Pixel-art scaling algorithms

Pixel-art scaling algorithms are graphical filters that are often used in video game console emulators to enhance hand-drawn 2D pixel art graphics. The re-scaling of pixel art is a specialist sub-field of image rescaling.

Sprite of a television set (center) resized using simple nearest-neighbor scaling (left) and the 2xSaI interpolation algorithm (right)

As pixel-art graphics are usually in very low resolutions, they rely on careful placing of individual pixels, often with a limited palette of colors. This results in graphics that rely on a high amount of stylized visual cues to define complex shapes with very little resolution, down to individual pixels and making image scaling of pixel art a particularly difficult problem.

A number of specialized algorithms[1] have been developed to handle pixel-art graphics, as the traditional scaling algorithms do not take such perceptual cues into account.

Since a typical application of this technology is improving the appearance of fourth-generation and earlier video games on arcade and console emulators, many are designed to run in real time for sufficiently small input images at 60 frames per second. This places constraints on the type of programming techniques that can be used for this sort of real-time processing. Many work only on specific scale factors: 2× is the most common, with 3×, 4×, 5× and 6× also present.

Comparison of common pixel-art scaling algorithms. View in full resolution to see the differences.

Contents

Algorithms

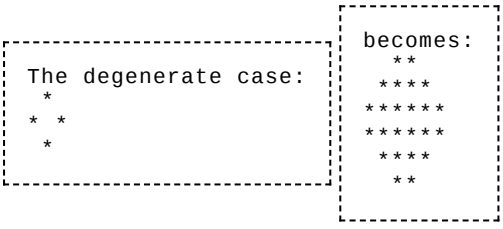
SAA5050 'Diagonal Smoothing'

The Mullard SAA5050 Teletext character generator chip (1980) used a primitive pixel scaling algorithm to generate higher-resolution characters on screen from a lower-resolution representation from its internal ROM. Internally each character shape was defined on a 5×9 pixel grid, which was then interpolated by smoothing diagonals to give a 10×18 pixel character, with a characteristically angular shape, surrounded to the top and to the left by two pixels of blank space. The algorithm only works on monochrome source data, and assumes the source pixels will be logical true or false depending on whether they are 'on' or 'off'. Pixels 'outside the grid pattern' are assumed to be off.[2][3][4]

The algorithm works as follows:

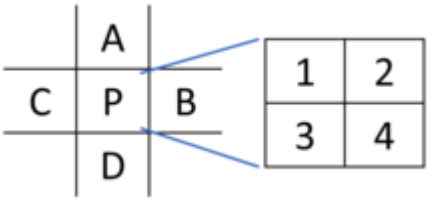
```
A B C --\ 1 2
D E F --/ 3 4

1 = B | (A & E & !B & !D)
2 = B | (C & E & !B & !F)
3 = E | (!A & !E & B & D)
4 = E | (!C & !E & B & F)
```

Note that this algorithm, like the Eagle algorithm below, has a flaw: If a pattern of 4 pixels in a hollow diamond shape appears, the hollow will be obliterated by the expansion. The SAA5050's internal character ROM carefully avoids ever using this pattern.



EPX/Scale2×/AdvMAME2×

Eric's Pixel Expansion (EPX) is an algorithm developed by Eric Johnston at LucasArts around 1992, when porting the SCUMM engine games from the IBM PC (which ran at 320×200×256 colors) to the early color Macintosh computers, which ran at more or less double that resolution.[5] The algorithm works as follows, expanding P into 4 new pixels based on P's surroundings:




```
1=P; 2=P; 3=P; 4=P;
IF C==A => 1=A
IF A==B => 2=B
IF D==C => 3=C
IF B==D => 4=D
IF of A, B, C, D, three or more are identical: 1=2=3=4=P
```

Later implementations of this same algorithm (as AdvMAME2× and Scale2×, developed around 2001) are slightly more efficient but functionally identical:

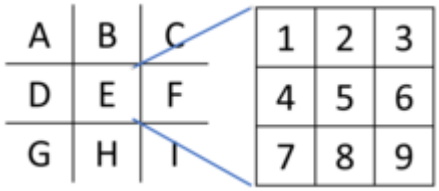
```
1=P; 2=P; 3=P; 4=P;
IF C==A AND C!=D AND A!=B => 1=A
IF A==B AND A!=C AND B!=D => 2=B
IF D==C AND D!=B AND C!=A => 3=C
IF B==D AND B!=A AND D!=C => 4=D
```

AdvMAME2× is available in DOSBox via the `scaler=advmame2x` dosbox.conf option.

