UNIVERSIDADE ESTADUAL PAULISTA "JÚLIO DE MESQUITA FILHO"
FACULDADE DE ENGENHARIA
CAMPUS DE ILHA SOLTEIRA

LEANDRO DE SOUZA SCHIARA

DESENVOLVIMENTO DE UM PROGRAMA COMPUTACIONAL PARA ANÁLISE DE TRINCAS VIA MÉTODO DOS ELEMENTOS DE CONTORNO:

Modelagem de Sólidos e Geração de Malha

Ilha Solteira
2023

UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Ilha Solteira

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA MECÂNICA

LEANDRO DE SOUZA SCHIARA

DESENVOLVIMENTO DE UM PROGRAMA COMPUTACIONAL PARA ANÁLISE DE TRINCAS VIA MÉTODO DOS ELEMENTOS DE CONTORNO:

Modelagem de Sólidos e Geração de Malha

Dissertação apresentada à Faculdade de Engenharia de Ilha Solteira – Unesp, como parte dos requisitos para obtenção do título de Mestre em Engenharia Mecânica.

Nome do Orientador
Amarildo Tabone Paschoalini

Ilha Solteira
2023

FICHA CATALOGRÁFICA
Desenvolvido pelo Serviço Técnico de Biblioteca e Documentação

S329d

Schiara, Leandro de Souza.
Desenvolvimento de um programa computacional para análise de trincas via método dos elementos de contorno: modelagem de sólidos e geração de malha / Leandro de Souza Schiara. -- Ilha Solteira: [s.n.], 2023
202 f. : il.

Dissertação (mestrado) - Universidade Estadual Paulista. Faculdade de Engenharia de Ilha Solteira. Área de conhecimento: Mecânica dos Sólidos, 2023

Orientador: Amarildo Tabone Paschoalini
Inclui bibliografia

1. Modelagem computacional de sólidos. 2. Geração de malha. 3. Método dos elementos de contorno. 4. Mecânica da fratura.

Raiane da Silva Santos
Supervisora Técnica de Seção
Seção Técnica de Referência, Atendimento ao usuário e Documentação
Diretoria Técnica de Biblioteca e Documentação
CRB/8 - 9999

UNIVERSIDADE ESTADUAL PAULISTA

Câmpus de Ilha Solteira

CERTIFICADO DE APROVAÇÃO

TÍTULO DA DISSERTAÇÃO: Desenvolvimento de um Programa Computacional para Análise de Trincas via Método dos elementos de Contorno: Modelagem de Sólidos e Geração de Malha

AUTOR: LEANDRO DE SOUZA SCHIARA
ORIENTADOR: AMARILDO TABONE PASCHOALINI

Aprovado como parte das exigências para obtenção do Título de Mestre em Engenharia Mecânica, área: Mecânica dos Sólidos pela Comissão Examinadora:

Assinado de forma digital por Amarildo Tabone Paschoalini:09280956817
Dados: 2023.03.03 11:41:58 -03'00'

Prof. Dr. AMARILDO TABONE PASCHOALINI (Participaçao Virtual)
Departamento de Engenharia Mecânica / Faculdade de Engenharia de Ilha Solteira - UNESP

Fabio Roberto Chavarette:15473115845
Assinado de forma digital por Fabio Roberto Chavarette:15473115845
Dados: 2023.03.03 12:05:53 -03'00'

Prof. Dr. FABIO ROBERTO CHAVARETTE (Participaçao Virtual)
Departamento de Engenharia, Física e Matemática / Instituto de Química de Araraquara - UNESP

Rodrigo Guerra Peixoto:01333027680
Digitally signed by Rodrigo Guerra Peixoto:01333027680
Date: 2023.03.03 13:08:02 -03'00'

Prof. Dr. RODRIGO GUERRA PEIXOTO (Participaçao Virtual)
Departamento de Engenharia de Estruturas / Universidade Federal de Minas Gerais UFMG

Ilha Solteira, 03 de março de 2023

DEDICATÓRIA

Aos meus pais:

Odete de Souza,

Alfredo Eustáquio Schiara

AGRADECIMENTOS

Em primeiro lugar agradeço aos meus pais Alfredo Eustáquio Schiara e Odete de Souza que, por seu amor, esforço e dedicação, são a *solução fundamental* de toda a cadeia de eventos que levaram à criação desse trabalho. A importância de vocês em minha vida é *singular*. Que nossos nomes fiquem registrados aqui até o dia em que a *entropia* da informação, inevitavelmente, dê um fim a esse texto.

Logo na primeira disciplina que cursei na UNESP tive o privilégio de conhecer o prof. Dr. Amarildo Tabone, hoje meu orientador. Obrigado por acolher, incentivar e direcionar esse trabalho, além da troca de experiências e sugestões.

Um forte abraço aos ex-colegas de trabalho do setor de inspeção de equipamentos da Refinaria Gabriel Passos (REGAP) pelos anos de saudável convivência, período no qual se encontra as origens desse trabalho. Sobretudo Paulo Pio, Rafael Reis e Ricardo Schayer sempre tinham alguma ideia interessante que levava à reflexão e à construção de um conhecimento mais coeso e profundo. Giuseppe Cristóvam (*in memoriam*) ajudava nas minhas inspeções quando eu não podia ficar depois do horário por causa da faculdade.

Meus sinceros agradecimentos à toda equipe de manutenção da Usina Termelétrica de Três Lagoas pelos anos de convivência e aprendizado mútuo. Destaco o gerente Flávio Altino, sempre muito lhano no trato, que entendeu e incentivou esse trabalho. Também lembro de Renato Barella que sempre dizia “Você tem que fazer um mestrado!”. Bárbara Ferro, de rápida passagem por lá, foi quem “desbravou” o programa de pós-graduação (PPGEM) da UNESP. Sabendo o que fazer, saí da inércia, e logo em seguida, comecei a cursar o mestrado.

RESUMO

A integridade das estruturas pode ser seriamente afetada pela presença de trincas, portanto, são necessários meios de se detectar, dimensionar e avaliar a severidade desses defeitos. Para essa avaliação, a mecânica da fratura tem se mostrado uma disciplina de crescente importância no projeto e na manutenção de estruturas e equipamentos em várias indústrias, especialmente nas de aviação e de óleo e gás. O uso dessa disciplina, no entanto, normalmente exige o emprego de métodos numéricos. O método dos elementos de contorno é uma técnica bastante adequada para a modelagem de defeitos planares em face se sua propriedade de redução de dimensionalidade. Dessa forma, a etapa de pré-processamento é bastante simplificada se comparada à técnica de elementos finitos, por exemplo. Não há muitas ferramentas de fácil utilização e que abranjam todas as etapas de uma simulação numérica via elementos de contorno, especialmente quando se trata de trincas, por terem topologia normalmente não aceita pelos programas usuais de modelagem de sólidos. Por causa disso, esse trabalho dedicou-se ao desenvolvimento de um *software* capaz de efetuar o modelamento dos sólidos adaptado à mecânica da fratura, bem como gerar a malha na superfície para análise via método dos elementos de contorno. O programa em desenvolvimento tem e terá mínima dependência de *softwares* externos. A união da modelagem computacional de sólidos, da mecânica da fratura e do método dos elementos de contorno permitirá o desenvolvimento de uma ferramenta computacional simples denominada CABEMT, mas de considerável flexibilidade e, portanto, capaz de ser utilizada na prática de engenharia.

Palavras-chave: modelagem computacional de sólidos; geração de malha; método dos elementos de contorno; mecânica da fratura.

ABSTRACT

Structural integrity can be seriously compromised by the presence of crack-like flaws, therefore, there should be ways to detect, measure and assess the severity of such damages. The latter is frequently addressed by fracture mechanics for whose importance has been growing in the design and maintenance of structures and equipment in many industries, especially in the aviation and oil & gas. However, to employ this technique, it's frequently necessary to use numerical methods The Boundary Element Method is a desirable choice to planar defect modeling due to its reduced dimensionality characteristics. In this way, the pre-processing phase is very simplified if compared to the finite element method, for example. There are not many easy handling tools that encompasses all steps to a numerical simulation by BEM, especially for cracks which topology is generally not accepted by conventional solid modelers. Because of that, this work developed a software capable of modeling cracked solid bodies and to perform mesh generation on its surfaces in order to analyze it by the BEM in a future expansion. The computer program under development has and will have minimum external dependency. The combination of solid modeling techniques, fracture mechanics and boundary element method allowed the development of a simple computational tool, but that poses considerably flexibility, therefore able to be utilized in the engineering practice.

Keywords: computational solid modeling; mesh generation; boundary element method; fracture mechanics.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

BEM	*Boundary Element Method*
BES	*Boundary Element System*
B-REP	*Boundary Representation*
CABEMT	*Crack Analysis and Boundary Element Tool*
CAD	*Computer Aided Design*
CAE	*Computer Aided Engineering*
CAM	*Computer Aided Manufacturing*
CGS	Construção Geométrica de Sólidos
CNC	Comando Numérico Computadorizado
CPU	Unidade Central de Processamento
EF	Elementos Finitos
FIT	Fator de Intensidade de Tensões
IDE	Interface de Desenvolvimento
IGU	Interface Gráfica do Usuário
MCS	Modelagem Computacional de Sólidos
MDF	Método das Diferenças Finitas
MEC	Método dos Elementos de Contorno
MEF	Método dos Elementos Finitos
MF	Mecânica da Fratura
NAN	*Not A Number*
NURBS	*Non-Uniform Rational B-Splines*
OB	Operação Booleana
OSM	*Object Solid Modeler*
POO	Programação Orientada a Objetos
RGB	Red Green Blue (especificação de cor)
SSD	*Solid State Drive*

LISTA DE SÍMBOLOS

LETRAS LATINAS

A	Área de secção
$[\boldsymbol{A}]$	Matriz de coeficientes
a	Número de arestas de um objeto
a_e	Dimensão do maior semieixo de uma elipse ou identificação de uma aresta
(a, b, c)	Vetor diretor de uma reta
a_{ik}	Semiaresta que vai de i a k
$\boldsymbol{b}$	Vetor binormal
b_e	Dimensão do menor semieixo de uma elipse
$B_{i,n}$	Função de base de uma curva de *Bézier*
D	Distância entre os pontos iniciais de duas linhas
$\bar{d}$	Variável de parametrização centrípeta de uma spline ou distância entre duas retas
d	Parâmetro da equação do plano
d'	Distância entre um ponto e uma reta
F	Número de faces de um objeto
F_k	k-ésima face
${}^{i}F_j$	Face j de um sólido i
$f; f'$	Uma face qualquer
G	Número de genus de um sólido
h_m	Maior distância entre os vértices de um elemento
I_k	Iterador
L	Dimensão de um objeto
l	Média dos comprimentos das arestas de um elemento
$[\boldsymbol{M}]$	Matriz de transformação de coordenadas
m	Número total de knots
$N_{i,p}$	Função de base de uma *B-spline*
NE	Número de elementos que compartilham um determinado nó
N_e	Número de elementos da malha
N_N	Número de nós de um elemento
N_s	Número de segmentos do sólido após a varredura rotacional
n	Número de nós
$\boldsymbol{n}; \vec{n}$	Vetor normal a uma superfície ou elemento de contorno
$\vec{n}_F$	Vetor normal de uma face
$\vec{n}_e$	Vetor normal de extrusão
$\vec{n}_r$	Vetor/eixo de revolução da varredura rotacional
$\boldsymbol{P}$	Vetor cujas coordenadas representam um ponto no espaço
$\boldsymbol{P'}$	$\boldsymbol{P}$ representado em coordenadas locais
$\boldsymbol{P_i}$	Ponto de controle de uma spline (vetor)
p	Pressão ou grau de interpolação de uma B-spline
p_t	Semi-perímetro de um triângulo
$\boldsymbol{Q_k}$	Ponto a ser interpolado em uma spline

r	Raio de um círculo ou arco
r_a	Razão de aspecto
$\bar{S}$	Área da face de um tetraedro
$\boldsymbol{S_k}$	Sólido
$\boldsymbol{S_u}; \boldsymbol{S_{uk}}$	Superfície
t	Parâmetro
$\bar{t}$	Parâmetro
V	Número de vértices
$\bar{V}$	Volume de uma célula
$V_k; V_k{}'$	Vértices de número k
x	Coordenada cartesiana
(x_0, y_0, z_0)	Ponto inicial de uma reta
(x_c, y_c, z_c)	Centro de um círculo
(x_r, y_r, z_r)	Ponto pertencente a uma reta

LETRAS GREGAS

θ	Parâmetro angular
θ_0	Rotação da elipse considerando o eixo X como base
ρ_m	Raio do círculo inscrito em um elemento triangular
σ_m	Parâmetro de qualidade de um elemento

SUMÁRIO

1 INTRODUÇÃO

Desde os primórdios da civilização as estruturas tem desempenhado um papel crucial no seu desenvolvimento. Primeiramente para proteção e abrigo, e posteriormente englobando as máquinas e equipamentos indispensáveis ao modo de vida moderno. Por se fazer tão presente em nosso dia-a-dia, a integridade dessas estruturas quase sempre guarda estreita relação com a nossa própria integridade física e com a segurança jurídica das empresas. O projeto das estruturas antigas era provavelmente obtido por testes de tentativa e erro (ANDERSON, 2005). De fato, as técnicas de construção antigas chegaram a um nível de complexidade elevado muito antes das teorias subjacentes (BENVENUTO, 1991). Estas vieram evoluindo aos poucos e, aliada às melhores técnicas, proporcionaram o projeto de estruturas otimizadas, muito embora sempre estejam expostas a mecanismos de dano que podem levar à sua degradação e colapso.

A fratura é um dos mais comuns e importantes danos aos quais as estruturas estão sujeitas, visto ser muito difícil tanto produzir elementos sem defeitos quanto monitorá-los ao longo da vida útil dos equipamentos e estruturas. Vários acidentes ao longo da história podem ser associados à presença de defeitos nos materiais e/ou projeto inadequado, ver por exemplo Anderson (2005), Broek (1984) e Jones (2001). Destacam-se as falhas de fadiga, que nada mais são que a propagação de trincas nas estruturas devido aos esforços cíclicos. Em 1978, estimou-se que os custos decorrentes de danos por fratura correspondiam a cerca de 4% do PIB dos EUA (ANDERSON, 2005). Além da perda financeira, há o risco de fatalidades, lesão pessoal e contaminação do meio ambiente (VENART, 2004). Danos por fratura tem um longo histórico associado aos setores de aviação, refino, petroquímica, geração elétrica e de construção civil (ANDERSON, 2005).

Pelas razões apresentadas, as trincas vêm sendo estudadas pelo menos desde a década de 20, dando origem à disciplina da Mecânica da Fratura (MF) que aproveitou bastante da análise matemática proveniente da teoria da elasticidade. Hoje existem normas para avaliação de estruturas, tais como o API-579 (ASME & API, 2016) e a BS 7910 (BRITISH STANDARDS, 2013) nas quais os parâmetros da mecânica da fratura são cruciais para avaliar uma ampla gama de mecanismos de danos, com destaque para a fadiga, fratura frágil, fluência e corrosão sob tensão. Há

ainda nessas normas, tabelas com soluções analíticas ou empíricas para avaliação de MF, além de existirem outras fontes para obtenção desses parâmetros, tais como em Laham *et al*. (1998).

Ainda assim, as soluções analíticas, por serem de difícil dedução, em geral limitam-se a problemas muito simples geometricamente, ao passo que nosso cotidiano é rodeado por formas geométricas e condições de contorno um tanto quanto complexas. As soluções com parâmetros tabelados costumam ser trabalhosas de se usar[1], exigindo muitas operações de consulta e cálculos intermediários; favorecendo, portanto, a ocorrência de equívocos. Não obstante, a essas normas tipicamente associaram-se *softwares* específicos para facilitar os cálculos. Pela BS-7910 tem-se o Crackwise (TWI GLOBAL, 2002) e pelo API-579 o IntegriWise, Becht FFS, entre outros (BECHT ENGINEERING, 2020) – cujo custo pode ser elevado[2] para o usuário, especialmente se levarmos em conta que as análises de vida remanescente em estruturas decerto não são atividades rotineiras em todas as empresas ou setores de manutenção/inspeção das empresas, portanto elevando o custo por análise.

Muitos desses cálculos podem ser feitos diretamente pelo uso de algum método numérico apropriado e foi somente com o advento desses, aliado à computação eletrônica, que a análise detalhada de estruturas foi possível. Dentre os métodos conhecidos, destacam-se o Método das Diferenças Finitas (MDF), o Método dos Elementos Finitos (MEF) e o Método dos Elementos de Contorno (MEC).

O MDF teve origem por volta dos anos 1930, evoluindo sob a colaboração de muitos pesquisadores. No decorrer dessa evolução, surgiu o MEF, por volta dos anos 1940 e 1950. Muitas ideias e técnicas são comuns a ambos os métodos, no entanto, devido a sua maior flexibilidade geométrica e à possibilidade de aplicá-lo a diferentes campos da engenharia de uma maneira sistemática, o MEF tornou-se dominante na indústria.

Em paralelo ao desenvolvimento do MDF e do MEF, trabalhava-se com as

---

[1] O API-579 especialmente possui uma miríade de tabelas e fórmulas, tornando difícil aplicá-lo diretamente aos problemas da indústria, em situações nas quais muitas vezes é necessária rapidez na avaliação da integridade da estrutura. Por causa disso, existem *softwares* com essas fórmulas embutidas.

[2] Os fabricantes desses softwares não costumam divulgar os seus preços, mas na internet foram encontrados valores de $ 8.500,00/ano para assinatura do Becht FFS (BECHT ENGINEERING, 2020) e de £4.995,00 para a licença do Crackwise, porém relativo ao ano de 2002.

equações integrais de contorno, baseando-se nos trabalhos de George Green, Erik Fredholm, Muskhelishvili e Kupradze – culminando-se no desenvolvimento do MEC nos anos 1970 (BREBBIA, 2017). Este método permite a redução da dimensionalidade do modelo, isto é, fazendo com que problemas tridimensionais possam ser resolvidos com uma discretização bidimensional. Por causa disso, o MEC é uma escolha natural ao se modelar trincas, dado que sua geometria é essencialmente planar, além de permitir um refinamento local da malha mais eficiente. Outras vantagens são a satisfação de condições de contorno no infinito, a melhor integração com sistemas CAD (*Computer Aided Design*) e soluções mais precisas, mesmo com uma malha mais grosseira que o MEF (ALIABADI, 2002).

Para aproveitar plenamente o potencial desses métodos numéricos é preciso uma etapa fundamental – denominada pré-processamento – na qual o modelo conceitual (geometria, topologia e dimensões) é transformado e representado em linguagem de máquina[3]. A Modelagem Computacional de Sólidos (MCS) nos dá a capacidade de descrever, modificar, manipular, armazenar e exibir entidades geométricas no computador para que então possam ser discretizadas por meio da geração de malha e finalmente encaminhada para algum método numérico para análise (CAE – *Computer Aided Engineering)* ou para a fabricação (CAM – *Computer Aided Manufacturing*). Apesar de sua importância, a quantidade de literatura técnica em MCS é consideravelmente menor que dos métodos numéricos, especialmente no Brasil.

Essas tecnologias de MCS e dos métodos numéricos foram transformadas em exitosos *softwares* comerciais. Destacam-se os *softwares* de elementos finitos: ABACUS®, ANSYS®, NASTRAM® para citar alguns. O MEF ainda conta com alternativas de software livre como o CODE_ASTER (AUBRY, 2013), desenvolvido pela empresa francesa EDF. Para o MEC há o *software* comercial BEASY® e um software gratuito (porém de código fechado) BES, desenvolvido pela Universidade de Cornell. Os programas de elementos finitos normalmente são acompanhados de ferramentas para modelagem de trincas em duas dimensões. Para três dimensões, as ferramentas inerentes a esses programas tendem a ser mais limitadas. O ANSYS®,

[3] No caso a função do programa de modelagem é exatamente essa. O programador, entretanto, raramente utilizará a linguagem de máquina diretamente. Ele trabalhará com uma linguagem de alto nível que traduzirá para a linguagem de máquina.

por exemplo, começou a possuir ferramentas mais robustas para modelagem 3D de trincas somente na sua versão 14.5[4], lançada em 2012. Essas limitações foram exploradas por softwares de apoio, tais como o Zencrack® (CHANDWANI; TIMBRELL; WIEHAHN, 2008), FEACrack e Trinca 3D (SCHIARA e RIBEIRO, 2016). Tipicamente, esses programas geram uma malha separadamente[5], em bloco, para inserção no modelo de elementos finitos. Técnicas similares permitem a integração entre o MEC e o MEF (CITARELLA *et al*., 2014).

O Franc3D[6] (CORNELL FRACTURE GROUP, 2003), que de certa forma inspirou o desenvolvimento desse trabalho, originou-se a partir do grupo de mecânica da fratura da Universidade de Cornell. Ele precisa de dois outros programas: o OSM (*Object Solid Modeler*) e o BES (*Boundary Element System*) que fazem a MCS e a solução das equações de contorno respectivamente. O Franc3D é responsável pelo pré e pós-processamento. Apesar de ser um bom software, a versão gratuita do Franc3D depende muito do OSM que por sua vez é bastante limitado, de tal forma que é difícil fazer modelos mais complexos. Ademais, o BES[7] é um solucionador lento, já que aparentemente não usa paralelismo no processamento.

A despeito da evolução dos modelos computacionais, falhas associadas à presença e propagação de trincas nas estruturas e equipamentos continuam acontecendo – vide figura 1.1. Compilações interessantes desse tipo de falha podem ser vistas em Pellicione *et al*. (2012) e Jones (2001), para citar alguns. Para combater a contínua ameaça de falhas catastróficas ocasionadas por trincas e, ao mesmo tempo reduzir as perdas financeiras associadas, faz-se necessário o permanente emprego das melhores técnicas de projeto, fabricação, inspeção e monitoramento. Mas para isso, softwares de custo acessível, amigáveis ao usuário e que não dependam de outros programas de elevado custo (ou de elevada complexidade) para funcionar, são de crucial importância.

Para se maximizar a difusão do uso da MF na engenharia, são necessárias

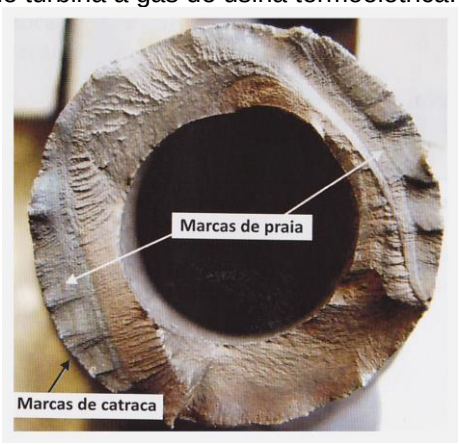
---

[4] Anteriormente a isso, ele possuía o comando CTMOPT para geração de um bloco de trinca elíptica, de manipulação não muito amigável ao usuário.
[5] Não se trata de submodelamento. O bloco com a trinca é inserido na malha completa da estrutura, ver (SCHIARA e RIBEIRO, 2016).
[6] Refere-se aqui ao Franc3D clássico, cuja última atualização ocorreu em 2007. A partir daí, foram lançadas versões comerciais. Esse programa, juntamente com o OSM e o BES ainda pode ser baixado no site do grupo de fratura da Universidade de Cornell por meio do endereço http://cfg.cornell.edu/software/
[7] Esse programa foi testado pelo autor.

ferramentas computacionais de baixo custo ou de código aberto e de simples manuseio. Essas duas características são especialmente importantes na engenharia de manutenção, uma vez que é pouco provável que ocorra necessidade frequente do uso de modelos computacionais para análise de tensões ou de MF no dia-a-dia. Então por um lado, a aquisição de *softwares* dispendiosos implicará automaticamente em um custo muito alto por análise realizada. Por outro lado, o uso de softwares livres ou gratuitos (quando adequados) costuma ser mais complexo[8] e exige um longo tempo para elaboração das análises porque o engenheiro não está habituado ao uso do programa por se tratar de uma utilização eventual.

Por conta disso, tem sido desenvolvido pelo autor deste trabalho desde 2017 o CABEMT (*Crack Analysis and Boundary Element Method Tool)* – um programa computacional de modelagem de sólidos e simulação pelo MEC escrito em Java® e OpenGL. Tem por objetivo ser um *software* leve, independente, de fácil uso e robusto, que permitirá a análise de tensões em problemas elásticos e termoelásticos transientes, assim como a introdução de trincas de formato arbitrário no modelo. O CABEMT não dependerá de outros *softwares* de engenharia para funcionar, se mostrando assim como uma ferramenta independente de análise.

Figura 1.1 – Exemplos de falhas devido à propagação de trincas. À esquerda: falha de um dreno de bomba de hidrocarbonetos em alta temperatura. À direita, falha por fadiga de uma pá de compressor de turbina a gás de usina termoelétrica.



Fonte: Pellicione *et al* (2012)

Fonte: Dados do próprio autor

Por serem temas de longa extensão, essa dissertação abordará a implementação da modelagem computacional de sólidos e a geração de malha, ao passo que o MEC será escopo de trabalhos vindouros. É importante ressaltar que

[8] Não é incomum que tais programas dependam de outros (de pré e pós-processamento) e de todo um interfaceamento entre eles para fazer uma análise e podem ter suporte técnico limitado.

apesar de existirem programas tipo CAD e geradores de malha de código aberto ou pelo menos gratuitos, estes não se encontram aptos a lidar diretamente com a geometria das trincas. Tais elementos possuem uma topologia normalmente não aceita pelos programas de modelagem. Ademais, a ideia central desse projeto é oferecer uma solução totalmente integrada, de tal forma que o usuário não precise ficar migrando para vários programas de forma a efetuar uma simulação. Dessa forma, a elaboração de um modelador de sólidos especificamente para o CABEMT se mostra necessária.

Busca-se também contribuir, ainda que de maneira limitada, a enriquecer a bibliografia brasileira sobre modelagem computacional de sólidos.

2 MODELAGEM COMPUTACIONAL DE SÓLIDOS

2.1 INTRODUÇÃO

A representação de objetos ou formas geométricas por computador é de profundo interesse prático para muitos campos do conhecimento: engenharia, arquitetura, medicina (LEORDEAN; VILAU; DUDESCU, 2021), artes em geral, geologia (KAVOURAS; SMART, 1989), arqueologia (BRUTTO; MELI, 2012) e outros. Na engenharia, por exemplo, é difícil imaginar um projeto que não esteja representado digitalmente de alguma forma e mais recentemente, em três dimensões. A introdução do CAD no projeto eleva a produtividade, facilita as revisões de documentos, a detecção de falhas de projeto e de interferências, permite a integração com máquinas CNC (Comando Numérico Computadorizado) para fabricação e auxilia no resgate de informações para apoio à manutenção. A modelagem computacional de sólidos ainda desempenha um papel importante na robótica, reconhecimento de padrões e visão de máquina (MORASSO, 1986), uma vez que nesses sistemas em geral é preciso virtualizar objetos do mundo real, porém não necessariamente da mesma forma que em sistemas CAD.

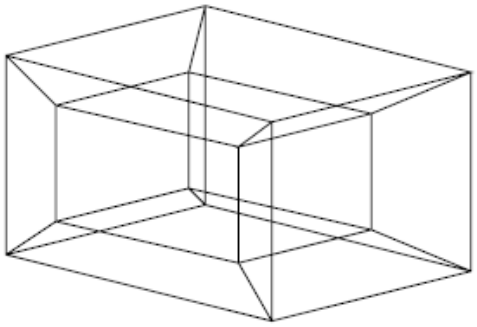
Há distintas formas de idealização das estruturas e seus componentes em modelos digitais:

- Modelos geométricos: são a representação em desenho de algum objeto ao invés do objeto em si (MÄNTYLÄ, 1988). Um exemplo de modelo gráfico são os desenhos em três vistas de alguma peça ou ainda sua representação em perspectiva. As primeiras tentativas de representação de sólidos computacionalmente foram por meio de modelos gráficos correspondentes às arestas do sólido – modelos *wireframe*. Entre muitos inconvenientes, a representação do sólido é por vezes ambígua (HOFFMANN, 1992), conforme nota-se na figura 2.1. Nesta ilustração, não se pode determinar se a peça tem realmente um furo e se tiver, em qual direção ele atravessa o objeto;
- Modelos de superfície: contêm informações detalhadas de superfícies, mas nem sempre possuem informações suficientes para se determinar todas as propriedades geométricas do objeto formado pela união das superfícies representadas (MÄNTYLÄ, 1988), isto é, há carência de informações acerca de

como as superfícies estão conectadas (topologia). No entanto, não seria possível fazer a distinção visual com clareza superior de tal forma que ocorreria ambiguidade na interpretação da figura 2.1. De fato, a representação visual seria equivalente à figura 2.2;

- Modelos sólidos: consiste na representação computacional inequívoca de um objeto físico (REQUICHA; ROSSIGNAC, 1992) de tal forma que possam ser respondidas questões arbitrárias sobre sua geometria de maneira automática, isto é, por algoritmos (MÄNTYLÄ, 1988).

Figura 2.1 – Representação ambígua de um sólido por *wireframe*



Fonte: Hoffman (1992)

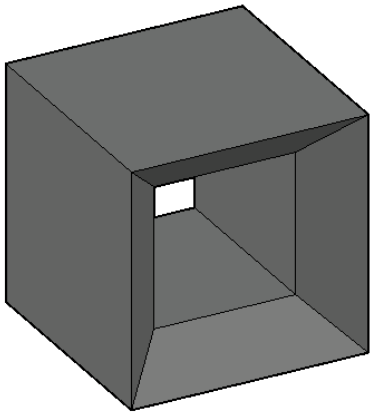
Os modelos sólidos são os mais completos, já que contêm as informações geométricas (vértices, linhas, curvas e superfícies) e as topológicas (a maneira pela qual as entidades geométricas se conectam) do sólido. Na figura 2.2 é possível observar que o sólido mostrado anteriormente agora possui uma representação inequívoca. Isso não se dá somente pela coloração e sombreamento adequados das superfícies[9] – características adequadas à interpretação humana – mas também pela conexão das arestas que formam esse objeto. Há dois conjuntos de quatro arestas que formam as extremidades do furo. Esse tipo de informação não existe nos modelos gráficos.

