4
Compressão de Malhas Irregulares

No Capítulo 3 foi apresentada uma nova estrutura de dados econômica para a representação de superfícies irregulares, ou seja, compostas por triângulos e/ou quadrângulos.

Neste Capítulo apresenta-se um método para a compressão de superfícies irregulares com gênus. Este método é uma extensão dos algoritmos propostos por Lopes et al. [9] e King et al. [7].

A estrutura de dados *CHalfEdge* será usada na descrição do algoritmo.

4.1
Compressão EdgeBreaker

O algoritmo proposto para a compressão e descompressão de malhas irregulares é uma generalização dos algoritmos de compressão EdgeBreaker que foi introduzido por Rossignac em [15].

O *EdgeBreaker* é um método que codifica uma malha de triângulos construindo para isto uma árvore geradora no grafo dual da superfície, onde os nós representam as faces e as linhas representam as relações de adjacências entre elas. Ele nomeia cada triângulo com uma letra que codifica tanto a estrutura da árvore, como também a informação necessária para reconstruir os triângulos, e, assim, recuperar a malha original.

A letra que o *EdgeBreaker* atribui a um triângulo depende dos triângulos adjacentes e de um de seus vértices, caso estes já tenham sido visitados ou não. O algoritmo percorre cada triângulo da superfície entrando em cada um deles através de uma de suas arestas, que será chamada de *porta*. Conseqüentemente, a porta e os seus vértices incidentes já foram

previamente visitados. As letras codificam, portanto, o estado (visitado ou não) do vértice oposto à porta e dos triângulos à esquerda e à direita da face em questão. São 5 as combinações possíveis desses estados, para os quais são atribuídas as seguintes letras:

C: triângulo em que o vértice oposto à porta ainda não foi visitado, assim como as faces vizinhas à esquerda e à direita.

R: triângulo em que o vértice oposto à porta e a face vizinha à direita já foram visitados, mas a face à esquerda não.

L: triângulo em que o vértice oposto à porta e a face vizinha à esquerda já foram visitados, mas a face à direita não.

S: triângulo cujo o vértice oposto à porta já foi previamente visitado porém as duas faces vizinhas ainda não foram.

E: triângulo cujo vértice oposto e as duas faces vizinhas já foram visitados.

Após identificada a letra de cada triângulo, é necessário identificar qual será o próximo triângulo a ser visitado.

C: segue para a face à direita.

R: segue para a face à esquerda.

L: segue para a face à direita.

S: segue para a face à direita até não poder mais (encontrar um triângulo do tipo E), e depois segue para a esquerda.

E: é o fim do caminho.

A Figura 4.1 mostra os casos para codificação de um triângulo e a direção a seguir.

Note, que o *Edgebreaker* sempre tenta andar sempre que possível para a direita. Cada ramo que é gerado árvore geradora dual finaliza com um *label* do tipo E, e cada ramo na arvore é gerado inicialmente pela raíz da árvore ou por um *label* do tipo S (Figura 4.2).

Um triângulo do tipo S depois de andar para a direita, volta e anda para a esquerda. Isso é sempre possível para superfície homeomorfas a uma esfera. Quando a superfície sem bordo possui g gênus, existem casos em que ao retornar do ramo à direita, a face esquerda já foi visitada durante

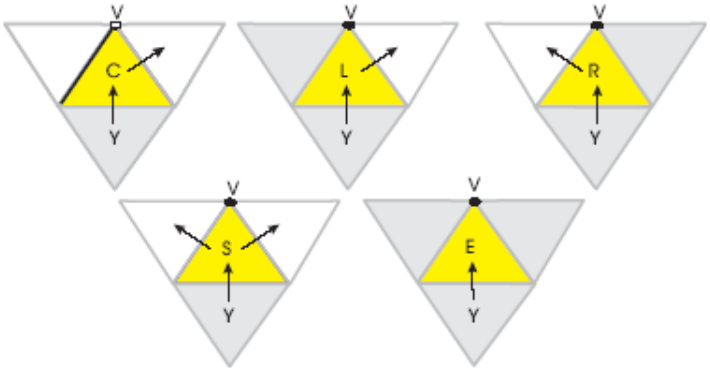


Figura 4.1: A codificação do *EdgeBreaker*.

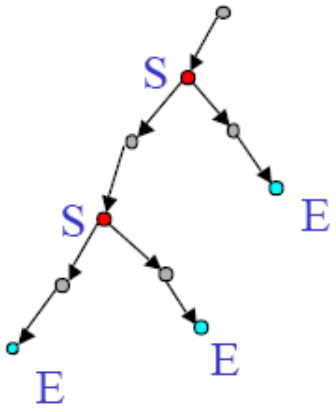


Figura 4.2: Na arvore TST(Triangle Spanning Tree), cada *label* S introduz um novo ramo e cada *label* E fecha o ramo.

o percurso do ramo a direita, e assim não seria mais possível andar, pois não se pode visitar uma face mais de uma vez. Lopes et al [9]. propuseram que que nesses casos para se obter uma correta reconstrução da topologia se transmita num arquivo separado, utilizando uma certa codificação, a aresta esquerda do triângulo S. Lopes et al. provaram que para uma superfície sem bordo com g gênus, existirão exatamente $2g$ triângulos S desse tipo. Esse número $2g$ corresponde exatamente ao número de operadores de alças do tipo 1 que devem ser aplicados, na descompressão à superfície para reconstruir corretamente a sua topologia. Por essa razão, essas arestas especiais serão, por abuso de linguagem, igualmente chamadas de alças.

4.2
Extensão do EdgeBreaker para Quadrângulos

Uma maneira simples de comprimir malhas de quadrângulos ou uma malha contendo outros polígonos com mais de três vértices, é de dividir cada polígono em dois triângulos e aplicar o algoritmo de compressão para malhas de triângulos. Porém, no momento da descompressão da malha se desejaria poder reconhecer quais são os faces a mais que foram adicionadas ao comprimir a malha e desta forma obter as faces originais. Para isso, seria necessário adicionar mais informação ao codificador para poder identificar corretamente no momento da decodificação qual é o número de vértices de cada face.

O ponto principal do método de compressão para malhas compostas por quadrângulos proposto por King et al [7]. é a de dividir cada quadrângulo em dois triângulos de acordo a uma regra que assegure que os dois triângulos criados a partir de um quadrângulo sejam adjacentes ao percurso da arvore geradora do grafo dual. Esta regra de divisão é fácil de implementar, e permite recuperar os quadrângulos originais durante a descompressão, simplesmente juntando o par de triângulos adjacentes.

Para um triângulo existem 5 possíveis combinações de arestas e vértices visitados e não visitados, os quais correspondem aos 5 *labels* dos CLERS já apresentados no capítulo anterior. Para outros tipos de polígonos, o número de combinações pode ser calculado via a relação de recorrência, baseado no fato que nenhum vértice não visitado pode ser incidente a uma aresta previamente já visitada. A solução para esta recorrência, proposta por King et al [7], para um polígono com n arestas é o número de Fibonaci $F(2n-1)$. Logo, para um quadrângulo, temos $F(7) = 13$ possíveis combinações de arestas e vértices não adjacentes a porta visitados e não visitados.

