Gernot Hoffmann

Interpolations for Image Warping

Contents

1. Introduction

This document shows some algorithms and the results for different interpolation strategies for scaling, rotation, morphing and perspective rectification.
Perspective transformations are explained in [2],[3].

Bilinear Interpolation is applied by the well known method, but additionally we use an offset of 0.5 pixels for the access to the source image.
This results in a balanced blur effect for no rotation and no scaling and for arbitrary rotations and moderate scaling as well. It works for downsampling as well. Especially we can downsample screenshots by factor 0.5.

Biquadratic and Bicubic Interpolation are straightforward applications of parabolas, but both create weak halos (same effect with Photoshop Bicubic).

The interpolation algorithms are based on simple mathematics without special references.

Bicubic B-Spline Interpolation is based on Paul Bourke´s document [1].

In fact this is not an interpolation. Because B-Splines are used, it is more a sort of averaging - will be always blurry.
But the result is very convincing if a weak sharpening filter is applied afterwards.

The tests on the next page were made without and with Gamma correction.

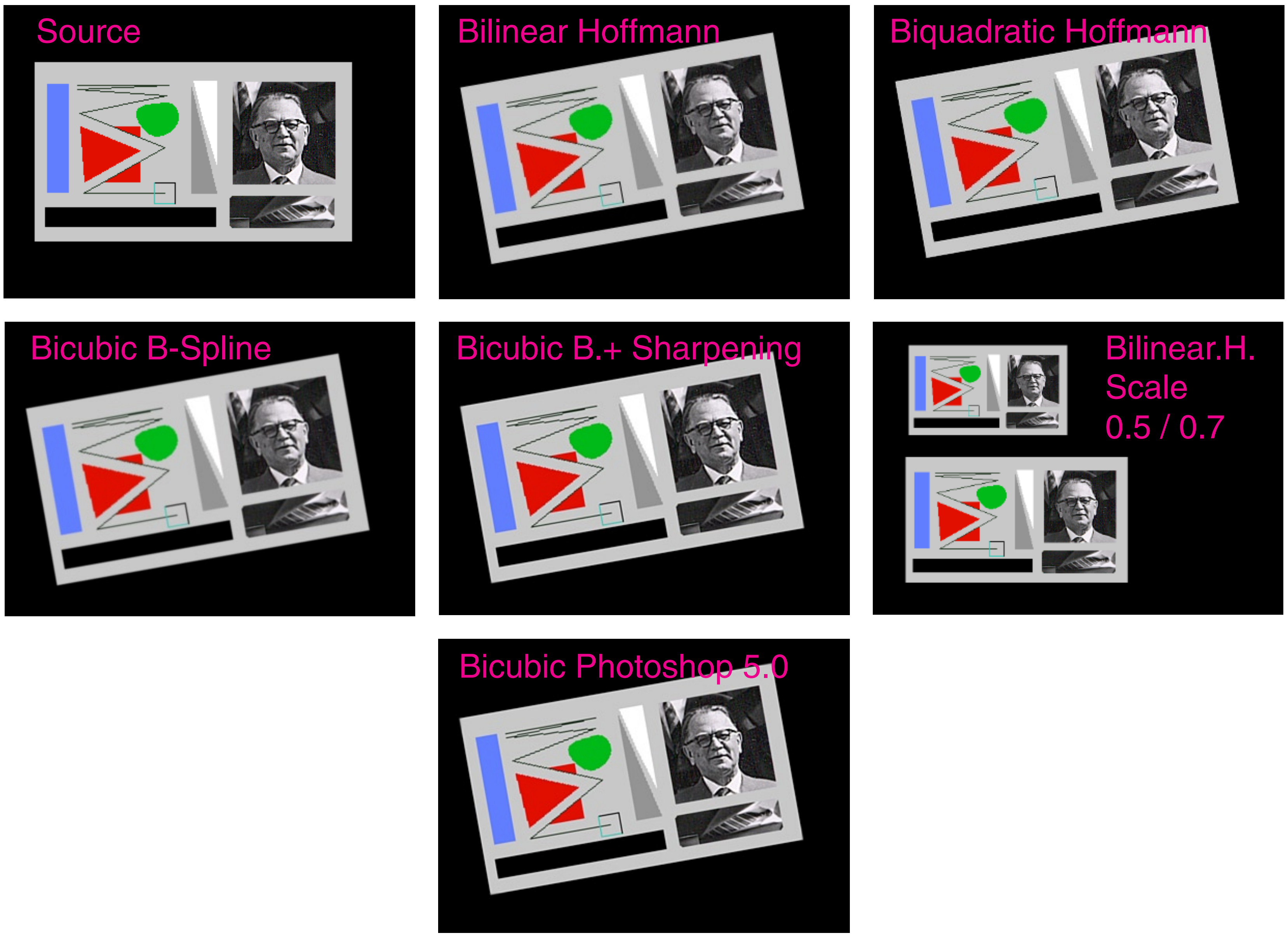
The last pages show algorithms for transformations, including different interpolations.

This document is a pixel synchronized PDF. Download by right mouse click.
View directly by Acrobat. Browsers are sometimes not accurate.

Please use resolution 72 dpi in Acrobat.
Images are pixel synchronized for zoom 100% or 200% for 72 dpi.

