Implementation Details for "Predictable High-Quality Image Completion"

1 Image Completion Pseudocode

In this supplementary document we provide commented pseudocode for our image completion algorithm. It is more or less a straight-forward implementation of [Wexler et al. 2007] using PatchMatch [Barnes et al. 2009] for fast approximate nearest-neighbor search. Refer to Figure 1 for a list and explanations of symbols. In practice, the source image $\mathbf{s}$ and target image $\mathbf{t}$ are stored interleaved in a single image, however, we are using distinct symbols here for clarity. The FILLSMOOTH function is not included in the code below. It fills the missing region with a smooth membrane. We implemented it using iterated laplacian smoothing.

```
    // The outer loop
 1: procedure COMPLETION
 2:     for l ← l^max downto 0 do
 3:         if l = l^max then
                // Initialize nearest-neighbor field
 4:             for all (x_t, y_t) ∈ T^l do
 5:                 f^l(x_t, y_t) ← randomly choose from S^l
 6:             end for
                // Smoothly fill the target image
 7:             t^L ← FILLSMOOTH(l)
                // Do a few passes without updating the result image
 8:             PASS(l, false)
 9:         else
10:             UPSAMPLE(l)
11:         end if
12:         PASS(l, true)
13:     end for
14: end procedure

    // Synthesize current level.
15: function PASS(l, freeze-result)
16:     N_passes ← max(N_min, N_max − N_reduce · (l^max − l))
17:     repeat N_passes times
18:         for k ← 1 to N_iter do
19:             down-direction ← [k mod 2 = 0]
20:             ITERATE(l, down-direction)
21:         end for
            // No updates when initializing the coarsest level.
22:         if freeze-result = false then
23:             AVERAGE(l)
24:         end if
25:     end repeat
26: end function

    // Upsample nearest-neighborfield from coarse level.
27: function UPSAMPLE(l)
28:     for all (x_t, y_t) ∈ T^l do
            // Upsample coordinate from coarse level.
29:         (x_s, y_s) ← f^{l+1}(⌊x_t/2⌋, ⌊y_t/2⌋)·2+(x_t mod 2, y_t mod 2)
            // Assign upsampled coordinate only if it is valid.
30:         f^l(x_t, y_t) ← { (x_s, y_s), if (x_s, y_s) ∈ S^l
                            { randomly choose from S^l, else
31:     end for
32:     AVERAGE(l)
33: end function
```

x_s, x_s	pixel coordinates in source image
x_t, x_t	pixel coordinates in target image
l	level
$l^{\max}$	coarsest level
$\mathbf{f}^l$	nearest-neighbor field
$\mathbf{t}^l$	result
$\mathbf{s}^l$	source
$\mathbf{T}^l$	set of valid patch centers in target image
$\mathbf{S}^l$	set of valid patch centers in source image
$\mathbf{K}^l$	set of known pixel locations
$\mathbf{H}^l$	set of missing pixel locations
$\mathbf{R}^l(x, y)$	restricted search space of pixel (x, y)
$\mathbf{d}^l$	distance to closest known pixel
$P(x, y)$	7×7 neighborhood centered around (x, y)
$N_{\max} = 50$	number of passes at coarsest level
$N_{\text{reduce}} = 5$	reduction in number of passes per level
$N_{\min} = 2$	minimum number of passes per level
$N_{\text{iter}} = 4$	number of PatchMatch iterations per pass
w^l, h^l	image dimensions