Para identificarmos as 13 possíveis combinações de arestas e vértices visitados em cada quadrângulo, dividimos o quadrângulo em dois triângulos de acordo a regra de divisão,a qual garante que os triângulos são adjacentes na arvore geradora dual resultante. Logo, cada quadrângulo será codificado utilizando duas letras.

O percurso de *EdgeBreaker* tenta andar à direita sempre que possível. King et al. propuseram que ao entrar num quadrângulo se adicione implicitamente uma aresta diagonal, dividindo, portanto, a face em dois triângulos.

O ideal é que os dois triângulos de um quadrângulo permaneçam adjacentes na árvore geradora do grafo dual. Isto pode ser feito dividindo cada quadrângulo ao longo da diagonal de tal forma que o segundo triângulo esteja sempre à direita do primeiro (Figura 4.3).

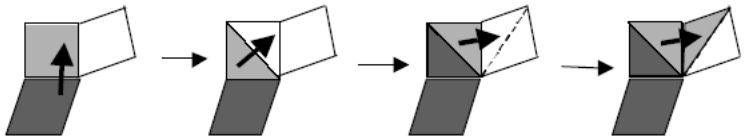


Figura 4.3: Divisão de um quadrângulo em dois triângulos.

Logo, as seguintes situações ocorrem: após a adição de uma aresta devido à subdivisão de um quadrângulo, que por razões obvias não foi previamente visitada, a regra de divisão não permite que o primeiro *label* do quadrângulo seja um R ou E. Além disso, se o primeiro triângulo de um quadrângulo começa com uma letra do tipo C, a segunda letra não pode ser um L ou E (Figura 4.4).

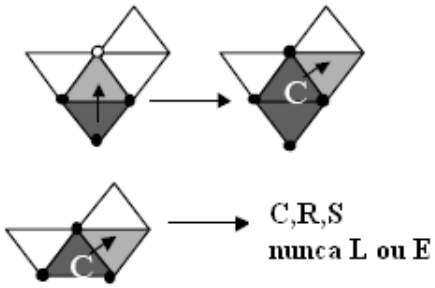


Figura 4.4: Processo de compressão iniciando com um *label* do tipo C.

A regra de divisão acima proposta leva a 13 possíveis pares de combinações de letras que um quadrângulo pode ter: CC, CR, CS, LC, LL, LE, LR, LS, SC, SL, SE, SR e SS.

Para mover de um quadrângulo adjacente para outro a fim de construir a árvore geradora do grafo dual, utilizam-se as regras definidas para os triângulos no *Edgebreaker* original para cada um dos triângulos do retângulo.

4.3
Codificação para superfícies irregulares

Com base no exposto na seção anterior, será apresentada uma nova proposta para codificação de superfícies compostas por triângulos e/ou quadrângulos.

Essa proposta consiste nas seguintes regras para a codificação de uma face:

1. Se a face for um triângulo, utiliza-se as letras C,L,R,E, ou S como no *EdgeBreaker* original.

2. Se a face for um quadrângulo, divide-se ele em dois triângulos segundo a proposta de King et al [7]., e codifica-se o primeiro triângulo utilizando as letras c, l, ou s, e o segundo triângulo com as letras C,L,R,E, ou S.

A maneira de classificar o primeiro triângulo do quadrângulo como c,l e s é a mesma de classificação dos triângulos do *Edgebreaker* original. A direção de saída dessas faces é sempre o segundo triângulo do quadrângulo.

Uma grande vantagem dessa escolha é que desta forma pode se identificar claramente quando é que se está entrando num quadrângulo ou num triângulo. Assim as codificações possíveis para os dois triângulos de um quadrângulo são: cC, cR, cS, lC, lL, lE, lR, lS, sC, sL, sE, sR e sS.

Os *labels* c,l e s assim como os labels C,L e S são utilizadas respectivamente para indicar a codificação de um quadrângulo ou um triângulo. Cada vez que um quadrângulo é encontrado codifica-se de tal forma que este é divido em dois triângulos adjacentes. Desta forma temos os 13 casos possíveis de codificação para quadrângulos indicados no parágrafo anterior.

Para codificar superfícies conexas com g gênus, será usada a mesma estratégia proposta por Lopes et al [9]. para os casos especiais do triângulo S, que consiste em transmitir, em um arquivo separado, a aresta esquerda desse triângulo. Será provado nesse trabalho, que mesmo para superfícies com triângulo e/ou quadrângulos, existirão exatamente $2g$ arestas a serem transmitidas à parte.

4.4
Compressão de superfícies irregulares com gênus

Nesta seção será descrita a implementação do algoritmo de compressão 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 $\mathcal{S}$ está instanciada como um objeto da classe *CHalfEdge*.

4.4.1
Inicialização da Compressão

A seguir descreve-se o procedimento que inicia o processo da compressão, nomeado de *WriteEB*. Tem-se como parâmetros de entrada o índice da semi-aresta que define a primeira face a ser visitada, e uma cadeia de caracteres que define o nome dos arquivos que serão gerados.

No procedimento *WriteEB*, inicialmente são definidas e iniciadas algumas variáveis auxiliares que serão usadas globalmente nas rotinas do algoritmo de compressão. Essas variáveis auxiliares são:

- int *MVertices; uma tabela de inteiros de tamanho NV, em que a entrada i indica se o vértice i já foi visitado ou não. Atribui-se o valor inicial 0 para todas as entradas.
- int *MTriangles; uma tabela de inteiros de tamanho NT, em que a entrada i indica se o triângulo i já foi visitado ou não. Atribui-se o valor inicial -1 para todas as entradas.
- int *MQuads; uma tabela de inteiros de tamanho NQ, em que a entrada i indica se o quadrângulo i já foi visitado ou não. Atribui-se o valor inicial -1 para todas as entradas.
- int *H; uma tabela de inteiros de tamanho $3NT + 4NQ$, em que a entrada i indica o índice que a semi-aresta i receberá no processo de descompressão. Atribui-se o valor inicial -1 para todas as entradas.
- int T; um número inteiro que indica o índice do próximo triângulo a ser visitado. O seu valor inicial é 0.

– int Q; um número inteiro que indica o índice do próximo quadrângulo a ser visitado. O seu valor inicial é 0.

A rotina *InitGlobalVariables* inicia essas variáveis.

No algoritmo são três os arquivos a serem gerados. O arquivo da seqüência de símbolos c, l, s, C, L ,E ,R e S das faces, chamado de arquivo *Fclers*. O arquivo da geometria dos vértices, chamado de arquivo *Fgeometry*. E, finalmente, o arquivo das alças, chamado de arquivo *Fhandle*. A esses três diferentes arquivos foram convencionadas as seguintes extensões, respectivamente: *.eb*, *.geo*, *.top*.

