A FAST METHOD FOR SCALING COLOR IMAGES

Jaana Parkkinen, Mikko Haukijärvi, Petri Nenonen
Nokia Corporation
P.O. Box 1000, FI-33721 Tampere
Finland
Jaana.Parkkinen@nokia.com, Mikko.Haukijarvi@nokia.com, Petri.Nenonen@nokia.com

ABSTRACT

Image scaling is an important processing step in any digital imaging chain containing a camera sensor and a display. Although resolutions of the mobile displays have increased to the level that is usable for imaging, images often have larger resolution than the mobile displays. In this paper we propose a software based fast and good quality image scaling procedure, which is suitable for mobile implementations. We describe our method in detail and we present experiments showing the performances of our approach on real images.

KEY WORDS

Mobile phone, camera, display, image scaling

1. Introduction

Over 5 megapixel size sensors are typical to digital cameras also in the imaging phones, and the sizes are constantly increasing. The display sizes have not increased at the same pace on the mobile devices. Therefore the captured image size has to be reduced so that it fits into the display. This means image downscaling using decimation methods. Sometimes an image is smaller than the display, or it contains an interesting detail. In this case the image size can be enlarged. The zooming requires upscaling using interpolation methods. The basic decimation and interpolation methods are very low in complexity and effectively implementable, but produce severe aliasing and pixelization artifacts. The downscaling and upscaling algorithms have to have an adequate quality. Otherwise artifacts, such as aliasing effects, jagged edges, excessive smoothing or pixelization, are introduced to images.

Mobile platforms set strict limits to the amount of memory and processing power available for image processing and enhancement algorithms. Large images consume lot of memory and processing power. The amount is directly or exponentially relative to the number of pixels in an image.

The sampling of signal is an important part of signal processing theory, and it is widely covered in the literature. [1] There are presented several possibilities to do image downscaling and upscaling in the literature of the signal and image processing. [2,3] In downscaling many input pixels correspond to one output pixel and in upscaling vice versa. In the basic downscaling method only one of the input pixels is selected to be the output pixel. This is called the nearest neighbor method. The nearest neighbor method produces severe aliasing artifacts. The common downscaling methods include antialias filter and re-sampling. The downscaled data are most often taken as a linear combination of the sampled input data and a certain kernel.

Sometimes only a part of the image contains interesting information. Varying level of zooming with panning support is required for showing the details at the area of interest. The zooming can be implemented using upscaling algorithms. The basic upscaling method is called pixel copy, which means copying one input pixel to multiple output pixels. This method causes pixelization and blocking artifacts. Better results are achieved by using more advanced methods that use some spatial filtering. There exist many methods with different complexity.

Because of varying sizes of source and target images methods supporting all possible scaling ratios are needed. Bilinear interpolation is a generally known method. In this method the output pixel is a weighted average of the nearest input pixels. The weights can be computed effectively for any scaling ratio. Therefore the bilinear interpolation is a good compromise between complexity and quality. A weighted average of the input pixels can be used also in the decimation case.

In this paper we introduce a novel and computationally effective solution to the problem of scaling of digital images. Our proposed method is a LUT (Look-up Table) based weighted average processing and it is fast and suitable for mobile implementations. We provide detailed description of the algorithm.

2. Proposed methods

In this section we describe a novel approach for color image scaling suitable for mobile implementations. The computational workflow is explained in more detailed way for downscaling, upscaling and pan and cropping use cases.

Downscaling algorithm

The proposed downscaling algorithm is designed to avoid unnecessary repetitive calculation while processing the images in order to reduce processing latency. Since the scaling factors in horizontal (x-direction) and vertical (y-direction) directions are constant for each row and column, all the scaling weights, indexes, the start and end pixels for the algorithm can be calculated and stored in index and weight vector LUTs. When performing the image looping and actual scaling the weights and indexes can be read from LUTs instead of recalculating them again for each pixel separately.

The downscaling algorithm operates by scanning pixels in one image row at time, once the row has been processed then continues with the next row. As an example the algorithm is described here in x-direction, and the operation is analogue in y-direction. The downscaling algorithm starts by calculating the start and end pixels in the original image as shown in figure 1.

