5
Descompressão de Malhas Irregulares

No Capítulo 4 apresentou-se um algoritmo para compressão de malhas irregulares com suporte a alças.

Neste Capítulo apresenta-se a decomposição *Handlebody* para descompressão de malhas irregulares. Logo a seguir apresenta-se o algoritmo para reconstrução de malhas proposto no Capítulo 4, é necessário para isto os arquivos (*Fclers*, *Fhandles*,*Fgeometry*) gerados pelo método de compressão.

5.1
Descompressão Edgebreaker

A metodologia Wrap&Zip para a descompressão de malhas apresentado em [14] assim como a estratégia de descompressão pelo algoritmo *Spirale Reversi* apresentado em [6] serão utilizados neste trabalho para realizarmos a descompressão de malhas irregulares.

Para reconstruir uma malha codificada pelo algoritmo de compressão apresentado no capítulo 4 são necessários dois passos.

O primeiro passo chamado de Wrap decodifica e utiliza os *labels* armazenados no arquivo *Fclers* para decidir aonde será adicionado um novo triângulo ou quadrângulo à malha que será reconstruída. O resultado é um malha topológica conectado que corresponde a árvore TST(*Triangle Spanning Tree*) da malha original.

O segundo passo da descompressão é o processo de Zip, o qual fecha as curvas de borda que não estão ligadas para finalizar a reconstrução da conectividade e geometria da superfície. Para ligar corretamente as arestas de bordo da malha, o processo de Zip utiliza os *labels* para orientar as arestas

livres no sentido anti-horário para os labels L, R e E. Sentido horário para os *labels* do tipo C (Figura 5.1).

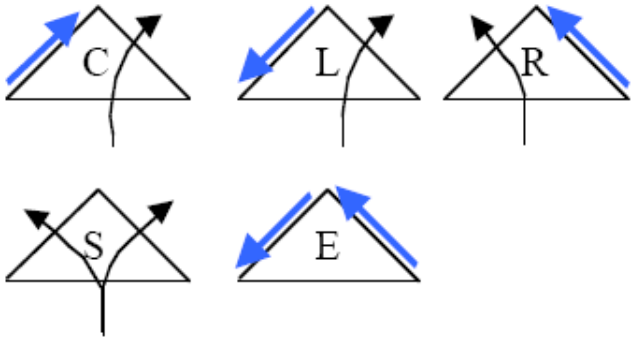


Figura 5.1: Orientação das arestas no processo Wrap&Zip.

O procedimento de descompressão também faz uso de pilhas para poder realizar uma busca em profundidade e poder reconstruir a malha corretamente sem alterar a sua topologia. São necessários duas pilhas para armazenar os *labels* do tipo S assim como também os *labels* do tipo C.

A pilha que guarda os *labels* do tipo S indica o próximo ramo a esquerda da árvore TST a ser descomprimida, caso o ramo a esquerda de S já foi visitado pega-se o proximo *label* da pilha e assim sucessivamente.

O processo de Zip é realizado utilizando a pilha que guarda os labels do tipo C, estes labels são os que tem as arestas livres iguais a -1 e o seu oposto simétrico igual a -2. O processo de Zip é realizado após ter-se feito o processo de Wrap, logo o primeiro elemento da pilha que guarda o label do tipo C será o ultimo label a ser desempilhado, isto é o processo de Zip será realizado então detrás para frente.

O processo Wrap&Zip decodifica e reconstrói a malha na mesma ordem em que os triângulos foram visitados e codificados pelo processo de compressão. Este tipo de decodificação tem o tempo de complexidade no pior caso de $O(n^2)$, pois este processo requer procedimentos para procurar adiante as subseqüências encapsuladas pelo label do tipo S e E para posteriormente decodifica-las corretamente. O procedimento *Spirale Reversi*decodifica a malha lendo o *string* de CLERS de trás para frente, desta forma evita-se procedimentos para procurar subseqüências encapsuladas pelos labels S e E. Este tipo de decodificação é feita em tempo linear.

O nosso processo de decodificação é parecido ao processo de decodificação apresentado em [6], porém a leitura dos *CLERS* é feita na ordem em que estas foram geradas. Na metodologia apresentada por Isenburg a leitura

dos CLERs é feita na ordem inversa em que estes foram gerados. A cada leitura de um triângulo ou quadrângulo do tipo C ou c empilha-se estes *labels*. Desta forma o primeiro triângulo ou quadrângulo a ser reconstruído após o processo de Wrap será o ultimo triângulo ou quadrângulo que entrou na pilha e este por sua vez tem um label do tipo c ou C. Desta forma temos um procedimento do tipo pseudo *Spirale Reversi* para o processo de Zip.

O procedimento de decodificação não é recursivo pois depende dos dados armazenados na pilha de labels S e C, desta forma não precisamos criar procedimentos para procurar subseqüências geradas à frente pelos labels S e E.

Na próxima seção mostra-se a Decomposição de *HandleBody* presente no processo de descompressão e decodificação de malhas irregulares, depende do caso utiliza-se um dos operadores de alças definidos no Capítulo 2.

5.2
Decomposição de HandleBody do EdgeBreaker

A descompressão do método *EdgeBreaker* consta de dois processos Wrap&Zip definido em [14], porém o processo de busca em profundidade não é recursivo devido ao uso de estrutura de dados de pilhas para os labels do tipo S, s, c e C.

O processo de Wrap decodifica a superfície comprimida adicionando um novo triângulo de cada vez ao triângulo adjacente, de tal forma que serão lidos todos os *labels* armazenados no arquivo de CLERS . Além disso a rotina *ReadEB* atualiza a tabela M salvando as semi-arestas opostas das $2g$ arestas armazenadas no arquivo de *Fhandles*. A superfície gerada pela chamada da rotina *Decompress* é homeomorfa a superfície original M.

Estes dois passos da descompressão, de fato definem uma decomposição de *Handlebody* para superfície M. A seqüência finita de superficies combinatórias $M_i, i = 0, .., n$ tais que $M_0 = \emptyset$ e $M_n = M$, os quais foram geradas pelo algoritmo de descompressão *EdgeBreaker* pode ser analisado por meio do uso dos operadores de handle definidos no Capitulo 2, onde n corresponde ao número de operadores de alça que usaremos para construir a superfície.

Em esta seção mostraremos que este número é $2|T(M)| + |V(M)| +$

$2g - 2$. Note que a identificação das duas arestas de bordo é executada na estrutura de dados *CHalfEdge*, isto é armazenando na tabela M os mates opostos correspondentes.

O algoritmo de descompressão *EdgeBreaker* inicializa com um triângulo, neste caso, um operador de alças de índice 0 é aplicado para criar o triângulo que corresponde a M_1. No caso de iniciar a descompressão com um quadrângulo são aplicados dois operadores de alças de índice 0 pois cada quadrângulo e sub-dividido em dois triângulos adjacentes.

