Research Documentation

Corel, Inc.
Product Development Department

Title

Mip-Map Interpolator for Image Warping.

Authors

Dmitry Chernichenko (DChernichenko@bublik.com) – theory and experiments

Vladimir Shakh (VShakh@bublik.com) – implementation joint action WarpBrush and Mip-Map Interpolator.

Pavel Polozov - implementation of Mip-Map Interpolator's algorithm.

Dates

2002-2003 years

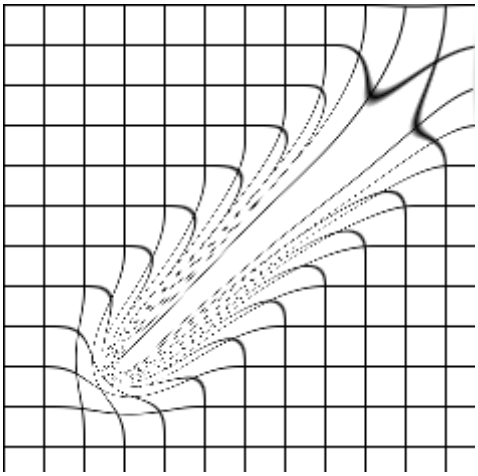
Abstract

Main goal of this research is avoiding aliasing (jaggedness effects) in Warp Brush tool. All existing JASC interpolators are not able to do it, because they are not taking into account of property of image transformation (all of them are isotropic filter). The Mip-Map Interpolator is anisotropic filter and it reduces aliasing.

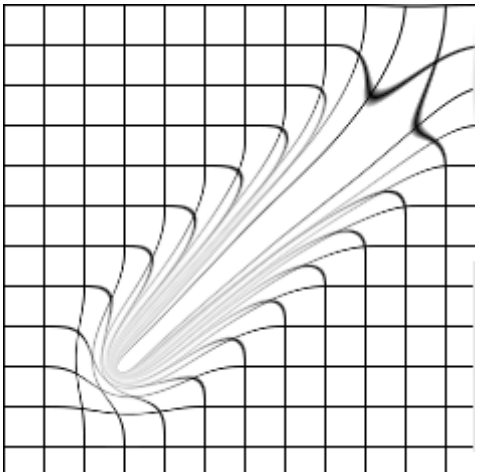
This interpolation scheme can be used for processing multidimensional signals. This scheme isn't taking into account the signal's property but only transformation's property. On the other hand, speed of execution is not depending from transformation's property. Mip-Map interpolator is based on JASC's Bilinear Interpolator.

Relevance to Corel

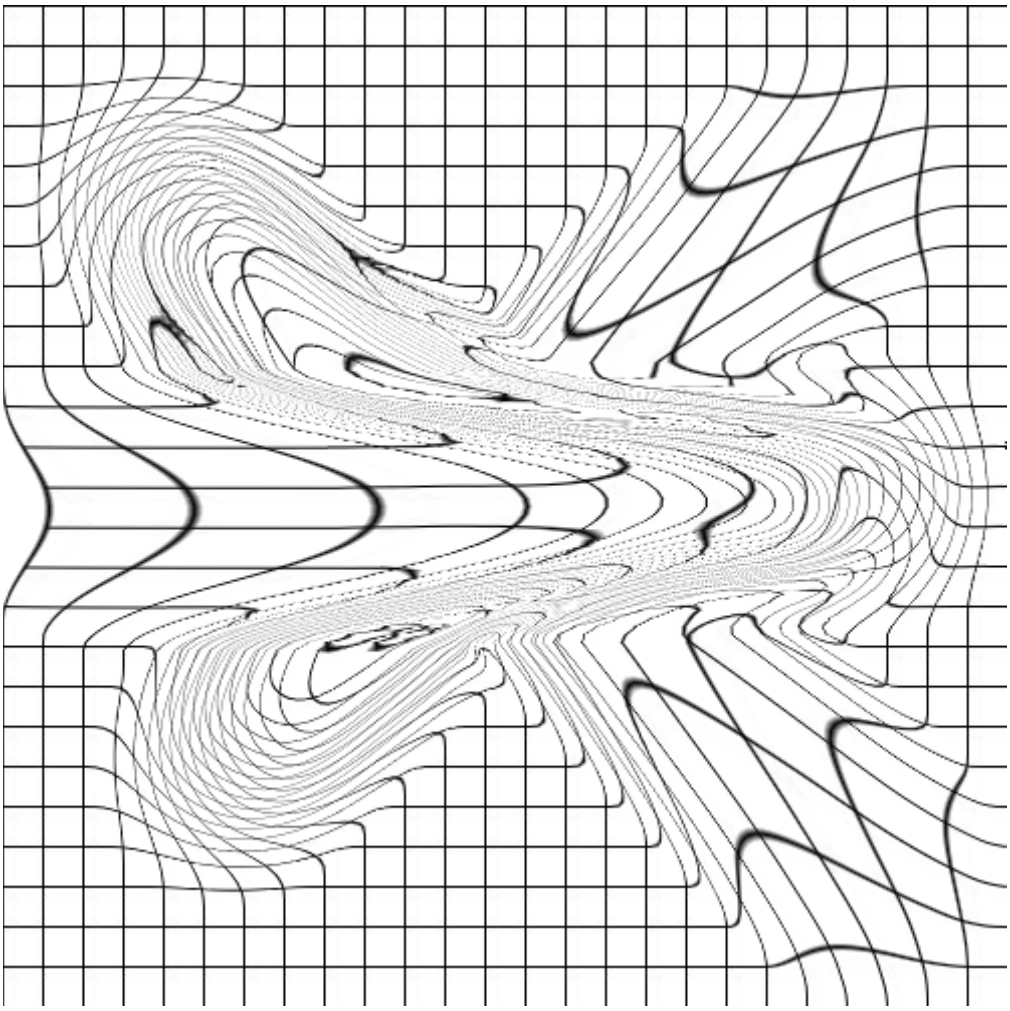
We developed this interpolation scheme, because competitor software (Photoshop Liquid filter) has it and anisotropic filtration gives result more accurately and more visual attractive. Our scheme has several advantages in comparison with Photoshop's: more rapid and more accurate in some cases.



