Reconstruction of 3-D Surface Object from its Pieces

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.

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 Kevin *et al.*. 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 [7] 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 [8] proposes a spline fit which enables the easy incorporation of torsion as a stable signature. In another work [9], 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. 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

and quadratically added to form a circular string of a single value. A noise tolerant matching algorithm serves to find the best match of two such circular strings even for cases where the match is fragmented.

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 [10, 11]) 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 $\vec{r}'$, 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 $\vec{r}' = \Omega \circ \vec{r} + \vec{c}$.*

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

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

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:

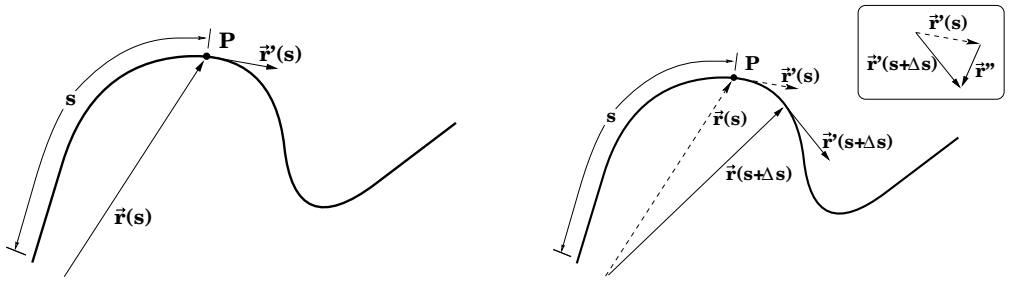
$$\vec{r}\,' = \frac{d\vec{r}}{ds}$$

As known s is defined by:

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

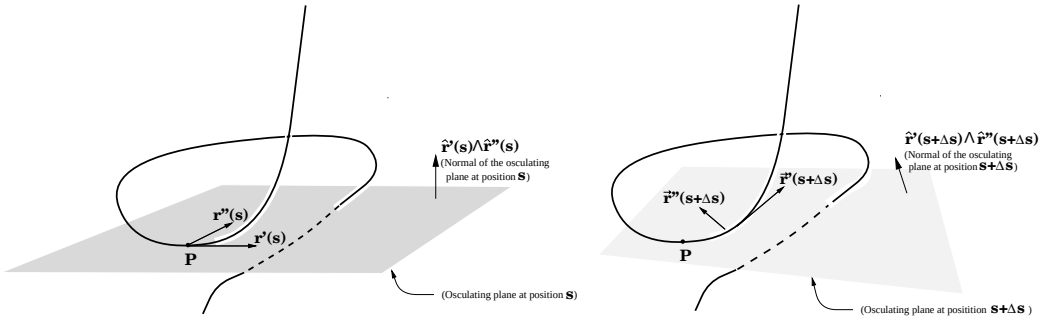
Where t is the *parameter* of the curve that maps each value in an interval in $\mathbf{R}$ into a point $r(t) = (x(t), y(t), z(t)) \in \mathbf{R^3}$ in such a way that the functions $x(t)$, $y(t)$, $z(t)$ are differentiable.

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.

The κ_i and τ_i values form a 2-dimensional feature vector ξ_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

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

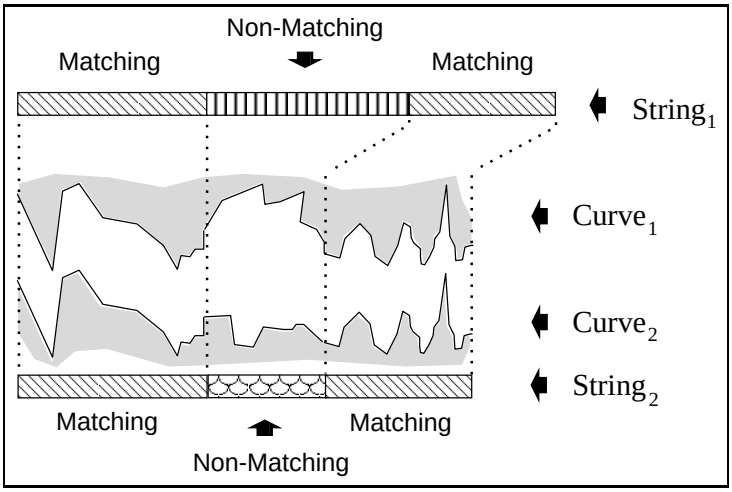


Figure 1: Two matching segments having a missing part

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

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 matrix Λ as

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

So Λ is a symmetric with nonnegative entries.

In the following algorithm a two dimensional array $\mathbf{M}$ is filled out. $\mathbf{M}$ will be holding the start and end positions of the matching segments. 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. A position information of a start (or end) is a pair of indices, namely the row and column numbers of the Λ matrix where the segment starts (or ends).