No passo seguinte, O *Edgebreaker* começa a decodificar a string de CLERS, note que a seqüência de CLERS tem $|T(M)| - 1$ símbolos.

A leitura de um símbolo $CLERS_i$, i em $[1...(|T(M)| - 1)]$, cria um novo triângulo (operador de alça de tipo 0) e anexa uma das arestas deste novo triângulo a aresta porta da superfície anterior. Veja a Figura 2.9(a).

A superfície M_{2i+1} é definida para ser a superfície resultante da aplicação destes dois operadores de alças. A característica de Euler da superfície M_{2i+1} é a mesma de M_{2i-1}, desde que M_{2i+1} tenha mais um triângulo, mais duas arestas e mais um vértice que M_{2i-1}.

Como conclusão, a superfície $M_{2|T(M)|-1}$ é uma superfície homeomorfa conectada a um disco (com uma única componente de bordo) sem nenhum vértice interior. Isto é, $\chi(M_{2|T(M)|-1}) = 1$.

Se o arquivo de *Fhandle* está vazio, isto é $(g = 0)$, podemos partir diretamente ao processo de Zip da decomposição de *Handlebody*. Caso contrario, tem-se que identificar os pares $2g$ de arestas de bordo representados pelos semi-arestas armazenadas no arquivo de *Fhandle*.

Para cada gênus, identificamos o primeiro par de arestas de bordo para gerar a superfície $M_{2|T(M)|}$ pelo uso dos operadores de alças de índice 1 mostrados na Figura 2.9(a). Veja que estas arestas não tem vértices em comum e pertencem a mesma componente de bordo. Além disso, este operador divide a curva de bordo em dois componentes. Após isto, identificamos o segundo par de arestas de bordo os quais pertencem a diferentes curvas de bordo, pelo uso do operador de alças de índice 1 da 2.9(c) para gerar a superfície $M_{2|T(M)+1|}$.

Este operador adiciona um genus a superfície e concatena duas curvas de bordo. A característica de Euler de $M_{2|T(M)|+1}$ é calculada de acordo a seguinte expressão:

$$\chi(M_{2|T(M)|+1}) = \chi(M_{2|T(M)|}) - 1 = \chi(M_{2|T(M)|-1}) - 1 = 1 - 2 = -1.$$

Este processo é repetido ate que todos os pares arestas de bordo no arquivo *Fhandle* seja identificado. Como conseqüência, a superfície $M_{2|T(M)|+2g-1}$ é uma superfície conectada com gênus g o qual tem uma única componente de bordo e nenhum vértice interior. A sua característica de Euler é:

$$\chi(M_{2|T(M)|+2g-1}) = \chi(M_{2|T(M)|-1}) - 2g = 1 - 2g$$

Logo, a processo de Zip é chamado para identificar os $|V(M)|-1$ pares de arestas adjacentes de bordo com indices -1 e -2. Estas arestas, depois da identificação, corresponderão a arvore T_{VE}.

Quando o primeiro par é encontrado, uma operação de alça de índice 2 é aplicado (Figura 2.10(b)). Este operador identifica as duas arestas de bordo que tem somente um vértice em comum para criar um vértice interior na superfície $M_{2|T(M)|+2g-1}$. Isto não muda a característica de Euler da superfície, por tanto a superfície resultante é homeomorfa a $M_{2|T(M)+2g-1}$

A processo de Zip continua a buscar iterativamente e a construir uma nova superfície para cada label retirado da pilha de labels do tipo C. Este terminará quando o ultimo par de arestas de bordo com indices -1 e -2 seja encontrado ou a pilha esteja vazia.

Veja que todos os $|V(M)| - 2$ pares de arestas de bordo inicialmente encontrado pelo processo de Zip tem somente um vértice em comum. Por tanto, a superfície $M_{(2|T(M)|+2g-1)+|V(M)|-2}$ ainda é homeomorfa a $M_{2|T(M)|+2g-3}$, e:

$$\chi(M_{2|T(M)|+2g+|V(M)|-3}) = 1 - 2g$$

Finalmente, o ultimo par de arestas de bordo existente na superfície é identificada pelo operador de alça de tipo 2 mostrado na Figura 2.10(a), o qual remove a componente de bordo. Podemos concluir que $M_{2|T(M)|+2g+|V(M)|-2}$ é a superfície original sem borda M para o qual a característica de Euler é $\chi(M) = 2 - 2g$.

Na próxima seção mostra-se em detalhes a implementação da extensão do *Edgebreaker* para a descompressão de malhas irregulares;

5.3
Descompressão de superfícies irregulares com gênus

Nesta seção será descrita a implementação do algoritmo de descompressão de uma superfície conexa orientável $\mathcal{S}$ sem bordo, composta por triângulos e/ou quadrângulos. Essa descrição será feita do topo para a base. Ele implementa a nova proposta apresentada na seção anterior utilizando a classe *CHalfEdge*. Os algoritmos, funções e procedimentos mostrados nesta seção, são métodos que pertencem a essa classe.

Assume-se, a partir de agora, que a superfície $\mathcal{S}$ será instanciada como um objeto da classe *CHalfEdge* a partir dos arquivos gerados pela compressão(*Fclers*, *Fhandles* e *Fgeometry*).

5.3.1
Inicialização da Descompressão

A seguir descreve-se o procedimento que inicia o processo da descompressão, nomeado de *ReadEB*. Tem-se como parâmetros de entrada os arquivos *Fclers*, *Fhandles* e *Fgeometry* a partir dos quais serão lidos o número de vértices, triângulos e quadrângulos. A partir destes dados calcula-se a quantidade de semi-arestas que a malha possui.

Inicialmente são definidas e iniciadas algumas variáveis auxiliares que serão usadas globalmente nas rotinas do algoritmo de descompressão. Essas variáveis auxiliares são:

- int T; um número inteiro que indica o índice do próximo triângulo a ser visitado. O seu valor inicial é -1.
- int Q; um número inteiro que indica o índice do próximo quadrângulo a ser visitado. O seu valor inicial é -1.
- int N; um número inteiro que indica o índice do último vértice visitado. O seu valor inicial é 0.
- int Flag; um número inteiro que indica se a primeira face da superfície a ser reconstruída começa com um triângulo ou um quadrângulo. Caso este valor seja igual a 3 é um triângulo, caso contrario o seu valor é igual a 4 indicando que a superfície começa com um quadrângulo.

Após a leitura de todos os vértices do primeiro triângulo ou quadrângulo, armazena-se os seus valores nos indices correspondentes as tabelas V, M e G. Incrementa-se os valores de T ou Q dependendo do caso, assim como a valor da variável N. A leitura do primeiro quadrângulo é feita numa sub-rotina chamada *ReadFirsQuad* 32.