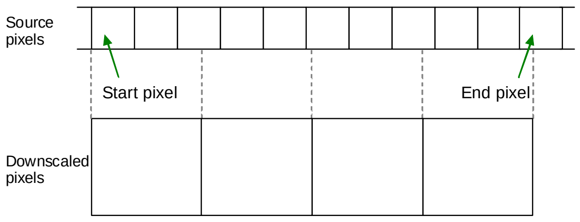


Figure 1 Start and end pixel alignment

The start and end pixels for the algorithm in x direction are calculated by:

$$start\_pixel = \frac{input\_image\_size}{2} - \frac{output\_image\_size}{2zoom\_factor} + \frac{panning}{zoom\_factor}$$

$$end\_pixel = \frac{input\_image\_size}{2} + \frac{output\_image\_size}{2zoom\_factor} + \frac{panning}{zoom\_factor}$$

in which the input image size is the size of the original image to be downscaled, output image size is the size of the downscaled image, panning is the amount of panning as pixels in the original image and the zoom factor defines the relation of the downscaled image compared to the original image. Since cropping and panning in all 4 directions (up, down, right and left) need to be supported for searching and viewing important image details that may be cropped out if only part of the image is shown and the rest is cropped, the panning value needs to be taken into account in the equations.

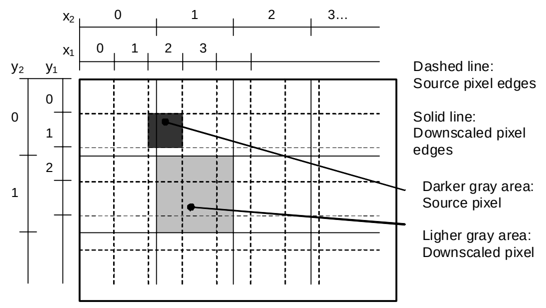

x2
0
1
2
3...
x1
0
1
2
3
y2
y1
0
1
2
0
1
Dashed line: Source pixel edges
Solid line: Downscaled pixel edges
Darker gray area: Source pixel
Ligher gray area: Downscaled pixel

Figure 2. Downscaled pixel value is a weighted average of the source pixel values that “belong to” the area of the downscaled pixel [3].

Next the source pixel indexes (index = 0, 0, 1, 0, …) and subpixel indexes (index = 1, 2, 3, …) for the downscaling filters with weighted average filter are defined as shown in figures 2 and 3.

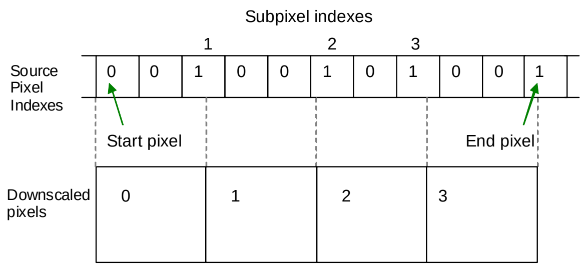


Figure 3. Source pixel indexing and subpixel indexing.

The source pixel index defines all the input pixels that are needed to calculate two downscaled pixels. The source pixel indexes are calculated by:

$$index(x) = (x - start\_pixel + 1)zoom\_factor - (x - start\_pixel)zoom\_factor;$$
$$x \neq start\_pixel$$

Hence the source pixel index is set to 1 if the input pixel is needed to calculate two downscaled pixels in x-direction, otherwise it is set to 0. While calculating the source pixel indexes the subpixel index is marked and incremented every time the source pixel is needed to calculate 2 downscaled pixels.

Next the algorithm calculates 2 weights in x-direction (in total 4 weighting factors are needed for one downscaled pixel: 2 for x-direction and 2 for y-direction) for each subpixel defined in previous step as shown in figure 4.

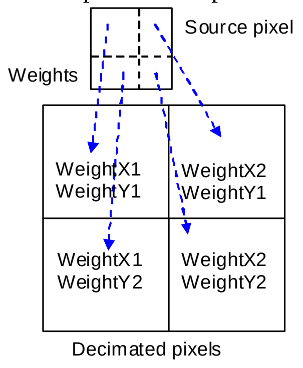


Figure 4. Defining weights.

The weight coefficients are proportional to how much of the input pixel area is inside the output pixel area. The 2 weights are calculated by:

$$weight_1(x) = \begin{cases} (1 - start\_pixel - x)1024\,zoom\_factor; x = start\_pixel \\ 1024\,zoom\_factor; index(x) = 0 \\ \left(\frac{subpixel\_index}{zoom\_factor} - x - start\_pixel\right)1024\,zoom\_factor \end{cases}$$

$$weight_2(x) = \begin{cases} 0; x = start \\ 0; index(x) = 0 \\ 1024\,zoom\_factor - weight1() \end{cases}$$

where $1 \leq x \leq end$

The weight1 can be considered as the left hand side weight for the scaled pixel and weight2 as the right hand side weight. The sum of weight1 and weight2 needs to be 1 as defined in weighted average filtering. Note that the weights and indexes are calculated as integers to minimize the need of using real numbers in calculations and also to minimize processing latencies and memory usage. As example 10bit accuracy (1024 multiplier) is used here to convert real values into integers.

