Implementação de um Gerador Tridimensional de Malhas de Elementos Finitos, com Aplicações à Simulação Computacional em Odontologia

Mauro Massayoshi Sakamoto

Dissertação de Mestrado

Pós-Graduação em Matemática Aplicada, MAP-058

Implementação de um Gerador Tridimensional de Malhas de Elementos Finitos, com Aplicações à Simulação Computacional em Odontologia

MAURO MASSAYOSHI SAKAMOTO

Orientador: Prof. Dr. José Márcio Machado

São José do Rio Preto

2001

Resumo

Este trabalho mostra a utilização do algoritmo de Delaunay em domínios tridimensionais para a geração de uma malha que leva em conta a anatomia (assimetrias e estruturas internas) de um dente. Os pontos que compõem o domínio foram mapeados de cortes feitos num dente verdadeiro. O trabalho também mostra uma das formas de se aplicar o Algoritmo em domínios não convexos, considerado um dos aspectos críticos do método. Inicialmente o programa foi executado no ambiente Mathematica For Windows, mas devido a complexidade do modelo geométrico, foi feita a conversão do algoritmo para o compilador C++ For Windows sendo o Mathematica usado apenas para gerar os resultado gráficos. Os resultados obtidos nos testes realizados confirmam a eficiência do método adotado.

Palavras Chaves: Método dos Elementos Finitos; Triangulação de Delaunay; Malha Triangular; Malha Tetraédrica; Implementação de Watson

Abstract

This study shows the use of the Delaunay algorithm in three dimensional domains for the generation of a mesh that considers the anatomy (asymmetries and internal structures) of a tooth. Some cuts in a real tooth were scanned . The study also regards a way to use the Algorithm in non convex domains, which is considered a critical feature of the method. Initially the program was executed in Mathematica for Windows enviroment, but due to the geometric model complexity, the algorithm was converted for compiler C++ for Windows, and the Mathematica was just applied for the graphic results generation. The tests results showed the efficacy of this method.

“Para ser grande, sê inteiro:
nada teu exagera ou exclui.
Sê todo em cada coisa. Põe quanto
és no mínimo que fazes.
Assim em cada lago a lua toda
Brilha, porque alta vive”.

Ricardo Reis

AGRADECIMENTOS

A Deus, pela iluminação.

Aos meus pais, de quem sempre recebi e continuarei recebendo apoio e amor incondicionais;

À meu orientador, Prof. Dr. José Márcio Machado, pela orientação acadêmico-científica, sem a qual seria impossível a realização deste trabalho;

À Faculdade de Odontologia de Araçatuba pela colaboração e fornecimento do material.

Aos professores do curso, pela oportunidade de poder compartilhar seus ensinamentos;

Aos meus colegas de curso, pelo companheirismo e troca de experiências profissionais ao longo desses anos;

À minha amiga Carina Alexandra Rondini Marreto por fornecer exemplos e a versão inicial do algoritmo.

À minha irmã, Carina Sakamoto, pela colaboração com o material odontológico e incentivo nos momentos de dúvidas pelos quais passei.

A todos estes e outros que não foram citados e que contribuíram para a execução deste trabalho, a minha mais sincera gratidão.

SUMÁRIO

1 INTRODUÇÃO

Na pesquisa odontológica tem-se empregado recentemente o método dos elementos finitos para simular, em computador, os esforços mecânicos e tensões que ocorrem em dentes e próteses durante o processo de mastigação.

Embora isso signifique um grande avanço na área odontológica, essas simulações vêm utilizando modelos planos ou simplificados da geometria de um dente real, o que nem sempre leva a resultados plenamente confiáveis.

Uma simulação realista deve ser feita necessariamente em três dimensões, com um pré-processamento geométrico e uma geração de malha tetraédrica que leve em conta todas as assimetrias e estruturas internas da geometria de um dente.

Atualmente o grande avanço tecnológico principalmente de hardware tem possibilitado o acesso a computadores de grande capacidade de processamento e memória a custo cada vez menor, possibilitando assim simulações para modelos bem mais complexos, como a anatomia de um dente.

Contudo uma exigência fundamental para a simulação é determinar com precisão o domínio geométrico de forma a reproduzir com o maior grau de precisão possível o modelo real em estudo.

Muitos modelos geométricos podem ser obtidos a partir de fórmulas ou combinações de fórmulas matemáticas que definem de modo aceitável as fronteiras do domínio geométrico requerido. Apesar disso, não é possível reproduzir a partir de fórmulas matemáticas (ou combinações das mesmas) modelos altamente irregulares como o de um dente, sendo necessário adotar outras técnicas que possibilitem essa reconstrução. Essas técnicas serão descritas mais adiante.

Outra exigência fundamental para qualquer programa de simulação que empregue o Método dos Elementos Finitos é a capacidade de gerar malhas em modelos geométricos que respeitem as fronteiras do domínio em estudo; em outras palavras é a reprodução do domínio original de forma consistente através da união de um conjunto de elementos geométricos topologicamente regulares (não necessariamente iguais) e disjuntos.

Atualmente os algoritmos de geração automática de malhas em três dimensões não são completamente flexíveis , ou seja não existe um algoritmo genérico que possa ser usado em todas as situações e domínios geométricos possíveis. Cada um deles possui suas qualidades e deficiências e muitas vezes requerendo a intervenção do usuário em pontos críticos do domínio.

Embora existam várias abordagens disponíveis para a geração de malhas tridimensionais, uma das mais populares e explorada é o algoritmo de Delaunay. Ainda que esse método apresente dificuldades na reprodução de geometrias com concavidades, é possível contornar esse problema de forma relativamente simples tornando o algoritmo viável mesmo para domínios não convexos. Além disso o algoritmo de Delaunay

com a implementação de Watson se mostrou bastante eficiente para sólidos regulares, não regulares e multiplamente conexos [1].

As técnicas apresentadas neste trabalho foram implementadas em dois ambientes distintos. As operações matemáticas do método foram executadas em linguagem C++ for Windows [2] devido a eficiência em tempo de processamento; já a análise dos resultados gráficos foi produzida no ambiente Mathematica for Windows [3] que fornece funções pré existentes para a obtenção do resultado desejado.

A idéia do método apresentado é preparar um modelo geométrico tridimensional de fácil interação e portabilidade que reconstitui realisticamente as formas dos dentes tornando-o confiável em problemas de simulação.

2 DESCRIÇÃO DA TÉCNICA

A ferramenta de modelagem tridimensional para representar os sólidos em computação gráfica é definida a partir de um conjunto finito de pontos adequadamente escolhidos que no ambiente computacional descrevem de modo aproximado e aceitável a fronteira do sólido (contorno) e sua região interior.

Sólidos regulares, e irregulares podem ser obtidos a partir de diferentes métodos a serem aplicados de acordo com a geometria original.

A partir desses pontos o Algoritmo de Delaunay é capaz de construir automaticamente um conjunto de tetraedros cuja união reproduz o sólido original.

Este capítulo abrange superficialmente os métodos utilizados para determinar domínios tridimensionais, enfatizando apenas o método utilizado no mapeamento do dente. Também será brevemente discutido o algoritmo de Delaunay e a implementação de Watson usada para a sua construção. Uma abordagem mais aprofundada sobre os métodos para a implementação do algoritmo de Delaunay pode ser integralmente encontrada em [1, 4]. Este trabalho pode ser considerado uma continuação de [5] que é um software que reproduz no computador um dente tridimensional com um princípio de malha triangular que só não se deu por

completo devido às limitações impostas pelo equipamento da época, máquinas com memória limitada trabalhando a 16 bits.

2.1 Descrevendo o domínio tridimensional

Um domínio tridimensional descreve computacionalmente a estrutura gráfica espacial de um modelo real; em outras palavras é a reprodução no computador de toda a anatomia que o sólido em estudo apresenta. Na verdade, os objetos são construídos de grupos de faces tridimensionais como um cubo que é formado por seis polígonos (quadrados) em cada face que o descreve. O mapeamento em si consiste em escolher o conjunto de pontos que melhor represente o modelo requerido, para em seguida armazenar numa lista de valores relativos a sua posição horizontal, vertical e sua profundidade. Quanto maior o número de pontos mapeados melhor será a definição obtida. É importante salientar que para regiões irregulares, o número de pontos mapeados é muito importante para evitar rompimentos em sua fronteira quando manipulados. Vejamos alguns dos métodos utilizados para esse fim.

2.1.1 Modelos gerados a partir de sólidos regulares.

As nuvens de pontos pertencentes a tipos básicos de sólidos podem ser obtidas por meio das suas equações paramétricas dos sólidos. Essas equações podem ser facilmente encontradas em livros de Geometria Analítica ou Análise Vetorial.

Como exemplo podemos mostrar as equações paramétricas da esfera:

$x=R\,Cos[\theta]\,Cos[\phi]$,

$y=R\,Sen[\theta]\,Cos[\phi]$,

$z=R\,Sen[\phi]$,

onde R é o raio da esfera com $0 \leq \theta \leq 2\pi$ e $-\pi/2 \leq \phi \leq \pi/2$.

Uma explicação mais detalhada de equações paramétricas além de outras topologias podem ser encontradas em [6]. Figuras mais complexas podem ser montadas a partir da concatenação adequada dessas equações [7].

Outro modo para obter domínios mais complexos é através do agrupamento de formas pré-definidas de sólidos regulares que podem descrever todos os tipos de domínios em aplicações de engenharia elétrica [8]. A título de ilustração, damos a seguir na figura 2.1.1 um sólido gerado a partir do agrupamento de vários cilindros.

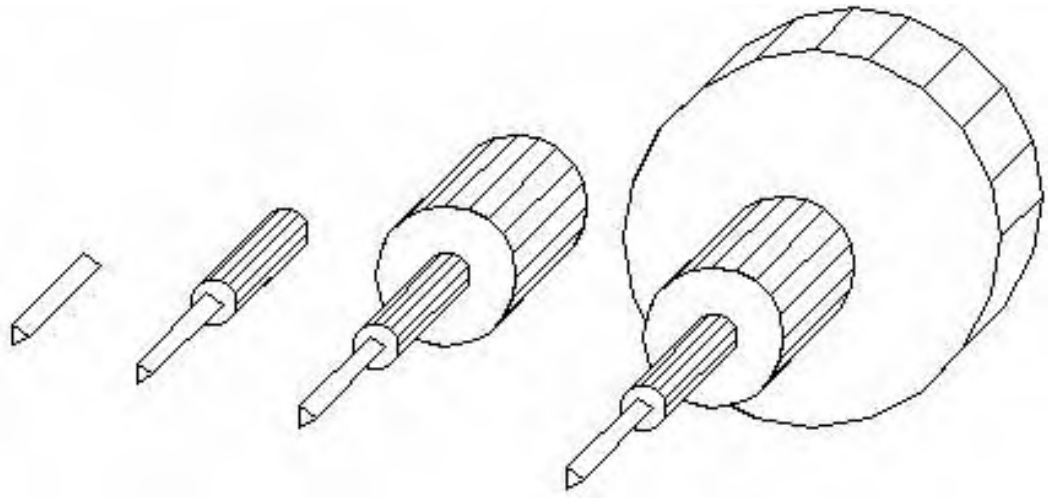

Figura 2.1.1: Sólido gerado a partir de quatro cilindros

A desvantagem desse método é que não podemos obter domínios de modelos altamente irregulares como um rosto humano, estrutura óssea ou um dente. Para esses exemplos existem métodos mais eficazes.

2.1.2 Modelos gerados por scanner

As tecnologias atuais permitem uma maior liberdade para reproduzir formas cada vez mais complexas em representações cada vez mais reais.

Essas novas tecnologias permitem que scanners capturem a forma e a superfície de modelos bem complexos como uma cabeça humana. Em geral esse tipo de scanner trabalha com um feixe de raios laser que emite milhares de raios e captura os reflexos medindo a profundidade da superfície a ser escaneada. O resultado é um conjunto de milhões de pontos tridimensionais.

Figura 2.1.2: Rosto gerado a partir de um Scanner 3D

No entanto, como o scanner se movimenta somente na horizontal e vertical, o mapeamento é feito apenas na superfície, sendo impossível reproduzir estruturas internas.

2.1.3 Modelos obtidos por tomografia computadorizada

Para muitos problemas a reprodução da superfície externa não é suficiente para obter resultados confiáveis sendo necessária a análise das estruturas internas. Tirando proveito da tecnologia atual esse mapeamento vem sendo feito através de tomografia computadorizada [9], em outras palavras esse método consiste em usar imagens digitalizadas de secções transversais do modelo real, obtidas através de tomografia computadorizada. Essas imagens obtidas são então processadas para detecção de bordas que descrevam perfeitamente a fronteira. Como exemplo mostramos na figura 2.1.3 uma imagem digitalizada de uma secção transversal de um fêmur e sua imagem processada.

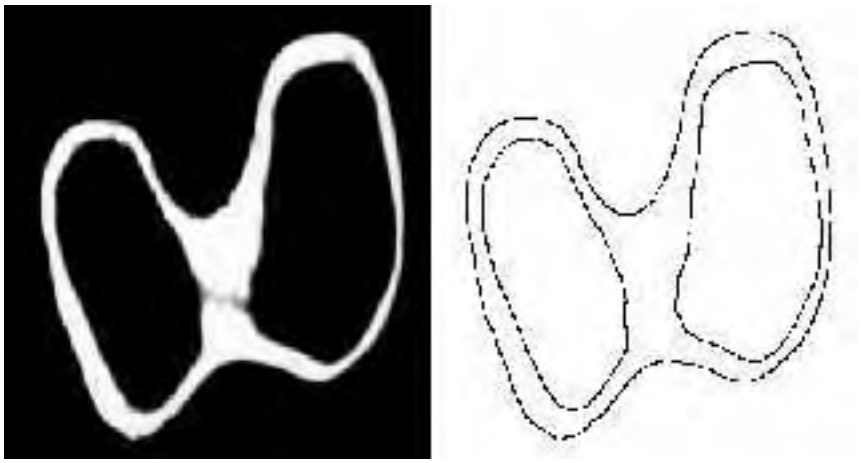

Figura 2.1.3: Mapeamento de uma secção transversal do fêmur

Apesar da flexibilidade desse método, o modelamento apresenta algumas dificuldades devido às características dos materiais que compõe as estruturas internas.

2.1.4 Modelos mapeados a partir de fotos.

A idéia básica desse método é a mesma do método descrito anteriormente, porém não é feito com o mesmo grau de otimização da tomografia computadorizada. Esse processo requer fotos (sempre com a mesma escala) de cortes em um modelo real.

Essas fotos são minuciosamente analisadas em busca de cada fronteira que compõe sua superfície e suas estruturas. Em seguida é escolhido um número finito de pontos, os quais são adequadamente distribuídos e descrevem a estrutura original da forma mais próxima do real.

A técnica aqui descrita foi a utilizada nesse projeto devido a facilidade de se obter o domínio sem a necessidade de equipamentos sofisticados. Na próxima seção explicaremos detalhadamente a aplicação desse método para o mapeamento do dente.

2.2 Mapeamento do dente

Inicialmente um dente Molar Inferior foi seccionado obtendo-se doze cortes com espessura máxima de um milímetro, sendo que cada um desses cortes foi posicionado entre duas réguas que reproduziram

um sistema de coordenadas. A partir dessas normas cada corte foi fotografado com ampliação.

Essas fotos foram quadriculadas utilizando-se as escalas das réguas e em seguida delimitada a fronteira de cada estrutura interna (esmalte, dentina e polpa) [10]. Os vértices e concavidades encontrados foram armazenados em uma lista com seus respectivos valores de coordenadas tridimensionais (posição horizontal, vertical e profundidade) e assim sucessivamente para cada uma das fotos. A inserção desses pontos na lista deve ser em ordem crescente, ou seja deve-se armazenar os pontos da polpa, dentina e esmalte nessa ordem, para evitar problemas na geração da malha. Para as regiões com grande irregularidades foram mapeados mais pontos a fim de melhor definir o contorno e evitar o rompimento da fronteira ao manipular de forma efetiva esses pontos.

Após mapearmos todos os cortes, obtivemos doze secções bidimensionais que reproduziram de forma aceitável cada uma das fotos. Apesar de serem secções bidimensionais cada uma delas está em um plano diferente reconstituindo o volume original do dente. Em outras palavras, o resultado computacional dos valores medidos são doze conjuntos de pontos que quando agrupados reproduzem a geometria do dente com todo seu volume, irregularidades e estruturas.

Podemos observar esses critérios de medição pela foto da figura 2.2.1 e seu respectivo mapeamento na figura 2.2.2. O agrupamento de todos os pontos dos cortes é visto na figura 2.2.3.

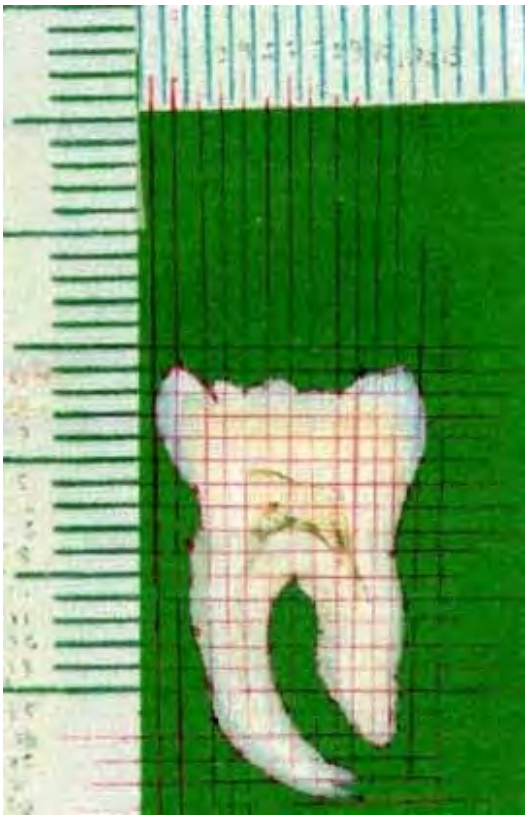

Figura 2.2.1: Foto de corte do dente

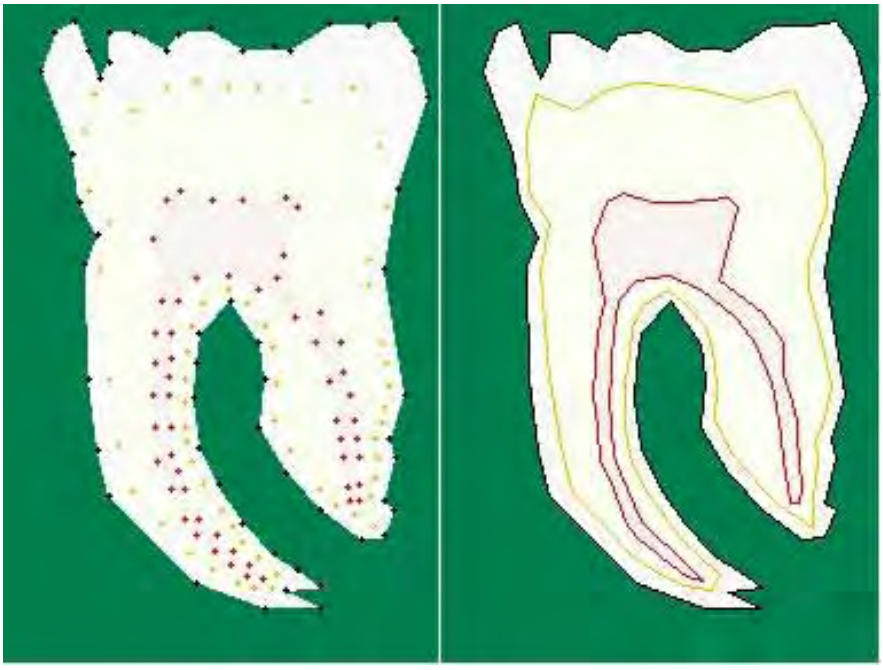

Figura 2.2.2: Mapeamento do corte da figura 2.2.1 para o Mathematica

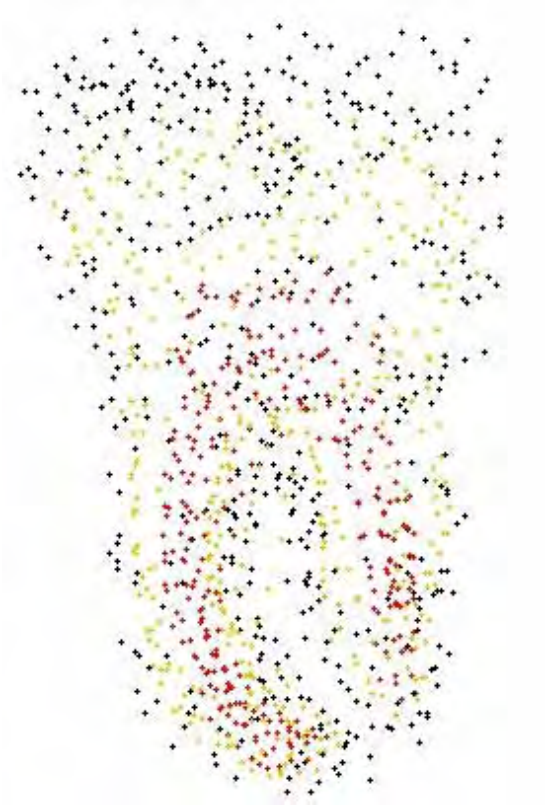

Figura 2.2.3: Agrupamento de todos os pontos medidos

Esse mapeamento também poderia ser feito através da digitalização das fotos e o posterior processamento das imagens obtidas em busca das fronteiras que determinam o domínio, porém como no caso da tomografia computadorizada, é difícil separar os materiais que compõem as estruturas internas.