Finalmente é chamado o módulo Decompress. Este módulo faz a descompressão propriamente dita e executará os sub-módulos necessários ao algoritmo de descompressão.

```
algoritmo ReadEB(char *name)
 1 char clersname[50]; char geoname[50]; char topname[50];
 2 //Gera os arquivos para leitura de dados
 3 sprint(clersname,"%s.eb",name);
 4 sprint(geoname,"%s.geo",name);
 5 sprint(topname,"%s.top",name);
 6 fstream Fclers(clersname,ios::in);
 7 fstream Fgeometry(geoname,ios::in);
 8 fstream Fhandles(topname,ios::in);
 9 //Lê do arquivo número de vértices
10 Fgeometry >> NV;
11 //Lê do arquivo número de triângulos
12 Fgeometry >> NT;
13 //Lê do arquivo número de quadrângulos
14 Fgeometry >> NQ;
15 for (h=0; h<3*NT+4*NQ;h++)
16 {
17    V[h] = -1; M[h] = -3;
18 }
19 //Inicializa variáveis de controle
20 T = -1; Q = -1; N = 0;
21 //Variavel q indica se compressão inicia com triângulo
22 //ou com um quadrangulo
23 Fgeometry >> Flag;
24 if (Flag == 3)
25 {
26 //Lê dados do triângulo inicial
27    Fgeometry >> G[0][0] >> G[0][1] >> G[0][2];
28    Fgeometry >> G[1][0] >> G[1][1] >> G[1][2];
29    Fgeometry >> G[2][0] >> G[2][1] >> G[2][2];
30    V[0] = 2; V[1] = 1; V[2] = 0;
31    M[0] = -1; M[1] = -1;
32    N = 2; quad T++;
33 }
34 then
35 {
36 //Lê dados do quadrângulo inicial
37    ReadFirstQuad();
38 }
39 Decompress(HalfEdge);
40 //Fecha arquivos
41 Fclers.close(); Fgeometry.close(); Fhandles.close();
fim
```

Algoritmo 31: Inicialização do módulo de Descompressão ReadEB.

```
algoritmo ReadFirstQuad()
   1 //Le dados do primeiro quadrângulo
   2 Fgeometry >> G[0][0] >> G[0][1] >> G[0][2];
   3 Fgeometry >> G[1][0] >> G[1][1] >> G[1][2];
   4 Fgeometry >> G[2][0] >> G[2][1] >> G[2][2];
   5 Fgeometry >> G[3][0] >> G[3][1] >> G[3][2];
   6 //Atualiza valores nos vetores V e M
   7 V[0] = 3; V[1] = 2; V[2] = 1; V[3] = 0;
   8 M[0] = -1; M[1] = -1; M[2] = -1;
   9 //Incrementa número de vértices visitados
  10 N = 3;
  11 //Incrementa número de quads visitados
  12 Q++;
fim
```

Algoritmo 32: Sub-módulo da inicialização ReadEB.

5.3.2

A Descompressão

A rotina *Decompress* é a principal do algoritmo de descompressão. Para reconstruir a malha cria-se uma arvore geradora no grafo dual da superfície a partir dos arquivos *Fclers*, *Fhandles* e *Fgeometry*, os quais foram gerados pela rotina *Compress*.

Como foi explicado nas seções anteriores, o processo de descompressão de malhas decodifica e utiliza os *labels* CLERS para decidir aonde adicionar um novo triângulo ou quadrângulo na nova malha que esta sendo reconstruída cujo resultado será uma malha topológica conectada que corresponderá a árvore TST da malha original.

O processo de descompressão inicia lendo o arquivo de *Fhandles*. para armazenar as 2g semi-arestas que são do tipo alças que são necessárias para reconstruir a topologia da malha corretamente (neste caso diz-se que a malha possui g gênus, caso caso contrário g=0.) e termina lendo todos os *labels* pertencentes ao arquivo *Fclers*. Assim como no algoritmo de descompressão uma busca em profundidade é feita.

É muito comum que algoritmos para busca em profundidade sejam implementados recursivamente. Mas neste caso, ficou mais simples utilizar uma pilha, pois é sabido exatamente quais são os pontos possíveis de

bifurcações, que são: um triângulo do tipo S ou um dos triângulos do quadrângulo do tipo s ou S.

Essa pilha, na realidade, armazena o índice da *semi-aresta pivô* de cada face, que é definida como sendo a semi-aresta antecessora da semi-aresta que é a porta de entrada da face.

Toda vez que a face de uma semi-aresta pivô é decodificada como s ou S, então a semi-aresta pivô é inserida na pilha. Da mesma forma se esta aresta pivô e decodificada do tipo C ou c também é inserida numa pilha.

Quando for encontrado um triângulo do tipo E, ou quando a segunda face do quadrângulo é do tipo E, duas situações podem ocorrer. O algoritmo volta para a última face decodificada como s ou S, ou não há mais face a visitar. No primeiro caso, uma semi-aresta esquerda de uma face do tipo S ou s é desempilhada, e um teste é feito para saber se a face à esquerda adjacente já foi visitada ou não. Se ela ainda não foi visitada, inicia-se um novo ramo na busca em profundidade, e caso contrário a aresta corresponde a uma alça, neste caso a próxima semi-aresta do tipo s ou S é desempilhada e o algoritmo de decodificação continua normalmente. Finalmente, o caso em que não há mais faces a decodificar acontece quando não existir mais *labels* no arquivo *Fclers* ou ate que a pilha que armazena os *labels* do tipo s ou S esteja vazia.

Uma vez que a pilha que guarda os *labels* do tipo s ou S estiver vazia inicia-se o processo de Zip ligando as semi-arestas cujos valores são igual a -1 e a sua aresta simétrica é igual a -2. Este processo é feito utilizando-se a pilha que guarda os *labels* do tipo c ou C, as semi-arestas do primeiro triângulo ou quadrângulo que iniciaram o processo de descompressão e os quais são iguais a -1 também estão armazenados nesta pilha. Desta forma reconstrói-se a superfície S corretamente.

Durante o percorrimento de um ramo na busca em profundidade, é testado de qual é o tipo de cada face: triângulo ou quadrângulo. Uma vez classificada, é chamado a rotina *DecompressTriangle* ou *DecompressQuad*, respectivamente.

O algoritmo 33 mostra a implementação do esquema de descompressão.

```
algoritmo Decompress(int HalfEdge)
 1 int Handle1, Handle2, NH;
 2 //Le numero de alças do arquivo
 3 Fhandles >> NH;
 4 //Para cada alça armazena dados no vetor M
 5 for (i=0; i<NH; i++)
 6 {
 7   Fhandles >> handle1 >> handle2;
 8   M[handle1]=handle2; M[handle2]=handle1;
 9 }
10 //Empilha dados do primeiro triângulo ou quadrângulo
11 for (i = Flag-2 ; i >= 0 ; i–)
12 { stackC.Push(i);}
13 //Empilha dados na pilha cujo label é do tipo S
14 stackS.Push(HalfEdge);
15 //Inicia processo de Wrap
16 while(!(stackS.isEmpty()))
17 {
18   Halfedge = stackS.Pop();
19   if (M[prev(HalfEdge)] >= 0) continue;
20   //Decodifica superfície ate achar label do tipo E
21   do
22   {
23     Clers1 = fgetc(Fclers);
24     //Decodifica novo triângulo
25     switch(Clers1)
26     {
27       case 'C': case 'L': case 'E': case 'R': case 'S':
28         IsAnEnd = DecompressTriangle(HalfEdge);
29       break;
30     }
31     //Decodifica novo quadrângulo
32     switch (Clers2)
33     {
34       case 'c': case 'l': case 's':
35       Clers2 = fgetc(Fclers);
36         IsAnEnd = DecompressQuad(HalfEdge);
37       if (IsAnEnd) break;
38   }while(true);
39 }
40 //Inicia processo de Zip
41 while(!(stackC.isEmpty()))
42 { HalfEdge = stackC.Pop(); Zip(HalfEdge);}
fim
```

Algoritmo 33: Módulo Geral para Descompressão de Malhas Irregulares.

