# MMPX Style-Preserving Pixel-Art Magnification

Morgan McGuire
University of Waterloo & NVIDIA

Mara Gagiu
University of Waterloo


(a) Input Image


(b) Magnified 2× with Nearest


(c) Magnified 2× with MMPX

**Figure 1**. (a) Pixel art combining sprites, text, and UI elements (b) magnified with nearest-neighbor filtering becomes blocky relative to the original pixel size; (c) MMPX filtering refines lines, curves, and patterns for the new resolution, while mostly preserving the palette, transparency, and shape aspects of the original artwork.

*Input derived from Oryx Design Lab licensed sprites https://www.oryxdesignlab.com/products/16-bit-fantasy-tileset*

## Abstract

We present MMPX, an efficient filter for magnifying pixel art, such as 8- and 16-bit era video-game sprites, fonts, and screen images, by a factor of two in each dimension. MMPX preserves art style, attempting to predict what the artist would have produced if working at a larger scale but within the same technical constraints.

Pixel-art magnification enables the displaying of classic games and new retro-styled ones on modern screens at runtime, provides high-quality scaling and rotation of sprites and raster-font glyphs through precomputation at load time, and accelerates content-creation workflow.

MMPX reconstructs curves, diagonal lines, and sharp corners while preserving the exact palette, transparency, and single-pixel features. For general pixel art, it can often preserve more aspects of the original art style than previous magnification filters such as nearest-neighbor, bilinear, HQX, XBR, and EPX. In specific cases and applications, other filters will be better. We recommend EPX and base XBR for content with exclusively rounded corners, and HQX and antialiased XBR for content with large palettes, gradients, and antialiasing. MMPX is fast enough on embedded systems to process typical retro 64k-pixel full screens in less than 0.5 ms on a GPU or CPU. We include open source implementations in C++, JavaScript, and OpenGL ES GLSL for our method and several others.

 ISSN 2331-7418

## 1. Introduction

### 1.1. Pixel Art

In the graphics and gaming community, "pixel art" refers to rasterized graphics elements including sprites, fonts, and whole framebuffer images consistent with the technical constraints of 8- and 16-bit consumer computers from the 1980's and 1990's, which are now referred to as retro consoles. This style has remained artistically and commercially important for decades, with increased interest in recent years enabled by retro gaming platforms built on low-cost embedded processors.

Figure 1 shows an example of pixel art authored by hand (a), magnified by nearest neighbor filtering (b), and automatically magnified (c) by the MMPX filter introduced in this paper. Magnification is useful for increasing the resolution of existing assets, for scaling the rendered output of entire games, or for creating fonts and sprites that represent larger objects beside the originals while approximating a consistent style, as shown in Figure 2.

**Figure 2.** A magnified minotaur stylistically compatible with the original soldier.

Pixel art arose from limitations in retro console hardware, such as 8×8 font glyphs, 16×16 or 32×32 individual sprites, 320×240-or-smaller screens, and 256 or fewer colors. These are not hard limits on modern devices, but aesthetic guidelines. Critically, because of the scale and color limitations, image features such as character eyes or hands are often a single pixel, edges are often jagged instead of antialiased with intermediate shades, and dithering is often used to approximate shading and more colors as well as texture. This makes the art style extremely sensitive to small artifacts at the pixel scale during rendering, unlike high-resolution 3D game rendering or natural images in which individual pixels may change without significant change to perception of the whole image.

The retro consoles include the Nintendo Entertainment System/Famicom (1983), Super Nintendo (1990), Game Boy (1989), Game Boy Color (1998), and Game Boy Advance (2001); Sega Master System (1985) and Genesis/Mega Drive (1988); Atari 7800 (1986), and Commodore 64 (1982).

Interest in such platforms has not waned. New hardware and emulator products mapping pixel-art games to modern hardware include Nintendo's Switch NES/SNES Online (2018), the new Nintendo Classic NES (2016) and SNES (2017) consoles, Sega Genesis Mini (2019), PiBoy DMG (2020), Retroid Pocket 2 (2020), Odroid-GO Advance (2019). See more examples at https://obscurehandhelds.com/.

Just as the platforms that display it may be contemporary, pixel-art content itself does not necessarily originate from the 8- or 16-bit era. Many recent games and

standalone artworks adopt similar constraints as an aesthetic even when practically unconstrained by hardware. Some motivating examples are critically lauded indie games such as *Superbrothers: Sword & Sworcery EP* (2011), *Towerfall* (2013), *Undertale* (2015), *Crawl* (2017) and *Celeste* (2018). There are hundreds of modern pixel art games for sale on the Steam, itch.io, Epic, Apple, and Android game stores.

"Fantasy consoles"–emulators for newly-designed retro hardware specifications–are currently popular for hobbyist development, game jams, and educational use. These adopt restrictions similar to the real retro consoles but integrate more modern development environments. The constraints reduce the complexity of game development to make the platforms more accessible for new programmers and help limit scope for projects. Specifically, 8-bit and 16-bit era style pixel art presents a Goldilocks zone. The constraints ensure that almost anyone can draw something such as an $8 \times 8$, 16-color sprite in reasonable in a short period of time. Yet, the design space is rich enough that good artists can still draw beautiful images. In contrast, the tighter constraints of earlier platforms such as Atari 2600 tend to produce art that is less appealing to a modern audience, and the looser ones of later Amigas leave too much freedom, in which amateurs can get lost. Examples of platforms targeting this zone include PICO-8 (2014), TIC-80 (2020), Pixel Vision 8 (2017), and our own quadplay✜ (2018).

## 1.2. Magnification Filters

Three important, common, and non-trivial rendering operations on pixel art are magnification (scaling up), minification (scaling down), and rotation at angles other than multiples of $90°$. These are non-trivial because they require resampling the source in ways that necessarily cannot exactly preserve each pixel and feature. This paper addresses magnification, which is an ongoing focus of algorithmic development within the pixel-art community, and rotation indirectly via magnification. It does not address the minification problem.

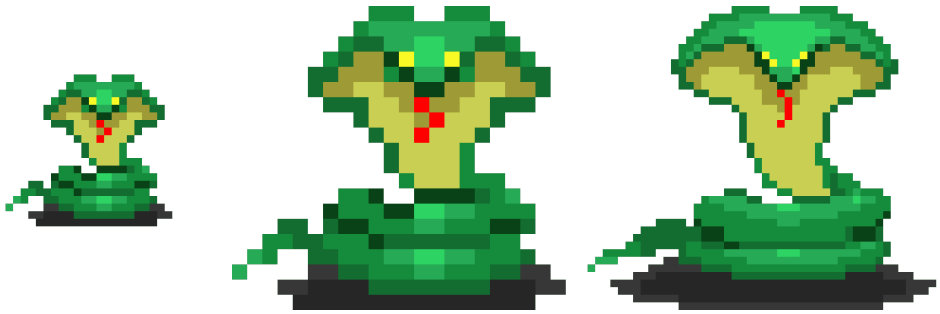



**Figure 3.** Nearest-neighbor filtering fails to take advantage of increased resolution during magnification, compared to a hand-drawn magnified image. *Derived Oryx Design Lab sprites https://www.oryxdesignlab.com/products/ultimate-fantasy-tileset*

Magnification can be posed as an image-filtering problem. The relationship between points on a continuous domain in a source and transformed destination is mathematically predetermined. The filter's role is to produce an output pixel value that compensates for the source pixel locations not aligning with its pixel grid.

The simplest algorithm for magnification is *nearest-neighbor* filtering, also known as point sampling. This maps each destination pixel center back to the source image and then samples from the nearest pixel center. Point sampling is efficient, preserves transparency, and preserves sharpness. However, it will also produce results that diverge significantly from what a human artist might create if drawing the magnified or rotated image directly. For example, under magnification the destination image will exhibit exaggerated sharpness and jagged edges on curves and lines as shown in Figure 3. Under rotation, dithering patterns will alias, single-pixel features may be lost or exaggerated, single-pixel lines may be broken, and silhouettes will have sharp projections as shown in Figure 17(a).

Any magnification filter is also a rotation filter when applied under variations of Xenowhirl's RotSprite method [2007]. Here, the source image is pre-magnified, and then filtered with nearest neighbor during combined rotation and minification back to the original scale. If the magnification filter is nearest neighbor this provides no advantage over naive rotation using nearest neighbor because it is mathematically identical. However, for more sophisticated magnification filters, this process generally produces better results than nearest-neighbor filtering of rotation on the original source. Xenowhirl describes additional heuristics that can further improve the output if applied during the rotation filtering.

We characterize a pixel-art filter as *style-preserving* if the output resembles what the original artist might have drawn were it intended as a seamless part of the original artwork. This is subjective, and it is probably impossible for any algorithm (let alone one that runs in a few nanoseconds per pixel!) to perfectly achieve. Yet many aspects of style preservation are achievable and non-controversial to judge. It is often obvious to a careful observer when one magnification filter is performing better than another for a specific piece of pixel-art content. Style-preserving properties include:

- Sharp convex and concave corners remain sharp;
- Palette preservation; destination colors are from the corresponding source region;
- Transparency is preserved;
- Curved and diagonal edges are refined to the destination resolution;
- Line and dot thickness is preserved relative to scale;
- Intersecting lines continue to intersect;
- Non-intersecting lines do not intersect.

Section 5 compares filters on images constructed to test these criteria.

### 1.3. Motivation and Contributions

We identified three important scenarios for applying pixel art filters.

*Interactive* content-creation tools accelerate artist workflow by producing a good approximation of the desired output, which an artist can then retouch to perfect. For example, Figure 2 showed a reasonable automated result for the large minotaur compared to nearest-neighbor filtering, but given the opportunity to retouch it, we would thin the outlines to better match the smaller soldier and round the nose. For the sizes of images considered in pixel art, there is little performance concern in this case and quality is the primary goal. Interactive tools are useful for creating larger or rotated fonts, background images, and sprites from initial artwork for new games or standalone pixel-art compositions. They are also useful when remastering existing pixel-art content for new display resolutions.

*Load-time* filter implementations allow automated preprocessing to generate additional sizes or orientations of sprites and fonts at high quality. That content then can be sampled with nearest-neighbor filtering without requiring expensive filtering at run-time. This case combines quality and performance targets, as the method should conserve loading time by running in tens to hundreds of milliseconds per spritesheet. There is convenience during development in preferring load-time to interactive methods, and an advantage for reducing download times and physical distribution sizes. Load-time filtering is particularly valuable for hobbyist and student programmers who are working without the benefit of artists on their development team. This case targets creation of new games with pixel-art styles.

*Run-time* filter implementation processes individual sprites or full-screen images in a few milliseconds per frame, with performance on low-end processors of paramount importance. This case is most important for systems that work with either modern or historical content that did not anticipate access to a higher-resolution display.

The authors have roles as the developer of the quadplay✤ fantasy console, an educator in game development courses, and pixel artists. As such, we were motivated by the interactive and load-time cases to develop an improved magnification filter for *new* pixel art on modern displays. Our MMPX filter has been in development and use for game jams and education through the open source quadplay✤ platform for a year. It is now stable and performant. This paper contributes implementations of MMPX in three languages, reference implementations of other popular filters for comparison, and extensive performance and quality evaluation.

Although our development focussed on the first two cases, this paper also shows in Section 6 that MMPX is sufficiently fast and high-enough quality for run-time application in retro game/hardware emulators. Yet, we offer two cautions about using any filter in this manner. The first caution is that we believe MMPX and previous magnification filters inherently produce lower-quality results when applied to composited full-screen content than when applied to the individual sprites before compositing.