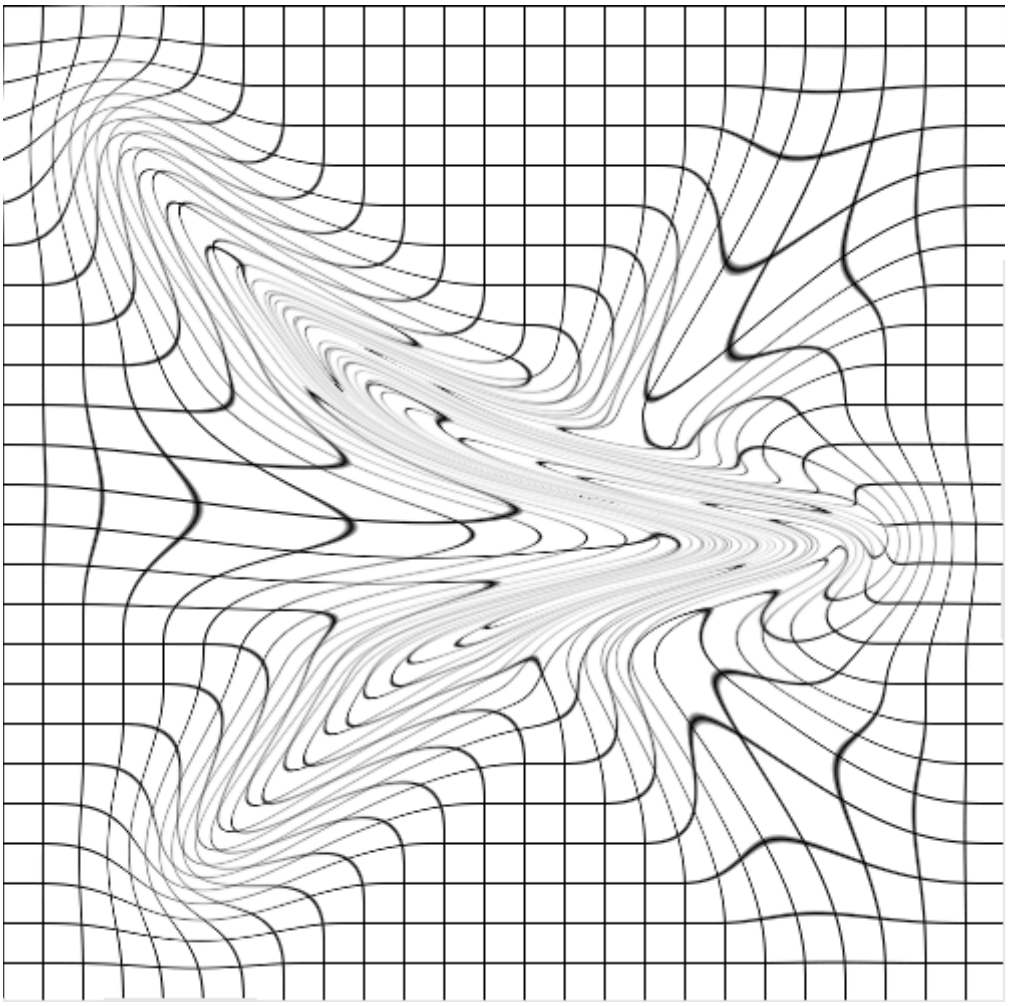
Pic. 1 Image warping with bilinear interpolation (aliasing effect).



Pic. 2 Image warping with Mip-Map interpolation (no aliasing).



Pic. 3. Adobe Photoshop CS interpolation



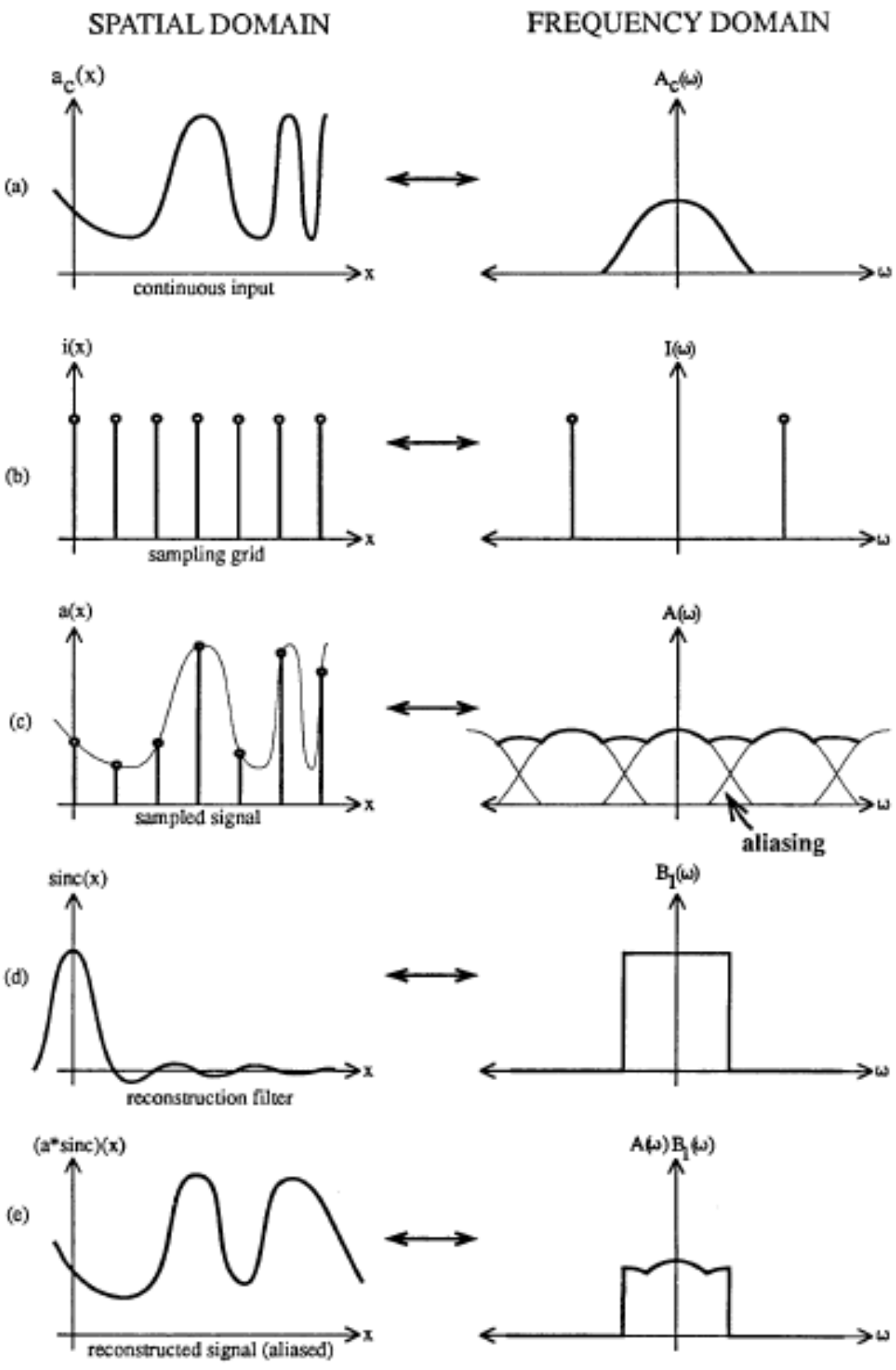
Pic. 4. PSP interpolation

Assumptions

The main assumption is that our transformation is continuous and has smooth derivatives.

Detailed Description

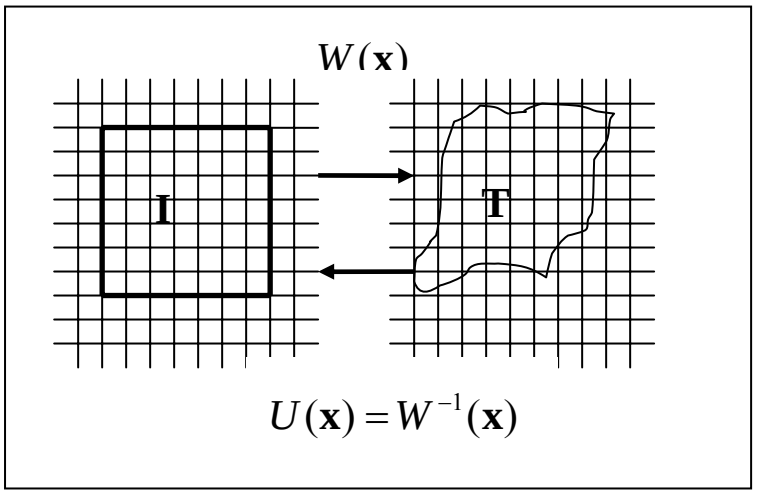
Main goal of this research is avoiding aliasing in Warp Brush tool. Aliasing appears when sampling rate is not enough. If sampling rate less then Nyquist frequency we can see spectrum aliasing. So, if transformation increases sampling rate (compress signal) we have potential possibility of aliasing. In order to decrease aliasing effect we have to filter out the signal using low frequency filter and frequency cut-off must be in coordination with sampling rate changing.



