1 Description of the Recognizers

1.1 Static Recognizers

Table 1 summarizes the invariance properties of the static recognizers.

Recognizer	Type of invariance			
	Position	Scale	Rotation	Direction
$P^3+	●	●	○	●
GPSDa	●	●	●	●

Table 1: Invariance properties of the static recognizers.

1.1.1 $P^3+

A 3D version of *$P+* algorithm [1]. It is position-, direction-, and scale-invariant [2], but not rotation-invariant. It is described into more details in Appendix 1.2.

1.1.2 GPSDa

A novel position-, scale-, direction-, and rotation-invariant recognizer for hand poses regardless of the position of the hand. Its rotation-invariance makes it well-suited to situations where hand poses should be recognized regardless of their direction (*e.g.*, rotate a picture by performing a "grab" static gesture and gradually rotating the hand).

1.2 Dynamic Recognizers

Table 2 compares the invariance properties of the dynamic recognizers.

Recognizer	Type of invariance			
	Position	Scale	Rotation	Direction
$P^3	●	●	○	●
$Q^3	●	●	○	●
$P^3+	●	●	○	●
$P^3+X	●	●	○	◐
3 cent	●	●	○	○
Jackknife	●	●	○	○
$F	●	●	○	●
FreeHandUni	●	●	○	●
Rubine3D	●	●	○	○
Rubine-Sheng	●	●	○	○

Table 2: Invariance properties of the dynamic recognizers.

1.2.1 $P^3

A generalization of $P [3] towards supporting 3D multi-stroke gestures which is similar to the $P3D recognizer implemented by[4]. However, unlike $P3D the recognizer $P^3 does not support 3D static poses and 2D dynamic gestures recognition. To keep memory usage and execution time low, gestures are represented as unordered sets of points, called *point clouds*. Gesture recognition happens in two phases: normalization and cloud-matching. The normalization process is similar to other $-family algorithms and is divided into three steps: (1) resample the gesture to a fixed number of equidistant points; (2) scale the gesture uniformly to keep its shape; (3) translate the centroid of the point cloud to the origin, *i.e.*, $(0, 0, 0)$. The cloud-matching phase matches a gesture's point cloud to the point cloud of each template by associating each point from the template to exactly one point from the gesture. It then computes the resulting distance, as the sum of Euclidean distances between all pairs of matching points, and returns the template that minimizes it. This recognizer is position-, direction-, and scale-invariant, but not rotation-invariant.

1.2.2 Q^3

A 3D variant of $Q [5], which achieved a 46X speedup on average over *$P* with no loss of accuracy. It brings two key changes to P^3: (1) early abandonment of templates, as soon as the distance exceeds the current shortest distance; (2) each point cloud has a 16×16×16 3D look-up table (LUT), where each location (x, y, z) refers to the closest point. These LUTs allow Q^3 to compute a lower bound of the distance between a gesture and a template in $\mathcal{O}(n)$. If it exceeds the current shortest distance, the template can be rejected without computing the exact distance. It has the same invariance properties as P^3.

1.2.3 P^3+

A more accurate version of P^3, adapted from Vatavu's $P+ [1]. It brings three key improvements to P^3: (1) each point from the template can now be matched to more than one point of the gesture; (2) the distance between a gesture and a template takes into account the connections between consecutive points (in the form of their turning angles); (3) early abandonment of templates. This recognizer has the same invariance properties as P^3.

1.2.4 P^3+X

A variant of P^3+, which supports partial direction-invariance by keeping track of conflicting templates (*i.e.*, templates that represent the same gesture but drawn in different directions). If a gesture matches a conflicting template, its direction is compared with the direction of each conflicting template and the closest one is returned.

1.2.5 3 cent

An optimized 3D port of Wobbrock *et al.*'s $1 [6] by Caputo *et al.* [7], which recognizes uni-stroke gestures in two phases: the gesture is first normalized (similarly to *$P*) and its distance to each template is then computed as the sum of the squared distances of corresponding points. The template with the shortest distance is returned. 3¢ is position- and scale-invariant, but neither rotation- nor direction-invariant.

1.2.6 Jackknife

A general-purpose 3D gesture recognizer [8] that supports any number of articulations. Unlike most $-family recognizers, it represents gestures as time-series. It uses the nearest neighbor approach, where it compares a gesture to each template and returns the closest one. It uses *Dynamic Time Warping* as a distance measure but applies a series of correction factors to take into account differences in gesture scale and span. As a result, this recognizer is both position- and scale-invariant, but neither direction- nor rotation-invariant.

