Automatic Reconstruction of Broken 3-D Surface Objects

Göktürk Üçoluk İ. Hakki Toroslu

Dept. of Computer Engineering
Middle East Technical University, Ankara

ucoluk@ceng.metu.edu.tr
toroslu@ceng.metu.edu.tr

Abstract

The problem of reconstruction of broken surface objects embedded in 3-D space is handled. A coordinate independent representation for the crack curves is developed. A new robust matching algorithm is proposed which serves for finding matching pieces even when some brittle pieces are missing. A prototype system having an X based GUI has been developed. This system generates artifical wire-frame data of broken pieces (with some noise) for a pot-shaped 3-D object and then recombines it using the proposed algorithms.

1 Introduction

The handled problem appears heavily in field archeology where reconstruction of hollow objects becomes a tedious and laborious task. It is the problem of jigsaw puzzle assembling of 3-D surfaces with no texture or color hints provided.

Previous work of [1, 2, 3] and the work of Wolfson [4] attack the 2-D problem and propose appropriate matching algorithms. Although Wolfson's algorithm is not the most efficient ($\mathcal{O}(n \log n + \epsilon n)$) it is especially well designed to deal with noise. In his work, 2-D objects are represented by *shape signatures* that are strings which are obtained by polygonal approximation of the boundary curve. Freeman [5] describes 2-D shapes by a set of *critical points* (like discontinuities in curvature) and computes features between consecutive critical points. This method is weak in treating curves that do not possess such points. Ayache and Faugeras [6] attack a more difficult problem where rotation, translation and scale change is allowed. Their matching algorithm is based on finding correspondence between sides of polygons that approximate the 2-D shape curves. Another special feature based recognition technique is the one developed by Kalvin *et al.* [7]. This technique makes use of *breakpoints* and carry by nature the handicap mentioned for [5].

Works dealing with 3-D also exists. Kishon and Wolfson [8] introduce the arclength, curvature and torsion as signatures of a 3-D curve but decide not use torsion because its requirement to the third derivative. The matching problem is attacked as a longest substring search problem in their work. Kishon, in his work [9] proposes a spline fit which enables the easy incorporation of torsion as a stable signature. In another work [10], Schwartz and Sharir propose various metrics (like color on the boundaries) and a smoothing operation on the data.

There exists real world problems where a 2-D solution is insufficient *(Reconstruction from broken pieces of solid objects is one of them)* so a 3-D solid model is inevitable. Furthermore, in many of those real world problems a perfect match between two subjects is not possible. Environmental aging effects, imperfections in the digitization environment, the accumulation of systematic errors in numerical operations all contribute to this imperfection. Therefore, a robust, fault tolerant

partial matching is required. This work proposes such a solution. [1] In our work 3-D surface piece objects are represented by their boundary curves. These closed curves are parameterized by their *curvature* and *torsion* scalars which are calculated from the discrete 3-D boundary curve data. A noise tolerant matching algorithm serves to find the best match of two such circular strings even for cases where the match is fragmented. By means of a search over all pairings of pieces the best fit is picked and for this pair the pieces are removed from the piece pool and then reinserted as a joint single piece. Henceforth this iteration is continued until a single piece only remains.

The heart of this proposed technique is based on a matching algorithm devised for the representation of two piece. The algorithm can be schematized as follows:

- Construct the internal representation for each 3-D piece making use of its discrete boundary curve data, namely κ_i the curvature, and τ_i the torsion values (where i is the discretization index over the boundary). Consider $[\kappa_i, \tau_i]$ as the feature vector at the point i.
- For each pair of the representation:
  - Construct a similarity matrix Λ with elements Λ_{ij} defined as the Euclidean distance between the feature vectors ξ_i and η_j where ξ reside on one piece and η on the other. The indices i and j range over all possible discretization points of the two pieces, respectively.
  - Λ_{ij} values which are less then a noise treshold ε are considered as matching points. By processing of the similarity matrix, all matching fragments are determined.
  - Among sequences of such matching points the longest sequence of matching fragments will be determined in a noice tolerant manner.[a] Considering the start and end information (the row and column indices in the similarity matrix) of the fragments the longest non overlapping sequence of fragments is determined (this is a non trivial job, since due to the fault tolerence, a point on a piece can belong to more then one matching fragment, hence it is very possible to have overlapping fragments)
- The best fit is picked on a maximum count of matching point base. The join procedure is carried out and the new created piece is replaced in place of the joined pieces.

[a] Noise tolerance in the algorithm is introduced in two aspects: (1) introducing a measure of closeness of two feature vectors (as mentioned above), (2) having a control over the count of fragmentation.

This work mainly focuses on the matching algorithm part of the above described procedure.

The rest of this paper is organized as follows: The mathematical foundation information is introduced in the next section; section 3 covers the proposed matching algorithm; in section 4 a brief coverage of the prototype system is given; the last section presents some concluding remarks are presented.