Pic. 5 Demonstration of aliasing effect

Let we have discrete signal $I(\mathbf{x})$, some transformation $W(\mathbf{x})$ and we must find discrete signal $T(\mathbf{x})$ that is transformation of input signal $I(\mathbf{x})$.

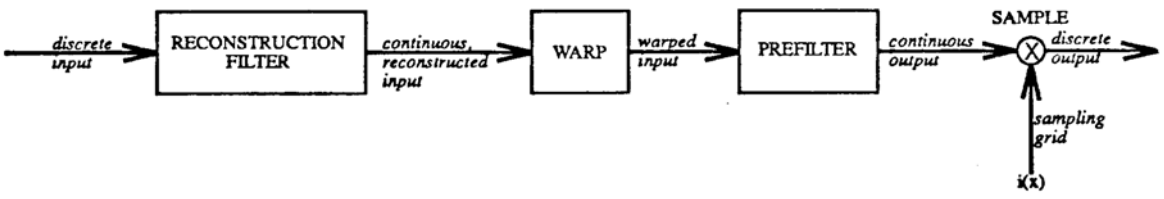
Where: $\mathbf{x} \in R^N, W(\mathbf{x}) : R^N \to R^N$ and has inverse transformation $U(\mathbf{x}) = W(\mathbf{x})^{-1}$.



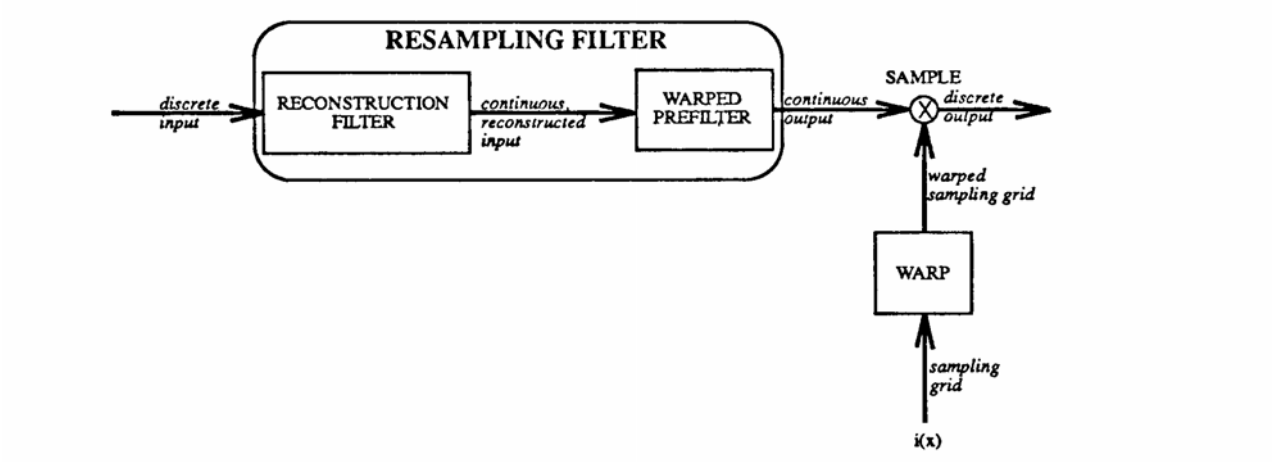
Forward and inverse transformation

So, common scheme for finding signal $T(\mathbf{x})$ is:

- We have input discrete signal $I(\mathbf{x})$
- Reconstruct continuous signal $I^C(\mathbf{x})$ from discrete signal $I(\mathbf{x})$ using reconstruction filter $r(\mathbf{x})$: $I^C(\mathbf{x}) = I(\mathbf{x}) * r(\mathbf{x})$, where * is convolution
- Warp continuous signal $I^C(\mathbf{x})$ using inverse transformation $U(\mathbf{x})$: $T^C(\mathbf{x}) = I^C(U(\mathbf{x}))$
- Filter $T^C(\mathbf{x})$ using low frequency filter $h(\mathbf{x})$: $T_L^C(\mathbf{x}) = T^C(\mathbf{x}) * h(\mathbf{x})$
- Discrete continuous signal $T_L^C(\mathbf{x})$ in order to receive discrete result $T(\mathbf{x})$



Pic. 6. Flow of finding output signal T



Pic. 7. Implementation of signal warping

$$T(\mathbf{x}) = \sum_{\mathbf{i}\in Z^n} T_L^C(\mathbf{x})\delta(\mathbf{x}-\mathbf{i}),$$

$$T_L^C(\mathbf{x}) = T^C(\mathbf{x}) * h(\mathbf{x}) = \int_{R^n} T^C(\mathbf{t})h(\mathbf{x}-\mathbf{t})d\mathbf{t}$$

$$T^C(\mathbf{x}) = I^C(U(\mathbf{x}))$$

$$I^C(\mathbf{x}) = I(\mathbf{x}) * r(\mathbf{x}) = \sum_{\mathbf{t}\in Z^n} I(\mathbf{t})r(\mathbf{x}-\mathbf{t})$$

Combine all equation we receive:

$$T_L^C(\mathbf{x}) = \int_{R^n} I^C(U(\mathbf{t}))h(\mathbf{x}-\mathbf{t})d\mathbf{t} = \int_{R^n} h(\mathbf{x}-\mathbf{t})\left[\sum_{\mathbf{k}\in Z^n} I(\mathbf{k})r(U(\mathbf{t})-\mathbf{k})\right]d\mathbf{t} = \sum_{\mathbf{k}\in Z^n} I(\mathbf{k})\rho(\mathbf{x},\mathbf{k})$$

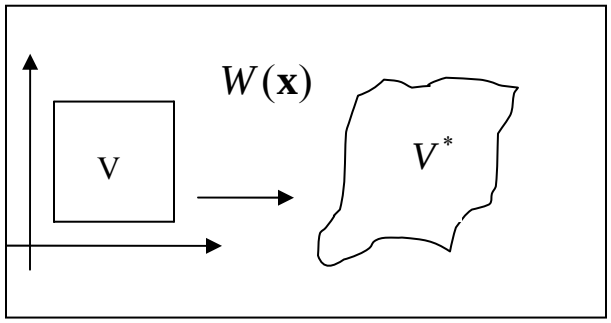
Where $\rho(\mathbf{x},\mathbf{k}) = \int_{R^n} h(\mathbf{x}-\mathbf{t})r(U(\mathbf{t})-\mathbf{k})d\mathbf{t}$ - is resampling filter.

Theory and Principles

Construction of resampling filter

Construction of the resampling filter depends on property of transformation. If transformation magnifies signal – it is simple interpolation filter, if transformation compresses signal – it is low frequency filter.