1.2.7 $F

A new recognizer that adds $P+'s flexible cloud matching [1] to P^3. As for most $-recognizers, the points of the candidate and the templates are resampled to equidistantly-spaced points, scaled within a unit box (isometric [9]), and translated so their centroid is at the origin $(0, 0, 0)$. The template with the lowest dissimilarity score is considered as the best matching template for the candidate. As opposed to P^3, $F's cloud matching is flexible, as points can be matched to more than one point. It consists of matching the points from the first cloud with their closest point from the second cloud, then matching the points from the second cloud that have not been matched yet with their closest point in the first cloud.

1.2.8 FreeHandUni

A recognizer derived from Vatavu *et al.*'s Free-Hand recognizer [10], which extends $P [3] to support 3D hand gestures. It replaces the hand pose structure with a 3D point structure (x, y, z). With this modification, FreehandUni improves P^3 using a flexible cloud matching based on a one-to-many alignment between points [1]. The pre-processing stays the same but the matching process is more flexible: each point of the template cloud is matched with the closest point from the candidate cloud, then the remaining points from the candidate cloud are matched with the closest point from the template

cloud. It returns the gesture class with the lowest dissimilarity score. FreeHandUni is different from $F in that the early abandoning is not implemented, to align the computational complexity to $P^3.

1.2.9 Rubine3D

Inspired by the iGesture framework [11], Rubine3D is a feature-based recognizer that combines a set of three individual 2D Rubine recognizers [12], one for each plane XY, YZ, and ZX. Before it can recognize 3D trajectories, Rubine3D pre-processes the training templates to compute weights for each feature of each gesture class. Then, it can recognize gestures in four steps: (1) it pre-processes the gesture by scaling and filtering its points; (2) it projects it onto each plane and extracts a feature vector $(f_1, .., f_{13})$ from each of the three projections; (3) for each projection, it selects the gesture class that maximizes the similarity score (computed by combining the weights of the gesture class with the feature vector of the gesture); (4) it determines the resulting class by merging the results from the three projections.

1.2.10 Rubine-Sheng

Rubine-Sheng (RS) is a 3D recognizer inspired by Rubine. It supports 3D gestures by adding three new features proposed in the AdaBoost recognizer [13] to Rubine's existing 13 features. Aside from this difference, it is extremely similar to its 2D counterpart.

Note The implemented multipath recognizers include an optimization which avoids comparing the candidate gesture performed with one hand against the gestures performed with both hands in the training set.

2 Pseudocode of the Recognizers

2.1 P^3 Recognizer

P^3 is based on Vatavu *et al.*'s $P[3]. The modifications to the original algorithm are highlighted in blue. POINT is a structure with the `x`, `y`, `z`, and `strokeId` properties. POINTS is a list of POINT and TEMPLATES is a list of POINTS with their associated gesture class data (*e.g.*, the name of the gesture). Please note that the templates should be pre-processed to improve performance, but this is not shown in the pseudocode in order to keep it concise.

P^3RECOGNIZER(POINTS `pts`, TEMPLATES `templates`)

```
1: n ← 32                                                    ▷ Number of points
2: NORMALIZE(PTS, N)
3: score ← ∞
4: for each template in templates do
5:     NORMALIZE(TEMPLATE, N)                                ▷ Should be pre-processed
6:     d ← GREEDYCLOUDMATCH(pts, template, n)
7:     if d < score then
8:         score ← d
9:         result ← template
10: return template
```

GREEDYCLOUDMATCH(POINTS `ptsA`, POINTS `ptsB`, int `n`)

```
1: e ← .50
2: step ← ⌊n^(1−e)⌋
3: min ← ∞
4: for i = 0 to n − 1 step step do
5:     d_1 ← CLOUDDISTANCE(ptsA, ptsB, n, i)
6:     d_2 ← CLOUDDISTANCE(ptsB, ptsA, n, i)
7:     min ← MIN(min, d_1, d_2)
8: return min
```

CLOUDDISTANCE(POINTS `ptsA`, POINTS `ptsB`, int `n`, int `start`)

```
1: matched ← new bool [n]
2: sum ← 0
3: i ← start
4: repeat
5:     min ← ∞
6:     for each j such that not matched[j] do
7:         d ← EUCLIDEANDISTANCE(ptsA[i], ptsB[j])
8:         if d < min then
9:             min ← d
10:            index ← j
11:    matched[index] ← true
12:    weight ← 1 − ((i − start + n)%n)/n
13:    sum ← sum + weight ∗ min
14:    i ← (i + 1)%n
15: until i == start
16: return sum
```

