SIMPLIFICANDO O CONTROLE TOPOLÓGICO DE UMA T-SNAKES

Vitor Vasconcelos Araújo Silva

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Antônio Alberto Fernandes Oliveira, D.Sc

Prof. Ailton Fernando Dias, Docteur

Prof. Gilson Antônio Giraldi, D.Sc.

Prof. Paulo Roma Cavalcanti, D.Sc.

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 2006

SILVA, VITOR VASCONCELOS ARAÚJO

T-Snakes, Simplificando o Controle Topológico de uma [Rio de Janeiro] 2006

IX, 89 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2006)

Dissertação – Universidade Federal do Rio de Janeiro, COPPE

1. Loop-trees. 2. Curvas transformadas.

3. Curvas projetadas. 4. Mapeamentos elementares.

I. COPPE/UFRJ II. Título (série)

*Ao pai, mãe e bro,*
*com amor.*

# Agradecimentos

Este trabalho teve muitos incentivadores e colaboradores. Tentarei agradecer em ordem cronológica, começando pelo amigo Lúcio França - que hoje está trabalhando num lugar mais distante - por plantar a semente do mestrado. Aos amigos do querido VERLAB/UFMG, pelos primeiros contatos com a pesquisa e pelos excelentes momentos de aprendizado.

Aos meus novos irmãos, mas nem tão novos, Marquinhos Ferida e Dário, por toda a convivência, amizade e companheirismo durante boa parte dessa caminhada e pelo que certamente continua pela vida afora.

Ao Ailton, por me permitir agarrar minha oportunidade.

Aos amigos da CNEN, e aqui devo respeitosamente separá-los: Sérgio e Léo. Obrigado pelo aprendizado através convivência, pela referência profissional que todo garoto precisa. Aos "desagregadores", Mr. Emerson, André Januário e Paulo Leal, valeu pela amizade.

Ao Rio de Janeiro, pela Lagoa, pela Lapa, por Ipanema, pelo Bracarense, pelos Chopps, pela Tijuca, pelo Salgueiro, pelo Osvaldo, pela vista da praia de Botafogo, pela Cinelância, por Santa Tereza (que maravilha!), pelo Democráticos, pela Mauá. Ah, Escravos da Mauá, obrigado pela inundação de alegria. Como é bom esse Rio.

Aos amigos do LCG: Ricardo, Fábio Franco, Saulo, Karlitos, Bubu, Disney, Yalmar, Guina, André, Caíque. Colegas de trabalho e de diversão também. Aos amigos, que não são do LCG - como se amigos precisassem de carta de referência

- Marquinhos e Leandro.

Ao irmão Ricardo - não coincidentemente o mesmo amigo do LCG - pela grande amizade, viagens e bons momentos. Tanto de farra como de responsabilidade. Obrigado pela acolhida num seleto grupo de amigos antigos. Aos meus pais adotivos, Dona Ana e Seu Murilo.

À (Ah!) Dani. Pela paciência quase infinita, pelo bom-humor e por todo carinho e amor. Sempre vão faltar palavras pra te agradecer.

Aos amigos de BH, claro! Impossível citar nomes. Obrigado por não deixarem a distância fazer diferença.

Finalmente, à família - presta atenção que essa é pra senhora, Vó! - a família mais divertida, alegre que qualquer um poderia ter inventado. A todos os almoços de domingo já com saudades de ter que ir, aos abraços e beijos das tias e primos, à República - depois da tese começo a redigir nossa Constituição, Tio André.

À Moeda, minha outra Cidade Maravilhosa.

Às viagens, que me mantém sempre querendo ir mais longe - literamente.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

SIMPLIFICANDO O CONTROLE TOPOLÓGICO DE UMA T-SNAKES

Vitor Vasconcelos Araújo Silva

Março de 2006

Orientador: Antônio Alberto Fernandes Oliveira

Programa: Engenharia de Sistemas e Computação

Snakes topologicamente adaptáveis, ou simplesmente T-snakes, são uma ferramenta padrão para identificação automática de segmentos em uma imagem. O modelo Loop-snakes é uma nova abordagem de controle da topologia de uma T-snake. Neste trabalho, uma versão do modelo com maior eficiência computacional é apresentado em detalhes. Esta abordagem é focada nos *loops* formados pela, assim chamada curva projetada, que é obtida a cada estágio da evolução da T-snake. A idéia é tornar esta curva a imagem de uma mapeamento linear por partes de uma classe adequada. Com a ajuda de uma estrutura adicional, a Loop-Tree, é possível decidir em complexidade $O(1)$ se a região delimitada por um desses loops já foi ou não explorada pela snake. Isto torna possível construir um algoritmo ótimo para implementar o processo de evolução de uma T-snakes, cujo desempenho é demonstrado, também no trabalho, por meio de estatísticas e de uma série de exemplos.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

SIMPLIFYING THE TOPOLOGYCAL CONTROL OF A T-SNAKES

Vitor Vasconcelos Araújo Silva

March 2006

Advisor: Antônio Alberto Fernandes Oliveira

Department: Computing and Systems Engineering

Topollogically adaptable snakes, or simply T-snakes, are a standard tool for automatically identifying multiple segments in an image. The Loop-snakes model is a novel approach for controlling the topology of a T-snake. In this work, a version of that model with improved computational efficiency is described in details. That approach 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. Using 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 allowing to construct an enhanced algorithm for evolving T-snakes whose performance is assessed by means of statistics and examples.

# Sumário

# Lista de Figuras

# Capítulo 1

# Introdução

Modelos de Contornos Ativos, também chamados *Snakes*, são modelos deformáveis propostos por [11] que têm sido aplicados com sucesso numa grande variedade de problemas em Visão Computacional e Análise de Imagens [3]. Sua formulação matemática torna simples integrar num único processo de extração [3, 9, 30] os dados da imagem, um contorno inicial estimado, propriedades do contorno desejado e restrições que ele deve satisfazer em função de um conhecimento a priori que se tem dele.

Os modelos de *Snakes* paramétricas consistem basicamente de uma curva (ou superfície) elástica que pode se deformar em resposta a forças internas (forças elásticas) e forças externas (forças da imagem e das restrições impostas). Estas forças podem ser obtidas por um processo de minimização global de uma dada função de energia ou determinadas com base em informações locais. [4, 27, 5]

Entretanto, uma limitação conhecida da maioria dos métodos de *Snakes* é que a topologia das estruturas de interesse deve ser conhecida a priori, uma vez que o modelo matemático não consegue lidar com mudanças topológicas sem mecanismos extras [18, 19]. Dentre os trabalhos propostos que lidam com essa limitação [18, 7, 12, 8, 2, 21], o modelo das *T-Snakes* tem a vantagem de ser de aplicação

geral. Uma *T-Snake* tem a capacidade de mudar sua topologia, tanto por subdivisões como por meio de junções, permitindo assim a identificação de mais de um segmento alvo na imagem. A idéia principal é projetar uma *Snake* paramétrica sobre uma triangulação na imagem e calcular a *Função Característica* que distingue o grupo de vértices interiores à *Snake* dos que estão de fora.

A matriz que contém o valor desta função característica para cada nó constitui, no caso do modelo original das *T-Snakes*, a chamada *Estrutura de controle topológico (Topologycal Control Structure - TCS.* A TCS é uma matriz contendo a informação usada para associar *snaxels* que são próximos em 2D mas distantes ao longo da *Snake*. Em outros trabalhos, os elementos da TCS são relacionados às arestas [2] ou células [21] de uma malha quadrada cobrindo o domínio da imagem.

Foi proposto em [21] o modelo original de *Loop Snakes*. Nesta dissertação, é apresentada uma variação desse modelo original, com diversos aprimoramentos computacionais. Este modelo mantém a idéia de partição do domínio, mas ao invés de utilizar uma função característica, as mudanças topológicas são feitas baseadas nos *loops* formados pelas curvas que são obtidas no processo de atualização das *Snakes*. Por essa razão, as *Snakes* obtidas por esse modelo foram batizadas *Loop-Snakes*.

A motivação original no nosso trabalho era lidar com algumas dificuldades conhecidas do modelo original de *T-Snakes*, que é descrito em detalhes no capítulo 2 juntamente com outras procedimentos alternativos. Na seção 3.1 são descritos todos os detalhes necessários à formalização do modelo das *Loop-Snakes*. Neste modelo, $PC_k$ é considerada como a imagem de uma versão dilatada da *Snake* $S_k$ por um mapeamento linear $\Gamma_k$. Definindo $\Gamma_k$ na versão dilatada de $S_k$ torna mais simples satisfazer algumas condições locais que são necessárias para tornar a estratégia computacionalmente interessante. A seção 3.2 descreve tais condições e a seção 3.3 é dedicada a mostrar como elas podem ser implementadas de modo

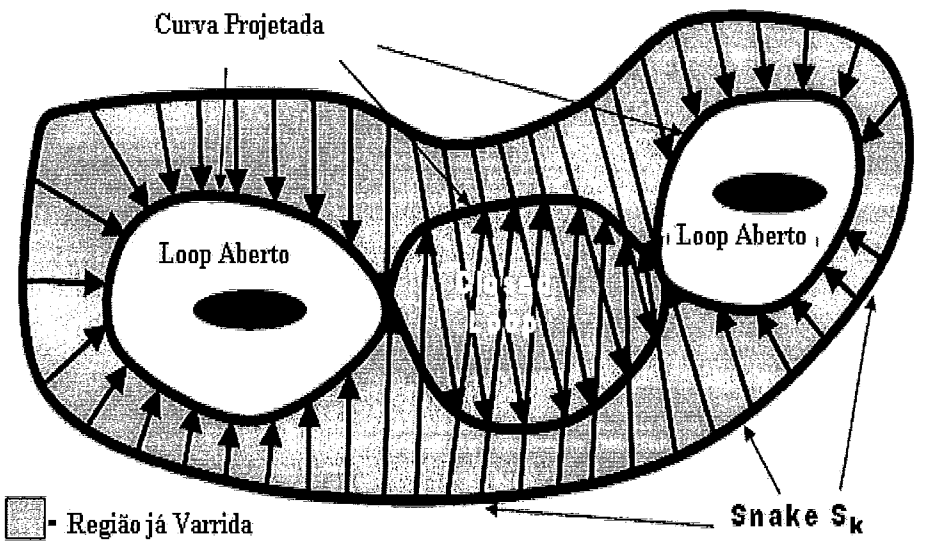


Figura 1.1: Curva projetada e seus *loops*

eficiente.

Para um *snaxel* de uma *Loop-Snake*, o processo é extremamente simples: deslocá-lo de acordo com o modelo físico, projetar um novo segmento da curva transformada na malha, verificar e atualizar a TCS. Eventualmente, uma ação corretiva deve ser executada para garantir que $\Gamma_k$ seja um mapeamento adequado. Quando $PC_k$ retorna a uma célula já visitada, duas possibilidades devem, então, ser exploradas: ou a célula C se torna uma *célula dupla*, ou seja, uma célula com um vértice da curva $PC_k$ em cada aresta, ou uma mudança topológica envolvendo as arestas de $PC_k$ contidas em C deve ser realizada. A implementação de tal mudança depende de $PC_k$ possuir uma auto-intersecção (*knot*) em C ou revisitar uma aresta de C. Em ambos os casos, um novo *loop* é criado do modo descrito na seção 4.1. Os resultados serão apresentados na seção 4.2, demonstrando que é possível inferir o rótulo do *loop* - aberto ou fechado - simplesmente examinando os rótulos dos

*loops* adjacentes encontrados anteriormente ou meramente analisando a posição dos vértices de $PC_k$ na célula C. Mesmo que $PC_k$ tenha um único *loop*, este pode ser corretamente rotulado em $O(1)$, apesar de um procedimento um pouco mais elaborado ser necessário neste caso. Na seção 4.3 são explicados os procedimentos para se encontrar e rotular os *loops*.

Revisitar uma célula, demanda um processo mais elaborado, mas isso ocorre muito raramente se comparado ao enorme número de *snaxels* gerados, como indicado nas estatísticas mostradas no capítulo 5. O capítulo 6 é destinado às conclusões e aos trabalhos futuros. Por motivo de concisão, é apresentado aqui apenas o caso onde as *Snakes* se contraem. *Snakes* que se expandem, entretanto, podem ser tratadas de maneira análoga. Como o foco deste trabalho é o controle da topologia da *Snake*, podemos considerar, em princípio, que o deslocamento físico dos snaxels é computado por uma "caixa-preta". Algumas alternativas para o cálculo destes deslocamentos, entretanto, como a descrita na seção 3.3, reduzem o número de operações necessárias para o controle da topologia.

Todas as propostas são de melhoramentos na metodologia apresentada em [24]. Apresentamos abaixo esquematicamente quais são eles:

- Uma nova maneira de computar o deslocamento dos snaxels elimina a existência de quadriláteros de varredura reversos, simplificando o controle da topologia;
- O registro das estruturas estreitas já encontradas simplifica o processo de rotulação;
- Uso de mecanismos para *by-passar* seqüências de snaxels que estão provisória/definitivamente imobilizadas;
- Nova maneira de identificar o rótulo de um *loop* que contém toda a curva projetada;

- Aprimoramento das rotinas que tratam o caso em que a curva projetada volta a uma dada célula em relação às utilizadas em [24].

Com essas modificações propostas no modelo das *Loop-Snakes*, é possível manter a qualidade da segmentação, e até mesmo melhorá-la, juntamente com uma diminuição do custo computacional, resultando num processo mais eficiente.

# Capítulo 2

# Trabalhos relacionados

Neste trabalho, focamos especialmente os modelos de *snakes* que são topologicamente adaptáveis - ou seja, *snakes* que podem mudar sua topologia durante a evolução. Por adaptabilidade topológica pode-se entender basicamente a habilidade de um contorno se dividir em duas ou mais *snakes* ou $N$ contornos se unirem num único. No campo das *snakes* implícitas, tais operações são inerentes ao modelo, devido à formulação matemática de dimensão maior que utilizam [23, 22, 25, 14].

Tal formulação consiste basicamente em considerar a *snake* como o nível zero de um funcional cuja definição varia de acordo com uma equação de movimento. Tais metodologias são bastante empregadas na recuperação de objetos com formas complexas e topologias desconhecidas. Entretanto, devido à sua formulação própria, os modelos implícitos não são convenientes como os modelos paramétricos, para análise de forma e visualização e, principalmente, para a interação com o usuário. Em relação ao custo computacional, mesmo se métodos do tipo "*narrow band*" são empregados, uma iteração de uma *snake* implícita consome mais tempo. Isto, pelo simples fato de que o funcional precisa ser atualizado tem todos os vértices de uma malha que pertençam ao "*narrow band*" para que a

nova curva de nível zero seja obtida por interpolação.

Nos modelos de *snakes* paramétricas, o controle da topologia requer que um maquinário extra seja acrescentado ao modelo. Em [29] é descrita uma metodologia na qual partículas são posicionadas na superfície de um objeto deformável imerso no domínio de uma imagem 3D até que sua densidade atinja um determinado limiar. O modelo utiliza forças inter-partículas para manter suave a superfície do objeto deformável durante o processo de evolução e as forças externas atraem as partículas para onde o gradiente de imagem é alto, ou seja, na borda dos objetos representados na imagem 3D. Esta técnica tem a vantagem de lidar tanto com superfícies abertas ou fechadas e é topologicamente adaptável. Entretanto, tem as desvantagens de ter alto custo computacional - $O(n^2)$, sendo $n$ o número de partículas - além de ser difícil encontrar bons locais para colocação inicial das partículas se pretendemos fazer isso de forma automática [15].

[8] apresenta uma metodologia para subdividir um contorno fechado em duas partes também fechadas. Isto é feito primeiramente com a construção do histograma da norma das forças da imagem ao longo da *snake*, de modo a identificar a região apropriada para cortá-la - a região onde o campo dessas forças é mais fraco. No próximo passo, o método identifica dois pontos nesta região para serem os pontos finais do segmento que cortará a curva em duas partes. O critério para se fazer isso é baseado na direção de uma força adicional - a força de área - que é usada para fazer o contorno se ajustar nas partes côncavas. Esta metodologia tem algumas desvantagens, como não tratar da junção de contornos e ainda possui um custo computacional considerável devido aos vários passos que exige.

Uma abordagem mais geral para incorporar mudanças topológicas ao modelo de *snakes* é através de um procedimento em que a *snake* é imersa em uma malha de células triangulares que particiona a imagem. Inspirados na filosofia dos *Level Sets*, **McInerney e Terzopoulos** propuseram uma abordagem mais geral para in-

corporar mudanças topológicas em modelos de *snakes* paramétricas, os quais eles chamaram *T-Snakes* [18, 19, 15].

A curva é reparametrizada fazendo-se com que suas interseções com as arestas de triangulação se tornem os novos vértices. Tal reparametrização é chamada *projeção da curva na estrutura particionadora.* As mudanças topológicas são possíveis através da utilização de uma *Função característica*, que distingue os nós da malha interiores à curva fechada dos nós exteriores a ele. O conjunto de arestas nas quais a Função característica muda de valor *(arestas de transição)* oferece uma forma simples de executar as mudanças topológicas. Este método é capaz de lidar tanto com *snakes* contráteis quanto expansíveis e sua extensão para modelos de superfícies deformáveis *(T-Surfaces)* é direta. Em [28], este modelo é empregado junto com técnicas de extração iso-superfícies [15] para gerar o modelo de T-Snakes duais [10].

Em relação ao modelo físico empregado, em [18] é proposta a evolução da *T-Snake* baseada em forças (suavizadas) de tensão ($B_i$), uma força (tipo *baloon*) normal ($F_i$) e uma força (baseada na imagem) externa ($f_i$) [18]. Estas forças são dadas respectivamente pelas seguintes expressões:

$$B_i = b_i \left( v_i - \frac{1}{2}(v_{i-1} + v_{i+1}) \right) \tag{2.1}$$

$$F_i = k_i \, sign_i \, n(i), \tag{2.2}$$

$$f_i = \gamma_i \|\nabla P\|, \tag{2.3}$$

onde $n_i$ é a normal ao *snaxel* $v_i$ e, $b_i$, $k_i$ e $\gamma_i$ são fatores de escala de força, $P = -\|\nabla I\|^2$, $sign_i = 1$, se $I(v_i) \leq T$ e $sign_i = 0$ caso contrário ($T$ é um limiar para a imagem $I$). Estatísticas regionais também podem ser usadas para definir $sign_i$ [18]. Assim, a atualização da posição da *T-Snake* segue a seguinte equação:

$$v_i^{t+\Delta t} = v_i^t + h_i \left( B_i^t + F_i^t + f_i^t \right), \tag{2.4}$$

onde $h_i$ é o passo de evolução.

No que tange ao controle da topologia, a evolução da *T-Snake* pode ser expressa pelo procedimento *Evolving a T-Snake* apresentado a seguir. Seja $S_k = [s_i, i = 0, \cdots, J]$ a *T-Snake* na iteração $k$. Os elementos da matriz TCS - que contém os valores da função característica nos vértices da malha - são todos inicializados como *não-visitados*. Além disso, façamos $\tau$ ser a triangulação $J_1$ dos vértices da malha, que é a obtida pelo corte de cada célula por sua diagonal principal. Uma aresta de transição será um vértice de $\tau$ ligando um nó visitado a um nó não-visitado. A figura 2.1 representa uma iteração do método. Por motivo de visibilidade, os *snaxels* de $S_k$ e os vértices de $PC_k$ nas arestas diagonais foram omitidos.

Procedimento ***Evolving a T-Snake***

1. Aplique a cada *snaxel* $s_i$, o movimento determinado pelo modelo físico empregado. Os pontos obtidos $(t_i, i = 0, \cdots, I)$ definem a curva transformada, $TC_k$.

2. Construir a curva projetada, $PC_k = [s'_j, j = 0, \cdots, J]$ fazendo a concatenação das arestas $[s'_j, s'_{j+1}]$, definidas por duas interseções sucessivas de $TC_k$ com arestas de $\tau$.

3. Para $i = 0, \cdots, I$, faça $Q_i$ seu quadrilátero definido por $s_{i-1}$, $s_i$, $t_i$ e $t_{i-1}$. Estes serão chamados *quadriláteros de varredura* de $S_k$. Verificar se $Q_i$ contém vértices da malha que permanecem não-visitados e modificar o elemento de TCS correspondente a eles para *visitado*.

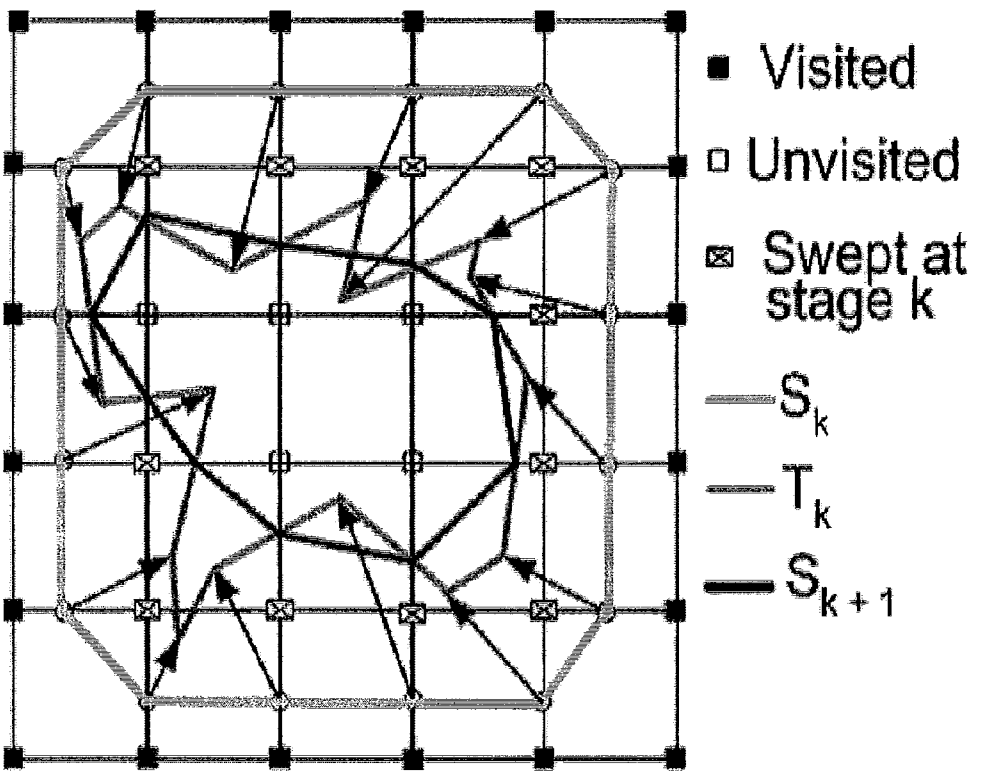


Figura 2.1: Uma iteração da *T-Snake*

4. Todas as arestas de transição são desmarcadas e $PC_k$ é percorrida. Toda vez que uma aresta de transição $e_0$ é encontrada, execute:

   (a) Faça $e = e_0$ e $t$ ser um dos triângulos adjacentes a $e_0$. Então, execute o seguinte:

      i. Marque $e$ e escolha um ponto $x_e$ dela, o que é feito considerando as posições dos vértices de $PC_k$ ou $S_k$ em $e_0$.

      ii. Substitua $e$ por $e'$, a outra aresta adjacente a $t$, e $t$ por $t'$, o outro triângulo adjacente a $e'$.

      iii. Repita i e ii até que $e = e_0$.

