Supplementary Material - Pattern-Based Cloth Registration and Sparse-View Animation

OSHRI HALIMI, Technion – Israel Institute of Technology, Israel and Meta Reality Labs, USA
TUUR STUYCK, Meta Reality Labs, USA
DONGLAI XIANG, Carnegie Mellon University, USA and Meta Reality Labs, USA
TIMUR BAGAUTDINOV, Meta Reality Labs, USA
HE WEN, Meta Reality Labs, USA
RON KIMMEL, Technion – Israel Institute of Technology, Israel
TAKAAKI SHIRATORI, Meta Reality Labs, USA
CHENGLEI WU, Meta Reality Labs, USA
YASER SHEIKH, Meta Reality Labs, USA
FABIAN PRADA, Meta Reality Labs, USA

ACM Reference Format:
Oshri Halimi, Tuur Stuyck, Donglai Xiang, Timur Bagautdinov, He Wen, Ron Kimmel, Takaaki Shiratori, Chenglei Wu, Yaser Sheikh, and Fabian Prada. 2022. Supplementary Material - Pattern-Based Cloth Registration and Sparse-View Animation. *ACM Trans. Graph.* 1, 1 (September 2022), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 ADDITIONAL REGISTRATION DATA

In addition to the shirt registration data, we captured dynamics of a skirt under varying legs motion and hand-cloth interaction. The registered mesh sequence contains 2k frames. The pattern comprises 123k cells, with a resolution of 2.6mm. We used 73 cameras for the triangulation. Figure 1 presents the skirt registrations.

2 PATTERN REGISTRATION

2.1 Image Pattern Detector

The image pattern detector has to address several challenges such as the non-rigid deformation, projective transformation invariance, low-resolution observations, variable illumination and blur. In addition, the detector must have reasonable computation time to allow processing of the multi-view sequence and support training from sparse annotations to reduce the annotation time. To this end we propose *PatterNet*, an image pattern detector designed with the above requirements in mind. *PatterNet* is composed of two separate networks *SquareLatticeNet* which detects the corners and centers and *ColorBitNet* classifies the pixel color. We elaborate on each in detail in the following sections.

Authors' addresses: Oshri Halimi, Technion – Israel Institute of Technology, Israel and Meta Reality Labs, USA, oshri.halimi@gmail.com; Tuur Stuyck, Meta Reality Labs, USA, tuur@fb.com; Donglai Xiang, Carnegie Mellon University, USA and Meta Reality Labs, USA, donglaix@cs.cmu.edu; Timur Bagautdinov, Meta Reality Labs, USA, timurb@fb.com; He Wen, Meta Reality Labs, USA, hewen@fb.com; Ron Kimmel, Technion – Israel Institute of Technology, Israel, ron@cs.technion.ac.il; Takaaki Shiratori, Meta Reality Labs, USA, tshiratori@fb.com; Chenglei Wu, Meta Reality Labs, USA, chenglei@fb.com; Yaser Sheikh, Meta Reality Labs, USA, yasers@fb.com; Fabian Prada, Meta Reality Labs, USA, fabianprada@fb.com.

0730-0301/2022/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2.1.1 Corner and Centers Detector. We detect the centers and corners in the square lattice by the network *SquareLatticeNet*. This network is a 3-way pixel classifier, operating on the gray version of the image and assigning the labels 1 and 2 to corners and centers, respectively, and 0 to background pixels. The architecture of SquareLatticeNet is based on UNet; see the supplementary for the structure. We trained the pixel classifier using sparse annotations, containing the centers and corners pixel location in every annotated patch. In addition, the annotation included a color label in the range 0-6 for every annotated center, see FIGURE. We trained the network with categorical cross-entropy (CCE) loss, measured between the network predictions over the image and a dense ground-truth signal we constructed by assigning the labels 1 and 2 to the annotated pixels, and 0 for the other pixels. To balance the inherent bias towards the background class, which occurs with much higher frequency than the keypoint annotations, we used another cross-entropy loss term evaluated exclusively in the annotated pixels and hence doesn't contain background labels in the ground-truth signal. We used the dense loss term and the sparse one with a fixed ratio 1:1 between their weights. Our training set consisted of 2000 annotated patches in varying sizes 32^2-150^2, cropped from our capture frames for a few selected cameras. The typical keypoint density was 0.02-0.1 keypoints per pixel. We augmented the training set by rotation, reflection, resizing, gaussian blur, and brightness variation for robustness. The inference time per 2668x4096 image was <1 sec on an A100 GPU. The predictions of SquareLatticeNet appear in FIGURE.

