FOUNDATIONS OF COMPUTING AND DECISION SCIENCES
Vol. 44 (2019) No. 2
ISSN 0867-6356
DOI: 10.2478/fcds-2019-0011 e-ISSN 2300-3405

sciendo

Extensible Implementation of Reliable Pixel Art Interpolation

Paweł M. Stasik, Julian Balcerek * †

Abstract. Pixel art is aesthetics that emulates the graphical style of old computer systems. Graphics created with this style needs to be scaled up for presentation on modern displays. The authors proposed two new modifications of image scaling for this purpose: a proximity-based coefficient correction and a transition area restriction. Moreover a new interpolation kernel has been introduced. The presented approaches are aimed at reliable and flexible bitmap scaling while overcoming limitations of existing methods. The new techniques were introduced in an extensible .NET application that serves as both an executable program and a library. The project is designed for prototyping and testing interpolation operations and can be easily expanded with new functionality by adding it to the code or by using the provided interface.

Keywords: image processing, pixel art, image upscaling, bitmap interpolation, proximity measure, proximity-based coefficient correction (PBCC), p-lin interpolation, transition area restriction (TAR)

1. Introduction

Old computer systems, in comparison to modern systems, were heavily restricted in their graphical capabilities (in the sense of the amount of available colors and the possible resolutions). Pixel art is an artistic form that was aimed at handling these limitations, but them should not prevent it from being presented with graphical possibilities of the modern systems. However, due to its original small size, pixel art needs to be sized up to be presented on modern high-resolution displays. The upscaling process has to preserve specific traits of pixel art: important pixel-level details, limited color palette, and strong contrast. Because of that a scaling method has to be picked carefully.

*Faculty of Computing, Poznań University of Technology, Poznań, PL, pawel.m.stasik@gmail.com, julian.balecerek@put.poznan.pl

† This paper is an expanded and significantly revised version of "Improvements in Upscaling of Pixel Art" that appeared in SPA 2017 conference proceedings.

Pixel art does not only appear nowadays in emulation of the old systems. It is also aesthetics of choice for new released games that are designed to be played on modern systems while staying true to the pixel art limitations. A list of few examples of such games is presented in Table 1.

Table 1. Estimated number of owners for the selected games on the Steam platform (as of December 1, 2017) [23]

Game	Owners	Release date	Developer
FTL: Faster Than Light	$2{,}925{,}530 \pm 52{,}448$	Sep 14, 2012	Subset Games
Owlboy	$188{,}162 \pm 13{,}339$	Nov 1, 2016	D-Pad Studio
Papers, Please	$1{,}915{,}985 \pm 42{,}489$	Aug 8, 2013	3909
Terraria	$9{,}547{,}040 \pm 94{,}092$	May 16, 2011	Re-Logic
Undertale	$3{,}001{,}685 \pm 53{,}122$	Sep 15, 2015	tobyfox

While there are standard, well-known ways of displaying pixel art [4, 24, 26], there is still a room for improvements in the state-of-the-art approaches to pixel art scaling [7, 22]. The list of problems includes: a disproportionate replication of pixels, loss of sharpness, restricted scale factors, and questionable decisions in image reinterpretation. As a result of that, the authors designed new and reliable techniques: the proximity-based coefficient correction (PBCC), transition area restriction (TAR), and p-lin [22]. These methods are reliable as they allow for scaling up bitmaps with any scale factor while providing a faithful and well-formed square visualization of the original pixels.

In this paper these new methods are presented in comparison to classic algorithms. Further the implementation of the new techniques is presented. The implementation was aimed at experimentation and showcase of the proposed algorithms. Moreover, the implementation is extensible, as it can be easily modified and expanded with new algorithms with help of included and in-code documentation.

The implementation relies on .NET Framework and while it uses it for optimization, it is not aimed at working in the real time. Shader support would have improved the execution time, but would also have made the implementation harder to modify. The application was designed in mind with easy prototyping various scaling methods and testing them, comparing them and their processing times. Moreover, the implementation can work both as an executable program and as a library, which expands its use even more.

In the following sections the methods are shown. Then the extensible software implementation is presented with an explanation of choices made for the project. At the end there is a showcase of results of scaling pixel art followed by conclusions.

2. State of the art

Several approaches of bitmap resizing can be used for scaling up pixel art. They can be divided into two basic groups — classic scaling methods and dedicated methods.

The easiest classic methods are the linear interpolations and the nearest neighbor [1]. Both are typically implemented in modern graphic processing units [15]. The linear interpolation does not impact on overall proportions of resized picture elements, but provides a blur (see Fig. 1a). The blur gets stronger with increase of the scale factor (Fig. 1a — keep in mind that the left image was upsampled for presentation). This drawback makes this method unsuitable for scaling pixel art.

On the other hand, the nearest neighbor is usually advised for scaling pixel art [4]. However, while providing great contrast, for fractional scale factors the nearest neighbor does not copy pixels equally. Small disfigurements occurring in such a scenario are an undesired side effect (see Fig. 1b). The proposed methods [22] were designed to fit in the middle ground between both the nearest neighbor and linear interpolation methods. Their results presented on displays create an illusion of sharp and undeformed pixels (see Fig. 1h, 1i, 1j and 1k).

Other classic methods of image scaling are the cubic (bicubic for 2D image processing) interpolation and Lanczos resampling [1]. Their major issue that makes them unfitted for scaling pixel art is overshots, which can be seen in Figures 1c and 1d.

Two typical pixel art-scaling methods were selected as examples — hqx and xBRZ. Both algorithms rely on a binary comparison of each pixel with its eight neighbors (whether both pixels are or are not similar). In case of hqx each of 256 possible similarity scenarios for a given pixel has assigned a combination of blending functions to fill an area representing the pixel [24]. An approach taken by xBRZ is different — the area is filled with one color and the similarity tests are used to detect specific patterns like sharp or round corners, applying appropriate corrections if needed [26].

While both hqx and xBRZ are designed with pixel art in mind and are aimed at providing readable images, they are not free from their own drawbacks. The main problem is that these methods allow only for specific, integer scale factors (up to 4 for hqx and up to 6 for xBRZ). Because they were not prepared to handle their own output, it is not advised to combine several passes of them to get the scale factors that were not defined (e.g. combining $\times 2$ and $\times 4$ to get $\times 8$ is not valid). The classic scaling methods might be used to deal with this problem, but such approach would include problems related to the used classic techniques. Due to complexity of the code of both methods an addition of new scale factors would require a lot of work.

The other issue with hqx and xBRZ is that the ways they handle image features were arbitrary chosen by their creators. On subjective level this may lead to either nice or questionable connections of pixels in the outcome images. The second kind might be either not liked by a viewer or not representing intentions of a creator of the image. Moreover, the shapes may be misinterpreted, which will result in errors in the output, like those that can be seen in Fig. 1e or 1f.