   (b) Assumir a linha poligonal fechada definida pelos pontos $x_e$ como a *snake* do estágio $k + 1$.

Em **4.a**, uma curva representando os limites de cada componente conexa do conjunto de pontos ainda não visitados pela *snake* é construída e dessa curva é e feita uma *snake* em **4.b**.

À despeito das propriedades das *T-Snakes/T-Surfaces*, algumas considerações são pertinentes:

1. Verificar se vértices não visitados são cobertos pelos quadriláteros de varredura $Q_i$ (**passo 3**) é um procedimento que consome tempo, uma vez que isto deve ser feito para cada novo *snaxel*. McInerney [17] lista 16 diferentes casos que devem ser considerados para fazer isso.

2. Mover *snaxels* em arestas diagonais já se provou não ser uma boa opção, uma vez que o ganho em precisão a ser obtido é fortemente ultrapassado pelo esforço em fazê-los evoluir.

3. O uso de uma triangulação $J_1$ permite que curvas com 4 *snaxels* numa célula quadrada sejam gerados somente se a curva não cortar a diagonal principal

da célula. Isto torna o processo viesado com respeito à orientação, uma vez que existem curvas que podem ser aproximadas por ele, mas que perderiam essa propriedade se rotacionadas.

A metodologia introduzida em [2] evita todos estes problemas e, ainda, tenta reduzir o quanto possível o esforço gasto no controle topológico. Isto é possível através das seguintes simplificações: a) Eliminar a necessidade de encontrar uma curva projetada, o *snaxel* fica restrito a movimentos somente ao longo da linha da malha onde está localizado. b) Evitar casos de topologia complicada, o *snaxel* não pode ir além do primeiro nó que ele encontra. Neste ponto, ele é explodido em três *snaxels*, um para cada aresta adjacente àquela de onde ele veio. Esta abordagem faz a Curva Transformada herdar a simplicidade da *snake* inicial. Uma mudança topológica será feita se, e somente se, uma aresta contém dois *snaxels* não sucessivos.

O preço de tal simplificação é pago de diferentes modos: complicação do cálculo dos deslocamentos, introdução de um claro peso em favor da malha de vértices e, principlamente, se torna mais lenta a evolução da *snake*, graças à limitação imposta ao deslocamento dos *snaxels*. A versão original propõe que o intervalo de tempo entre duas iterações seja diminuído e, assim, num caso genérico, um único nó é cruzado por iteração.

É óbvio que o deslocamento do *snaxel* deve ser limitado para não comprometer a convergência do processo. A idéia é não limitá-lo ainda mais somente para facilitar o controle topológico. Este último método foi generalizado para superfícies imersas em uma malha tetraedral [1].

# Capítulo 3

# Metodologia

## 3.1 Coordenadas CEP, *Loops* e Estruturas Estreitas

Este capítulo e o próximo são dedicados a descrever formalmente as *loop-snakes*. Este é especialmente dedicado a explicar precisamente o que é um *loop* no contexto deste trabalho e introduzir uma maneira de representar os vértices de $PC_k$ que facilita a detecção e rotulação dos *loops*.

Daqui em diante, $\mu$ vai se referir à malha contendo as células representadas na $TCS$. Chamamos de $\mu$-curva qualquer linha poligonal tal que: a) Seus vértices são os pontos nos quais ela intercepta as arestas de $\mu$. b) Nenhum vértice da curva coincide com qualquer nó de $\mu$. *Uma $\mu$-curva* é dita regular se ela for uma curva simples e tiver um único vértice numa aresta da malha. Uma *T-Snake*($S_k$) deve ser uma $\mu$-curva regular enquanto que a curva projetada $PC_k$ é simplesmente uma $\mu$-curva. Uma vez que as *T-Snakes* devem ser $\mu$-curvas regulares, serão também os *loops* obtidos pelo procedimento dado. Duas $\mu$-curvas cruzando a mesma sequência de arestas da malha são ditas *equivalentes*.

Para representar uma $\mu$-curva $S = [s_i, i = 0, \cdots, I-1]$ é suficiente poder representar pontos no interior relativo das arestas-$\mu$. Para isso, nós usamos o sistema

*Célula - Aresta da célula - Ponto da aresta* (*Cell, Edge e Pixel - CEP*, figura 3.1) no qual cada $s_i$ é representado por: a) A coordenada da célula $C(s_i)$ indicando a $\mu$-célula contendo $\overrightarrow{[s_i, s_{i+1}]}$; b) A coordenada da aresta $E(s_i)$ indicando qual das quatro arestas de $C(s_i)$ contém $s_i$ (as arestas da esquerda, superior, direita e inferior são numeradas 0, 1, 2 e 3, respectivamente); c) $p(s_i)$, a distância entre $s_i$ e seu nó à esquerda expressa em pixels. O *nó à esquerda* de $s_i$, chamado $v_i$ é o nó da $\mu$-aresta contendo $s_i$ que está a esquerda do segmento orientado $\overrightarrow{[s_{i-1}, s_i]}$ (o *nó à direita* é definido analogamente). Para uma $\mu$-curva *regular* perçorrida no sentido horário, o nó à esquerda de $s_i$ está fora da curva e o nó à direita está dentro dela. Neste caso, eles serão chamados mais apropriadamente, *nó-externo* e *nó-interno* de $s_i$ respectivamente. Também definimos $E(s_i)^{-1}$ de maneira similar a $E(s_i)$, indicando o código de arestas referente à célula adjacente $C(s_{i-1})$.

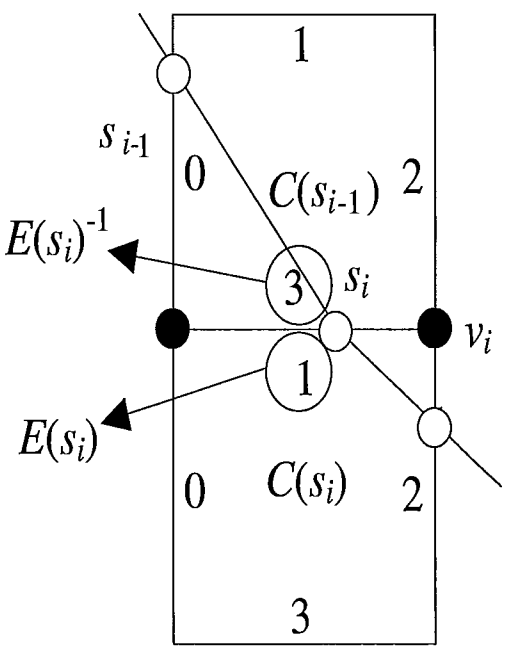


Figura 3.1: Coordenadas CEP de $S_i$

A razão para a adoção do sistema *CEP* é que este torna o processamento dos *loops* mais direto do que a utilização coordenadas cartesianas. De fato, $C_i$ e $E_i$ devem ser calculadas de qualquer maneira, não importa o sistema utilizado. Se $S$ é *regular*, a **$\mu$-Dilatação** de $S$ ($\mu D(S)$) é a curva obtida ao se substituir cada $s_i$ por $w_i = v_i + 0^+.(s_i - v_i)$. $w_i$ é uma abstração, sendo um ponto com coordenadas $v_i$ que pertence a uma aresta simples - que contém $s_i$. Utilizar $w_i$ ao invés de $v_i$ na definição de $\mu D(S)$ a torna uma $\mu$-curva regular. A figura 3.2 descreve uma $\mu$-curva regular e sua **$\mu$-Dilatação**. A **$\mu$-Contração** de $S$ é definida de forma similar.

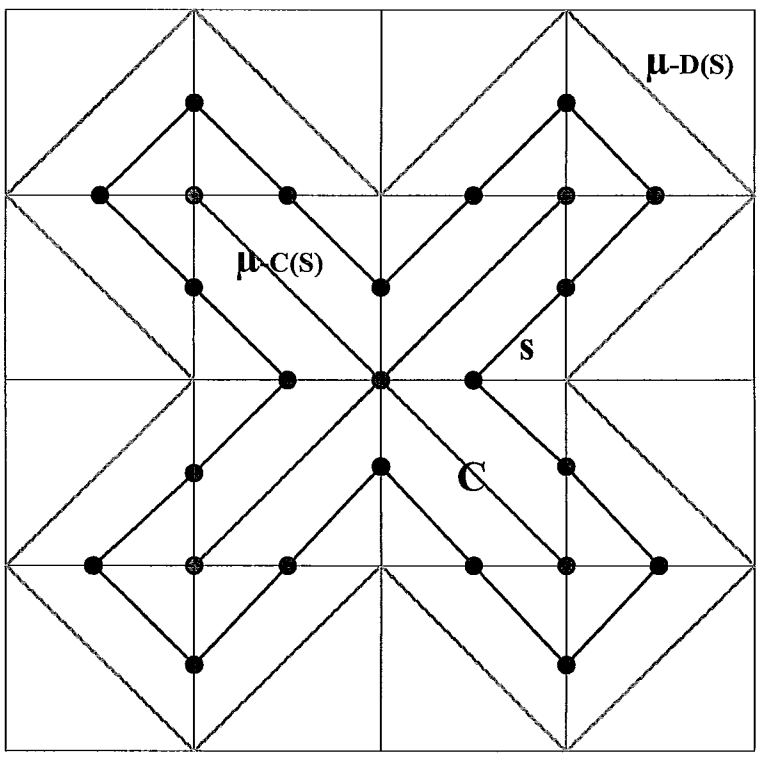


Figura 3.2: Uma $\mu$-curva $S$, sua $\mu$-dilatação ($\mu$-$D(S)$) e sua $\mu$-contração ($\mu$-$C(S)$)

Uma célula - como $C$ na figura 3.2 - que tenha um vértice da $\mu$-curva regular $S$ em cada uma de suas arestas é dita uma *célula dupla* de $S$. $S$ tem uma *cruz em* $C$ se ela tem dois pares de vértices não consecutivos - ($s_k$, $s_{k+1}$) e ($s_j$, $s_{j+1}$) - tais

que:

1. $s_k$ e $s_{k+1}$ estejam em arestas paralelas da célula $C$.

2. $s_j$ e $s_{j+1}$ estejam nas outras arestas da célula $C$.

Seja agora $S = [s_i, i = 0, \cdots, m]$ uma $\mu$-curva. No restante deste capítulo $W$ será uma linha poligonal formada por $n$ vértices de $S$, $w_i = s_{j,1}, \cdots, w_k = s_{j,k}, \cdots, w_n = s_{j,n}$, os quais satisfazem a seguinte propriedade:

"Atravessando $S$ de $w_1$ para $w_n$ na direção dada por $(s_i)_0^m$, os vértices $w_k$ serão alcançados na ordem dada por $k$."

$w_1$ e $w_n$, os vértices de $W$ que possuem os menores e maiores índices em $(s_i)_0^m$, são chamadas *extremidades* de $S$. O antecedente de $w_1$ e o sucessor de $w_n$ em $(s_i)_0^m$ são os *adjacentes* de $S$.

$W$ será um *loop*, se e somente se:

1. ele é uma $\mu$-curva regular na qual as extremidades - $w_1$ e $w_n$ - estão na mesma célula e, incorporando-se a $W$ os vértices adjacentes a ela no seu fim, a linha poligonal obtida não é mais uma $\mu$-curva regular. Esta segunda condição evita que uma $\mu$-curva regular com extremidades em uma de suas células duplas seja considerada um loop.

2. se $w$ é uma $\mu$-curva, $n = 2p$ e $w_1, \cdots, w_n, \cdots, w_{2p}$ são tais que $w_k$ e $w_{2n-k}$ estão sobre a mesma aresta da malha para $k = 1, \cdots, n-1$, então $W$ é dito uma *estrutura estreita* de $S$ com tamanho $p$.

Neste trabalho, vamos considerar somente *estruturas estreitas* que são maximais, no sentido de que não há outra com as mesmas extremidades e contendo

todos os seus vértices. Se $w_{n-1}$ e $w_n$ são vértices consecutivos de $S$, então a *estrutura estreita* é chamada *$\mu$-whisker*. Em caso contrário, ele é chamado gargalo. A figura 3.12 mostra dois exemplos de *$\mu$-whiskers*. A figura 3.4 mostra um gargalo.

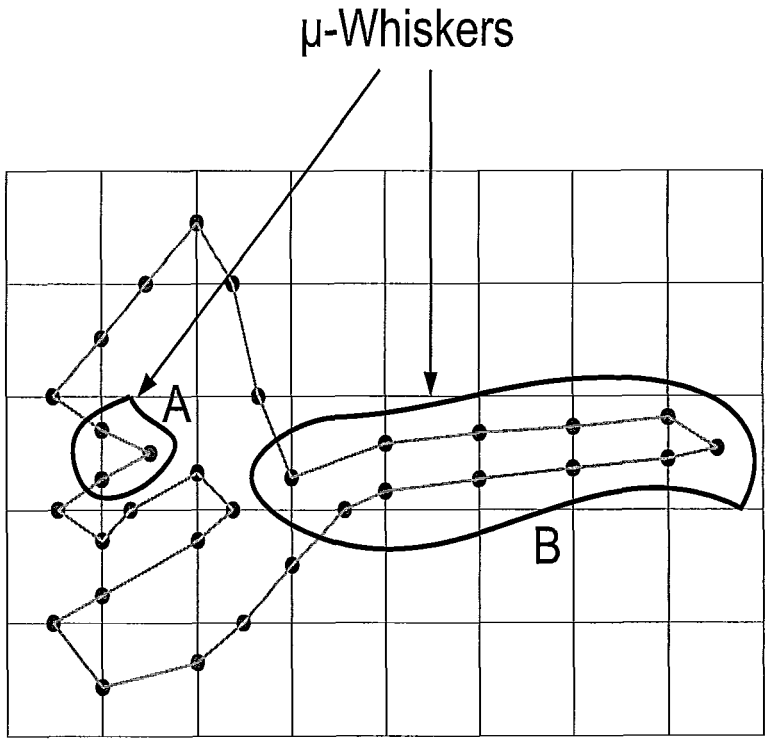


Figura 3.3: Dois $\mu$-whiskers

O nome *$\mu$-whisker* deriva do fato de que se todos os pontos de uma aresta da malha estão colapsados num único ponto dessa aresta, então um $\mu$-whisker de $S$ se tornará um whisker da região delimitada pela curva resultante de tal identificação, se ele não for nulo. As células cruzadas pelo gargalo formam uma conexão estreita entre dois *loops*.

Estruturas estreitas não podem ser parte de *loops* abertos de $PC_k$ uma vez que eles serão as *loop-snakes* do próximo estágio, as quais devem, obrigatoriamente, ser $\mu$-curvas regulares. Quando uma delas é encontrada, entretanto, ainda não se sabe que tipo de sub-estrutura irá conter suas adjacências. Por esse motivo, são

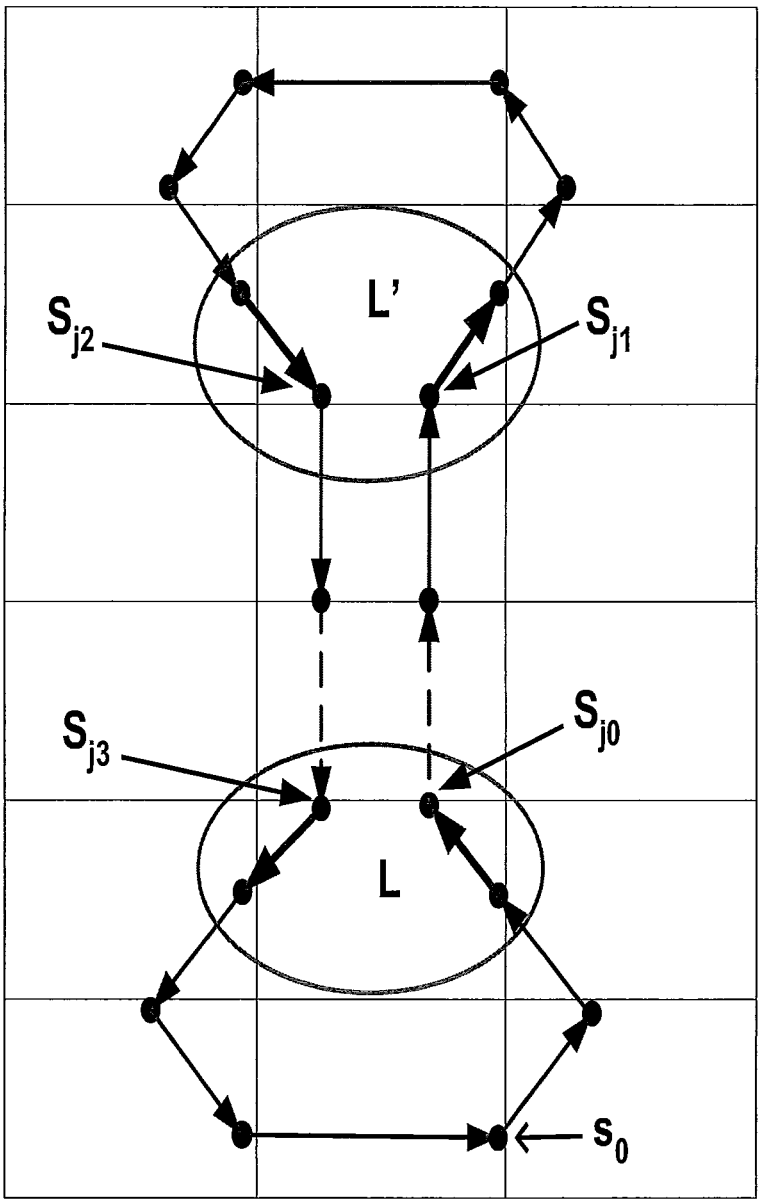


Figura 3.4: Um gargalo

todas removidas de $PC_k$.

Tendo obtido a curva $PC_k$ completa, todos os seus $\mu$-whiskers podem ser eliminados pelo simples procedimento dado abaixo, no qual assumimos que $v_0$ é o vértice inicial dos $m$ contidos em $PC_k$, os quais devem estar armazenados numa lista duplamente encadeada.

```
v=vo;
for(j=0; j<m; j++) {
  if(v->edge == (v->next)->edge) {
    (v->previous)->next = (v->next)->next;
    v = v->previous;
  }
  else
  v = v->next;
}
```

Este esquema pode, perfeitamente, ser empregado mas requer que $PC_k$ seja atravessada três vezes: uma para a geração dos seus vértices, uma para a eliminação dos *whiskers* e uma para identificação e rotulação dos *loops*. Uma análise um pouco mais elaborada da questão, entretanto, indica que é possível identificar/rotular os *loops* e eliminar todas as estruturas estreitas - e não somente *whiskers* - numa única passagem pela curva $PC_k$.

## 3.2 Mapeamentos com Propriedades Desejadas

O modelo de *loop-snakes* considera que $PC_k$ é a imagem de um mapeamento com propriedades desejadas definidas na $\mu$-dilatação de $S_k$. Esta seção especifica quais são tais propriedades.

Chamamos um *mapeamento elementar (e-map)* a transformação entre curvas obtidas no mesmo estágio $k$ de evolução da *T-snake*. Neste trabalho serão utilizados dois mapeamentos. O primeiro consiste em tomar a *snake* inicial num estágio $S_k$ e transformá-la, respectivamente, em $TC_k$ e depois $PC_k$, referidas como $T_k$ e $P_k$. O mapeamento chamado $\Gamma_k$ é como $P_k$, só que definido na $\mu$-dilatação de $S_k$. $\Gamma_k$ será utilizada por ser mais fácil fazê-la ter certas propriedades desejáveis do que $P_k$.

$T_k$ pode ser facilmente definida devido a correspondência entre os vértices de $S_k$ e $TC_k$. Entretanto, não há definição natural para $P_k$ e $\Gamma_k$. O primeiro objetivo é fazer esses mapeamentos serem lineares contínuos, um a um, preservando a ordem dos *snaxels*, e associando pontos próximos uns aos outros. Para estabelecer este último requisito de uma maneira mais formal, o conceito de mapeamento elementar (*e-map*) limitado pela malha é introduzido.

Se $f$ é um elemento da malha: vértice, aresta ou célula, chamamos $N(f)$ a união de todas as células que são adjacentes aos vértices de $f$. Um *e-map* $M$ será dito *limitado pela malha* se $s_i \in f \Rightarrow M(s_i)$ está no interior de $N(f)$. Obviamente, se a distância física entre todos os *snaxels* é menor do que $d$, $T_k$ será limitada por $\mu$ e, neste caso, $P_k$ e $\Gamma_k$ podem ser limitadas por $\mu$ também.

Chamamos de *raia* um destes mapeamentos, $M$, no ponto $s$ ao segmento aberto delimitado por $s$ e $M(s)$. Os pontos $s_i$, $s_{i+1}$, $M(s_{i+1})$ e $M(s_i)$ definem um quadrilátero de varrdura de $M$ nos vértices $s_i$ ou $s_{i+1}$. $[M(s_i), M(s_{i+1})]$ definem a parte de cima do quadrilátero e $[s_i, s_{i+1}]$ a parte debaixo. Dois *e-maps* $M$ e $M'$ são ditos $\mu$-equivalentes se suas imagens são $\mu$-curvas equivalentes e $\forall s_i$, $M(s_i)$ e $M'(s_i)$ estão na mesma célula da malha.

Um *e-map* $M$ com todas as propriedades apresentadas até agora, incluindo ser contrátil e limitado pela malha, será dito adequado se não gera quadriláteros reversos de varredura. Um mapeamento adequado é dito ideal, se e somente se,

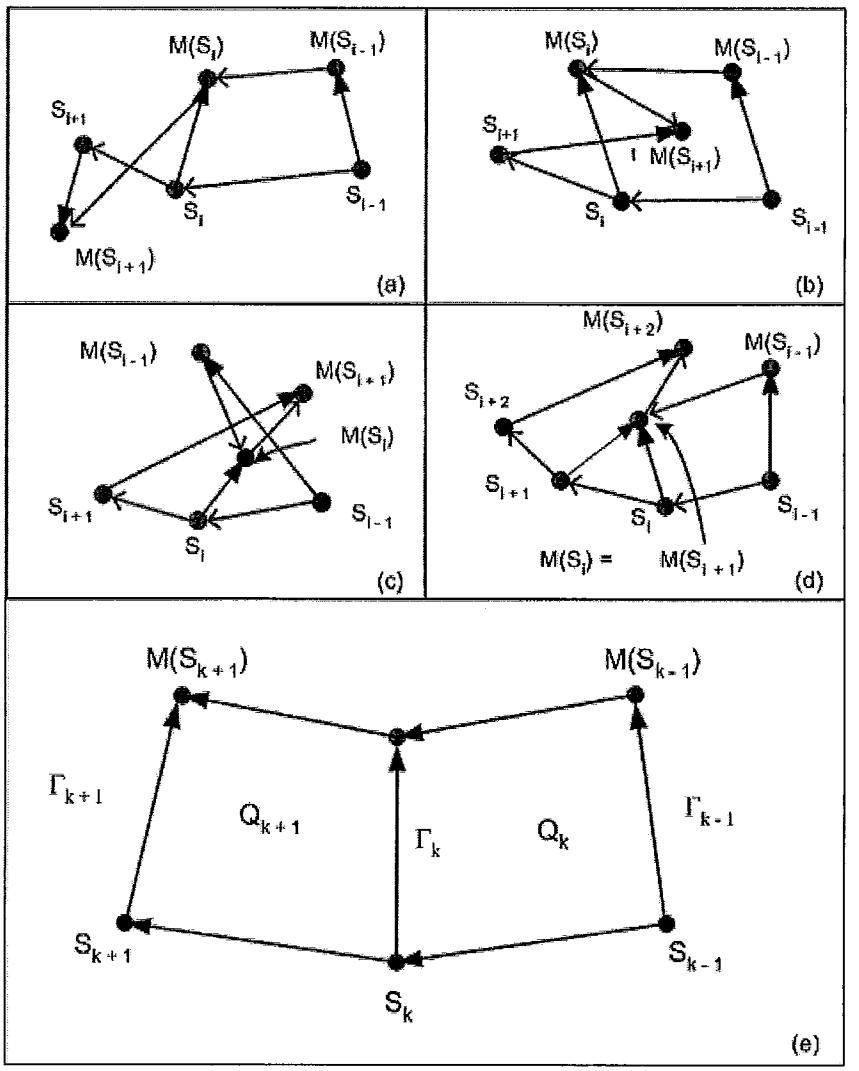


Figura 3.5: Em (a) e (b) têm-se dois *e-maps* não adequados. Em (c) e (d) dois mapementos adequados e , finalmente, em (e) um mapeamento ideal.

não existem quadriláteros de varredura degenerados e,

$$\forall s_i, M(s_i) \text{ está na borda de } Q_{i-1} \cup Q_i \tag{3.1}$$

Esta condição é claramente satisfeita se os interiores de quaisquer dois quadriláteros de varredura consecutivos são disjuntos. Observando a figura 3.5, os *e-maps* representados em (a) e (b) não são adequados uma vez que formam quadriláteros reversos. Em (c) e (d) são mapeamentos adequados mas não ideais, pois $M(s_i)$ é interior a $Q_{i-1} \cup Q_i$ em (c) e há um quadrilátero de varredura degenerado - [$s_i$, $M(s_i)$ e $s_{i_1}$] - em (d). Finalmente, o mapeamento em (e) é ideal.

