This is an old topic I made in another forum. As Retroarch benefits from this filter, I decided to post here too. Sorry if some format are broken.

xBR Algorithm

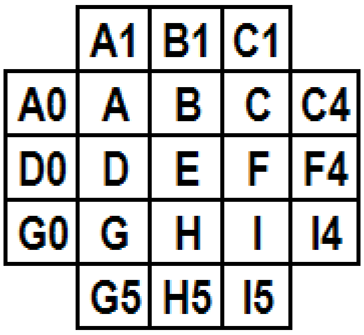
Ok, here it is the algorithm. I've divided the explanation in modules. Some of them are obligatory (red) and others are optionals (green). You can implement the filter using only the obligatory modules and enhance it by implementing the optional modules later (though they may cost performance). So, let's begin!

Introduction

This filter works by detecting edges and interpolating pixels along those edges. Edges are pixel regions in the image were pixels are very distinct (high color frequency) among them along some direction and very similar (low color frequency) perpendicular to that direction. I've divided it basically in two modules: Edge Detection Rule (EDR) and Interpolation Rules (IRs).

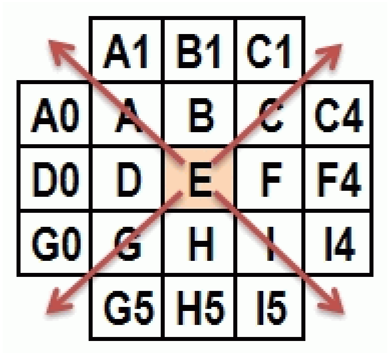
1) Edge Detection Rule (EDR)

Consider a pixel E and its neighbors like the configuration below:

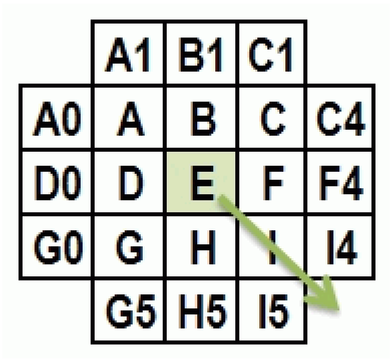


The nomenclature is that way because I'm used to it and facilitate my memorization.

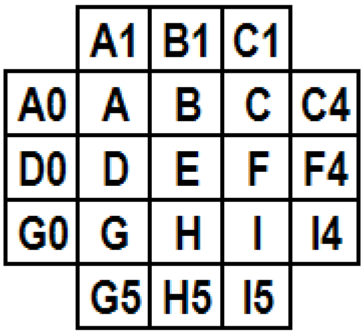
Before introducing the EDR, some symmetries must be shown. In the image below, it can be seen that the pixel E has a four-edge symmetry, so that any rules applied to an edge, must be applied to the others.



For that reason, from now on, all algorithm terms will be related to the down-right edge. You'll have to apply the same rules to the other three edges by symmetry.

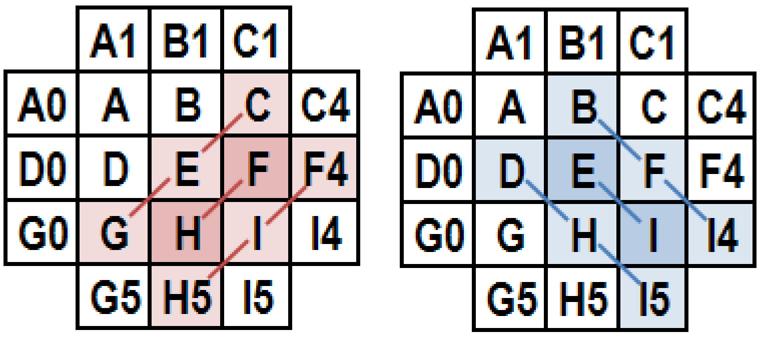


At last, the EDR! It took me some months thinking until I figured out this simple rule. Consider the first picture again:



I have to know if there's an edge along pixels H and F. If yes, then pixel E must be interpolated. If no, then the predominant edge is along pixels E and I, so that E doesn't need to be interpolated. So, how do I know that?

Consider the pictures below:



If the weighted distance (wd) among pixels in those red directions are smaller than those in blue, then I can assume there's a predominant edge in the H-F direction. Here's my EDR:

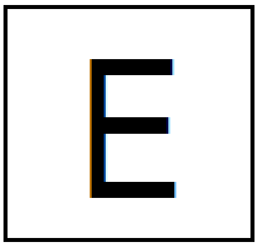
```
wd(red) = d(E,C) + d(E,G) + d(I,F4) + d(I,H5) + 4*d(H,F)

wd(blue)= d(H,D) + d(H,I5) + d(F,I4) + d(F,B) + 4*d(E,I)

EDR = (wd(red) < wd(blue))
```

EDR is a bool variable.

obs: The d function is presented in Annex 1.

