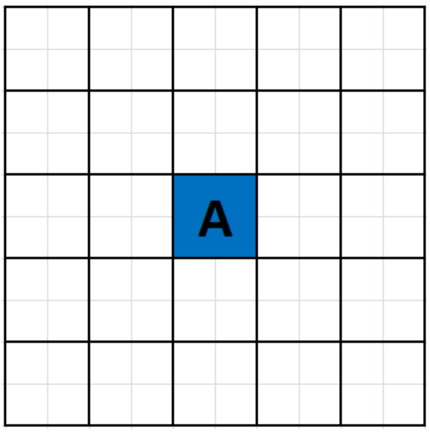
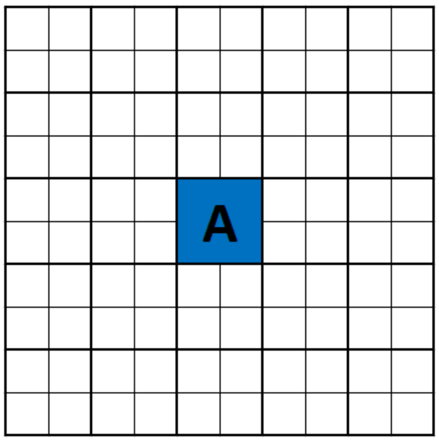
This is a brief explanation on how Super-xBR works.

Super-xBR is an algorithm developed by me in 2015. It uses some combinations of known linear filters along xBR edge detection rules in a non-linear way. It works in two passes and can only scale an image by 2 (or multiples of 2 by reapplying it).

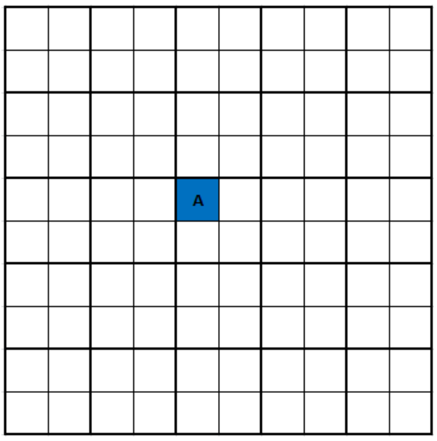
Think about a source image and pick up a small region of it (5x5 pixels). See pixel grid picked up from the source image below:



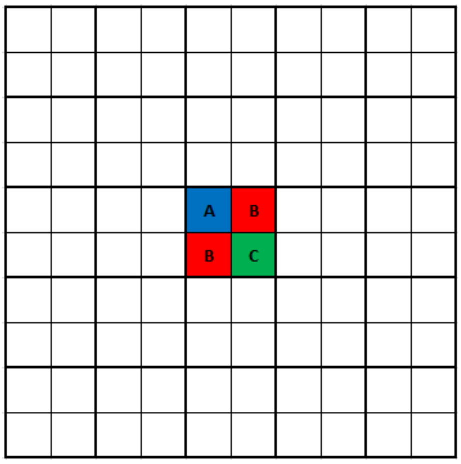
Here we have a central pixel A from the source image in its original resolution. The first step is to subdivide the grid in two subpixels for each dimension, like this:



The first assumption we'll have to take here is to assume that the subpixel located at the upper-left position gets the same value as the original pixel A:

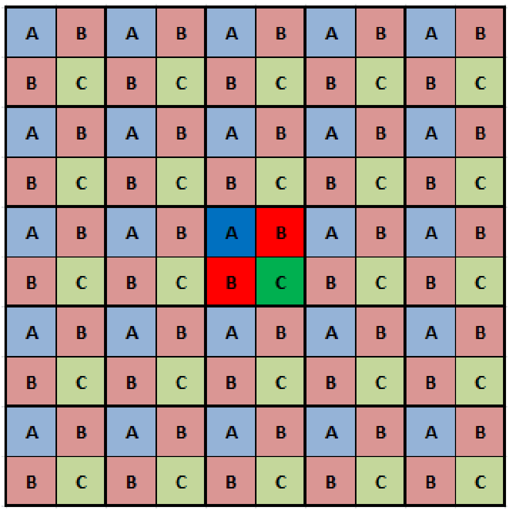


This is a common assumption in image processing area. There is another approach to this, but I won't give the details here. So, where the original pixel A was located, we now have four subpixels, three of them are unknown:



From now on, I'll call the subpixels as pixels, because, well, they are pixels anyway! ☺

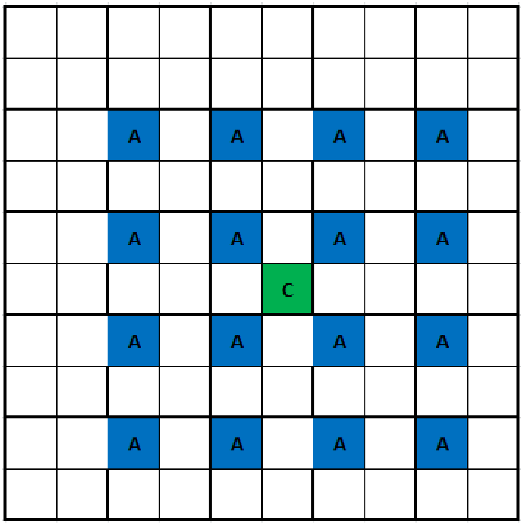
As you can see, there are three types of pixels in the grid: A, B and C. Pixel A is a known pixel and has the same value as the original in the source image. Pixels B are adjacent neighbors to A and are unknown. Pixel C is a diagonal neighbor to A and is unknown too. We can see how the grid will look by generalizing the idea throughout of it:



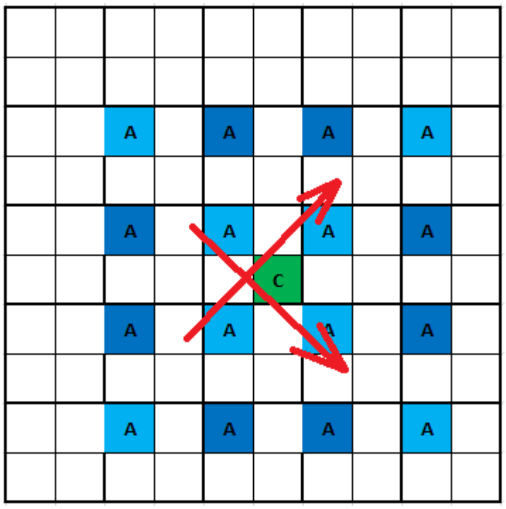
Pass 1

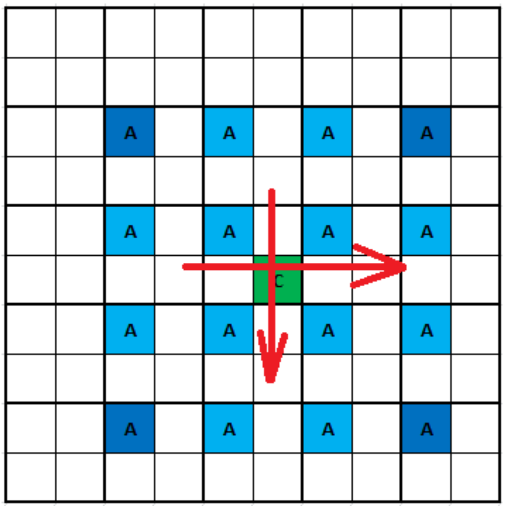
Now we have a grid completely described and the objective of Super-xBR (and any other filter you can think of) is to guess the values of all pixels B and C.

As Super-xBR works in two passes, it firstly estimates a value for the unknown C pixels. For this, Super-xBR uses the first sixteen nearest known pixels to C as inputs, see below:



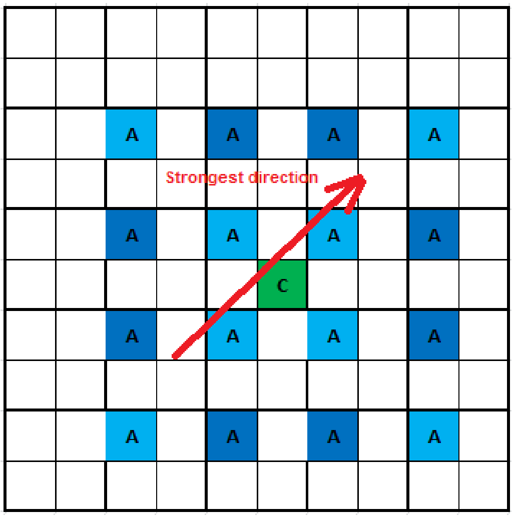
See how centralized is pixel C in relation to the pixel A neighbors. Pixel C can be estimated in many ways and this is where Super-xBR differentiates from other filters. It employs a non-linear approach that I'll explain now.

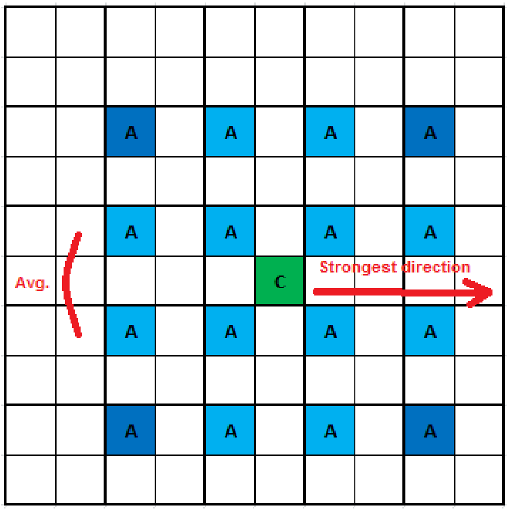




To estimate pixel C, firstly Super-xBR has to get the two strongest edge directions. It considers that edges can have two possible directions, one in vertical/horizontal direction and the other in diagonal direction. So, it compares the two diagonal directions and chooses the strongest (based in an xBR heuristic) one. And apply the same method for the horizontal/vertical directions. The implementation of edge calculation can vary with the chosen method. In my Super-xBR implementation it employs some heuristics I had developed for my other xBR algorithm (used only by pixel art games) and it works very well with Super-xBR (that's why I use this name).