Let V is some volume in R^N, centered in point $\mathbf{x}_0$, if we apply transformation $W(\mathbf{x})$ to this volume we receive volume V^* - some image of V. Volume of V^* is volume V multiply by $\left|\det(W'(\mathbf{x}_0))\right|$.



We can take Taylor series of transformation $W(\mathbf{x})$. If we restrict Taylor series by linear part we receive:

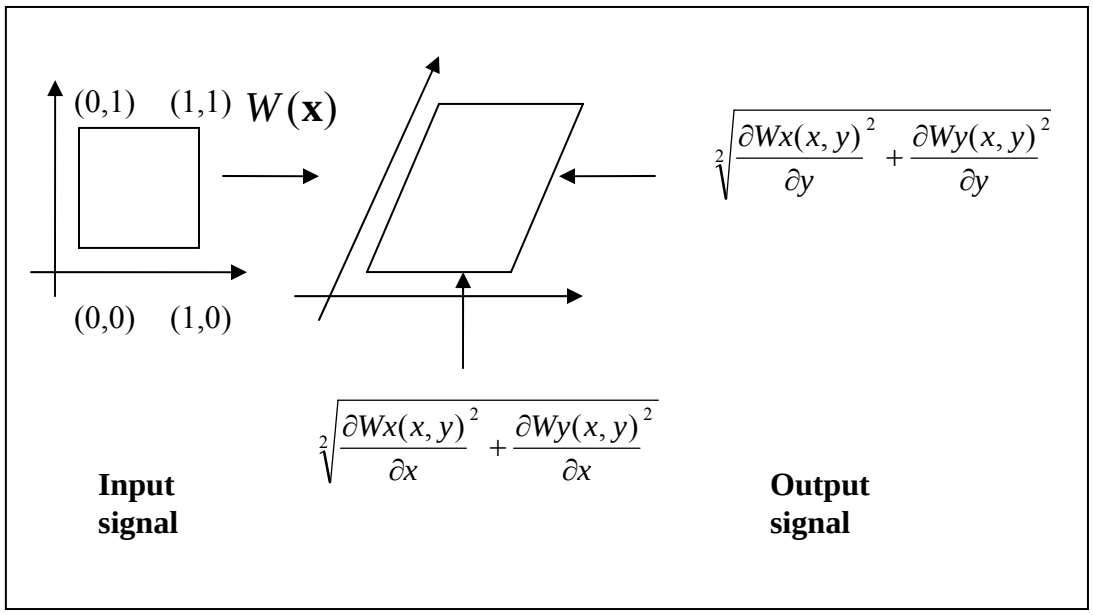
$$W(\mathbf{x}) \approx W(\mathbf{x}_0) + W'(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$$

So we linearize out transformation $W(\mathbf{x})$ in each point. The aliasing effect will be appearing if transformation $W(\mathbf{x})$ compresses volume V in some direction.

If **N**=2 (i.e. we process image), $\mathbf{x} = (x, y)$, $W(\mathbf{x}) : R^2 \to R^2$, $W(\mathbf{x}) = \begin{pmatrix} Wx(x,y) \\ Wy(x,y) \end{pmatrix}$

$$W'(x,y) = \begin{pmatrix} \dfrac{\partial Wx(x,y)}{\partial x} & \dfrac{\partial Wx(x,y)}{\partial y} \\ \dfrac{\partial Wy(x,y)}{\partial x} & \dfrac{\partial Wy(x,y)}{\partial y} \end{pmatrix},$$

So when we apply linearized transformation $W(\mathbf{x})$ to image $I^C(\mathbf{x})$, than we pass in each point $\mathbf{x} = (x, y)$ from canonical basis $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ to $\begin{pmatrix} \dfrac{\partial Wx(x,y)}{\partial x} & \dfrac{\partial Wx(x,y)}{\partial y} \\ \dfrac{\partial Wy(x,y)}{\partial x} & \dfrac{\partial Wy(x,y)}{\partial y} \end{pmatrix}$. Unit square in input coordinate system pass to parallelogram with sides equal $\sqrt{\left(\dfrac{\partial Wx(x,y)}{\partial x}\right)^2 + \left(\dfrac{\partial Wy(x,y)}{\partial x}\right)^2}$ and $\sqrt{\left(\dfrac{\partial Wx(x,y)}{\partial y}\right)^2 + \left(\dfrac{\partial Wy(x,y)}{\partial y}\right)^2}$.



Forward transformation

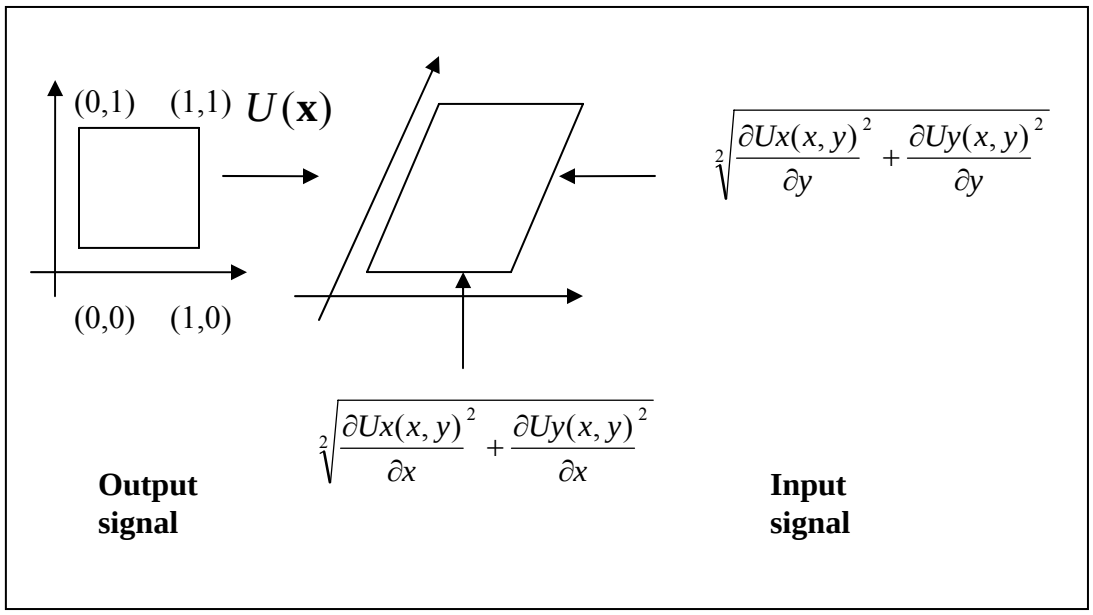
If parallelogram is less than unit square in some direction – we have decimation, if more – magnification.

So our resample filter in each point must be:

$$\rho_{\mathbf{U_0}}(\mathbf{x}_0,\mathbf{k}) = \int_{R^n} h(\mathbf{x} - W_{\mathbf{U_0}}(\mathbf{u}))r(\mathbf{u}-\mathbf{k})\left|J_{\mathbf{U_0}}\right|d\mathbf{u},$$

Where $\mathbf{u} \approx U(\mathbf{x}_0) = \mathbf{u}_0$

Often inverse transformation $U(\mathbf{x}) = W(\mathbf{x})^{-1}$ is more suitable and convenient than forward transformation $W(\mathbf{x})$. All argumentations above may be rewritten concerning transformation $U(\mathbf{x})$. Unit square in output coordinate system pass to parallelogram with sides equal $\sqrt[2]{\frac{\partial Ux(x,y)^2}{\partial x}+\frac{\partial Uy(x,y)^2}{\partial x}}$ and $\sqrt[2]{\frac{\partial Ux(x,y)^2}{\partial y}+\frac{\partial Uy(x,y)^2}{\partial y}}$.