An interesting approach to scaling pixel art was presented by Kopf and Lishinski [7]. The idea was to vectorize the bitmap by turning it into Voronoi map. The next step was to make connections between cells and to optimize the resulting splines, by analyzing similarity of neighboring pixels. The final vector image is then rendered on the display. The main disadvantage of this method is, as it was stated by its creators themselves, that it does not handle well all possible scenarios. While there exists demo allowing for real time computations of the algorithm [8], it tends to

render image with artifacts that are especially visible at large scale factors (see Fig. 2). The other available implementations can be found as libdepixelize [5] which was included in Inkscape vector graphics editor [6]. In [20] an improvement over the Kopf-Lishinski depixelizing algorithm was proposed, named pixel art remasterization, aiming at shorter processing time. However, both methods are prone to make wrong connections as objects move in the rendered scene (see Fig. 1g for an example of the depixelizing algorithm and see [19] for the remasterization algorithm).

So far the presented techniques depend on predefined rules. These rules can take various forms - from a simple kernel function to to be as complex as xBRZ, hqx, or depixelization. However, there are techniques that can, instead of having set rules, adapt to the data.

There are more experimental approaches that instead of relying on predefined data first need to be trained to be able to scale up images. Good example of such algorithms aimed at synthetic pictures are waifu2x [2, 14] and pixcaler [25]. Both algorithms rely on neutral networks to learn about how images are supposed to be scaled. Waifu2x was designed with scaling digital artwork while pixcaler is aimed at scaling pixel art. Both algorithms need a learning dataset, but are also provided with trained data of their own. Sadly, in case of pixel art no reference model is typically provided. In case of pixcaler, models are automatically generated from the provided set of images. Looking at the output of these two algorithms with their default datasets, we can see that waifu2x does not handle pixel art well (Fig. 3b), while pixcaler provides quite promising results (Fig. 3c). However both algorithms have limited use and allow only for the highest scale factor to be of 2.

We proposed in our work [22] an approach to pixel art scaling that solely relies on proper modifications of interpolation kernel. These techniques are the proximity-based coefficient correction (PBCC) and the transition area restriction (TAR). The aim of PBCC is to improve sharpness of the linear interpolation by revaluing its coefficients in a regard to the real two-dimensional distance of the interpolated point from the defined source points (see Fig. 1h). From PBCC a new interpolation function, called p-lin (see Fig. 1j), was proposed. TAR affects the coordinates of the interpolated point and is similar to a clamping function, but TAR ties the size of the transition to the target resolution. That way TAR allows for achieving a good contrast with various target display resolutions (see Fig. 1i). Moreover, because these techniques modify normal interpolation kernel, they allow for any scale factors.

When it comes to software designed for testing various scaling algorithms, there are several options. Custom interpolation kernels can be implemented in Matlab [9], however the environment is proprietary and not freely available. In regards to freeware and open-source applications like Octave [3], they can be modified to include new interpolation options. This is nevertheless not always an easy task. In regards to optimized implementations, there is a project that tests various kernels using OpenGL [18], but the use of shaders makes addition of new methods more difficult and less straightforward.

(a) Linear (×1.5 on the left, ×3 on the right)

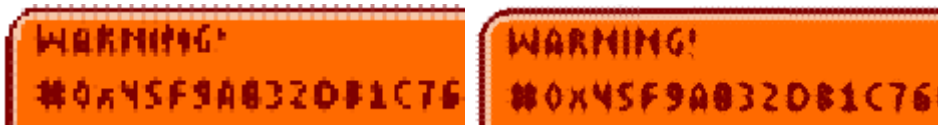
(b) Nearest neighbor (×1.5 on the left, ×3 on the right)

(c) Bicubic (×3)

(d) Lanczos-3 (×3)

(e) hqx (×3)

(f) xBRZ (×3)



(g) Depixelize (Kopf-Lishinski, ×1.5 on the left, ×3 on the right)

(h) Linear + PBCC (×3)

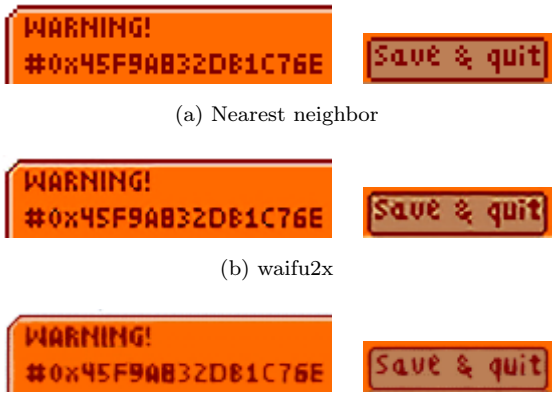
(i) Linear + TAR set to 2 (×3)

(j) P-lin (×1.5 on the left, ×3 on the right)

(k) P-lin + TAR set to 2 (×1.5 on the left, ×3 on the right)

Figure 1. An overview of the scaling methods that rely on predefined rules — classic, pixel art-dedicated and introduced by the authors. The images for scale factor ×1.5 were upsampled two times for readability. The outcome of the nearest neighbor ×3 serves as a ground truth.

Figure 2. An example of artifacts in depixelization algorithm. The image is from game *Owlboy* (by courtesy of D-Pad Games).



(a) Nearest neighbor

(b) waifu2x

(c) pixcaler

Figure 3. An overview of the scaling methods that rely on neutral networks — waifu2x and pixcaler. The scale factor is $\times 2$. The nearest neighbor was added for comparison.

3. Classic interpolations

Let us assume we have a bitmap $\mathbf{I}$ that is M pixels height and N pixels wide. A value assigned to a pixel at (x, y) is represented by $\mathbf{I}(x, y)$, where

$$x \in [0, N-1] \cap \mathbb{Z}\,, \qquad y \in [0, M-1] \cap \mathbb{Z}\,. \tag{1}$$

If we want to obtain values for each pixel of the image $\mathbf{I}$ scaled up with a scale factor S, the new values have to be interpolated using the existing original pixels. For this purpose let us assume that each pixel (from both the source image and target image) has its value assigned to its geometrical center. The shift is important for obtaining a proper outcome. The transition between the position of the (x, y) pixel in an bitmap to (x_v, y_v) coordinates of its value is

$$(x_v, y_v) = (x + 0.5, y + 0.5)\,. \tag{2}$$

Let us assume that (x_r, y_r) is a position of an pixel in the bitmap that is a result of scaling up. The corresponding coordinate in the source bitmap is

$$(x, y) = \left(\frac{x_r + 0.5}{S} - 0.5, \frac{y_r + 0.5}{S} - 0.5\right)\,, \tag{3}$$

which is a result of the transition to the center of the target image pixel, casting into source image, and transition back from the center of the source image pixel. Notice, that the coordinates might end up with a fractional value which means that the point lies between two pixels. A reference pixel assigned to these coordinates is

$$(x_0, y_0) = (\lfloor x \rfloor\,, \lfloor y \rfloor) \tag{4}$$

and the position of the interpolated point from the reference point is

$$(x', y') = (x - x_0, y - y_0)\,. \tag{5}$$

It is worth noting that the pixels in bitmaps are uniformly spaced from one another in a square grid. Because of that we can write a formula for more coordinates of the defined pixels of the source bitmap in relation to the interpolated point:

$$(x_p, y_r) = (x_0 + p, y_0 + r)\,. \tag{6}$$

We can also describe a relative position of the defined points as

$$(x'_p, y'_r) = (p, r)\,. \tag{7}$$

When an image is scaled up, the value for a pixel at (x_r, y_r) is the value of the image $\mathbf{I}$ at (x, y). This value is obtained with an interpolation function. All the presented classic approaches are convolution-based and are represented with an interpolation kernel that is applied to the defined values of the source image:

$$\begin{aligned} \mathbf{I}(x, y) &= \sum_{p,r} W(x - x_0 - p, y - y_0 - r) \cdot \mathbf{I}(x_0 + p, y_0 + r) = \\ &= \sum_{p,r} W(x' - x'_p, y' - y'_r) \cdot \mathbf{I}(x_p, y_r)\,, \end{aligned} \tag{8}$$

where $W(x, y)$ is the window function of the interpolation kernel. The values u and v should be limited to the size of the window (to the range within which its values are non-zero). In case of pixels close to the border of the image, there might be a need to read values outside of the bounds of the picture. In such scenario the values can be acquired by selecting a strategy of handling the out-of-bounds cases, e.g. mirroring the image.