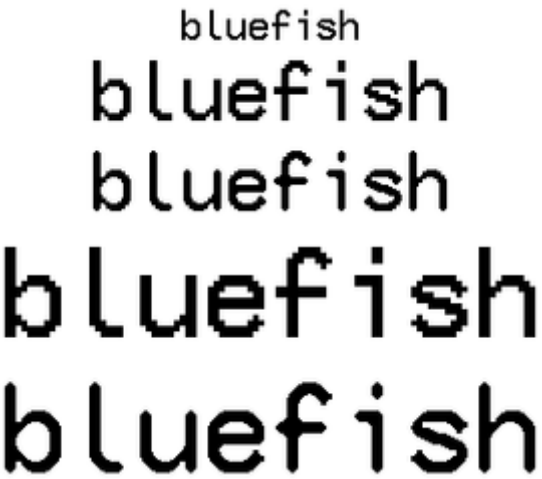
The AdvMAME4×/Scale4× algorithm is just EPX applied twice to get 4× resolution.

Scale3×/AdvMAME3× and ScaleFX

The AdvMAME3×/Scale3× algorithm (available in DOSBox via the `scaler=advmame3x` dosbox.conf option) can be thought of as a generalization of EPX to the 3× case. The corner pixels are calculated identically to EPX.




```
1=E; 2=E; 3=E; 4=E; 5=E; 6=E; 7=E; 8=E; 9=E;
IF D==B AND D!=H AND B!=F => 1=D
IF (D==B AND D!=H AND B!=F AND E!=C) OR (B==F AND
B!=D AND F!=H AND E!=A) => 2=B
IF B==F AND B!=D AND F!=H => 3=F
IF (H==D AND H!=F AND D!=B AND E!=A) OR (D==B AND
D!=H AND B!=F AND E!=G) => 4=D
5=E
IF (B==F AND B!=D AND F!=H AND E!=I) OR (F==H AND
F!=B AND H!=D AND E!=C) => 6=F
IF H==D AND H!=F AND D!=B => 7=D
IF (F==H AND F!=B AND H!=D AND E!=G) OR (H==D AND
H!=F AND D!=B AND E!=I) => 8=H
IF F==H AND F!=B AND H!=D => 9=F
```



EPX can be used to scale bitmap fonts. From top to bottom: a) original font size; b) nearest-neighbor 2× scaling; c) EPX 2× scaling; d) nearest-neighbor 3× scaling; e) EPX 3× scaling.

There is also a variant improved over Scale3× called ScaleFX, developed by Sp00kyFox, and a version combined with Reverse-AA called ScaleFX-Hybrid.[6][7][8]

Eagle

> Eagle works as follows: for every in pixel we will generate 4 out pixels. First, set all 4 to the color of the in pixel we are currently scaling (as nearest-neighbor). Next look at the three pixels above, to the left, and diagonally above left: if all three are the same color as each other, set the top left pixel of our output square to that color in preference to the nearest-neighbor color. Work similarly for all four pixels, and then move to the next one.[9]

Assume an input matrix of 3×3 pixels where the center most pixel is the pixel to be scaled, and an output matrix of 2×2 pixels (i.e., the scaled pixel)

```
first:        |Then
. . . --\ CC  |S T U  --\ 1 2
. C . --/ CC  |V C W  --/ 3 4
. . .         |X Y Z
              | IF V==S==T => 1=S
              | IF T==U==W => 2=U
              | IF V==X==Y => 3=X
              | IF W==Z==Y => 4=Z
```

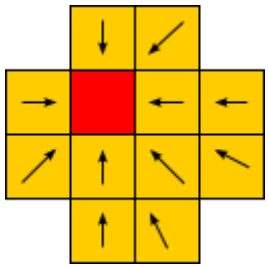
Thus if we have a single black pixel on a white background it will vanish. This is a bug in the Eagle algorithm, but is solved by other algorithms such as EPX, 2xSaI and HQ2x.

2×SaI

2×SaI, short for 2× Scale and Interpolation engine, was inspired by Eagle. It was designed by Derek Liauw Kie Fa, also known as Kreed, primarily for use in console and computer emulators, and it has remained fairly popular in this niche. Many of the most popular emulators, including ZSNES and VisualBoyAdvance, offer this scaling algorithm as a feature. Several slightly different versions of the scaling algorithm are available, and these are often referred to as *Super 2×SaI* and *Super Eagle*.

