Fast Box Filter with Subpixel Accuracy

Siniša Petrić, SigmaPi Design

January 21, 2017

Abstract

Box filter is simple and well known technique for image resizing. This technical paper describes an algorithm for image downscaling (shrinking) using box filter with subpixel accuracy, based on color space rescaling, which avoids floating point arithmetics. All operations in accompaniment C++ code are completely integer based, which significantly improves image shrinking speed and quality.

1 Introduction

Box filter is the simplest image resizing technique involving pixel interpolation. It takes a set of pixels, calculates mean value and stores this value as an output pixel. Considering 1D **single-channel 8-bit** "image" simplification, if we have two images I_1(input) and I_2 (output) of the length: $L_1 = 9$ and $L_2 = 4$, shrinking image $L_1$to L_2involves calculating length ratio:

$$w = \frac{L_1}{L_2} = 2,25$$

This ratio tells us that each pixel in output image will be constructed from two and quarter pixels from input image. So, the first pixel in output image will be:

$$p_2(1) = \frac{1 \cdot p_1(1) + 1 \cdot p_1(2) + 0,25 \cdot p_1(3)}{2,25}$$

Generally, formula for output pixel can be expressed as affine linear combination:

$$p_2 = \sum_{i=1}^{ceil(w)} \alpha_i p_1$$

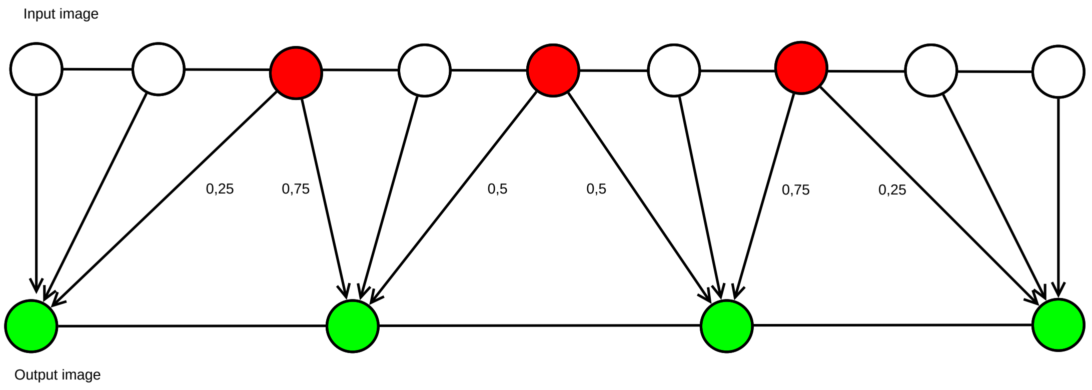
where:

$$\sum_{i=1}^{ceil(w)} \alpha_i = 1 \, and \, \alpha_i = \frac{a_i}{w}$$

or simply:

$$p_2 = \frac{1}{w} \sum_{i=1}^{ceil(w)} a_i p_1$$

where $ceil(w)$ is the lowest integer value higher or equal to w, and $a_i$is **contribution factor** for each pixel from input image. contribution factor values depend on desired interpolation (Bilinear, Bicubic, Lanczos, etc...). We will stick to our modified box filter. This process can be shown as diagram:



White circles the diagram above, denote input image pixels with contribution factor $= 1$, while red circles denote pixels with some real number fraction (0,1). Green circles denote output image pixels calculated using above formulas.

2 Algorithm

To avoid floating point arithmetics and still obtain subpixel accuracy, we will rescale 8-bit value of resulting pixel using mapping:$[0, 255] \mapsto [0, 2^{30} - 4]$, which gives as much better subpixel resolution. As we will use **unsigned long** type (4 bytes) for storing pixel summation, we can even extend upper boundary using bigger exponent value. To accommodate "richer" color space, length ratio is mapped to constant value: $w \mapsto 2^{22}$. As any value of w is mapped to constant value, for higher ratio values, less resolution is available for each contribution factor. We can use modification that uses higher exponent in $w \mapsto 2^n$mapping for bigger w, but for sake of simplicity, we will retain fixed exponent and give a name to our normalization value: $2^{22} = 4194304 = norm$. To construct the output pixel we need to calculate contribution factor in new color space. First, we will divide our norm with length ratio and extract integer (unsigned long) fraction and initialize summation variable to norm value:

$$\begin{aligned} pixC &= Ulong(\frac{norm}{w}) \\ sumC &= norm \end{aligned}$$

Now we can calculate the first output pixel from our example:

$$\begin{aligned} p_2^{22}(1) &= pixC \cdot p_1(1) + pixC \cdot p_1(2) + (sumC - 2 \cdot pixC) \cdot p_1(3) \\ p_2(1) &= p_2^{22}(1) \gg 22 \end{aligned}$$