[1] An early version of this paper appeared in [11]

2 Mathematical Representation of the Problem

We will assume that the object which will be reassembled has no thickness, hence can be represented by a surface in a 3-D Euclidean space. The pieces of a surface structure embedded in a 3-D space are surfaces with boundaries that are closed curves of the 3-D space. Since a matching over these closed curves corresponds to the task of reassembling, a coordinate independent parameterization of these curves are very desirable. The fundamental theorem of the local theory of curves (see [12, 13]) reads as

> *Given differentiable functions $\kappa(s) > 0$ and $\tau(s)$, $s \in \mathbf{I}$, there exists a regular parameterized curve $\vec{r} : \mathbf{I} \to \mathbf{R}^3$ such that s is the arc length, $\kappa(s)$ is the curvature, and $\tau(s)$ is the torsion of $\vec{r}$. Moreover, any other curve $\vec{\bar{r}}$, satisfying the same conditions, differs from $\vec{r}$ by a rigid motion; that is, there exists an orthogonal linear map Ω of $\mathbf{R}^3$, with positive determinant, and a vector $\vec{c}$ such that $\vec{\bar{r}} = \Omega \circ \vec{r} + \vec{c}$.*

What we can conclude from this theorem is exactly what we were looking for:

> *If two different curves which are parameterized by their arc length produce the same torsion and curvature values then we can conclude that these curves are the same (modulo rotation and translation).*

Furthermore, the converse is also true. *Curvature* is defined as

$$\kappa = |\vec{r}\,''|$$

Torsion is defined as

$$\tau = \frac{1}{\kappa^2}\left[\vec{r}\,'\vec{r}\,''\vec{r}\,'''\right]$$

where the square brackets $[\cdots]$ have the special meaning of

$$\left[\vec{A}\vec{B}\vec{C}\right] \equiv \begin{vmatrix} A_x & A_y & A_z \\ B_x & B_y & B_z \\ C_x & C_y & C_z \end{vmatrix}$$

Furthermore the prime denotes differentiation with respect to the arc length s:

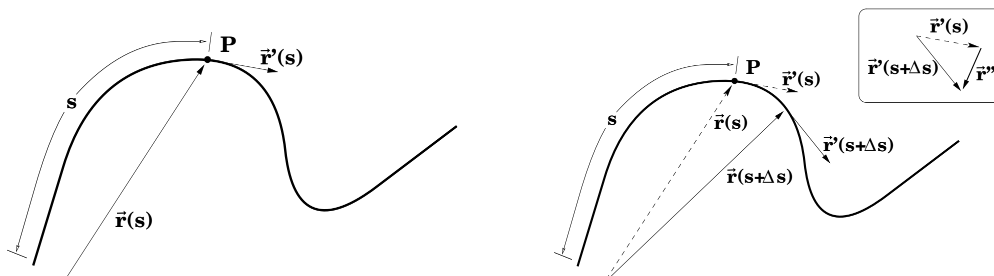
$$\vec{r}\,' = \frac{d\vec{r}}{ds}$$

As known s is defined by:

$$s(t) = \int_0^t ds = \int_0^t \sqrt{d\vec{r}\cdot d\vec{r}} = \int_0^t \sqrt{dx^2 + dy^2 + dz^2}$$

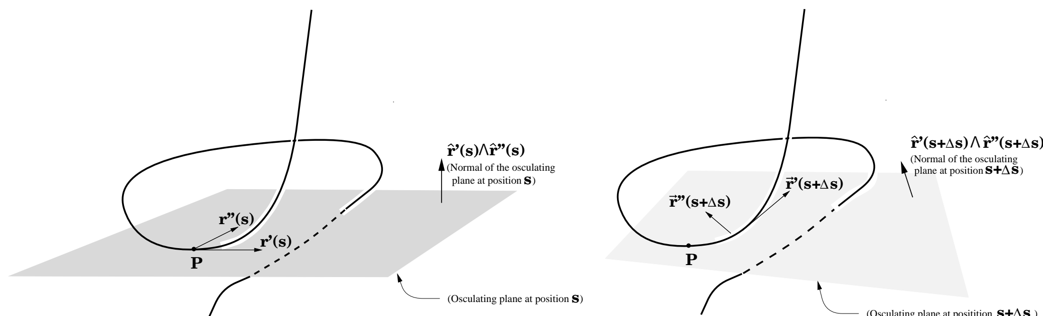
Where t is the *parameter* of the curve that maps each value in an interval in $\mathbf{R}$ into a point $r(t) = (x(t), y(t), z(t)) \in \mathbf{R}^3$ in such a way that the functions $x(t)$, $y(t)$, $z(t)$ are differentiable.

Intuitively speaking, the *curvature* at a point on the curve is the measure of how rapidly the curve pulls away from the tangent line at that point (so in a close neighborhood of that point we will have a deviated tangent line).



Tangent is nothing else than the change in the position vector $\vec{r}$ namely $\vec{r}\,'$. The magnitude of the change rate of this vector $|\vec{r}\,''|$is called *curvature*.

Consider at any point on the curve the plane formed to include the vectors $\vec{r}\,'$ and $\vec{r}\,''$ (at that point). This plane is called the osculating plane of that point. Again intuitively speaking, the *torsion* at a point on the curve is the measure of how rapidly the curve pulls away form the osculating plate at that point (so in a close neighborhood of that point we will have a deviated osculating plane).



