Loop Snakes: Snakes with Enhanced Topology Control

Antonio Oliveira, Saulo Ribeiro, Ricardo Farias, Cláudio Esperança
Dept. of Systems Engineering and Computer Sciences - COPPE/UFRJ
Rio de Janeiro - R.J. , Brazil
oliveira, saulo, rfarias, esperanc@lcg.ufrj.br

Gilson Giraldi
National Lab. of Scientific Computing
Petrópolis - R.J. , Brazil
gilson@lncc.br

Abstract

Topologically adaptable snakes, or simply T-snakes, are a standard tool for automatically identifying multiple segments in an image. This work introduces a novel approach for controlling the topology of a T-snake. It focuses on the loops formed by the so-called projected curve which is obtained at every stage of the snake evolution. The idea is to make that curve the image of a piecewise linear mapping of an adequate class. Then, with the help of an additional structure—the Loop-Tree—it is possible to decide in O(1) time whether the region enclosed by each loop has already been explored by the snake. This makes it possible to construct an enhanced algorithm for evolving T-snakes whose performance is assessed by means of statistics and examples.

1. Introduction

The use of Active Contour Models or Snakes has become a standard technique for segmenting an image. Hundreds of works about the plain snakes model have been produced since it was introduced in [2]. One of its most obvious limitations, however, is the fact that for each contour that must be identified, a different snake has to be initialized by the user. Further, in some applications, all segments having certain properties must be found or simply counted and the number of them is large enough as to render this approach totally inappropriate, for instance: identifying cellular structures of a given type in microscope images, all blood vessels in an angiogram or electronic components on a board.

That limitation has been overcome with the introduction by McInerney and Terzopoulos in [3], of the T-snakes, short for topologically adaptable snakes. These snakes have the ability of changing their topology either by subdivision or by aggregation allowing every segment contour to be approached by exactly one snake or one snake from inside and another from outside. In ideal circumstances, T-snakes are used according to one of the following schemes:

(A) Initially, the whole image is encircled by a closed snake. During its evolution, that snake is continuously contracted and eventually broken into smaller ones which are subjected to the same contraction process. When all snakes have either been adjusted to a target contour or become too small, the process stops. (B) In a dual way, a series of very small snakes, the *seed snakes*, are randomly spread over the image. They are continuously expanded and when two of them collide they are merged into a single one. A contour having seeds in its interior will have a snake approaching it from the inside, whereas seeds in the background generate snakes approaching the contours from outside.

The framework of a T-snake, is that of a common snake plus a 2D-structure—the *Auxiliary Structure*, in short: *AS*—containing information that makes it possible to directly associate vertices of a polygonal curve which are close in 2D but far apart along the curve. This additional structure consists of a matrix whose elements are associated to the vertices [3], edges [1] or, as done here, cells of a square mesh covering the image domain.

The original motivation of our approach was to address some well-known difficulties of the original strategy proposed in [3], which is described in detail in Section 2. Similarly to that strategy, the approach proposed in this work also generates at a stage k, two curves: (I) The **physically transformed curve**—TC_k—which is defined by the new locations of the snaxels, after they have been moved by the forces of the chosen physical model from their positions in S_k, **the initial snake of the stage**. (II) The so-called **pro-**

jected curve—PC_k—, which is defined by the sequence of intersections between TC_k and mesh edges.

Assume that the snake moves continuously from S_k to PC_k. Then, it sweeps (or burns or visits, which in this context are used as synonyms), a whole strip of points. Call *visited set* the set of points swept in all stages up to the current one. While McInerney's approach uses the Auxiliary Sructure to explicitly control that visited set, this is accomplished implicitly in our approach. We focus on the loops formed by PC_k, some of which will give rise to the new snakes in stage $k+1$. For this reason, we choose to call them ***loop snakes***. PC_k is considered as the image of a dilated version of the snake S_k by a piecewise linear mapping Γ_k. Defining Γ_k on a dilated version of S_k makes it easier to satisfy a few local conditions which are necessary for making the strategy computationally attractive. Section 3 describes those conditions while Section 4 is dedicated to show how they can be enforced in straightforward way.

Some loops of PC_k, labeled *open* or *unexplored*, delimit regions yet to be visited by the snake while others, the *closed* or *explored* loops, enclose regions that have already been explored. Results shown in Section 5 demonstrate that it is possible to infer the *label of a loop*, simply by examining the label of adjacent loops found earlier. Even loops that have no earlier adjacent ones can be correctly labeled in $O(1)$ time.

The process is quite simple except for the case when PC_k returns to a cell already visited. Section 6 shows how to handle that case. Two possibilities must then be explored: either C becomes a *double cell*, that is, one with a PC_k vertex on each edge, or a topological change involving the edges of PC_k contained in C must be realized. The implementation of such a change depends on whether PC_k has a self-crossing or *knot* in C, or revisits an edge of C. Revisiting a cell triggers a more elaborated process but this occurs very infrequently if compared to the enormous number of snaxels generated, as is shown by statistics presented in Section 7.