Figure 1: *Symbols used throughout the paper and in the pseudocode*

```
    /* The overlapping patches are averaged together to generate
       an updated result image. */
34: function AVERAGE(l)
35:     for all (x_t, y_t) ∈ H^l do
36:         color ← (0, 0, 0), w_sum ← 0
            // Loop over all patches who overlap the current pixel
37:         for all (x, y) ∈ P(x_t, y_t) do
38:             (x_s, y_s) ← f^l(x, y) + (x_t − x, y_t − y)
                /* Give higher weight to pixels closer to the hole
                   boundary. γ = 2. */
39:             w ← γ^{d^l(x_t,y_t) − d^l(x,y)}
40:             color ← color + w · s^l(x_s, y_s)
41:             w_sum ← w_sum + w
42:         end for
43:         t^l(x_t, y_t) ← color / w_sum
44:     end for
45: end function

    // A PatchMatch iteration over the whole image
46: function ITERATE(l, down-direction)
        // left/up offset on down pass and right/down on up pass
47:     ofs ← { −1, if down-direction = true
              { +1, else
        // Use reverse scanline order if down-direction = false
48:     for all (x_t, y_t) ∈ T^l in normal/reverse scanline order do
49:         cur-dist ← PATCHDIST(l, x_t, y_t, f^l(x, y))
            // Horizontal propagation
50:         if (x_t + ofs, y_t) ∈ T^l then
51:             (x_new, y_new) ← f^l(x_t + ofs, y_t) − (ofs, 0)
52:             cur-dist ← TRYUPDATE(l, x_t, y_t, x_new, y_new, cur-dist)
53:         end if
```

```
        // Vertical propagation
        if (x_t, y_t + ofs) ∈ T^l then
            (x_new, y_new) ← f^l(x_t, y_t + ofs) − (0, ofs)
            cur-dist ← TRYUPDATE(l, x_t, y_t, x_new, y_new, cur-dist)
        end if
        // Window search
        rad ← max(w^l, h^l)
        while rad > 0 do
            // Try a random coordinate within square window
            (x_new, y_new) ← f^l(x, y) + random2D(−rad, rad)
            cur-dist ← TRYUPDATE(l, x_t, y_t, x_new, y_new, cur-dist)
            rad ← ⌈rad/2⌉
        end while
    end for
end function

/* Updates the nearest-neighbor field at (x_t, y_t) only if the new
   coordinate is valid and provides a lower patch distance. */
function TRYUPDATE(l, x_t, y_t, x_new, y_new, cur-dist)
    if (x_new, y_new) ∈ R^l(x_t, y_t) then
        new-dist ← PATCHDIST(l, x_t, y_t, x_new, y_new)
        if new-dist < cur-dist then
            f^l(x_t, y_t) = (x_new, y_new)
            return new-dist
        end if
    end if
    return cur-dist
end function

/* Computes the SAD of two 7×7 patches. */
function PATCHDIST(l, x_t, y_t, x_s, y_s)
    dist ← 0
    for all (x'_t, y'_t) ∈ P(x_t, y_t) do
        (x'_t, y'_t) = (x_s + x'_t − x_t, y_s + y'_t − y_t)
        dist ← dist + |t^l(x'_t, y'_t) − s^l(x'_s, y'_s)|
    end for
    return dist
end function
```

2 Features for Quality Prediction

2.1 Overview

Features for a pixel's quality prediction are computed as follows. First, for every 16x16 region in the image that contains known pixels, we pre-compute 19 statistics of basic visual features in the known area (edges, curves, lines, color, see subsections below for details). Then, given an unknown pixel, we find all its associated regions and compute a weighted average of the 19 statistics (using the number of known pixels in a region as its weight). Finally, we augment the pixel's feature vector by the its y-coordinate and Euclidian distance to the closest known pixel in the image.

2.2 Edge Features (1+3)

We compute a set of edge segments using steerable filters. Steerable filters are useful as they have the same response to step edges and ridges. We use a second order quadrature pair of filters (G2 and H2, as described in [Freeman and Adelson 1991]). These are the steps in computing an edge response: 1) We compute the dominant orientation using the analytical solution for the orientation energy (this only finds one dominant direction as we use the second order filters). 2) Then we use the computed orientation angle to steer the filters in that direction and get filter responses G2 and H2. 3) We then find a peak in the magnitude = (G2*G2 + H2*H2) by searching for a maximum in the direction perpendicular to the direction of the dominant orientation. 4) If we find a peak in step (3) that indicates the presence of a line or edge. 5) We do a quadratic fit on the magnitude to find the x,y position of the edge reponse up to sub-pixel accuracy. We derive four edge features from this data: the first one is a scalar describing the fraction of pixels that are edges. The other three are the mean, standard deviation, and entropy of the distribution of edge orientations (computed from a 180-bin histogram).

2.3 Curve and Line Features (3+3)

If there are more than eight edges in a region, we detect curves from edges by greedily growing edge elements along their orientation as long as there are neighboring edges with matching orientation. Then, we use the Ramer-Douglas-Peucker algorithm for line simplification [Douglas and Peucker 1973]: at each stage a curve is examined and split at the point that is the farthest from the line joining the end points of the curve. If the distance is less than a threshold then the simplification stops otherwise it recursively operates on the two sub curves split at the farthest point. The lengths of the resulting curve and line segments are binned into two 500-bin histograms, respectively. The mean, standard deviation, and entropy of the two length distributions serve as curve and line features.

2.4 Color Features (3+3+3)

For color statistics, we simply use mean, standard deviation, and entropy of 256-bin color histograms, one for each channel (R, G, B).

References

BARNES, C., SHECHTMAN, E., FINKELSTEIN, A., AND GOLDMAN, D. B. 2009. Patchmatch: a randomized correspondence algorithm for structural image editing. *ACM Trans. on Graphics (Proceedings of Siggraph) 28*, 24:1–24:11.

DOUGLAS, D., AND PEUCKER, T. 1973. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization 10*, 112–122.

FREEMAN, W. T., AND ADELSON, E. H. 1991. The design and use of steerable filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence 13*, 891–906.

WEXLER, Y., SHECHTMAN, E., AND IRANI, M. 2007. Space-time video completion. *TPAMI*.