2. Examples 1 / Zoom 200%

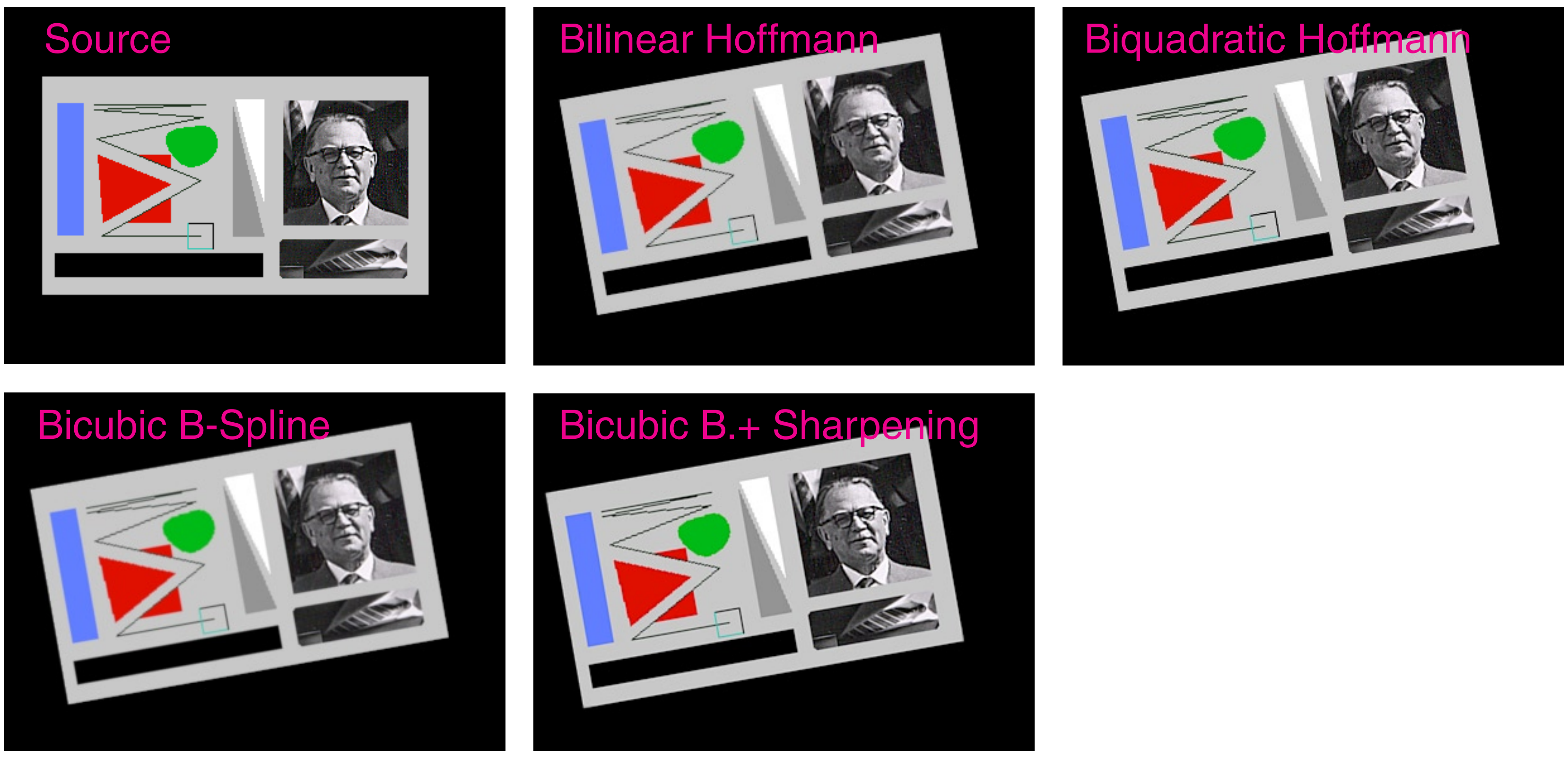
Operations without Gamma Correction



Operations with Gamma Correction

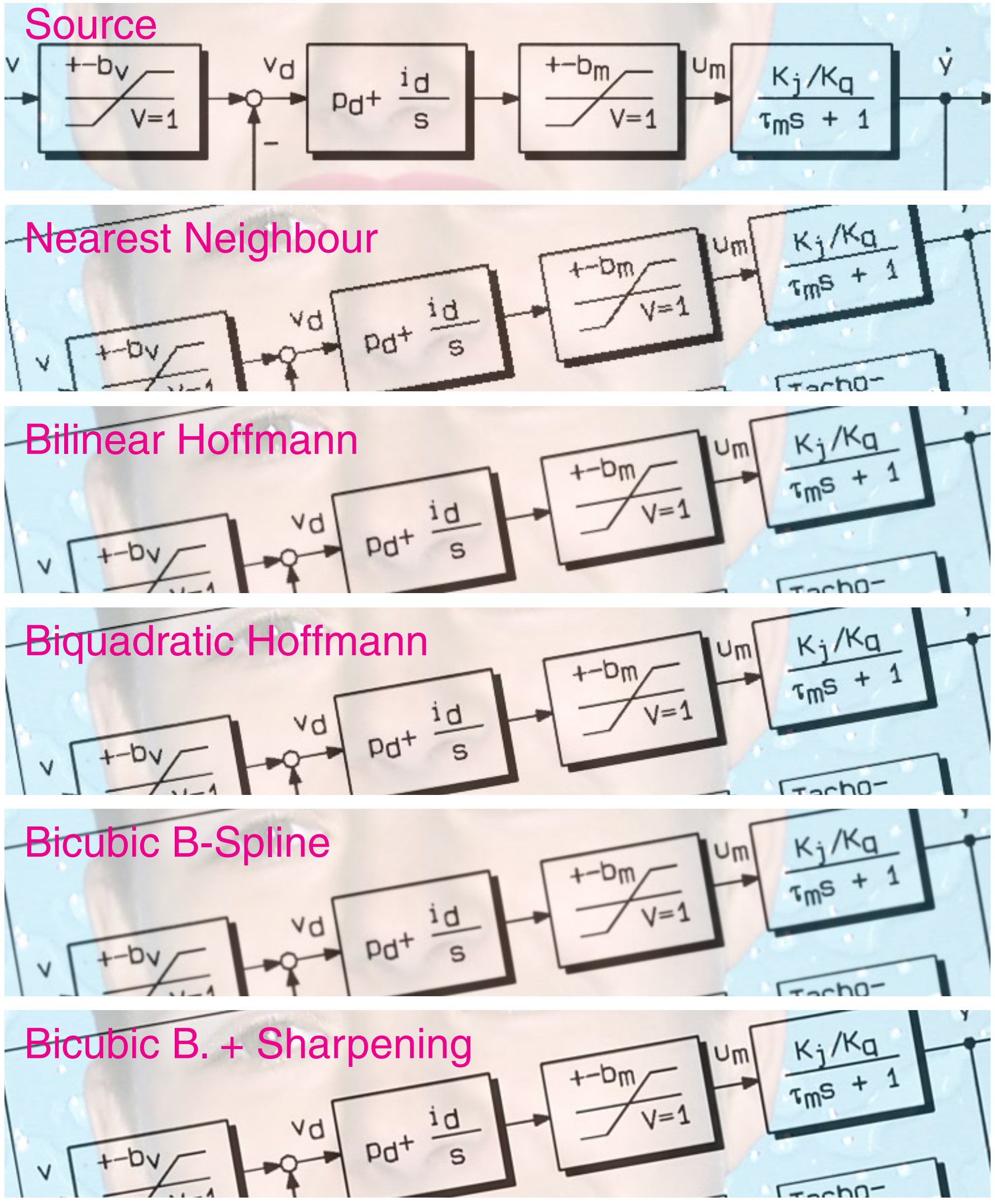
1. $c=c^{1.6}$ 2. Apply operations 3. $c=c^{(1/1.6)}$

Best results for 1.6 instead of 2.2



3. Examples 2 / Zoom 200%

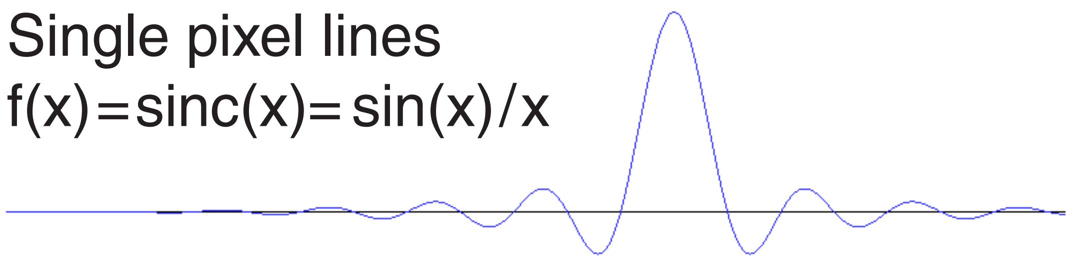
Operations without Gamma Correction

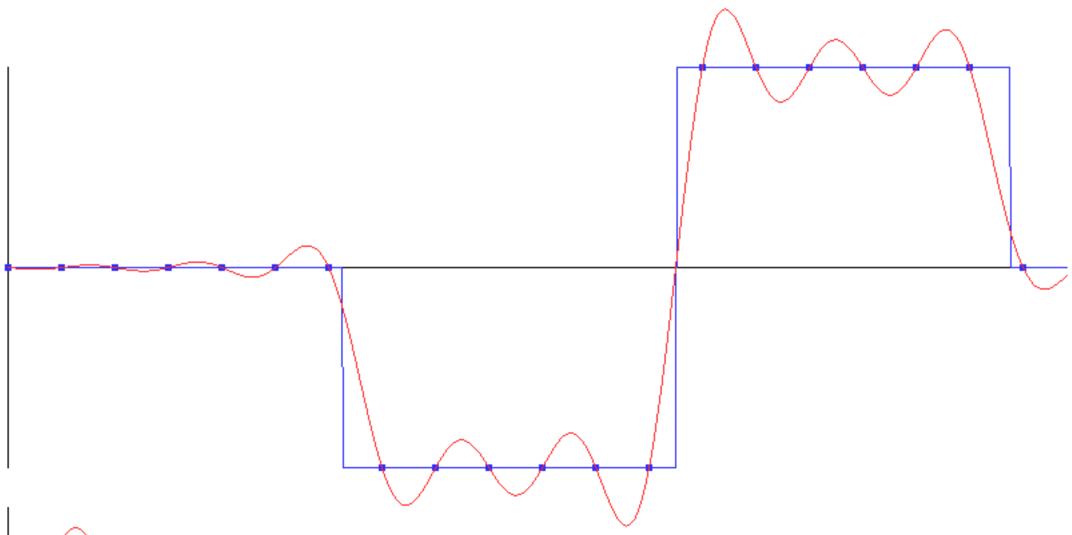


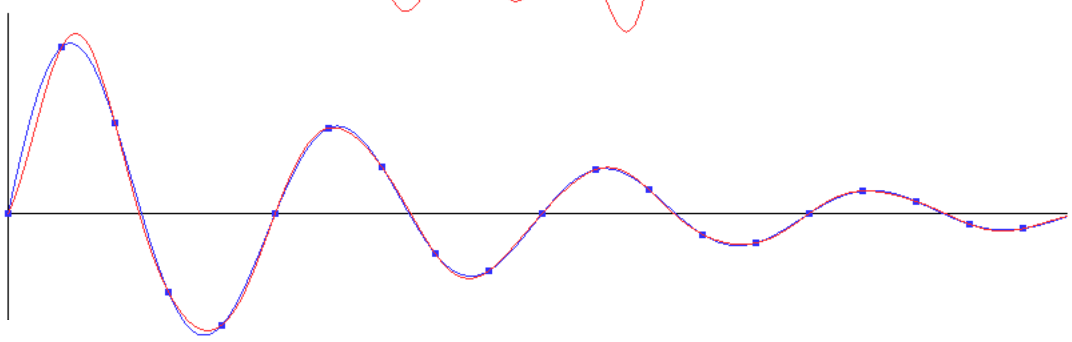
The author, after rotating
18 times by 10° and once by 180°,
then downscaled by factor 0.5
Digital Photo
Biquadratic Interpolation

4. Examples 3 / Zoom 200%

Single pixel lines
$f(x)=\mathrm{sinc}(x)=\sin(x)/x$



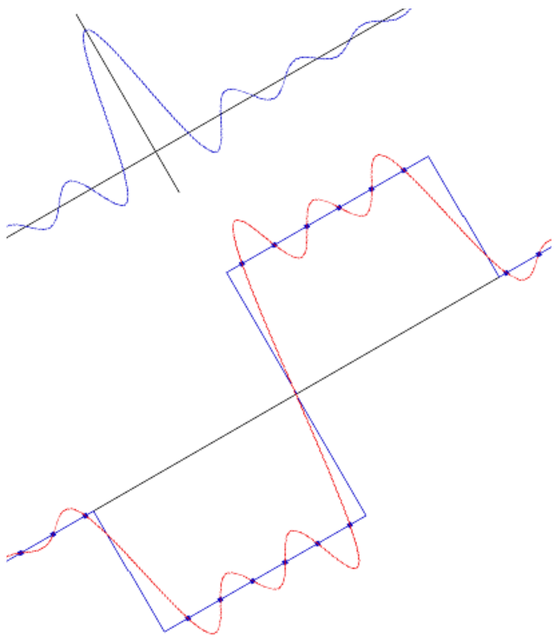




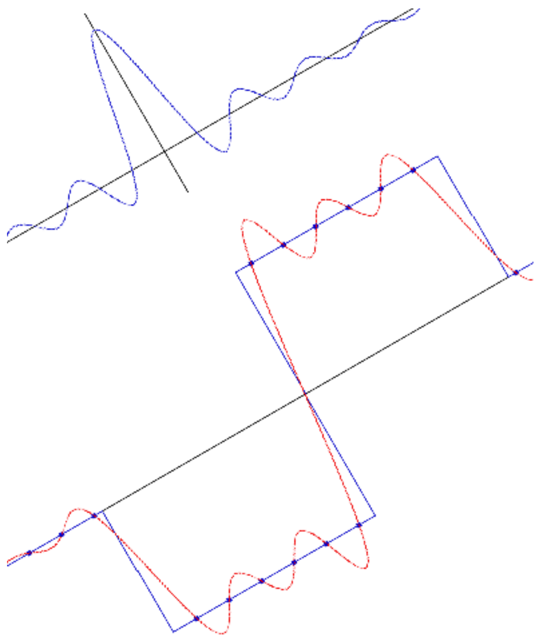
Sometimes the function
$f(x)=\mathrm{sinc}(x)=\sin(x)/x$
is recommended for interpolation. Results are very bad if the signal is not band limited, e.g. the step function (graphic left).