Um mapeamento adequado $M$ pode se tornar um *e-map* ideal - $M'$ - simplesmente pela introdução, para cada $i$ no qual a equação 3.1 não é válida, de novos vértices $v_i$ em $S$ e $v_i' = M'(v_i)$, como mostrado na figura 3.6. $v_i$ pode ser qualquer ponto no interior relativo de $[s_{i-1}, s_i]$ equanto $v_i'$ deve ser um ponto tal que:

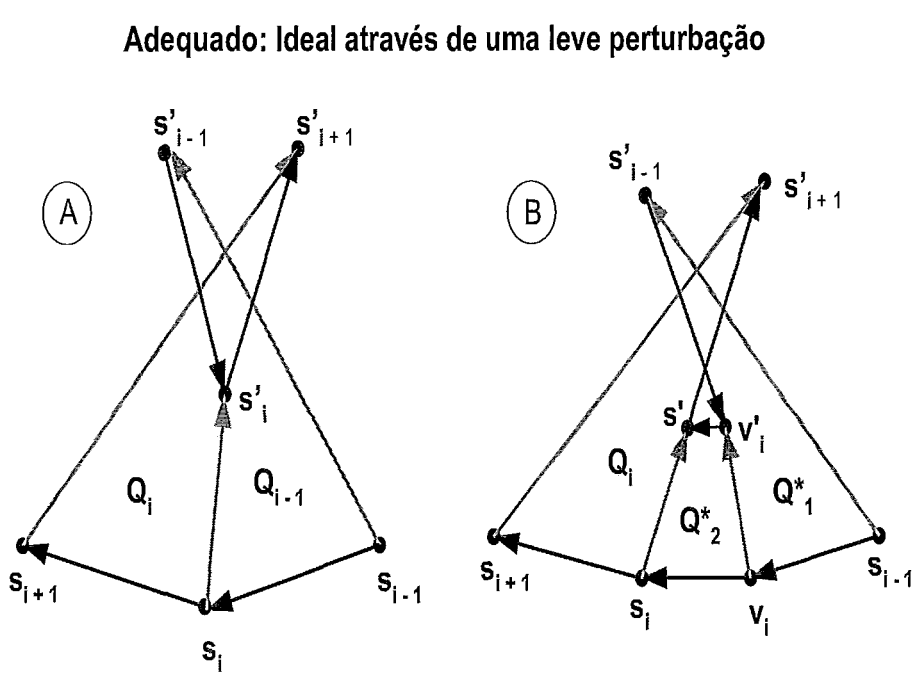


Figura 3.6: Tornando um mapeamento adequado ideal

$$[v_i', s_i') \subseteq Q_{i-1}^0/Q_i \quad e \tag{3.2}$$

$$[v_i, v_i', s_{i-1}'] \subseteq Q_{i-1} \tag{3.3}$$

Tal ponto sempre existe desde que $M$ seja monotônica. Tendo $v_i$ e $v_i'$ substituído o quadrilátero $Q_{i-1}$ por $Q_1^*$=[$s_{i-1}$, $s_{i-1}'$, $v_i'$, $v_i$] e $Q_2^*$=[$v_i$, $v_i'$, $s_i'$, $s_i$]. A condição 3.2 implica que $s_i'$ está na borda de $Q_2^* \cup Q_i$ e 3.3 determina que $Q_1^*$ e $Q_2^*$ têm interiores disjuntos. Assim, $M'$ satisfaz a condição 3.1 em $v'$ e $s'$. Repetindo o processo para outros $s_i$ nos quais $M$ não satisfaz a equação 3.1, terminamos obtendo um mapeamento ideal.

Como descrito na figura 3.6, a inserção de $v_i'$ faz $S'$ ter um *loop* contendo $[s_i', v_i']$. Desde que seja sempre possível fazer $[v_i, v_i']$ arbitrariamente próximo a $[s_i, s_i']$, este *loop* pode, em qualquer caso, ser infinitamente pequeno. Isto, em particular, significa que ele é disjunto do restante de $S'$ e, em consequência, a estrutura de *loops* da imagem de um *e-map* adequado é a mesma que a da imagem de um mapeamento ideal, exceto por *loops* arbitrariamente pequenos que não cruzam o restante da curva. Como veremos na sequência deste trabalho, tais *loops* são folhas da *Loop-tree* de $S'$. Assim, a topologia das *Loop-trees* de curvas geradas por um mapeamento ideal e um adequado, só diferem pela existência de tais folhas.

## 3.3 Tornando um Mapeamento Elementar $\Gamma_k$ Adequado

Quando um *snaxel* $s_i$ é processado, tudo o que se deve fazer em relação ao controle topológico, além - é claro - de se verificar e atualizar a $TCS$, é garantir que é

possível construir um mapeamento $\Gamma_k$ que seja localmente adequado em $v_i$ - o nó externo a $s_i$. Esta seção descreve como fazer isto, especialmente, se os *snaxels* estão espacialmente separados de forma adequada. Neste caso, basta fazer com que $\Gamma_k$ seja contrátil.

Inicialmente, temos que justificar a escolha de $\Gamma_k$ em detrimento de $P_k$. Tornar $P_k$ um mapeamento adequado pode ser excessivamente restritivo, pois neste caso, as raias de $P_k$ devem estar contidas no cone determinado pelo ângulo externo de $S_k$ no *snaxel*. Esta, entretanto, pode ser uma restrição muito forte se esse ângulo é muito pequeno.

Tornar $\Gamma_k$ adequada é suficiente para evitar que o *snaxel* não retorne a uma célula que já tenha sido completamente varrida. Além disso, como os ângulos internos de $\mu D(S_k)$ têm ao menos 90°, as raias não estarão excessivamente restringidas.

Para fazer $\Gamma_k$, inicialmente, simultaneamente limitada pela malha e monótona em cada *snaxel* $s_i$, devemos restringir $T_k(s_i)$ às quatro células adjacentes a $u_i$ - o nó interno de $s_i$. Isso pode ser feito calculando $T_k(s_i)$ por meio do seguinte procedimento:

1. Considere $GNI$ uma imagem na qual o valor de cada pixel $p$ é proporcional a $\|\nabla I(p)\|$, estimado de modo padrão.

2. Obter $GNI_{cell}$ que é $GNI$ ao nível das células. Para isso, atribuímos a cada célula a média dos valores dos seus *pixels*.

3. O campo é, então, calculado. No nível dos *pixels* calculamos o ponto

$$t_{pixel} = \frac{\sum_{p \in V(u_i)} GNI(p).p}{\sum_{p \in V(u_i)} GNI(p)} \tag{3.4}$$

onde $V(u_i)$ é a união das 4 células adjacentes a $u_i$. $t_{pixel}$ é o centro de massa de $V(u_i)$ se os pesos forem dados por $GNI$. Observe que $t_{pixel}$ é o mesmo para todo *snaxel* cujo nó interno é $u_i$.

4. Se for considerado que os gradientes em $V(u_i)$ não são suficientemente relevantes, em relação a um valor $G$ pré-estabelecido para o gradiente, pode-se, basicamente, repetir o procedimento anterior, só que no nível das células. Neste caso, $s_i$ é associado a $u_i$. Explicitamente determinamos:

$$t_{cell} = \frac{\sum_{C \subseteq (W_j(u_i) - V(u_i))} GNI_{cell}(C).\rho(u_i, C)}{\sum_{C \subseteq W_j(u_i)} GNI_{cell}(C)} \tag{3.5}$$

onde:

(a) $C$ é qualquer célula $\subseteq (W_j(u_i) - V(u_i))$

(b) Para definir $W_j(u_i)$, fazemos $W_k(u_i)$ uma janela *kd x kd* centrada em $u$ e, dados valores pré-estabelecidos de $k$ e $G$, que podem variar nas iterações, obtemos $j$ por:

$$\{j = 0; \text{while } (\, j < k \text{ and } \sum_{C \subseteq W_j(u_i)} GNI_{cell}(C) < G); j++\}$$

(c) $\rho(u_i, C) = (d-1) * (\textit{Centro de } C - u_i)/(\|\textit{Centro de } C - u_i\|)^2$ é previamente calculada, lembrando que $d$ é o deslocamento máximo permitido para um *snaxel*. Esta função nada mais é do que o "peso" de uma célula divido por um fator - $(\|\textit{Centro de } C - u_i\|)^2$ - o que favorece as células próximas a $u_i$. Assim, o objetivo no passo 4 é permitir, com apenas um esforço computacional moderado, que um *snaxel* que não esteja suficientemente próximo a nenhum dos contornos seja afetado por eles, ao invés de se mover apenas em função das forças internas que atuam sobre ele. Forças estas que também podem ser pequenas.

5. Finalmente, calcular $t_{field}$ como se segue:

$$t_{field} = \left( \frac{GNI(p)}{\sum_{C \subseteq W_j(u_i)} \rho(u_i, C).GNI_{cell}(C)} \right) t_{pixel} + t_{cell}, \tag{3.6}$$

onde $t_{field}$ é o ponto no qual $s_i$ sofre ação do campo. Observar que devido à introdução do fator $\rho(u_i, C)$, $t_{field}$ está em $V(u_i)$.

6. Somente uma força interna, determinada por uma energia de primeira ordem, é utilizada. Ela é calculada por uma aproximação discreta de $\frac{\partial^2 S}{\partial S^2}(S_i)$ multiplicada por um peso $w'$. Um limite razoável para tal peso - considerando que o uso de uma força interna tende a suavizar a curva - é um valor que faça $s_i$ se mover para uma posição que miniminize localmente sua energia. Por exemplo, se a distância $D_1(s_i)$ determinada pela energia de primeira ordem é dada por $w' * (s_{i+1} + s_{i-1} - 2s_i)$, então o limite deve ser $\frac{1}{2}$. Para este valor, $s_i$ é levado em $\frac{(s_{i+1}+s_{i-1})}{2}$, o ponto no qual a função $s \rightarrow w' * (s_{i+1} + s_{i-1} - 2s_i)^2$ é zero. Entretanto, fazer $w' = \frac{1}{2}$ pode não ajudar a suavizar a curva, como na figura 3.7 demonstra. Este valor é ótimo quando os *snaxels* são deslocados um de cada vez, não quando todos se movem simultaneamente. Em vista disso, e porque isto torna mais direto o controle topológico, um valor $\leq \frac{1}{3}$ é utilizado para $w'$.

   Para $w' < \frac{1}{2}$, $t_{internal} = s_i + D_1(s_i)$ no interior do triângulo $[s_{i-1}, s_i, s_{i+1}]$, então, $t_{internal} \in V(u_i)$ desde que os três vértices deste triângulo estejam no fecho de $V(u_i)$.

7. Uma combinação convexa é usada para obter a nova posição de $s_i$ - $T_k(s_i)$ - em função de $t_{internal}$, $t_{field}$ e $u_i$. A introdução de $u_i$ na combinação corresponde a aplicar uma força interna normal com direção da aresta contendo $s_i$. Os coeficientes usados na combinação podem variar dependendo, por exemplo, do valor de $\sum_{p \in V(u_i)} GNI(p)$ e do número de iterações nas quais $C(s_i)$ foi visitada pela *snake*.

Uma vez que $V(u_i)$ contém os três pontos: $t_{internal}$, $t_{field}$ e $u_i$, ele deve conter $T_k(s_i)$.

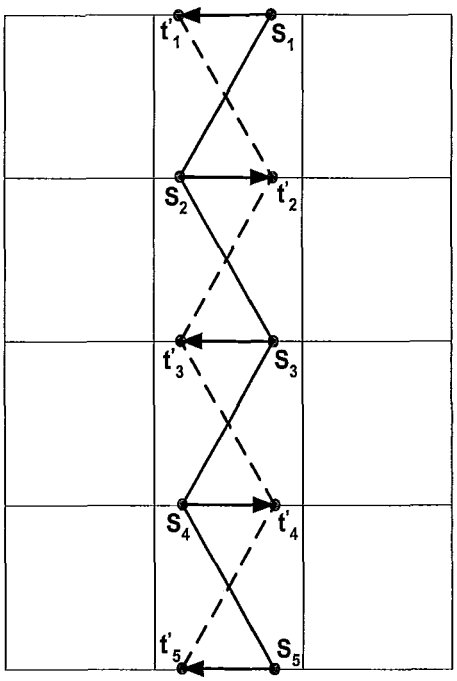


Figura 3.7: Utilizar $w' = \frac{1}{2}$ pode não suavizar a curva

Portanto, calcular os deslocamentos dos *snaxels* do modo descrito anteriormente para fazer $T(s_i) \in V(u_i)$ não requer qualquer medida adicional específica. Assim, podemos considerar que esta condição pode ser admitida sem custo computacional extra.

Se $s_i$ está na célula contendo uma diagonal $\delta$ que é uma aresta de $\mu D(S_k)$, então é possível que $T_k(s_i)$ esteja fora dessa curva. Entretanto, como podemos ver na figura 3.8, a projeção na malha resolverá isto, uma vez que nas células onde estão essas diagonais, $TC_k$ pode somente cruzar arestas da malha que estão no interior de $\mu D(S_k)$.

Infelizmente, o mesmo não acontece numa situação em que vértices sucessivos de $PC_k$ tem o mesmo nó a esquerda, no qual $\mu D(S_k)$ forma, internamente, um ângulo $> 180°$. Esta situação, que é mostrada na figura 3.9, torna necessária uma intervenção que é feita no momento em que um vértice de $PC_k$ é gerado. Nesta

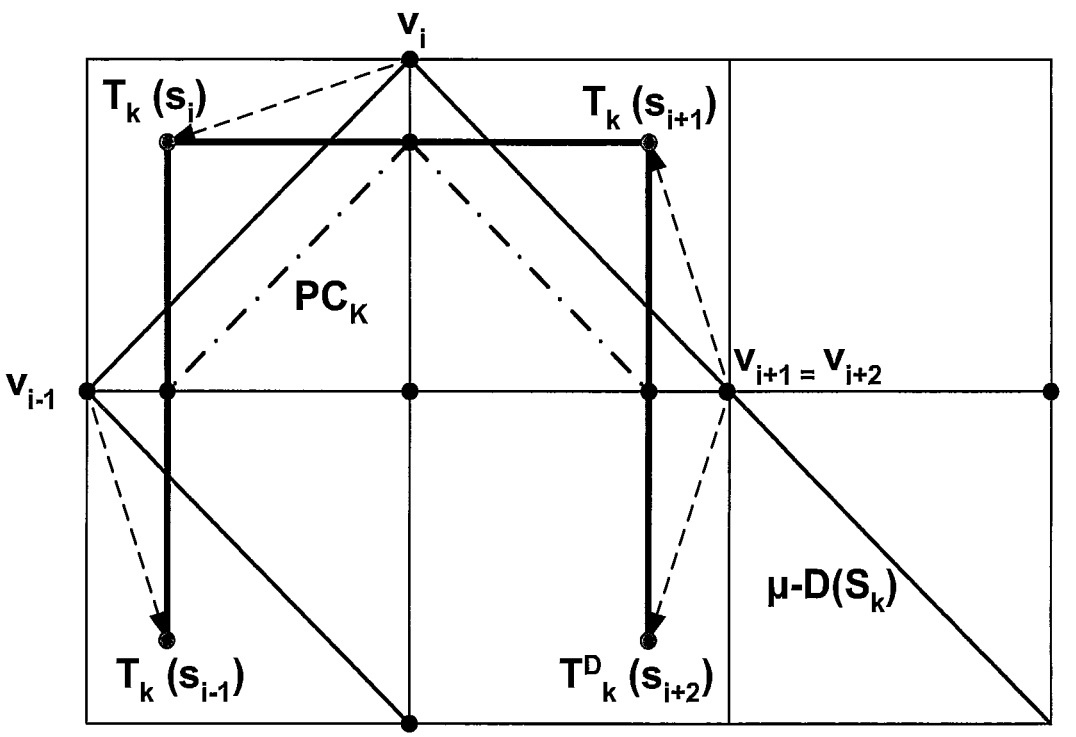


Figura 3.8: A curva transformada corta $\mu D(S_k)$ mas não a sua projeção $PC_k$

situação, $PC_k$ cruza $\mu D(S_k)$. Ela fica perfeitamente caracterizada por:

- Dois *snaxels* consecutivos, $s_{i-1}$ e $s_i$, contidos numa célula $C_1$ e que possuem os mesmo vértice esquerdo $v_i$.
- A intersecção de $[T_k(s_{i-1}), T_k(s_i)]$ com a célula $C_2$, diagonalmente oposta a $C_1$ em relação a $v_i$, é o segmento delimitado pelos vértices $s'_{j-1}$ e $s'_j$ de $PC_k$, os quais estão ambos sobre arestas adjacentes a $v_i$.
- $s_i$ e $s'_j$ devem pertencer a mesma célula, $C$.

$C_2$ é chamada de **célula proibida** (Figura 3.9) pois $PC_k$ tem que cruzar $\mu D(S_k)$ para entrar em $C_2$. Ela é totalmente identificada se conhecemos $C_1$ e a coordenada da aresta de $s_i$ ($E_i$). Suponhamos que $M_c$ e $N_c$ sejam as dimensões da malha de células e que a célula($i$,$j$) seja representada por $(i * N_c + j)$. Neste caso $C_2$ será uma célula proibida se e somente se $C_2 - C_1 = \Delta(E_i)$, onde $\Delta(.)$ é definida como:

$$\Delta(0) = -N_c + 1, \Delta(1) = N_c + 1, \Delta(2) = N_c - 1, \Delta(3) = -N_c - 1 \quad (3.7)$$

Se $C_2$ é identificada com uma célula proibida, a maneira mais simples de fazer com que $PC_k$ não cruze $\mu D(S_k)$ é substituir os vértices $s'_{j-1}$ e $s'_j$ por $s_{i-1}$ e $s_i$, respectivamente.

Observe que $s_i$ é o *snaxel* corrente no momento em que o processo de construção de $PC_k$ calcula $s'_{j-1}$. Nesse momento, a célula proibida será a próxima célula a ser cruzada por $PC_k$ - que será referida aqui por *Next_cell_of_PC* - e $C1$ é célula na qual está a aresta $[s_{i-1}, s_i]$ cuja imagem por $T_k$ está sendo regularizada - referida aqui como *Current_cell_of_Snake*. Como $i$ e $j-1$ são os índices correntes das seqüências de *snaxels* e vértices de $PC_k$, vamos denotá-los por $i_{cur}$ e $j_{cur}$ respectivamente. Considerando toda esta notação, podemos escrever o comando que

deve ser executado para evitar que $PC_k$ entre em uma célula proibida, da seguinte maneira:

$$\{ \textbf{if}(\ \textit{Next\_cell\_of\_PC} = \textit{Current\_cell\_of\_Snake} + \Delta(E_{jcur})) \textbf{ then } \{$$
$$s'(j_cur) = s(i_{cur} - 1);\ s'(j_{cur} + 1) = s(i_{cur})\ \}$$

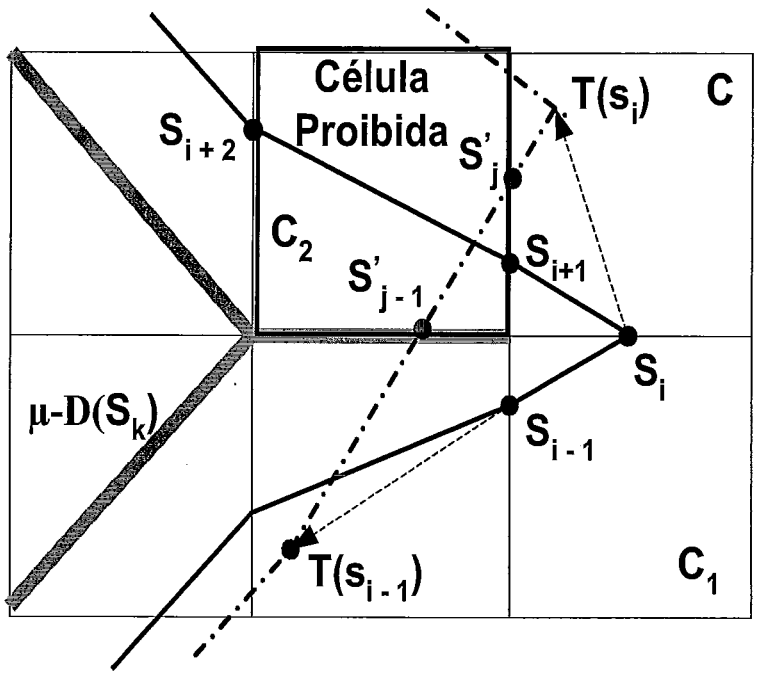


Figura 3.9: Configuração indesejada

A utilização do esquema apresentado anteriormente para deslocar os *snaxels* faz com que não seja necessário executar nenhum processo especial de eliminação de quadriláteros reversos de $\Gamma_k$. Em conseqüência, o comando indicado acima é tudo que deve ser, eventualmente, feito quando se processa um *snaxel* para controlar a topologia da *snake*, além de, é claro, se verificar e atualizar a estrutura de controle topológico.

Chamamos uma célula que não contenha vértices de $\Gamma_k$ de **nula**. Uma célula proibida deve ser nula. Assim, o teste descrito acima necessita ser aplicado ape-

nas quando tal condição exige. Entretanto, não é necessário verificar tal condição explicitamente. Na regularização de um segmento $\rho$ de $TC_k$, a cada nova inserção com a malha que é calculada, deve ser verificado se a célula ao final da extremidade de $\rho$ foi alcançada. Senão, a nova célula alcançada é nula. Desse modo, se $PC_k$ entra numa célula não-nula num de seus vértices, pode-se dizer que nada deve ser feito em tal vértice para controlar a topologia, exceto consultar $TCS$ e atualizá-la.