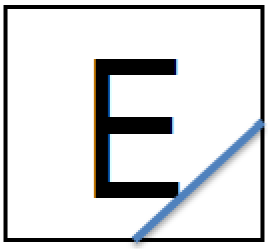
2) Interpolation Rules (IRs)

Let's expand the pixel E and see how the interpolation works.



2.1) Interpolation level 1

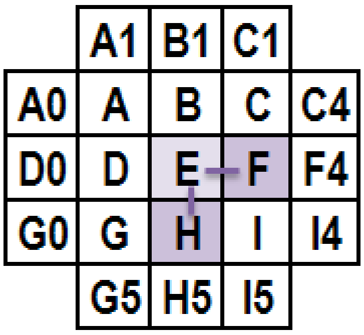
This interpolation is the most basic and only can address 45 degree diagonals. It must change the region below blue line in this picture:



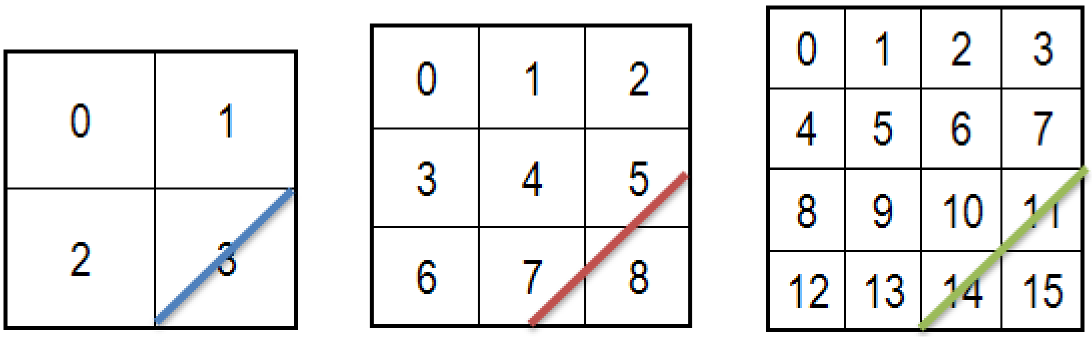
That blue line extends from the middle of F side to the middle of H side.

Once EDR is true, that region below blue line must receive a new color. This new color can be F or H. There's a rule to decide which color it'll receive. This is the rule:

```
new color = (d(E,F) <= d(E,H)) ? F : H;
```



Depending on the scale factor, that region below blue line can encompass a number of pixels. Below you can see which pixels are affected for 2x, 3x and 4x scale factors, respectively:

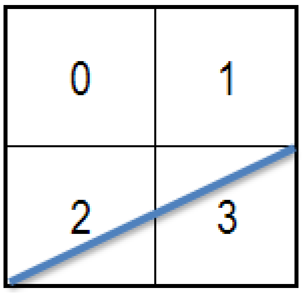


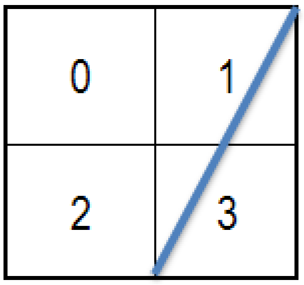
Let me explain what happens to E when you use 2x scale factor. In the picture above (2x), pixel 3 is affected by 50% of its area. So, you have to blend F or H (depending on new color rule) with E in 50%. In pixel shader language, it's the same as mix(new_color, E, 0.5). The same logic you can apply to other scale factor. Just do some geometry calculations, :P!

2.2) Interpolation level 2

This interpolation is easier said than done.

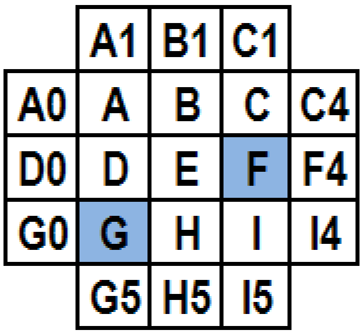
It only happens if interpolation level 1 can happen. It uses the same “new color” calculated for level 1. The difference is the region it changes. See the pictures below:

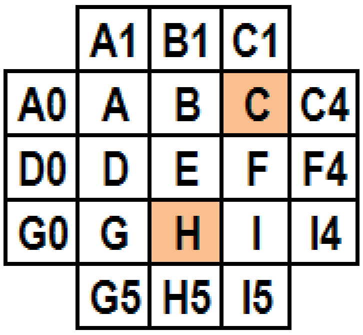




To do this interpolation, you have to test two rules:

```
do_LVL2_left_vertice = (F==G) ? true : false

do_LVL2_up_vertice  = (H==C) ? true : false
```





These two rules are independent. So that, if the left vertice rule is true and the other is false, you can do that LVL2 interpolation for the left vertice region. Also, if the two are true, you must interpolate the intersection of those two LVL2 regions!

