Pontifícia Universidade Católica do Rio de Janeiro

Jéferson Rômulo Pereira Coêlho

Uma Solução Eficiente para Subdivisão de Malhas Triangulares

Dissertação de Mestrado

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática da PUC–Rio

Orientador: Prof. Marcelo Gattass

Rio de Janeiro
Março de 2013

PUC-Rio - Certificação Digital Nº 1112650/CA

Pontifícia Universidade Católica do Rio de Janeiro

Jéferson Rômulo Pereira Coêlho

Uma Solução Eficiente para Subdivisão de Malhas Triangulares

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós–graduação em Informática do Departamento de Informática do Centro Técnico Científico da PUC–Rio. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Marcelo Gattass
Orientador
Departamento de Informática — PUC–Rio

Prof. Hélio Côrtes Vieira Lopes
Departamento de Informática — PUC–Rio

Prof. Luiz Fernando Martha
Departamento de Engenharia Civil — PUC–Rio

Dr. Pedro Mário Cruz e Silva
Instituto Tecgraf/PUC–Rio

Prof. Waldemar Celes
Departamento de Informática — PUC–Rio

Prof. José Eugenio Leal
Coordenador Setorial do Centro Técnico Científico — PUC–Rio

Rio de Janeiro, 26 de Março de 2013

PUC-Rio - Certificação Digital Nº 1112650/CA

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador.

Jéferson Rômulo Pereira Coêlho

Graduou-se em Bacharel em Ciência da Computação pela Universidade Federal de Viçosa em janeiro de 2011.

PUC-Rio - Certificação Digital Nº 1112650/CA

Ficha Catalográfica

Coêlho, Jéferson Rômulo Pereira

Uma Solução Eficiente para Subdivisão de Malhas Triangulares / Jéferson Rômulo Pereira Coêlho; orientador: Prof. Marcelo Gattass. — Rio de Janeiro : PUC–Rio, Departamento de Informática, 2013.

v., 92 f: il. ; 29,7 cm

1. Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui referências bibliográficas.

1. Informática – Tese. 2. Subdivisão de Malhas Triangulares. 3. Estruturas de Dados Topológicas. 4. Subdivisão Global. 5. Subdivisão Local. I. Gattass, Marcelo. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Agradecimentos

A Deus por ter me dado força para suportar e superar todas as dificuldades.

Ao meu orientador, professor Marcelo Gattass, pelo apoio, atenção, incentivo e amizade durante toda a realização deste trabalho.

À minha familia, em especial aos meus pais Romilda Santos e Hélio Gilson os quais privei da minha companhia para me dedicar a este trabalho.

À Juliette Zanetti pelo companherismo e pelo apoio.

À todos os meus colegas do v3o2, em especial ao André Campos pelas ideias sempre interessantes e paciência.

À CAPES, a PUC-Rio e ao TECGRAF, pelo auxilios concedidos, sem os quais este trabalho não poderia ser realizado.

PUC-Rio - Certificação Digital Nº 1112650/CA

Resumo

Coêlho, Jéferson Rômulo Pereira; Gattass, Marcelo. **Uma Solução Eficiente para Subdivisão de Malhas Triangulares**. Rio de Janeiro, 2013. 92p. Dissertação de Mestrado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Subdivisão de superfícies triangulares é um problema importante nas atividades de modelagem e animação. Ao deformar uma superfície a qualidade da triangulação pode ser bastante prejudicada na medida em que triângulos, antes equiláteros, se tornam alongados. Uma solução para este problema consiste em refinar a região deformada. As técnicas de refinamento requerem uma estrutura de dados topológica que seja eficiente em termos de memória e tempo de consulta, além de serem facilmente armazenadas em memória secundária. Esta dissertação propõe um *framework* baseado na estrutura *Corner Table* com suporte para subdivisão de malhas triangulares. O *framework* proposto foi implementado numa biblioteca C++ de forma a dar suporte a um conjunto de testes que comprovam a eficiência pretendida.

Palavras–chave

Subdivisão de Malhas Triangulares; Estruturas de Dados Topológicas; Subdivisão Global; Subdivisão Local;

PUC-Rio - Certificação Digital Nº 1112650/CA

Abstract

Coêlho, Jéferson Rômulo Pereira; Gattass, Marcelo (advisor). **An efficient solution for Triangular Mesh Subdivision**. Rio de Janeiro, 2013. 92p. MSc Dissertation — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Subdivision of triangular surfaces is an important problem in modeling and animation activities. Deforming a surface can be greatly affected the quality of the triangulation when as equilateral triangles become elongated. One solution to this problem is to refine the deformed region. Refinement techniques require the support of topological data structure. These structures must be efficient in terms of memory and time. An additional requirement is that these structures must also be easily stored in secondary memory. This dissertation proposes a framework based on the Corner Table data structure with support for subdivision of triangular meshes. The proposed framework was implemented in a C ++ library. With this library this work presents a set of test results that demonstrate the desired efficiency.

Keywords

Subdivision of triangle meshes; Topological Data Structures; Global Subdivision; Local Subdivision;

PUC-Rio - Certificação Digital Nº 1112650/CA

Sumário

Lista de figuras **8**

Lista de tabelas **11**

Lista de Algoritmos **12**

1 Introdução **14**
1.1 Motivação 14
1.2 Contribuições desta dissertação 15
1.3 Estrutura da Dissertação 15

2 Métodos de subdivisão **17**
2.1 Subdivisão Global 17
2.2 Subdivisão Adaptativa 23

3 Estruturas de Dados Topológicas **30**
3.1 Corner Table 30
3.2 CHE - Compact Half-Edge 36
3.3 Corner Table vs CHE 41

4 Operadores na Corner Table **42**
4.1 Operadores Estelares para Malhas Triangulares 42
4.2 Edge Flip 43
4.3 Edge Flip Inverso 45
4.4 Edge Split 46
4.5 Edge Weld 48

5 Subdivisão em Corner Table **51**
5.1 Composição de operações 51
5.2 Subdivisão Global 53
5.3 Subdivisão Local 58
5.4 Undo e Redo para Subdivisão 68

6 Resultados **71**
6.1 Subdivisão Global 71
6.2 Subdivisão adaptativa 80

7 Conclusão e Trabalhos Futuros **89**

Referências Bibliográficas **90**

Referências Bibliográficas **90**

PUC-Rio - Certificação Digital Nº 1112650/CA

Lista de figuras

1.1 Interpretação de superfícies a partir de um plano de corte 15

2.1 Subdivisão *1 para 4* 18
2.2 Máscara para posicionamento dos vértices pares e ímpares em *Loop*. 19
2.3 Três níveis de subdivisão *Loop*. As arestas em destaque mostram o refinamento *1 para 4*. Fonte: (Pakdel and Samavati, 2007) 20
2.4 Máscara para posicionamentos dos novos vértices para o caso de malhas regulares em *Butterfly*. 20
2.5 Máscara para posicionamentos dos novos vértices para o caso de malhas regulares em *Butterfly* considerando $w = 0$. 21
2.6 Máscara para posicionamento dos vértices ímpares para o caso em que a valência de $\mathbf{v}^i$ é três. 22
2.7 Máscara para posicionamento dos vértices ímpares para o caso em que a valência de $\mathbf{v}^i$ é quatro. 22
2.8 Máscara para posicionamento dos vértices ímpares para valências de $\mathbf{v}^i$ diferentes de três e quatro. 22
2.9 Suavização usando *Butterfly* modificado. Fonte: (Zorin *et al*, 1996) 22
2.10 Fissura gerada pela simples subdivisão de uma face e corrigida por uma triangulação simples. 23
2.11 Uso de triangulação simples para remover fissuras. A partir do segundo passo de subdivisão os vértices não são reposicionados de forma correta 25
2.12 Triangulação usando *red-green* e malha restrita. Fonte: (Pakdel and Samavati, 2007) 26
2.13 Expansão $E^1(S)$, $E^2(S)$ e $E^3(S)$ dos vértices selecionados. Os números mostram a distância do vértice à área selecionada 27
2.14 Dois passos de subdivisão aplicados sobre a mesma região com expansão $E^1(S)$. As fissuras foram resolvidas usando a triangulação simples, ou seja, conectando os vértices T aos vértices O. A área em destaque identifica a região que deve ser dividida a cada passo e os vértices em vermelho pertencem a S'. 28
2.15 Exemplo da aplicação da subdivisão incremental em uma malha de controle usando *Loop* e *Butterfly*. Fonte: (Pakdel and Samavati, 2007) 29

3.1 Relação entre *corner* e *half-edge*. Adaptado de: (Ferreira, 2006) 31
3.2 Representação de uma malha de triângulos usando *Corner Table*. Adaptado de (Vieira, 2003) 32

PUC-Rio - Certificação Digital Nº 1112650/CA

3.3 *Corners* $n(c)$, $p(c)$, $l(c)$ e $r(c)$ que resultam das operações de *next*, *previous*, *left* e *right*, respectivamente. O índice do triângulo ao qual pertence o *corner* c, é denotado por $t(c)$. Adaptado de (Rossignac *et al*, 2001) e (Vieira, 2003) 32
3.4 Representação incluindo a tabela C 33
3.5 A estrela de um vértice e a representação implícita de arestas 34
3.6 Vizinhança de uma aresta 36
3.7 Nível 0 da CHE: Sopa de triângulos. Adaptado de (Ferreira, 2006) 37
3.8 Relação entre vértice e *half-edge*. Adaptado de (Ferreira, 2006) 38
3.9 Nível 1 da CHE: Adjacência entre os triângulos. Adaptado de (Ferreira, 2006) 38
3.10 *Half-edges* opostas. Adaptado de (Ferreira, 2006) 38
3.11 Escolha das *half-edges* para serem armazenadas na tabela VH. Adaptado de (Ferreira, 2006) 39
3.12 Exemplo de uma malha representada com a CHE 40

4.1 Operações estelares de subdivisão. Fonte: (Velho, 2003) 42
4.2 *Edge Flip* como uma composição de *edge split* + *edge weld* 43
4.3 *Edge Flip* na Corner Table 44
4.4 A aplicação do *Edge Flip* faz com que as faces girem no sentido anti-horário 45
4.5 *Edge Flip* inverso 45
4.6 *Edge Split* na Corner Table 47
4.7 Convenção para posicionamento das faces após a realização da operação de *Edge Split* 47
4.8 *Edge Weld* na Corner Table 49

5.1 Subdivisão *1 para 4* 51
5.2 Subdivisão *1 para 4* como uma sequência de 3*Edge Split* + *Edge Flip* 52
5.3 Subdivisão *1 para 4* subdividindo faces adjacentes 52
5.4 a) Malha inicial a ser subdividida. b-e) Os pares de *cornes* (v, e) são retirados da fila e as faces em suas estrelas que ainda não foram processadas são subdivididas. Ao processar um vértice, as faces de sua estrela são marcadas como processadas e as faces opostas são marcadas como visitadas. Após subividir toda estrela do vértice, o operador de *Edge Flip* é aplicado sobre a aresta oposta ao *corner* e. f) Malha refinada. 54

PUC-Rio - Certificação Digital Nº 1112650/CA

5.5 a) Malha inicial. b) O operador de *Edge Split* é inicialmente aplicado a partir do *corner* c. O par (v, e) é colocado na fila Q. c) Ao subdvidir as faces na estrela do vértice $V[v]$, o operador de *Edge Split* é aplicado a partir do *corner* t e o par (e, g) é colocado na fila Q. d) Após processar o vértice $V[v]$, o operador de *Edge Flip* é aplicado a patir do *corner* e. Apenas o corner $t = prev(e)$ associado ao vértice que já foi processado muda de vértice, logo os outros *corners* da face que podem já está na fila associados com os seus vértices continuam associados com os mesmos vértices. 57
5.6 Exemplo da aplicação do algoritmo de subdivisão em um triângulo usando *Corner Table*. Sequência de aplicação dos operadores: *EdgeSplit(* 0 *)*, *EdgeSplit(* 1 *)*, *EdgeSplit(* 5 *)* e *EdgeFlip(* 2 *)* 58
5.7 Ilustração do funcionamento do Algoritmo 18. As faces em azul são aquelas que serão subdivididas posteriormente. 65
5.8 Inversão do operador de *Edge Split* pela aplicação do operador de *Edge Weld* e vice-versa. 69
5.9 Invrsão do operador de *Edge Flip* pela aplicação do *Edge Flip* Inverso e vice-versa. 69

6.1 Geração da malha de uma esfera através da subdivisão de um icosaedro regular 73
6.2 Subdivisão de uma malha com dois triângulos para aproximar a função $z = sin(\pi x)cosy + 0.1sin(2\pi x)$ 74
6.3 Subdivisão de uma um tetraedro regular usando *Butterfly* 75
6.4 Subdivisão de uma estrela de 6 pontas usando Butterfly 76
6.5 Subdivisão do modelo cow usando Butterfly. 77
6.6 Quatro níveis de subdivisão do modelo bunny usando Butterfly. 78
6.7 Subdivisão do modelo bimba usando Butterfly. O triângulo muito fino na em 6.7(a) provoca um efeito indesejado na subdivisão 79
6.8 Vários níveis da subdivisão incremental aplicada sobre um icosaedro regular para geração da malha de uma esfera. 83
6.9 Subdivisão adaptativa de uma malha inicial com 8 triângulos para aproximar a função $z = sin(\pi x)cosy + 0.1sin(2\pi x)$. A cada passo a região onde o ponto médio de uma aresta dista mais que 0.005 da posição onde deveria está é selecionada para subdivisão 84
6.10 Vários níveis da subdivisão incremental aplicada sobre um icosaedro regular para geração da malha de uma esfera. 85
6.11 Subdivisão incremental do modelo *cow* usando *Butterfly* para posicionar os novos vértices criados. Os pontos em destaque pertecem ao conjunto S', ou seja, o pontos a partir de onde o algoritmo faz a expansão. 86
6.12 Subdivisão incremental do modelo *bunny* usando *Butterfly* para posicionar os novos vértices criados. 87
6.13 Subdivisão incremental do modelo *bimba* usando *Butterfly* para posicionar os novos vértices criados. 88

PUC-Rio - Certificação Digital Nº 1112650/CA

Lista de tabelas

5.1 Operações Inversas 70

6.1 Tempo para realizar a subdivisão global. *Core i5 3.10Ghz, 8GB RAM 71
6.2 Dados da realização da subdivisão adaptativa com $r = 1$. *Core i5 3.10Ghz, 8GB RAM 80
6.3 Dados da realização da subdivisão adaptativa com $r = 1$. *Core i5 3.10Ghz, 8GB RAM 80
6.4 Dados da realização da subdivisão adaptativa com $r = 5$. *Core i5 3.10Ghz, 8GB RAM 81
6.5 Dados da realização da subdivisão adaptativa com $r = 5$. *Core i5 3.10Ghz, 8GB RAM 81

PUC-Rio - Certificação Digital Nº 1112650/CA

Lista de Algoritmos

1	edge(c)	34
2	VizinhançaDeUmvértice(v). Fonte: (Vieira, 2003)	35
3	VizinhançaDeUmaAresta(e). Fonte: (Vieira, 2003)	35
4	VizinhançaDeUmaFace(f). Fonte: (Vieira, 2003)	36
5	*Percorrimento das curvas de bordo( ). Fonte: (Ferreira, 2006)*	40
6	EdgeFlip(c_0). Adaptado de (Vieira, 2003)	44
7	EdgeFlipInverso(c_1)	46
8	EdgeSplit(c_0)	48
9	EdgeWeld(c_1). Adaptado de (Vieira, 2003)	49
10	SubdivisaoGlobal	55
11	runSplit	56
12	SubdivisaoIncremental	58
13	obtemCornersDosVertices	59
14	obtemCornersDosVertices	60
15	obtemCornersNaVizinhancaR	61
16	obtemVerticesVizinhos	62
17	expandeVertice	63
18	obtemFacesFormadasPorErS	64
19	runSplit	66
20	SubdivideFaces	67

PUC-Rio - Certificação Digital Nº 1112650/CA

PUC-Rio - Certificação Digital Nº 1112650/CA

Quando se culpa os outros, renuncia-se à capacidade de mudar.

Douglas Adams, .

1
Introdução

Métodos de subdivisão são cada vez mais utilizados em pacotes de animação e modelagem computacional, substituindo ferramantes tradicionais de modelagem, como *NURBS*. As operações de subdivisão são eficientes e permitem a construção de superfícies que tem suavidade variável, além ferramentas de modelagem intuitivas que podem ser aplicadas em uma malha de topologia arbitrária. No entanto, em muitos casos, a subdivisão é realizada tendo como suporte uma estrutura de dados topológica que consome um grande volume de memória, como uma *Half-Edge*, tornando difícil a sua aplicação em modelos massivos.

1.1
Motivação

A motivação para realizar subdivisão local em *Corner Table* surgiu do problema de iterativamente ajustar superfícies a dados sísmicos. Seguindo uma prática comum no processo de interpretação sísmica, a superfície, que geralmente representa um horizonte, uma falha ou um sal, é posicionada no volume a partir de uma manipulação direta de suas curvas de interseção com um plano de corte qualquer (ver Figura 1.1).

À medida que o usuário vai deformando a superfície, a malha triangular inicial precisa localmente ser simplificada ou refinada para se adaptar à geometria e manter a qualidade dos triângulos.

O refinamento e a simplificação podem ser baseados na curvatura da superfície ou mesmo na área dos triângulos, ou seja, se um triângulo atingir uma determinada área limite ele é refinado ou se sua área se tornar muito pequena ele é simplificado.

Uma vez que dados sísmicos usam grandes volumes de memória, é desejável que a interpretação de superfícies faça uso eficiente da memória. Por este motivo, a estrutura de dados topológica escolhida para representação da superfície foi a *Corner Table*. Esta estrutura é simples e usa apenas um vetor de inteiros da dimensão da lista de triângulos para representar a topologia da superfície.

PUC-Rio - Certificação Digital Nº 1112650/CA

Por se tratar de uma aplicação de modelagem interativa, também é desejável fornecer algum mecanismo de *undo/redo* para que o usuário possa desfazer ou refazer uma sequência de passos durante o processo.

(Vieira *et al*, 2004) propuseram a simplificação de malhas triangulares baseadas em *Corner Table*, o autor entretando não encontrou na literatura trabalhos com subdivisão e suporte para *undo/redo* em *Corner Table*. Desta forma, o objetivo deste trabalho é realizar subdivisão global e local de forma eficiente usando a *Corner Table* como estrutura de dados topológica de suporte, além de fornecer uma forma para implementação do mecanismo de *undo/redo* que preserve a relação biunivica entre a triangulação e a estrutura de dados que a implementa, ou seja, ao executar uma operação de *undo*, por exemplo, a estrutura de dados volta à mesma configuração que possuía anteriormente..

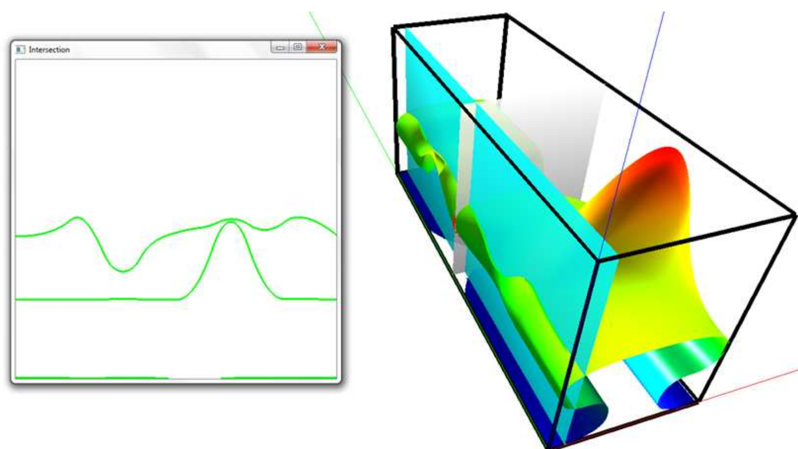


Figura 1.1: Interpretação de superfícies a partir de um plano de corte

PUC-Rio - Certificação Digital Nº 1112650/CA

1.2 Contribuições desta dissertação

Como contribuição desta dissertação destacam-se:

1. Nova implementação dos operadores estelares inversíveis.

2. Dois algoritmos para realizar subdivisão global e local para malhas triangulares em *Corner Table*

1.3 Estrutura da Dissertação

Esta dissertação está dividida da seguinte forma: no Capítulo 2 apresentamos os esquemas de subdivisão global e local que são utilizados neste trabalho, no Capítulo 3 apresentamos duas estruturas de dados topológicas: a *Corner Table* e a *Compact Half-Edge*. Ambas são estruturas que utilizam um pequeno volume para

representação de superfícies triangulares e poderiam ser utilizadas neste trabalho. Concluímos este Capítulo com a escolha da *Corner Table* por ser mais simples. No Capítulo 4 apresentamos uma nova implementação em *Corner Table* para um conjunto de operadores estelares inversíveis que servem de base para realizar a subdivisão e as suas operações inversas. No Capítulo 5 propomos dois novos algoritmos para realizar subdivisão global e local e o mecanismo de *undo/redo* baseados nestes operadores. No Capítulo 6 apresentamos os resultados e obtidos, e por fim, no capítulo 7, apresentamos algumas conclusões e propomos trabalhos futuros.

PUC-Rio - Certificação Digital Nº 1112650/CA

2
Métodos de subdivisão

Métodos de subdivisão de superfícies são cada vez mais usados em pacotes de animação e modelagem computacional (Pakdel and Samavati, 2007). A subdivisão é definida por operações simples, de refinamento e de médias, que são uniformemente aplicados em uma dada malha de controle. Repetidas aplicações destas operações produzem uma sequência de malhas que convergem para uma superfície que é suave em toda sua extensão (Cashman, 2012). Algumas regras especiais de subdivisão possibilitam a construção de superfícies que variam a sua suavidade. Além disso, as operações de subdivisão são eficientes e permitem a construção de ferramentas de modelagem intuitivas e podem ser aplicadas em malhas de polígonos com topologia arbitrária. Consequentemente, subdivisão tem substituido ferramentas de modelagem tradicionais, tais como *NURBS*, em aplicações de modelagem e sólidos de forma livre (DeRose *et al*, 1998).

Iniciando com uma malha de polígonos de entrada, cada passo da subdivisão refina suas faces e reposiona seus vértices através de combinações afim dos vértices próximos, produzindo uma superfície suave no limite (Zorin *et al*, 1996). À medida que repetidos passos de subdivisão são aplicados, o número de faces aumenta exponencialmente, o que pode aumentar a carga de computação rapidamente (Pakdel and Samavati, 2007). No entanto, em muitas aplicações não é necessário, e muitas vezes nem desejável, que toda a malha seja refinada. Em visualização de modelos 3D, por exemplo, apenas as regiões visíveis da malha devem ser detalhadas, ou em outro caso, projetistas muitas vezes precisam focar apenas na área em que estão trabalhando.

Desta forma, as operações de subdivisão podem ser classificadas em dois tipos: subdivisão global e subdivisão adaptativa. No caso da subdivisão global, todas as faces da malha são subdivididas, ja na subdivisão adaptativa, apenas um conjunto das faces é subdividido.

2.1
Subdivisão Global

Muitos métodos de subdivisão de superfícies foram desenvolvidos nas últimas décadas, como *Doo-Sabin* (Doo and Sabin, 1978), *Catmull-Clark*

PUC-Rio - Certificação Digital Nº 1112650/CA

(Catmull and Clark, 1998), *Loop* (Loop, 1987) e *Butterfly* (Dyn *et al*, 1990). Nós focaremos nos métodos de subdivisão *Loop* e *Butterfly*, pois, como citado em (Pakdel and Samavati, 2007), estes esquemas operam em malhas de triângulos, e são comumente utilizados em modelagem, renderização e sistemas de animação. Esses dois algoritmos compartilham a mesma operação de refinamento, mas os operadores de média, que definem as posições dos novos vértices, são diferentes. A Figura 2.1 mostra a operação de refinamento nos esquemas *Loop* e *Butterfly*. Em cada passo de subdivisão, cada face é dividida em quatro novas faces.

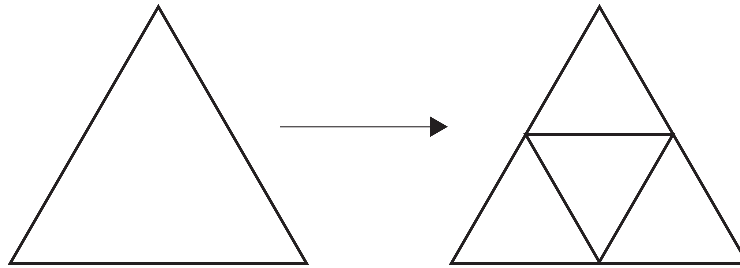


Figura 2.1: Subdivisão *1 para 4*

PUC-Rio - Certificação Digital Nº 1112650/CA

Esta operação de refinamento tem duas características desejáveis: não altera a valência dos vértices pré-existentes na malha e todos os novos vértices criados tem valência 6, com exceção daqueles que pertencem ao bordo da superfície.

Como destaca (Pakdel and Samavati, 2007), os esquema de subdivisão *Loop* produz uma superfície que é uma aproximação da malha de controle, uma vez que ele reposiciona os vértices da malha, já a superfície criada pelo esquema de subdivisão *Butterfly* interpola a malha de controle.

2.1.1 Loop

Começando com uma malha M^0, a malha M^{i+1} é obtida através da divisão de todas as faces da malha M^i e reposicionando seus vértices resultantes. A malha M^i tem um nível de subdivisão i, além disso a resolução da malha e seu nível de subdivisão são diretamente relacionados, ou seja, a medida que o nível de subdivisão aumenta, as faces se tornam menores e a resolução da malha aumenta (Pakdel and Samavati, 2007).

No esquema de subdivisão *Loop*, proposto por (Loop, 1987), os operadores geométricos que definem as posições dos novos vértices são representados através de máscaras. Os vértices $\mathbf{v}^i$ da malha M^i são reposicionados como uma combinação linear de sua posição com a dos seus vizinhos $\mathbf{v}^i_j$ de acordo com a equação (2-1).

$$\mathbf{v}^{i+1} = \beta\mathbf{v}^i + \alpha\sum_{j=0}^{n-1}\mathbf{v}^i_j \tag{2-1}$$

onde n é a valência do vértice $\mathbf{v}^i$,

$$\alpha = \frac{1}{n}\left(\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\cos(\frac{2\pi}{n})\right)^2\right) \tag{2-2}$$

$$\beta = 1 - n\alpha \tag{2-3}$$

Os vértices $\mathbf{v}^{i+1}$ são chamados de vértices pares da malha M^{i+1}. A operação de divisão introduz vértices $\mathbf{e}^{i+1}$ sobre as arestas denominados vértices ímpares. Os vértices ímpares são calculados como na equação (2-4)

$$\mathbf{e}^{i+1} = \frac{3}{8}\mathbf{a} + \frac{3}{8}\mathbf{b} + \frac{1}{8}\mathbf{c} + \frac{1}{8}\mathbf{d} \tag{2-4}$$

onde **a**, **b**, **c** e **d** são definidos como na Figura 2.2.

De acordo com (Pakdel and Samavati, 2007) a correta aplicação das equações (2-1) e (2-4) requerem que todos os vértices da malha M^i esteja no mesmo nível de subdivisão. Isto é particularmente importante para subdivisão adaptativa.

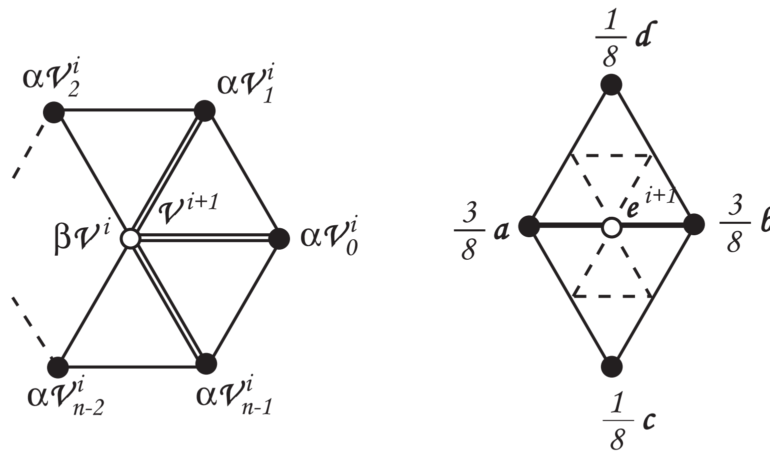


Figura 2.2: Máscara para posicionamento dos vértices pares e ímpares em *Loop*.

Em uma malha de triângulos, o esquema de subdivisão *Loop* tem continuidade paramétrica C^2 em todos os vértices de valência 6, ditos vértices ordinários. Nos vértices extraordinários, aqueles os quais tem valência diferente de 6, tem continuidade geométrica G^1 (Loop, 1987). Como o *Loop* usa a operação de refinamento *1 para 4*, os vértices ímpares são sempre ordinários enquanto os vértices pares mantém a sua valência.

