Intra Paint Deringing Filter

Jean-Marc Valin

July 10, 2015

1 Introduction

Intra paint is a pixel domain technique designed to reconstruct images based only on a direction and the pixels that lie on block boundaries. Intra paint can be used either for directly encoding images, or as a post-processing deringing filter to reduce coding artefacts.

2 Intra Paint Algorithm

The intra paint algorithm works in four steps:

Step 1: Dividing Into Blocks

The first step is to divide the image into blocks of fixed or variable size (Fig. 1). Variable-size blocks make it possible to use large blocks on long, continuous edges and small blocks where edges intersect or change direction. Fixed block size is easier to implement, especially in the deringing application where we do not wish to signal the sizes on a block-by-block basis. For deringing purposes a block size around 8x8 is generally suitable.

Step 2: Direction Search

Once the image is divided into blocks, we determine which direction best matches the pattern in each block (Fig. 3). The direction is only signaled when operating as an image encoder. One way to determine the direction is to minimize the pixel variance. For each direction, we assign a line number to each pixel, as shown in Fig. 2.

For each direction d, the variance is defined as:

$$\sigma_d^2 = \frac{1}{N} \sum_{k \in \text{block},d} \left[\sum_{p \in P_{d,k}} (x_p - \mu_{d,k})^2 \right] , \tag{1}$$

where x_p is the value of pixel p, $P_{d,k}$ is the set of pixels in line k following direction d, N is the total number of pixels in the block, and μ_k is the pixel average for line k:

$$\mu_{d,k} = \frac{1}{N_{d,k}} \sum_{p \in P_{d,k}} x_p , \tag{2}$$

where $N_{d,k}$ is the cardinality of $P_{d,k}$. Substituting (2) into (1) and simplifying, we get

$$\sigma_d^2 = \frac{1}{N} \left[\sum_{p \in \text{block}} x_p^2 - \sum_{k \in \text{block},d} \frac{1}{N_{d,k}} \left(\sum_{p \in P_{d,k}} x_p \right)^2 \right] , \tag{3}$$

Considering that the first term of Eq. (3) is constant with respect to d, we simply find the optimal direction d_{opt} as:

$$d_{opt} = \max_d \sum_{k \in \text{block},d} \frac{1}{N_{d,k}} \left(\sum_{p \in P_{d,k}} x_p \right)^2 . \tag{4}$$

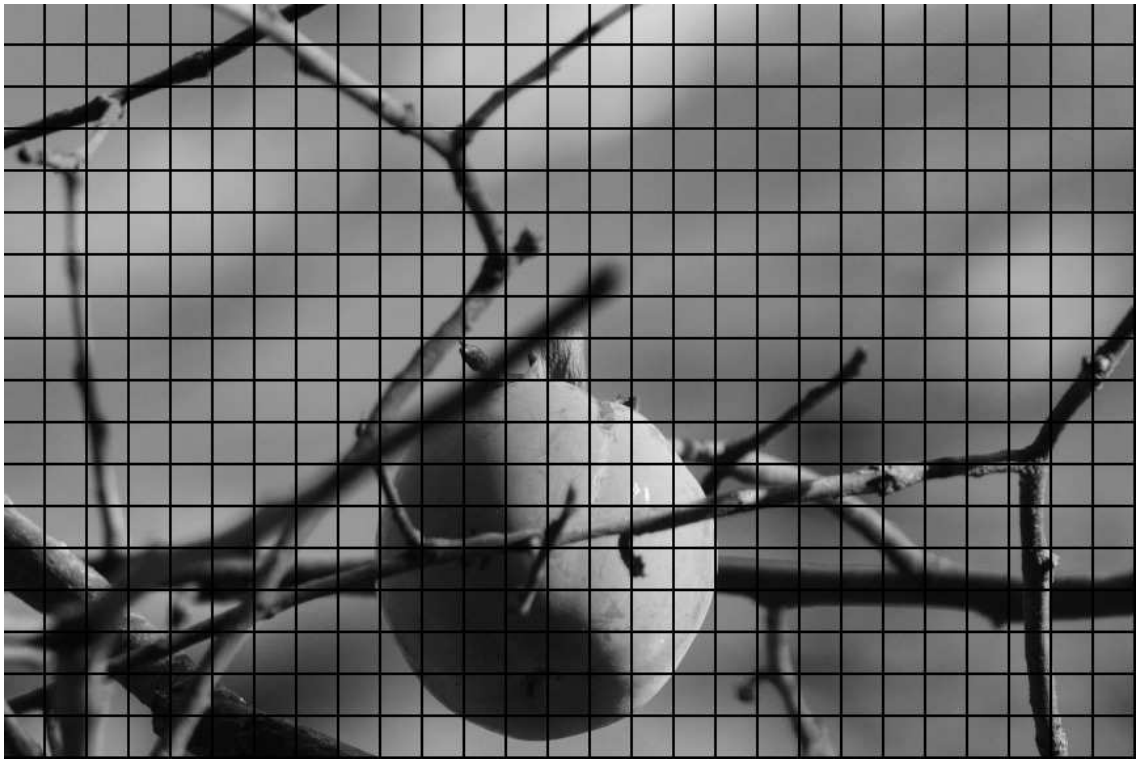

Figure 1: Input image divided into fixed 32x32 blocks.