Formulas for the mentioned (in the state-of-the-art section) classic approaches are enlisted below. Notice, that all presented functions can be separated into a product of two independent one-dimensional functions (which are exactly the same). Also notice, that in the case of the Lanczos resampling there is n parameter, which denotes size of the window of the function. The most popular sizes of the window are $n = 2$ and $n = 3$ [1]. Plots of the one-dimensional kernels of the presented interpolation functions are shown in Fig. 4.

The classic methods have kernels as follows:

- nearest neighbor

$$W_{\mathrm{nn}}(x, y) = \begin{cases} 1 & ; \ -0.5 < x, y < 0.5 \\ 0 & ; \text{otherwise,} \end{cases} \tag{9}$$

- bilinear interpolation

$$W_{\mathrm{lin}}(x, y) = \begin{cases} 1 - |x| - |y| - |xy| & ; \ 0 \leq |x|, |y| < 1 \\ 0 & ; \text{otherwise,} \end{cases} \tag{10}$$

- bicubic interpolation

$$W_{\mathrm{bic}}(x, y) = w_{\mathrm{cub}}(x) \cdot w_{\mathrm{cub}}(y)$$
$$w_{\mathrm{cub}} = \begin{cases} |x|^3 - 2 \cdot |x|^2 + 1 & ; \ 0 \leq |x| < 1 \\ -|x|^3 + 5 \cdot |x|^2 - 8 \cdot |x| + 4 & ; \ 1 \leq |x| < 2 \\ 0 & ; \text{otherwise,} \end{cases} \tag{11}$$

- Lanczos-n resampling &

$$W_{\mathrm{L}n}(x, y) = w_{\mathrm{L}n}(x) \cdot w_{\mathrm{L}n}(y) \quad w_{\mathrm{L}n} = \begin{cases} 1 & ; \ 0 = |x| \\ n \cdot \dfrac{sin(\pi x/n) \cdot sin(\pi x)}{\pi^2 x^2} & ; \ 0 < |x| < n \\ 0 & ; \text{otherwise.} \end{cases} \tag{12}$$

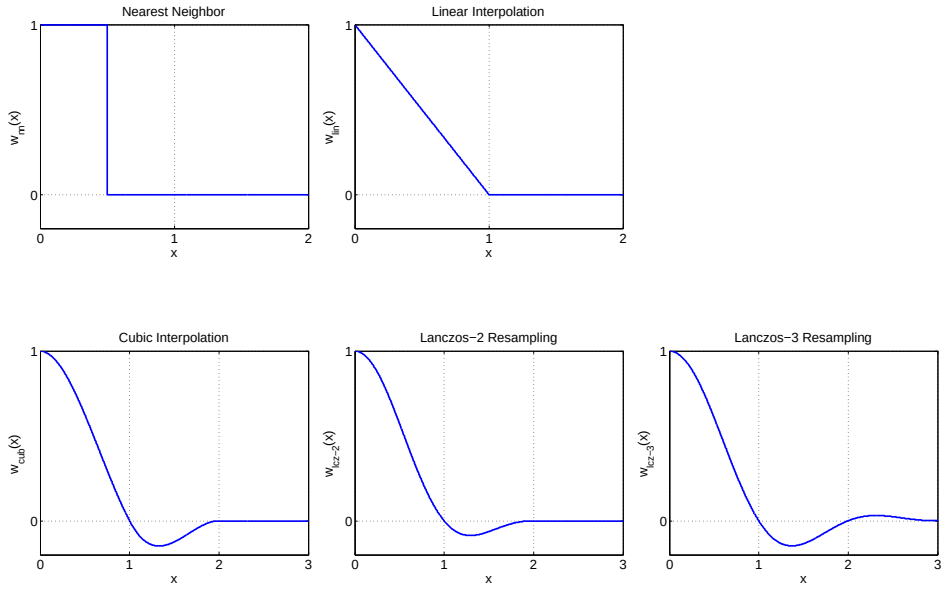


Figure 4. Positive values of the classic kernels. The negative values are mirrored.

4. Proposed methods

4.1. PBCC

Let us consider that we use the linear interpolation to obtain coefficients assigned for each of the defined points. These factors are in practice a product of two independent one-dimensional interpolations, i.e. horizontal and vertical. Let us now consider that we want to revalue these coefficients in regards to the true Euclidean two-dimensional distance from the interpolated point. The following proximity function [21] was used for this purpose:

$$b_{p,r} = b(x', y', x'_p, y'_r) = \begin{cases} 1 & ; [x'y'] = [x'_p y'_r] \\ 1 - \sqrt{\dfrac{(x' - x'_p)^2 + (y' - y'_r)^2}{2}} & ; \text{otherwise,} \end{cases} \tag{13}$$

where x', y' are the relative coordinates of the interpolated point, and x'_p, y'_r are relative coordinates of a defined point in relation to the reference point (x_0, y_0). Because only four defined points are used in the case of the linear interpolation, forming a square around the interpolated point, we have $x'_p, y'_r = 0, 1$. Moreover, the proximity function returns one if the interpolated point is overlapping with the defined point and zero if both across the diagonal. Note, that the proximity function was simpli-

fied to encompass only the case of $x', y' \in [0, 1]$. Outside of this range, value of the function should be considered to be zero.

With the proximity function in mind, we can redefine the linear interpolation function as:

$$\mathbf{I}(x, y) = \sum_{p,r=0}^{1} W_{\text{lin}} \cdot \frac{b_{p,r}}{B} \cdot \mathbf{I}(x_p, y_r) = \sum_{p,r=0}^{1} W_{\text{lin+PBCC}}(x, y) \cdot \mathbf{I}(x_p, y_r)\,, \tag{14}$$

where B is a sum of all four proximity values (this sum is needed to keep convexity of the interpolation):

$$B = \sum_{p,r=0}^{1} b_{p,r}\,. \tag{15}$$

This is how PBCC is applied to the linear interpolation. If we would like to define only the coefficients of the new, proximity-corrected interpolation, we get:

$$W_{\text{lin+PBCC}}(x, y) = W_{\text{lin}}(x, y) \cdot b(x, y, 0, 0)/B\,. \tag{16}$$

The results of the linear interpolation with PBCC are a little bit sharper than the linear interpolation and involve a small rounding effect to the corners of the pixels. It is worth noting, that PBCC can be applied multiple times to achieve a stronger effect.