5.3.3
Descompressão de Triângulo

O algoritmo 34 decodifica cada triângulo da malha. Recebe como parâmetro a semi-aresta pivô do triângulo atual, nomeada $HalfEdge$. Corresponde à semi-aresta antecessora a semi-aresta que foi escolhida como porta de entrada da face. A rotina retorna o inteiro 1 caso o triângulo tenha sido decodificado como E e 0 caso contrário.

A seguir, verifica se o vértice oposto à porta, que corresponde ao vértice da semi-aresta pivô, ainda não foi visitado. No caso afirmativo, o triângulo é decodificado como C e o seguinte passo é feito, lê-se do arquivo *Fgeometry* as coordenadas do vértice oposto; marca-se em seguida o vértice como visitado; atribui-se o novo valor da semi-aresta pivô à variável $HalfEdge$, para indicar à face a direita como sendo a próxima a ser visitada e finalmente empilha-se esta semi-aresta no pilha de *labels* do tipo C.

No caso em que o vértice oposto à porta já tenha sido visitado, verifica-se se a face à direita e a face à esquerda também já foram. E com isso o triângulo é decodificado como do tipo L, E, R, ou S, de acordo com a proposta original do *Edgebreaker*. A rotina *DecompressTriangle* faz essa decodificação (algoritmo 34).

Como mencionado anteriormente, se o triângulo for decodificado como sendo do tipo S, a sua semi-aresta pivô será armazenada numa pilha que guarda especificamente estes dados.

```
algoritmo DecompressTriangle(int &HalfEdge)
 1 //Incrementa índice do triângulo visitado
 2 T++;
 3 he = 3*T;
 4 //Constrói adjacências do triângulo na tabela M
 5 M[prev(HalfEdge)] = next(he);
 6 M[next(he)] = prev(HalfEdge);
 7 //Constroi ids para os vértices do triângulo
 8 V[next(he)] = V[HalfEdge];
 9 V[prev(he)] = V[prev(HalfEdge)];
10 //move halfedge para o próximo triângulo
11 HalfEdge = he;
12 //Decodifica e empilha labels segundo cada caso especificado
13 //e retorna um inteiro para cada caso
14 Switch (Clers)
15 {
16   case 'C':
17     N++;
18     Armazena dados da coordenada do vértice no vetor G
19     Fgeometry >> G[N][0] >> G[N][1] >> G[N][2];
20     M[HalfEdge] = -1;
21     V[HalfEdge] = N;
22     stackC.Push(HalfEdge);
23     return 0;
24   case 'L':
25     if (M[HalfEdge] < 0)
26       M[HalfEdge] = -2;
27     return0;
28   case 'R':
29     if (M[prev(HalfEdge)] < 0)
30       M[prev(HalfEdge)] = -2;
31     HalfEdge = next(HalfEdge);
32     return 0;
33   case 'S':
34     stackS.Push(next(HalfEdge));
35     return 0;
36   case 'E':
37     if (M[HalfEdge] < 0)
38       M[HalfEdge]=-2;
39     Se (M[prev(HalfEdge)] < 0)
40       M[prev(HalfEdge)]=-2;
41     return 1;
42 }
fim
```

Algoritmo 34: Módulo de Descompressão para triângulos.

5.3.4
Descompressão de Quadrângulos

A compressão de cada quadrângulo é obtida pelo algoritmo 35. Para decodificar um quadrângulo, deve-se ler dois *labels* do arquivo *Fclers*. o primeiro *label* é do tipo c, l, ou s e o segundo *label* é do tipo C, L, E, R ou S.

O módulo para compressão de quadrângulos foi naturalmente dividido em duas etapas. Uma para identificar o primeiro triângulo do quadrângulo (algoritmo 35), e outra para decodificar o segundo triângulo (algoritmo 36).

A decodificação de quadrângulos, é similar a decodificação de quadrângulos porém deve-se ter muito cuidado para reconstruir cada quadrângulo, pois estes foram divididos em dois triângulos adjacentes.

Segue na seguinte folha o algoritmo 35.

```
algoritmo DecompressQuad(int &HalfEdge)
 1 //Incrementa numero de quadrângulos visitados
 2 Q++;
 3 //Calcula halfedge corrente
 4 he = 3*ntriangles + 4*Q;
 5 Atualiza vetor M
 6 M[prev(HalfEdge)] = next(he);
 7 M[next(he)] = prev(HalfEdge);
 8 Atualiza vetor V
 9 V[next(he)] = V[HalfEdge];
10 V[next(next(he))] = V[prev(HalfEdge)];
11 HalfEdge = he;
12 //Codifica halfedge corrente segundo o seu label respectivo
13 Switch (Clers1)
14 {
15   case 'c':
16     //Incrementa numero de vértices visitados
17     N++;
18     //Armazena dados da coordenada do vértice no vetor G
19     Fgeometry >> G[N][0] >> G[N][1] >> G[N][2];
20     M[HalfEdge] = -1;
21     V[HalfEdge] = N;
22     stackC.Push(HalfEdge);
23     break;
24   case 'l':
25     if (M[HalfEdge] < 0)
26       M[HalfEdge] = -2;
27     break;
28   case 's':
29     stackS.Push(next(HalfEdge));
30     break;
31 }
32 //Decodifica segundo triângulo do quadrangulo
33 IsAnEnd = DecompressSecondTriangle(prev(HalfEdge));
34 retorna IsAnEnd;
fim
```

Algoritmo 35: Módulo de Descompressão para quadrângulos.