De maneira geral, quando o interesse é somente de visualização dos objetos, o modelo sólido é dispensável, podendo ser substituído por outros mais simples, que não possuem a informação topológica dos seus elementos. Contudo, na geração de malha para os métodos numéricos a topologia é fundamental, uma vez as informações

[9] Essa coloração também só é possível pela representação adequada do sólido e as normais da superfície. O algoritmo de traçagem usa as normais para adequar o sombreamento da superfície, tornando a visualização mais realista. Uma técnica comum é o sombreamento de Gouraud (GOURAUD, 1971). É o sombreamento que dá ideia de profundidade na representação gráfica.

de conexão entre os vários elementos discretos da malha são necessárias para a montagem das matrizes do sistema de equações desses métodos.

Figura 2.2 – Um modelo sólido de um objeto.



Fonte: Elaboração do próprio autor

2.1.1 Primitivas básicas de modelagem

As primitivas básicas de modelagem são as entidades fundamentais que constituirão os modelos sólidos e correspondem aos pontos, linhas e curvas. Os pontos são a base das outras entidades, embora sozinhos não sejam de grande utilidade. As linhas são um dos elementos fundamentais de construção das faces dos sólidos. Com frequência são representadas pelos pontos correspondentes às suas extremidades que podem estar conectadas geométrica e topologicamente a outras linhas e curvas para formar uma face ou elemento. As linhas/retas são fundamentais nos algoritmos de intersecção, conforme será demonstrado mais adiante. Na representação das entidades geométricas, pode-se utilizar mais de uma forma para descrever a primitiva. A representação paramétrica (SANTOS, 2007) da reta mostra-se bastante conveniente:

$$\begin{cases} x_r = x_0 + \bar{t}a \\ y_r = y_0 + \bar{t}b \\ z_r = z_0 + \bar{t}c \end{cases} \tag{2.1}$$

Nesse caso, é desejável tomar o ponto (x_0, y_0, z_0) como o início da linha e o parâmetro $\bar{t}$ variando de 0 (início) a 1 (final).

As curvas mais comuns na modelagem de sólidos são as cônicas – correspondentes ao círculo/arco e a elipse. A representação paramétrica do círculo ou do arco é:

$$\begin{cases} x_c = x_0 + r\cos\theta \\ y_c = y_0 + r\,\mathrm{sen}\,\theta \\ z_c = z_0 \end{cases} \tag{2.2}$$

Nesse caso, o ponto (x_0, y_0, z_0) corresponde ao centro do círculo/arco e o parâmetro θ varia de 0 a 2π; ou menos, no caso do arco. Notar que essa equação representa o círculo no plano x, y apenas, entretanto, utilizando-se um eixo de coordenadas local, é possível representar o círculo em qualquer posição e ainda fazer com que a origem do eixo local sempre coincida com o centro do círculo, de forma que x_0, y_0 e z_0 se anulam. Nessa condição, a equação implícita do círculo é:

$$x_c^2 + y_c^2 - r^2 = 0 \tag{2.3}$$

Trata-se de outra forma de representar matematicamente o círculo.

A elipse é uma forma importante na engenharia, especialmente na mecânica da fratura, tendo em vista que as trincas normalmente são modeladas em formato elíptico – sua equação na forma implícita é (HOFFMANN, 1992):

$$\frac{x^2}{a_e^2} + \frac{y^2}{b_e^2} - 1 = 0 \tag{2.4}$$

Ao passo que na forma paramétrica é (HOFFMANN, 1992):

$$x(t) = a_e \frac{1 - \bar{t}^2}{1 + \bar{t}^2}\,; \qquad y(t) = b_e \frac{2\bar{t}}{1 + \bar{t}^2} \tag{2.5}$$

Há ainda a possibilidade de parametrizar pelo ângulo:

$$x(\theta) = x_e + r\cos(\theta + \theta_0)\,;\; y(\theta) = y_e + r\,\mathrm{sen}(\theta + \theta_0)\,;$$

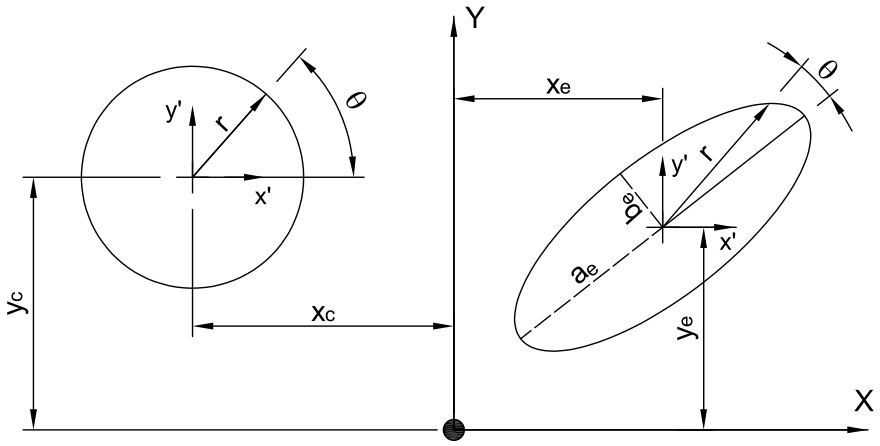
$$r = \sqrt{\frac{a_e^2 b_e^2}{\cos^2\theta\, b_e^2 + sen^2\theta\, a_e^2}} \tag{2.6}$$

A figura 2.3 ilustra essas primitivas e a notação de seus parâmetros.

As formas paramétricas são adequadas para a segmentação de curvas, por sua vez úteis nos algoritmos de exibição e intersecção. A exibição de curvas é normalmente aproximada por segmentos de reta, tal como sugerido no manual do OpenGL (JACKIE; DAVIS; DIXON, 1997) – um conhecido software de computação gráfica. A segmentação pode ser feita por meio de incrementos no parâmetro, porém isso seria muito difícil na representação implícita. Contudo, verificar se um ponto coincide com uma curva ou uma superfície é mais fácil na representação implícita. Há

métodos para converter de uma forma de representação para outra. A princípio, todas as curvas e superfícies paramétricas racionais podem ser convertidas para a forma implícita, mas o contrário não é necessariamente verdadeiro (HOFFMANN, 1992).

Figura 2.3 – O círculo e a elipse: duas primitivas de modelagem geométrica.



Fonte: Elaboração do próprio autor

Quando se torna necessário representar curvas de formatos arbitrários, é usual recorrer às curvas de *Bézier* e às *splines*. Elas são definidas pela seguinte expressão (PIEGL e TILLER, 1997):

$$\boldsymbol{C}(\bar{t}) = \sum_{i=0}^{n} B_{i,n}(\bar{t})\boldsymbol{P}_i \qquad 0 \leq \bar{t} \leq 1 \tag{2.7}$$

$$B_{i,n}(\bar{t}) = \frac{n!}{i!\,(n-i)!}\bar{t}^{i}(1-\bar{t})^{n-i} \tag{2.8}$$

na qual $\boldsymbol{C}(\bar{t})$ e $\boldsymbol{P}_i$ são os vetores de posição dos pontos da curva e dos pontos de controle, respectivamente. $B_{i,n}$ são os polinômios de Bernstein e n o número de pontos de controle. Embora as curvas de *Bézier* ofereçam uma maneira intuitiva de manipulação, sofrem de alguns problemas (PIEG; TILLER, 1997):

- Curvas de elevado grau são requeridas para satisfazer um número alto de restrições, como na interpolação de pontos, por exemplo. Isso as tornam instáveis numericamente;
- Um elevado grau é requerido para ajustar as curvas a um conjunto de pontos – regressão;
- Embora as curvas de *Bézier* possam ser ajustadas pelos seus pontos de

controle, seu ajuste não é suficientemente local[10].

A solução é usar curvas que sejam polinomiais ou racionais por trechos, como se ela fosse uma emenda de várias outras. Dessa forma, é possível ajustar muitos pontos com uma curva de grau baixo, que geralmente tem comportamento melhor para a modelagem e desenho. Um exemplo desse tipo de curva são as B-*splines*, definidas por (PIEGL e TILLER, 1997):

$$\boldsymbol{C}(\bar{t}) = \sum_{i=0}^{n} N_{i,p}(\bar{t})\boldsymbol{P}_i\,; \qquad \alpha \leq t \leq \beta \tag{2.9}$$

onde $N_{i,p}$ são as funções de base de grau p da B-spline que podem ser calculadas pela fórmula recursiva de Boor, Cox e Mansfield (PIEG; TILLER, 1997):

$$\begin{gathered} N_{i,0}(\bar{t}) = \begin{cases} 1, & se\ t_i \leq \bar{t} < t_{i+1} \\ 0, & caso\ contrário \end{cases}\ ; \qquad \alpha \leq t \leq \beta \\ N_{i,p}(t) = \frac{\bar{t} - t_i}{t_{i+p} - t_i} N_{i,p-1}(t) + \frac{t_{i+p+1} - \bar{t}}{t_{i+p+1} - t_{i+1}} N_{i+1,p-1}(t) \end{gathered} \tag{2.10}$$

na qual t é o parâmetro da curva, p o seu grau, $m = n + p + 1$ e as variáveis discretas t_k formam o vetor não-decrescente de *knots* $T = \{t_0, \dots, t_m\}$, tipicamente utilizado na forma (PIEGL e TILLER, 1997):

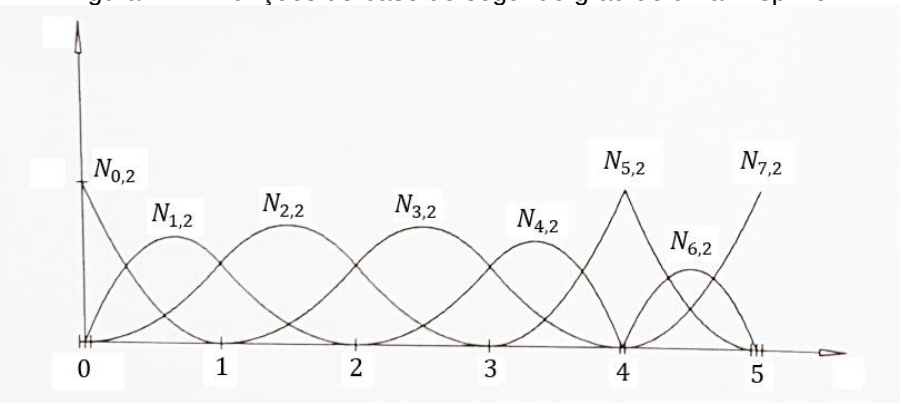
$$T = \{\underbrace{\alpha, \dots, \alpha}_{p+1}, t_{p+1}, \dots, t_{m-p-1}, \underbrace{\beta, \dots, \beta}_{p+1}\}$$

Esse vetor limita a atuação das funções de base, gerando a propriedade desejável de controle (ou suporte) local, como se pode observar na figura 2.4.

A propriedade de suporte local garante que $N_{i,p}(\bar{t}) = 0$ se t estiver fora do intervalo $[t_i, t_{i+p+1})$ (PIEGL; TILLER, 1997). De fato, observa-se na figura 2.4 que as funções de base respeitam essa propriedade. Os valores numéricos do eixo das abcissas marcam os *knots*, nesse caso $T = \{0,0,0,1,2,3,4,4,5,5,5\}$. Reparar que a multiplicidade de *knots* no início e no fim do vetor garante que a B-spline obrigatoriamente passará pelo ponto de controle inicial e final, conforme a figura 2.5:

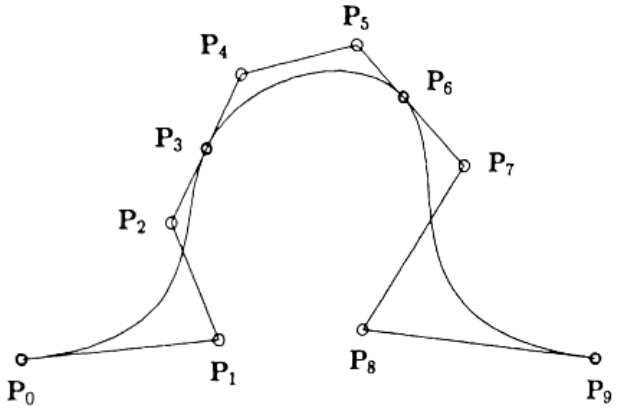
[10] Isso quer dizer que ao se deslocar um ponto de controle da curva, grande parte dela é modificada, ao invés de somente uma região próxima ao ponto manipulado.

Figura 2.4 – Funções de base de segundo grau de uma B-spline.



Fonte: Adaptado de Piegl e Tiller (1997)

Figura 2.5 – Representação de uma B-spline e seu polígono de controle. Reparar que a curva coincide com os pontos inicial e final devido à multiplicidade do *knots*.



Fonte: Piegl e Tiller (1997)

Se os pontos de controle ($\boldsymbol{P_i}$) forem conhecidos, determinar os pontos da B-spline ($\boldsymbol{C}(\bar{t})$) é relativamente simples:

- Define-se um vetor de knots (T) de tamanho $m+1$ com $p+1$ multiplicidade nas extremidades. Esse vetor varia de 0 a 1 e pode ser uniformemente espaçado nesse intervalo ou não;
- Para um determinado parâmetro $\bar{t}$, encontra-se a i-ésima posição em T que respeite $t_i \leq \bar{t} < t_{i+1}$;
- Calcula-se as funções de base não nulas por meio da equação (2.10);
- Por meio da equação (2.9) determina-se o ponto na curva correspondente ao parâmetro $\bar{t}$.

Em muitos casos o usuário deseja traçar uma curva por meio da entrada de pontos de interpolação ($\boldsymbol{Q}_k$) ao invés de pontos de controle ($\boldsymbol{P_i}$). Conforme a equação (2.9) tem-se:

$$\boldsymbol{Q}_k = \boldsymbol{C}(\bar{t}_k) = \sum_{i=0}^{n} N_{i,p}(\bar{t}_k)\boldsymbol{P}_i \tag{2.11}$$

Primeiramente é preciso criar uma parametrização para a curva. Há três métodos comuns para se fazer isso: igual espaçamento, comprimento da corda e o centrípeto. O método centrípeto é mais adequado que os demais quando os pontos resultam em curvas com trechos de pequeno raio (PIEG; TILLER, 1997):

$$\begin{gathered} \bar{d} = \sum_{k=1}^{n} \sqrt{|\boldsymbol{Q}_k - \boldsymbol{Q}_{k-1}|} \\ \bar{t}_0 = 0 \qquad\qquad \bar{t}_n = 1 \\ \bar{t}_k = \bar{t}_{k-1} + \frac{\sqrt{|\boldsymbol{Q}_k - \boldsymbol{Q}_{k-1}|}}{\bar{d}} \qquad k = 1, \dots, n-1 \end{gathered} \tag{2.12}$$

Em relação aos knots, pode-se utilizar a técnica da média (PIEG; TILLER, 1997):

$$\begin{gathered} t_0 = \cdots = t_p = 0 \qquad\qquad t_{m-p} = \cdots = t_m = 1 \\ t_{j+p} = \frac{1}{p} \sum_{i=j}^{j+p-1} \bar{t}_i \qquad\qquad j = 1, \dots, n-p \end{gathered} \tag{2.13}$$

Reescrevendo a equação (2.11) como:

$$\boldsymbol{Q} = \mathbb{N}\boldsymbol{P} \tag{2.14}$$

na qual $\mathbb{N}$ é uma matriz formada pelas funções de base, os pontos de controle podem ser encontrados pela resolução do sistema linear (2.14).

Uma extensão das B-splines são as NURBS (*Non-Uniform Rational B-Splines*) que possuem forma muito parecida com a equação (2.11), porém com um fator que pode ser diferente de 1 no denominador. As NURBS estão presentes na maioria dos softwares de computação gráfica de hoje e o livro de Piegl e Tiller (1997) trata com detalhes esses elementos. Como as NURBS são uma generalização das B-Splines, as funções de NURBS da biblioteca GLU do OpenGL (SHREINER *et al*., 2013) podem ser utilizadas para exibi-las.

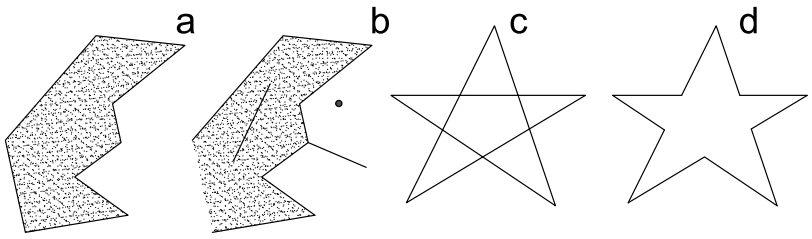
Convém lembrar que as curvas podem ser aproximadas por segmentos de reta e assim representadas, às custas de mais memória.

2.2 PRINCIPAIS TÉCNICAS DE MODELAGEM DE SÓLIDOS

Para fins de implementação computacional, um sólido é uma região delimitada e fechada de um subconjunto de E^3 (espaço euclidiano tridimensional) e que além disso apresenta essas propriedades:

- Rigidez: invariante às transformações rotacionais e translacionais (MÄNTYLÄ, 1988);
- Regularidade, isto é, suas entidades não podem ter partes isoladas, tal como ilustra a figura 2.6 a) e b) para subconjuntos de E^2;
- Finitude: os elementos que perfazem o sólido não devem possuir singularidades. Na prática, é conveniente admitir que as superfícies sejam algébricas ou pelo menos analíticas (MÄNTYLÄ, 1988);
- Superfícies sem auto-intersecção: as arestas de uma superfície não devem se intersectar, a não ser em suas extremidades.

Figura 2.6 – Regularidade na representação de sólidos: a) subconjunto regular b) subconjunto não-regular c) superfícies com arestas auto-intersectantes d) superfície adequada.



Fonte: Elaboração do próprio autor

Os sólidos precisam ser representados de uma maneira conveniente que permita a obtenção de suas propriedades geométricas (volume, centro de massa, normais da superfície, etc.), a sua combinação com outros sólidos (intersecção, subtração, adição) e, para o caso dos métodos numéricos, a geração de malha de tal forma que os elementos em faces adjacentes se conectem. Jansen (1987), Mäntylä (1988) e Hoffman (1992) descrevem três tipos principais de representação: por decomposição, por construção e por representação de superfícies (B-REP).

2.2.1 Técnicas de decomposição

Define-se um envelope como uma caixa que engloba totalmente o objeto (superfície ou sólido). As coordenadas de qualquer um de seus vértices podem ser

$\{X_{e+}, X_{e-}, Y_{e+}, Y_{e-}, Z_{e-}, Z_{e+}\}$ e as coordenadas extremas do objeto são $\{X_{mín}, X_{máx}, Y_{mín}, Y_{máx}, Z_{mín}, Z_{máx}\}$. Então $\{X_{e-} < X_{mín}\}$, $\{X_{e+} > X_{máx}\}$ e assim por diante. Em suma, nenhum vértice do objeto deve ficar fora do envelope.

As estruturas de decomposição são geradas por meio da divisão sucessiva e hierárquica de células[11], de tal sorte que a célula pai de hierarquia superior é o próprio envelope do objeto. Uma das maneiras de se gerar as árvores é mostrada na instrução 2.1:

Instrução 2.1 - Geração de quadtree ou octree

a) Se a célula atual for passível de divisão conforme algum critério, divida-a em quatro (ou oito para 3D) outras células – denominadas células filhas;

b) Para cada uma das filhas, realizar nova divisão se ela se intersectar com uma aresta ou face do objeto ou ainda se o seu tamanho atual é maior que um limite especificado;

c) Armazenar o endereço de cada célula e demais informações que possam ser úteis, tal como o endereço das células vizinhas. Voltar em a).

Quando a decomposição ocorre em um espaço bidimensional, diz-se que a estrutura resultante é um *quadtree*, enquanto no espaço tridimensional essa estrutura recebe o nome de *octree*. De fato, o constructo lógico dessas entidades assemelha-se a uma árvore, como pode ser observado na figura 2.7.

Denomina-se folha, uma célula do nível hierárquico mais baixo que possua interesse (contém alguma intersecção, um nó, elemento, etc.) na árvore. As células órfãs são aquelas do nível hierárquico mais baixo e que não possuem interesse. O endereço de cada célula na árvore pode ser indicado por um vetor. Por exemplo, a célula marcada com uma seta na figura 2.7 tem o endereço $\{2,1,3\}$[12].

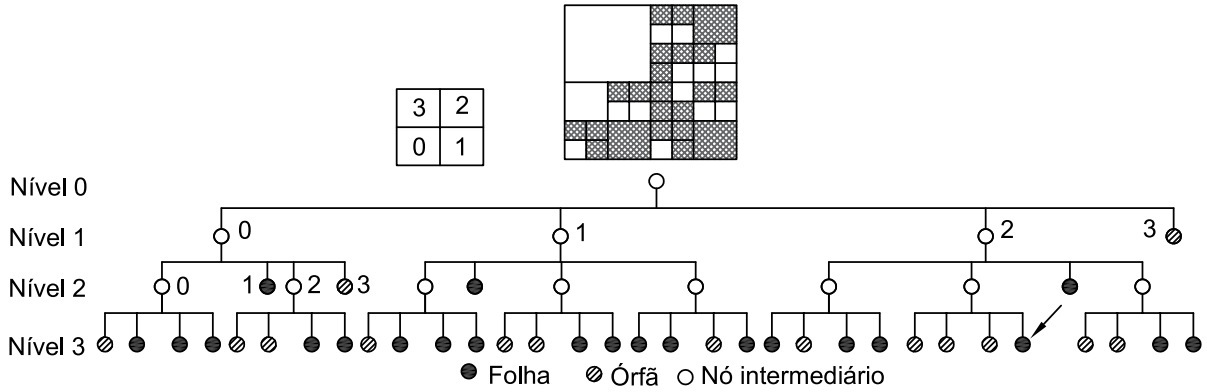
A figura 2.8 mostra dois exemplos de decomposição. O *quadtree* é gerado por um algoritmo de divisão baseado na intersecção com os contornos da superfície. Esse tipo de *quadtree* é apropriado para a representação da superfície, e também geração de malha como mostrado por Bern e Plasmann (2000). O *octree* mostrado na figura é

[11] No contexto desse trabalho essas células podem ser quadrados, retângulos, cubos e paralelepípedos. O uso da palavra célula no livro de Mäntylä é um pouco diferente.

[12] Não é necessário marcar o endereço do nível 0 por razões óbvias. Aqui foram usados números inteiros para o endereço. Na programação, pode-se utilizar um tipo de variável que use menos memória, tal como o *char* ou *byte*.

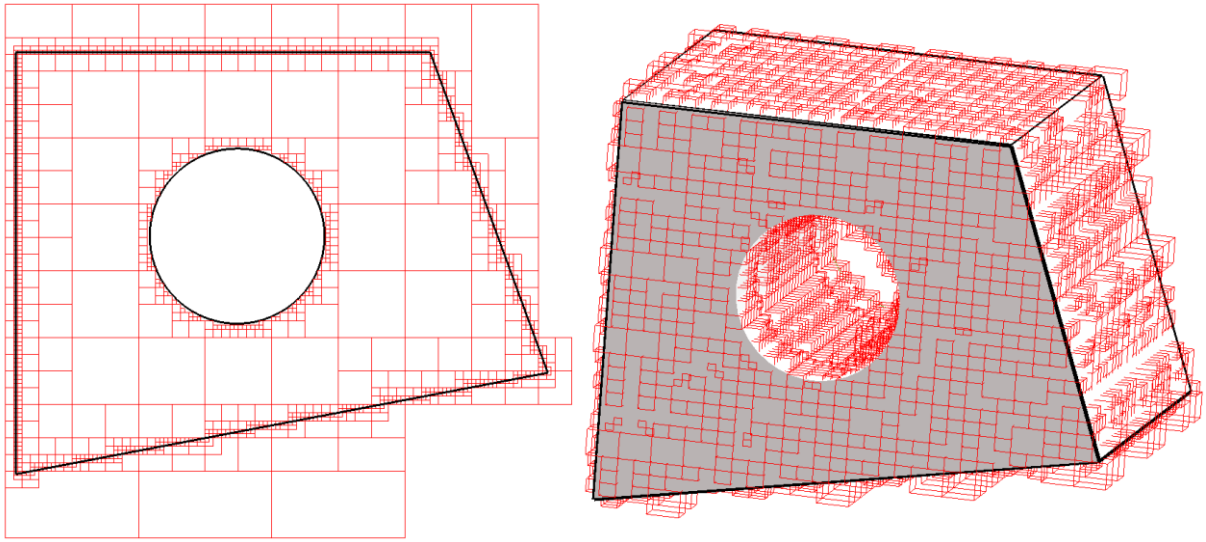
para uso do MEC com multipolos rápidos e a divisão dos cubos se dá pelo número máximo aceitável de elementos (não mostrados aqui) no interior de cada célula, ao invés das intersecções com a superfície do objeto. Dessa maneira, podem existir regiões da superfície sem células.

Figura 2.7 – Esquema de uma árvore para um quadtree



Fonte: Elaboração do próprio autor

Figura 2.8 – Decomposições geradas pelo CABEMT: à esquerda – de uma superfície, correspondendo a um *quadtree* (células internas ao círculo foram omitidas); à direita – de um sólido, correspondendo a um *octree*



Fonte: Elaboração do próprio autor

Percebe-se que a criação de sólidos ou superfícies diretamente é uma tarefa muito difícil. Na realidade, os modelos de decomposição não são utilizados na construção direta e sim, para auxiliar ou acelerar outras operações, tais como a geração de malha, algoritmos de inflação (MÄNTYLÄ, 1988), algoritmos de classificação de pontos[13] e operações booleanas[14]. Mäntylä (1988) cita que o *software*

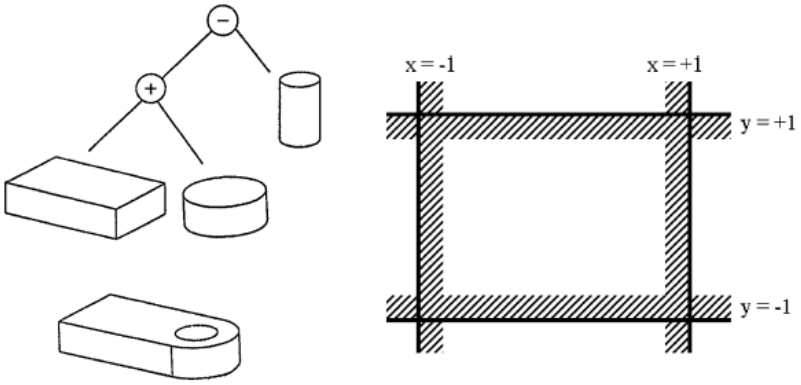
[13] Por exemplo, determinar se um ponto está dentro ou fora de um sólido.
[14] Por exemplo: adição, subtração e intersecção entre sólidos.

MCS TIPS usa um *octree* regular tridimensional para acelerar vários algoritmos do programa.

2.2.2 Técnica da construção geométrica de sólidos (CGS)

Essa técnica busca representar os sólidos por meio da combinação booleana de outros sólidos elementares, tais como: caixa retangular, cilindro, cone, esfera, toro e cunha (STROUD; NAGY, 2011).

Figura 2.9 – Criação de objetos pela técnica da construção.

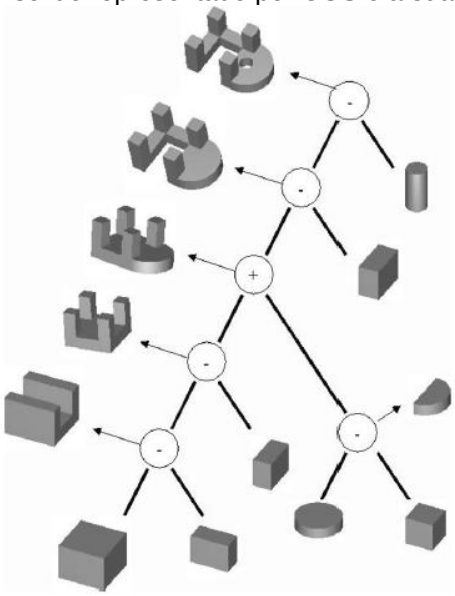


Fonte: Stroud e Nagy (STROUD e NAGY, 2011)

A figura 2.9 mostra a combinação de sólidos para formação de um olhal, à esquerda – técnica da CGS. À direita percebe-se a formação de um retângulo pela combinação dos semi-espaços: $\{(x \geq -1) \wedge (x \leq 1) \wedge (y \geq -1) \wedge (y \leq 1)\}$ – modelo de semi-espaços. Este último usa uma descrição lógico-matemática dos objetos e são menos intuitivos que a construção por blocos elementares, mas por sua vez, tais blocos podem ser interpretados como modelos de semi-espaços que se interceptam sem deixar regiões abertas. Dessa forma, a representação por sub-espaços é de certa forma o fundamento da CGS.

A CGS pode ser descrita por uma árvore, cujas folhas correspondem às entidades elementares, ao passo que os nós intermediários correspondem às operações booleanas entre dois sólidos elementares (duas folhas) ou entre o sólido que está sendo construído e outro elementar ou ainda a uma operação de rotação ou translação. Um exemplo de árvore de CGS pode ser visto na figura a seguir:

Figura 2.10 – Um sólido representado por CGS e a sua árvore geradora



Fonte: Stroud e Nagy (STROUD; NAGY, 2011)

O problema de como descrever um objeto por meio da junção de sólidos mais simples por ser resolvido pelas árvores, contudo, ainda resta uma questão: como a imagem desses sólidos poderá ser formada?