NORMALIZE(POINTS `pts`, int `n`)

```
1: pts ← RESAMPLE(pts, n)
2: SCALE(pts)
3: TRANSLATETOORIGIN(pts, n)
```

RESAMPLE(POINTS pts, int n)

```
I ← PATHLENGTH(pts)/(n − 1)
D ← 0
resampledPts ← {pts[0]}
for each p_i in pts such that i ≥ 1 do
    if p_i.strokeId == p_{i−1}.strokeId then
        d ← EUCLIDEANDISTANCE(p_i, p_{i−1})
        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.z ← p_{i−1}.z + ((I − D)/d) ∗ (p_i.z − p_{i−1}.z)
            APPEND(resampledPts, q)
            INSERT(pts, i, q)                          ▷ q will be the next p_i
            D ← 0
        else
            D ← D + d
while resampledPts.length ≤ n do
    APPEND(resampledPts, pts[pts.length - 1])
return resampledPts
```

PATHLENGTH(POINTS pts)

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

SCALE(POINTS pts)

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

TRANSLATETOORIGIN(POINTS pts, int n)

```
c ← (0, 0, 0)
for each p in pts do                                   ▷ Compute the centroid
    c ← (c.x + p.x, c.y + p.y, c.z + p.z)
c ← (c.x/n, c.y/n, c.z/n)
for each p in pts do                                   ▷ Translate each point
    p ← (p.x − c.x, p.y − c.y, p.z − c.z, p.strokeId)
```

2.2 $P^3+ Recognizer

$P^3+ is an improved version of $P^3 based on Vatavu *et al.*'s $P+[1]. We only show the parts that were updated from $P^3. The modifications to the original algorithm ($P+) are highlighted in blue.

$P^3+RECOGNIZER(POINTS pts, TEMPLATES templates)

```
1: n ← 32                                                          ▷ Number of points
2: NORMALIZE(PTS, N)
3: score ← ∞
4: for each template in templates do
5:     NORMALIZE(TEMPLATE, N)                                      ▷ Should be pre-processed
6:     d ← MIN(CLOUDDISTANCE(pts, template, n, score),
7:             CLOUDDISTANCE(template, pts, n, score))
8:     if d < score then
9:         score ← d
10:        result ← template
11: return template
```

CLOUDDISTANCE(POINTS ptsA, POINTS ptsB, int n, float minSoFar)

```
1: matched ← new bool [n]
2: sum ← 0
3: for i = 0 to n − 1 do        ▷ Match points from cloud ptsA with points from ptsB; one-to-many matchings allowed
4:     min ← ∞
5:     for j = 0 to n − 1 do
6:         d ← POINTDISTANCE(ptsA[i], ptsB[j])
7:         if d < min then
8:             min ← d
9:             index ← j
10:    matched[index] ← true
11:    sum ← sum + min
12:    if sum ≥ minSoFar then
13:        return sum
14:    for each j such that not matched[j] do     ▷ Match remaining points from cloud ptsB with points from ptsA; one-to-many matchings allowed
15:        min ← ∞
16:        for i = 0 to n − 1 do
17:            d ← POINTDISTANCE(ptsA[i], ptsB[j])
18:            if d < min then
19:                min ← d
20:        sum ← sum + min
21:        if sum ≥ minSoFar then
22:            return sum
23:        return sum
```

POINTDISTANCE(POINT a, POINT b)

```
1: return √((a.x − b.x)² + (a.y − b.y)² + (a.z − b.z)² + (a.θ − b.θ)²)
```

NORMALIZE(POINTS pts, int n)

```
1: pts ← RESAMPLE(pts, n)
2: SCALE(pts)
3: TRANSLATETOORIGIN(pts, n)
4: COMPUTENORMALIZEDTURNINGANGLES(pts, n)
```

ComputeNormalizedTurningAngles(Points pts, int n)