2.1.2 Butterfly

O esquema de subdivisão *Butterfly* foi primeiro proposto por (Dyn *et al*, 1990). Ele é um esquema de interpolação, uma vez que não reposiciona os vértices da malha durante a subdivisão. Em suas primeiras versões, *Butterfly* era aplicável apenas em casos regulares, ou seja, casos em que os vértices eram ordinários. No caso regular, com os vértices a e b ordinários, de acordo com uma

PUC-Rio - Certificação Digital Nº 1112650/CA

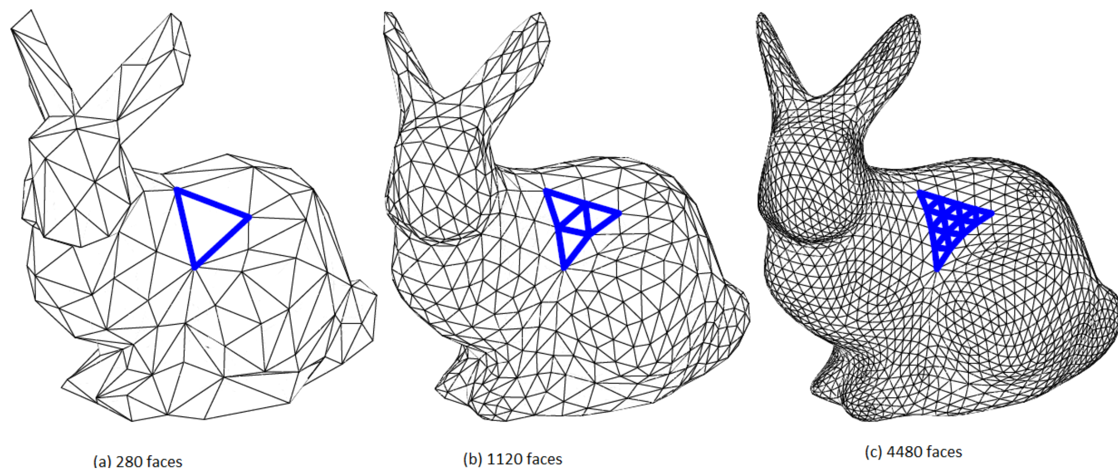


Figura 2.3: Três níveis de subdivisão *Loop*. As arestas em destaque mostram o refinamento *1 para 4*. Fonte: (Pakdel and Samavati, 2007)

extensão do esquema *Butterfly* apresentada em (Dyn *et al*, 1993) a posição do novo vértice ímpar $\mathbf{e}^{i+1}$ sobre a aresta **ab** é dada por

$$\mathbf{e}^{i+1} = (\frac{1}{2}-w)(\mathbf{a}+\mathbf{b})+(\frac{1}{8}+2w)(\mathbf{c}+\mathbf{d})-(\frac{1}{16}-w)(\mathbf{e}+\mathbf{f}+\mathbf{g}+\mathbf{h})+w(\mathbf{i}+\mathbf{j}) \quad (2\text{-}5)$$

onde **a**, **b**, **c**, **d**, **e**, **f**, **g**, **h**, **i** e **j** são definidos como na Figura 2.4 e w é um valor que deve ser escolhido de forma que seja muito pequeno. Como em (Zorin *et al*, 1996) e (Pakdel and Samavati, 2007) escolhemos $w = 0$, simplificando o cálculo da posição do novo vértice $\mathbf{e}^{i+1}$ para

$$\mathbf{e}^{i+1} = \frac{1}{2}\mathbf{a} + \frac{1}{2}\mathbf{b} + \frac{1}{8}\mathbf{c} + \frac{1}{8}\mathbf{d} - \frac{1}{16}\mathbf{e} - \frac{1}{16}\mathbf{f} - \frac{1}{16}\mathbf{g} - \frac{1}{16}\mathbf{h} \quad (2\text{-}6)$$

onde **a**, **b**, **c**, **d**, **e**, **f**, **g** e **h** são definidos como na Figura 2.5.

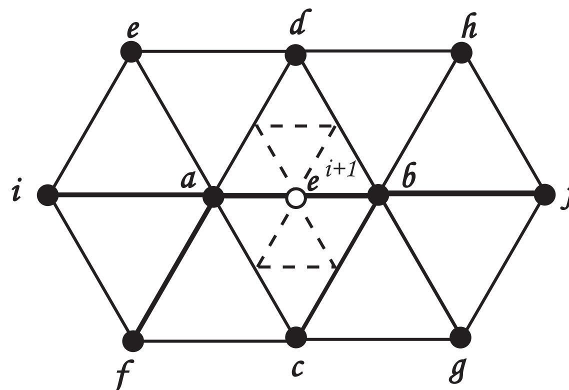


Figura 2.4: Máscara para posicionamentos dos novos vértices para o caso de malhas regulares em *Butterfly*.

Mais uma vez, para um resultado correto, todos os vértices envolvidos na computação de $\mathbf{e}^{i+1}$ devem está no mesmo nível de subdivisão.

Com o esquema original, as superfícies resultantes da subdivisão tem continuidade C^0 nos vértices extraordinários e C^1 nos demais. A fim de

PUC-Rio - Certificação Digital Nº 1112650/CA

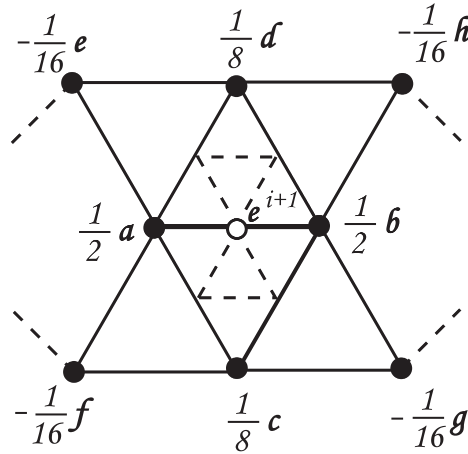


Figura 2.5: Máscara para posicionamentos dos novos vértices para o caso de malhas regulares em *Butterfly* considerando $w = 0$.

garantir continuidade C^1 também em vértices extraordinários, (Zorin *et al*, 1996) modificaram a máscara original, de forma que, a posição do vértice ímpar $\mathbf{e}^{i+1}$ na aresta $\mathbf{v}^i\mathbf{v}^i_j$ depende se um ou ambos os pontos extremos $\mathbf{v}^i$ e $\mathbf{v}^i_j$ são extraordinários. Este esquema ficou conhecido como *Butterfly* modificado.

No *Butterfly* modificado, se o vértice $\mathbf{v}^i$ é extraordinário e $\mathbf{v}^i_j$ é ordinário, a máscara de $\mathbf{e}^{i+1}$ é determinada de acordo com a valência n de $\mathbf{v}^i$. Para valência três (ver Figura 2.6):

$$\mathbf{e}^{i+1} = \frac{9}{12}\mathbf{v}^i + \frac{5}{12}\mathbf{v}^i_j - \frac{1}{12}\mathbf{v}^i_{j+1} - \frac{1}{12}\mathbf{v}^i_{j-1} \tag{2-7}$$

Para valência quatro (ver Figura 2.7):

$$\mathbf{e}^{i+1} = \frac{6}{8}\mathbf{v}^i + \frac{3}{8}\mathbf{v}^i_j - \frac{1}{8}\mathbf{v}^i_{j+2} \tag{2-8}$$

E para todas as outras valências do vértice extraordinário $\mathbf{v}^i$ (ver Figura 2.8):

$$\mathbf{e}^{i+1} = \beta\mathbf{v}^i + \sum_{j=0}^{n-1} \alpha_j \mathbf{v}^i_j \tag{2-9}$$

onde

$$\alpha_j = \frac{1}{n}\left(\frac{1}{4} + \cos\frac{2j\pi}{n} + \frac{1}{2}\cos\frac{4j\pi}{n}\right) \tag{2-10}$$

e

$$\beta = 1 - \sum_{j=0}^{n-1} \alpha_j \tag{2-11}$$

Se ambos os vértices $\mathbf{v}^i$ e $\mathbf{v}^i_j$ são extraordinários, então o processo acima é repetido para cada vértice e o resultado final é dado pela média.

PUC-Rio - Certificação Digital Nº 1112650/CA

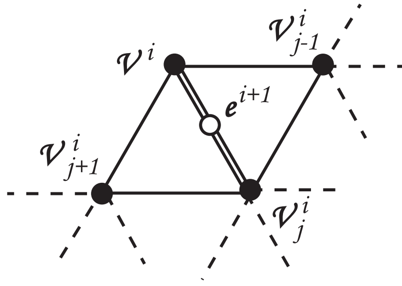


Figura 2.6: Máscara para posicionamento dos vértices ímpares para o caso em que a valência de $\mathbf{v}^i$ é três.

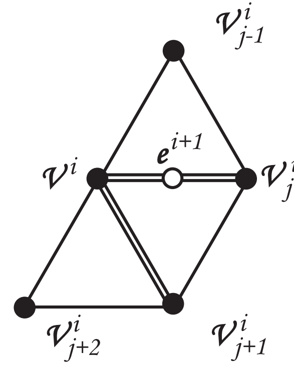


Figura 2.7: Máscara para posicionamento dos vértices ímpares para o caso em que a valência de $\mathbf{v}^i$ é quatro.

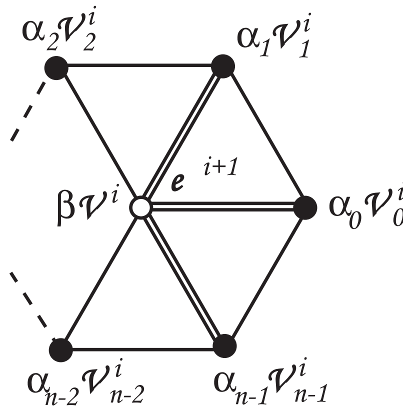


Figura 2.8: Máscara para posicionamento dos vértices ímpares para valências de $\mathbf{v}^i$ diferentes de três e quatro.

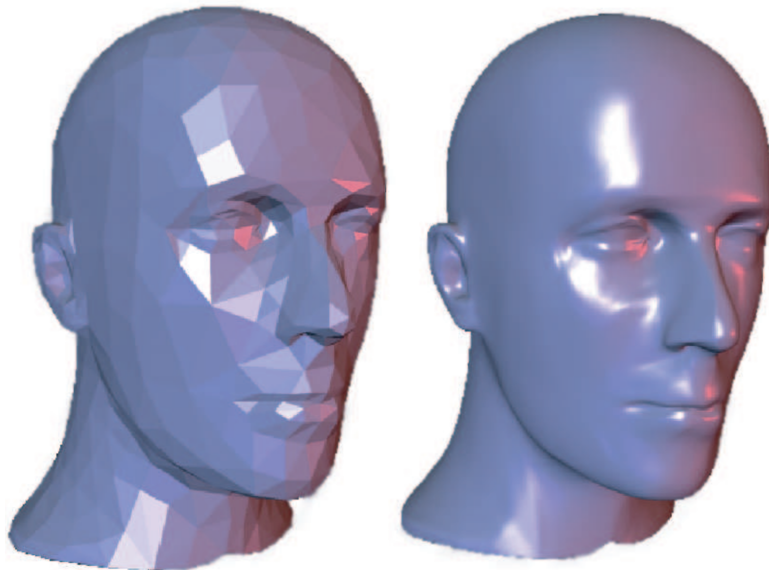


Figura 2.9: Suavização usando *Butterfly* modificado. Fonte: (Zorin *et al*, 1996)

PUC-Rio - Certificação Digital Nº 1112650/CA

PUC-Rio - Certificação Digital Nº 1112650/CA

2.2 Subdivisão Adaptativa

Como colocado por (Pakdel and Samavati, 2007), alguns esquemas de subdivisão são projetados de forma que o refinamento adaptativo é uma extensão natural do processo, como é o caso do $\sqrt{3}$-*subdivision* de (Kobblet, 2000) e o 4-8 *subdivision* de (Velho and Zorin, 2001). No entanto, pelas qualidades em relação às valências dos vértices da operação de refinamento *1 para 4* citadas na Seção 2.1, focaremos nos métodos que utilizam esse tipo de refinamento.

Na subdivisão adaptativa, apenas um subconjunto das faces da malha de controle e subdividida, no entanto, um cuidado maior deve ser tomado para lidar com a conectividade e as incossistências geométricas que podem surgir no processo. Como mostra a Figura 2.10, um simples passo para subdividir uma única face criou fissuras na malha, onde triângulos de níves de subdivisão diferentes se encontram em uma aresta. Uma malha 2-*manifold* é conforme se o resultado da interseção entre quaisquer duas faces é um vértice simples, uma aresta ou um conjunto vazio (Jones *et al*, 1997). Uma malha com fissuras não é conforme, desta forma, fissuras devem ser removidas para que a renderização, edição, animação e subdivisão da malha sejam feitas de maneira adequada (Pakdel and Samavati, 2007).

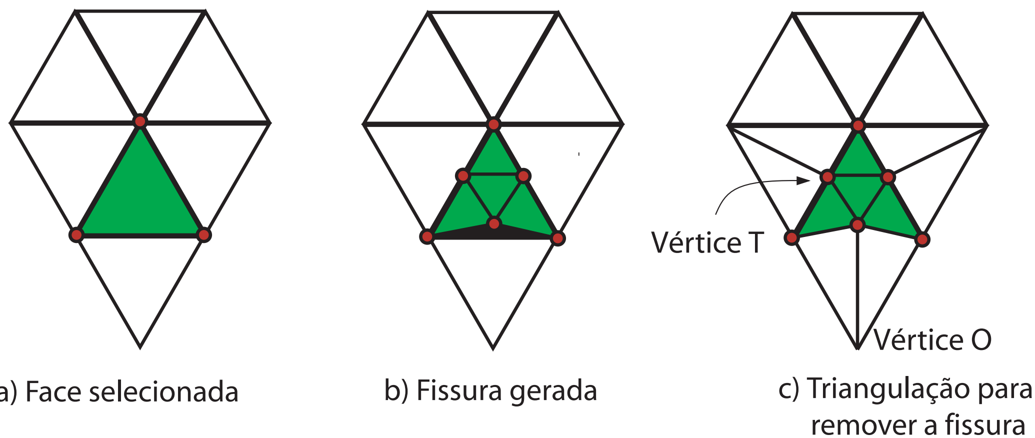


Figura 2.10: Fissura gerada pela simples subdivisão de uma face e corrigida por uma triangulação simples.

De acordo com (Pakdel and Samavati, 2007), para remover as fissuras, é comum que as faces que as contém sejam trianguladas para produzir uma malha conforme. Essa triangulação muda a conectividade dos vértices dentro da área de subdivisão selecionada. Como alguns métodos, como *Loop* e *Butterfly*, usam os vizinhos para calcular a posição de um novo vértice, a mudança na conectividade dos vértices da área selecionada para subdivisão afeta o processo de subdivisão, e conduz a uma geometria inconsistente na superfície limite. Ainda de acordo com (Pakdel and Samavati, 2007), o ideal seria que a subdivisão adaptativa produzisse, na região selecionada da malha refinada, uma superfície limite igual à superfície

gerada se toda a superfície fosse subdividida regularmente. Desta forma, todos os vértices dentro da região selecionada, deveriam ter a mesma concetividade como quando toda malha fosse subdividida.

Um outro efeito da subdivisão adaptativa é que repetidos refinamentos de uma região selecionada aumenta sua resolução exponencialmente em relação ao restante da malha. Em muitas aplicações, é desejável que a malha que foi adaptativamente subdividida seja balanceada, ou seja, que para quaisquer duas faces vizinhas, seus níveis de subdivisão não difiram em mais de uma unidade (Kobblet, 2000), de forma que o resultado seja um aumento gradual na resolução da regiões grosseiras para as regiões refinadas da malha.

Em (Pakdel and Samavati, 2007) são destacadas três propriedades que são desejáveis em uma subdivisão adaptativa:

1. conectividade consistente.
2. geometria consistente.
3. mudança gradual de resolução na superfície.

PUC-Rio - Certificação Digital Nº 1112650/CA

2.2.1 Manipulação das inconsistências

Como mencionado anteriormente, a subdivisão de um subconjunto de faces da malha gera fissuras. Estas fissuras são devido à inserção de vértices ímpares em arestas compartilhadas por faces que estão em diferentes níveis de subdivisão. Para tratar tais inconsistências (Pakdel and Samavati, 2007) listam alguns algoritmos que trataremos de forma superficial abaixo.

Triangulação simples: Para resolver o problema das fissuras, (Amresh *et al*, 2002) propuseram um método de triangulação simples para dividir as faces vizinhas em duas, três ou quatro, dependendo do número de vértices ímpares. Para cada vértice ímpar, a divisão da face de menor nível de subdivisão remove a fissura. Os vértices ímpares são chamados de vértices T e os vértices opostos que são conectados aos vértices T são chamados de vértices O (ver Figura 2.10).

O método de triângulação simples remove as fissuras da malha de forma bastante eficiente, porém, de acordo com (Pakdel and Samavati, 2007), ele apresenta alguns efeito colaterais indesejados. Primeiro, ele muda a conectividade e a valência dos vértices ímpares. Isto altera a superfície limite e ainda reduz a suavidade da superfície para G^1. Segundo, os vértices O têm um nível diferentes de subdivisão em relação aos vértices pares da área selecionada, portanto, os vértices ímpares não serão corretamente reposicionados se a área selecionada for subdividida mais de uma vez (ver Figura 2.11). Por último, repetidas subdvisões e

triangulações simples da área selecionada produzem uma alta valência dos vértices O, o que torna as faces alongadas.

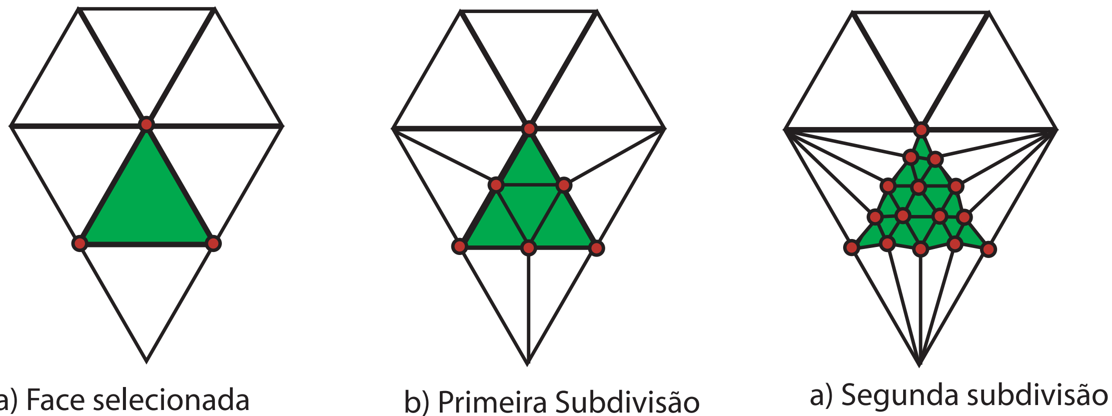


Figura 2.11: Uso de triangulação simples para remover fissuras. A partir do segundo passo de subdivisão os vértices não são reposicionados de forma correta

Triangulação *red-green*: outro método para remoção de fissuras pela inserção de vértices ímpares sobre arestas da malha é a triangulação *red-green*. Este método foi desenvolvido para permitir o refinamento adaptativo de malhas em análise de elementos finitos, onde uma malha é usada para aproximar equações em aplicações de análises numéricas.

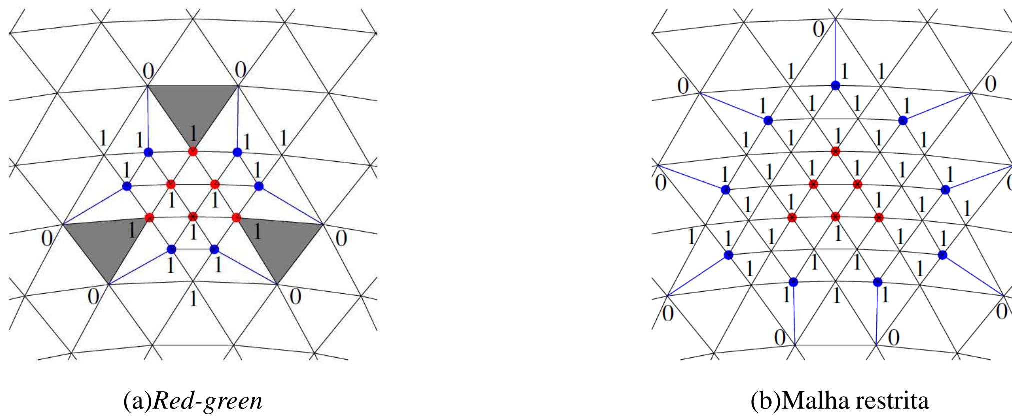
A triangulação *red-green* garante que a conectividade da área selecionada não é afetada, além de evitar as altas valências dos vértices extraordinários e garantir que o nível de subdivisão entre quaisquer duas faces nunca difira de uma unidade, produzinho assim uma malha balanceada. No entanto, os vértices ímpares ainda têm vizinhos de diferentes níveis de subdivisão. Uma descrição mais detalhada da triangulação *red-green* pode ser encontrada em (Andrew *et al*, 1983).

Malha restrita: para evitar uma mudança na superfície gerada pela subdivisão adaptativa, vértices pares e ímpares devem estar no mesmo nível de subdivisão que seus vizinhos. (Zorin *at al*, 1997) definiram essa malha de malha restrita. Para obter uma malha restrita, faces são construídas de forma que os vértices da área selecionada tenham o mesmo nível de subdivisão. Uma descrição mais detalhada do algoritmo para obtenção de uma malha restrita pode ser encontrado em (Zorin *at al*, 1997).

Um algoritmo combinado: para uma solução que abranja as três propriedades desejáveis em uma subdivisão adaptativa colocadas anteriormente, (Pakdel and Samavati, 2007) cita um algoritmo que combina a triangulação *red-green* com a malha restrita, uma vez que a triangulação *reg-green* resolve as questões de conectividade da subdivisão adaptativa e da mudança gradual entre os níveis de subdivisão e o algoritmo de malha restrita resolve a questão da computação correta dos vértices. No entanto, (Pakdel and Samavati, 2007) ressalta que este

PUC-Rio - Certificação Digital Nº 1112650/CA

algoritmo não é eficiente e nem simples, uma vez que novos vértices T podem ser gerados no processo e algoritmo deveria ser recursivamente aplicado até que não exista nenhum vértice T.



(a)*Red-green* (b)Malha restrita

Figura 2.12: Triangulação usando *red-green* e malha restrita. Fonte: (Pakdel and Samavati, 2007)

PUC-Rio - Certificação Digital Nº 1112650/CA

2.2.2 Subdivisão Incremental

A subdivisão incremental, um esquema para subdivisão adaptativa proposto por (Pakdel and Samavati, 2007), satisfaz as três propriedades desejáveis para uma subdivisão adaptativa, ou seja, garante uma geometria e conectividade consistentes e a superfície resultante muda gradualmente sua resolução das áreas mais grosseiras para as áreas mais refinadas e, além disso, é um algoritmo eficiente e simples de implementar. Este algoritmo se propõe a realizar uma subdivisão adaptativa de forma que a área selecionada para a subdivisão tenha um resultado final como se a superfície fosse subdividida de forma global, além de resolver de forma eficiente as inconsistências geradas no processo de subdivisão adaptativa. Além disso, métodos para subdivisão global como *Loop* e *Butterfly* podem ser usados junto com a subdivisão incremental para realizar a subdivisão adaptativa. Por esses motivos, a subdivisão incremental foi o esquema escolhido pelo autor como método de subdivisão adaptativa.

Abaixo segue a descrição formal da subdivisão incremental. A descrição formal a seguir contém trechos retirados de (Pakdel and Samavati, 2007), onde o esquema é definido com mais detalhes.

Descrição Formal

Tome $V = \{\mathbf{v}_0, \mathbf{v}_1, ..., \mathbf{v}_{m-1}\}$ como sendo o conjunto de vértices da malha corrente. Tome S como sendo um sub-conjunto de V. É desejável subdividir adaptativamente S de forma que a superfície limite gerada a partir de S seja

PUC-Rio - Certificação Digital Nº 1112650/CA

exatamente a mesma como quando V é subdividido. Para isso, o conjunto S é expandido para um novo conjunto, maior que S, e então esse novo conjunto é subdividido.

Em cada nível de subdivisão, S é expandido para $E^r(S)$ incluindo os vértices de V que estão dentro de um raio r, em número de arestas, de pelo menos um vértice de S

$$E^r(S) = \bigcup_{v \in S} N^r(\mathbf{v}), r > 0 \tag{2-12}$$

onde $N^r(S)$ denota os vértices na vizinhança de $\mathbf{v}$ que distam r arestas dele. Portanto, $w \in V$ está em $N^r(v)$ somente se existir uma caminho de $\mathbf{v}$ até w com no máximo r arestas (ura 2.13). Após fazer a expansão, a subdivisão (*Loop* ou *Butterfly* por exemplo) deve ser realizada em $E^r(S)$ e as fissuras podem ser removidas eficientemente usando a triangulação simples (ver Figura 2.14). Ao final, tome S' como sendo a nova área selecionada (ver Figura 2.14), que é resultado da subdivisão adaptativa realizada em S.

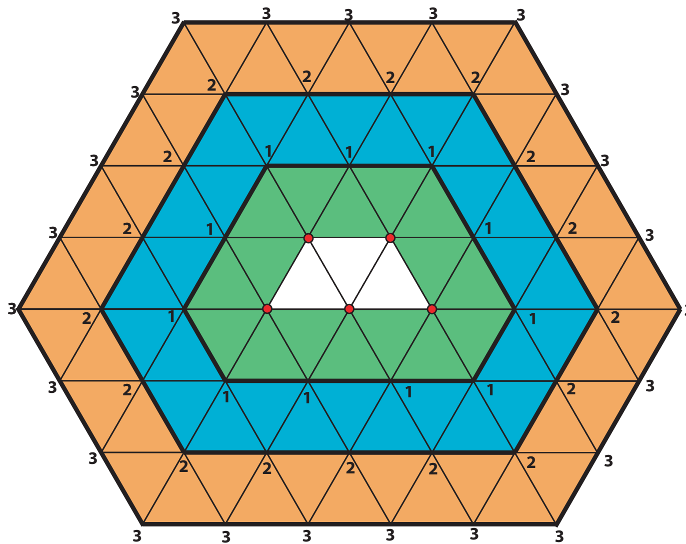


Figura 2.13: Expansão $E^1(S)$, $E^2(S)$ e $E^3(S)$ dos vértices selecionados. Os números mostram a distância do vértice à área selecionada

(Pakdel and Samavati, 2007) destacam algumas características da subdivisão incremental. Estas características são listadas abaixo:

– A subdivisão adaptativa de $E^r(S)$ produz uma superfície limite a partir de S que é exatamente a mesma como quando toda a malha é subdividida, uma vez que os vértices O e T ficam de fora do conjunto $E^r(S)$ (ver Figura 2.14), logo a conectividade dos vértices dentro de S permanece inalterada. Além disso, os vértices de S' e seus vizinhos possuem o mesmo nível de subdivisão, uma vez que $E^r(S)$ inclui todos os vértices que tem uma distância, em arestas, menor

ou igual a r a partir de S no processo de subdivisão. Com isso a superfície limite não é alterada devido à subdivisão adaptativa.

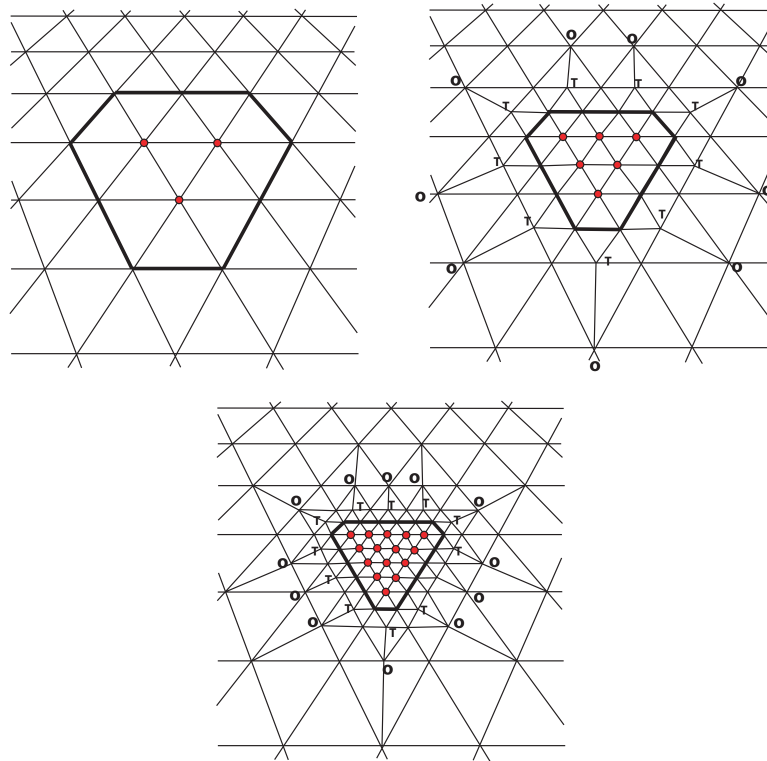


Figura 2.14: Dois passos de subdivisão aplicados sobre a mesma região com expansão $E^1(S)$. As fissuras foram resolvidas usando a triangulação simples, ou seja, conectando os vértices T aos vértices O. A área em destaque identifica a região que deve ser dividida a cada passo e os vértices em vermelho pertencem a S'.

PUC-Rio - Certificação Digital Nº 1112650/CA

- A subdivisão incremental mantém os vértices O fora da área selecionada, uma vez que os vértices ímpares criados sobre as arestas que estavam a uma distânca r do conjunto S antes da subdivisão, passam a está a uma distância $2r$ após a subdivisão. Os vértices T da subdivisão adaptativa de $E^r(S)$ estão sempre a $2r$ unidades de S' e os vértices O sempre a $3r$ unidades. Isto impede que a subdivisão incremental produza altas valências nos vértices O (ver Figura 2.14).
- Na subdivisão incremental os vértices à uma distância $3r$ de S' têm sempre um nível de subdivisão abaixo do que os vértices que estão a uma distância $2r$, isso significa que a superfície é balanceada e que a superfície resultante tem um aumento gradual no nível de subdivisão das áreas mais grosseiras para as áreas que foram incrementalmente subdivididas.
- O parâmetro r controla o grau de suavidade da transição entre os níveis de subdivisão. Um grande valor de r incluirá um grande número de vizinhos na área selecionada o que permite uma transição suave das áreas mais grosseiras para as regiões mais refinadas. O aumento da área de transição é benéfico para

os casos em que a região selecionada é subdividida muitas vezes. Na prática, poucos refinamentos são necessários para obter uma superfície suave e em muitos casos a expansão $E^1(S)$ é utilizada.