Uma última medida para fazer com que $\Gamma_k$ se contraia: se a diagonal de uma célula dupla $C$ de $S_k$ está totalmente contida em seu interior, então os *snaxels* em $C$ estão imobilizados ou a *snake* é dividida em $C$. Isto evita que $PC_k$ volte a cruzar uma célula que já foi totalmente varrida pela *snake*. Uma vez que tais células exigem um tratamento específico, elas são chamadas **especiais**. Células duplas especiais são detectadas pelo processo de atualização da *snake* e tratadas antes que a *snake* evolua. Assim, quando a *snake* é percorrida para gerar $TC_k$ e $PC_k$, não há necessidade de verificar se foi alcançada alguma célula especial.

Há, ainda, um importante **by-product** em se utilizar o esquema apresentado anteriormente para calcular o deslocamento de um *snaxel*. Ele é consequência do fato de que o campo externo leva ao mesmo ponto todo *snaxel* que tenha o mesmo nó externo. Ainda, há a ação das forças internas que possuem um peso suave ($w' < \frac{1}{3}$). Portanto, não é necessária nenhuma ação específica para eliminar quadriláteros reversos de varredura de $\Gamma_k$.

Seja, agora:

1. $C$ é a célula contendo o segmento $e_i = [s'_j, s'_{j-1}]$ de $PC_k$.

2. $z_i = \Gamma_k^{-1}(s'_i)$, $i = j, j+1$.

3. $r_j$ a linha suporte de $e_i$. Então, se as raias $[z_i, s'_i]$ e $[z_{i+1}, s'_{i+1}]$ se intersectam, três alternativas são possíveis:

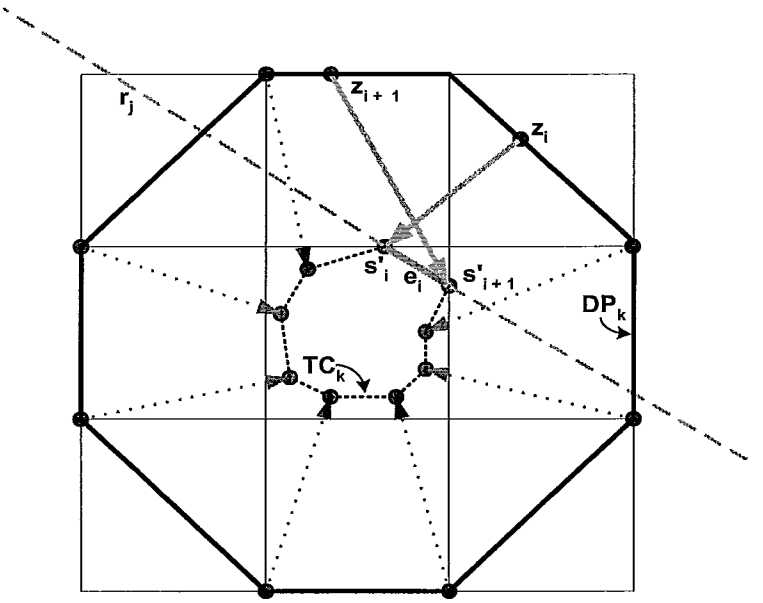


Figura 3.10: Se $\Gamma_k^{-1}(e_i)$ corta $r_j$, então $\Gamma_k$ não tem quadrilátero reverso de varredura com a parte de cima em $e_i$

(a) Se $\Gamma_k^{-1}(e_i)$ cruza $r_j$, então $\Gamma_k$ não possui quadrilátero reverso de varredura com a parte superior em $e_i$, de acordo com a figura 3.10.

(b) Se $\Gamma_k^{-1}(e_i)$ não cruza $r_j$ mas contém um vértice externo de $S_k$, então existe um mapeamento $\Gamma_k^j = \gamma_j(\Gamma_k)$, $\mu$-equivalente a $\Gamma_k$, tal que:

   i. O segmento $e_i'$ da imagem de $\Gamma_k^j$, que corresponde a $e_i$, tem uma extremidade no vértice $s_i'$.

   ii. $\Gamma_k^j$ não possui um quadrilátero de varredura reverso com a parte superior em $e_i$. Ver figura 3.11.

(c) Se $\Gamma_k^{-1}(e_i)$ não contém nenhum vértice externo de $S_k$, então ou $e_i$ pertence a um *$\mu$-whisker* - como mostrado na figura 3.12 - ou se torna adequado com a inserção de um ponto extra.

Se os *snaxels* não forem deslocados pela aplicação do esquema apresentado ante-

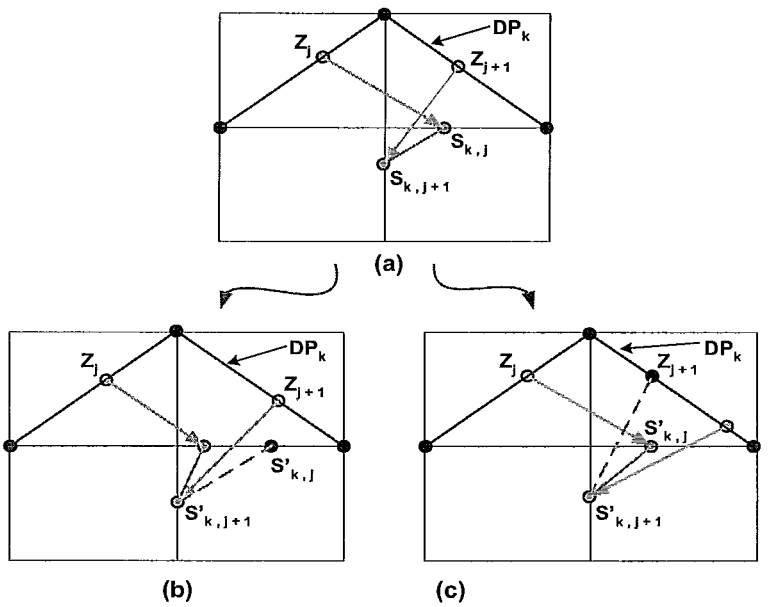


Figura 3.11: Se a posição de $z_{j+1}$ muda como o mostrado nas figuras (b) e (c), a interseção entre $[z_j, s'_i]$ e $[z_{j+1}, s'_{i+1}]$ desaparece. Isto elimina o quadrilátero reverso. Observe que $z_j$ permanece imóvel.

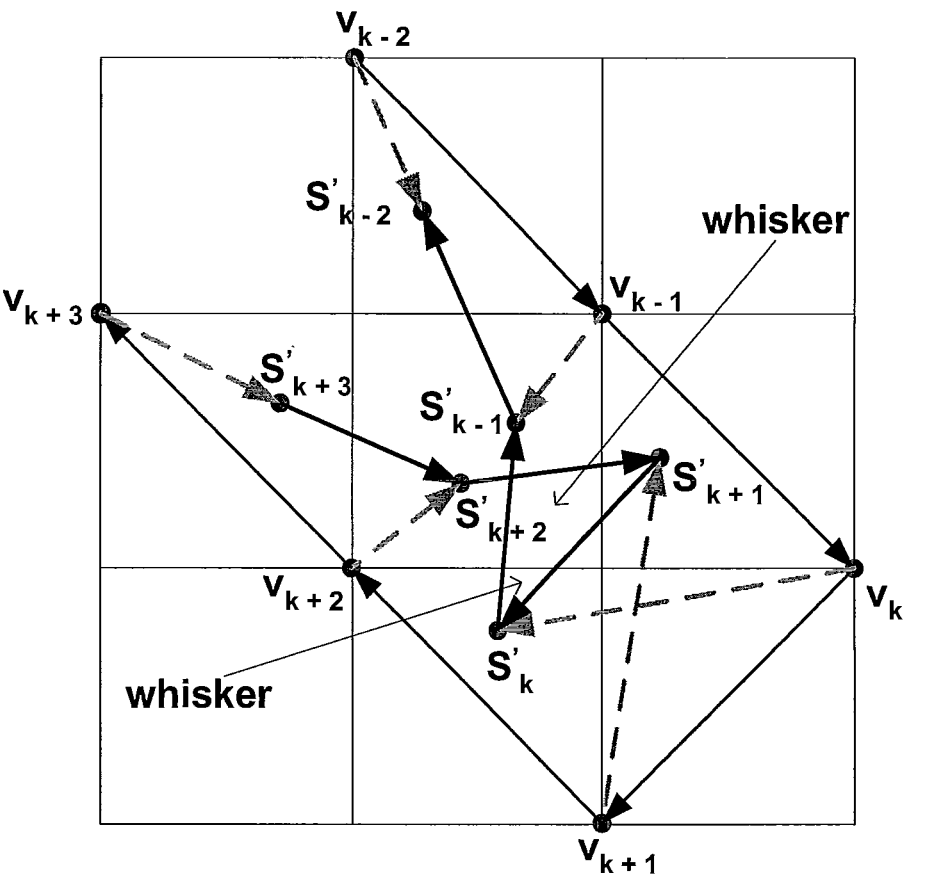


Figura 3.12: A aplicação repetida do item (b) torna $\Gamma_k$ adequada.

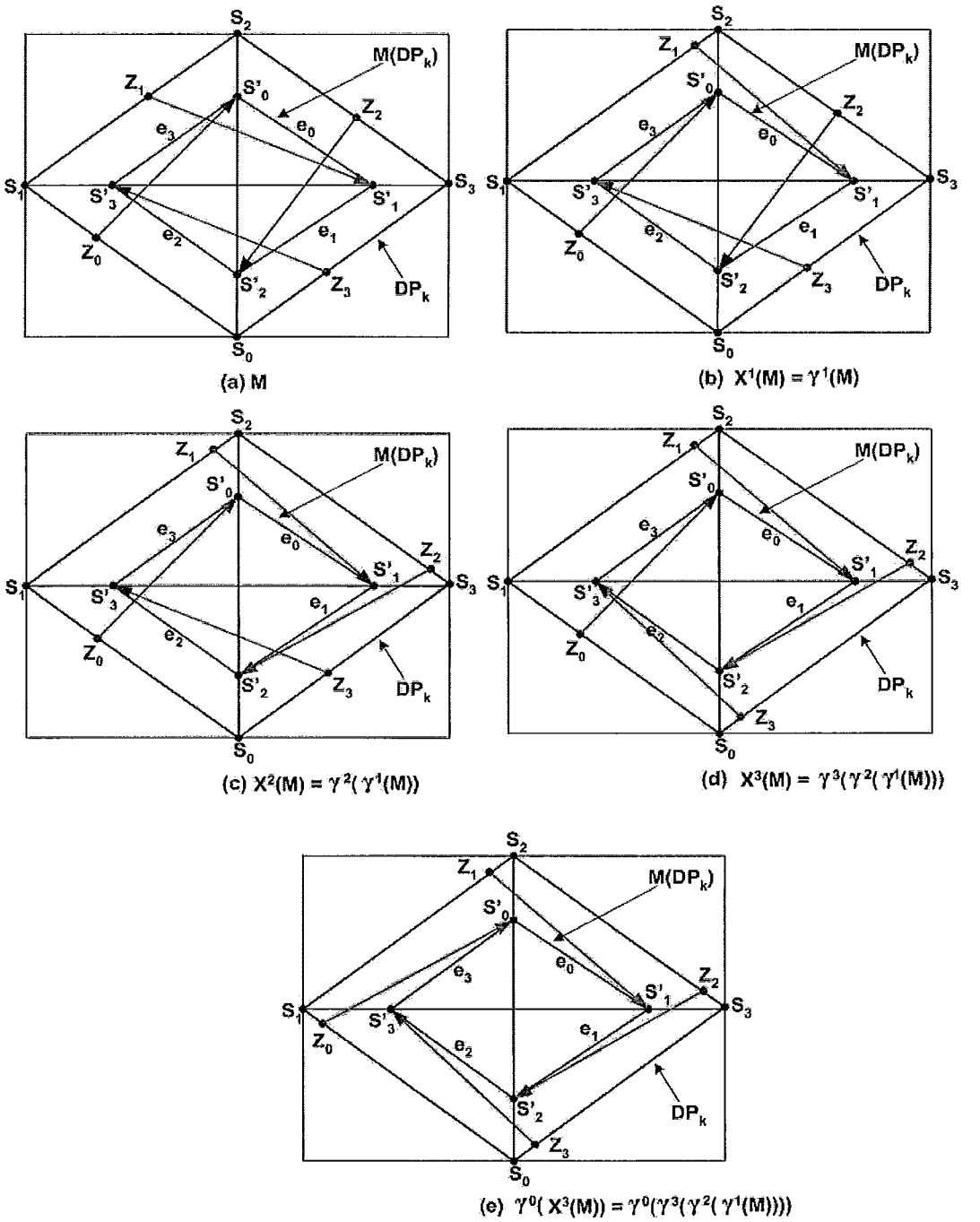


Figura 3.13: A aplicação repetida do item 3b torna $\Gamma_k$ adequada.

riormente, deve-se seguir alguns passos para evitar que quadriláteros reversos se formem. Isto pode ser feito como indicado em [21] e exige um teste a mais para snaxels cuja inclusão faça $PC_k$ entrar numa célula nula. A aplicação repetida do resultado indicado em 3b acima pode ser usada para eliminar todos os eventuais quadriláteros reversos de $\Gamma_k$, um por um. Isto é feito obtendo-se os *e-maps* da seguinte sequência:

$$\gamma_1(\Gamma_k), \gamma_2(\gamma_1(\Gamma_k)), \gamma_3(\gamma_2(\gamma_1(\Gamma_k)), \cdots \tag{3.8}$$

como indicado na figura 3.13. O *e-map* obtido ao fim desse processo é adequado. A imagem deste mapeamento, entretanto, não é mais $PC_k$ mas a curva $\chi$, $\mu$-equivalente a ela. Isto não é um problema, pois a topologia destas duas curvas será a mesma, exceto por eventuais *loops* pequenos não contendo um único vértice da malha. Assim, no nível de precisão em que estamos trabalhando, tais *loops* podem ser ignorados e a topologia de $\chi$ adotada. Isto não significa absolutamente que os vértices de $PC_k$ devem ser substituídos pelos de $\chi$. A idéia é a topologia de $\chi$, não sua geometria. $PC_k$ simplesmente herda as boas propriedades que $\chi$ tem, como a imagem de um mapeamento adequado definido numa curva $\mu$-regular. Tais propriedades tornarão mais simples rotular seus *loops*, como será visto na seção 4.1. Nessa seção, será mostrado, ainda, que o conjunto de *loops* que é obtido depende somente da seqüência das arestas da malha cortadas pela $\mu$-curva, que é a mesma para $PC_k$ ou $\chi$. Desse modo, é indiferente considerar uma ou outra.

Uma nota final: fazer $\Gamma_k$ ideal requer um esforço computacional que não vale a pena. Mesmo detectar simplesmente os vértices nos quais um mapeamento não é ideal requer que três *snaxels* consecutivos e os vértices de $PC_k$ sejam considerados. Assim, a identificação destes vértices tem um custo por *snaxel* que não é compensado pelo benefício determinado por isso. Este benefício é simplesmente

o processo de rotulação, que é somente aplicado uma única vez por *loop*. O fato de que $\Gamma_k$ é somente adequada, parece gerar um enfoque mais eficiente.

# Capítulo 4

# Algoritmo

## 4.1 Determinando *Loops* e construindo a árvore de *Loops*

Agora, que sabemos como um *snaxel* é processado, devemos considerar como tratar o caso mais complicado, no qual a adição de um *snaxel* à curva transformada faz com que a curva transformada - ou sua projeção na malha, $PC_k$ - retorne a uma célula que já foi visitada anteriormente. Começamos definindo a **árvore de *loops*** de uma curva contínua.

Uma árvore de *loops* (**Loop-tree**, figura 4.1) de uma curva fechada $C$ sem múltiplos pontos de auto-interseções é um grafo que pode ser obtido pelo processo simples, que se segue: Escolhe-se um ponto $s$ em $C$ e um sentido $D$ que pode ser horário ou anti-horário. Percorre-se então $C$ na direção escolhida começando por $s$. A cada momento em que um ponto $x$ é revisitado, cria-se um nó para representar o *loop* formado pelo trecho de $C$ entre as duas visitas a $x$. Esse *loop* é então colapsado em $x$ e continua-se o percurso. Depois de completá-lo, para cada *loop* $L_1$ que foi concentrado em um ponto de outro *loop* $L_2$, cria-se uma

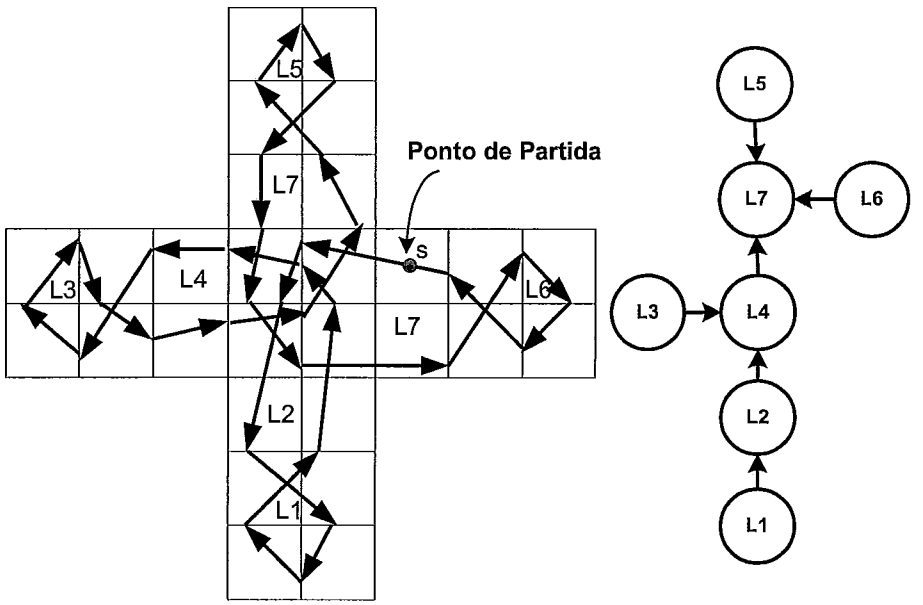


Figura 4.1: Uma curva e sua *Loop-tree*

aresta orientada do nó $L_1$ para $L_2$. Uma curva e a *Loop-tree* associada a ela, são mostradas na figura 4.1

Para adaptar o conceito de *Loop-tree* à definição de *loop* dada no capítulo 3.1, devemos considerar algumas situações. Uma delas é que os *loops* devem ser curvas $\mu$-regulares que são determinadas não somente por auto-interseções de $PC_k$, mas também pelo fato de que $PC_k$ retornou para uma aresta da malha que ela já havia cruzado anteriormente. Além disso, deve-se considerar que nem todos os vértices da curva pertencem a um *loop*, mas podem estar numa estrutura estreita. Neste caso, o processo simples para construir a *loop-tree*, apresentado acima, deve ser substituído por outro mais elaborado.

Considere que $PC_k$ é percorrida na ordem em que seus vértices são computados pelo processo de sua obtenção e defina o ***Loop* em construção** - $LeC$ - como a linha poligonal definida pelos vértices de $PC_k$ que já foram alcançados e não pertencem a um *loop* ou a uma estrutura estreita já encontrados. O $LeC$ deve ser

uma $\mu$-curva regular porque sua parte final pode estar num *loop* aberto ainda a ser gerado, uma vez que *loops* abertos são transformados em *snakes* no próximo estágio. Assim, durante o percurso de $PC_k$ o $LeC$ deve ser mantido regular.

Equanto isso é verdadeiro, deve-se extender o $LeC$ acrescentando o próximo vértice de $PC_k$ à sequência que o define. Se, com tal extensão, $S$ se torna não-regular, uma das seguintes condições deve ocorrer:

- Com a extensão, o $LeC$ intersecta uma aresta ($e$) da malha duas vezes. Neste caso, seja $s_{j1}$ o vértice de $PC_k$ localizado na primeira intersecção do $LeC$ com $e$. O último vértice de $S$ - $s_{j2}$ - está localizado na segunda intersecção. Se $s_{j1}$ e $s_{j2}$ são vértices consecutivos do $LeC$, eles estão numa estrutura estreita e devem ser removidos da definição do $LeC$. $s_{j2}.prev$ se torna, então, seu vértice terminal. Se o próximo vértice que é adicionado ao $LeC$ - $s_{j2+1}$ - está na aresta de $S_{j2}.prev$, deve-se removê-los também. E continua-se, assim, eliminando o vértice recém acrescentado e seu antecedente no $LeC$ até que eles estejam em arestas diferentes da malha.

  Neste momento, a estrutura estreita terminou. Coloca-se, então, os dois últimos vértices do $LeC$, que são adjacentes a ele, numa lista $NS$ (*Lista de estruturas estreitas*).

  Se $s_{j1}$ e $s_{j2}$ não são consecutivos, um *loop* - $L$ - deve ser formado e removido do $LeC$ para evitar que *loops* encontrados posteriormente contenham alguma de suas arestas. Isto é fundamental para evitar que, na próxima iteração, duas *snakes* diferentes tenham *snaxels* em comum. De acordo com o lado de $e$ em que o $LeC$ retorna a ela, diferentes regras devem ser empregadas tanto para formar $L$ quanto para manter o $LeC$ uma $\mu$-curva regular depois da remoção de $L$.

  Se em $S_{j2}$ o $LeC$ retornar para $e$ vindo da mesma célula que ele está en-

trando em $S_{j1}$, então forme $L$ pela ligação de $S_{j2}.prev$ a $S_{j1}.next$. *Loops* formados dessa maneira são chamados *loops* de um lado (**one-side *loops***). Formado o *loop*, remova $L$, $S_{j1}$ e $S_{j2}$ do $LeC$, o que faz com que $S_{j1}.prev$ se torne seu vértice final. Essa situação é apresentada na figura 4.2 (A).

Se a curva retorna a $e$ vindo de uma célula que está do outro lado de $e$, $L$ é formado pela conexão de $S_{j2}.prev$ a $S_{j1}$. Neste caso, ele é chamado de *loop* de dois lados (**two-side *loop***). $L$ é então removido do $LeC$ para manter essa curva conexa, ligando-se $S_{j1}.prev$ a $S_{j2}$. Observe que neste caso, os vértices de $PC_k$ adjacentes ao *loop* são mantidos na definição do $LeC$.