$predecessor(i,j) \leftarrow \{$ **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 $(k,l)\}$ $\}$

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

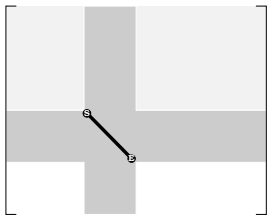
```
match() ← {
        S ← min{R, C}
        p ← 0
        for i ← 1 ... R do
            for j ← 1 ... C do
                if Λ_ij ∧ Λ_predecessor(i,j) > ε then
                  { (k,l) ← (i,j)
                    m ← 0
                    repeat { m ← m + 1
                             (k,l) ← successor(k,l) }
                    until m ≥ S ∨ Λ_kl > ε
                    p ← p + 1
                    M_p^start ← (i,j)
                    M_p^end ← predecessor(k,l) } }
        }
```

From know 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*.

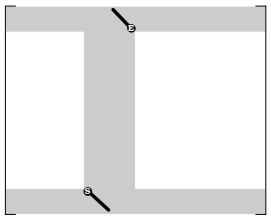
The next task is to determine, among the segments found, which can follow which. As was stated, due to the circular structure of the matched curves a special treatment is necessary in finding the answer to this question. To avoid the halting problem of the algorithm we impose a canonical order onto the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

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

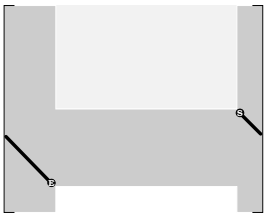
Of course this is 'a necessary but not sufficient' criteria that has to be met. (The converse is not always true: you can have *non-following* two segments α and β where $\alpha_{row}^{end} < \beta_{row}^{start}$ still holds). To complete the definition of the following segments we consider the possible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



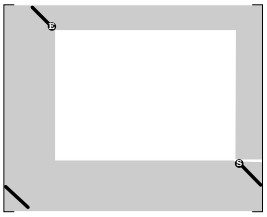
CASE:I



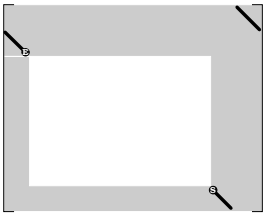
CASE:II



CASE:III



CASE:IV



CASE:V

Start and end point-wise, CASE IV and CASE V are not different from each other, so they will be considered as the same. 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)*:

$$\begin{aligned}
\alpha \prec \beta \quad \leftarrow \quad & \textbf{if } wrapped_{row}(\alpha) \textbf{ then } \beta_{row}^{start} \in (\alpha_{row}^{end}, \alpha_{row}^{start}) \ \wedge \ \beta_{row}^{end} \in (\beta_{row}^{start}, \alpha_{row}^{start}) \\
& \qquad\qquad \textbf{else } \beta_{row}^{start} \in (\alpha_{row}^{end}, R] \ \wedge \ \beta_{row}^{end} \in (\beta_{row}^{start}, R] \cup [1, \alpha_{row}^{start}) \\
& \wedge \\
& \textbf{if } wrapped_{col}(\alpha) \textbf{ then } \beta_{col}^{start} \in (\alpha_{col}^{end}, \alpha_{col}^{start}) \ \wedge \ \beta_{col}^{end} \in (\beta_{col}^{start}, \alpha_{col}^{start}) \\
& \qquad\qquad \textbf{else } \beta_{col}^{start} \in [1, \alpha_{col}^{start}) \cup (\alpha_{col}^{end}, C] \\
& \qquad\qquad\qquad \wedge \\
& \qquad\qquad\qquad \textbf{if } \beta_{col}^{start} \in (\alpha_{col}^{end}, C] \textbf{ then } \beta_{col}^{end} \in [1, \alpha_{col}^{start}) \cup (\alpha_{col}^{end}, C] \\
& \qquad\qquad\qquad\qquad\qquad \textbf{else } \beta_{col}^{end} \in [1, \alpha_{col}^{start})
\end{aligned}$$

Where we have defined

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

The $\prec$ operator will yield always the correct answer for the cases where α is following β or visa versa. For segments, though, 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. So, we will perform a topological sort on the $\mathbf{M}$ array where $\prec$ is the ordering criteria. A simple bubble sort will do the job *($\leftrightarrow$ stands for 'content exchange')*:

$$\begin{aligned}
sort_M() \leftarrow & \textbf{for } j \leftarrow 1 \ldots lastindex(\mathbf{M}) \textbf{ do} \\
& \quad \textbf{for } i \leftarrow 1 \ldots (lastindex(\mathbf{M}) - j) \textbf{ do} \\
& \quad\quad \textbf{if } \mathbf{M}_{i+1} \prec \mathbf{M}_i \textbf{ then } \mathbf{M}_i \leftrightarrow \mathbf{M}_{i+1}
\end{aligned}$$