2.1.2 Color Detector. As explained in the paper, with the predicted square centers and corners pixel location, we reconstruct the graph structure of the pattern in the image domain in regions where the cloth pattern is visible. To determine the hash code of every 3×3 local grid subgraph, we need to detect the color labels at the graph nodes. To this end, we propose ColorBitNet, an 8-way pixel classifier, operating on the color image and assigning the labels 1-7 identifying

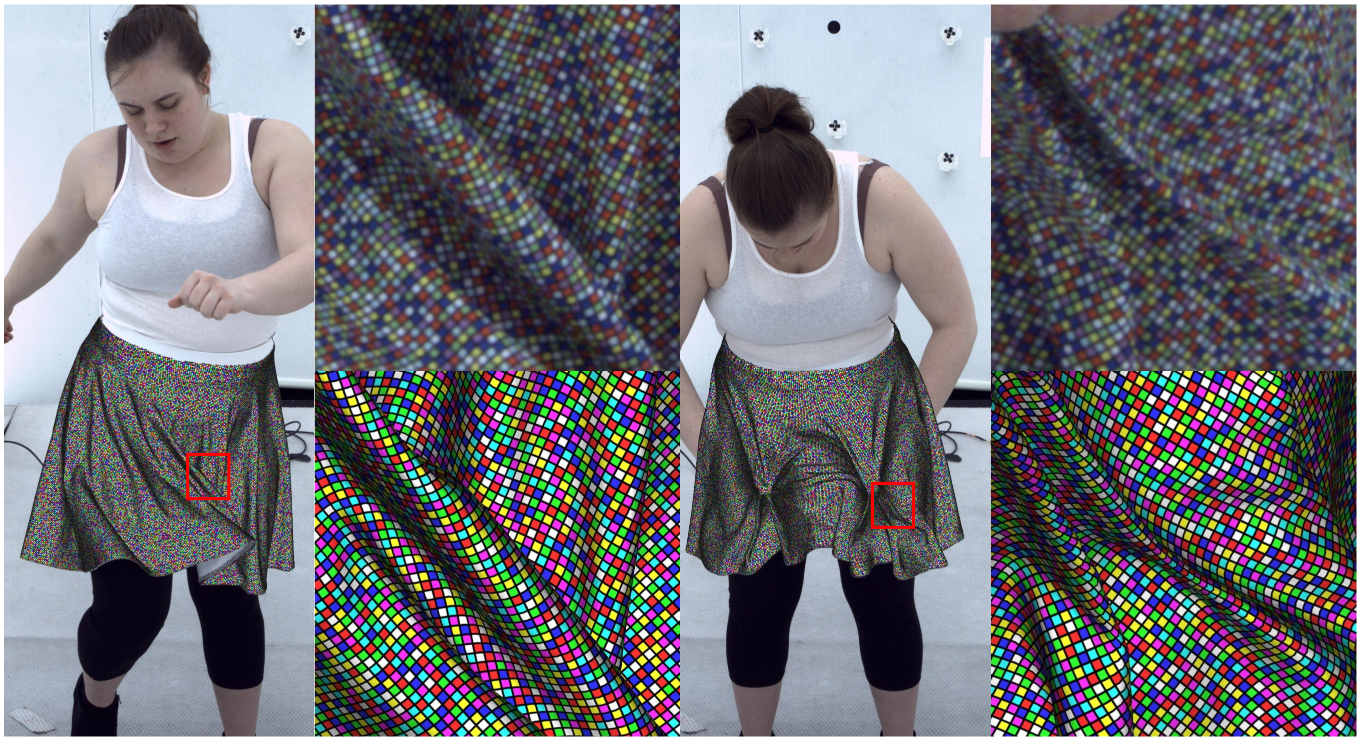

Fig. 1. We registered a dynamic skirt under diverse motion and hand-cloth interaction. We show two example frames, zooming on the original frame's wrinkles and rendered registration, respectively.