where operator $\gg$ stands for right shift operation. After summation is done, the output pixel must be scaled back to it's normal 8-bit value. For the next output pixel $p_2(2)$ we need to carry over last difference between $pixC$ and last contribution factor ($sumC - 2 \cdot pixC$), and so on. Here is the complete algorithm for 1D image: When dealing with 2D images, we need to perform box filtering

```
norm = 4194304
pixC = Ulong(norm/w)
sumC = norm
outVal = 0, Xi =0, Xo = 0
// start of input image pixel loop //
loop while Xi < InpImage.Width
outX = false
Pin = Ulong(InpImage(Xi))
if (sumC > pixC) then
        outVal += Pin * pixC
        sumContribX -= pixC
        goto loop
endif
outVal += Pin * sumC
//scale back and write output pixel
OutImage(Xo) = outVal>>22, Xo += 1
outX=true
// contribution factor for next pixel block //
sumC = pixC - sumC
outVal = Pin * sumC
sumC = norm - sumC
if (Xo >= OutputImage.Width) then exit loop
goto loop
if (!outX) then OutImage(Xo) = outVal>>22
```

Algorithm 1: *1D image box filter*

in both x and y direction. We can do it in two steps, or as suggested in this article in a single step using additional memory space for y-cumulative, which consists of one array of size $ImgeOutput.Width$. Without further pestering with 2D version of algorithm, here is complete source code for 2D three-channel 8-bit RGB image shrinking.

3 C++ code

```
void fbfDownScale(spImage<ptRGB> *imgIn, spImage<ptRGB> *imgOut)
{
unsigned long norm = 4194304;
unsigned long shift = 22;
```

```
float fX = (float)imgIn->Width() / (float)imgOut->Width();
float fY = (float)imgIn->Height() / (float)imgOut->Height();
unsigned long pixContribX = (int)((float)norm / fX);
unsigned long pixContribY = (int)((float)norm / fY);
unsigned long sumContribX, sumContribY, restContribY, yContrib;
unsigned long outVal[3];
unsigned long *cumulY[3];
unsigned int x, y, yi, xi;
bool outX, outY, outLastRow;
for (int i = 0; i < 3; i++)
 {
 cumulY[i] = new unsigned long [imgOut->Width()];
 memset(cumulY[i], 0, imgOut->Width() * sizeof(unsigned long));
 }
y = 0;
sumContribY = norm - pixContribY;
yContrib = pixContribY;
restContribY = 0;
outY = false;
outLastRow = false;
for (yi = 0; yi < imgIn->Height(); yi++)
 { // start yi loop
 x = 0;
 outVal[0] = outVal[1] = outVal[2] = 0;
 sumContribX = norm;
 for (xi = 0; xi < imgIn->Width(); xi++)
  {  // start xi loop
  outX = false;
  if (sumContribX > pixContribX)
   {
   outVal[0] += (unsigned long)imgIn->Pixel(yi, xi).ch[0] * pixContribX;
   outVal[1] += (unsigned long)imgIn->Pixel(yi, xi).ch[1] * pixContribX;
   outVal[2] += (unsigned long)imgIn->Pixel(yi, xi).ch[2] * pixContribX;
   sumContribX -= pixContribX;
   continue;
   }
  // rest of contribution factor (sumContrib < pixContrib)
  outVal[0] += (unsigned long)imgIn->Pixel(yi, xi).ch[0] * sumContribX;
  outVal[1] += (unsigned long)imgIn->Pixel(yi, xi).ch[1] * sumContribX;
  outVal[2] += (unsigned long)imgIn->Pixel(yi, xi).ch[2] * sumContribX;
  // done - output pixel to y culmulative
  cumulY[0][x] += (outVal[0]>>shift) * yContrib;
  cumulY[1][x] += (outVal[1]>>shift) * yContrib;
  cumulY[2][x] += (outVal[2]>>shift) * yContrib;
  if (outY)
    {
        // done - output pixel
        imgOut->Pixel(y, x).ch[0] = (Byte)(cumulY[0][x] >>shift);
        imgOut->Pixel(y, x).ch[1] = (Byte)(cumulY[1][x] >>shift);
        imgOut->Pixel(y, x).ch[2] = (Byte)(cumulY[2][x] >>shift);
        // contribution factor for next pixels block (pixContribY-sumContribY)
        cumulY[0][x] = (outVal[0]>>shift) * restContribY;
        cumulY[1][x] = (outVal[1]>>shift) * restContribY;
        cumulY[2][x] = (outVal[2]>>shift) * restContribY;
        }
      // contribution factor for next pixels block (pixContribX-sumContribX)
      sumContribX = pixContribX - sumContribX;
      outVal[0] = (unsigned long)imgIn->Pixel(yi, xi).ch[0] * sumContribX;
      outVal[1] = (unsigned long)imgIn->Pixel(yi, xi).ch[1] * sumContribX;
      outVal[2] = (unsigned long)imgIn->Pixel(yi, xi).ch[2] * sumContribX;
      sumContribX = norm - sumContribX;
      //
      outX = true;
      x++;
      if (x >= imgOut->Width())
         break;
      } // end xi loop
    // output last pixel
    if (!outX)
      {
      cumulY[0][x] += (outVal[0]>>shift)*yContrib;
      cumulY[1][x] += (outVal[1]>>shift)*yContrib;
      cumulY[2][x] += (outVal[2]>>shift)*yContrib;
      if (outY)
        {
        // done - output pixel
        imgOut->Pixel(y, x).ch[0] = (Byte)(cumulY[0][x] >>shift);
        imgOut->Pixel(y, x).ch[1] = (Byte)(cumulY[1][x] >>shift);
        imgOut->Pixel(y, x).ch[2] = (Byte)(cumulY[2][x] >>shift);
        // contribution factor for next pixels block (pixContribY-sumContribY)
        cumulY[0][x] = (outVal[0]>>shift) * restContribY;
        cumulY[1][x] = (outVal[1]>>shift) * restContribY;
        cumulY[2][x] = (outVal[2]>>shift) * restContribY;
        }
      }
    if (outY)
      {
      if (y == imgOut->Height()-1)
        {
           outLastRow = true;
           break;
           }
    y++;
    }
  if (y >= imgOut->Height())
      break;
  outY = false;
  if (sumContribY > pixContribY)
    {
    yContrib = pixContribY;
    sumContribY -= pixContribY;
    continue;
    }
  yContrib = sumContribY;
  restContribY = pixContribY - sumContribY;
  sumContribY = norm - restContribY;
  outY = true;
  }  // end yi loop
// output last row
if (!outLastRow)
    {
    for (int xo = 0; xo < imgOut->Width(); xo++)
            {
            imgOut->Pixel(y, xo).ch[0] = (Byte)(cumulY[0][xo] >>shift);
            imgOut->Pixel(y, xo).ch[1] = (Byte)(cumulY[1][xo] >>shift);
            imgOut->Pixel(y, xo).ch[2] = (Byte)(cumulY[2][xo] >>shift);
            }
    }
for (int i = 0; i < 3; i++)
        delete []cumulY[i];
}
```