O algoritmo mostrado na pagina anterior é a primeira parte da decodificação de quadrângulos, a continuação mostra-se a segunda parte que é relativamente o mesmo modulo da descompressão de triângulos apresentado nas seções anteriores.

```
algoritmo DecompressSecondTriangle(int &HalfEdge)
 1 //Decodifica segundo triângulo do quadrângulo
 2 //segundo os labels para cada caso e retorna um inteiro
 3 Switch (Clers2)
 4 {
 5   case 'C':
 6     N++;
 7     //Armazena coordenadas do vértice no vetor G
 8     Fgeometry >> G[N][0] >> G[N][1] >> G[N][2];
 9     M[HalfEdge] = -1;
10     V[HalfEdge] = N;
11     stackC.Push(HalfEdge);
12     return 0;
13   case 'L':
14     //Atualiza vetor M
15     if (M[HalfEdge] < 0)
16       M[HalfEdge] = -2;
17     return 0;
18   case 'R':
19     //Atualiza vetor M
20     if (M[prev(HalfEdge)] < 0)
21       M[prev(HalfEdge)] = -2;
22     //Move-se para a próxima halfedge
23     HalfEdge = next(HalfEdge);
24     return 0;
25   case 'S':
26     //Empilha halfedge na pilha
27     stackS.Push(next(HalfEdge));
28     return 0;
29   case 'E':
30     if (M[HalfEdge] < 0)
31       //Atualiza vetor M
32       M[HalfEdge]=-2;
33     if (M[prev(HalfEdge)] < 0)
34       //Atualiza vetor M
35       M[prev(HalfEdge)]=-2;
36     return 1;
37 }
fim
```

Algoritmo 36: Módulo de compressão para quadrângulos para o segundo triângulo do quadrângulo.

A primeira parte do processo de descompressão (processo de Wrap) foi apresentado nas seções anteriores a continuação apresenta-se o processo de Zip, o qual irá juntar as semi-arestas que ainda estão livres com valor igual a -1.

O seguinte algoritmo implementa o processo de Zip, para cada *Halfedge* igual a -1, procura-se pelo sua semi-aresta oposta que deve ser igual a -2. Logo após deve-se fazer o percurso da estrela à semi-aresta corrente para atualizar os seus vértices. Este processo é realizado enquanto houver *labels* do tipo C ou c na pilha(algoritmo 37).

```
algoritmo Zip(HalfEdge)
 1 //Pega halfedge corrente
 2 he1 = HalfEdge;
 3 //Pega halfedge anterior a halfedge corrente
 4 he2 = prev(HalfEdge);
 5 //Percorre halfedges ate achar um com valor negativo
 6 while (M[he2] >= 0)
 7 {
 8    he2 = prev(he2);
 9 }
10 //Achou halfedge com valor negativo
11 //atualiza os seus opostos respectivos a uma halfedge
12 if (M[he2] == -2)
13 {
14    M[he1] = he2;
15    M[he2] = he1;
16    he = M[prev(he1)];
17 //Faz o percurso da estrela ligando os vertices correspondentes
18    while (he != he1)
19    {
20       V[he] = V[he1];
21       he = M[prev(he)];
22    }
23 }
fim
```

Algoritmo 37: Módulo para zipar arestas livres de uma *HalfEdge*.

5.4
Ilustração do algoritmo

5.4.1
Exemplo 1: Uma pirâmide com base quadrangular

Para ilustrar o algoritmo, considere a superfície que é uma pirâmide de base quadrangular como mostrado na Figura 5.2(b). A Figura 5.2(a) mostra o modelo planar que será usado para explicar o algoritmo.

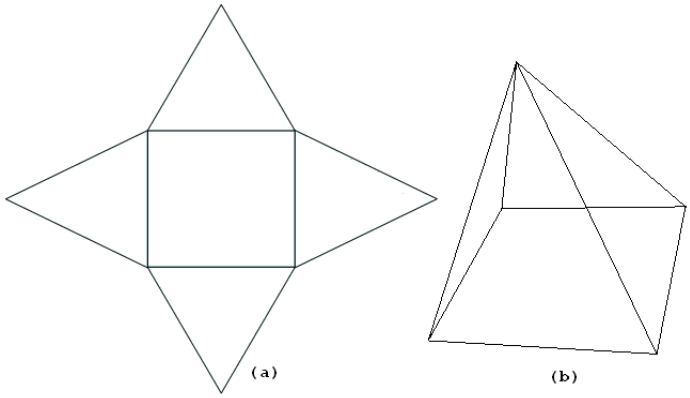
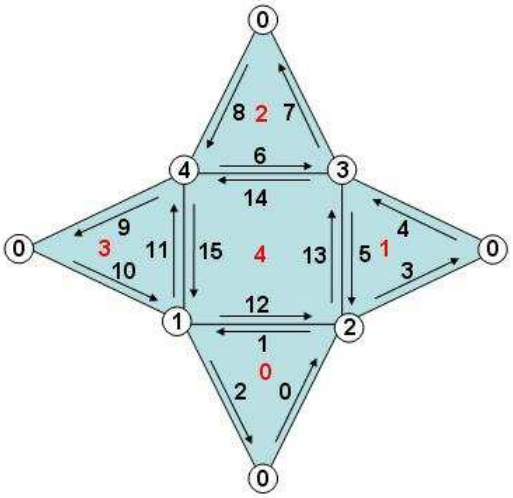


Figura 5.2: Exemplo de uma pirâmide de base quadrangular.

Assume-se que a superfície será instânciada como um objeto da classe *CHalfEdge*. As tabelas V e M foram apresentadas no Capítulo 3, as quais são mostrados na Figura 5.3. Estas tabelas serão reconstruída nos processo de descompressão a partir dos arquivos *Fclers*, *Fgeometry* e *Fhandles* porem no final terão uma numeração diferente de semi-arestas devido ao percurso no grafo dual.

A decodificação dos arquivos de entrada é seqüêncial de acordo com os CLERS codificados pelo método da compressão. Devem-se construir as tabelas de incidências e adjacências V e M. Esta malha da pirâmide é uma superfície homeomorfa fechada e esta não contem gênus. Em seguida será apresentado o processo de descompressão de uma malha irregular com genus como foi apresentada também no Capitulo 4.

O processo de decodificação e reconstrução da pirâmide começa com a leitura dos dados do primeiro triângulo ou quadrângulo de inicio, e as suas faces incidentes e adjacentes serão decodificados a cada leitura do arquivo



V[0] = 0	M[0] = 3
V[1] = 2	M[1] = 12
V[2] = 1	M[2] = 10
V[3] = 2	M[3] = 0
V[4] = 0	M[4] = 7
V[5] = 3	M[5] = 13
V[6] = 4	M[6] =14
V[7] = 3	M[7] = 4
V[8] = 0	M[8] = 9
V[9] = 4	M[9] = 8
V[10] = 0	M[10] = 2
V[11] = 1	M[11] = 15
V[12] = 1	M[12] = 1
V[13] = 2	M[13] = 5
V[14] = 3	M[14] = 6
V[15] = 4	M[15] = 11

Figura 5.3: Tabelas V e M da pirâmide a ser reconstruída..

de CLERS. Logo inicialmente temos a seguinte configuração das tabelas V e M (Figura 5.4).