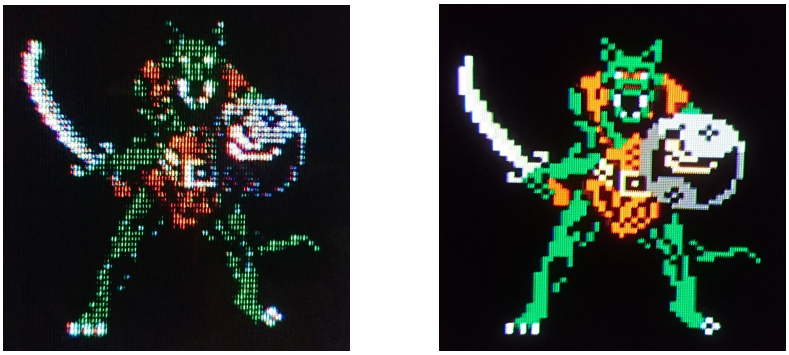


**Figure 4**. Beware that (left) content originally authored for CRT displays is misrepresented when displayed directly on (right) modern uniform, square pixel displays, and magnification filters may further distort the presentation instead of enhancing it. *Sprites from Wizardry, ©1981 Sir-Tech, image capture from https://twitter.com/MOG4791/status/886922645375139841*

That is because the individual sprites have more information, including the alpha masks separating their features and the continuation of background features which are partly obscured by the foreground after compositing. This is especially the case for consistent results under motion, where an animated foreground sprite will obscure different parts of the background and thus change the interpretation of features when processed after compositing.

The second caution is that true retro hardware and content was designed for cathode-ray tube displays. These displays often had non-square aspect ratios, and pixels on them did not appear as uniformly-filled squares but as analog "filtered" shapes closer to Gaussian splats with stronger blurring along horizontal scan lines. Artists created content with these characteristics in mind and often exploited them. Unless an emulator models such a display itself, true retro content on a modern display will be overly jagged and bright as shown in Figure 4. Display-independent magnification algorithms will not properly take frequency content and display filtering into account. They produce net results that may be attractive and useful, but one must acknowledge that the perceived image is not faithful to the original artistic intent.

## 2. Related Work

### *Natural image filters*

We review filters for the related problem of magnifying natural images (i.e., photographs) or high-resolution rendered 3D content. These are generally not appropriate for pixel art because they do not preserve style and are especially insensitive to pixel-scale features and conventions.

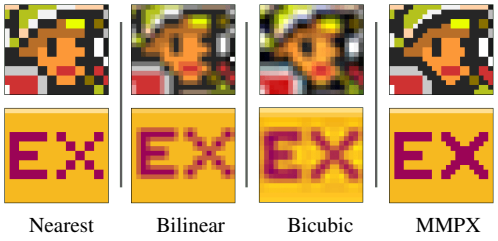



**Figure 5**. Biased bilinear and bicubic overblur.

Unbiased *bilinear* (`GL_LINEAR` in OpenGL) magnification linearly interpolates between the four nearest samples. It creates gradient ramps and diamond artifacts in the output. At 2×, every destination pixel lies exactly between four source pixels and is thus the average of four values and very blurry. *Biased bilinear* magnification shifts the destination by half a source pixel on each axis, so that one quarter of the output pixels are directly copied from the source, half are the average of two pixels, and only one quarter are the average of four pixels. This yields sharper results but, as shown in Figure 5, is still blurry and is of course offset slightly from the source. All bilinear results in this paper are the less-blurry biased version. See our supplement for unbiased bilinear results for all figures.

GPUs contain filter circuits for bilinear interpolation. However, even if the overblurring and introduction of new colors were acceptable, in practice hardware bilinear filtering cannot be used for most pixel art. That is because pixel-art images are typically encoded paletted or at four- or eight-bits per color channel. At such low precision, the alpha (i.e., transparency) channel must be unassociated instead of premultiplied, because premultiplication destroys the precision of low-alpha or low-value colors at low bit rates. Current GPUs do not support proper hardware bilinear filtering for unassociated alpha, which requires first scaling each color by its alpha value before filtering and then normalizing by the average alpha value afterwards [Glassner 2015]. This can make bilinear filtering more expensive as well as lower quality for pixel art than filters such as MMPX and EPX that perform almost no arithmetic and contain no expensive division operations. We include reference implementations and results of bilinear and biased bilinear magnification in our supplement.

*Lanczos*, *sinc*, *bicubic*, and other sharpening filters [Turkowski 1990] use a higher-order kernel than the simple tent shape of a bilinear filter in order to preserve some of the high frequencies that are attenuated by bilinear filters. This requires that they have negative filter coefficients and necessarily can produce negative pixel values as output that must be clamped to black. They also have the same issues as bilinear with transparency. While frequently preferred over bilinear for magnification of natural images, these still produce too many colors outside of the original palette, too much blurring, and tend to destroy single-pixel features (see Figure 5).

*Bilateral* filters [Tomasi and Manduchi 1998] combine a static filter kernel, such as a 2D Gaussian, with a spatially-varying mask. The mask zeroes out filter coefficients that appear to be from different features than the central pixel, determined by measuring the color difference. This can create results similar to manual airbrushing, where small blemishes in skin or noise signals are blurred away but significant feature

edges are preserved. When applied to a magnified image using a sharpening kernel, this can reduce jagged edges without overblurring in the way that bilinear filtering does.

There is a body of work on magnification or super-resolution filtering of natural images by supervised machine learning from databases of low- and high-resolution image pairs [Dai et al. 2015; Salvi et al. 2017; Xiao et al. 2020]. The NVIDIA DLSS [Burnes 2020] algorithm increases the performance of ray-traced 3D games using a sophisticated magnification filter. While the algorithm is unpublished, the marketing materials indicate that it uses a combination of temporally amortized supersampling via reprojected previous frames, machine learning inference, and traditional sharpening filters.

### *Filters developed for emulators*

Unsurprisingly, the filters developed for the games industry and emulator community tend to be the fastest and highest quality for processing pixel art.

*EPX* ("Eric's Pixel Scaler") [Johnston 1992] was the first 2× pixel-art magnification filter with a clear goal of preserving style. Eric Johnston created it when porting the LucasArts SCUMM game engine from PC to the then-faster, higher-resolution Macintosh [Thomas 1999]. The algorithm was independently reinvented and popularized by Mazzoleni [2001] for the MAME retro-game emulator. Our MMPX algorithm builds directly on EPX by extending its rule set to recognize more patterns, and our algorithm name follows Johnston's convention using our first initials.

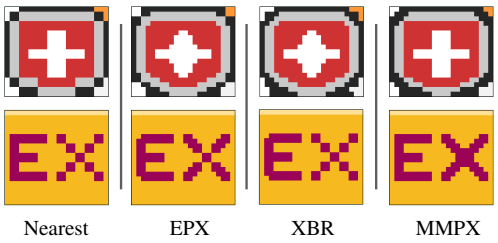



**Figure 6.** EPX and XBR indiscriminately round all corners and can fail at intersections.

EPX is very simple, extremely fast, and is in current use in both retro emulators and new pixel-art games. It is well-suited to the LucasArts content of its era, which had bubbly character and font designs, and in our judgement it remains superior in both performance and quality to other filters we evaluated for that kind of content. For content that contains corners and straight lines, including many pixel-art fonts, the primary drawback of EPX is that it rounds everything too much as shown in Figure 6.

Maxim Stepin developed *HQX* ("high quality scaling," sometimes stylized as hq2x or HQx) [2003] for use in PC emulators of retro consoles as a 2× scaling filter. He noted that it can be run twice to produce 4× scaling and later released a separate algorithm 3×. HQX is essentially a bilateral filter with a $3 \times 3$ kernel that is implemented using lookup tables (it is roughly analogous to the Marching Cubes [Lorensen and Cline 1987] isosurface tessellation algorithm in structure).

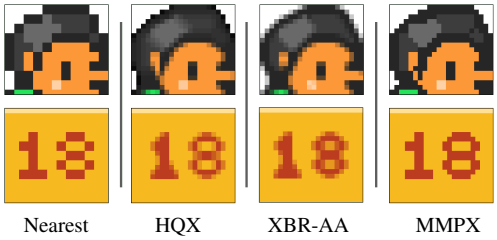



**Figure 7**. HQX and XBR-AA do not preserve the palette or sharpness.

HQX produces beautifully antialiased results with smooth gradients. It is superior to EPX when the introduction of new colors is desired but frequently inferior when more strict style preservation is the goal. That is because HQX introduces new colors (Figure 7) and does not support transparency. Because HQX relies on lookup tables for its many cases, the performance varies highly with the bandwidth and cache coherence available in different implementations and platforms. We do not consider HQX sufficiently style-preserving to meet our goals, but because of its popularity we include performance results in this paper and provide implementations and result comparisons in our supplement.

Hylian developed the *XBR* ("scale by rules," sometimes stylized as xBR) [2011] family of filters as a successor to HQX. It is also a form of bilateral filter, but uses explicit color-space bounding boxes instead of enumerated cases in a lookup table. The base version of the algorithm is more style-preserving than HQX because it handles transparency and does not introduce new colors. It tends to handle more edge slopes than EPX but still rounds corners and does not deal well with 45-degree slopes, leaving them too jagged (Figure 6). More sophisticated versions correct the edge limitations but introduce new colors while antialiasing (XBR-AA) and round corners as shown in Figure 7. We include those variants in the supplement. XBR is orders of magnitude more expensive than EPX, yet it does not provide consistently better quality when the palette must be preserved.

Stasik and Balcerek [2017] derived a relatively complicated magnification filter for pixel art within the general scientific image-processing literature. Their filter is designed to support arbitrary scaling ratios instead of being hard-coded for $2\times$ or $3\times$. In practice, the results are similar to Lanczos or bicubic results. They preserve more frequency content than nearest or bilinear filtering, but almost none of the style of the source image in the way that the EPX, XBR, MMPX, or even HQX do.

Xenowhirl developed *RotSprite* [2007] as an interactive tool for generating rotated sprites while working on the *Sonic* video game series. It magnifies sprites in a spritesheet before rotation and then applies various heuristics for minifiying them afterward. In production, Xenowhirl manually retouched RotSprite output to build spritesheets. We observe that when using a filter that is better than nearest for magnification (such as EPX, XBR, HQX, or MMPX), RotSprite produce high-quality results when using a fast, nearest-neighbor minification filter. This enables fully automatic run-time use because the magnification can be precomputed. In combination with RotSprite, our results demonstrate that MMPX can produce slightly better thin

lines and outlines than EPX and XBR, but all three are much better than naive rotation without magnification.

*Style transformation*

Some related work explores pixel-art filters with the goal of producing different source or destination styles.

Han et al. [2018] introduced a minification and stylizing filter for transforming natural images into pixel art. The PixaTool https://kronbits.itch.io/pixatool program uses an undisclosed algorithm for the same transformation.

Coeurjolly et al. [2018] presented a method for converting 3D voxel art to vector shapes. The shapes are smooth but the textures remain only nearest-neighbor magnified, which may be aesthetically desirable in some applications.

Kopfs et al.'s [2011] depixelization algorithm converts pixel art into vector art by maximally smoothing the contours. It runs in minutes per frame and does a remarkable job of magnifying sprites, but alters the style significantly, where thin lines in the source image have varying output thickness and all shapes become rounded, reminiscent of clip art in a sketch style. A later implementation presented as a poster [Kreuzer et al. 2015] produces similar high quality at $4\times$ magnification. It runs in about 140 ns/pixel on a GPU, which is sufficient for real-time on a desktop but two orders of magnitude slower than the pixel-art filters such as EPX, XBR, and MMPX.

Several texture-synthesis or style-transfer methods can convert between natural images and abstracted styles [Gatys et al. 2015; Park et al. 2019; Rebouas Serpa and Formico Rodrigues 2019].

## 3. Expressing Filters as Rules

The EPX, XBR, and MMPX filters compute four destination pixels for a single source pixel `E` using rules that recognize local features such as corners and edges. When a small neighborhood of `E` matches a rule's pattern, it assigns one or two of the four output destination pixels. Figure 8 shows the pixel indexing for this paper, which follows Mazzoleni's notation. The $2\times$ magnification filters produce a destination image that is twice as large in each dimension as the source, so each source pixel `E` maps to four destination pixels `J`, `K`, `L`, and `M`. The pixels in the $3 \times 3$ source neighborhood are named `A-I` in raster order, and the diamond points are named `P-S`. We use row-major, top-down linear pixel packing where source pixel `E = src(x, y)` is stored at `srcBuffer[x + y * srcWidth]`. To simplify the presentation in this paper, we clamp out-of-bounds reads to the edges of the source image.