Now we use the graphic just as an example with single pixel lines, rotate and scale (below) .
A computer graphic may consist of vertical lines, alternating black and white. With some normalization this is in horizontal direction the sequence +1,-1,+1,...sampled by frequency f_o.
The lowest frequency is the Nyquist frequency $f_n=f_o/2$.
This image cannot be 'reconstructed', not even by an ideal low pass filter. Any real image, graphic or scanned, can contain components with f_n.
Single pixel line images have a considerable contribution of f_n. Therefore a perfect interpolation is never possible for arbitrary images.

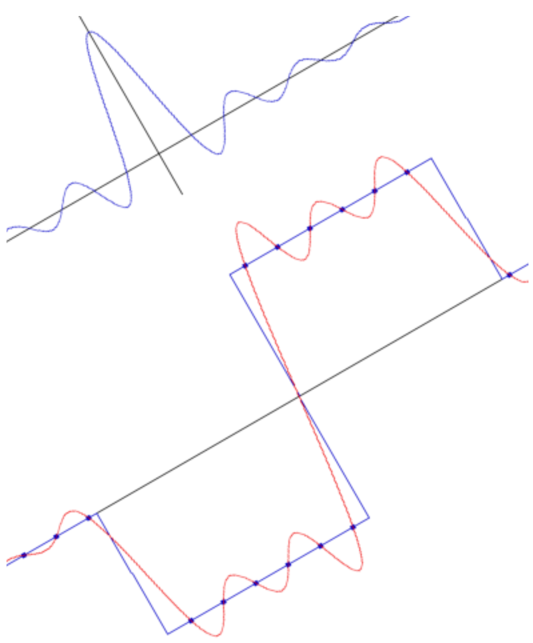
Downscaled 70%, rotated 30°,in one pass



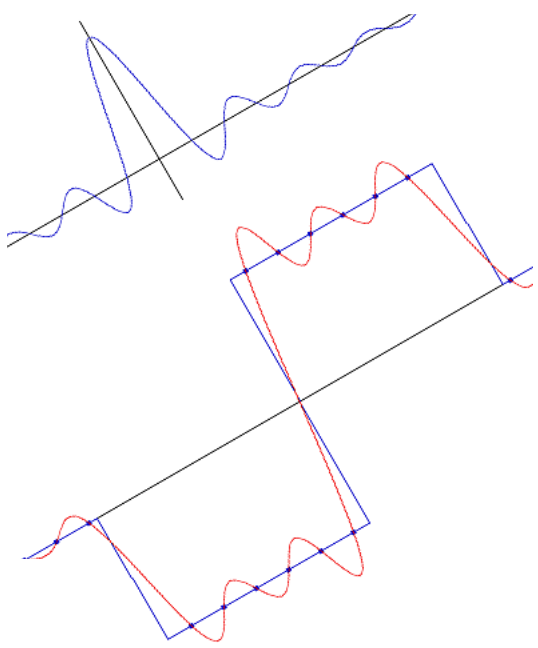
Bilinear Hoffmann



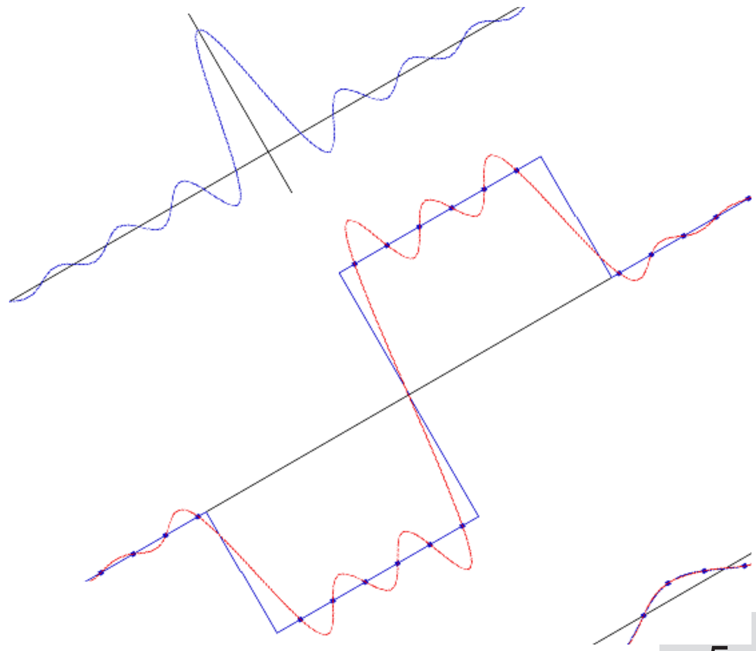
Biquadratic Hoffmann



Bicubic B-Spline

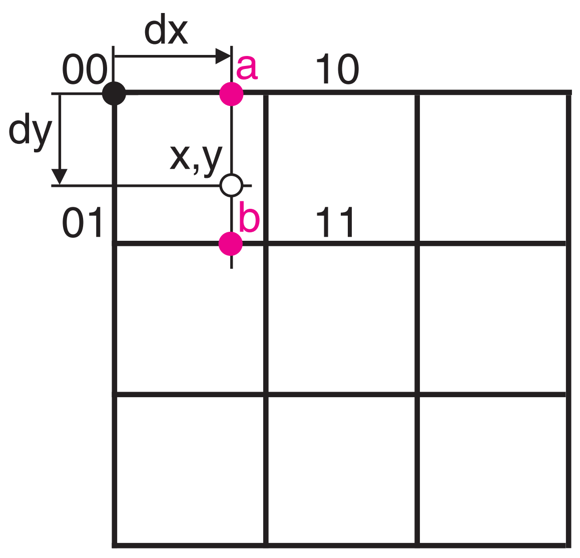


Bicubic B+Sharpened



Bicubic Hoffmann

5. Bilinear Interpolation

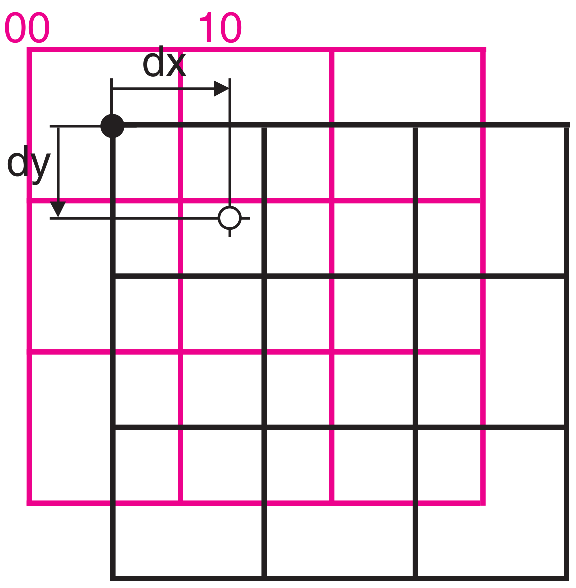


$$C_a = C_{00} + (C_{10} - C_{00})dx$$
$$C_b = C_{01} + (C_{11} - C_{01})dx$$
$$C = C_a + (C_b - C_a)dy$$

A destination image is filled by a double loop in u,v directions. The destination image is a copy of the source image, translated, rotated, scaled and eventually perspectively rectified.
The coordinates x,y in the source image are calculated for each destination pixel u,v.
Generally, x and y are non-integer numbers.
The drawing shows the source image. Each knot represents a pixel.
Normally, the nearest pixel p_{00} is found by truncation (integer part of x,y).
Next neighbours are p_{10}, p_{01}, and p_{11}.
dx and dy are the local non-integer deviations.
For each channel C = R,G,B the interpolation is executed by

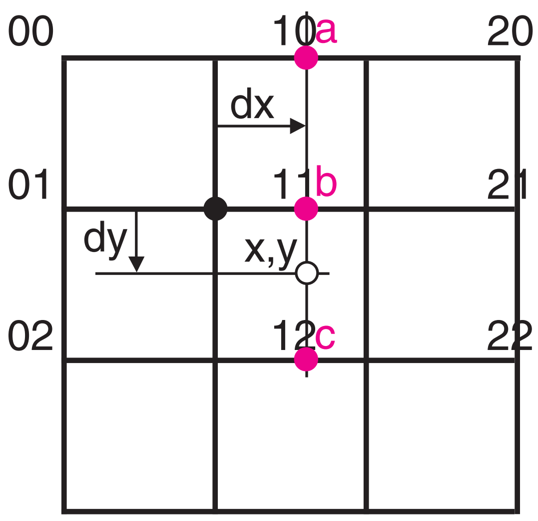
$$C = C_{00}(1-dx)(1-dy) + C_{10}dx(1-dy) + C_{01}(1-dx)dy + C_{11}dxdy$$

Now let us assume a single pixel line in vertical direction. Scaling by factor 1.0 would reproduce the line. Scaling by factor 0.5 would either reproduce and ignore the line alternating. This means: thin lines in arbitrary directions would appear with gaps. Therefore we use a shift of 0.5 pixels in the source image. Then a scaling by factor 1.0 would show all lines slightly blurred, because the mean value of neighbours is used. E.g. a scale factor 1.01 would cause a balanced blur, instead of alternating sharp and blurred lines. Scaling by factor 0.5 shows interpolated values. Thin lines don´t have gaps. This is valid for scale factors down to 0.5.
Upscaling works anyway as usual. The average shift of 0.5 pixels in either direction doesn´t affect or deteriorate photos. A calculated dx=0 results in a effective dx=0.5.