Osculating plane is the plane that contains the $\vec{r}\,'$ and $\vec{r}\,''$ vectors. Of course this plane changes from point to point. *torsion* is the scalar measure of the rate of deviation of this plane (the deviation of the normal of the plane). *torsion* is defined as the change in the magnitude of this deviation. This is so because calculation reveals that the direction of the change is always in the direction of $\vec{r}\,''$

In the discrete case we have instances of r which are labeled with an index i. We assume that the labeling is done such that for any two r_i and r_{i+1} instances there exist no provided r_k value that corresponds to a curve point that is between them. Hence, the index is the discrete form of the *curve parameter*. Differentials will be replaced by differences with the following definitions

$$\Delta x_i = x_i - x_{i-1} \qquad \Delta y_i = y_i - y_{i-1} \qquad \Delta y_i = y_i - y_{i-1}$$

$$\Delta s_i = \sqrt{\Delta x_i^2 + \Delta y_i^2 + \Delta z_i^2}$$

So for the arc length we have $s_i = \sum_{k=1}^{i} \Delta s_i$. Once obtained the tuples $(\vec{r}_i, s_i)$ the $\vec{r}\,', \vec{r}\,'', \vec{r}\,'''$ are calculated for equally spaced (δs) points in the usual manner. To avoid local divergent behaviors the derivatives are calculated as an *average* value in a given radius of neighborhood. Experimentation has shown that a δs value which is large enough to accommodate $\sim 20\ \Delta s_i$ values performs very well.

At each discrete boundary curve point of the piece the κ_i and τ_i values form a 2-dimensional feature vector $[\kappa_i, \tau_i]$ which we will denote as ξ_i (or sometimes as η_i). The sequence of feature vectors ξ_i forms the shape signature string. Since the objects dealt with are defined to have closed boundary curves, in all algorithms operating on the shape signature strings the assumption that these strings round over (i.e. be circular) will be made.

3 The Matching Algorithm

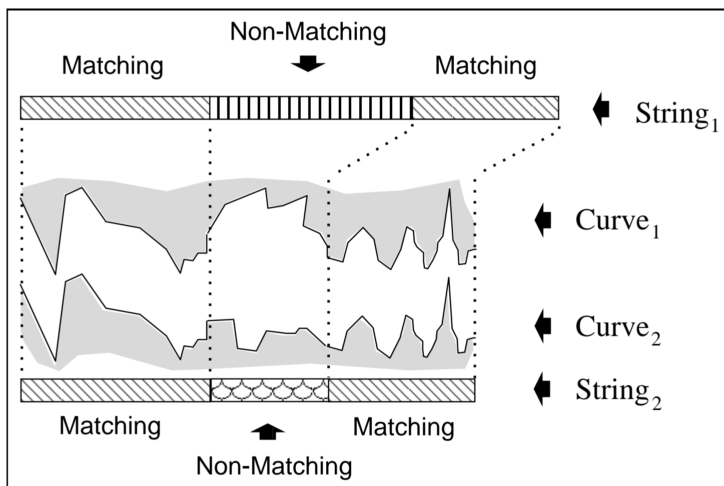


Figure 1: Two matching segments having a missing part

Since we are dealing with broken pieces which might have worn off contours the algorithm shall be

- robust in matching (i.e. fault tolerant),
- allow the non-existence of some minor pieces.

In Figure 1 two pieces with some missing portion and the affect of this on the string representation is illustrated. In the chosen representation, this corresponds to

- accepting numerical matches with an ε tolerance,
- being able to resume the match after a gap of non-matching data.

In this way each broken piece boundary is represented by an *array of feature vectors*. Throughout the rest of the paper we will refer to an example of two such feature vector arrays (Figure 2).

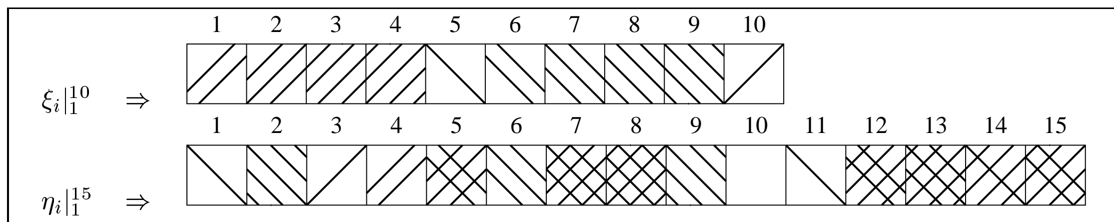


Figure 2: Example for the feature vector arrays: $\xi_i|_1^R, \eta_i|_1^C$ where each feature vector is represented by hatches. The pictorial rule of having a match of ξ_i with η_j is that either the count of up inclined hatches or the count of down inclined hatches matches. (e.g. ξ_4 matches (only-and-only) η_8, η_{13} and η_{15}).