For efficiency and given the target domain of pixel art, all versions operate on packed 32-bit unsigned integer pixel values that we call `ABGR8` (matching both OpenGL and JavaScript's default encoding across the platforms from our experiments) that are 8-bit normalized sRGB values with an unassociated alpha channel.

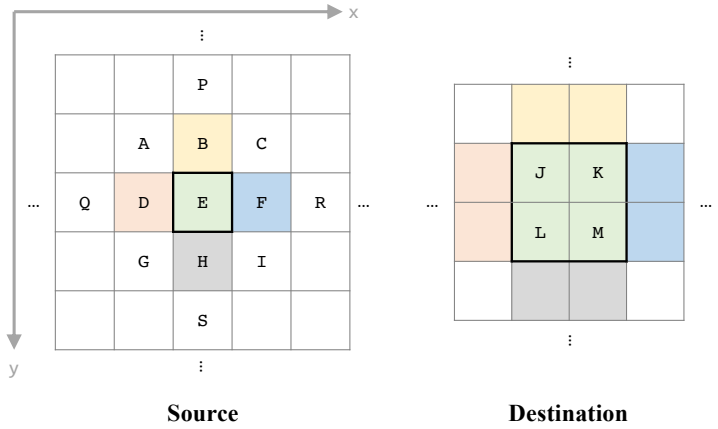



**Figure 8.** A 2× magnification filter computes destination pixels J, K, L, and M from source pixel E and its neighborhood. Colors in this figure show corresponding pixels under Nearest.

A single-pass filter comprises loops over both dimensions and a main, inner-loop body. On a GPU, these loops are implicit in the shader launch. A GPU compute shader or CPU implementation iterates over the source dimensions and writes four destination pixels per body execution, amortizing the cost of reading the neighborhood and of executing the rules that apply to multiple destination pixels. Because it iterates along rows, we structure a CPU implementation to only read the right-most edge of the neighborhood per iteration, and pass along the remainder of the neighborhood from the previous one (see Listing 5 at the end of the paper). This keeps most of the working set in registers or at the top of the stack in L1 cache for a measurable performance advantage. See our supplemental files for the JavaScript equivalent.

A similar optimization could be performed for very large images on a GPU compute-shader implementation. However, given that the target spritesheet and screen resolutions are comparable to the number of lanes on even an embedded GPU, it is more efficient to process each set of four destination pixels in its own GPU lane and let the GPU's higher-cache bandwidth capture the data reuse pattern.

Listing 6 (at the end of the paper) shows our GLSL compute shader framework. We use OpenGL ES shaders because they execute under both full OpenGL and the reduced OpenGL ES standard embedded systems GPUs. We use GLSL version 3.10 because that is the highest level supported by the popular Broadcom Videocore VI GPU on the Raspberry Pi 4-series system-on-a-chip.

A GPU pixel-shader implementation iterates over the destination dimensions and writes a single pixel per body execution. See the code supplement for our GLSL pixel shader framework supporting previous XBR and HQX implementations, which is not used by our own MMPX algorithm, Nearest, or EPX.

When processing a source image with transparency, we recommend extending boundary clamping to treat out of bounds pixels as fully transparent. When process-

```
ABGR8 J = E, K = E, L = E, M = E;
```

**Listing 1**. Nearest body rule (GLSL & C++).

ing a font sheet or spritesheet instead of a full image, process each glyph or sprite independently for bounds, clamping so that pixels of adjacent elements are not misinterpreted as features that cross the boundary. Because MMPX resolves ambiguous patterns by assuming a dark foreground, for processing font sheets with black backgrounds we recommend converting the background to transparent or inverting the image for filtering, and then restoring it in the destination. The version of MMPX deployed in quadplay✤ follows all of these practices.

Filters that preserve the local palette, such as MMPX, are implemented with a series of rules for deciding which pixel value from the source neighborhood each of the destination pixels should be assigned. In this context, the Nearest filter is described by the single, unconditional rule in Listing 1: make all four destination pixels equal to the central source value, `E`.

The EPX [Johnston 1992; Mazzoleni 2001] magnification filter in Listing 2 has five rules. It begins with the Nearest rule, and then tests for overriding conditions. Each of the four remaining rules considers one corner of the source neighborhood. If the two spatially close source neighbors match each other and are different from the other two, then the destination pixel corresponding to that corner is set to the value of the matching source pixels. The first such rule examines the top-left corner: if the pixel above and to the left of `E` have the same value and differ from the pixels to the right and below, the corresponding to destination pixel `J` receives that value.

```
// Default to Nearest magnification
ABGR8 J = E, K = E, L = E, M = E;

// Round some corners
if (D == B && D != H && D != F) J = D; // First corner rule
if (B == F && B != D && B != H) K = B;
if (H == D && H != F && H != B) L = H;
if (F == H && F != B && F != D) M = F;
```

**Listing 2**. EPX body rules (GLSL & C++).

Figure 9 visualizes the first EPX corner rule. The remaining three rules are symmetric to it. On the left (source) side of the rule diagram, we use colors to denote pixel values that must be the same as each other for the rule to apply (here, only red-coded), shades of gray to denote pixels that must be different from all of the colors but not necessarily the same as other gray-coded pixels. White pixels are unconstrained. Colors in rule diagrams solely denote patterns, not actual pixel colors.

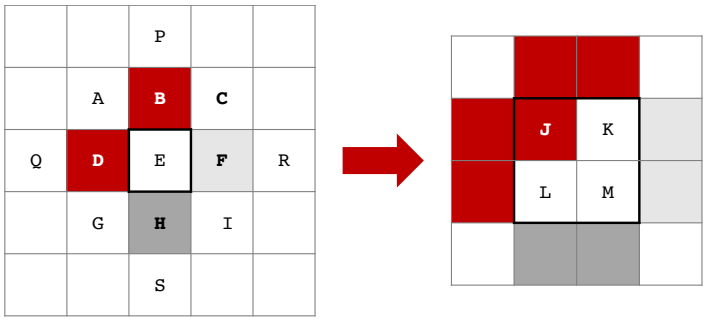



**Figure 9**. The first EPX corner rule. In our rule diagrams, the colors label pattern-matching variables and not actual pixel values. Each saturated color, such as red, denote pixels that must exactly match others marked with that color in the diagram. Gray matches any pixel that is different from all saturated labels. White is unconstrained for the rule.

The right side of the rule diagram shows which destination pixels were set. Destination values outside of the central $2\times2$ box are not set by the filter on this iteration and may not have the values depicted. In this example, we can see that EPX extends Nearest by rounding corners and smoothing the stair stepping on diagonal lines.

## 4. MMPX Algorithm

### 4.1. Utility Functions

We define some utility functions in Listing 3 to simplify the MMPX implementation. The `luma()` function computes the sum of the three color channels plus 1 and weighs it by `256 - alpha`. We use this to identify dark or opaque values, which MMPX assumes are foreground/positive-space pixels in otherwise ambiguous conditions. Because the luminance is used only to coarsely make this distinction, we avoid the expense perceptually weighting of color channels.

The `all_eqN()` functions return true if and only if all of their $N$ arguments are equal. The `any_eqN()` functions return true if and only if the first argument is equal to any of the remaining $N$. The `none_eqN()` functions return true if and only if the first argument is different from all of the remaining $N$.

Through profiling, we found that the `all_eqN()` and `none_eq8()` functions are faster when implemented with bitwise operations, while the others are faster when implemented in a straightforward manner with logical operations.

### 4.2. Fallback to Nearest (Lines 4–6)

Listing 4 gives the body of the MMPX algorithm in GLSL and C++, which have similar syntax. It uses a preprocessor branch to distinguish between the languages for the only implementation difference, which is that in C++ we pass two source pixels labelled `Q` and `R` from the previous loop iteration instead of reading them each

```
1  // Fast luminance approximation with transparency, assuming a bright
2  // background
3  uint luma(ABGR8 C) {
4      uint alpha = (C & 0xFF000000u) >> 24;
5      return (((C & 0x00FF0000u) >> 16) + ((C & 0x0000FF00u) >> 8) +
6              (C & 0x000000FFu) + 1u) * (256u - alpha);
7  }
8
9  // True if all values are equal
10 bool all_eq2(ABGR8 B, ABGR8 A0, ABGR8 A1) {
11     return ((B ^ A0) | (B ^ A1)) == 0u;
12 }
13
14 bool all_eq3(ABGR8 B, ABGR8 A0, ABGR8 A1, ABGR8 A2) {
15     return ((B ^ A0) | (B ^ A1) | (B ^ A2)) == 0u;
16 }
17
18 bool all_eq4(ABGR8 B, ABGR8 A0, ABGR8 A1, ABGR8 A2, ABGR8 A3) {
19     return ((B ^ A0) | (B ^ A1) | (B ^ A2) | (B ^ A3)) == 0u;
20 }
21
22 // True if any B == any An
23 bool any_eq3(ABGR8 B, ABGR8 A0, ABGR8 A1, ABGR8 A2) {
24     return B == A0 || B == A1 || B == A2;
25 }
26
27 // True if no B == any An
28 bool none_eq2(ABGR8 B, ABGR8 A0, ABGR8 A1) {
29     return (B != A0) && (B != A1);
30 }
31 bool none_eq4(ABGR8 B, ABGR8 A0, ABGR8 A1, ABGR8 A2, ABGR8 A3) {
32     return B != A0 && B != A1 && B != A2 && B != A3;
33 }
34
35 bool none_eq8(ABGR8 B, ABGR8 A0, ABGR8 A1, ABGR8 A2, ABGR8 A3,
36               ABGR8 A4, ABGR8 A5, ABGR8 A6, ABGR8 A7) {
37     return ((A0^B) | (A1^B) | (A2^B) | (A3^B) |
38             (A4^B) | (A5^B) | (A6^B) | (A7^B)) != 0u;
39 }
```

**Listing 3**. Utility functions (GLSL & C++)

time. See our supplement for the JavaScript implementation, which differs only in the equality operator syntax.