Section 8 is devoted to conclusions and future works. For concision sake, here we discuss the case where snakes only contract. Nevertheless, the method can be easily extended in order to handle expanding snakes.

2. Other T-snakes Models

Essentially, in the original approach [3], the T-snake of step k—S_k **= [s_i , i = 0,...,I]**— is evolved as indicated below. The elements of the AS Matrix, which are related to the mesh vertices, are all initialized with "unvisited". In the following, τ will be the $K1$ triangulation of the mesh vertices, that is, the one obtained by cutting each cell along its main diagonal. A ***transition edge*** will be an edge of τ linking a visited and an unvisited vertex. Figure 1 represents an iteration of the method. For the sake of visibility S_k-snaxels and PC_k-vertices on diagonal edges have been excluded.

Procedure Evolving_an_original_T-Snake

- **Step 1.** Apply to each snaxel s_i, the movement determined by the physical model employed. The obtained points $(t_i, i = 0, \ldots, I)$ define the Transformed Curve, TC_k.
- **Step 2.** Build the Projected Curve, $PC_k = [s'_j, j = 0, ..., J]$ by concatenating the edges $[s'_j\ ,\ s'_{j+1}]$, defined by two successive intersections of TC_k with edges of τ.
- **Step 3.** For $i = 0, ..., I$, let Q_i be the quadrilateral defined by s_{i-1}, s_i, t_i and t_{i-1}. These will be termed the sweeping quadrilaterals of S_k. Check whether Q_i contains vertices of the mesh which are still unvisited and change the *AS* corresponding element to *"visited"*.

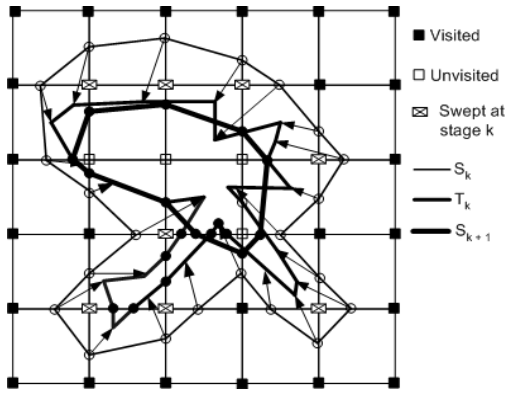

Figure 1. One iteration of the original T-snake.

- **Step 4.** Consider that all transition edges are initially unmarked and traverse PC_k. Every time an unmarked transition edge e_0 is reached execute:
  - **a)** Let $e = e_0$ and let t be one of the triangles adjacent to e_0. Then, do:
    - ***i***. Mark e and choose a point x_e on it considering the positions of the vertices of PC_k or S_k on e.
    - **ii**. Replace e by *e'*, the other transition edge adjacent to t , and t by *t'*, the other triangle adjacent to *e'*.
    - **iii**. Repeat $i - iii$ until $e = e_0$.
  - **b)** Take the closed polygonal line defined by points x_e as a snake of stage $k+1$.

Thus, a curve representing each connected component of the visited set boundary is constructed from its intersections with the transition edges and becomes a new snake in step 4. Some observations regarding the procedure above are pertinent:

- **1**. Checking whether unburned vertices are covered by the sweeping quadrilaterals Q_i (Step 3) is time consuming, since this must be done for every new snaxel. [3] lists 16 different cases that must be treated.
- **2**. PC_k may be traversed up to three times. Once in step 3, where it is constructed and twice in step 4, while searching for initial transition edges and during the determination of a new snake contour.
- **3**. Moving snaxels on diagonal edges has already been proved not to be a good option. The possible gain in precision to be obtained by considering is largely offset by the effort necessary to make them evolve.
- **4**. The use of a $K1$ triangulation allows curves with 4 snaxels in a square cell to be generated only if that curve does not cross the main diagonal of the cell. This makes the process orientation biased since there are curves which can be approximated by it but would lose that property if rotated.

From these observations a set of goals which must be pursued by a new T-snake schema can be established: (A) **No vertex burning** because this can be costly. (B) No triangulations or, more generally, **no orientation bias**. (C) All curves of a stage must be traversed only once (**one turn** property). We will add that **no history** must be necessary — a stage should only process information obtained within the stage itself. This property makes it easier to refine the mesh during the process.

A first approach with those properties was introduced by Bischoff and Kobbeit in [1]. That schema reduces the time step thus allowing the Transformed Curve to inherit the simplicity of snake S_k. The price of such a simplification is paid mainly by slowing the snake evolution and thus requiring a large number of iterations.

3. Theoretical background

We propose a method which repeats steps 1 and 2 of the algorithm presented earlier except that a vertex of PC_k is computed as soon as the segment of TC_k containing it becomes available. Also, our method does not use a triangulated mesh.