The devised algorithm to match two curves represented respectively by the strings $\xi_i|_1^R$ and $\eta_i|_1^C$ (ξ_i and η_i are feature vectors) is as follows: we define a *similarity matrix* Λ as

$$\Lambda_{ij} = \| \xi_i - \eta_j \|$$

This is nothing else but the Euclidean distance of two feature vectors ξ_i and η_j where the row index i and the coulumn index j take values over the the first and second pieces, respectively. Below, in Figure 3 an example of such a similary matrix is given.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	×	×	×	●	×	×	×	×	×	×	×	×	×	×	×
2	×	×	×	×	●	×	×	×	×	×	×	×	×	×	×
3	×	×	×	×	×	×	●	×	×	×	×	●	×	●	×
4	×	×	×	×	×	×	×	●	×	×	×	×	●	×	●
5	●	×	×	×	×	×	×	×	×	×	●	×	×	●	×
6	×	×	×	×	●	×	×	×	×	×	×	●	×	×	●
7	×	×	×	×	×	●	●	×	×	×	×	×	●	×	×
8	×	×	×	×	×	×	×	●	×	×	×	×	×	×	×
9	×	●	×	×	×	×	×	×	●	×	×	×	×	×	×
10	×	×	●	×	×	×	×	×	×	×	×	×	×	×	×

Figure 3: An example for the Λ matrix. In this example we consider two pieces, each of which is represented by a sequence of feature vectors with 10 and 15 elements, respectively. (×) represent a value greater than ε; others (●) are values less than ε

In the following *match* algorithm a two dimensional array $\mathbf{M}$ is filled out. $\mathbf{M}$ will be holding the start and end positions of the matching segments. Each such position is represented by two integers standing for the row and column number of that matrix entry, respectively. So, one index takes values as *start* or *end*. The second index runs through an enumeration of the found matching segments. $\mathbf{M}_p^{start}$ and $\mathbf{M}_p^{end}$ hold the start and end *position informations* of the found p^{th} segment, respectively.

$predecessor(i,j) \leftarrow \{$
  if $i = 1$ **then** $k \leftarrow R$ **else** $k \leftarrow i - 1$
  if $j = 1$ **then** $l \leftarrow C$ **else** $l \leftarrow j - 1$
  return $(k,l)\}$
$\}$

$successor(i,j) \leftarrow ((i \bmod R) + 1, (j \bmod C) + 1)$

$match() \leftarrow \{$
  $S \leftarrow \min\{R, C\}$
  $p \leftarrow 0$
  for $i \leftarrow 1 \ldots R$ **do**
    for $j \leftarrow 1 \ldots C$ **do**
      if $\Lambda_{ij} \leq \varepsilon \ \wedge \ \Lambda_{predecessor(i,j)} > \varepsilon$ **then**
        $\{\ (k,l) \leftarrow (i,j)$
        $m \leftarrow 0$
        repeat $\{\ m \leftarrow m + 1$
          $(k,l) \leftarrow successor(k,l)\ \}$
        until $m \geq S \ \vee \ \Lambda_{kl} > \varepsilon$
        $p \leftarrow p + 1$
        $\mathbf{M}_p^{start} \leftarrow (i,j)$
        $\mathbf{M}_p^{end} \leftarrow predecessor(k,l)\ \}$
$\}$

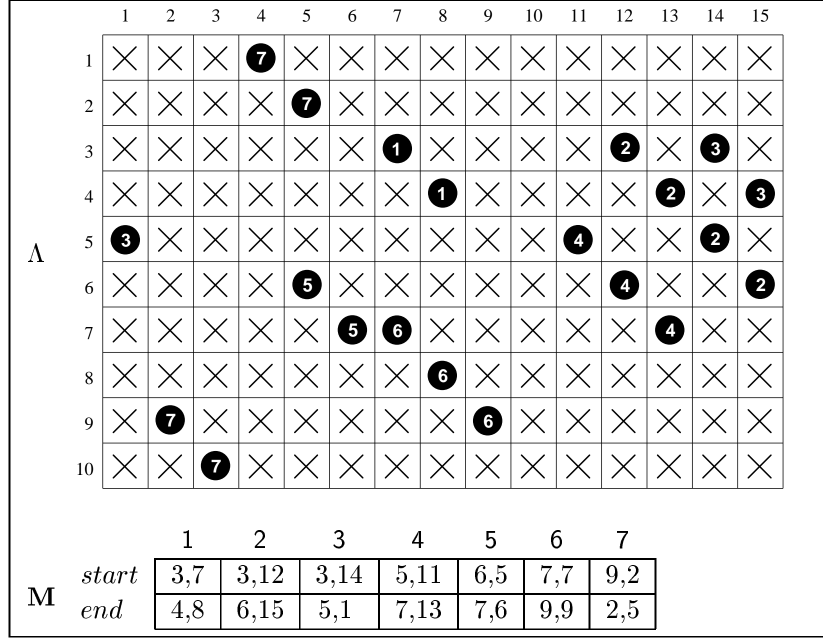


Figure 4: An example for the Λ matrix and the $\mathbf{M}$ array formed. The match algorithm determines the matching segments and stores the corresponding start and end positions in the Λ matrix into the $\mathbf{M}$ array. Numbers shown in the (●) (bullets) are the segment numbers.

Figure 4 shows an example Λ matrix, corresponding two broken pieces with 15 and 10 feature points, respectively, which will be used in this section to explain the matching process. When the *match*() procedure is executed for the above example it will detect seven matching segments. It will construct an $\mathbf{M}$ array with those segments being its members. For example the third segment will be represented in this array with its start and end entries in the matrix as $\mathbf{M}_3^{start} = (3, 14)$ and $\mathbf{M}_3^{end} = (5, 1)$.