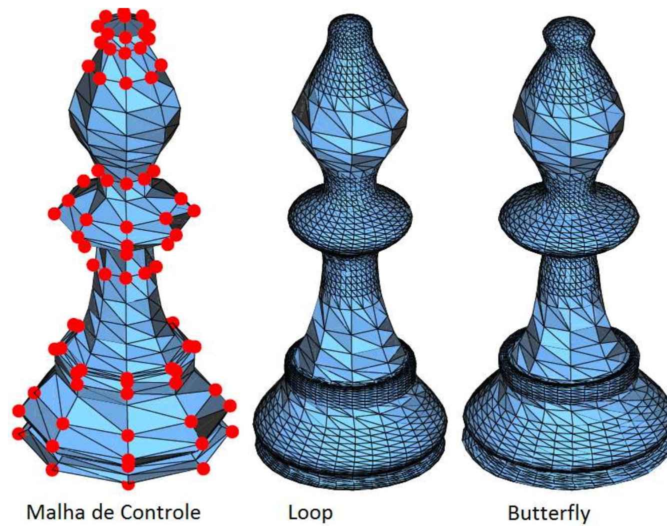


Figura 2.15: Exemplo da aplicação da subdivisão incremental em uma malha de controle usando *Loop* e *Butterfly*. Fonte: (Pakdel and Samavati, 2007)

PUC-Rio - Certificação Digital Nº 1112650/CA

3
Estruturas de Dados Topológicas

Como destaca (Ferreira, 2006), a estrutura de dados *Winged-Edge*, proposta por (Baumgart, 1972), foi uma das primeiras estruturas para representação de superfícies em $\mathbb{R}^3$. Posteriormente essa estrutura sofreu muitas modificações a fim de aumentar a variedade de objetos a serem representados por ela. Entre essas modificações, destacam-se a inserção de laços (Braid *et al*, 1980) e a criação de uma nova estrutura de dados a partir dela chamada *Half-Edge* (Mantyla, 1987). Estas estruturas são muito poderosas no que diz respeito às suas capacidades de representação,uma vez que são capazes de representar subdivisões planares quaisquer, no entanto elas utilizam uma quantidade dispendiosa de memória, tornando-as muitas vezes inviáveis para aplicações com modelos muito grandes.

Mais recentemente, (Rossignac *et al*, 2001) e (Ferreira, 2006) propuseram duas estruturas de dados para malha triangulares muito concisas, *Corner Table* e CHE respectivamente. Ambas as estruturas se utilizam apenas de vetores de inteiros e um conjunto de regras para representar superfícies.

Neste capítulo, faremos uma revisão das estruturas de dados *Corner Table* e CHE, umas vez que estas duas estruturas são eficientes em termos de memória na representação de malhas triangulares e podem ser usadas em aplicações que envolvem grandes modelos.

3.1
Corner Table

A *Corner Table* é uma estrutura de dados topológica muito concisa que foi introduzida por (Rossignac *et al*, 2001) para ser usada em compressão de malhas triangulares. A estrutura foi usada para codificar e decodificar, em um formato compacto, malhas triangulares em um esquema chamado *Edgebreaker*.

A *Corner Table* foi apresentada de maneira bastante superfícial em (Rossignac *et al*, 2001), mas (Vieira, 2003) fez uma apresentação mais didática da mesma, e com alguns exemplos de algumas buscas topógicas básicas que facilitam a compreensão do funcionamento desta estrutura de dados, além de apresentar as implementações de algumas operações que serão utilizadas neste trabalho. Por estes motivos, optamos por apresentar a *Corner Table* da mesma forma, utilizando-se de

PUC-Rio - Certificação Digital Nº 1112650/CA

alguns trechos e exemplos apresentados em (Vieira, 2003), que serão devidamente citados.

3.1.1 Representação

Como destaca (Ferreira, 2006), a *Corner Table* usa um conceito equivalente ao conceito de *half-edge*: o *corner*. Um *corner* é uma associação de uma face a um dos seus vértices, enquanto que a *half-edge* é uma associação de uma face à uma de suas arestas. As duas abordagens são equivalentes, uma vez que para cada *corner*, sempre existe uma *half-edge* oposta a ele (Figura 3.1).

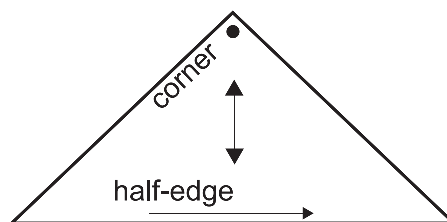


Figura 3.1: Relação entre *corner* e *half-edge*. Adaptado de: (Ferreira, 2006)

PUC-Rio - Certificação Digital Nº 1112650/CA

Na *Corner Table*, cada triângulo orientado é representado por 3 *corners* consecutivos que são indexados de acordo com a sua orientação. Assim uma superfície triangular formada por n triângulos possuirá $3n$ *corners* sendo, por definição, o triângulo orientado de índice 0 representado pelos *cornes* 0, 1 e 2, o triângulo de índice 1 representado pelos *corners* 3, 4 e 5, de modo que o i-ésimo triângulo posssui os *corners* $3i$, $3i+1$ e $3i+2$. Desta forma, um *corner* c é associado ao triângulo de índice t da seguinte forma:

$$t(c) = \lfloor c \div 3 \rfloor$$

Como apresentado em (Vieira, 2003) e (Rossignac *et al*, 2001), a representação da topologia na *Corner Table* consiste em duas tabelas de inteiros, O e V, com dimensões iguais a $3n$, onde n é o número de triângulos. Para cada *corner* c é gravado: $V[c]$, vértice do *corner*, e $O[c]$, *corner* oposto ao *corner* c. Os atributos dos vértices, juntamente com sua geometria, são armazenados em uma outra tabela G, cuja dimensão é dada pelo número de vértices da superfície.

A fim de representar superfícies com bordo, um *corner* c que for oposto à uma aresta de bordo da superfície terá o seu *corner* oposto definido com -1, ou seja, $O[c] = -1$.

A Figura 3.2 ilustra a configuração das tabelas O e V para uma malha triangular simples. O exemplo foi adaptado de (Vieira, 2003), onde este usava a malha de um tetraedro como exemplo. Optamos aqui por uma malha aberta para ficar evidente a representação de bordas.

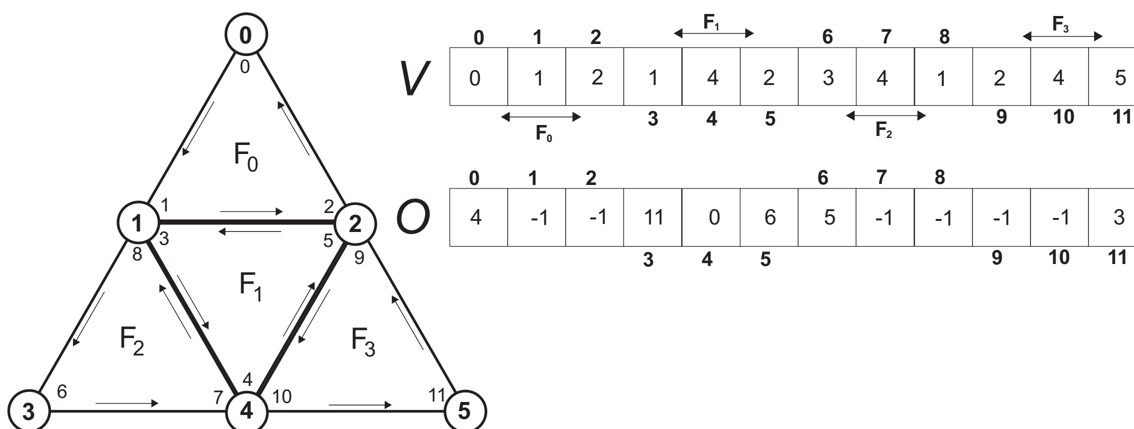


Figura 3.2: Representação de uma malha de triângulos usando *Corner Table*. Adaptado de (Vieira, 2003)

Como os *corners* são indexados de forma consecutiva e seguindo a orientação da face, é possível escrever rotinas para obtenção do *corners* denominadas *next* e *previous* de um *corner* c, usando apenas operações entre inteiros (Vieira, 2003) :

$$n(c) = next(c) = 3\lfloor c \div 3 \rfloor + (c+1) \mod 3$$

$$p(c) = prev(c) = 3\lfloor c \div 3 \rfloor + (c+2) \mod 3$$

Com estes operadores, as faces da direita e da esquerda podem ser obtidas da seguinte forma:

$$l(c) = left(c) = O[prev(c)]$$

$$r(c) = right(c) = O[next(c)]$$

A Figura 3.3 ilustra os *corners* que resultam das operaçãos acima a partir de suas aplicações no *corner* c.

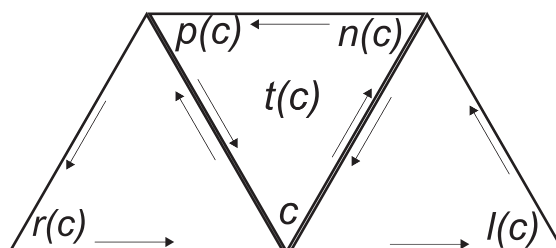


Figura 3.3: *Corners* $n(c)$, $p(c)$, $l(c)$ e $r(c)$ que resultam das operações de *next*, *previous*, *left* e *right*, respectivamente. O índice do triângulo ao qual pertence o *corner* c, é denotado por $t(c)$. Adaptado de (Rossignac *et al*, 2001) e (Vieira, 2003)

O leitor ao olhar a Figura 3.3, pode imaginar que os *corners* $l(c)$ e $r(c)$ estão trocados, uma vez que aparentemente $l(c)$ está à direita e $r(c)$ está á esquerda de c. No entando, esta notação pode ser compreendida com base nas regras da mão direita

PUC-Rio - Certificação Digital Nº 1112650/CA

PUC-Rio - Certificação Digital Nº 1112650/CA

e esquerda, ou seja, partindo de c e aplicando a regra da mão direita seguindo a orientação do triângulo se obtém o *corner* $r(c)$ e da mesma forma aplicando a regra da mão esquerda se obtém o *corner* $l(c)$.

Note que os *corners*, representados na Figura 3.2, não são gravados de fato. Um *corner* é o índice de um vértice armazenado na lista de triângulos, de forma que este existe de forma implícita.

A conectividade da malha triangular pode ser completamente representada pelas tabelas O e V. Além disso, a tabela G guarda os atributos de cada vértice, assim como sua geometria. Com as tabelas O e V é possível recuperar a vizinhança de um vértice a partir de um *corner* c, no entanto, a operação de recuperar um *corner* de um vértice tem custo linear, de forma que aplicações que tem como entrada os vértices da malha podem ter problemas de eficiência. Uma solução para este problema foi prosposta por (Vieira, 2003). Neste caso, a estrutura original (Rossignac *et al*, 2001) seria modificada para ter mais uma tabela C, do tamanho da lista de vértices, que armazenaria para cada vértice um dos seus *corners* incidentes.

A fim de representar de forma implícita o bordo da superfície, a tabela C pode ser construída de forma que se um vértice pertence ao bordo da superfície, o *corner* c associado a ele a ser armazenado é aquele em que $next(c)$ leva a um outro vértice de bordo.

A Figura 3.4 ilustra o exemplo da Figura 3.2 com a inclusão da tabela C.

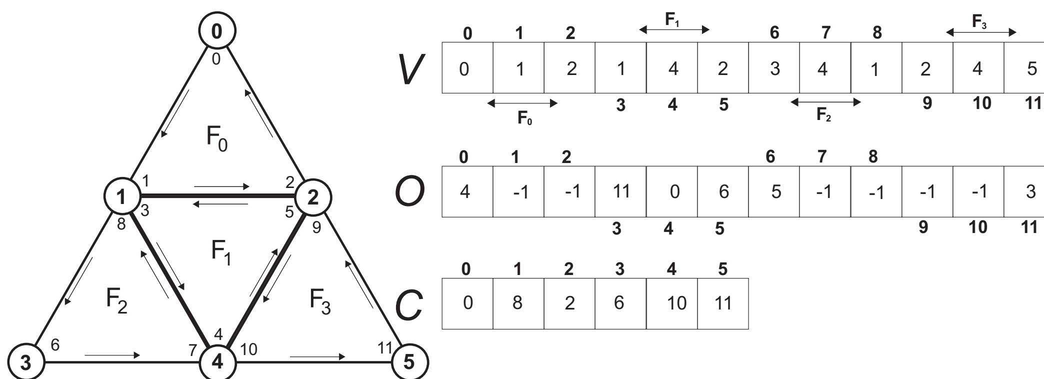


Figura 3.4: Representação incluindo a tabela C

3.1.2
Incidências e Adjacências

Nesta seção mostra como obter as relações de incidência e adjacência necessárias para identificação da conectividade dos vértices, arestas e das faces de uma malha triangular usando a *Corner Table*. Estas buscas topológicas foram implementadas por (Vieira, 2003), e são repoduzidas aqui para facilitar a leitura.

Em (Vieira, 2003) a notação a seguir é utilizada:

V: um grupo de vértices;

v: um determinado vértice;
E: um grupo de arestas;
e: uma determinada aresta;
F: um grupo de faces
f: uma determinada face;
$\{grupo\}$: um conjunto não ordenados de elementos;

A *Corner Table* não tem uma representação explícita para as arestas, uma vez que uma aresta pode ser representada implicitamente por um dos seus *corners* opostos. Um *corner* oposto c representa a aresta formada pelos vértices $V[next(c)]$ e $V[prev(c)]$ (Vieira, 2003). Note que se a aresta e for interna à malha, ela pode ser representada por dois *corners* e quando esta pertence ao bordo da superfície pode ser representada apenas por um *corner*.

Na Figura 3.5 a aresta e_1 pode ser representada pelos *corners* 9 e 14, já a aresta e_2 só pode ser representada pelo *corner* 1.

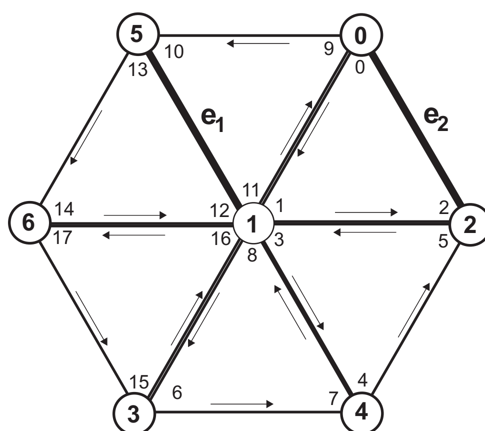


Figura 3.5: A estrela de um vértice e a representação implícita de arestas

Para simplificar os algoritmos vamos introduzir uma função que retorna a aresta oposta a um *corner* c (ver Algoritmo 1).

Algoritmo 1 edge(c)

$e = \{V[prev(c)], V[next(c)]\}$
return e;

Vizinhanças de um Vértice

Dado um vértice v, pode-se obter um *corner* c associado a v da tabela C, ou no caso desta não existir, percorrer a tabela V em tempo linear. Dado c, pode-se percorrer a estrela de v, obtendo os conjuntos de vértices $\{\mathbf{V}\}(v)$ adjacentes a v e arestas $\{\mathbf{E}\}(v)$ e faces $\{\mathbf{F}\}(v)$ incidentes a v em tempo $O(Degree(v))$ (Vieira, 2003)(ver Algoritmo 2).

PUC-Rio - Certificação Digital Nº 1112650/CA

Algoritmo 2 VizinhançaDeUmvértice(v). Fonte: (Vieira, 2003)

Inicializar: $c = i = C[v]$;
$\{\mathbf{V}\}(v) = \emptyset$;
$\{\mathbf{E}\}(v) = \emptyset$;
$\{\mathbf{F}\}(v) = \emptyset$;
repetir
$j = next(i)$;
$e = edge(j)$;
$\{\mathbf{V}\}(v) = \{\mathbf{V}\}(v) \cup V[j]$;
$\{\mathbf{E}\}(v) = \{\mathbf{E}\}(v) \cup e$;
$\{\mathbf{F}\}(v) = \{\mathbf{F}\}(v) \cup \lfloor\, i \div 3 \,\rfloor$;
$i = next(right(i))$;
até que $(i == c)$

Por simplicidade o Algoritmo 2 não trata superfícies de bordo. O laço que gira na estrela do vértice pode ser interrompido ao se deparar com o bordo da superfície, neste caso o a busca deve ser retomada pelo *corner* inicial, mas girando no sentido oposto desta vez.

Vizinhanças de uma aresta

Dada uma aresta e, obtemos o *corner* c oposto a e, assim pode-se obter os conjuntos de vértices $\{\mathbf{V}\}(e)$ e faces $\{\mathbf{F}\}(e)$ incidentes a e e arestas $\{\mathbf{E}\}(e)$ adjacentes a e em tempo constante (Vieira, 2003)(ver Algoritmo 3).

Note que é necessário um tempo linear obter um *corner* c oposto a uma aresta e, uma vez que os *corners* deveriam ser visitados até que se encontrasse o *corner* c oposto a aresta que tenha os mesmos vértices extremos que e. Assim como a tabela C, que associada um *corner* para cada vértice, uma nova tabela pode ser alocada para associar um *corner* a cada aresta, no entanto, esse tipo de consulta explícita não será necessária em nossa aplicação, não necessitando assim desta tabela extra.

Algoritmo 3 VizinhançaDeUmaAresta(e). Fonte: (Vieira, 2003)

Inicializar: c com o valor de um $corner$ oposto a e
$e_1 = edge(prev(c))$;
$e_2 = edge(next(c))$;
$e_3 = edge(prev(O[c]))$;
$e_4 = edge(next(O[c]))$;
$\{\mathbf{V}\}(e) = \{V[prev(c)], V[next(c)]\}$;
$\{\mathbf{E}\}(e) = \{e_1, e_2, e_3, e_4\}$;
$\{\mathbf{F}\}(e) = \{\lfloor\, c \div 3 \,\rfloor, \lfloor\, O[c] \div 3 \,\rfloor\}$;

PUC-Rio - Certificação Digital Nº 1112650/CA

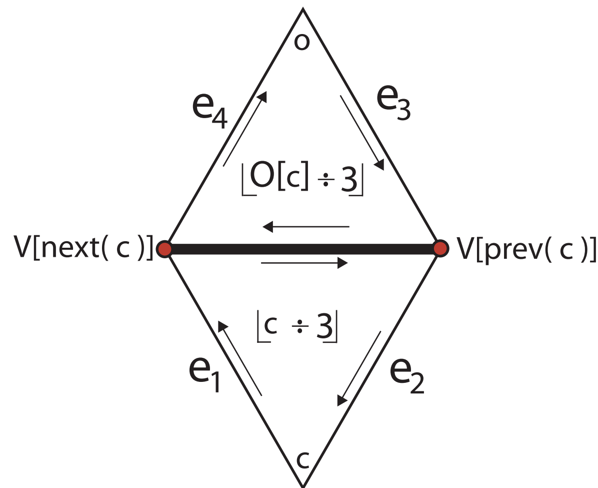


Figura 3.6: Vizinhança de uma aresta

Vizinhanças de uma face

Dada uma face f, obtemos o primeiro *corner* c associado a ela fazendo $c = 3f$, e então pode-se obter os conjuntos de vértices $\{\mathbf{V}\}(f)$ e arestas $\{\mathbf{E}\}(f)$ incidentes a f e faces $\{\mathbf{F}\}$ adjacentes a f em tempo constante (Vieira, 2003)(ver Algoritmo 4).

Algoritmo 4 VizinhançaDeUmaFace(f). Fonte: (Vieira, 2003)

Inicializar: $c = 3f$;
$e_1 = edge(c)$;
$e_2 = edge(next(c))$;
$e_3 = edge(prev(c))$;
$\{\mathbf{V}\}(f) = \{V[c], V[prev(c)], V[next(c)]\}$;
$\{\mathbf{E}\}(f) = \{e_1, e_2, e_3\}$;
$\{\mathbf{F}\}(f) = \{\lfloor O[c] \div 3 \rfloor, \lfloor O[prev(c)] \div 3 \rfloor, \lfloor O[next(c)] \div 3 \rfloor\}$;

3.2 CHE - Compact Half-Edge

A CHE, ou *Compact Half-Edge*, é uma estrutura de dados topológica, proposta por (Ferreira, 2006), que tem como uma de suas principais características o fato de ser escalonável, além de usar o conceito de *half-edge*. A estrutura é dividida em quatro níveis, de forma que, à medida que haja disponibilidade física, uma quantidade adicional de mémoria pode ser alocada para obter melhor performace nas consultas topológicas, ou por outro lado, liberar mémoria com

PUC-Rio - Certificação Digital Nº 1112650/CA

o objetivo de processar modelos maiores, com o custo de perder desempenho no tempo de resposta das informações topológicas. Assim, a estrutura busca um balanceamento entre o uso de memória e a eficiência da representação topológica.

A seguir, serão descritos os quatro níveis da CHE, assim como as informações que são acrescidas a cada nível para melhorar a efiências nas consultas topológicas. A descrição a seguir foi retirada de (Ferreira, 2006), onde pode ser encontrada uma descrição mais deltalhada da CHE.

3.2.1 Nível 0: Sopa de Triângulos

No nível 0 são armazenadas apenas as informações essenciais para representação da malha triangular, como a geometria dos vértices e a lista de triângulos em si. Neste nível não há intenção de representar de forma eficiente as relações de incidência e adjacência, uma vez que não são guardadas informações de vizinhança entre os triângulos de forma explícita.

PUC-Rio - Certificação Digital Nº 1112650/CA

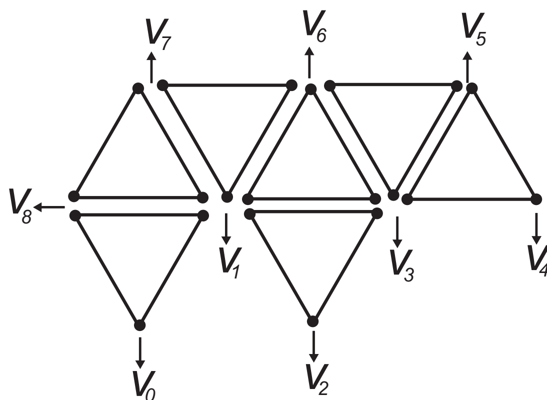


Figura 3.7: Nível 0 da CHE: Sopa de triângulos. Adaptado de (Ferreira, 2006)

A geometria dos vértices é armazenada em uma tabela G, que tem uma entrada para cada vértice. A lista contendo os triângulos é armazenada na tabela V. Nesta tabela é amazenado para cada *half-edge* o índice do seu vértice inicial, ou seja, o inteiro $v = V[he]$ corresponde ao índice do vértice inicial de uma *half-edge* he.

De forma semelhante à *Corner Table*, cada triângulo é representado por 3 *half-edges* consecutivas que são indexadas de acordo com a sua orientação. Assim, as *half-edges* 0, 1 e 2 pertencem ao triângulo de índice 0, as *half-edges* 3, 4 e 5 pertencem ao triângulo de índice 1, e assim por diante. Uma *half-edge* de índice he pertence ao triângulo $\lfloor he \div 3 \rfloor$ e um triângulo de índice t contém as *half-edges* de índices $3t$, $3t + 1$ e $3t + 2$. Além disso, pode-se obter as *half-edges* denominadas *next* e *previous* de uma *half-edge* he usando apenas operações de inteiros, assim como na *Corner Table*:

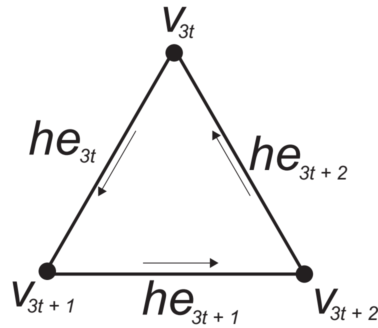


Figura 3.8: Relação entre vértice e *half-edge*. Adaptado de (Ferreira, 2006)

$$h_n = next(he) = 3\lfloor he \div 3 \rfloor + (he + 1) \mod 3$$

$$h_p = prev(he) = 3\lfloor he \div 3 \rfloor + (he + 2) \mod 3$$

3.2.2 Nível 1: Adjacência entre Triângulos

No nível 1, a CHE passa a tratar a malha como um conjunto de triângulos conectados (Figura 3.9). Isso é feito pela adição de um novo vetor O, que tem mesma dimensão da lista triângulos, e que em cada posição indica o índice da *half-edge* oposta. Uma *half-edge* h é oposta a *half-edge* he se h tem os mesmos vértices de he, mas orientação oposta (Figura 3.10). Quando he for incidente à uma aresta de bordo, esta não terá uma *half-edge* oposta. Neste caso, $O[he] = -1$.

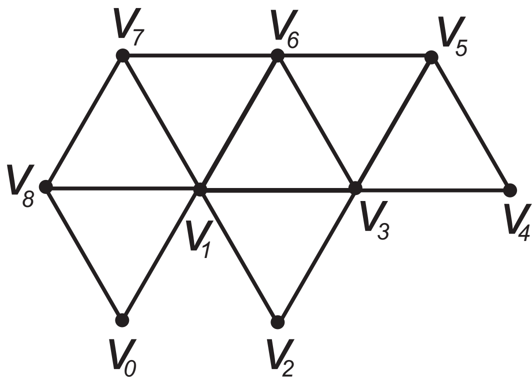


Figura 3.9: Nível 1 da CHE: Adjacência entre os triângulos. Adaptado de (Ferreira, 2006)

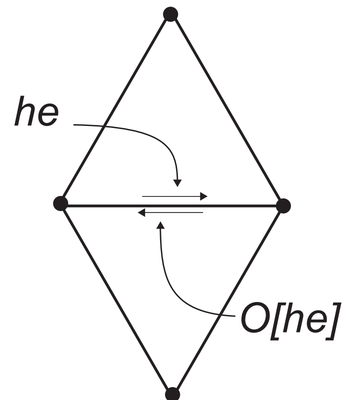


Figura 3.10: *Half-edges* opostas. Adaptado de (Ferreira, 2006)

3.2.3

PUC-Rio - Certificação Digital Nº 1112650/CA

Nível 2: Representação das Células

No nível 2 da CHE, é acrescentada uma representação explícita para cada célula da malha. Tal representação é feita por meio de duas novas tabelas, que serão chamadas de EH e VH.

A tabela EH tem a sua dimensão determinada pelo número de arestas, e associa cada aresta com uma de suas *half-edges* incidentes, já que a outra pode ser obtida através do vetor O. Essa representação explícita permite que atributos relacionados às arestas possam ser armazenados em uma outra tabela, que pode ter a mesma indexação da tabela EH.

Note que os vértices que compõe a aresta são facilmente recuperados através dos vértices iniciais de suas *half-edges*.

Já na tabela VH, cada vértice é associado à uma de suas *half-edges* incidentes. Como podem existir várias *half-edges* que podem ser associadas ao vértice, (Ferreira, 2006) definiu um critério de escolha. Se o vértice é de bordo, então a *half-edge* escolhida será a *half-edge* de bordo que tem o vértice como vértice inicial. Se o vértice for de interior, a *half-edge* armazenada poderá ser qualquer *half-edge* incidente que tenha o vértice como vértice inicial.

Desta forma, se torna bastante simples classificar os vértices da malha como interior ou de bordo. Dado um vértice v, basta verificar se $O[VH[v]] = -1$. Em caso afirmativo, o vértice é de bordo.

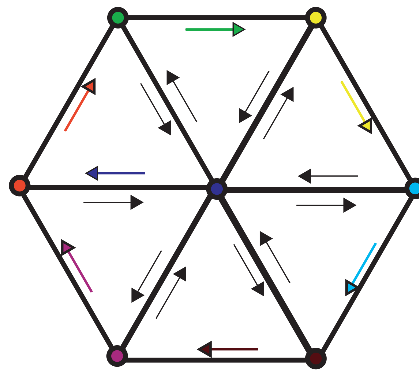


Figura 3.11: Escolha das *half-edges* para serem armazenadas na tabela VH. Adaptado de (Ferreira, 2006)

3.2.4
Nível 3: Representação das Curvas de Bordo

No nível 3 da CHE, é adicionada uma estrutura para representação explícita para curvas de bordo de superfícies. Para isto, uma nova tabela é CH é criada. Cada posição desta tabela armazena uma curva de bordo. Cada curva de bordo é representada por uma das *half-edges*, uma vez que a partir desta, todas as outras

PUC-Rio - Certificação Digital Nº 1112650/CA

podem ser obtidas de forma eficiente através das regras aritméticas e da tabela O, como no Algoritmo 5.

Algoritmo 5 *Percorrimento das curvas de bordo( ). Fonte: (Ferreira, 2006)*

---

para $he \in CH$
  $he_0 = he$
  repetir //Percorre as *half-edges* da borda
    borda.insert(he);
    repetir //Busca a próxima *half-edge* que pertence à borda
      $he = next(O[next[he]])$
    até que $(O[he] == -1)$
  até que $(he_0 \neq he)$
fim para

A localização das arestas na curva de bordo pode ser otimizada através da de um atributo gravado na tabela O. Nesta tabela pode ser armazenado o índice da componente de bordo a qual pertence a *half-edge* de bordo. Por exemplo, se uma *half-edge* de bordo he pertence à componente de número 3, então $O[he] = -3$.