As the focus of this work is the control of a snake topology, we consider that the physical displacement of a snaxel is computed by a "black-box". We only assume, as is usual in the context of T-snakes, that physical displacements have amplitudes which are smaller than the edge length—d—of cells in the *AS*. Also, hereafter, μ will refer to the mesh containing these cells.

We call a μ-curve any polygonal line such that: (a) Its vertices are the points where it intersects the edges of μ. (b) No vertex of the curve coincides with any vertex of μ. A regular μ-curve is one which is simple and has a single vertex on a mesh edge. Since T-snakes must be regular μ-curves, so are the loops obtained by the approach given here. Two μ-curves crossing the same sequence of mesh edges are said to be *equivalent* .

To represent a μ-curve $S = [s_i; i = 0, \ldots, I-1]$ we use the *Cell—Edge of the cell—Point of the edge (CEP)* system where each s_i is represented by: (a) The **cell coordinate**—C_i —indicating the μ-cell containing $[s_i, s_{i+1}]$. (b) The **edge coordinate** $E(s_i)$ indicating which of the four edges of C_i contains s_i(the left, top, right and bottom edges of C_i are represented by 0, 1, 2 and 3, respectively). (c) $p(s_i)$, the distance between s_i and its out-vertex expressed in pixels. The **out-vertex** of s_i, termed v_i is the vertex of the μ-edge containing s_i which is outside the region delimited by S (the in-vertex of s_i is defined analogously). Also, define $E(s_i)^{-1}$, similarly to $E(s_i)$, as the code of the edge where s_i is on as an edge of the adjacent cell $C(s_{i-1})$. See Figure 2 below.

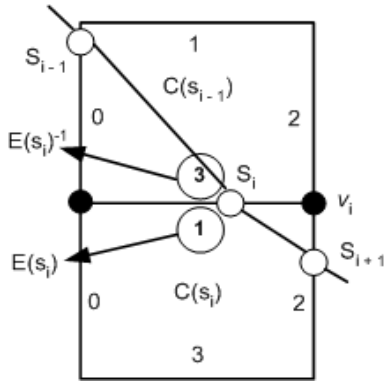

Figure 2. Elements used in the CEP system.

The reason for adopting the CEP-system is that it makes loop-processing more straightforward than using plain line-column coordinates. In fact C_i and $E(s_i)$ have to be computed anyway, no matter the system used.

The *μ-Dilation* of *S—$\mu D(S)$*—is the curve obtained by replacing every s_i by $w_i = v_i + \epsilon(s_i - v_i)$, where ϵ is a small positive number. *$\mu D(S)$* is a curve, μ-equivalent to S, which passes very close to the out-vertices of $s_i, i = 0, \ldots, I-1$.

Given two μ-curves, S and R , let $C_1, \ldots, C_k$ be a maximal sequence of mesh cells intersected by consecutive edges of both curves. We say that *S μ-intersects R* within $C_1, \ldots, C_k$ if any μ-curve equivalent to S crosses R within one of those cells. This concept is illustrated in Figure 3

The existence or not of a **μ-intersection** within the sequence $C_1, \ldots, C_k$ can be determined by computing the product of three functions taking values in {-1,1}which can be tabulated: $X_{ini} : \{0,1,2,3\}^3 \rightarrow \{-1,1\}$ depending on the three edges of C_1 crossed by the μ-curves,

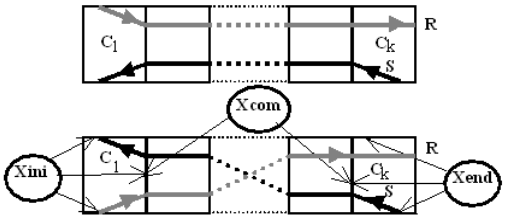

Figure 3. (a) Two μ-curves which do not μ-intersect and (b) A μ-intersection and the three functions used to detect it.

$X_{com} : \{0,1,2,3\}^2 \rightarrow \{-1,1\}$, which depends on the first and last μ-edges crossed by both curves within the sequence of cells, and $X_{end} : \{0,1,2,3\}^3 \rightarrow \{-1,1\}$, which is like X_{ini} for C_n.

We call an elementary mapping (or *e-map*) a transformation between curves obtained at the same stage k of the T-snake evolution. In this work, three e-maps will be used: Two e-maps take the initial snake of the stage S_k onto TC_k and onto PC_k. These will be termed T_k and P_k, respectively. E-map Γ_k takes the μ-dilation of S_k onto PC_k.

T_k can be easily defined due to the correspondence between the vertices of S_k and TC_k. On the other hand, there is no natural definition for P_k and Γ_k. The first objective is to make them continuous order-preserving piecewise linear mappings which associate points close to each other. To establish this last requirement in a more formal way the concept of a μ-bounded e-map is introduced below.