Para exibir o sólido, a representação deve ser convertida em um tipo de representação volumétrica (como no item 2.2.1) ou por fronteiras (item 2.2.3). Contudo essas conversões podem ser lentas (JANSEN, 1987).

A CGS pode representar e guardar todas as informações construtivas do objeto, facilitando a implementação da função desfazer, crucial no modelamento sólido. Além disso, as fronteiras de cada elemento da CGS possuem uma função matemática conhecida. Apesar dessas vantagens, a representação por CGS possui algumas desvantagens importantes:

- Normalmente a árvore que representa um determinado objeto não é unívoca, impedindo a comparação entre dois objetos somente através de suas árvores associadas (STROUD; NAGY, 2011);
- Nem sempre é conveniente realizar uma modificação do sólido por meio de operações booleanas[15] (STROUD; NAGY, 2011). Detalhes como esse fazem da modelagem construtiva bastante morosa se comparada as técnicas de

[15] Por exemplo: as operações locais de chanfro e arredondamento são mais fáceis de se utilizar na representação por fronteira. De fato, um usuário de um sistema baseado em CGS não gostaria de precisar criar o “negativo” do chanfro para depois subtraí-lo do modelo principal. Ele poderia fazer isso com apenas alguns comandos no caso da representação por fronteira.

modificação local da geometria permitida pela modelagem de fronteira;

- Operações de extrusão são feitas de forma muito indireta na CGS (STROUD; NAGY, 2011) e esse tipo de operação, aliada à de varredura, é amplamente empregada na maneira moderna de criar sólidos, uma vez que são bastante intuitivas.

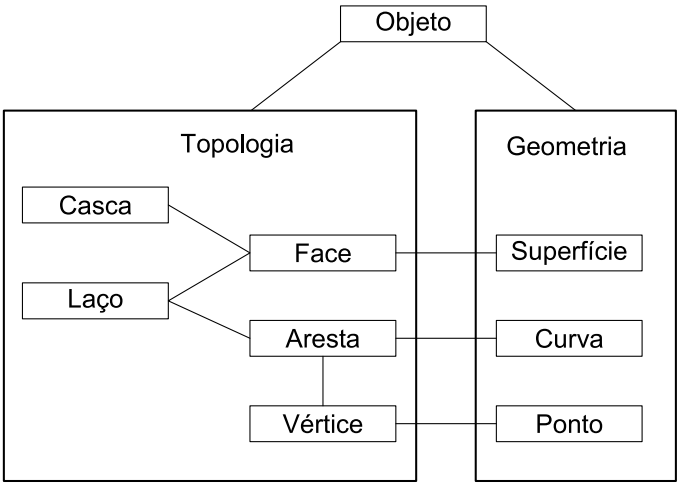
Devido a essas características, a CGS, que já foi amplamente empregada em softwares comerciais (MÄNTYLÄ, 1988), hoje perdeu espaço para a representação por fronteiras, em virtude dessa ser mais completa e permitir a criação do modelo de maneira mais rápida, em geral. Contudo a CGS tem despertado interesse no campo de reconstrução de modelos a partir de nuvens de pontos ou malhas de superfície. Du, Liu e Dong (2018), por exemplo, elaboraram um algoritmo de engenharia reversa para transformar uma malha grosseira em uma representação completa em CGS.

2.2.3 Técnica da representação por fronteiras (B-rep)

2.2.3.1 B-REP – Técnica, Geometria e Topologia

Essa forma de representação (*Boundary Representation* – B-rep) tem sido largamente utilizada nos softwares de modelagem atualmente (STROUD; NAGY, 2011) e também foi escolhida para representar os sólidos no CABEMT. A figura a seguir apresenta um esquema lógico de uma B-REP:

Figura 2.11 – Esquema lógico da representação de um objeto por fronteira (B-REP)



Fonte: Adaptado de Stroud e Nagy (2011)

O esquema separa as entidades topológicas das geométricas, mas isso não é estritamente necessário na implementação computacional. Para ilustrar isso, pode-se imaginar duas classes: vértice e ponto. O vértice pode ter como variável uma entidade

da classe ponto, além das informações de conectividade (topologia). Tudo isso ficaria armazenado em uma mesma estrutura de dados ou objeto. A seguir serão apresentadas sucintamente algumas características dos elementos apontados na figura 2.11:

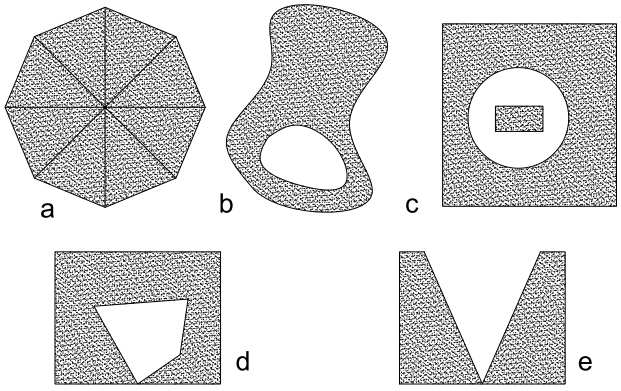
- Ponto: é um conjunto de coordenadas que representam uma posição. Deve ter pelo menos um número identificador para facilitar sua manipulação e localização na estrutura de dados;
- Curva: pode ser uma linha, um conjunto de linhas descrevendo uma curva ou uma equação da curva. Nesse último caso, pelo menos a equação paramétrica e os valores inicial e final do parâmetro da curva são necessários;
- Superfície: esse elemento pode conter a equação que o descreve e/ou conjunto de faces que o constitui. É comum na B-rep o requisito de que as superfícies devam ser variedades[16] orientáveis. Uma variedade bidimensional tem a propriedade que todos os seus pontos apresentem uma vizinhança que seja homeomorfa ao plano, ou seja, podemos deformar a superfície localmente para que a região se torne um plano, sem que isso leve a um “rasgamento” ou separação de pontos na superfície. A variedade será orientável se é possível distinguir dois lados diferentes da superfície (HOFFMANN, 1992). No CABEMT, uma superfície é definida como um conjunto de faces. As informações geométricas e topológicas ficam armazenadas nas faces, ao passo que a superfície é apenas objeto um agregador;
- Vértices: são os elementos fundamentais de conexão. No sólido, podem se conectar a um número arbitrário de arestas. Em uma face, conectam-se obrigatoriamente a apenas duas arestas (quatro semi-arestas) para variedades em $\mathbb{R}^2$. Vértices podem estar associados a pontos por referência ou ponteiros;
- Aresta: conectam-se obrigatoriamente a dois vértices;
- Semi-aresta: pares de semi-arestas estão sempre associados a uma aresta e em sentidos opostos conforme figura 2.13. Cada semi-aresta pertence a apenas uma face que por sua vez é constituída por um conjunto de semi-arestas denominado laço, orientado de modo que ao percorrer as arestas orientadas, a face fica sempre

[16] Do inglês *manifolds*

à esquerda delas;

- Cascas são regiões completamente fechadas internas a um sólido;
- Face: contém pelo menos um número identificador e as referências às semi-arestas (ou arestas + sentido) que a compõe. Faces também podem conter informações geométricas, tais como sua normal, área, centroide, nós e elementos associados, etc. As faces convenientes para a B-rep podem ter muitas curvas no seu contorno, desde que formem um objeto conexo. As faces são como "continentes" que podem ter "lagos", mas não "ilhas" (MÄNTYLÄ, 1988). Na figura 2.12 o caso (a) representa uma superfície formada por um conjunto de faces, perceber como o vértice central pode se conectar a um número arbitrário de arestas. No caso (b) tem-se uma face e furo delimitados por curvas. As semi-arestas do furo possuem a mesma orientação que as semi-arestas da face. A figura em (c) deve ser separada em duas faces. O caso (d) pode ser aceito como uma representação válida (não é o caso do CABEMT) – reparar que essa face, e outras obtidas após a sua extrusão, não constituem em uma variedade em $\mathbb{R}^2$, assim como no caso (e). Neste último, é melhor dividir a face em duas, ao passo que (d) poderia ser tratado como um caso especial.

Figura 2.12 – Exemplos de faces

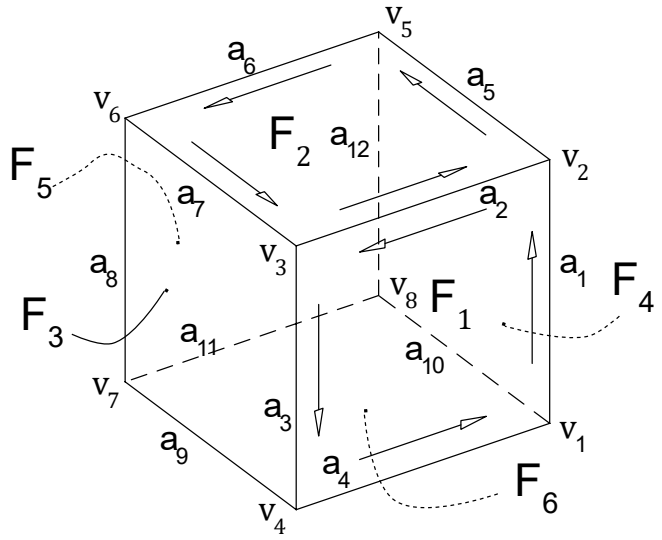


Fonte: Adaptado de Mäntylä (1988)

A programação orientada a objeto (POO) se mostra bastante adequada à B-rep devido à interdependência e organização hierárquica das entidades topológicas e geométricas, cada qual podendo ser considerada como um objeto (ponto, aresta, curva, etc.). Há muitas formas de se estruturar a B-rep, sendo o esquema da figura 2.11 um exemplo possível. O uso de ponteiros ou referências permite uma busca eficiente pelas conexões topológicas e o seu armazenamento não consome muita memória, tal característica será melhor discutida no capítulo 4.

A figura 2.13 mostra um cubo e as entidades que o constituem: vértices (V_i), arestas (a_i), semi-arestas (a_{ij}) e faces (F_i).

Figura 2.13 – Um cubo e suas entidades topológicas associadas



Fonte: Elaboração do próprio autor

Tabela 2.1 – Conectividade topológica dos elementos de um cubo. Alguns itens foram omitidos (-)

Vért.	Aresta	Semi-aresta	Aresta	Semi-Aresta	Face
V_1	-	-	-		
V_2	$a_1(V_1 \leftrightarrow V_2)$	$a_{12}(V_1 \rightarrow V_2)$ $a_{21}(V_2 \rightarrow V_1)$	$a_7(V_6 \leftrightarrow V_3)$	$a_{25}(V_2 \rightarrow V_5)$ $a_{52}(V_5 \rightarrow V_2)$	$F_1(a_{12} \rightarrow a_{23} \rightarrow a_{34} \rightarrow a_{41})$
V_3	$a_2(V_2 \leftrightarrow V_3)$	$a_{23}(V_2 \rightarrow V_3)$ $a_{32}(V_3 \rightarrow V_2)$	$a_8(V_6 \leftrightarrow V_7)$	$a_{56}(V_5 \rightarrow V_6)$ $a_{65}(V_6 \rightarrow V_5)$	$F_2(a_{32} \rightarrow a_{25} \rightarrow a_{56} \rightarrow a_{63})$
V_4	$a_3(V_3 \leftrightarrow V_4)$	$a_{34}(V_3 \rightarrow V_4)$ $a_{43}(V_4 \rightarrow V_3)$	$a_9(V_7 \leftrightarrow V_4)$	.	.
V_5	$a_4(V_2 \leftrightarrow V_5)$	$a_{41}(V_4 \rightarrow V_1)$ $a_{14}(V_1 \rightarrow V_4)$	$a_{10}(V_1 \leftrightarrow V_9)$	.	.
V_6	$a_5(V_5 \leftrightarrow V_6)$	.	$a_{11}(V_7 \leftrightarrow V_8)$	.	.
V_7	$a_6(V_1 \leftrightarrow V_2)$	.	$a_{12}(V_8 \leftrightarrow V_5)$		
V_8		.			

Fonte: Elaboração do próprio autor

2.2.3.2 A fórmula de Euler-Poincaré

A validade da representação simbólica da topologia de um sólido está sujeita a alguns erros, principalmente devido a:

- Entrada de parâmetros incorretos pelo usuário associada à ausência de verificador de consistência;
- Erros em operações de ponto flutuante podem fazer com que a classificação de um vértice, por exemplo, seja incorreta e afete em cadeia as entidades topológicas associadas. Por exemplo: um vértice pode ter sido classificado como pertencente a uma aresta quando na realidade não é;

- Erro lógico oculto em algum algoritmo do programa, especialmente naqueles de operações booleanas devido a sua complexidade.

Então é importante haver uma maneira de verificar a validade da representação topológica. Se considerarmos um sólido simples, isto é, cujas faces sejam contornadas por um único laço e que tal sólido não possua vazios ou furos, então a seguinte fórmula de Euler-Poincaré é válida (HOFFMANN, 1992):

$$V - a + F - 2 = 0 \tag{2.15}$$

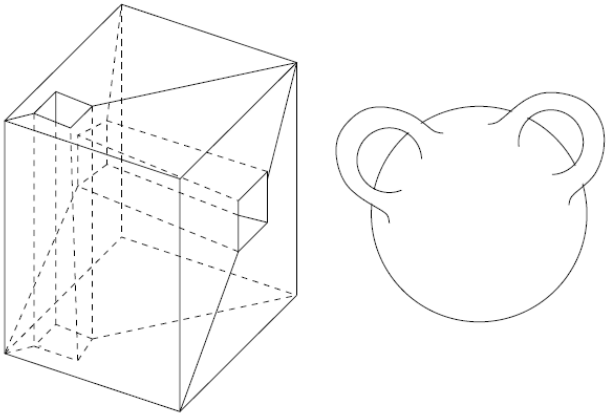
na qual V é o número de vértices, a o número de arestas e F o número de faces.

Se o sólido possui furos passantes, então é homeomorfo a uma esfera com uma alça. O número dessas alças é denominado genus da superfície (G). Assim, a equação de Euler-Poincaré fica (HOFFMANN, 1992):

$$\mathrm{V} - a + \mathrm{F} - 2(1 - G) = 0 \tag{2.16}$$

A figura a seguir mostra dois sólidos com genus = 2.

Figura 2.14 – Dois sólidos homeomorfos com genus igual a 2



Fonte: Hoffmann (1992)

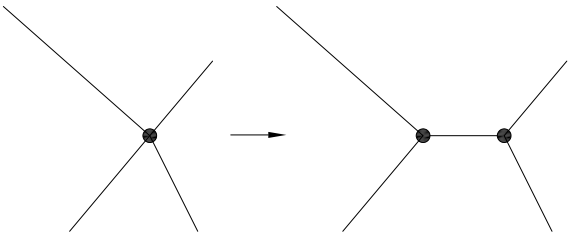
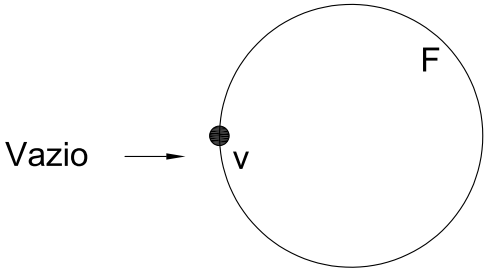
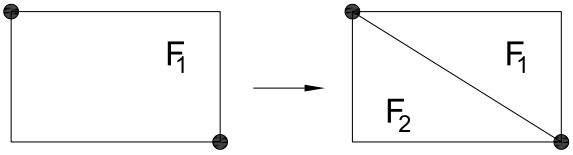
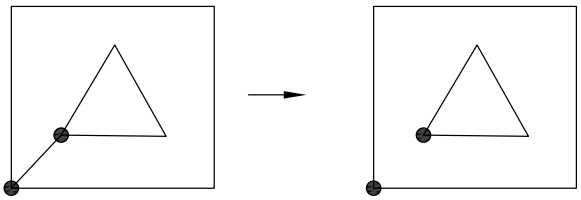
2.2.3.3 Operadores de Euler

Os operadores de Euler podem ser imaginados como uma forma de criar e modificar a topologia de variedades de maneira robusta e consistente. Eles podem criar superfícies fechadas e modificá-las por meio da adição e exclusão de faces, arestas e vértices. Os operadores de Euler são usados como se fossem uma camada de abstração intermediária em alguns sistemas (HOFFMANN, 1992).

Mäntylä (1988) utiliza os operadores como base para seu programa MCS GWB (*geometric workbench*). Esses operadores normalmente possuem comandos e entidades. Os comandos podem ser: fazer (*make*), eliminar (*kill)*, dividir (*split*), unir

(*join*). E as entidades: vértice (*vertex*), aresta (*edge*), face (*face*), sólido (*solid*), buraco (hole) e anel (*ring*). Se a letra C representar um comando e E uma entidade topológica, então os operadores de Euler podem ser tipicamente representados por $CECE$, CEE, $CEEE$, $CECEE$, etc. Por exemplo: MEV significa fazer uma aresta e vértice (*make edge vertex)*. A figura 2.15 exemplifica alguns desses comandos:

Figura 2.15 – Alguns operadores de Euler comuns

Operador de Euler	Ilustração
MEV *Make Edge Vertex* Cria um vértice a partir de outro e uma aresta que liga os dois	
MVFS *Make Vertex Face Solid* *Cria um vértice, uma face vazia e um sólido que contém essa face*	
MEF *Make Edge Face* *Cria uma aresta e uma nova face*	
KEML *Kill Edge Make Loop* *Elimina uma aresta e faz um laço*	

Fonte: Adaptado de Mäntylä (1988)

2.2.3.4 Estrutura de Dados na B-rep

Há muitas formas de se armazenar a topologia de um sólido na memória do computador. Algumas maneiras podem ser mais compactas e favorecem o armazenamento, porém requerendo mais processamento durante a modelagem. Outras formas mais ricas em detalhes e até com alguma redundância de informações podem ocupar mais espaço na memória, porém facilitam as inúmeras operações computacionais ao longo da modelagem, possibilitando acesso mais rápido ou direto

às entidades topológicas. Contudo, não há empecilho em usar uma representação[17] mais compacta para armazenamento do arquivo e no momento de carregá-lo no programa, construir a representação que melhor se adapte à modelagem e processamento. Isso ocorre porque a informação elementar de construção do sólido é a mesma, a maneira de acessar essa informação é que pode mudar conforme ela esteja explícita ou implícita na representação.

2.2.3.4.1. B-rep com estrutura baseada em vértices

Este é o modelo mais simples de representação e aplicável a modelos poliédricos, ou seja, nos quais todas as faces são planares. Consiste em uma lista principal de vértices e suas coordenadas, além de uma lista de faces contendo a sequência de vértices que a perfazem, na orientação correta. Esta última é uma lista de ponteiros ou referências aos elementos da lista de vértices. Para efeito de demonstração, consideremos a representação das faces F_1 e F_2 do cubo ilustrado na figura 2.13 conforme a tabela a seguir:

Tabela 2.2 – Estrutura do cubo da figura 2.13 baseada em vértices

Vértice	Coordenadas	Face	Vértices
V_1	$x_1 \quad y_1 \quad z_1$	F_1	$V_1 \quad V_2 \quad V_3 \quad V_4$
V_2	$x_2 \quad y_2 \quad z_2$	F_2	$V_3 \quad V_2 \quad V_5 \quad V_6$
V_3	$x_3 \quad y_3 \quad z_3$	F_3	$V_4 \quad V_3 \quad V_6 \quad V_7$
V_4	$x_4 \quad y_4 \quad z_4$	F_4	$V_2 \quad V_1 \quad V_8 \quad V_5$
V_5	$x_5 \quad y_5 \quad z_5$	F_5	$V_6 \quad V_5 \quad V_8 \quad V_7$
V_6	$x_6 \quad y_6 \quad z_6$	F_6	$V_1 \quad V_4 \quad V_7 \quad V_8$
V_7	$x_7 \quad y_7 \quad z_7$		
V_8	$x_8 \quad y_8 \quad z_8$		

Fonte: Adaptado de Mäntylä (1988)

Essa forma estrutural, embora bastante compacta, certamente vai exigir mais processamento em tempo de execução, especialmente nas operações booleanas. Nesse caso, com frequência é necessário determinar intersecções com arestas que não estão representadas explicitamente na estrutura. Assim como as faces vizinhas às arestas, vértices sucessores e predecessores, etc.

2.2.3.4.2. B-rep com estrutura baseada em arestas

Nesse caso, as arestas são explicitamente armazenadas na estrutura, reduzindo o tempo de processamento na manipulação do sólido. Além de facilitar o acesso às informações topológicas, quando superfícies curvas estão presentes no

[17] O termo representação aqui se refere ao modo como os elementos são estruturados e armazenados na memória e não ao tipo de representação do sólido (B-rep, CGS, decomposição).

modelo, é útil incluir vértices explicitamente[18] nos dados de fronteira dessas curvas, quando gerados por intersecções diversas (MÄNTYLÄ, 1988). Nessa circunstância, é frequente a necessidade de computar intersecções dessas curvas com outras superfícies, ao se efetuar operações booleanas. A tabela abaixo exemplifica a representação baseada em arestas para o cubo da figura 2.13:

Tabela 2.3 – Estrutura do cubo baseada em arestas

Aresta	Vértices		Vértice	Coordenadas			Face	Arestas			
a_1	V_1	V_2	V_1	x_1	y_1	z_1	F_1	a_1	a_2	a_3	a_4
a_2	V_2	V_3	V_2	x_2	y_2	z_2	F_2	a_2	a_5	a_6	a_7
a_3	V_3	V_4	V_3	x_3	y_3	z_3	F_3	a_3	a_7	a_8	a_9
a_4	V_4	V_1	V_4	x_4	y_4	z_4	F_4	a_1	a_{10}	a_{12}	a_5
a_5	V_2	V_5	V_5	x_5	y_5	z_5	F_5	a_6	a_{12}	a_{11}	a_8
a_6	V_5	V_6	V_6	x_6	y_6	z_6	F_6	a_4	a_{10}	a_{11}	a_9
a_7	V_6	V_3	V_7	x_7	y_7	z_7					
a_8	V_6	V_7	V_8	x_8	y_8	z_8					
a_9	V_7	V_4									
a_{10}	V_1	V_8									
a_{11}	V_8	V_7									
a_{12}	V_5	V_8									

Fonte: Adaptado de Mäntylä (1988)

Todas as arestas são ordenadas no sentido anti-horário ou conforme a regra da mão direita aplicada às normais das faces. Cada aresta tem somente duas faces vizinhas, de modo que em uma face a aresta é percorrida num sentido e na outra, no sentido inverso.

2.2.3.4.3. B-rep com estrutura alada (*winged-edge*)

Essa representação foi introduzida por Baumgart (1974) e além de manter os elementos das estruturas mostradas anteriormente, associa sucessores e predecessores das arestas.

Considerando uma aresta $a_i(V_n \rightarrow V_m)$, a variável ncw (*next clockwise*) indica a próxima aresta na face onde a_i tem orientação positiva, ou seja, a sequência de vértices $V_n \rightarrow V_m$ coincide com a ordenação de vértices na face. Por outro lado, a variável $nccw$ (*next counterclockwise*) indica a próxima aresta da face onde a_i é percorrida no sentido negativo, ou seja, a face que contém uma sequência de vértices igual a $V_m \rightarrow V_n$. A tabela a seguir ilustra essa estrutura para o cubo da figura 2.13,

[18] Não se trata obrigatoriamente de representar uma curva por vários segmentos de reta ou sequência de vértices. A representação pode ser utilizada somente para determinar de maneira rápida uma primeira aproximação do ponto de intersecção entre uma aresta e uma superfície. O restante da curva é determinado por algoritmos de traçagem, tal como em Hoffmann (1992).

porém desconsiderar a orientação mostrada na ilustração, neste caso, todas as faces estão orientadas no sentido horário quando vistas de fora do cubo:

Tabela 2.4 – Estrutura topológica tipo alada (*winged-edge*) - parte I

Aresta	V_{inicio}	V_{fim}	ncw	nccw
a_1	V_1	V_2	a_5	a_4
a_2	V_2	V_3	a_7	a_1
a_3	V_3	V_4	a_9	a_2
a_4	V_4	V_1	a_{10}	a_3
a_5	V_2	V_5	a_{12}	a_2
a_6	V_5	V_6	a_8	a_5
a_7	V_6	V_3	a_3	a_6
a_8	V_6	V_7	a_{11}	a_7
a_9	V_7	V_4	a_4	a_8
a_{10}	V_1	V_8	a_{11}	a_1
a_{11}	V_8	V_7	a_9	a_{12}
a_{12}	V_5	V_8	a_{10}	a_6

Fonte: Adaptado de Mäntylä (1988)

Tabela 2.5 – Estrutura topológica tipo alada (*winged-edge*) - parte II

Vértice	Coordenadas	Face	Primeira aresta	Sinal
V_1	x_1 y_1 z_1	F_1	a_1	-
V_2	x_2 y_2 z_2	F_2	a_2	+
V_3	x_3 y_3 z_3	F_3	a_3	+
V_4	x_4 y_4 z_4	F_4	a_1	+
V_5	x_5 y_5 z_5	F_5	a_6	+
V_6	x_6 y_6 z_6	F_6	a_4	-
V_7	x_7 y_7 z_7			
V_8	x_8 y_8 z_8			

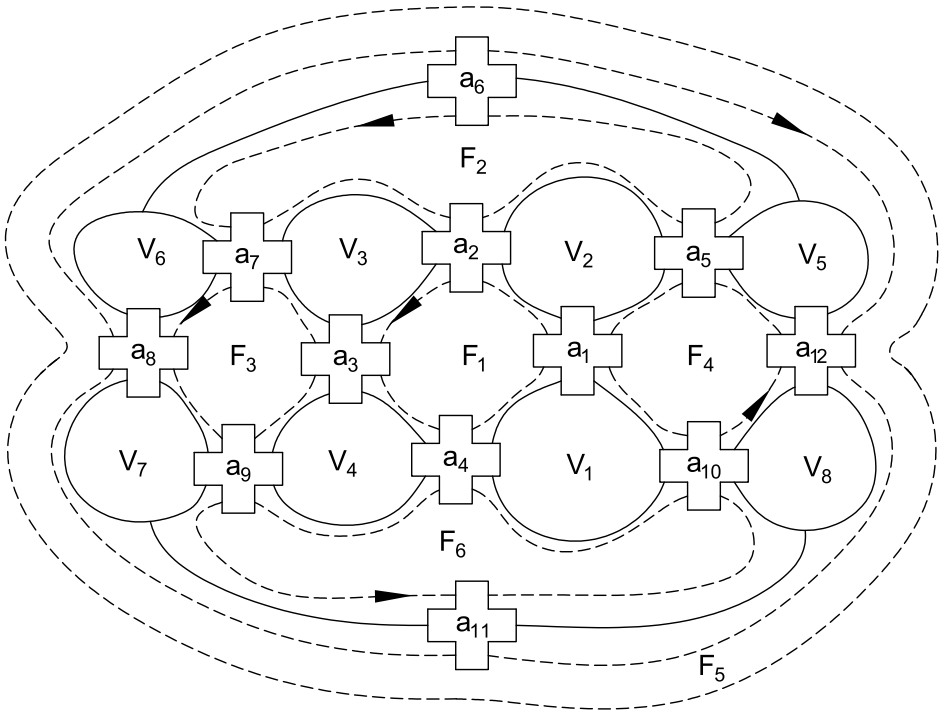
Fonte: Adaptado de Mäntylä (1988)

2.2.3.4.4. B-rep com estrutura *quad-edge*

Essa estrutura foi introduzida por Guibas e Stolfi (1985) com fins à geração de malha e muitos artigos científicos a utilizam. Não há empecilho, porém, à representação de sólidos. Como o próprio nome indica, as arestas são o principal elemento dessa representação. Cada uma guarda referência para outras quatro: antecessora e sucessora na face à esquerda e antecessora e sucessora na face à direita. Além disso, cada *quad-edge* guarda as referências de seus vértices. O cubo da figura 2.13 ficaria representado como na figura 2.16.

Guibas e Stolfi (1985) assinalam que a provável maior vantagem da estrutura *quad-edge* sobre as demais é que a modificação e construção da topologia pode ser feita com apenas dois operadores básicos. Eles aplicaram essa estrutura e seus operadores para efetuar a triangulação de domínios, algo a ser visto no item 3.3.2.

Figura 2.16 – Um cubo representado graficamente pela estrutura de *quad-edge*



Fonte: Elaboração do próprio autor

2.2.3.4.5. Estrutura triangular

É uma estrutura dedicada à representação de faces ou elementos triangulares. Cada triângulo armazena os seus três vizinhos e seus três vértices. O espaço necessário para armazenar uma *quad-edge* ou estrutura triângulo é o mesmo, porém, quando consideradas as redundâncias no agrupamento de vários elementos, a estrutura triangular é mais eficiente em termos de armazenamento e em velocidade de acesso a seus elementos (SHEWCHUK, 1996).

2.3 OPERAÇÕES BÁSICAS PARA GERAÇÃO DE SÓLIDOS

A geração dos sólidos é uma etapa que envolve bastante interface humana e por isso é importante que tal processo se dê de maneira natural e amigável ao usuário, dando liberdade criativa e possibilidade de experimentar diferentes abordagens.

Os modelos de decomposição certamente não são adequados para o trabalho humano, visto que seria extremamente laborioso informar vértice por vértice do *octree* de um sólido ou algum conjunto de informações equivalentes. É por isso que as técnicas de decomposição em geral processam um sólido existente, ao invés de criá-lo, a não ser que os dados de construção provenham de técnicas de escaneamento.