The 2xSaI family work on a 4×4 matrix of pixels where the pixel marked A below is scaled:

```
I E F J
G A B K --\ W X
H C D L --/ Y Z
M N O P
```



The matrix of surrounding pixels that Super2xSaI uses to scale a single pixel.

For 16-bit pixels, they use pixel masks which change based on whether the 16-bit pixel format is 565 or 555. The constants `colorMask`, `lowPixelMask`, `qColorMask`, `qLowPixelMask`, `redBlueMask`, and `greenMask` are 16-bit masks. The lower 8 bits are identical in either pixel format.

Two interpolation functions are described:

```
INTERPOLATE(uint32 A, UINT32 B)
  if (A == B) return A;
  return (
    ((A & colorMask) >> 1)
  + ((B & colorMask) >> 1)
  + (A & B & lowPixelMask) );

Q_INTERPOLATE(uint32 A, uint32 B, uint32 C, uint32 D)
  x = ((A & qColorMask) >> 2)
```

```
    + ((B & qColorMask) >> 2)
    + ((C & qColorMask) >> 2)
    + ((D & qColorMask) >> 2);
  y = (A & qLowPixelMask)
    + (B & qLowPixelMask)
    + (C & qLowPixelMask)
    + (D & qLowPixelMask);
  y = (y >> 2) & qLowPixelMask;
  return x + y;
```

The algorithm checks A, B, C, and D for a diagonal match such that `A==D` and `B!=C`, or the other way around, or if they are both diagonals, or if there is no diagonal match. Within these, it checks for three or four identical pixels. Based on these conditions, the algorithm decides whether to use one of A, B, C, or D, or an interpolation among only these four, for each output pixel. The 2xSaI arbitrary scaler can enlarge any image to any resolution, and uses bilinear filtering to interpolate pixels.

Since Kreed released[10] the source code under the GNU General Public License, it is freely available to anyone wishing to utilize it in a project released under that license. Developers wishing to use it in a non-GPL project would be required to rewrite the algorithm without using any of Kreed's existing code.

It is available in DosBox via `scaler=2xsai` option.

hq*n*x family

Maxim Stepin's hq2x, hq3x, and hq4x are for scale factors of 2:1, 3:1, and 4:1 respectively. Each works by comparing the color value of each pixel to those of its eight immediate neighbours, marking the neighbours as close or distant, and using a pregenerated lookup table to find the proper proportion of input pixels' values for each of the 4, 9 or 16 corresponding output pixels. The hq3x family will perfectly smooth any diagonal line whose slope is ±0.5, ±1, or ±2 and which is not anti-aliased in the input; one with any other slope will alternate between two slopes in the output. It will also smooth very tight curves. Unlike 2xSaI, it anti-aliases the output.[11][8]

Image enlarged 3× with the nearest-neighbor interpolation

Image enlarged by 3× with hq3x algorithm

hq*n*x was initially created for the Super NES emulator ZSNES. The author of bsnes has released a space-efficient implementation of hq2x to the public domain.[12] A port to shaders, which has comparable quality to the early versions of xBR, is available.[13] Prior to the port, a shader called "scalehq" has often been confused for hqx.[14]

xBR family

There are 6 filters in this family: **xBR** , **xBRZ**, **xBR-Hybrid**, **Super xBR**, **xBR+3D** and **Super xBR+3D**.

xBR ("scale by rules"), created by Hyllian, works much the same way as HQx (based on pattern recognition), and would generate the same result as HQx when given the above pattern.[15] However, it goes further than HQx by using a 2-stage set of interpolation rules, which better handle more complex patterns such as anti-aliased lines and curves. Scaled background textures keep the sharp characteristics of the original image, rather than becoming blurred like HQx (often ScaleHQ in practice) tends to do. Newest xBR versions are multi-pass and can preserve small details better. There is also a version of xBR combined with Reverse-AA shader called xBR-Hybrid.[16] xBR+3D is a version with a 3D mask that only filters 2D elements.

xBRZ by Zenju is a modified version of xBR. It is implemented from scratch as a CPU-based filter in C++ .[17] It uses the same basic idea as xBR's pattern recognition and interpolation, but with a different rule set designed to preserve fine image details as small as a few pixels. This makes it useful for scaling the details in faces, and in particular eyes. xBRZ is optimized for multi-core CPUs and 64-bit architectures and shows 40–60% better performance than HQx even when running on a single CPU core only. It supports scaling images with an alpha channel, and scaling by integer factors from 2× up to 6×.