The nearest pixel p_{00} is calculated by truncation, as usual.
Scaling and Rotation refers to the center of the image x_c, y_c.
For image width 0...1279 pixels or 1...1280 pixels:
Bilinear Standard uses x_c=640
Bilinear Hoffmann uses x_c=640.5
The interpolation code itself is not modified.

6. Biquadratic Interpolation

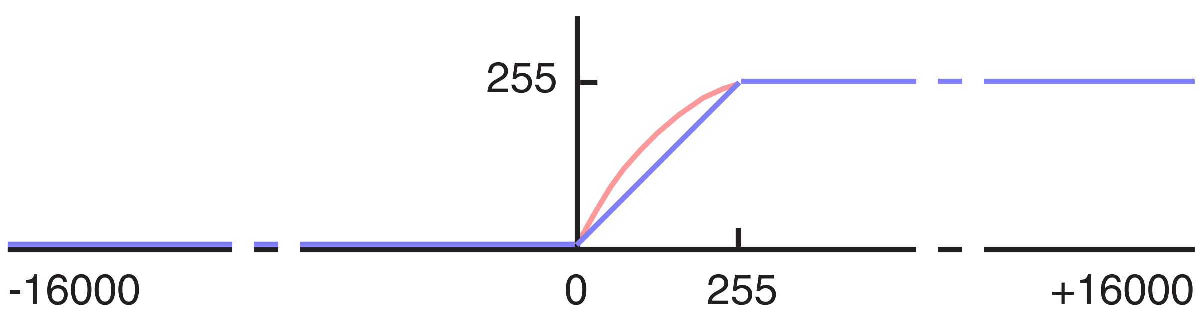


A destination image is filled by a double loop in u,v directions. The destination image is a copy of the source image, translated, rotated, scaled and eventually perspectively rectified.
The coordinates x,y in the source image are calculated for each destination pixel u,v.
Generally, x and y are non-integer numbers.
The drawing shows the source image. Each knot represents a pixel.
By rounding the nearest pixel p_{11} is found.
Next neighbours are 8 pixels p_{00} to p_{22} .
dx and dy are the local non-integer deviations.
For each channel C = R,G,B the interpolation is executed by these parabolas:

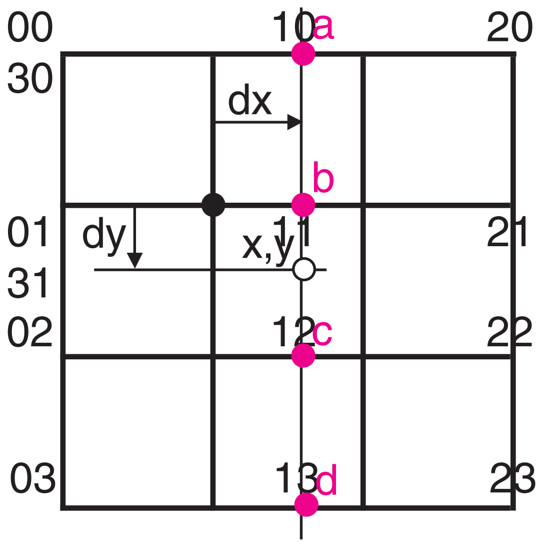
$$qx = 0.5\,dx^2$$
$$qy = 0.5dy^2$$
$$dx = 0.5dx$$
$$dy = 0.5dy$$
$$C_a = C_{10} + (C_{20}-C_{00})dx + (C_{00}-2C_{10}+C_{20})qx$$
$$C_b = C_{11} + (C_{21}-C_{01})dx + (C_{01}-2C_{11}+C_{21})qx$$
$$C_c = C_{12} + (C_{22}-C_{02})dx + (C_{02}-2C_{12}+C_{22})qx$$
$$C = C_b + (C_c - C_a)dy + (C_a - 2C_b + C_c)qy$$

The method works good for photos, but sharp lines create halos. This is slightly better with Gamma Correction.

The results need to be clipped. This can be done by a clipping table. Clipping by tables is much faster than logical clipping, because the instruction queue is not disturbed by jumps.
Gamma compensations can be applied here as well (red graph).



7. Bicubic Interpolation



A destination image is filled by a double loop in u,v directions. The destination image is a copy of the source image, translated, rotated, scaled and eventually perspectively rectified.
The coordinates x,y in the source image are calculated for each destination pixel u,v.
Generally, x and y are non-integer numbers.
The drawing shows the source image. Each knot represents a pixel.
By truncation the nearest pixel p_{11} is found.
Next neighbours are 15 pixels p_{00} to p_{33} .
dx and dy are the local non-integer deviations. The powers 2,3 of dx,dy are calculated in advance by multiplication, because they are constant for the whole interpolation.
For each channel C = R,G,B the interpolation is executed by these parabolas:

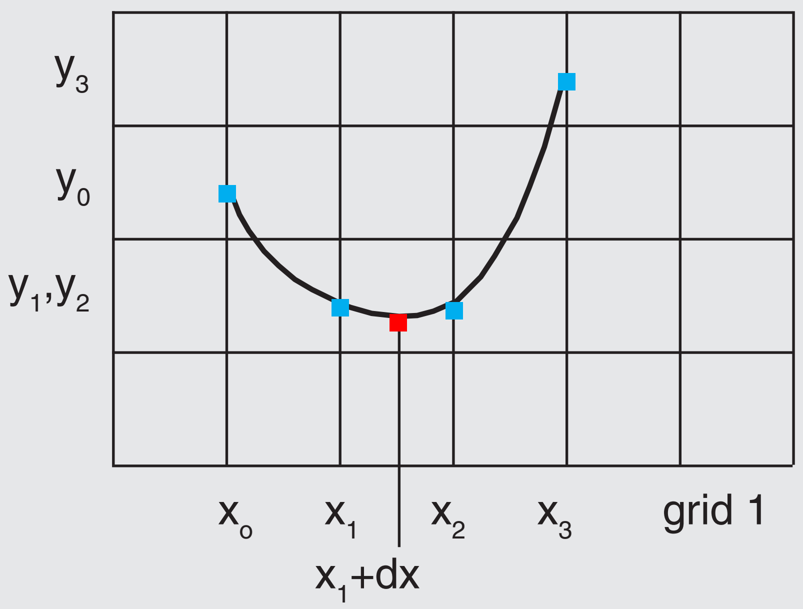
For each row j = 0,1,2,3 = a,b,c,d :

$$
\begin{aligned}
d_0 &= C_{0j} - C_{1j} \\
d_2 &= C_{2j} - C_{1j} \\
d_3 &= C_{3j} - C_{1j} \\
a_0 &= C_{1j} \\
a_1 &= -(1/3)d_0 + d_2 - (1/6)d_3 \\
a_2 &= (1/2)d_0 + (1/2)d_2 \\
a_3 &= -(1/6)d_0 - (1/2)d_2 + (1/6)d_3 \\
C_j &= a_0 + a_1 dx + a_2 dx^2 + a_3 dx^3
\end{aligned}
$$

This delivers C_a, C_b, C_c, C_d

Then these values are interpolated in vertical direction:

$$
\begin{aligned}
d_0 &= C_a - C_b \\
d_2 &= C_c - C_b \\
d_3 &= C_d - C_b \\
a_0 &= C_b \\
a_1 &= -(1/3)d_0 + d_2 - (1/6)d_3 \\
a_2 &= (1/2)d_0 + (1/2)d_2 \\
a_3 &= -(1/6)d_0 - (1/2)d_2 + (1/6)d_3 \\
C &= a_0 + a_1 dy + a_2 dy^2 + a_3 dy^3
\end{aligned}
$$



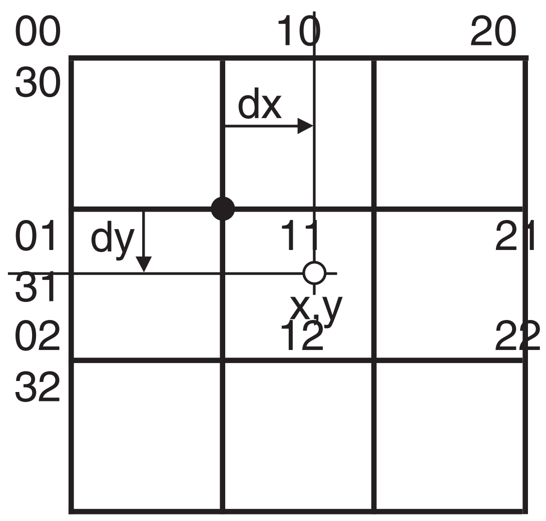
$$y(x) = a_0 + a_1 dx + a_2 dx^2 + a_3 dx^3$$

$$
\begin{aligned}
y_0 &= a_0 - a_1 + a_2 - a_3 \\
y_1 &= a_0 \\
y_2 &= a_0 + a_1 + a_2 + a_3 \\
y_3 &= a_0 + 2a_1 + 4a_2 + 8a_3
\end{aligned}
$$

Solve by *Cramer's* rule

Clipping is performed by a clipping table, as described on the previous page.
The results are very similar to Photoshop Bicubic Interpolation.