Em busca de uma melhor definição das fronteiras podemos aplicar técnicas de refinamento como B-Spline, que aproximam a curva obtida por uma polinomial. Esse tipo de interpolação funciona muito bem quando o algoritmo de geração de malhas não requer intervenção do usuário nos pontos críticos (técnica da propagação frontal, por exemplo). Como esse não é o caso do algoritmo de Delaunay onde muitas vezes foi

necessário perturbar os pontos para suprir as degenerecências, foi preferível trabalhar com pontos conhecidos (mapeados) do que pontos interpolados, uma vez que as funções B-Spline são apenas um ajustador de aproximação, sendo que a curva gerada não passa pelos pontos originais[*]. Uma discussão mais detalhada sobre a aplicação de B-Spline num dente mapeado pode ser encontrada em [5].

Um problema desse tipo de mapeamento é que os cortes no dente causam perda de material dentário que causa pequenos prejuízos na continuidade de duas secções consecutivas. Na figura 2.2.4 podemos observar esse tipo de problema; pois apesar de serem dois cortes consecutivos a raiz da direita de ambos são bem diferentes.

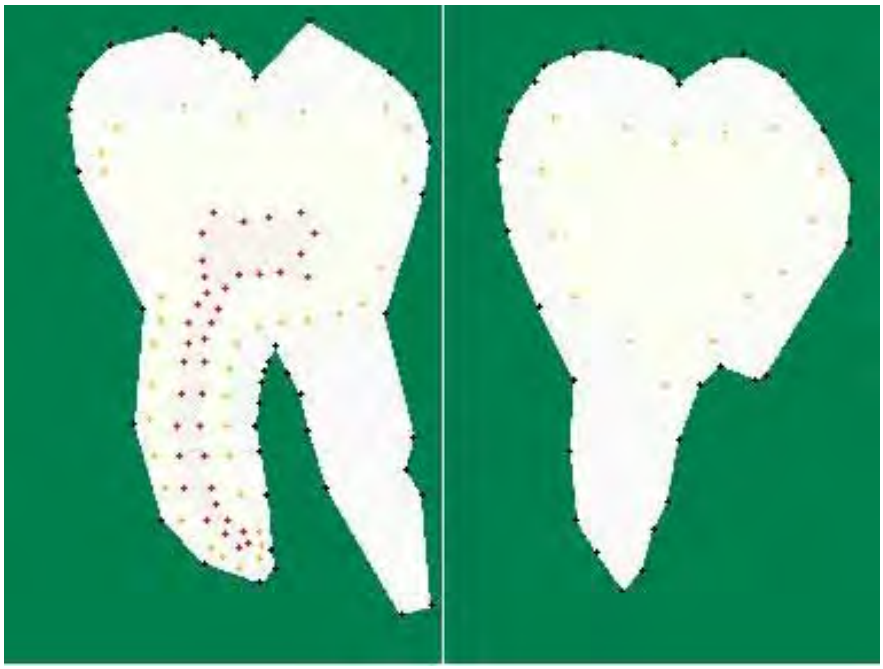

Figura 2.2.4: Secções consecutivas com problemas de continuidade na raiz

[*] Seminário Bases para Spline apresentado em 1999 na disciplina de Análise de Métodos Numéricos

2.3 O Algoritmo de Delaunay

O algoritmo de Delaunay é um processo que possibilita a construção automática de uma malha de triângulos ou tetraedros a partir de um conjunto finito de pontos. A malha triangular é obtida quando trabalhamos com domínios bidimensionais e a tetraédrica em domínios tridimensionais.

Para a sua implementação existem alguns métodos encontrados na literatura que serão brevemente discutidos neste capítulo, mas primeiramente faremos uma rápida explanação matemática do método. Um estudo mais aprofundado do mesmo pode ser encontrado em [1, 4].

Definição 2.3.1 Um simplexo K é uma lista ordenada de vértices $\{P_i\}$ com $1 \leq i \leq d + 1$ onde d é a dimensão do espaço euclidiano afim E. Chama-se de det(K), ao determinante de ordem $d + 1$ dado por:

$$\det(K) = \begin{vmatrix} 1 & \dots & \dots & \dots & 1 \\ P_1^1 & \dots & \dots & \dots & P_{d+1}^1 \\ P_1^2 & \dots & \dots & \dots & P_{d+1}^2 \\ \vdots & & & & \vdots \\ P_1^d & \dots & \dots & \dots & P_{d+1}^d \end{vmatrix}$$

onde P_i^j é a coordenada j do ponto P_i para $1 \leq i \leq d + 1$ e $1 \leq j \leq d$.

Por definição tem-se que, se det(K) > 0, K é dito ser positivamente orientado e se det(K) = 0, K é dito ser degenerado (ou seja, todos os seus vértices pertencem ao mesmo hiperlano)[11].

Definição 2.3.2 Entende-se por "triangulação" de um domínio Ω, que se supõe limitado e poliédrico, um conjunto ζ de simplexos. Para esse conjunto temos as seguintes condições:

1. A intersecção de dois elementos de ζ é vazia ou se reduz a um vértice, a uma aresta (no caso bidimensional) ou a uma face (no caso tridimensional).

2. A união dos elementos de ζ é igual a Ω.

3. Os elementos de ζ devem ser topologicamente regulares: no caso bidimensional por exemplo, esses elementos devem ser o mais próximo possível da equilateralidade, o que não os obriga a serem iguais [12, 13].

4. Eventualmente a incidência de elementos deve ser maior nas regiões do domínio onde isto é necessário [14].

Definição 2.3.3 Segundo Delaunay [15] quando existir um elemento de ζ que não é um simplexo os pontos $P_1,....,P_n$ serão ditos *especiais* (figura 2.2.3).

Portanto, se os pontos $P_1,....,P_n$ não forem especiais, concluí-se que ζ é uma triangulação da envolvente convexa dos pontos $P_1,....,P_n$, caso contrário, obtém-se uma triangulação chamada "deduzida" de ζ, dividindo em simplexos cada elemento de ζ, que não seja um simplexo (figura 2.2.4).

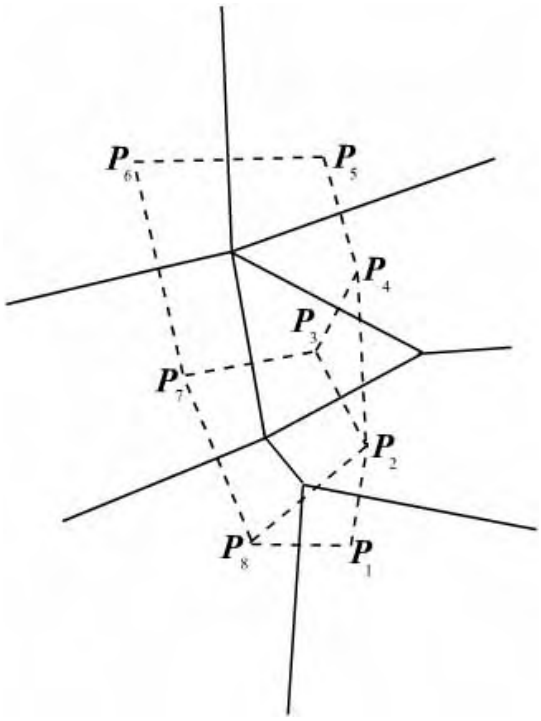

Figura 2.3.3: Divisão em simplexos de pontos especiais

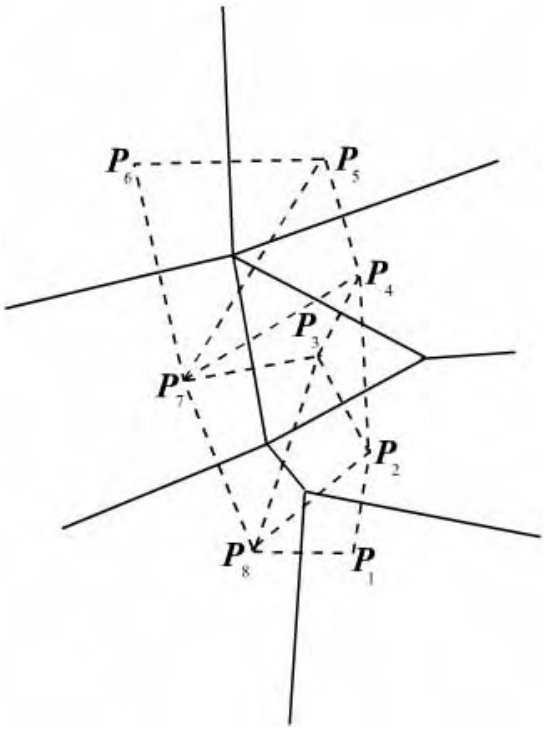

Figura 2.3.4: Triangulação deduzida de ζ

Resumidamente, chamar-se-á triangulação de Delaunay associada aos pontos $P_1,....,P_n$ a triangulação ζ (se os pontos $P_1,....,P_n$ não forem especiais), ou toda triangulação deduzida de ζ (se os pontos $P_1,....,P_n$ forem especiais).

2.3.1 Triangulação de Delaunay

Iniciamos construindo uma primeira triangulação ζ_{d+1} que é composta de um só simplexo $K = P_1,....,P_{d+1}$. Esse simplexo não é degenerado e positivamente orientado desde que: det(K) > 0. Uma vez que os pontos $P_1,....,P_n$ não pertençam a um mesmo hiperplano esta condição é sempre verificável: basta fazer uma permutação conveniente desses pontos.

Para incluirmos um novo ponto P_{i+1} devemos primeiro determinar a situação geométrica do (i+1)-ésimo ponto em relação a triangulação.

Distinguem-se três tipos de situação geométrica do ponto P_{i+1} em relação aos conjuntos $\cup K$ (com $K \in \zeta_i$) e $\cup B$ (com $B \in \beta_i$) (onde $\cup K$ é igual à envolvente convexa dos pontos $P_1,....,P_n$ e $\cup B$ é o conjunto das esferas circunscritas aos elementos ζ_i):

i. $P_{i+1} \in \cup K$

ii. $P_{i+1} \notin \cup B$

iii. $P_{i+1} \notin \cup K$ e $P_{i+1} \in \cup B$

Na figura 2.3.5 temos um exemplo onde os pontos P, Q e T estão respectivamente nas situações i, ii e iii.

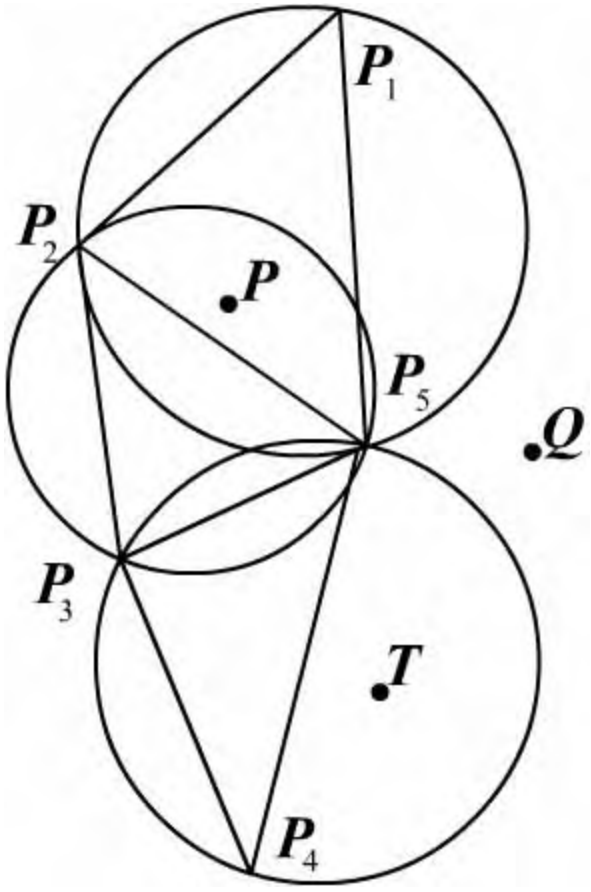

Figura 2.3.5: Situação geométrica dos pontos P, Q e T

Essas situações induzem a três tipos de modificações da triangulação de Delaunay ζ_i para obter uma triangulação de Delaunay ζ_{i+1} que compreende o ponto P_{i+1}.

Para a situação *i*, onde $P_{i+1} \in \cup K$ então $P_{i+1} \in \cup B$, suponha que λ seja o conjunto dos elementos de ζ_i cuja esfera circunscrita contém *P* e sejam $F_1,....,F_w$ as faces de elementos de λ que não são comuns a dois elementos de λ.

O algoritmo gera um conjunto $\zeta_{i+1}=(\zeta_i/\lambda)\cup(P_{i+1}F_j)1 \leq j \leq w$ que é uma triangulação de Delaunay associada aos pontos $P_1,....,P_{i+1}$ [16].

Podemos observar essa situação na figura 2.3.6 e 2.3.7 onde $P_{i+1}=P$ e $P_{i+1}=P_9$ respectivamente.

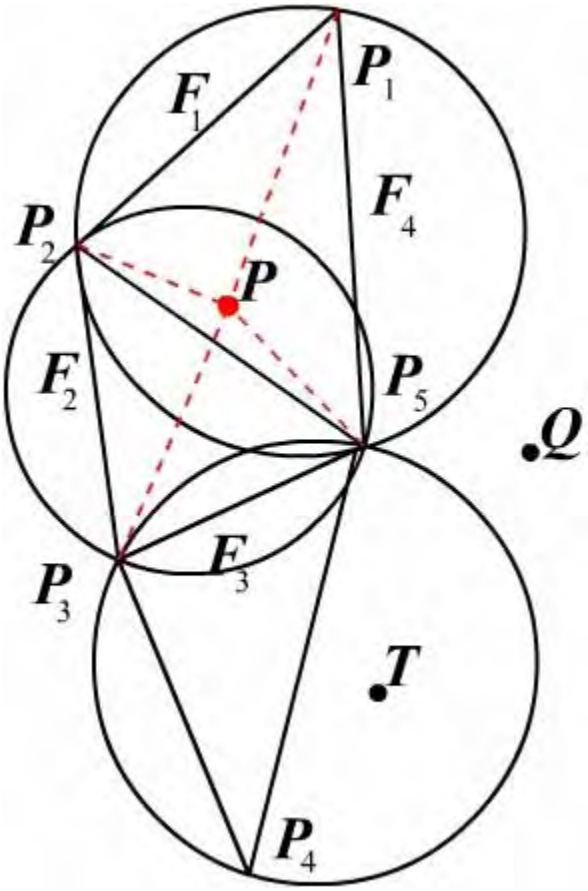

Figura 2.3.6: Inclusão do ponto P (situação i)

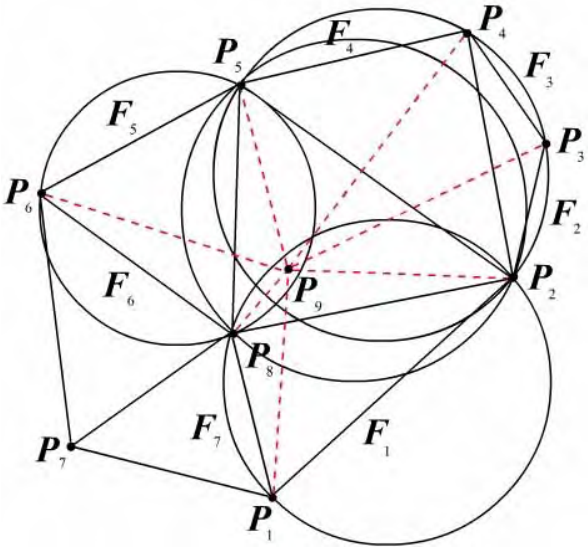

Figura 2.3.7: Inclusão do ponto P_9 (situação i)

A união dos elementos de λ é igual ao polígono $P_1P_2P_3P_5$ (figura 2.3.6) e $P_1P_2P_3P_4P_5P_6P_8$ (figura 2.3.7) ao passo que os novos elementos $P_{i+1}F_j$ aparecem pontilhados.

Para a situação *ii*, onde $P_{i+1} \notin \cup B$ então $P_{i+1} \notin \cup K$. Toma-se $F_1,....,F_w$ as faces externas do politopo $\cup K$ que definem um hiperplano separando estritamente P_{i+1} e $\cup K$.

O algoritmo gera um conjunto $\zeta_{i+1}=(\zeta_i)\cup(P_{i+1}F_j)1 \leq j \leq w$ que é uma triangulação de Delaunay associada aos pontos $P_1,....,P_{i+1}$ [16].