The remaining of the algorithm is basically a search for the longest path in a graph where vertices are the segments and unidirectional edges between these vertices are introduced from segments to the *following* segments. The weights on the vertices are nothing else than the lengths of the segments. The task of finding the longest match is converted into a task in which the longest

path of the described graph is found. The longest path will yield a maximal weight (length of segments) sum of the vertices (segments) on the path. To start a search for the longest path we have to identify the *starting* and *terminating* vertices of the graph (i.e. those segments which are not following any segment and segments which are not followed by any segment). By two linear scans over the $\mathbf{M}$ array we are able to identify and mark those segments:

$$
\begin{aligned}
&mark\_terminatings() \leftarrow \{ \\
&\qquad \textbf{for } i \leftarrow 1 \ldots lastindex(\mathbf{M}) \textbf{ do } mark\_as\_terminating(\mathbf{M}_i) \\
&\qquad r \leftarrow \left[\mathbf{M}_{lastindex(\mathbf{M})}\right]_{row}^{start} \\
&\qquad c \leftarrow \left[\mathbf{M}_{lastindex(\mathbf{M})}\right]_{col}^{start} \\
&\qquad \textbf{for } i \leftarrow (lastindex(\mathbf{M}) - 1) \ldots 1 \textbf{ do} \\
&\qquad\quad \textbf{if } [\mathbf{M}_i]_{row}^{end} < r \ \wedge\ [\mathbf{M}_i]_{col}^{end} < c \textbf{ then } remove\_terminating\_mark(\mathbf{M}_i) \\
&\qquad\qquad\qquad\qquad \textbf{else if } [\mathbf{M}_i]_{row}^{start} > r \ \wedge\ [\mathbf{M}_i]_{col}^{start} > c \textbf{ then} \\
&\qquad\qquad\qquad\qquad\qquad \{\ r \leftarrow [\mathbf{M}_i]_{row}^{start} \\
&\qquad\qquad\qquad\qquad\qquad\ \ c \leftarrow [\mathbf{M}_i]_{col}^{start}\ \} \\
&\qquad \}
\end{aligned}
$$

$$
\begin{aligned}
&mark\_startings() \leftarrow \{ \\
&\qquad \textbf{for } i \leftarrow 1 \ldots lastindex(\mathbf{M}) \textbf{ do } mark\_as\_starting(\mathbf{M}_i) \\
&\qquad r \leftarrow [\mathbf{M}_1]_{row}^{end} \\
&\qquad c \leftarrow [\mathbf{M}_1]_{col}^{end} \\
&\qquad \textbf{for } i \leftarrow 2 \ldots lastindex(\mathbf{M}) \textbf{ do} \\
&\qquad\quad \textbf{if } [\mathbf{M}_i]_{row}^{start} > r \ \wedge\ [\mathbf{M}_i]_{col}^{start} > c \textbf{ then } remove\_starting\_mark(\mathbf{M}_i) \\
&\qquad\qquad\qquad\qquad \textbf{else if } [\mathbf{M}_i]_{row}^{end} < r \ \wedge\ [\mathbf{M}_i]_{col}^{end} < c \textbf{ then} \\
&\qquad\qquad\qquad\qquad\qquad \{\ r \leftarrow [\mathbf{M}_i]_{row}^{end} \\
&\qquad\qquad\qquad\qquad\qquad\ \ c \leftarrow [\mathbf{M}_i]_{col}^{end}\ \} \\
&\qquad \}
\end{aligned}
$$

For convenience we introduce two dummy vertices, namely at start $\mathbf{M}_0$ and at end $\mathbf{M}_\lambda$ (where $\lambda = lastindex(\mathbf{M}) + 1$) with length zero and satisfying the conditions

$$
\begin{aligned}
\forall i \ni marked\_as\_terminating(\mathbf{M}_i) &: \mathbf{M}_0 \prec \mathbf{M}_i \\
\forall i \ni marked\_as\_starting(\mathbf{M}_i) &: \mathbf{M}_i \prec \mathbf{M}_\lambda
\end{aligned}
$$

The below given greedy type algorithm uses dynamic programming to find the longest path. Gradually it fills out an array that we will name as *longest*:

$$
\begin{aligned}
&find\_longest\_path() \leftarrow \textbf{for } i \leftarrow lastindex(\mathbf{M}) \ldots 0 \textbf{ do} \\
&\qquad\qquad longest_i \leftarrow \max\{\, \omega(length(\mathbf{M}_k)) + longest_j \mid \mathbf{M}_i \prec \mathbf{M}_j \,\}
\end{aligned}
$$

length is a function that returns the count of match points of the segment given as argument to it. It is defined as:

$$
\begin{aligned}
&length(\alpha) \leftarrow \textbf{if } \alpha_{row}^{start} < \alpha_{row}^{end} \textbf{ then } \alpha_{row}^{end} - \alpha_{row}^{start} \\
&\qquad\qquad\qquad\qquad \textbf{else } R + \alpha_{row}^{end} - \alpha_{row}^{start}
\end{aligned}
$$

$\omega()$ is a function which defines the weight contribution of the count of match points for a continuous match segment given to it as the argument. The idea is to allow a penalty treatment for short matches. If no such penalty is favored then it is possible to simply define $\omega(m) = m$.

4 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 simple:

- 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.

For a possible implementation we would propose a visual workbench approach in which the user has a full control over the matching parameters and the matching itself and the availability of an undo operation over the construction history. A project of such an implementation has been started.

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

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] 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.

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

[9] 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.

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

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