the different colors and 0 to background pixels, such as the square grid wireframe. In this case, we aimed for a piecewise constant color segmentation rather than a narrow pulse like in SquareLatticeNet, to accommodate correct color prediction even when the centers are sampled slightly around the exact location. Therefore, we couldn't assume that all the unannotated pixels belong to the background class, as we did in SquareLatticeNet. Instead, we had to train the network in a semi-supervised setting and regularize the predictions at the unconstrained pixels. Our solution is a UNet-based architecture with a single downsampling-upsampling operation while the other layers are purely convolutional to encourage color inference based on local image features. Indeed, while we didn't explicitly provide background annotations except at the corner pixels, the network's inductive bias led to the correct prediction of background labels along the square edges and piecewise constant prediction of the color labels outside the square centers. Specifically, we trained the network with categorical cross-entropy (CCE) loss, evaluated exclusively at the sparsely annotated pixels; The center pixels were labeled according to the color annotation 1-7, and the corners were labeled as background with the label 0. We used similar augmentation to SquareLatticeNet, except for the color variation, leaving the critical feature for classification uninterfered. ColorBitNet exhibited computation time similar to SquareLatticeNet, <1 sec per 2668x4096 image on an A100 GPU. The predictions of ColorBitNet appear in FIGURE X. ColorBitNet's network architecture can be found in the supplementary.

3 GRAPH PROCESSING ALGORITHMS

The heterogeneous graph To generate the heterogeneous graph, we apply Algorithm 1. As a final step, we handle the boundary detections. At the boundary keypoints, the number of the mutually opposite-type nearest neighbor is <4. Thus the neighbors cannot form a valid parallelogram around the detected keypoint. To complete the heterogeneous graph, we extend it with edges between mutually opposite type neighbors, updating the cyclically sorted neighbor list upon incidence with a node. We designate the newly added edges by E_{new}. Then, we calculate the set of cycles in the extended graph generated according to the following procedure: 1) Choose a starting node v_0, an edge exiting this node e_0, and an orientation - clockwise or anti-clockwise 2) Walk four steps on the graph, start along the initial edge and determine the next direction by turning from the last edge according to the chosen orientation. The result is a sequence of nodes and oriented edges $(v_0, e_0, v_1, e_1, v_2, e_2, v_3, e_3, v_4, e_4)$. Next, we validate the generated cycle. A cycle is said to be *valid* if the nodes (v_1, v_2, v_3, v_4) are four unique nodes AND $v_4 = v_0, e_4 = e_0$. Finally, if an edge $e \in E_{new}$ participates in a valid cycle, we keep it as a permanent edge in the heterogeneous graph. The results are shown in Figure **??**.

The homogeneous graph After the last stage, every pattern square potentially maps to four cyclically oriented edges around a center-type node in the heterogeneous graph. We apply the following scheme to reconstruct the grid graph in the visible image regions. For each center-type node, we iterate its corner-type neighbors in the heterogeneous graph, picking a pair of cyclically adjacent neighbors each time. We get each pair's mutual center neighbors and subtract the original center from the resulting set. Next, if the result is a singleton, we create a directed edge between the former center and the resulting center. When we finish iterating all the center-type nodes, we have a set of directed edges. Finally, we create an undirected edge for every pair of opposite edges found in the directed edge set. This final set of undirected edges defines the homogeneous graph's edges. The homogeneous graph's nodes are center-type nodes with at least two edges in the last edge set. Nodes with less than two edges don't participate in any 3×3 grid subgraph of the homogeneous graph, even as boundary nodes - when the subgraph lies on the complete graph boundary; therefore, we can safely ignore them.