Na modelagem construtiva as coisas se tornam mais simples. O usuário deve decompor mentalmente o objeto a modelar, baseando-se nos sólidos primitivos disponíveis no software. Em seguida, criam-se os primitivos e aplicam-se operações de rotação e/ou translação para posicioná-los, caso já não tenham sido criados na posição adequada por meio de eixos de coordenadas auxiliares. Na sequência, as operações booleanas são aplicadas conforme necessário.

O paradigma da representação por fronteiras é certamente aquele que fornece as ferramentas mais intuitivas e que na maioria dos casos permitirá uma modelagem mais rápida. Devido à importância da B-rep e ao fato de sua implementação no CABEMT, serão discutidas adiante as maneiras mais comuns de se criar modelos sólidos nesse tipo de representação, ainda que algumas dessas técnicas também possam ser utilizadas na CGS (JANSEN, 1987), mas que são certamente mais simples de implementar e mais naturais à B-rep.

As instruções mostradas terão como pano de fundo a criação de sólidos poliédricos que é o tipo de objeto gerado pelo CABEMT. O conceito geral dessas instruções será muito similar para modelos mais exatos, cujas superfícies possam ser representadas por quádricos ou NURBS.

2.3.1 Geração direta

A geração direta é comum à CGS e à B-REP. Tal como o nome diz, consiste na criação de uma entidade sólida simples (caixa, cilindro, esfera, etc.) por meio de comandos ou pela IGU (interface gráfica do usuário). O exemplo abaixo mostra a criação de um cilindro (Ø 30 x 100, centro em 0,0,0) no AutoCAD®:

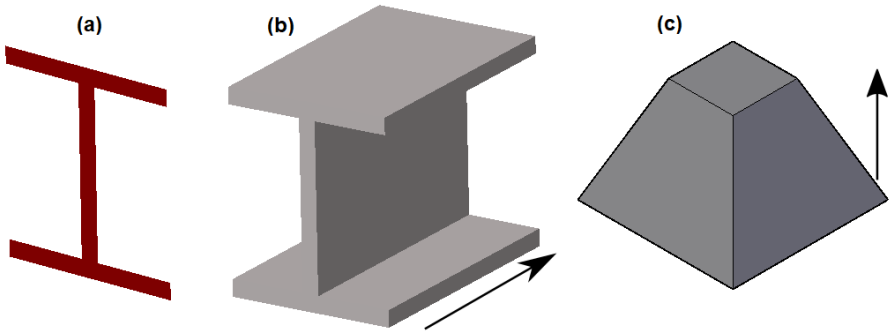
Command: CYLINDER
Specify center point of base or [3P/2P/Ttr/Elliptical]: 0,0
Specify base radius or [Diameter]: 30
Specify height or [2Point/Axis endpoint]: 100

A geração direta de sólidos não precisa se restringir somente às formas básicas. O software *Salome* (RIBES; BRUNETON; GEAY, 2017), por exemplo, dá opção de criar conexões de tubos diretamente, por meio de uma IGU. Entretanto, o mais comum é que a geração direta seja apenas uma etapa da modelagem –, a ela estarão tipicamente associadas outras operações para criação de um modelo complexo.

2.3.2 Varredura translacional (extrusão)

É a operação de varredura mais simples e a mais utilizada. Consiste em "arrastar" uma face ao longo de um caminho (normalmente retilíneo), formando assim o sólido. Na figura 2.17 (a) uma face correspondente a um perfil I foi criada e em seguida extrudada. O resultado é exibido em (b). O sólido mostrado em (c) passou por um processo de extrusão em ângulo.

Figura 2.17 – Exemplos de sólidos formados por extrusão no sentido indicado pelas setas



Fonte: Elaboração do próprio autor

O processo de extrusão será detalhado com base na figura 2.18 e na instrução a seguir:

Instrução 2.2 - Extrusão de uma face

Dados: uma face F composta pela sequência de vértices $V_1, V_2, V_3, \dots, V_N$ e as semi-arestas associadas $a_{1\to2}, a_{2\to3}, \dots, a_{N-1\to N}$, cuja normal é $\vec{n}_F$ e o vetor diretor da extrusão é $\vec{n}_e$.

a) Se $\vec{n}_f$ e $\vec{n}_e$ possuírem a mesma orientação, isto é, $\overrightarrow{n_f}.\overrightarrow{n_e} > 0$, então inverter a orientação de F. Não é absolutamente necessário que $\vec{n}_f$ e $\vec{n}_e$ sejam paralelos, mas não podem ser perpendiculares;

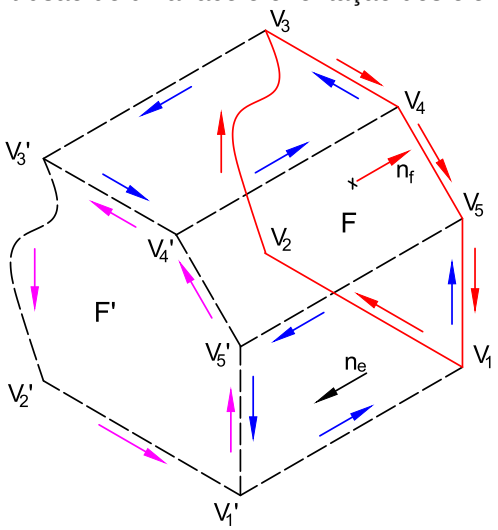
b) Para cada vértice V_k em F, criar um vértice correspondente $V_k{}'$ que formará uma nova face F', mas com orientação e normais invertidas em relação à F;

c) As faces laterais então serão criadas pela sequência: $V_k \to V_k' \to V_{k+1}' \to V_{k+1}$. Se $k+1 > N$, adota-se $k+1 = 0$, fechando o sólido.

Essa instrução lida apenas com os aspectos topológicos, a geometria da face $V_2 \to V_2' \to V_3' \to V_3$ por exemplo, poderia ser representada por NURBS, quádricos ou um conjunto de faces. Se $V_4 \to V_5$ for um arco, por exemplo, a face criada na extrusão poderia ser representada por um cilindro seccionado ou um conjunto de muitas faces

adjacentes que se aproxima de um cilindro.

Figura 2.18 – Extrusão de uma face e orientação dos elementos criados.



Fonte: Elaboração do próprio autor

2.3.3 Varredura rotacional[19]

Também denominada de “revolução” em alguns *softwares*, essa operação forma um sólido por meio da rotação das arestas de uma face em torno de um eixo. Essa operação é muito utilizada para modelagem de eixos com muitos detalhes e quaisquer sólidos de revolução. De maneira geral, a rotação de cada aresta resulta numa superfície de revolução, mas na representação de sólidos unicamente por poliedros, essa superfície é um agrupamento de faces planas. As instruções para essa operação são similares à de extrusão, como pode ser visto abaixo e na figura 2.19:

Instrução 2.3 - Varredura rotacional de uma face

Dados: uma face F composta pela sequência de vértices $V_1, V_2, V_3, \dots, V_N$ e as semi-arestas associadas $a_{1\to2}, a_{2\to3}, \dots, a_{N-1\to N}$, cuja normal é $\vec{n}_F$. O eixo de rotação pode ser representado pelo par de pontos $\boldsymbol{P}_{r1}$ e $\boldsymbol{P}_{r2}$, cujo vetor diretor é $\vec{n}_r$. Por último, deve ser fornecido também o ângulo da varredura ente 0 e 360º. Para modelos poliédricos ainda é necessário fornecer o número de segmentos N_s.

a) Verificar se o eixo de rotação intercepta alguma aresta da face. Se sim, cancelar a operação, senão, prosseguir;

b) Se $\left(\overrightarrow{\boldsymbol{P}_{r1}V_1}\ x\ \vec{n}_r\right).\vec{n}_F < 0$, deve-se inverter a orientação de F ;

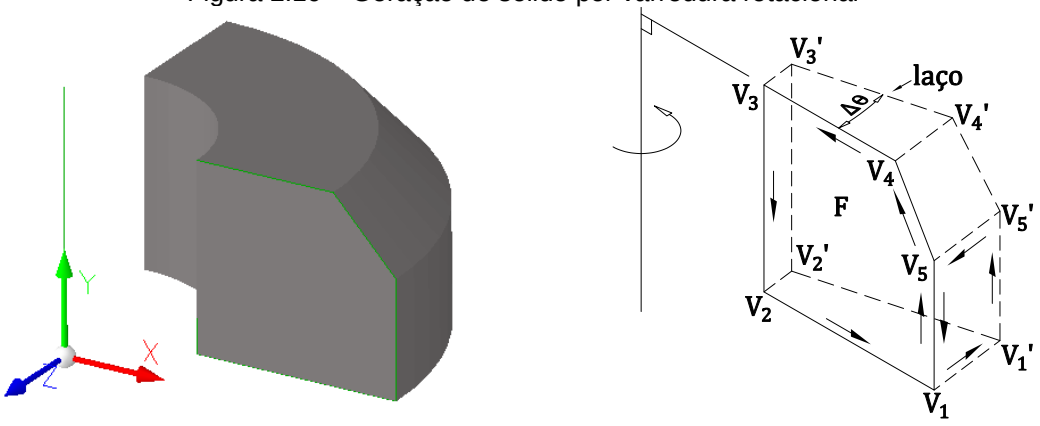
[19] Em inglês – r*evolve*.

c) Para cada vértice V_k em F, criar um vértice correspondente $V_k{}'$ na próxima face que formará um laço l, mas com orientação e normais invertidas em relação à F. A posição de $V_k{}'$ é dada pelo raio em relação ao eixo de revolução e o ângulo θ (ângulo de varredura);

d) As faces laterais então serão criadas pela sequência: $V_k \rightarrow V_k' \rightarrow V_{k+1}' \rightarrow V_{k+1}$. Se $k+1 > N$, adota-se $k+1 = 1$;

e) Repetir os passos b), c) e d) até fazer todos os segmentos. Cada novo laço terá como base o laço anterior. Ao término, se o ângulo de varredura for menor que 360°, cria-se uma face F' baseando-se no último laço formado. Se o ângulo for igual a 360°, exclui-se F e unem-se os vértices do último conjunto de faces com o primeiro;

Figura 2.19 – Geração de sólido por varredura rotacional



Fonte: Elaboração do próprio autor

Essas instruções não contemplam a varredura rotacional de faces com furos. No caso de modelos não poliédricos, as superfícies do sólido seriam formadas conforme a geometria da aresta geradora. Se uma linha, a superfície será um trecho de casca cônica, cilíndrica ou o segmento de um disco. Se um arco, a superfície será um segmento de um toro.

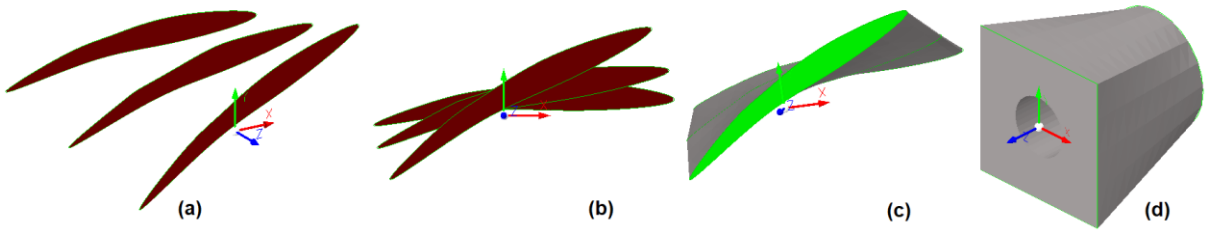
2.3.4 Varredura ordenada (*loft*)[20]

É utilizada para criação de sólidos que apresentem alguma torção (pá de compressor, por exemplo) e/ou transição de forma. Na figura 2.20 em (a) e (b) estão 3 faces[21] ordenadas para a varredura que se diferem pela coordenada z e por rotações

[20] Também conhecida como *general duct* (JANSEN, 1987).
[21] Poderiam ser laços fechados e orientados também. O CABEMT utiliza faces.

em torno do eixo z. Em (c) observa-se o resultado da operação. Em (d) realizou-se um *loft* entre uma face quadrada e outra circular. Os dois sólidos são poliédricos, gerados pelo CABEMT.

Figura 2.20 – Sólidos criados por varredura ordenada



Fonte: Elaboração do próprio autor (CABEMT)

A varredura ordenada para criação de sólidos poliédricos pode ser implementada seguindo a instrução abaixo e observando-se a figura 2.21.

Instrução 2.4 - Varredura Ordenada

Dados: um conjunto de faces planares ($F_1, \dots, F_i$) ou laços ($l_i, \dots, l_{i+1}$) espaçados entre si. O espaçamento não precisa ser uniforme e as faces não precisam estar alinhadas. Pode-se adotar a restrição de que as normais das faces sejam paralelas.

a) Supondo-se que o usuário possa informar as faces em qualquer ordem, primeiramente é preciso verificar se as faces ou laços estão realmente ordenados. Para isso, é necessário determinar a primeira ou a última face da ordenação. Para cada face F_i, construir um vetor saindo de qualquer ponto de F_i até qualquer outro ponto de cada uma das outras faces. Assim, para cada face F_i, haverá um vetor ($V_{F_i \to F_{i+1}}$). Toma-se então o produto escalar do primeiro vetor com cada um dos subsequentes (ou seja, das outras faces). Se ocorrer mudança de sinal do produto escalar, a face F_i não é uma extrema da ordenação[22]. Repetem-se esses passos até que para uma determinada face F_k o produto escalar não mude de sinal quando testada contra todas as outras. Para esclarecimento, notar que na figura 2.21 (a) a face F_1 não é uma extremidade da ordenação, já que $\vec{V}_{12}.\vec{V}_{13} < 0$. O produto final desse passo é uma face extrema. Se o número de faces for de apenas 2, esse passo não é necessário, pode-se tomar qualquer uma das faces;

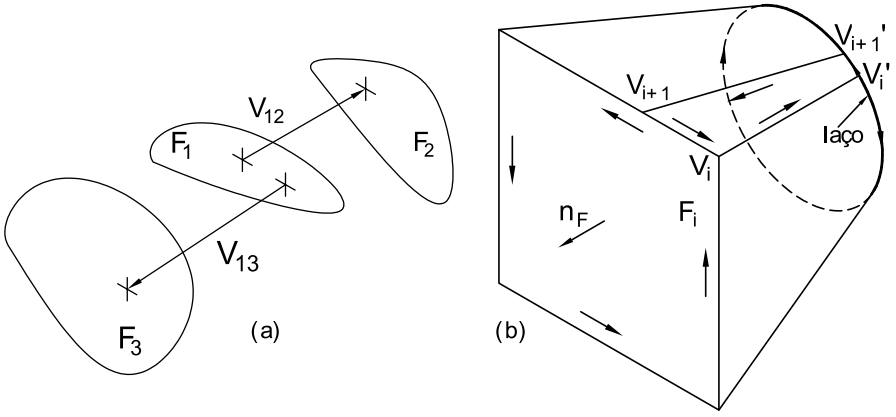
b) Uma vez selecionada a face extrema da ordenação, é preciso organizar as

[22] Não é a primeira face, nem a última da série ordenada.

demais em sequência baseada na distância entre os planos que contém essas faces;

c) Para cada par de faces ou laços subsequentes (F_i, F_{i+1}), verificar se o número de subdivisões total de cada face/laço é idêntico. Se não for, subdividir as arestas dos laços com menor número de segmentos até que todas as faces/laços possuam o mesmo número de subdivisões. Observar que na figura 2.21 (b) as arestas do quadrado precisaram ser segmentadas;

Figura 2.21 – Estrutura topológica de uma varredura ordenada com transição de forma



Fonte: Elaboração do próprio autor

d) Mudar a orientação das arestas das faces de modo que elas fiquem alternadas. Por exemplo: F_1(horário), F_2(anti-horário), F_3(horário)... Ajustar as normais conforme a orientação;

e) Selecionar um vértice V_i qualquer[23] em F_i e determinar o seu correspondente V_i' (o oposto) em F_{i+1}. Poderá ser aquele em que $\left|\boldsymbol{n}_{f_i}\, x\, \overrightarrow{V_i V_i'}\right|$ assumir um valor mínimo;

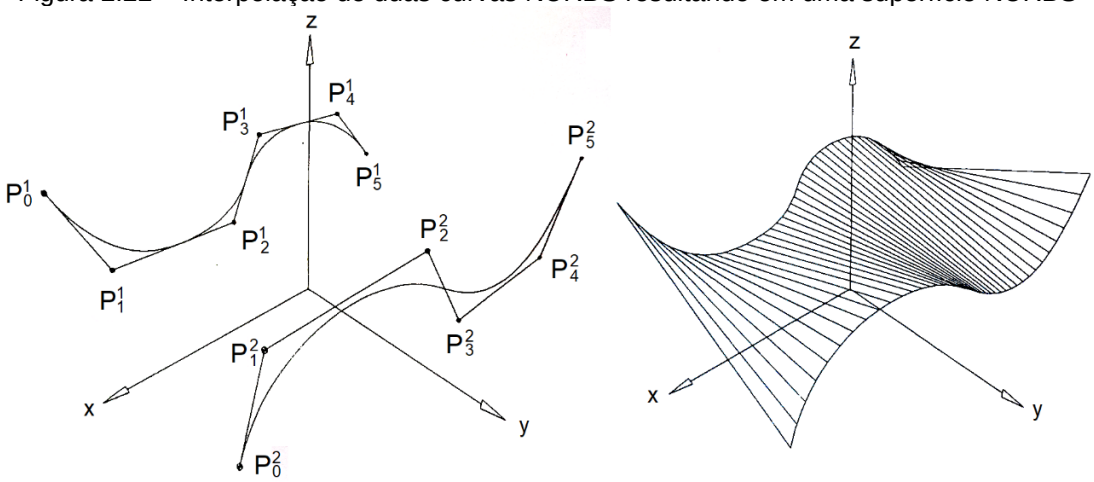
f) Construir as faces laterais para cada par de faces (F e F') seguindo a sequência $V_i \rightarrow V_i' \rightarrow V_{i+1}' \rightarrow V_{i+1} \rightarrow V_i$. Observar que essas arestas que vão da face F_i à face F_{i+1} podem não ser necessariamente retilíneas, mas para fins desse trabalho serão consideradas como tal, ou seja, a interpolação é linear;

Para modelos não poliédricos, os passos para a execução da varredura são similares, mas nesse caso, a criação das faces laterais vai usar diretamente a

[23] Uma melhoria do algoritmo de varredura ordenada poderia procurar vértices cuja vizinhança seja o mais suave possível tanto em F_i, quanto em F_{i+1}.

geometria das arestas ao invés de segmenta-las. Uma forma de se fazer isso é pela utilização de NURBS, conforme pode ser verificado em Piegl e Tiller (1997, p. 337-340). Nesse caso, toma-se duas curvas NURBS ($\boldsymbol{C_1}(\bar{t}) e\, \boldsymbol{C_2}(\bar{t})$) e cria-se uma superfície obtida pela interpolação linear entre elas. Um exemplo dessa técnica é ilustrado na figura abaixo para um par de curvas.

Figura 2.22 – Interpolação de duas curvas NURBS resultando em uma superfície NURBS

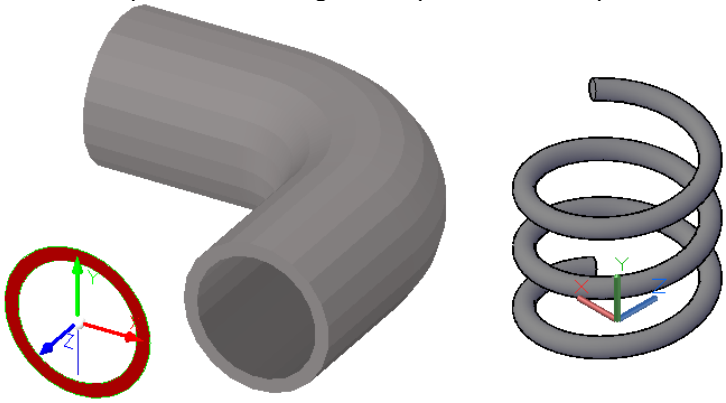


Fonte: Piegl e Tiller (PIEGL; TILLER, 1997)

2.3.5 Varredura por caminho

Essa operação é mais conhecida simplesmente como varredura (*sweep*) e consiste na geração de um sólido a partir da extrusão de uma face ou laço fechados ao longo de um caminho geralmente definido por linhas e curvas. Esse tipo de varredura é muito útil na criação de tubulações e helicoides em geral, como mostrado na figura a seguir:

Figura 2.23 – Exemplos de sólidos gerados pela varredura por caminho (*sweep*)

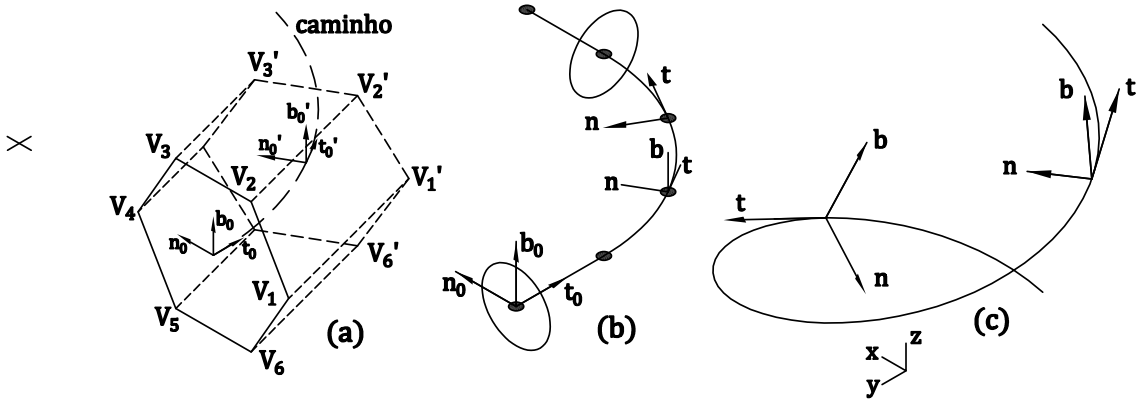


Fonte: Elaboração do próprio autor

Na modelagem de sólidos por poliedros, o caminho é segmentado nas curvas e a face é posicionada em cada extremidade dos segmentos que compõem o

caminho. O plano da face deve ficar perpendicular à tangente da curva, tal como exibido na figura 2.24. Quando o caminho da varredura não está contido em um plano, a varredura deve levar em conta a torção, rotacionando a face em cada segmento de maneira a alinhar o vetor bi-normal ($\boldsymbol{b}$) do ponto antecessor com o sucessor no caminho. Nesse caso, se o vetor $\boldsymbol{b}_i$ passa por um ponto $\boldsymbol{P}_i$ em F_i – quando em F_{i+1} , o vetor $\boldsymbol{b}_{i+1}$ deve coincidir com $\boldsymbol{P}_i$ na face F_{i+1}.

Figura 2.24 – Exemplo de caminhos para varredura e os eixos de coordenadas locais



Fonte: Elaboração do próprio autor

As seguintes relações do cálculo são úteis para se determinar a tríade ($\boldsymbol{t}, \boldsymbol{n}, \boldsymbol{b}$), assumindo-se que a curva que descreve o caminho pode ser parametrizada em u (STEWART, 2006):

$$\boldsymbol{t}(u) = \frac{\mathbf{r}'(u)}{|\mathbf{r}'(u)|}; \qquad \vec{n}(u) = \frac{\boldsymbol{t}'(u)}{|\boldsymbol{t}'(u)|}; \qquad \mathbf{B}(u) = \mathbf{t}(u) \times \mathbf{n}(u) \tag{2.17}$$

No caso da curva ser representada de maneira discreta, ou seja, por um conjunto de linhas, pode-se fazer uma aproximação numérica da equação (2.17).

A criação de um sólido poliédrico por varredura pode ser feita por meio da instrução abaixo:

Instrução 2.5 - Varredura por caminho

Dados: uma face F ou um laço para a varredura contidos no plano $\boldsymbol{nb}$ da tríade ($\boldsymbol{t_0}, \boldsymbol{n_0}, \boldsymbol{b_0}$) e um caminho (fechado ou não) que pode ser definido por uma equação paramétrica ou um agrupamento de linhas.

a) Considerando que dois pontos iniciais do caminho sejam $\mathbf{P_1}$ e $\boldsymbol{P_2}$ e a normal de F seja $\boldsymbol{n}_f$, então se $\boldsymbol{n}_f . \overrightarrow{\boldsymbol{P_1 P_2}} > 0$, inverter a orientação de F, caso contrário, pular para o passo b);

b) A partir da face F contida no plano $\boldsymbol{nb}$ de $(\boldsymbol{t_0}, \boldsymbol{n_0}, \boldsymbol{b_0})$ criar outra idêntica (F') porém no sistema $(\mathbf{t}_0', \boldsymbol{n}_0', \boldsymbol{b}_0')$ que fica em um ponto subsequente do caminho – vide figura 2.24;

c) Construir as faces laterais de maneira análoga à extrusão (ver instrução 2.2);

d) Designar F' como F e repetir os passos b) e c) até que todo o caminho seja percorrido;

e) Se o caminho for aberto, a primeira e última face farão parte do sólido, caso contrário, apagá-las e unir as extremidades;

f) Apagar quaisquer faces intermediárias F'.

No caso de modelos não-poliédricos, a varredura por caminho pode ser realizada com NURBS, tanto na definição da face ou laço a varrer, como no caminho. Detalhes dessa técnica podem ser encontrados no livro de Piegl e Tiller (1997, p. 472-485).

2.4 OPERAÇÕES BOOLEANAS

2.4.1 Introdução

Embora as técnicas descritas no item 2.3 tenham uma capacidade razoável de modelagem, em muitos casos somente a combinação de dois ou mais sólidos pode representar um objeto adequadamente. Imagine que o perfil I mostrado na figura 2.17 tivesse um furo circular transversal à sua alma. As operações de varredura ou geração direta não conseguiriam fazer a modelagem. No entanto, podemos interpretar a peça final como um perfil I subtraído de um cilindro transversal à alma com diâmetro igual ao do furo. Fez-se, portanto, uma operação booleana (OB) de subtração[24] entre dois sólidos. Há também as operações de adição e intersecção e desta última derivam-se as primeiras.

Além das limitações da geração direta de sólidos ou por varredura, há de se considerar que as operações booleanas oferecem uma maneira de subdividir um problema complexo em um conjunto de operações em componentes mais simples, fazendo com que todo o processo de geração do sólido fique mais inteligível ao

[24] Os termos subtração e adição não são operações booleanas de fato, mas possuem um sentido bastante intuitivo na modelagem de sólidos. No fim das contas, a “adição” e “subtração” de sólidos são resultado de uma ou mais operações booleanas (AND, OR, NOT).

usuário do sistema. Fora do contexto da modelagem, essas funções também podem ser utilizadas para detecção de colisão em máquinas CNC (MÄNTYLÄ, 1988).

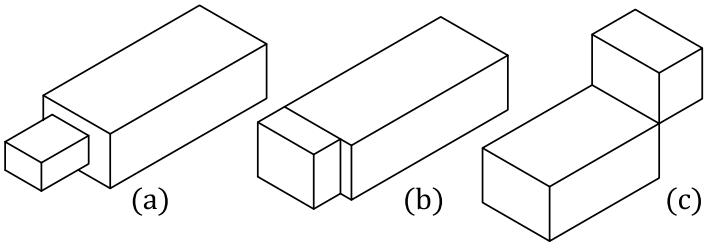
Uma vez que na CGS a modelagem é feita sempre na forma de combinação entre vários sólidos, as operações booleanas são algo intrínseco à técnica e correspondem basicamente a operações simbólicas. A dificuldade concentra-se no momento de gerar a visualização do sólido, conforme descrito no item 2.2.2 e cujas técnicas são amplamente discutidas por Jansen (1987).

De acordo com Mäntylä (1988), as operações booleanas na representação por fronteira são os algoritmos tecnicamente mais complexos. De fato, o conjunto de algoritmos responsáveis por tais operações foram provavelmente os mais difíceis de se implementar no CABEMT devido a principalmente três fatores:

a) Complexidade lógica: por mais que se reparta o algoritmo em funções/métodos menores e simples, o funcionamento se dá mediante a interação de todos esses trechos de código. Erros podem ocorrer em algum método e se propagar para as outras funções sem causar parada do programa, dificultando sua localização. Enfim, a complexidade pode emergir das operações mais simples, basta que o resultado final dependa da sua interação com outras funções e que estas ocorram em número elevado;

b) Variáveis de entrada: os operandos podem estar em orientações desfavoráveis que geram casos específicos de intersecção e devem ser classificados e tratados separadamente. Para todo caso especial existirá provavelmente um algoritmo de tamanho e complexidade razoável para elaboração. Alguns casos especiais de alinhamento encontram-se na figura 2.25. Em (a) o algoritmo deverá detectar duas faces coplanares e se há intersecção ou não entre elas, assim como avaliar se uma está contida totalmente no interior de outra. Em (b), além de fazer as verificações de (a), será necessário identificar arestas que se tocam e uni-las de acordo. Em (c) a operação de adição resultará em um sólido *non-manifold*, isto é, que não é uma variedade em $\mathbb{R}^3$. Alguns programas não aceitam esse tipo de sólido. O CABEMT aceita estruturas não orientáveis nos modelos de trincas;

Figura 2.25 – Casos que requerem tratamento especial durante operações booleanas



Fonte: Elaboração do próprio autor

c) Erros numéricos: no decorrer do processamento das operações booleanas é necessário calcular curvas e pontos de intersecção entre superfícies, classificar pontos quanto à sua posição relativa a uma superfície ou sólido e detectar casos especiais como em a) e b). Todas essas funções estão sujeitas a erros numéricos, visto que variáveis para armazenamento das características geométricas são normalmente de ponto flutuante. Hoffmann (1992) cita três erros típicos da representação por ponto flutuante:

- Erro de conversão: dados de entrada são geralmente em formato decimal que por vezes se tornam dízimas em sistema binário;
- Erro de truncamento: a precisão é limitada pela quantidade de memória reservada ao tipo de variável[25];
- Erro de cancelamento de dígito: ocorre quando a diferença entre dois números próximos é tão pequena que dígitos significativos do resultado possam ser truncados.