From now on, denotationally, we will represent segments by a naming (e.g. α, β or α_i). Each segment, naturally, has four values associated: its start position (a *row* and a *column* number) in the matrix Λ and its end position (a *row* and a *column* number). These are represented by the appropriate combination of an superscript which is either *start* or *end* and a subscript that is either *row* or *column*. For example, the segment $\mathbf{M}_3$ would be represented as:

$$\alpha 3_{row}^{start} = 3 \quad \text{and} \quad \alpha 3_{col}^{start} = 15$$
$$\alpha 3_{row}^{end} = 5 \quad \text{and} \quad \alpha 3_{col}^{end} = 1$$

The next task is to determine, among the segments found, which can follow which. As was stated, due to the circular structure of the matched curves a special treatment is necessary in finding the answer to this question. To avoid the halting problem of the algorithm we impose a canonical order onto the concept of following. The canonical order we will impose says that if a segment β is *following* a segment α then

$$\alpha_{row}^{end} < \beta_{row}^{start}$$

Of course this is 'a necessary but not sufficient' criteria that has to be met. (The converse is not always true: you can have *non-following* two segments α and β where $\alpha_{row}^{end} < \beta_{row}^{start}$ still holds). To complete the definition of the following segments we consider the possible positions of a segment α (which is going to be followed by β) in the Λ matrix as it is picturized in Figure 5.

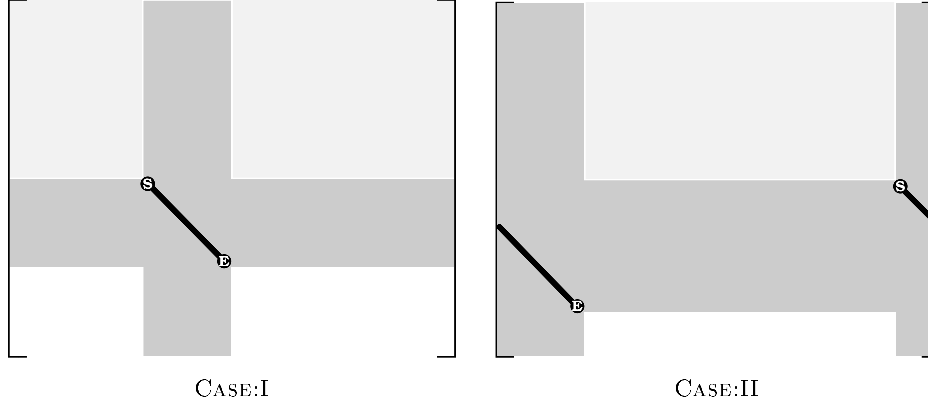


Figure 5: Visual representation of the following segments concept: Light shaded areas are forbidden zones for the following segment to start in due to the imposed canonical order but it may end in there; dark shades are the regions where an overlapping would occur, so the following segment shall have no points in there.

Consider the given similarity matrix of Figure 4. An example for CASE I is:

Segment ❶ is followed by Segment ❹ and Segment ❺

Similarly, an example for CASE II is:

Segment ❸ is followed by Segment ❺ and Segment ❻

We define a comparison operator $\prec$ that will admit two segments as operands and return True if the right operand is a *following segment* of the left one and False otherwise. Formally this operator can be defined as *(we are making use of the mathematical notation for representing closed/open/semiclosed sets; in our cases set elements are integral values)*:

$\alpha \prec \beta \quad \leftarrow \quad \{$

if $\neg wrapped_{col}(\alpha)$ **then**
  if $\beta^{start}_{row} \in (\alpha^{end}_{row}, R]$ **then**
    if $\beta^{start}_{col} \in (\alpha^{end}_{col}, C]$ **then**
      $\{$
        if $wrapped_{col}(\beta)$ **then**
          if $\beta^{end}_{col} \notin [1, \alpha^{start}_{col})$ **then return** $(FALSE)$
        else
          if $\beta^{end}_{col} \notin (\alpha^{end}_{col}, C]$ **then return** $(FALSE)$
        if $wrapped_{row}(\beta)$ **then**
          if $\beta^{end}_{row} \notin [1, \alpha^{start}_{row})$ **then return** $(FALSE)$
        else
          if $\beta^{end}_{row} \notin (\alpha^{end}_{row}, R]$ **then return** $(FALSE)$
      $\}$
  else
    if $\beta^{start}_{col} \in [1, \alpha^{start}_{col})$ **then**
      $\{$
        if $wrapped_{row}(\beta)$ **then**
          if $\beta^{end}_{row} \notin [1, \alpha^{start}_{row}) \ \vee \ \beta^{end}_{col} \notin [1, \alpha^{start}_{col})$ **then return** $(FALSE)$
        else
          if $\beta^{end}_{row} \notin (\beta^{start}_{row}, R] \ \vee \ \beta^{end}_{col} \notin (\beta^{start}_{col}, \alpha^{start}_{col})$ **then return** $(FALSE)$
        if $wrapped_{col}(\beta)$ **then return** $(FALSE)$

```
            }
         else return (FALSE)
      else return (FALSE)
   else
      if β_row^start ∈ (α_row^end, R] ∧ β_col^start ∈ (α_col^end, α_col^start) then
        {
          if wrapped_row(β) then
             if β_row^end ∉ [1, α_row^start) ∨ β_col^end ∉ (α_col^end, α_col^start)  then return (FALSE)
          else
             if β_row^end ∉ (β_row^start, R] ∨ β_col^end ∉ (β_col^start, α_col^start) then return (FALSE)
          if wrapped_col(β) then return (FALSE)
        }
      else return (FALSE)
   if wrapped_row(α) then return (FALSE)
   return (TRUE)
}
```