Line 4 is the same as Nearest filtering. Line 6 then tests whether any other pixel in the central 3×3 grid is the same as the central source pixel E. If none are equal, then there are no identifiable features in the four destination pixels corresponding to E, so Nearest is the best that can be done and the algorithm terminates for that iteration.

```
// Input:  A-I central 3x3 grid

// Output pixels default to the input value
ABGR8 J=E, K=E, L=E, M=E;

if (none_eq8(E,A,B,C,D,F,G,H)) {
 // Read additional values at the tips of the diamond pattern (GLSL).
 ABGR8 P = src(srcX, srcY-2), S = src(srcX, srcY+2);

 // In C++, Q and R are passed from the previous pixel, outside of this branch.
#if defined(GL_core_profile) || defined(GL_ES)
 ABGR8 Q = src(srcX-2, srcY), R = src(srcX+2, srcY);
#endif

 // Precompute luminances
 ABGR8 Bl = luma(B), Dl = luma(D), El = luma(E), Fl = luma(F), Hl = luma(H);

 // 1:1 slope rules, extended from EPX
 if ((D==B && D!=H && D!=F) && (El>=Dl || E==A) && any_eq3(E,A,C,G) && (El<Dl || A!=D
      || E!=P || E!=Q)) J=D;
 if ((B==F && B!=D && B!=H) && (El>=Bl || E==C) && any_eq3(E,A,C,I) && (El<Bl || C!=B
      || E!=P || E!=R)) K=B;
 if ((H==D && H!=F && H!=B) && (El>=Hl || E==G) && any_eq3(E,A,G,I) && (El<Hl || G!=H
      || E!=S || E!=Q)) L=H;
 if ((F==H && F!=B && F!=D) && (El>=Fl || E==I) && any_eq3(E,C,G,I) && (El<Fl || I!=H
      || E!=R || E!=S)) M=F;

 // Intersection rules
 if ((E!=F && all_eq4(E,C,I,D,Q) && all_eq2(F,B,H)) && (F!=src(srcX+3, srcY))) K=M=F;
 if ((E!=D && all_eq4(E,A,G,F,R) && all_eq2(D,B,H)) && (D!=src(srcX-3, srcY))) J=L=D;
 if ((E!=H && all_eq4(E,G,I,B,P) && all_eq2(H,D,F)) && (H!=src(srcX, srcY+3))) L=M=H;
 if ((E!=B && all_eq4(E,A,C,H,S) && all_eq2(B,D,F)) && (B!=src(srcX, srcY-3))) J=K=B;

 // Triangle tip rules
 if (Bl<El && all_eq4(E,G,H,I,S) && none_eq4(E,A,D,C,F)) J=K=B;
 if (Hl<El && all_eq4(E,A,B,C,P) && none_eq4(E,D,G,I,F)) L=M=H;
 if (Fl<El && all_eq4(E,A,D,G,Q) && none_eq4(E,B,C,I,H)) K=M=F;
 if (Dl<El && all_eq4(E,C,F,I,R) && none_eq4(E,B,A,G,H)) J=L=D;

 // 2:1 edge rules
 if (H!=B) {
   if (H!=A && H!=E && H!=C) {
     if (all_eq3(H,G,F,R) && none_eq2(H,D,src(srcX+2, srcY-1))) L=M;
     if (all_eq3(H,I,D,Q) && none_eq2(H,F,src(srcX-2, srcY-1))) M=L;
   }

   if (B!=I && B!=G && B!=E) {
     if (all_eq3(B,A,F,R) && none_eq2(B,D,src(srcX+2, srcY+1))) J=K;
     if (all_eq3(B,C,D,Q) && none_eq2(B,F,src(srcX-2, srcY+1))) K=J;
   }
 }

 if (F!=D) {
   if (D!=I && D!=E && D!=C) {
     if (all_eq3(D,A,H,S) && none_eq2(D,B,src(srcX+1, srcY+2))) J=L;
     if (all_eq3(D,G,B,P) && none_eq2(D,H,src(srcX+1, srcY-2))) L=J;
   }

   if (F!=E && F!=A && F!=G) {
     if (all_eq3(F,C,H,S) && none_eq2(F,B,src(srcX-1, srcY+2))) K=M;
     if (all_eq3(F,I,B,P) && none_eq2(F,H,src(srcX-1, srcY-2))) M=K;
   }
 }
}
```

**Listing 4**. MMPX implementation body (GLSL & C++).

### 4.3. 1:1 Edges (Lines 19–22)

We call a horizontal, vertical, or diagonally-connected run of pixels of the exact same value an *edge*. We call an edge at $\pm 45^\circ$ to horizontal a 1:1 [slope] edge.

Lines 19-22 of Listing 4 recognize 1:1 edges with four rules (`if` statements) corresponding to the four ways an edge can appear adjacent to central pixel `E`. As with all of the following MMPX rules, we only describe the first because the four patterns are rotationally symmetric with each other. The first is:

```
if ((D==B && D!=H && D!=F) && // Adjacent 1:1 edge
    (El>=Dl || E==A) &&       // Luminance tie-break or thick
       feature
    any_eq3(E,A,C,G) &&       // Not a sharp corner
    (El<Dl || A!=D || E!=P || E!=Q)) // Not a single-pixel bump
    J=D; // Fill the destination corner along the 1:1 edge
```

The test expression contains four parenthesized clauses. The first clause is identical to the corresponding EPX rule from Figure 9. It matches when two diagonal pixels are identical and are different from both of the opposite diagonal pixels.

The other three clauses are unique to MMPX. These avoid the double refinement of thick lines and the over-rounding of corners observed under EPX. For a thick line or large shape with a 1:1 edge, the first clause from EPX will trigger on both sides. Matching on one side refines a blocky Nearest edge. Matching on both sides restores the blocky Nearest output, shifted by half a pixel relative to the source. This can be observed on the $45^\circ$ rotated central brown box in Figure 14(b). MMPX avoids this problem by only refining one side. Lacking any other way to break the tie using only a small kernel, it assumes that the lower luminance/more opaque side is the foreground feature and refines it at the expense of the higher luminance, background side's pixels. In the case of a perfect luminance tie, it fails but still is no worse than EPX.

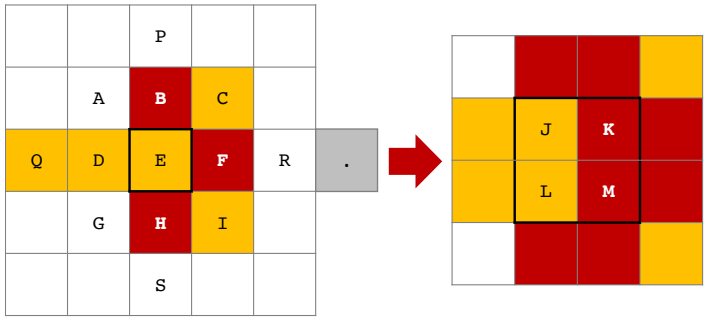
Note that the first clause does not consider the value of the central source pixel `E` itself, which is why EPX undesirably rounds all corners. In MMPX, the third clause preserves sharp corners by only refining edges where `E` also matches a of diagonals that is not part of the edge.

The 4th clause prevents rounding single-pixel bumps that appear on horizontal or vertical lines of a darker color than the background, such as the tail on a pixel art "4."

### 4.4. Intersections (Lines 25–38)

As depicted in Figure 10, code lines 25–28 connect intersections that the previous 1:1 edge rules leave disconnected. There are two parenthesized clauses. The first recognizes the inside of a diagonal corner formed by two lines. The second clause ensures that the pattern is not a notch at the edge of a checkerboard dithering pattern.

Note that in ambiguous cases, the intersection rule does not attempt to break ties using luminance. That is because doing so at an intersection could disconnect the wrong feature.




```
if ((E!=F && all_eq4(E,C,I,D,Q)&& all_eq2(F,B,H))&& (F!=src(srcX+3, srcY)))K=
M=F;
```

**Figure 10**. The MMPX intersection rule. The gray pixel is necessary to break symmetry and distinguish intersections from the edge of checkerboard dithering patterns.

Consider the 4×4 pixel region shown on the left of Figure 11, with the goal of computing destination pixel values within the red outline. An artist might interpret the source as depicting a black letter "R" on a white background and extend the stem as shown on the top-right subimage. But that interpretation cannot be made objectively given the local context. The same pixels might depict a white "Я" on a black background as shown on the bottom right subfigure; an equally plausible interpretation in Russian. We designed MMPX intersection rules to discriminate correctly when there is clear background on all sides and to intentionally not match when other features are nearby (detected by the gray pixel in Figure 10) to avoid disconnecting potential features.

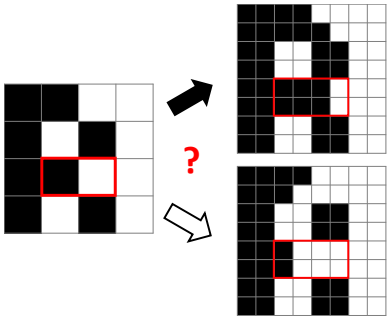


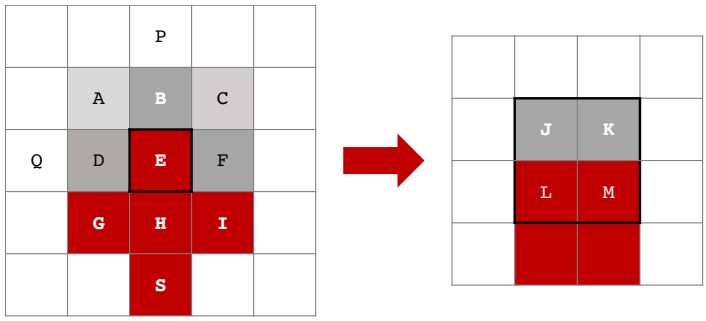

**Figure 11**. Is this a black "R" on white, or a white "Я" on black? Each choice disconnects one feature within the red outline.

### 4.5. Triangle Tips (Lines 31–34)

A triangle's tip is the positive space, convex corner interpretation of a surrounding negative space intersection or concave corner. The luminance condition for 1:1 edges will flatten the tips of bright triangles against dark backgrounds because it refines the two incoming edges at the intersection. The rules on lines 30–33 depicted in Figure 12 correct for this to restore the corner.

### 4.6. 2:1 Edges (Lines 36–58)

As depicted in Figure 13, lines 36–58 of Listing 4 handle 2:1 edges. When it applies, each 2:1 edge rule copies a pixel value already assigned by a previous rule, rather



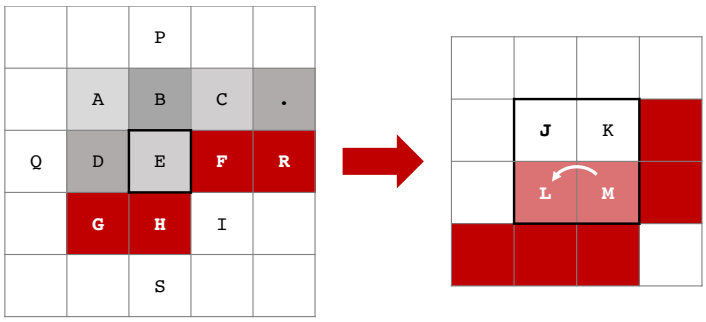

```
if (Bl<El && all_eq4(E,G,H,I,S)&& none_eq4(E,A,D,C,F))J=K=B;
```

**Figure 12**. A MMPX triangle tip rule.

than directly reading using a source pixel value. They effectively extend the reach of the 1:1 edge rules by additional pixel. Implicit luminance tie breaks are thus inherited from previous rules. There are eight ways to copy a pixel horizontally or vertically, so there are eight 2:1 edge rules.

The first clause of each rule identifies a 2:1 edge of constant color. The second clause verifies that the edge is separated from every adjacent pixel on one side and thus not part of a more complex pattern. Common subexpressions from the first clause are lifted to the two outer tests.

MMPX cannot perfectly refine edges with 3:1 or higher slopes. It will treat those as a series of stretched 2:1 steps, creating a slightly wiggly result that is better than previous filter methods but worse than what an artist would produce.




```
if (H !== B) { if (H !== A && H !== E && H !== C)
{ // Nothing else matches the edge
     if (all_eq3(H, G,F,R) && none_eq2(H, D,src(srcX+2, srcY-1))) // 2:1 edge
        L = M; // Extend the previous 1:1 rule's reach
     ...
```

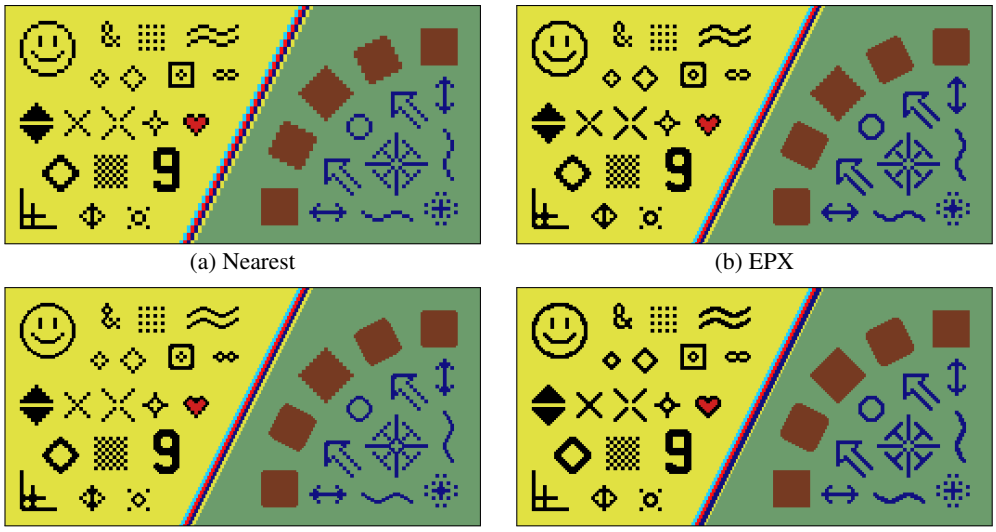
**Figure 13**. A MMPX 2:1 edge rule, which is divided over four statements to lift common subexpressions from the other symmetric 2:1 rules not shown here.