4.2. P-Lin

Let us consider a one-dimensional kernel of the linear interpolation:

$$w_{\text{lin}}(x) = \begin{cases} 1 - |x| & ;\, 0 \le |x| < 1 \\ 0 & ;\, \text{otherwise} \end{cases} \tag{17}$$

and a one dimensional variant of the previously used proximity function (13) [21]:

$$b_p = b(x', x'_p) = \begin{cases} 1 - |x| & ;\, 0 \le |x| < 1 \\ 0 & ;\, \text{otherwise.} \end{cases} \tag{18}$$

The results of applying PBCC in such case would yield two possible non-zero coefficients:

$$w_{\text{p-lin}}(0) = \frac{(1 - x')^2}{(1 - x')^2 + x'^2}\,, \qquad w_{\text{p-lin}}(1) = \frac{x'^2}{(1 - x')^2 + x'^2}\,. \tag{19}$$

These factors can be combined to get a two-dimensional interpolation:

$$W_{\text{p-lin}}(x, y) = w_{\text{p-lin}}(x) \cdot w_{\text{p-lin}}(y)\,. \tag{20}$$

This interpolation kernel was named ‘p-lin’, which stands for ‘the proximity-corrected (one-dimensional) linear interpolation’. The results achieved with this method are sharper than for the bilinear interpolation with PBCC and the corners are not rounded. This difference on the corners can be either good or bad effect depending on what is expected and needed.

4.3. TAR

The transition area restriction (TAR for short) is a technique of reducing the between-pixels area that is a subject to interpolation. The idea is that if an interpolated pixel is close enough to one of the defined points, its value is assigned to be of that closest defined point. The points that are going to be interpolated have their coordinates modified to encompass for the reduced transition area between pixels. TAR works in a similar manner to a clamping function.

Let us assume that we want to dedicate Δl_{x} pixels vertically and Δl_{y} pixels horizontally on the display for the purpose of the transition. These values define a desired size of the transition area on the target image. We can use these parameters to obtain the respective size in the space of the source image (with the scale factor being S):

$$\Delta l'_{\mathrm{x}} = \Delta l_{\mathrm{x}}/S\,, \qquad \Delta l'_{\mathrm{y}} = \Delta l_{\mathrm{y}}/S\,. \tag{21}$$

Both values $\Delta l'_{\mathrm{x}}$ and $\Delta l'_{\mathrm{y}}$ have to be limited to $[0,1]$ range to prevent exceeding size of a pixel in the source image. If these values are zeros, then the nearest neighbor should be used instead of the desired interpolation.

Once the relative size of the transition area have been computed (and limited if that was needed), the second pair of parameters can be computed:

$$l'_{\mathrm{x}} = \frac{1-\Delta l'_{\mathrm{x}}}{2}\,, \qquad l'_{\mathrm{y}} = \frac{1-\Delta l'_{\mathrm{y}}}{2}\,. \tag{22}$$

Finally, new coordinates can be obtained:

$$x'' = \frac{x'-l'_{\mathrm{x}}}{\Delta l'_{\mathrm{x}}}\,, \qquad y'' = \frac{y'-l'_{\mathrm{y}}}{\Delta l'_{\mathrm{y}}}\,. \tag{23}$$

These values have to be limited to $[0,1]$ range as well. In this case negative values and values larger that one correspond to 'snapping' to the nearest defined point. The new coordinates replace the old ones when computing interpolation coefficients, i.e. we have $W(x''-x'_p, y''-y'_r)$ instead of $W(x'-x'_p, y'-y'_r)$ in (8).

TAR can improve sharpness of the images very well. Moreover, it ties the new transition area to the effective target image size. That way it allows for a consistent perceived sharpness regardless of the resolution of the target display. However, it is worth mentioning, that for the smaller scale factors than the desired size of the transition area this technique would simply not work — there would be not enough pixels left in the first place.

5. Extensible software implementation

An application was created by us to test the proposed algorithms on bitmaps. The application was created with .NET Framework and works both is the command line and as a dynamically linked library (DLL). The prime concepts behind its design were availability, transparency, and extensibility.

The first big idea was that this project needed to be easy to use in further research. Moreover the upgrades and modifications to the program created by others should not be difficult to publish and share. Because of that the application was made available in multiple ways. Firstly, its code was intended to be open and publicly available. Moreover, thanks to .NET the software is cross-platform. The code either uses functions provided with .NET or has its own, causing no licensing issues.

Another important issue was to ensure that further improvements over the existing functions should be easy to be made by anyone. Such transparency allows others to analyze the algorithms and approaches with ease, which is beneficial in further work. The transparency of the application comes from documentation of its code. Both its functions and the main code for the command-line functionality are commented, making it easy to understand the whole resizing process. Additionally, the structure of the program helps in following what are the responsibilities of each part of the code.

The extensibility of the software benefits from the transparency, because modification of the program and addition of new options is supposed to be a simple and clear process. This is supposed to have a positive impact on both further testing and getting insightful input from other researchers. On top of that, using the program as the library, some options, like kernels or image casting functions, can be customized on the run. This can be used for dynamic prototyping of new functions and testing them in the real life.

The application can serve well in prototyping and testing various interpolation functions. Thus the speed of the whole process was not the main concern. While the program can be capable of providing results fast, it is not optimized for any real-time scenarios. The faster processing could be achieved with inclusion of shaders, but that would impair the straightforwardness of the design (as it has been mentioned in the state-of-the-art section).

5.1. Technology

As it was mentioned, the application can be used on various platforms and that is one of the reasons behind using .NET Framework. The minimal version selected of .NET for the project was 4.5 (default for Windows 10) due to availability of `Parallel.For` function [13], which clarifies the code in the part regarding the image processing. Parallel processing isn't required for the program to be functional, but is beneficial for computational time. The way the processing functions are designed is aimed at separating the exact interpolation functions from the processing, so inclusion of the parallel processing shouldn't be a problem. Of course, because the way the parallel computations are implemented it's hard to rely on them for comparison of processing time. Because of that an option to disable the parallel processing was also included.

Another benefit of .NET is an included support for various bitmap formats both for reading and for writing [12]. However, the default bitmap class cannot be accessed by multiple threads. Because of that each read bitmap is cast into a simple three-dimensional array, which is encapsulated into its own class.

The project was designed to work as both console application and as a DLL. This approach allows for the application to be used in the most preferable manner. While the latter approach allows for prototyping in the run-time, the former allows for a straightforward access to the implemented functions. The library mode allows for more flexibility and exposes most parts of the project, but users have to construct and run the scaling process themselves. In case of the command line mode the process is constructed by the application and all required information has to be provided as parameters of the program.

The parameters are gathered in the code in arrays of strings which are used for matching and recognizing the parameters and options. Some exclusive options, like interpolation method, use two arrays — one for acceptable parameter names and the other for a finally-assigned labels. New parameters can be included by adding new arrays and providing a support for their values in the code.