$wrapped_{\delta}(\chi) \quad \leftarrow \quad \chi_{\delta}^{start} > \chi_{\delta}^{end} \quad , \quad {\scriptstyle \delta \in \{row, col\}}$

The $\prec$ operator will yield always the correct answer for the cases where α is following β or β si following α. On the other hand for segments that have overlapping regions the answer is undetermined. Hence we are able to define a partial order among the set of all found segments, namely the $\mathbf{M}$ array.

For our example the following relations among the matching segments will be true.

$$\mathbf{M}_1 \prec \mathbf{M}_4,\ \mathbf{M}_1 \prec \mathbf{M}_5,\ \mathbf{M}_1 \prec \mathbf{M}_7,\ \mathbf{M}_2 \prec \mathbf{M}_6,\ \mathbf{M}_2 \prec \mathbf{M}_7,\ \mathbf{M}_3 \prec \mathbf{M}_5,$$
$$\mathbf{M}_3 \prec \mathbf{M}_6,\ \mathbf{M}_3 \prec \mathbf{M}_7,\ \mathbf{M}_4 \prec \mathbf{M}_7,$$

After all the match segments are determined, a sequence of these segments should be chosen satisfying the non-overlapping property in order to match and combine the two broken pieces. The join will be performed over those sequences of (matching) segments. Therefore, first, we should find all possible segment sequences which are not subsequences of each others. We might then prefer the longest one of the possible segment sequences representing the best possible match between two pieces.

Our algorithm works as follows: each match segment is processed one by one from 1 to p (number of segments determined by the *match* procedure between these two pieces), trying to generate new sequences. Each time a new segment is processed, it is tried against all the previously obtained sequences to determine if it is possible to add the new segment to those sequences, or to generate a new sequence from of segments of the sequence which are followed by the new segment.

In the following program n is the global variable which is the counter for the sequences ($\mathbf{S}_j$'s) generated from the match segments ($\mathbf{M}_i$'s where i is beween $1 \ldots p$ obtained by the *match* procedure).

$\mathbf{S}_j \leftarrow\oplus \mathbf{M}_i$ is a special operator which adds the match segment $\mathbf{M}_i$ into the sequence $\mathbf{S}_j$. Each sequence $\mathbf{S}_j$ holds both the number of match segments ($\mathbf{S}_j.ln$) and the match segments in it ($\mathbf{S}_j.\mathbf{M}_{j_k}$'s).

find_all_match_sequences() is the main procedure which generates all sequences of the match segments. Initially it generates a trivial sequence $\mathbf{S}_1$ from the first segment $\mathbf{M}_1$. After that, all the segments are considered by the outer for loop, processing all the previously generated sequences ($\mathbf{S}_j$'s) with the new segment $\mathbf{M}_i$ in the inner for loop.

```
find_all_match_sequences() ← {
        n ← 1
        S_n ←⊕ M_1
        for i ← 2 ... p do
            tn ← n
            for j ← 1 ... tn do
                if (create_new_sequence(S_j, M_i)) then add_if_not_exist(S_n, tn)
        }
```

The below defined $create_new_sequence(\mathbf{S}_j, \mathbf{M}_i)$ function creates, if possible, a new sequence from the previously generated sequence $\mathbf{S}_j$ and a match segment $\mathbf{M}_i$. First, from $\mathbf{S}_j$, all those match segments which are followed by $\mathbf{M}_i$ are determined. If all the match segments of $\mathbf{S}_j$ are followed by $\mathbf{M}_i$, then instead of creating new sequence, $\mathbf{M}_i$ is appended to $\mathbf{S}_j$ and the procedure returns FALSE. If $\mathbf{M}_i$ is followed ony by the subset of the match segments of the sequence $\mathbf{S}_j$, then using those match segments and $\mathbf{M}_i$ a new sequence is generated, and the procedure returns TRUE. *(Below $\mathbf{M}_{j_k}$ is the kth match segment of sequence j)*

```
create_new_sequence(S_j, M_i) ← {
        n ← n + 1
        S_n.ln ← 0
        for k ← 1 ... S_j.ln do
            if S_j.M_{j_k} ≺ M_i then
                        { S_n.ln ← S_n.ln + 1
                          S_n ←⊕ S_j.M_{j_k} }
            if S_n.ln = S_j.ln then
                        { n ← n − 1
                          S_j.ln ← S_j.ln + 1
                          S_j ←⊕ M_i
                          return (FALSE) }
                    else
                        { S_n.ln ← S_n.ln + 1
                          S_n ←⊕ M_i
                          return (TRUE) }
        }
```