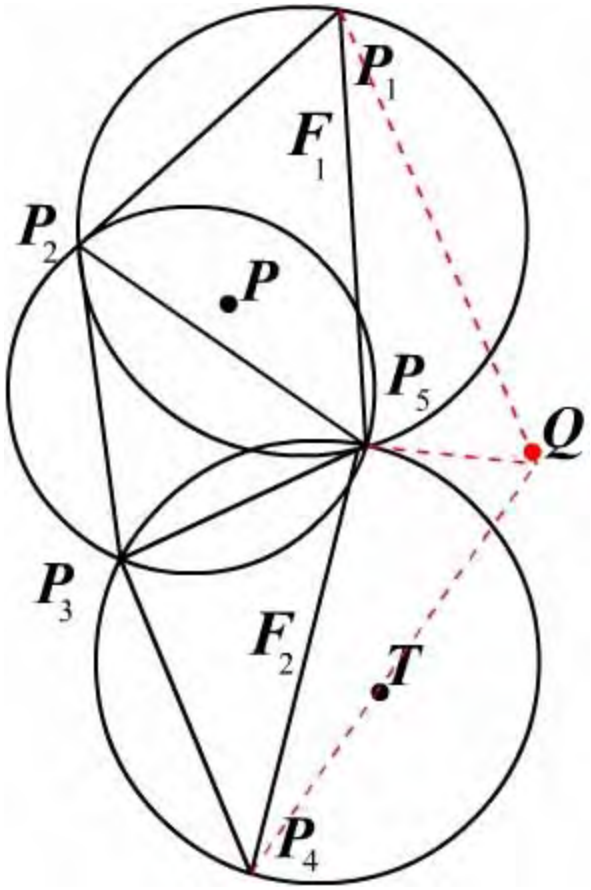

Figura 2.3.8: Inclusão do ponto *Q* (situação *ii*)

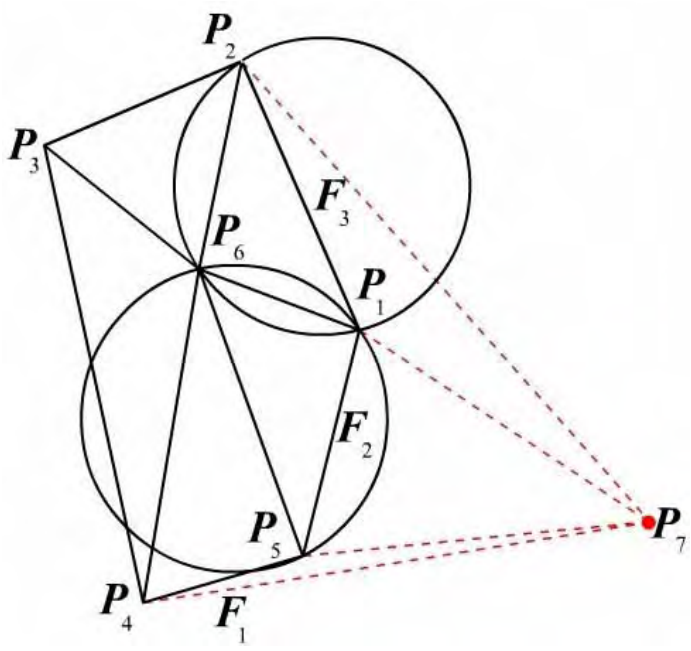

Figura 2.3.9: Inclusão do ponto P_7 (situação *ii*)

Nas figuras 2.3.8 e 2.3.9 pode-se observar esse processo de construção tomando respectivamente $P_{i+1} = Q$ e $P_{i+1} = P_7$, onde os novos simplexos $P_{i+1}F_j$ aparecem pontilhados.

A situação *iii*, coincide com a exclusão das situações *i* e *ii*. Neste caso tem-se $P_{i+1} \notin \cup K$ mas P_{i+1} pode pertencer à fronteira de $\cup K$.

Denota-se por λ o conjunto dos elementos de ζ_i cuja esfera circunscrita contém P_{i+1} e sejam:

- $F_1,....,F_w$ as faces de elementos de λ que não são comuns a dois elementos de λ e que determinam um hiperplano não separando P_{i+1} e $\cup K$.

- $F_{w+1},....,F_{w+q}$ as faces externas de $\cup K$ que não são faces dos elementos de λ e que determinam um hiperplano separando estritamente P_{i+1} e $\cup K$.

O algoritmo gera um conjunto $\zeta_{i+1} = (\zeta_i / \lambda) \cup (P_{i+1}F_j)$ $1 \leq j \leq w + q$ que é uma triangulação de Delaunay associada aos pontos $P_1,....,P_{i+1}$ [16].

As figuras 2.3.10 e 2.3.11 permitem observar este processo de construção partindo respectivamente de $P_{i+1} = T$, $w = 2$, $q = 0$ e $P_{i+1} = P_{12}$, $w = 5$, $q = 2$.

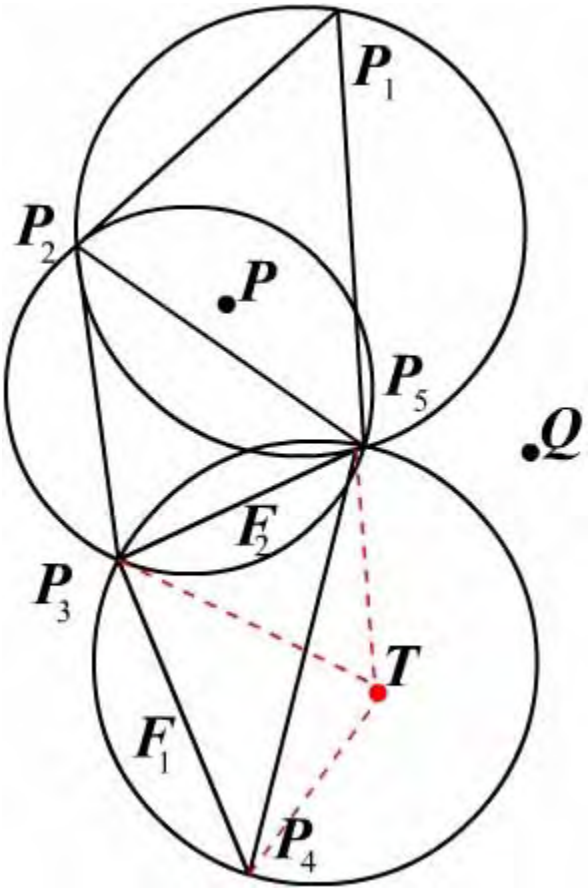

Figura 2.3.10: Inclusão do ponto T (situação *iii*)

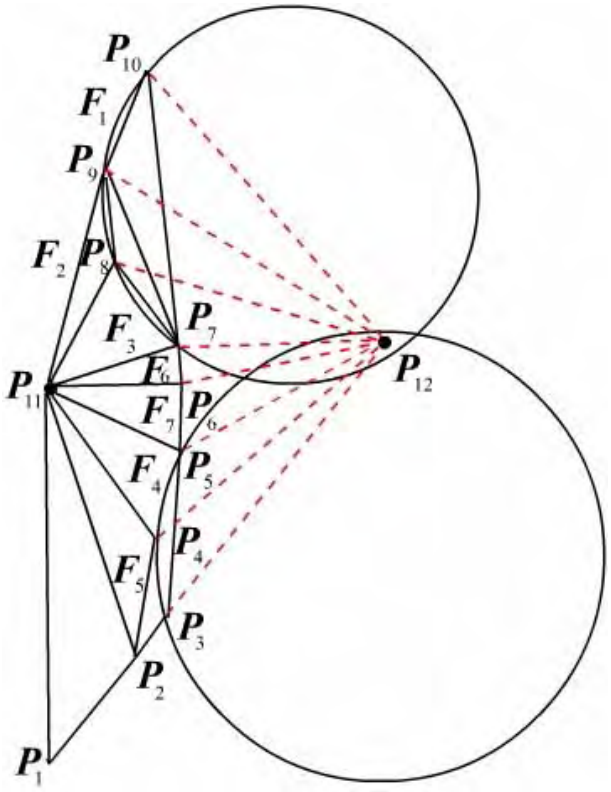

Figura 2.3.11: Inclusão do ponto P_{12} (situação *iii*)

Note que a união dos elementos de λ é igual ao polígono $P_3P_4P_5$ (figura 2.3.10) e à reunião dos polígonos $P_7P_{10}P_9P_8$ e $P_3P_5P_4$ (figura 2.3.11) enquanto que os novos elementos aparecem pontilhados.

Em todos os casos vistos anteriormente deve-se ter cuidado para que os "novos" simplexos $P_{i+1}F_j$ sejam positivamente orientados.

2.3.2 Descrição da fronteira de $\boldsymbol{\Omega}$

Em geral a descrição de um poliedro coincide com a de sua fronteira. Quando o poliedro é convexo a fronteira é descrita por uma lista de pontos onde $\boldsymbol{\Omega}$ é a envolvente convexa bastando fornecer todos os pontos extremos de $\boldsymbol{\Omega}$.

Para um poliedro “quase convexo” podemos descrever essa fronteira por meio de uma lista de pontos tais que, para cada um dentre eles e para cada poliedro **Ω**, o valor *1* ou *0* lhe seja atribuído caso pertença ou não à fronteira de Ω_i. Deste modo podemos descrever a fronteira de um poliedro “quase convexo” fornecendo a lista das coordenadas dos pontos e uma tabela anexa que permita reconhecer em quais lugares a fronteira de **Ω** é “convexa” ou “côncava”. Este processo pode ser observado pelo exemplo da figura 2.3.12 onde usamos a lista das coordenadas e sua tabela anexa.

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9
Ω_1	0	0	0	0	0	0	0	0	0
Ω_2	1	0	0	0	0	0	0	0	0
Ω_3	0	0	0	0	0	0	0	0	0
	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}	P_{17}	P_{18}
Ω_1	0	0	0	0	0	0	0	0	0
Ω_2	0	0	0	0	0	0	0	0	1
Ω_3	0	0	0	0	1	1	1	1	1
	P_{19}	P_{20}	P_{21}	P_{22}	P_{23}	P_{24}	P_{25}	P_{26}	P_{27}
Ω_1	0	0	1	1	1	1	1	1	1
Ω_2	1	1	0	0	0	0	0	0	0
Ω_3	0	0	0	0	0	0	0	0	0

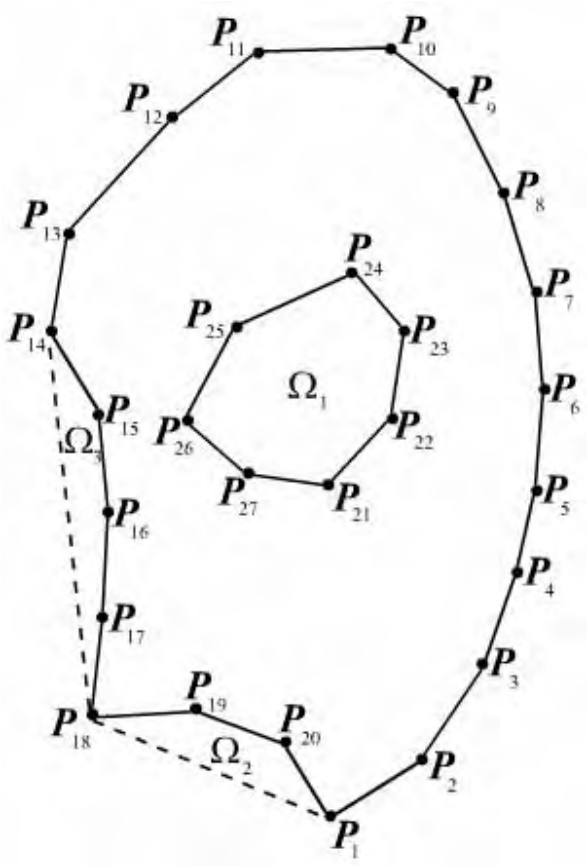

Figura 2.3.12: Fronteira de um poliedro quase convexo

2.3.3 Tratamento dos pontos internos

Após uma triangulação fronteiriça, é preciso “refinar” essa triangulação, isto é, incluir um conjunto de pontos internos convenientemente distribuídos em **Ω**, de maneira a obter uma triangulação que se apoia nestes pontos e naqueles da fronteira e que verifica as condições já descritas.

Um ponto interno *Q* é *compatível* com a fronteira de **Ω** se *Q* não pertencer a nenhuma esfera fechada circunscrita a um elemento

exterior de **Ω**. Um exemplo de ponto não compatível com a fronteira é dado por meio da figura 2.3.13.

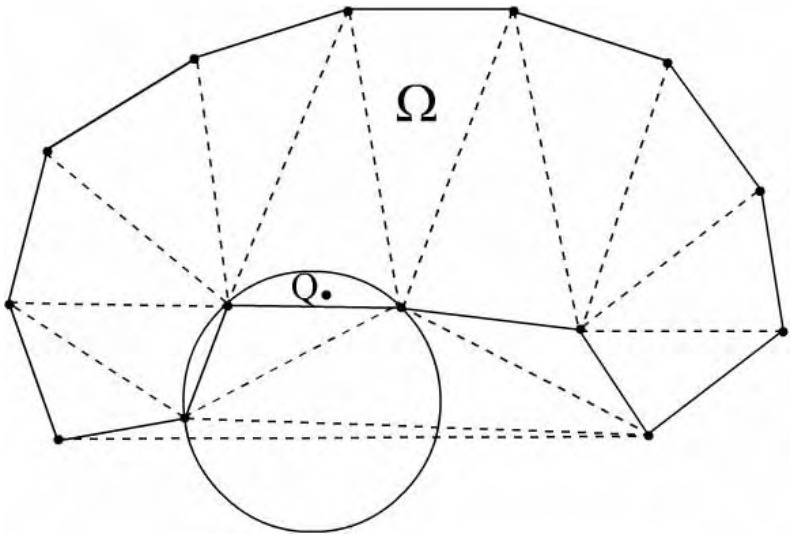

Figura 2.3.13: Exemplo de um ponto não compatível com a fronteira de **Ω**

Se os pontos internos Q_i's são compatíveis com a fronteira de **Ω**, é demonstrável que a triangulação de Delaunay associada aos vértices situados na fronteira e a esses pontos Q_i's "respeita" a fronteira de **Ω** [16].

Na prática não é restrito exigir que os pontos Q_i's sejam "compatíveis" com a fronteira de **Ω**, pois se um ponto Q não for compatível, isso significa que a fronteira de **Ω** é mal descrita (poucos vértices dados para descrever a curvatura dessa fronteira) ou que Q é muito próximo da fronteira. A "qualidade" dessa triangulação depende da escolha dos pontos Q_i's.

2.4 Implementação de Watson

Para obter a triangulação de Delaunay podemos adotar alguns dos métodos de geração de malhas encontrados na literatura [1], como o método de Shelto e Cendes ou mesmo outros algoritmos similares.

Dentre esses algoritmos existentes para o desenvolvimento da triangulação de Delaunay a técnica mais citada na literatura para a construção de uma malha de Delaunay é a implementação de Watson [17].

A maioria dos métodos inicia com um tetraedro que contém o novo ponto e a partir daí novos tetraedros são inseridos para cada novo ponto adicionado. Já na implementação de Watson temos um tetraedro que contém todos os pontos e assim novos tetraedros internos são formados enquanto os pontos são introduzidos.

As etapas da implementação são [18]:

1. Construir uma lista com as coordenadas dos n pontos que descrevem o sólido

2. Determinar um paralelepípedo englobante a partir do “extent” do sólido. O “extent” é o cálculo das coordenadas máximas de x,y e z dos n pontos.

3. Decompor o paralelepípedo em 6 tetraedros com os n pontos forçosamente espalhados por cada um deles

4. O primeiro ponto da lista é inserido na tetraedrização da etapa 3 em seguida calculam-se as esferas que

circunscrevem os tetraedros verificando se o ponto está incluído em alguma dessas esferas. Os tetraedros dentro das esferas que incluem o ponto são analisados, e suas faces comuns são eliminadas e novos tetraedros formados, unindo o ponto às faces restantes não comuns. E assim sucessivamente para os demais pontos.

5. Após a inclusão de todos os pontos da lista teremos tetraedros com vértices que coincidem com vértices do "extent". Esses tetraedros são eliminados da lista.

A implementação de Watson pode ser assim descrita:

- Inicia-se com um parelelepípedo englobante decomposto em seis tetraedros onde os pontos estão espalhados por cada um deles; novos tetraedros internos são formados enquanto os pontos são introduzidos um a um.

- Num típico estágio do processo, um novo ponto é testado para determinar quais esferas circunscritas dos tetraedros existentes contém o ponto. Os tetraedros associados são removidos, restando uma região poliédrica contendo o novo ponto.

- Arestas unindo o novo ponto às faces triangulares da superfície de região poliédrica são criadas, definindo tetraedros que preenchem a região poliédrica. Combinando esses com os tetraedros externos da região poliédrica produz-se uma nova triangulação de Delaunay que contém o recente ponto adicionado.

Escrevendo na forma de um algoritmo esse método pode ser estruturado da seguinte maneira:

PARA i=1 ... Número de Pontos
INICIO
  PARA j=0 ... Número de Tetraedros
  INICIO
    Calcula a circunsfera C_j do tetraedro T_j
    SE C_j contém o ponto i ENTÃO
    INICIO
      Adiciona a face de T_j para a lista de faces.
      Adiciona T_j para a lista de tetraedros deletados.
      Marca T_j para deleção da lista de tetraedros na malha.
    FIM;
  FIM;
  Apaga todos os tetraedros que foram marcados para deleção. Deleta todas as faces comuns dos tetraedros deletados da lista de faces. Formar tetraedros usando o ponto i e a lista de faces e adicionar esses tetraedros para a lista de tetraedros.
FIM.

A implementação computacional desse algoritmo será apresentada no próximo capítulo.

3 IMPLEMENTAÇÃO DOS ALGORITMOS

Neste capítulo apresentaremos um breve histórico dos softwares utilizados, passando-se então, para uma visão geral dos algoritmos empregados, mostrando algumas otimizações para acelerar o processo matemático.

As implementações dos algoritmos foram divididas em dois módulos que são matemático e gráfico. No módulo matemático trataremos essencialmente das funções que utilizam processamento matemático para gerar a malha, enquanto que no módulo gráfico mostraremos as funções utilizadas para visualizar as malhas dos tetraedros.

3.1 Softwares utilizados

Para a geração da malha tridimensional do dente foi inicialmente utilizado o Mathematica 3.0 for Windows e posteriormente adotado o Mathematica 4.0 for Windows devido ao seu gerenciamento de memória mais sofisticado, que permite trabalhar com um número muito maior de pontos.

Com o objetivo de diminuir o tempo de processamento, parte dos algoritmos foram convertidos para a linguagem C++ for Windows.

3.1.1 O ambiente Mathematica

O grande diferencial do Mathematica é a variedade de recursos para cálculos algébricos, numéricos e visualização gráfica, por isso é uma ferramenta largamente utilizada por engenheiros e cientistas para desenvolvimento em simulação computacional.

O Mathematica produz com pouco volume de código, gráficos em duas e três dimensões, bem como contornos e equipotenciais, gerados a partir de funções ou listas. Ele é dotado de muitas opções para controlar a saída de gráficos que são produzidos no formato PostScript que permite uma grande portabilidade.

Quando as aplicações consomem grande tempo com processamento, é conveniente converter partes críticas do código em rotinas FORTRAN ou C que interagem com o Mathematica. Uma abordagem mais detalhada dos recursos disponíveis nesse ambiente pode ser encontrado em [19, 20, 21].

3.1.2 A linguagem C++ for Windows.

As primeiras linguagens de programação foram criadas para projetos de pequena escala, mas à medida que os programas tornaram-se maiores, os erros se proliferaram afetando todas as partes do programa, ou seja, ocorreram conflitos em seções diferentes do código. Algumas linguagens (BASIC, FORTRAN e C) surgiram e contornaram esse problema, mas foram a velocidade e flexibilidade (estrutura de dados,

encapsulamento e modularidade) da linguagem C que tornaram a programação mais dinâmica.