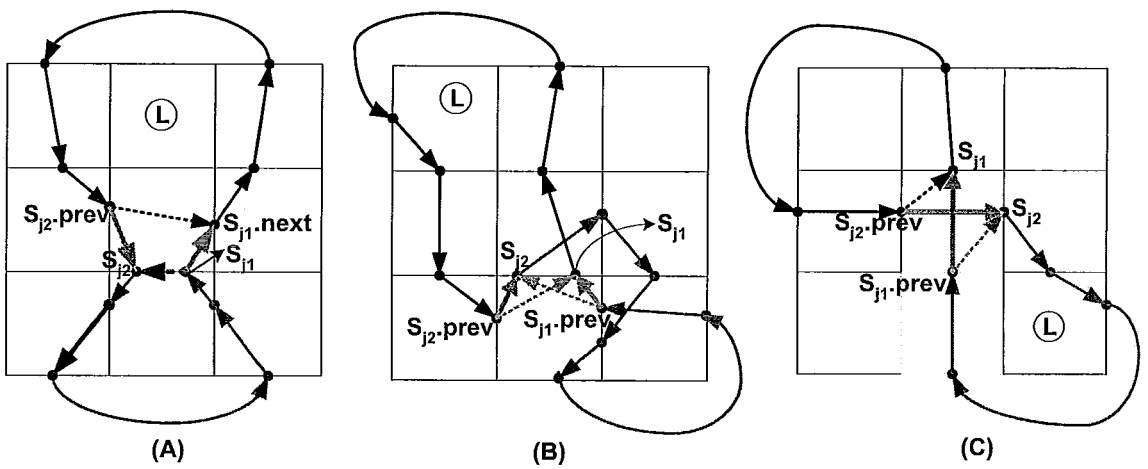


Figura 4.2: Os três tipos de *loops*

- O $LeC$ extendido tem um cruzamento na célula revisitada C. Neste caso, $PC_k$ tem uma auto-intersecção em $C$ que não pode ser removida por sua substituição por uma $\mu$-curva equivalente a ela. O *loop* $L$ que deve ser formado, neste caso, é obtido pela ligação de $S_{j2}.prev$ a $S_{j1}$. Além disso, deve-se remover $L$ do $LeC$ e, para manter essa curva conexa, liga-se $S_{j1}.prev$ a $S_{j2}$. Veja a figura 4.2 (C).

Se um *loop* é criado, associa-se um novo nó do grafo $G$ a ele. Tendo-se com-

pletado o processo indicado acima, continua-se a percorrer $PC_k$ - ou continuar gerando vértices de $PC_k$ se percorrê-lo tiver o significado de construí-lo gradualmente. Repete-se o processo acima cada vez que $LeC$ se tornar não-regular. Ao fim da circulação da $PC_k$ - ou quando ela estiver totalmente construída, determinar as arestas de $G$ como se segue:

Para cada nó $n$ de $G$, inicializar uma linha poligonal $P_n$ com o *loop* associado a $n$. Executar:

```
enquanto (NS não for vazia) faça
{ para cada estrutura fina ns em NS faça
  { se as adjacências v_1 e v_2 de ns
    pertencem a linha poligonal de um nó:
    substituir[v1, v2] por ns na linha poligonal;
    remover ns de NS;
  }
}
```

Então, para cada par de nós - $N_1$ e $N_2$, criar uma aresta de $N_1$ a $N_2$, se e somente se as adjacências de $N_1$ estão na linha poligonal de $N_2$.

O grafo obtido por este processo não contém ciclos e pode se tornar conexo pela adição a $PC_k$, se necessário, de um vértice inicial artificial que não esteja contido em nenhum *loop* e associar um nó a este vértice. Assim, com esta medida simples o grafo se torna uma árvore onde cada nó está relacionado a um *loop*. Desse modo, podemos chamá-la de *loop-tree* de $PC_k$. Ainda, denotaremos $LT(PC_k)$ para se referir a loop-tree de $PC_k$ obtida pelo processo acima quando a curva é percorrida no sentido horário a partir do primeiro de seus vértices a ser determinado. Chamamos atenção para o fato de que as *loop-trees* de diferentes topologias ou com a mesma topologia, mas com diferentes associações *loop*-nó,

podem ser obtidas se o processo descrito acima é aplicado a $PC_k$ considerando um ponto inicial diferente e/ou a outra direção de circulação na curva. Se os *loops* abertos forem dependentes desses fatores, então as *snakes* do próximo estágio também serão e, em consequência, os resultados de toda a evolução das *snakes*. Isso seria um claro inconveniente, mas, felizmente, tal não pode acontecer se $\Gamma_k$ é adequada.

Façamos $\delta_1, \cdots, \delta_M$ serem os contornos das componentes conexas de um conjunto de pontos que ainda não foram visitados pelas *snakes* até o momento em que $PC_k$ é totalmente formada. Se $\Gamma_k$ é adequado, cada $\delta_i$ será $\mu$-equivalente a um *loop* representado em cada árvore de *loop* de $PC_k$ não dependendo do ponto inicial e da circulação utilizada para obtê-la. Podemos, então, definir um *loop* aberto de uma árvore de *loops* como todo aquele que é $\mu$-equivalente a um $\delta_i$ e, neste caso, para cada *loop* aberto de uma árvore de *loops* de $PC_k$ haverá um, $\mu$-equivalente a ele, em outra árvore de *loops* desta curva. Assim, se identificamos corretamente estes *loops*, o conjunto de *snakes* do próximo estágio gerado empregando-se uma dada árvore de *loops* cruzará as mesmas arestas que o conjunto de *snakes* gerado a partir de outra árvore de *loops*. Isso é plenamente satisfatório no nível de precisão em que estamos trabalhando.

## 4.2 Rotulando os *Loops*

Após a explicação de como são formados os *loops*, se faz necessário explicar como eles são rotulados. Para descrever como o rótulo de um *loop* pode ser obtido de acordo com o rótulo de um filho na árvore de loops da curva $LT(PC_k)$ é preciso introduzir o conceito de $\mu$-intersecção.

Dados os *loops* $L_1$ e $L_2$, que são adjacentes em $LT(PC_k)$, definimos $P_{12}$ como a linha poligonal mais curta, contida em $PC_k$ e tendo a mesma orientação, que liga

um ponto $L_1$ a um ponto $L_2$. Definimos $P_{21}$ de forma análoga, exceto que ela liga um ponto de $L_2$ a s um ponto de $L_1$. Remover os $\mu$-whiskers destas curvas. Se os $P_1$ e $P_2$ resultantes se interseptam e substituindo cada um por alguma curva $\mu$-equivalente, esta interseção continua a existir, então dizemos que $L$ e $L'$ têm uma $\mu$-interseção. Os *loops* $L_1$ e $L_2$ representados na figura 4.4 possuem uma $\mu$-interseção. Observe que o fato de $L_1$ e $L_2$ serem adjacentes implica que a linha poligonal definida pelos vértices $P_{12}$ e $P_{21}$, excluídas as extremidades de ambas, é uma estrutura estreita.

Removendo os *$\mu$-whiskers* contidos nela, a curva se torna um gargalo. Isto é o que ocorre entre as ilustrações da figura 4.4.

Para ver como verificar a existência de uma $\mu$-interseção de uma maneira computacionalmente eficiente, consideramos, inicialmente, a função $X_c$ que é definida a seguir:

Dados três pontos $s_1$, $s_2$ e $s_3$ pertencentes a diferentes arestas de uma célula $C$, definir $X_C(s_1, s_2, s_3) = 1$ se $s_1$ é alcançada antes de $s_2$ quando, iniciando-se no ponto $s_3$, a borda de $C$ é percorrida no sentido horário. De outro modo, $X_C(s_1, s_2, s_3) = 2$.

Como os três pontos estão em arestas diferentes de $C$, $X_C$ não depende da localização precisa destes pontos, mas somente das arestas nas quais eles estão. Faça $E_i \in \{0, 1, 2, 3\}$ ser o valor indicativo de qual aresta contém $s_i$, $i = 1, 2, 3$ se a mesma convenção usada para definir coordenadas das arestas for adotada. Suponhamos, por enquanto, que os pontos $s_i$ sejam vértices de uma curva-$\mu$ $S$. Neste caso:

1. $E_i$ será a aresta coordenada de $s_i$, chamada $E(s_i)$, se $S$ entra em $C$ por $S_i$.

2. $E_i = E(s_i)^{-1}$ se $S$ sai de $C$ em $s_i$.

$X_C(s_1, s_2, s_3)$ pode ser calculada pelo algoritmo abaixo. Cada um dos casos

apontados no algoritmo está representado na figura 4.3.

```
function Xc(s1, s2, s3)
  if(E1==E3 mod 2)
    if(E2==(E3+1) mod 4)
      then X=2; // Caso 4
    else
      then X=1; // Caso 3
  else
    if(E1==(E3+1) mod 4)
      then X=1; // Casos 1 e 2
    else
      then X=2; // Casos 5 e 6
end function
```

Utilizando a função $X_C$, podemos verificar a existência de uma $\mu$-interseção entre um filho $L'$ e um pai $L$ da seguinte forma:

1. Se as adjacências de $L'$ pertencem a $L$, o que somente é possível nos casos representados na figura 4.2, a $\mu$-interseção deve existir.

2. Por outro lado, seja $C_2$ a célula contendo as extremidades de $L_2$. As extremidades são $s_{j1}$ onde $P_{12}$ termina e $s_{j2}$ onde $P_{21}$ começa. Ainda, seja $C_1$ a referência à célula contendo as outras extremidades de $P_{12}$ e $P_{21}$ - $s_{j0}$ e $s_{j3}$ respectivamente - ambas em $L_1$.

3. Calcular $X_{C1}(s_{j1}, s_{j2}, s_{j1}.prev)$ e $X_{C2}(s_{j3}.next, s_{j0}, s_{j3})$. Elas são iguais se e somente se $L_1$ e $L_2$ têm diferentes orientações, como pode ser visto na figura 4.3. Isto, entretanto, somente é possível se a $\mu$-interseção existe.

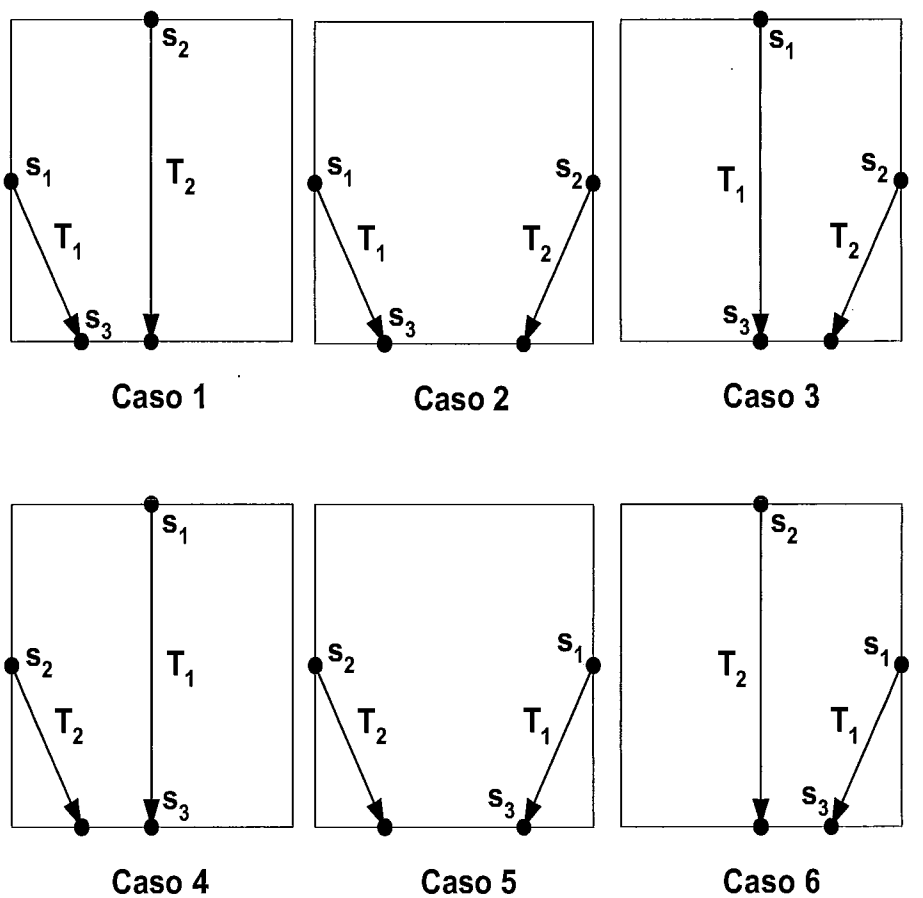


Figura 4.3: Casos considerados no cálculo de $X_c$

Temos, então, um processo para verificar se dois *loops* adjacentes têm uma $\mu$-intersecção que exige no máximo 5 testes. Calcular $X_C$ exige, no máximo, 2 testes. Como o cálculo deve ser feito duas vezes e os resultados comparados, o número total de testes é arredondado para 5.

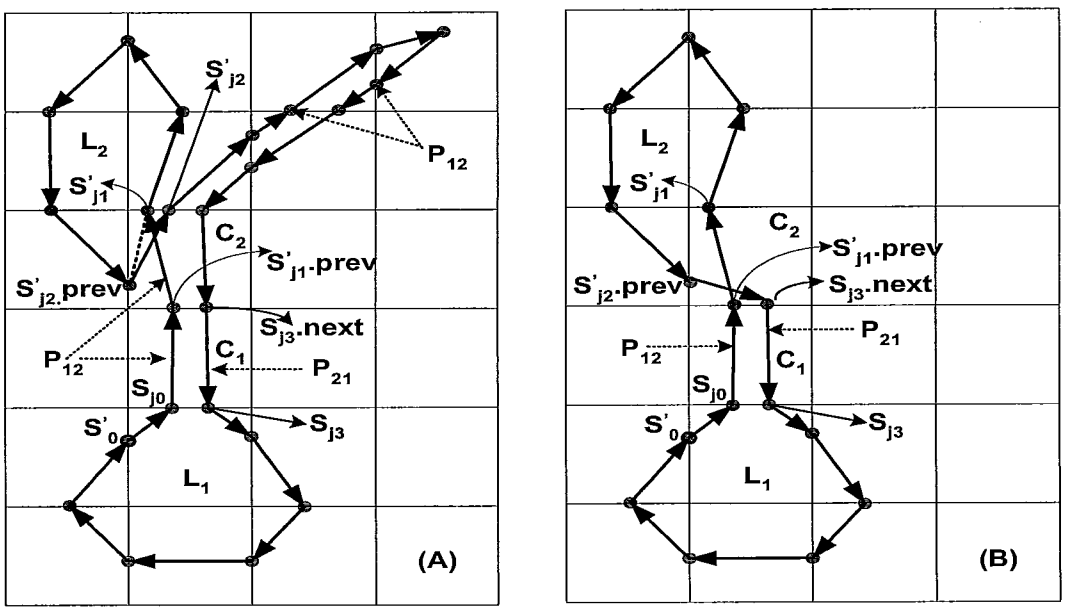


Figura 4.4: Verificando a existência de uma $\mu$-interseção.

Agora, é possível descrever como um *loop* pode ser corretamente rotulado em tempo $O(1)$ se $\Gamma_k$ tem as propriedades necessárias. Os seguintes fatos relativos aos elementos de $LT(PC_k)$ podem ser empregados:

**Fato 1** Se $\Gamma_k$ é ideal, então toda folha de $LT(PC_k)$ é aberta.

**Fato 2** Se um *loop* e um filho não têm uma $\mu$-interseção, seus rótulos são os mesmos.

**Fato 3** Se $\Gamma_k$ é adequada, então um *loop* que tenha uma $\mu$-intersecção com um filho aberto na $LT(PC_k)$ é fechado.

**Fato 4** Supondo que $\Gamma_k$ seja adequada e que $L$ seja um filho fechado do *loop* $L'$. Então $L'$ será fechado se e somente se existe uma célula $C$, diferente da célula inicial de $L$ ($C_L$), tal que tanto os segmentos de $L'$ e $L$ em $C$ formam uma cruz ou ambos $L$ e $L'$ cruzam a mesma aresta em $C - C_L$.

Usando estes 4 fatos, se $\Gamma_k$ é ideal, os loops de $PC_k$ podem ser rotulados como se segue: as folhas da $LT(PC_k)$ são *loops* abertos. *Loops* que não tenham uma $\mu$-interseção com um filho, recebem o mesmo rótulo deste filho. Para loops que tenham interseções-$\mu$ com todos os filhos, deve ser aplicada uma verificação especial. Esta consiste em verificar se um dos filhos é aberto, e, neste caso, o *loop* é fechado. Se o *loop* tem apenas filhos fechados, ele será, então, também fechado, se uma das condições do fato **fato 4** for verdadeira para qualquer um dos filhos. De outro modo, o *loop* será aberto.

Usando um esquema um pouco diferente, a rotulação de um loop que não seja folha pode ser obtida somente em função do rótulo do último filho dele a ser determinado. Este último filho é sempre o último loop a ser encontrado antes de $L$.

Em relação às condições **fato 4**, devemos observar que verificar se um pai e filho cruzam a mesma aresta de uma célula $C$ ou tenham um cruzamento nela, pode ser feito em tempo constante, simplesmente comparando as coordenadas das arestas dos vértices dos dois *loops* que estão em $C$. Como uma célula satisfazendo as condições **fato 4** deve ser adjacente a $C_L$, essas condições podem ser verificadas em $O(1)$.

Para verificar se $L$ e $L'$ têm uma $\mu$-interseção em $O(1)$, podemos utilizar o procedimento proposto acima. Isto, entretanto, requer em particular que se conheça $s_{j0}$ e $s_{j3}$, as extremidades das linhas poligonais $P_1$ e $P_2$ que estão no nó pai. Suponhamos que seja adotado o esquema no qual um loop pai $L$ é rotulado somente em função da rotulação do seu último filho a ser gerado - $L'$. Neste caso, para

identificar as extremidades de $P_1$ e $P_2$ que estão no pai, é necessário verificar, para cada estrutura estreita $ns$ criada depois de $L'$, se esta está entre $L$ e $L'$. Isto é verificado comparando os índices dos vértices em $L'$, $ns$ e $L$. $s_{j0}$ e $s_{j3}$ serão as extremidades da última estrutura estreita entre $L$ e $L'$ a ser encontrada.

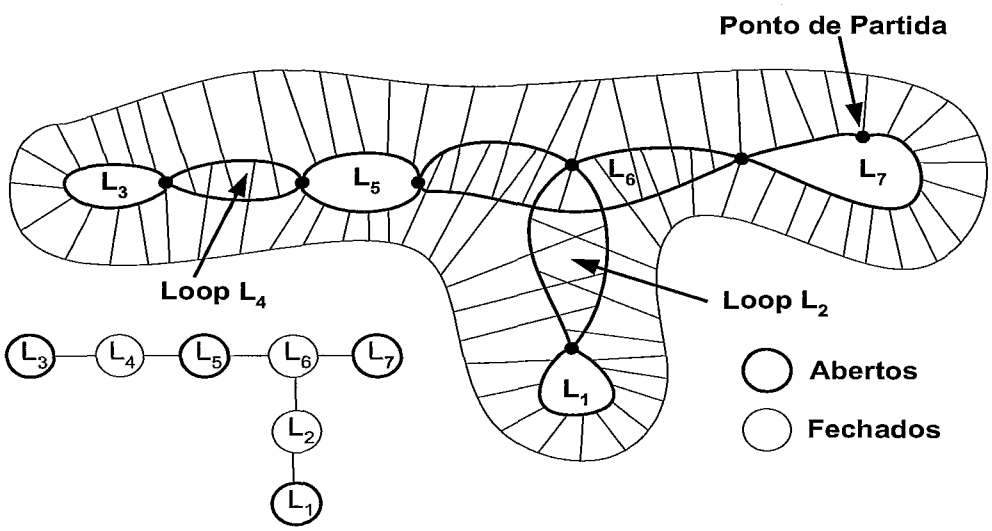


Figura 4.5: Um exemplo do processo de rotulação quando $\Gamma_k$ é ideal. As folhas $L_1$ e $L_3$ são abertas. Portanto, seus pais $L_2$ e $L_4$ são fechados, respectivamente. Como $L_5$ é disjunta de $L_4$, ela é aberta. O *Loop* $L_6$ pode ser rotulado como fechado, seja porque ele tem um filho aberto $L_5$ ou um filho fechado $L_2$ que é cortado por ele mesmo. Finalmente, como o nó raiz $L_7$ é disjunto do filho fechado $L_6$, ele é aberto.

Se $\Gamma_k$ é somente adequada, é preciso de um modo de rotular as folhas de $LT(PC_k)$, uma vez que o **fato 1** não pode mais ser usado. Para que seja possível rotular uma folha neste caso, e evitar, o quanto possível, a verificação das condições **fato 4** ou a existência de $\mu$-interseções - que são os passos mais custosos do esquema apresentado - deve ser adotada uma estratégia alternativa de rotulação dos *loops*. Esta estratégia é baseada no seguinte raciocínio:

1. Supor que a *snake* $S_k$ é percorrida no sentido horário e considerar que um *loop* de $PC_k$ é percorrido na direção dada pela ordem em que seus vértices são gerados. Neste caso, *loops* abertos são também percorridos no sentido horário, enquanto *loops* fechados são percorridos no sentido anti-horário. Desse modo, pode-se determinar o rótulo de um *loop* a partir do sentido em que ele é percorrido.

2. Agora, seja $q$ um ponto na aresta $[s_{j1}, s_{j2}]$, sendo $s_{j1} < s_{j2}$, de um *loop* $L$. Seja $p$ outro ponto tal que $[p, q)$ é externo a $L$. Neste caso, a direção em que $L$ é percorrido pode ser derivada da posição relativa de $[p, q)$ em relação ao segmento orientado $\overrightarrow{[s_{j1}, s_{j2}]}$. Se ele está do lado esquerdo, o *loop* é atravessado no sentido horário, o que significa que ele é aberto. Se ele está do lado direito do *loop*, o *loop* é percorrido em sentido anti-horário e, conseqüentemente, é fechado. Desse modo, tudo que é necessário para rotular um *loop* é encontrar um par de pontos como $p$ e $q$.

3. Se $\Gamma_k$ é limitada pela malha, então tal par de pontos pode ser obtido, para um *loop* de $PC_k$, do seguinte resultado:

**Fato 5** Chamamos um *loop* $L$ de $PC_k$ **antecipado** se o antecedente de seu vértice inicial $s_{j1}.prev$ pertence à célula inicial de $PC_k$. Se $L$ é não é um *loop* **antecipado** de $PC_k$ e $s_{j1}$ é seu vértice inicial, então a aresta - $e$ -da malha contendo $s_{j1}.prev$ está totalmente fora de $L$.

Façamos $s_{j2}$ ser o vértice final de $L$ e $e_i$, $i = 1, 2$, a aresta da malha na qual está $s_{ji}$. Tanto $e_1$ como $e_2$ têm um nó $v$ em comum com $e$. Se $v \in e_i$, então $[v, s_{ji})$ é externo a $L$ e para marcá-lo, tudo que é necessário, é determinar em qual lado de $\overrightarrow{[s_{j1}, s_{j2}]}$ o nó $v$ está. De acordo com o modo como as coordenadas das arestas são definidas, $v$ está do lado esquerdo, o que significa que $L$ é aberto se e somente se:

```
E(sj1.prev)=(E(sj1)-1)mod 4 or (E(sj2)-1)mod 4
```

Caso contrário, ele será fechado. Assim, um *loop* não **antecipado** pode ser rotulado com somente dois testes envolvendo as coordenadas das arestas de suas extremidades.

Para explicar como *loops* **antecipados** podem ser rotulados em $O(1)$, alguns conceitos são necessários. Observe que se $PC_k$ é um *loop*, ele será **antecipado**.

Dada uma sub-estrutura $S$ de $PC_k$, *loop* ou estrutura estreita, chamamos o conjunto de células que são adjacentes àquelas contendo as extremidades de **vizinhança crítica** desta sub-estrutura. Como $\Gamma_k$ é limitada pela malha, somente vértices na vizinhança crítica de $S$ podem ser parte de outras sub-estruturas que tenham sido encontradas depois da determinação de $S$. $S$ é chamada **curta** se ela está totalmente contida na sua vizinhança crítica. Caso contrário, é chamada **longa**. Na figura 3.12, o *$\mu$-whisker* $A$ é curto, enquanto $B$ é longo.