Métodos que buscam detectar coincidências de entidades geométricas, tais como as arestas na Figura 2.25 (b), são especialmente vulneráveis a erros numéricos, de tal modo que nesses algoritmos é costumeiro adotar tolerâncias para a classificação dos elementos geométricos – porém sob risco de classificá-los erroneamente.

A seguir serão discutidas em detalhes cada uma das OB fundamentais, começando pela intersecção, das quais as outras se derivam. Será adotado o enfoque da representação por fronteiras e assumido que todos os sólidos sejam poliédricos, assim como no CABEMT. Essa última consideração não simplifica muito o problema

[25] Em muitas linguagens de programação a precisão simples consiste numa variável do tipo *float* de 4 bytes e a precisão dupla é denominada *double*, com 8 bytes.

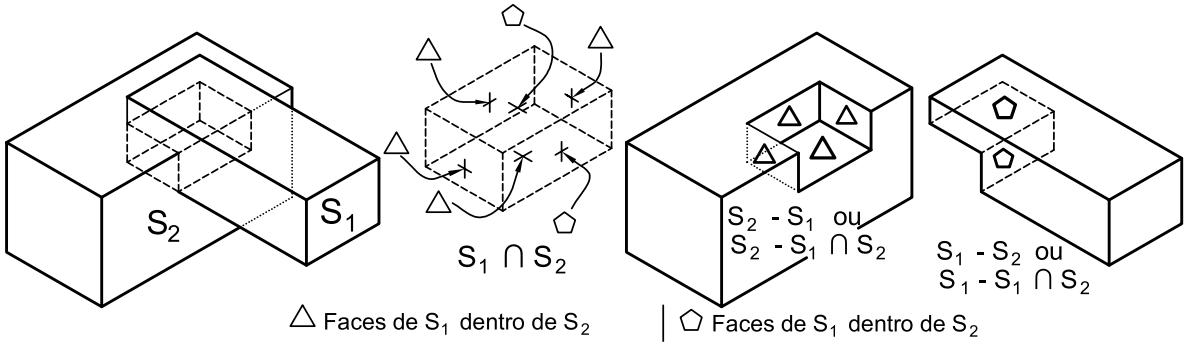
do ponto de vista topológico, uma vez que as OB estão bastante relacionadas à topologia e menos à geometria. De fato, nas obras de Mäntylä (1988) e Hoffmann (1992) as operações booleanas são apresentadas para poliedros e somente depois são sugeridas extensões dos algoritmos para se levar em conta superfícies curvas.

Não obstante, o uso direto de superfícies curvas pode ser evitado aproximando-as por faces triangulares por exemplo, caso seja conveniente. Até mesmo as NURBS podem ser aproximadas por superfícies poligonais por meio do refinamento de *knots* (PIEGL; TILLER, 1997, p. 162-179), caso seja desejável evitar cálculos de intersecção complicados entre esses elementos.

2.4.2 Intersecção de dois sólidos (AND)

Observando a figura abaixo é possível notar duas características essenciais da intersecção entre dois sólidos ($\boldsymbol{S}_1$ e $\boldsymbol{S}_2$) que também se aplicam à adição e subtração. A primeira é que as OB não criam regiões novas, ou seja, todo o conjunto de pontos que compõem o sólido $\boldsymbol{S}_1 \cap \boldsymbol{S}_2$ pertencia a $\boldsymbol{S}_1$ ou a $\boldsymbol{S}_2$, ou ambos. A segunda característica é que as faces de $\boldsymbol{S}_1 \cap \boldsymbol{S}_2$ podem ser recortes das faces de $\boldsymbol{S}_1$ ou $\boldsymbol{S}_2$ ou alguma face inteira deles. Os cortes são sempre feitos nas superfícies, o interior do sólido não importa, a não ser que possua algum vazio ou furo.

Figura 2.26 – Operações booleanas entre dois sólidos



Fonte: Elaboração do próprio autor

Sabe-se então que as faces constituintes de $\boldsymbol{S}_1 \cap \boldsymbol{S}_2$ são trechos (ou a face inteira se for o caso) das faces de $\boldsymbol{S}_1$ que estão dentro de $\boldsymbol{S}_2$[26] e vice-versa. Com base nessas observações, pode-se implementar a instrução a seguir para intersecção de dois sólidos poliédricos. Os detalhes da implementação serão vistos no capítulo 4.

[26] Uma face está totalmente no interior de um sólido se todos os seus vértices assim estiverem.

Instrução 2.6 - intersecção entre dois sólidos

Dados: sólidos $\boldsymbol{S}_1$ e $\boldsymbol{S}_2$ (operandos) representados por fronteiras (B-rep).

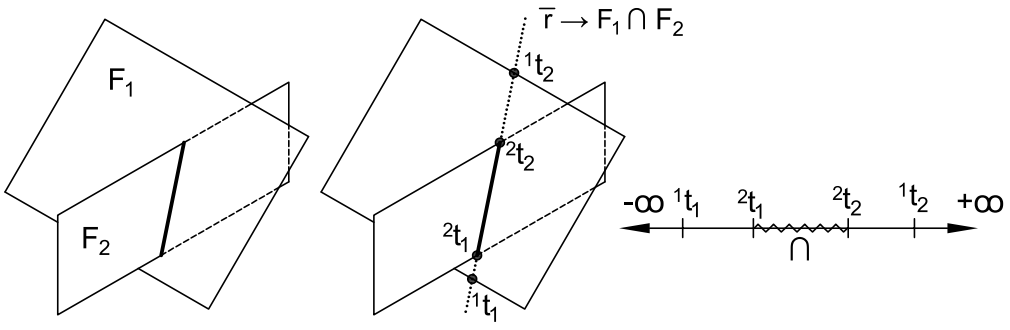
a) Iterar por todos os vértices de $\boldsymbol{S}_1$ e $\boldsymbol{S}_2$, identificando aqueles de $\boldsymbol{S}_1$ que estão dentro de $\boldsymbol{S}_2$ e vice-versa. Esses vértices estão marcados com um quadrado na figura 2.28 b) e c). Se para uma determinada face todos os seus vértices estiverem dentro do sólido oposto, então essa face já deve ser automaticamente armazenada como uma face do sólido de intersecção que está para se formar;

b) Sendo ${}^1F_k$ a k-ésima de ${}^1N$ faces de $\boldsymbol{S}_1$e ${}^2F_j$ a j-ésima de ${}^2N$ faces de $\boldsymbol{S}_2$, determinar e armazenar as curvas geradas por ${}^1F_k \cap {}^2F_j$ $(k = 1, \dots, {}^1N; j = 1, \dots, {}^2N$. No caso de os operandos serem poliédricos, as "curvas" serão sempre retas e o problema de intersecção entre as faces resume-se à intersecção entre dois planos para cada par de faces. Vide APÊNDICE D sobre como determinar a equação da reta de intersecção;

c) Para cada reta $\overline{r_n}$ gerada em b), estarão associadas as duas faces (${}^1F_k$ e ${}^2F_j$) que a formaram, denominadas faces geradoras. É conveniente que essas retas estejam na forma paramétrica conforme a equação (2.1);

d) O próximo passo é transformar as retas em segmentos, bastando que sejam atribuídos um parâmetro inicial ${}^F t_i$ e um final ${}^F t_f$ para cada face e que devem ser determinados para que o segmento formado esteja simultaneamente dentro de ambas as faces geradoras, como pode ser observado na figura 2.27. Para tal, basta organizar os parâmetros em ordem crescente e tomar os centrais. Os parâmetros de intersecção são dados pelas das equações do APÊNDICE E;

Figura 2.27 – Segmentação de uma reta de intersecção entre duas faces



Fonte: Elaboração do próprio autor

e) Efetuar a intersecção dos segmentos de retas obtidos no passo anterior com cada uma das arestas das faces. Dando origem a novos vértices – denominados divisores – marcados com um círculo na figura 2.28. Observar que as arestas antigas de $\boldsymbol{S}_1$ e $\boldsymbol{S}_2$ devem ser subdivididas pelos novos vértices. Por exemplo, na figura 2.28, a aresta ($V_{11} \rightarrow V_4$) torna-se ($V_{11} \rightarrow V_2 \rightarrow V_4$). Pode acontecer de uma face ser dividida em duas ou mais. Quando dois novos vértices forem coincidentes a menos de uma certa tolerância, eles devem ser unidos. Essa união não é meramente geométrica, mas também topológica – ou seja, as entidades que em separado conectavam-se a cada um dos vértices, agora devem conectar ao vértice gerado na união;

f) Das faces que sobraram, somente aquelas que possuem pelo menos um vértice dentro do sólido oposto[27] (vértice interior) ou um vértice divisor são elegíveis para fazer parte de $\boldsymbol{S}_1 \cap \boldsymbol{S}_2$;

g) Para cada uma das faces dos sólidos (${}^{i}F_k$), selecionar um de seus vértices, desde que ele seja interior ou divisor. Armazenar também as faces (ou os seus identificadores) de $\boldsymbol{S}_1$ e $\boldsymbol{S}_2$ que o originaram – faces geradoras. Aqui se dá o início de um laço l_n;

h) Ainda processando a face ${}^{i}F_k$, a partir do vértice selecionado anteriormente (V_i), determinar o próximo (V_{i+1}) que dará origem à primeira aresta ($V_i \rightarrow V_{i+1}$). Se este nó for um divisor, ${}^{i}F_k$ deverá obrigatoriamente ser uma de suas faces geradoras. Mas caso ele seja um vértice interior, basta que ele se conecte diretamente a V_i e que pertença à face sendo processada, isto é, à ${}^{i}F_k$. Por exemplo, na figura 2.28, supondo-se que a face ${}^{2}F_1$ esteja sendo processada, podemos selecionar o vértice divisor V_3, produto das seguintes intersecções: ${}^{2}F_1 \cap {}^{1}F_3$ e ${}^{2}F_1 \cap {}^{1}F_4$. V_3 conecta-se diretamente ao vértice interior V_4. Todavia V_4 não faz parte de ${}^{2}F_1$ e por conseguinte não é sequência de V_3 quando se está processando ${}^{2}F_1$. Mas V_3 também se conecta a V_1 e V_6 de tal modo que qualquer um deles pode ser o próximo vértice pois ambos satisfazem a condição de serem divisores e possuírem a face ${}^{2}F_1$ como uma de suas geradoras. Para efeitos desse exemplo, consideremos que o vértice V_6

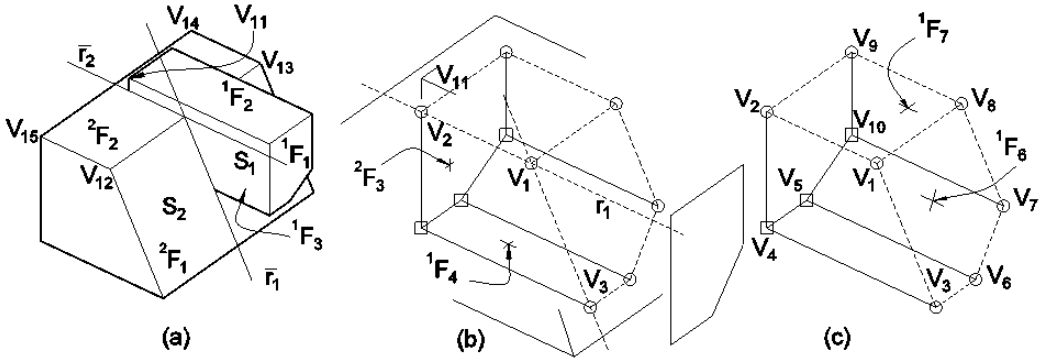
[27] Um vértice de uma face de S_1 dentro do sólido S_2.

foi escolhido. Então o laço l até agora possui V_3 e V_6;

i) A escolha dos vértices posteriores é análoga ao passo h), mas vértices já utilizados no laço corrente (l_n) não podem mais ser utilizados, caso contrário o algoritmo poderia entrar em recursão infinita. Continuando o exemplo, observa-se que V_6 se conecta a V_5 e V_7. O vértice V_5 não pode ser o próximo porque suas faces geradoras são ${}^1F_4$, ${}^1F_6$ e ${}^2F_3$, ou seja, ele não tem a face corrente ${}^2F_1$ como uma de suas geradoras. V_7, por outro lado, é um vértice do tipo divisor e uma de suas faces geradoras é ${}^2F_1$, portanto é o próximo da sequência. O laço l_n agora fica $V_3 \rightarrow V_6 \rightarrow V_7$;

j) Seguindo a instrução, continua-se acrescentando vértices ao laço e buscando os candidatos para o próximo vértice até que um desses candidatos seja o primeiro vértice do laço. Nesse caso, o laço estará completo. É claro que para essa condição ser atendida o número corrente de vértices do laço deve ser de pelo menos dois;

Figura 2.28 – Retas de intersecção e vértices entre as faces de dos sólidos. Os números das faces são todos relativos à ilustração (a), mesmo estando em (b) ou (c).



Fonte: Elaboração do próprio autor

k) Uma vez fechado o laço, é necessário verificar se a orientação de suas arestas aponta para o sentido da normal da face que o gerou. Se apontarem para sentidos opostos, a orientação do laço deve ser invertida. Após isso, o laço pode ser imediatamente transformado em uma face;

l) Repetem-se os passos g) a k) para cada face de $\boldsymbol{S}_1$ e $\boldsymbol{S}_2$;

m) Ao final, unem-se todas as faces criadas em l) em um novo sólido que nesse caso será o resultado da intersecção de $\boldsymbol{S}_1$ e $\boldsymbol{S}_2$. Os dois operandos são então excluídos para completar a operação.

2.4.3 Subtração de dois sólidos

O algoritmo de subtração entre dois sólidos toma emprestado a maior parte do código de intersecção. De fato, o resultado de $\boldsymbol{S}_2 - \boldsymbol{S}_1$ será um sólido $\boldsymbol{S}_3$ que:

- Contém todas as faces de $\boldsymbol{S}_2$ que não se interceptam com nenhuma face de $\boldsymbol{S}_1$e que também não estejam integralmente dentro de $\boldsymbol{S}_1$;
- Contém todas as faces de $\boldsymbol{S}_2$ que se interceptam com alguma face de $\boldsymbol{S}_1$. Mas nesse caso, as faces devem ser recortadas. Se tomarmos como exemplo a figura 2.28, a face ${}^2F_2$ deve ser recortada pela face formada por $V_1 \rightarrow V_8 \rightarrow V_9 \rightarrow V_2$, ao passo que ${}^2F_1$ deve ser recortada pela face formada por $V_3 \rightarrow V_6 \rightarrow V_7 \rightarrow V_8 \rightarrow V_1$;
- Contém todas as faces do sólido $\boldsymbol{S}_1$ (recortadas ou não) que fazem parte de $\boldsymbol{S}_1 \cap \boldsymbol{S}_2$.

Para atender a esses requisitos, as etapas a seguir podem ser utilizadas:

Instrução 2.7 - subtração entre dois sólidos

Dados: dois sólidos $\boldsymbol{S}_1$ e $\boldsymbol{S}_2$, um deles será o minuendo e o outro, subtraendo. Para efeito de exemplificação, assume-se a criação do sólido $\boldsymbol{S}_3 = \boldsymbol{S}_2 - \boldsymbol{S}_1$.

a) Executar os passos a) a l) da Instrução 2.6;

b) Para cada face ${}^2F_k$ de $\boldsymbol{S}_2$ verificar se ela possui intersecção com $\boldsymbol{S}_1$. Se não possuir, essa face ficará em $\boldsymbol{S}_3$. Tal verificação pode ser feita percorrendo as arestas de cada face ${}^kF_2$. Se alguma aresta contiver um vértice divisor, então a face sofreu uma intersecção com $\boldsymbol{S}_1$ e dessa forma não pode ser levada diretamente para $\boldsymbol{S}_3$;

c) As faces de $\boldsymbol{S}_2$ que possuem alguma intersecção com $\boldsymbol{S}_1$ precisam antes ser recortadas pelas faces do sólido de intersecção. Voltando à figura 2.28, notamos que a face ${}^2F_2$ possui intersecção com $\boldsymbol{S}_1$ e, portanto, deve ser recortada pela face ($V_1 \rightarrow V_8 \rightarrow V_9 \rightarrow V_2$). Inicia-se em qualquer vértice de ${}^2F_2$ e vai armazenando os vértices pela conectividade das arestas até que o vértice atual e o próximo sejam novos, ou seja, foram obtidos por meio da segmentação das arestas de ${}^2F_2$ durante o algoritmo de intersecção. Esse par de vértice forma uma aresta proibida e outro caminho deve ser encontrado. O

próximo vértice não estará em ${}^{2}F_2$, mas será algum dos vértices de $\boldsymbol{S}_1 \cap \boldsymbol{S}_2$ que tiverem sido gerados através da intersecção de ${}^{2}F_2$ com faces de $\boldsymbol{S}_1$ (${}^{1}F_k$) e que a ele estejam ligados. Utilizando a figura 2.28 como exemplo, escolhemos qualquer vértice de ${}^{2}F_2$, digamos o V_{15}. Então os próximos da sequência são V_{12}, V_1, V_8, etc. Mas $V_1 \rightarrow V_8$ é uma aresta proibida e a sequência deve ser encontrada em $\boldsymbol{S}_1 \cap \boldsymbol{S}_2$. No sólido de intersecção, os vértices V_1, V_8, V_9, V_2 provêm da intersecção de ${}^{2}F_2$ com faces de $\boldsymbol{S}_1$, por conseguinte, são candidatos a sequenciar $V_{15} \rightarrow V_{12} \rightarrow V_1$. V_8 é um candidato porque se conecta a V_1, mas formaria com este uma aresta proibida, logo, a continuação se dará por V_2;

d) Após eleito um vértice de $\boldsymbol{S}_1 \cap \boldsymbol{S}_2$ para o sequenciamento, os vértices seguintes serão aqueles da face de $\boldsymbol{S}_1$ mencionada no passo anterior até que se chegue ao vértice rejeitado no passo c). Pela figura 2.28, teremos a sequência $V_{15} \rightarrow V_{12} \rightarrow V_1 \rightarrow V_2 \rightarrow V_9 \rightarrow V_8$;

e) Uma vez alcançado o vértice rejeitado, é momento de se retornar à face ${}^{k}F_2$, e determinando os vértices subsequentes pela topologia. Voltando ao exemplo, na face ${}^{2}F_2$, V_8 conecta-se a V_1 e a V_{13}. Como $V_1 \leftrightarrow V_8$ é uma aresta proibida, a única opção é seguir por V_{13};

f) Continuar seguindo os vértices de ${}^{k}F_2$ até que o próximo seja o primeiro da face/laço que está sendo criado ou até que se encontre uma aresta proibida, nesse caso, repetem-se os passos c), d) e e) até fechar a face e então transferi-la para $\boldsymbol{S}_3$;

g) Iterar pelas faces de $\boldsymbol{S}_1 \cap \boldsymbol{S}_2$. Excluir aquelas nas quais todos os seus vértices são divisores. No exemplo, as faces ${}^{1}F_6$ e ${}^{1}F_7$ seriam excluídas;

h) Adicionar as faces de $\boldsymbol{S}_1 \cap \boldsymbol{S}_2$ que sobraram do passo anterior à $\boldsymbol{S}_3$;

i) Checar a orientação de todas as arestas de todas as faces e finalizar.

2.4.4 Adição de dois sólidos

Uma das formas de se efetuar a adição ou união de sólidos é usar as operações de intersecção e subtração por meio das etapas a seguir:

Instrução 2.8 - adição de sólidos

Dados: dois sólidos (operandos) – $\boldsymbol{S}_1$ e $\boldsymbol{S}_2$.

a) Efetuar $\boldsymbol{S}_2 - \boldsymbol{S}_1$;

b) Efetuar $\boldsymbol{S}_1 - \boldsymbol{S}_2$;

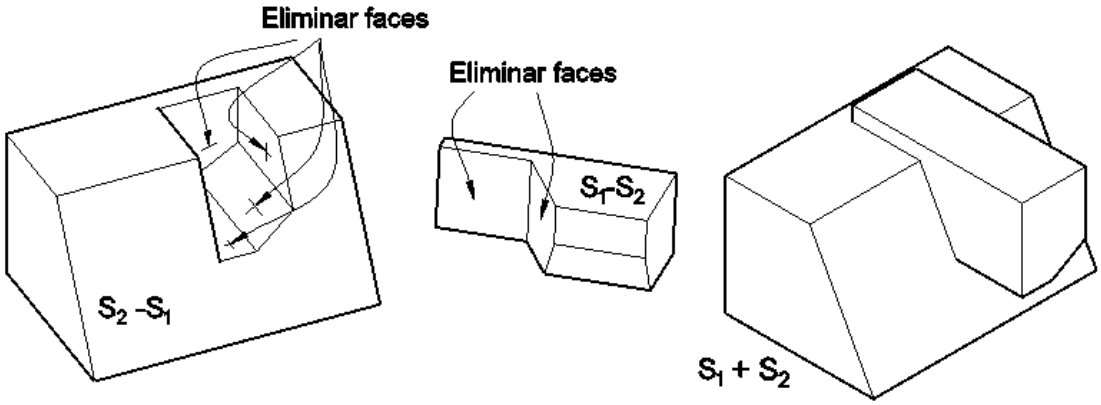
c) Em $\boldsymbol{S}_2 - \boldsymbol{S}_1$ excluir todas as faces que tenham todos os seus vértices iguais a uma face de $\boldsymbol{S}_1 \cap \boldsymbol{S}_2$;

d) Em $\boldsymbol{S}_1 - \boldsymbol{S}_2$ excluir todas as faces que tenham todos os seus vértices iguais a uma face de $\boldsymbol{S}_1 \cap \boldsymbol{S}_2$;

e) Reunir as faces dos passos c) e d) que formarão $\boldsymbol{S}_1 + \boldsymbol{S}_2$.

A figura abaixo ilustra essas etapas; o sólido intermediário foi girado para melhor visualização.

Figura 2.29 – Etapas da adição de dois sólidos



Fonte: Elaboração do próprio autor

3 GERAÇÃO DE MALHAS

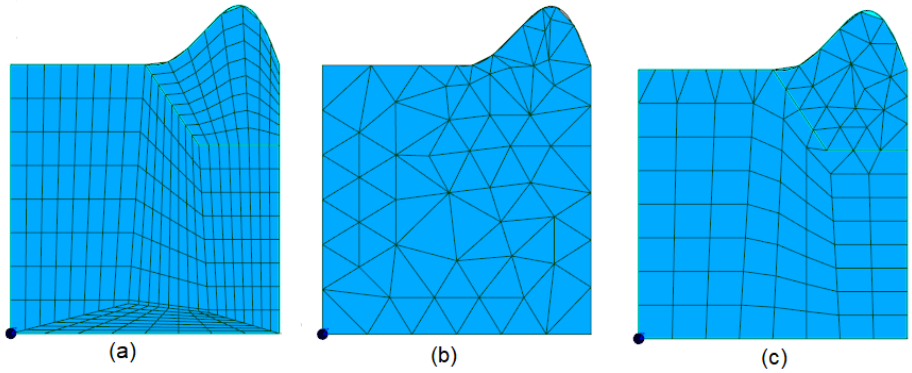
3.1 INTRODUÇÃO

A maioria dos métodos numéricos requerem uma subdivisão do objeto em elementos menores e de geometria simples, processo denominado discretização do domínio e que resulta em um conjunto de elementos chamado de malha. No MEF tridimensional por exemplo, os sólidos podem ser subdivididos em tetraedros, hexaedros e pirâmides. Ao passo que em duas dimensões as formas comuns são a quadrangular e triangular, sendo também as utilizadas no MEC tridimensional, uma vez que nessa técnica a dimensionalidade do problema é reduzida. Por tratar os sólidos sempre como poliedros, o CABEMT gera malhas com elementos bidimensionais planares.

As malhas podem ser classificadas em três principais grupos:

- Estruturadas: todos os seus nós internos possuem um número igual de elementos vizinhos. Esse tipo de malha tem melhor aspecto visual e os nós, em geral, encontram-se bem distribuídos, apesar de não ser um requisito obrigatório. Malhas desse tipo são tipicamente formadas por uma face com quatro lados, ainda que seja necessário concatenar algumas arestas para tal. Esse tipo de malha é normalmente utilizado quando se é possível fazer uma grade com nós regularmente distribuídos, algo possível em geometria mais simples. É comum subdividir regiões mais complicadas em outras mais simples de modo que se possa empregar esse tipo de malha;
- Não-estruturadas: seus nós internos possuem um número variável de elementos vizinhos. São normalmente utilizadas para discretização de geometrias mais complexas, onde existe uma grade não regular de nós;
- Híbridas: quando parte do domínio é constituído por malha estruturada e outra parte não. Isso é feito normalmente quando uma região específica possui uma complexidade geométrica que dificulta ou impede a criação de malha estruturada.

Figura 3.1 – Tipos de malha: a) estruturada, b) não-estruturada e c) híbrida



Fonte: Elaboração do próprio autor

A figura 3.1 ilustra esses principais tipos de malha bidimensionais. Nesse caso, a geometria apresenta um ressalto onde ainda foi possível a criação de malha estruturada, porém nem sempre isso é possível. Malhas estruturadas podem requerer menos memória para armazenamento, fornecem soluções mais precisas e oferecem maior velocidade para acessar seus elementos, por outro lado, pode ser difícil ou impossível aplicá-la em domínios com geometria desfavorável (BERN; PLASSMANN, 2000). As malhas não-estruturadas possuem as vantagens de serem mais flexíveis ao se ajustar em domínios complexos, serem capazes de promover uma rápida mudança de tamanho dos elementos e facilidade no refinamento (BERN; PLASSMANN, 2000). Esse tipo de malha é normalmente associado a elementos triangulares. Esses têm a característica de se adaptar mais facilmente a geometrias difíceis, mas também podem ser encontrados em malhas regulares.

3.2 QUALIDADE DA MALHA

Idealmente, os elementos utilizados em qualquer método numérico, devem possuir a menor distorção possível, ou seja, seus ângulos internos e tamanhos das arestas devem assumir valores próximos entre si. Para se fazer uma avaliação quantitativa dessa distorção, a métrica mais comum é a razão de aspecto. Sua definição depende do tipo de elemento. Para elementos quadrangulares, é a razão entre a dimensão mais longa e a mais curta do elemento. Para triângulos, é o comprimento da maior aresta dividido pela altura do triângulo tendo essa aresta como base (EDELSBRUNNER, 2000). Em muitos casos, à medida que a razão de aspecto aumenta, a precisão dos resultados diminui (LOGAN, 2007). Mas há situações em que mesmo razões de aspecto elevadas ainda levam a resultados satisfatórios, como demonstrado por Rice (1985). Isso costuma acontecer se os elementos distorcidos

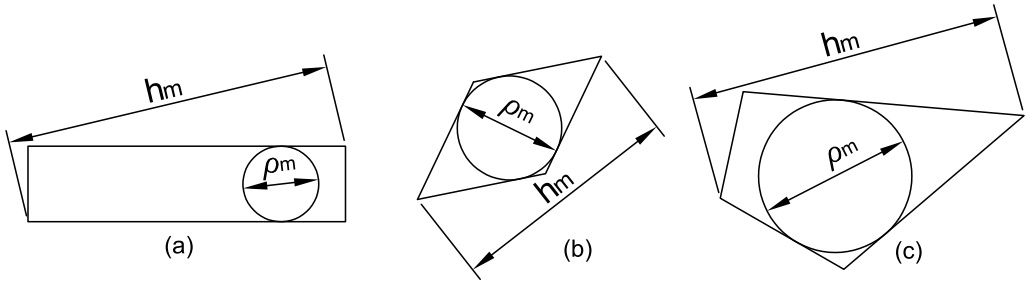
ficarem em áreas de pequeno gradiente de tensões (LOGAN, 2007).

Bathe (2014) usa outro parâmetro denominado σ_m para verificar a regularidade da malha, definido como:

$$\sigma_m = \frac{h_m}{\rho_m} \tag{3.1}$$

em que h_m é a maior dimensão do elemento e ρ_m é o diâmetro do maior círculo que pode ser inscrito no elemento, como ilustrado na figura 3.2. Em um triângulo, ρ_m pode ser calculado facilmente pelo quociente da área pelo semi-perímetro do triângulo (BRONSHTEIN *et al*., 2015).

Figura 3.2 – Parâmetros para cálculo de distorção: a) distorção da razão de aspecto; b) distorção em paralelogramo; c) distorção angular



Fonte: Adaptado de Bathe (2014)

Outras métricas para averiguar a qualidade da malha são os ângulos dos elementos, de tal forma que os algoritmos de otimização procuram (BERN e PLASSMANN, 2000):

- Maximizar os menores ângulos (*maxmin*);
- Minimizar os maiores ângulos (*minmax*).

Há ainda outras métricas, conforme mostram Pébay e Baker (2003). No entanto todas tem por objetivo estimar a distorção da malha e em geral os elementos “bons” apresentam um valor da métrica próximo à unidade.