Após a iniciação das variáveis e a abertura desses arquivos, a rotina *WriteEB* grava no arquivo *Fgeometry* o número de vértices, o número de triângulos e o número de quadrângulos da superfície $\mathcal{S}$ e no arquivo *Fhandles* o número de alças, que corresponde a $2-(NT+NQ)-\frac{(3NT+4NQ)}{2}-NV$. Em seguida, o algoritmo visita a primeira face gravando no arquivo *Fgeometry* as coordenadas dos vértices dessa face em uma ordem fixada através das rotinas *SaveFirstTriangle* ou *SaveFirstQuad*. Essa ordem dos vértices dá início à forma espiralada de visitação que o *Edgebreaker* implicitamente impõe ao tentar andar sempre que possível para a direita. Finalmente, é chamada a rotina Compress que faz a compressão propriamente dita.

Segue o código da rotina *WriteEB*, *InitGlobalVariables*, e das rotinas *SaveFirstTriangle* e *SaveFirstQuadrangle*. As duas últimas recebem como parâmetro a semi-aresta da face inicial.

algoritmo `void CHalfEdge::WriteEB`(int nStartHalfEdge, char *name)

```
 1 InitGlobalVariables();
 2 char clersname[50];
 3 char geoname[50];
 4 char topname[50];
 5 sprint(clersname,"%s.eb",name);
 6 sprint(geoname,"%s.geo",name);
 7 sprint(topname,"%s.top",name);
 8 //Gera os arquivos para escritura de dados
 9 fstream Fclers(clersname,ios::out);
10 fstream Fgeometry(geoname,ios::out);
11 fstream Fhandles(topname,ios::out);
12 int NH = 2 - ((NT+NQ) -(3*NT+4*NQ)/2+ NV);
13 //Grava no arquivo número de alças
14 Fhandles << NH;
15 //Grava no arquivo número de vértices
16 Fgeometry<< NV;
17 //Grava no arquivo número de triângulos
18 Fgeometry<< NT;
19 //Grava no arquivo número de quadrângulos
20 Fgeometry<< NQ;
21 //Comprime primeiro triângulo da malha
22 if (facetype(nStartHalfedge) == 3)
23 {
24    Fgeometry<< 3;
25    SaveFirstTriangle(nStartHalfEdge);
26 }
27 //Comprime primeiro quadrângulo da malha
28 else
29 {
30    Fgeometry<< 4;
31    SaveFirstQuad(nStartHalfEdge);
32 }
33 //Processo de compressão da malha
34 Compress(prev(M[prev(nStartHalfEdge)]),)
35 //Remove variáveis usadas da memória
36 delete [] H; delete [] Mvertices;
37 delete [] Mtriangles; delete [] Mquads;
38 //Fecha arquivos utilizados
39 Fclers.close(); Fgeometry.close(); Fhandles.close();
```

fim

Algoritmo 21: Iniciação do módulo de Compressão WriteEB.

```
algoritmo void InitGlobalVariables(int he)
 1 //Modulo que inicia as variáveis globais respectivas
 2 //Gera vetor de vértices
 3 Mvertices = new int[NV];
 4 //Gera vetor de triângulos
 5 Mtriangles = new int[NT];
 6 //Gera vetor de quadrangulos
 7 Mquads = new int[NQ];
 8 //Gera vetor de halfEdges para triângulos e quadrângulos
 9 H = new int[3*NT+4*NQ];
10 //Inicializa valores dos vetores
11 for ( i = 0 ; i < NV ; i++ )
12 {
13    Mvertices[i] = 0;
14 }
15 for ( i = 0 ; i < NT ; i++ )
16 {
17    Mtriangles[i] = -1;
18 }
19 for ( i = 0 ; i < NQ ; i++ )
20 {
21    Mquads[i] = -1;
22 }
23 for ( i = 0 ; i < 3*NT+4*NQ ; i++ )
24 {
25    H[i] = -1;
26 }
27 //Índice do próximo triângulo a ser visitado
28 T = 0;
29 //Índice do próximo quadrângulo a ser visitado
30 Q = 0;
fim
```

Algoritmo 22: Sub-módulo 1 da Compressão WriteEB.

```
algoritmo void SaveFirstTriangle(int he)
 1 //Grava as coordenadas do triângulo no arquivo de geometria
 2 Fgeometry<< G[V[prev(he)]][0] << G[V[prev(he)]][1] <<
   G[V[prev(he)]][2]);
 3 Mvertices[V[prev(he)]] = 1;
 4 Fgeometry<< G[V[next(he)]][0] << G[V[next(he)]][1] <<
   G[V[next(he)]][2]);
 5 Mvertices[V[next(he)]] = 1;
 6 Fgeometry<< G[V[he]][0] << G[V[he]][1] << G[V[he]][2] <<
   endl;
 7 //Marca o vértice e triângulo como visitados
 8 Mvertices[V[he]] = 1;
 9 Mtriangles[trig(he)] = 0;
10 T++;
11 //Constrói tabela de halfedges
12 H[he]= 0; H[next(he)] = 1; H[prev(he)] = 2;
fim
```

Algoritmo 23: Sub-módulo 2 da Compressão WriteEB.

```
algoritmo void SaveFirstQuad(int he)
 1 //Grava as coordenadas do quadrângulo no arquivo de geome-
   tria
 2 Fgeometry<< G[V[prev(he)]][0] << G[V[prev(he)]][1] <<
   G[V[prev(he)]][2]);
 3 Mvertices[V[prev(he)]] = 1;
 4 Fgeometry<< G[V[next(next(he))]][0];
 5 Fgeometry<< G[V[next(next(he))]][1];
 6 Fgeometry<< G[V[next(next(he))]][2]);
 7 Mvertices[V[next(next(he))]] = 1;
 8 Fgeometry<< G[V[next(he)]][0];
 9 Fgeometry<< G[V[next(he)]][1];
10 Fgeometry<< G[V[next(he)]][2]);
11 Mvertices[V[next(he)]] = 1;
12 Fgeometry<< G[V[he]][0] << G[V[he]][1] << G[V[he]][2]);
13 //Marca o vértice e quadrângulo como visitados
14 Mvertices[V[he]] = 1;
15 Mtriangles[quad(he)] = 3*NT;
16 Q++;
17 //Constrói tabela de halfedges
18 H[he]= 0; H[next(he)] = 1;
19 H[next(next(he))] = 2; H[prev(he)] = 3;
fim
```

Algoritmo 24: Sub-módulo 3 da Compressão WriteEB.

4.4.2

A Compressão

A rotina *Compress* é a principal do algoritmo de compressão. Ela visita todas as faces da superfície, passando por cada uma delas uma só vez, i.e., ela cria uma árvore geradora no grafo dual da superfície. Para cada triângulo visitado atribui-se uma letra (C,L,E,R, ou S). A cada quadrângulo visitado atribuem-se duas letras são atribuídas (cC, cR, cS, lC, lL, lE, lR, lS, sC, sL, sE, sR ou sS).