The second part of the the program is the library part. Thanks to .NET rules, an executable file can work as a dynamically linked library by making its functions public. providing more functionality than the command-line variant. New options, like interpolation or transformation functions, can be added in the code. Some of the new options can be even obtained by using functions provided by the library. Such approach allows for an easy prototyping. The main reasoning behind exposing the program functions in this manner was providing a convenient way for experimenting with scaling methods. While such access to be useful requires creating a separate application, the client program could be tailored to the needs of actual prototyping.

The image resizing is handled by a dedicated `Resizer` class. The class takes the input image, an empty output image, and an interpolation setup (which includes casting function, TAR, PBCC, and interpolation functions). The resizing process can work with an interpolation kernel of any finite size and various transformations (like rotations) can be turned into proper casting functions. However, the excessive points, like in case of rotation, have to be removed in post-processing (after the `Resizer` object provides its output).

Figure 5 shows an outline for the resizing process. For each point of the output image a respective location within the source image is computed. The location is then turned into a structure of positioning values: the reference point and the relative position to that point. The relative coordinates can be modified with TAR. Then the coordinates, whether restricted or not, are used to obtain the interpolation coefficients. These coefficients together with the restricted coordinates are then processed with PBCC. The final coefficients are then multiplied with values of the respective pixels of the original image, whose locations are obtained from the location of the reference point. The result of the multiplication is then written into right pixel of the output image.

The way this process is implemented recreates the whole proposed way TAR combined with PBCC. This step-by-step processing allows to affect how the values are computed at each stage, usually allowing to provide a new function. TAR in this situation can be substituted with any function that is supposed to modify positions only for the purpose of computing the coefficients. In a similar manner PBCC can be replaced with any method that modifies the coefficients after obtaining them. The

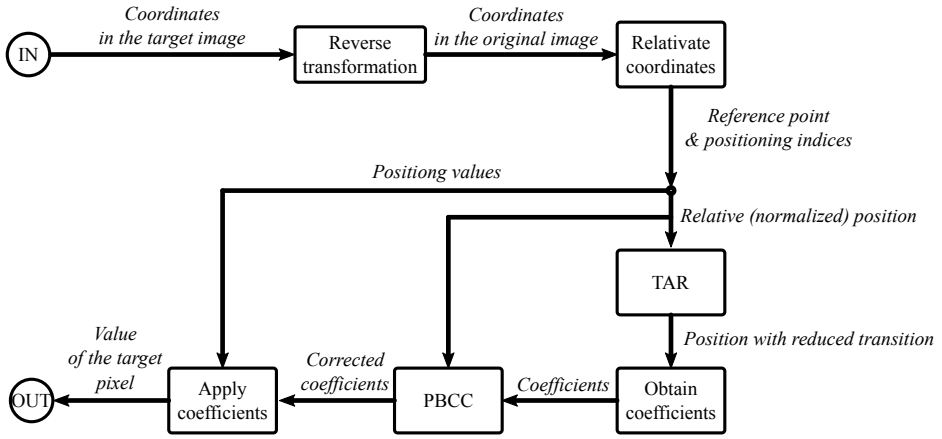


Figure 5. An outline of the resizing process implemented in the `plin` Project

presented approach turns the action of computing coefficients into a tree-stage process:

- non-transitioning shift of positions (it does not affect the real positions, just how they are provided to the kernel),
- kernel,
- coefficient reevaluation.

The available control over the process allows at implementing any new scaling functions and test them in the same environment as the existing ones. The only drawback of this approach is that it requires to use the program in the library mode or to recode it.

The `Resizer` class involves an optional optimization process. This process selects one of many variants of the resizing functions. The selection depends on the information on: how many colors image has, are the coefficients buffered, whether TAR or PBCC should be applied, and even if the whole process is allowed to be computed using `Parallel.For`. Every variant of the resizing process is stripped to include only the needed functions and operations. While at the existing point this is not required, further modifications might tackle this part of the program to provide even more flexibility and control options to the users.

5.2. Functions

We can distinguish four basic fields related to how the program works and scales images. First and foremost are casting operations, which are used to obtain position

of the new point in the source image. We can simply distinguish as a special field transformation operations, which after setting up the desired sequence of operations, can be turned into casting function. After obtaining the position of the point, its value needs to be interpolated, which is another field of operations of our library. The last field of our program is the exact processing of the image by applying those operations. The library part of the project was divided into four namespaces related to the aforementioned fields:

- `Casting` — supplies basic structures for point coordinates and positioning of interpolation coefficients. It also provides definition for delegates [11] of casting, blending and transition functions. It also gives methods for composing one-dimensional functions into two-dimensional ones, generating casting and blending functions.

- `Transformations` — provides a definition of an affine transformation matrix and allows for a creation of such a matrix by stating desired operations or by composing them.

- `Interpolations` — is responsible for delegates needed for interpolation functions and their compositions. It also provides support for PBCC and TAR. Moreover, it defines an interpolation setup object class needed for the processing.

- `Processing` — provides object class for image processing and its optimization.

Whenever it is stated that the library provides a function, it means the library generates the function. The generation process returns an anonymous function that works accordingly to the proper delegate type. Thanks to that the whole process works exactly the same regardless of origins of used functions, allowing for more reliable comparison of processing time.

The `Casting` namespace is supposed to cover all all structures and definitions required to apply casting functions. The most important are definitions of casting functions, structures to hold related data and means to compose those together functions. The most notable options provided by `Casting` namespace are:

- `InterpPoint1D` and `InterpPoint2D` structures are responsible for providing an integer position — an index of the reference pixel — and a float value — the displacement from the reference point or the normalized position of the interpolated point.

- A static `Function` class is responsible for combing one-dimensional casting functions into two-dimensional ones. The class is also used for a generation of the most simple casting methods from provided scale factors or from transformation matrices. Moreover, the class can generate blending functions (which are used to clip the image after processing) from the casting functions. For this purpose it supplies three basic blending functions — a linear one, a point one, and one based on p-lin. This class also provides functions for creating an array of the buffered coefficients.

When we want to design more complex transformations than just image scaling, it is much easier using transformation matrices and composing them than trying to design simple casting functions. For that reason the whole `Transformation` namespace was made in separation to the Casting namespace. The `Transformation` namespace consists of:

- `TransformationMatrix` — a structure of the affine transformation matrix with methods for composing it with other transformation matrices and applying it to points. It also provides a definition of a unity transformation.
- `TransformationPrototype` is a structure for storing the matrices for transformations from the source image to the target image and back. It also holds dimensions of both the original and the target image.
- `TransfomationSetup` — a structure for designing an affine transformation from provided operations — translation, scaling, rotation and expansion (addition of empty pixels on sides of the bitmap) in that order. The expansion can also be moved to be the first operation. The structure also provides static methods for obtaining size of the bitmap after rotation.