3.2.5 Exemplo

A seguir é mostrado um exemplo de uma simples malha triangular representada pela CHE. Neste exemplo é mostrada uma CHE nível 3, com todas as tabelas preenchidas como definido acima.

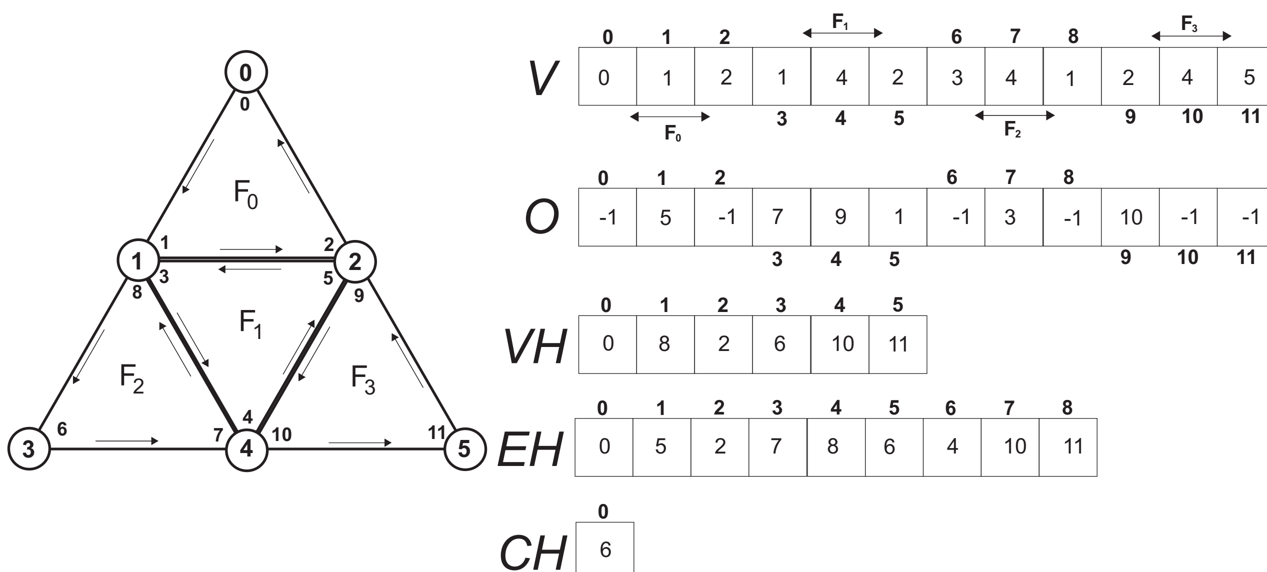


Figura 3.12: Exemplo de uma malha representada com a CHE

PUC-Rio - Certificação Digital Nº 1112650/CA

3.3
Corner Table vs CHE

A CHE pode ser vista como uma evolução da *Corner Table*, com um poder de representação maior e com maior eficiência em algumas operações topológicas, como recuperar uma curva de bordo, e ainda tem a propriedade de ser escalonável. Este trabalho poderia ser realizado usando ambas as estruturas, no entanto optamos por usar a *Corner Table* por ser mais simples, permitir implementações eficientes das buscas topológicas que necessitamos e satisfazer a principal condição do nosso problema, que é o uso de pouca mémoria.

PUC-Rio - Certificação Digital Nº 1112650/CA

PUC-Rio - Certificação Digital Nº 1112650/CA

4
Operadores na Corner Table

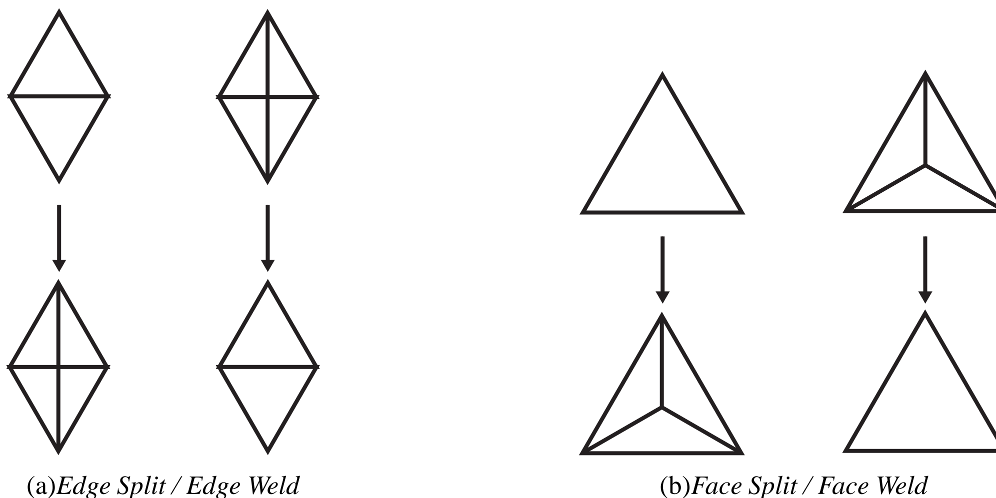
Neste capítulo são apresentadas as implementações usando a *Corner Table* dos operadores utilizados para realizar as subdivisões global e adaptativa e o mecanismo de *undo/redo*. São eles: *edge split* e seu inverso *edge weld*, *edge flip* e uma implementação de um *edge flip* inverso. O motivo da necessidade da implementação de um *flip* inverso está detalhado na seção 4.3.

Estes operadores podem ser classificados como operadores estelares, que por sua vez tem propriedades importantes, apresentadas superficialmente na seção 4.1.

4.1
Operadores Estelares para Malhas Triangulares

De acordo com (Velho, 2003), teoria estelar estuda as equivalências entre complexos simpliciais, tendo como principal resultado o seguinte teorema:

Teorema 4.1 *Duas n-variedades combinatórias são linearmente homeomorfas por partes, se e somente se, são relacionadas por uma sequência de operações estelares elementares (Newman, 1926) (Pachner, 1991).*



(a)*Edge Split / Edge Weld* (b)*Face Split / Face Weld*

Figura 4.1: Operações estelares de subdivisão. Fonte: (Velho, 2003)

Em outras palavras, o teorema 4.1 diz que os operadores estelares realizam transformações entre *manifolds* combinatórios equivalentes. Este resultado é muito

importante, uma vez que a subdivisão tem o objetivo apenas de refinar a superfície e não de alterar a sua topologia. Usando estes operadores, garantimos que a superfície subdividida é sempre homeomorfa à superfície grosseira inicial.

Para o caso de malhas de triângulos os operadores são os seguintes: *face split* e seu inverso *face weld*; *edge split* e seu inverso *edge weld* (Velho, 2003)(ver Figura 4.1).

O operador de *edge flip* também é um operador estelar, uma vez que ele pode ser obtido através de uma composição de *edge split* e *edge weld*, como mostrado na Figura 4.2.

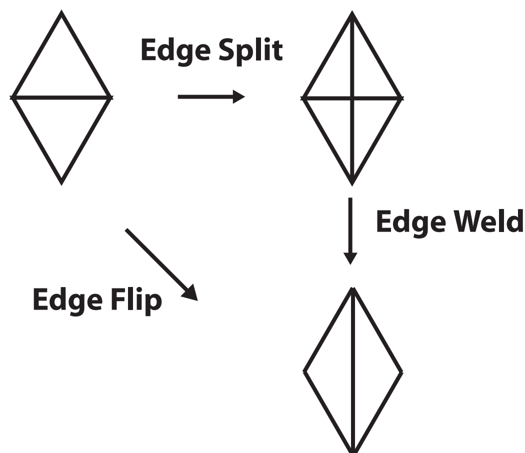


Figura 4.2: *Edge Flip* como uma composição de *edge split* + *edge weld*

Uma descrição mais detalhada sobre teoria estelar pode ser encontrada em (Lickorish1999) e (Velho, 2003).

4.2 Edge Flip

Uma implementação para o operador de *Edge Flip* na *Corner Table* foi primeiro proposta por (Vieira, 2003). No nosso trabalho utilizaremos uma adaptação desta implementação. Na prática o resultado das duas implementações são equivalentes, há apenas uma translação dos *corners* dentro das duas faces envolvidas.

O operador de *Edge Flip* consiste em transformar duas faces adjacentes em outras duas faces também adjacentes pelo *swap* da aresta em comum às duas.

Seja $e = \langle uv \rangle$ e s e t dois vértices opostos à aresta e, como na Figura 4.3. A operação de *Edge Flip* substitui a aresta e por $\langle st \rangle$ e as faces incidentes a e por $\langle tsu \rangle$ e $\langle vst \rangle$. Note ainda que a valência dos vértices u e v diminui, enquanto que a valência dos vértices s e t aumenta.

O Algoritmo 6 realiza a operação de *Edge Flip* em uma aresta oposta ao *corner* c_0.

PUC-Rio - Certificação Digital Nº 1112650/CA

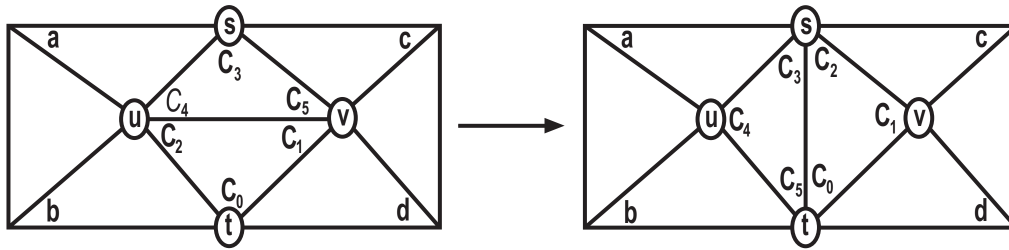


Figura 4.3: *Edge Flip* na Corner Table

Algoritmo 6 EdgeFlip(c_0). Adaptado de (Vieira, 2003)

// Identifica corners incidentes
c_1 = next(c_0); c_2 = previous(c_0);
b = O[c_1]; c = O[c_4];
c_3 = O[c_0]; c_4 = next(c_3);
c_5 = previous (c_3);

// Identifica vértices incidentes
t = V[c_0]; u = V[c_2];
s = V[c_3]; v = V[c_1]

// Efetua swap da aresta
V[c_2] = s;
V[c_5] = t;

// Atualiza a tabela C se esta existir
se existe a tabela C **então**
    C[t] = c_0;
    C[v] = c_1;
    C[u] = c_4;
    C[s] = c_2;
fim se

// Redireciona corners opostos
O[c_1] = c_4; O[c_4] = c_1;
O[c_0] = c; O[c] = c_0;
O[c_3] = b; O[b] = c_3;

Uma propriedade importante deste operador é que apenas dois *corners*, que são $prev(c_0)$ e $prev(O[c_0])$, mudam de vértices, ou seja, os *corners* c_0, $O[c_0]$, $next(c_0)$ e $next(O[c_0])$ permanecem associados aos mesmos vértices. Esta propriedade serve de base para implementação do algoritmo de subdivisão mostrado no capítulo 5.

PUC-Rio - Certificação Digital Nº 1112650/CA

4.3 Edge Flip Inverso

Para implementação de um mecanismo de *undo/redo* é importante ter operadores inversíveis, mas apesar do operador *edge flip* ter ele mesmo como seu inverso, a sua aplicação em uma aresta na *Corner Table* tem o efeito de girar as faces, de forma que a aplicação deste operador em uma aresta duas vezes seguidas produz uma configuração na estrutura de dados diferente da inicial, como mostrado na Figura 4.4.

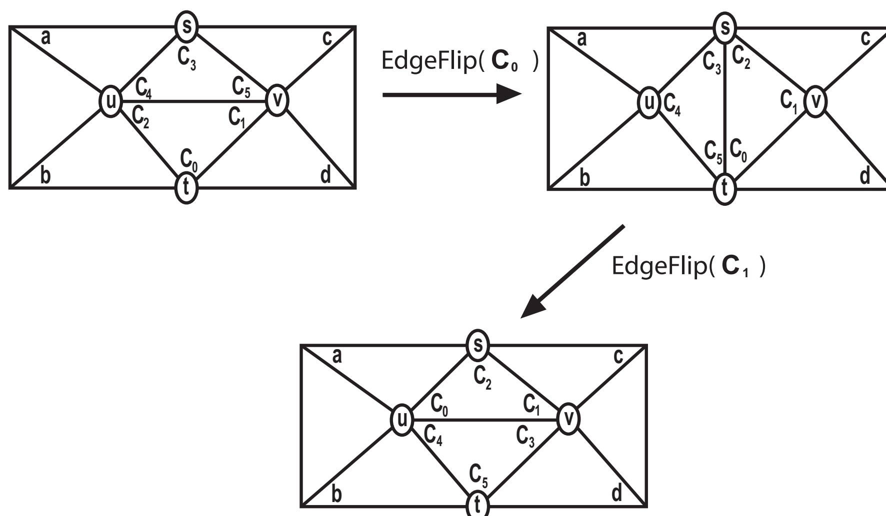


Figura 4.4: A aplicação do *Edge Flip* faz com que as faces girem no sentido anti-horário

Para voltar ao estado inicial seria necessário a aplicação do operador *Edge Flip* três vezes em sequência. Uma outra opção seria realizar um *swap* entre as duas faces e em seguida aplicar o operador *Edge Flip*, no entanto ainda teriamos duas operações. Por questões de eficiência optamos pela implementação de um operador *Edge Flip* que faça as faces girarem no sentido horário, como na Figura 4.5.

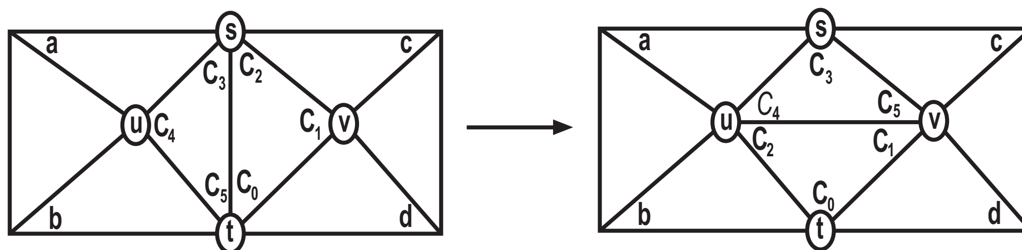


Figura 4.5: *Edge Flip* inverso

O Algoritmo 7, com pequenas mudanças para o Algoritmo 6, implenta o operador de *Edge Flip* Inverso.

PUC-Rio - Certificação Digital Nº 1112650/CA

Além de retornar à configuração inicial da *Corner Table*, este operador também mantém os *corners* $c_1, O[c_1], prev(c_1), prev(O[c_1])$ associados aos mesmos vértices.

Algoritmo 7 EdgeFlipInverso(c_1)

// Identifica corners incidentes
c_2 = next(c_1); c_0 = previous(c_1);
c_4 = O[c_1]; c_5 = next(c_4);
c_3 = previous (c_4);
b = O[c_3];c = O[c_0];

// Identifica vértices incidentes
u = V[c_4]; v = V[c_1];
s = V[c_2]; t = V[c_0];

// Efetua swap da aresta
V[c_5] = v; V[c_2] = u;

// Atualiza a tabela C se esta existir
se existe a tabela C **então**
  C[s] = c_3;
  C[u] = c_2;
  C[v] = c_1;
  C[t] = c_0;
fim se

// Redireciona corners opostos
O[c_0] = c_3; O[c_3] = c_0;
O[c_1] = b; O[b] = c_1;
O[c_4] = c; O[c] = c_4;

PUC-Rio - Certificação Digital Nº 1112650/CA

4.4 Edge Split

A operação de *Edge Split* consiste em transformar duas faces adjacentes em quatro faces pela inserção de um novo vértice sobre a aresta comum às duas.

Seja $e = \langle uv \rangle$ e s e t, dois vértices opostos à aresta e, como na Figura 4.6. A operação de *Edge Split* criará um novo vértice n sobre a aresta e, e este será conectado aos vértices s e t gerando as novas arestas $e_1 = \langle nt \rangle$ e $e_2 = \langle ns \rangle$. Além disso as faces $\langle tvu \rangle$ e $\langle suv \rangle$ são substituidas pelas faces $\langle tnu \rangle$, $\langle snv \rangle$, $\langle tvn \rangle$ e $\langle snu \rangle$.

O novo vértice n sempre terá valência 4, e os vértices s e t terão suas valências aumentadas.

Na Corner Table, cada face pode ser representada implicitamente por um índice inteiro f_i. Desta forma, ao subdividir a aresta e existem várias formas de reposicionar as faces F_1 e F_2. Para facilitar a implementação dos algoritmos de

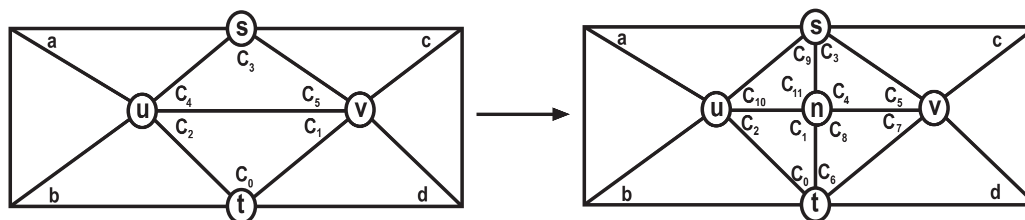


Figura 4.6: *Edge Split* na Corner Table

subdivisão e do mecanismo de *undo/redo*, será adotada a convenção exibida na Figura 4.7.

Nesta convenção as faces F_1 e F_2 existentes antes de realizar a operação de *Edge Split* são posicionadas na nova configuração da mesma forma, independente se a operação foi realizada no *corner* c ou no *corner* o, oposto ao corner c.

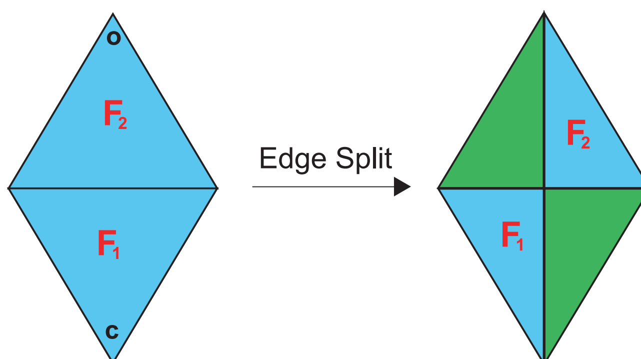


Figura 4.7: Convenção para posicionamento das faces após a realização da operação de *Edge Split*

O Algoritmo 8 realiza a operação de *Edge Split* em uma aresta oposta ao *corner* c_0, seguindo a convenção adotada.

Ao aplicar o operador de *Edge Split*, as faces F_1 e F_2 ocupam as mesmas posições na tabela V, sendo que apenas um dos vértices de cada uma muda durante a operação. Já as duas novas faces são adicionadas no final da tabela V. Se não houver espaço a tabela V é realocada de acordo com uma constante $m > 1$ definida pelo usuário. Desta forma o novo tamanho da n da tabela a cada vez que esta for realocada é dado por

$$n = n_c * m; \tag{4-1}$$

onde n_c é o número máximo de elementos que cabem na tabela corrente. Se o usuário não definir explicitamente um valor para m, este valor é definido como sendo 2.

PUC-Rio - Certificação Digital Nº 1112650/CA

Algoritmo 8 EdgeSplit(c_0)

Inicializar: F_n = número de triângulos da malha corrente
Inicializar: $F_{n+1} = F_n + 1$
// Identifica corners incidentes
c_2 = previous(c_0); c_1 = next(c_0);
d = O[c_2]; c_3 = O[c_0];
c_4 = next(c_3); c_5 = previous (c_3);
a = O[c_5];

// Identifica vértices incidentes
t = V[c_0]; v = V[c_1];
s = V[c_3]; u = V[$_2$];

// Refaz a triangulação
V[c_1] = n; V[c_4] = n;
V[$3 * F_n$] = t; V[$3 * F_n + 1$] = v;
V[$3 * F_n + 2$] = n; V[$3 * F_{n+1}$] = s;
V[$3 * F_{n+1}$] = u; V[$3 * F_{n+1} + 2$] = n;

// Atualiza a tabela C se esta existir
se existe a tabela C **então**
C[t] = c_0;
C[v] = c_5;
C[u] = c_2;
C[s] = c_3;
C[n] = c_1;
fim se

// Redireciona corners opostos
O[c_0] = c_9; O[c_9] = c_0;
O[c_2] = c_7; O[c_7] = c_2;
O[c_3] = c_6; O[c_6] = c_3;
O[c_5] = c_{10}; O[c_{10}] = c_5;
O[c_{11}] = a; O[a] = c_{11};
O[c_8] = d; O[d] = c_8;

4.5 Edge Weld

Como colocado na seção 4.1, o operador de *Edge Weld* é inverso ao operador de *Edge Split*. Desta forma, utilizaremos este operador no mecanismo de *undo/redo* para reverter o efeito do operador *Edge Split*.

Uma implementação para este operador foi primeiro proposta por (Vieira, 2003). Assim como no caso do operador de *Edge Flip*, utilizaremos uma adaptação desta implementação, a fim de garantir que a aplição de um *Edge Split*

PUC-Rio - Certificação Digital Nº 1112650/CA

seguido de um *Edge Weld* faça com que a estrutura de dados tenha uma configuração igual à aquela de antes da aplicação do operador de *Edge Split*.

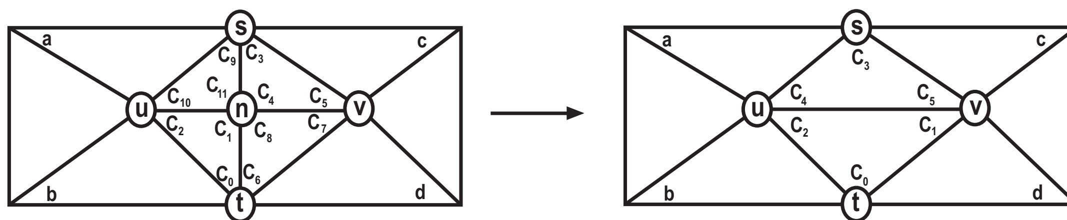


Figura 4.8: *Edge Weld* na Corner Table

Seja n um vértice de valência 4, adjacente aos vértices s, t, u e v, como na Figura 4.8. O operador *Edge Weld* remove o vértice v e refaz a triangulação do quadrilátero resultante dividindo-o em duas faces. Com a aplicação o operador a partir do *corner* c_1, a arestas $\langle tn \rangle$ e $\langle sn \rangle$ são removidas, de forma que as duas faces resultantes são $\langle tvu \rangle$ e $\langle suv \rangle$.

Algoritmo 9 EdgeWeld(c_1). Adaptado de (Vieira, 2003)

// Identifica corners incidentes
c_2 = next(c_1); c_0 = previous(c_1);
c_7 = O[c_2]; c_3 = O[prev(c_7)];
c_4 = next(c_3); c_5 = previous (c_3);
d = O[next(c_7)]; a = O[next(O[c_5])];

// Identifica vértices incidentes
v = V[c_7]; u = V[c_2];
s = V[c_3]; t = V[c_0];

// Refaz a triangulação
V[c_4] = u;
V[c_1] = v;

// Atualiza a tabela C se esta existir
se existe a tabela C **então**
    C[t] = c_0;
    C[v] = c_1;
    C[u] = c_2;
    C[s] = c_3;
fim se

// Redireciona corners opostos
O[c_5] = a; O[a] = c_5;
O[c_2] = d; O[d] = c_2;
O[c_0] = c_3; O[c_3] = c_0;

Como este operador é utilizado para reverter o operador de *Edge Split*, é garantido que as duas faces removidas estão no fim do vetor V, como visto na seção

PUC-Rio - Certificação Digital Nº 1112650/CA

4.4. Desta forma, para remover as duas faces, basta decrementar a variável que guarda o número de faces. Isto evita a necessidade de uma operação de *garbage-collector* para redimensionar as tabelas retirando os espaços não utilizados, uma vez que as tabelas são sempre contínuas. Para evitar que um grande montade de memória esteja alocada para o vetor V sem que este esteja utilizando, o vetor V é realocado sempre que

$$\frac{n_f}{n_c} < \frac{1}{m} \tag{4-2}$$

onde n_f é o número de faces da malha corrente, n_c é o número de máximo de faces que cabem na estrutura sem a necessidade dos vetores serem realocados e m é como definido na seção 4.4. O Algoritmo 9 implementa o operador de *Edge Weld* na *Corner Table*.

PUC-Rio - Certificação Digital Nº 1112650/CA

5
Subdivisão em Corner Table

Neste capítulo será apresentado uma forma de realizar o refinamento *1 para 4* pela composição de operadores estelares (seção 5.1), além de algoritmos para realização de subdivisão global (seção 5.2) e subdivisão adaptativa (seção 5.3) e uma forma de realizar *undo/redo* (seção 5.4) para estas subdivisões usando *Corner Table*. Os esquemas de subdivisão implementados usam a operação de refinamento *1 para 4* que foi apresentado no capítulo 2, mas a forma como a geometria dos novos vértices e calculada é independente. Desta forma, o usuário deve fornecer a geometria dos novos vértices. Isto proporciona uma flexibilidade maior, uma vez que a geometria pode ser definida de acordo com as necessidades do modelo, ou através de métodos como *Loop* e *Butterfly*.

5.1
Composição de operações

Diferente de estruturas de dados topológicas como *Half Edge* e *Winged Edge*, a *Corner Table* não representa malhas híbridas, uma vez que esta representa apenas elementos triangulares, sem vértices T. Desta forma, qualquer passo intermediário na *Corner Table* não pode criar vértices T e deve produzir apenas elementos triangulares, a fim de manter uma estrutura consistente durante toda execução do algoritmo. Para garantir tal restrição, a subdivisão *1 para 4* (Figura 5.1) é realizada pela composição de operações estelares elementares.

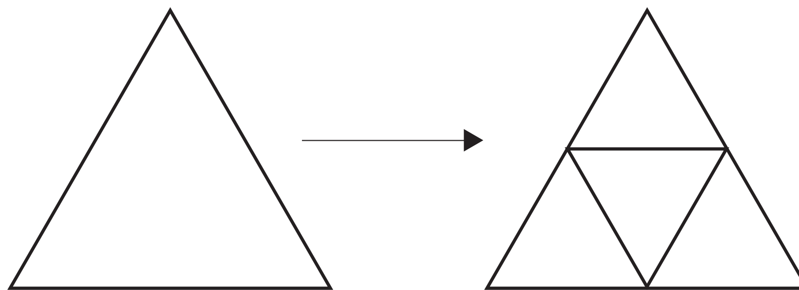


Figura 5.1: Subdivisão *1 para 4*

Esta operação pode ser dividida em 4 operações básicas. Uma sequência com três operações de *Edge Split* e ao final, uma operação de *Edge Flip*. As operações de *Edge Split* subdividem as três arestas do triângulo, e a operação de *Edge Flip* é

PUC-Rio - Certificação Digital Nº 1112650/CA

realizada sobre a aresta criada em decorrência do primeiro *Edge Split* realizado na face. É importante frisar que o operador *Edge Flip* deve ser aplicada somente ao final. Esta sequência de passos é ilustrada, para a subdivisão de um único triângulo, na Figura 5.2.

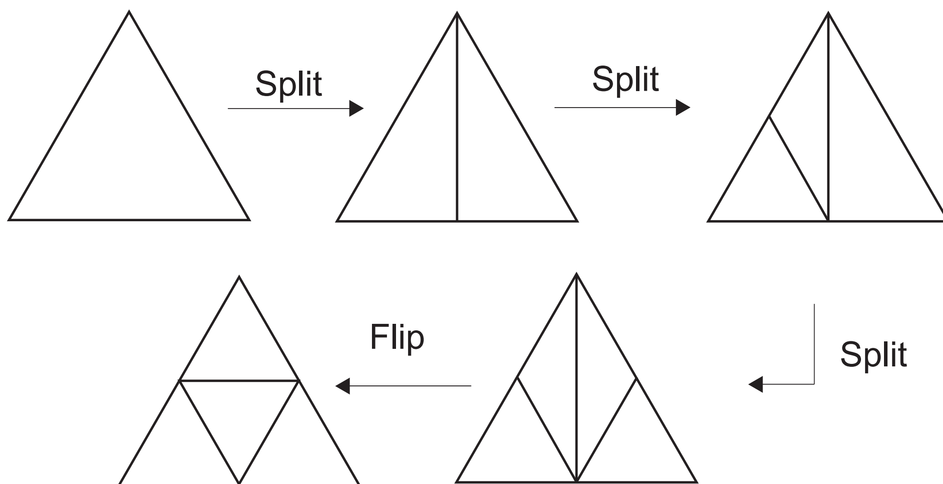


Figura 5.2: Subdivisão *1 para 4* como uma sequência de 3*Edge Split* + *Edge Flip*

PUC-Rio - Certificação Digital Nº 1112650/CA

A fim de manter uma malha sem vértices T, uma operação de *Edge Split* deve ser realizada também na face adjacente à aresta subdividida, isso evita a criação de vértices em *T*, mas obriga a subdivisão das faces opostas, como ilustrado na Figura 5.3. Note que ao subdividir as faces adjacentes, o que acontece de fato é uma triangulação simples (capítulo 2).