0	0	1	1	2	2	3	3
1	1	2	2	3	3	4	4
2	2	3	3	4	4	5	5
3	3	4	4	5	5	6	6
4	4	5	5	6	6	7	7
5	5	6	6	7	7	8	8
6	6	7	7	8	8	9	9
7	7	8	8	9	9	10	10

Figure 2: Line numbers for pixels following one direction in an 8x8 block.

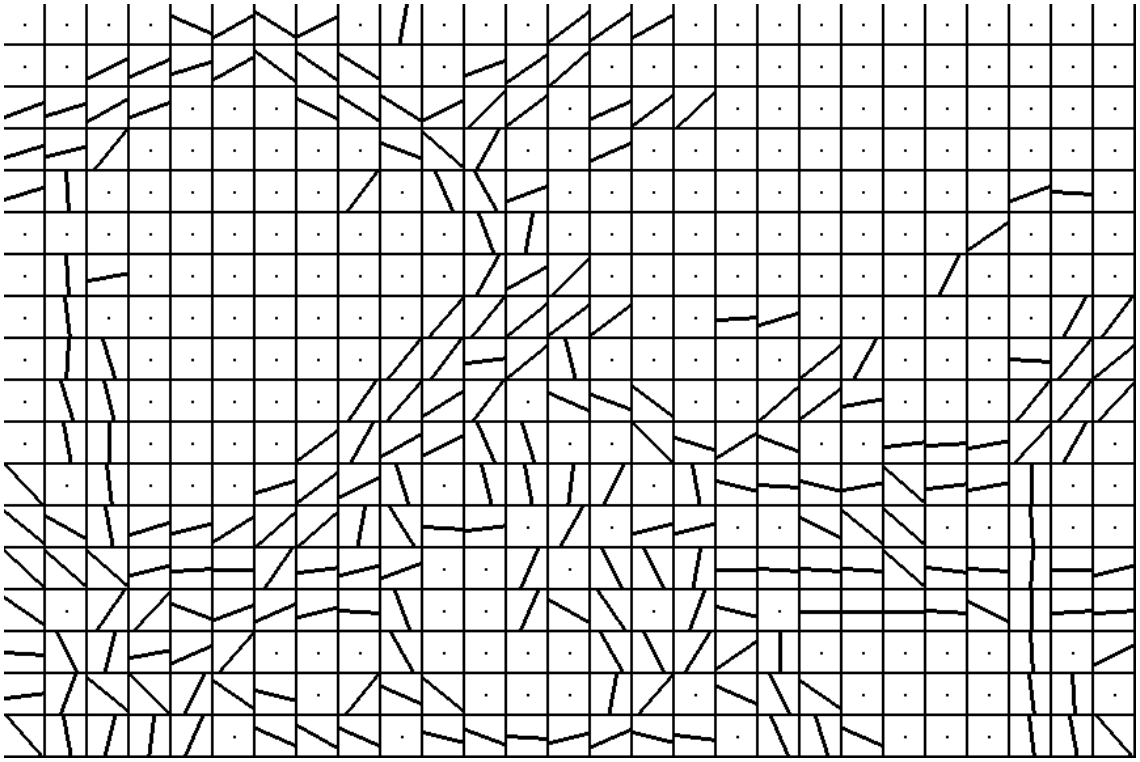

Figure 3: Dominant direction for each block of the image (a dot means no dominant direction).

Step 3: Boundary Pixels

Determine the pixel values at block boundaries (Fig. 4) that optimally match the image using the directions found in step 2. While directly using the pixels that lie on the block boundaries would work to some extent, we can generate a better image when also considering the pixels within the blocks.

When operating as an encoder, we need to code these pixels, but we can use intra prediction from other block boundaries to help save bits. This can be done by copying already coded pixels in the same line as the pixels being predicted – in a similar way to existing video codecs, except that only boundary pixels are predicted.