The homogeneous graph attributes To allow the hash code extraction, we assign every node in the homogeneous graph a color attribute and a 3×3 matrix containing the vertex IDs participating in the local 3×3 grid graph around that node. For the color attribute, we query the precalculated heterogeneous graph with the corresponding center-type node ID to get its corner-type neighbors. We fetch all the non-background color labels in the parallelogram area defined by the corners and assign this color set as the color attribute for every node in the homogeneous graph. The following section explains how we handle potentially ambiguous codes resulting from a non-singleton color set. Next, try to assign a local 3x3 grid to every node with a graph degree equal to four. Nodes with a lesser degree cannot serve as the center of a 3x3 grid. Additionally, nodes containing a neighbor with a degree less than three also fall in the same category. Therefore, we set the grid attribute with an empty value for these nodes. To get the 3x3 vertex ID matrix in the 4-degree nodes, we define a local coordinate system, determined by an arbitrary edge that serves as the local positive x-direction, inducing the local +x, +y, -x, -y directions on the other edges, in a cyclical order. The neighboring vertices along those edges fill the (1, 0), (0, 1), (-1, 0), (0, -1) entries of the 3x3 matrix, respectively, while the central node ID fills the (0,0) entry. To complete the four remaining diagonal entries, we cyclically iterate the off-diagonal entry pairs, successively fetching the corresponding vertices' mutual neighbors and subtracting the central vertex from the resulting set. If the final set is a singleton, we set the vertex ID in the corresponding diagonal entry; otherwise, we assign an empty value for the grid attribute. When this process terminates, each node contains a 3×3 grid or an empty value as a grid attribute and a color attribute.

3.1 Hash Code inference

The hash code is calculated from the homogeneous graph can the color classification of its nodes via the *Neighbor-Voting* Algorithm 2

4 MULTI-VIEW SURFACE ALIGNMENT

4.1 Triangulation

The RANSAC triangulation is a two-step algorithm that takes camera parameters and pixel coordinates of a grid point in each camera as input, and produces the 3D world coordinate of the grid point. In

ALGORITHM 1: Heterogeneous graph generation

```
Input      : Set of nodes $V$
             Type attribute $T : V \longrightarrow [0, 1]$
             Image coordinates attribute $F : V \longrightarrow \mathbb{R}^2$
Output     : The undirected heterogeneous graph $G = (V, E)$,
             admitting $\forall (u, v) \in E : T(u) \neq T(v)$.
             Neighbor cyclic ordering $\forall v \in V : c(v) = (v_1, v_2, ...)$,
             where $(v_1, v_2, ...)$ is $v$'s neighbor sequence $N_G(v)$,
             ordered cyclically by the angle of the vector
             $F(v_i) - F(v) \in \mathbb{R}^2$.
Complexity : $O(V)$
// Note: We use a numerical threshold in practice to evaluate
geometric conditions, but we present them abstractly here for
conciseness.
initiallize cycNbr      // Represents the output function $c()$ - a
dictionary with keys = vertex IDs and values = cyclically
ordered opposite neighbors

initialize geomValid                          // array of size $|V|$
for vertex $v \in V$ do
    /* Get the 4 non-co-directed opposite type nearest
       neighbors, as induced by $F : V \longrightarrow \mathbb{R}^2$          */
    counter = 0
    i = 1
    $\mathcal{N} = \emptyset$
    while $counter < 4$ do
        $u \longleftarrow v$'s i'th nearest-neighbor in $\{u \mid T(u) \neq T(v)\}$
        i++
        if $\{w \in \mathcal{N} \mid P(u) - P(v) \upuparrows P(w) - P(v)\} \neq \emptyset$ then
            continue
        else
            $\mathcal{N} = \mathcal{N}^\frown(u)$
                                  // append $u$ at the end of the sequence
            counter++
        end
    end
    sort $\mathcal{N}$ by the vector's angle $\langle P(\mathcal{N}) - P(v), \hat{x} \rangle$ // the reference
     axis can be chosen arbitrarily
    if $P(\mathcal{N})$ is a geometrically valid parallelogram AND
     $\sum_{v_i \in \mathcal{N}} P(v_i) = v$ then
        geomValid[$v$] = True
        cycNbr[$v$] = $\mathcal{N}$
    else
        geomValid[$v$] = False
        cycNbr[$v$] = ()
    end
end
/* Connect the edges                                          */
$E = \emptyset$
for vertex $v \in V$ do
    for neighbor $u \in cycNbr[v]$ do
        if $geomValid[u]$ AND $v \in cycNbr[u]$ then
            $E = E \cup \{(u, v)\}$
        else
            Remove $u$ from cycNbr[$v$]
        end
    end
end
```

