g>0xff00) g=0xff00;   
    b=b>>16;   
    if(b>255) b=255;   
    \*p=(r&0xff0000)+(g&0xff00)+b;   
  }

(ici col correspond à a et tol à c)

les  "& 0xffff0000" permet de limiter la propagation aux fortes incrémentations de couleur pendant le calcul ce qui augmente le contrate et évite le flou.

Voila, j'espère que j'ai été assez clair, n'hésitez pas à envoyer vos remarques!

Deemphasis