Quando os programas C ficaram muito longos a linguagem foi aperfeiçoada, surgindo assim a linguagem C++ que apresenta os conceitos de classe, herança, instância e objeto. À medida que os programas se tornaram mais sofisticados o endereçamento de memória tornou-se um problema pois o MS-DOS não tinha muito suporte para o gerenciamento de memória (trabalha a 16 bits). O lançamento do Windows 95 foi a solução para esses problemas, uma vez que Windows 95 trabalha com endereçamento de memória de 32 bits e, portanto pode acessar 2^{32} , ou 4294967296 bytes de memória física [2, 22].

Todos esses recursos citados tornam a linguagem C++ for Windows indicada para programas que utilizam muita memória e requerem grande tempo de CPU.

3.2 Módulo matemático

O módulo matemático engloba as funções de geração de malha ou seja trata do aspecto computacional da implementação de Watson para o algoritmo de Delaunay. O algoritmo inicialmente implementado para o ambiente Mathematica foi primeiramente otimizado (para o Mathematica) e posteriormente convertido para a linguagem C++ for Windows. Faremos um estudo detalhado apenas das funções implementadas em C.

Maiores informações das funções para o ambiente Mathematica podem ser encontradas de forma detalhada em [1].

3.2.1 Implementação no ambiente Mathematica

A função principal é dada por:

```
Do[trip=lcord[[i]];
    px=trip[[1]]; py=trip[[2]]; pz=trip[[3]];
    Do[lindex=tetra[[j]];
        Do[pi[[k]]=lindex[[k]];
            pts=lcord[[pi[[k]]]];
            xx[[k]]=pts[[1]];
            yy[[k]]=pts[[2]];
            zz[[k]]=pts[[3]],
         {k,1,4}];
         spher[xx,yy,zz];
         R2=(a^2+b^2+c^2-4d)/4.;
         R=(px+a/2)^2+(py+b/2)^2+(pz+c/2)^2;
         If[R<R2,fac=addfac[fac,lindex];T[[j]]=0],
    {j,1,Length[tetra]}];
     tetraold=deltetra[tetra,T];
     delcomfac[fac];
     If[tetra=={},tetra=t,tetra=Join[tetraold,t]];
     T=Table[1,{Length[tetra]}];fac={},
{i,9,Length[lcord]}]
```

A função acima precisa ler a lista *lcord* que descreve o domínio do sólido, em seguida a função *spher* resolve um sistema de equações lineares. Com o resultado obtido verificamos se há algum ponto no interior da circunsfera. Caso isso ocorra chamamos a função *addfac* que armazena as faces do novo tetraedro e em seguida executamos as funções

que removem os tetraedros que não passaram no teste de inclusão e também as triplas repetidas. Essas funções são *deltetra* e *delcomfac* respectivamente.

A função *spher* usa uma função pré existente no Mathematica que é *LinearSolve* e resolve sistemas lineares.

```
spher[xx:{_,_,_}, yy:{_,_,_}, zz:{_,_,_}]:=Module[{m,const},
    m={{xx[[1]],yy[[1]],zz[[1]],1},
        {xx[[2]],yy[[2]],zz[[2]],1},
        {xx[[3]],yy[[3]],zz[[3]],1},
        {xx[[4]],yy[[4]],zz[[4]],1}};
    const={-xx[[1]]^2-yy[[1]]^2-zz[[1]]^2,
            -xx[[2]]^2-yy[[2]]^2-zz[[2]]^2,
            -xx[[3]]^2-yy[[3]]^2-zz[[3]]^2,
            -xx[[4]]^2-yy[[4]]^2-zz[[4]]^2};
    vars=LinearSolve[m,const];
    sol=Flatten[vars];
    a=sol[[1]]; b=sol[[2]]; c=sol[[3]]; d=sol[[4]];
]
```

Com o objetivo de diminuir o tempo de processamento algumas modificações foram feitas no programa principal e na rotina *spher*. Essas mudanças são pouco significativas e consistem em diminuir o número de listas utilizadas e melhorar a manipulação das listas restantes. O ganho em desempenho com essas modificações foi pequeno, porém num processamento demorado, o tempo poupado é considerável. As modificações no programa principal podem ser vistas abaixo:

```
Do[px=lcord[[i,1]]; py=lcord[[i,2]]; pz=lcord[[i,3]];
   Do[lindex=tetra[[j]];
```

```
    Do[matA[[k,1]]=lcord[[lindex[[k]],1]];
        matA[[k,2]]= lcord[[lindex[[k]],2]];
        matA[[k,3]]= lcord[[lindex[[k]],3]];
        matA[[k,4]]= lcord[[lindex[[k]],4]],
    {k,1,4}];
    spher[matA];
    R2=(a^2+b^2+c^2-4d)/4.;
    R=(px+a/2)^2+(py+b/2)^2+(pz+c/2)^2;
    If[R<R2,fac=addfac[fac,lindex];T[[j]]=0],
    {j,1,Length[tetra]}];
    tetraold=deltetra[tetra,T];
    delcomfac[fac];
    If[tetra=={},tetra=t,tetra=Join[tetraold,t]];
    T=Table[1,{Length[tetra]}];fac={},
{i,9,Length[lcord]}]
```

Neste algoritmo modificado eliminamos as listas *trip*, *pi*, *pts*, *xx*, *yy*, *zz* e passamos diretamente o valor de *lcord* para *px*,*py*,*pz*. Também inserimos esses pontos em *matA* já no formato apropriado para a função *LinearSolve*. A função *spher* foi então alterada para o seguinte formato:

```
spher[xx:{_,_,_}, yy:{_,_,_}, zz:{_,_,_}]:=Module[{m,const},
    m=matA;
    const={-m[[1,1]]^2-m[[1,2]]^2-m[[131]]^2,
            -m[[2,1]]^2-m[[2,2]]^2-m[[2,3]]^2,
            -m[[3,1]]^2-m[[3,2]]^2-m[[3,3]]^2,
            -m[[4,1]]^2-m[[4,2]]^2-m[[4,3]]^2};
    vars=LinearSolve[m,const];
    a=vars[[1]]; b=vars[[2]]; c=vars[[3]]; d=vars[[4]];
```

]

As demais funções utilizadas não foram modificadas.

3.2.2 Implementação na Linguagem C++

Com base nos algoritmos modificados apresentados acima, traduzimos o código para a linguagem C++ for Windows. O primeiro passo para essa conversão foi a definição das estruturas de armazenamento. A estrutura escolhida foi uma lista ligada duplamente encadeada que permite uma grande flexibilidade na manipulação de seus nós. As principais listas implementadas foram a *tetra*, *lcord* e *fac* que guardam respectivamente os pontos dos tetraedros, as coordenadas do domínio do sólido e as novas faces que formarão novos tetraedros. Todas as listas utilizadas usam a mesma estrutura (figura 3.2.2) mudando apenas o tipo de valor armazenado em cada nó da lista.

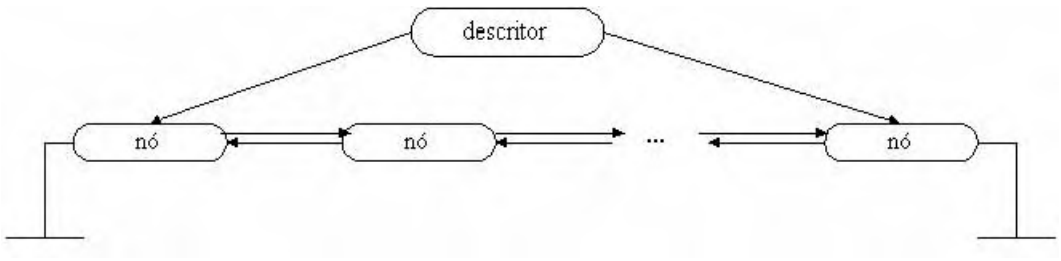

Figura 3.2.2: Representação da estrutura de armazenamento

Faremos um estudo detalhado apenas das funções fundamentais para o algoritmo de Delaunay.

Inicialmente o programa lê dois arquivos *tetra.dat* e *lcord.dat*, onde *tetra.dat* possui a discretização inicial {{1,2,3,6}, {1,3,4,6}, {1,4,5,6}, {3,4,6,7}, {4,6,7,5}, {4,7,8,5}} e *lcord.dat* que é composta dos vértices que formam o domínio do sólido. A lista armazenada em *lcord.dat* possui o seguinte formato: lcord={{_,_,_},{_,_,_},...{_,_,_},{_,_,_},...}. Em seguida iniciamos a lista T ={1,1,1,1,1,1}. Esta lista informa quais pontos passaram no teste de inclusão.

Esses valores então são transferidos para as listas *tetra* e *lcord* respectivamente.

A seguir vemos a implementação do programa principal:

```
void __fastcall TFrmDelaunay::principal(void)
{
  int i,j,k,l,lindex[4], Lentetra,;
  double R2,R,px,py,pz, X[4];
  i=9;
  Lentetra=Dtetra->Len;
  do
  { Lcord(i); px=lcord->n[0]; py=lcord->n[1]; pz=lcord->n[2];
    j=1;
    do
    { Tetra(j,Lentetra);
      for(l=0; l<4; l++) lindex[l]=tetra->n[l];
      k=0;
      do
      {Lcord(tetra->n[k]);
        A[k][0]=lcord->n[0]; A[k][1]=lcord->n[1];
        A[k][2]=lcord->n[2]; A[k++][3]=1;
```

```
        }
        while (k<4);
        spher(X);
        R2=(pow(X[0],2)+pow(X[1],2)+pow(X[2],2)-4*X[3])/4;
        R=pow(px+X[0]/2,2)+pow(py+X[1]/2,2)+pow(pz+X[2]/2,2);
        if (R<R2){
            addfac(lindex); BuscaT(j); T->n=0;
        }
        j++;
    }
    while(j<=Lentetra);
    deltetra(Lentetra);
    delcomfac(i);
    if(Lentetra==0) tTotetra();
    else Lentetra=tetraoldJoint();
    IniciaT(Lentetra);
    Esvaziafac();
    Esvaziat();
    Esvaziatetraold();
    i++;
  }
  while(i<=Dlcord->Len);
  ListaMath(Lentetra);
}
```

Essa função lê a lista *tetra* onde cada inteiro que a compõe contém a posição das triplas da lista *lcord*, ou seja, para cada vértice do tetraedro da lista *tetra*, temos os valores correspondentes para suas coordenadas na lista *lcord*.

A partir das coordenadas associadas a uma determinada quadrupla calculamos a esfera que passa por esse quatro pontos. Esses cálculos são realizados na função *spher* que listamos a seguir.

```
void __fastcall TFrmDelaunay::spher(double *X)
{ double B[4];
  B[0]=-pow(A[0][0],2)-pow(A[0][1],2)-pow(A[0][2],2);
  B[1]=-pow(A[1][0],2)-pow(A[1][1],2)-pow(A[1][2],2);
  B[2]=-pow(A[2][0],2)-pow(A[2][1],2)-pow(A[2][2],2);
  B[3]=-pow(A[3][0],2)-pow(A[3][1],2)-pow(A[3][2],2);
  LinearSolve(B,X,4);
}
```

A função *LinearSolve* resolve sistema lineares. Para a sua implementação foi escolhido o Método de Eliminação de Gauss pois não apresenta problemas com tempo de execução [23]. A implementação do algoritmo pode ser encontrada no Apêndice A (o método é simples e pode ser encontrado com facilidade em livros de Cálculo Numérico ou Análise de Métodos Numéricos). É importante salientar que o método deve adotar a estratégia do pivoteamento completo, pois apesar do ganho em performance sem pivoteamento ou com pivoteamento parcial os erros de arredondamento comprometem a qualidade da malha gerada.

Após calcularmos os valores associados à equação da esfera, verificamos se o novo ponto pertence ao interior da circunsfera. Se o ponto for detectado a função *addfac* é chamada.

Essa função tem como parâmetro o vetor *lindex* que é um elemento de *tetra*. O objetivo de *addfac* é montar todas as combinações três

a três dos quatro elementos que compõem *lindex* e adicioná-los a lista *fac*, ou seja a lista *fac* recebe as faces do novo tetraedro.

```
void __fastcall TFrmDelaunay::addfac(int *lindex)
{
  int imax=4,i,j,k=0;
  for (j=0;j<imax;j++)
  {
    k=0;
    fac=(Myfac *)malloc(sizeof(Myfac));
    for (i=0;i<imax;i++)
       if (i!=j) fac->n[k++]=lindex[i];
    Insertfac();
  }
}
```

Quando um ponto passa no teste de inclusão é preciso informar isso para o sistema, então o seu valor correspondente na lista *T* é alterado de *1* para *0*.

Após obtermos a nova lista chamamos a função *deltetra* que gera a lista *tetraold* que contém os tetraedros que não passaram no teste de inclusão.

```
void __fastcall TFrmDelaunay::deltetra(int Len)
{ int i,j=1;
  do
  { BuscaT(j);
    if(T->n==1) Inserttetraold(j, Len);
```

```
    j++;
  }
  while(j<=DT->Len);
}
```

Após todos esses processos é preciso ainda remover as triplas iguais, e esse processamento é feito na função *delcomfac*. O resultado dessa função é uma nova lista sem tetraedros repetidos. Essa lista é chamada *t.* A idéia desse processo é ordenar as listas e marcar as triplas iguais com *–1* para depois removê-las deixando apenas as triplas diferentes de *–1*.

```
void __fastcall TFrmDelaunay::delcomfac(int ind)
{Myfac *aux,*aux2;
  int n,i,j,z,r,indmax,indmin,max=-10000,min=10000;
  int ordfac[3];
  for (i=1;i<=Dfac->Len;i++)
  {max=-10000;min=10000;
    for(j=0;j<3;j++)
    { Fac(i);
      if(fac->n[j]>max) max=fac->n[j]; indmax=j;
      if(fac->n[j]<min) {min=fac->n[j]; indmin=j;}
    }
    ordfac[0]=fac->n[indmin];
    ordfac[1]=fac->n[3-indmin-indmax];
    ordfac[2]=fac->n[indmax];
    fac->n[0]=ordfac[0]; fac->n[1]=ordfac[1];  fac->n[2]=ordfac[2];
 }
 n=Dfac->Len;
 for (z=1;z<=n;z++)
 {   Fac(z);  aux=fac;
```

```
    for (r=1;r<=n;r++)
    { Fac(r); aux2=fac;
      if ((aux->n[0]==aux2->n[0]) &&
          (aux->n[1]==aux2->n[1]) &&
          (aux->n[2]==aux2->n[2]) &&
          (aux->n[0]!=-1) && (aux2->n[0]!=-1) &&
          (aux->n[1]!=-1) && (aux2->n[1]!=-1) &&
          (aux->n[2]!=-1) && (aux2->n[2]!=-1) && (z!=r))
      { aux2->n[0]=-1;aux2->n[1]=-1;aux2->n[2]=-1;
        aux->n[0]=-1;aux->n[1]=-1;aux->n[2]=-1;
      }
    }
 }
  Insertt(aux, n, ind);
}
```

As demais funções encontradas como *Insertt(Myfac *, int, int), Lcord(int), Tetra(int,int), BuscaT(int), IniciaT(int), tTotetra(void), tetraoldJoint(void), insertfac(), inserttetraold(int,int)*, etc, são funções de manipulação de lista e sua implementação pode ser totalmente encontrada no Apêndice A.

Já a função *ListaMath(int)* transfere o conteúdo da lista *tetra* para um arquivo formatado para o ambiente Mathematica de nome *lista.nb*. Dessa forma a lista de tetraedros gerada está pronta para ser manipulada diretamente dentro do Mathematica, ou seja é possível aplicá-la no módulo gráfico sem a necessidade de formatações adicionais.

A título de informação mostraremos como é o formato de uma lista dentro do ambiente Mathematica, mas sua implementação será

mostrada apenas no Apêndice A, uma vez que ela é obtida através da formatação da função *fprintf()*.

```
Notebook[{
Cell[BoxData[
   \(\(tetra = {{_, _, _, _}, {_, _, _, _},{_, _, _, _}, ..,
               {_, _, _, _},.....,{_, _, _, _}};\)\)], "Input"]
},
FrontEndVersion->"4.0 for Microsoft Windows",
ScreenRectangle->{{0, 800}, {0, 527}},
WindowSize->{792,500},
WindowMargins->{{0, Automatic}, {0, Automatic}}
]
```

Os valores de *ScreenRectangle*, *WindowSize* e *WindowMargins* podem assumir valores diferentes dos definidos acima. Os valores pré-definidos acima abrem a janela de forma maximizada para uma resolução de 800x600.

3.3 Módulo gráfico

O módulo gráfico está vinculado ao módulo matemático pois a discretização só estará concluída quando os vértices em comum da lista *tetra* e do *extent* forem excluídos. Esse processo é realizado em *elimina*.

```
elimina[tetra_List]:=Module[{aux,aux1},
   aux1={};
   Table[x=True; aux=tetra[[v]];
```

```
        For[u=1,u<5,u++, If[aux[[u]]>0&&aux[[u]]<9,x=False]];
        If[x==True,AppendTo[aux1,aux],];
    ,{v,1,Length[tetra]}];
   Return[aux1]]
```

Para visualizarmos a malha precisamos basicamente de duas funções: *Comb* e *Plot4Edro*.

```
Comb[tetra_list]:=
{ aux1={};
  Do[aux=tetra[[i]];
      aux2={{aux[[1],aux[[2]],aux[[3]]},
             {aux[[1],aux[[2]],aux[[4]]},
             {aux[[1],aux[[3]],aux[[4]]},
             {aux[[2],aux[[3]],aux[[4]]}};
  Map[AppendTo[aux1,#]&,aux2],{i,1,Length[tetra]}];
  h=Flatten[aux1]; H=Partitio[b,3];
}

Plot4Edro[lcord_,tetra_]:=
{  Comb[tetra];
   H=H/.{x_Integer->lcord[[x]]};
   H=Mpa[Polygon,H]; H=Map[Graphics3D,H];
   Show[H,Boxed->False];
}
```

Como mencionado anteriormente o algoritmo de Delaunay não conserva a fronteira quando os domínios utilizados são não convexos. Para sanar esse problema dois novos algoritmos foram implementados um

para retirar tetraedros das concavidades e outro para delimitar as fronteiras das estruturas internas. Essas funções são chamadas ao longo de *Plot4Edro*.

```
.... H=H/.{x_Integer->lcord[[x]]};
areas={{{_,_,_},{_,_,_},...,{_,_,_}},...,{{_,_,_},...,{_,_,_}}};
H=EliminaTetraedros[H,areas];
areas={{_,_,_},.....{_,_,_}};
H=SeparaRegioes[H,areas]; .....
```

A função *EliminaTetraedros* recebe como argumento a lista de tetraedros e as áreas de concavidade. Em seguida testamos se um tetraedro é formado exatamente por quatro pontos da lista de concavidades. Quando isso acontece o tetraedro é eliminado da lista de tetraedros. Já a função *SeparaRegioes* recebe como argumento a lista de tetraedros e as fronteiras e os pontos interiores das estruturas internas. Em seguida testa se o tetraedro é formado por quatro pontos dessa lista, em caso positivo esses pontos são transferidos para uma outra lista. Essas funções estão codificadas a seguir.

```
EliminaTetraedros[vertices:{{{_,_,_},{_,_,_},{_,_,_}}..},areas_]:=
Module[{vertex,regiao,pos,flag,jini,aux,listavertex},
   vertex=vertices; regiao=areas;  pos={}; aux=0; flag=0;
   Do[listavertex={{0,0,0},{0,0,0},{0,0,0}};
     aux=0; jini=i;
     listavertex=Union[ Flatten[vertices[[{i,i+1,i+2,i+3}]],1] ] ;
     Do[Do[
        If[MemberQ[areas[[w]],listavertex[[k]]],aux++;  Break[]],
         {w,1,Length[areas]}],
     {k,1,4}];
     If[aux==4,
```

```
        pos=Join[pos,{{jini},{jini+1},{jini+2},{jini+3}}];
        flag=1;],
      {i,1,Length[vertex],4}];
      indice=pos;
      If[flag>0,vertex=Delete[vertices,pos]];
     Return[vertex];    ]
SeparaRegioes[vertices:{{{_, _, _},{_, _, _},{_, _, _}}..},
 areas:{{_, _, _}..}] := Module[
  {vertex, regiao, flag, pos, jini, aux, listavertex},
   vertex = vertices; regiao = areas; pos = {};flag = 0; aux = 0;
   Do[listavertex = {{0, 0, 0}, {0, 0, 0}, {0, 0, 0}};
    aux = 0;jini = i;
    listavertex=Union[Flatten[vertices[[{i,i + 1,i + 2,i + 3}]],1]];
    Do[If[MemberQ[areas, listavertex[[k]] ], aux++], {k, 1, 4}];
     If[aux == 4,
      pos = Join[pos, {{jini}, {jini + 1}, {jini + 2}, {jini + 3}}];
      flag = 1]
    , {i, 1, Length[vertex], 4}];
   indice = pos;
   If[flag > 0, vertex = Delete[vertices, pos]];
   Return[vertex];]
```

3.4 Considerações sobre a Implementação

A conversão das funções do Mathematica para a linguagem C++ se mostrou eficiente quanto ao tempo de processamento, uma vez que os tempos de CPU foram aceitáveis para estruturas complexas como um

dente. O desempenho das otimizações citadas pode ser visto no final do capítulo 4.