Image container used for the source code can be replaced by any other image container or an 2D array holding RGB pixel values.

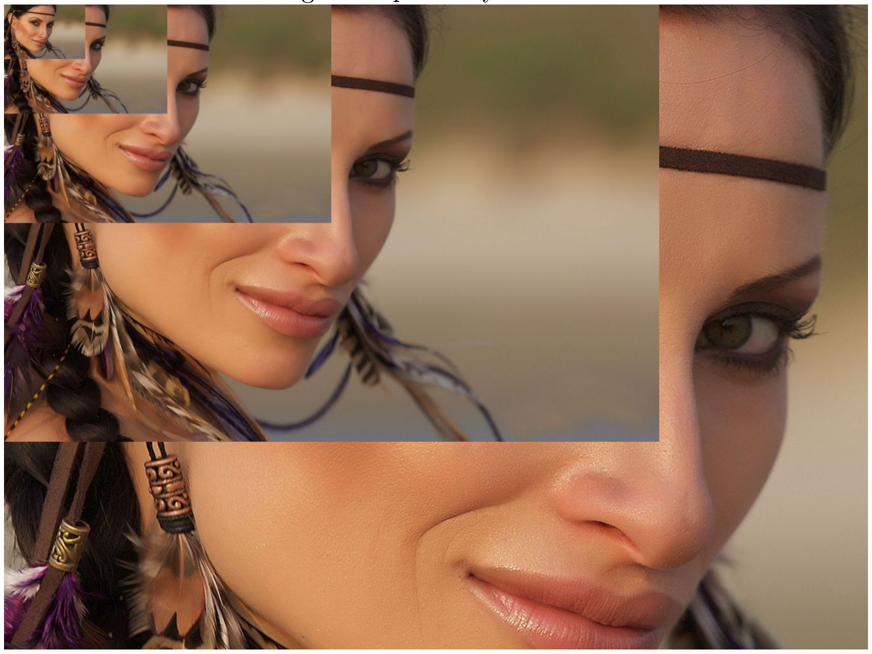
4 Performance test

Here is a table with some ad-hoc performance measurement (shrinking by 10, both width and height => w/10, h/10) . Test configuration: Intel i5-2400@3.10GHz, 6GB memory, Win7 64, program platform: 32-bit.

image size	speed (sec)
1024x1024	0.011
1115x1621	0.021
2048x1365	0.031
3240x4320	0.085
12000x12000	0.781

Last image tested is a huge image downloaded from NASA site.

Figure 1: photo by Jassi Oberai



An example of subsequent image shrinking by factor 2 (w/2, h/2).

5 Conclusion

Algorithm and accompaniment source code presented in this technical paper gives quite decent results with acceptable speed. Any improvements regarding this method are welcomed and highly appreciated. For any questions and suggestions please mail to: sigmapi@sigmapi-design.com.