No início desse capítulo ficou definido como classificar essas faces e ficou fixado qual o caminho a seguir depois de atribuir um símbolo a ela. Foi dito que um triângulo do tipo s ou S pode gerar uma bifurcação no caminho.

O processo de visitação começa com uma face inicial dada, e termina quando não existe mais triângulos ou quadrângulos a serem visitados. Para isso uma busca em profundidade é feita.

É muito comum que algoritmos para busca em profundidade sejam implementados recursivamente. Mas nesse 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 classificada como s ou S, então a semi-aresta pivô é inserida na 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 classificada 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 e será armazenado no arquivo *Fhandles*. Essa aresta é codificada, escrevendo nesse arquivo dois números inteiros que correspondem ao índice que as suas duas semi-arestas receberão no decodificador. Esses índices são

computados durante a compressão e armazenados na tabela *H*. Finalmente, o caso em que não há mais face a visitar acontece quando não existir mais elementos na pilha.

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 *CompressTriangle* ou *CompressQuadrangle*, respectivamente.

O algoritmo 25 mostra a implementação do esquema de compressão.

```
algoritmo Compress(int HalfEdge)
 1 //Variáveis que indicam fim de compressão e
 2 //alças á esquerda e direita respectivamente
 3 int IsAnEnd,heL,heS;
 4 Stack stack();
 5 //Empilha halfedge
 6 stack.Push(HalfEdge);
 7 //Compressão da malha enquanto pilha não estiver vazia
 8 //ou encontra-se indicador isAndEnd com valor igual a 1
 9 While(!(stack.isEmpty()))
10 {
11   //Desempilha halfedge
12   HalfEdge = stack.Pop();
13   //Caso a Face já foi visitada achou-se alcas
14   if (VisitedFace(HalfEdge,Mtriangles,Mquads))
15   {
16     //Armazena dados no arquivo de alças
17     heL = H[next(HalfEdge)];
18     heS = H[M[next(HalfEdge)]];
19     Fhandles << heL << heS;
20   }
21   do
22   {
23     //Verifica tipo de face a comprimir
24     if (facetype(nStartHalfedge) == 3)
25       IsAnEnd = CompressTriangle(HalfEdge);
26     then
27       IsAnEnd = CompressQuad(HalfEdge);
28     if (IsAnEnd)
29       break;
30   }while(true);
31 }
fim
```

Algoritmo 25: Modulo Geral para Compressão de Malhas Irregulares.

4.4.3
Compressão de Triângulo

O algoritmo 26 codifica 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 classificado como E e 0 caso contrário.

Primeiramente, atualiza-se a tabela H armazenado para cada semi-aresta o índice que ela receberá na descompressão. Esses índices são extremamente importantes para a codificação das alças. Em seguida, o triângulo é marcado como visitado e o índice T para o próximo triângulo é incrementado.

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 classificado como C e os seguintes passos são seguidos: grava-se no arquivo *Fclers* a letra C; armazena-se no 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.

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 classificado como do tipo L, E, R, ou S, de acordo com a proposta original do *Edgebreaker*. A rotina *CodeTriangle* faz essa classificação (algoritmo 26).

```
algoritmo CompressTriangle(int &HalfEdge)
 1 //Compressão de um triângulo dada uma halfedge
 2 //Constrói tabela de halfedges
 3 H[HalfEdge] = 3*T; H[next(HalfEdge)] = 3*T+1;
 4 H[prev(HalfEdge)] = 3*T+2;
 5 //Marca triângulo como visitado
 6 Mtriangles[trig(HalfEdge)] = 3*T; T++;
 7 //Verifica estado do vértice corrente,
 8 //isto é, se foi visitado ou não foi visitado
 9 if (Mvertices[V[HalfEdge]] != 1)
10 {
11   //Vértice não foi visitado e codifica com label tipo C
12   //Grava informações das coordenadas do vértice
13   Fgeometry << G[V[HalfEdge]][0];
14   Fgeometry << G[V[HalfEdge]][1];
15   Fgeometry << G[V[HalfEdge]][2] << endl;
16   Mvertices[V[HalfEdge]] = 1;
17   Fclers << 'C';
18   HalfEdge = prev(M[prev(HalfEdge)]);
19   return 0;
20 }
21 //Vértice já foi visitado e codifica com label tipo L,E, R ou S
22 then
23 {
24   return CodeTriangle(HalfEdge);
25 }
fim
```

Algoritmo 26: Módulo de compressão para triângulos.

```
algoritmo CodeTriangle(int &HalfEdge)
 1 //Codifica triângulo com labels L, E, R ou S
 2 if (VisitedFace(M[prev(HalfEdge)],Mtriangles,Mquads))
 3 {
 4   //Caso em q face a esquerda e direita do vértice
 5   //já foram visitadas e codifica como E.
 6   if (VisitedFace(M[HalfEdge],Mtriangles,Mquads))
 7   {
 8     Fclers << 'E';
 9     return 1;
10   }
11   //Face a direita do vértice foi visitada
12   //codifica como R.
13   then
14   {
15     Fclers << 'R';
16     HalfEdge = prev(M[HalfEdge]);
17     return 0;
18   }
19 }
20 then
21 {
22   //Face a esquerda do vértice foi visitada
23   //codifica como L.
24   if (VisitedFace(M[HalfEdge],Mtriangles,Mquads))
25   {
26     Fclers << 'L';
27     HalfEdge = prev(M[prev(HalfEdge)]);
28     return 0;
29   }
30   //Nem a face a esquerda e direita do vértice foram visitadas
31   //codifica como S.
32   then
33   {
34     Fclers << 'S';
35     stack.Push(prev(M[HalfEdge]));
36     HalfEdge = prev(M[prev(HalfEdge)]);
37     return 0;
38   }
39 }
fim
```

Algoritmo 27: Sub-módulo de codificação de triângulos.

4.4.4
Compressão de Quadrângulo

A compressão de cada quadrângulo é obtida pelo algoritmo 28. Para comprimir um quadrângulo, deve-se dividir o quadrângulo em dois triângulos adjacentes de tal forma que o primeiro sempre esteja à esquerda do segundo.

O módulo para compressão de quadrângulos foi naturalmente dividido em duas etapas. Uma para identificar o primeiro triângulo do quadrângulo (algoritmo 29), e outra para codificar o segundo triângulo (algoritmo 30).