Let f be a facet of μ (i.e, a vertex, edge or cell). Call *N(f)* the union of all cells which are adjacent to the vertices of f. An e-map M is said to be *μ-bounded* if for every $s \in f$, $M(s)$ is in the interior of *N(f)*. Of course, if the physical displacement of all snaxels is less than d, then T_k will be μ-bounded, and in that case, we may enforce this property on P_k and Γ_k, too.

The *ray* of a mapping M at point s is defined as the open segment delimited by s and *M(s)*. If $e_i = [s_i, s_{i+1}]$ is an edge of the domain of M, the points $s_i, s_{i+1}, M(s_i) and M(s_{i+1}$ define the *sweeping quadrilateral* of M generated by e_i (Q_i).

A contracting μ-bounded e-map is said to be *adequate* if every one of its rays r_i is interior to the union of the sweeping quadrilaterals Q_{i-1} and Q_i, adjacent to it. Figure 4 illustrates this concept.

If P_k is adequate, then it generates no reverse sweeping quadrilaterals and thus all of its rays will be interior to the strip swept by the snake of stage k. This means that the border of the visited set is completely contained in PC_k. If Γ_k is adequate, each connected component of that border is μ-equivalent to a loop of PC_k, which is sufficient for our purposes. Also, a series of results, given in Section 5 can be explored to label these loops.

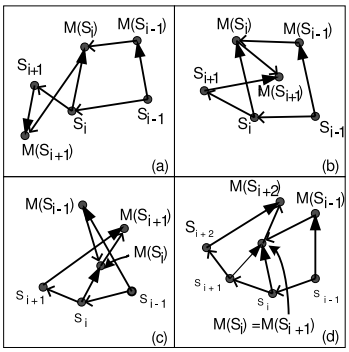

Figure 4. Non-Adequate ((a)-(b))and Adequate Mappings ((c)-(d)).

4. Making the Mapping Γ_k Adequate

Making P_k adequate can be too restrictive since the ray of P_k at a given snaxel must be constrained to the cone determined by the internal angle of S_k at that snaxel. This can be too strong a restriction if that angle is small.

Making Γ_k adequate avoids the case where a snake revisits a cell it has already totally swept and, as the internal angles of $\mu D(S_k)$ have at least 90°, the rays will not be too constrained. Moreover, this new objective requires considerably less computation. It can be achieved by acting in two moments. First, when physical displacements are computed, we avoid moving a snaxel S_i across the line orthogonal to its edge which passes by its out-vertex If Δ_i is the coordinate of the displacement applied to a snaxel s_i in the direction (horizontal or vertical) of its edge, prevent that possibility by doing:

if $E(s_i) > 1$ **then** $\Delta_i = \min\left(\Delta_i, p(s_i)\right)$
else $\Delta_i = \max\left(\Delta_i - p(s_i)\right)$.

There are no additional restrictions to be imposed on the movement in the other direction.

The second intervention is performed when a vertex of PC_k is generated. It aims at avoiding two undesirable configurations composed of S_k snaxels and PC_k vertices. In the first configuration, depicted in Figure 5(A), PC_k crosses $\mu D(S_k)$ which makes Γ_k not contracting. That configuration is characterized by: (1) Snaxels s_{i-1} and s_i, contained in a cell C_1 and having the same out-vertex v_i; (2) the intersection of $[T_k(s_{i-1}), T_k(s_i)]$ with cell *C2*, diagonally opposed to *C1* in relation to v_i, is the segment delimited by the PC_k vertices, s'_{j-1} and s'_j, both lying on edges adjacent to v_i, and (3) s_i and s'_j must belong to the same cell C. If such a configuration is detected, the intersection between PC_k and $\mu D(S_k)$ is eliminated by replacing s'_{j-1} and s'_j by s_{i-1} and s_i, respectively.

The second configuration to be avoided concerns the existence of reverse sweeping quadrilaterals. It is shown in

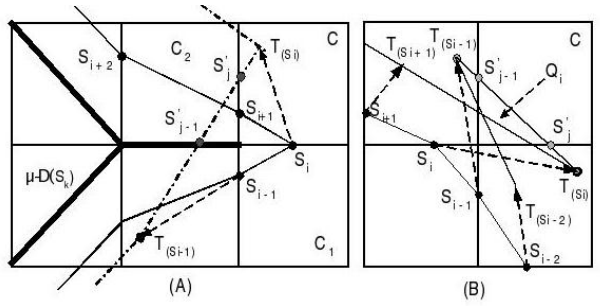

Figure 5. Undesirable configurations.

Figure 5(B), where snaxels s_i and s_{i+1} lay on edges which are external to a cell C, but are adjacent to the same vertex of C, which is cut into two parts by $\left[T_k(s_{i-1}), T_k(s_i)\right]$. The two vertices of PC_k in C, s'_j and s'_{j+1}, lay on the same mesh lines as s_i and s_{i+1}, respectively, which is only possible if the quadrilateral Q_i is reverse. The correction, in this case, consists merely of replacing $T_k(s_i)$ by s'_j.