## 5. Qualitative Evaluation

The Nearest, EPX, XBR, and MMPX filters preserve palette by construction, so we evaluate other elements of style preservation in detail here. Our supplement contains additional results from non-palette preserving HQX, XBR-Antialiased, Bilinear, and Unbiased Bilinear filters. The input image is the same as Nearest at half resolution, so we do not show it in the result figures. All results can be reproduced by running `js-demo.html` from our supplement in a web browser and then dragging the test images onto the page (also at https://morgan3d.github.io/quadplay/tools/scalepix.html).

Figure 14 shows results on a pixel art magnification filter test image by Stepin [2003]. It covers a variety of difficult cases. On the right half of the image, a series of brown squares at different orientations on a green background test all combinations of 1:1 and 2:1 edges on large features. MMPX refines these all ideally. EPX and XBR fail on the 1:1 edges because they process both the brown and green sides; double-refinement yields a shifted Nearest result. They succeed on the 2:1 edges because they add one pixel on each side. None of the filters processed 30°- and 60°-rotated box corners well. EPX and MMPX over-round slightly, and XBR skews and incorrectly rounds.

MMPX retains the sharp corners of the brown axis-aligned boxes and the convex and concave corners of the black-on-yellow features on the left side that EPX and XBR incorrectly round.



(a) Nearest (b) EPX

(c) XBR (d) MMPX

**Figure 14.** Comparison of four filters on the pixel-art magnification test image by Stepin [2003]. Examine the 2:1 and 1:1 edges on the brown boxes, the corners and curves, and intersections in the blue line patterns. All methods round the 30°-rotated box corners and MMPX produces inconsistent thickness for the 2:1 stripes in the center because of its luminance tie-break. MMPX performs better than the others for most other patterns and does particularly well on the red heart outline, ampersand, and diamonds.

All three filters refine the 2:1 edges of the 2-pixel thick colored stripes in the center well. MMPX produces stripes of different thickness because its luminance tie break prefers dark blue over bright yellow.

On the black-on-yellow happy face and the blue curves, MMPX and XBR produce highly refined curves. EPX falls back to Nearest for these. On the black ampersand and red heart, only MMPX correctly refines both the curves and lines.

EPX is better than XBR at refining the straight blue lines and shapes, but it disconnects their intersections. MMPX handles both the line refinement and intersections well. Its result is not as good for the blue double-arrow heads, however.

All three filters preserve the checkerboard pattern by falling back to Nearest. Only more context and an artist's intention could distinguish whether it is a dither pattern that refined to single pixels or a checkerboard to magnify to $2 \times 2$ pixel squares.

We created the binary image in Figure 15 to test all of the style-preservation criteria listed in Section 1.2. For each feature, it contains both black-on-white and inverse versions so that the luminance tests in MMPX have no advantage.

The properties observed for each of the filters are consistent with the previous example. MMPX handles lines of varying thickness and slope well, refines curves, keeps both convex and concave corners sharp, and maintains intersections properly. XBR is equally good on curves and 2:1 edges, but rounds all corners and cannot handle 1:1 edges. EPX is poor on 1:1 edges, 2:1 thin lines, intersections, and curves, and rounds everything.

The brick and grid patterns on the right of the test image show MMPX's corner preservation very clearly over the other methods.

Within the thin line examples, note that MMPX preserves the single-pixel bumps on horizontal and vertical lines that EPX and XBR both thicken and round.

Because of the luminance rules, for certain size disks MMPX produces better results for black on white instead of white on black. EPX and XBR are consistently good for both because for disks, rounding everything is the ideal strategy. Note that if the white disks were filtered individually as sprites on a transparent background, MMPX would produce equally good results.

MMPX shows a strong advantage for text. That combines the cases for which it performs better than the other methods with a scenario where small changes can radically affect readability and perception of shape. For both the labels such as "DIAMONDS" and the explicit text and symbol area on the lower right of the test image, MMPX better captures the style of various fonts and preserves readability. EPX and XBR's rounding and poor 1:1 edge handling here can make a "D" look like an "O" for EPX, or destroy the art deco style of "ABCDE" and curves of "S" under XBR.

Figure 16 shows results on a spritesheet containing sprites in various pixel-art styles by ourselves and others. For this evaluation, we magnified and filtered the spritesheet with transparency and then composited it onto the gray background for

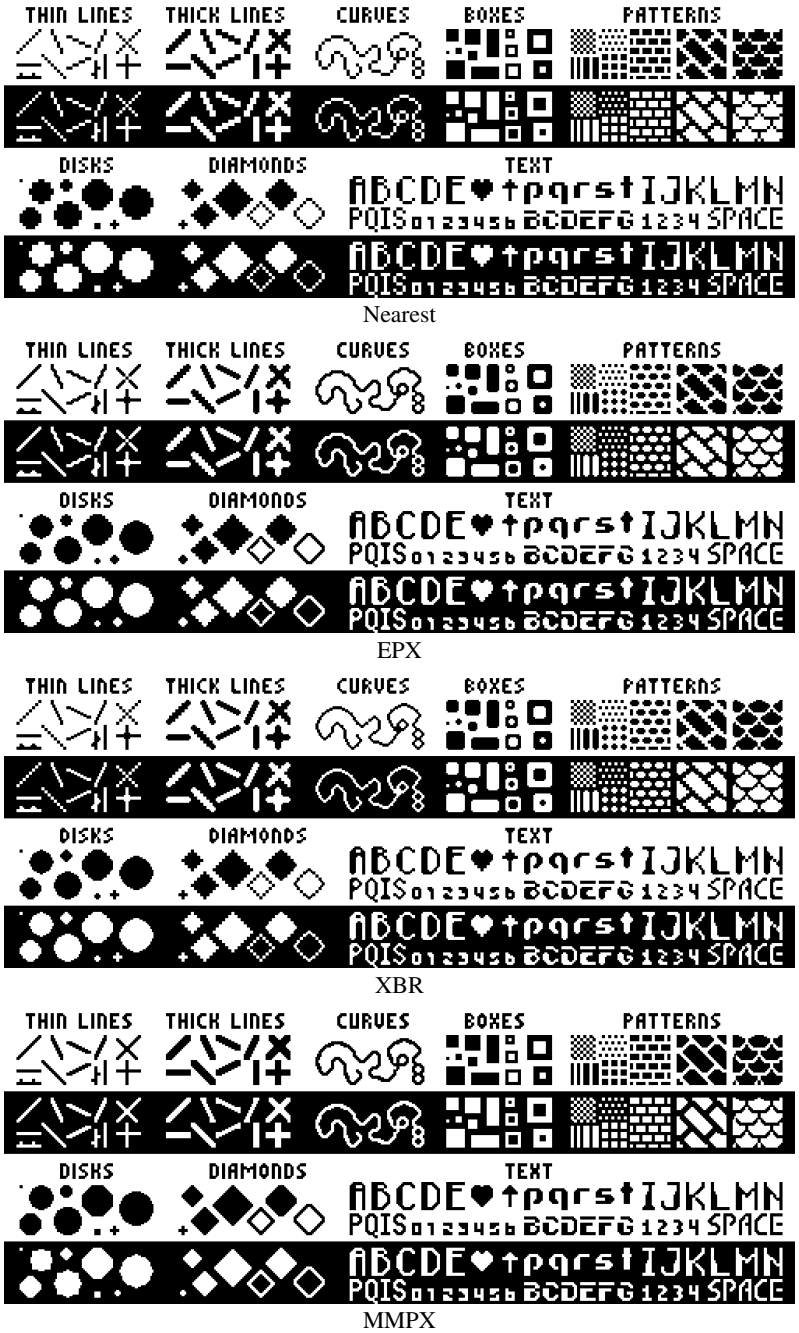



**Figure 15**. Comparison of four filters on an image we designed to test style preservation of edges, curves, lines, patterns, and intersections. In the case of light disks on a dark background, the MMPX luminance rules produce under-smoothed results, while the over-smoothing of EPX and XBR happens to match the desired outcome better in this case. In all other situations, MMPX generally preserves style better than the other filters, notably at sharp corners, intersections, 2:1 edges, and the thin features found in text.

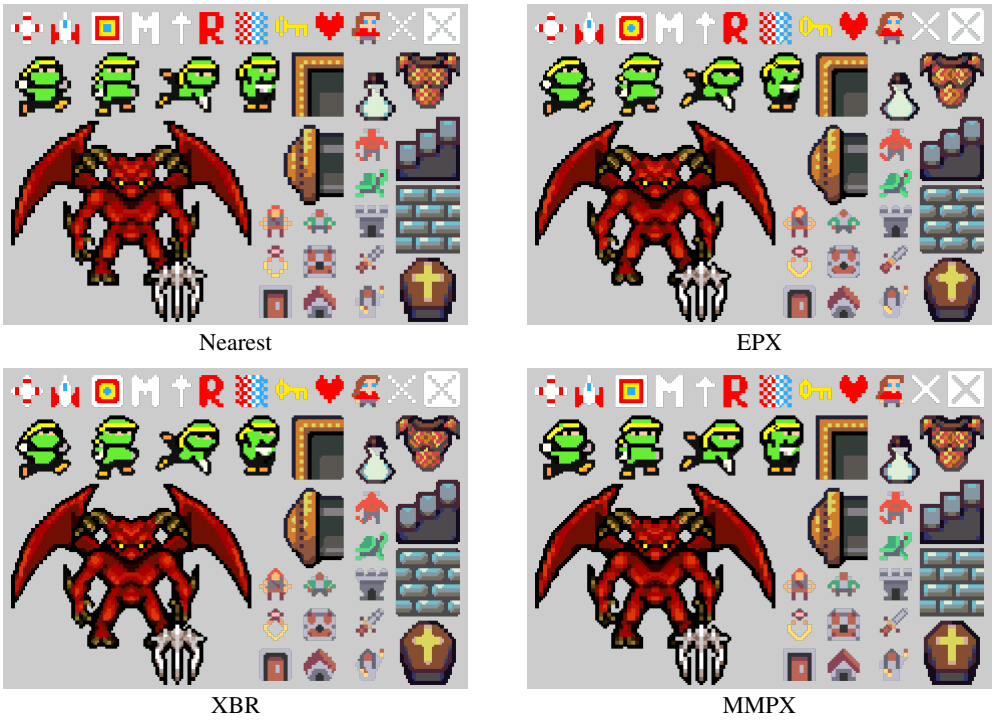
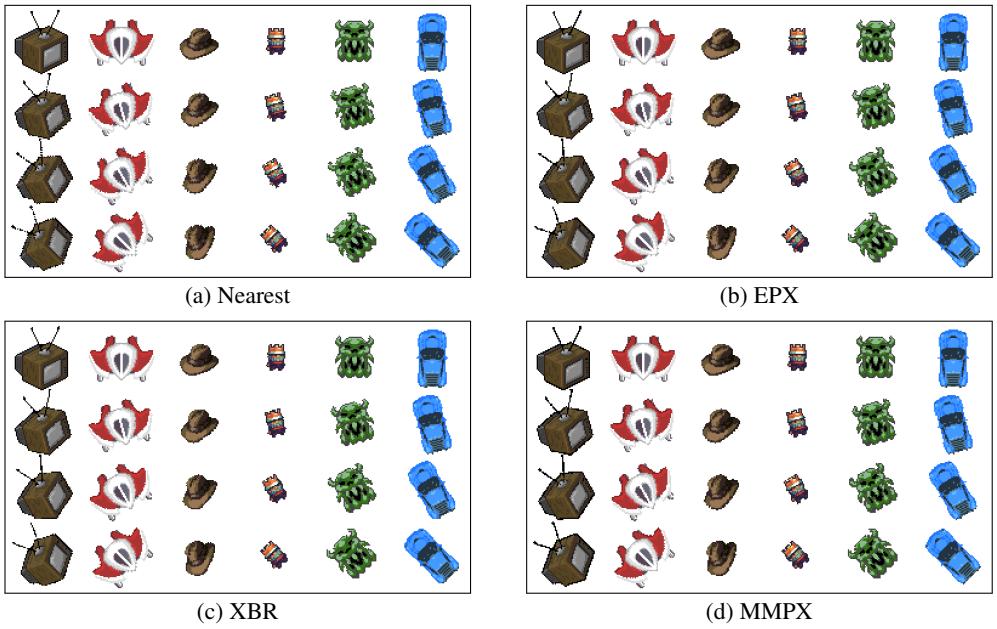



**Figure 16.** Spritesheet with transparency magnified with various filters. MMPX best smoothes the silhouettes of the characters, preserves the sharp corners on the blue bricks, maintains the intersection in the X's on the top right, and preserves the single-pixel eyes of the green ninja.