As funções gráficas codificadas podem ser perfeitamente implementadas também em linguagem C++ [24,25], porém os recursos gráficos disponíveis no Mathematica simplificam bastante esse trabalho e mantêm um desempenho eficiente para visualizar o dente discretizado.

Contudo a implementação acima não é a única forma de se obter o algoritmo de Delaunay. Funções mais rápidas podem ser obtidas melhorando-se a manipulação das listas e talvez adotando uma estratégia de pivoteamento que seja tão eficiente quanto o pivoteamento completo e diminua o tempo de processamento.

No próximo capítulo apresentaremos exemplos que ilustram a construção do dente e como contornar problemas com ruptura de fronteira e outras degenerecências.

4 TESTES DE VALIDAÇÃO

Até agora fizemos apenas um estudo teórico do mapeamento do dente e da implementação do algoritmo de Delaunay.

Neste capítulo, faremos a validação desse algoritmo para a modelagem do dente e suas estruturas internas e veremos como contornar os problemas já mencionados. Discutiremos o tempo de processamento gasto para o algoritmo original, otimizado e para a implementação em C++.

4.1 Validação para as secções do dente

Devido a complexidade do domínio do dente é conveniente suprir os problemas em cada corte, ou seja inicialmente fazemos a discretização bidimensional de cada um dos cortes usando a função PlanarGraphPlot (pré existente no ambiente Mathematica.) que realiza a triangulação de Delaunay. Para dois cortes consecutivos aplicamos o algoritmo de Delaunay (em três dimensões) e comparamos esses cortes com essa secção. Essa comparação pode ser vista nas figuras 4.1.1 e 4.1.2.

Além da função PlanarGraphPlot existe a função TriangularSurfacePlot que realiza a discretização de um domínio tridimensional porém a malha obtida é triangular sendo gerada apenas na superfice da geometria.

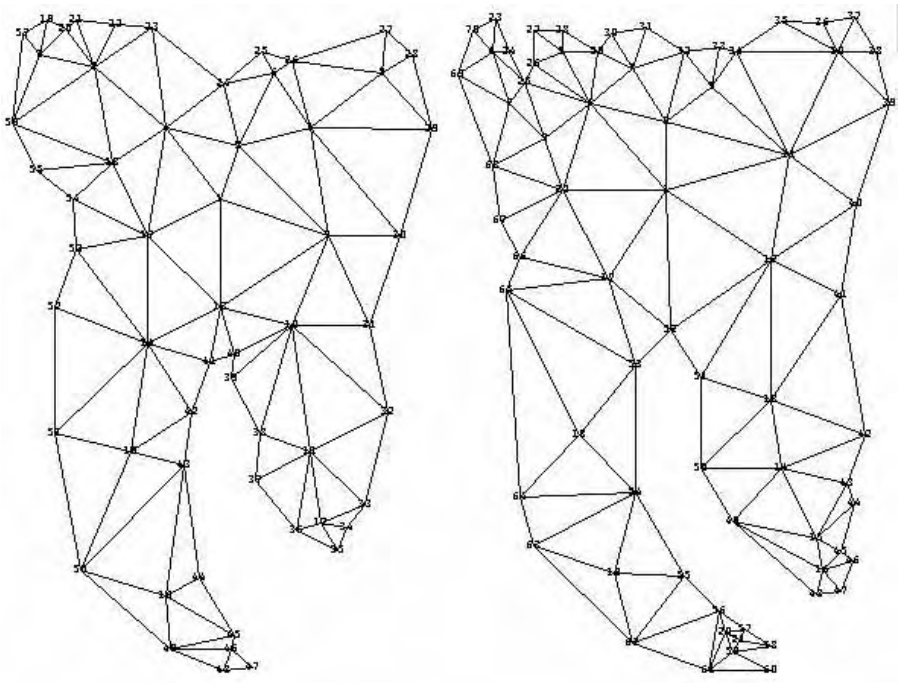

Figura 4.1.1: Cortes 5 e 6 respectivamente.

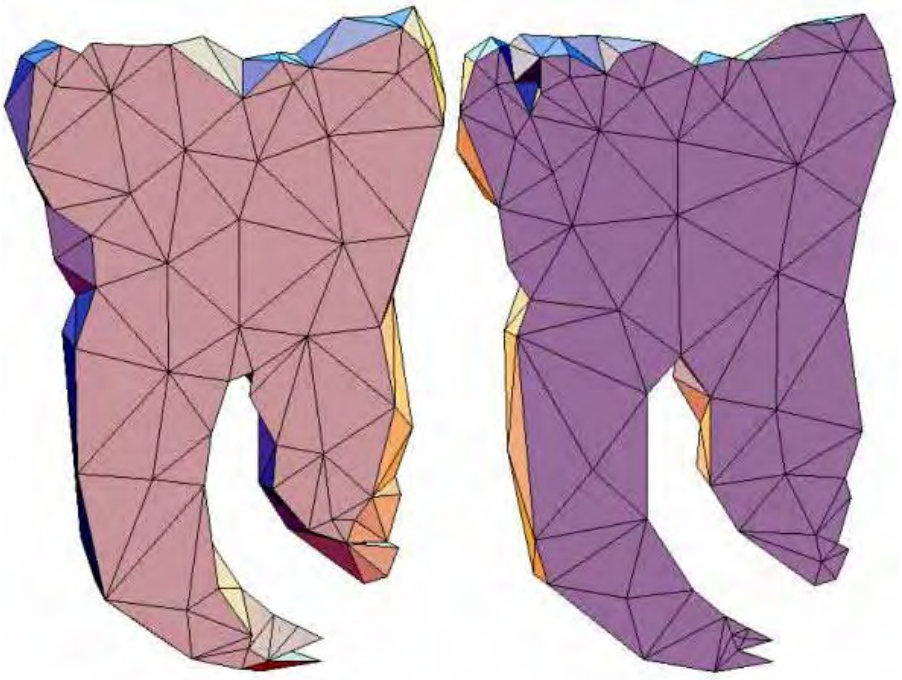

Figura 4.1.2: Secção 5-6 do dente

4.2 Validação para regiões não convexas.

Como já mencionamos anteriormente, o algoritmo de Delaunay mesmo enfrentando os problemas com a quebra de geometria, pode ser aplicado eficientemente para regiões não convexas.

Para isso, primeiro discretizamos todos os pontos de forma independente da geometria do sólido (figura 4.2.1) e em seguida deletamos os tetraedros cujos vértices são todos da parte convexa do dente. A aplicação desse algoritmo pode ser vista na figura 4.2.2.

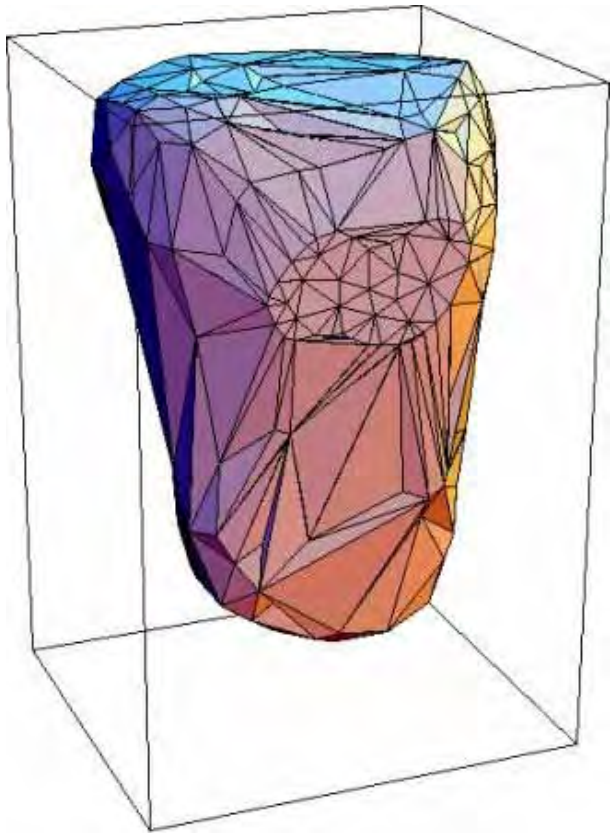

Figura 4.2.1: Discretização com ruptura da geometria do dente.

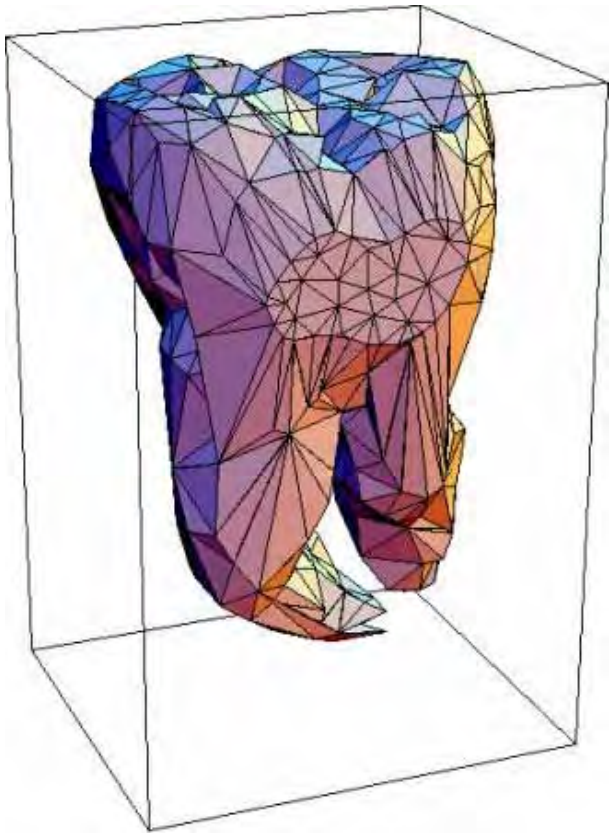

Figura 4.2.2: Discretização da figura 4.2.1 corrigida

Muitas vezes a distribuição dos pontos nas regiões convexas não está bem definida gerando tetraedros formados por pontos da concavidade e pontos internos. Esse tipo de problema pode ser corrigido de duas maneiras: inserção de novos pontos e perturbação dos pontos afetados. Essas perturbações são pequenas e não afetam a geometria do dente. Na figura 4.2.3 observamos esse tipo de problema.

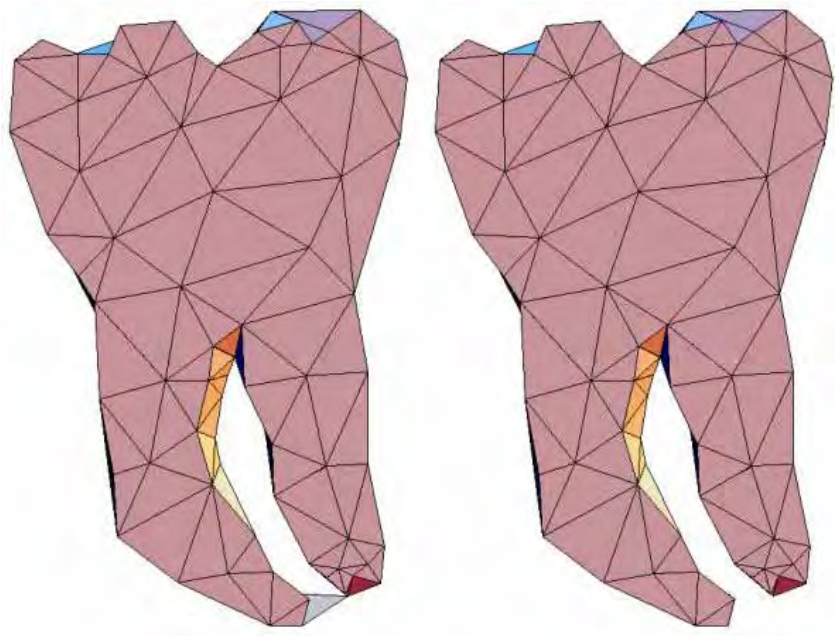

Figura 4.2.3: Correção de geometria com má distribuição dos pontos.

4.3 Validação para estruturas internas

Algum sólidos necessitam que suas estruturas internas sejam preservadas, a principal condição para obtermos essa representação é respeitar a ordem de inserção dos pontos que são tomados das regiões mais internas para as mais externas.

Isso nem sempre é suficiente; muitas vezes temos o rompimento das fronteiras das estruturas internas. Esse problema pode ser corrigido com a perturbação dos pontos afetados, mas isso nem sempre funciona pois na maioria das vezes provocamos a ruptura de outras fronteiras. O método que se mostrou totalmente confiável foi a inserção de novo ponto próximo ao ponto médio da fronteira afetada. Na figura 4.3.1 podemos observar a ruptura da estrutura interna (linha amarela) da raiz esquerda.

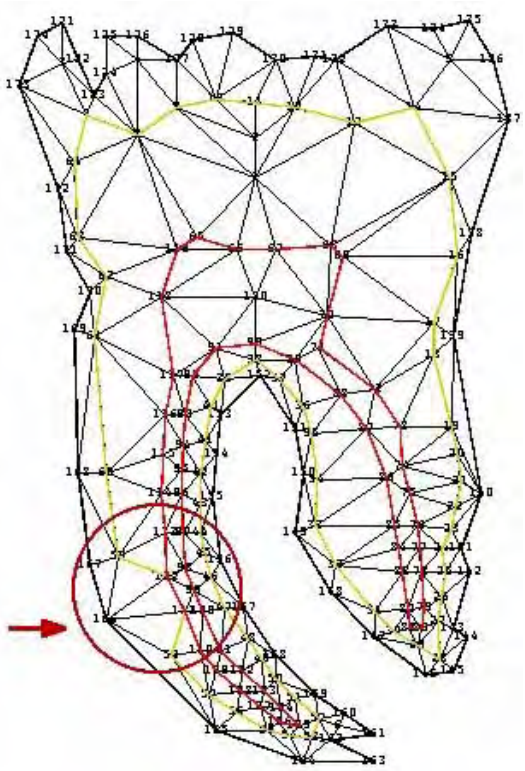

Figura 4.3.1: Estrutura interna com ruptura na raiz

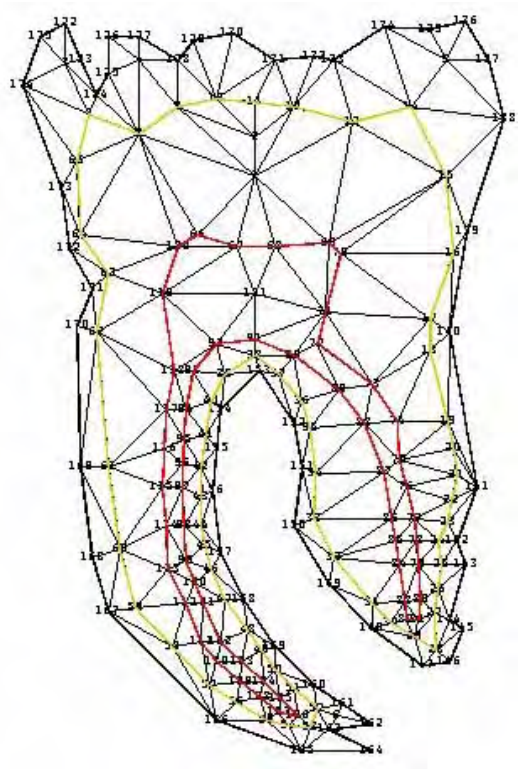

Figura 4.3.2: Correção para a figura 4.3.1 com a inserção de um novo ponto

4.4 Validação para a anatomia do dente

Após corrigirmos os problemas anteriormente apresentados podemos aplicar sem grandes dificuldades o algoritmo para todo o domínio tridimensional do dente e suas estruturas internas. Eventualmente ocorreram algumas degenerecências do tipo “overlaping” em vértices com duas coordenadas iguais. Como exemplo podemos citar a degenerecência ocorrida no tetraedro formado pelos vértices {{7,6,11.8}, {7,6,13}, {7.5,7,10.8},{7.5,7,14}}. Neste tetraedro temos os dois primeiros vértices com duas coordenadas repetidas (7,6). Essa degenerecência pode ser sanada perturbando-se o segundo vértice para o {6.999,6,13}. Contudo a ocorrência desses erros foi muito pequena e isso acontece pois domínios irregulares são menos sujeitos a esse tipo de ocorrência [26].

Podemos observar a eficiência da aplicação dos métodos descritos sobre o dente nas figuras abaixo.