The source pixel index is set to 0 for the first source pixel since the first downscaled pixel has no left side source pixel, and also when one source pixel goes to only one downscaled pixel. The index is set to 1 when one source pixel goes to two downscaled pixels or when the left borders of the source pixel and the downscaled pixel align exactly inside the image borders.

The index(x), weight1(x) and weight2(x) vectors are stored in look-up tables to minimize the need of recalculating them for each row separately. This can be done since the zoom factor is constant for each image row. The use of LUTs in processing the image reduces the processing latency significantly and the main target in the proposed method was to utilize LUTs as much as possible.
Once the weights and indexes have been calculated in x direction the weights and indexes in y direction need to be calculated using the above formulas.

As a final step of the algorithm the actual image downscaling is done by calculating the downscaled pixels. The source image pixels are read from the previously defined start pixel to the end pixel, and the indexes and weights are read from Look-up tables. The processing is done by looping one source image row at the time. The downscaled pixels are calculated by:

$$out\_pixel\,(x1, y1) += weight\,1(x) * weight\,1(y) * input\_pixel\,,$$
$$out\_pixel\,(x2, y1) += weight\,2(x) * weight\,1(y) * input\_pixel\,,$$
$$out\_pixel\,(x1, y2) += weight\,1(x) * weight\,2(y) * input\_pixel\,,$$
$$out\_pixel\,(x2, y2) += weight\,2(x) * weight\,2(y) * input\_pixel\,,$$

Since only those output pixels need to be calculated for which the subpixel index of the source pixel is 0, only 1 or 2 downscaled pixels may need to be calculated in stead of all 4. This reduces the processing latency significantly for very small scaling factors since for majority of source pixels the subpixel index is set to 0.
The downscaled pixels for each row are calculated by adding the original input pixel values to the 2 output buffers in each processing loop as shown in figure 5.

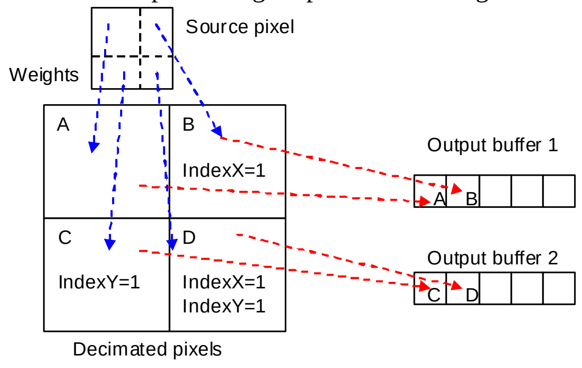


Figure 5 Output pixel buffering

One input pixel may affect 4 downscaled pixels (A, B, C, D) depending on the index values indexX and indexY i.e one source pixel can affect 2x2 downscaled pixels as described earlier. If the input pixel is affecting only one downscaled pixel (source pixel index for x is 0, source pixel index for y is 0) the input pixel value is added to the output buffer as such (pixel A in figure 8). If the indexX is set to 1 the pixel B is calculated using the weighted input pixels and added to the corresponding output buffer index. If the indexY is set to1 the pixel C is calculated using weighted input pixels and added to the second output buffer to the corresponding index. Finally if both indexes are set to 1, the pixel D is calculated and added to the second output buffer to the corresponding index.

When the whole image length is processed the added pixel values in the output buffer 1 are converted back to 8-bit numbers by dividing with 1024 as used in this example (limited to values 0…255) and written to the output image row buffer. If the current source row affects to two downscaled rows (indexY=1) and if the downscaled row is ready processed in the output buffer 1 the row can be written to output image. After writing the row to the output image the output buffers 1and 2 are swapped and the processing starts again from the pixel downscaling and output buffering for the next input image row.

Note that for RGB images each color component is processed separately so the above calculations need to be done for R, G, and B components separately.

In addition to the low computational load the downscaling algorithm uses only 1 source image row, 2 destination image rows and 2 output buffers to downscale the images. Only 2 subpixel index vectors (1 for original image length and 1 for original image height) and 4 weight vectors (2 for original image length and 2 for original image height) are also allocated for processing. The algorithm therefore needs only very little memory and is very suitable for applications which have limited memory available such as mobile phones.

Upscaling algorithm

The proposed upscaling method is based on bilinear interpolation method. Similarly to the downscaling method the proposed interpolation method is also designed to avoid unnecessary repetitive calculations while processing the images. Therefore since the scale factors in x- and y-direction are constant for each row and column, all the scaling weights, indexes and the starting and ending pixels for the algorithm can again be calculated and stored in index and weight vectors before performing the image looping and the actual scaling. As example the algorithm is described here in x-direction.