Super xBR[18][19] is an algorithm developed by Hylian in 2015. It uses some combinations of known linear filters along with xBR edge detection rules in a non-linear way. It works in two passes and can only scale an image by two (or multiples of two by reapplying it and also has anti-ringing filter). Super xBR+3D is a version with a 3D mask that only filters 2D elements. There is also a Super xBR version rewritten in C/C++.[20][8]

RotSprite

RotSprite is a scaling and rotation algorithm for sprites developed by Xenowhirl. It produces far fewer artifacts than nearest-neighbor rotation algorithms, and like EPX, it does not introduce new colors into the image (unlike most interpolation systems).[21]

Left: Original pixel art image
Middle: Image rotated using nearest-neighbor rotation algorithm
Right: Image rotated using RotSprite algorithm

The algorithm first scales the image to 8 times its original size with a modified Scale2× algorithm which treats similar (rather than identical) pixels as matches. It then (optionally) calculates what rotation offset to use by favoring sampled points which are not boundary pixels. Next, the rotated image is created with a nearest-neighbor scaling and rotation algorithm that simultaneously shrinks the big image back to its original size and rotates the image. Finally, overlooked single-pixel details are (optionally) restored if the corresponding pixel in the source image is different and the destination pixel has three identical neighbors.[22]

Fast RotSprite

Fast RotSprite is a fast rotation algorithm for pixel art developed by Oleg Mekekechko for Pixel Studio (https://play.google.com/store/apps/details?id=com.PixelStudio) app. It's based on RotSprite but has better performance with slight quality loss. It's able to process larger images in realtime. Instead of the 8× upscale, Fast RotSprite uses single 3× upscale. Then it simply rotates all pixels with rounding coordinates. Finally, it performs 3× downscale without introducing new colors. As all operations on each step are independent, they can be done in parallel to greatly increase performance.

Kopf–Lischinski

The Kopf–Lischinski algorithm is a novel way to extract resolution-independent vector graphics from pixel art described in the 2011 paper "Depixelizing Pixel Art".[23] A Python implementation is available.[24]

The algorithm has been ported to GPUs and optimized for real-time rendering. The source code is available for this variant.[25]

Edge-Directed Interpolation (EDI)

Edge-directed interpolation (EDI) describes upscaling techniques that use statistical sampling to ensure the quality of an image when scaling it up.[26][27] There were several earlier methods that involved detecting edges to generate blending weights for linear interpolation or classifying pixels according to their neighbour conditions and using different otherwise isotropic interpolation schemes based on the classification. Any given interpolation approach boils down to weighted averages of neighbouring pixels. The goal is to find optimal weights. Bilinear interpolation sets all the weights to be equal. Higher order interpolation methods like bicubic or sinc interpolation consider a larger number of neighbours than just the adjacent ones.

NEDI

NEDI (New Edge-Directed Interpolation) computes local covariances in the original image, and uses them to adapt the interpolation at high resolution. It is the prototypic filter of this family.[28]

EDIUpsizer

EDIUpsizer[29] is a resampling filter that upsizes an image by a factor of two both horizontally and vertically using NEDI (new edge-directed interpolation).[28] EDIUpsizer also uses a few modifications to basic NEDI in order to prevent a lot of the artifacts that NEDI creates in detailed areas. These include condition number testing and adaptive window size,[30] as well as capping constraints. All modifications and constraints to NEDI are optional (can be turned on and off) and are user configurable. Just note that this filter is rather slow

FastEDIUpsizer

FastEDIUpsizer is a slimmed down version of EDIUpsizer that is slightly more tuned for speed. It uses a constant 8×8 window size, only performs NEDI on the luma plane, and only uses either bicubic or bilinear interpolation as the fall back interpolation method.

eedi3

Another edge-directed interpolation filter. Works by minimizing a cost function involving every pixel in a scan line. It is slow.

EEDI2

EEDI2 resizes an image by 2× in the vertical direction by copying the existing image to 2·y(n) and interpolating the missing field. It is intended for edge-directed interpolation for deinterlacing (i.e. not really made for resizing a normal image, but can do that as well). EEDI2 can be used with both TDeint and TIVTC, see the discussion link for more info on how to do this.[31]

SuperRes

The SuperRes[32] shaders use a different scaling method which can be used in combination with NEDI (or any other scaling algorithm). This method is explained in detail here.[33] This method seems to give better results than just using NEDI, and rival those of NNEDI3. These are now also available as an MPDN renderscript.