ALGORITHM 2: Neighbor-Voting Algorithm: Registering board location to the homogeneous graph nodes

```
Input      : The homogeneous graph $G = (V, E)$
Output     : board location over the vertices
             $B(V) = (B_v = (i_v, j_v))_{v \in V}$
Complexity : $O(V)$
initialize boardLocationOverVertices  // a dictionary with keys =
    vertex IDs and values = registered board location

for vertex $v \in V$ do
    votes = {}
    // visit the vertices containing $v$ in their $3 \times 3$ grid-graph
    for vertex $u \mid v \in Nbr_{3\times3}(u)$ do
        store all 9-color-bit codes of the $3 \times 3$ graph centered at $u$, in
          $C = \{c_0, c_1, \dots\}$
        for code $c$ in $C$ do
            result ← HashFunction($c$)
            if result is empty then
                continue
            end
            boardLocation, boardRotation ← result
            nbrVote ← getNbrVote(u, v, boardLocation,
              boardRotation)          // get $(i_v^{u,c}, j_v^{u,c})$ the board
              coordinates of $v$ according to the $(u, c)$ pair
            votes.Append(nbrVote)
        end
    end
    majorVote, majorVoteCount ← CalcMajorVote(votes)
    if majorVoteCount == 1 then
        boardLocationOverVertices[$v$] = NULL  // The vote count
          has the form [1, 1, ...], we reject the board
          registration
    else
        boardLocationOverVertices[$v$] = majorVote
    end
end
```

the RANSAC step, we first generate candidate 3D points by triangulating from 100 random pairs of cameras. Then, for each candidate we calculate the number of inliers by counting the number of cameras with less than 1mm point-to-ray distance, and pick the one with the greatest number of inliers as our best candidate. In the triangulation refinement step, we take all the inliers from our best candidate and use direct linear transformation (DLT) to produce an initial estimation. To further refine the result, we use an iterative procedure to reweigh each camera in the DLT until convergence to find the densest cluster and make the result more stable. More precisely, in each iteration we assign a weight to each camera that is inversely proportional to the exponential of the point-to-ray distance, and solve the DLT again.

4.2 Camera Setup Analysis

In Table 1 we report the average precision and coverage of pattern registration as a function of number of cameras on a collection of 200 representative frames. For this we use incremental sets of uniformly distributed cameras. We measure the registration accuracy as the root mean square distance between the triangulated vertices and the ground-truth vertices, which we assume are given by the registration obtained with 212 cameras. We observe that the registration error is stable at around 0.35 millimeters, even for a significantly smaller number of cameras than we used. The surface coverage (visualized in Figure 2) increases monotonously with the number of cameras. Interestingly, using only 12 cameras is enough to get over 70% coverage, while the improvement gained when changing from 100 cameras to 212 cameras is only 1.2% (!). For these results we use 2 views and a radius of 1mm as the parameters of the RANSAC algorithm.

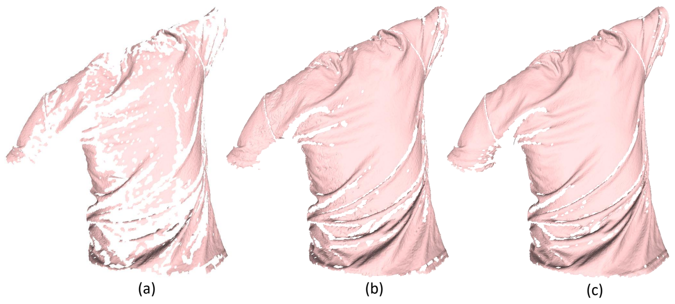

Fig. 2. Surface coverage from left to right for 6 cameras (43%), 25 cameras (76%) and 212 cameras (82%).

5 SPARSE-VIEW ANIMATION

5.1 Pixel-Driving Network Architecture

The pixel driving network consists of the following layers:

(1) Input convolution
(2) UNet
(3) Output convolution

The **Input convolution** processes the input features, which are the stacked pixel coordinates signals of each camera channel. The input shape is $P_t \in \mathbb{R}^{U \times V \times 2C}$, where $C = 2$ is the number of cameras, and U, V is the UV size. The input convolution performs a single convolution operation, producing an output of dimensions $F_t \in \mathbb{R}^{U \times V \times 64}$. The network uses a kernel size equal to 3, padding=1, applied in all 4 sides of the UV signal, untied bias, and a LeakyReLU activation function. The **UNet** consists of 6 downsampling operations, and 6 upsampling operations. Both downsampling and upsampling are performed with a convolution / transposed-convolution layer that uses untied bias, and operates with kernel size=4, stride=2, and padding=1, applied to each of the 4 sides of each UV signal. The layers use LeakyReLU activation function. In the downsampling direction, every convolution operation decreases the input by a factor of two, while the feature dimension increases by a factor of two. In the upsampling direction, every convolution operation increases the input by a factor of two, while the feature dimension decreases by a factor of two. The features in the upsampling path are merges with the features in the downsampling path by addition operation. The **Output convolution** finally we apply an output convolution. The output convolution performs a single convolution, producing an output with the dimenstions of $O_t \in \mathbb{R}^{U \times V \times 3}$. The network uses

Camera setup analysis		
Number of cameras	Registration accuracy Vertex RMSEe [mm]	Surface Coverage [%]
6	0.328	43.86
12	0.392	71.96
25	0.359	73.73
50	0.388	75.76
100	0.383	79.05
212	0.000 (Assumed "ground-truth")	80.25

Table 1. Influence of the number of cameras on surface coverage and registration accuracy.

a kernel size equal to 3, padding=1, applied in all 4 directions, untied bias, and a LeakyReLU activation function.

5.2 Kinematic Model Construction

Our kinematic model $\mathcal{K}$ is automatically constructed from template $\mathcal{M}$. It is a skeleton with leaf nodes at a subset of vertices of the template mesh $\mathcal{M}$. We compute leaf nodes by doing iterative farthest point insertion, providing a uniform coverage of the surface. We compute intermediate nodes of the skeleton using procedural clustering. For our experiments, we use a skeleton model that has 156 nodes, organized in 4 hierarchical levels with 1 (root), 5, 25, and 125 (leaves) nodes, respectively. We compute skinning weights for each vertex in $\mathcal{M}$ by normalizing geodesic distance to nearby leaf nodes.

5.3 Surface Augmentation

To make our driving network insensitive to the precise shape of the coarse geometry used for normalization, we augment the coarse surface produced by the kinematic model at training time. We deform the coarse geometry using a deformation space spanned by the Laplace-Beltrami eigenfunctions, using a randomized filter to select their linear combination. The deformation is defined as the displacement along the surface normal modulated by the random scalar function. We expect the coarse geometry to be projected close to the pixel detections. Therefore, the most significant ambiguity is likely to be along the local vector parallel to the surface normal. Since the kinematic model produces surfaces with coordinate functions of limited frequency, we used an exponential function as the spectral amplitude to decay the high-frequency modes of the displacement function. We used Bernoulli random variables for the filter coefficients to mix different modalities for each augmentation instance. Finally, we scale the resulting displacement function with a Gaussian random variable to allow displacements of varying magnitudes. More precise, our displacement function is:

$$D(x) = A \sum_{n=1}^{\infty} q_n e^{-\alpha n} \phi_n(x), x \in \mathcal{S} \tag{1}$$

where, ϕ_n are the corresponding Laplace-Beltrami functions on the coarse surface $\mathcal{S}$, $\alpha = 50$ is the filter decay rate in the spectral domain, $A \sim \mathcal{N}(0, \sigma = 20mm)$ is the random scale, and $q_n \sim$ $Bernoulli(p = 1/2)$ are the random coefficients selecting the different modes. To a good approximation, $\mathcal{M}$ is isometric to the coarse surface $\mathcal{S}$ and Laplace-Beltrami eigenfunctions are isometry invariant. Therefore, we calculated Laplace-Beltrami eigenfunctions once on the template mesh $\mathcal{M}$, and map them to the coarse geometry. At inference time, the coarse geometry is passed directly without deforming it.

6 PERFORMANCE

Our current effort targeted the highest driving quality but not real-time performance. More thorough engineering would help close the performance gap. Computation times of the two-camera driving system operating on two 4096x2668 images using a single GPU and 16 CPU cores are: 1)Pattern registration (4.2-4.4): 4 seconds. 2)Coarse geometry fitting (6.3.3): 30 seconds. 3)Pixel-driving network (6.2): 1 second. (2) uses gradient-descent iterations in PyTorch, expecting a significant acceleration converting to CUDA implementation and higher-order solvers or, alternatively, to a regression model. (1,3) will accelerate by converting from python to a compiled framework.