O processo de compressão é o mesmo processo para comprimir triângulos, porem com algumas restrições. O triângulo de início de um quadrângulo somente poderá ser codificado com os seguintes *labels*: c, l, ou s. Os triângulos do segundo quadrângulo serão codificados com a mesma codificação indicada no algoritmo 26. Isto é os *labels* serão C, L, E, R, e S.

```
algoritmo CompressQuad(int &HalfEdge)
 1 //Codifica quadrângulo dada uma halfedge
 2 //Constrói tabela de halfedges referente a este quadrângulo
 3 H[HalfEdge] = 3*ntriangles+4*Q;
 4 H[next(HalfEdge)] = 3*ntriangles+4*Q+1;
 5 H[next(next(HalfEdge))] = 3*ntriangles+4*Q+2;
 6 H[prev(HalfEdge)] = 3*ntriangles+4*Q+3;
 7 //Marca quadrângulo como visitado
 8 Mquads[quad(HalfEdge)] = 3*ntriangles+4*Q;
 9 //Incrementa índice do quadrângulo já visitado
10 Q++;
11 //Codifica primeiro triângulo do quadrângulo
12 EncodeFirstTriangle(HalfEdge);
13 //Codifica segundo triângulo do quadrângulo
14 IsAnEnd = EncodeSecondTriangle(prev(HalfEdge));
15 return IsAnEnd;
fim
```

Algoritmo 28: Módulo de compressão para quadrângulos.

```
algoritmo EncodeFirstTriangle(int &HalfEdge)
 1 //Codifica primeiro triângulo do quadrângulo
 2 //verifica se o vértice ainda não foi visitado
 3 if (Mvertices[V[HalfEdge]] != 1)
 4 {
 5   //Caso nao foi visitado codifica com label c,
 6   //labels em minúscula indicam inicio de um quadrângulo
 7   //Grava coordenadas do vértice e codifica como label c
 8   Fgeometry << G[V[HalfEdge]][0] << G[V[HalfEdge]][1];
 9   Fgeometry << G[V[HalfEdge]][2];
10   Mvertices[V[HalfEdge]] = 1;
11   Fclers << 'c';
12 }
13 then
14 {
15   //A face já visitada codifica com label l
16   if (VisitedFace(M[HalfEdge],Mtriangles,Mquads))
17   {
18     Fclers << 'l';
19   }
20   //caso contrario codifica com label s
21   then
22   {
23     Fclers << 's';
24     Empilha halfedge de label do tipo s
25     stack.Push(prev(M[HalfEdge]));
26   }
27 }
fim
```

Algoritmo 29: Sub-módulo para codificar o primeiro triângulo de um quadrângulo.

```
algoritmo EncodeSecondTriangle(int &HalfEdge)
 1 //Pergunta se vértice já foi visitado
 2 if (Mvertices[V[HalfEdge]] != 1)
 3 {
 4   //Grava coordenadas do vértice e codifica como label C
 5   Fgeometry << G[V[HalfEdge]][0] << G[V[HalfEdge]][1];
 6   Fgeometry << G[V[HalfEdge]][2];
 7   Mvertices[V[HalfEdge]] = 1;
 8   Fclers << 'C'; HalfEdge = prev(M[prev(HalfEdge)]);
 9   return 0;
10 }
11 then
12 {
13   if (VisitedFace(M[prev(HalfEdge)],Mtriangles,Mquads))
14   {
15     //Face a esquerda e direita já foram visitadas
16     if (VisitedFace(M[HalfEdge],Mtriangles,Mquads))
17     { Fclers << 'E'; return 1;}
18     else
19     {
20       //Face a direita do vértice foi visitada
21       Fclers << 'R'; HalfEdge = prev(M[HalfEdge]);
22       return 0;
23     }
24   }
25   else
26   {
27     //Face a esquerda do vértice foi visitada
28     if (VisitedFace(M[HalfEdge],Mtriangles,Mquads))
29     {
30       Fclers << 'L';
31       prev(M[prev(HalfEdge)]); return 0;
32     }
33     //Face a esquerda e direita do vértice não foram visitadas
34     then
35     {
36       Fclers << 'S'; stack.Push(prev(M[HalfEdge]));
37       HalfEdge = prev(M[prev(HalfEdge)]); return 0;
38     }
39   }
40 }
fim
```

Algoritmo 30: Sub-módulo para codificar o segundo triângulo de um quadrângulo.

4.5
Ilustração do algoritmo

4.5.1
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.

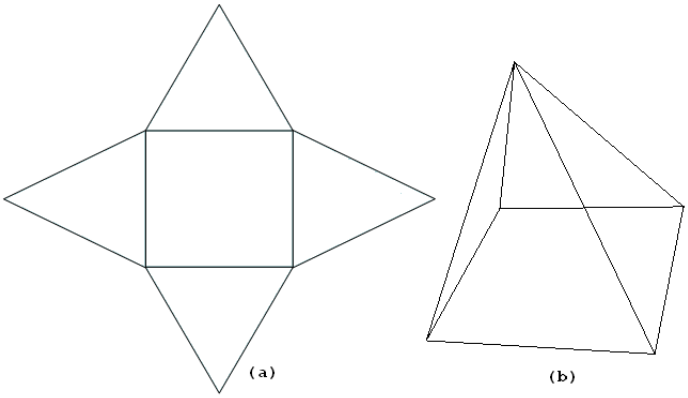
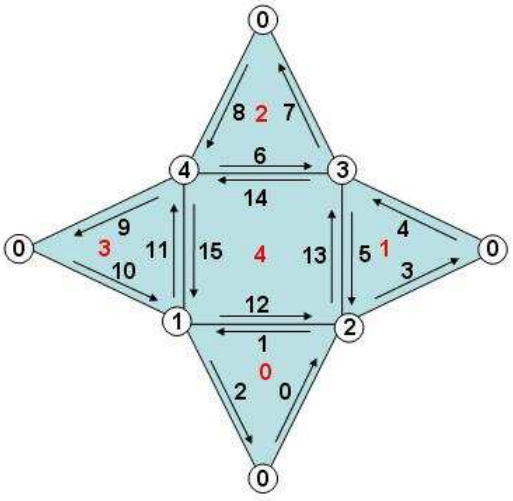


Figura 4.5: Exemplo de uma pirâmide de base quadrangular.

Assume-se que a superfície foi instânciada como um objeto da classe *CHalfEdge*. As tabelas V e M foram apresentadas no Capítulo anterior, na Figura 5.3.



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 4.6: Tabelas da Estrutura de Dados *CHalEdge*.

No processo de iniciação da compressão são gravados no arquivo *Fgeometry* o número de vértices, o número de triângulo e o número de quadrângulos que nesse caso são 5, 4, e 1, respectivamente. É gravado também no arquivo *Fhandles* o número 0, pois a superfície não possui gênus.

Como a primeira face escolhida ficou sendo a da semi-aresta 0, deve-se gravar no arquivo *Fgeometry* o número 3, indicando que a face inicial é um triângulo. Em seguida, no arquivo *Fgeometry* são armazenados os três vértices da face inicial (desenhado em cinza na Figura 4.7).

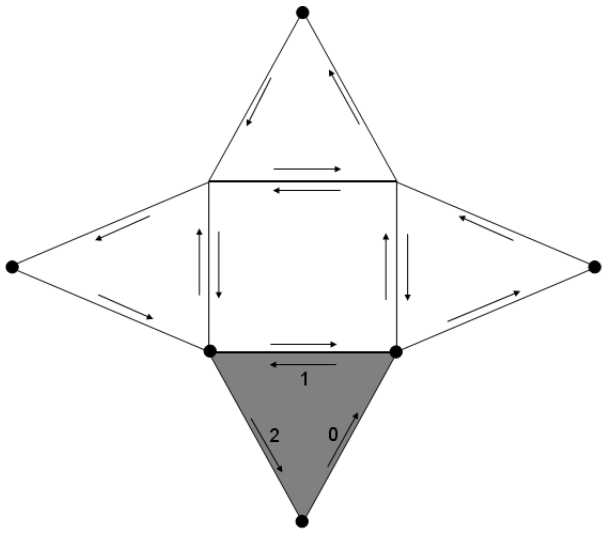


Figura 4.7: Inicio da compressão.