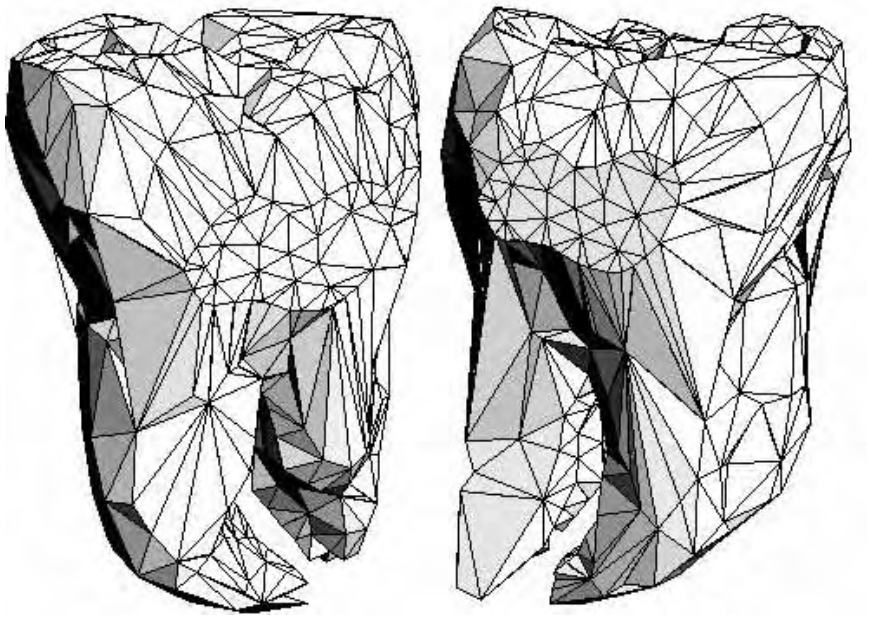

Figura 4.4.1: Visualização das faces do dente

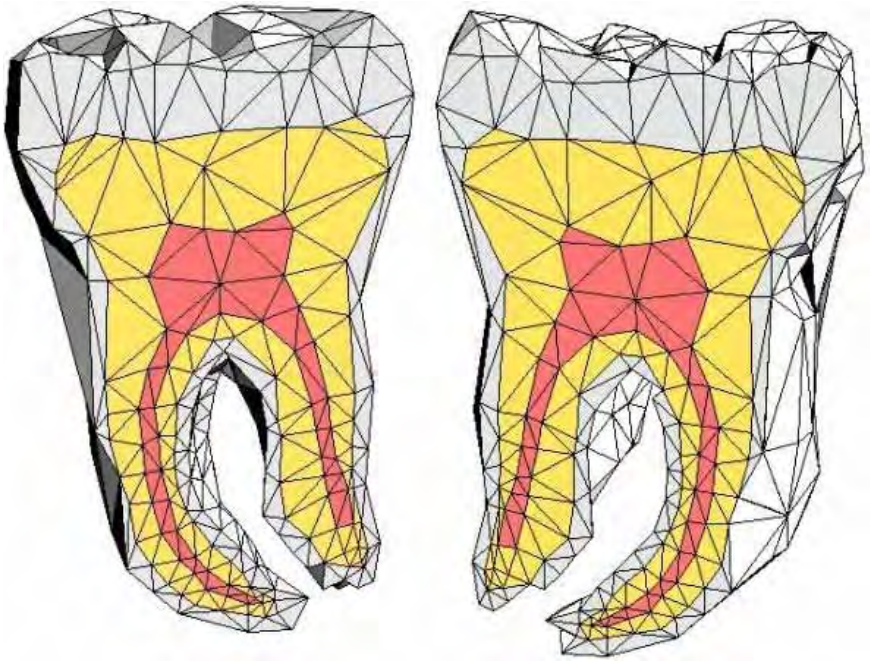

Figura 4.4.2: Visualização em corte da figura 4.4.1

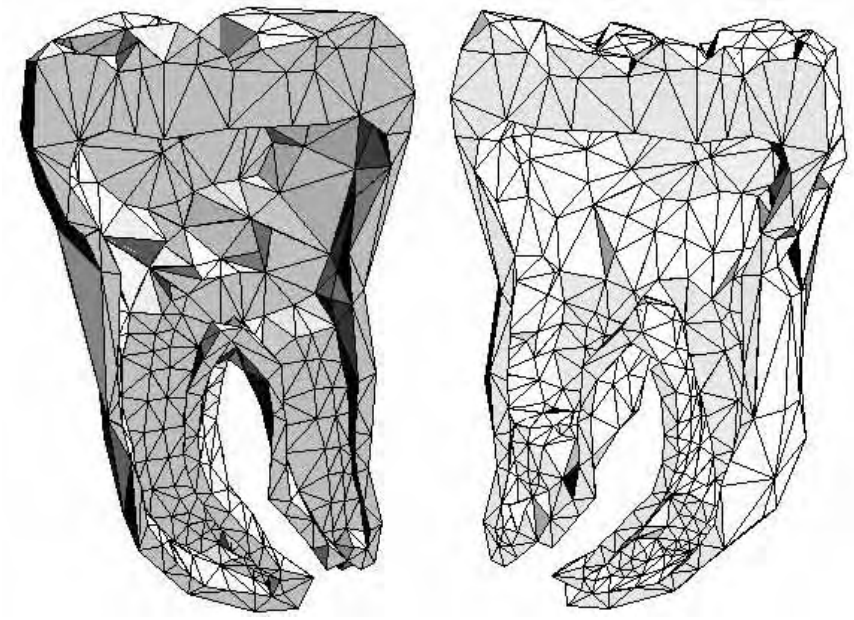

Figura 4.4.3: Visualização do esmalte sem as estruturas internas

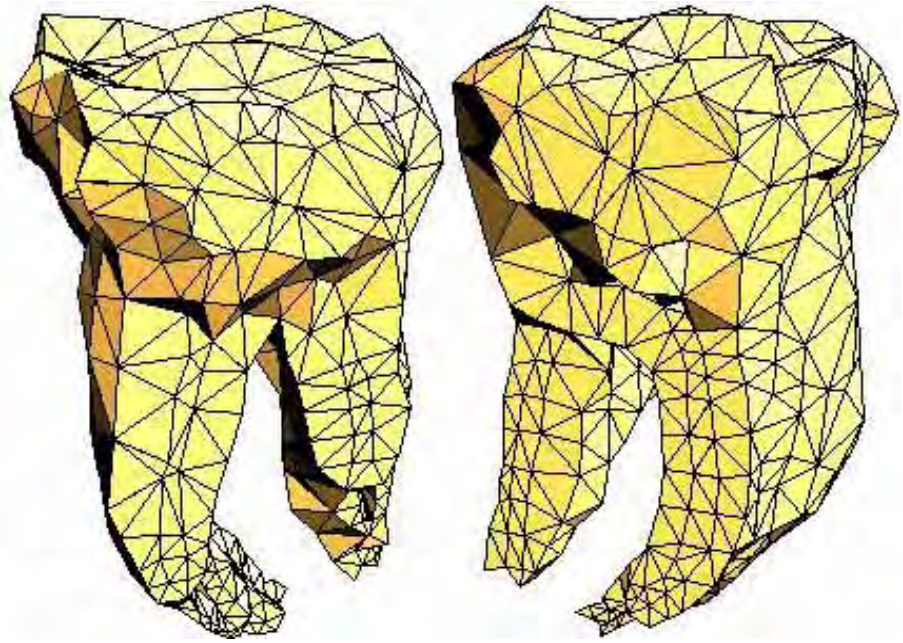

Figura 4.4.4: Estrutura interna da dentina

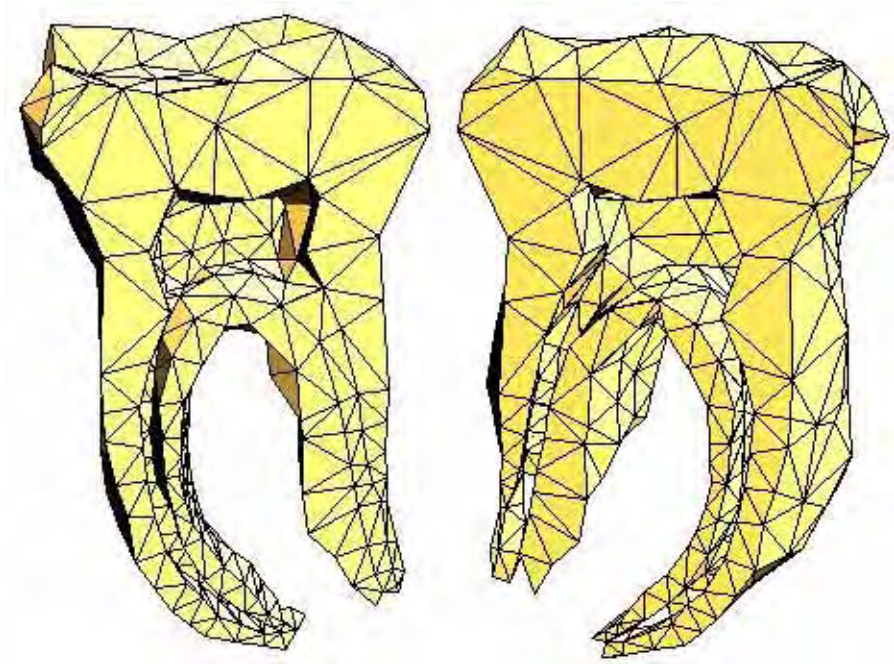

Figura 4.4.5: Visualização em corte da figura 4.44

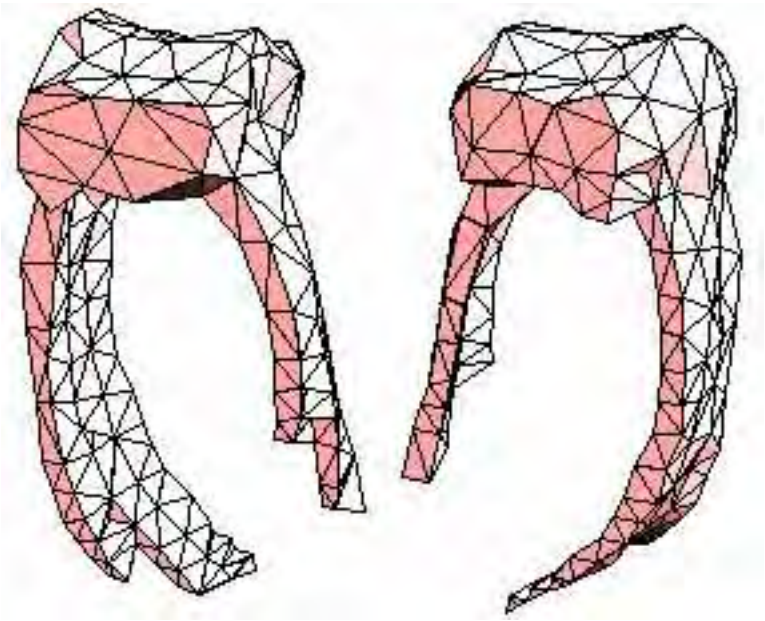

Figura 4.4.6: Estrutura interna da polpa

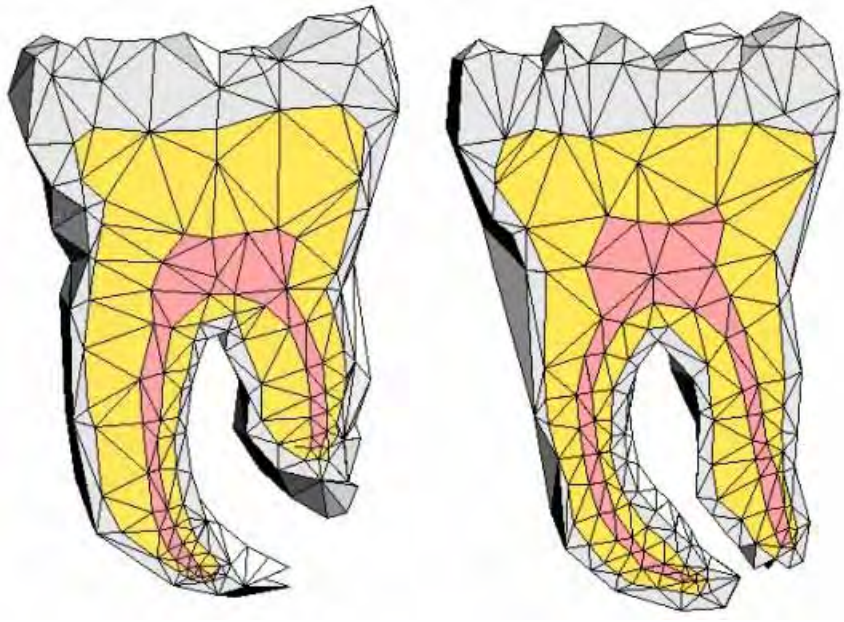

Figura 4.4.7: Secções (não consecutivas) do dente

4.5 Desempenho computacional

Apresentamos até agora três formas para o desenvolvimento do algoritmo de Delaunay: o algorimto original implementado no Mathematica [1], o algoritmo original ligeiramente otimizado e por fim a conversão desse último algoritmo para a linguagem C++.

Primeiramente medimos o desempenho do algoritmo original para alguns sólidos conhecidos (cubo, esfera e toro) e com o dente. Esse teste foi então repetido para as outras duas implementação (otimizado e C++). O tempo de processamento está parametrizado em segundos e pode ser observado na tabela abaixo.

Sólido	nº tetraedros	original	otimizado	C++
Cubo	6	0.71 s	0.55 s	0 s
Esfera	297	88.65 s	79.4 s	12 s
Toro	561	300.05 s	234.97 s	38 s
dente(superfície)	3251	5549.41 s	4448.47 s	1126 s
dente(completo)	7094	36735.3 s	32950.2 s	9418 s

Essas avaliações foram obtidas num AMD K6-2 500 Mhz, 96 Mb sempre após reiniciá-lo para evitar que vestígios dos testes anteriores não interfiram na medição dos tempos.

A análise dos resultados e perspectivas em relação a continuidade do projeto serão apresentadas no próximo capítulo.

5 CONCLUSÃO

A estrutura obtida ao final do desenvolvimento do projeto confirma o objetivo do trabalho, que é gerar em computador um modelo geométrico apropriado para uso em simulações, uma vez que o resultado obtido representa um dente de forma realística, levando em conta todas as irregularidades de sua anatomia e preservando suas regiões internas.

O algoritmo de Delaunay se mostrou eficiente em aplicações realísticas uma vez que a discretização foi executada sobre um domínio complexo, cheio de assimetrias e concavidades.

As técnicas de otimização adotadas como o aperfeiçoamento na manipulação de algumas listas e a conversão do algoritmo para a linguagem C++ se mostraram bastante produtivas, uma vez que o sistema foi capaz de trabalhar com uma malha bastante grande com um desempenho de processamento aceitável.

Os resultados obtidos nesse trabalho podem ser bastante úteis em aplicações odontológicas uma vez que nos casos de simulação publicados até agora na literatura especializada [27], trabalha-se normalmente com representações planas ou modelos que simplificam bastante as geometrias de dentes reais.

5.1 Perspectivas

Os bons resultados obtidos até agora proporcionam uma grande perspectiva em relação à continuidade desse projeto. Umas das possibilidades é refinar a malha inserindo novos pontos nos baricentros dos tetraedros da malha existente. Outra possibilidade é obter e analisar valores de tensão que o modelo pode fornecer quando empregamos simulações realistas. Podemos também refinar a malha mapeando mais pontos interiores. Contudo essas operações consomem muito tempo de CPU sendo necessário aplicar outras técnicas de otimização.

Algumas dessas possibilidades de aperfeiçoamento são:

- Conversão das funções *Comb, EliminaTetraedros* e *SeparaRegiões* para a linguagem C++;

- Melhorar a forma de armazenamento construindo uma estrutura que suporte ao mesmo tempo os tetraedros e suas respectivas coordenadas diminuindo assim o número de listas utilizadas;

- Evitar as constantes desalocações dos nós da lista, adotando uma estratégia de marcá-las como vazias e reaproveitá-las sem a necessidade de se alocar novas posições de memória (essa técnica foi empregada com sucesso apenas na lista de tetraedros);

- Para função LinearSolve devemos pesquisar uma outra estratégia de pivoteamento que não execute tantos testes quanto o pivoteamento completo e que não forneça muitos erros de arredondamento, quanto a velocidade, o pivoteamento parcial é muito superior, mas a malha gerada não apresenta uma boa qualidade. Outra solução seria adotar outra técnica mais rápida que o Método de Eliminação de Gauss com pivoteamento completo;

- Migração para uma plataforma de Hardware de maior capacidade de memória e processamento;

- Desenvolvimento de versões para os ambientes Linux ou Unix, pois ambos trabalham com a linguagem C e o pacote Mathematica e possuem um gerenciamento de memória muito superior ao Windows.

APÊNDICE A

A.1 Código Fonte da Implementação em C++

A implementação do Algoritmo de Delaunay em linguagem C++ não é tão trivial quanto no Ambiente Mathematica pois não existem tantas funções pré-existentes em C, principalmente em relação às listas que devem ser totalmente implementadas.

A.2 Estruturas de Armazenamento

As listas utilizadas para armazenamento seguem sempre o mesmo padrão, ou seja, possuem um campo para guardar a posição de cada nó na lista, e outro campo relativo aos pontos manipulados. Cada uma dessas estruturas possui um descritor que indica o início e o fim da lista. Os tipos de listas são Mytetra que armazena a lista de tetraedros, Mylcord para os pontos do sólido, Myfac para as novas faces geradas e MyT que indica os tetraedros que passaram no teste de inclusão.

```
typedef struct Etetra
{int id, n[4];
  struct Etetra *back, *next;
}Mytetra;
Mytetra *tetra,*tetraold,*t;

typedef struct
{int Len;
  Mytetra *Inicio, *Fim;
```

```
}Desctetra;
Desctetra *Dtetra, *Dtetraold, *Dt;

typedef struct Elcord
{ int id;
  double n[3];
  struct Elcord *back, *next;
}Mylcord;
Mylcord *lcord;

typedef struct
{ int Len;
  Mylcord *Inicio, *Fim;
}Desclcord;
Desclcord *Dlcord;

typedef struct Efac
{ int id, n[3];
  struct Efac *back, *next;
}Myfac;
Myfac *fac;

typedef struct
{ int Len;
  Myfac *Inicio, *Fim;
}Descfac;
Descfac *Dfac,*Dordfac;

typedef struct ET
{ int id, n;
  struct ET *back, *next;
```

```
}MyT;
MyT *T;

typedef struct
{ int Len;
  MyT *Inicio, *Fim;
}DescT;
DescT *DT;
```

A.3 Protótipo das funções

*void __fastcall Pivoteamento(double *B, int i,int n);* - Executa o pivoteamento para o Método da Eliminação de Gauss em LinearSolve.

*void __fastcall LinearSolve(double *B, double *X, int n);* - Executa o Método da Eliminação de Gauss.

*void __fastcall spher(double *X);* - Calcula a circunsfera que passa pelos pontos de um tetraedro.

*void __fastcall addfac(int *lindex);* - Adiciona as faces do novo tetraedro.

void __fastcall deltetra(int Len); - Deleta todas as faces que já estão incluídas na lista fac.

void __fastcall delcomfac(int ind); - Remove as triplas duplicadas na lista fac.

void __fastcall principal(void); - Essa função lê a lista lcord e tetra e executa as funções subordinadas que realizam a discretização.

void __fastcall CreateListas(void); - Aloca memória para o descritor das listas.

void __fastcall DestroeListas(void); - Libera o espaço de memória utilizado pelas listas e pelos descritores.

void __fastcall EsvaziaListas(void); - Libera o espaço de memória utilizado apenas pelas listas.

void __fastcall IniciaTetra(void); - Abre o arquivo tetra.dat e transfere os valores para a lista tetra.