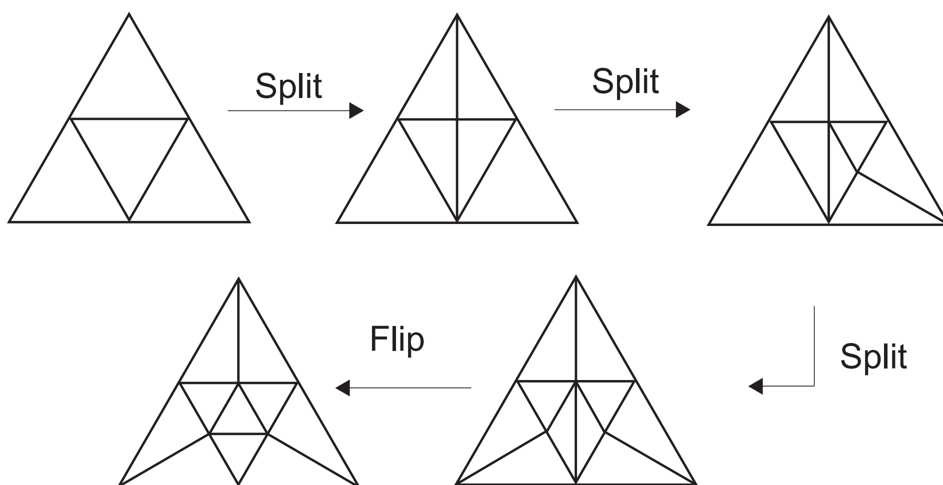


Figura 5.3: Subdivisão *1 para 4* subdividindo faces adjacentes

Como o operador de subdivisão *1 para 4* pode ser decomposto em operações estelares elementares, então este operador também é um operador estelar, desta forma, como descrito no capítulo 4, este operador não altera a topologia da superfície inicial durante o processo de subdivisão.

5.2 Subdivisão Global

Na subdivisão global, todas as faces da malha são recursivamente subdivididas aplicando o refinamento *1 para 4* descrito na seção anterior. A cada refinamento da malha M^i é aplicado um algoritmo de busca em largura que se baseia nos operadores de *Edge Split* e *Edge Flip* como ilustrado na Figura 5.3. Naturalmente todas as referências para vértices, arestas e faces se baseiam nos seus respectivos *corners*.

Uma dificuldade do algoritmo de subdivisão é que ao aplicar o refinamento *1 para 4* em um triângulo, os seus vizinhos são subdivididos para evitar a criação de vértices T, o que implica em um gerenciamento dos estados das faces da malha durante o processo de subdivisão para garantir que a cada refinamento da malha M^i todas as faces sofram apenas uma subdivisão *1 para 4*. Para este controle o algoritmo requer a utilização de vetores boleanos, *facesVisitadas* e *facesProcessadas*, de dimensão igual ao número de triângulos da malha refinada. O primeiro diz se a face sofreu o seu primeiro *Edge Split*, vindo de uma face vizinha, e o segundo indica se todas as arestas da face já foram subdivididas.

Uma outra dificuldade gerada pelo algoritmo mostrado na Figura 5.3 é o registro de qual aresta deve ser aplicado o operador de *Edge Flip* após todas todas as arestas da face terem sido subdivididas. Note que a aresta que sofre esta operação é sempre a correspondente a primeira aresta a dividir a face.

A seguir segue uma ideia geral de um algoritmo proposto.

1. Inicializar o vetores *facesVisitadas* e *facesProcessadas* .
2. Aplicar o operador de *Edge Split* em uma aresta correspondente a um *corner* inicial c_0 qualquer da malha. Esta operação gera o primeiro par vértice e aresta (v, e) que deve ser colocado na fila Q, onde v é o novo vértice gerado e e é a aresta que subdiviu a face oposta. Guarde uma referência para a aresta e_0 que dividiu a face que contém o *corner* inicial e marque as novas faces geradas como visitadas.
3. Enquanto a fila Q não for vazia, faça:
   (a) Retire o primeiro par de *corners* correspondentes do vértice v e aresta e da fila Q.
   (b) Processe o vértice v, subdividindo todas as faces de sua estrela que ainda não foram processadas. Cada aresta subdividida neste processo, gera um novo par (v, e) deve ser colocado na fila Q. Cada face da estrela de v é marcada como processada e as opostas

PUC-Rio - Certificação Digital Nº 1112650/CA

como visitadas. Se uma face oposta já tiver sido visitada, então as faces geradas por sua subdivisão são marcadas como processadas.

(c) Aplique o operador de *Edge Flip* na aresta e.

4. Ao final da busca, aplicar o operador de *Edge Flip* na aresta e_0.

A Figura 5.4 illustra o funcionamento deste algoritmo. As arestas em destaque são aquelas as quais o operador de *Edge Flip* deve ser aplicado após cada vértice ser processado.

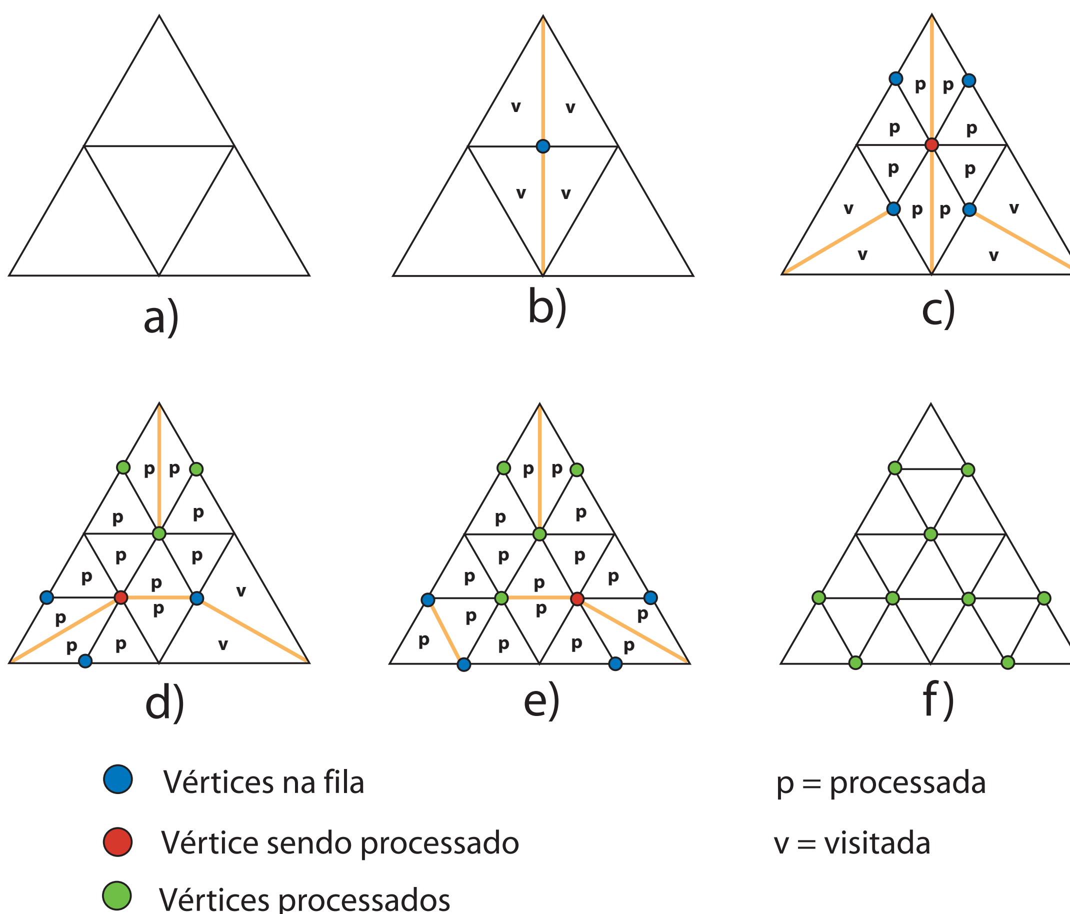


Figura 5.4: a) Malha inicial a ser subdividida. b-e) Os pares de *cornes* (v, e) são retirados da fila e as faces em suas estrelas que ainda não foram processadas são subdivididas. Ao processar um vértice, as faces de sua estrela são marcadas como processadas e as faces opostas são marcadas como visitadas. Após subividir toda estrela do vértice, o operador de *Edge Flip* é aplicado sobre a aresta oposta ao *corner* e. f) Malha refinada.

No Algoritmo 10 apresentamos uma implementação para o algoritmo da Figura 5.4 em *Corner Table*. O Algoritmo 10 usa o Algoritmo 11 para aplicar o operador de *Edge Split* sobre as arestas e realizar as atualizações necessárias, enquanto o Algoritmo 10 é responsável pela busca em si.

PUC-Rio - Certificação Digital Nº 1112650/CA

Algoritmo 10 SubdivisaoGlobal

```
1: //Declara as estruturas usadas pela subdivisão global
2: nT = número de triângulos da malha inicial;
3: Fila Q; bool facesVisitadas[4 * nT] = false, facesProcessadas[4 * nT] = false;
4: //realiza a operação de split em uma aresta inicial
5: runSplit( c0, facesVisitadas, facesProcessadas, Q );
6: enquanto (!Q.vazia( )))
7:     (v, e) = Q.remove();
8:     inicial = v;
9:     corrente = inicial;
10:    //percorre a estrela do vértice realizando a operação
11:    //de split nas arestas opostas
12:    repetir
13:        //realiza o split apenas se a face não tiver sido processada.
14:        se ( !facesProcessadas[corrente/3] ) então
15:            runSplit( corrente, facesVisitadas, facesProcessadas, Q );
16:        fim se
17:        aux = right( operation );
18:        corrente = next( aux );
19:    até que (inicial == corrente || aux == -1)
20:    //Verifica se parou por voltar ao inicio. Se sim vai para o próximo vértice
21:    se (aux ≠ -1) então
22:        se (e ≠ - 1) então
23:            EdgeFlip( e );
24:        fim se
25:        continue;
26:    fim se
27:    //por parar em uma borda, retotna o ao inicio e gira no sentido contrário
28:    corrente = v;
29:    repetir
30:        se ( !facesProcessadas[corrente/3] ) então
31:            runSplit( corrente, facesVisitadas, facesProcessadas, Q );
32:        fim se
33:        aux = left( operation );
34:        corrente = prev( aux );
35:    até que  aux == -1)
36:    se (e ≠ - 1) então
37:        EdgeFlip( e );
38:    fim se
39: fim enquanto
40: EdgeFlip( prev( c0 ) );
```

Note que ao aplicar o operador de *Edge Split*, o *corner* do próximo vértice a entrar na fila é sempre o $next$ do *corner* corrente, e a aresta a qual será aplicado o operador de *Edge Flip* após o vértice ser processada será dada pelo *previous* do *opposite* do *corner* corrente. A convenção para o operador de *Edge Split* definida no capítulo 4 garante que mesmo para as faces opostas o *corner* oposto à aresta a

PUC-Rio - Certificação Digital Nº 1112650/CA

Algoritmo 11 runSplit

Entrada: *corner:* corner a partir do qual o operador de Edge Split deve ser aplicado.
Entrada: *facesVisitadas:* vetor que diz se uma face já foi visitada.
Entrada: *facesProcessadas:* vetor que diz se uma face já foi processada.
Entrada: *Q:* fila com os corners dos próximos vértices a serem processados e as arestas as quais o operador de *Edge Flip* deve ser aplicado após cada vértice ser processado.

```
1: //marca a face corrente e a nova face gerada como processadas
2: size = número de triângulos da malha corrente
3: facesProcessadas[ corner / 3] = true;
4: facesProcessadas[size] = true;
5: opposite = O[corner];
6:
7: //Aplica o operador de Edge Split
8: EdgeSplit( corner );
9:
10: //Salva um corner do vértice gerado para ser inserido na fila
11: novoVertice = next( corner );
12: arestaParaFlip = -1;
13: se (opposite ≠ -1) então
14:
15:     //Se a face oposta já foi visita, então marca a face oposta e nova face
16:     //gerada a partir dela como processadas
17:     se (facesVisitadas[opposite / 3]) então
18:         facesProcessadas[opposite / 3] = true;
19:         facesProcessadas[size + 1] = true;
20:     senão
21:         //Como a face ainda nao foi visitada, a nova que cruza face
22:         //é aquela a qual o operador de Edge Flip deve ser
23:         //aplicado após o vértice ser processado.
24:         arestaParaFlip = prev( opposite );
25:     fim se
26:
27:     //Marca a face oposta e a face gerada a partir dela como visitadas
28:     facesVisitadas[opposite / 3] = true;
29:     facesVisitadas[size + 1] = true;
30: fim se
31: //Insere o par (v, e) na fila
32: Q.insere( (novoVertice, arestaParaFlip)) ;
33:
```

ser “flipada”continua oposto à mesma aresta mesmo após as outras duas arestas da face serem subdivididas. Isto é ilustrado na Figura 5.6. Note ainda que, o operador de *Edge Flip* muda *corners* de posição, o que poderia mover o *corner* associado ao vértice enfileirado de posição, no entanto essa referência nunca é perdida pois o *corner* enfileirado continua associado ao mesmo vértice. Como descrito no capítulo

PUC-Rio - Certificação Digital Nº 1112650/CA

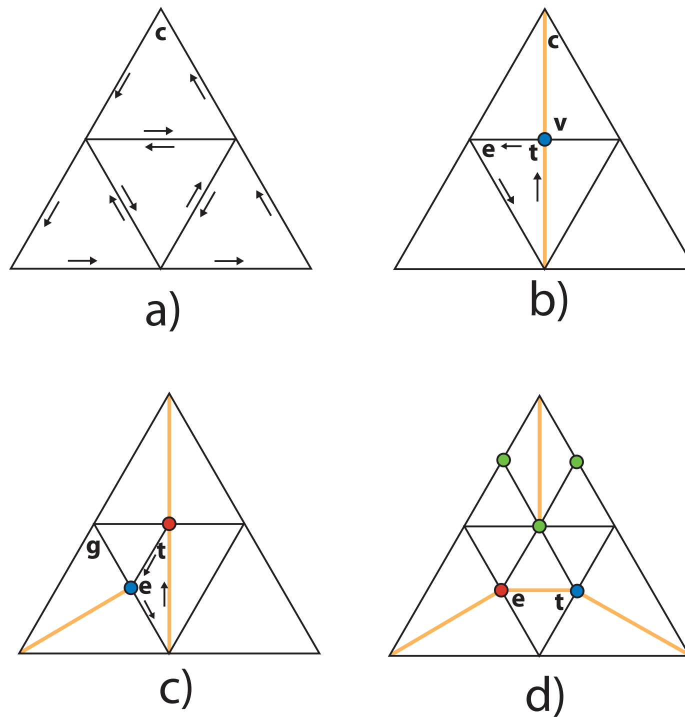


Figura 5.5: a) Malha inicial. b) O operador de *Edge Split* é inicialmente aplicado a partir do *corner* c. O par (v, e) é colocado na fila Q. c) Ao subdvidir as faces na estrela do vértice $V[v]$, o operador de *Edge Split* é aplicado a partir do *corner* t e o par (e, g) é colocado na fila Q. d) Após processar o vértice $V[v]$, o operador de *Edge Flip* é aplicado a patir do *corner* e. Apenas o corner $t = prev(e)$ associado ao vértice que já foi processado muda de vértice, logo os outros *corners* da face que podem já está na fila associados com os seus vértices continuam associados com os mesmos vértices.

PUC-Rio - Certificação Digital Nº 1112650/CA

4 e ilustrado na Figura 5.5, ao aplicar aplicar o operador de *Edge Flip* a partir de um *corner* e, o único *corner* da face que muda de vértice é o *prev*(e), que está associado ao vértice que está sendo processado. Logo, a única referência perdida é para o vértice $V[v]$, que já foi retirado da fila. A mesma propriedade é valida para a face oposta.

Na Figura 5.6, a face inicial é primeiro subdividida a partir do *corner* 0, logo o *corner* 1 é colocado na fila para que as faces na estrela do vértice $V[1]$ sejam subdivididas posteriormente, e o *corner* 2 é guardado para que ao final o operador de *Edge Flip* seja aplicado a partir dele.

Para realizar a subdivisão global, a *Corner Table* em seu estado original original é suficiente, logo a tabela C não é necessária. Com isso, um montante de memória é economizado sem nenhuma perda em eficiência. Além disso o Algoritmo 10 usa uma fila como estrutura auxiliar para realizar a busca na malha e dois vetores de boleanos, que juntos são responsáveis por um consumo de memória equivalente a 2 bits por face refinada.

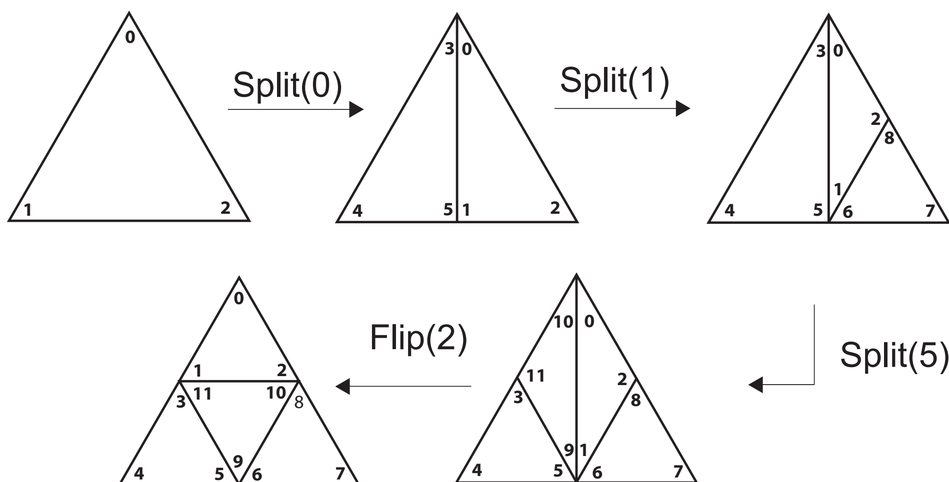


Figura 5.6: Exemplo da aplicação do algoritmo de subdivisão em um triângulo usando *Corner Table*. Sequência de aplicação dos operadores: *EdgeSplit(* 0 *)*, *EdgeSplit(* 1 *)*, *EdgeSplit(* 5 *)* e *EdgeFlip(* 2 *)*

5.3 Subdivisão Local

Na subdivisão local apenas um subconjunto das faces é subdividido. Como mostrado no capítulo 2 isso pode gerar algumas inconsistências. O esquema de subdivisão adaptativa proposto por (Pakdel and Samavati, 2007) resolve estas inconsistências, por esse motivo adotamos o método de subdivisão incremental como o método de subdivisão adaptativa a ser implementado.

Algoritmo 12 SubdivisaoIncremental

Entrada: S: conjunto dos vértices na região a ser subdividida.
Entrada: r: distância máxima em arestas para expansão do conjunto S.
Saída: S'

1: *//Obtém para cada vértice de* S *um dos seus corners*
2: corners[] = obtemCornersDosVertices(S);
3:
4: *//A partir dos corners do conjunto* S*, faz a expansão para* $E^r(S)$.
5: *//Aqui* $E^r(S)$ *em vez de vértices, tem um corner para cada vértice da expansão*
6: $E^r(S)$ = obtemCornersNaVizinhancaR(corners, r);
7:
8: *//Calcula as faces formadas pelos vértices do conjunto* $E^r(S)$
9: faces[] = obtemFacesFormadasPorErS($E^r(S)$);
10:
11: *//Aplica a subdivisão no conjunto de faces e retorna o novo conjunto* S'
12: **return** subdivideFaces(faces);

A entrada para o algoritmo de subdivisão incremental é um conjunto S de vértices de uma área selecionada para subdivisão. Em seguida esse conjunto é expandido para um novo conjunto $E^r(S)$ que contém todos os vértices de S e mais

PUC-Rio - Certificação Digital Nº 1112650/CA

aqueles que distam ao menos r arestas de S, e então a região que contém estes vértices é subdividida. No entanto o operador de subdivisão *1 para 4* é aplicado sobre faces da malha, logo alguns passos anteriores à subdivisão das faces de fato devem ser efetuados. O Algoritmo 12 realiza os passos necessários para subdividir adaptativamente a malha de acordo com a subdvisão incremental e a seguir cada passo é explicado com mais detalhes.

5.3.1 Obtenção de corners de um conjunto de vértices

A implementação da função que obtém um *corner* para cada vértice da linha 2 do Algoritmo 12 depende se a *Corner Table* implementada tem a tabela C ou não. Se a tabela C existir a obtenção destes *cornes* tem um custo linear com o número de vértices, uma vez que basta consultar esta tabela, caso contrário, a obtenção de cada *corner* tem tempo linear, logo o custo para obtenção de todos os *corners* é $O(n_t n_v)$, onde n_v é o número de vértices e n_t é o número de triângulos da malha corrente. A implementação desta rotina, usando a *Corner Table*, é mostrada no Algoritmo 13.

Algoritmo 13 obtemCornersDosVertices

Entrada: S: conjunto dos vértices
Saída: corners[]: vetor com um *corner* para cada vértice de entrada

```
1: se existe tabela C então
2:     //Consulta tabela C salvando os corners
3:     para i = 0 até S.size()
4:         corners[i] = C[ S[i] ];
5:     fim para
6: senão
7:     n_t = número de triângulo na malha corrente
8:     para i = 0 até S.size()
9:         para j = 0 até 3n_t
10:            //Verifica se o vértice do corner j é igual ao vértice procurado
11:            se V[j] == S[i] então
12:                corners[i] = j;
13:                break;
14:            fim se
15:        fim para
16:    fim para
17: fim se
18: return corners;
```

Note que algumas aplicações requerem que a entrada seja um conjunto de faces. Desta forma o algoritmo para obter os *corners* dos vértices é mais simples e não requer a tabela C (ver Algoritmo 14).

Apesar de mais lenta em buscas topológicas que tem como entrada os vértices, a implementação da *Corner Table* sem a tabela C é indicada para aplicações que

PUC-Rio - Certificação Digital Nº 1112650/CA

Algoritmo 14 obtemCornersDosVertices

Entrada: F: conjunto de faces
Saída: corners[]: vetor com *corners* dos vértices da faces de entrada

```
1: //Salva corners dos vértices das faces
2: para i = 0 até F.size()
3:     corners[3 * i] = 3 * F[i];
4:     corners[3 * i + 1] = 3 * F[i] + 1;
5:     corners[3 * i + 2] = 3 * F[i] + 2;
6: fim para
```

tenham limitação no uso de memória física ou aquelas que as regiões a serem subdivididas não tenham como entrada um conjunto muito grande de vértices.

Este é o único passo onde a tabela C se faz necessária para aumentar a eficiência na realização da subdivisão incremental na *Corner Table*. Nos próximos passos usaremos apenas as tabelas V e O.

5.3.2
Cálculo da r-vizinhança

O segundo passo para realizar a subdivisão incremental é fazer a expansão do conjunto S para um novo conjunto $E^r(S)$. Esse novo conjunto possui todos os vértices de S e mais aqueles que distam ao menos r arestas de algum vértice de S. Esta expansão é basicamente uma busca em largura a partir do conjunto S que deve ser interrompida quando alcançar vértices que distam mais que r arestas de S.

O Algoritmo 15 calcula a expansão de S, gerando um novo conjunto $E^r(S)$ que por sua vez tem um *corner* para cada vértice da r-vizinhança. O Algoritmo 15 usa o Algoritmo 17 para fazer a expansão a partir de um determinado vértice, incluindo seus vértices vizinhos no conjunto $E^r(S)$. Por sua vez, o Algoritmo 17 usa o Algoritmo 16, para que a partir de um *corner* c se obtenha um vetor com um *corner* para cada vértice na estrela do vértice $V[c]$.

O Algoritmo 15 realiza a expansão do conjunto S para um novo conjunto $E^r(S)$ usando uma fila e um vetor de inteiros como estruturas auxiliares. A fila é usada para armazenar os corners dos próximos vértices a serem expandidos na busca em largura, já o vetor de inteiros é usado para armazenar a menor distância ao conjunto S para cada vértice. À medida que os vértices são alcançados as suas distâncias são atualizadas e seus vizinhos, se não estiverem a uma distância para S maior que r, são adicionados na fila. Note que por se tratar de uma busca em largura, no primeiro momento em que um vértice fora do conjunto S é alcançado, a distância corrente já é a sua menor distância.

O Algoritmo 16 obtém um *corner* para cada vértice na estrela de um determinado vértice. A partir de um *corner*, o algoritmo gira em um sentido na

PUC-Rio - Certificação Digital Nº 1112650/CA

Algoritmo 15 obtemCornersNaVizinhancaR

Entrada: *corners[]:* vetor com um corner para cada vértice do conjunto S.
Entrada: *r:* distância máxima em arestas que um vértice v deve está de S para que v seja incluido em $E^r(S)$.
Saída: $E^r(S)$

```
1:  //Fila usada na busca em largura com os corners dos vértices a serem visitados
2:  //na expansão.
3:  Fila filaDeBusca;
4:
5:  //inicializa o vetor distancia com ∞ para todos os vértices.
6:  //Este vetor é responsável por guardar a distancia, em arestas,
7:  //de cada vértice ao conjunto S.
8:  numVertices = número de vértices da malha corrente
9:  para (i = 0 até numVertices)
10:     distancia[i] = ∞;
11: fim para
12:
13: para i = 0 até corners.size()
14:     //os corners dos vértices do conjunto S são adicionados na fila. Os vértices
15:     //entram para o conjunto E^r(S) e são marcados com distância 0, uma
16:     //vez que pertencem a S.
17:     filaDeBusca.insere( corners[i] );
18:     vertice = V[ corners[i] ];
19:     E^r(S) = E^r(S) ∪ corners[i];
20:     distance[vertice] = 0;
21:
22:     //Realiza a expansão para o vértice V[corner]
23:     enquanto !filaDeBusca.vazia()
24:         corner = filaDeBusca.remove();
25:         //chama o Algoritmo 17 para fazer a expansão do vértice V[corner],
26:         //atualizar o vetor de distância e inserir novos vértices na fila
27:         //caso algum entre para E^r(S)
28:         expandeVertice( corner, distancia, filaDeBusca);
29:     fim enquanto
30: fim para
31:
32: return E^r(S)
```

estrela de um determinado vértice obtendo os *corners* dos vértices vizinhos. O algoritmo pode se deparar com o bordo da superfície, neste caso o algoritmo retoma a partir do *corner* inicial e gira em outro sentido, novamente obtendo os *corners* dos vértices vizinhos até se deparar mais uma vez com o bordo da superfície.

O Algoritmo 17 realiza a expansão de um determinado vértice v atualizando os vetores de distância para os vizinhos de v e a fila usada para fazer a busca em largura. O algoritmo recebe um *corner* c, consulta os vértices na estrela de $V[c]$ e atualiza a distância destes vértices caso o caminho até S seja menor passando por

PUC-Rio - Certificação Digital Nº 1112650/CA

Algoritmo 16 obtemVerticesVizinhos

Entrada: c: corner do vértice o qual se deseja encontrar sua estrela

Saída: vizinhos[]: vetor com um corner para cada vértice vizinho ao vértice de corner c

```
1:  cornerInicial = next( c );
2:  cornerCorrente = cornerInicial;
3:
4:  //gira na estrela do vértice salvando os corners até voltar ao início ou encontrar
    uma borda
5:  repetir
6:      vizinhos.push( cornerCorrente );
7:      cornerCorrrente = right( cornerCorrente );
8:  até que (cornerCorrente == cornerInicial || cornerCorrente == -1)
9:
10: //Verifica se o laço anterior parou por voltar ao corner inicial.
11: Se sim retorna o vetor com a lista de corners vizinhos
12: se  cornerCorrente ≠ -1 então
13:     return vizinhos;
14: fim se
15:
16: //Como encontrou a borda, gira para o outro lado, a partir do corner inicial,
17: //até encontrar a borda novamente
18: cornerCorrrente = previous( c );
19: repetir
20:     vizinhos.push( cornerCorrente );
21:     cornerCorrrente = left( cornerCorrente );
22: até que ( cornerCorrente == -1 )
23:
24: return vizinhos;
```

$V[c]$. Neste caso, os *corners* destes vértices são adicionados na fila, para que seus vizinhos também sejam consultados posteriormente. Além disso, um *corner* para cada um destes vértices é adicionado ao conjunto $E^r(S)$.

Note que o vetor distância funciona como uma espécie de *cache* das buscas dos outros vértices já realizadas, uma vez que as operações da linha 19 a linha 27 do Algoritmo 17 só são executadas para um vértice se for descoberto um caminho mais curto para ele, evitando assim que estas operações sejam executadas várias vezes para o mesmo vértice.

PUC-Rio - Certificação Digital Nº 1112650/CA

Algoritmo 17 expandeVertice

Entrada: c: corner do vértice a ser expandido.
Entrada: distancia[]: informa a distância de cada vértice ao conjunto S. Esse vetor será atualizado na função.
Entrada: filaDeBusca: Fila com os corners dos vértices a serem visitados na expansão. Ela será usada apenas para adicionar o corner de um novo vértice.

```
1: //Obtem os corners dos vértices vizinhos ao vértice V[c] (Algoritmo 16)
2: vizinhos[] = obtemVerticesVizinhos( c );
3:
4: //Obtem a distância do vértice V[ c] ao conjunto S
5: distanciaCorrente = distancia[ V[ c ] ];
6:
7: para Para i = 0 até vizinhos.size()
8:     //obtém o vértice do corner
9:     vertice = V[ vizinhos[i] ];
10:
11:    //Verifica se o vértice vizinho ao vértice V[c] tem uma distância maior
12:    //do que a distância passando por V[c]. Se sim, foi encontrado
13:    //um caminho menor para o vértice
14:    se (distancia[vertice] > distanciaCorrente + 1) então
15:        //verifica se o vértice que está sendo visitado foi alcançado
16:        //dentro da região de expansão
17:        se distanciaCorrente + 1 <= r então
18:            //adiciona o vértice no conjunto E^r(S)
19:            E^r(S) = E^r(S) ∪ vizinhos[i];
20:
21:            //atualiza a distancia do vértice
22:            distancia[vertice] = distanciaCorrente + 1;
23:
24:            //Uma vez que foi encontrado um caminho mais curto para o vértice
25:            /corrente, um dos seus corners é adicionado na fila para que uma
26:            //nova busca a partir dele seja realizada.
27:            filaDeBusca.insere( vizinho[i] );
28:        fim se
29:    fim se
30: fim para
```

5.3.3 Calculando as faces a partir de $E^r(S)$

Uma vez que o operador de subdivisão *1 para 4* é aplicado sobre as faces da malha triangular, um terceiro passo é necessário para calcular quais as faces são formadas com os vértices do conjunto $E^r(S)$.