Both cases are simultaneously handled by the procedure given below. The following notation is used: *s'_cur* and *s'_prev* refer to the two most recently obtained PC_k vertices; *i_cur* and *i_prev* denote the indices of the two consecutive snaxels such that $s'\_cur \in [T_k(s_{i\_prev}), T_k(s_{i\_cur})]$. Also, s_{i_cur} and s_{i_prev} are denoted by *s_cur* and *s_prev*, respectively and *t-cur* is the current vertex of TC_k.

Procedure ***Making*** Γ_k ***Adequate***
if *i_cur* = *i_prev* **then**
  if $E(s'\_cur) = \left(E(s\_cur)\right)^{-1}$ **then**
    $t\_cur \leftarrow s'\_cur$;
  else
    if $E(s'\_cur) = E(s\_prev)$ **and** $C(s'\_cur) = C(s\_cur)$ **then**
      $s'\_prev \leftarrow s\_prev$; $s'\_cur \leftarrow s\_cur$

It should be noted that this procedure embodies all that must be done when computing a new vertex of PC_k in order to ensure that Γ_k is adequate. We consider remarkable that, in spite of the many complicate cases determined by sequences of consecutive reverse sweeping quadrilaterals or clusters of TC_k vertices in the same cell, the problem of making Γ_k adequate admits such a simple solution.

5. Loop-Trees and the Labeling Process

A *Loop-Tree* of a closed curve C with no multiple self-intersection points is a graph that can be obtained by the following process: choose a point s in C and a circulation D (either clockwise or counter-clockwise). Traverse C in that direction starting at s. Every time a point x is revisited create a node to represent the loop formed by the part of C between the two visits to x. Then, collapse that loop into x and continue the tour. After having completed it, for every loop L_1 which has been collapsed to a point of another loop L_2, create an oriented edge from the node of L_1 to the node of L_2 (See Figure 6).

Loop-Trees of different topologies or with the same topology, but with different loop-node associations can be obtained for the same curve depending on the initial point taken and the circulation used in traversing the curve. Fortunately, any problems originated from this fact are avoided if Γ_k is adequate, as can be derived from Lemma 5.1 below.

Lemma 5.1. *If k is an adequate mapping then (A) every open loop of PC_k is disjoint from other loops found in the process, and (B) any loop in a sub-tree rooted at an open loop L will be disjoint from those in the other sub-trees rooted at L and also from the ancestors of L in the tree.*

So, if Γ_k is adequate, item (A) leads us to conclude that the regions enclosed by the open loops of PC_k will be totally unexplored. In view of that, these loops can be made the T-snakes of stage k+1. Also, (A) and (B) together imply that these loops are represented in any Loop-Tree of PC_k. This means that these new T-snakes are independent of both the initial point from which PC_k is traversed and the circulation used for that.

With respect to the labeling process, if Γ_k is an adequate mapping, then the following results regarding the nodes of a PC_k Loop-Tree can be applied:

Lemma 5.2. *The parent of an open node is a closed node. A closed node, however, can have both open and closed parents.*

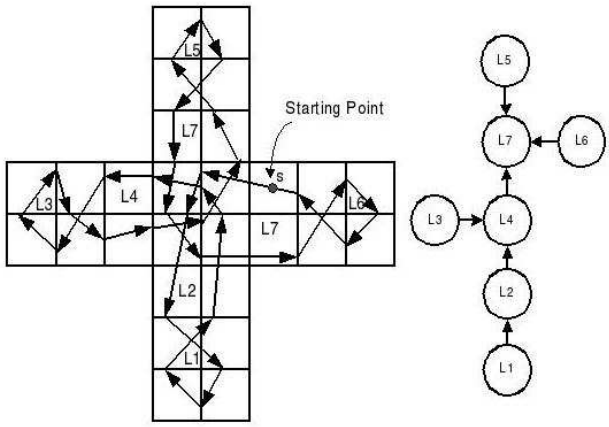

Figure 6. A curve and its Loop-Tree.

Lemma 5.3. *If both parent and child are closed they must intersect each other.*

These results allow us to label a loop by examining the label of its children in the Loop-Tree. Lemma 5.2 can be

used to label as *closed* every node having an *open* child. A loop having only *closed* children, will be *open* , if it is disjoint from its children (Lemma 5.1), or closed, if it intersects those children (Lemma 5.3). To start the process, however, we must devise a method for labeling the leaves. Even if Γ_k is adequate, it is not possible to get the label of a PC_k Loop-Tree leaf if we focus only on the curve itself. It is necessary to analyze the neighborhood of a snaxel in the loop. The information necessary for that analysis is provided by the Auxiliary Structure and the following result can be applied if the leaf is a proper loop of PC_k.