3.3 CONSTRUÇÃO DA MALHA

Para a construção da malha são necessárias pelo menos duas etapas fundamentais: o posicionamento de nós e a formação dos elementos. O posicionamento precisa distribuir os nós nas fronteiras e no interior do domínio de tal modo que a geometria original seja bem representada após a malha ser construída. Pode ser necessária a variação de tamanhos dos elementos nas regiões de interesse da simulação ou que apresentem variações mais bruscas de alguma propriedade

geométrica, tais como a curvatura ou a normal da borda/superfície.

Estando os nós posicionados, é preciso conectá-los entre si para que formem os elementos da malha. Essa conexão deve ser feita de tal modo a minimizar a distorção e a razão de aspecto dos elementos, pois caso contrário, a solução do método numérico pode ficar comprometida.

O posicionamento de nós em geral pode ser feito via inserção de uma grade divisória, pela expansão progressiva das fronteiras do domínio ou por decomposição espacial. A conexão dos nós, especialmente em malhas não estruturadas, é feita por triangulação de Delaunay ou diretamente através das células da decomposição espacial.

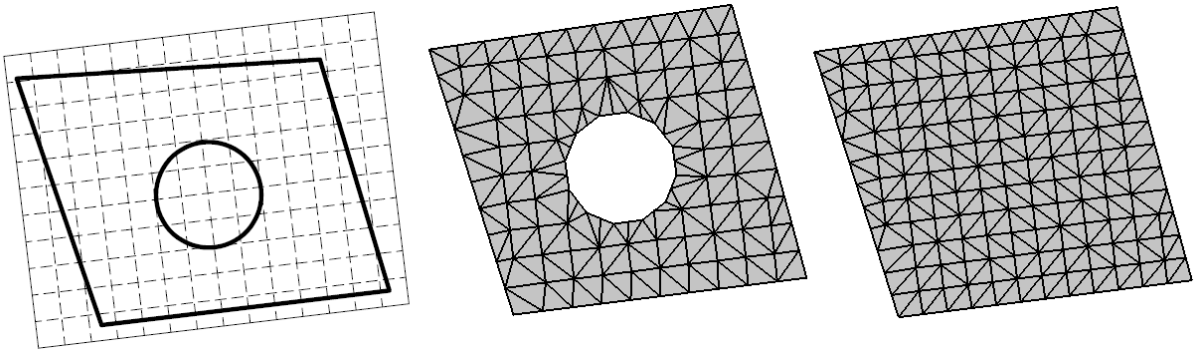
3.3.1 Posicionamento dos nós

Para que a malha contenha elementos de qualidade, ou seja, que apresentem correta orientação, pouca distorção e adequada razão de aspecto, é fundamental que a distribuição de nós seja o mais regular possível.

Dada uma face qualquer F_i é primeiro necessário atribuir nós a seus vértices e em seguida subdividir as arestas com espaçamento uniforme ou não, dependendo das necessidades do usuário e/ou de alguma propriedade geométrica local, tal como a curvatura, por exemplo.

Atribuídos nós aos vértices e arestas, é preciso distribui-los no interior do domínio. A forma mais óbvia é através do posicionamento de uma grade de nós regularmente espaçados, excluindo aqueles que ficarem dentro de furos, além das fronteiras do domínio ou ainda muito próximos a elas. A figura 3.3 mostra esse tipo de inserção de nós. Observar que a qualidade da discretização no entorno do furo é inferior ao restante do domínio. Isso ocorre porque o posicionamento em grade não contorna de maneira satisfatória a curvatura do furo.

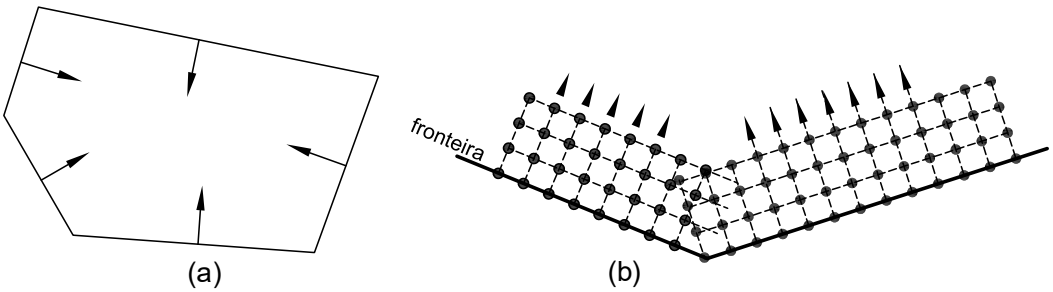
Figura 3.3 – Posicionamento de nós através de uma grade regular no interior de um domínio com e sem furo



Fonte: Elaboração do próprio autor

Há outros métodos de posicionamento dos nós, tais como a combinação de malhas estruturadas e por avanço de fronteira (BERN; PLASSMANN, 2000). Neste último, os pontos da malha partem da fronteira do domínio (ou de algum furo em seu interior) e vão sendo posicionados em camadas sucessivas. Essa técnica pode resultar em elementos de baixa qualidade onde as frentes se encontram, como pode ser visto na figura 3.4 b), mas isso pode ser resolvido por técnicas de suavização de malha tais como aquelas apresentadas por Bern e Plasmann (2000).

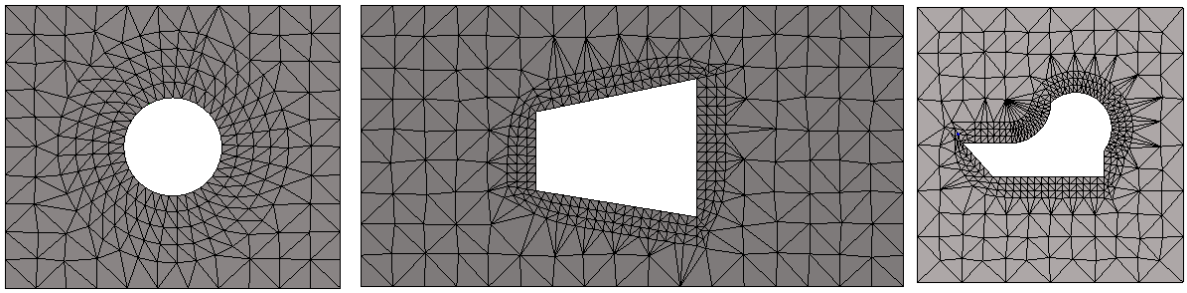
Figura 3.4 – Colocação de pontos por expansão de fronteira: a) esquema básico; b) intersecção de duas frentes em um canto



Fonte: Elaboração do próprio autor

Há ainda a opção de conjugar mais de um tipo de geração de pontos: o posicionamento por avanço de fronteira em furos, aliado ao posicionamento por grade no restante do domínio. Nesse caso, os nós por avanço de fronteira são posicionados primeiro, em seguida, os nós da grade regular cuja distância em relação aos primeiros seja maior que uma certa tolerância, são adicionados. A figura 3.5 exibe malhas desse tipo criadas no CABEMT. Nas duas ilustrações, central e à direita, a rápida transição de tamanho entre os elementos gerados por avanço de fronteira e os da grade originaram elementos de baixa qualidade na região de transição. A redução de tamanho dos elementos da grade e/ou o gradual aumento dos elementos nas camadas da fronteira em expansão devem amenizar esse problema.

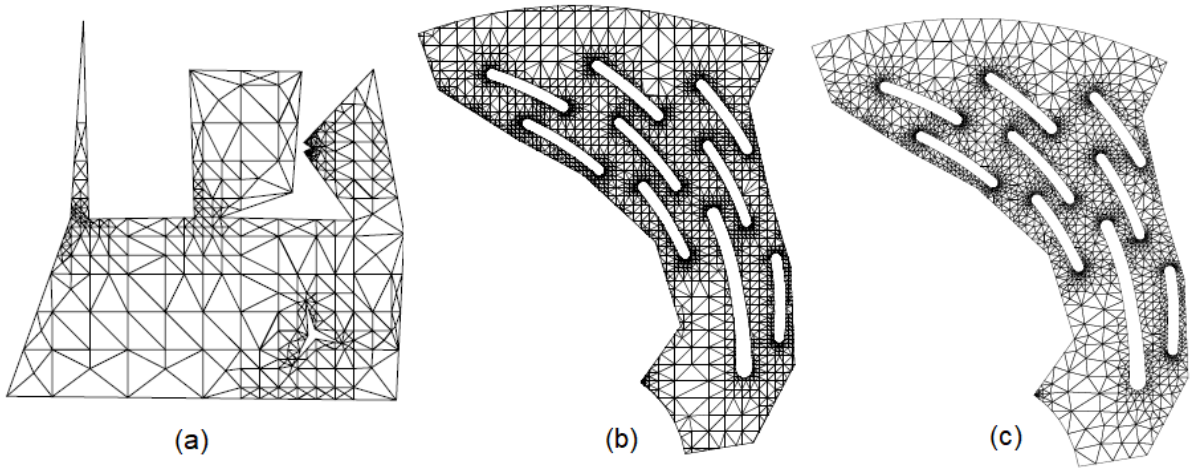
Figura 3.5 – Malhas geradas por posicionamento de nós por grade regular e avanço de fronteira nas bordas dos furos. Malhas não otimizadas/suavizadas geradas no CABEMT



Fonte: Elaboração do próprio autor

A outra maneira de dispor os nós é por meio da decomposição espacial, utilizando os *quadtrees* para o caso bidimensional, cuja construção foi mostrada no item 2.2.1. Nessa técnica, os nós são posicionados nos vértices dos quadrantes quando eles se encontram integralmente dentro da face. Se estiverem parcialmente dentro do domínio, eles podem ser seccionados para satisfazer o contorno (BATISTA, 2005). O algoritmo de Bern e Plasmann (2000) divide as células do *quadtree* de tal modo que cada uma das células resultantes tenha no máximo um vértice do contorno da face. Isso simplifica a subdivisão e a posterior triangulação dos pontos nesta célula. Frey e Marechal (1998) usam as arestas da fronteira e seus nós para a subdivisão do *quadtree*. Nesse caso, cada célula pode conter no máximo uma aresta ou nó pertencente à fronteira. Em seguida ocorre uma etapa de balanceamento, na qual as células vizinhas são verificadas quando ao seu tamanho. Se forem maiores que um fator de 2 vezes, são divididas. Isso visa simplificar a etapa de criação dos elementos (FREY; MARECHAL, 1998). A figura a seguir mostra alguns exemplos de malhas geradas por *quadtrees*:

Figura 3.6 – Malhas geradas por quadtree: a) uma face com ângulos agudos, b) malha gerada por octree – antes de otimização, c) malha após a otimização.



Fonte: a) Bern e Plasmann (2000); b) e c) Frey e Marechal (1998)

A formação dos elementos nos *quadtrees* pode se dar por triangulação de Delaunay dos nós ou pela conexão direta dos pontos utilizando as informações de vizinhança de cada célula.

3.3.2 Triangulação de delaunay 2D

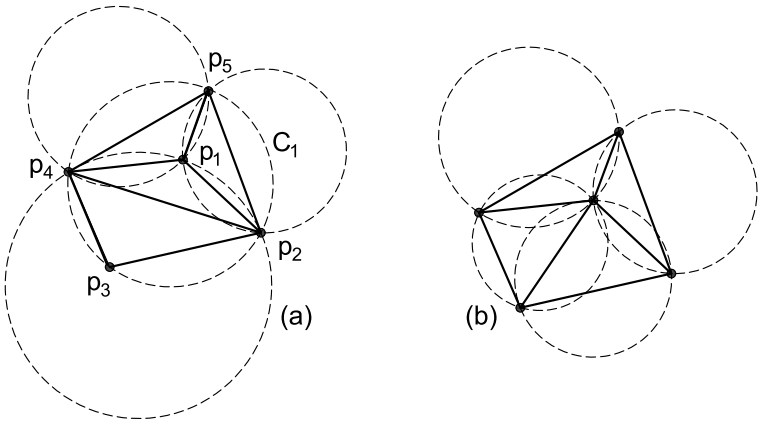
Dado um conjunto de pontos numa face, existem inúmeras formas de se combinar esses pontos em conjuntos de triângulos, mas a triangulação de Delaunay[28] é a mais utilizada, pois normalmente resulta em uma malha de melhor qualidade e garante a não formação de elementos cujas arestas se interceptam[29], a não ser em suas extremidades. Ademais, a triangulação de Delaunay maximiza os menores ângulos (EDELSBRUNNER, 2000), evitando então elementos muito desfavoráveis numericamente.

Lawson (1972), citado por Sibson (1978), foi um dos primeiros a criar um algoritmo de triangulação baseado na proposta de Delaunay. Sibson (1978) demonstrou que há somente uma triangulação localmente equiangular de um conjunto convexo de pontos, e essa é a triangulação de Delaunay. Isso implica que a mudança de diagonal de um quadrilátero formado por dois triângulos compartilhando uma aresta, não resultará em ângulos mínimos maiores que os atuais, caso os triângulos respeitam a condição de Delaunay. Essa propriedade é desejável, pois queremos maximizar os ângulos mínimos, portanto, se dois triângulos respeitam a condição de Delaunay, mudança de arestas não resultará em triângulos “melhores” que os já existentes.

Supondo um conjunto de pontos $P = \{\boldsymbol{P}_1, \boldsymbol{P}_2, \dots, \boldsymbol{P}_n\}$ distintos e não colineares, a triangulação de Delaunay é definida pela condição do círculo vazio. Um triângulo $\boldsymbol{P}_i, \boldsymbol{P}_j, \boldsymbol{P}_k$ é Delaunay se o seu circuncírculo não engloba nenhum outro ponto de P. No caso de haver sub-conjuntos de P com quatro ou mais pontos cocirculares, qualquer triangulação é válida entre esses pontos (BERN; PLASSMANN, 2000).

[28] Em homenagem ao matemático russo Boris Delaunay que em 1934 publicou *Sur La Sphère Vide* (Na Esfera Vazia), dando origem à triangulação que recebe seu nome.
[29] Supondo que não há 4 ou mais nós cocirculares.

Figura 3.7 – Duas triangulações possíveis para um conjunto de cinco pontos: a) não-Delaunay; b) Delaunay



Fonte: Elaboração do próprio autor

Na figura 3.7 o ponto P_1 está dentro de C_1, violando a condição de círculo vazio. Notar que a triangulação em (a) realmente tem um aspecto visualmente pior que a triangulação em (b), pois apresenta ângulos muito agudos e outros obtusos demais.

Como visto, uma etapa fundamental na triangulação de Delaunay é verificar se um ponto está dentro de um círculo que passa por outros três, que nada mais são que um potencial triângulo a se formar ou não. Guibas e Stolfi (1985) oferecem um teste em forma de determinante para avaliar isso. Suponha um conjunto de pontos $P = \{\boldsymbol{P}_1(x_{p1}, y_{p1}); \boldsymbol{P}_2(x_{p2}, y_{p2}); \boldsymbol{P}_3(x_{p3}, y_{p3}); \boldsymbol{P}_4(x_{p4}, y_{p4})\}$ no qual é necessário testar se $\boldsymbol{P}_4$ está dentro do circuncírculo formado por $\boldsymbol{P}_1, \boldsymbol{P}_2, \boldsymbol{P}_3$. Essa condição será verdadeira se (GUIBAS e STOLFI, 1985):

$$\mathcal{D}(\boldsymbol{P_1}, \boldsymbol{P_2}, \boldsymbol{P_3}, \boldsymbol{P_4}) = \begin{vmatrix} x_{p1} & y_{p1} & x_{p1}^2 + y_{p1}^2 & 1 \\ x_{p2} & y_{p2} & x_{p2}^2 + y_{p2}^2 & 1 \\ x_{p3} & y_{p3} & x_{p3}^2 + y_{p3}^2 & 1 \\ x_{p4} & y_{p4} & x_{p4}^2 + y_{p4}^2 & 1 \end{vmatrix} > 0 \qquad (3.2)$$

Portanto, em uma malha com n pontos, cada agrupamento de três pode ser testado por meio da inequação (3.2) contra todos os outros $n - 3$ pontos. Basta que um ponto resulte em um determinante positivo para que o agrupamento seja desclassificado como possível candidato a triângulo Delaunay. Tal observação nos leva automaticamente à seguinte instrução para realizar uma triangulação Delaunay em um conjunto de pontos:

Instrução 3.1 - triangulação de Delaunay – método direto e muito lento

Dados: uma face planar em $\mathbb{R}^3$, contendo ou não furos, e associada a um conjunto de n pontos ou nós.

a) Criar um eixo de coordenadas cujo plano XY seja coincidente com o plano da face;

b) Transformar todos os pontos para o eixo de coordenadas criado em a);

c) Criar três iteradores em cascata[30] (I_1, I_2, I_3), cada qual percorrendo todos os nós da face. Assim, numa determinada iteração, haverá três nós ($^{I_1}N_i$, $^{I_2}N_j$, $^{I_3}N_k$) candidatos a formar um elemento provisório E_l;

d) Criar mais um iterador de nós (I_4) hierarquicamente inferior aos demais e responsável por percorrer os nós de teste $^{I_4}N_m$;

e) Testar se o nó $^{I_4}N_m$ está dentro do circuncírculo formado pelos nós de E_l por meio da inequação (3.2);

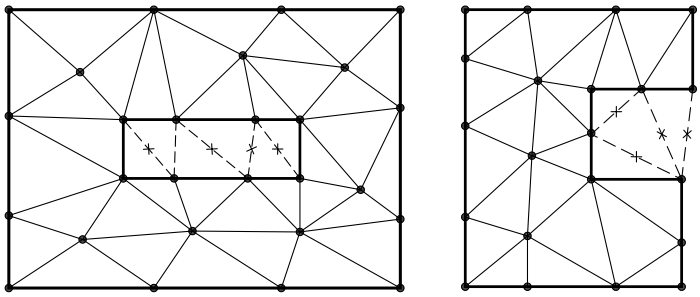
f) Se $\mathcal{D} > 0$, $^{I_4}N_m$ estará dentro de E_l e portanto os nós $^{I_1}N_i$, $^{I_2}N_j$, $^{I_3}N_k$ não formam um triângulo de Delaunay. Nesse caso, deve-se resetar I_4 e formar o próximo elemento provisório $^{I_1}N_i$, $^{I_2}N_j$, $^{I_3}N_{k+1}$. Por outro lado, se o iterador I_4 terminar sem que a condição de $\mathcal{D} < 0$ tenha sido atingida, o elemento provisório E_l deve ser promovido a definitivo;

g) Após finalizadas todas as iterações, uma triangulação de Delaunay foi estabelecida;

h) As etapas anteriores não garantem que um elemento não possa ser criado erroneamente onde deveria ser um furo ou preenchendo indevidamente uma concavidade do domínio, tal como ilustrado na figura 3.8. Uma das maneiras de se resolver isso é fazer um processamento posterior dos elementos, apagando aqueles que possuírem arestas no interior de um furo ou exterior à face.

[30] Os iteradores possuem hierarquia. Quando I_1 iterar uma vez, I_2 já passou por n iterações, I_3 por n^2 e I_4 por n^3.

Figura 3.8 – Instrução 3.1 aplicada diretamente em um domínio com furo (a) e outro côncavo (b)



Fonte: Elaboração do próprio autor

A geração de elementos indevidos na figura (b) decorre de a triangulação sempre formar a envoltória convexa dos pontos. De fato, a maioria dos algoritmos da literatura tratam da triangulação em um conjunto de pontos ao invés de superfícies ou polígonos. Quando a triangulação se dá nesses últimos, recebe o nome de triangulação restrita. Por isso, concavidades e furos no domínio devem ter um tratamento especial quando o algoritmo utilizado leva em conta apenas os pontos a triangular, ignorando as arestas do domínio.

A implementação direta da instrução 3.1 leva a uma complexidade computacional elevada de $O(n^4)$, tornando-a pouco atrativa para a geração de malha com muitos nós, mesmo com o emprego dos processadores velozes de hoje em dia. A triangulação de uma só vez de cerca de 680 nós, dura cerca de 10 min em um processador mediano, uma demora incômoda. No entanto, além desse ser o algoritmo mais fácil de programar, ainda pode ser utilizado quando o modelo não tem muitos nós a triangular de uma só vez. Isso é comum na modelagem de sólidos poliédricos, porque a segmentação das superfícies permite a triangulação por bateladas, lembrando o processo de divisão e conquista que será discutido mais à frente.

Se nos dias de hoje a geração de malha na complexidade $O(n^4)$ já leva um certo tempo, no início de desenvolvimento desses algoritmos nos anos 70 e 80, a geração de malhas com esse custo computacional não era viável. Os pesquisadores então desenvolveram diferentes técnicas para reduzir o tempo de triangulação.

Uma das técnicas mais empregadas é a divisão e conquista. Lee e Schachter (1980) e Guibas e Stolfi (1985) descrevem algoritmos similares para essa técnica e Peterson (1998) descreve a sua implementação em detalhes.

Instrução 3.2 - Triangulação de Delaunay pela técnica da divisão e conquista

Dados: um conjunto de pontos distintos em $\mathbb{R}^3$, porém localizados no mesmo plano.

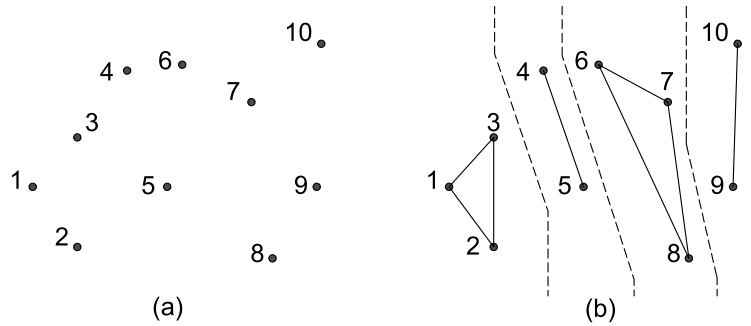
a) Criar um eixo de coordenadas no qual o plano XY seja paralelo ao plano que contém os pontos/nós. Em seguida, transformar as coordenadas desses pontos para o novo eixo;

b) Colocar em ordem crescente os pontos com base na coordenada x, porém no caso de coincidência, usar a coordenada y, também em ordenação crescente;

c) Agrupar os pontos ordenados em subconjuntos contendo entre dois e três pontos;

d) Triangular os subconjuntos. No caso de dois pontos, apenas traçar uma aresta entre eles. Os passos b), c) e d) encontram-se ilustrados na figura 3.9;

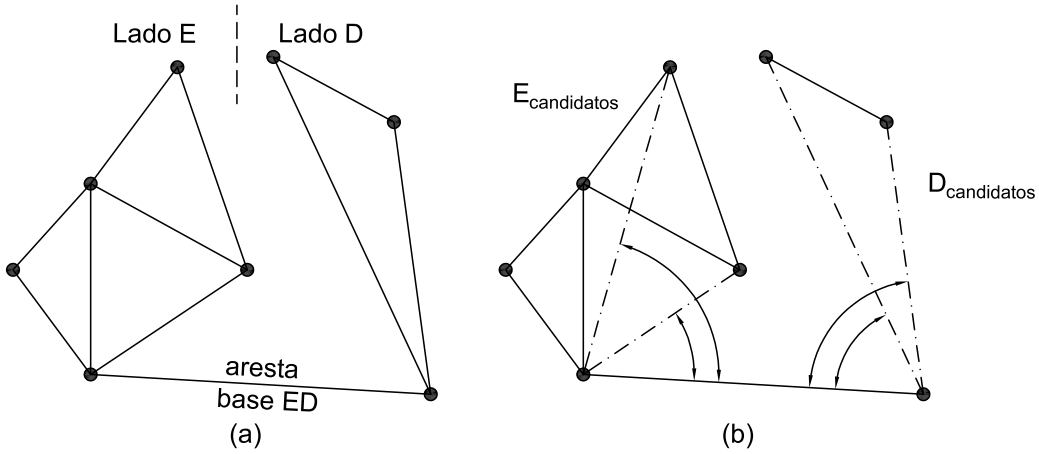
Figura 3.9 – Primeiros passos na triangulação de Delaunay por divisão e conquista: a) ordenação de pontos; b) divisão em subconjuntos e triangulação



Fonte: Adaptado de Peterson (1998)

e) Dado um par adjacente de subconjuntos – as arestas do subconjunto à esquerda serão denominadas EE. Da direita, DD. Arestas que vão de um subconjunto para outro, ED. É preciso unir os pares adjacentes de subconjuntos para concluir a malha. Inicia-se determinando a aresta inferior entre um par de subconjuntos. Podemos chamá-la de aresta base ED, conforme mostrado na figura abaixo:

Figura 3.10 – Designação de aresta base no algoritmo de divisão e conquista



Fonte: Adaptado de Peterson (1998)

Verificando a figura 3.10 (a) e considerando o sentido da aresta base como sempre da esquerda para direita, percebe-se que todos os demais pontos estão à sua esquerda, exceto obviamente pelas suas extremidades. Essa é uma maneira de se determinar a aresta base, percorrendo os nós até que todos eles fiquem à esquerda da aresta. Para se verificar a posição relativa de um ponto C em relação a uma aresta formada e orientada pelos pontos $A \rightarrow B$, recorre-se à equação (GUIBAS e STOLFI, 1985):

$$\begin{vmatrix} x_A & y_A & 1 \\ x_B & y_B & 1 \\ x_C & y_C & 1 \end{vmatrix} > 0 \qquad (3.3)$$

Sendo a desigualdade verdadeira, o ponto C formará um triângulo orientado no sentido anti-horário. Uma vez sabido que a aresta $A \rightarrow B$ vai da esquerda para direita, é certo que se a desigualdade (3.3) for satisfeita, o ponto C estará à esquerda de $A \rightarrow B$;

f) Agora é necessário encontrar os candidatos a serem arestas dos novos triângulos a serem formados para união dos subconjuntos. Em cada lado, toma-se uma extremidade da aresta base e segrega os pontos que unidos ao vértice da aresta base formam os menores ângulos com ela, tal como ilustrado na figura 3.10 (b). A união desses pontos segregados com a extremidade da aresta base em cada lado da triangulação dá origem a arestas ou candidatos a serem inseridos nos novos triângulos. Arestas do lado direito recebem a terminologia $D_{candidatos}$, e do lado esquerdo $E_{candidatos}$;

g) Verificar se os candidatos do lado direito são válidos. Para tal, seu ângulo com

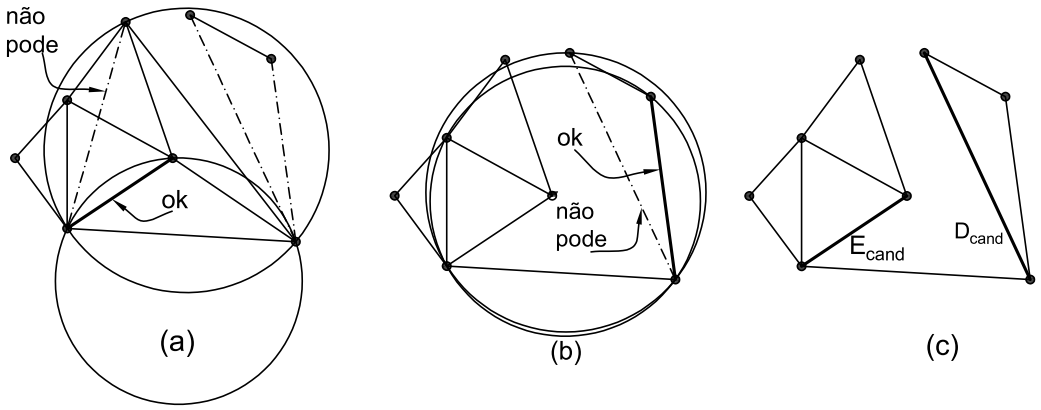
a aresta base deve ser inferior a 180° e o circuncírculo formado pelos pontos da aresta base e o ponto do candidato não deve conter o ponto do próximo candidato. Como essa última condição é a própria condição de Delaunay, no máximo um candidato será selecionado se os quatro pontos envolvidos nesse processo não forem cocirculares;

h) Repetir o passo anterior para o lado esquerdo. Para o exemplo da figura 3.10 (b), a seleção dos candidatos foi ilustrada na figura 3.11;

i) Se nenhum candidato for selecionado em ambos os lados a fusão entre os dois subconjuntos estará completa, formando então um novo subconjunto que deverá ser unido a outro adjacente. Portanto retornar ao passo e). Se não houver mais subconjuntos, a triangulação estará concluída;

j) Se no total somente um candidato for selecionado, então ele definirá a aresta ED que queríamos encontrar. Essa aresta será a nova base e retorna-se ao passo e);

Figura 3.11 – Seleção de candidatos para triangulação: a) à esquerda; b) à direita; c) resultado da seleção



Fonte: Elaboração do próprio autor

k) Se dois candidatos forem selecionados (obrigatoriamente será um de cada lado) então o candidato vencedor será aquele cujo circuncírculo por seus três pontos não contém nenhum outro ponto do lado direito ou esquerdo;

l) O candidato vencedor define uma aresta ED entre o seu vértice oposto à intersecção com a aresta base e o vértice desta no lado oposto – fechando o triângulo. Essa aresta será a nova base;

m) Retorna-se ao passo f).

Lee e Schachter (1980) e Guibas e Stolfi (1985) relatam uma complexidade

computacional de $O(n \log n)$ no pior caso para o algoritmo de divisão e conquista, sendo então um dos mais velozes. Entretanto, essa técnica pode falhar em malhas contendo agrupamentos de quatro ou mais nós quase-cocirculares, caso não seja utilizada aritmética precisa nos cálculos (SHEWCHUK, 1996).

Dwyer (1987) propôs uma simples modificação no algoritmo clássico de divisão e conquista, melhorando seu desempenho para $O(n \log \log n)$. Ao invés de segmentar o conjunto de nós somente no eixo x, ele dividiu o domínio em uma grade regular na qual cada célula continha pontos que pudessem ser imediatamente triangulados. Em seguida, as células adjacentes em cada linha da grade eram unidas de acordo com o algoritmo clássico. Finalmente, as linhas eram unidas da mesma forma. Esse método não é só mais veloz, como também menos sujeito a erros aritméticos quando não se usa aritmética exata (SHEWCHUK, 1996).