No passo seguinte da compressão, como ilustrado na Figura 4.8 , a próxima face é um triângulo e este é classificado como C porque o vértice oposto à porta ainda não foi visitado.

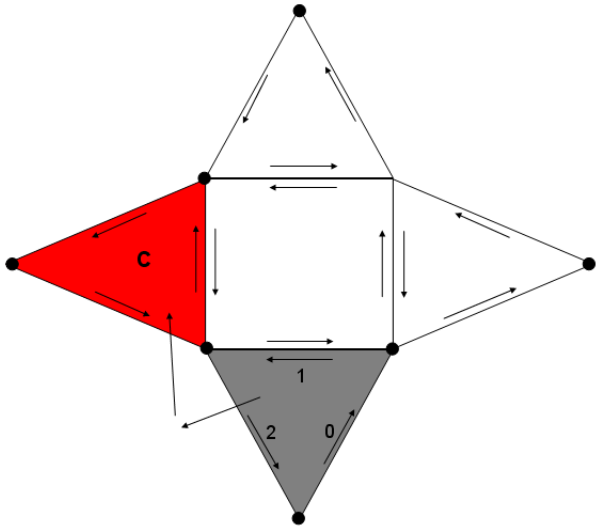


Figura 4.8: Codificação do segundo triângulo da pirâmide.

Ao sair desse triângulo do tipo C, entra-se num quadrângulo. A Figura 4.9 mostra a divisão dessa face em dois triângulos, onde o primeiro ficou classificado como c, pois o vértice oposto á porta também não foi ainda visitado.

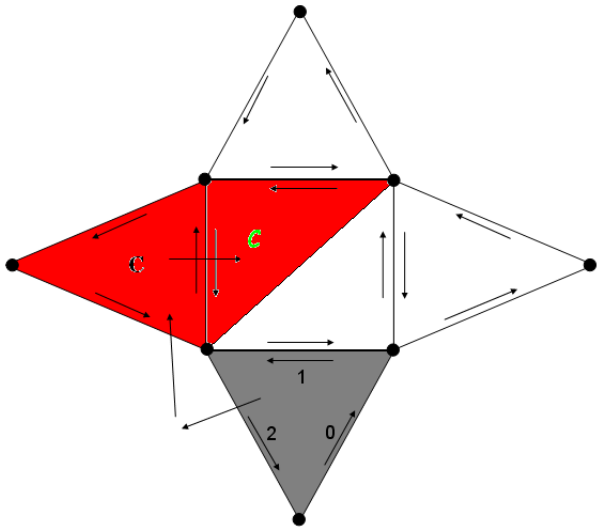


Figura 4.9: Codificação do primeiro triângulo do quadrângulo da pirâmide.

A Figura 4.10 mostra a codificação do segundo triângulo do quadrângulo que foi codificado como R. A próxima face a ser visitada é face à esquerda.

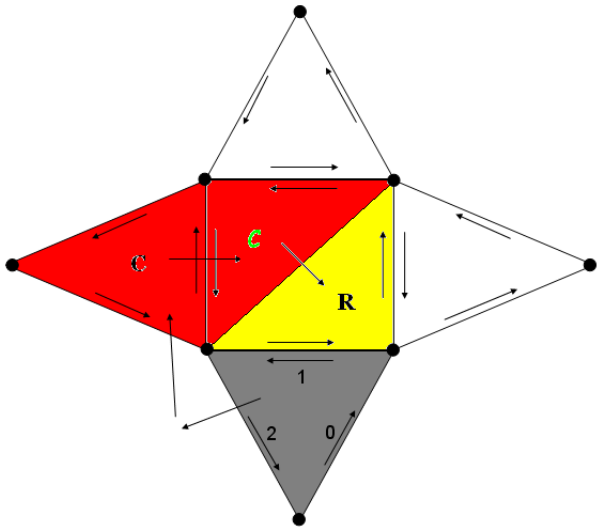


Figura 4.10: Codificação do segundo do quadrângulo da pirâmide.

No quinto passo da compressão, a face atual que é um triângulo é codificada como R pois o triângulo a direita já foi visitado. Após a codificação deste triângulo move-se novamente para a face à esquerda (Figura 4.11).

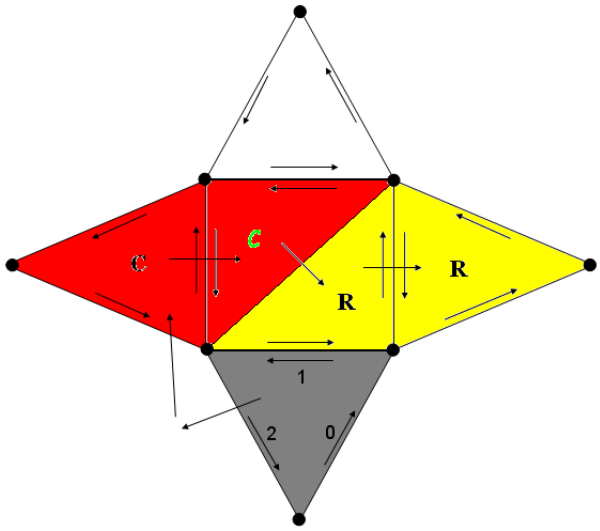


Figura 4.11: Codificação do terceiro triângulo da pirâmide.

No sexto e último passo da compressão, a face em questão é o quarto triângulo da malha a ser visitado e é classificado como E (Figura 4.12). E assim o algoritmo termina.

A rotina de compressão gera a seguinte seqüência no arquivo *Fclers*: CcRRE para o exemplo acima considerado.

A variação da seqüencia dos *labels* de minúscula para maiúscula deve-

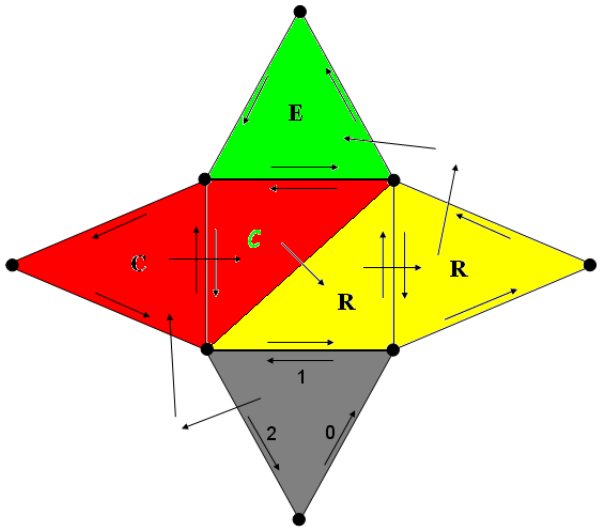


Figura 4.12: Codificação do ultimo triângulo da pirâmide.

se ao seguinte fato: Cada triângulo é codificado de acordo a máquina de estados que *EdgeBreaker* impõe para os *labels* C, L, E, R ou S. A mudança de um label para outro indicam a codificação de um triângulo corrente para outro adjacente a este.