*void __fastcall IniciaLcord(char *filename);* - Abre o arquivo *filename* e transfere os valores para a lista lcord.

void __fastcall IniciaT(int n); - Inicia cada posição da lista T com 1 (um). Essa lista tem o mesmo comprimento da lista tetra.

void __fastcall Lcord(int i); - Retorna a posição i da lista lcord.

void __fastcall Tetra(int i,int Len); - Retorna a posição i da lista tetra.

void __fastcall BuscaT(int i); - Retorna a posição i da lista T.

void __fastcall Buscat(int i); - Retorna a posição i da lista t. Essa lista armazena os novos tetraedros.

void __fastcall Fac(int i); - Retorna a posição i da lista fac.

void __fastcall tTotetra(void); - Transfera o valor da lista t para tetra.

int __fastcall tetraoldJoint(void); - Essa função une as listas tetraold e t, retornando o tamanho da lista resultante.

void __fastcall Esvaziafac(void); - Libera a memória apenas da lista fac.

void __fastcall Esvaziat(void); - Libera a memória apenas da lista t.

void __fastcall Esvaziatetraold(void); - Libera a memória apenas da lista tetraold.

void __fastcall ListaMath(int Lentetra); - Transfere os tetraedros gerados para a lista lista.nb .

void __fastcall Insertfac(void); - Aloca e insere novas faces na lista fac.

void __fastcall Inserttetraold(int j, int Len); - Aloca e insere novas faces na lista tetraold.

*void __fastcall Insertt(Myfac *aux,int n, int ind);* - Aloca e insere novas faces na lista t.

*void __fastcall Abrir1Click(TObject *Sender);* - Abre uma caixa de diálogo para selecionar a lista de coordenadas.

*void __fastcall Executar1Click(TObject *Sender);* - Marca o tempo inicial em seguida chama a função *Principal()* e armazena o tempo final para cálculo do tempo total de processamento.

A.4 Implementação das funções

```
void __fastcall TFrmDelaunay::Pivoteamento(double *B,int i,int n)
{
    int p,k,j;
    double aux=0.0;
     p=i;
     for (k=i;k<n;k++)
       if (fabs(A[p][i])<fabs(A[k][i])) p=k;
    if (i!=p)
    {
       for (j=i;j<n;j++)
       {
              aux=A[i][j];
              A[i][j]=A[p][j];
              A[p][j]=aux;
       }
       aux=B[i]; B[i]=B[p]; B[p]=aux;
    }
```

```
}

void __fastcall LinearSolve(double *B,double *X,int n)
{
     int i,l,k,j;
     double mult,aux=0.0,Aaux[4][4];
    char msg[60];
    for (k=0;k<n;k++)
    {
      for(j=0;j<n;j++)  Aaux[k][j]=A[k][j];
    }
     for (k=0;k<n-1;k++)
     {
        Pivoteamento(B,k,n);
        for (i=k+1;i<n;i++)
         { mult=1/A[k][k];
           mult=mult*A[i][k];
           A[i][k]=mult;
            for (j=k+1;j<n;j++)
           {
              A[i][j]=A[i][j]-mult*A[k][j];
           }
           B[i]=B[i]-mult*B[k];
        }
     }
    if (fabs(A[n-1][n-1])==0.0)
    {
      Canvas->Brush->Color = clBtnFace;
      Canvas->TextOut(10,10+20*(erro),"A matriz de está mal
                                  condicionada.");
      A[n-1][n-1]=0.00000001;
```

```
    }
    X[n-1]=B[n-1]/A[n-1][n-1];
    for (l=n-2;l>=0;l--)
    {
            for(j=l+1;j<n;j++)   aux=aux+A[l][j]*X[j];
            X[l]=(B[l]-aux)/A[l][l];
            aux=0.0;
    }
}

void __fastcall TFrmDelaunay::spher(double *X)
{
  double B[4];
  B[0]=-pow(A[0][0],2)-pow(A[0][1],2)-pow(A[0][2],2);
  B[1]=-pow(A[1][0],2)-pow(A[1][1],2)-pow(A[1][2],2);
  B[2]=-pow(A[2][0],2)-pow(A[2][1],2)-pow(A[2][2],2);
  B[3]=-pow(A[3][0],2)-pow(A[3][1],2)-pow(A[3][2],2);
  LinearSolve(B,X,4);
}

void __fastcall TFrmDelaunay::addfac(int *lindex)
{
  int imax,i,j,k;
  imax=4; k=0;
  for (j=0;j<imax;j++)
  {
    k=0;
    fac=(Myfac *)malloc(sizeof(Myfac));
    for (i=0;i<imax;i++)
    {
      if (i!=j)
```

```
        {
          fac->n[k++]=lindex[i];
        }
      }
      Insertfac();
    }
}

void __fastcall TFrmDelaunay::deltetra(int Len)
{
  int i,j=1;
  do
  {
    BuscaT(j);
    if(T->n==1) Inserttetraold(j,Len);
    j++;
  }
  while(j<=DT->Len);
}

void __fastcall TFrmDelaunay::delcomfac(int ind)
{
  int max=-10000,min=10000;
  Myfac *aux,*aux2;
  int n,i,j,z,r,indmax,indmin;
  int ordfac[3];
  for (i=1;i<=Dfac->Len;i++)
  {
    max=-10000;min=10000;
    for(j=0;j<3;j++)
    {
```

```
      Fac(i);
      if(fac->n[j]>max)
      {
        max=fac->n[j]; indmax=j;
      }
      if(fac->n[j]<min)
      {
        min=fac->n[j]; indmin=j;
      }
    }
    ordfac[0]=fac->n[indmin];
    ordfac[1]=fac->n[3-indmin-indmax];
    ordfac[2]=fac->n[indmax];
    fac->n[0]=ordfac[0];
    fac->n[1]=ordfac[1];
    fac->n[2]=ordfac[2];
  }
  n=Dfac->Len;
  for (z=1;z<=n;z++)
  {
    Fac(z);
    aux=fac;
    for (r=1;r<=n;r++)
    {
      Fac(r);
      aux2=fac;
      if ((aux->n[0]==aux2->n[0]) &&
          (aux->n[1]==aux2->n[1]) &&
          (aux->n[2]==aux2->n[2]) &&
          (aux->n[0]!=-1) && (aux2->n[0]!=-1) &&
          (aux->n[1]!=-1) && (aux2->n[1]!=-1) &&
```

```
          (aux->n[2]!=-1) && (aux2->n[2]!=-1) && (z!=r))
        {
          aux2->n[0]=-1;aux2->n[1]=-1;aux2->n[2]=-1;
          aux->n[0]=-1;aux->n[1]=-1;aux->n[2]=-1;
        }
      }
    }
    Insertt(aux,n,ind);
}

void __fastcall TFrmDelaunay::principal(void)
{
  int i,j,k,l,lindex[4],Lentetra;
  double R2,R,px,py,pz,X[4];
  i=9;
  Lentetra=Dtetra->Len;
  do
  {
    Lcord(i);
    px=lcord->n[0]; py=lcord->n[1]; pz=lcord->n[2];
    j=1;
    do
    {
      Tetra(j,Lentetra);
      for(l=0; l<4; l++)
        lindex[l]=tetra->n[l];
      k=0;
      do
      {
       Lcord(tetra->n[k]);
       A[k][0]=lcord->n[0]; A[k][1]=lcord->n[1];
```

```
          A[k][2]=lcord->n[2]; A[k++][3]=1;
        }
        while (k<4);
        spher(X);
        R2=(pow(X[0],2)+pow(X[1],2)+pow(X[2],2)-4*X[3])/4;
      R=pow((px+X[0]/2),2)+pow((py+X[1]/2),2)+pow((pz+X[2]/2),2);
        if (R<R2)
        {
          addfac(lindex);
          BuscaT(j);
          T->n=0;
        }
        j++;
      }
      while(j<=Lentetra);
      deltetra(Lentetra);
      delcomfac(i);
      if(Lentetra==0) tTotetra();
      else Lentetra=tetraoldJoint();
      IniciaT(Lentetra);
      Esvaziafac();
      Esvaziat();
      Esvaziatetraold();
      i++;
    }
    while(i<=Dlcord->Len);
    ListaMath(Lentetra);
}

void __fastcall TFrmDelaunay::CreateListas(void)
{ aloca=1;
```

```
  Dtetra=(Desctetra *)malloc(sizeof(Desctetra));
  Dtetra->Len=0;
  Dtetra->Inicio=Dtetra->Fim=NULL;
  Dlcord=(Desclcord *)malloc(sizeof(Desclcord));
  Dlcord->Len=0;
  Dlcord->Inicio=Dlcord->Fim=NULL;

  Dfac=(Descfac *)malloc(sizeof(Descfac));
  Dfac->Len=0;
  Dfac->Inicio=Dfac->Fim=NULL;
  DT=(DescT *)malloc(sizeof(DescT));
  DT->Len=0;
  DT->Inicio=DT->Fim=NULL;
  Dtetraold=(Desctetra *)malloc(sizeof(Desctetra));
  Dtetraold->Len=0;
  Dtetraold->Inicio=Dtetraold->Fim=NULL;
  Dt=(Desctetra *)malloc(sizeof(Desctetra));
  Dt->Len=0;
  Dt->Inicio=Dt->Fim=NULL;
}

void __fastcall TFrmDelaunay::DestroeListas(void)
{
  int i;
  Mytetra *aux;
  Mylcord *aux2;
  MyT *aux3;
  Myfac *aux4;
  if ((Dtetra->Inicio!=NULL) && (Dtetra->Fim!=NULL))
  {
    for (i=1;i<=Dtetra->Len;i++)
```

```
  {
    aux=Dtetra->Inicio;
    Dtetra->Inicio=aux->next;
    if (Dtetra->Inicio!=NULL) Dtetra->Inicio->back=NULL;
    free(aux);
  }
  free(Dtetra);
}

if ((Dlcord->Inicio!=NULL) && (Dlcord->Fim!=NULL))
{
  for (i=1;i<=Dlcord->Len;i++)
  {
    aux2=Dlcord->Inicio;
    Dlcord->Inicio=aux2->next;
    if (Dlcord->Inicio!=NULL)
      Dlcord->Inicio->back=NULL;
    free(aux2);
  }
  free(Dlcord);
}
if ((DT->Inicio!=NULL) && (DT->Fim!=NULL))
{
  for (i=1;i<=DT->Len;i++)
  {
    aux3=DT->Inicio;
    DT->Inicio=aux3->next;
    if (DT->Inicio!=NULL)
      DT->Inicio->back=NULL;
    free(aux3);
  }
```

```
    free(DT);
  }

  if ((Dfac->Inicio!=NULL) && (Dfac->Fim!=NULL))
  {
    for (i=1;i<=Dfac->Len;i++)
    {
      aux4=Dfac->Inicio;
      Dfac->Inicio=aux4->next;
      if (Dfac->Inicio!=NULL) Dfac->Inicio->back=NULL;
      free(aux4);
    }
    free(Dfac);
  }
}

void __fastcall TFrmDelaunay::EsvaziaListas(void)
{
  int i;
  Mytetra *aux;
  Mylcord *aux2;
  MyT *aux3;
  Myfac *aux4;
  if ((Dtetra->Inicio!=NULL) && (Dtetra->Fim!=NULL))
  {
    for (i=1;i<=Dtetra->Len;i++)
    {
      aux=Dtetra->Inicio;
      Dtetra->Inicio=aux->next;
      if (Dtetra->Inicio!=NULL)
        Dtetra->Inicio->back=NULL;
```

```
      free(aux);
    }
  }
  if ((Dlcord->Inicio!=NULL) && (Dlcord->Fim!=NULL))
  {
    for (i=1;i<=Dlcord->Len;i++)
    {
      aux2=Dlcord->Inicio;
      Dlcord->Inicio=aux2->next;
      if (Dlcord->Inicio!=NULL)
        Dlcord->Inicio->back=NULL;
      free(aux2);
    }
  }
  if ((DT->Inicio!=NULL) && (DT->Fim!=NULL))
  {
    for (i=1;i<=DT->Len;i++)
    {
      aux3=DT->Inicio;
      DT->Inicio=aux3->next;
      if (DT->Inicio!=NULL)
        DT->Inicio->back=NULL;
      free(aux3);
    }
  }
  if ((Dfac->Inicio!=NULL) && (Dfac->Fim!=NULL))
  {
    for (i=1;i<=Dfac->Len;i++)
    {
      aux4=Dfac->Inicio;
      Dfac->Inicio=aux4->next;
```

```
        if (Dfac->Inicio!=NULL)
          Dfac->Inicio->back=NULL;
        free(aux4);
      }
    }
}

void __fastcall TFrmDelaunay::IniciaTetra(void)
{
  FILE *ftetra;
  int k=0,x,y,z,w;
  char ch;
  Dtetra=(Desctetra *)malloc(sizeof(Desctetra));
  Dtetra->Len=0;
  Dtetra->Inicio=Dtetra->Fim=NULL;
  ftetra=fopen("tetra.dat","r+");
  k=0;
  do
  {
   ch=getc(ftetra);
   if (isdigit(ch))
   {
     if (k==0) x=(int)ch-int('0');
     if (k==1) y=(int)ch-int('0');
     if (k==2) z=(int)ch-int('0');
     if (k==3) w=(int)ch-int('0');
     k++;
   }
   if (k==4)
   {
     k=0;
```

```
      tetra=(Mytetra *)malloc(sizeof(Mytetra));
      if (Dtetra->Len==0)
      {
        Dtetra->Len++;
        tetra->id=Dtetra->Len;
        tetra->n[0]=x; tetra->n[1]=y; tetra->n[2]=z; tetra->n[3]=w;
        tetra->back=tetra->next=NULL;
        Dtetra->Inicio=Dtetra->Fim=tetra;
      }
      else
      {
        Dtetra->Len++;
        tetra->id=Dtetra->Len;
        tetra->n[0]=x; tetra->n[1]=y; tetra->n[2]=z; tetra->n[3]=w;
        tetra->next=NULL; tetra->back=Dtetra->Fim;
        Dtetra->Fim->next=tetra;
        Dtetra->Fim=tetra;
      }
    }
  }
  while (!feof(ftetra));
  fclose(ftetra);
}

void __fastcall TFrmDelaunay::IniciaLcord(char *filename)
{
  FILE *flcord,*fptmp;
  int k=0,j=0,w=0;
  char ch,chold,numb[20];
  double x,y,z;
  flcord=fopen(filename,"r+");
```

```
fptmp=fopen("lcord.tmp","w+");
j=0;
do
{
  ch=getc(flcord);
  if ((isdigit(ch)) || (ch=='{') || (ch=='}') || (ch=='.') || (ch==','))
  {
    if (ch==',') j++;
    fprintf(fptmp,"%c",ch);
    if (j==18)
    {
      fprintf(fptmp,"\n");
      j=0;
    }
  }
}
while(!feof(flcord));
fclose(flcord);
fclose(fptmp);
remove(filename);
rename("lcord.tmp",filename);
remove("lcord.tmp");
ch='\0';
Dlcord=(Desclcord *)malloc(sizeof(Desclcord));
Dlcord->Len=0;
Dlcord->Inicio=Dlcord->Fim=NULL;
flcord=fopen(filename,"r+");
k=0,w=0;
do
{
 chold=ch;
```

```
ch=getc(flcord);
if ( (isdigit(ch)) || (ch=='.') || (ch==',') || (ch=='}') )
{
  if ( ((isdigit(chold)) || (chold=='.')) && ((ch==',') || (ch=='}')) )
  {
    numb[w]='\0';
    w=0;
    if (k==0) x=atof(numb);
    if (k==1) y=atof(numb);
    if (k==2) z=atof(numb);
    k++;
  }
  else
  {
    if ((isdigit(ch)) || (ch=='.')) numb[w++]=ch;
  }
}
if (k==3)
{
  k=0;
  lcord=(Mylcord *)malloc(sizeof(Mylcord));
  if (Dlcord->Len==0)
  {
    Dlcord->Len++;
    lcord->id=Dlcord->Len;
    lcord->n[0]=x; lcord->n[1]=y; lcord->n[2]=z;
    lcord->back=lcord->next=NULL;
    Dlcord->Inicio=Dlcord->Fim=lcord;
  }
  else
  {
```

```
          Dlcord->Len++;
          lcord->id=Dlcord->Len;
          lcord->n[0]=x; lcord->n[1]=y; lcord->n[2]=z;
          lcord->next=NULL; lcord->back=Dlcord->Fim;
          Dlcord->Fim->next=lcord;
          Dlcord->Fim=lcord;
        }
     }
   }
   while (!feof(flcord));
   fclose(flcord);
}

void __fastcall TFrmDelaunay::IniciaT(int n)
{
   MyT *aux;
   int i,len;
   DT=(DescT *)malloc(sizeof(DescT));
   len=DT->Len=0;
   DT->Inicio=DT->Fim=NULL;
   for (i=1;i<=n;i++)
   {
      if ((len==0) || (i>len))
        T=(MyT *)malloc(sizeof(MyT));
      if (DT->Len==0)
      {
        DT->Len++;
        T->id=DT->Len;
        T->n=1;
        T->back=T->next=NULL;
        DT->Inicio=DT->Fim=T;
```

```
      }
      else
      {
        if (i<=len) T->n=1;
        else
        {
          DT->Len++;
          T->id=DT->Len;
          T->n=1;
          T->next=NULL; T->back=DT->Fim;
          DT->Fim->next=T;
          DT->Fim=T;
        }
      }
  }
  aux=DT->Inicio;
}

void __fastcall TFrmDelaunay::Lcord(int i)
{
  int j;
  lcord=Dlcord->Inicio;
  for (j=1;j<=Dlcord->Len;j++)
  {
    if (lcord->id==i) break;
    lcord=lcord->next;
  }
}

void __fastcall TFrmDelaunay::Tetra(int i,int Len)
{
```

```
  int j;
  tetra=Dtetra->Inicio;
  for (j=1;j<=Len;j++)
  {
    if (tetra->id==i) break;
    tetra=tetra->next;
  }
}

void __fastcall TFrmDelaunay::Buscat(int i)
{
  int j;
  t=Dt->Inicio;
  for (j=1;j<=Dt->Len;j++)
  {
    if (t->id==i)  break;
    t=t->next;
  }
}

void __fastcall TFrmDelaunay::BuscaT(int i)
{
  int j;
  T=DT->Inicio;
  for (j=1;j<=DT->Len;j++)
  {
    if (T->id==i)   break;
    T=T->next;
  }
}
```

```
void  __fastcall TFrmDelaunay::Fac(int i)
{
  int j;
  fac=Dfac->Inicio;
  for (j=1;j<=Dfac->Len;j++)
  {
    if (fac->id==i)
      break;
    fac=fac->next;
  }
 }

void __fastcall TFrmDelaunay::tTotetra(void)
{
  int l;
  for (l=1;l<=Dt->Len;l++)
  {
    Buscat(l);
    tetra=(Mytetra *)malloc(sizeof(Mytetra));
    tetra->n[0]=t->n[0];
    tetra->n[1]=t->n[1];
    tetra->n[2]=t->n[2];
    tetra->n[3]=t->n[3];
    if (Dtetra->Len==0)
    {
      Dtetra->Len++;
      tetra->id=Dtetra->Len;
      tetra->back=tetra->next=NULL;
      Dtetra->Inicio=Dtetra->Fim=tetra;
    }
    else
```

```
    {
      Dtetra->Len++;
      tetra->id=Dtetra->Len;
      tetra->next=NULL; tetra->back=Dtetra->Fim;
      Dtetra->Fim->next=tetra;
      Dtetra->Fim=tetra;
    }
  }
}

int __fastcall TFrmDelaunay::tetraoldJoint(void)
{
  int Lentetraaux,l;
  tetra=Dtetra->Inicio;
  Lentetraaux=Dtetra->Len;
  tetraold=Dtetraold->Inicio;
  t=Dt->Inicio;
  for (l=1;l<=(Dtetraold->Len+Dt->Len);l++)
  {
    if (l<=Lentetraaux)
    {
      tetra->id=l;
      if (l<=Dtetraold->Len)
      {
        tetra->n[0]=tetraold->n[0];
        tetra->n[1]=tetraold->n[1];
        tetra->n[2]=tetraold->n[2];
        tetra->n[3]=tetraold->n[3];
      }
      else
      {
```

```
      tetra->n[0]=t->n[0];
      tetra->n[1]=t->n[1];
      tetra->n[2]=t->n[2];
      tetra->n[3]=t->n[3];
    }
    tetra=tetra->next;
  }
  else
  {
    tetra=(Mytetra *)malloc(sizeof(Mytetra));
    if (l<=Dtetraold->Len)
    {
      tetra->n[0]=tetraold->n[0];
      tetra->n[1]=tetraold->n[1];
      tetra->n[2]=tetraold->n[2];
      tetra->n[3]=tetraold->n[3];
    }
    else
    {
      tetra->n[0]=t->n[0];
      tetra->n[1]=t->n[1];
      tetra->n[2]=t->n[2];
      tetra->n[3]=t->n[3];
    }
    if (Dtetra->Len==0)
    {
      Dtetra->Len++;
      tetra->id=Dtetra->Len;
      tetra->back=tetra->next=NULL;
      Dtetra->Inicio=Dtetra->Fim=tetra;
    }
```

```
        else
        {
          Dtetra->Len++;
          tetra->id=Dtetra->Len;
          tetra->next=NULL; tetra->back=Dtetra->Fim;
          Dtetra->Fim->next=tetra;
          Dtetra->Fim=tetra;
        }
      }
      if (l<=Dtetraold->Len) tetraold=tetraold->next;
      else t=t->next;
    }
    return(Dtetraold->Len+Dt->Len);
}

void __fastcall TFrmDelaunay::Esvaziafac(void)
{
    int l;
    Myfac *aux;
    if ((Dfac->Inicio!=NULL) && (Dfac->Fim!=NULL))
    {
      for (l=1;l<=Dfac->Len;l++)
      {
        aux=Dfac->Inicio;
        Dfac->Inicio=aux->next;
        if (Dfac->Inicio!=NULL)
        Dfac->Inicio->back=NULL;
        free(aux);
      }
      Dfac->Len=0;
      Dfac->Inicio=Dfac->Fim=NULL;
```

```
  }
}

void __fastcall TFrmDelaunay::Esvaziat(void)
{
  int l;
  Mytetra *aux;
  if ((Dt->Inicio!=NULL) && (Dt->Fim!=NULL))
  {
    for (l=1;l<=Dt->Len;l++)
    {
      aux=Dt->Inicio;
      Dt->Inicio=aux->next;
      if (Dt->Inicio!=NULL)  Dt->Inicio->back=NULL;
      free(aux);
    }
    Dt->Len=0;
    Dt->Inicio=Dt->Fim=NULL;
  }
}

void __fastcall TFrmDelaunay::Esvaziatetraold(void)
{
  int l;
  Mytetra *aux;
  if ((Dtetraold->Inicio!=NULL) && (Dtetraold->Fim!=NULL))
  {
    for (l=1;l<=Dtetraold->Len;l++)
    {
      aux=Dtetraold->Inicio;
      Dtetraold->Inicio=aux->next;
```

```
      if (Dtetraold->Inicio!=NULL)  Dtetraold->Inicio->back=NULL;
      free(aux);
    }
    Dtetraold->Len=0;
    Dtetraold->Inicio=Dtetraold->Fim=NULL;
  }
}

void __fastcall TFrmDelaunay::ListaMath(int Lentetra)
{
  int l,k;
  FILE *ftet;
  ftet=fopen("lista.nb","w+");
  fprintf(ftet,"Notebook[{\n");
  fprintf(ftet,"Cell[BoxData[\n");
  fprintf(ftet,"\\(\\(tetra={");
  tetra=Dtetra->Inicio;
  k=0;
  for (l=1;l<=Lentetra;l++)
  {
    if (k==4)
    {
      k=0;
      fprintf(ftet,"\n");
    }
    if (l<Lentetra)
    {
      fprintf(ftet,"{%d,%d,%d,%d},",tetra->n[0],tetra->n[1],
                    tetra->n[2],tetra->n[3]);
    }
    else
```

```
    {
      fprintf(ftet,"{%d,%d,%d,%d}};\\)\\)], \"Input\"]\n",
                    tetra->n[0],tetra->n[1],tetra->n[2],tetra->n[3]);
    }
    tetra=tetra->next;
    k++;
  }
  fprintf(ftet,"},\n");
  fprintf(ftet,"FrontEndVersion->\"4.0 for Microsoft Windows\",\n");
  fprintf(ftet,"ScreenRectangle->{{0, 800}, {0, 527}},\n");
  fprintf(ftet,"WindowSize->{496, 431},\n");
  fprintf(ftet,"WindowMargins->{{33, Automatic},
                                  30, Automatic}}\n");
  fprintf(ftet,"]");
  fclose(ftet);
}

void __fastcall TFrmDelaunay::Insertfac(void)
{
  if (Dfac->Len==0)
  {
    Dfac->Len++;
    fac->id=Dfac->Len;
    fac->back=fac->next=NULL;
    Dfac->Inicio=Dfac->Fim=fac;
  }
  else
  {
    Dfac->Len++;
    fac->id=Dfac->Len;
    fac->next=NULL; fac->back=Dfac->Fim;
```

```
    Dfac->Fim->next=fac;
    Dfac->Fim=fac;
  }
}

void __fastcall TFrmDelaunay::Inserttetraold(int j, int Len)
{
  int i;
  tetraold=(Mytetra *)malloc(sizeof(Mytetra));
  Tetra(j,Len);
  for (i=0;i<4;i++) tetraold->n[i]=tetra->n[i];
  if (Dtetraold->Len==0)
  {
    Dtetraold->Len++;
    tetraold->id=Dtetraold->Len;
    tetraold->back=tetraold->next=NULL;
    Dtetraold->Inicio=Dtetraold->Fim=tetraold;
  }
  else
  {
    Dtetraold->Len++;
    tetraold->id=Dtetraold->Len;
    tetraold->next=NULL; tetraold->back=Dtetraold->Fim;
    Dtetraold->Fim->next=tetraold;
    Dtetraold->Fim=tetraold;
  }
}

void __fastcall TFrmDelaunay::Insertt(Myfac *aux, int n, int ind)
{
  int z;
```

```
  for (z=1;z<=n;z++)
  {
    Fac(z);
    if ((fac->n[0]!=-1) && (fac->n[1]!=-1) && (fac->n[2]!=-1))
    {
       t=(Mytetra *)malloc(sizeof(Mytetra));
       t->n[0]=fac->n[0];
       t->n[1]=fac->n[1];
       t->n[2]=fac->n[2];
       t->n[3]=ind;
       if (Dt->Len==0)
       {
         Dt->Len++;
         t->id=Dt->Len;
         t->back=t->next=NULL;
         Dt->Inicio=Dt->Fim=t;
       }
       else
       {
         Dt->Len++;
         t->id=Dt->Len;
         t->next=NULL; t->back=Dt->Fim;
         Dt->Fim->next=t;
         Dt->Fim=t;
       }
    }
  }
}

void __fastcall TFrmDelaunay::Abrir1Click(TObject *Sender)
{
```

```
  FILE *flcord;
  char ch;
  Repaint();
  if (aloca==1) EsvaziaListas();
  CreateListas();
  IniciaTetra();
  IniciaT(6);
  if(OpenDialog1->Execute())
    IniciaLcord(OpenDialog1->FileName.c_str());
}

void __fastcall TFrmDelaunay::Executar1Click(TObject *Sender)
{
  time_t tini,tfim;
  char ch[30],chnum[10];
  int i,j;
  tini=time(NULL);
  principal();
  tfim=time(NULL);
  tfim=tfim-tini;
}
```