Podemos obviamente determinar o rótulo de um *loop* curto e percorrer totalmente uma estrutura estreita curta em $O(1)$. Seja $L_1$ a primeira estrutura longa a ser encontrada, **antecipada**. Apagar as indicações aos seus vértices nos elementos da $TCS$ que são relativos às células da sua vizinhança crítica. Então, mover o ponto inicial de $PC_k$ ao vértice $s_m$ de $L_1$ tendo o último índice dentre os que estão fora da vizinhança crítica. Deste momento em diante, cada *loop* encontrado, exceto o que contém $s_m$, será não-antecipado e, conseqüentemente, pode ser rotulado pela utilização do **fato 5**. Se existirem outros *loops* juntamente com os contendo $s_m$, ele terá filhos na árvore de *loops* e, neste caso, seu rótulo pode ser obtido a partir do rótulo de qualquer dos seus filhos. Assim, o único caso que não é resolvido pela movimentação do ponto inicial de $PC_k$ para $s_m$ é o que $L_1$ é o único *loop* de $PC_k$.

Quando há um único *loop* que é **antecipado**, o contorno de $PC_k$ pode não conter informação suficiente para sua rotulação em $O(1)$. Mesmo neste caso, po-

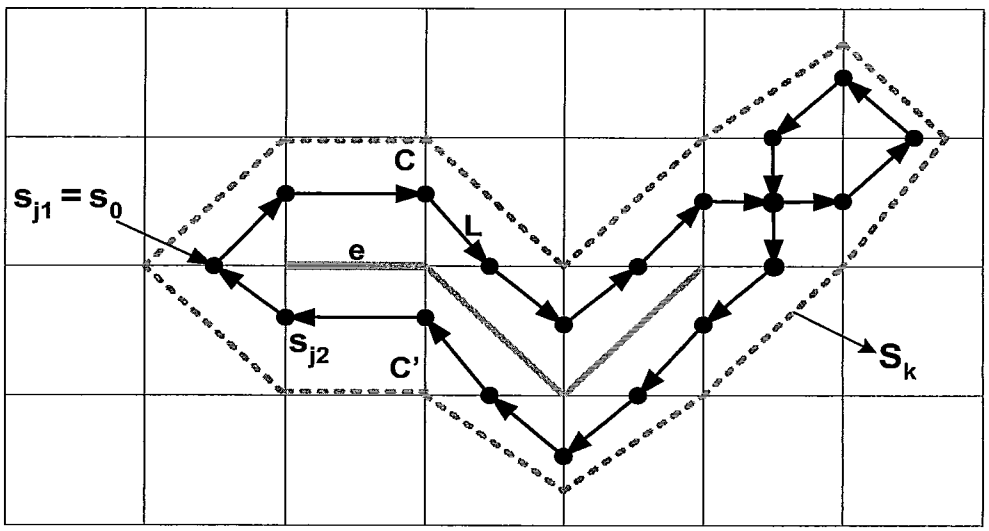


Figura 4.6: Um *Loop* $L$ fechado antecipado, a aresta $e$ e as células $C$ e $C'$

demos rotular um *loop* aberto se ele não corta todas as quatro células adjacentes ao nó esquerdo de uma de suas extremidades. Se ele corta todas as quatro células, entretanto, pode ser necessário saber a posição de alguns *snaxels* de $S_k$. O processo a ser aplicado neste caso faz uso de algumas propriedades dos *loops* fechados. Suponhamos que $PC_k$ tenha um único *loop* $L$ que é fechado. Um *loop* fechado pode envolver um único caminho $P$ formado pelas arestas da malha e as diagonais de suas células duplas, como pode ser visto na figura 4.6. Então, se $L$ tem mais do que quatro vértices, o caminho não colapsará num ponto e, conseqüentemente, vai conter ao menos uma aresta ou diagonal da malha. Uma das extremidades de $P$ - $v$ - é um nó da célula contendo as extremidades de $L$. Seja $e$ o segmento de $P$ adjacente a esta extremidade. Suponha, inicialmente, que $e$ é uma aresta da malha. Neste caso, considere duas células $C$ e $C'$ adjacentes a ela. Ao menos uma delas, digamos $C$, contém tanto uma aresta de $S_k$ quanto uma aresta de $PC_k$. De outro modo, ambas as células vão conter os pontos finais de $\mu$-*whiskers* e $L$ teria sido dividida em uma delas. O rótulo de $L$ pode, então, ser derivado das coordenadas das arestas dos vértices de $PC_k$ e dos *snaxels* de $S_k$ em $C$ como se segue:

1. Se $C$ é dupla para $S_k$ ou $L$, então estas curvas terão segmentos em $C$ cujas extremidades estão nas mesmas arestas da malha. Desde que $L$ seja fechada, a orientação destes segmentos deve ser diferente. Caso contrário a hipótese é falsa e $L$ deve ser aberta.

2. Se $C$ é simples para $S_k$ e $L$, considerar as interseções $C_S$ e $C_L$ de $S_k$ e $L$ com $C$. $L$ será fechado se e somente se uma das seguintes condições for válida:

   (a) O vértice terminal de um destes segmentos está na mesma aresta do vértice inicial do outro.

   (b) Se $C$ não possui arestas contendo uma extremidade de $C_S$ e $C_L$ então seus vértices inicial ou terminal estão em arestas opostas.

Assim, tendo encontrado uma célula simultaneamente cortada por $S_k$ e $L$, é possível encontrar o rótulo de $L$ em complexidade $O(1)$. O problema é que, devido a possível existência de *$\mu$-whiskers*, que devem ser removidos para tornar $L$ uma $\mu$-curva regular, ir de uma das extremidades de $S_k$ ao seu segmento $e_C$ em $C$, precisamos passar por um número de *snaxels* que não pode ser limitado por uma constante. Entretanto, se um destes *$\mu$-whiskers* é longo, podemos, uma vez mais, mudar o vértice inicial de $PC_k$, fazendo então $L$ não **antecipado**. Ainda, como $L$ é **antecipado**, existe uma extremidade $s_{ext}$ de $S_k$, tal que o número de *$\mu$-whiskers* que deve ser percorrido para ir de $s_{ext}$ a $e_C$ é um número pequeno. Se todos eles devem ser curtos, o número de *snaxels* entre $s_{ext}$ a $e_C$ é também limitada por uma constante.

Se $e$ é uma aresta diagonal, façamos $C$ ser uma das duas células adjacentes a $v$ que não seja dupla para $L$, e repetir o raciocínio anterior. Considerando que $S_k$ é uma $\mu$-curva regular, em qualquer das duas definições possíveis de $C$, o número

de *snaxels* entre $s_{ext}$ a $e_C$ é teoricamente limitado por 12. Na maioria dos casos, entrentanto, tanto $s_{ext}$ está em $C$ ou é adjacente a um *snaxel* em $C$.

Assim, todos os casos foram resolvidos e podemos, finalmente, dizer que a abordagem descrita não é somente $O(1)$ por *loop*. Suas operações mais custosas, como verificar se existe uma $\mu$-interseção entre dois *loops* ou procurar por uma célula cortada por um *loop* e $S_k$, são feitas apenas uma vez por iteração.

## 4.3 Encontrando e rotulando *Loops* passo-a-passo

Por motivo de simplicidade, vamos assumir que a **Estrutura de Controle Topológico** contém, para cada célula $C$, um registro composto de dois campos. O primeiro, *Last_Stage_in[C]* contém uma codificação em dois bits. O primeiro bit indica se $C$ já foi cortada pela curva projetada numa iteração até a atual. O segundo bit indica a paridade da iteração em que a última visita ocorreu. Observe o fato de que uma célula, cruzada por $PC_{k-2n}$ mas não visitada por $PC_k$, tem o mesmo código daquelas que já foram cruzadas pela curva projetada. Isto não origina a criação de falsos *loops*. Uma vez que o processo seja de contração, $PC_k$ não cruza qualquer célula que foi visitada pela última vez na iteração anterior $k-1$.

Agora, seja $S_C$ definida como se segue: se o $LeC$ corrente cruza $C$, ele é o último vértice do $LeC$ em $C$. Caso contrário, ele é, dentre os vértices em $C$ das curvas projetadas em todas as iterações, desde o início do processo, o último a se tornar vértice de um *loop*. Ainda, seja $k_C$ a iteração na qual $S_C$ foi determinada. O segundo campo do registro *Last_Vertex[C]* contém o índice de $S_C$ na lista de vértices da curva projetada da iteração $k_C$. Obviamente, este campo só é útil se $k_C$ está na iteração atual.

Embora opções mais compactas sejam claramente possíveis, a lista de vértices

de $PC_k$ pode ser estruturada como uma lista duplamente encadeada, na qual cada vértice esteja num registro contendo sua aresta, coordenadas do *pixel* e ainda, um ponteiro especial (*.old*). Este é usado na seguinte situação: suponha que o segundo segmento de um $LeC$ em uma de suas células duplas - $C_{double}$ - seja removido, devido a criação de um *loop* que o contenha, enquanto o primeiro segmento, que não é parte do *loop*, permanece.

Devemos, então, atualizar *Last_Vertex*[$C_{double}$] substituindo o último vértice do segundo segmento - $S_{j2}$ - pelo último vértice do primeiro, $S_{j1}$. Para fazer a substituição mais diretamente, $S_{j1}$ é indicado por $S_{j2}$. Este ponteiro pode também ser usado para identificar qual célula é uma célula dupla do $LeC$. Se seu valor é nulo, a célula é uma célula simples.

Podemos agora tornar operacional cada passo executado no processo de encontrar/criar os *loops*, apresentado na seção 4.1. Façamos *Current_Vertex* e *Current_Cell* se referirem, respectivamente, ao último vértice de $PC_k$ que foi gerado e para a célula contendo este vértice e o anterior.

O fato de $PC_k$ estar revisitando a célula atual é identificado pela verificação de que *Last_Stage*[*Current_Cell*] contém a paridade do estágio atual.

Remover um *loop* $L$ significa evitar que qualquer um dos seu vértices esteja contido em qualquer *loop* ou estrutura estreita determinada posteriormente. Este objetivo pode ser alcançado apagando na **Estrutura de Controle Topológico** cada referência aos vértices de $L$. Entretanto, um vértice $v$ de $L$ pode somente estar em um *loop*, uma estrutura estreita ou mesmo ser adjacente a um deles, se e somente se ele pertence à vizinhança crítica de $L$. Somente as referências a vértices dessa vizinhança devem ser apagados. Como o seu número é aproximado por uma constante, remover um *loop*, não interessando seu tamanho, é uma operação de complexidade $O(1)$.

Um *loop* de dois lados pode ser detectado verificando a seguinte condição:

*E(Current_Vertex)=(E(Last_Vertex[Current_Cell])),*

e um *loop* de um lado por:

*E(Current_Vertex)=E(Last_Vertex[Current_Cell].prev).*

Quando ambas as condições não são válidas, uma célula com um cruzamento pode ser identificada por:

*E(Current_Vertex)=E(Current_Vertex).prev,*

caso contrário, a célula é uma célula dupla do $LeC$.

Neste caso, as células duplas, definidas na seção 3.3, podem ser diferenciadas das comuns pela condição:

*E(Current_Vertex)=(E(Current_Vertex.prev)-1)mod 4.*

Assim, quase todas as decisões podem ser tomadas por comparação de coordenadas de arestas. Índices de vértices são usados essencialmente para decidir se um *loop* é descendente de outro ou se uma estrutura estreita está entre pai e filho. Dado um *loop* ou estrutura estreita $S$, façamos o *intervalo de* $S$ se referir ao intervalo definido pelos índices dos seus vértices extremos. Uma vez que $LT(PC_k)$ é construída a partir das folhas, um loop $L_1$ é um descendente de um loop $L2$ se e somente se o intervalo de $L_1$ está contido no intervalo de $L_2$. De forma similar, uma estrutura estreita $ns$ está entre um filho $L_1$ e seu pai $L_2$, se e somente se o intervalo de $L_1 \subseteq$ no intervalo de $ns \subseteq$ no intervalo de $L_2$.

# Capítulo 5

# Avaliação e Segmentação

A abordagem introduzida neste trabalho explica detalhadamente como podem ser alcançadas as melhorias no controle topológico das *loop-snakes*, inclusive descrevendo a complexidade $O$ de várias funções propostas. Contudo, uma avaliação meticulosa se faz importante no processo de segmentação. Para isso, é proposta aqui uma maneira de se avaliar o esforço computacional total, baseado nos *snaxels* separadamente.

Os *snaxels* devem ser classificados em função do número de operações relativas ao controle topológico que foram realizadas no momento do seu processamento. Um agrupamento possível pode ser dado pelas seguintes classes de *snaxels*:

A) *snaxels* cuja inclusão determina que $PC_k$ entra numa célula nula;

B) *snaxels* nos quais é necessária uma ação corretiva para manter $\Gamma_k$ adequada;

C) *snaxels* nos quais $PC_k$ revisita uma célula.

Seguindo uma única classificação, como a dada anteriormente, pode-se computar os *snaxels* para diferentes imagens, por exemplo: sintéticas, com ruído,

vários segmentos etc. Com isso, é possível obter uma classificação dos *snaxels* especificamente em relação a cada tipo de imagem, verificando para quais delas é mais adequado o método proposto neste trabalho.

Já para avaliação da qualidade da segmentação, uma maneira de exibir as imagens segmentadas é o ideal. Para isso, foi construída uma interface gráfica que será brevemente descrita a seguir.

## 5.1 Interface

Com a finalidade de facilitar os testes dos algoritmos, melhorar o controle dos parâmetros e permitir a utilização amigável, foi criada uma *GUI (do inglês, Graphical User Interface)*. Por se tratar de um processo inerentemente visual, a segmentação é melhor verificada e avaliada com a exibição da imagem a ser segmentada e o contorno gerado pela *snake*. A necessidade de se apresentar os resultados e facilitar a utilização das *snakes* foi mencionada em [16]. Na figura 5.1 é mostrada a interface juntamente com a janela gerada para exibição da imagem.

Além da exibição final da imagem segmentada, a utilização da interface permite o acompanhamento visual do processo de segmentação. Isso é fundamental para verificar o comportamento dos *snaxels* e, a partir disto, modificar e fazer ajustes paramétricos inerentes ao modelo de *snakes* utilizado. Obviamente, ajustes de exibição como: linhas, pontos, cores e quaisquer outros são facilmente fornecidos via interface, cooperando para a avaliação da qualidade da segmentação.

A seguir, explicaremos brevemente a construção da interface e as bibliotecas que foram utilizadas.

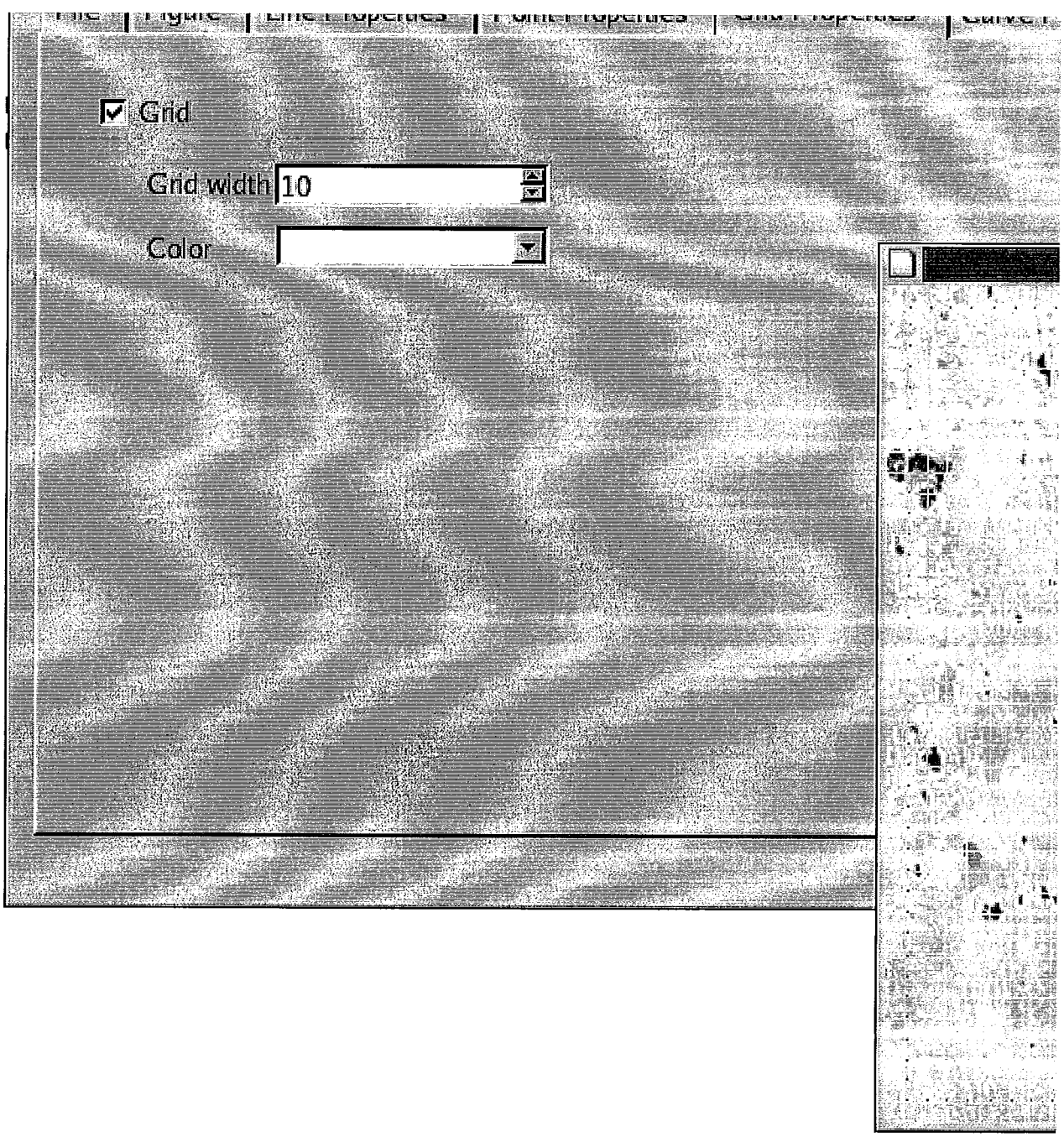


Figura 5.1: Interface gráfica em Qt/OpenGL exibindo a janela da imagem a ser segmentada.

### 5.1.1 Qt

A classificação e avaliação de interfaces é objeto de estudo em outras áreas em Ciência da Computação. Por isso, nos limitamos aqui a descrever vantagens de determinado *framework*, unicamente no contexto deste trabalho e dos recursos computacionais disponíveis.

O Qt [6], desenvolvido pela empresa Trolltech, é um *framework* amplamente utilizado na construção de interfaces gráficas de uso geral. Dentre as suas vantagens, é inteiramente desenvolvido em C++, o que torna transparente e natural sua utilização com código C/C++, que é o caso das implementações dos algoritmos propostos. Outra vantagem, especialmente no contexto deste trabalho, é a de que o Qt possui componentes específicos que englobam o *OpenGL*, utilizados para a exibição da imagem a ser segmentada e das próprias *snakes*.

Como desvantagens, devemos salientar que o Qt exige a incorporação de código específico fora do padrão C/C++ para o funcionamento das *mensagens* entre os componentes gráficos (*widgets*) e as funções. Além disso, é necessário um compilador interno do Qt para gerar código específico para seu funcionamento. Felizmente, as diretrizes de compilação do Qt podem ser incluídas nas do compilador do sistema operacional, tornando o processo automático e invisível ao programador, não constituindo em incômodo no processo de desenvolvimento.

### 5.1.2 OpenGL

O OpenGL é a especificação de uma linguagem *cross-plataform* para construção de aplicações de computação gráfica em $2D$ e $3D$. É amplamente utilizada e desenvolvida pela indústria de dispositivos gráficos, com objetivo de explorar ao máximo as capacidades do *hardware* gráfico existente. Isso a torna o padrão *de facto* em aplicações gráficas.

É utilizada neste trabalho via extensões do Qt para mesclar em vídeo a imagem a ser segmentada e os movimentos das *snakes*. São amplas as fontes de informação sobre o OpenGL, sendo uma referência inicial [26].

### 5.1.3 OpenCV

A empresa Intel desenvolve há alguns anos e de forma aberta um conjunto de funções de Visão Computacional e Processamento de Imagens. Estas funções são desenvolvidas para tirar proveito das capacidades específicas dos microprocessadores *Intel* e são distribuídas em forma de uma biblioteca chamada OpenCV - Intel Open Source Computer Vision Library [13].

Esta biblioteca fornece as mais diversas funções e, juntamente com elas, estruturas de dados que facilitam o desenvolvimento e testes de algoritmos. Ela foi utilizada nos primeiros protótipos da interface e nos protótipos de alguns algoritmos, sendo retirada apenas em trechos nos quais o desenvolvimento de partes dos algoritmos descritos neste trabalho a substituiu.

Nesta implementação da interface, a *OpenCV* é utilizada unicamente para abertura de imagens - salientado o uso da estrutura de dados da imagem - e cálculo do gradiente da imagem. Essa decisão vem do fato de que a interface é funcional e, a permanência da biblioteca permite fácil agregação de funcionalidades que podem vir a ser utilizadas não somente em relação às *loop-snakes*, mas em outras implementações, comparações de desempenho etc.

A figura 5.2 mostra a interface executando um algoritmo de *Snakes* fornecido pela *OpenCV*. Este algoritmo foi utilizado inicialmente para os testes da interface e exibição de *snaxels* na malha.

Enfim, a avaliação dos resultados será muitíssimo simplificada pela utilização da interface, sendo a metodologia de avaliação implementada em conjunto, o que levará a um rápido resultado, tanto da segmentação, em termos visuais, quanto da

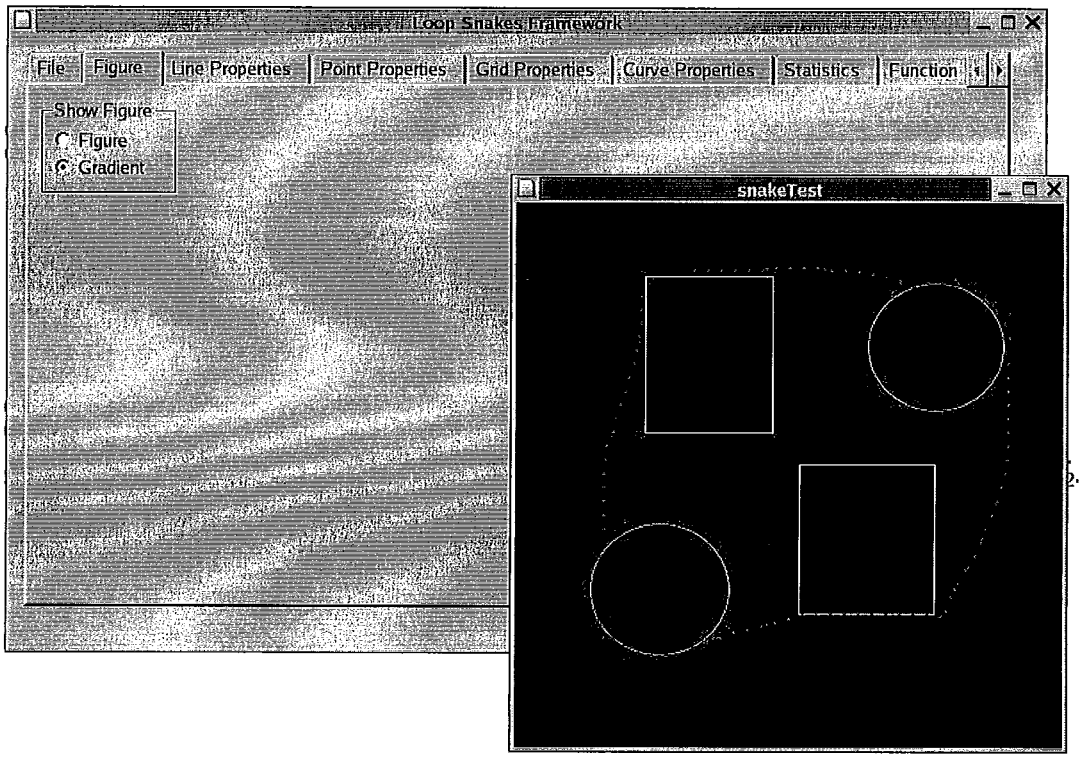


Figura 5.2: Interface e a janela com a exibição da evolução da *Snake*.

complexidade computacional total exigida para o conjunto de imagens de teste.