*Top row of* $8 \times 8$ *PICO-8 sprites by Morgan McGuire 2020 in the Public Domain; Green Ninja by DezrasDragons 2015 in the Public Domain https://opengameart.org/content/ninja-animated; Large Red Imp ©2012, Redshrike & William Thompson CC-BY 3.0 https://opengameart.org/content/lpc-imp; Gold and gray UI elements ©2013 Buch CC-BY-SA 3.0 https://opengameart.org/content/golden-ui-bigger-than-ever-edition; 8x8 roguelike sprites by Morgan McGuire & Kenney.nl 2020 in the Public Domain; 16x16 dungeon sprites ©2014 Dragon De Platino & DawnBringer CC-BY 4.0 https://opengameart.org/content/dawnlike-16x16-universal-rogue-like-tileset-v181*

display. EPX and XBR produce better rounding than MMPX on the turtle shell, and each method has a different undesirable outcome for the white arrow. For all other cases, MMPX performs as well or better than EPX and XBR. Note in particular MMPX's correct handling of the "X" in the upper-right corner, sharp bricks and coffin cross, the eyes of the green ninja, and the refinement of the demon's curved wings.

Figure 17 shows our variation of the RotSprite algorithm using each of the filters for magnification and Nearest for minification. EPX, XBR, and MMPX produce similar results and all are significantly better than Nearest. We prefer MMPX slightly for keeping the TV antennae connected, fewer stray pixels on the silhouette of the spaceship, and consistently maintaining the gray outline on the green monster skull.



(a) Nearest (b) EPX

(c) XBR (d) MMPX

**Figure 17**. Rotation of sprites with transparency using RotSprite with different magnification filters. All other filters are much better than Nearest. MMPX preserves the thin TV antennae and spaceship silhouette slightly better than the other filters in these examples.

*TV ©2004 LucasVB CC BY-SA 3.0 https://commons.wikimedia.org/wiki/File:Pixelart-tv-iso.png; Spaceship by Kenney.nl 2016, in the Public Domain https://opengameart.org/content/space-shooter-extension-250; King by Buch 2014, in the Public Domain https://opengameart.org/content/a-platformer-in-the-forest; Skull ©2015 Lunarsignals, CC-BY-SA 3.0 https://opengameart.org/content/overhead-action-rpg-forest; Hat from Dungeons of Dredmor, ©2011 Gaslamp Games. Used with permission from Nicholas Vining for this research paper. Redistribution and further use is prohibited; Sports car by Morgan McGuire 2020, in the Public Domain.*

Figures 18 and 19 show that the filter behaviors we examined through isolated test cases are consistently maintained in complex whole-screen images. These show a screenshot from a pixel-art game and a standalone artwork in which MMPX produces good results for the varying edge slopes, sharp corners, and intersections.

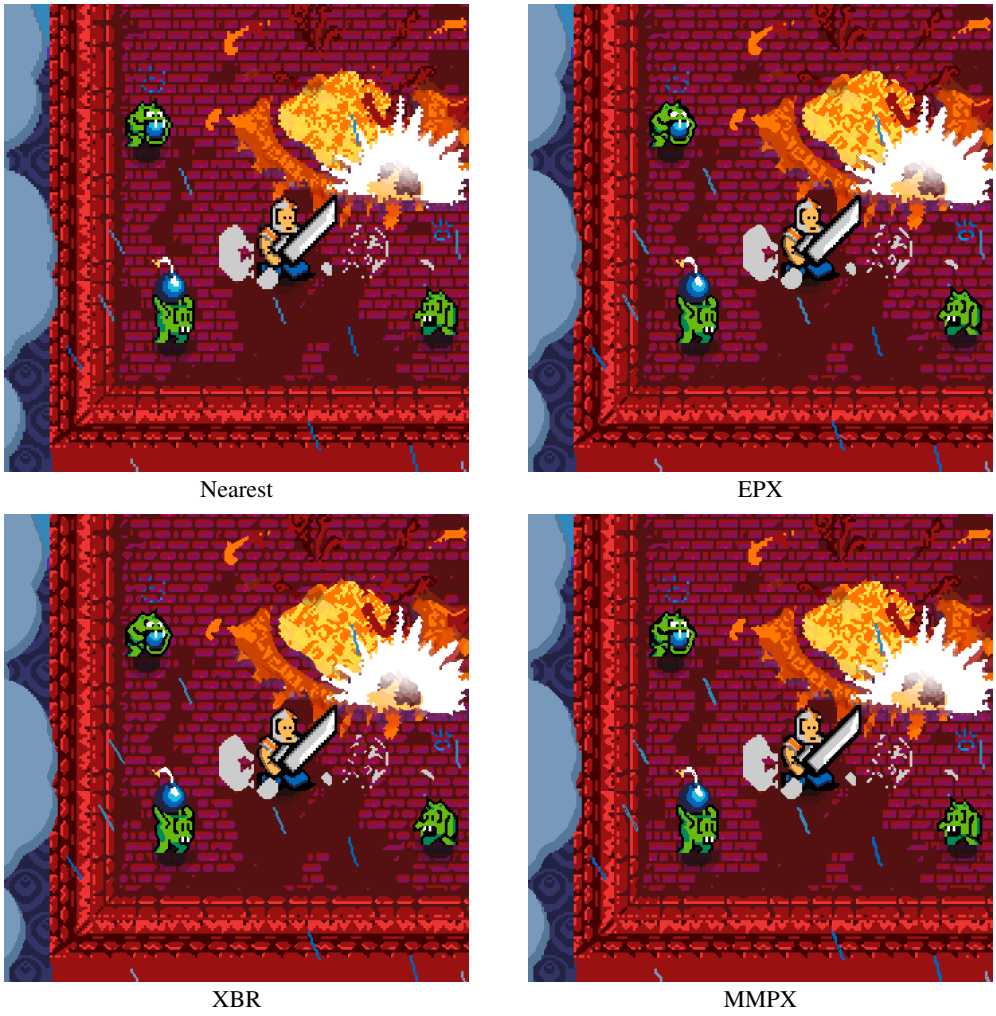



**Figure 18.** Crop of a full-screen game image magnified with different filters. *Beat the Gobblins* ©*2019 Stephan Steinbach, used with permission*

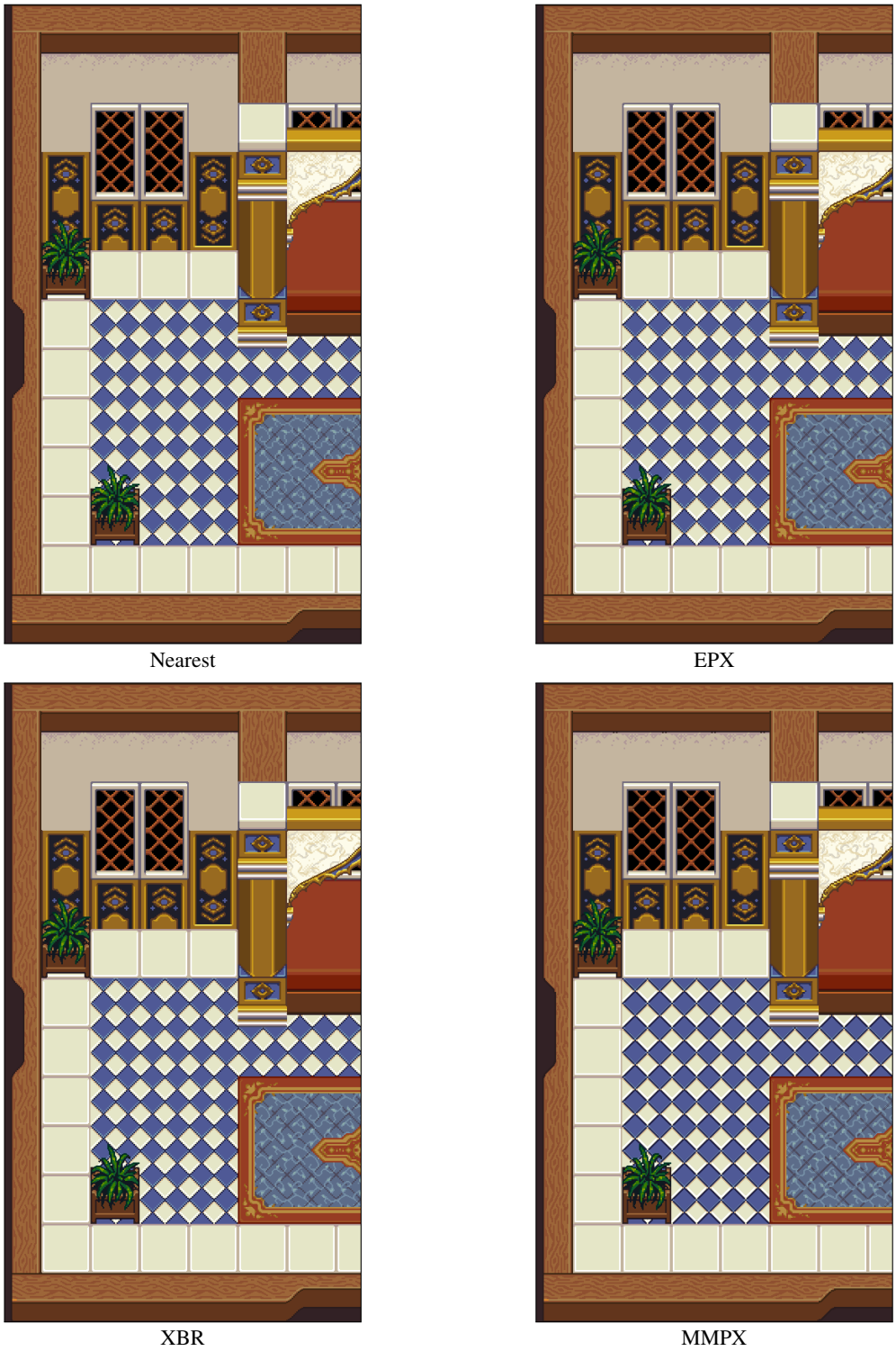
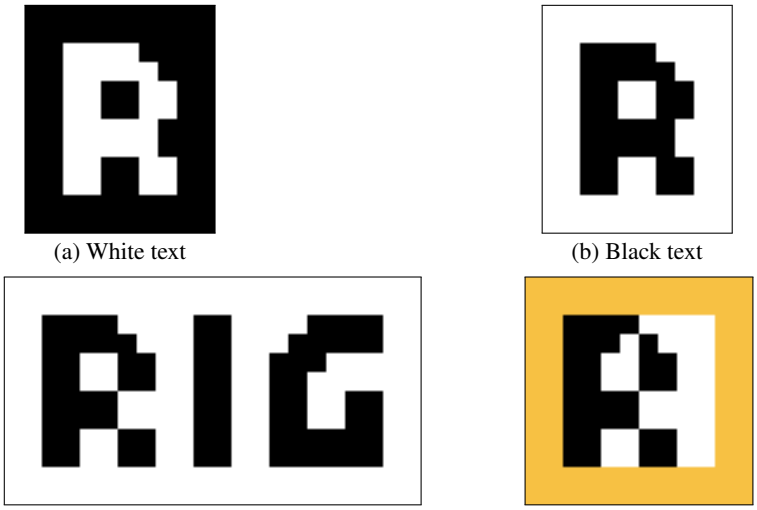



**Figure 19**. Crop of a full-screen pixel-art image processed with Nearest, EPX, XBR, and MMPX. 



(a) White text (b) Black text

(c) Nearby features (d)Varying background

**Figure 20.** (a, b) Independent of luminance or alpha, the MMPX rules produce ideal results on the "R" test when the foreground is unambiguous because it is surrounded by a consistent background. This occurs in a font sheet or spritesheet. (c, d) Where the local neighborhood leaves foreground and background ambiguous, MMPX conservatively falls back to Nearest at those pixels. This can occur in full-screen images.