1: $\texttt{pts}[0].\theta \leftarrow 0$
2: $\texttt{pts}[n].\theta \leftarrow 0$
3: **for** $\texttt{i} = 2$ **to** $\texttt{n} - 2$ **do**
4: $\quad \texttt{dpX} \leftarrow (\texttt{pts}[i+1].x - \texttt{pts}[i].x) * (\texttt{pts}[i].x - \texttt{pts}[i-1].x)$
5: $\quad \texttt{dpY} \leftarrow (\texttt{pts}[i+1].y - \texttt{pts}[i].y) * (\texttt{pts}[i].y - \texttt{pts}[i-1].y)$
6: $\quad \texttt{dpZ} \leftarrow (\texttt{pts}[i+1].z - \texttt{pts}[i].z) * (\texttt{pts}[i].z - \texttt{pts}[i-1].z)$
7: $\quad \texttt{pts}[i].\theta \leftarrow \frac{1}{\pi} \arccos\left(\frac{\texttt{dpX}+\texttt{dpY}+\texttt{dpZ}}{\|\texttt{pts}[i+1]-\texttt{pts}[i]\| * \|\texttt{pts}[i]-\texttt{pts}[i-1]\|}\right)$

2.3 $F Recognizer

$F is an improved version of $P^3 with the flexible cloud matching of $P+ [1]. We only show the parts of $F that were updated. The modifications to the original algorithms ($P^3 and $P+) are highlighted in blue.

$FRecognizer(Points pts, Templates templates)

```
1: n ← 32                                                          ▷ Number of points
2: Normalize(pts, n)
3: score ← ∞
4: for each template in templates do
5:     Normalize(template, n)                                      ▷ Should be pre-processed
6:     d ← Min(CloudDistance(pts, template, n, score),
7:             CloudDistance(template, pts, n, score))
8:     if d < score then
9:         score ← d
10:        result ← template
11: return template
```

CloudDistance(Points ptsA, Points ptsB, int n, float minSoFar)

```
1: matched ← new bool [n]
2: sum ← 0
3: for i = 0 to n − 1 do        ▷ Match points from cloud ptsA with points from ptsB; one-to-many matchings allowed
4:     min ← ∞
5:     for j = 0 to n − 1 do
6:         d ← EuclideanDistance(ptsA[i], ptsB[j])
7:         if d < min then
8:             min ← d
9:             index ← j
10:    matched[index] ← true
11:    sum ← sum + min
12:    if sum ≥ minSoFar then
13:        return sum
14: for each j such that not matched[j] do ▷ Match remaining points from cloud ptsB with points from ptsA; one-to-many matchings allowed
15:    min ← ∞
16:    for i = 0 to n − 1 do
17:        d ← EuclideanDistance(ptsA[i], ptsB[j])
18:        if d < min then
19:            min ← d
20:    sum ← sum + min
21:    if sum ≥ minSoFar then
22:        return sum
23: return sum
```

2.4 $P^3+X Recognizer

$P^3+X is based on Vatavu *et al.*'s $P3+[1]. We only show the parts that were modified from $P^3.

$P^3+XRecognizer(Points pts, Templates templates)

```
1: n ← 32                                                          ▷ Number of points
2: conflictThreshold ← 1.25          ▷ Maximum score to consider the template as conflicting
3: loadedTemplates ← {templates[0]}
4: for each template in templates do                              ▷ Should be pre-processed
5:     ⟨template2, score⟩ ← RecognizeHelper(template, loadedTemplates, n)
6:     if (template2.name ≠ template.name) and (score < conflictThreshold) then
7:         conflicts[template] ← template2
8:         conflicts[template2] ← template
9:     Append(loadedTemplates, template)
10: template ← RecognizeHelper(pts, templates, n)
11: if template ∈ conflicts then          ▷ If two templates are conflicting, compare their direction
12:     s1 ← DirectionalSimilarity(pts, template, n)
13:     s2 ← DirectionalSimilarity(pts, conflicts[template], n)
14:     if s2 > s1 then
15:         template ← conflicts[template]
16: return template
```

RecognizeHelper(Points pts, Templates templates, int n)

```
1: Normalize(pts, n)
2: score ← ∞
3: for each template in templates do
4:     d ← Min(CloudDistance(pts, template, n, score),
5:             CloudDistance(template, pts, n, score))
6:     if d < score then
7:         score ← d
8:         result ← template
9: return ⟨template, score⟩
```