Step 4: Painting

For each block, use all four boundaries as well as the direction to paint the pixels inside the block (Fig. 6). The boundary pixels encoded in step 3 are not taken directly from the original image, but rather optimized to give the best prediction of the original image at this stage. Block discontinuities are avoided by blending within blocks based on the distance to each boundary. Each pixel is reconstructed from up to 4 boundary pixels as

$$y_p = \sum_{i=1}^{4} w_{p,i,d} \cdot x_{p,i,d} \ , \tag{5}$$

where $x_{p,i,d}$ are the 4 boundary pixel values used to predict pixel p for direction d and $w_{p,i,d}$ are the corresponding weights and sum to one. The weights are computed in such a way as to interpolate between adjacent boundary pixels, while also blending the prediction coming from each end of the prediction line. An example is provided in Fig. 5.

It turns out that the weights used in 5 are exactly the same as those we want to use in Step 3. In other words, for each pixel in the block in Step 3, we use the weights to do the accumulation on the corresponding boundary pixels. This in turn minimizes the distance between the original and the painted image

$$D = \sum_{p} (x_p - y_p)^2 \tag{6}$$

over the entire image.

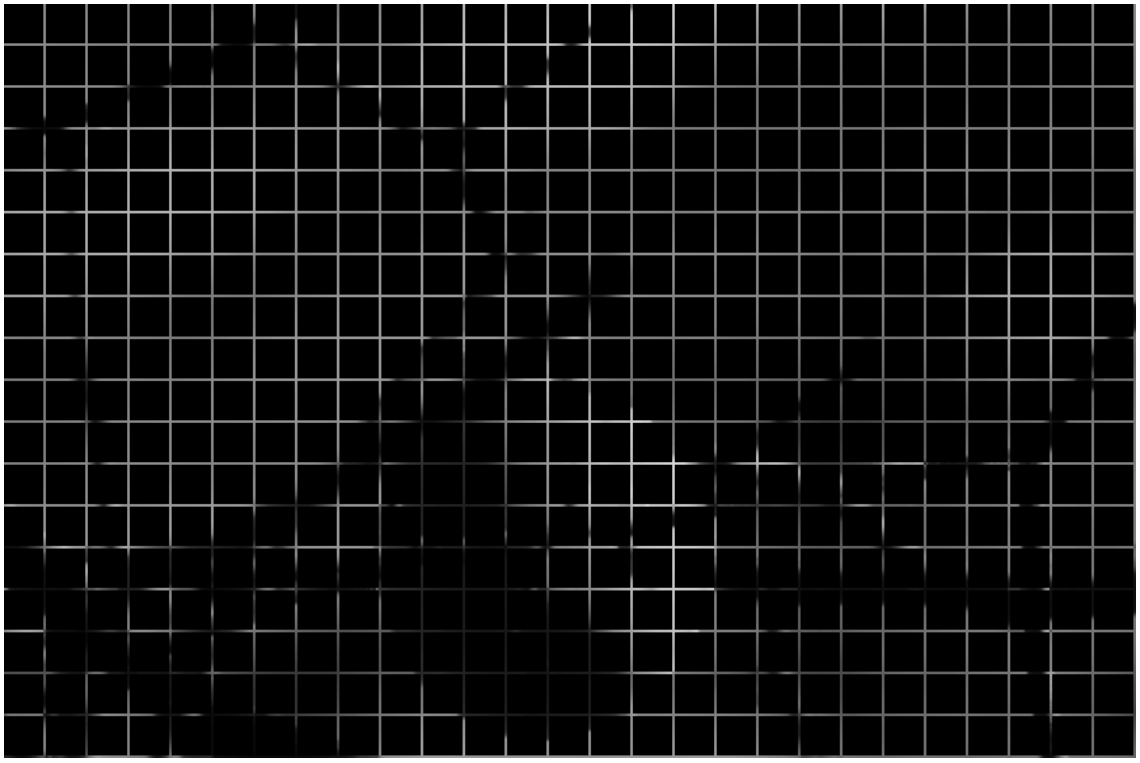

Figure 4: Optimal pixel values at block boundaries.

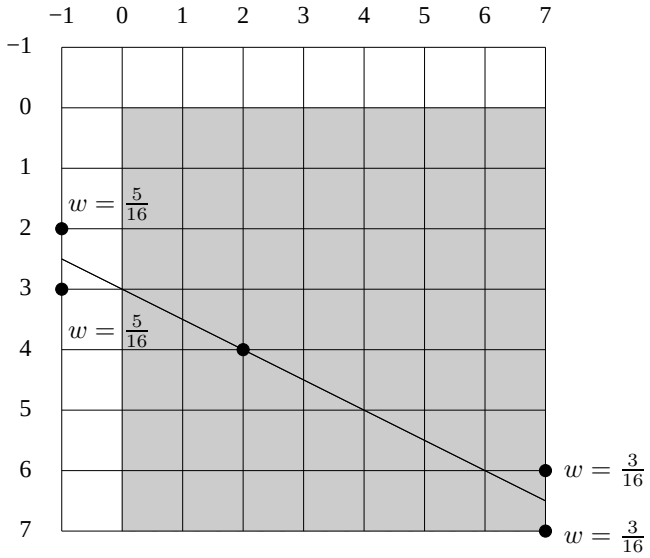

Figure 5: Interpolation of pixel (2,4) from pixels (-1,2), (-1,3), (7,6), and (7,7). The current block shown in gray.