V[0] = 2	M[0] = -1
V[1] = 1	M[1] = -1
V[2] = 0	M[2] = -3
V[3] = -1	M[3] = -3
V[4] = -1	M[4] = -3
V[5] = -1	M[5] = -3
V[6] = -1	M[6] = -3
V[7] = -1	M[7] = -3
V[8] = -1	M[8] = -3
V[9] = -1	M[9] = -3
V[10] = -1	M[10] = -3
V[11] = -1	M[11] = -3
V[12] = -1	M[12] = -3
V[13] = -1	M[13] = -3
V[14] = -1	M[14] = -3
V[15] = -1	M[15] = -3

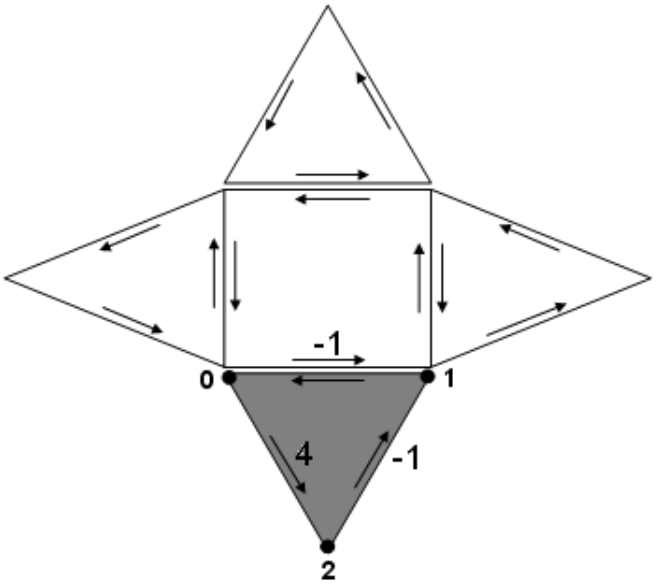


Figura 5.4: Tabelas V e M depois da leitura e reconstrução da primeira face da pirâmide.

A seguir inicia-se a leitura da seqüência de CLERS para reconstruir a malha da pirâmide. Este primeiro passo da reconstrução é o processo de Wrap.

V[0] = 2	M[0] = -1
V[1] = 1	M[1] = -1
V[2] = 0	M[2] = 4
V[3] = 3	M[3] = -1
V[4] = 2	M[4] = 2
V[5] = 0	M[5] = -3
V[6] = -1	M[6] = -3
V[7] = -1	M[7] = -3
V[8] = -1	M[8] = -3
V[9] = -1	M[9] = -3
V[10] = -1	M[10] = -3
V[11] = -1	M[11] = -3
V[12] = -1	M[12] = -3
V[13] = -1	M[13] = -3
V[14] = -1	M[14] = -3
V[15] = -1	M[15] = -3

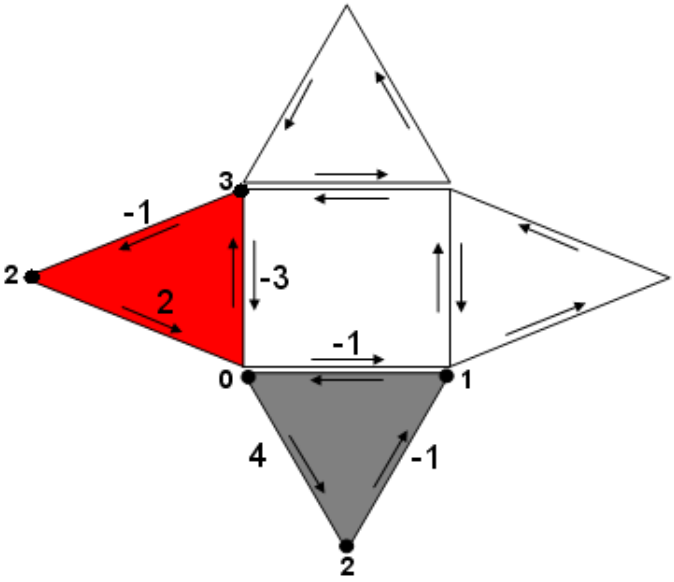


Figura 5.5: Tabelas V e M depois da decodificação do primeiro elemento da seqüência de CLERS.

O primeiro *label* a ser decodificado é do tipo C, e tem-se a seguinte configuração das tabelas V e M, conforme mostra a Figura 5.5. As semi-arestas que são portas as quais indicam o percurso de um face a outra tem os seus indices maiores que 0.

O próximo *label* do arquivo de CLERS a ser decodificado é do tipo c, o qual indica que a próxima face a ser reconstruída é um quadrângulo. Logo deve-se ler mais *label* para poder reconstruí-lo, dado que na compressão um quadrângulo é dividido em dois triângulos adjacentes. O próximo *label* da seqüência de CLERS é um R, logo temos a seqüência cR com o qual podemos reconstruir o quadrângulo (Figura 5.6).

V[0] = 2	M[0] = -1
V[1] = 1	M[1] = -1
V[2] = 0	M[2] = 4
V[3] = 3	M[3] = -1
V[4] = 2	M[4] = 2
V[5] = 0	M[5] = 13
V[6] = -1	M[6] = -3
V[7] = -1	M[7] = -3
V[8] = -1	M[8] = -3
V[9] = -1	M[9] = -3
V[10] = -1	M[10] = -3
V[11] = -1	M[11] = -3
V[12] = 4	M[12] = -1
V[13] = 3	M[13] = 5
V[14] = 0	M[14] = -2
V[15] = -1	M[15] = -3

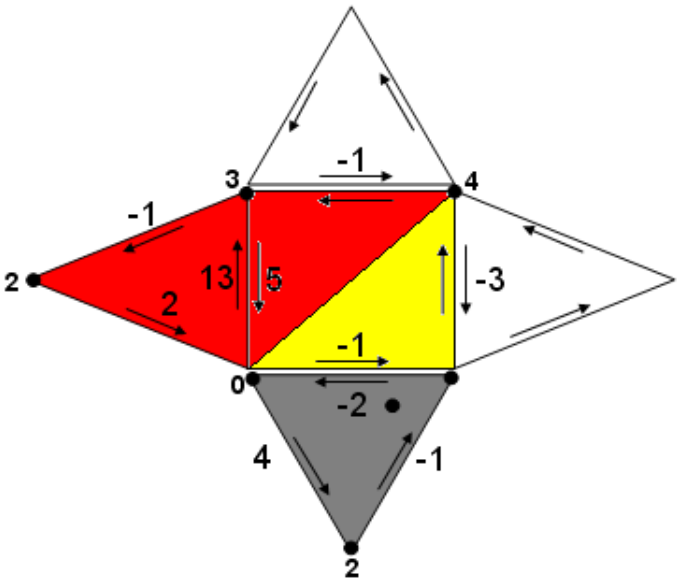


Figura 5.6: Tabelas V e M depois da decodificação da seqüência CcR de CLERS.

A Figura 5.6 mostra a reconstrução do quadrângulo pertencente à malha da pirâmide. É importante salientar que a diagonal que divide o quadrângulo em dois triângulos não existe. Esta informação está implícita na seqüência de CLERS e na regra adotada para subdivisão de quadrângulos no processo de compressão e codificação.