The algorithm starts by defining the start and end pixels in the source image as shown in figure 6.

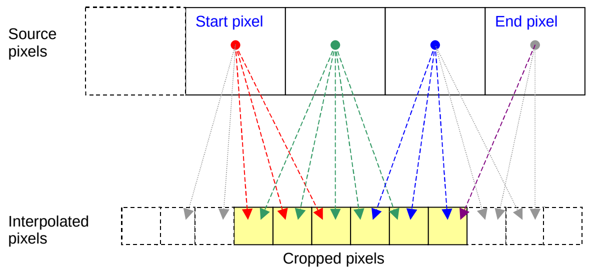


Figure 6 Start and end pixels for upscaling

The start pixel is calculated by:

$$start = \frac{source\_image\_size - 1}{2} - \frac{destination\_image\_size - 1}{2zoom\_factor} + \frac{panning}{zoom\_factor}$$

Since the upscaled image can be panned, the start pixel position depends on the zoom factor and the panning position. The panning origin is defined to be the center pixel of the scaled image. Also note that the panning values are defined as scaled pixels. There is no need to calculate the end pixel index because the algorithm is designed so that processing is done by looping the upscaled pixels instead of looping the original source pixels, and since the viewed image is smaller than the upscaled image due to cropping.
Each scaled pixel is an output of 4 original pixels weighted as in bilinear interpolation. The processing is done by looping two source image rows at a time and using 2x2 filtering window to calculate the interpolated pixels. Again the scaling can be done in x- and y-directions separately while scanning the source image rows. The source pixel indexes and weights are in x direction for are calculated by:

$$index(x) = \begin{cases} 0, start + \frac{x}{zoom\_factor} < 0 \\ 0, source\_image\_size - 1 < start + \frac{x}{zoom\_factor} \\ \left\lfloor start + \frac{x+1}{zoom\_factor} \right\rfloor - \left\lfloor start + \frac{x}{zoom\_factor} \right\rfloor \end{cases}$$

where $0 < x < destination\_image\_size.$

$$weight_1(x) = \begin{cases} 1024, start + \frac{x}{zoom\_factor} < 0 \\ 1024, source\_image\_size - 1 < start + \frac{x}{zoom\_factor} \\ 1024 - 1024\left(start + \frac{x+1}{zoom\_factor} - \left\lfloor start + \frac{x}{zoom\_factor} \right\rfloor \right) \end{cases}$$

$$weight_2(x) = \begin{cases} 0, start + \frac{x}{zoom\_factor} < 0 \\ 0, source\_image\_size - 1 < start + \frac{x}{zoom\_factor} \\ \left\lfloor start + \frac{x+1}{zoom\_factor} \right\rfloor - - \left\lfloor start + \frac{x}{zoom\_factor} \right\rfloor \end{cases}$$

where $0 < x < destination\_image\_size.$

The index is set to 1 when the next interpolated pixel value is calculated from the next source pixel pair, i.e. the center of the next interpolated pixel is on the right side of the center of the current right source pixel, and to 0 otherwise.
The above equations are then used to calculate the indexes and weights in y-direction also.
The weights and indexes are calculated as integers and 10bit accuracy is used here as an example to convert the real values into integers. Note that when the center of the destination image pixel is on the left (or upper) side of the center of first source input pixel (start + x/zoom_factor < 0) the algorithm has no left (or the upper) side source pixel to calculate the weights. The first source pixel only is used to calculate the left side (or upper) weight and the right side (or lower) weight is set to 0. Also note that when the center of the destination image pixel is on the right (or lower) of the center of last source input pixel (input_image_size – 1 < start_pixel + x/zoom_factor) the algorithm has no right (or lower) side source pixel to calculate the weights. The last source pixel only is used to calculate the left side (or the upper) weight and the right side (or the lower) weight is set to 0.
When the whole image length is processed the upscaled pixels are converted back to 8-bit numbers by dividing with 1024 as used in this example (limited to values 0…255) and written to the output image row buffer.
Using LUTs makes the method very low in computational complexity. The upscaling method uses only 2 source image rows and 1 destination image row to upscale the images. 2 pixel index vectors (1 for source image length and 1 for source image height) and 4 weight vectors (2 for source image length and 2 for source image height) are also allocated for processing. These make the method very fast and memory efficient and therefore applicable to mobile implementations.

Pan and crop

When the images are zoomed in to 100% or viewed in original size (1:1 zoom) one pixel in the original image corresponds to one pixel in the displayed image. In this case the original image is only cropped to the correct size if needed and only panning position needs to be taken into account instead of zoom factor.