The most important part of the presented idea is image interpolation which transforms position of a point into weights that are applied to pixels of the source image in order to obtain the value at the interpolated point. We proposed the `Interpolation` namespace which provides definitions of functions to all three stages of interpolation. It also provides means to store the setup for the processing and means to generate it to make creation of the setup easier. The `Interpolations` namespace provides:

- `Interpolate1D` and `Interpolate2D` delegates for functions that return an array of coefficients from the normalized position of the interpolated point. The indexes (coordinates) of the pixels to which these factors are supposed to be applied are provided by positioning functions defined by `Interp1DStart` and `Interp2DStart` delegates.
- PBCC is supposed to be handled by functions of `PBCCFunc` delegates. They take an array of coefficients and the normalized position to return an array of corrected coefficients. Handling of TAR is provided with `TransitionReductionFunc` delegate, which transforms the normalized coordinates of the interpolated point.
- A static class `Interpolations` provides enumerations of the PBCC options and defined in the library interpolation functions (linear, nearest neighbor, and p-lin). It gives methods for obtaining two-dimensional interpolations from one-dimensional ones — both for the interpolation functions and the positioning functions. It can generate interpolations from the selection of the defined ones — linear, nearest, and p-lin, — or from a transition function. The class also is capable of generating basic positioning function for these interpolations and of creating PBCC and TAR functions.

- `Interp` class is designed to store the delegates required for the scaling process along with the dimensions of both the source and target images. It also provides methods for simplified assignment of its parameters from provided input informations.

The `Processing` namespace was made to apply the the image transformation and interpolation process presented in this paper and consists of:

- `Image` class is designed to store a bitmap array and provide access to it. It allows for an arbitrary quantity of color channels, but it also gives preassigned indexes for layers of the classic RGB modes. For reading out-of-bounds pixels it is assumed that the border pixels are expanded infinitely (which is sufficient for the provided interpolations).
- The aforementioned `Resizer` class takes `Image` objects of both the source and the target images and the `Interp` interpolation configuration. It provides an optimization function which selects the most efficient resizing method. It also gives a support for coefficient buffering. Its `Resize` method handles the rescaling process safety, however there are options for forcing parallelism or for doing the resize without ensuring all provided objects are compatible.

5.3. Extensibility and use

The most convenient approach to program extension would be keeping the approach to the structure of the code intact. That's why the new options can be included in the library code by simply adding them in the same manner as the existing options were implemented. In most cases it would require an addition of a method that returns an anonymous function. It is worth remembering that addition of new kernels should be accompanied with an inclusion of a proper positioning function (if the existing ones do not fit the kernel already). If it is needed, the existing functions, like the initializations of the `Interp` object might be modified to allow for a simpler use of the new possibilities.

The expansion of features available in the library part should be followed by providing access to those options in the command-line part. That way the accessibility of the project is preserved and the new options can be easily tested. The manner in which the parameters are introduced allows them to be very customizable. The options can behave like toggles, can take input parameters, or even allow for more than one tag for them. That put an requirement for providing a system to parse the arguments, compare them with a list of possible names and include a procedure of obtaining the input value from them.

The new options can also be added to the main program and the whole support for them is handled with string arrays. The command-line arguments are, at first, handled by the system, which separates them on the basis of position of quotation marks and then on position of empty spaces. Next, the arguments are separated into two groups — basic (the input filename, the desired size, and the output filename) and special (e.g. used interpolation function, rotation). This distinction separates the

required parameters - input, output and scale - from the optional ones. Moreover, the optional parameters can allow for additional input values to customize their behavior. The difference is that the special options are preceded with a dash mark '-' and the basic are not. The new options should always be added as the special ones.

The special parameters are then split into two parts — an argument and its extension. The split is done according to position of a separator sign (colon or equality sign by default). Then the argument part can be matched and parsed. The matching should be done by `Array.IndexOf<string>` method [10] — the returned index indicates which position of the argument array was matched and negative value tells that there was no match. In case of detection of the argument, proper information has to be provided to the program — a meaningful value has to be assigned to a right variable. After the testing of all arguments these variables are used to setup the scaling operation.

If an option has exclusive variants (like the interpolation method), the option array is accompanied with another of the same length. The second array should contain tags that represent the presented variant. These tags can appear more than once if an argument has more than one variant. The index, that was a result of matching the argument, is going to be the same as the one of the tag to assign.

The extension of the argument is a specific parameter for the selected argument. The extension can be matched the same way the arguments are and the program already has a support for boolean options. If the extension is a number to parse, the program already provides encapsulated culture-invariant parsing functions for single- and double-precision floating point numbers.

As it has already been mentioned, the program can serve as a library. Because all image processing functions used by the program were assigned `public` specifier, they are available outside of the command line. There are two ways of accessing them. The first one is to include the executable the same way as custom .NET libraries are added to the project. The second one is to dynamically link them (the default .NET linking process works the same way in this case). That way there are even more options available than from the command line. What is more, some options, like new kernels or transformations can be used this way without resorting to modify the code of the library. Interestingly, the same mechanisms allow for other kinds of operations to be conducted with the library, like filtering.

5.4. Sample use: plin_showcase

To showcase the possibilities of the project, an example application was created (Fig. 6). The goal of this application — 'plin_showcase' — was to provide an outlook on how each of the implemented interpolations and their modifiers work. The application was made as a Windows Form application, allowing for an easy access to all options and for an overview of the outcome.

The program allows for reading a bitmap and for increasing its size by any scale factor (even fractional). One of three kernels can be used for scaling — the nearest neighbor, the linear interpolation, or the p-lin interpolation. Moreover, there is an

option to turn on PBCC and set the value of TAR. As an addition, the image can also be rotated.

The `plin` project is linked as a project library in this case and all options are accessed that way.

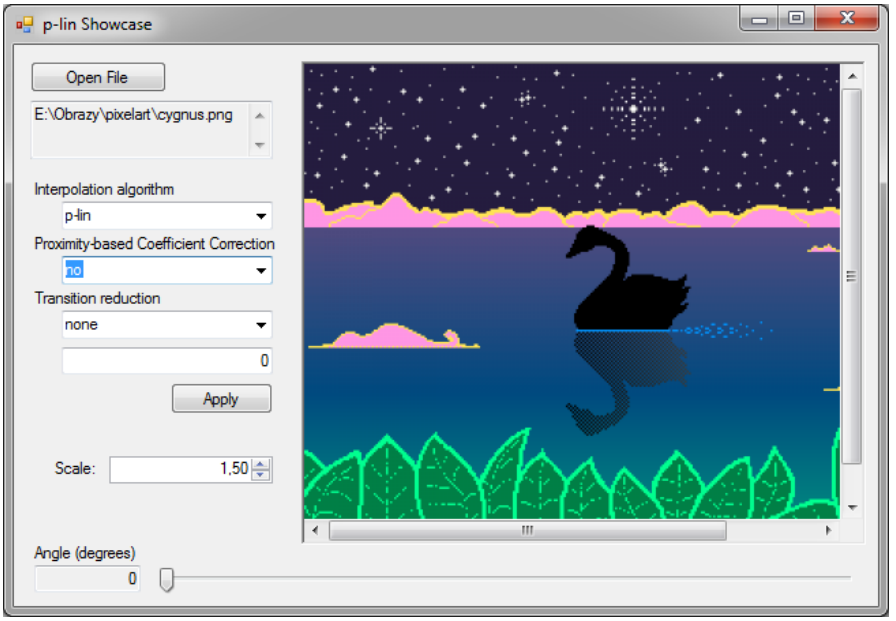


Figure 6. The look of the plin_showcase apllication.