No próximo passo, o *label* a ser lido pelo decodificador é do tipo R. A seguinte Figura 5.7 mostra as configurações das tabelas V e M após a reconstrução do *label* R.

A proxima seqüência a ser lida do arquivo de CLERS é o *label* E, o

V[0] = 2	M[0] = -1
V[1] = 1	M[1] = -1
V[2] = 0	M[2] = 4
V[3] = 3	M[3] = -1
V[4] = 2	M[4] = 2
V[5] = 0	M[5] = 13
V[6] = -1	M[6] = -3
V[7] = 4	M[7] = 15
V[8] = -1	M[8] = -2
V[9] = -1	M[9] = -3
V[10] = -1	M[10] = -3
V[11] = -1	M[11] = -3
V[12] = 4	M[12] = -1
V[13] = 3	M[13] = 5
V[14] = 0	M[14] = -2
V[15] = -1	M[15] = 7

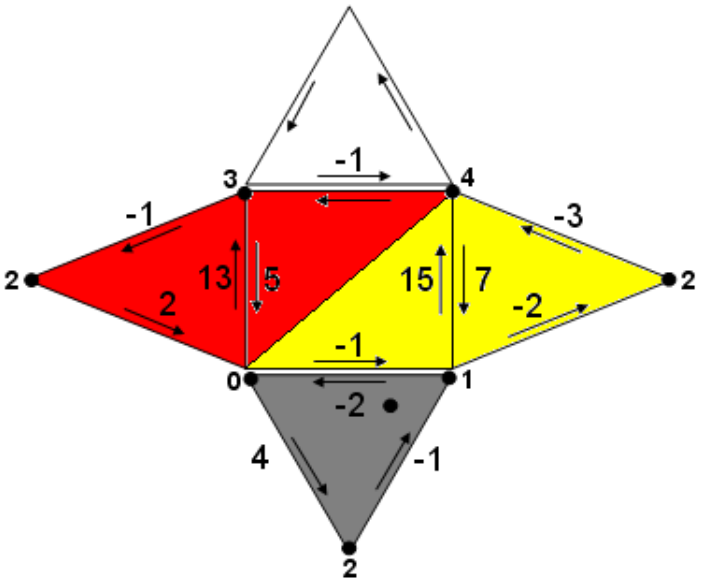


Figura 5.7: Tabelas V e M depois da decodificação da seqüência *CcRR* de CLERS.

qual indica o fim da decodificação (Figura 5.8).

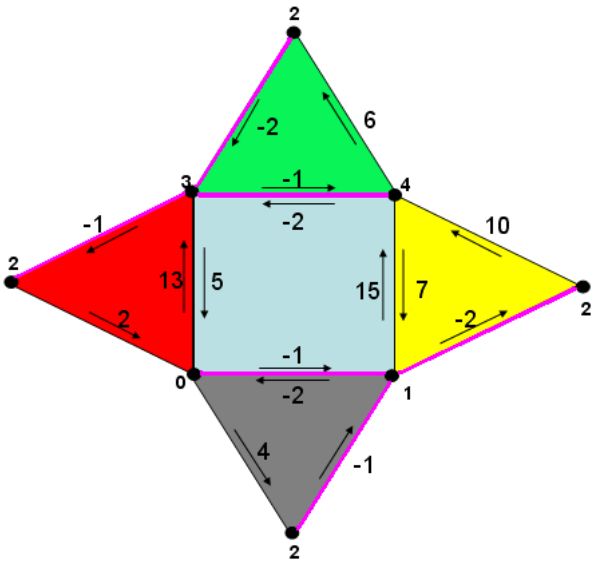


Figura 5.8: Malha da pirâmide após a decodificação da seqüência *CcRRE* de CLERS.

A Figura 5.9 mostra a configuração das tabelas V e M após a decodificação da seqüência de CLERS *CcRRE*, o processo acima especificado é chamado de Wrap, o qual reconstrói a malha codificada.

Após o processo de Wrap, deve-se atualizar os valores para cada semi-

V[0] = 2	M[0] = -1
V[1] = 1	M[1] = -1
V[2] = 0	M[2] = 4
V[3] = 3	M[3] = -1
V[4] = 2	M[4] = 2
V[5] = 0	M[5] = 13
V[6] = -1	M[6] = 10
V[7] = 4	M[7] = 15
V[8] = -1	M[8] = -2
V[9] = -1	M[9] = -2
V[10] = 4	M[10] = 6
V[11] = -1	M[11] = -2
V[12] = 4	M[12] = -1
V[13] = 3	M[13] = 5
V[14] = 0	M[14] = -2
V[15] = -1	M[15] = 7

Figura 5.9: Tabelas V e M após o processo de Wrap.

aresta fazendo-se o percurso da estrela para cada semi-aresta igual a -1 até achar a sua semi-aresta oposta com valor igual a -2. A Figura 5.8 mostra as arestas cujos valores são igual a -1 e o seus opostos estão marcados em roxo. Estas arestas deverão passar pelo processo de Zip.

O processo de Zip começa costurando o último triângulo ou quadrângulo que tem o seu *label* do tipo C ou c. Este processo de Zip é diferente ao processo de Zip implementado por [14], pois neste algoritmo utilizam-se pilhas para armazenar os *labels* do tipo S assim como também armazena-se os *labels* de tipo C ou c. Desta forma o algoritmo implementado não e recursivo, assim como também não e necessário a implementação de sub-funções para buscar as subseqüências a frente geradas pelo *label* S.

Após o processo de Zip atualizar os índices de incidências e adjacências cujos valores dos opostos das semi-arestas são menores que zero temos a malha decodificada e reconstruída (Figuras 5.10 e 5.11).

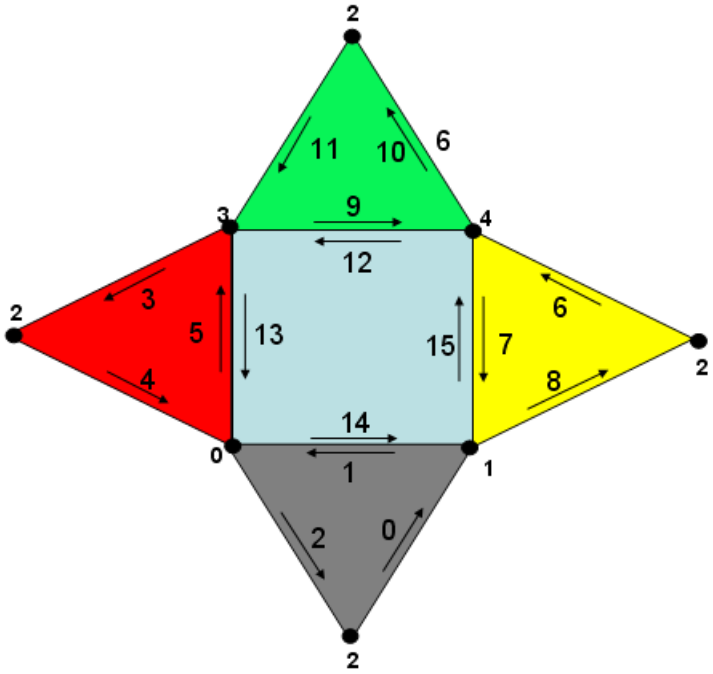


Figura 5.10: Malha da pirâmide após a decodificação completa.