Figure 20 shows MMPX results on the ambiguous positive/negative space "R" test case that we used to motivate the intersection rules. When there are no other features near the ambiguous intersection, as in (a) and (b), the rules refine the leg of the "R" as an artist would. This case occurs in font sheets, and is an example of why processing the input prior to rendering is preferable to processing the output. When other features are nearby as in (c) it falls back to Nearest because a much larger (and thus slower and more complex) kernel would be needed to discriminate foreground from background. In truly ambiguous cases such as (d), Nearest is the best objective result so the rules fall back to it.

We stress that while the results in this section showed that MMPX can improve on the quality of Nearest, EPX, and XBR for *general* pixel art, for any *specific* asset, one filter might be better than another. The LucasArts games often contained fonts and sprites for which EPX's rounding is superior, and for full-screen, high-color pixel art HQX and XBR-AA can be more consistent with style. One could imagine designing filter rules using the strategies of MMPX tailored to a particular artist or game.

## 6. Performance Evaluation

The measurements in Table 1 demonstrate that MMPX is within the same performance range as alternative pixel-art magnification filters. Our supplement includes C++, JavaScript, and GLSL versions of MMPX and selected previous filters to which

we compare it, as well as the profiling code used in these experiments. We choose those three languages because they are the common source languages for emulators and pixel-art tools. Note that GLSL shaders can be run from OpenGL ES on an embedded processor or mobile, from WebGL in a browser, or from desktop OpenGL or Vulkan. We did not evaluate C++ compiled to WebAssembly because the performance in that case will be bracketed by native C++ and JavaScript.

We implemented Nearest, MMPX, and EPX ourselves. We used the following popular, open source implementations of XBR and HQX.

**XBR:** JS Josep del Rio 2020 https://github.com/joseprio/xBRjs
C++ Treeki 2015 https://github.com/Treeki/libxbr-standalone
GLSL Hyllian 2016 https://github.com/libretro/glsl-shaders/blob/master/xbr

**HQX:** JS Eliastik 2015 https://github.com/Eliastik/javascript-hqx
C++ Treeki 2015 https://github.com/Treeki/libxbr-standalone
GLSL Jules Blok 2014 https://github.com/CrossVR/hqx-shader

| **JavaScript** | RPi | Nano | Netbook | Laptop | Desktop |
|---|---|---|---|---|---|
| Nearest | 3.7 | 3.4 | 1.0 | 1.2 | 0.7 |
| EPX | 6.3 | 10.2 | 2.9 | 3.3 | 2.0 |
| MMPX | 140.5 | 126.4 | 32.7 | 39.9 | 25.3 |
| HQX | 215.7 | 229.7 | 52.6 | 58.7 | 36.4 |
| XBR | 465.3 | 491.8 | 135.7 | 144.4 | 91.5 |
| **C++** | | | | | |
| Nearest | 3.0 | 0.6 | 0.3 | 0.7 | 0.3 |
| EPX | 4.3 | 5.8 | 2.6 | 3.1 | 1.6 |
| MMPX | 15.7 | 23.2 | 7.4 | 9.1 | 4.9 |
| HQX | 836.3 | 516.9 | 96.0 | 130.5 | 71.2 |
| XBR | 841.7 | 526.3 | 100.5 | 130.0 | 71.4 |
| **GLSL** | | | | | |
| Nearest$^{C}$ | N/A | 0.170 | 0.050 | 0.015 | 0.004 |
| Nearest$^{P}$ | N/A | 0.202 | 0.085 | 0.019 | 0.005 |
| EPX$^{C}$ | N/A | 0.317 | 0.076 | 0.019 | 0.004 |
| MMPX$^{C}$ | N/A | 0.955 | 0.156 | 0.039 | 0.005 |
| HQX$^{P}$ | N/A | 2.631 | N/A | 0.159 | 0.025 |
| XBR$^{P}$ | N/A | 3.457 | 0.986 | 0.204 | 0.038 |

**Table 1**. Performance in nanoseconds per pixel (lower is better), measured over 50 CPU trials on $512 \times 512$ input and 5000 GPU trials on $1024 \times 1024$ input, after cache and JIT warmup. For context, $57\text{ns/pix}$ is the throughput required to scale a SNES $256 \times 240$ full screen to $512 \times 480$ in $1\text{ms}$ per frame. "P" = pixel shader, "C" = compute shader. "N/A" = entry is not applicable because RPi does not support GPU timing and the HQX OpenGL ES shader does not compile on Intel or Broadcom GPUs.

The `README.md` file in the source code supplement for this paper gives the full copyright information for these implementations.

These implementations (including our own) are designed for readability and easy porting between the many platforms on which emulators, educational software, and fantasy consoles must run. So, they do not explicitly use CPU threading, platform-specific CPU SIMD features such as SSE and AVX, or GPU features beyond base specification OpenGL ES 3.1.

We measured GPU execution time with a `GL_TIME_ELAPSED` query, native CPU C++ execution time with `std::chrono::system_clock::now()` on Visual Studio (Windows) and clang++ (Linux), and CPU JavaScript time with `performance.now()` in Chrome 86. In each case, we processed a $512 \times 512$ input image 50 times and then reported time per output pixel in nanoseconds for the 67 million output pixels computed. We warmed caches and shader JITs by running the full test multiple times and recording only the final 50-image run.

Our test image combines diverse Creative Commons pixel-art sprites, fonts, and full-screen images to cover all branches from the algorithms.

HQX is very slow for the JIT to compile in JavaScript. The first few executions can take seconds or minutes as a result. The numbers here are after the JIT has compiled and when timing is at steady state.

Note that the Native C++ XBR and HQX implementations from the third-party libXBR are slower than the JavaScript implementations on many platforms. We believe that it is possible to implement those algorithms more efficiently in C++.

However, identifying the fastest XBR and HQX implementations is beyond the scope of this paper. For peak performance the GLSL implementations are the most interesting, and regardless, the goal of these experiments is only to show that MMPX is well within the performance profile of some available implementations already in production use for emulators, which it clearly is.

On all platforms MMPX is fast enough to operate within a sprite renderer for scaling and rotation of small numbers of sprites. It can be applied for load-time MIP-map generation at negligible additional cost over disk/network access and image decompression, and it is fast enough to use within an interactive content-creation tool even for megapixel images. On every platform configuration except JavaScript on the embedded systems, MMPX can process runtime SNES-resolution full frames in less than a millisecond. Even on RPi, MMPX can process a full SNES frame in two milliseconds in a browser, which is sufficient for real-time emulators because that 4-core processor can run the magnification on a separate one from the game.

## 7. Conclusions

MMPX addresses a niche problem in computer graphics in a way that is practical and interesting. We hope that it also provides insight into the challenges and conven-

tions of drawing recognizable features at small scales and demonstrates how sensitive the eye is to slight rounding or squaring of features in the abstracted style of pixel art. MMPX has already proven useful for over a year as a tool for the quadplay and PICO-8 fantasy consoles. It provides another magnification-filter option for the many pixel-art indie game titles, retro re-releases from major vendors, and pixel-art content-creation tools that use its predecessors such as EPX and HQX. We intend the observations on pixel-art filtering, test data, and evaluation methodology in this paper as a resource supporting further innovation in this area.

Having worked with the filtering problem for some time, we remain in awe of EPX's economy. It produces significant quality from its four simple rules, and we needed much more complexity to address the cases in which one can do better than EPX. While we believe additional rules for further improving filtering quality await discovery, that discovery will not be easy. We attempted and rejected many rules as too expensive to justify the diminishing increase in quality beyond what MMPX currently achieves on top of EPX.

Designing the algorithm to run efficiently under the four different processing models of interpreted JavaScript, native C++, full OpenGL, and OpenGL ES for embedded systems was a significant challenge. We found that we needed per-platform optimizations for reading and writing pixels efficiently. Yet, we were able to create a single algorithm body that runs efficiently on all four targets and remains elegant, relatively understandable, and free of data or library dependencies.

There are a few interesting limitations and directions for future work on this problem. For 1:1 slopes, we were able to avoid luminance bias on thin stripes by corresponding downstream patterns. However, that necessarily created inconsistency between the filtering of light and dark disk shapes that we showed as an undesirable result in Figure 15 and varying stripe thickness in Figure 14. We have not yet found a set of patterns that can efficiently avoid this bias for 2:1 edges, or a better way to break ties between opaque pixels than by luminance. For example, we rejected breaking ties by pixel position because it destroys translation invariance and causes shapes to crawl as they move across the screen and disrupts the slope of diagonal lines. An ideal magnification filter would detect and maintain special pixel-art conventions for individual pixels, such as dither patterns and single-pixel outlines. Doing so for specific colors (e.g., black) or uniform dithering is relatively easy, but recognizing arbitrary outline colors and dithered gradients is challenging.

As in most previous work, MMPX operates as a single-pass filter with a fixed maximum kernel size. This simplifies the implementation and gives predictable performance. An inherent limitation of this is that it is sometimes ambiguous which color is positive space/foreground when only looking at the context of the small kernel. Our solution has three parts: 1) apply the heuristic that dark is foreground and light is background within the algorithm, 2) recommend magnifying font and spritesheets

instead of the final image to increase the chance encountering of an isolated figure surrounded by transparent pixels, and 3) do no harm.

The heuristic that dark indicates positive space is motivated by the observation that many games use dark text on light backgrounds and black outlines around sprites. For a given set of content, the luminance tests can of course be inverted, as we do in our JavaScript demo application when a binary font sheet is detected with a black background instead of one with a transparent or white background. To avoid "doing harm," in situations as in Figure 11, where heuristically preferring one color over another would disconnect a feature of the other color, MMPX simply does nothing and falls back on Nearest.

An alternative approach for future work is to abandon the fixed kernel size or to make multiple passes when resolving ambiguous patterns. For example, one could adopt techniques from post-process antialiasing methods such as MLAA and FXAA (see the survey by Jimenez et al. [2011] and Getreuer et al.'s [2011]) to search along feature edges and adaptively extend the kernel as needed.

Deep neural networks have proven exceptionally well-suited to image inpainting and superresolution problems, albeit for natural images with large, labelled training sets. There are some reasons to think that DNNs are a poor fit for the style-preserving pixel-art scaling problem when applied in the same way. Large amounts of hand-scaled pixel art are not available for supervised training. The cases where DNNs succeeded at image filtering tended to be ones where the pixel values were on a continuous, interpolatable range with large features. For a limited palette and single-pixel features, we speculate that previous networks designed for natural images will significantly overblur.

However, other machine-learning methods have excelled at binary segmentation and classification problems. If we pose pixel-art scaling as classifying which pixels are art of the same feature (similar to how EPX and MMPX operate), then DNNs may be a good choice. A separate challenge with DNNs for this problem is bandwidth. MMPX reads only about four input pixels per output pixel due to amortization and a small working-set size that fits into CPU stack or GPU registers. In contrast, a typical DNN will use hundreds or thousands of weights, connections, and inputs per pixel.