6. Experimental results of interpolation

The proposed algorithms can be compared with the classic methods for the computation time. All processing was done using the presented project. The times are shown in Table 2. As it can be clearly seen, more refined techniques needed more time to process the image. The use of PBCC brought the biggest increase (about 15-16%). On the other hand, TAR did not affect the processing time significantly. The p-lin interpolation seems to provide as much time increase as TAR set to 2. The increase in time is less significant for the smaller scale factor.

Let us now look at the results of bitmap scaling. Figure 7 presents results of scaling a *lorem* bitmap [22] with various methods. The bitmap was created for the purpose of testing the project. On the top there is the source image, below are the results of scaling up selected areas.

It can be clearly seen that for smaller scale factors the nearest neighbor method (Fig. 7b) has problems with keeping the squared shapes of the original elements, causing visible deformations. The effect, even if less strong, is still visible for larger

Table 2. Processing time for various scaling methods using `plin` Project. Median and mean time of 10 cycles were provided. The median value was used to compensate for unexpected background operations running on the testing computer. The processed image was of size 640×360. Done without parallel computations.

Method	**PBCC**	**TAR**	**Median time [ms]**	**Mean time [ms]**
Scale factor $\times 1.5$				
nearest	no	no	109.2	107.6
linear	no	no	109.2	109.2
linear	yes	no	124.8	125.9
linear	no	1	109.2	110.8
linear	no	2	109.2	109.2
linear	yes	2	124.8	128.3
p-lin	no	no	109.2	109.2
p-lin	no	2	109.2	109.2
Scale factor $\times 4$				
nearest	no	no	821.0	821.3
linear	no	no	828.8	839.2
linear	yes	no	998.4	998.4
linear	no	1	842.4	849.3
linear	no	2	842.4	851.1
linear	yes	2	998.4	1003.6
p-lin	no	no	858.0	852.8
p-lin	no	2	858.0	863.2

(a) Source

(b) Nearest neighbor (×1.5)

(c) Nearest neighbor (×3.5)

(d) Linear (×1.5)

(e) Linear (×3.5)

(f) P-lin (×1.5)

(g) P-lin (×3.5)

(h) Linear + PBCC (×3.5)

(i) Linear + TAR set to 2 (×3.5)

(j) Linear + PBCC + TAR set to 2 (×3.5)

(k) P-lin + PBCC (×3.5)

(l) P-lin + TAR set to 2 (×3.5)

(m) P-lin + PBCC + TAR set to 2 (×3.5)

Figure 7. The source image *lorem* and results of scaling up parts of it with `plin` Project. The fragments scaled with a factor of 1.5 were sampled up two times for readability.

scale factors (Fig. 7c). It is worth noting, that this method provides perfect contrast and does not add any new values to the image.

The linear interpolation does not have a problem with providing undistorted shape regardless of the scale factor (Fig. 7d and 7e). However, the method introduces a strong blur, which is more noticeable at larger scale factors. Use of PBCC (Fig. 7h slightly improves the contrast and rounds up corners of the picture elements. Use of TAR can strongly improve contrast, especially together with PBCC, as it can be seen in Fig. 7i and 7j. Interestingly, using TAR not only provides improvement in contrast, but also keeps the elements less deformed (thanks to the small area in which the blur is applied). An bitmap scaled this way appears on the display both to be of a rather good contrast and to contain no deformed pixels.

The p-lin interpolation is capable of giving a slightly sharper outcome without rounding up the corners (Fig. 7g). The contrast can be further improved with TAR (Fig. 7l). However PBCC does not impact this kernel with the same strength it did for the linear interpolation — with or without TAR. This can be seen in Fig. 7k and 7m. The use of TAR with p-lin can provide both very good contrast (even better than for the linear interpolation with TAR) and perception of undeformed pixels.

Similar effects can be seen for other test images — Fig. 8 [16] and 9.

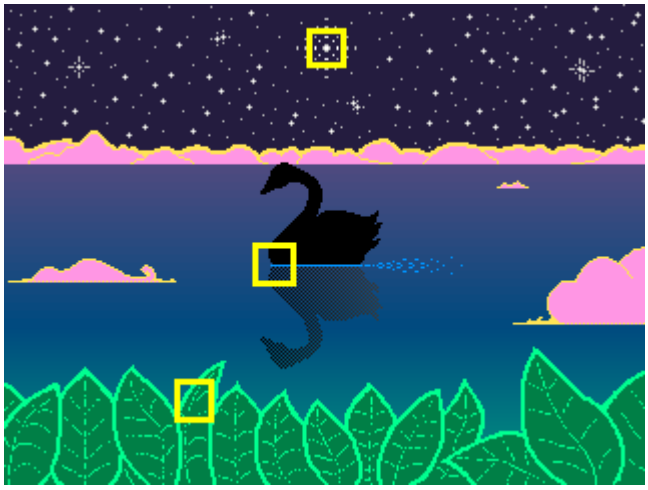
It is not an easy task to acquire objective measure of the quality of the proposed methods. Normal measures, like MSE or PSNR [17] would require comparison of the scaled image to the reference one. But as there are no reference images available, the only one available for comparison is the original one. However, scaling either bitmap would affect its features, corrupting the result of the comparison. This means that further evaluation of the methods shall be different.

The other approach would be a subjective measure of the quality. In such test viewers should be presented with the same picture scaled with various methods and rate how these techniques deal with the same problem for the changing scale factor. As it is clear that the scaling problem in this case requires balancing between sharpness and distortions, these two aspects should be assessed. Such survey is a goal of further work.

Our initial observations suggest that our methods handle scaling pictures better than the classic ones. While the impact might depend on the content — more single-pixel details and checkerboards benefit our methods — the general outcome should still point at PBCC, p-lin and TAR as improvements over the existing methods.

The most significant benefit of the proposed techniques comes from the reduction of transition area. It effectively reduces the blur of the interpolation. However, because few pixels are usually dedicated to transition between values, the distortion that appears in the case of the nearest neighbor is not present. Thanks to small size of singular pixels on the modern displays this should result in illusion of sharp-enough image without disfiguring the original image. The use of PBCC also effectively reduces the transition area a bit (both in the pure form, like in the form of p-lin). The pure PBCC furthermore introduces actual two-dimensional interpolation, which can be observed in the way it rounds the corners of the pixels.

A more in-depth analysis of our methods would require a full survey to obtain subjective reception and a mean of obtaining an objective measure of quality. The



(a) Source

(b) Nearest neighbor (×1.5)

(c) Nearest neighbor (×3.5)

(d) Linear (×1.5)

(e) Linear (×3.5)

(f) P-lin (×1.5)

(g) P-lin (×3.5)

(h) Linear + PBCC (×3.5)

(i) Linear + TAR set to 2 (×3.5)

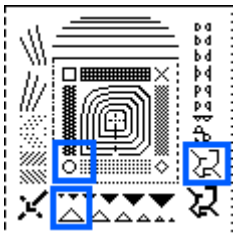
(j) Linear + PBCC + TAR set to 2 (×3.5)

(k) P-lin + PBCC (×3.5)

(l) P-lin + TAR set to 2 (×3.5)

(m) P-lin + PBCC + TAR set to 2 (×3.5)

Figure 8. The source image *cygnus* and results of scaling up parts of it with `plin` Project. The fragments scaled with a factor of 1.5 were sampled up two times for readability.