## 5.2 Evolução e Controle Topológico da *Snake*

Alguns dos métodos apresentados no trabalho foram implementados e produziram alguns resultados iniciais.

### 5.2.1 Evolução da *Snake*

Um dos grandes diferenciais deste trabalho para as metodologias anteriores, consiste na forma com que é computado o deslocamento dos *snaxels*. A contribuição aqui é a proposta de um modelo ótimo de deslocamento, no sentido de que não seja muito conservativo nem "descontrolado", mas o melhor entre as duas alternativas. No primeiro caso, a simplificação obtida pelo deslocamento pequeno no controle topológico, leva a um aumento no tempo de convergência da segmentação, o que desencoraja a utilização do método. Do contrário, deslocamentos grandes dos *snaxels* impossibilitam o controle topológico de modo como é proposto para as *snakes* paramétricas. Isso pode permitir convergência rápida, mas é uma limitação enorme no método, impossibilitando sua aplicação em grande parte dos problemas.

O deslocamento dos *snaxels*, aqui, permanece baseado no modelo clássico de forças interna, externa e elástica. Entretanto, o movimento inicial dos *snaxels* segundo o modelo é apenas uma componente do deslocamento total. Afim de manter as propriedades de um mapeamento adequado, o deslocamento dos *snaxels* deve ser limitado às quatro células adjacentes a $u_i$ - o vértice interno relativo a $s_i$ - *snaxel* $i$.

Este processo foi detalhadamente descrito na seção 3.3, e descrevemos, abaixo, algumas detalhes de sua implementação.

Como resultado deste modelo de deslocamento, aplicamos o algoritmo a uma imagem artificial gerada para testes. Obtivemos, então, o campo vetorial da figura 5.3, cujos vetores estão associados um a um a cada célula da malha. Estes vetores tem direção e sentido da influência dos gradientes das células vizinhas. Além disso, seus valores de deslocamento não ultrapassam o tamanho máximo da célula, mantendo uma restrição fundamental para o controle topológico.

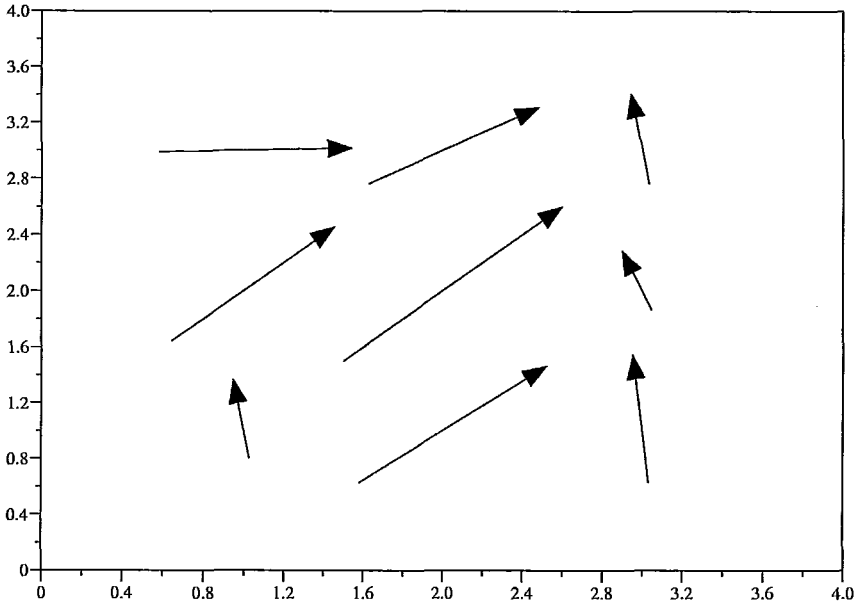


Figura 5.3: Campo vetorial gerado para uma imagem de 40x40 pixels e células de aresta 10 pixels. Esta é uma imagem artificial pequena utilizada apenas para ilustrar os resultados. Os tamanhos dos vetores são proporcionais entre si, mas não em escala da célula.

Somado à utilização do campo, existe ainda um detalhe importante no cálculo dos deslocamentos, que é a utilização dos *snaxels* vizinhos. O cálculo para um

*snaxel* leva em consideração as posições de seus dois vizinhos à frente e dois vizinhos atrás, cada par com um peso diferente. Isso faz com que a curva mantenha uma certa suavidade, mas faz com que seja mais fortemente atraída para os alvos a serem segmentados. Isso é suficiente para garantir as propriedades desejadas para o mapeamento.

### 5.2.2 Controle Topológico

O controle topológico envolve a implementação das diversas funções descritas no capítulo 4. Além da implementação propriamente dita, tais funções exigem um cuidado extra, uma vez que determinadas otimizações são fundamentais para que o algoritmo execute na complexidade proposta no modelo.

Nesta seção serão brevemente descritos os algoritmos utilizados no controle topológico, fornecendo uma descrição em alto nível de como é organizado o processo de controle topológico como um todo. Recordando como se dá o processo de construção do *loop*, temos que o *loop* é a linha poligonal definida pelos vértices da curva projetada que não pertencem a nenhum loop já encontrado, mantida a ordem em que foram gerados. O *loop em construção* é então chamado $LeC$.

Existem dois tipos de *loops*: **válido** e *inválido*. Um *loop* válido é um que está completamente contido no $LeC$, enquanto um não válido compartilha parte de seu contorno com algum *loop* encontrado anteriormente. A diferenciação de ambos é feito de modo relativamente: armazena-se os seus vértices iniciais e finais. O segundo passo é comparar, ao se revisitar um célula, os vértices aos dos filhos do $LeC$. Caso esteja contido, tem-se um *loop* inválido. Caso contrário, um *loop* válido e processado.

Entretanto, para garantir o complexidade $O(m)$, depende-se do armazenamento da informação relativa aos $m$ *loops* já gerados. Desse modo, são utilizados ponteiros específicos, como o *Last_Vertex*, responsável por manter uma ligação

direta entre alguns vértices, evitando que se percorra desnecessariamente a curva projetada.

### Detecção de *Loops* e divisão da curva projetada

Abaixo são apresentadas e explicadas várias funções definidas e utilizados na função que verifica se um vértice está sobre uma aresta repetida. Optou-se por apresentar as funções numa abordagem **bottom-up** para facilitar a compreensão. É fundamental notar que, individualmente as funções apresentadas aqui não resolvem o problema de controle topológico, mas o fazem quando utilizadas em conjunto no algoritmo.

Tem-se, em princípio, as seguintes variáveis :

```
1) boolean Self_Intersection;
2) boolean PrevVertexAndLastVertex_of_PrevCellonSameEdge;
3) vertex LinkVertex;
```

Os nomes significam, respectivamente, que uma curva possui auto-interseção e que a condição de haver, sempre em referência a um vértice atual, que o vértice anterior e o último vértice da célula anterior estão localizados na mesma aresta. Com isso, é possível iniciar procedimentos responsáveis pelo controle topológico. A variável **3** denota o vértice que deve ser conectado ao próximo vértice a ser gerado no $LeC$ para manter a conectividade dos *loops*.

A seguir, a lista de funções:

```
f1) boolean CheckWheterLeCContains(Vertex);
f2) label LabeltheLoopBetween(Vertex, Vertex);
f3) vertex X(Vertex, Vertex, Vertex);
f4) boolean PushOntoStack(Vertex*, Vertex);
```

Em todas as funções, os parâmetros são vértices ou ponteiros para vértices. Na linha **f1** é apresentada a função que verifica se um determinado vértice pertence a um $LeC$ e também é responsável por atualizar uma variável que contém o rótulo que o pai do *loop* no topo da pilha deve ter em função do *loop* atual.

Na linha **f2** a função $Label_the_Loop_Between$ rotula o *loop* localizado entre os vértices inicial e final, passados como parâmetro, retornado uma cadeia de caracteres (ou outro tipo de rótulo).

A função $X$ - linha **f3** - ainda a ser batizada, retorna o vértice final de um gargalo. Dados como parâmetros três vértices (*snaxels*), a função identifica se existe uma $mu$-interseção e retorna a vértice atingido antes ao se percorrer a célula no sentido anti-horário a partir do terceiro. Esta função é fundamental no tratamento de gargalos e identificação de *loops*.

A última função deste bloco, na linha **f4**, simplesmente coloca no topo da pilha informações a respeito do *loop* encontrado, contendo seus vértices inicial e final, rótulo e informações junto ao $LeC$.

Com isso em mente, é possível descrever como é a verificação de um vértice numa aresta repetida. Tendo como parâmetro o último vértice do $LeC$, o primeiro passo é verificar se este vértice pertence realmente ao $LeC$. A função **f1** é então utilizada para esse propósito. Após a confirmação, verifica-se a qual aresta já visitada a curva retorna. Caso o retorno seja à aresta do vértice do parâmetro, verifica-se o valor do *flag* **2**. Sendo falso, o vértice está dando início a um novo gargalo e deve-se rotular adequadamente o *loop* formado. Isso é feito pela função **f2**. Além disso, calcula-se o vértice inicial pela função **f3**. Independentemente do vértice em questão, procede-se com a atualização de variáveis envolvidas no processo **2** e **3**, retornando sempre o valor de verdadeiro para estes vértices.

Uma vez determinado o vértice inicial do gargalo, deve-se determinar o vértice final. Nesse caso, leva-se em consideração a possível existência de uma célula du-

pla. Para isso, é feita uma comparação entre o vértice parâmetro e o $Link_V ertex$ e, caso negativo, executar os procedimentos específicos de fim de *loop*. É fundamental levar em consideração que o $Link_V ertex$ determina o início de outro gargalo, que será tratado tão logo seja tratado o primeiro. Compara-se os vértices - utilizando aqui a função **f3** - e, caso positivo, acrescenta-se o novo *loop* à pilha onde estão todos os filhos do $LeC$. Este acréscimo é feito pela função **f4**. Com isso, temos o fim do tratamento do primeiro gargalo. Utilizando as funções **f2** e **f3** são, respectivamente rolutado o *loop* - obviamente entre os vértices atuais - e definida a aresta em que ele entra na célula. É, então, atualizado o $Link_V ertex$ e a variável **2**. Finalmente, retorna-se verdadeiro.

Continuando com o algortimo, define-se agora a caso em que a função **f1** falha para o teste. Neste caso, deve haver uma $\mu$-interseção. Não havendo, um teste retorna falso e é o fim do algoritmo. De outro modo, a verificação da $\mu$-intersecção é feita verificando-se o vértice corrente e os adjacentes a ele na curva projetada. Como anteriormente, testa-se a variável **2**, obtém-se o vértice final pela função **f3**, testa-se uma auto-interseção pela variável **1** e através de **f4** o *loop* é inserido na pilha. Finalmente, procede-se com a atualização do $Link_V ertex$ afim de reestabelecer a conectividade do $LeC$.

Prosseguimos então com o detalhamento da função que verifica se um vértice está sobre uma aresta não repetida. São dois os casos a serem tratados caso a aresta não seja repetida: a) a célula cruzada imediatamente já foi visitada. b) a célula jamais foi cruzada ou, menos comumente, a aresta já foi cruzada, mas antes da criação do vértice atual da curva projetada.

Como feito anteriormente, serão listadas as variáveis e funções utilizadas neste algoritmo, continuando com a abordagem **bottom-up**.

```
4) boolean PrevCell_has_3_Crossings;
```

```
f5) boolean CheckWheter(vertex, condition);
```

A variável **4**, como o nome já diz, identifica se a célula anterior a célula sendo cruzada no instante atual foi cruzada 3 vezes, o que implica em visita anterior. A função **f5** funciona como a função **f1** descrita anteriormente, mas restrita apenas a identificar duas situações: se está sendo gerada uma célula dupla válida ou verificar uma interseção do $LeC$ com com um filho fechado dele determinado anteriormente. Esta função é utilizada com dois parâmetros, sendo o primeiro o vértice a ser verificado e o segundo qual das duas condições descritas será verificada.

Retornando ao algoritmo completo, ele se inicia com o teste de validade da variável (4). Caso seja falsa, vamos diretamente verificar, pela variável **2** a condição de compartilhamento de aresta. Caso haja o compartilhamento, o algoritmo funciona de modo parecido com a finalização anterior de gargalo. Entretanto, a lista de arestas, neste caso, compreende a aresta do vértice anterior ao do parâmetro, uma vez mais utilizada a função **f3**. As arestas inicias e finais são então comparadas e, caso sejam iguais, o *loop* é colocado na pilha - função **f4**. A conectividade é restaurada entre o $Link_Vertex$ e o vértice parâmetro e é atualizado o vértice da célula corrente.

Sendo a variável **4** verdadeira, parte-se para novo teste, comparando-se a aresta do vértice parâmetro com a aresta do vértice anterior ao vértice parâmetro. Com isso é possível inferir uma possível formação de um *knot-loop*. Sendo esta comparação positiva, verifica-se outra condição, que consiste na falha do teste dos vértices parâmetro e seu anterior. Isto significa que o segmento analisado não pertence ao $LeC$. Desse modo, é utilizada a função **f1** com parâmetros indicando um tipo específico de interseção do $LeC$: dele com um filho seu fechado. que é o obrigatoriedade do vértice Se o *loop* for válido, simplesmente se procede com a rotulação desse novo *loop* com a função **f2**, empilha-se os vértices formadores do *loop* utilizando a função **f4** e, finalmente, restaura-se a conectividade, natural-

mente nos vértices devidos, da curva.

Existe ainda um caso a ser tratado, se as arestas comparadas não forem paralelas. Essa avaliação juntamente com as estruturas já visitadas, permite a identificação de uma célula dupla. Caso esta célula dupla seja válida, seus vértices do loop formado nela são armazenados na pilha, mais uma vez utilizando a função **f4** e é atualizada a estrutura de dados de controle, promovendo-se uma ligação entre o vértice "memória" e o último vértice na célula anterior. Com isto, é finalizado o algoritmo da verificação sobre aresta não repetida.

Assim, finalizamos a explicação dos dois principais algoritmos responsáveis pela detecção de *loops* e divisões na curva projetada. A partir deste ponto, devem ser tratados a validação dos *loops* encontrados, sua rotulação e como se dá o processamento destes *loops* na árvore de *loops*.

### Validação, Rotulação e Processamento de um loop

Todos os procedimentos relativos aos *loops* já identificados e separados - Validação, Rotulação e Processamento - são baseados nas definições e lemas apresentadas no Capítulo 4.

Na subseção anterior foram discutidos dois algoritmos e ambos utilizavam internamente a função:

```
f1) boolean CheckWheterLeCContains(Vertex);
```

Seu funcionamento foi explicado e, nesta seção, será detalhado, uma vez que ela é uma das responsáveis por rotular os *loops* e é peça fundamental no desempenho total do algoritmo. Devido ao seu funcionamento torna-se possível promover todas as operações necessárias ao controle topológico percorrendo-se a curva uma única vez.

A função **f1** recebe um único parâmetro que é um vértice e que pode possuir três significados diferentes.

- Pode ser o último vértice gerado na curva, significando que o $LeC$ retorna a uma aresta.

- Pode ser o vértice final do penúltimo segmento da curva projetada na célula atual. Isso ocorre quando a célula corrente já foi cruzada pelo menos duas vezes e o último segmento da curva projetada encontra-se dentre de um *loop* já encontrado. Tanto neste caso quanto no anterior, é evitada a geração de um falso *loop* por repetição de aresta.

- Pode ser o último vértice da célula gerado antes do antecessor do vértice corrente. Isso acontece quando a célula é revisitada mas os vértices estão em arestas ainda não cruzadas pelo $LeC$. Isso evita a geração de falsos *knot-loops*.

Com isso, é possível distinguir entre as várias mudanças topológicas permitidas de forma que o custo computacional seja da ordem do número de *loops* verdadeiros encontrados em cada estágio.

A distinção de *loops* falsos e verdadeiros pode ser feita de forma simples, com a seguinte seqüência de comandos:

```
while(ParameterVertex < TopOfStack.initialVertex)
{
    Pop(TopOfStack);
    If(ParameterVertex <= TopOfStack.finalVertex)
        return FALSE;
    Else
        return TRUE;
}
```

Para o processo de rotulação, entretanto, a simples retirada dos elementos da pilha traz conseqüências. Isso é devido à necessidade de informação sobre os

filhos de um *loop* para sua rotulação. Caso um *loop* não seja folha, as seguintes premissas são necessárias:

- É suficiente conhecer o rótulo de um único filho, caso este filho não seja separado do pai por uma $\mu$-interseção ou seja aberto.
- saber se um filho fechado é disjunto do *loop* ou o intercepta formando um falso *loop* com ele.

Graças a isso, bastam informações adequadas de um dos filhos de um *loop* para sua rotulação. Estas informações são armazenadas juntamente com os *loops* na pilha, sempre observando a relação do último *loop* com o $LeC$. Todavia, esta informação pode variar no momento da inserção deste *loop* na pilha. Este caso é tratado observando-se a posição relativa do *loop* no topo da pilha em relação ao $LeC$ no caso dele ser fechado. O registro desta mudança é mantido através da substituição do rótulo do topo para "fechado e formando falso *loop*". O rótulo "fechado" é mantido apenas para um *loop* disjunto do $LeC$.

Com os vários rótulos convenientemente atribuídos, juntamente com a informação relativa à existência ou não de uma $\mu$-interseção com o $LeC$, o rótulo de um *loop* pai pode ser diretamente obtido do seu filho. Portanto, com um procedimento simples de comparação, o rótulo é obtido e armazenado de forma global para ser utilizado adiante.

Finalmente, para efetivamente rotular o *loop* completamente, basta verificar os vértices deste e comparar com os do *loop* anteriormente encontrado. Caso seu vértice inicial seja maior do que o último vértice do último *loop* encontrado, ele é uma folha da *loop-tree*, e será rotulado com os métodos de rotulação de *loops* sem pai. Caso contrário, o *loop* receberá o rótulo obtido a partir do último filho do *loop* a ser retirado da pilha.

Além disso, é atualizada a variável:

```
1) FinalVertexOfTheLastLoopFound;
```

que permite detectar possíveis futuras folhas.

Foi completada a descrição do modo como são tratados os *loops* formados, tanto na validação quanto na rotulação. Na próxima seção será apresentado como é gerenciados o processo de re-visita à uma célula.

**Processo de re-visitar uma célula**

O processo de re-visita de uma célula verifica a possível formação de um *loop*. Isso exige um cuidado especial com os parâmetros a serem considerados e a ordem da operações, uma vez que o procedimento pode ser aplicado mais de uma vez numa mesma re-visita. São dois os casos a serem considerados:

- Uma célula $C$ dupla que já foi visitada, o é novamente. Isso implica numa nova interseção da curva projetada com uma aresta já intersectada. Isso pode acarretar na formação de um novo *loop* se a curva projetada tiver um vértice válido na aresta em questão. Para lidar com esse caso, deve-se verificar na estrutura $Last_Vertex$ qual o vértice em questão. É fundamental ressaltar que as estruturas de dados são fundamentais neste procedimento, pois a sua formulação é que permite a verificação do vértice na complexidade de tempo proposta.

- Neste caso, um pouco mais complexo, pode haver a formação de um *knot-loop* ou que a célula anterior tenha se tornado dupla. A verificação destes dois casos é feita através da função:

  ```
  f1) CheckWhetherVertexOnRepeatedEdge();
  ```

  De acordo com os parâmetros, verfica-se - o vértice de entrada da célula atual, num dos casos - se a célula atual foi ou não atingida antes. Situação

idêntica seria constatada no caso da formação de um gargalo, com a geração do último nó e sua entrada numa célula não visitada. A complexidade do procedimento está em evitar que, em toda célula atingida pela curva projetada, a verificação das suas arestas e da célula anterior.

Assim, é feita a verificação que a célula atual contém três vértices pertencentes ao $LeC$ e é gerado o próximo vértice. Verifica-se, então, se a nova célula a ser atingida foi ou não visitada e, caso positivo, em que estágio isso ocorreu. Se sim, é verificada então se o vértice gerado está sobre uma aresta não repetida. Neste momento é executado o procedimento de fim de gargalo, ou a separação de um *knot-loop* ou indicar que a célula anterior é dupla. A ordem como é feita a verificação do vértice anterior e do vértice gerado é fundamental para garantir a completeza das possíveis mudanças topológicas.

Enfim, foram descritos os procedimentos relativos ao controle topológico. É importante ressaltar que este conjunto de procedimentos é muito mais elaborado do que um simples procedimento para traçar uma curva de nível de uma função cujo valor é conhecido nos vértices da malha, como o que dá o traçado final de uma T-Snake. Entretanto, apesar da sua complexidade, o conjunto de procedimentos descritos permite, num único percurso na *snake*, gerar as curvas transformada e projetada, detectando e rotulando os *loops* desta última.

# Capítulo 6

# Conclusão e trabalhos futuros

Uma série de resultados teóricos foram desenvolvidos para permitir a abordagem de controle da topologia de uma $T$-Snake, que é apresentada neste trabalho. Baseado nesta teoria, foi possível criar uma metodologia com um claro ganho computacional em relação aos métodos existentes, podendo ser avaliada pela proposta apresentada no capítulo 5.

A metodologia descrita aqui pode ser considerada "lazy", no sentido de que ela faz o mínimo necessário num *snaxel* simples e adia todas as complicações até o momento que um *loop* é criado. Alguns outros aspectos interessantes dessa metodologia podem ser apontados:

1. Um estágio processa apenas informações obtidas nele mesmo. Isso torna simples refinar a malha durante a evolução da *snake* ou invertendo a direção do seu movimento.

2. Se a curva projetada for a imagem de um mapeamento ideal, então a topologia desta curva determina o rótulo de todos os seus *loops*. Olhar ao redor somente é necessário para tornar mais simples a detecção de onde realizar a divisão.

3. Assim que um *loop* é criado, seu rótulo pode ser determinado. Em abordagens alternativas é necessário esperar que toda a *snake* seja transformada. Isto permite que em uma única volta em $S_k$, as curvas transformada e projetada sejam geradas, *loops* sejam rotulados, e as *snakes* do próximo estágio sejam determinadas.