NNEDI

NNEDI is a family of intra-field deinterlacers which can also be used to enlarge images by powers of two. When being used as a deinterlacer, it takes in a frame, throws away one field, and then interpolates the missing pixels using only information from the kept field. There are so far three major generations of NNEDI.

NNEDI, the original version, works with YUY2 and YV12 input.[34] NNEDI2 added RGB24 support and a special function `nnedi2_rpow2` for upscaling. NNEDI3 enhances NNEDI2 with a predictor neural network. Both the size of the network and the neighborhood it examines can be tweaked for a speed-quality tradeoff:[35]

> This is a quality vs speed option; however, differences are usually small between the amount of neurons for a specific resize factor, however the performance difference between the count of neurons becomes larger as you quadruple the image size. If you are only planning on

doubling the resolution then you won't see massive differences between 16 and 256 neurons. There is still a noticeable difference between the highest and lowest options, but not orders of magnitude different.[36]

References

1. "Pixel Scalers" (http://www.datagenetics.com/blog/december32013/index.html). Retrieved 19 February 2016.
2. "Mullard SAA5050 Datasheet" (https://amigan.yatho.com/saa5050.pdf) (PDF).
3. "SAA5050 Smoothing source code from the MAME project" (https://github.com/mamedev/mame/blob/master/src/devices/video/saa5050.cpp#L445). *GitHub*.
4. "Forum post showing Teletext reference test page on SAA5050 chip" (https://forums.bannister.org/ubbthreads.php?ubb=showflat&Number=102137#Post102137).
5. Thomas, Kas (1999). "Fast Blit Strategies: A Mac Programmer's Guide" (http://www.mactech.com/articles/mactech/Vol.15/15.06/FastBlitStrategies/index.html). *MacTech*.
6. libretro. "common-shaders/scalenx at master · libretro/common-shaders · GitHub" (https://github.com/libretro/common-shaders/tree/master/scalenx). *GitHub*. Retrieved 19 February 2016.
7. "ScaleNx - Artifact Removal and Algorithm Improvement [Archive]" (https://web.archive.org/web/20160527015550/https://libretro.com/forums/archive/index.php?t-1655.html). Archived from the original (https://libretro.com/forums/archive/index.php?t-1655.html) on 2016-05-27. Retrieved 2016-05-27.
8. "PixelArt Scalers" (https://github.com/janert/pixelscalers). Retrieved 12 October 2022.
9. "Eagle (idea)" (http://everything2.com/index.pl?node_id=1859453). *Everything2*. 2007-01-18.
10. "Kreed's Homepage: 2xSaI" (https://vdnoort.home.xs4all.nl/emulation/2xsai/). Retrieved 25 April 2020.
11. Stepin, Maxim. "hq3x Magnification Filter" (https://web.archive.org/web/20070703061942/http://www.hiend3d.com/hq3x.html). Archived from the original (http://www.hiend3d.com/hq3x.html) on 2007-07-03. Retrieved 2007-07-03.
12. Byuu. Release announcement (http://nesdev.parodius.com/bbs/viewtopic.php?p=82770#82770) Accessed 2011-08-14.
13. libretro. "common-shaders/hqx at master · libretro/common-shaders · GitHub" (https://github.com/libretro/common-shaders/tree/master/hqx). *GitHub*. Retrieved 19 February 2016.
14. Hunter K. "Filthy Pants: A Computer Blog" (http://filthypants.blogspot.com/2014/06/true-hq2x-shader-comparison-with-xbr.html). Retrieved 19 February 2016.
15. "xBR algorithm tutorial" (https://forums.libretro.com/t/xbr-algorithm-tutorial/123). 2012-09-18. Retrieved 19 February 2016.
16. libretro. "common-shaders/xbr at master · libretro/common-shaders · GitHub" (https://github.com/libretro/common-shaders/tree/master/xbr). *GitHub*. Retrieved 19 February 2016.
17. zenju. "xBRZ" (https://sourceforge.net/projects/xbrz/). *SourceForge*. Retrieved 19 February 2016.
18. "Super-xBR.pdf" (https://drive.google.com/file/d/0B_yrhrCRtu8GYkxreElSaktxS3M/view?pref=2&pli=1). *Google Docs*. Retrieved 19 February 2016.
19. libretro. "common-shaders/xbr/shaders/super-xbr at master · libretro/common-shaders · GitHub" (https://github.com/libretro/common-shaders/tree/master/xbr/shaders/super-xbr). *GitHub*. Retrieved 19 February 2016.
20. "Super-XBR ported to C/C++ (Fast shader version only))" (http://pastebin.com/cbH8ZQQT). 6 March 2016.