Figure 6: Reconstructed "painted" image.

3 Deringing Filter

Intra paint can be used as a post-processing step to attenuate the coding artefacts. The idea is that there are regions of the image (e.g. close to edges) where the painted version looks better than the coded image, especially at low bitrate. Of course, it would be bad to replace the entire image with the painted version. We would only want to do use it in places where it improves quality. We also want to avoid spending too many bits on the painting process or on signaling which are the pixels that benefit.

For deringing purposes, instead of running the paint algorithm on the original image and sending information about the directions and edges, we can simply run the algorithm on the decoded image. The second part is trickier. How can we know which pixels should be replaced with the painted pixels and which ones are best unmodified? Intuitively, we see that in regions of the image that have clear directional patterns, the painted image should be much closer to the decoded image than in regions with unpredictable texture. Not only that, but knowing the quality at which the image was coded, we have an idea of the amount of quantization noise that was introduced, which is also the magnitude of the difference we can expect between decoded image and its painted version.

Let $\mathbf{x}_k$ be the decoded pixels on a line k and $\mathbf{y}_k$ be the painted version of the same line, the deringing output is given by

$$\mathbf{z}_k = (1 - g_k)\,\mathbf{x}_k + g_k \mathbf{y}_k \ , \tag{7}$$

where g_k is the filter gain. The gain that minimizes the expected mean squared error is given by

$$g_k = \min\left[1,\ \frac{\alpha Q^2}{12\sigma_k^2}\right] \tag{8}$$

where Q is the image quality (quantization step size), $\sigma_k^2 = \|\mathbf{x}_k - \mathbf{y}_k\|^2$ is the mean squared distance between decoded image and the painted image, and α is a tunable parameter between 0 and 1. Computing a weight along each line of a diven direction provides more control than computing the weight at the block level.

Instead of computing (7) and (8) separately on each line of each block, we can compute σ_k^2 on block boundary pixels in the same way as we computed the pixel averages in Step 3. This allows us to compute the partial gain

$$g_k^* = \frac{Q^2}{12\sigma_k^2} \tag{9}$$

on each boundary pixel. From there, we can treat the g_k^* values in the same way as pixels and apply the paint algorithm in Step 4 to fill the g_k^* values that are inside the blocks. This provides us with a continuously-varying gain across the image. The final gain then becomes

$$g_k = \min\left[1,\ \alpha g_k^*\right] \tag{10}$$

where the α parameter is the only information coded for the deringing filter. Typically, the encoder selects α by comparing the filter output to the original image and picking the α value that minimizes a certain cost metric (e.g. PSNR). To reduce the signaling overhead, α is typically signaled over a larger block size than the painting itself. For example, good results were obtained when signaling α only on 32x32 superblocks.

3.1 Simplifications

In some applications, applying the algorithm as described above may require too much CPU time. In that case, some simplifications are possible. It is possible to reduce complexity by skipping Step 3 of the paint algorithm and painting each block independently. Although this may increase boundary effects around blocks, it may still be acceptable for deringing purposes. Alternatively, the painting may be performed on overlapping blocks, which would reduce blocking artefacts at the cost of higher complexity.

It is also possible to reduce the complexity of the filter gain computation by only computing g on a block-by-block basis.

4 Implementation

A deringing filter based on the intra paint algorithm has been implemented for the Daala [1] codec. It is available from the Daala Git repository [2], in the `exp_paint_deringing4` branch. To maximize the resulting quality, the implementation does not include the simplifications described in Section 3.1.

5 Conclusion

We have demonstrated an effective algorithm to remove ringing artefacts from coded images and videos. While this algorithm does not significantly improve objective quality metrics we have been using on Daala (PSNR, PSNR-HVS, SSIM, FAST-SSIM [3]), the improvement in subjective quality is significant, mostly on edges. See this technilogy demo [4] for more information.

References

[1] Daala website, Xiph.Org Foundation. http://xiph.org/daala/

[2] Daala Git repository. http://git.xiph.org/?p=daala.git;a=summary

[3] T. Daede, J. Moffitt, *Video Codec Testing and Quality Measurement*, IETF Internet draft, 2015. https://tools.ietf.org/html/draft-daede-netvc-testing

[4] J.-M. Valin, *Daala: Painting Images for Fun (and Profit?)*. http://people.xiph.org/~jm/daala/paint_demo/