8. Bicubic B-Spline Interpolation



A destination image is filled by a double loop in u,v directions. The destination image is a copy of the source image, translated, rotated, scaled and eventually perspectively rectified.
The coordinates x,y in the source image are calculated for each destination pixel u,v.
Generally, x and y are non-integer numbers.
The drawing shows the source image. Each knot represents a pixel.
By truncation the nearest pixel p_{11} is found (integer part of x,y).
Next neighbours are 15 pixels p_{00} to p_{33} .
dx and dy are the local non-integer deviations.
For each channel C = R,G,B the interpolation is executed by these B-Spline functions:

$$C = \sum_{k=0}^{3} \sum_{i=0}^{3} C_{ik}\, R(i-1-dx)\, R(k-1-dy)$$

$$R(x) = [\,P(x+2)^3 - 4P(x+1)^3 + 6P(x)^3 - 4P(x-1)^3\,]/$$

$$\begin{array}{llll}
P(x+2): & \text{If } x+2>0 & \text{Then } P=x+2 & \text{Else } P=0 \\
P(x+1): & \text{If } x+1>0 & \text{Then } P=x+1 & \text{Else } P=0 \\
P(x): & \text{If } x>0 & \text{Then } P=x & \text{Else } P=0 \\
P(x-1): & \text{If } x-1>0 & \text{Then } P=x-1 & \text{Else } P=0
\end{array}$$

The nomenclature differs slightly from P.Bourke's document. The argument of R(x) can have either sign because R(x) is symmetric. The speed can be improved by a table with -200 to 200 entries for x = -2 to +2.
The interpolation is based on a kind of Gaussian blur. The bell is centered at the actual non-integer position x,y. The application of splines is not really essential.

Function R(x) / blue

Function R(x) Substitute / red

$R(x) = (2/3)e^{-1.40xx}$

This delivers practically the same result for images, but the colors in uniformly colored areas are not accurately reproduced

9. Sharpening Filter for Bicubic B-Spl. Interpolation

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

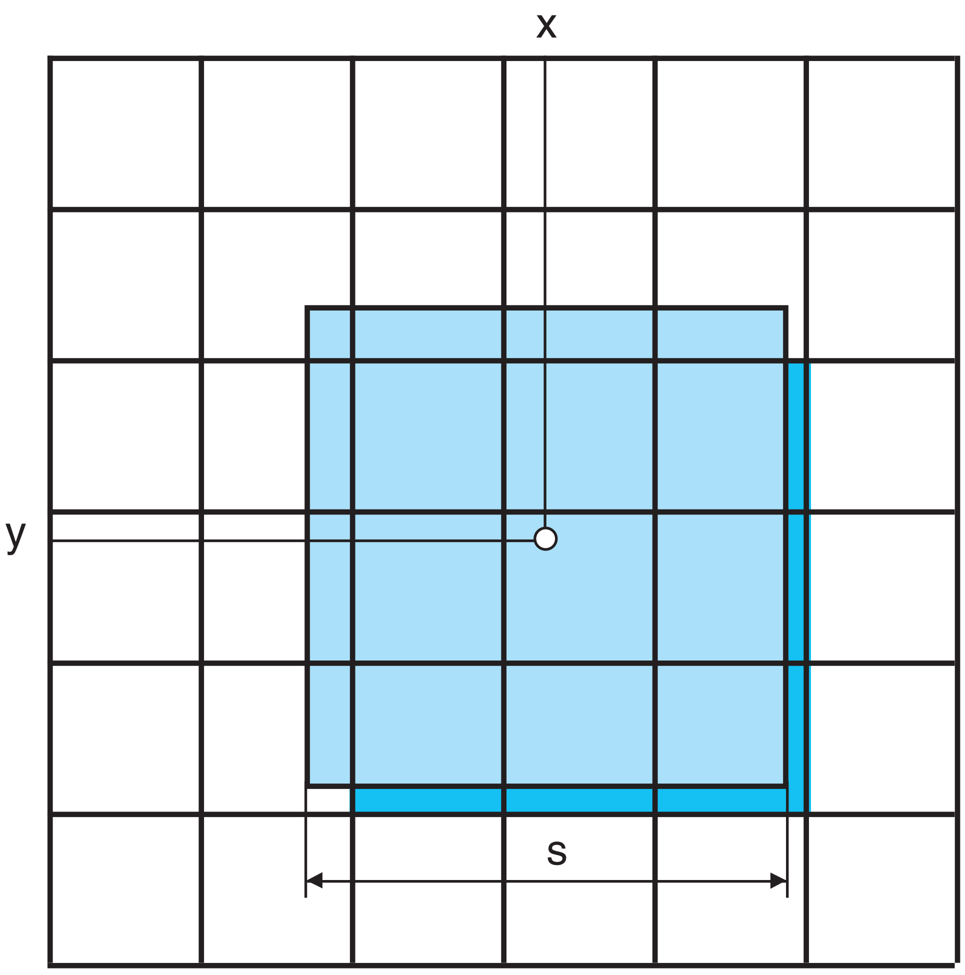
-0.0035	-0.0159	-0.0262	-0.0159	-0.0035
-0.0159	-0.0712	-0.1173	-0.0712	-0.0159
-0.0262	-0.1173	2.0	-0.1173	-0.0262
-0.0159	-0.0712	-0.1173	-0.0712	-0.0159
-0.0035	-0.0159	-0.0262	-0.0159	-0.0035

The drawing shows the numbering and the weight factors (right side) for the kernel for a sharpening filter, here for n=2. The algorithm works for any $n \geq 1$. The total number of elements is $m=(2 \cdot n+1)^2$.
The center weight factor fs[13]=2.0 is a positive peak. The other weight factors fs[k] are calculated by a negative Gauss Bell, according to the code below.
The sum of negative weight factors is -1.0 , therefore a uniformly colored area remains unfiltered, as required.
For Bicubic B-Spline Interpolation without scaling, e.g. a rotation, n=1 is recommended.

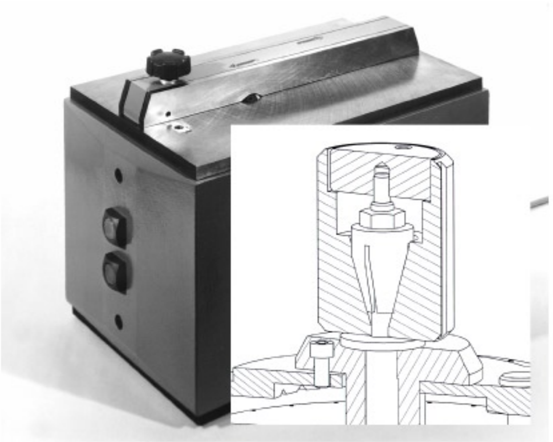
```
Tutorial code, not optimized

sm:=0;
k :=1;
For j:=-n to n Do
For i:=-n to n Do
Begin
 ra:=Sqrt(Sqr(i)+Sqr(j))/n;
 ra:=exp(-2*Sqr(ra));
 fs[k]:=-ra;
 If (i<>0) Or (j<>0) Then sm:=sm+ra;
 Inc(k);
End;
k :=1;
For j:=-n to n Do
For i:=-n to n Do
Begin
 fs[k]:=fs[k]/sm;
 If (i=0) And (j=0) Then fs[k]:=2.0;
 Inc(k);
End;
```

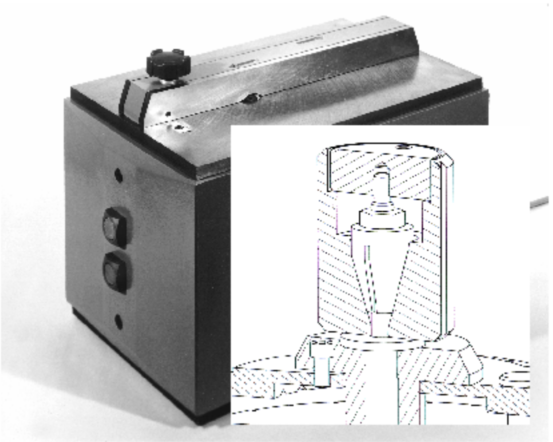
10.1 Box Averager for Downsampling / Zoom 200%



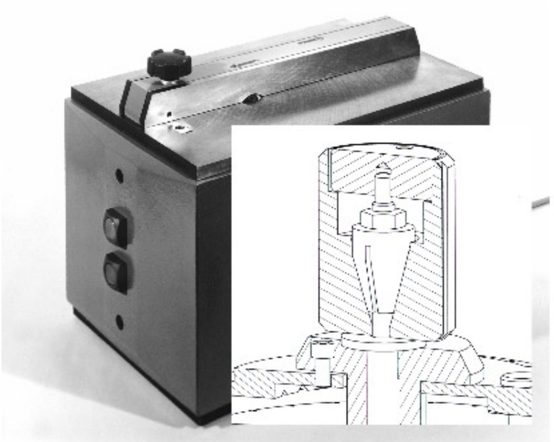
Downscaling by a scale factor less 0.5 can be done by a Box Averager. For convenience we write here s=3.2 for downscaling by 1/3.2.
The coordinates x,y in the source image are calculated for each destination pixel u,v. x and y are generally non-integer numbers. Then we average all pixels for i= round(x-s/2) to i=round(x+s/2) and for j in y direction similarly (dark cyan). It´s quite useless to take fragments of pixels into consideration (light cyan). The Box Averager delivers the best results for images with single pixel lines. The example shows a scanned technical drawing. For programmed single pixel lines the effect is even more impressive.
Finally the best solution was found by averaging only those pixels which are entirely in the box:
i= round(x-s/2+0.5) to i=round(x+s/2-0.5) . This delivers rather sharp results.