O Algoritmo 18 recebe o conjunto $E^r(S)$ e retorna as faces formadas por seus vértices.

PUC-Rio - Certificação Digital Nº 1112650/CA

Algoritmo 18 obtemFacesFormadasPorErS

Entrada: $E^r(S)$: conjunto dos vértices da região a ser subdividida.
Saída: faces[]: conjunto das faces a serem subdivididas

```
1:  para todo corner c ∈ E^r(S)
2:      corrente = next( c );
3:      inicial = corrente;
4:
5:      //Gira na estrela do de cada vértice V[c] verificando se seus vizinhos
6:      //também pertencem ao conjunto E^r(S)
7:      repetir
8:          se (V[next( corrente )] ∈ E^r(S) && V[prev(corrente)] ∈ E^r(S)) então
9:              faces.insert( corrente / 3 );
10:         fim se
11:         corrente = right( corrente );
12:     até que (corrente == -1 || corrente == inicial)
13:
14:     //Se a busca parou por voltar ao início pula para o próximo corner
15:     se (corrente ≠ -1) então
16:         continue;
17:     fim se
18:
19:     //Se a busca parou por chegar a uma borda, gira no outro sentido
20:     //executando as mesmas operações
21:     corrente = prev( c );
22:     repetir
23:         se (V[next( corrente )] ∈ E^r(S) && V[prev(corrente)] ∈ E^r(S)) então
24:             faces.insert( corrente / 3 );
25:         fim se
26:         corrente = left( corrente );
27:     até que (corrente == -1)
28: fim para
29:
30: return faces;
```

O Algoritmo 18 verifica para cada vértice dos *corners* em $E^r(S)$ se seus vértices vizinhos, dentro da mesma face, também pertecem a $E^r(S)$. Para cada *corner* $c \in E^r(S)$, o algoritmo gira na estrela do vértice $V[c]$ verificando se *next(c)* e *prev(c)* também pertecem a $E^r(S)$ (ver Figura 5.7), em caso a afirmativo a face $f = \lfloor c \div 3 \rfloor$ é adicionada no conjunto das faces que serão subdivididas posteriormente. O algoritmo pode se de deparar com o bordo da superfície. Nesse caso, o algoritmo retoma a busca a partir do *corner* inicial, mas girando no sentido oposto desta vez.

PUC-Rio - Certificação Digital Nº 1112650/CA

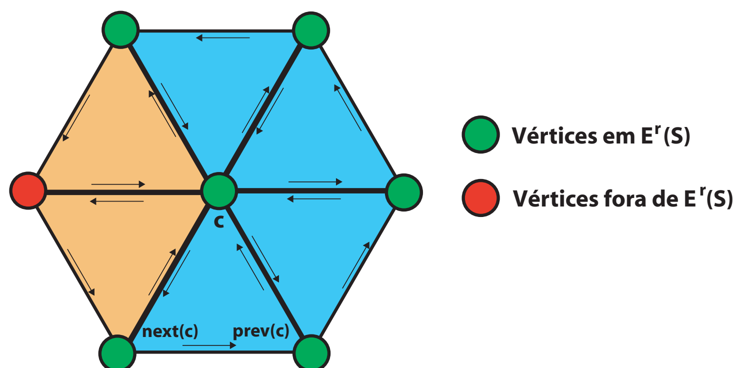


Figura 5.7: Ilustração do funcionamento do Algoritmo 18. As faces em azul são aquelas que serão subdivididas posteriormente.

5.3.4
Subdivisão da região expandida

Com o conjunto das faces a serem subdivididas calculado, o último passo da subdivisão adaptativa pode ser realizado. Neste passo as faces calculadas são subdivididas e para possibilitar vários níveis de subdivisão da mesma região, o novo conjunto S' é retornado. Lembrando que S' é o conjunto de vértices formado por S mais os vértices criados sobre as arestas das faces formadas por S (ver Figura 2.14).

O algoritmo para subdivisão adaptativa das faces selecionadas é bastante semelhante ao algoritmo de subdivisão global, no entanto como nem todas as faces são subdivididas, uma estrutura auxiliar é necessária para informar se uma face é subdividida ou não. Esta estrutura pode ser um vetor de boleano de tamanho $4n_t$, onde n_t é o número de triângulos da malha corrente, que informa em tempo constante se uma face é subdividida ou não, ou até mesmo uma árvore balanceada, onde os nós são os índices das faces a serem subdivididas. Note que ao subdividir uma face, a nova face gerada pela subdivisão deve ser colocada na árvore também. Desta forma, ao fim da subdivisão a árvore terá $4n$ nós.

Se o número de faces a serem subdivididas for pequeno em relação ao número total de faces da superfície, a implementação usando uma árvore balanceada pode usar um montante de memória menor, no entanto o custo para determinar se uma face deve ser subdividida ou não é de $log(n)$, onde n é o número de faces a serem subdivididas.

O vetor de booleano pode usar um bit por face, já a árvore usa um inteiro de 32 bits por nó, logo se o fator determinante para escolha da estrutura auxiliar for o montante de memória usado durante a subdivisão, a árvore deve ser escolhida sempre que

$$32(4n) < 4n_t \tag{5-1}$$

PUC-Rio - Certificação Digital Nº 1112650/CA

Algoritmo 19 runSplit

Entrada: *corner:* corner onde o operador de Edge Split deve ser aplicado.
Entrada: *subdivide:* vetor para determinar se uma face deve ser subdividida.
Entrada: *facesVisitadas:* vetor que diz se uma face já foi visitada.
Entrada: *facesProcessadas:* vetor que diz se uma face já foi processada.
Entrada: *Q:* fila com os corners dos próximos vértices a serem processados e as arestas as quais o operador de *Edge Flip* deve ser aplicado após cada vértice ser processado.
Entrada: S*:* conjunto de vértices da região selecionada.
Entrada: S'*:* conjunto S' que deve ser atualizada.

1: size = número de triângulos da malha corrente
2: n_t = número de vértices da malha corrente
3:
4: *//Verifica se o novo vértice será criado sobre uma aresta da região selecionada.*
5: *//Se sim adiciona o índice do vértice vértice ao conjunto* S'
6: **se** (V[next(corner)] $\in S$ && V[prev(corner)] $\in S$) **então**
7: $S' = S' \cup n_t$;
8: **fim se**
9:
10: facesProcessadas[corner / 3] = true;
11: facesProcessadas[size] = true;
12: opposite = O[corner];
13:
14: *//Aplica o operador de* Edge Split
15: **EdgeSplit(corner**);
16:
17: *//Salva um corner do vértice gerado para ser inserido na fila*
18: novoVertice = next(corner);
19: arestaParaFlip = -1;
20: **se** (opposite $\neq$ -1) **então**
21: *//Salva corner oposto a aresta que deve ser "flipada".*
22: **se** (subdivide[opposite / 3] && !facesVisitadas[opposite / 3]) **então**
23: arestaParaFlip = prev(opposite);
24: **fim se**
25: *//Se a face oposta já foi visita, então marca a face oposta e nova face*
26: *//gerada a partir dela como processadas*
27: **se** (facesVisitadas[opposite / 3]) **então**
28: facesProcessadas[opposite / 3] = true;
29: facesProcessadas[size + 1] = true;
30: **fim se**
31: *//Marca a face oposta e a face gerada a partir dela como visitadas*
32: ...
33: **fim se**
34: *//Insere o par* (v, e) *na fila*
35: Q.insere((novoVertice, arestaParaFlip)) ;

PUC-Rio - Certificação Digital Nº 1112650/CA

caso contrário, o vetor de booleano deve ser escolhido.

Algoritmo 20 SubdivideFaces

Entrada: *faces*: conjunto das faces a serem subdivididas.
Entrada: S: conjunto dos vértices da região selecionada.

```
1: //Declara as estruturas usadas pela subdivisão global
2: n = número de triângulos da malha inicial;
3: Fila Q; bool facesVisitadas[4 * nT] = false, facesProcessadas[4 * nT] = false;
4: bool subdivide[4*nT] = false;
5: ...
6: //define quais faces devem ser subdivididas
7: para todo (i = 0 até faces.size())
8:     subdivide[i] = true;
9: fim para
10:
11: //inicializa S' com o conjunto S
12: S' = S;
13:
14: //realiza a operação de split na aresta oposto ao primeiro corner
15: //do primeiro triangulo da lista de faces
16: runSplit( 3 * faces[0], facesVisitadas, facesProcessadas, Q, S' );
17: enquanto (!Q.vazia( ))
18:     ...
19:     //percorre a estrela do vértice realizando a operação
20:     //de split nas arestas opostas
21:     repetir
22:         //realiza o split apenas se a face tiver um nível inferior a 2.
23:         se ( !facesProcessadas[corrente / 3] && subdivide[corrente / 3]) então
24:             runSplit( corrente, facesVisitadas, facesProcessadas, Q, S' );
25:         fim se
26:         aux = right( operation );
27:         corrente = next( aux );
28:     até que (inicial == corrente || aux == -1)
29:     ...
30:     repetir
31:         se ( !facesProcessadas[corrente / 3] && subdivide[corrente / 3]) então
32:             runSplit( corrente, facesVisitadas, facesProcessadas, Q, S' );
33:         fim se
34:         aux = left( operation );
35:         corrente = prev( aux );
36:     até que (aux == -1)
37:     se (e ≠ - 1) então
38:         EdgeFlip( e );
39:     fim se
40: fim enquanto
41: EdgeFlip( prev( 3 * faces[0] ) );
42:
43: return S'
```

PUC-Rio - Certificação Digital Nº 1112650/CA

Os Algoritmos 19 e 20 implementam a subdivisão das faces selecionadas usando um vetor de booleano para determinar quais faces devem ser subdivididas. Pela semelhança com os algoritmos 11 e 10, algumas linhas serão omitidas. Além disso, o Algoritmo 20 retorna o conjunto S'. É importante retornar esse conjunto de vértices, pois é a partir dele que a subdivisão incremental deve ser aplicada para se produzir vários níveis de subdivisão em uma mesma região da superfície.

5.4 Undo e Redo para Subdivisão

Para algumas aplicações, como por exemplo modelagem, é importante fornecer uma forma de desfazer ou refazer um conjunto de passos. Por isso, provemos um mecanismo para realizar *undo* e *redo* para as operações de subdivisão.

Uma forma de realizar *undo* e *redo* nas operações de uma superfície, seria amarzenar uma superfície a cada modificação sofrida pela superfície corrente. No entanto, isto seria muito ineficiente no que diz respeito ao uso de memória. Uma forma eficiente seria pela aplicação das operações inversas às operações que produziram o estado corrente da superfície.

Como mostrado na seção 5.1, as subdivisões global e adaptativa são realizadas através da aplicação dos operadores estelares *Edge Split* e *Edge Flip* e no capítulo 4 foi visto que o operador inverso ao *Edge Split* é o operador de *Edge Weld*, e o operador inverso ao *Edge Flip* é o próprio *Edge Flip*. Ainda no capítulo 4 foi mostrada a necessidade de haver uma implementação para reverter o operador de *Edge Flip*, que chamamos de *Edge Flip Inverso*. Baseado nisso e nas convenções que assumimos para implementar as operações, as figuras 5.8 e 5.9 ilustram a aplicação dos operadores inversos usados na subdivisão.

Como ilustrado na Figura 5.8, para reverter o efeito provocado pelo operador *Edge Split* aplicado em um *corner* c_0, basta aplicar o operador *Edge Weld* a partir do *corner next(*c_0*)* e para reverter o efeito provocado pelo operador *Edge Weld* aplicado em um *corner* c_1, basta aplicar o operador *Edge Split* a partir do *corner prev(*c_1*)*. Já para reverter o efeito provocado pelo operador *Edge Flip* aplicado a partir de um *corner* c_0, basta aplicar o *Edge Flip* Inverso a partir do mesmo *corner* $next(c_0)$ e para reverter o efeito provocado pela aplicação do *Edge Flip* Inverso a partir de um *corner* c_1, basta aplicar o operador de *Edge Flip* a partir do mesmo *corner* $prev(c_0)$.

O operador de *Edge Weld* remove um vértice e dois triângulos da malha, se o vértice removido não for do bordo da superfície, mas é desejável que a operação de *undo* matenha sempre as tabelas V e G contíguas, ou seja, não remova triângulos ou vértices no meio destas tabelas, de forma que se a malha tiver n triângulos, estes estão gravados nas $3n$ primeiras posições da tabela V e da mesma forma a tabela G. Note que esta condição é automaticamente satisfeita, uma vez que ao aplicar o

PUC-Rio - Certificação Digital Nº 1112650/CA

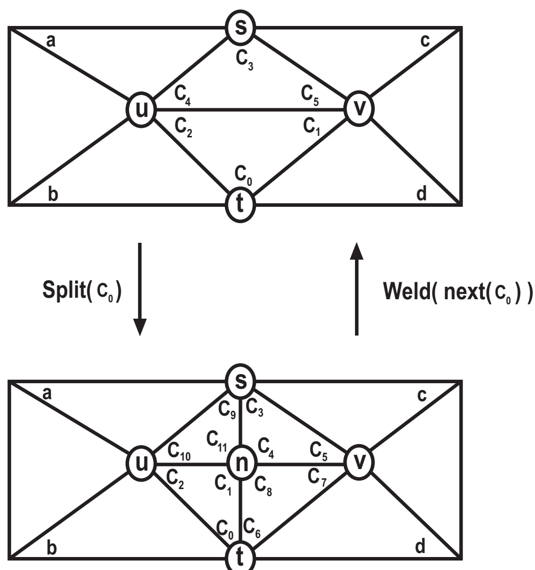


Figura 5.8: Inversão do operador de *Edge Split* pela aplicação do operador de *Edge Weld* e vice-versa.

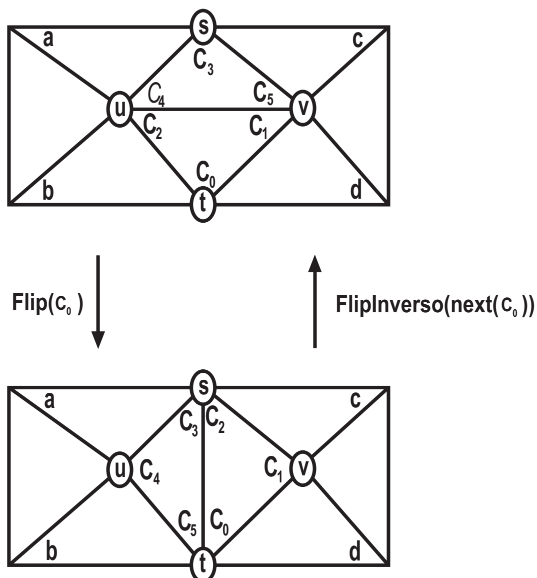


Figura 5.9: Invrsão do operador de *Edge Flip* pela aplicação do *Edge Flip* Inverso e vice-versa.

operador de *Edge Split*, este sempre posiciona os novos triângulos e o novo vértice ao final das tabelas V e G respectivamente, logo ao reverter uma sequência de passos o operador de *Edge Weld* sempre removerá vértices e triângulos do final de suas tabelas, mantendo assim as tabelas V e G sempre contíguas. Isso evita um passo extra para rearranjar estes vetores e nenhum montante de memória adicional para execução do mecanismo de *undo/redo* é requerido.

PUC-Rio - Certificação Digital Nº 1112650/CA

Operação	**Operação Inversa**
EdgeSplit(c)	EdgeWeld(next(c))
EdgeWeld(c)	EdgeSplit(previous(c))
EdgeFlip(c)	EdgeFlipInverso($next(c)$)
EdgeFlipInverso(c)	EdgeFlip($prev(c)$)

Tabela 5.1: Operações Inversas

A tabela 5.1 mostra as operações inversas que devem ser empilhadas para executar o mecanismo de *undo/redo*.

PUC-Rio - Certificação Digital Nº 1112650/CA

PUC-Rio - Certificação Digital Nº 1112650/CA

6
Resultados

Neste capítulo apresentamos alguns resultados da subdivisão global e adaptativa. Para avalizar a eficiência foram feitas medições de tempo para os dois algoritmos. Ambos também foram aplicados em alguns modelos para avaliar a malha resultante e qualidade de seus triângulos.

6.1
Subdivisão Global

Para avaliar a eficiência do algoritmo de subdivisão global, aplicamos recursivamente o algoritmo de subdivisão em uma malha triangular de uma superfície plana com bordo. Subdividimos recursivamente um triângulo onde os novos vértices criados pelo processo de subdivisão são sempre posicionados sobre o ponto médio das arestas. A Tabela 6.1 mostra o número de faces antes e depois de cada subdivisão o tempo de execução.

Número de faces antes da subdivisão	**Número de faces após a subdivisão**	**Tempo (segundos*)**
1.024	4.096	< 0.01
4.096	16.384	< 0.01
16.384	65.536	< 0.01
65.536	262.144	0.02
262.144	1.048.576	0.08
1.048.576	4.194.304	0.38
4.194.304	16.777.216	1.76
16.777.216	67.108.864	5.74

Tabela 6.1: Tempo para realizar a subdivisão global. *Core i5 3.10Ghz, 8GB RAM

Como segundo exemplo executamos uma subdivisão global sobre um icosaedro regular para gerar a malha de uma esfera (Figura 6.1). O icosaedro regular está mostrado na Figura 6.1(a). Nas Figuras de 6.1(b)-6.1(e) cada novo vértice gerado é posicionado sobre a superfície da esfera interpolante deste icosaedro. A Figura 6.1(f) mostra o icosaedro de 6.1(a) com o mesmo nível de subdivisão de 6.1(e), mas com os novos vértices posicionados usando *Butterfly*.

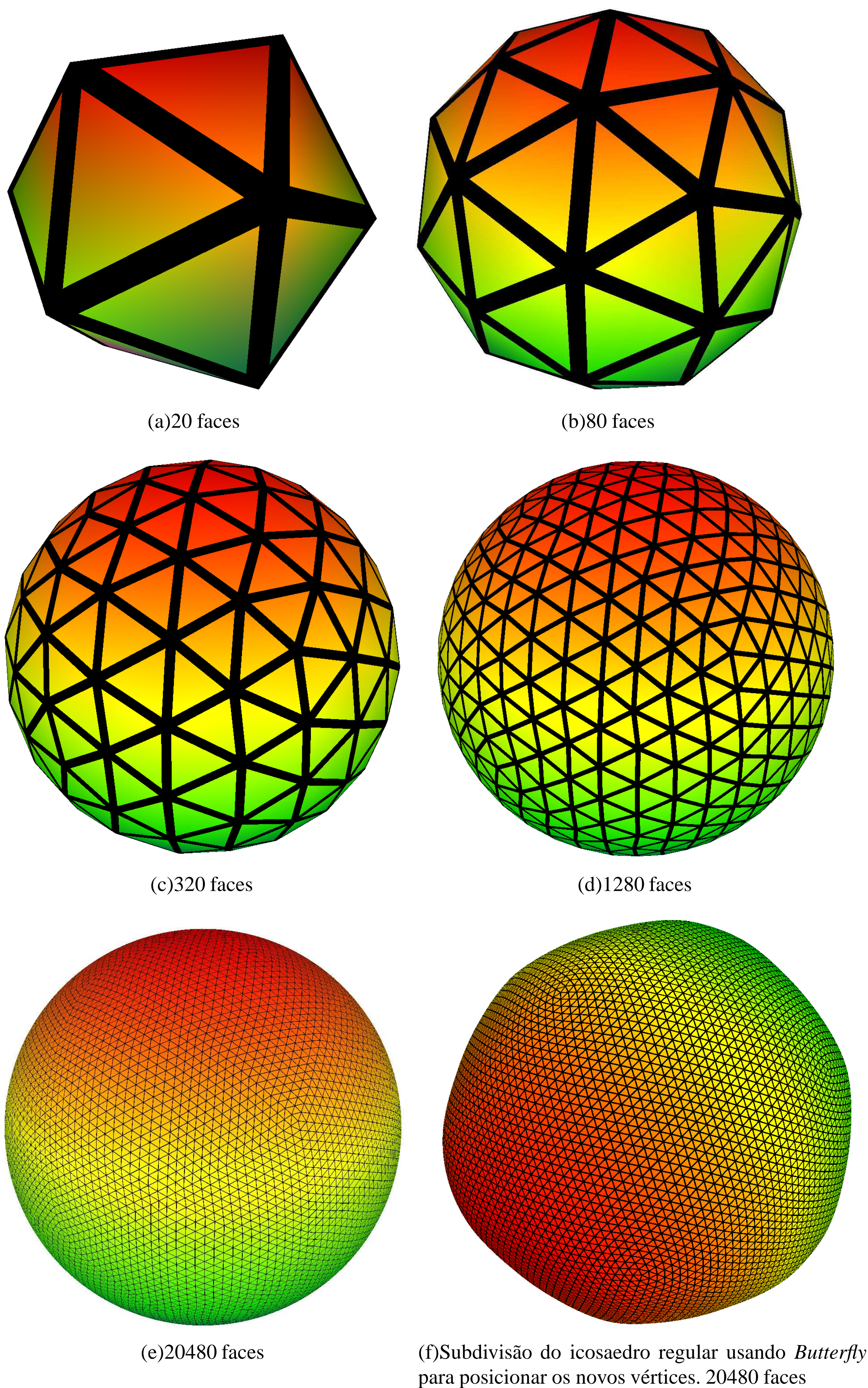
Na Figura 6.2, a subvisão global é usada para aproximar uma malha plana da função $z = sin(\pi x)cosy + 0.1sin(2\pi x)$. Em todos os outros exemplos de

subdivisão global, o esquema *Butterfly* foi utilizado para posicionar os novos vértices.

A Figura 6.3, 6.4, 6.5, 6.6(a), 6.7 ilustram a subdivisão de um tetraedro, de uma estrela de 6 pontas, e dos modelos *cow*, *bunny* e *bimba* respectivamente.

Apesar do *Butterfly* modificado poder ser aplicado em uma malha de topologia arbitrária, o método assume uma certa regularidade nos triângulos da malha. De forma que a presença de triângulos muito finos causam efeitos indesejados, como ilustra a Figura 6.7, onde o triângulo fino destacado 6.7(a) causa uma dobra na superfície.

PUC-Rio - Certificação Digital Nº 1112650/CA



(a)20 faces

(b)80 faces

(c)320 faces

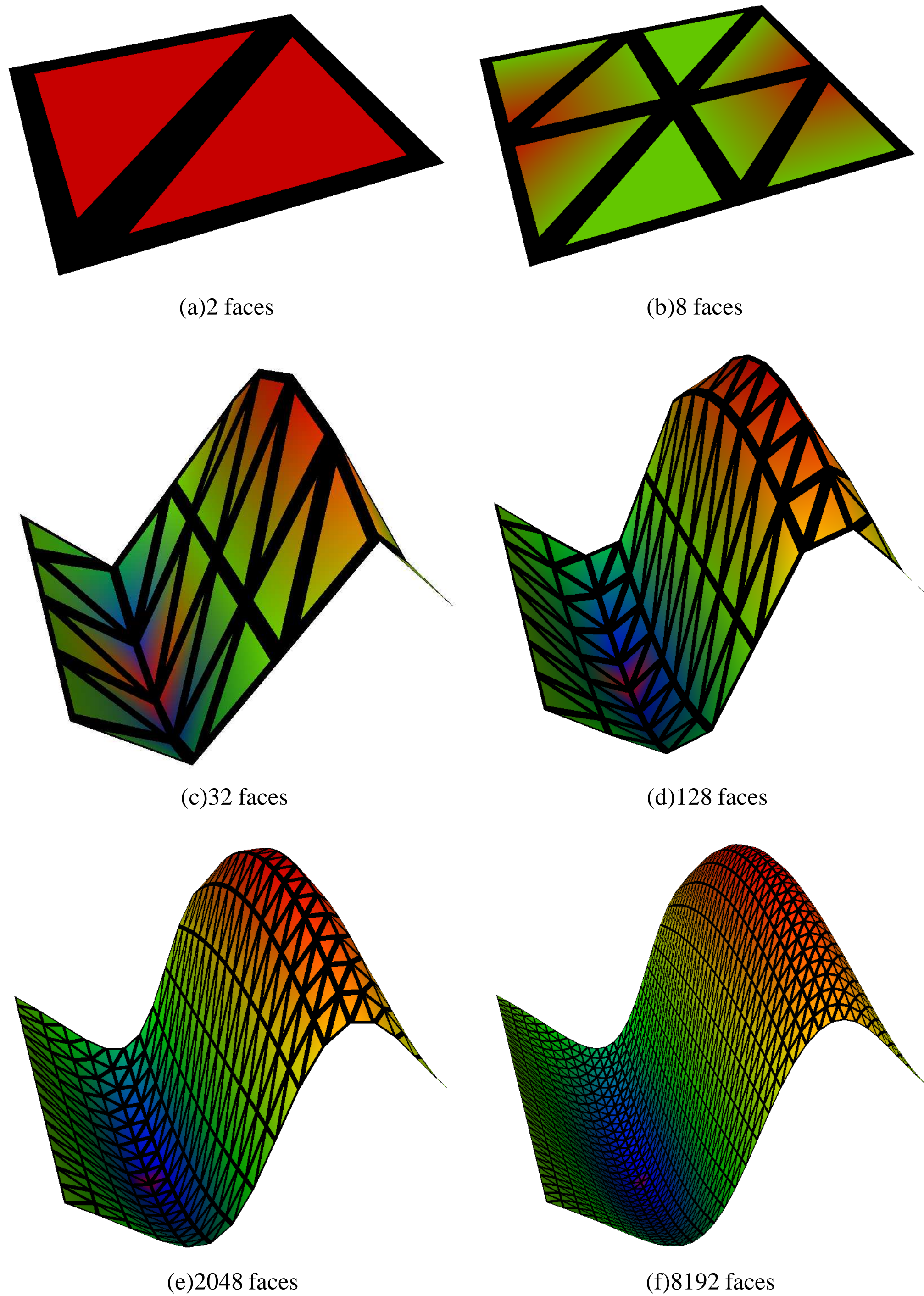
(d)1280 faces

(e)20480 faces

(f)Subdivisão do icosaedro regular usando *Butterfly* para posicionar os novos vértices. 20480 faces

Figura 6.1: Geração da malha de uma esfera através da subdivisão de um icosaedro regular

PUC-Rio - Certificação Digital Nº 1112650/CA



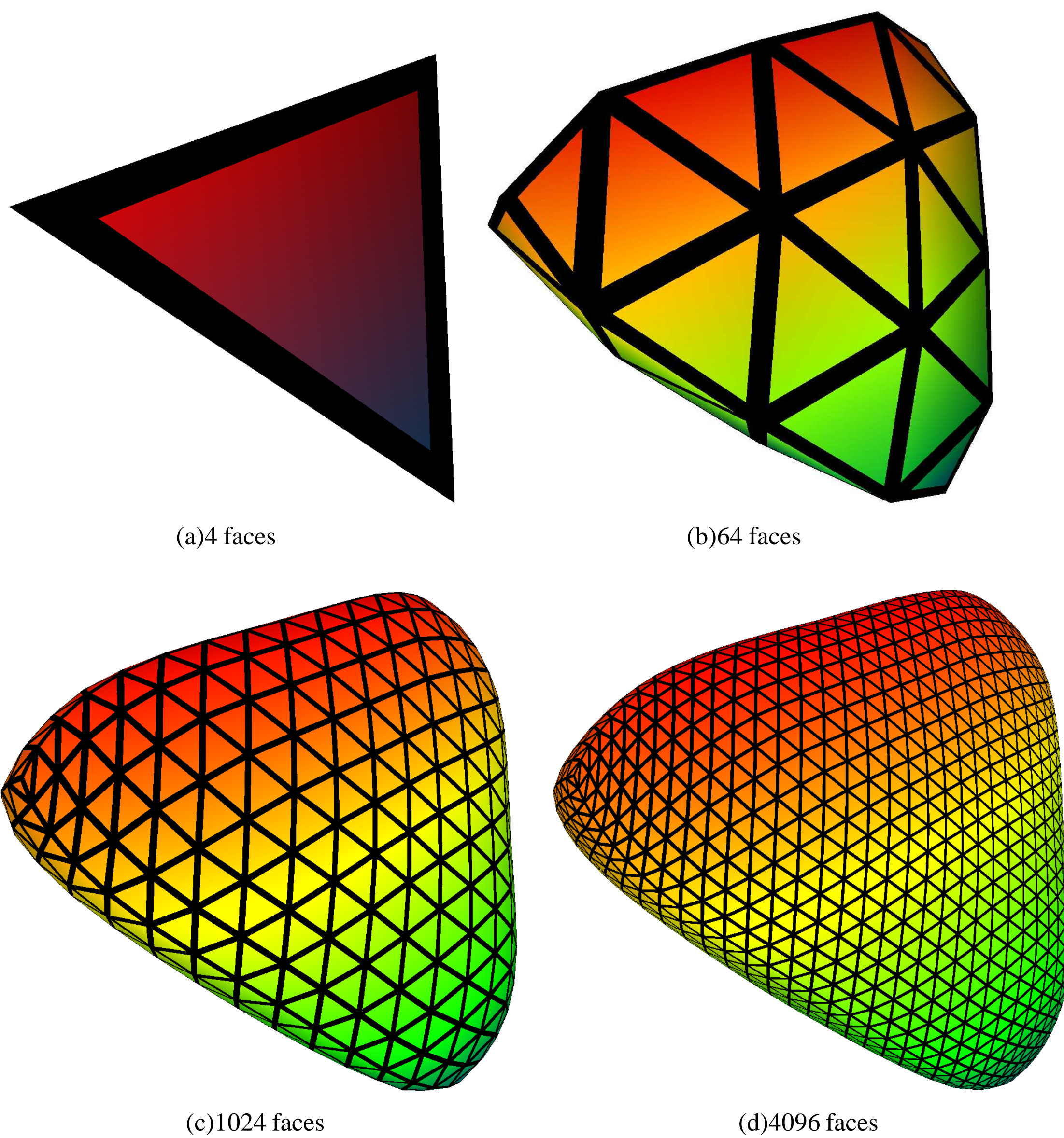
(a)2 faces (b)8 faces

(c)32 faces (d)128 faces

(e)2048 faces (f)8192 faces

Figura 6.2: Subdivisão de uma malha com dois triângulos para aproximar a função $z = sin(\pi x)cosy + 0.1sin(2\pi x)$