After the two strongest edge directions are calculated, the next step is to estimate a value for pixel C using each direction. This means we'll get two C estimates based on each edge direction. The estimation can be done by applying a linear filtering in each direction. So, for the diagonal direction, the four known A pixels are used to estimate a C_1 value (by using the four A pixels from the strongest diagonal direction as input to a 2-tap sinc filter, for example). For the horizontal/vertical direction, there are eight pixels A, so firstly it's necessary to average A neighbors 2 by 2 so that we get four A averaged pixels and then apply the same 2-tap sinc filter and calculate a C_2 value.

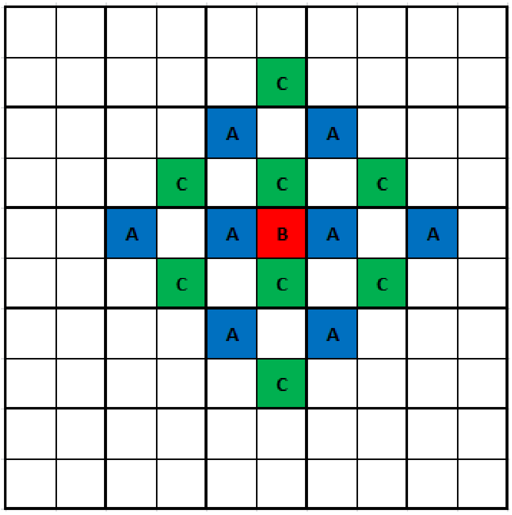




With C_1 and C_2, the last Super-xBR step before second pass is to blend these two C values to obtain final C value estimation. The implementation of that blend can vary. I chose to use the strength of the diagonal edge as a guide to the blend process. The implementation can be seen at the final of this explanation.

Pass 2

The second pass is analogous to the first one, with some small changes. As we now have access to all C values, we can apply the same method to estimate B pixel values. See below the new grid formed to estimate a B pixel:



If you rotate that grid by 45 degrees, it's exactly the same grid as the one in the first pass. The only difference is that now we use pixels A and C as inputs, instead of only pixels A.

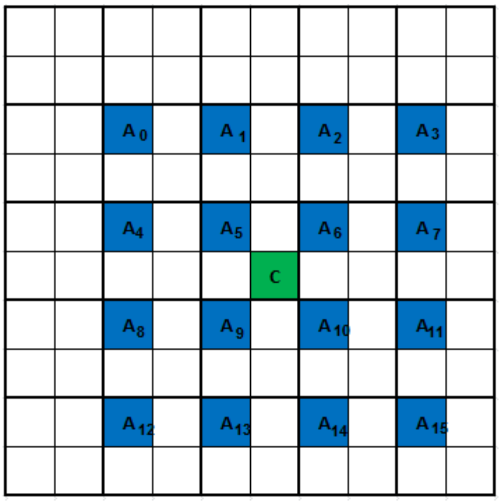
Implementations

Choosing the strongest directions

To compare the strength of each direction, it's necessary a method to evaluate the edge strength. The xBR algorithm calculates the edge strength by adding weighted pixel luma differences.

If pixel A and pixel B have luma values, the absolute difference between them is defined by:

```
float df(float A, float B) {return abs(A-B);}
```



So, calculating the luma values for the sixteen pixels in the grid, we can evaluate the edge strength using these formulas:

For diagonal directions (for left-down to right-upper diagonal below):

$D\_Str = 2*[df(A_5,A_2) + df(A_5,A_8) + df(A_{10},A_{13}) + df(A_{10},A_7)] + df(A_6,A_3) + df(A_9,A_{12}) - [df(A_9,A_3) + df(A_6,A_{12})] + 4*df(A_6,A_9) - [df(A_2,A_8) + df(A_7,A_{13})] + df(A_1,A_4) + df(A_{14},A_{11})$

For horizontal/vertical directions (for horizontal direction below):

$HV\_Str = 4*(df(A_5,A_6)+df(A_9,A_{10})) + 2*(df(A_5,A_4)+df(A_6,A_7)+df(A_8,A_9)+df(A_{10},A_{11})) - [df(A_5,A_7)+df(A_9,A_{11})+df(A_4,A_6)+df(A_8,A_{10})]$

For the other directions, you just need to rotate the grid and use the correspondent pixels. So, the two diagonal strengths are compared and the strongest is chosen. The same comparison is done for horizontal/vertical directions and the strongest is chosen too.

Blending the two color estimates from the strongest directions

The blending process is heuristically developed. I have noticed that if the diagonal edges were too strong, then the horizontal/vertical color estimate was a bit useless. On the other hand, if the diagonal strength was too weak when compared to horizontal/vertical ones, the blend should consider both edges as relevant. So, my heuristic implementation is to use two formulas:

```
float edge_strength = smoothstep(0.0, limits, abs(D_Str1 - D_Str2));

float3 final_C =  lerp(C_hv_edge, C_d_edge,  edge_strength);
```

Where *limits* is an input parameter that can vary from 0.0 to 5.0.