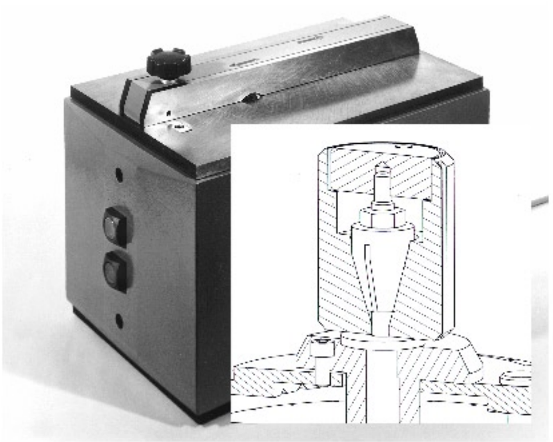
Box Averager



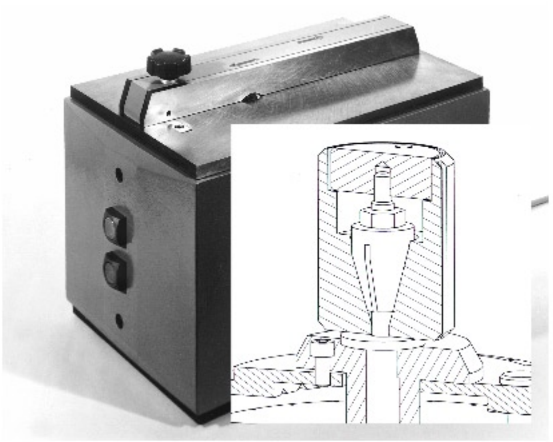
Nearest Neighbour



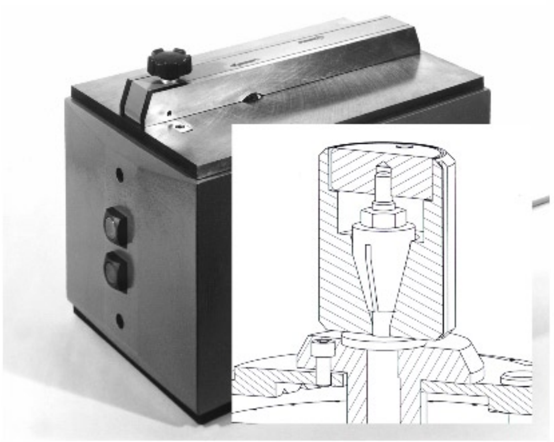
Bilinear



Biquadratic



Bicubic



Bicubic B-Spline

10.2 Box Averager for Thumbnails / Zoom 100%

This example shows the fast creation of thumbnails from JPEGs.

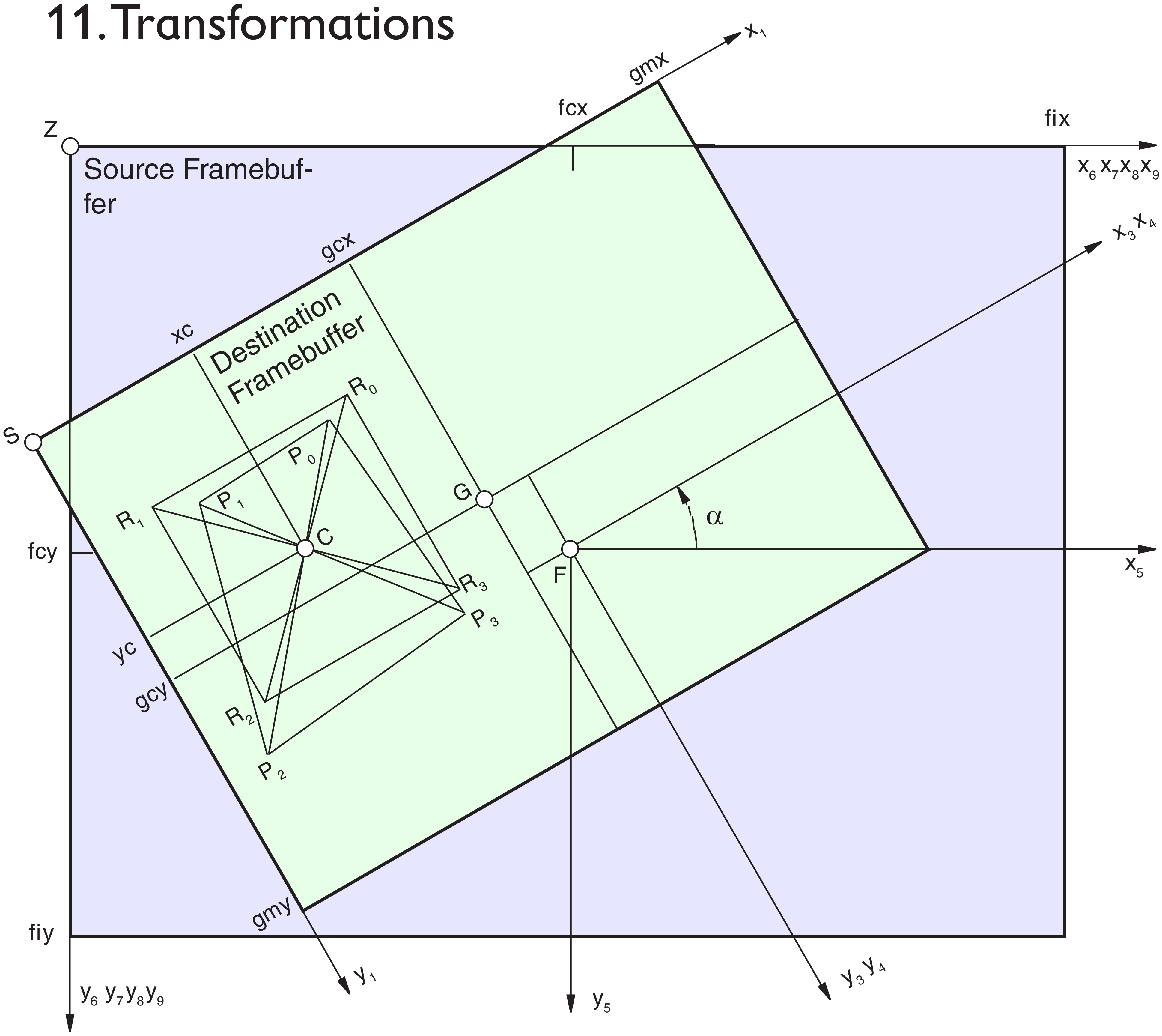
1. Use only the DC components or DC and the first two or four cosines.
2. Downscale by box averager, here for s=5.
3. Sharpen by the sharpening filter in chapter 9 for n=2 (5x5 pixel box).