PUC-Rio - Certificação Digital Nº 1112650/CA



(a)4 faces

(b)64 faces

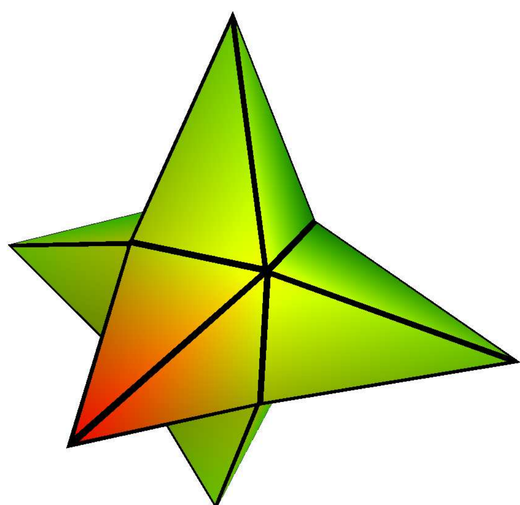
(c)1024 faces

(d)4096 faces

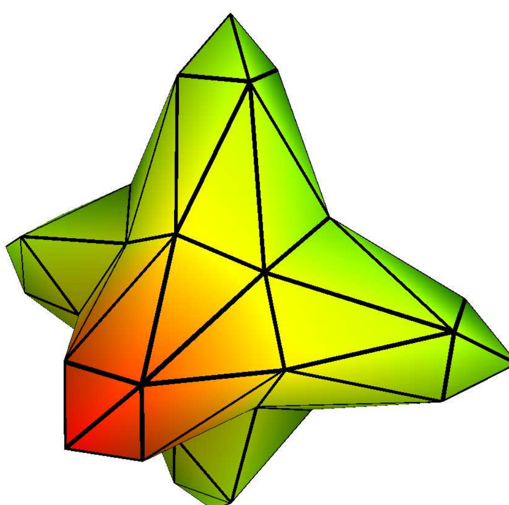
Figura 6.3: Subdivisão de uma um tetraedro regular usando *Butterfly*

PUC-Rio - Certificação Digital Nº 1112650/CA

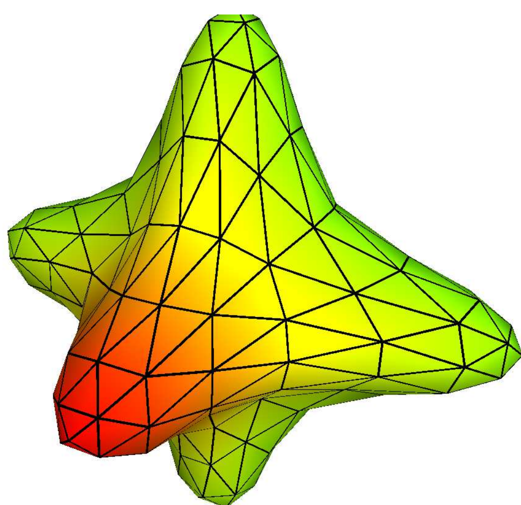
PUC-Rio - Certificação Digital Nº 1112650/CA



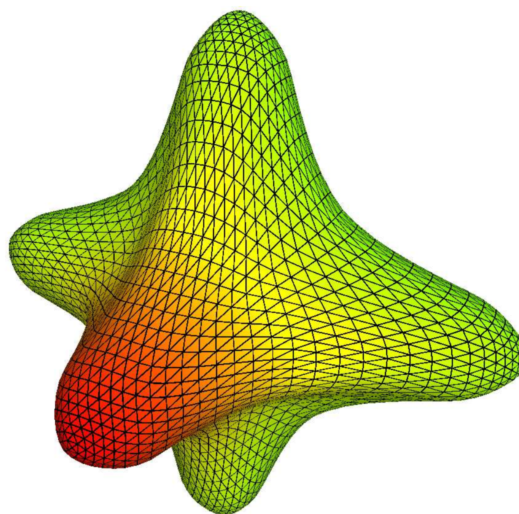
(a)24 faces



(b)96 faces

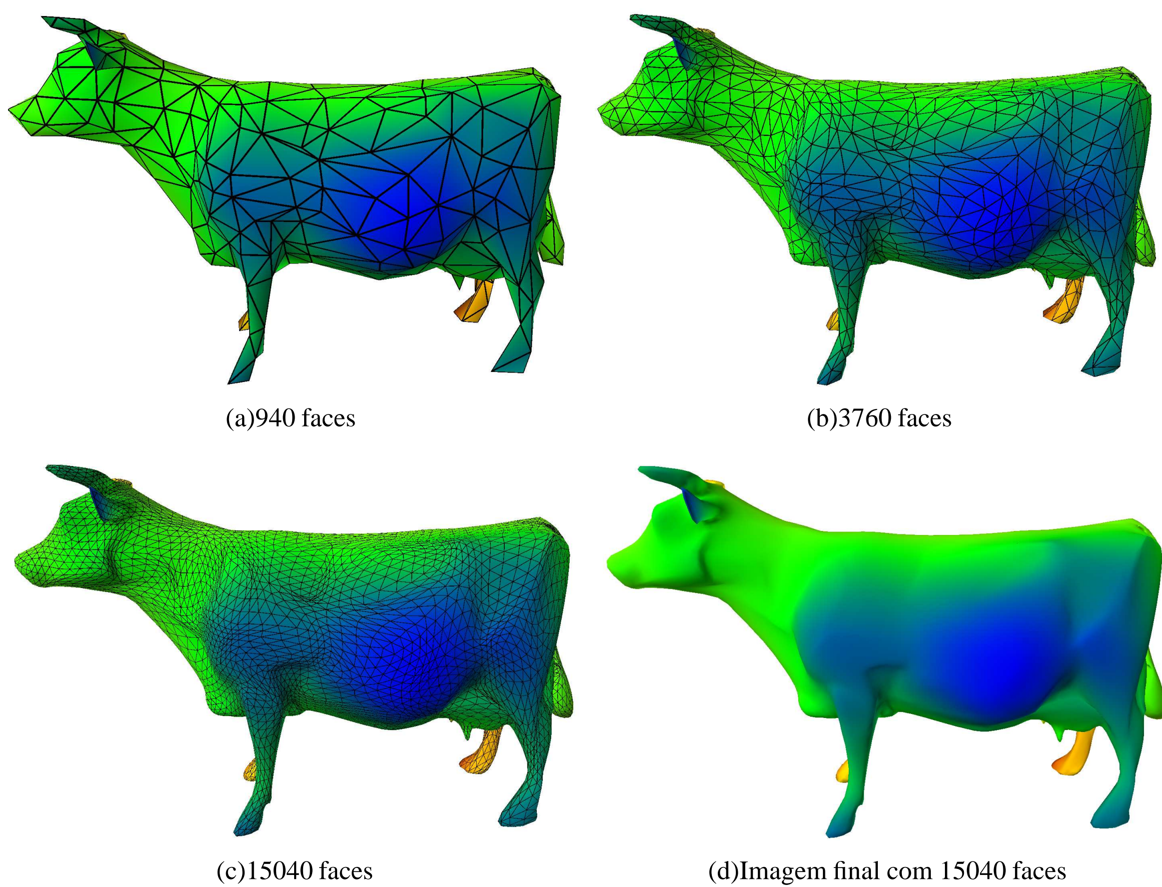


(c)384 faces



(d)6144 faces

Figura 6.4: Subdivisão de uma estrela de 6 pontas usando Butterfly



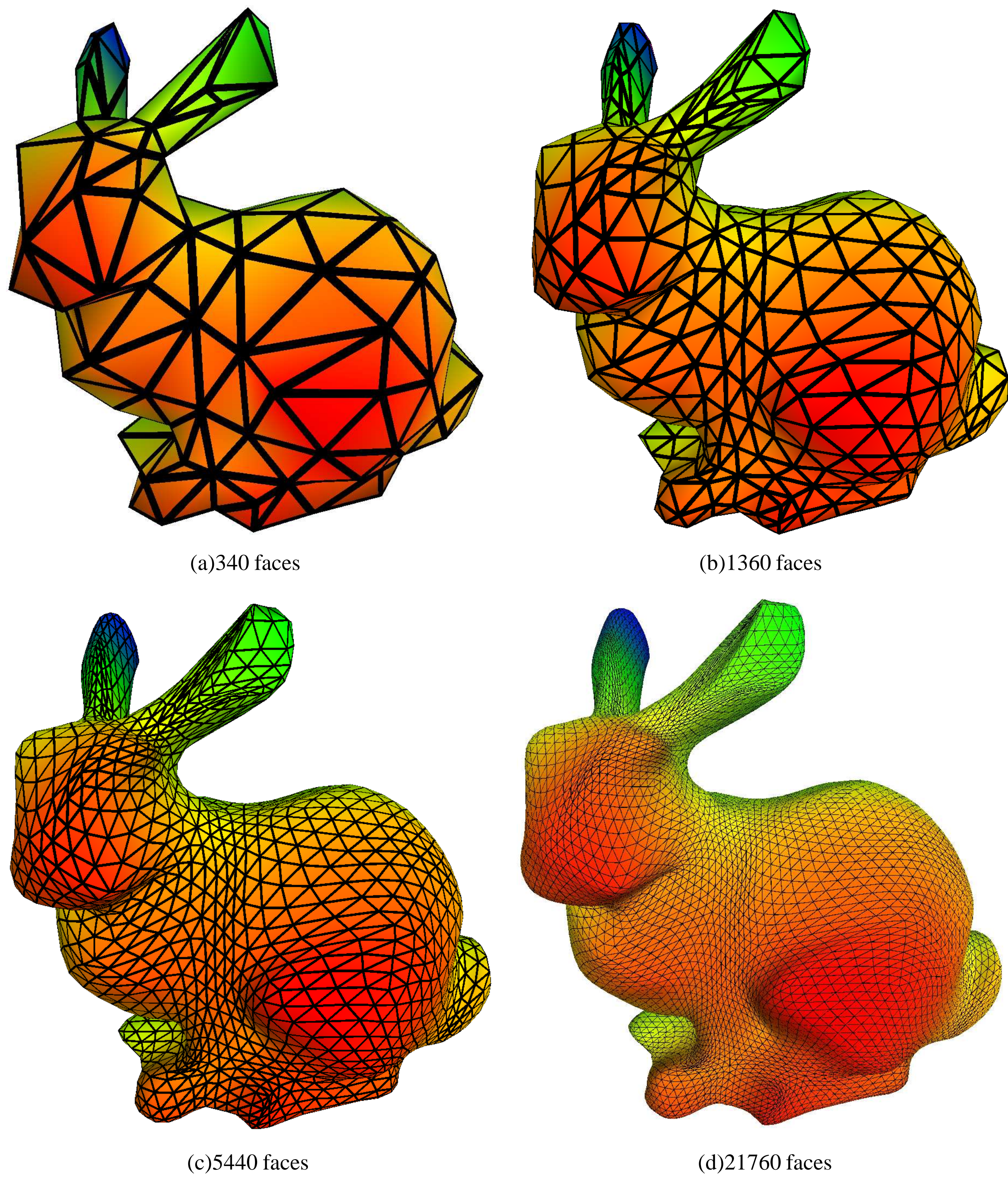
(a)940 faces (b)3760 faces

(c)15040 faces (d)Imagem final com 15040 faces

Figura 6.5: Subdivisão do modelo cow usando Butterfly.

PUC-Rio - Certificação Digital Nº 1112650/CA

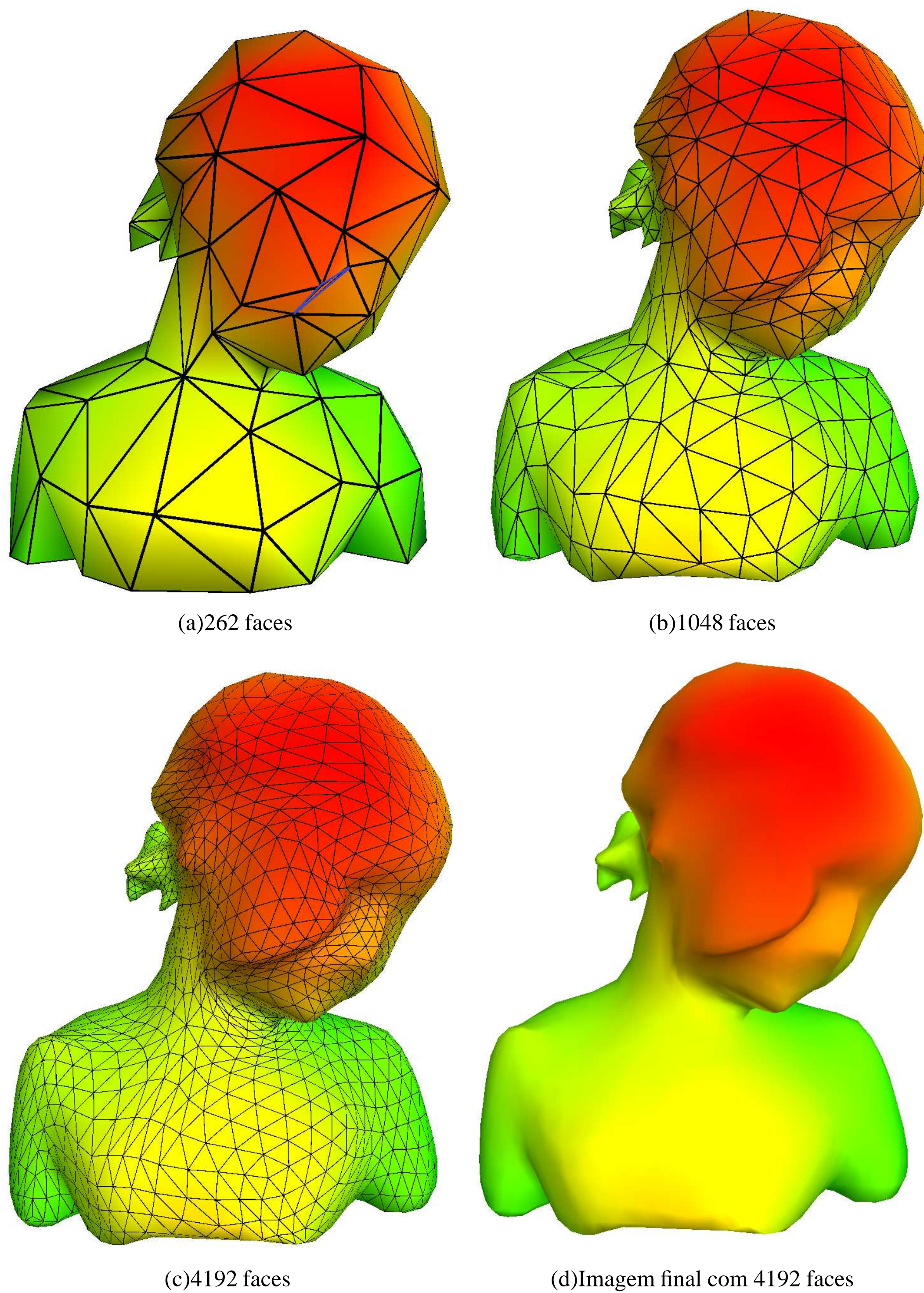
PUC-Rio - Certificação Digital Nº 1112650/CA



(a)340 faces (b)1360 faces

(c)5440 faces (d)21760 faces

Figura 6.6: Quatro níveis de subdivisão do modelo bunny usando Butterfly.



(a)262 faces (b)1048 faces

(c)4192 faces (d)Imagem final com 4192 faces

Figura 6.7: Subdivisão do modelo bimba usando Butterfly. O triângulo muito fino na em 6.7(a) provoca um efeito indesejado na subdivisão

PUC-Rio - Certificação Digital Nº 1112650/CA

6.2 Subdivisão adaptativa

Como na subdivisão global, avaliamos a eficiência do algoritmo de subdivisão adaptativa aplicando-o recursivamente em uma malha triangular de uma superfície plana com bordo. Subdividimos recursivamente um triângulo onde os novos vértices criados pelo processo de subdivisão são sempre posicionados sobre o ponto médio das arestas.

$\|S\|$	$\|E^r(S)\|$	**Faces em $E^r(S)$**	**Corners s/ tab. C**	**Corners c/ tab. C**	**Tem. calc. $E^r(S)$**
153	302	349	$< 0,01s$	$< 0,01s$	$< 0,01s$
561	1.034	1.213	$0.01s$	$< 0,01s$	$< 0,01s$
2.145	3.794	4.477	$0.03s$	$< 0,01s$	$< 0,01s$
8.385	14.498	17.149	$0.19s$	$< 0,01s$	$< 0,01s$
33.153	56.642	67.069	$2,52s$	$< 0,01s$	$< 0,01s$
131.841	223.874	265.213	$37,22s$	$< 0,01s$	$0.07s$
525.825	890.114	1.054.717	$\gg 37,22s$	$< 0,01s$	$0.26s$
2.100.225	3.549.698	4.206.589	$\gg 37,22s$	$< 0,01s$	$1.14s$
8.394.753	14.177.282	16.801.789	$\gg 37,22s$	$0,02s$	$4,78s$

Tabela 6.2: Dados da realização da subdivisão adaptativa com $r = 1$. *Core i5 3.10Ghz, 8GB RAM

$\|S\|$	$\|E^r(S)\|$	**Faces em $E^r(S)$**	**Tem. p/ calc. faces**	**Tem. p/ sub.**
153	302	349	$< 0,01s$	$< 0,01s$
561	1.034	1.213	$< 0,01s$	$< 0,01s$
2.145	3.794	4.477	$0,01s$	$< 0,01s$
8.385	14.498	17.149	$0,01s$	$< 0,01s$
33.153	56.642	67.069	$0,05s$	$0,01s$
131.841	223.874	265.213	$0,19s$	$0,12s$
525.825	890.114	1.054.717	$0,82s$	$0,36s$
2.100.225	3.549.698	4.206.589	$3,62s$	$1,5s$
8.394.753	14.177.282	16.801.789	$16,02s$	$7.3s$

Tabela 6.3: Dados da realização da subdivisão adaptativa com $r = 1$. *Core i5 3.10Ghz, 8GB RAM

As tabelas 6.2 - 6.5 mostram o número de vértices do conjunto S ($|S|$) da região selecionada para subdivisão, o número de vértices após realizar a expansão de S para $E^r(S)$ ($|E^r(S)|$), o número de faces formadas pelos vértices do conjunto $E^r(S)$ que devem ser subdivididas e os tempos para realizar cada passo do algoritmo de subdivisão incremental: obter os *corners* dos vértices do conjunto S usando uma *Corner Table* com e sem a tabela C, realizar a expansão de S para $E^r(S)$, calcular as faces de $E^r(S)$ e por fim aplicar o algoritmo de subdivisão sobre

PUC-Rio - Certificação Digital Nº 1112650/CA

$\|S\|$	$\|E^r(S)\|$	**Faces em** $E^r(S)$	**Corners s/ tab.** C	**Corners c/ tab.** C	**Tem. calc.** $E^r(S)$
153	719	661	$< 0,01s$	$< 0,01s$	$< 0,01s$
561	2.224	1.909	$0,01s$	$< 0,01s$	$< 0,01s$
2.145	7.362	5.941	$0,04s$	$< 0,01s$	$< 0,01s$
8.385	26.286	20.149	$0,3s$	$< 0,01s$	$< 0,01s$
33.153	98.809	73.141	$3,71s$	$< 0,01s$	$0.04s$
131.841	382.479	277.429	$47,61s$	$< 0,01s$	$0.14s$
525.825	1.504.315	1.079.221	$\gg 37,22s$	$< 0,01s$	$0,59s$
2.100.225	5.965.971	4.255.669	$\gg 37,22s$	$< 0,01s$	$2,52s$
8.394.753	23.761.219	16.900.021	$\gg 37,22s$	$0,03s$	$10.55s$

Tabela 6.4: Dados da realização da subdivisão adaptativa com $r = 5$. *Core i5 3.10Ghz, 8GB RAM

$\|S\|$	$\|E^r(S)\|$	**Faces em** $E^r(S)$	**Tem. p/ calc. faces**	**Tem. p/ sub.**
153	719	661	$< 0,01s$	$< 0,01s$
561	2.224	1.909	$< 0,01s$	$< 0,01s$
2.145	7.362	5.941	$0,01s$	$< 0,01s$
8.385	26.286	20.149	$0,02s$	$20,01s$
33.153	98.809	73.141	$0,07s$	$0,02s$
131.841	382.479	277.429	$0,32s$	$0,12s$
525.825	1.504.315	1.079.221	$1,34s$	$0,38s$
2.100.225	5.965.971	4.255.669	$5,8s$	$1,59s$
8.394.753	23.761.219	16.900.021	$25.01s$	$7.26s$

Tabela 6.5: Dados da realização da subdivisão adaptativa com $r = 5$. *Core i5 3.10Ghz, 8GB RAM

PUC-Rio - Certificação Digital Nº 1112650/CA

essas faces. As Tabelas 6.2 e 6.3 ilustram estes dados para um valor de $r = 1$ e as Tabelas 6.4 e 6.5 para $r = 5$.

Pela análise das tabelas é possível perceber que o passo que consome mais tempo é o cálculo das faces formadas por $E^r(S)$. Em seguida o cálculo da expansão do conjunto S para o conjunto $E^r(S)$ e só então o passo de subdivisão, logo, para uma melhora na eficiência do algoritmo estes dois passos devem ser otimizados.

Como esperado, ao aumentar o valor de r o número de vértices em $E^r(S)$ também aumenta, consequentemente aumentando o número de faces a serem subdivididas.

É importante notar que para aplicações que têm como entrada os vértices da região a ser subdividida a *Corner Table* com a tabela C é essencial para realizar a subdivisão com eficiência, uma vez que as tabelas de tempos mostram um tempo muito alto para obter os *corners* dos vértices quando o conjunto S atinge a ordem de 100 mil elementos. Em aplicações com modelos massivos, onde a lista de triângulos é muito grande, essa operação se torna muito cara.

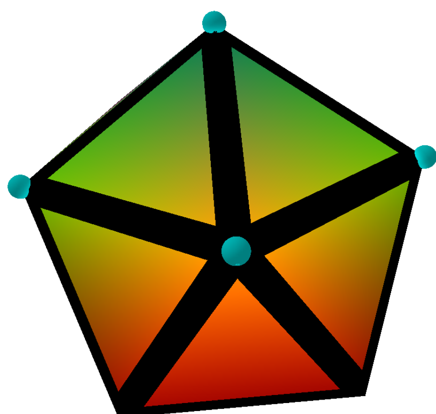
Executamos também a subdivisão adaptativa sobre um icosaedro regular para gerar a malha de uma esfera (Figura 6.8) aplicando vários níveis de subdivisão a partir de um conjunto de vértices usando $r = 3$.

Na Figura 6.9, a subvisão adaptativa é usada aproximar uma malha plana da função $z = sin(\pi x)cosy + 0.1sin(2\pi x)$. Em todos os outrsos exemplos da subdivisão adaptativa, o esquema *Butterfly* foi utilizado o para posicionar os novos vértices.

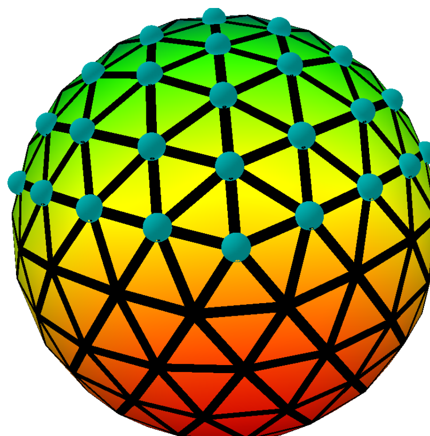
Nas figuras 6.10, 6.11, 6.12, 6.13, um conjunto de vértices S (em destaque) é selecionado para subdivisão e então vários níveis de subdivisão são aplicados sempre subdividindo o sonjunto S'. Todos os resultados mostram uma transição suave entre a região selecionada para subdvisão e a região que não foi subdividida, além de uma boa qualidade dos triângulos resultantes, salvo os casos com a presença de triângulos muito finos na malha, como pode ser observado na Figura 6.11.

PUC-Rio - Certificação Digital Nº 1112650/CA

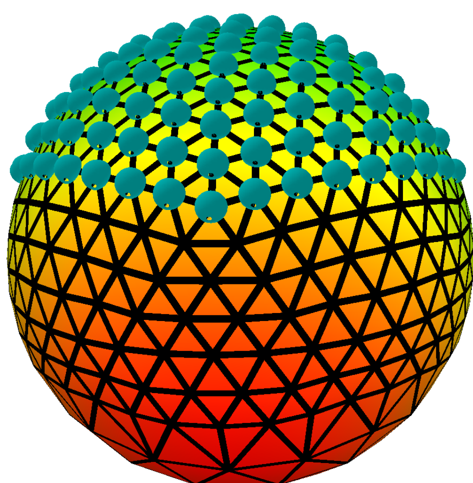
PUC-Rio - Certificação Digital Nº 1112650/CA



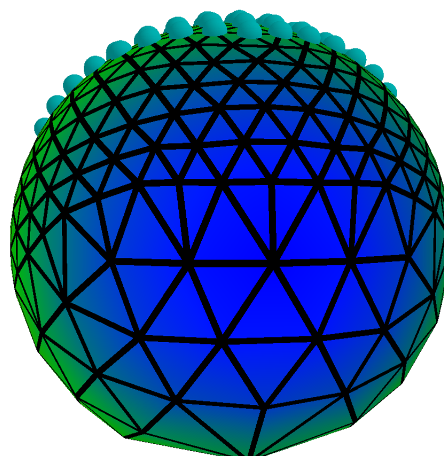
(a)Icosaedro regular com pontos selecionados para subdivisão. 20 faces



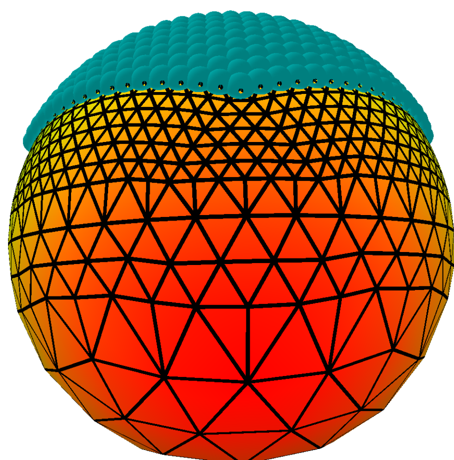
(b)2 leveis de subdivisão com $r = 3$. 292 faces



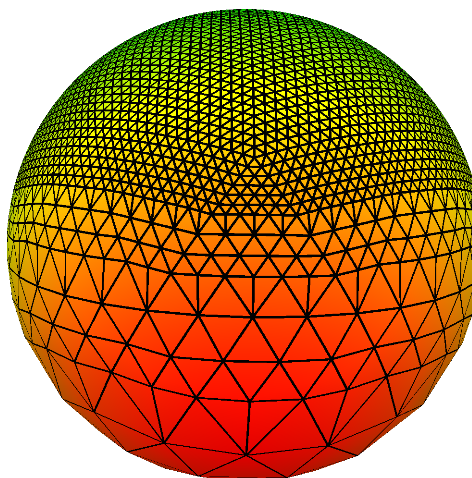
(c)4 leveis de subdivisão com $r = 3$. 752 faces



(d)Mesma esfera de c). Evidenciando a mudança de nível de subdivisão.



(e)5 leveis de subdivisão com $r = 3$. 1084 faces



(f)6 leveis de subdivisão com $r = 3$. 4016faces faces

Figura 6.8: Vários níveis da subdivisão incremental aplicada sobre um icosaedro regular para geração da malha de uma esfera.