Inverse transformation

If parallelogram is more than unit square in some direction – we have decimation, if less – magnification.

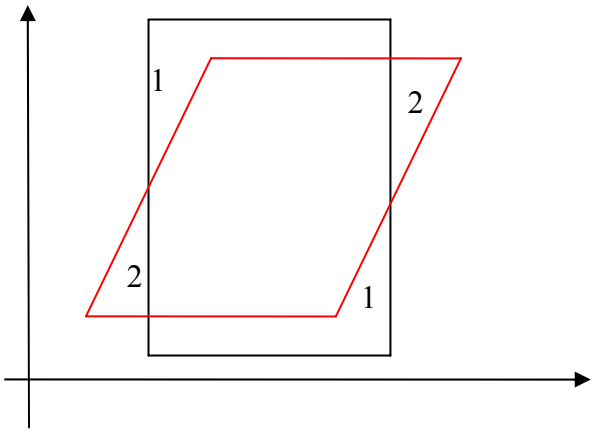
In order to avoid aliasing we must put to result pixel weighted sum of all input pixels that is covered by parallelogram, described be Jacobean of $U(\mathbf{x}) = W(\mathbf{x})^{-1}$ in each output point.

So, in order to construct output image without aliasing effect we must for each output pixel $\mathbf{x}_0$ determinate Jacobean of $U(\mathbf{x}_0) = W(\mathbf{x}_0)^{-1}$, $U'(x_0, y_0) = \begin{pmatrix} \frac{\partial Ux(x_0,y_0)}{\partial x} & \frac{\partial Ux(x_0,y_0)}{\partial y} \\ \frac{\partial Uy(x_0,y_0)}{\partial x} & \frac{\partial Uy(x_0,y_0)}{\partial y} \end{pmatrix}$ and sum all pixels, that covered parallelogram with sides equal $\sqrt[2]{\frac{\partial Ux(x,y)^2}{\partial x}+\frac{\partial Uy(x,y)^2}{\partial x}}$ and $\sqrt[2]{\frac{\partial Ux(x,y)^2}{\partial y}+\frac{\partial Uy(x,y)^2}{\partial y}}$, centered in point $\mathbf{w}_0 = U(\mathbf{x}_0)$.

It is very slow process (especially when parallelogram is not rectangle) and interpolation scheme depends on transformation property. In order to increase execution speed we approximate

parallelogram by rectangle with sides equal $\sqrt[2]{\frac{\partial Ux(x,y)}{\partial x}^2 + \frac{\partial Uy(x,y)}{\partial x}^2}$,

$\sqrt[2]{\frac{\partial Ux(x,y)}{\partial y}^2 + \frac{\partial Uy(x,y)}{\partial y}^2}$.



Red parallelogram is Jacobean of $U(\mathbf{x}_0) = W(\mathbf{x}_0)^{-1}$, black rectangle is approximation.

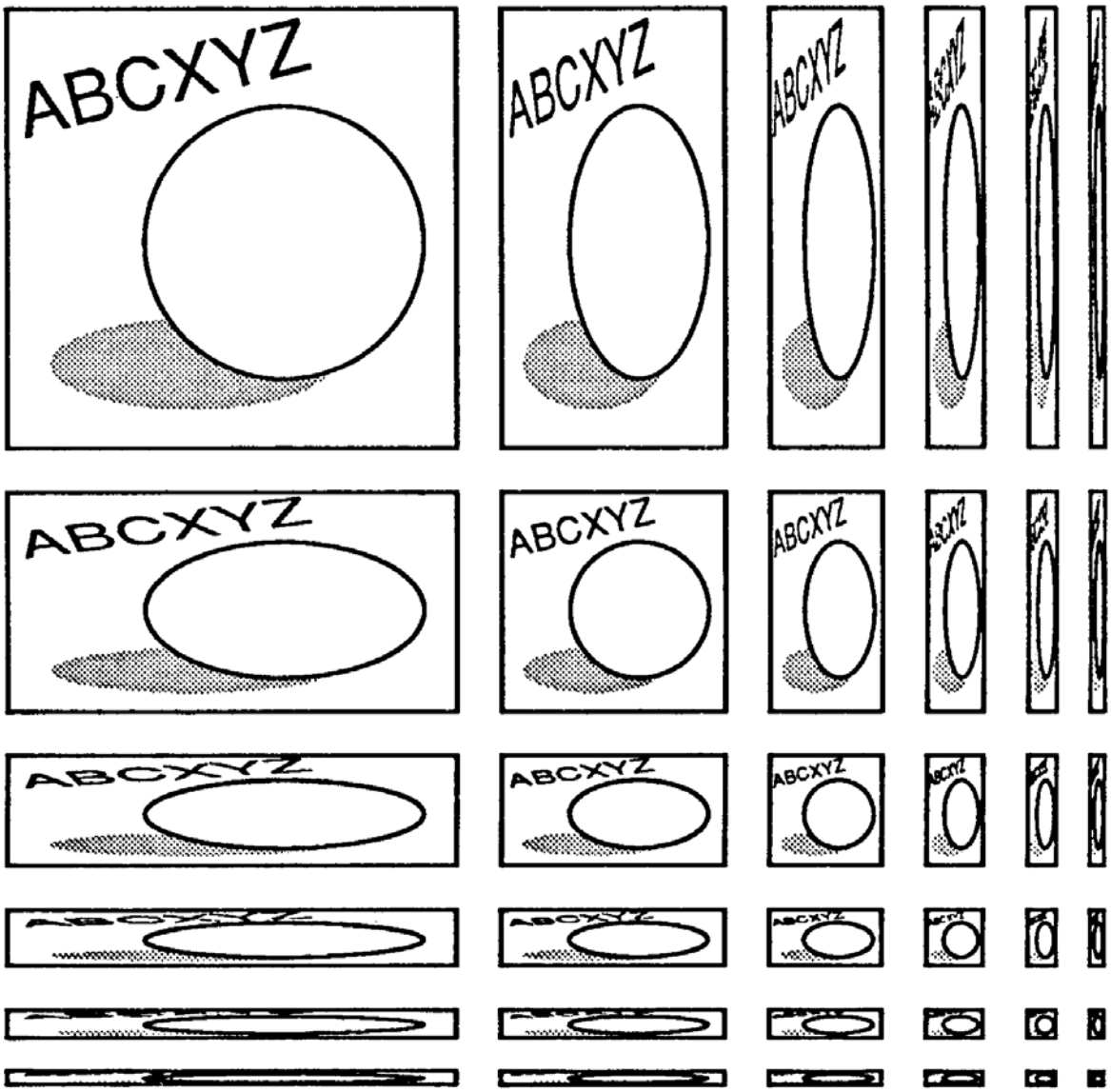
Of course, there are a few errors. Errors can be 2 types:

1. We integrate areas outside parallelogram.
2. We don't integrate areas inside parallelogram

The best case is when parallelogram is rectangle. The worst case is when parallelogram is narrow and rotated by 45 degree.

Image pyramid

In order to increase execution speed, we construct image pyramid. Pyramid consists of scaled on each dimensions input image.



Pic. 8. Image pyramid.

For construction image pyramid we use low frequency filtration and decimation by 2 times.