APÊNDICE B

O apêndice B descreve a forma de utilização do software descrito no Apêndice A. Aqui é estabelecido de forma clara e concisa o significado de cada item de menu ou de caixa de diálogo que aparecem no programa.

B.1 O Menu Arquivo

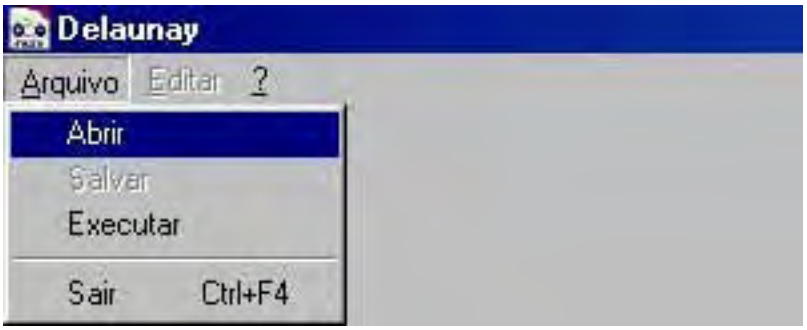

Figura B.1: Menu Arquivo

- Abrir

Para abrirmos a nuvem de pontos devemos primeiramente escolher a opção *Abrir* do menu e em seguida na caixa de diálogo apresentada escolhemos o arquivo que contém os pontos do sólido (página 35 da tese) a ser discretizado (figura B.1.2).

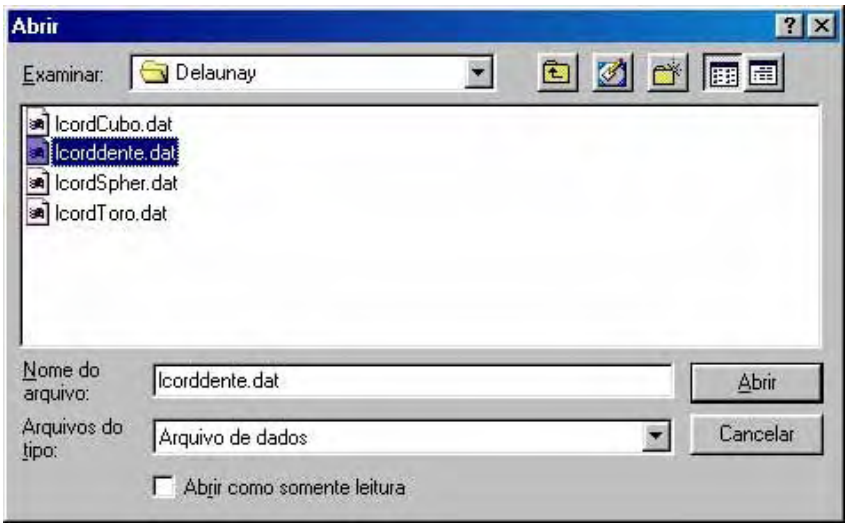

Figura B.1.2: Escolhendo o sólido para discretizar

- Salvar

A Opção salvar não foi implementada, uma vez que o *software* do Apêndice A salva automaticamente os pontos discretizadas para uma lista no formato Mathematica de nome *lista.nb*.

- Executar

Após a escolha da nuvem de pontos devemos aplicar os algoritmos de discretização, para isso escolhemos a opção *Executar* no menu *Arquivo* (figura B.1.3).

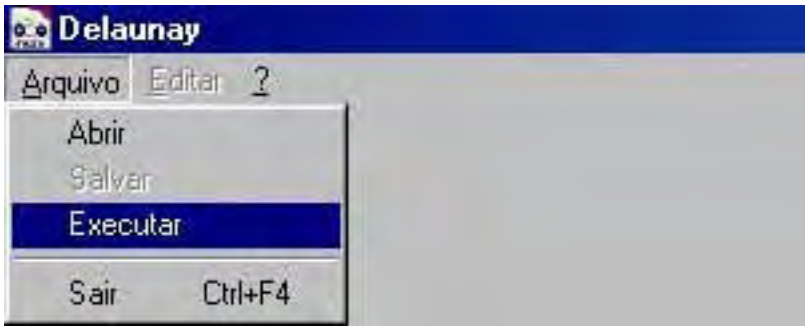

Figura B.1.3: Iniciando o Algoritmo de Delaunay

Enquanto o algoritmo é executado podemos observar sua evolução pela barra de progresso como na figura B.1.4. Ao término da execução será apresentada uma caixa de diálogo com o tempo total de processamento (figura B.1.5).

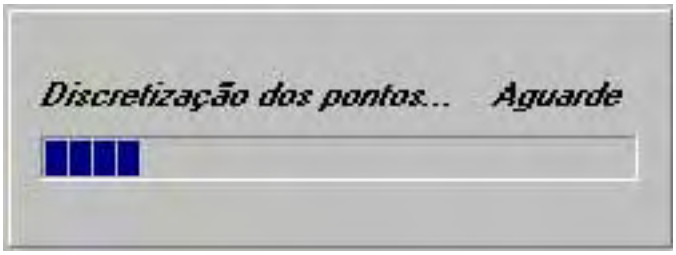

Figura B.1.4: Evolução do processamento

Figura B.5: Fim do Algoritmo

- Sair

Esta opção finaliza a utilização do *software*.

B.2 Outros Menus

Os outros menus apresentados *Editar* e *?* não foram implementados pois não eram essenciais para o funcionamento do programa.

B.3 Gerando os Gráficos com o Mathematica

A utilização da lista de pontos no Ambiente Mathematica é bem trivial basta abrir no Mathematica o arquivo *lista.nb* (figura B.3.1) gerado pelo *software* do Apêndice A e executá-lo com as teclas Contrl + Enter. Em seguida realizamos o mesmo processo para a lista lcord e para as funções gráficas.

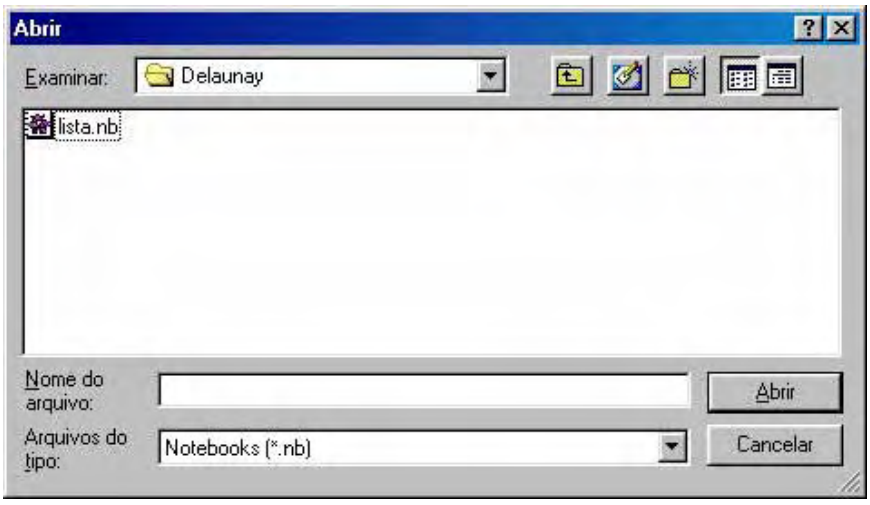

Figura B.3.1 Abrindo a lista de tetraedros

REFERÊNCIAS BIBLIOGRÁFICAS

[1] MARRETO, C. A. R. **O Algoritmo de Delaunay em Três Dimensões: Uma Implementação Computacional para o Ambiente Mathematica For Windos**. São José do Rio Preto, 1999. Dissertação (Mestrado Matemática Aplicada) – Instituto de Biociências, Letras e Ciências Exatas, UNESP.

[2] PETZOLD, C. **Programando para Windows 95**. Makron Books do Brasil Editora Ltda, 1997.

[3] MARRETO, C. A. R., MACHADO, J. M. **A compact library for automatic generation ot tetrahedra meshes using the Mathematica system**, Proc. of 8th International IGTE Symposium on Numerical Field Calculation in Electrical Engineering & European TEAM Workshop, Graz, Autria, Part I, 1998, 116-120.

[4] HERMELINE, F. **Triangulation automatique d'un polyèdre en dimension N**, R.A.I.R.O., Analyse numérique, Numerial Analysis, Vol. 16, nº 3, pp. 211-242, 1982.

[5] SAKAMOTO, M. M., et. al. **Modelador Geométrico para Aplicações Odontológicas**. II JOL – II Jornada Odontológica de Londrina, Londrina, 1997

[6] BOULOS, P., et. al. **Geometria Analítica Um tratamento Vetorial**. McGraw-Hill, São Paulo, 2 ed, 1987.

[7] MANTYLA M. **An Introduction solid modeling**, Computer Sciense Press, Inc, 1988.

[8] VIEIRA, C. **Implementação de um gerador automático de elementos finitos sobre domínios tridimensionais**. CBMAG-96 - Congresso Brasileiro de Eletromagnetismo, Ouro Preto, 1996.

[9] BARROS, G., FERNANDES, M. A. G. Estudo das propriedades mecânicas do Fêmur Humano pelo Método dos Elementos Finitos. **USP**, 2001. <http://www.mcca.ep.usp.br/~petmecan/pesquisa/realizad/femur.htm>

[10] ZAMBON. **Display Odontologico**. Op Editorial Ltda.

[11] GEORGE, P. L., HERMELINE, F. **Delaunay's mesh of a convex polyhedron in dimension d. Application to arbitrary polyhedra,** Int. J. for Num. Meth. in Engng., Vol. 33, pp. 975-995, (1992).

[12] BABUSKA Y., et. al. **On the angle condition in the finite element method**, Siam J. Num. Anal., Vol. 13, nº 2 (1976).

[13] CIARLET, P. G. **The finite element method for ellipitc problems**. North-Holland (1978).

[14] THACKER, W. C. **A brief review of techniques for generating irregular computational grids**, Int. J. Num. Met. Engng., Vol. 15, pp. 1335-1341, (1980).

[15] DELAUNAY, B. **Sur la sphère vide**, Bul. Acad. Sci. URSS Class. Sci. Nat., pp. 793-800, (1934).

[16] HERMELINE, F. **Une méthode automatique de maillage en dimension n**, Thèse Paris (1980).

[17] SCHROEDER W. J., et. al. **Geometry-based fully automatic mesh generation and the Delaunay triangulation**, Int. J. Numer. Methods Eng., Vol. 26, pp. 2503-2515, (1988).

[18] MACHADO, J. M. **LMAG-3D: Um software CAD/CAE para eletromagnetismo baseado no método dos elementos finitos em 3D**, Tese apresentada a Escola Politécnica da USP para obtenção do título de Doutor em Engenharia, São Paulo, (1993).

[19] WOLFRAM, S. Mathematica**: A system for doing mathematics by computer**. Addison-Wesley Publishing Company, Inc., second edition, 1991.

[20] JONES, T. W. **Mathematica graphics techniques & applications,** TELOS, Inc., 1994.

[21] BAHDER, T. B. **Mathematica for Scientists and Engineers**, Wesley Publishing Company, Inc, 1995.

[22] HOLZNER, S. **Borland C++ Programação For Windows**. Makron Books do Brasil Editora Ltda, 3 ed, 1995.

[23] RUGGIERO, M. A., G. et. al. **Cálculo Numérico Aspectos teóricos e computacionais**. Makron Books do Brasil Editora Ltda, 2 ed, 1997.

[24] WATKINS, C. D., et. al. **Programando em 3 Dimensões**. Berkeley Brasil Editora, 1993

[25] COY, S. B., et. al. **Photorealism and Ray Tracing in C**. M&T Books.

[26] BOWYER A. **Computing Dirichlet tessellations, The computer journal**, Vol. 24, nº 2, pp. 162-166, (1981).

[27] LEDLEY, R. S., et. al. **Linear Model of Tooth Displacement by Applied Forces**. J Dent Res May-June 1968 pp 427-432.