Lemma 5.4. *Let s be any snaxel of a leaf L which is a proper loop of PC_k. Then, L is a closed loop if all cells adjacent to the out-vertex of s are crossed by it.*

It remains to consider the case where the whole PC_k is a simple loop. The simplest alternative in this case is to check whether the in-vertex of s'_0 has been swept by the snake.

6. Revisiting a cell

If a cell is revisited, either it will become a double one or a loop has been formed. To describe the actions taken by the algorithm to identify a double cell or a loop and to make operational the results of Section 5 with respect to labeling a loop, some previous considerations are necessary.

(A) The *Loop_in_Construction—LiC—*is the polygonal line defined by the PC_k vertices already determined which neither belong to the loops already found nor have been eliminated for laying on the same edge as a neighbor one. The *LiC* can be considered as the root of the Loop-Tree of the part of PC_k already determined.

(B) To use the results of Section 5 it is not necessary to build the whole Loop_Tree of PC_k. It suffices to know the children of the *LiC* and their labels. For that reason, at the creation of a loop, the indexes of its initial and final vertices are pushed onto a stack relative to the loops with its label—either the Open or the Closed stack. When the parent of the loop is determined, and it is a child of the *LiC* no more, if those indexes are still there, they can be popped out.

(C) To make it possible to obtain the so-called, "one turn" property, it is necessary to have a direct link between the initial and final vertices of a loop. For that, the element of *AS* associated to a cell *C* stores a code identifying *Last_vertex(C)*, also noted S_c, the most recently found vertex of a projected curve which is in *C*. That code consists of: (1) the stage at which s_c was created and (2) a record containing both the CEP-coordinates of s_c and links (*.prv and .nxt*) to the records of its antecessor and successor in the last loop containing it or in the current *LiC* . If (1) is not the current stage then (2) is of no use.

(D) A loop just found is valid if it is totally contained in the *LiC* , while a non-valid one shares part of its contour with a loop found before and must not be added to the loop tree. The algorithm distinguishes between valid and non-valid loops in the following way: When PC_k revisits a cell *C*, all elements of the Open and Closed stacks greater than the index of *Last_vertex(C)* are popped out. If a new loop is formed when PC_k returns to *C* and the number of remaining elements in each stack is even, then that loop will be valid. Otherwise, it will be non-valid. For simplicity, henceforth we will assume that all loops formed are valid.

As loops must be regular μ-curves, if the *LiC* returns to an edge *e*, it is considered that a new loop has been formed. Other loops are determined by the knots of PC_k. These loops are identified by a configuration where a cell has two consecutive vertices of PC_k on its horizontal edges and other two consecutive ones on its vertical edges.

The fact that the characterization of one type of loop is related to an edge and the other to a cell determines that they are treated in different ways. Let us start with the case where PC_k revisits an edge.

A typical configuration in this case is depicted in Figure 7 A. $s_{j-m}, \ldots, s_j, s_i, \ldots, s_{i+m}$ with $j < i$ are *LiC* vertices which form what is called a **bottleneck**. That configuration is characterized by: (A) s_{i+n} and s_{j-n} are on the same edge, for n = 0,..., m. (B) If the *LiC* enters a cell C_n at s_{i+n} then, at the moment that vertex is generated, *Last_ vertex* $(C_n) = s_{j-n}$.

When s_i is generated, the loop L_0 , delimited by s_{j+1} and s_{i-1}, is identified. Also, any vertex of the bottleneck is discarded for laying on the same edge of another *LiC* vertex. However, the treatment of a bottleneck is not restricted to that because for labeling the loop, yet to be formed, containing s_{i+m+1}, it is necessary to check whether PC_k has a μ-self-intersection within the cells containing vertices of the bottleneck. If that intersection does not exist the label of that loop must be that of L_0 and both must be associated to a same Loop-Tree node. They are, in fact, parts of a larger loop which is subdivided into components which are regular μ-curves. On the other hand, if the μ-self-intersection exists L_0 becomes a child of the *LiC* and its extremities must be pushed onto the stack of its label. The existence of a μ-self-intersection is checked by using the tabulated functions X_{ini}, X_{com} and X_{end} introduced in Section 3. Thus, the complete treatment of a bottleneck can be described as follows: (A) At the beginning of the bottleneck: Let $E_{ini} = E(s_i)$, evaluate X_{ini} and find the label of L_0. (B) At the end of the bottleneck: Evaluate X_{end} and X_{com} which depends on E_{ini}, and $E(s_{j-n})$. If the product $X_{ini}.X_{com}.X_{end}$ indicates that a μ-self-intersection exists, then s_{j-n} and s_{i+m} must be pushed onto the stack of the label of L_0. In any case, s_{j-n-1} and s_{i+m+1} must be linked to maintain the *LiC* connected. (C) For concision sake, the unusual case where there is a loop between two consecutive elements of $s_{j-m}, \ldots, s_j$ will not be considered here. In view of that, no action is re-

quired at the non-extreme vertices of the bottleneck other than updating the ***Link_vertex*** . That vertex must be linked to the next vertex to be generated, if the bottleneck finishes at the current one, in order to maintain the *LiC* connected.