Scaling by $2\times$ is the smallest integer ratio on which to build a MIP chain, and it maintains efficient power-of-two sprite sizes. As with most previous filters, MMPX can perform $4\times$ scaling by simply running the $2\times$ magnification process twice. Although less common, $3\times$ scaling could be desirable to provide better intermediate stages for sprite and font zooming. HQX has a native $3\times$ scaling alternative that is a different filter from its $2\times$ variant, and we believe that a MMPX variant specifically designed for $3\times$ is worth investigation. In some ways, $3\times$ is easier: every $3\times3$ output tile has a center pixel that is copied directly from the source, which avoids some of the directional biasing problems that we had to resolve at $2\times$. However, all of the

```
typedef uint32_t ABGR8;

inline ABGR8 src(int x, int y) {
    // Use a single branch because clamping is only needed rarely, and
    // perform an unsigned test so that negatives wrap around and fail.
    if (uint32_t(x) > uint32_t(srcMaxX) || uint32_t(y) > uint32_t(srcMaxY)) {
        x = clamp(x, 0, srcMaxX); y = clamp(y, 0, srcMaxY);
    }
    return srcBuffer[y * srcWidth + x];
}

...

for (int srcY = 0; srcY < srcHeight; ++srcY) {
    int srcX = 0;

    // Inputs carried along rows
    ABGR8 A = src(srcX-1, srcY-1), B = src(srcX, srcY-1), C = src(srcX+1,
        srcY-1),
          D = src(srcX-1, srcY+0), E = src(srcX, srcY+0), F = src(srcX+1,
              srcY+0),
          G = src(srcX-1, srcY+1), H = src(srcX, srcY+1), I = src(srcX+1,
              srcY+1);

    ABGR8 Q = src(srcX - 2, srcY), R = src(srcX + 2, srcY);

    for (srcX = 0; srcX < srcWidth; ++srcX) {

        // ...main body here...

        int dstIndex = ((srcX + srcX) + (srcY << 2) * srcWidth) >> 0;
        ABGR8* dstPacked = (ABGR8*)dst + dstIndex;

        *dstPacked = J; dstPacked++;
        *dstPacked = K; dstPacked += srcWidth + srcMaxX;
        *dstPacked = L; dstPacked++;
        *dstPacked = M;

        A = B; B = C; C = src(srcX + 2, srcY - 1);
        Q = D; D = E; E = F; F = R; R = src(srcX + 3, srcY);
        G = H; H = I; I = src(srcX + 2, srcY + 1);
    }
}
```

**Listing 5**. Single-threaded scalar C++ framework.

neighborhood rules we introduced would have to be reconsidered specifically for the case of the nine outputs per input pixel.

## Acknowledgements

We thank Peter Shirley for editing this manuscript.

```
#version 310 es
#define ABGR8 uint

layout(local_size_x = 8, local_size_y = 8) in;

layout(std430, binding = 0) restrict writeonly buffer outputBuffer {
    ABGR8 dst_buffer[];
};

layout(std430, binding = 1) restrict readonly buffer inputBuffer {
    ABGR8 src_buffer[];
};

uniform ivec2       dst_size;
uniform ivec2       src_size;
uniform ivec2       src_max;

// Read a source pixel, clamping to bounds
ABGR8 src(int x, int y) {
    return src_buffer[clamp(x, 0, src_max.x) + clamp(y, 0, src_max.y) *
        src_size.x];
}

void main () {
    int srcX = int(gl_GlobalInvocationID.x), srcY = int(gl_GlobalInvocationID
        .y);

    ABGR8 A = src(srcX-1, srcY-1), B = src(srcX, srcY-1), C = src(srcX+1,
        srcY-1),
          D = src(srcX-1, srcY+0), E = src(srcX, srcY+0), F = src(srcX+1,
              srcY+0),
          G = src(srcX-1, srcY+1), H = src(srcX, srcY+1), I = src(srcX+1,
              srcY+1);

    // ...main body here...

    // Write four pixels at once
    int dst_index = 2 * srcY * dst_size.x + 2 * srcX;
    dst_buffer[dst_index] = J; ++dst_index;
    dst_buffer[dst_index] = K; dst_index += dst_size.x - 1;
    dst_buffer[dst_index] = L; ++dst_index;
    dst_buffer[dst_index] = M;
}
```

**Listing 6**. GLSL compute shader framework.

## References

BURNES, A., 2020. NVIDIA DLSS 2.0: A big leap in AI rendering, March. Blog Post. URL: https://www.nvidia.com/en-us/geforce/news/nvidia-dlss-2-0-a-big-leap-in-ai-rendering/. 90

COEURJOLLY, D., GUETH, P., AND LACHAUD, J.-O. 2018. Regularization of voxel art. In *ACM SIGGRAPH 2018 Talks*, Association for Computing Machinery, New York, NY,

USA, SIGGRAPH '18. URL: https://doi.org/10.1145/3214745.3214748. 92

DAI, D., TIMOFTE, R., AND VAN GOOL, L. 2015. Jointly optimized regressors for image super-resolution. *Computer Graphics Forum 34*, 2, 95–104. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12544. 90

GATYS, L. A., ECKER, A. S., AND BETHGE, M., 2015. A neural algorithm of artistic style. URL: https://arXiv.org/abs/1508.06576, arXiv:1508.06576. 92

GETREUER, P. 2011. Contour stencils: Total variation along curves for adaptive image interpolation. *SIAM Journal on Imaging Sciences 4*, 3, 954–979. URL: https://doi.org/10.1137/100802785. 112

GLASSNER, A. 2015. Interpreting alpha. *Journal of Computer Graphics Techniques (JCGT) 4*, 2 (May), 30–44. URL: http://jcgt.org/published/0004/02/03/. 89

HAN, C., WEN, Q., HE, S., ZHU, Q., TAN, Y., HAN, G., AND WONG, T.-T. 2018. Deep unsupervised pixelization. *ACM Trans. Graph. 37*, 6 (Dec.). URL: https://doi.org/10.1145/3272127.3275082. 92

HYLLIAN, 2011. Xbr. URL: https://github.com/Hyllian/glsl-shaders/blob/master/xbr/shaders/xbr-lv2.glsl. 91

JIMENEZ, J., GUTIERREZ, D., YANG, J., RESHETOV, A., DEMOREUILLE, P., BERGHOFF, T., PERTHUIS, C., YU, H., MCGUIRE, M., LOTTES, T., MALAN, H., PERSSON, E., ANDREEV, D., AND SOUSA, T. 2011. Filtering approaches for real-time anti-aliasing. In *ACM SIGGRAPH Courses*. Association for Computing Machinery, New York, NY, USA. URL: https://doi.org/10.1145/2037636.2037642. 112

JOHNSTON, E., 1992. EPX. Algorithm in the Macintosh port of the LucasArts SCUMM VM engine, Lucas Arts. 90, 94

KOPF, J., AND LISCHINSKI, D. 2011. Depixelizing pixel art. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2011) 30*, 4, 99:1 – 99:8. URL: https://doi.org/10.1145/2010324.1964994. 92

KREUZER, F., KOPF, J., AND WIMMER, M. 2015. Depixelizing pixel art in real-time. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, i3D '15. Association for Computing Machinery, New York, NY, USA, 130. URL: https://doi.org/10.1145/2699276.2721395. 92

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph. 21*, 4 (Aug.), 163–169. URL: https://doi.org/10.1145/37402.37422. 90

MAZZOLENI, A., 2001. Scale 2x. Website visited 2020-01-01. URL: https://www.scale2x.it/. 90, 94

PARK, T., LIU, M., WANG, T., AND ZHU, J. 2019. Semantic image synthesis with spatially-adaptive normalization. URL: https://arXiv.org/abs/1903.07291, arXiv:1903.07291. 92

REBOUAS SERPA, Y., AND FORMICO RODRIGUES, M. A. 2019. Towards machine-learning assisted asset generation for games: A study on pixel art sprite sheets. In *2019 18th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*, 182–191. URL: https://ieeexplore.ieee.org/document/8924853. 92

SALVI, M., PATNEY, A., LEFOHN, A. E., AND BRITTAIN, D. L., 2017. Temporally stable data reconstruction with an external recurrent neural network, 7. US Patent Application 20190035113A1. 90

STASIK, P. M., AND BALCEREK, J. 2017. Improvements in upscaling of pixel art. In *2017 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*. IEEE, Washington, dC, USA, 371–376. URL: https://ieeexplore.ieee.org/document/8166895. 91

STEPIN, M., 2003. hq2x, October. Website. URL: https://web.archive.org/web/20131205091805/http://www.hiend3d.com/hq2x.html. 90, 101

THOMAS, K. 1999. Fast blit strategies: A mac programmer's guide. *MacTech 15*, 6. URL: http://preserve.mactech.com/articles/mactech/Vol.15/15.06/FastBlitStrategies/index.html. 90

TOMASI, C., AND MANDUCHI, R. 1998. Bilateral filtering for gray and color images. In *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*. IEEE, Washington, DC, USA, 839–846. URL: https://ieeexplore.ieee.org/document/710815. 89

TURKOWSKI, K. 1990. *Filters for Common Resampling Tasks*. Academic Press Professional, Inc., Cambridge, MA, USA, 147–165. URL: https://dl.acm.org/doi/10.5555/90767.90805. 89

XENOWHIRL, 2007. Sprite rotation utility, Jan. Sonic Retro forum thread. URL: https://web.archive.org/web/20210506191421/https://forums.sonicretro.org/index.php?threads/sprite-rotation-utility.8848/. 86, 91

XIAO, L., NOURI, S., CHAPMAN, M., FIX, A., LANMAN, D., AND KAPLANYAN, A. 2020. Neural supersampling for real-time rendering. *ACM Trans. Graph. 39*, 4 (July). URL: https://doi.org/10.1145/3386569.3392376. 90

## Index of Supplemental Materials

The supplemental materials as described below can be found at http://jcgt.org/published/0010/02/04/supplement.zip.

**tests/** Source images used for all tests. The `license.txt` file gives copyrights and licenses for all content.

**results/** Selected MMPX, Nearest, EPX, XBR, XBR-Antialiased, Biased Bilinear, Unbiased Bilinear, Bicubic, and HQX results. `js-demo.html` can produce additional results.

**code/**

| | |
|---|---|
| `js-demo.html` | HTML/JavaScript single-file interactive demo, reference implementation, and profiling harness for MMPX, Nearest, EPX, XBR, XBR-AA, HQX, Unbiased Bilinear, and Biased Bilinear |
| `libxbr/` | C++ implementation of XBR |
| `data-files/` | GLSL implementations of MMPX, Nearest, Bilinear, EPX, XBR, and HQX |
| `source/` | |
| `hq2x.c` | C++ implementation of HQX |
| `cppPerf.cpp` | C++ implementations of MMPX, Nearest, Bilinear, and EPX, and the C++ profiling harness |
| `glslPerf.cpp` | C++ profiling harness for GLSL code |

## Author Contact Information

Morgan McGuire
University of Waterloo
200 University Avenue West
Waterloo, ON, Canada N2L 3G1
morgan@casual-effects.com
https://casual-effects.com

Mara Gagiu
University of Waterloo
200 University Avenue West
Waterloo, ON, Canada N2L 3G1
mara.gagiu@uwaterloo.ca

McGuire and Gagiu, MMPX Style-Preserving Pixel Art Magnification, *Journal of Computer Graphics Techniques (JCGT)*, vol. 10, no. 2, 83–117, 2021
https://jcgt.org/published/0010/02/04/

| | | | |
|---|---|---|---|
| Received: | 2021-01-26 | | |
| Recommended: | 2021-04-29 | Corresponding Editor: | Angelo Pesce |
| Published: | 2021-06-30 | Editor-in-Chief: | Marc Olano |