Quando o *EdgeBreaker* codifica quadrângulos a regra de codificação adotada foi a de dividir o quadrângulo em dois triângulos, de tal forma que o segundo triângulo sempre esteja adjacente ao primeiro. Deste modo deve-se indicar de alguma forma o inicio e fim de um quadrângulo para o qual o primeiro triângulo é codificado com um *label* em letras minúsculas e o segundo quadrângulo é codificado com um *label* em letras maiúsculas indicando desta forma o inicio e fim de um quadrângulo.

4.5.2

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.

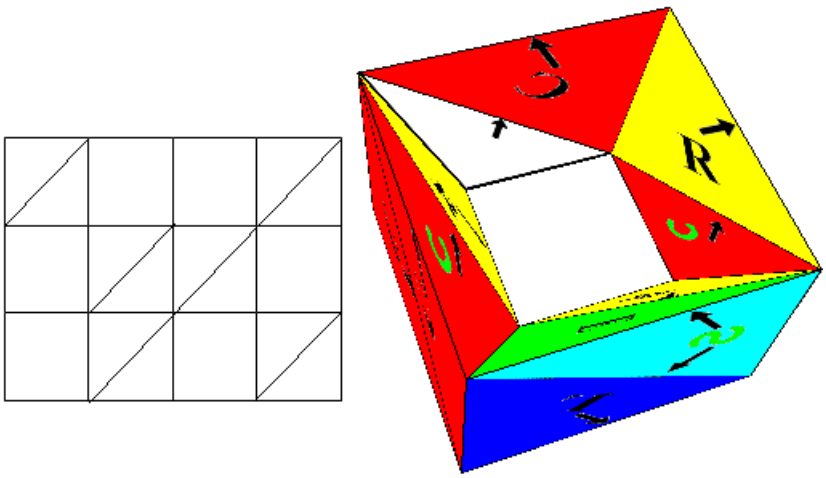


Figura 4.13: Exemplo de um toro e sua malha irregular.

A Figura 4.14 mostra alguns estágios da codificação pelo algoritmo proposto neste trabalho. Pode-se observar nessa figura a seqüência CCCCcR para as primeiras 6 faces da malha, incluindo a face inicial identificada como a do triângulo cinza.

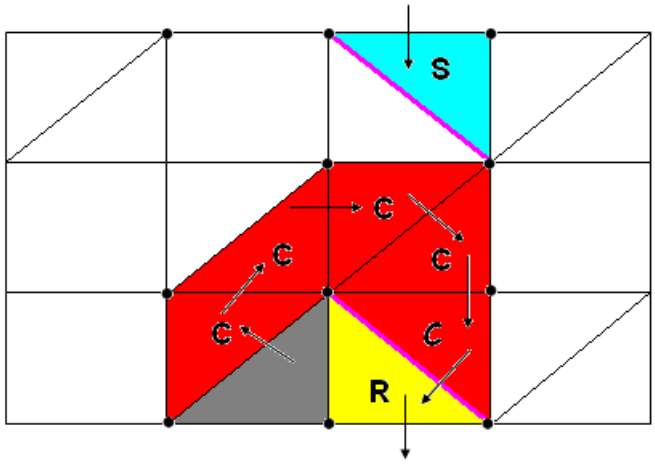


Figura 4.14: Alguns estágios da compressão de um toro.

A Figura 4.15 mostra os *labels* de todos os triângulos e quadrângulos do torus definido pelo algoritmo de compressão. No final do algoritmo, a seqüência gravada no arquivo *Fclers* é: CCCCcRsLcRCCcRsLEsLRLRE.

Neste exemplo a primeira e a terceira face classificada como S geram alças. As arestas alças estão desenhadas em marrom.

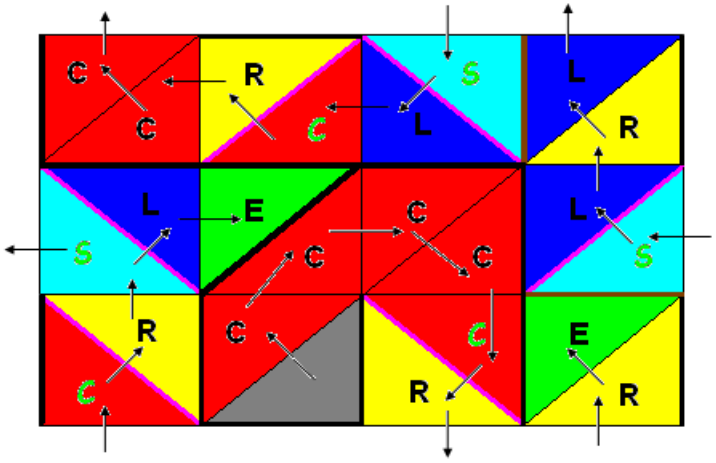


Figura 4.15: Seqüência de CLERS na compressão de um toro com malha irregular.

As figuras 4.16, 4.17, e 4.18, ilustram o resultado do algoritmo de compressão para o toro da Figura 5.12. Correspondem, respectivamente, aos arquivos *Fgeometry*, *Fhandles* e *Fclers*.

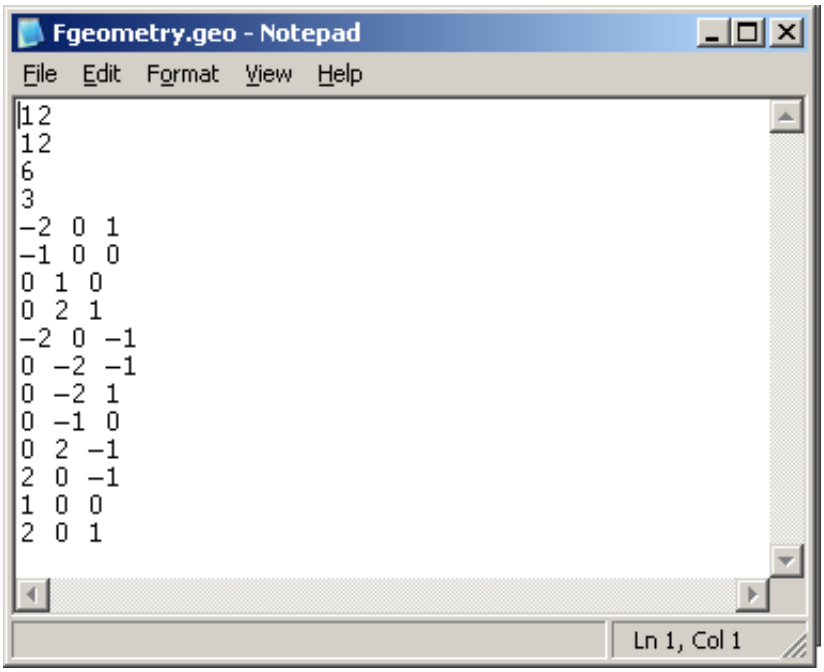


Figura 4.16: Estrutura do arquivo de saída Fgeometry.