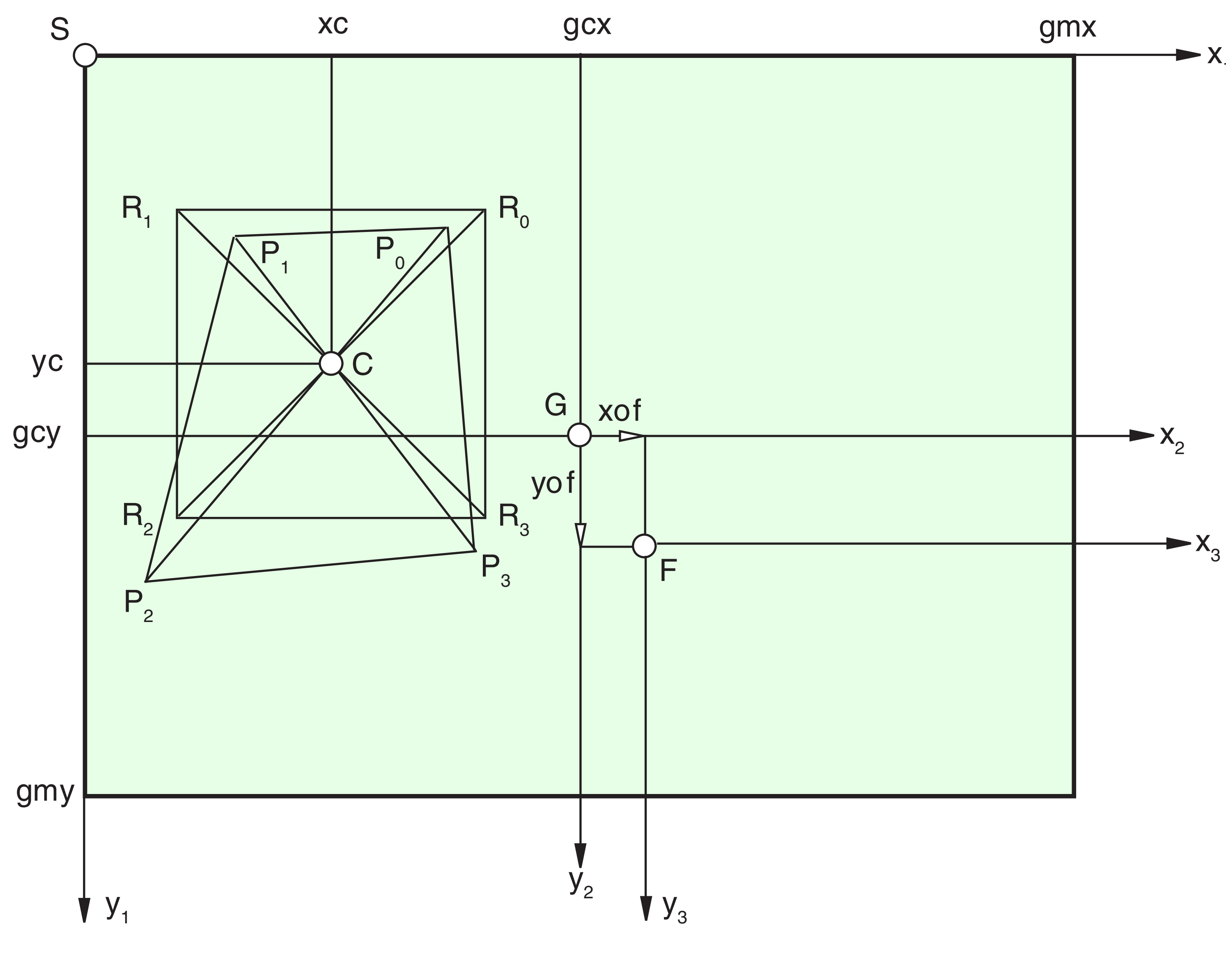
DC values of JPEG

Downscaled
Box Averager

Sharpened

11. Transformations





12.1 Tutorial Program

Tutorial programs explain algorithms.
They cannot be copied because they need external libraries.

```
{  Complete Translation, Rotation, Scaling, Projective Mapping    }
  Gernot Hoffmann
  April 08, 2002
  Quadratic Interpolation
  This is a tutorial version
  The  actual version uses incremental loops
  and all kernels are written in assembly language                }
```

```
      a1:=p8[1]; a2:=p8[2]; a3:=p8[3]; {   For Perspective        }
      b1:=p8[4]; b2:=p8[5]; b3:=p8[6]; {   Refer to [2]           }
      c1:=p8[7]; c2:=p8[8]; c3:=1.0;

  ColToScr(frab,baco);   { Background                             }

  For y1:=0 to gmy Do    { Loops in Destination Frame             }
  Begin
  For x1:=0 To gmx Do
  Begin
  x5:=sx*(x1-gcx-xof);   { sx,sy  Scaling                         }
  y5:=sy*(y1-gcy-yof);
  x6:=+cp*x5+sp*y5+fcx;  { cp=cos(ang) sp=sin(ang)                }
  y6:=-sp*x5+cp*y5+fcy;
  x7:= a1*x6+a2*y6+a3;   { a1..a3 external                        }
  y7:= b1*x6+b2*y6+b3;   { b1..b3 external                        }
  z7:= c1*x6+c2*y6+1.0;  { c1, c2 external                        }
  x8:= x7/z7;            { Projection                             }
  y8:= y7/z7;
  x9:= Round(x8);        { Reference in                           }
  y9:= Round(y8);        { Source Framebuffer FMem                }

  If (x9>0) And (x9<fix) And (y9>0) And (y9<fiy) Then
  Begin
    FramToRGB (FMem[y9-1]^[x9-1],r00,g00,b00);      { xy          }
    FramToRGB (FMem[y9-1]^[x9  ],r10,g10,b10);      { 00  10  20  }
    FramToRGB (FMem[y9-1]^[x9+1],r20,g20,b20);      { 01  11  21  }
    FramToRGB (FMem[y9  ]^[x9-1],r01,g01,b01);      { 02  12  22  }
    FramToRGB (FMem[y9  ]^[x9  ],r11,g11,b11);
    FramToRGB (FMem[y9  ]^[x9+1],r21,g21,b21);
    FramToRGB (FMem[y9+1]^[x9-1],r02,g02,b02);
    FramToRGB (FMem[y9+1]^[x9  ],r12,g12,b12);
    FramToRGB (FMem[y9+1]^[x9+1],r22,g22,b22);
    dx:=x8-x9;
    dy:=y8-y9;
    qx:=0.5*Sqr(dx);
    qy:=0.5*Sqr(dy);
    dx:=0.5*dx;
    dy:=0.5*dy;
```

```
                    ra:=r10+(r20-r00)*dx+(r00-2*r10+r20)*qx;
                    rb:=r11+(r21-r01)*dx+(r01-2*r11+r21)*qx;
                    rc:=r12+(r22-r02)*dx+(r02-2*r12+r22)*qx;
                    ga:=g10+(g20-g00)*dx+(g00-2*g10+g20)*qx;
                    gb:=g11+(g21-g01)*dx+(g01-2*g11+g21)*qx;
                    gc:=g12+(g22-g02)*dx+(g02-2*g12+g22)*qx;
                    ba:=b10+(b20-b00)*dx+(b00-2*b10+b20)*qx;
                    bb:=b11+(b21-b01)*dx+(b01-2*b11+b21)*qx;
                    bc:=b12+(b22-b02)*dx+(b02-2*b12+b22)*qx;
                    r11:=Round(rb+(rc-ra)*dy+(ra-2*rb+rc)*qy);
                    g11:=Round(gb+(gc-ga)*dy+(ga-2*gb+gc)*qy);
                    b11:=Round(bb+(bc-ba)*dy+(ba-2*bb+bc)*qy);
                    r11:=LimTab^[r11];
                    { Table Limiter }
                    g11:=LimTab^[g11];
                    b11:=LimTab^[b11];
                    RGBToFram(r11,g11,b11,fram);{ Pixel in Dest.Buffer }
                    SetSixel(x1,y1,fram);
          End;    { If... }
         End;     { x1... }
        End;      { y1... }
```

13.1 Incremental Algorithm

```
This is a tutorial example for an incremental algorithm
Translation, Rotation and Scaling with Linear Interpolation
G.Hoffmann, April 10 2002
```

```
Begin
  SicCoc(wrad*ang,sp,cp);         { Fast sine, cosine    }
  fix:=mxpix;     fiy:=mypix;      { Integer              }
  gcx:=xpx Div 2; gcy:=ypx Div 2; { Center of Screen     }
  fcx:=fix Div 2 +0.5;
  fcy:=fiy Div 2 +0.5;             { Center of Image FMem }

  xx:= sx*cp;          { Increment along x1=0..gmx,any y1 }
  yx:=-sx*sp;
  xy:= sy*sp;          { Increment along y1=0..gmy,x1=0   }
  yy:= sy*cp;
  x1:=0; y1:=0;
  x5:= sx*(x1-gcx-xof);
  y5:= sy*(y1-gcy-yof);
  x6:=+cp*x5+sp*y5+fcx;
  y6:=-sp*x5+cp*y5+fcy;
  x8:= x6;
  y8:= y6;
  x8o:=x8;             { Index o means values along y=0..gmy, x1=0}
  y8o:=y8;             { Incremented by xy, yy, zy               }

  For y1:=0 To gmy Do  {  Destination FrameBuffer        }
  Begin
  For x1:=0 To gmx Do
   Begin
   x9:=Trunc(x8);      { Reference                       }
   y9:=Trunc(y8);       { in Source Framebuffer           }
   If (x9>=0) And (x9<fix) And (y9>=0) And (y9<fiy) Then
    Begin
     FramToRGB(FMem[y9  ]^[x9  ],r00,g00,b00); { xy           }
     FramToRGB(FMem[y9  ]^[x9+1],r10,g10,b10); { 00  10       }
     FramToRGB(FMem[y9+1]^[x9  ],r01,g01,b01); { 01  11       }
     FramToRGB(FMem[y9+1]^[x9+1],r11,g11,b11);
     dx:=x8-x9;
     dy:=y8-y9;
     ra:=r00+(r10-r00)*dx;
     rb:=r01+(r11-r01)*dx;
     ga:=g00+(g10-g00)*dx;
     gb:=g01+(g11-g01)*dx;
     ba:=b00+(b10-b00)*dx;
     bb:=b01+(b11-b01)*dx;
     r11:=Round(ra+(rb-ra)*dy);
     g11:=Round(ga+(gb-ga)*dy);
     b11:=Round(ba+(bb-ba)*dy);
```

13.2 Incremental Algorithm

```
    RGBtoFram(r11,g11,b11,fram);
    SetSixel(x1,y1,fram);
    End; { If... }
    x8:=x8+xx;
    y8:=y8+yx;
    End;  { x1... }
    x8o:=x8o+xy;
    y8o:=y8o+yy;
    x8 :=x8o;
    y8 :=y8o;
  End;   { y1... }
End;
```

14.1 Barrel/Pincushion Correction

Lens optics generate either barrel or pincushion distortions. Here is a simple model for the correction.
The destination image (dst) is filled by a regular double loop. The pixels are taken from the source image (src), either by nearest neighbour for the preview or by bicubic interpolation for the final result.

x_c, y_c is the image center.

x_b, y_b is the offset for the optical center,

relative to x_c, y_c.

$dx \;= x_c + x_b$

$dy \;= y_c + y_b$

$R_{dst}^2 = (x_{dst} - dx)^2 + (y_{dst} - dy)^2$

$K_r \;= R_{src} / R_{dst} = 1 + K R_{dst}^2$

Scaling by the half image width $s_c = x_c$

and retaining the average image width:

$K_r \;= 1 - K + K(R_{dst}^2 / s_c^2) = 1 - K + (K / s_c^2) R_d^2$

$x_{src} = dx + K_r (x_{dst} - dx)$

$y_{src} = dy + K_r (y_{dst} - dy)$

Use $K = \pm n \cdot 0.01$.