The algorithm starts by calculating the first and last pixels in both x and y direction:

$$start\_pixel = \frac{input\_image\_size - output\_image\_size + 1}{2} + panning$$

$$end\_pixel = start\_pixel + output\_image\_size \cdot$$

Processing in this case is done by only picking the right amount of pixels from the original input image and copying them to the corresponding output image. Hence the algorithm is very fast since no filtering with weighting and indexing is needed but only memory copying.

Only one source image row and one destination image row need to be used for processing which makes the algorithm also very memory efficient.

3. Results

The computational complexity of the proposed scaling methods has been estimated by calculating the used operations per pixel. The results have been collected to Table 1, and are compared to different scaling methods collected in references [4] and [5]. As seen from the results the implementation has very low computational load.

Table 1. Comparisons between different methods. (a) is min number of operations per pixel for downscaling 2x2 pixels at the time, (b) is max number of operations per pixel for downscaling 2x2 pixels at the time, and (c) is the number of operations per pixel for upscaling 2x2 pixels at the time

	Scaling method					
	Proposed methods			Nearest (reference method)	Bilinear (reference method)	Bi-cubic (reference method)
Operation	*(a)*	*(b)*	(c)			
Addition	*3*	*12*	**9**	2	**16**	22
Multiplication	*6*	*24*	**12**	0	**18**	29

Other filtering methods, e.g. bi-cubic filtering, can be implemented using the proposed method by calculating the weights for the filter to be used and updating the filter LUTs. Hence changing the filter method does not add the computational load of the proposed methods.

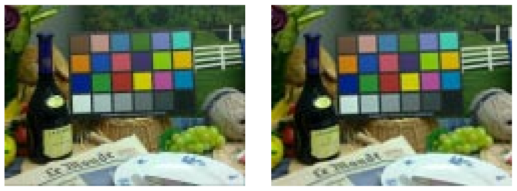
The memory consumption of the proposed scaling methods has also been estimated. Since the filtering LUTs are calculated before the actual image scaling, the scaling needs only 2 image rows for processing despite the used filtering method e.g. nearest neighbor, bilinear, or bicubic. As an example to scale an 8-bit VGA color image would require 2x3x640x8 bits, and a 1MPix color image would require 2x3x1024x8 bits.

The visual quality of the presented upscaling method corresponds to bilinear interpolation method, and the visual quality of the presented downscaling method corresponds to linear filtering methods. In Figure 7, there are presented two images from different downscaling methods: the proposed method is used in case (a), and weighted average method in commercial image editing software is used in case (b). Results show that the proposed method significantly improves the scaling performance when compared to the reference method. The visual quality of the proposed solution using simple integer operations with 10 bit accuracy is comparable to commercially available well-known implementation.

4. Conclusion

In this paper, a fast image scaling method based on weighted average of neighbouring pixels was presented. The method reduces aliasing and blocking artefacts resulting good image quality. The method is based on LUTs calculated beforehand and only once, so that excessive calculations are minimised. Then only simple LUT readings and memory accesses are used to scale the image. This makes the processing loop very simple and easy to optimise. The whole method can be further optimised using e.g. assembler coding.

Also only few image row buffers and LUTs are needed for processing. As a result the presented method has very low memory consumption and it is computationally effective, and the resulted scaled image quality is good. The method can be utilised in devices with low processing power, e.g. in mobile camera phones and in multimedia handsets.



(a) (b)

Figure 7. Examples of downscaling results. (a) Proposed downscaling method, (b) Commercially available downscaling method

References

[1] A.K. Jain, Fundamentals of Digital Image Processing (Englewood Cliffs, NJ: Prentice-Hall, 1989).

[2] J. A. Parker, R. V. Kenyon, and D. E. Troxel, Comparison of interpolation methods for image resampling, *IEEE Transactions on Medical Imaging, Vol. 2*, No. 1, 1983, 31–39.

[3] Chun-Ho Kim, et al., Winscale: An Image-Scaling Algorithm Using an Area Pixel Model, *IEEE Transactions on Circuits and Systems for Video Technology, Vol. 13*, No. 6, June 2003, 549-553.

[4] A. Amanatiadis and I. Andreadis, Digital Image Scaling, *Instrumentation and Measurement Technology Conference, Ottawa, Canada*, May 17-19, 2005.

[5] A. Amanatiadis and I. Andreadis, Performance Evaluation Techniques for Image Scaling Algorithms, *IEEE International Workshop on Imaging Systems and Techniques, Chania, Greece*, September 10-12, 2008.