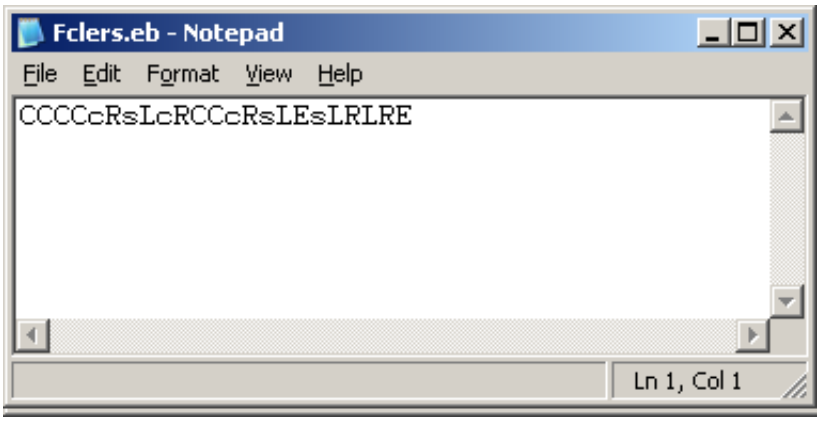


Figura 4.17: Estrutura do arquivo de saída Fclers.

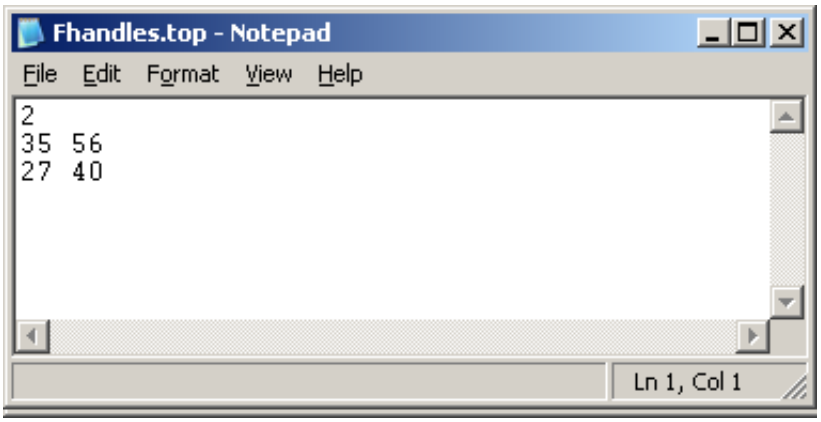


Figura 4.18: Estrutura do arquivo de saída Fhandles.

4.5.3

Análise do algoritmo

Primeiramente, é possível observar que esse programa termina, pois ele nada mais é que uma busca em profundidade para criar uma árvore geradora no grafo dual da superfície, onde os nós do grafo são as faces e as linhas do grafo indicam a relação de adjacência entre as faces. A complexidade de tempo do algoritmo é, portanto, linear no número de faces.

O algoritmo visita todas as faces classificando-as. Os testes forçam que cada face só seja visitada uma única vez. Essa árvore geradora no grafo dual possui $NT + NQ - 1$ linhas, que correspondem às arestas que foram portas de passagem de uma face para outra.

A cada face classificada como c ou C visita-se um novo vértice. Como todos os vértices necessariamente vão ser visitados, o número total de c ou C é igual a NV menos o número de vértices da face inicial que pode

ser 3 ou 4. Assim implicitamente, o algoritmo também cria uma árvore geradora no grafo primal da superfície, onde os nós do grafo são os vértices da superfície e as linhas do grafo são as arestas da superfície. As linhas dessa árvore geradora do grafo dual correspondem às arestas esquerdas dos triângulos do tipo C ou c, e todas as arestas da face inicial menos a porta. O número de arestas da árvore geradora do grafo primal é, por definição, $NV - 1$(Figura 4.19).

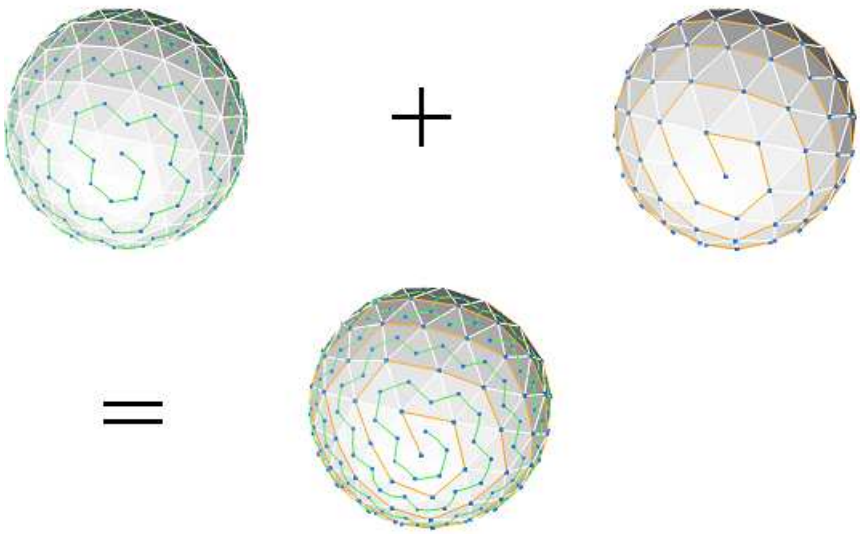


Figura 4.19: Compressão no grafo dual e primal.

As arestas codificadas implicitamente pelo algoritmo até então são: as arestas portas do grafo dual e as arestas da árvore geradora do grafo primal. O número total dessas arestas é: $(NT + NQ - 1) + (NV - 1) = NV + NT + NQ - 2$. A superfície a fórmula de Euler-Poincaré é:

$$\chi(\mathcal{S}) = NV - NE + (NT + NQ) = 2 - 2g.$$

Daí, pode-se escrever o número total de arestas da superfície como:

$$NE = NV + NT + NQ - 2 + 2g.$$

Assim, o número de arestas que ainda não foram implicitamente codificadas somam exatamente $2g$.

4.6
Conclusão

Neste Capítulo foi apresentada uma extensão do algoritmo *Edgebreaker* para compressão de superfícies compostas por triângulos e/ou quadrângulos com ou sem gênus. Uma implementação simples desse algoritmo foi descrita utilizando a estrutura de dados *CHalfEdge*, que foi introduzida no Capítulo 3.

No próximo Capítulo será apresentado o algoritmo para descomprimir a superfície codificada pelo algoritmo aqui proposto, através da leitura dos arquivos: $Fgeometry, Fhandles, Fclers$.