$P++ Gesture Recognizer Pseudocode

Radu-Daniel Vatavu
University Stefan cel Mare of Suceava
Suceava 720229, Romania
vatavu@eed.usv.ro

$P++ builds on top of the $P gesture recognizer[1], to which it adds (1) one-to-many point matchings and (2) processing of the gesture shape structure. In the following pseudocode, POINT is a structure that exposes x, y, and *strokeId* properties. *strokeId* is the stroke index a point belongs to (1, 2, ...) and is filled by counting pen down/up events.

Recognizer main function. Match *points* against a set of *templates* by employing the Nearest-Neighbor classification rule.

$P++RECOGNIZER (POINTS C, TEMPLATES *templates*)

```
n ← 32
NORMALIZE(C, n)
score ← ∞
for each T in templates do
    NORMALIZE(T, n) // should be pre-processed
    d ← min (CLOUD-DISTANCE(C, T), CLOUD-DISTANCE(T, C))
    if score > d then
        score ← d
        result ← T
return ⟨result, score⟩
```

Cloud matching function. Match two point clouds (*points* and *template*) by performing one-to-many alignments. Returns the minimum alignment cost.

CLOUD-DISTANCE (POINTS C, POINTS T, int n)

```
matched ← new bool[n]
sum ← 0
// match points from cloud C with points from T; 1-to-many matchings allowed
for i = 1 to n do
    min ← ∞
    for j = 1 to n do
        d ← POINT-DISTANCE(C_i, T_j)
        if d < min then
            min ← d
            index ← j
    matched[index] ← true
    sum ← sum + min
// match remaining points T with points from C; 1-to-many matchings allowed
for each j such that not matched[j] do
    min ← ∞
    for i = 1 to n do
        d ← POINT-DISTANCE(C_i, T_j)
        if d < min then min ← d
    sum ← sum + min
return sum
```

The following pseudocode addresses gesture preprocessing (or normalization) which includes resampling, scaling with shape preservation, and translation to origin.

Gesture normalization. Gesture points are resampled, scaled with shape preservation, and translated to origin. Normalized turning angles are computed.

Normalize (POINTS *points*, int n)

```
points ← RESAMPLE(points, n)
SCALE(points)
TRANSLATE-TO-ORIGIN(points, n)
COMPUTE-NORMALIZED-TURNING-ANGLES(points, n)
```

COMPUTE-NORMALIZED-TURNING-ANGLES (POINT C, int n)

```
C_{1.θ} ← 0, C_{n.θ} ← 0
for i = 2 to n − 1 do
    τ ← ((C_{i+1.x} − C_{i.x})·(C_{i.x} − C_{i−1.x}) + (C_{i+1.y} − C_{i.y})·(C_{i.y} − C_{i−1.y})) / (‖C_{i+1} − C_i‖·‖C_i − C_{i−1}‖)
    C_{i.θ} ← (1/π) arccos(τ)
return
```

Points resampling. Resample a *points* path into n evenly spaced points. We use $n = 32$.

RESAMPLE (POINTS *points*, int n)

```
I ← PATH-LENGTH(points) / (n − 1)
D ← 0
newPoints ← points_0
for each p_i in points such that i ≥ 1 do
    if p_i.strokeId == p_{i−1}.strokeId then
        d ← EUCLIDEAN-DISTANCE(p_{i−1}, p_i)
        if (D + d) ≥ I then
            q.x ← p_{i−1}.x + ((I − D)/d) · (p_i.x - p_{i−1}.x)
            q.y ← p_{i−1}.y + ((I − D)/d) · (p_i.y - p_{i−1}.y)
            q.strokeId ← p_i.strokeId
            APPEND(newPoints, q)
            INSERT(points, i, q) // q will be the next p_i
            D ← 0
        else D ← D + d
return newPoints
```

PATH-LENGTH (POINTS *points*)

```
d ← 0
for each p_i in points such that i ≥ 1 do
    if p_i.strokeId == p_{i−1}.strokeId then
        d ← d + EUCLIDEAN-DISTANCE(p_{i−1}, p_i)
return d
```

Points rescale. Rescale *points* with shape preservation so that the resulting bounding box will be $\subseteq [0..1] \times [0..1]$.

SCALE (POINTS *points*)

```
x_min ← ∞, x_max ← 0, y_min ← ∞, y_max ← 0
for each p in points do
    x_min ← MIN(x_min, p.x)
    y_min ← MIN(y_min, p.y)
    x_max ← MAX(x_max, p.x)
    y_max ← MAX(y_max, p.y)
scale ← MAX(x_max − x_min, y_max − y_min)
for each p in points do
    p ← ((p.x − x_min)/scale, (p.y − y_min)/scale, p.strokeId)
```

Points translate. Translate *points* to the origin $(0, 0)$.

TRANSLATE-TO-ORIGIN (POINTS *points*, int n)

```
c ← (0, 0) // will contain centroid
for each p in points do
    c ← (c.x + p.x, c.y + p.y)
c ← (c.x/n, c.y/n)
for each p in points do
    p ← (p.x - c.x, p.y - c.y, p.strokeId)
```

Point distance. Computes the distance between two points by considering their x, y coordinates, but also the turning angles θ.

POINT-DISTANCE (POINT a, POINT b)

```
return ((a.x − b.x)^2 + (a.y − b.y)^2 + (a.θ − b.θ)^2)^(1/2)
```

[1] http://dx.doi.org/10.1145/2388676.2388732

Project no. PN-II-RU-TE-2014-4-1187, financed by UEFISCDI, Romania.