21. "RotSprite" (http://info.sonicretro.org/RotSprite). *Sonic Retro*. Retrieved 19 February 2016.
22. "Sprite Rotation Utility" (http://forums.sonicretro.org/index.php?showtopic=8848&st=15&p=159754&#entry159754). *Sonic and Sega Retro Message Board*. Retrieved 19 February 2016.
23. Johannes Kopf and Dani Lischinski (2011). "Depixelizing pixel art" (http://johanneskopf.de/publications/pixelart/). *ACM Transactions on Graphics*. SIGGRAPH. **30** (4): 99:1–99:8. doi:10.1145/2010324.1964994 (https://doi.org/10.1145%2F2010324.1964994). Retrieved 2016-05-22.
24. Vemula, Anirudh; Yeddu, Vamsidhar (29 April 2019). "Pixel-Art: We implement the famous "Depixelizing Pixel Art" paper by Kopf and Lischinski" (https://github.com/vvanirudh/Pixel-Art). *GitHub*.
25. Kreuzer, Felix; Kopf, Johannes; Wimmer, Michael (2015). "Depixelizing Pixel Art in Real-time" (https://www.cg.tuwien.ac.at/research/publications/2015/KREUZER-2015-DPA). *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*. ACM: 130. doi:10.1145/2699276.2721395 (https://doi.org/10.1145%2F2699276.2721395). ISBN 9781450333924.
26. "Edge-Directed Interpolation" (http://chiranjivi.tripod.com/EDITut.html). *chiranjivi.tripod.com*. Retrieved 2019-05-07.
27. "Shader implementation of the NEDI algorithm - Doom9's Forum" (https://forum.doom9.org/showthread.php?s=7fb2fb184cfe82b7d76b63bb26df481a&t=170727). *forum.doom9.org*. Retrieved 2019-05-07.
28. Li, Xin (2010-11-26). "New Edge-Directed Interpolation" (https://web.archive.org/web/20101126091759/http://neuron2.net/library/nedi.pdf) (PDF). Archived from the original (http://neuron2.net/library/nedi.pdf) (PDF) on 2010-11-26. Retrieved 2019-05-07.
29. tritical's Avisynth Filters (https://web.archive.org/web/20110811050103/http://web.missouri.edu/~kes25c/)
30. "Archived copy" (https://web.archive.org/web/20041221052401/http://www.cs.ucdavis.edu/~bai/ECS231/finaltzeng.pdf) (PDF). *www.cs.ucdavis.edu*. Archived from the original (http://www.cs.ucdavis.edu/~bai/ECS231/finaltzeng.pdf) (PDF) on 21 December 2004. Retrieved 12 January 2022.
31. "TDeint and TIVTC - Page 21 - Doom9's Forum" (http://forum.doom9.org/showthread.php?p=744308#post744308). Retrieved 19 February 2016.
32. "nnedi3 vs NeuronDoubler - Doom9's Forum" (http://forum.doom9.org/showthread.php?t=170661). Retrieved 19 February 2016.
33. "Shader implementation of the NEDI algorithm - Page 6 - Doom9's Forum" (http://forum.doom9.org/showthread.php?p=1685124#post1685124). Retrieved 19 February 2016.
34. "NNEDI - intra-field deinterlacing filter - Doom9's Forum" (http://forum.doom9.org/showthread.php?t=129953). Retrieved 19 February 2016.
35. "Nnedi3" (http://avisynth.nl/index.php/Nnedi3). *AviSynth*. Retrieved 2019-05-07.
36. tritical (2019-04-30), *nnedi3 - Readme.txt* (https://github.com/jpsdr/NNEDI3), retrieved 2019-05-07

See also

- libretro - implements many aforementioned algorithms as shaders
- pixelscalers (https://github.com/janert/pixelscalers) - C++ implementations of ScaleNx, hqNx, and superXBR algorithms in a stand-alone tool