V[0] = 2	M[0] = 8
V[1] = 1	M[1] = 14
V[2] = 0	M[2] = 4
V[3] = 3	M[3] = 11
V[4] = 2	M[4] = 2
V[5] = 0	M[5] = 13
V[6] = 2	M[6] = 10
V[7] = 4	M[7] = 15
V[8] = 1	M[8] = 0
V[9] = 3	M[9] = 12
V[10] = 4	M[10] = 6
V[11] = 2	M[11] = 3
V[12] = 4	M[12] = 9
V[13] = 3	M[13] = 5
V[14] = 0	M[14] = 1
V[15] = 1	M[15] = 7

Figura 5.11: Tabelas V e M após a descompressão.

A ordem de numeração das tabelas V e M da malha cujo objeto é uma pirâmide de base quadrangular é diferente da ordem de numeração das tabelas V e M reconstruídas pelo modulo *Decompress*. A cada passo de decodificação é gerada uma entrada para a tabela V e M atualizando as suas respectivas adjacências e incidências nas respectivas tabelas. Esta ordem nem sempre será o mesmo para a malha original pois depende da ordem inicial de como as *halfedges* foram numeradas para cada face na malha original.

5.4.2

Exemplo 2: Um toro

A Figura 5.12 mostra outro exemplo, que corresponde a uma superfície homeomorfa a um toro (com gênus 1). A ilustração à esquerda dessa figura mostra o modelo planar dessa superfície, onde os lados opostos do retângulo são identificados. Observe que essa malha também possui triângulos e quadrângulos.

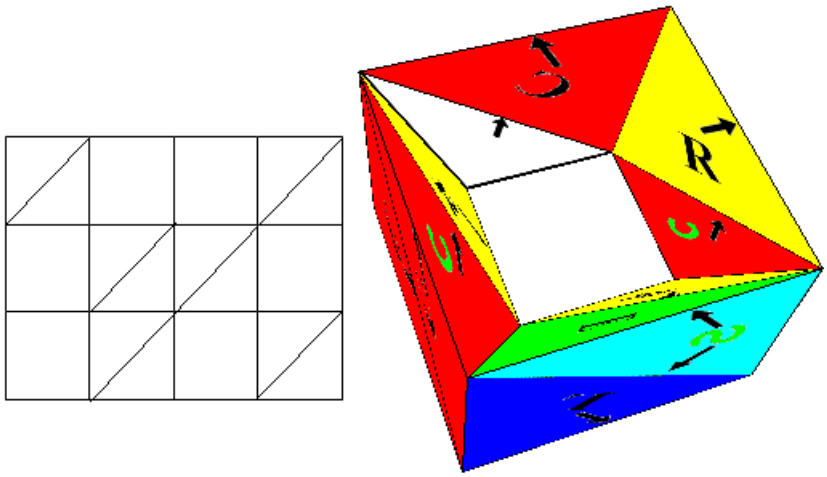


Figura 5.12: Exemplo de um toro e sua malha irregular.

Para ilustrar o algoritmo de descompressão de uma malha irregular com alças considere o torus apresentado na Figura 5.12 aqui reproduzida. Depois de atualizarmos a tabela M com os pares de semi-arestas geradas pelo arquivo *Fhandles*, reconstrói-se os índices de incidências e adjacências das tabelas V e M. A superfície resultante obtida depois da decodificação dos CLERS é mostrada na Figura 5.13. As arestas que são do tipo alças estão marcadas em marrom.

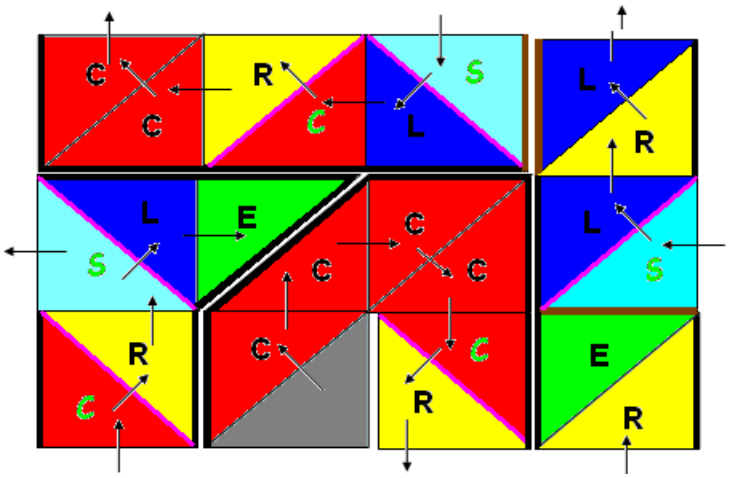


Figura 5.13: Decodificação de um torus antes de passar pelo processo de Zip.

O próximo passo da descompressão é o processo de Zip. Como mostrado no exemplo anterior, este processo começa ligando a última face que foi codificada com *label* de tipo C ou c. A Figura 5.14, mostra as arestas que foram ligadas pelo processo de Zip marcadas em roxo, e o primeiro quadrângulo que tem uma de suas arestas zipadas indicado pela desenho de uma mão.

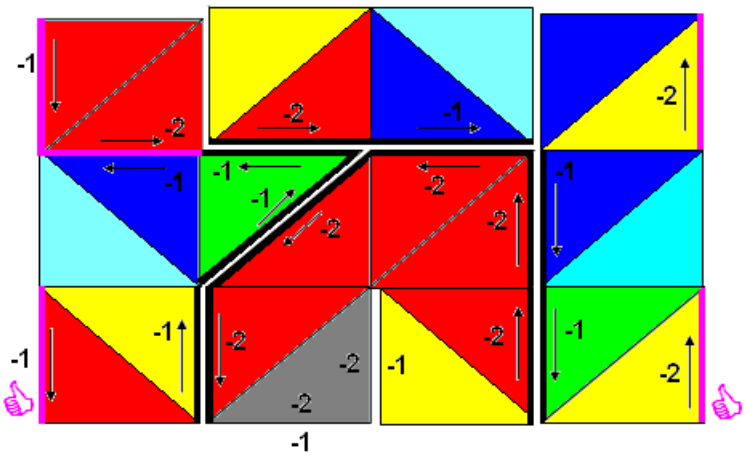


Figura 5.14: Alguns passos do processo de Zip.

Neste Capítulo foi apresentado em detalhes o algoritmo para descompressão de malhas irregulares com e sem gênus. Na próxima seção serão apresentados os resultados obtidos para vários modelos, que serão codificados e decodificados.