The following procedure, $add_if_not_exist(\mathbf{S}_n, tn)$, is called only if a new sequence is generated by processing match segment $\mathbf{M}_i$. The match segment $\mathbf{M}_i$ might follow the subset of (or exactly the same) match segments in some previously generated sequences. Such a case means the newly generated sequence is redundant and it is discarded from the list of sequences. One of the previously generated sequence could be a subset of the new sequence and in that case that previously generated sequence becomes redundant, hence must be eliminated. To do this check, the new sequence is only compared with the sequences generated during the processing of the match segment $\mathbf{M}_i$.

```
add_if_not_exist(S_n, tn) ← {
        for i ← tn + 1 ... n − 1 do
            if S_n ⊆ S_i then n ← n − 1
                    else if S_i ⊆ S_n then
                            { n ← n − 1
                              S_i ← S_n }
        }
```

For our example, after applying the above described procedure, finally, the following sequences would be generated:

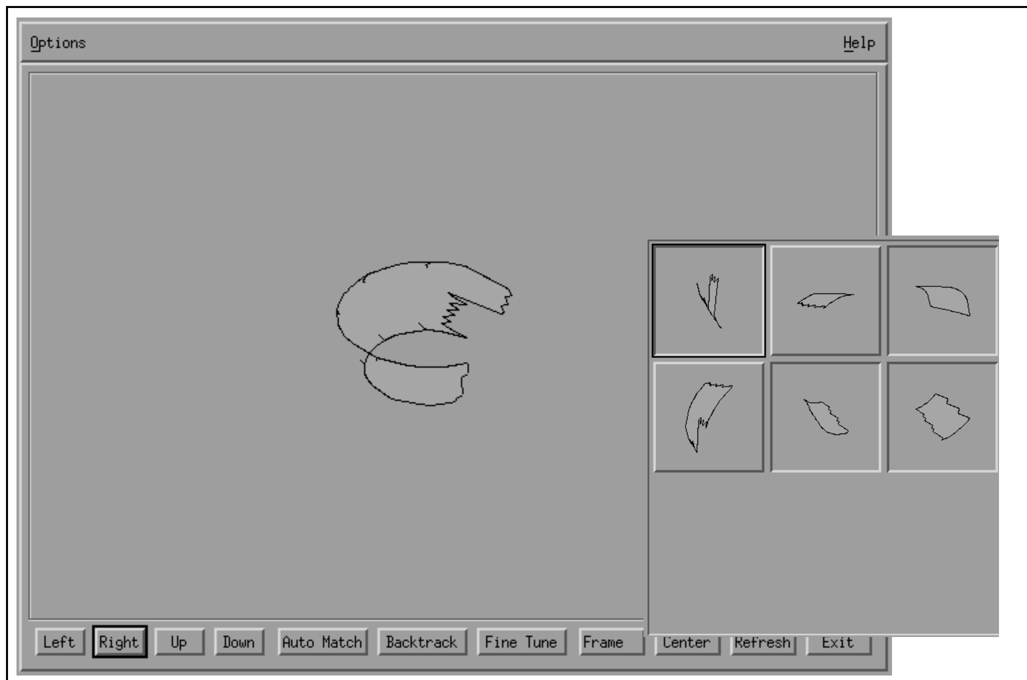
$$\mathbf{M}_1\mathbf{M}_4\mathbf{M}_7,\ \mathbf{M}_1\mathbf{M}_5,\ \mathbf{M}_2\mathbf{M}_6,\ \mathbf{M}_2\mathbf{M}_7,\ \mathbf{M}_3\mathbf{M}_5,\ \mathbf{M}_3\mathbf{M}_6,\ \mathbf{M}_3\mathbf{M}_7,\ \mathbf{M}_4\mathbf{M}_7$$

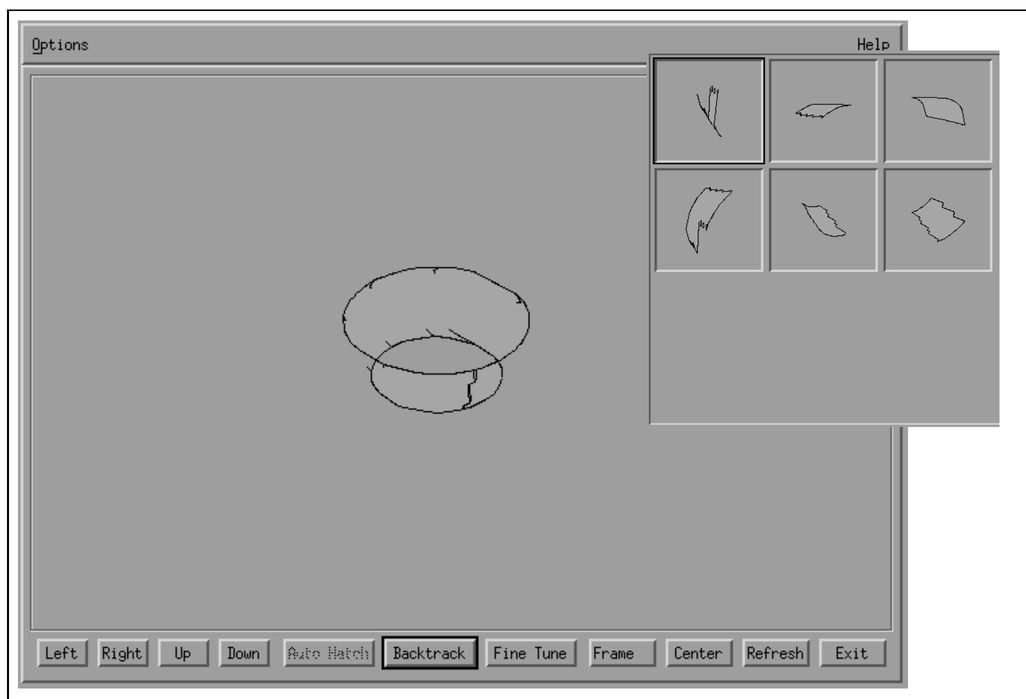
After all the sequences are generated, the longest can be choosen. However, instead of simply summing up the lengths of the match segments in the sequence, more complicated techniques might be used. As an example, we might prefer longer segments to the shorter ones, therefore, we might use the summation of the square of the lengths of the segments in determining the sequence that we choose.