Dentre outras, existe uma técnica para a triangulação de Delaunay particularmente fácil de implementar, trata-se do método por inserção de pontos ou incremental, proposto inicialmente por Weatheril e Hassan (1994) e também explicado em detalhes por Zienkiewicz e Taylor (2005). Esse método, apesar de não ser o mais rápido, é de programação trivial se comparado aos algoritmos de divisão e conquista (MOURA, 2006).

Instrução 3.3 - Triangulação de Delaunay pela técnica de inserção de pontos

a) Dada uma superfície planar com um conjunto de N pontos P_i cuja triangulação de Delaunay deseja-se obter, faça uma triangulação manualmente de tal forma que todos os pontos estejam contidos nela. Recomenda-se triangular o envelope (*bounding box*) a partir do conjunto dos pontos que engloba: os quatro vértices desse envelope, um ponto central e mais quatro pontos (um no centro de cada aresta do envelope);

b) Inserir o ponto $\boldsymbol{P_i}$ ($i = 1,2,3 \ldots n$) na malha e percorrer todos os elementos já criados. Para cada um desses elementos, tomar os três pontos de seus vértices e juntamente com P_i efetuar o teste do círculo - eq. (3.2). Selecionar todos os elementos (e suas respectivas arestas) para os quais $\boldsymbol{P_i}$ cai dentro do circuncírculo formado pelos vértices de cada elemento;

c) Eliminar todos os elementos selecionados no passo anterior;

d) Do conjunto de arestas selecionadas no passo b), eliminar aquelas que possuem elementos excluídos no passo c) em ambos os lados – em suma, eliminam-se as arestas que ficarem órfãs de elementos;

e) Unir $\boldsymbol{P_i}$ aos vértices das arestas que sobraram do passo d);

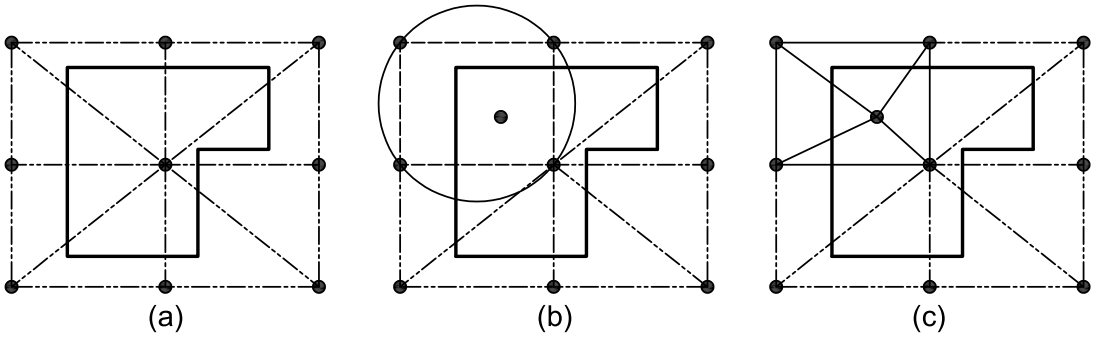
f) Retornar a b) até que todos os n pontos sejam inseridos;

g) Ao final do processo, elementos indesejados são excluídos, tratam-se daqueles externos ao domínio ou que estejam dentro de alguma cavidade (furo). Os primeiros são detectados por meio dos vértices do envelope e os últimos se pelo menos um de seus vértices estiver no interior da cavidade, mas existem outros meios.

No caso de $\boldsymbol{P_i}$ estar exatamente no circuncírculo de algum elemento, a triangulação não é única e pode ser resolvida manualmente.

A figura a seguir ilustra os procedimentos básicos para geração de malha por inserção de pontos:

Figura 3.12 – Triangulação por inserção de pontos: a) envelope inicial, b) inserção do ponto e exclusão dos elementos e arestas, c) criação dos novos elementos



Fonte: Elaboração do próprio autor

As técnicas de geração de malha aqui apresentadas trabalham em uma superfície plana. Uma vez que o CABEMT representa qualquer sólido como poliedro, a geração de malha localmente sempre se dará nessas condições. No entanto, a triangulação de Delaunay pode ser aplicada a elementos bidimensionais em superfícies não planares, conforme demonstra Chew (1993).

Embora a triangulação de Delaunay seja um elemento vital em muitos geradores de malha, há alguns métodos que não precisam dela. Um deles é a geração de malha por *quadtree*, pois nesse caso os elementos são triangulados com base nas

células e sua vizinhança, por meio do uso de padrões pré-estabelecidos como descrito por Frey e Marechal (1998).

3.3.3 Triangulação de delaunay 3D

Em geral a análise linear elástica pelo MEC requer apenas a discretização da superfície e no caso do CABEMT será feira pelos algoritmos descritos anteriormente. Entretanto, se simulações não-lineares foram requeridas (por exemplo: elasto-plásticas) faz-se necessária a discretização no interior do domínio, pelo menos nas partes cuja variável não-conservativa (ex.: deformação plástica) seja não-nula. Por isso, é importante que o programa tenha a capacidade de gerar malhas tridimensionais.

A geração desse tipo de malha segue os mesmos passos da instrução 3.3 porém os triângulos agora serão tetraedros e o teste expresso pela equação (3.2) pode ser naturalmente adaptado para avaliar se um ponto $\boldsymbol{P}_5$ está na circunsfera delimitada pelos quatro vértices ($\boldsymbol{P}_1, \boldsymbol{P}_2. \boldsymbol{P}_3, \boldsymbol{P}_4$) de um tetraedro, também denominado genericamente de célula.

A triangulação de Delaunay em domínios bidimensionais garante a maximização dos menores ângulos dos elementos e, portanto, alcança uma qualidade de malha otimizada[31]. Isso não ocorre em três dimensões, especialmente porque a triangulação de Delaunay pode formar elementos de qualidade muito baixa chamados *slivers* (LO, 1997). Pode ser necessário realizar algumas modificações na malha para eliminar tais elementos.

3.3.4 Otimização de malhas

A triangulação bidimensional de Delaunay é capaz de conectar os nós da melhor forma possível, mas isso não garante a qualidade da malha, uma vez que o posicionamento dos nós pode ser desfavorável, gerando elementos bastante distorcidos. Isso ocorre principalmente em malhas com variação grande e rápida do tamanho de elementos (ZIENKIEWICZ; TAYLOR, 2005). Essa característica pode ser notada na figura 3.5.

Uma das formas de aumentar a qualidade da discretização é através da técnica de suavização. Nela, a topologia da malha não se altera, apenas os nós interiores ao

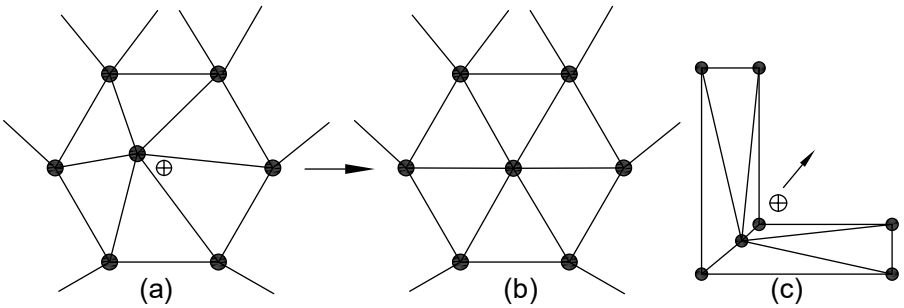
[31] A qualidade final da malha será boa ou não a depender do posicionamento dos nós.

domínio são reposicionados, como mostrado na figura 3.13 (a)(b). Um dos algoritmos mais velozes desse tipo é a suavização Laplaciana, que reposiciona um nó para o centroide dos seus vizinhos. A nova posição do nó é dada por (ZIENKIEWICZ; TAYLOR, 2005):

$$x_i = \frac{1}{N_{viz}} \sum_{j=1}^{N_{viz}} x_j \; ; \qquad y_i = \frac{1}{N_{viz}} \sum_{j=1}^{N_{viz}} y_j \tag{3.4}$$

Conforme Zienkiewicz e Taylor (2005) esse algoritmo deve ser executado repetidas vezes (três a cinco) para melhores resultados. Contrariando seu objetivo, a suavização Laplaciana pode piorar a qualidade de alguns elementos localizados em concavidades no domínio, tal como ilustrado na figura 3.13 (c). Esse problema pode ser contornado pelo monitoramento de alguma métrica de qualidade dos elementos que se conectam ao nó que está sendo movido, rejeitando o seu deslocamento caso seja identificada uma ação deletéria do algoritmo nos elementos associados a este nó. Como esse fenômeno, se ocorrer, será nas fronteiras do domínio, essa avaliação não precisa ser feita em todos os nós, somente naqueles que se conectam a elementos cujas arestas compartilham a fronteira.

Figura 3.13 – Suavização Laplaciana



Fonte: Adaptado de Zienkiewicz e Taylor (2005)

Em alguns casos, a baixa qualidade da malha pode ser inferida através da topologia. Zienkiewicz e Taylor (2005), por exemplo, recomendam o seguinte número ótimo de elementos conectados a um nó (NE), para uma malha triangular:

$$NE = \begin{cases} 6 & p/\ nós\ interiores \\ máx\left(\left\lfloor \frac{3\theta}{\pi} + \frac{1}{2} \right\rfloor, 1\right) & p/\ nós\ na\ fronteira \end{cases} \tag{3.5}$$

em que θ é o ângulo interno formado pelas arestas da fronteira que se conectam ao nó e $\lfloor c \rfloor$ é a parte inteira de c.

O reposicionamento de nós internos também pode ser feito por algum método

de otimização. Para um dado ponto, selecionam-se os elementos adjacentes. É estabelecida uma função objetivo que se queira minimizar, geralmente relacionada à alguma métrica de qualidade dos elementos. Utiliza-se um método numérico, tal como o do gradiente descendente para minimizar a função. Dai *et al*. (2014) fizeram o teste de otimização com diferentes métricas de qualidade e concluíram que em geral todas contribuem para substancial melhoria da malha tridimensional. No entanto, obtiveram melhores resultados com a métrica proposta por Geuzaine e Remacle (2009).

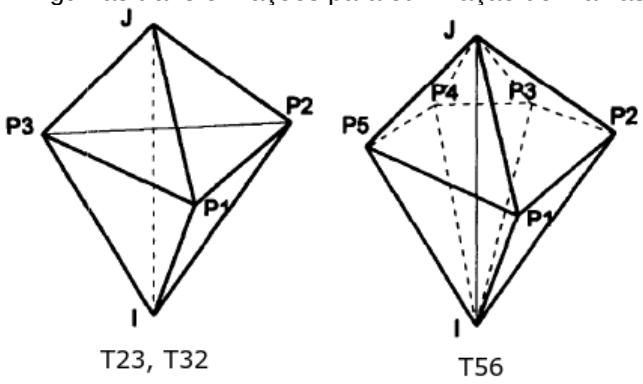
Além de reposicionamento, nós indesejados podem ser eliminados, bastando rastrear os elementos associados a eles e excluí-los. Então um novo elemento é gerado tomando-se as arestas comuns aos elementos excluídos.

No refinamento de malha, os pontos/nós podem ser inseridos estrategicamente para elevar a qualidade dos elementos. Uma técnica bastante difundida é o refinamento de Delaunay e consiste na inserção de nós, chamados de pontos de Steiner, normalmente no circuncentro de triângulos de baixa qualidade (RUPPERT, 1993). Erten e Üngör (2009) sugerem diferentes critérios para determinar o ponto de inserção conforme a posição do circuncentro do triângulo e do petal[32] de sua menor aresta.

No caso de malhas tridimensionais, além do reposicionamento dos nós internos, é comum otimizar a malha por meio de mudanças topológicas, ou seja, da conectividade de seus elementos (MA; WANG, 2021; LO, 1997) – escolhe-se um agrupamento pequeno de tetraedros (normalmente 2 ou três) e formam-se novos elementos no lugar dos anteriores através da permutação de arestas. Se a qualidade dos novos elementos for superior, a transformação é mantida, caso contrário, retorna-se à configuração inicial. O CABEMT usa as transformações T23, T32 e T56, conforme mostrado na figura a seguir:

[32] O petal de uma aresta é um círculo que passa pelos seus extremos (P_i, P_F) e está ao lado direito dela, obedecendo a sua orientação e cujo raio seja $2sen\ (\alpha_0)|\overline{P_iP_f}|$,onde α_0 é o menor ângulo admissível para um triângulo.

Figura 3.14 – Algumas transformações para otimização de malhas tetraédricas



Fonte: Adaptado de Lo (1997)

T23 significa passar de dois para três tetraedros. Como podemos ver na figura, a configuração de dois tetraedros consiste nos elementos $P_1P_2P_3J$ e $P_1P_3P_2I$. Podemos dividir esse conjunto de pontos em três, ao invés de dois, tetraedros: P_1P_2JI, P_3P_1JI, P_2P_3JI. Processos análogos ocorrem com as transformações T32 e T56. Se a métrica de qualidade do conjunto de elementos não melhorar após as transformações elas são desfeitas.

Para as malhas tridimensionais tetraédricas, o CABEMT calcula a qualidade dos elementos através da seguinte fórmula (GEUZAINE; REMACLE, 2009):

$$Q = \frac{6\sqrt{6}\bar{V}}{\left(\sum_{i=1}^{4}\bar{S}_i\right)L_{máx}} \tag{3.6}$$

na qual Q é a qualidade, $\bar{V}$ o volume do tetraedro, $\bar{S}_i$ é a área da face i do tetraedro e $L_{máx}$ é o maior comprimento de aresta do tetraedro. A função objetivo utilizada foi:

$$f_q = -\left(0{,}7\sum_{i=1}^{N}\frac{Q_i}{N_t} + 0{,}3Q_{min}\right) \tag{3.7}$$

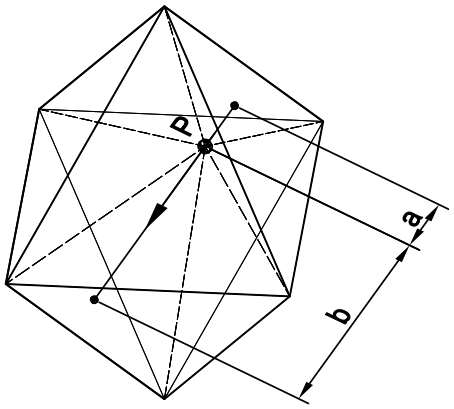
na qual N é o total de tetraedros que se conectam a um determinado nó, Q_{min} é a qualidade do pior elemento dentro do agrupamento. Observa-se então que se optou por estratégia de otimização local, procurando minimizar a função f_q em cada nó interno da malha. Fez-se uma média ponderada entre a qualidade do pior elemento (peso = 0,3) e a qualidade média dos demais (peso = 0,7) para que não ocorresse o risco de “sacrifício” da qualidade do pior elemento em detrimento aos outros.

Dado um determinado nó interno cuja posição se deseja otimizar de modo a minimizar f_q, esperando melhorar a qualidade da malha, calcula-se o vetor gradiente:

$$\boldsymbol{V_q} = -\left\{\frac{\partial f_q}{\partial x}, \frac{\partial f_q}{\partial y}, \frac{\partial f_q}{\partial z}\right\}^T \tag{3.8}$$

No caso do CABEMT, tal vetor é calculado numericamente mediante o deslocamento não simultâneo e não cumulativo da posição (x, y, z) do nó interno por Δx, Δy e Δz. A posição do nó será otimizada dentro de um segmento imaginário que se estende de uma distância a e vai até b, sendo estas as distâncias no nó até a intersecção com alguma face do conjunto de tetraedros, conforme mostra a figura 3.15. Admitiu-se o deslocamento máximo do nó interno no intervalo $\left\{-\frac{1}{5}a, +\frac{1}{5}b\right\}$.

Figura 3.15 – Otimização de posicionamento de um nó interno em malha tetraédrica



Fonte: Elaboração do próprio autor

O ponto ótimo (aquele que minimiza f_q) entre $-1/5a$ e $+1/5b$ pode ser determinado pelo conhecido algoritmo de bissecção.

4 O PROGRAMA DESENVOLVIDO: CABEMT

4.1 INTRODUÇÃO

A criação do CABEMT buscou o desenvolvimento de um *software* de simples manipulação e ao mesmo tempo flexível, no sentido de permitir ao usuário a simulação de estruturas de complexidade baixa e moderada. A prática de engenharia requer uma “caixa de ferramentas” para que possa realizar um trabalho de qualidade, mas também de maneira rápida. Os engenheiros e técnicos encontram suas ferramentas, com frequência, na forma de programas computacionais. O CABEMT englobará todas as etapas para realização de simulações numéricas, sem depender de programas externos, e é orientado para o emprego em pelo menos duas grandes disciplinas da engenharia – a manutenção e o projeto.

A manutenção exige uma abordagem multidisciplinar para que seja exercido um processo que:

- Reduza a quantidade de falhas mediante o emprego de estratégias adequadas de manutenções preventivas, preditivas, análises de falhas, entre outros;
- Atenda aos requisitos anteriores preservando a segurança, saúde, meio-ambiente e com o menor custo possível.

É evidente que esses aspectos são conflitantes entre si e, portanto, uma “solução de compromisso” deve ser determinada de modo a equilibrar os esforços direcionados para o cumprimento de seus objetivos. Um desses esforços consiste em avaliar as solicitações e mecanismos de danos aos quais as estruturas estão submetidas, de modo a acompanhar sua deterioração e definir períodos de inspeção ou substituição. Busca-se evitar ao máximo a ocorrência de falhas importantes durante a operação dos equipamentos.

Não se pode realizar tais avaliações de integridade com excelência sem o apoio de ferramentas numéricas e da mecânica da fratura, dado que a maior parte dessas análises requereria um tempo elevado de execução se fossem feitas manualmente, quando isso sequer é possível. Não obstante, ferramentas computacionais de métodos numéricos tendem a ser bem completas, porém se mostram de uso difícil quando não são rotineiramente manipuladas, situação típica nos setores de manutenção.

Ademais, o emprego da mecânica da fratura para problemas reais e que não podem ser simplificados para o caso bidimensional, pode esbarrar na dificuldade em modelar trincas no MEF. Por sinal, softwares foram desenvolvidos[33] somente para a geração da malha, necessitando de um programa de elementos finitos para calcular a solução. Tais programas geradores de malhas especiais podem ser dispendiosos se não utilizados corriqueiramente, não justificando sua aquisição em muitas empresas. Por outro lado, alternativas de código livre nem sempre são de uso simples.

O CABEMT, por outro lado, vai operar de maneira independente de outros programas e pode ser utilizado em qualquer sistema operacional que execute a máquina virtual do Java.

O CABEMT também se mostrará como uma fonte independente de análise de tensões ou de MF para confronto com outros resultados numéricos de maneira a emprestar mais confiança sobre alguma análise.

Por fim, segue um resumo das características do CABEMT e as razões pelas quais tais características são importantes.

- Simplicidade de uso: porque nem todos os seus usuários irão trabalhar constantemente com o programa. Em certas ocasiões, especialmente nas atividades de manutenção, soluções rápidas são necessárias e não há tempo para se reaprender os procedimentos de utilização de um *software* complexo;
- Independência: porque a dependência a outros programas além de poder violar a característica anterior de simplicidade, também pode acarretar custos quando tais softwares são de uso proprietário. Não obstante, a independência permite a comparação entre resultados numéricos de programas e técnicas diferentes;
- Custo: se vier a ser comercializado, seu custo será baixo, característica derivada de sua simplicidade. Também poderá existir uma versão gratuita com menos funcionalidades. Até o momento não foi decidido sobre qual caminho tomar;
- Portabilidade: o CABEMT ocupa alguns poucos *megabytes* de memória e por ser escrito em Java, pode ser executado em diferentes sistemas operacionais sem ser recompilado, uma vez que a execução se dá na máquina virtual do Java.

[33] Zencrack

4.2 AS LINGUAGENS DE PROGRAMAÇÃO E O INTERFACEAMENTO GRÁFICO

A escolha do *software* de interfaceamento gráfico, isto é, do programa responsável por interpretar as descrições matemáticas dos objetos de modo a exibi-los na tela, foi relativamente simples. Isso ocorre porque há basicamente duas opções principais: o Direct3D é um software da Microsoft® e o OpenGL, como o próprio nome diz, é um programa de código aberto e gratuito desenvolvido pela Silicon Graphics (JACKIE; DAVIS; DIXON, 1997). Pelas suas características de ampla aceitação, código aberto e generalidade, o OpenGL foi selecionado para o interfaceamento gráfico do CABEMT. Havia também a opção do Java3D, mas este é mais limitado que o OpenGL.

Se a escolha do *software* gráfico foi simples, por outro lado, há um número grande, talvez excessivo, de linguagens de programação à disposição hoje em dia. Torna-se desafiador selecionar a linguagem mais adequada à elaboração do programa tema deste trabalho caso algumas características desejáveis não sejam estabelecidas, como as descritas a seguir:

- Universalidade: a linguagem deve ser amplamente difundida. Isso facilita a busca de informações e detalhes cruciais à implementação dos algoritmos. Existindo uma grande base de usuários, também haverá informações abundantes sobre a linguagem;
- Persistência: a linguagem deve se mostrar capaz de resistir ao tempo e manter sua base de usuários. Isso tende a evitar a obsolescência e consequentemente, falta de suporte técnico, de IDEs atualizadas e de compatibilidade;
- Simplicidade: a linguagem deve ter a sintaxe simples e que facilite a leitura do algoritmo nela escrito. Isso facilita a programação em geral e também a revisão e modificação dos algoritmos quando necessário. Aqui se excluirão, portanto, linguagens de operem em baixo nível[34], pois sua leitura é bastante complicada;
- Orientação a objeto: a programação orientada a objeto (POO) é um exitoso paradigma de programação e que se adapta muito bem tanto à modelagem de sólidos quanto às entidades utilizadas nos métodos numéricos em geral (nós,

[34] São linguagens que operam quase ao nível de máquina, como o *Assembly*, por exemplo.

elementos, malha). Portanto, a linguagem deve ter suporte à POO;

- Compatibilidade com o OpenGL: a linguagem deve ser capaz de trabalhar com o OpenGL;
- Velocidade e uso de memória: embora os computadores modernos tenham um poder de processamento elevado, de tal forma que a maioria dos softwares não usam pleno poder computacional por longos períodos, o MEC exige uma quantidade enorme de operações matemáticas para a montagem do sistema linear e também para a sua resolução. Além disso, dado que as matrizes dos sistemas de equações do MEC são densas, o custo de memória também é elevado. Por isso, é de grande relevância que a linguagem de programação resulte em um programa veloz e que tenha baixo consumo de memória na medida do possível;
- Custo financeiro: a linguagem de programação deve ser gratuita;
- Portabilidade: é desejável que o programa compilado seja capaz de executar em várias plataformas, sem problemas de compatibilidade;
- Bibliotecas: a linguagem deve ter um bom repertório de bibliotecas de apoio, especialmente de funções matemáticas e de resolução de sistemas de equações;

Vamos estabelecer uma lista das linguagens ou interpretadores de *script* mais usuais: FORTRAN, C, C++, Pascal, Java, Visual Basic, Python, Scilab, Matlab, Octave.

- FORTRAN: pioneira e respeitada linguagem de programação de alto nível utilizada no mundo científico. Tem o inconveniente de não trabalhar com POO e perdeu espaço para linguagens mais modernas. No entanto, ainda é bastante utilizada;
- C: uma das mais antigas e ainda bastante utilizada. Não aceita POO;
- C++: é uma linguagem totalmente baseada em C, porém aceita POO. Seu uso, entretanto, não é dos mais simples. Ela apresenta muitas palavras chave e não tem recolhimento automático de lixo na memória. Provavelmente a mais veloz, excetuando-se a C. A linguagem requer um compilador para transformar o código em arquivos executáveis que são específicos para a arquitetura de CPU e do sistema operacional (AL-QAHTANI *et al*., 2010). Essa característica é comum a linguagens que não trabalham em máquinas virtuais;

- Pascal: tem caído em desuso;
- Java: muito popular e que atende aos requisitos da lista. Sua sintaxe é bastante parecida com a do C++, portanto não é das mais simples;
- *Visual Basic*: é uma linguagem da Microsoft voltada para uso geral. É incomum ver trabalhos científicos escritos em *Visual Basic*;
- Python: muito popular e com uma curva ascendente de adeptos. É voltada para uso científico, mas tem suporte a praticamente tudo. Atende aos requisitos da lista;
- Scilab e Octave: não são linguagens de programação, e sim, interpretadores de *scripts*. Muito utilizadas em meios acadêmicos e científicos. Tem o inconveniente de necessitar da instalação desses programas para executarem os algoritmos implementados. Além disso, são consideravelmente mais lentas que as outras linguagens descritas e a ligação direta com o OpenGL não é possível;
- Matlab: além dos problemas do Scilab e Octave, tem custo elevado.

  Ao confrontarmos as linguagens e interpretadores com as características desejadas, ficamos com as seguintes opções: C++, Java e Python.

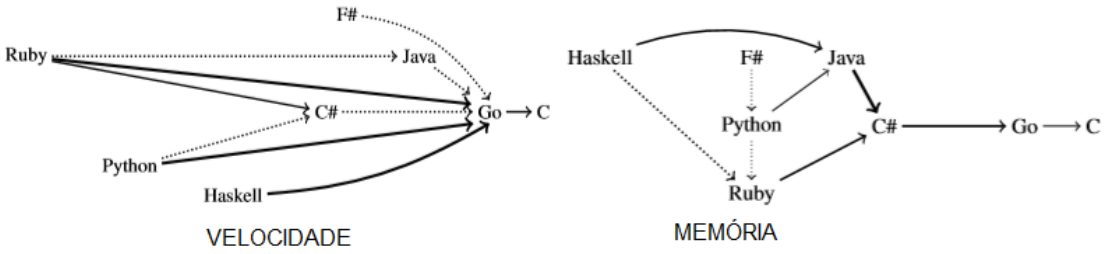
  O C++ não foi escolhido porque é de maneira geral uma linguagem mais complexa e não possui o valioso recurso de coleta automática de lixo da memória, exigindo que tal processo seja escrito pelo programador. Por consequência, o tempo de desenvolvimento torna-se maior. Além disso, os programas em C++ devem ser recompilados para execução em sistemas operacionais de natureza diferente, prejudicando um pouco a portabilidade do software.

  O Python é uma linguagem de programação simples e de sintaxe bastante limpa, tornando o código de fácil leitura (HUMER; FOSTER, 2014). Ele possui muitas bibliotecas técnico-científicas e uma ampla comunidade de usuários.

  O Java, embora mais voltado para uso geral, apresenta uma quantidade considerável de bibliotecas científicas em consequência de sua popularidade.

  Em termos de velocidade, pelo menos na época em que o CABEMT começou a ser desenvolvido, o Java era tipicamente mais veloz que o Python, como pode ser observado nos testes feitos por Nanz e Furia (2015) e Arouba e Villaverde (2015). Essa tendência também é mostrada em estudos mais recentes, tais como de Khoirom *et al*. (2020).

Figura 4.1 – Grafos comparativos de velocidade e consumo de memória para algumas linguagens de programação. O C é o mais veloz e consome menos memória. As distâncias estão aproximadamente em escala.



Fonte: Nanz e Furia (2015)

Observa-se que o consumo de memória do Java também é reduzido, mas não é difícil encontrar fontes que mostram o contrário, como por exemplo o estudo de Stein e Geyer-Schulz (STEIN; GEYER-SCHULZ, 2013). Isso pode depender do tipo de problema a ser avaliado.

Concluindo, Java, Python e C++ são boas linguagens para a implementação dos códigos do CABEMT. Optou-se pelo uso do Java por ser mais similar ao C++, linguagem familiar ao autor.

Algumas terminologias de POO e também do Java merecem um destaque para entendimento dos itens posteriores da dissertação:

- Classe: é um constructo, um escopo ou uma estrutura capaz de descrever ou representar um objeto por meio de seus atributos. Por exemplo, uma classe pode conter a estrutura de representação de um elemento finito cujos atributos seriam os seus nós, sua área, volume, cor ou qualquer outra característica de interesse. Além dos atributos, as classes contêm os métodos ou funções;
- Objeto: é um ente manipulável através dos atributos definidos por sua classe. Por exemplo, da classe “elemento”, podem ser criados (instanciados) os objetos elemento1, elemento2... Cada um deles contêm suas próprias características correspondentes aos atributos de sua classe (element1 cor = verde, área = 20...; element2 cor = azul, área = 15...);
- Métodos ou funções: são algoritmos e instruções pertencentes a uma classe. Eles podem ser não-estáticos quando necessitam obrigatoriamente serem chamados por um objeto daquela classe – ou podem ser estáticos, quando não necessitam de um objeto para serem invocados. Classes que possuam somente métodos estáticos comportam-se praticamente igual à uma rotina de programação procedural;

- Pacotes ou bibliotecas: são um conjunto de classes e métodos que estendem as funcionalidades básicas do programa. No desenvolvimento do CABEMT utilizaram-se os seguintes pacotes de terceiros: jBlas, jLapack, jogl, Jama, commons-math e vecmath. jBlas e jLapack são pacotes derivados das famosas bibliotecas Blas e Lapack escritas em Fortran e que fazem o tratamento de matrizes, solução de sistemas lineares e outras operações. Jogl é a biblioteca que permite o uso do OpenGL no Java. Jama é uma biblioteca de suporte para trabalho com matrizes e vetores. Commons-math é um pacote com diversas funções matemáticas e vecmath para trabalhos com vetores. Muitas funções matemáticas, no entanto, foram escritas especificamente para o CABEMT, de modo a serem o mais simples possível, tais como as funções abrigadas nas classes *Calculator* e *Calculator2*.