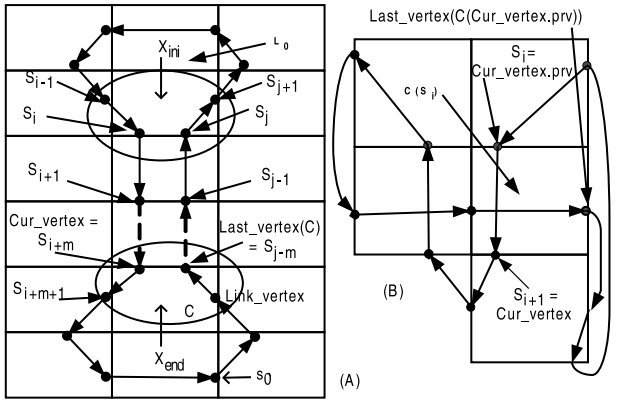

Figure 7. A bottleneck(7A) and a knot loop(7B).

In the case of a bottleneck, the repeated edge of a visited cell C is always that where *Last_vertex(C)* is on. It remains to consider the case where PC_k returns to the edge of *Last_vertex(C).prev* . In that second case, bottlenecks cannot occur and there is always a μ-intersection between the loop formed and the *LiC* . Thus, that loop must be represented in one of the stacks. Eventually, if that second case determines the end of a bottleneck, the specific treatment for that situation, given above, must be applied. In Procedure Repeated_Edge below, all steps described above for treating the two cases are put together.

Procedure Repeated_Edge
if $E(Cur\_vertex) = E(Last\_vertex(C))^{-1}$ **then**:
  if *Cur_ vertex.prv* is not in a bottleneck **then**:
    Loop(Last_vertex(C).nxt,Cur_ vertex.prv)
    Apply the Begining of a Bottleneck treatment
  else *Link_vertex* ← *Last_vertex(C).prv*
else if *E(Cur_vertex) = E(Last_vertex(C).prv)* **then**:
  if *Cur_vertex.prv* is in a bottleneck **then**:
    Apply the End of a Bottleneck treatment
  Loop(Last_vertex(C).prv, Cur_vertex.prv)

Now, assume that the *LiC* reenters an already visited cell C by crossing a non repeated edge - e_i - at vertex s_i (See Figure 7B). When the next vertex - s_{i+1} - is determined, there are three possibilities: (A) If s_{i+1} is on a repeated edge, a loop will be created by the routine Repeated_edge above. (B) If s_{i+1} is also on a non-repeated edge with the same orientation of e_i, $C \cap LiC$ will consist of two crossing segments with extremities on μ-edges of the same direction. In that case a knot loop must be generated. (C) If s_i and s_{i+1} are on non-repeated edges of different directions, C will become a double cell. Thus, at the creation of a second consecutive vertex in a non-repeated edge (like s_{i+1}) either case (B) or case (C) must be treated. At the creation of the first of these vertices (like s_i), however, the only action that may be necessary is to apply the End of Bottleneck treatment. All these steps are summarized in the following procedure.

Procedure Non_Repeated_Edge
if *Cur_vertex.prev* is also in a non repeated edge **then**:
  if *E*(*Cur_vertex*) = *E*(*Cur_vertex.prv*) **then**:
    Loop(Last_vertex(C(Curr_vertex_prev)), Cur_vertex.prv)
  else the cell has become a double one
else if *Cur_vertex.prev* is in a bottleneck **then**:
    Apply the End of a Bottleneck treatment

Finally, a loop can be labeled and processed by the procedure Loop below.

Procedure Loop(*Ini_vertex,End_vertex*)
if *Ini_vertex* > *Last_loop_end* **then** *Leaf_Label(End_vertex)*
else if *Ini_vertex* < *Last_intersection*
    or *Ini_vertex* < *Top*(*Open_Stack*) **then** *Label* ← *CLOSED*
else *Label* ← *OPEN*
Last_loop_end ← *End_vertex*
Remove the elements > *Ini_vertex* from the two stacks
Push *Ini_vertex* and *End_vertex* onto (Label)-Stack
Link *End_vertex.nxt* to *Ini_vertex.prv*

A leaf L of the Loop-Tree can be identified by the fact that its initial vertex has been created after any vertex in a loop already found. Its label must be determined by one of the schemes exposed in Section 5. Now, let us consider the way a non-leaf loop L is labeled. If a new loop intersects another already found then, by Lemma 5.3, it must be closed. That intersection exists if the loop initiates before the last intersection of the *LiC* with one of its children. If this is not the case, one may try to label L by using Lemma 5.2 which establishes that any loop having an open child is closed. An open child exists iff the initial vertex of L, *Ini_vertex* , has been generated before the one on the top of the Open-Stack. If L has not been labeled up to this point, then it must be open , since a non-leaf closed loop either has an open child or intersects its children. Besides labeling the new loop, it is necessary to update the stacks and extend the *LiC* .