DirectionalSimilarity(Points ptsA, Point ptsB, int n)

```
1: padding ← 2
2: dist ← 0
3: for i = padding to n − (2 + padding) do
4:     dir_{A,x} ← ptsA[i+1].x − ptsA[i].x
5:     dir_{A,y} ← ptsA[i+1].y − ptsA[i].y
6:     dir_{A,z} ← ptsA[i+1].z − ptsA[i].z
7:     dir_{B,x} ← ptsB[i+1].x − ptsB[i].x
8:     dir_{B,y} ← ptsB[i+1].y − ptsB[i].y
9:     dir_{B,z} ← ptsB[i+1].z − ptsB[i].z
10:    dist ← dist + dir_{A,x} * dir_{B,x} + dir_{A,y} * dir_{B,y} + dir_{A,z} * dir_{B,z}
11: return dist
```

2.5 Q^3 Recognizer

Q^3 is an improved version of P^3 based on Vatavu *et al.*'s $Q[5]. We only show the parts that were updated from P^3. The modifications to the original algorithm ($Q) are highlighted in blue.

$Q^3Recognizer(Points pts, Templates templates)

```
1: n ← 32                                                    ▷ Number of points
2: m ← 16                                                    ▷ Size of the LUT
3: Normalize(pts, n, m)
4: score ← ∞
5: for each template in templates do
6:     Normalize(template, n, m)                             ▷ Should be pre-processed
7:     d ← CloudMatch(pts, template, n, score)
8:     if d < score then
9:         score ← d
10:        result ← template
11: return template
```

CloudMatch(Points pts, Points template, int n, int min)

```
1: e ← .50
2: step ← ⌊n^(1−e)⌋
3: LB1 ← ComputeLowerBound(pts, template, step, template.LUT)
4: LB2 ← ComputeLowerBound(template, pts, step, pts.LUT)
5: min ← ∞
6: for i = 0 to n − 1 step step do
7:     if LB1[i/step] < min then
8:         min ← Min(min, CloudDistance(pts, template, n, i, min))
9:     if LB2[i/step] < min then
10:        min ← Min(min, CloudDistance(template, ps, n, i, min))
11: return min
```

CloudDistance(Points pts, Points template, int n, int start, float minSoFar)

```
1: unmatched ← {0, 1, 2, ..., n − 1}
2: i ← start
3: weight ← n
4: sum ← 0
5: repeat
6:     min ← ∞
7:     for each j in unmatched do
8:         d ← SqrEuclideanDistance(pts[i], template[j])
9:         if d < min then
10:            min ← d
11:            index ← j
12:    Remove(unmatched, index)
13:    sum ← sum + weight ∗ min
14:    if sum ≥ minSoFar then
15:        return sum
16:    weight ← weight − 1
17:    i ← (i + 1)%n
18: until i == start
19: return sum
```

ComputeLowerBound(Points pts, Points template, int step, int[] LUT)

```
1: LB ← new float[n/step + 1]                    ▷ Multiple lower bounds, one for each starting point
2: SAT ← new float[n]                            ▷ Summed area table for fast computations
3: LB[0] ← 0
4: for i = 0 to n − 1 do                         ▷ First, compute the lower bound for starting point index 0
5:     index ← LUT [pts[i].x, pts[i].y, pts[i].z]
6:     d ← SqrEuclideanDistance(pts[i], template[index])
7:     if i == 0 then
8:         SAT[i] ← d
9:     else
10:        SAT[i] ← SAT[i − 1] + d
11:    LB[0] ← LB[0] + (n − i) ∗ d
12: for i ← step to n − 1 step step do            ▷ Compute the lower bound for the other starting points
13:    LB[i/step] ← LB[0] + i ∗ SAT[n − 1] − n ∗ SAT[i − 1]
14: return LB
```

Normalize(Points pts, int n, int m)

```
1: pts ← Resample(pts, n)
2: Scale(pts)
3: TranslateToOrigin(pts, n)
4: LUT ← ComputeLUT(m, pts, n)
```

Scale(Points pts, int m)

```
1: x_min ← ∞, x_max ← −∞, y_min ← ∞, y_max ← −∞, z_min ← ∞, z_max ← −∞
2: for each p in pts do
3:     x_min ← Min(x_min, p.x)
4:     y_min ← Min(y_min, p.y)
5:     z_min ← Min(z_min, p.z)
6:     x_max ← Max(x_max, p.x)
7:     y_max ← Max(y_max, p.y)
8:     z_max ← Max(z_max, p.z)
9: scale ← Max(x_max − x_min, y_max − y_min, z_max − z_min)/(m − 1)
10: for each p in pts do
11:    p ← ((p.x − x_min)/scale, (p.y − y_min)/scale, (p.z − z_min)/scale, p.strokeId)
```