4. Em relação às possíveis desvantagens de outras abordagens, as curvas geradas não têm viés de direção e a variação de tempo é fixa, não uma função da posição dos *snaxels*.

5. Finalmente, deve ser mencionado que este é um modelo completamente paramétrico. Curvas não são obtidas como *conjuntos de nível* de qualquer função, provando que não é necessário manter sua topologia sob controle.

Uma primeira direção a ser seguida é a finalização da implementação de todas as funções de controle topológico. Com isso, além da comprovação da qualidade da metodologia, será possível compará-la com qualquer outro modelo de *snakes*, bastando para isso, acrescentar funções à interface já desenvolvida.

A extensão natural deste trabalho é generalizar o modelo, de modo que um sistema de várias *loop snakes*, contratoras ou expansoras, possa ser usado. Uma primeira tentativa de obter tal generalização é descrita em [20], mas ainda há espaço para melhorar o modo como se lida com particularidades da junção de contornos. A maioria das dificuldades vem do fato de que, na junção, um *loop* aberto pode ser modificado após sua formação. Entretanto, ainda há questões específicas a serem consideradas, por exemplo, a primeira interseção de uma curva projetada e uma componente conexa formada pela junção de outras curvas projetadas do mesmo estágio, não determinam a criação de um *loop* enquanto as outras sim.

Este fato determina que cada vértice de uma curva projetada deve ter um rótulo para indicar, direta ou indiretamente, a componente conexa na qual ele está.

Manter estes rótulos atualizados é uma versão do problema clássico "union-and-find", que é somente quase-linear. Estes rótulos seriam desnecessários se todas as interseções pudessem ser tratadas da mesma maneira sem o risco de gerar *snakes* que tenham partes em comum com outras ou regiões cortadas que já tenham sido varridas por outras *snakes*.

Existem ainda variantes $2D$ do método a serem exploradas. Uma promissora trata de não projetar a curva na malha, mas simplesmente determinar as células cruzadas por ela. A regularização da *snake* será, então, baseada na distância entre *snaxels* consecutivos.

Para expandir os resultados obtidos aqui para uma Superfície Topológica (**T-Surface**) encontrada em imagens $3D$, algumas dificuldades têm que ser superadas. A estrutura correspondente à árvore de *loops* não é mais um grafo. O *genus* de uma superfície evoluindo pode aumentar sem controle e toda a curva, não mais a introdução de um único vértice em $2D$, é necessária para uma separação.

Esquemas que mesclem características da abordagem descrita aqui e da *T-Snake* padrão podem também ser investigados. Por exemplo, se a superfície projetada na malha é a imagem de um mapeamento adequado, o trabalho de determinar os nós a serem "queimados"é enormemente simplificado.

# Apêndice A

# Apêndice - Regularização de Snaxels

Na seção 3.3 foi explicada a necessidade de manter o deslocamento dos snaxels dentro de limites estabelecidos afim de não prejudicar o método de controle topológico. A partir dessa necessidade, advém outra que é garantir, após o deslocamento dito "energético", o posicionamento dos snaxels sobre as arestas da malha. O modo como é garantido este processo, chamado **regularização**, será descrito detalhadamento nos próximos parágrafos.

O procedimento de **regularização** é aplicado na curva original $S_k$ após o cálculo das novas posições dos snaxels a partir da energia, gerando a curva transformada $TC_k$. Os snaxels de $TC_k$ são então deslocados às suas posições finais - obviamente considerando-se uma iteração da evolução da curva - formando a curva projetada $PC_k$

Existem diversos cálculos utilizados comumente nos processos de regularização e deslocamento a partir da energia. Assim, por motivo de economia de processamento, boa parte dos cálculos computados são armazenados, no contexto da uma iteração, de modo a servirem ao procedimento seguinte. Naturalmente, a economia em processamento exige utilização de memória para armazenagem dos valores. Entretanto, isso é justificado pela vantagem em se

computar uma única vez, e utilizar o poder de processamento dedicado o mais possível a processar todos os snaxels o mais rapidamente possível.

Com isso, o procedimento de **regularização** aproveita-se amplamente de valores que "aparecem" disponíveis em variáveis de memória, simplificando seu funcionamento.

## A.1 Detalhamento do algoritmo

A explicação do algoritmo será dada passo-a-passo sendo baseada no pseudo-código, havendo referência à origem das variáveis e estruturas de dados toda vez que se julgar necessário para melhorar a compreensão.

Inicia-se o procedimento chamado *Regularizing_TC* com as devidas inicializações de variáveis a serem utilizadas.

```
1. edge = {0, 1, 2, 3};
2. pixel = {0..7} ou {0..15};
```

A variável *edge* é um conjunto das possíveis arestas que podem ser utilizadas numa célula. A função da variável *pixel* é semelhante, indicando os possíveis pontos que podem ser ocupados sobre a aresta. Caso a célula tenha tamanho 8, os valores permitidas são de 0 a 7. Célula de tamanho 16, valores de 0 a 15. Ressaltando que estas são variáveis necessárias para utilização das coordenadas *CEP*.

```
3. forbidden_displacement[4] = {Number_Cell_Lines, 1,
                                -Number_Cell_Liner, -1};
```

Para controlar e verificar posições inválidas dos snaxels, os prováveis sentidos de movimentção são armazenados em forma de vetor, que será utilizado posteriormente no cálculo da posição de um snaxel na curva transformada - $TC_k$.

```
4. if ( cell == next_cell_TC)
5.     return;
6. else ...
```

O algoritmo se inicia efetivamente neste ponto, verificando se a célula atual é igual à célula do vértice final do segmento da curva transformada $TC_k$ que está sendo tratado. Esta é a condição que controla o fim do procedimento de regularização. Isso é possível pois todos os snaxels possuem a informação de snaxels vizinhos como numa lista encadeada.

```
7. if(orthogonal_to_TC_segment_X < 0)
8.     if(orthogonal_to_TC_segment_Y < 0)
```

As duas primeiras comparações (de um conjunto de outras) verificam os sinais dos vetores ortogonais aos vetores direção - separados nas suas componentes horizontal $x$ e vertical $y$ - formados pelo snaxel atual e seu próximo - $S_x$ e $S_y$. Nas figuras A.1 e A.2 estão convencionadas pelas cores, as direções dos vetores e seus vetores ortogonais correspondentes.

Neste caso, especificamente, a ortogonal está no segundo quadrante, indicando que a direção do segmento sendo regularizado está no primeiro quadrante. Com isso, definimos o *vértice de interesse*, que é o vértice no canto superior esquerdo, denominado vértice nordeste - $vNE$.

Na figura A.3 temos um esquema dos vértices válidos e as coordenadas *CEP* das arestas da célula.

Estabelecido o quadrante, é calculado o **produto interno**, cujo cálculo é diferente de acordo com o quadrante:

```
9. inner_product = (edge_length - TC_x) *
                    orthogonal_to_TC_segment.x -
                   (TC_y * orthogonal_to_TC_segment.y);
```

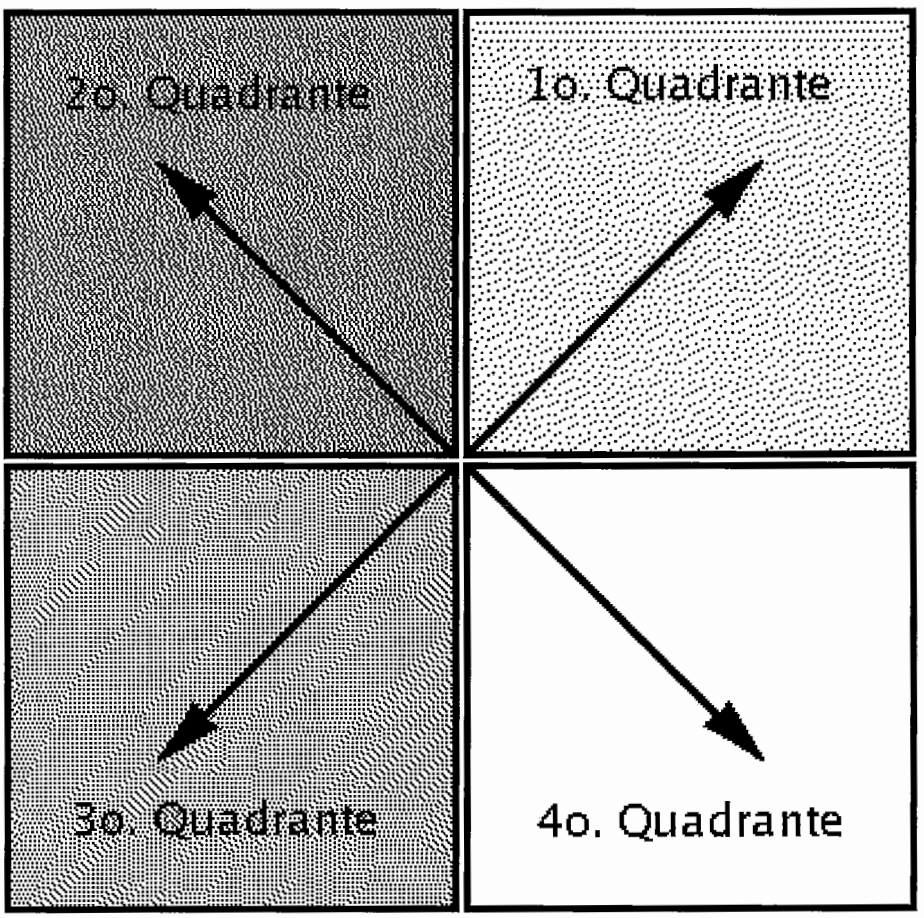


Figura A.1: Quadrantes dos vetores formados por dois snaxels consecutivos.

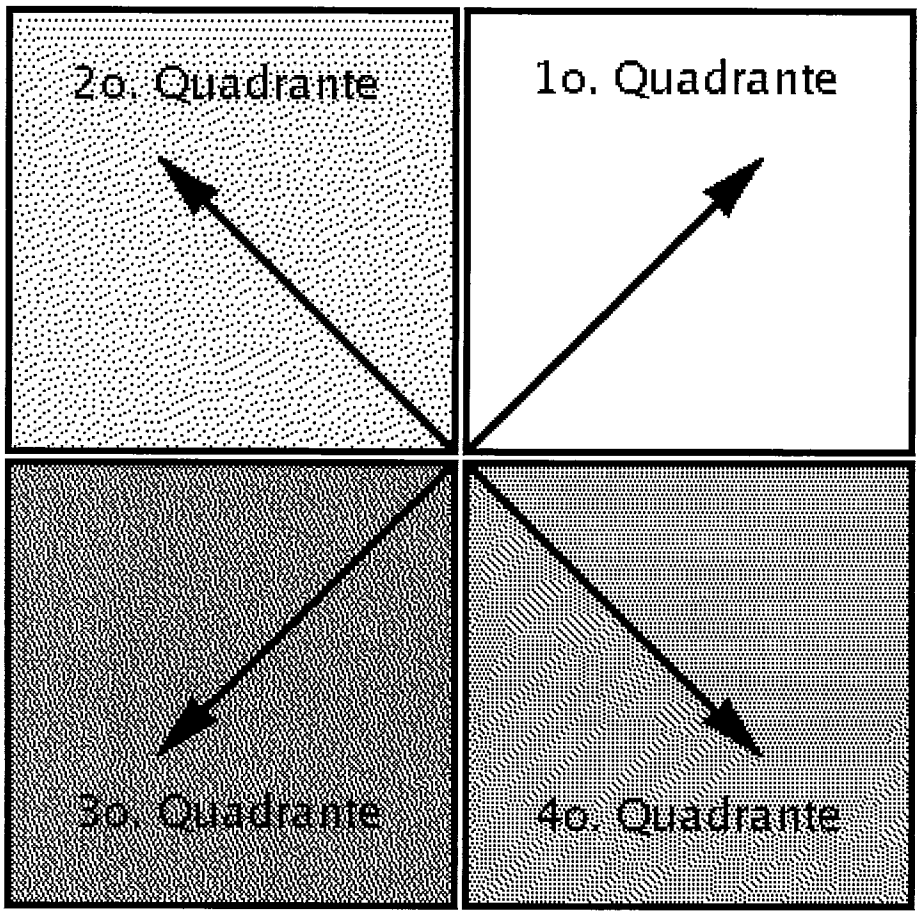


Figura A.2: Quadrantes correspondentes as vetores ortogonais aos vetores direção.

De posse do valor do produto interno, é iniciado um loop que, inicialmente incrementa o índice do snaxel sendo calculado. Caso o produto interno seja menor ou igual a zero, temos que o segmento a ser regularizado corta a aresta $e_1$ e, obrigatoriamente, a próxima célula a ser encontrada será a que está acima.

```
10. do {
11.     index ++ ;
12.     if(inner_product  <= 0) {
13.         cell = cell - num_cells_line;
```

Na linha 13 temos a atualização da célula atual para a próxima célula em que a curva está entrando.

```
14. if (cell != (snaxel_cell + forbidden_displacement[1])) {
15.      pixel[index] = edge_length -
              (int)(inner_product/orthogonal_to_TC_segment_x);
16.      inner_product = inner_product -
              (edge_length*orthogonal_to_TC_segment_y);
    }
```

Neste ponto (linha 14), deve-se verificar se uma célula proibida está sendo invadida. Caso não seja invadida, prossegue-se com o cálculo da posição do pixel na aresta, dado pelo comprimento da aresta menos o produto interno dividido pela ortogonal ao segmento regularizado na direção $x$ (linha 15). Na linha 16 procedemos com a atualização do produto interno. Essa atualização é devida ao vértice de interesse ser o vértice $vNE$. Como foi mudado o vértice, o produto interno é atualizado acrescentando-se a ele o produto escalar do vetor ortogonal ao segmento em regularização pela diferença entre o vértice de interesse atual e o anterior. Essa diferença é dada na direção $y$ pois os dois vértices são adjacentes a uma aresta vertical.

```
17. else {
18.     edge[index] = 0;
19.     pixel[index] = epsilon;
20.     index ++;
21.     inner_product = inner_product + edge_length*
                    (orthogonal_to_TC_segment_x
                     orthogonal_to_TC_segment_y);
22.     cell = snaxel_cell;
23.     pixel[index] = epsilon;
}
```

Caso seja detectado na comparação da linha 14 que uma célula proibida é invadida, devem ser acresentados dois snaxels da curva projetada $PC_k$. Ambos são acrescentados numa vizinhança do vértice de interesse. Procedendo assim, marca-se a aresta na linha 18 e posteriormente é acrescentado o primeiro snaxel (linha 19) numa distância $\epsilon$ (epsilon), que é uma constante, pequena em relação ao comprimento da aresta, definida anteriormente. O índice é incrementado (linha 20) para a inserção do próximo snaxel.

Antes da inserção, entretanto, é uma vez mais atualizado o produto interno, pois o novo snaxel a ser inserido está transladado para a direita em relação ao snaxel anterior (linha 21). A célula é corrigida, passando a ser a célula abaixo da célula do snaxel. Na linha 23 é acrescentado o novo vértice.

```
        }
24.     edge[index] = 3;
    }
```

Antes de finalizar a condição que verifica o valor do produto interno (linha 12), devemos marcar a aresta relativa ao snaxel inserido anteriormente. Dada o vértice

de interesse e a aresta do vértice anterior, estabelecemos qual a aresta relativa ao vértice atual.

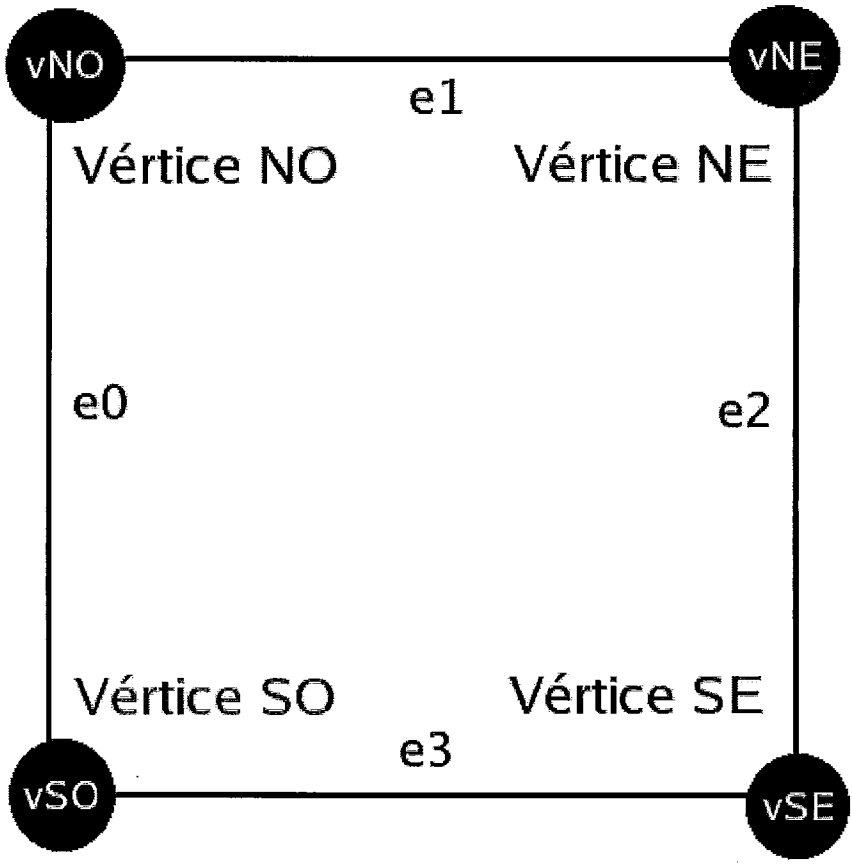


Figura A.3: Uma célula exemplificando os códigos *CEP* e os vértices de interesse.

Este procedimento é praticamente o mesmo para os quatro quadrantes. Naturalmente, há mudanças óbvias na forma como são calculados as arestas e *pixels*. Entretanto, fundamentalmente, o funcionamento segue os mesmos padrões no cálculo do produto interno e demais variáveis, não sendo necessário repetir sua explicação.

# Referências Bibliográficas


[1] S. Bischoff, T. Weyand, and L. Kobbelt. Snakes on triangle meshes. *In Bildverarbeitung fur die Medizin*, pages 208–212, 2005. http://www-i8.informatik.rwth-aachen.de/publications/downloads/sotm.pdf.

[2] S. S. Bischoff and L. Kobbeit. Snakes with topology control. *The Visual Computer*, 20:217–228, 2004.

[3] A. Black and A. (editors) Yuille. *Active Vision.* MIT Press, 1993.

[4] L. D. Cohen. On active contour models and balloons. *Computer Vision, Graphics, and Image Processing. Image Understanding*, 53(2):211–218, March 1991.

[5] L.D. Cohen and I. Cohen. Finite-element methods for active contour models and balloons for 2-d and 3-d images. *Pattern Analisys and Machine Intelligence*, 15(11):1131–1147, November 1993.

[6] Matthias Kalle Dalheimer. *Programming with Qt*, chapter 1, 2, 6, 7, 11, 17, 24. O'Reilly, second edition, January 2002.

[7] H. Delingette and J. Montagnat. Shape and topology constraints on parametric active contours. *Computer Vision and Image Understanding: CVIU*, 83(2):140–171, 2001.

[8] R. Durikovic, K. Kaneda, and H. Yamashita. Dynamic contour: a texture approach and contour operations. *The Visual Computer*, 11:277–289, 1995.

[9] Leymarie F. and M. D. Levine. Tracking deformable objects in the plane using an active contour model. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(6), June 1993.

[10] G. Giraldi, E. Strauss, and A. Oliveira. Dual-t-snakes model for medical image segmentation. *Pattern Recognition Letters*, 24(7):993–1003, April 2003.

[11] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active contour models. *Procedings of First International Conference in Computer Vision*, pages 259–268, 1987. London.

[12] J-O. Lachaud and A. Montanvert. Deformable meshes with automatic topology changes for carse-to-fine threedimensional surface extraction. *Computer Vision Image Understanding*, 3(2):187–207, 1999.

[13] Jérôme Landré. Programming with intel ipp and intel opencv under gnu linux. Technical report, Universtité de Bourgogne, Laboratoire Életronique, Informatique et Images, Le Creusot, France, July 2003.

[14] R. Malladi, J. A. Sethian, and B. C. Vemuri. Shape modeling with front propagation: a level set approach. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(2):158–175, 1995.

[15] T. McInerney. *Topologically Adaptable Deformable Models for Medical Image Analysis*. PhD thesis, Department of Computer Science, University of Toronto, Toronto, ON, January 1997.

[16] T. McInerney, M. Sharif, and N. Pashotanizadeh. Jess: Java extensible snakes system. In J. Fitzpatrick and J. Reinhardt, editors, *Medical Imaging 2005: Image Processing*, volume 5747, pages 1985–1992, April 2005.

[17] T. McInerney and D. Terzopoulos. Topologically adaptable snakes. In *International Conference on Computer Vision*, pages 840–845, Cambridge, MA, USA, June 1995.

[18] T. McInerney and D. Terzopoulos. Topology adaptive deformable sufaces for medical image volume segmentation. *IEEE Transactions on Medical Imaging*, 18(10):840–50, 1999.

[19] T. McInerney and D. Terzopoulos. T-snakes: topology adaptative snakes. *Medical Image Analysis*, 4(2):73–91, 2000.

[20] A. Oliveira, S. Ribeiro, C. Esperança, and G. Giraldi. Loop snakes: The generalized model. In *Internation Symposium on Graphical Models and Imaging - GMAI*, 2005.

[21] A. Oliveira, S. Ribeiro, G. Giraldi, R. Farias, and C. Esperança. Loop snakes: Snakes with enhanced topology control. In *XVII SIBGRAPI*, 2004.

[22] S. Osher and J. A. Sethian. Fonts propagating with curvature-dependent speed: algorithims based on hamilton-jacobi formulations. *Journal of Computational Physics*, 79(1):12–49, 1988.

[23] S. J. Osher and R. P. Fedkiw. *Level set methods and dynamic implicit surfaces*. Springer-Verlag, New York, 2002.

[24] Saulo Pereira Ribeiro. Loop-snakes - snakes com controle topológico otimizado. Master's thesis, Programa de Engenharia de Sistemas e Computação,

COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, February 2005.

[25] J. A. Sethian. *Level Set Methods: Evolving Interfaces in Geometry, Fluid Mechanics, Computer Vision and Materials Sciences.* Cambridge University Press, 1996.

[26] Dave Shreiner, Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*, chapter 1. Addison Wesley, fifth edition, July 2005.

[27] A. Singh, D. Terzopoulos, and D. B. Goldgof. *Deformable Models in Medical Image Analysis.* IEEE Computer Society Press, 1998.

[28] E. Strauss, W. Jimenez, G. Giraldi, R. Silva, and A. Oliveira. A semi-automatic surface reconstruction framework based on t-surfaces and isosurface extraction methods. In *International Symposium on computer Graphics, Image Processing and Vision*, 2002.

[29] R. Szeliski, D. Tonnesen, and D. Terzopoulos. Modeling surfaces of arbitrary topology with dynamic particles. In *Computer Vision and Pattern Recognition Conference*, pages 82–87, New York, 1993.

[30] S. C. Zhu and A. Yuille. Region competition: Unifying snakes, region growing, and bayes/mdl for multiband image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(9):884–900, September 1996.