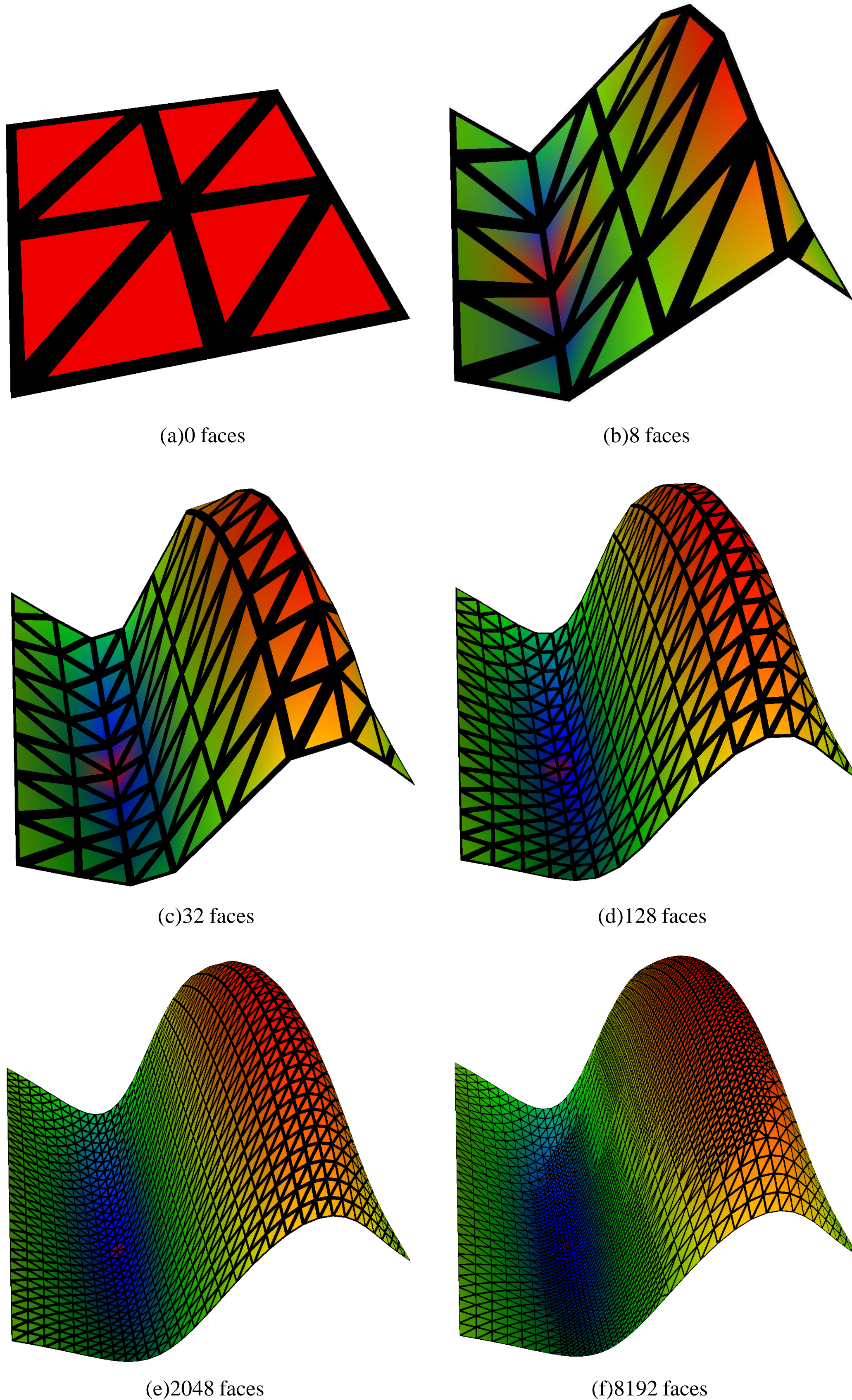
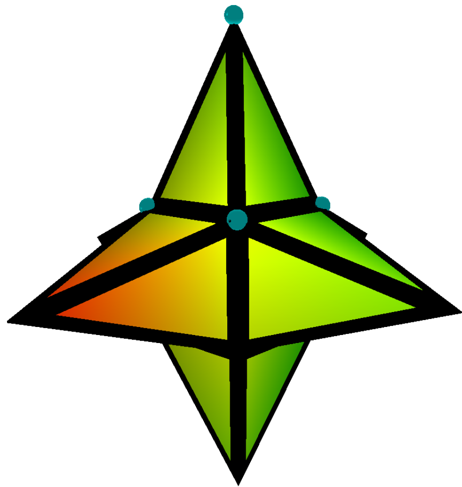
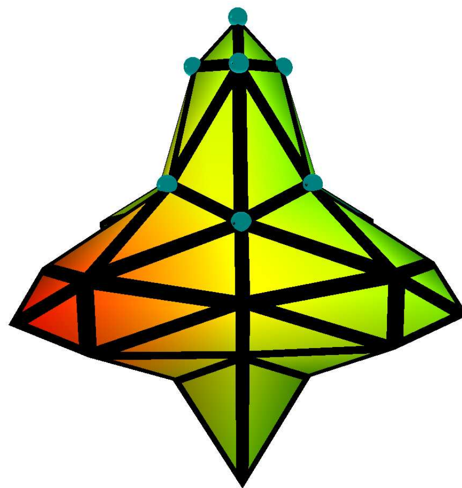


Figura 6.9: Subdivisão adaptativa de uma malha inicial com 8 triângulos para aproximar a função $z = sin(\pi x)cosy + 0.1sin(2\pi x)$. A cada passo a região onde o ponto médio de uma aresta dista mais que 0.005 da posição onde deveria está é selecionada para subdivisão

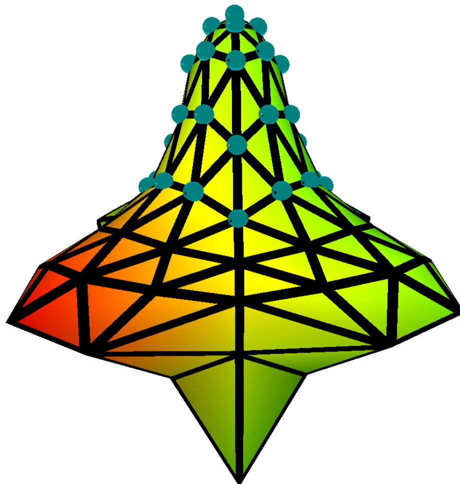
PUC-Rio - Certificação Digital Nº 1112650/CA



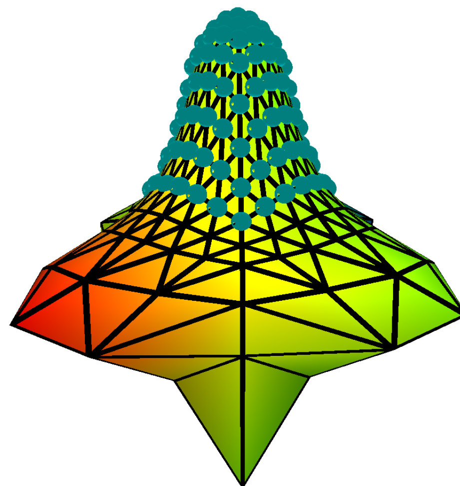
(a)Estrela de 6 pontos. 24 faces.



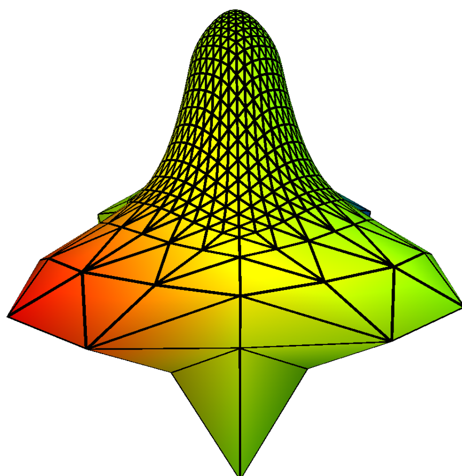
(b)1 nível de subdivisão com $r = 1$. 88 faces



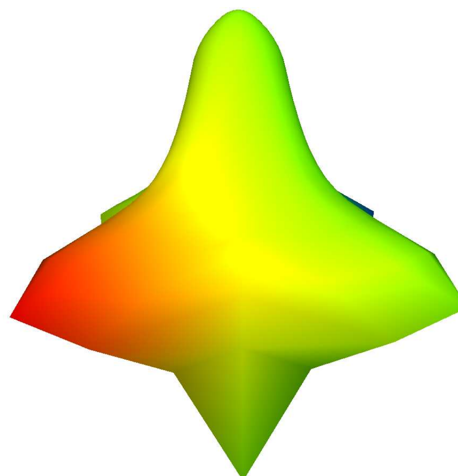
(c)2 níveis de subdivisão com $r = 1$. 208 faces.



(d)3 níveis de subdivisão com $r = 1$. 528 faces



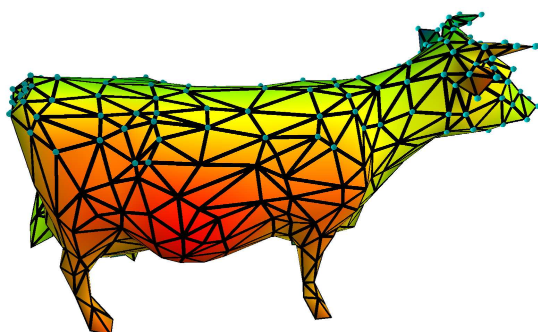
(e)4 níveis de subdivisão com $r = 1$. 1536 faces



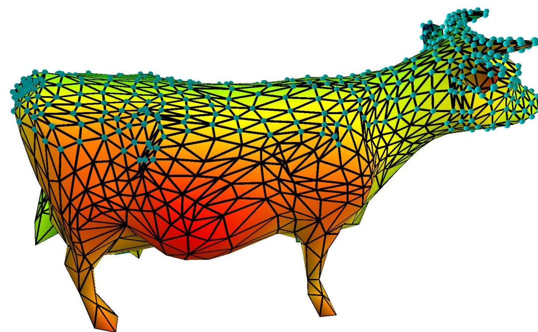
(f)Superfície final com 1536 faces faces

Figura 6.10: Vários níveis da subdivisão incremental aplicada sobre um icosaedro regular para geração da malha de uma esfera.

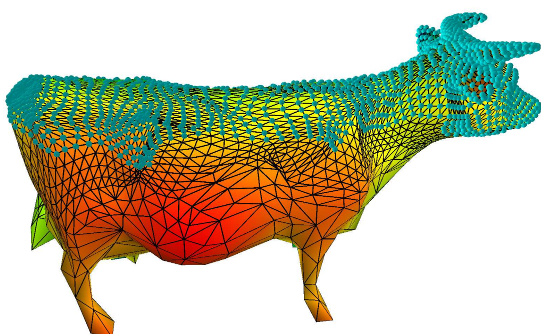
PUC-Rio - Certificação Digital Nº 1112650/CA



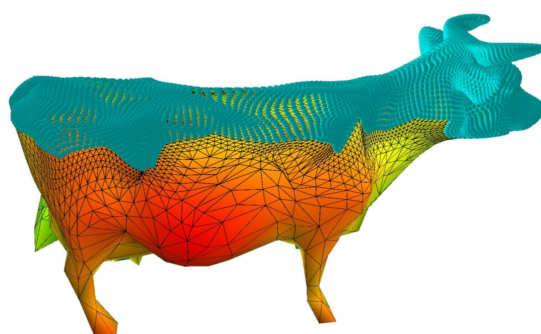
(a)Malha inicial do modelo *cow*. Os pontos selecionados são da região a ser subdividida, ou seja, o conjunto S. 940 faces



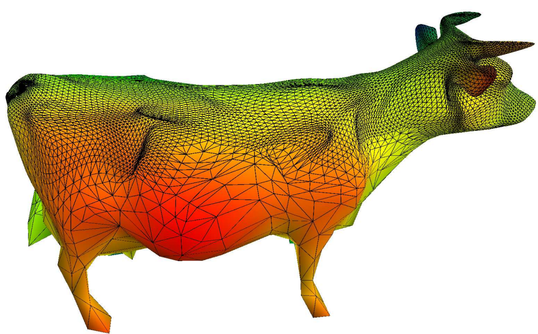
(b)1 nível de subdivisão com $r = 2$. 2628 faces.



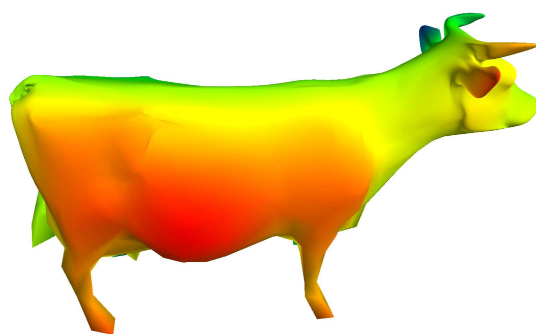
(c)2 níveis de subdivisão com $r = 2$. 8206 faces.



(d)3 níveis de subdivisão com $r = 2$. 28122 faces



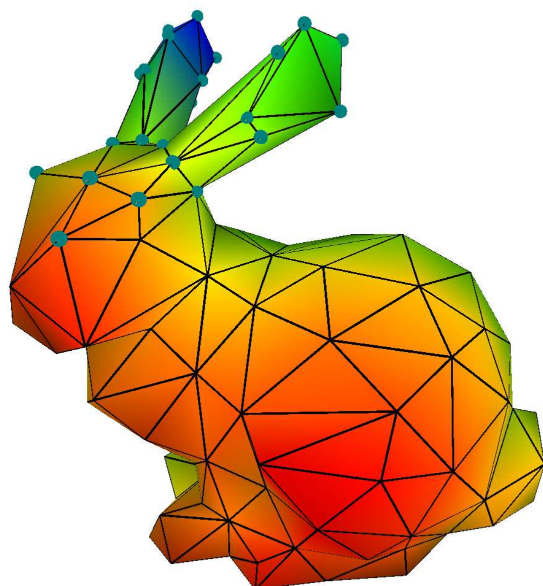
(e)Malha da superfície apresentada em c).



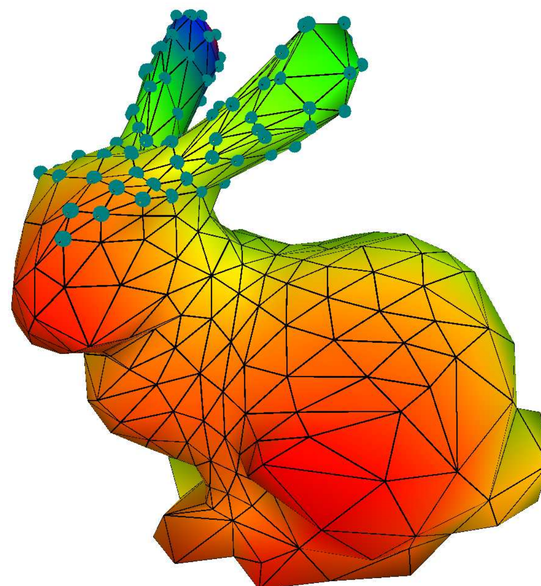
(f)Superfície final com 28122 faces.

Figura 6.11: Subdivisão incremental do modelo *cow* usando *Butterfly* para posicionar os novos vértices criados. Os pontos em destaque pertecem ao conjunto S', ou seja, o pontos a partir de onde o algoritmo faz a expansão.

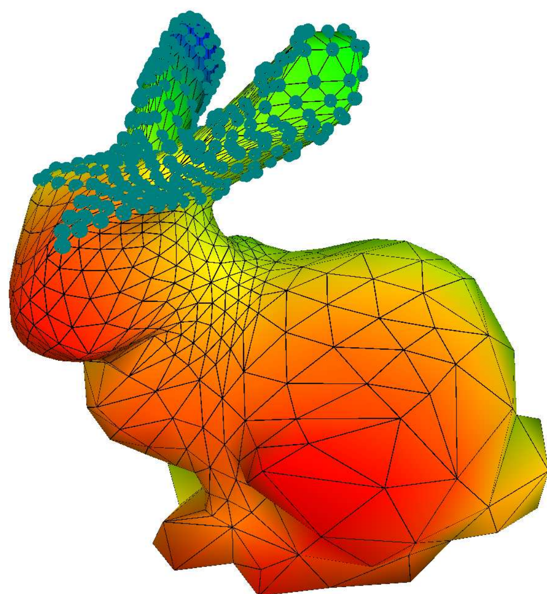
PUC-Rio - Certificação Digital Nº 1112650/CA



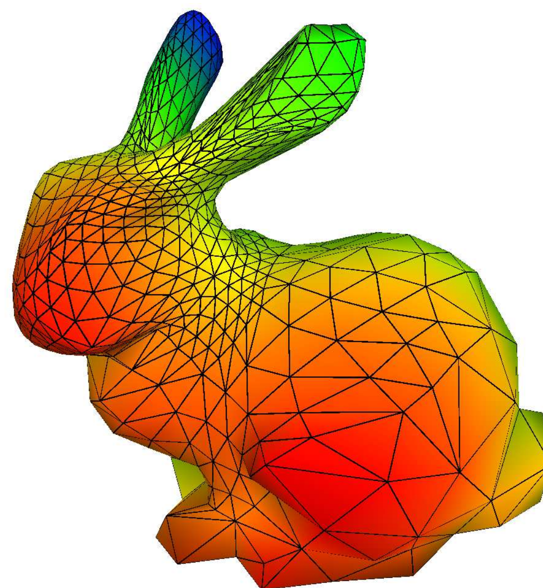
(a)Malha inicial do modelo *bunny*. Os pontos selecionados são da região a ser subdividida, ou seja, o conjunto S. 340 faces



(b)1 nível de subdivisão com $r = 4$. 800 faces.



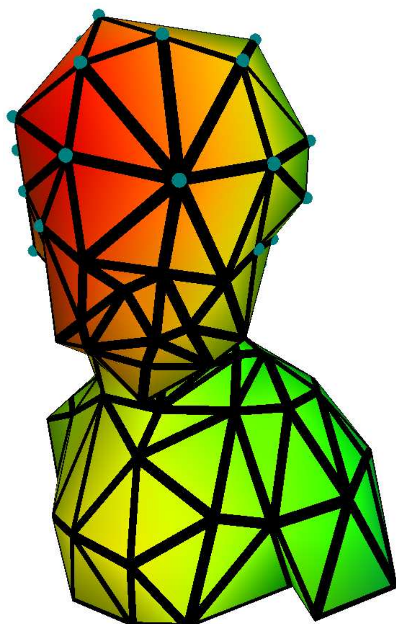
(c)2 níveis de subdivisão com $r = 4$. 1858 faces.



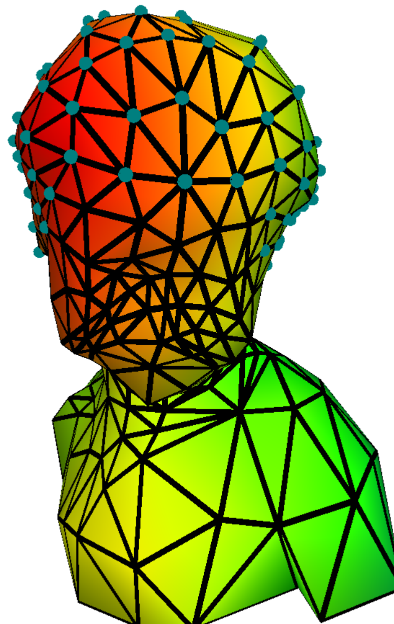
(d)Malha final com 1858 faces

Figura 6.12: Subdivisão incremental do modelo *bunny* usando *Butterfly* para posicionar os novos vértices criados.

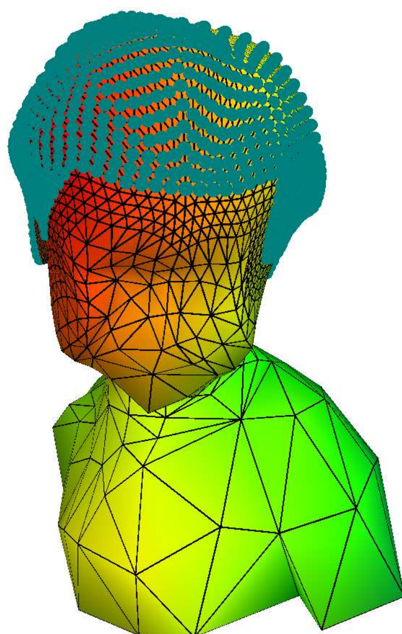
PUC-Rio - Certificação Digital Nº 1112650/CA



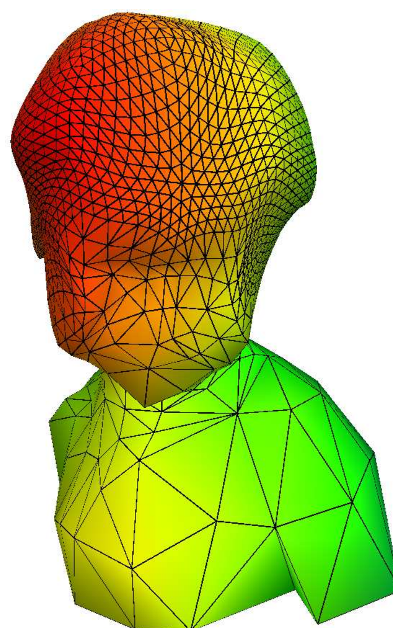
(a)Malha inicial do modelo *bimba*. Os pontos selecionados são da região a ser subdividida, ou seja, o conjunto S. 340 faces



(b)1 nível de subdivisão com $r = 2$. 768 faces.



(c)3 níveis de subdivisão com $r = 2$. 7288 faces.



(d)Malha final com 7288 faces

Figura 6.13: Subdivisão incremental do modelo *bimba* usando *Butterfly* para posicionar os novos vértices criados.

PUC-Rio - Certificação Digital Nº 1112650/CA

7
Conclusão e Trabalhos Futuros

O objetivo de realizar subdivisão global e adaptativa de forma eficiente usando a *Corner Table* como estrutura de dados topológica de suporte foi alcançado. A implementação dos operadores estelares inversíveis preservaram a relação entre a triangulação e a estrutura de dados que a implementa, dando suporte a uma implementação simples e eficiente ao mecanismo de *undo/redo*. Essas ideias todas formam uma sólida base topológica para o desenvolvimento de algoritmos de refinamento e simplificação.

Para todos os testes, a *Corner Table* deu suporte eficiente à subdivisões global e adaptativa, exceto nos casos onde a região selecionada para a subdivisão adaptativa é definida pelos vértices. Nestes casos a tabela C se faz necessária.

Os algoritmos propostos podem ser feitos independentemente da forma como a geometria dos novos vértices é calculada. No esquema *Butterfly* a qualidade da malha resultante depende da qualidade da malha inicial. De forma que malhas com triângulos distorcidos produzem dobras. Quando a geometria da superfície é conhecida, o refinamento se mostrou adequado no controle das transições.

Como trabalhos futuros sugerimos uma melhora nos algoritmos que efetuam a expansão do conjunto S para $E^r(S)$ e calculam as faces em $E^r(S)$.

PUC-Rio - Certificação Digital Nº 1112650/CA

Referências Bibliográficas

[Amresh *et al*, 2002] AMRESH, A.; FARIN, G. ; RAZDAN, A.. **Adaptive subdivision schemes for triangular meshes**. In: HIERARCHICAL AND GEOMETRIC METHODS IN SCIENTIFIC VISUALIZATION, p. 319–327. Springer-Verlag, 2002. 2.2.1

[Andrew *et al*, 1983] ANDREW, R. B.; SHERMAN, A. H. ; WEISER, A.. **Refinement algorithms and data structures for regular local mesh refinement**. In: SCIENTIFIC COMPUTING, volumen 1, p. 3–17. IMACS/North-Holland, 1983. 2.2.1

[Baumgart, 1972] BAUMGART, B. G.. **Winged edge polyhedron representation.** Technical report, Stanford, CA, USA, 1972. 3

[Braid *et al*, 1980] BRAID, I. C.; HILLYARD, R. C. ; STROUD, I. A.. **Stepwise construction of polyhedra in geometric modeling**, p. 123–141. 1980. 3

[Cashman, 2012] CASHMAN, T. J.. **Beyond catmull-clark? a survey of advances in subdivision surface methods**. Comput. Graph. Forum, 31(1):42–61, Feb. 2012. 2

[Catmull and Clark, 1998] CATMULL, E.; CLARK, J.. **Seminal graphics**. chapter Recursively generated B-spline surfaces on arbitrary topological meshes, p. 183–188. ACM, New York, NY, USA, 1998. 2.1

[DeRose *et al*, 1998] DEROSE, T.; KASS, M. ; TRUONG, T.. **Subdivision surfaces in character animation**. In: PROCEEDINGS OF THE 25TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, SIGGRAPH ’98, p. 85–94, New York, NY, USA, 1998. ACM. 2

[Doo and Sabin, 1978] DOO, D.; SABIN, M.. **Behavior of recursive subdivision surfaces near extraordinary points**. p. 356–360. Computer-Aided Design, 1978. 2.1

[Dyn *et al*, 1990] DYN, N.; LEVINE, D. ; GREGORY, J. A.. **A butterfly subdivision scheme for surface interpolation with tension control**. ACM Trans. Graph., 9(2):160–169, Abril 1990. 2.1, 2.1.2

PUC-Rio - Certificação Digital Nº 1112650/CA

[Dyn *et al*, 1993] DYN, N.; S, H. ; LEVIN, D.. **Subdivision schemes for surface interpolation**. Workshop in Computational Geomtry, 9(2):97–118, Abril 1993. 2.1.2

[Ferreira, 2006] DE OLIVEIRA LAGE FERREIRA, M.. **Estruturas de dados topológicas escalonáveis para variedades de dimensão 2 e 3**. Master's thesis, PUC Rio, Rio de Janeiro, Fevereiro 2006. (document), 3, 3.1.1, 3.1, 3.2, 3.7, 3.8, 3.9, 3.10, 3.2.3, 3.11, 5

[Jones *et al*, 1997] JONES, M. T.; PLASSMANN, P. E.. **Parallel algorithms for adaptive mesh refinement**. SIAM J. Sci. Comput., 18(3):686–708, May 1997. 2.2

[Kobblet, 2000] KOBBELT, L.. **$\sqrt{3}$-subdivision**. In: PROCEEDINGS OF THE 27TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, SIGGRAPH '00, p. 103–112, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. 2.2, 2.2

[Lickorish1999] LICKORISH, W. B. R.. **Simplicial moves on complexes and manifolds**. ArXiv Mathematics e-prints, Nov. 1999. 4.1

[Loop, 1987] LOOP, C.. **Smooth subdivision surfaces based on triangles**. Master's thesis, Department of Mathematics, University of Utah, Salt Lake City, August 1987. 2.1, 2.1.1, 2.1.1

[Mantyla, 1987] MÄNTYLÄ, M.. **An introduction to solid modeling**. Computer Science Press, Inc., New York, NY, USA, 1987. 3

[Newman, 1926] NEWMAN, M. H. A.. **On the foundations of combinatorial analysis situs**. In: PROC. ROYAL ACAD., p. 29:610–641, 1926. 4.1

[Pachner, 1991] PACHNER, U.. **P1 homeomorphic manifolds are equivalent by elementary shellings**. In: EUROP. J. COMBINATORICS, p. 12:129–145, 1991. 4.1

[Pakdel and Samavati, 2007] PAKDEL, H.; SAMAVATI, F. F.. **Incremental subdivision for triangle meshes**. Int. J. Comput. Sci. Eng., 3(1):80–92, July 2007. (document), 2, 2.1, 2.1, 2.1.1, 2.1.1, 2.3, 2.1.2, 2.2, 2.2, 2.2.1, 2.2.1, 2.12, 2.2.2, 2.2.2, 2.15, 5.3

[Rossignac *et al*, 2001] ROSSIGNAC, J.; SAFONOVA, A. ; SZYMCZAK, A.. **3d compression made simple: Edgebreaker on a corner-table**. In: SHAPE MODELING INTERNATIONAL CONFERENCE, p. 278–283, 2001. (document), 3, 3.1, 3.1.1, 3.3, 3.1.1

PUC-Rio - Certificação Digital Nº 1112650/CA

[Velho and Zorin, 2001] VELHO, L.; ZORIN, D.. **4-8 subdivision**. Comput. Aided Geom. Des., 18(5):397–427, June 2001. 2.2

[Velho, 2003] VELHO, L.. **Dynamic adaptive meshes**. In: IN DAGSTUHL SEMINAR ON HIERARCHICAL METHODS IN COMPUTER GRAPHICS, 2003. (document), 4.1, 4.1, 4.1

[Vieira, 2003] VIEIRA, A. W.. **A topological approach for mesh simplification**. Master's thesis, PUC Rio, Rio de Janeiro, Outubro 2003. (document), 3.1, 3.1.1, 3.2, 3.1.1, 3.3, 3.1.1, 3.1.2, 3.1.2, 2, 3.1.2, 3, 3.1.2, 4, 4.2, 6, 4.5, 9

[Vieira *et al*, 2004] VIEIRA, A. W.; LEWINER, T.; VELHO, L.; LOPES, H. ; TAVARES, G.. **Stellar mesh simplification using probabilistic optimization**. Computer Graphics Forum, 23(4):825–838, october 2004. 1.1

[Zorin *et al*, 1996] ZORIN, D.; SCHRÖDER, P. ; SWELDENS, W.. **Interpolating subdivision for meshes with arbitrary topology**. In: PROCEEDINGS OF THE 23RD ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, SIGGRAPH '96, p. 189–192, New York, NY, USA, 1996. ACM. (document), 2, 2.1.2, 2.1.2, 2.9

[Zorin *at al*, 1997] ZORIN, D.; SCHRÖDER, P. ; SWELDENS, W.. **Interactive multiresolution mesh editing**. In: PROCEEDINGS OF THE 24TH ANNUAL CONFERENCE ON COMPUTER GRAPHICS AND INTERACTIVE TECHNIQUES, SIGGRAPH '97, p. 259–268, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. 2.2.1

PUC-Rio - Certificação Digital Nº 1112650/CA