ComputeLUT(Points pts, int n, int m)

```
1: LUT ← new int[m, m, m]
2: for x = 0 to m − 1 do
3:     for y = 0 to m − 1 do
4:         for z = 0 to m − 1 do
5:             min ← ∞
6:             for i = 0 to n − 1 do
7:                 d ← SqrEuclideanDistance(pts[i], newPoint(x, y, z))
8:                 if d < min then
9:                     min ← d
10:                    index ← i
11:            LUT[x, y, z] ← index
12: return LUT
```

SqrEuclideanDistance(Point a, Point b)

```
1: return (a.x − b.x)^2 + (a.y − b.y)^2 + (a.z − b.z)^2
```

References

[1] Radu-Daniel Vatavu. Improving gesture recognition accuracy on touch screens for users with low vision. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 4667–4679, New York, NY, USA, 2017. ACM.

[2] Gordon Kurtenbach, George Fitzmaurice, Thomas Baudel, and Bill Buxton. The Design of a GUI Paradigm Based on Tablets, Two-hands, and Transparency. In *Proceedings of the ACM International Conference on Human Factors in Computing Systems*, CHI '97, pages 35–42, New York, NY, USA, 1997. ACM.

[3] Radu-Daniel Vatavu, Lisa Anthony, and Jacob O. Wobbrock. Gestures as point clouds: A $p recognizer for user interface prototypes. In *Proceedings of the 14th ACM International Conference on Multimodal Interaction*, ICMI '12, pages 273–280, New York, NY, USA, 2012. ACM.

[4] Harisson Cook, Quang Vinh Nguyen, Simeone Simoff, and Mao Lin Huang. Enabling gesture interaction with 3D point cloud. In *Proceedings of the 24th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, volume 2602, page 59–68. Computer Science Research Notes CSRN, June 2016.

[5] Radu-Daniel Vatavu, Lisa Anthony, and Jacob O. Wobbrock. $q: A super-quick, articulation-invariant stroke-gesture recognizer for low-resource devices. In *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services*, MobileHCI '18, pages 23:1–23:12, New York, NY, USA, 2018. ACM.

[6] Jacob O. Wobbrock, Andrew D. Wilson, and Yang Li. Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes. In *Proceedings of the 20th Annual ACM Symposium on User Interface Software and Technology*, UIST '07, pages 159–168, New York, NY, USA, 2007. ACM.

[7] F. M. Caputo, P. Prebianca, A. Carcangiu, L. D. Spano, and A. Giachetti. A 3 Cent Recognizer: Simple and Effective Retrieval and Classification of Mid-Air Gestures from Single 3D Traces. In *Proceedings of the Conference on Smart Tools and Applications in Computer Graphics*, STAG '17, page 9–15, Goslar, DEU, 2017. Eurographics Association.

[8] Eugene M. Taranta II, Amirreza Samiei, Mehran Maghoumi, Pooya Khaloo, Corey R. Pittman, and Joseph J. LaViola Jr. Jackknife: A reliable recognizer with few samples and many modalities. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, pages 5850–5861, New York, NY, USA, 2017. ACM.

[9] Jean Vanderdonckt, Paolo Roselli, and Jorge Luis Pérez-Medina. !!ftl, an articulation-invariant stroke gesture recognizer with controllable position, scale, and rotation invariances. In *Proceedings of the 20th ACM International Conference on Multimodal Interaction*, ICMI '18, page 125–134, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Elena-Gina Craciun, Ionela Rusu, and Radu-Daniel Vatavu. Free-Hand Gesture Recognizer Pseudocode. `http://www.eed.usv.ro/mintviz/projects/GIVISIMP/data/Pseudocode2.pdf`, 2016. [Online; accessed 09-August-2020].

[11] Beat Signer, U. Kurmann, and Moira C. Norrie. igesture: A general gesture recognition framework. In *9th International Conference on Document Analysis and Recognition (ICDAR 2007), 23-26 September, Curitiba, Paraná, Brazil*, pages 954–958. IEEE Computer Society, 2007.

[12] Dean Rubine. Specifying gestures by example. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '91, pages 329–337, New York, NY, USA, 1991. ACM.

[13] Jia Sheng. A study of adaboost in 3D gesture recognition. technical report CSC2515, Department of Computer Science, University of Toronto, 2004.