Let we have input image $I(x,y)$ and image has size [X x Y]. We construct image pyramid that is matrix by $[floor(\log_2(X)-1)\ x\ floor(\log_2(Y)-1)]$. Each cell (i,j) is image, which has size $[ceil(X/2^j)xceil(Y/2^i)]$. i.e. it is filtered input image $I(x,y)$ by low frequency filter $H_{(i,j)}$ and compressed by $(2^j,2^i)$ times in each dimension. Our filter $H_{(i,j)}$ is separable, so $H_{(i,j)}(x,y)=h_i(x)h_j(y)$, filter $h_j(x)$ is $\otimes_{k=1}^{j} 2^{-k/2}h(2^k x)$,

Here $h(x)$ is indicator function: $h(x)=I\{x\in(0,1]\}=\begin{cases}0, x\notin(0,1]\\ 1, x\in(0,1]\end{cases}$

1. Construction image pyramid
2. Calculate $N = floor(\log_2(X) - 1)$, $M = floor(\log_2(Y) - 1)$
3. Create matrix P, P has size MxN
4. $P_{1,1} = I(x, y)$
5. For first row (i=1)

   for columns $j = 2 : N$

   $P_{1,j} = \left(P_{1,j-1} * h(x)\right)\downarrow (2)$ along X coordinate
6. For rows $i = 2 : M$

   $P_{i,1} = \left(P_{i-1,1} * h(y)\right)\downarrow (2)$ along Y coordinate

   for columns $j = 2 : N$

   $P_{i,j} = \left(P_{i,j-1} * h(x)\right)\downarrow (2)$ along X coordinate

Here

Filter $h(x)$ has mask $\begin{bmatrix} 0.5 & 0.5 \end{bmatrix}$,

Filter $h(y)$ has mask $\begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix}$,

Operation $I\downarrow(2)$ along X coordinate is down sampling image I along coordinate X by 2 times, i.e. if $I = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$ then

$$I\downarrow(2) \text{ along X coordinate} = \begin{pmatrix} a_{11} & a_{13} \\ a_{21} & a_{23} \\ a_{31} & a_{33} \\ a_{41} & a_{43} \end{pmatrix},$$

$$I\downarrow(2) \text{ along Y coordinate} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}.$$

In PSP we used RESIZE command using Weighted Average Interpolator with “Repeat” boundary condition for image pyramid construction.

When image pyramid is constructed, we can use the following interpolation procedure:

For each point in output image T, $\mathbf{x}_0 = (x_0, y_0)$

1. Calculate Jacobean of $U(\mathbf{x}_0) = W(\mathbf{x}_0)^{-1}$,

$$U'(x_0, y_0) = \begin{pmatrix} \frac{\partial Ux(x_0, y_0)}{\partial x} & \frac{\partial Ux(x_0, y_0)}{\partial y} \\ \frac{\partial Uy(x_0, y_0)}{\partial x} & \frac{\partial Uy(x_0, y_0)}{\partial y} \end{pmatrix}$$

2. Calculate

$$LoD\_X(x_0, y_0) = \log_2\left(\sqrt[2]{\frac{\partial Ux(x_0, y_0)}{\partial x}^2 + \frac{\partial Uy(x_0, y_0)}{\partial x}^2}\right)$$

$$LoD\_Y(x_0, y_0) = \log_2\left(\sqrt[2]{\frac{\partial Ux(x_0, y_0)}{\partial y}^2 + \frac{\partial Uy(x_0, y_0)}{\partial y}^2}\right)$$

3. Calculate levels of details

$$S_x = floor(LoD\_X(x_0, y_0))$$

$$S_y = floor(LoD\_Y(x_0, y_0))$$

4. Then we extract 4 points from image pyramid

$$P_1 = P_{S_y+1, S_x+1}\left(\frac{x_0}{2^{S_x}}, \frac{y_0}{2^{S_y}}\right)$$

$$P_2 = P_{S_y+1, S_x+2}\left(\frac{x_0}{2^{S_x+1}}, \frac{y_0}{2^{S_y}}\right)$$

$$P_3 = P_{S_y+2, S_x+1}\left(\frac{x_0}{2^{S_x}}, \frac{y_0}{2^{S_y+1}}\right)$$

$$P_4 = P_{S_y+2, S_x+2}\left(\frac{x_0}{2^{S_x+1}}, \frac{y_0}{2^{S_y+1}}\right)$$

The bilinear interpolation is used for these points extraction.

5. Then make bilinear interpolation:

$$T(x_0, y_0) = P_1 * K_1 * K_2 + P_2 * (1 - K_1) * K_2 + P_3 * K_1 * (1 - K_2) + P_4 * (1 - K_1) * (1 - K_2)$$

Where

$$K_1 = 1 - LoD\_X(x_0, y_0) + S_x$$

$$K_2 = 1 - LoD\_Y(x_0, y_0) + S_y$$

If S_x or S_y less than 0, it means, that our transformation magnified image and we use simple bilinear interpolation.

So, our interpolation scheme takes account of transformation's property, but speed of execution is not depending from transformation's property.

The speed of execution is about 5 times slowly than simple bilinear interpolation. So we need addition memory for storing image pyramid (less than 4*(X*Y)).

Mip-Map Interpolator and Displacement Map.

If we don't know forward and inverse transformation (for example we have only displacement map that determinates image transformation indirectly), we can use the following equations:

Let's we have displacement map $\mathbf{D}(\mathbf{x})$

$$\mathbf{D}(\mathbf{x}) = \begin{pmatrix} Dx(x, y) \\ Dy(x, y) \end{pmatrix}$$

and $T(x, y) = I(x + Dx(x, y), y + Dy(x, y))$.

Where $T(x, y)$ is an output image, $I(x, y)$ is an input image.

So we can calculate $U'(x_0, y_0) = \begin{pmatrix} \frac{\partial Ux(x_0, y_0)}{\partial x} & \frac{\partial Ux(x_0, y_0)}{\partial y} \\ \frac{\partial Uy(x_0, y_0)}{\partial x} & \frac{\partial Uy(x_0, y_0)}{\partial y} \end{pmatrix}$ numerically as first order finite difference.

$$\frac{\partial Ux(x_0, y_0)}{\partial x} = 1 + \frac{Dx(x_0 + 1, y_0) + Dx(x_0 - 1, y_0)}{2}$$

$$\frac{\partial Uy(x_0, y_0)}{\partial x} = \frac{Dy(x_0 + 1, y_0) + Dy(x_0 - 1, y_0)}{2}$$

$$\frac{\partial Ux(x_0, y_0)}{\partial y} = \frac{Dx(x_0, y_0 + 1) + Dx(x_0, y_0 - 1)}{2}$$

$$\frac{\partial Uy(x_0, y_0)}{\partial y} = 1 + \frac{Dy(x_0, y_0 + 1) + Dy(x_0, y_0 - 1)}{2}$$

On the boundary of displacement map we assume that derivative is constant (but not zero) in order to avoid of discontinuity.

Interfaces

To implement the MipMap interpolation, we have written class CMipMapResampler, which is contained in files MipMapInterpolator.h/cpp. The class in section `public` comprises the following functions:

CMipMapResampler ( ) – default constructor.