14.2 Barrel/Pincushion Correction

```
Here are two tutorial examples for Barrel/Pincushion Correction
1.  Interpolation by Nearest Neighbour
2.  Interpolation Bicubic
G.Hoffmann, January 15/2006
```

```
Procedure ColTranBP1(xb,yb: Integer; K: Single);
{   Barrel/Pincushion correction
    Nearest neighbour
    Write from FMem to Screen
    Uses 0..gmx, 0..gmy Pixels on Screen and in FrameBuffer
    xc,yc   :   Center of image
    xb,yb   :   Center of distortion, relative to xc,yc
    K       :   Barrel/Pincushion correction
    1       :   Destination
    9       :   Source                                          }
Var     x1,y1,x9,y9                                : Integer;
        x5,y5,x8,y8,k1,k2,xc,yc,dx,dy,Kr,Rd   : Single;
Begin
    xpx:=gmx+1;  ypx:=gmy+1;   { Integer                        }
    xc:=0.5*xpx; yc:=0.5*ypx;  { Center                         }
    k1:=1-K;
    k2:=K/Sqr(xc);
    dx:=xb+xc;
    dy:=yb+yc;
    ColToScr(frab,Whit);       { white background               }
    For y1:=0 To gmy Do
    Begin
    For x1:=0 To gmx Do
        Begin
        x5:= x1-dx;
        y5:= y1-dy;
        Rd:=Sqr(x5)+Sqr(y5);
        Kr:=k1+k2*Rd;
        x8:=dx+x5*Kr;
        y8:=dy+y5*Kr;
        x9:=Round(x8);         { Reference                      }
        y9:=Round(y8);        { in Source Framebuffer          }
        If (x9>0) And (x9<gmx) And (y9>0) And (y9<gmy) Then
            Begin
             SetSixel(x1,y1,FMem[y9]^[x9]); { Pixel on Screen}
            End; { If... }
        End; { x1... }
  End;       { y1... }
End;
```

14.3 Barrel/Pincushion Correction

```
Procedure ColTranBP3(xb,yb: Integer; K: Single);
{   Barrel/Pincushion correction
    Bicubic interpolation
    Write from FMem to Screen
    Uses 0..gmx, 0..gmy Pixels on Screen and in FrameBuffer
    xc,yc   :   Center of image
    xb,yb   :   Center of distortion, relative to xc,yc
    K       :   Barrel/Pincushion correction
    1       :   Destination
    9       :   Source                  }
Var     x1,y1,x9,y9                     : Integer;
        x5,y5,x8,y8,dx1,dy1             : Single;
        Kr,Rd1,k1,k2,xc,yc              : Single;
Var     r00,r10,r20,r30,r01,r11,r21,r31,
        r02,r12,r22,r32,r03,r13,r23,r33: Integer;
        g00,g10,g20,g30,g01,g11,g21,g31,
        g02,g12,g22,g32,g03,g13,g23,g33: Integer;
        b00,b10,b20,b30,b01,b11,b21,b31,
        b02,b12,b22,b32,b03,b13,b23,b33: Integer;
        a1,a2,a3,b1,b2,b3,c1,c2         : Single;
        dx,dy,dx2,dx3,dy2,dy3           : Single;
        ra,rb,rc,rd                     : Single;
        ga,gb,gc,gd                     : Single;
        ba,bb,bc,bd                     : Single;
        prgb                            : LongInt;
Const   k16:Single=1/6;
        k13:Single=1/3;
        k12:Single=1/2;
Procedure FramToRgb(Fram: LongInt; Var ri,gi,bi: Integer);
Begin
    bi:=Fram        AND $000000FF;
    gi:=Fram SHR  8 AND $000000FF;
    ri:=Fram SHR 16 AND $000000FF;
End;
Procedure RgbToFram(ri,gi,bi: Integer; Var Fram: LongInt);
Var lr,lg,lb: LongInt;
Begin { Byte range for Integer inputs expected }
    lr:=ri; lg:=gi; lb:=bi;
    Fram:= lr SHL 16 + lg SHL 8 + lb ;
End;
Procedure Par3(c0,c1,c2,c3:Integer;dx,dx2,dx3:Single;Var cj:Single);
Var d0,d2,d3    : Integer;  a1,a2,a3    : Single;
Begin
    d0:= c0-c1;
    d2:= c2-c1;
    d3:= c3-c1;
    a1:= d2-k13*d0-k16*d3;
    a2:= k12*(d0+d2);
    a3:= k16*(d3-d0)-k12*d2;
    cj:= c1+a1*dx+a2*dx2+a3*dx3;
End;
```

14.4 Barrel/Pincushion Correction

```
Procedure Pbr3(c0,c1,c2,c3,dy,dy2,dy3: Single; Var cj: Integer);
Var     d0,d2,d3        : Single;
            a1,a2,a3    : Single;
            ci          : Integer;
Begin
    d0:= c0-c1;
    d2:= c2-c1;
    d3:= c3-c1;
    a1:= d2-k13*d0-k16*d3;
    a2:= k12*(d0+d2);
    a3:= k16*(d3-d0)-k12*d2;
    { Table Limiter for cj in 0..255 }
    cj:=LimTab^[Round(c1+a1*dy+a2*dy2+a3*dy3)];
End;

Begin
    xpx:=gmx+1;  ypx:=gmy+1;      { Integer                      }
    xc:=0.5*xpx; yc:=0.5*ypx;     { Center                       }
    k1:=1-K;
    k2:=K/Sqr(xc);
    dx1:=xb+xc;
    dy1:=yb+yc;
    { Extrapolation }
    For x1:=0  To gmx     Do FMem[gmy+1]^[x1]:=FMem[gmy]^[x1];
    For x1:=0  To gmx     Do FMem[gmy+2]^[x1]:=FMem[gmy]^[x1];
    For y1:=0  To gmy+2   Do FMem[y1]^[gmx+1]:=FMem[y1]^[gmx];
    For y1:=0  To gmy+2   Do FMem[y1]^[gmx+2]:=FMem[y1]^[gmx];
    ColToScr(frab,Whit);          { White Background             }
    For y1:=0 To gmy Do
    Begin
    For x1:=0 To gmx Do
        Begin
        x5:= x1-dx1;
        y5:= y1-dy1;
        Rd1:=Sqr(x5)+Sqr(y5);
        Kr:=k1+k2*Rd1;
        x8:=dx1+Kr*x5;
        y8:=dy1+Kr*y5;
        x9:=Trunc(x8);            { Reference                    }
        y9:=Trunc(y8);             { in Source Framebuffer       }
        If (x9>0) And (x9<gmx) And (y9>0) And (y9<gmy) Then
            Begin
                dx:=x8-x9;        { Difference                   }
                dy:=y8-y9;
                dx2:=Sqr(dx); dx3:=dx*dx2;
                dy2:=Sqr(dy); dy3:=dy*dy2;
```

14.5 Barrel/Pincushion Correction

```
        Dec(y9); { -1 }
            FramToRGB(FMem[y9]^[x9-1],r00,g00,b00);   { 00  10  20 30 }
            FramToRGB(FMem[y9]^[x9  ],r10,g10,b10);   { 01  11  21 31 }
            FramToRGB(FMem[y9]^[x9+1],r20,g20,b20);   { 02  12  22 32 }
            FramToRGB(FMem[y9]^[x9+2],r30,g30,b30);   { 03  13  23 33 }
            Inc(y9); {  0 }
            FramToRGB(FMem[y9]^[x9-1],r01,g01,b01);
            FramToRGB(FMem[y9]^[x9  ],r11,g11,b11);
            FramToRGB(FMem[y9]^[x9+1],r21,g21,b21);
            FramToRGB(FMem[y9]^[x9+2],r31,g31,b31);
            Inc(y9); { +1 }
            FramToRGB(FMem[y9]^[x9-1],r02,g02,b02);
            FramToRGB(FMem[y9]^[x9  ],r12,g12,b12);
            FramToRGB(FMem[y9]^[x9+1],r22,g22,b22);
            FramToRGB(FMem[y9]^[x9+2],r32,g32,b32);
            Inc(y9); { +2 }
            FramToRGB(FMem[y9]^[x9-1],r03,g03,b03);
            FramToRGB(FMem[y9]^[x9  ],r13,g13,b13);
            FramToRGB(FMem[y9]^[x9+1],r23,g23,b23);
            FramToRGB(FMem[y9]^[x9+2],r33,g33,b33);
            Par3(r00,r10,r20,r30,dx,dx2,dx3,ra);
            Par3(r01,r11,r21,r31,dx,dx2,dx3,rb);
            Par3(r02,r12,r22,r32,dx,dx2,dx3,rc);
            Par3(r03,r13,r23,r33,dx,dx2,dx3,rd);
            Pbr3(ra, rb ,rc ,rd, dy,dy2,dy3,r11);
            Par3(g00,g10,g20,g30,dx,dx2,dx3,ga);
            Par3(g01,g11,g21,g31,dx,dx2,dx3,gb);
            Par3(g02,g12,g22,g32,dx,dx2,dx3,gc);
            Par3(g03,g13,g23,g33,dx,dx2,dx3,gd);
            Pbr3(ga, gb ,gc ,gd, dy,dy2,dy3,g11);
            Par3(b00,b10,b20,b30,dx,dx2,dx3,ba);
            Par3(b01,b11,b21,b31,dx,dx2,dx3,bb);
            Par3(b02,b12,b22,b32,dx,dx2,dx3,bc);
            Par3(b03,b13,b23,b33,dx,dx2,dx3,bd);
            Pbr3(ba, bb ,bc ,bd, dy,dy2,dy3,b11);
            RgbToFram(r11,g11,b11,prgb);
            SetSixel(x1,y1,prgb); { Pixel on Screen }
      End; { If... }
     End;  { x1... }
    End;   { y1... }
End;
```

15. References

[1] http://astronomy.swin.edu.au/~pbourke/colour/bicubic/

[2] G.Hoffmann
Rectification by Photogrammetry
http://docs-hoffmann.de/sans04012001.pdf

[3] G.Hoffmann
Planar Projections
http://docs-hoffmann.de/project18032004.pdf

This document:
http://docs-hoffmann.de/bicubic03042002.pdf

February 03 / 2013:
Conversion from PageMaker to InDesign
May have caused minor layout bugs