You can see that if any of those LVL2 rules are true, the LVL1 rule is satisfied. Then, for implementation, you may test LVL2 rules before LVL1.

3) Font Enhancement

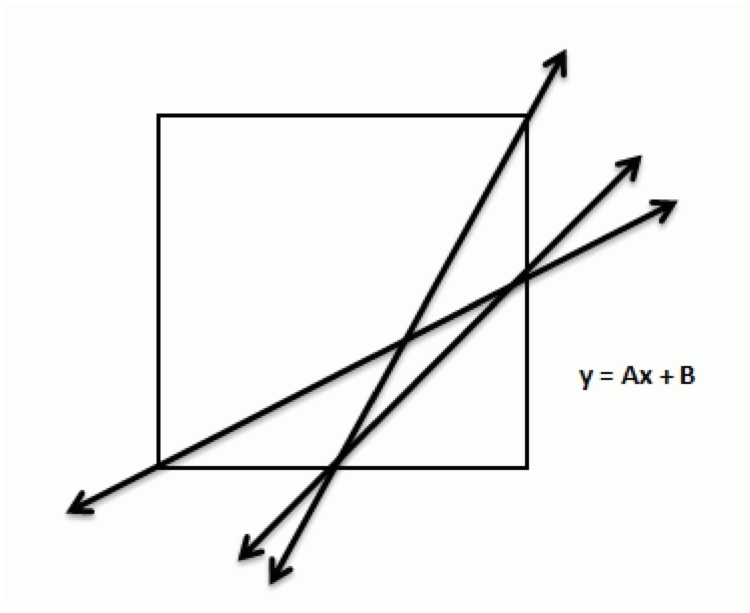
…

4) Advanced Topics

4.1) Using LVL2 Interpolation in regions plagued by shades of colors

…

4.2) Using linear equation of the Straight Line to interpolate in odd scale factors



…

Annex 1: The distance (d) function

d(A,B) calculates the distance of A and B by transforming these pixels in its YUV parts. I use the threshold used by HQx filters: 48*Y* + *7*U + 6*V.

Here’s my slow C implementation of d(A,B):

```
float d(unsigned int A, unsigned int B)
{
    unsigned int r, g, b;
    unsigned int y, u, v;

    b = abs(((A & pg_blue_mask  )>>16) - ((B & pg_blue_mask  )>> 16));
    g = abs(((A & pg_green_mask)>>8  ) - ((B & pg_green_mask )>>  8));
    r = abs(( A & pg_red_mask        ) - ( B & pg_red_mask          ));

    y = abs(0.299*r + 0.587*g + 0.114*b);
    u = abs(-0.169*r - 0.331*g + 0.500*b);
    v = abs(0.500*r - 0.419*g - 0.081*b);

    return 48*y + 7*u + 6*v;
}
```

Annex 2: Pseudo-code implementing XBR LVL1

```
wd(red) = d(E,C) + d(E,G) + d(I,F4) + d(I,H5) + 4*d(H,F);
wd(blue)= d(H,D) + d(H,I5) + d(F,I4) + d(F,B) + 4*d(E,I);
EDR = (wd(red) < wd(blue));

if (EDR)
{
      new_color = (d(E,F) <= d(E,H)) ? F : H;

      do_INT_LVL1(new_color);
}
```

Yes, it's that simple!

Annex 3: Pseudo-code implementing XBR LVL2

```
wd(red) = d(E,C) + d(E,G) + d(I,F4) + d(I,H5) + 4*d(H,F);
wd(blue)= d(H,D) + d(H,I5) + d(F,I4) + d(F,B) + 4*d(E,I);
EDR = (wd(red) < wd(blue));

if (EDR)
{
      new_color = (d(E,F) <= d(E,H)) ? F : H;

      if (F==G && H==C) {
           do INT_LVL2_left(new_color);
           do INT_LVL2_up(new_color);
      }
      else if (F==G) {
           do INT_LVL2_left(new_color);
      }
      else if (H==C) {
           do INT_LVL2_up(new_color);
      }
      else
           do_INT_LVL1(new_color);
}
```

- and repeat for the other three edges according to symmetry.
- the POC I released in the other thread implements this code in shader language.

Adapted from: https://forums.libretro.com/t/xbr-algorithm-tutorial/123
Accessed Sept 2022