CMipMapResampler (CRowIterator& rkImage, EEdgeMode eEdgeMode = keEdgeBackground, DWORD dwRGBA = 0, int iLODX256 = -1, int iLODY256 = -1), which is the constructor with parameters described below in function Attach( );

~CMipMapResampler ( ) – destructor.

void Attach (CRowIterator& rkImage, EEdgeMode eEdgeMode = keEdgeBackground, DWORD dwRGBA = 0, CStatusControl* pStatusControl = NULL, int iLODX256 = -1, int iLODY256 = -1) is the function for initializing an object of the MipMap interpolator. The iterator of the source image and the interpolation mode are set by the parameters which are described in more detail below. During initialization, an array of averaged representations of the source image is created.

bool IsAttached ( ) – this checks the presence of the link between the interpolator object and the iterator of the source image.

void Detach ( ) - this detaches the object from the iterator of the source image, clears the memory for averaged images, and resets all the parameters.

CErrorAutoThrow GetLine (BYTE* pbyDest, POINT* ppt256, POINT* pptLoad256, int iCount) - this interpolates a manifold of the points of the result image over the points of the source image. The coordinates of the points in the result image and corresponding coordinates of the points in the source image are set by the parameters which are described in more detail below.

The main functions of the class are GetLine ( ) and Attach ( ), which are described in more detail below.

Function GetLine ( )

The interpolation of the manifold of the points of the result image over the points of the source image is performed by the function

CErrorAutoThrow GetLine (BYTE* pbyDest, POINT* ppt256, POINT* pptLoad256, int iCount);

Parameters of the function:

BYTE* pbyDest – indicates the array of output data for placing the interpolation results. The interpolation is performed channel-by-channel, the number of channels and the number of bytes per channel being defined by the iterator of the result image. (Currently, only one-byte channels can be used, while an upgrade of some `private` –functions of the class is necessary). To this effect, it is convenient to use function

pbyDest = JGDI_GetPixelRow(oDestIter, iy);

CrowIterator oDestIter – the result iterator,

int iy – a line of the result image.

POINT* ppt256 – indicates the array of coordinates of the source image points from which the result points are computed. Each coordinate pair (X, Y) is represented by integers with a sign and fixed point so that the lower 8 digits of the integers represent the fractional part, that is, all the coordinates are multiplied by 256:

DestImage[i] = SrcImage(ppt256[i].x/256, ppt256[i].y/256);

The values ppt256 at point (x, y) are computed via coordinate transformation functions or via the deformation map as follows:

ppt256.x = round(256* Ux(x, y)) = round(256*(DX(x, y) + x));

ppt256.y = round(256* Uy(x, y)) = round(256*(DY(x, y) + y));

POINT* pptLoad256 – indicates the array of scale factors for each interpolated point. Each pair of the scale factors (ptLoad256.x, ptLoad256.y) is represented by positive integers with fixed point so that the lower 8 digits of the integers represent the fractional part, that is, all the scale factors are multiplied by 256:

The scale factors are computed via derivatives dUx/dx, dUx/dy, dUy/dx, dUy/dy of the coordinate transformation functions:

ptLoad256.x= round(256*(log2(sqrt((dUx/dx)^2 + (dUx/dy)^2))));

ptLoad256.y= round(256*(log2(sqrt((dUy/dx)^2 + (dUy/dy)^2))));

log2 ( ) – hereafter this is logarithm with base 2.

or via derivatives dDX/dx, dDX/dy, dDY/dx, dDY/dy of the deformation magnitudes:

ptLoad256.x= round(256*(log2(sqrt((1 + dDX/dx)^2 + (dDX/dy)^2))));

ptLoad256.y= round(256*(log2(sqrt((dDY/dx)^2 + (1 + dDY/dy)^2))));

In practice, the computations are carried out according to the following scheme:

double dK = 128.0/log(2.0);

```
ptLoad256.x = ptLoad256.y = 0;
if (ptDx.x || ptDx.y)
{
        double tmp = sqr(512 + ptDx.x) + sqr(ptDx.y);
        if (tmp > 262144.0)
                ptLoad256.x = round(dK*log(tmp)) - 2304;
}
if (ptDy.x || ptDy.y)
{
        double tmp = sqr(512 + ptDy.y) + sqr(ptDy.x);
        if (tmp > 262144.0)
                ptLoad256.y = round(dK*log(tmp)) - 2304;
}
```

POINT ptDx, ptDy – are computed via the differences in deformations of two points of the result image along the vertical (Y) and horizontal (X), respectively:

ptDx.x = 256*(DX(x + 1, y) - DX(x - 1, y)) = 256*(Ux(x + 1, y) - Ux(x - 1, y)) - 512 = ppt256(x + 1, y).x - ppt256(x - 1, y).x - 512;

ptDx.y = 256*(DX(x, y + 1) - DX(x, y - 1)) = 256*(Ux(x, y + 1) - Ux(x, y - 1)) = ppt256(x, y + 1).x - ppt256(x, y - 1).x;

ptDy.x = 256*(DY(x + 1, y) - DY(x - 1, y)) = 256*(Uy(x + 1, y) - Uy(x - 1, y)) = ppt256(x + 1, y).y - ppt256(x - 1, y).y;

ptDy.y = 256*(DY(x, y + 1) - DY(x, y - 1)) = 256*(Uy(x, y + 1) - Uy(x, y - 1)) - 512 = ppt256(x, y + 1).y - ppt256(x, y - 1).y - 512;

In fact, this corresponds to

```
ptLoad256(x, y) =
  round(256*(
        log2(sqrt(
            ((ppt256(x + 1, y) - ppt256(x – 1, y))/512)^2 +
            ((ppt256(x, y + 1) - ppt256(x, y - 1))/512)^2
          ))
));
```

For the edge points in the image, when, for example, x + 1 extends beyond the image and the deformation at this point is not defined, the difference is computed in a different manner:

ptDx.x = 512*(DX(x, y) - DX(x - 1, y)) = 512*(Ux(x, y) - Ux(x - 1, y)) - 512 = 2*(ppt256(x, y).x - ppt256(x - 1, y).x) –512;

ptDy.x = 512*(DY(x, y) - DY(x - 1, y)) = 512*(Uy(x, y) - Uy(x - 1, y)) = 2*(ppt256(x, y).y - ppt256(x - 1, y).y);

For other edge points, the procedure is similar. For x – 1:

ptDx.x = 512*(DX(x + 1, y) - DX(x, y)) = 512*(Ux(x + 1, y) - Ux(x, y)) - 512 = 2*(ppt256(x + 1, y).x - ppt256(x, y).x) - 512;

ptDy.x = 512*(DY(x + 1, y) - DY(x, y)) = 512*(Uy(x + 1, y) - Uy(x, y)) = 2*(ppt256(x + 1, y).y - ppt256(x, y).y);

For y + 1:

ptDx.y = 512*(DX(x, y) - DX(x, y - 1)) = 512*(Ux(x, y) - Ux(x, y - 1)) = 2*(ppt256(x, y).x - ppt256(x, y - 1).x);

ptDy.y = 512*(DY(x, y) - DY(x, y - 1)) = 512*(Uy(x, y) - Uy(x, y - 1)) - 512 = 2*(ppt256(x, y).y - ppt256(x, y - 1).y) - 512;

For y - 1:

ptDx.y = 512*(DX(x, y + 1) - DX(x, y)) = 512*(Ux(x, y + 1) - Ux(x, y)) = 2*(ppt256(x, y + 1).x - ppt256(x, y).x);

ptDy.y = 512*(DY(x, y + 1) - DY(x, y)) = 512*(Uy(x, y + 1) - Uy(x, y)) - 512 = 2*(ppt256(x, y + 1).y - ppt256(x, y).y) - 512;

int iCount – is the number of interpolated points.