4 Implementation

An early version of the system is implemented as a prototype. The GUI is based on X-Motif and fully capable of 3-D view-operations. The system starts by generating a user determined number of random broken pieces of a rotational surface shape where the rotational cross section function is defined by the user. The user has control over the noise introduced on the boundary curve of the broken pieces which simulates, in a very natural way, worn outs due to ageing effects. The system continues by generating widgets of buttons for each broken-piece each of which is marked by a 2-D view of that broken piece. The user sellects a broken piece by pressing the corresponding button. The system then starts to search the workspace which contains the partially reconstructed object (at the start of the session this space is empty) for a best match with the chosen piece. If such a match is found the coordinate transformation necessary to 'stick' the piece into its found place is automatically calculated. The piece moves to its place and 'updates' the partially reconstructed object by integrating itself into it: The matched curve portion is removed and the boundary is updated to include the boundary of the unmatched part of the new piece. Tests reveals that the system reacts almost instantly on a SUN-4 system where each piece (about 20 pieces), on average, is represented by a feature vector of 100 elements.

The pictures below displays the two snapshots of the system-user interaction window just before and after the last move which completes the reconstruction.





5 Conclusion

We presented a method for matching two closed space curves which are holding discrete feature values, in a robust manner. Unlike in other related works the problem of the proper treatment of missing parts in a match is put under focus and a complete solution is proposed. The reconstruction of the object is just an exhaustive search over all 'pieces' and choosing the best fittings. The idea is as follows:

- Find the best match.
- Join the matching portions (perform in parallel the necessary bookkeeping).
- Removing the parts of the joint obtain the representation of a single piece.
- Add this new obtained piece and remove the two pieces which were joined from the database, hence reducing the count of pieces by one, continue until only one piece is left.

Further efforts can go into the implementation details where a suitable data representation and efficient retrieval mechanisms will be the main concern.

6 Acknowledgement

We would like to thank Mr. Yılmaz Çeken for the development of the tools used for the generation of test data and the user interface as part of his MS thesis [14].

References

[1] H. Freeman and L. Garder. A pictorial jigsaw puzzles: The computer solution of a problem in pattern recognition. *IEEE Trans. Electron. Comput.*, EC-13:118–127, 1964.

[2] G. M. Radack and N. I. Badler. Jigsaw puzzle matching using a boundary-centered polar encoding. *Comput. Graphics Image Processing*, 19:1–17, 1982.

[3] H. Wolfson, E. Schonberg, A Kalvin, and Y. Lambdan. Solving jigsaw puzzle using computer vision. *Ann. Oper. Res.*, 12:51–64, 1988.

[4] H. Wolfson. On curve matching. *IEEE, Trans. Pattern. Anal. Machine. Intell.*, 12:483–489, 1990.

[5] H. Freeman. Shape description via the use of critical points. *Pattern Recogn.*, 10:159–166, 1978.

[6] N. Ayache and O. D. Faugeras. Hyper: A new approach for the recognition and positioning of two-dimensional objects. *IEEE, Trans. Pattern. Anal. Machine. Intell.*, 8:44–54, 1986.

[7] A. Kalvin, E. Schonberg, J. T. Schwartz and M. Sharir. Two-dimensional model-based boundary matching using footprints. *The Int. J. of Robotics Research*, 5:38–55, 1986.

[8] E. Kishon and H. Wolfson. 3-d curve matching. In *Proceeding of the AAAI Workshop on Spatial Reasoning and Multi-sensor Fusion*, pages 250–261, 1987.

[9] E. Kishon, T. Hastie, and H. Wolfson. 3d curve matching using splines. In *First Europian Conference on Computer Vision*, pages 589–591, 1990.

[10] J. T. Schwartz and M. Sharir. Identification of partially obscured objects in two and three dimension by matching noisy characteristic curves. *IEEE, Trans. Pattern. Anal. Machine. Intell.*, 8:44–54, 1986.

[11] G. Ucoluk and I.H. Toroslu. Reconstruction of 3-d surface object from its pieces. In *Proceeding of the Ninth Canadian Conference on Computational Geometry*, pages 187–192, 1997.

[12] Manfredo P. do Carmo. *Differential geometry of curve and surfaces.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[13] Abraham Goetz. *Introduction to differential geometry of curve and surfaces.* Prentice-Hall, Englewood Cliffs, New Jersey, 1970.

[14] Yılmaz Çeken. *Three dimensional object reconstruction from boundary data.* METU, Ankara, Turkey, 1995.