7. Statistics and Examples of Segmentation

A program for evolving Loop-Snakes has been implemented and applied to a series of test examples. The statistics obtained from these validation tests are the best argument in favor of the approach introduced here. For evaluating the overall computational effort required by that approach, the snaxels have been classified according to the number of operations, with regard to the topology control, that must be performed when they are processed. Here, for simplicity, they will be grouped in the three following classes: (A) snaxels s_i such that $[T_k(s_{i-1}), T_k(s_i)]$ cuts a cell into two parts. (B) snaxels at which the procedure to make

Γ_k adequate takes a corrective action . (C) Snaxels s_i such that $[T_k(s_{i-1}), T_k(s_i)]$ intersects an already visited cell. The number of snaxels in each of these classes was computed for images of four different types - Synthetized images (I), Noisy images (II), Images with many segments (III) and Images with cells on a textured background (IV). The results obtained are presented in Table 1. It can be observed

	Images I	Images II	Images III	Images IV
Group A	142.605	937.837	360.687	425.618
Group B	1.806	95.357	22.453	44.866
Group C	6.359	8.540	12.541	3.808
Overall	557.282	2.847.313	1.183.100	1.212.631

Table 1. The number of snaxels computed.

that the number of most costly snaxels, that is, those determining that PC_k revisits a cell (Group C), is considerably small compared with the total number of snaxels, not reaching 1.2%. Also, the snaxels in group A are no more than 36%, of the total which means that for the other, at least 64% of them, the only action specific of this approach is testing whether *i_cur = i_prev*, since limiting the displacement of the snaxels and checking and updating the *AS* are common to all approaches. Moreover, for the snaxels not in B, at least 96% of the overall number , that action is limited to 4 tests. This means that: (1) the number of snaxels requiring additional work is a negligible fraction of the total and (2) for those 96% of the snaxels, the performance of Loop-Snakes is clearly hard to beat. In fact, the numbers above attest the adequacy of the strategy employed in this work: spend minimum effort when processing a plain snaxel and delay all the complication to the moment a loop may be found.

Two examples where the method has been applied are illustrated in figures 8 and 9. The first one shows the ability of ignoring small artifacts present in the background. The second is an example where the T-snake splits multiple times. The solutions indicated in both cases, have been obtained without refining the mesh or applying any post-processing.

8. Conclusions and Future Work

A series of theoretical results (Sections 3 - 5) had to be developed to support the Loop-Snakes approach. Based on those results it was possible to create a methodology satisfying the four desired properties indicated in Section 2 and having a clear computational gain when compared to the existing methods, as seen in Section 7.

With respect to future works , the possibility of expanding the results obtained here for T-surfaces evolving in 3D-images can be considered in the following context: If Γ_k is adequate, the work of burning the mesh vertices swept by the T-snake can be reduced to label as visited the out-vertex of every PC_k vertex generated. This property can be extended to T-Surfaces and give origin to a faster method. There are also 2D variants of the method yet to be explored. The most promising one does not even require that the projected curve be determined.

References

[1] Bischoff and L. Kobbeit. Snakes with topology control. *The Visual Computer*.

[2] A. W. M. Kass and D. Terzopoulos. Snakes:active contour models. *The Visual Computer*.

[3] T. McInerney and D.Terzopoulos. Topologically adaptable snakes. *Proc. of Int. Conf on Computer Vision*, pages 840–845, 1995.

[4] A. Oliveira and S. Ribeiro. The loop snakes page: http://ganimede.lcg.ufrj.br/projetos/loopsnakes. 2004.

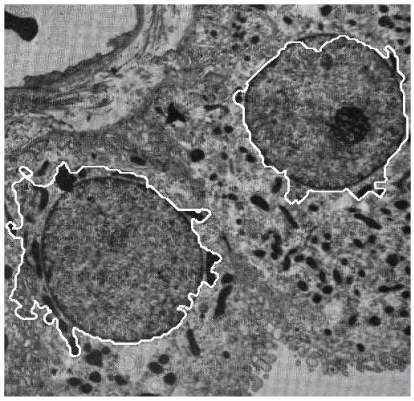

Figure 8. An image with two cells.

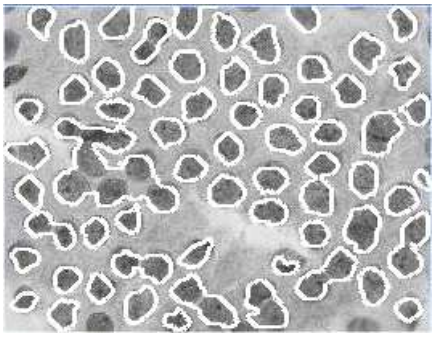

Figure 9. Evolution with multiple splits.