MipMap interpolation mechanism and Attach ( ) function

The essence of the MipMap interpolation mechanism is as follows.

The result value is calculated by bilinear interpolation between four values, each of which is calculated allowing for the scale factors over averaged representations of the source image. First, we determine from which averaged representations the result should be calculated:

LODX = ptLoad256.x >> 8 ;

LODY = ptLoad256.y >> 8 ;

Values LODX and LODY mean that the following representations should be used:

Image1 – the source image averaged over a window of size (2^LODX)*(2^LODY);

Image2 – the source image averaged over a window of size (2^(LODX + 1))*(2^LODY);

Image3 – the source image averaged over a window of size (2^LODX)*(2^(LODY + 1));

Image4 – the source image averaged over a window of size (2^(LODX + 1))*(2^(LODY + 1));

The sizes of the averaging windows are always multiples of a power of 2 and the averaging at all the levels is performed sequentially: first, by a factor of 2, then, applying the same averaging procedure to the already averaged image, we obtain averaging by a factor of 4, and so on along each of the coordinates X and Y. At each stage, the size of the averaged image also reduces by a factor of 2. If the size of the averaged image is odd, the image is complemented to the nearest even size by reiterating the last pixel in the image. For example, for a source image of size 256*256 pixels, we shall obtain the following averaged images (the first and second numbers are the horizontal and vertical sizes of the averaging window, respectively): 1:1 (this is the source image without averaging, which is also used for calculations), 2:1, 4:1, 8:1, 16:1, 32:1, 64:1, 128:1, 256:1, 1:2; 2:2, 4:2, 8:2, 16:2, 32:2, 64:2, 128:2, 256:2, 1:4, 2:4, 4:4, ….., 128:256, 256:256. Altogether, there will be 64 images, the size of the last one being (256:256) – 1*1 pixels. All these images are stored in a 2-D matrix:

CRowIterator** m_pMipData;

m_pMipData[i] [j] – is the averaged representation corresponding to (2^j):(2^i).

i = 0.. MaxLODX - 1, j = 0.. MaxLODY - 1;

MaxLODX = (int)ceil(log2(SrcImage.GetWidth()));

MaxLODY = (int)ceil(log2(SrcImage.GetHeight()));

In each averaged representation, the consistence of coordinates should be preserved. This is illustrated by an example. Let the source image be of horizontal size 256 pixels, i.e., the range of the X coordinate equals 0..255. In the representation with the averaging window of size 2 along coordinate X, the range of the X-coordinate variation will be 0..127. In this case, to extract a point with coordinate X with respect to the source image, we should address to coordinate

X*127.0/255.0 in the averaged image and, in a similar manner, to coordinate Y. In a general case, when addressing to the averaged image, the coordinates are corrected by the equations

Xcor(i) = X*(2^(MaxLODX - i) - 1)/(2^ MaxLODX - 1);

Ycor(j) = Y*(2^(MaxLODY - i) - 1)/(2^ MaxLODY - 1);

These calculations are fairly exactly approximated by simpler expressions (X = pt256.x/256.0; Y = pt256.y/256.0):

```
int bx = (pt256.x + (pt256.x >> MaxLODX)) >> MaxLODX;
int by = (pt256.y + (pt256.y >> MaxLODY)) >> MaxLODY;
int ax = pt256.x + bx;
int ay = pt256.y + by;
int iX256cor0 = (ax >> i) - bx ;
int iX256cor1 = (ax >> (i + 1)) - bx ;
int iY256cor0 = (ay >> j) - by ;
```

```
int iY256cor1 = (ay >> (j + 1)) - by;
```

Thereby four points are bilinearly interpolated:

```
Pixel1 = Image1(iX256cor0, iY256cor0);
Pixel2 = Image2(iX256cor1, iY256cor0);
Pixel3 = Image3(iX256cor0, iY256cor1);
Pixel4 = Image4(iX256cor1, iY256cor1);
```

Now, it remains to accomplish bilinear interpolation of these points:

```
DestImage(pt256, ptLoad256) = iK1*Pixel1 + iK2*Pixel2 + iK3*Pixel3 + iK4*Pixel4) >> 16;
```

with coefficients:

```
int iK1 = ( 256 - iAlphaX ) * ( 256 - iAlphaY ) ;
int iK2 = ( 256 - iAlphaY ) * iAlphaX ;
int iK3 = ( 256 - iAlphaX ) * iAlphaY ;
int iK4 = iAlphaX  * iAlphaY ;
```

где iAlphaX, iAlphaY – lower 8 digits of ptLoad256:

```
int  iAlphaX = ptLoad256.x & 255 ;
int  iAlphaY = ptLoad256.y & 255 ;
```

A specific feature of calculation of the averaged images in the keEdgeRepeat mode is the following: Two extreme lines at j > 0 or two extreme columns at i > 0 are copied into each new averaging level (i, j) from levels (i, 0) or (0, j), respectively.

The averaged images are formed at a stage of initializing an object of the MipMap interpolator in the function

```
void Attach( CRowIterator& rkImage, EedgeMode eEdgeMode = keEdgeBackground, DWORD dwRGBA = 0, CStatusControl* pStatusControl = NULL, int iLODX256 = -1, int iLODY256 = -1 ) ;
```

Parameters of the function are the following:

CRowIterator& rkImage – the iterator of the source image. The number of channels and the number of bytes per channel should be the same as in the iterator of the result image (in the GetLine function). (Currently, only one-byte channels can be used, while an upgrade of some `private` –functions of the class is necessary).

EedgeMode eEdgeMode – the mode for processing the points beyond the boundary of the source image.

DWORD dwRGBA – the color for points located beyond the boundary of the source image. This is used only in the eEdgeMode = keEdgeBackground mode.

CStatusControl* pStatusControl – for organization of Status control, since initializing the MipMap interpolator may take significant time.

int iLODX256, int iLODY256 – the maximum values of the scale factors in the pptLoad256 array. By setting these values the initialization time may be reduced due to computation of a smaller number of the averaged images. This will provide appreciable speed-up at small values of iLODX256, iLODY256. With the default parameters, the whole array of the averaged images is calculated.

Limitations

Suggestions for a User Interface

Research Software

Possible Uses

Although specific uses for the research were certainly identified before the project was started, it is possible that during the course of the work, additional applications may have been uncovered. If so, these should be documented.

Possible Extensions

Similarly, at the end of the project, extensions to the technology may have been found along the way that, due to time or budget constraints were not explored. A record of these extensions should be made to "bookmark" possible continuation efforts in the future.

Patents

The research work must fully conform to Corel's patent policies.

Bibliography

A list should be made of research papers, technical publications, books, URLs, etc. that are relevant to the work done or that provide useful background information. A description should be included indicating how this particular research project compares to the cited references (e.g. how it is unique or novel, how it is better, faster, more efficient, etc.).

1. Fundamentals of Texture Mapping and Image Warping. Master's Thesis. Paul S. Heckbert. Dept. of Electrical Engineering and Computer Science University of California, Berkeley. 1989.

2. Run-Time MIP-Map Filtering. Andrew Flavell. Gamasutra.. 1998. http://www.gamasutra.com/features/19981211/flavell_01.htm