(a) Source

(b) Nearest neighbor (×1.5)

(c) Nearest neighbor (×3.5)

(d) Linear (×1.5)

(e) Linear (×3.5)

(f) P-lin (×1.5)

(g) P-lin (×3.5)

(h) Linear + PBCC (×3.5)

(i) Linear + TAR set to 2 (×3.5)

(j) Linear + PBCC + TAR set to 2 (×3.5)

(k) P-lin + PBCC (×3.5)

(l) P-lin + TAR set to 2 (×3.5)

(m) P-lin + PBCC + TAR set to 2 (×3.5)

Figure 9. The source image *bw_ test* and results of scaling up parts of it with `plin` Project. The fragments scaled with a factor of 1.5 were sampled up two times for readability.

survey would require a presentation of various outcomes of image scaling with various methods and assessment of the quality by a group of respondents. A possible mean of filtering the responses to achieve more reliable results is considered. On the other hand obtaining a reliable objective measure will require a solid mathematical approach.

Both the survey and development of the objective measure are outside of the scope of this paper, but are going to be approached in our further works.

7. Conclusions

The new interesting methods for scaling pixel art and their implementation were presented in this paper. The proposed techniques can improve quality of displaying pixel art. They achieve that by allowing for scaling up the bitmaps in a manner that is for human perception both sharp and undeformed. The propounded approaches allow for this to appear at any scale factor. Because our aim was to allow for the most unaltered experience, the flexibility of the proposed methods is an advantage. Moreover, p-lin with TAR, which seems to provide the best results, does not add a lot of additional computation time, making its real-time implementation possible.

The introduced methods were implemented in the C# project. The provided solution is a flexible, easily extensible and multi-platform application. The application was made easy-to-read and modify with a possibility of adding new interpolations without any need to recompile the program. The project might not be suited for real-time applications, but it is good enough for simple prototyping of scaling methods. Moreover, the way the project is designed, the other point and filter operations can be done using the provided library.

Future work with the project is to further improve its extensibility. In first place it would require an introduction of an interface for transformation matrices, which would allow for much more complex image casting. The class designed for storing image needs to be adapted for easy inheritance for allowing other approaches for handling border-case scenarios. Moreover, there might be a need to find a way to speed up PBCC and to allow for other coefficients- and coordinates-modifying functions.

The project can be found at https://github.com/PawelMStasik/plinProject (contains code and executables for both the main program and the showcase application).

Acknowledgements

The text of this article was prepared within the DS 2018 Project.

References

[1] Burger W., Burge M. J., Principles of digital image processing: core algorithms, Springer, pp. 210–237, 2009.

[2] Dong C., Loy C. C., He K., Tang X., Image Super-Resolution Using Deep Convolutional Networks, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 38, no. 2, pp. 295-307, 2016.

[3] GNU Octave, https://www.gnu.org/software/octave/, 2018 [on-line, accessed: 20.09.2018].

[4] Graphics Section in *GameMaker: Studio Settings for Windows*, https://help.yoyogames.com/hc/en-us/articles/216753338-GameMaker-Studio-Settings-for-Windows, YoYo Games Ltd, 2015 [on-line, accessed: 20.09.2018].

[5] Inkscape Developers, libdepixelize, https://launchpad.net/libdepixelize, 2014 [on-line, accessed: 20.09.2018].

[6] Inkscape tutorial: Tracing Pixel Art, https://inkscape.org/en/doc/tutorials/tracing-pixelart/tutorial-tracing-pixelart.html, 2018 [on-line, accessed: 20.09.2018].

[7] Kopf J., Lishinski D., Depixelizing pixel art, *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011)*, vol. 30, no. 4, 99:1-99:8, 2011.

[8] Kreuzer F., Depixelizing Pixel Art on GPUs, *Institute of Computer Graphics and Algorithms, Vienna University of Technology*, 2014.

[9] Mathworks, imresize, *Matlab Documentation*, https://www.mathworks.com/help/images/ref/imresize.html, 2018 [on-line, accessed: 20.09.2018].

[10] Microsoft Developer Network, Array.IndexOf Method, https://msdn.microsoft.com/en-us/library/system.array.indexof, 2018 [on-line, accessed: 20.09.2018].

[11] Microsoft Docs, Delegates (C# Programming Guide), https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/, 2015 [on-line, accessed: 20.09.2018].

[12] Microsoft Developer Network, ImageFormat Class, https://msdn.microsoft.com/en-us/library/system.drawing.imaging.imageformat, 2018 [on-line, accessed: 20.09.2018].

[13] Microsoft Developer Network, Parallel.For Method, https://msdn.microsoft.com/en-us/library/system.threading.tasks.parallel.for, 2018 [on-line, accessed: 20.09.2018].

[14] Nagatomi, Image Super-Resolution for Anime-Style Art, https://github.com/nagadomi/waifu2x, 2018 [on-line, accessed: 17.10.2018].

[15] OpenGL Wiki "Sampler Object", https://www.khronos.org/opengl/wiki/Sampler_Object, 2017 [on-line, accessed: 20.09.2018].

[16] Pahvl, *Cygnus*, https://pahvl.tumblr.com/post/161126319888/cygnus, 2017 [on-line, accessed: 20.09.2018].

[17] Riebman A. R., Bell R. B., Gray S., Quality Assessment for Super-Resolution Image Enhancement, *2006 International Conference on Image Processing*, pp. 2017-2020, 2006.

[18] Santhosh G_, Zoom An Image With Different Interpolation Types, *CodeProject*, https://www.codeproject.com/Articles/236394/Bi-Cubic-and-Bi-Linear-Interpolation-with-GLSL, 2011 [on-line, accessed: 20.09.2018].

[19] Silva M. A. G., Real Time Pixel Art Remasterization on GPUs with CUDA, https://github.com/marcoc2/pixel-art-remaster-gpu, 2017 [on-line, accessed: 20.09.2018].

[20] Silva M. A. G., Montenegro A., Clua E., Vasconcelos C., Lage M., Real Time Pixel Art Remasterization on GPUs, *2013 XXVI Conference on Graphics, Patterns and Images, Arequipa*, 274-281, 2013.

[21] Stasik P. M., Euclidean proximity function in image processing, *2016 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, 254-258, Poznan, 2016.

[22] Stasik P. M., Balcerek J., Improvements in upscaling of pixel art, *2017 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, 371-376, Poznan, 2017.

[23] SteamSpy, www.steamspy.com, 2017 [on-line, accessed: 01.12.2017].

[24] Steppin M., Zemek C., Gannaz F., hqx 1.1, https://code.google.com/p/hqx/, 2015 [on-line, accessed: 20.09.2018].

[25] Mitani Y., Pixel-Art Upscaler Based on pix2pix Network, https://github.com/mitaki28/pixcaler, 2018 [on-line, accessed: 17.10.2018].

[26] Zenju, xBRZ 1.6, http://sourceforge.net/projects/xbrz/, 2018 [on-line, accessed: 20.09.2018].

This paper is a revised and extended version of work originally presented at IEEE Signal Processing Algorithms, Architectures, Arrangements, and Applications Conference, 20-22 September 2017, Poznań, Poland

Received 19.06.2018, Accepted 6.12.2018