O código foi escrito na interface de desenvolvimento (IDE) *NetBeans* versão 7.2 e posteriormente na versão 10. O *NetBeans* é bastante completo e tem boas ferramentas de debug e perfilamento. *Debug* é o processo de avançar passo a passo ou por etapas no código e ler as variáveis, de modo a encontrar possíveis erros. O perfilamento consiste em estudar o uso de processamento e memória por cada função do programa de maneira a identificar gargalos no desempenho do código.

4.3 HISTÓRICO, PRINCÍPIOS E PREMISSAS DO PROJETO

O CABEMT iniciou-se como um projeto experimental e de aprendizado, com foco especial em modelagem de sólidos. Por conseguinte, não havia objetivos claros a respeito de quais caminhos e marcos deveriam ser percorridos ao longo de seu desenvolvimento.

Entretanto, conforme o tempo foi passando e algumas linhas de desenvolvimento foram descartadas, percebeu-se a possibilidade de criação de um modelador de sólidos funcional, embora simples. Uma vez que a experiência do autor no ramo de inspeção de equipamentos indicava uma carência de programas efetivos e amigáveis para o estudo de mecânica da fratura, vislumbrou-se estender a funcionalidade do modelador para que pudesse tratar esse tipo de problema.

Surgiu então um objetivo norteador do trabalho: a criação de um programa de análise de tensões com o foco em mecânica da fratura e que fosse utilizável na prática

de engenharia. Ainda era necessário definir o método numérico para implementação.

Naturalmente, o método dos elementos finitos surgiu como primeira opção. Contudo, percebida a dificuldade em gerar malhas tridimensionais de qualidade e especialmente criação de trincas nesse tipo de malha. Além disso, existe uma miríade de programas computacionais de MEF –, não seria interessante criar mais uma. Sendo assim, alternativas foram procuradas.

O MEC se mostrou como uma opção bastante interessante, uma vez que necessitava de elementos somente na superfície do sólido. Isso iria poupar muito tempo na implementação do gerador de malha, além de ser conveniente para a modelagem de trincas porque afinal, são superfícies. Outras vantagens do MEC são a melhor precisão para o mesmo tamanho de malha se comparado ao MEF e também a capacidade de atribuição não-aproximada de condições de contorno em fronteiras infinitas (ALIABADI, 2002).

Terminada a implementação dos códigos de MCS, iniciou-se a programação do módulo elástico-linear, em seguida será implementada a capacidade de realizar simulações de problemas termoelásticos transientes, haja vista que ciclagens de tensões térmicas são comuns geradores/propagadores de defeitos. Por fim, será implementada a capacidade de tratar problemas elasto-plásticos.

Embora os objetivos foram se consolidando aos poucos, o desenvolvimento do CABEMT sempre teve como premissa de que o programa:

- Seria de simples utilização: por isso procurou-se limitar o número de comandos ao essencial, implementar dicas de uso em seus ícones (*tooltips*), fazer com que sua interface fosse semelhante a outros *softwares*;
- Permitiria a interação por mouse e por comandos: o programa deveria ser utilizado por meio do mouse e do uso de comandos. Isso agiliza a elaboração dos modelos;
- Todos os modelos sólidos seriam representados por poliedros: pois caso fossem adotadas as representações por NURBS ou quádricos, o tempo de implementação seria muito estendido. A representação por poliedros é adequada para a maioria dos problemas.

No que tange ao código em si, procurou-se utilizar o idioma inglês para nomear funções, objetos, classes e variáveis. Esse idioma, além de sua universalidade,

parece ter uma boa capacidade de condensação de ideias *versus* número de caracteres utilizado para expressá-las. Isso deixa o código mais limpo e de fácil leitura. Os comentários, que são importantes para o entendimento dos algoritmos, foram elaborados em português. A partir daqui então, uma certa mistura dos idiomas ocorrerá, visto parecer razoável manter os mesmos nomes de classes na dissertação e no programa.

Quando se está programando, por vezes surge um conflito do tipo memória *vs* processamento. Peguemos um círculo, por exemplo. Podemos armazenar uma série de pontos subsequentes que o descrevem e no momento de desenhá-lo, basta invocar esses pontos de uma lista e passá-los como argumento para a função OpenGL apropriada[35]. Por outro lado, poderíamos armazenar somente o centro e o raio do círculo e toda vez que este precisasse ser desenhado, calcularíamos os pontos. Este método certamente consumirá menos espaço na memória, mas exigirá mais processamento.

Procurou-se um equilíbrio entre o consumo dos recursos de memória e processamento, uma vez que ambos são altamente exigidos no MEC. Entretanto, há um maior privilégio ao uso do processamento, já que ele pode ser interrompido em alguns setores do programa quando funções exigentes ao processador são invocadas. Isso em geral pode ser feito com a memória, a menos que se opte pelo lento processo de armazenar dados em disco.

É boa prática de programação atribuir nomes significativos às variáveis e atributos de classes ainda que para isso o nome acabe se tornando mais longo que a prudência estética permitiria. É também importante dividir as tarefas em objetivos simples e, portanto, mais facilmente conquistáveis. Ficar “preso” em partes do código não é incomum e tal divisão pode reduzir o efeito psicológico da frustração em não se avançar.

Ter soluções conhecidas para as quais se escreve um algoritmo é de suma importância para testar e, pelo menos reduzir, as chances de ocorrerem problemas quando uma parte maior do código entra em funcionamento. Erros podem ser difíceis de se encontrar, especialmente após troca de informações entre muitos algoritmos –

[35] O OpenGL pode ser considerado uma linguagem de baixo nível. Não existe função para desenhar o círculo, deve ser informado o conjunto de segmentos.

situação na qual podem acumular mais de um tipo de erro, inclusive. Daí a importância do teste por etapas, algoritmos ou trechos de código. Utilizar uma IDE com boas capacidades de *debug* é crucial no tratamento de erros.

O tratamento de erros e exceções é uma tarefa importante na programação. Geralmente os erros acontecem quando o usuário insere parâmetros incorretos ou não previstos na idealização do programa. Então condicionantes especiais devem ser dispostas no código, de modo a capturar essas impertinências e exibi-las ao usuário. Isso não só ajuda o mesmo a contornar o problema, como também facilita o programador encontrar em qual parte do código o erro ou exceção está sendo gerada.

O CABEMT possui um sistema básico de tratamento de erros em algumas regiões críticas do código, porém, foi dada prioridade ao desenvolvimento de novas funções em vez de refinar o tratamento. De qualquer modo, a investigação de problemas nos algoritmos pode ser feita via *debug*, bastando possuir o estado e as condições geradoras de um determinado tipo de erro em mãos.

O CABEMT não tem como premissa fundamental ser à prova de erros, de tal sorte que o usuário pode gerar formas não realistas ao usar o modelador de maneira inapropriada. Essas anomalias geométricas, entretanto, devem ficar bastante explícitas e também causar erros em estágios mais avançados da modelagem. Cercar todas as possibilidades que levariam a irrealismos demandaria um tempo elevado e com pouco resultado em termos de confiabilidade do programa. Esse equilíbrio entre tempo de programação de um algoritmo e vantagens a serem obtidas com ele (custo benefício) foi sempre um norteador no desenvolvimento do CABEMT.

4.4 DIAGRAMA DE FUNCIONAMENTO DO CABEMT

Na figura 4.2 está representado o diagrama de funcionamento do CABEMT. Nele encontram-se a maioria das classes utilizadas – cada caixa do fluxograma representa uma classe que por sua vez podem conter muitas funções/métodos. Caminhos com setas bidirecionais indicam que a informação sai da classe de origem, é processada na classe destino e retorna para a origem. Caixas tracejadas são classes formadas basicamente por métodos estáticos, ou seja, normalmente não são criados objetos baseados nelas. Funcionam como um repositório de funções.

Caixas contínuas representam classes que formarão objetos a serem utilizados no CABEMT. Linhas fantasma (traço dois pontos) representam um agrupamento de

funções.

A classe principal é a *Modeler* e ela basicamente gerencia as outras, além de receber e “escutar” comandos do usuário, é responsável por monitorar todos os menus do programa. *Modeler* também gera sinais periódicos para que *Drawer* possa atualizar a tela. Uma vez que o usuário possui uma forte interação com os sólidos e as primitivas de modelagem, a classe *Modeler* apresentará um fluxo significativo de informação para *Solid* e *Geometric Object*, representado pelas linhas contínuas espessas.

Geometric Object é uma classe abstrata[36] que fornece o escopo comum a uma série de entidades geométricas fundamentais conforme ilustrado do fluxograma. Essas entidades, ao serem processadas por certos comandos, se transformam em superfícies e arestas que por sua vez podem constituir sólidos ou permanecerem como superfícies livres. Neste último caso, para a geração de trincas ou cortes nas superfícies de algum sólido. Via de regra, as entidades primitivas tem dimensões chaves designadas por coordenadas através de objetos da classe *Vertex*. Além disso, como as curvas são representadas por vários segmentos de reta, *arcdiv* se encarrega de alterar esse número de divisões mediante intervenção do usuário.

Drawer contém as funções responsáveis por transformar as informações matemáticas das entidades geométricas em gráficos/imagens na tela. Essa classe trabalha com muitas funções da biblioteca *jogl*. Há nela uma chave de reconhecimento de mudança de cena que reenvia os dados para processamento somente quando alguma alteração substancial é feita. Caso contrário, todas as informações gráficas ficam em *calllists* (um recuso do OpenGL), de modo a acelerar significativamente a exibição dos gráficos.

Drawer também chama funções de auxílio ao desenho, tais como linhas de traçagem e magnetos. Esses são atratores do cursor do *mouse* para pontos notáveis, tais como as extremidades, pontos intermediários, quadrantes e pontos de intersecção das primitivas. Tais pontos são determinados pelas funções da classe *intersectCalculator* que executa toda vez que alguma primitiva de modelagem é

[36] Uma classe abstrata precisa ser concretizada por meio de outra classe. A classe abstrata pode guardar métodos e propriedades comuns às classes concretas que dela se derivam. Por exemplo, todas as entidades geométricas elementares possuem em comum cor, espessura de linha, status de seleção e a sua função de desenho.

inserida ou removida da cena. Os magnetos, por rastrearem pontos de frequente interesse, auxiliam bastante na criação dos modelos geométricos.

Modifier produz alterações nas primitivas, tais como copiar, mover, girar, aparar, criar cópias paralelas e criar arcos de concordância.

Solid armazenará todas as entidades necessárias à representação dos sólidos, além de funções básicas como de translação, rotação, desenhar e outras auxiliares. Observar que a origem de um objeto tipo *Solid* provém da operação de funções das classes *Extrude, Revolve, Loft* ou *Sweep* sobre alguma *Face* que é, por sua vez, constituída por primitivas. *Solid* pode ser modificado (observar setas bidirecionais) por meio de operações booleanas, chanfros, arredondamento de arestas, corte ou a criação de uma trinca. Uma vez que essas operações podem desorganizar os dados de *Solid, Refactor* tem o objetivo de organizar a estrutura de dados e refazer as devidas referências[37] entre seus elementos. *Hole* é a entidade que representa furos nas faces.

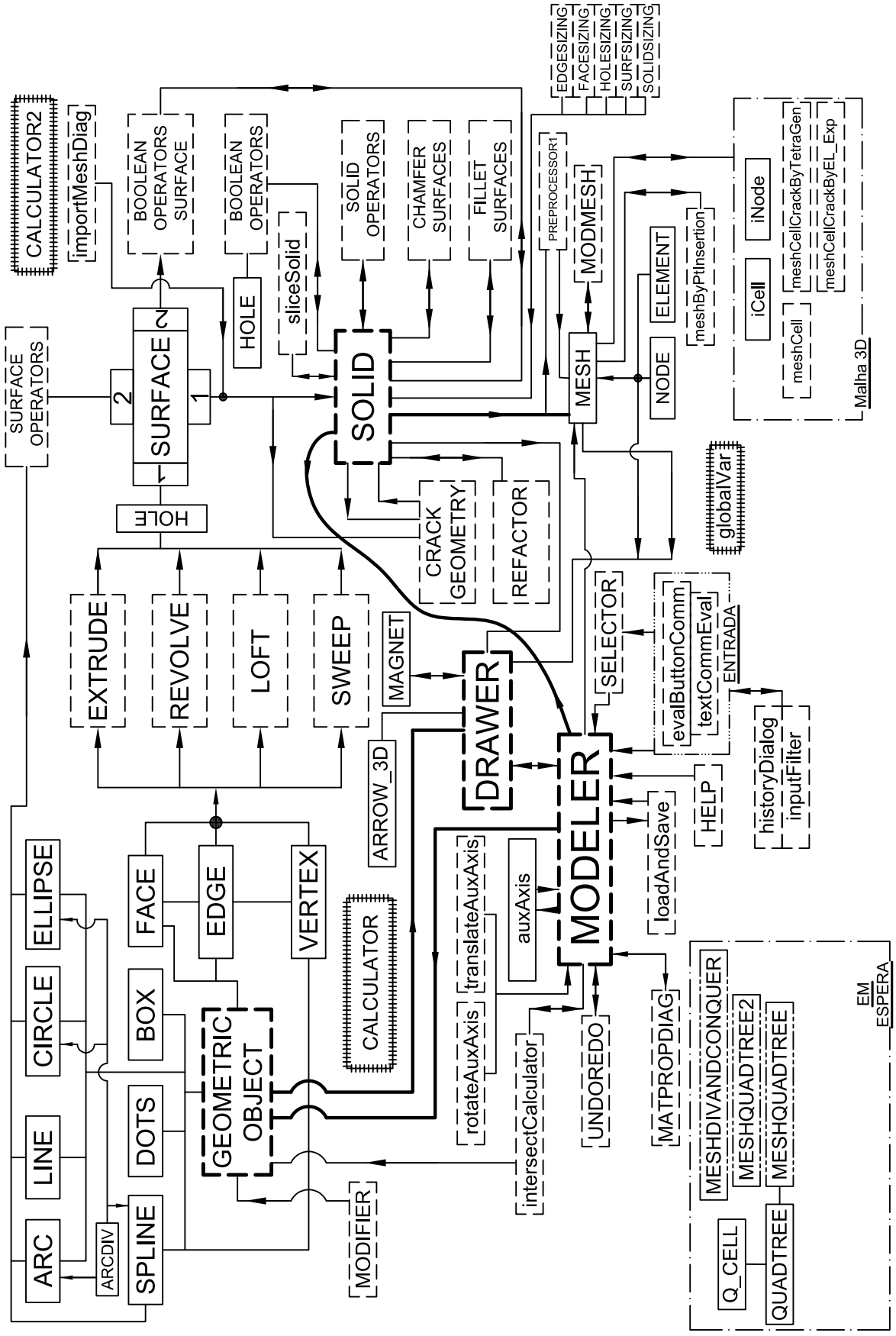
Observa-se que a "matéria-prima" de *Solid* são as *Surfaces,* porém nessa caixa do fluxograma existe um filtro. Somente superfícies provenientes dos operadores *Extrude, Revolve, Loft* ou *Sweep* são destinadas à formação de sólidos. Superfícies oriundas por extrusão ou varredura direta de primitivas, isto é, criadas por *Surface Operators*, são destinadas a outras funções, dentre as quais modificar um sólido por corte ou inserção de trinca, mas jamais formarão um sólido.

Solid envia informações para a construção da malha na classe *Mesh,* cujas características podem ser modificadas por *Modmesh*. *Mesh* dá origem a um objeto que contém outros das classes *Node* e *Element.* As condições de contorno são inseridas via comandos do usuário, lidos por *textCommEval* e *evalButtonComm,* processados por *Modeler* e enviados para *Mesh.* Há classes (*edgesSizing, faceSizing,* etc) responsáveis por receber parâmetros de tamanho de malha do usuário. Esses parâmetros são transferidos aos objetos sólidos e seus componentes por meio de *Preprocessor1* e *Mesh,* alterando as propriedades dos objetos cabíveis (tamanho dos elementos nas arestas e faces).

---

[37] Referência em Java tem o mesmo significado de ponteiros em C/C++, embora estes apresentem meios de operar em seus endereços diretamente. Referência a um determinado objeto é o seu endereço. Isso permite uma economia grande de memória, pois um objeto pode ser referenciado por vários outros. Ao invés de criarmos cópias desses objetos, armazenamos apenas seu endereço.

Figura 4.2 – Macrofluxograma do CABEMT



Fonte: Elaboração do próprio autor

A classe *undoRedo* é responsável pelos comandos desfazer e refazer.

No agrupamento “Em Espera” existem classes que não foram plenamente desenvolvidas de modo a serem utilizadas no CABEMT.

Selector é a importante classe que permite a seleção dos objetos pelo usuário através do mouse, é fundamental na modelagem de sólidos. O *OpenGL* tem uma função nativa de seleção baseado nos pixels ocupados pelas entidades gráficas. O processo é complicado e escrever as próprias funções de seleção pareceu mais apropriado.

HistoryDiag armazena e exibe o histórico de comandos utilizados no terminal do CABEMT, assim como os retornos textuais de certas funções, além de mensagens de aviso ou erro.

LoadAndSave é a classe responsável por transferir as informações acerca dos objetos ativos no CABEMT, portanto armazenados na memória RAM, para um arquivo. Em uso posterior do programa os objetos podem ser reconstruídos através da leitura deste arquivo, também realizada pela classe *loadAndSave.*

A importante classe auxAxis, além de permitir ao usuário a definição de sistemas de coordenadas auxiliares, tem uso em várias funções no CABEMT, especialmente quando se faz necessário trabalhar com coordenadas no plano das faces ou de elementos. A posição do eixo auxiliar, além de ser definida no momento de sua criação, pode ser modificada pelos métodos das classes *rotateAuxAxis* e *translateAuxAxis.* Objetos da classe *auxAxis* são também associados à cada face, representando o sistema de coordenadas particular de cada uma.

Se olharmos as classes *calculator* e *calculator2* no diagrama, veremos que o contorno de sua caixa está diferente. Isso é para representar que essas classes, compostas basicamente de funções estáticas, se conectam a praticamente todas as outras e ficaria inviável representar isso no diagrama. Elas são uma coletânea de funções matemáticas e auxiliares indispensáveis ao funcionamento do CABEMT e que complementam as funções importadas de bibliotecas/pacotes.

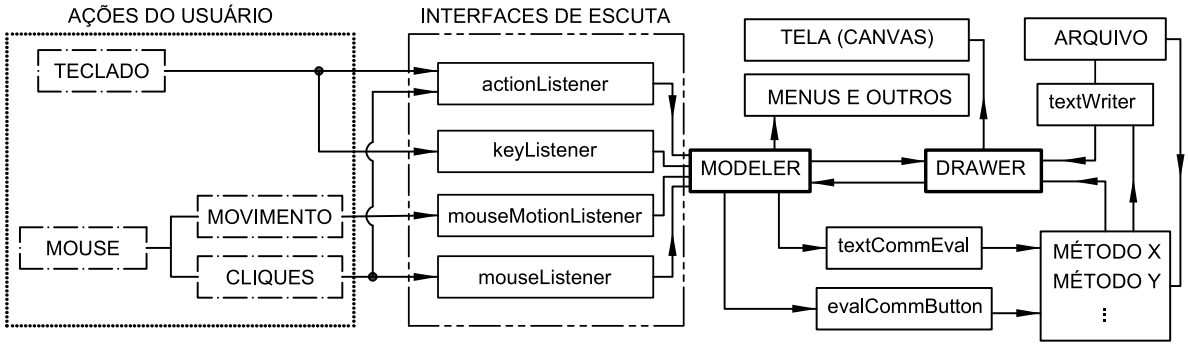
A classe *globalVar* tem a mesma natureza universal de *calculator* e *calculator2*, porém, ao invés de métodos, armazenam variáveis globais, cujo uso é recorrente em muitas funções.

4.5 INTERFACES E PROCESSAMENTO DE ENTRADA E SAÍDA

4.5.1 Diagrama

Na figura 4.3 há um esquema dos principais fluxos de dados relativos à entrada e saída de dados do CABEMT que consolida as explicações anteriores. Notar o papel central da classe *Modeler e suas interfaces,* e da classe *Drawer*.

Figura 4.3 – Diagrama do processamento de entradas e saídas do CABEMT



Fonte: Elaboração do próprio autor

4.5.2 Descrição

As interfaces de entrada e saída são ponto crucial de qualquer programa. Afinal de contas, passam por elas as informações que dão razão de existência ao *software*. As entradas do usuário sempre são recebidas pelas funções da classe *Modeler* e depois encaminhadas para os métodos devidos. *Modeler* implementa interfaces de monitoramento de eventos oriundas da classe principal de escuta – *eventListener.* As interfaces implementadas por *Modeler* são: *actionListener* (monitora ações nos menus)*, keyListener* (monitora pressionamentos de teclas), *mouseListener* (monitora cliques do *mouse*) e *mouseMotionListener* (monitora os movimentos do cursor do mouse).

As saídas não necessariamente passam por *Modeler*. Os dados podem ser enviados diretamente para um arquivo ou para a tela, por exemplo.

4.5.3 Entradas

O CABEMT recebe três tipos de entradas: via teclado, *mouse* ou por um arquivo externo.

- Comandos por teclado: são códigos digitados em um campo especial (*prompt*), marcado com a letra (e) na figura 4.4 ou então teclas de atalho para facilitar a execução de algumas tarefas, com destaque para seleção de entidades e a

movimentação do ponto de vista (ver APÊNDICE B).

Esses comandos tem uma classe dedicada a seu tratamento: *textCommEval.* Quando o usuário digita algo no *prompt* e tecla *enter* (ou espaço) –, *textCommEval* (chamado por *Modeler*) lê a *string*[38] completa do *prompt*. Se for digitado "*line*", a string lida será "*Command:line*". Subtrai-se a *string padrão "Command:"* ficando então apenas com o comando do usuário que então percorre uma série de operações condicionais (*if*) em cascata e caso coincida com algum comando do CABEMT, faz a sua ativação, senão, apenas limpa o *prompt.*

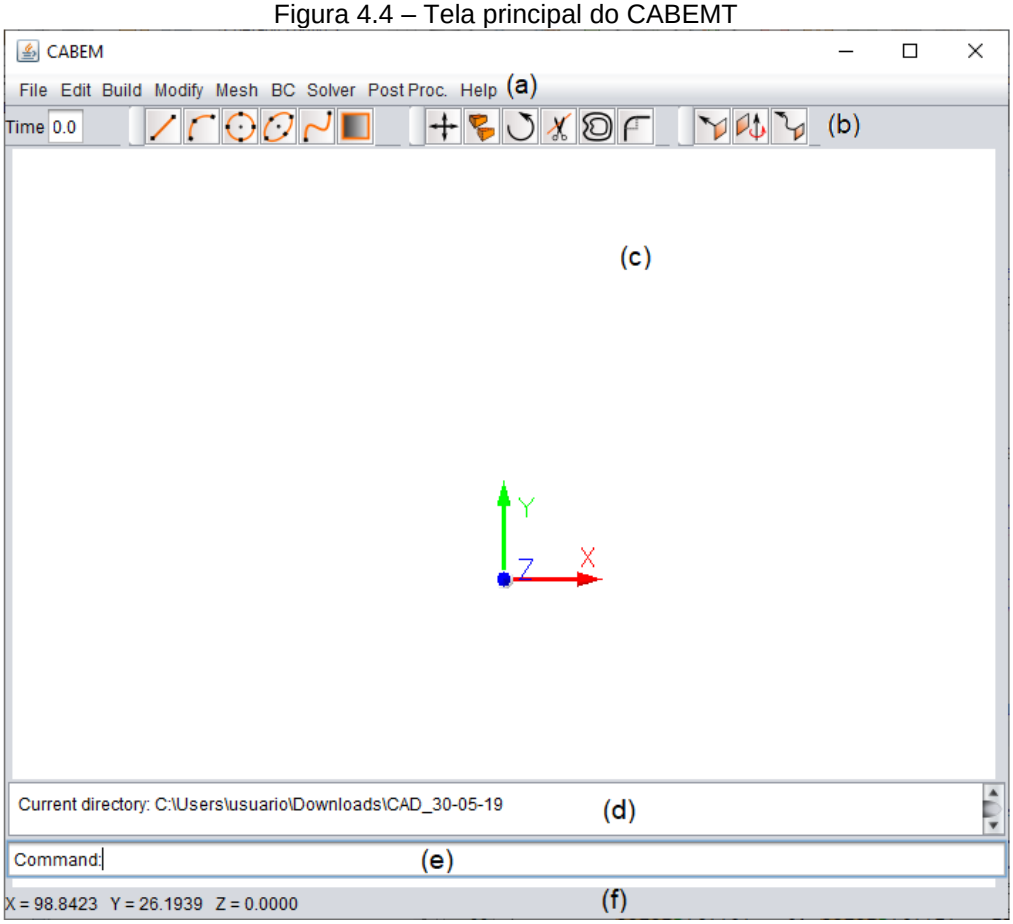
Alguns comandos requerem mais dados. É o caso de desenhar uma linha por exemplo. O usuário, após entrar o comando "*line",* é perguntado sobre o ponto inicial e em seguida o ponto final para especificação da linha.

A entrada desses pontos pode ocorrer via teclado, então *textCommEval* transfere as coordenadas digitadas para um método da classe *line* (no caso o método *drawLineByComm*) que fará a validação do comando inserido e se estiver correto, cria o primeiro vértice da linha. Como *textCommEval* sabe que deve enviar a *string* para o método *drawLineByComm?* Toda vez que um comando requer intermediários, ele incrementa uma variável específica em *globalVar* (por exemplo: *lineStatusIndicator*), sendo assim, *textCommEval* verifica se a variável está alterada ou não. Se estiver, transfere o comando para a função apropriada. Ao final dos comandos, a variável em *globalVar* retorna ao seu valor original, tipicamente zero;

- Comandos pelo cursor do *mouse:* por causa de sua agilidade e poder expressivo, o mouse é talvez a melhor interpretação da agilidade manual do usuário. Praticamente todas as tarefas no CABEMT podem ser realizadas com este instrumento. Voltando à figura 4.4, em (a) o usuário tem acesso aos diversos menus do CABEMT por meio do *mouse*. Em (b) alguns ícones com as funções mais utilizadas, de modo a agilizar a modelagem. Em (c) tem-se a tela principal de trabalho (*canvas*) na qual o usuário pode interagir com as entidades geométricas, modificá-las, selecioná-las e manipular o ponto de vista.

[38] Cadeias de caracteres.

Figura 4.4 – Tela principal do CABEMT

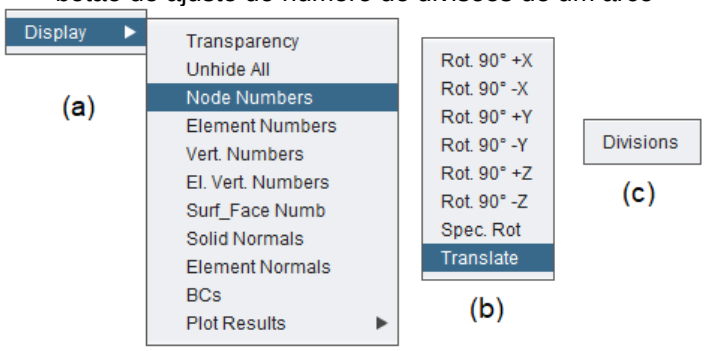


Fonte: Elaboração do próprio autor

Para conveniência do utilizador, a roda do *mouse* faz a aproximação e afastamento do ponto de vista (*zoom*) e quando mantida pressionada executa a translação (*pan*). Cliques com o botão esquerdo na tela principal geralmente selecionam algum objeto ou determinam algum ponto no momento em que uma entidade é criada.

Cliques com o botão direito são programados para abrir menus auxiliares, conforme mostrado na figura 4.5. Eles são processados pelos métodos da classe *evalButtonComm*.

Figura 4.5 – Menus abertos por meio de clique com o botão direito na área de trabalho do CABEMT: a) menu auxiliar geral; b) menu de modificação de sistema auxiliar de coordenadas; c) menu com botão de ajuste do número de divisões de um arco



Fonte: Elaboração do próprio autor

A posição do cursor é continuamente monitorada e registrada por *Modeler,* que atualiza as variáveis globais (*mouseX*, *mouseY*) – marcadoras da posição em pixels do cursor. Entretanto, não queremos as coordenadas da tela em pixels e sim, as coordenadas de nosso espaço virtual em uma unidade de medida “real” sobre o sistema de coordenadas que desejarmos. Para entender como esse processo de conversão funciona, precisamos observar a figura 4.6. Ela mostra o volume de projeção ortográfica do *OpenGL*. A face deste volume, paralela ao plano XY e que está mais próxima do ponto de observação é o plano de corte próximo (*nearClipPlane)*, ao passo que a mais distante é o plano de corte distante (*farClipPlane)*. Uma linha de visada (traço ponto) atravessa esses dois planos e também o plano $X'Y'$ do qual as coordenadas queremos determinar. Essa linha se intercepta com o cursor do *mouse* mostrado com uma seta na figura.

A função *gluUnProject* pode determinar as coordenadas globais (XYZ) nos planos próximo e distante do volume de projeção de tal forma que se pode calcular a equação da linha de visada. Como todas as informações do eixo auxiliar de coordenadas ($X'Y'Z'$) são conhecidas, simples manipulações algébricas nos dão as coordenadas globais de $\boldsymbol{P}$.

Consideremos os vetores diretores globais $X = (1,0,0); Y = (0,1,0); Z = (0,0,1)$ e também os vetores unitários X', Y' e Z' expressos em coordenadas globais. A matriz de transformação do eixo auxiliar e a conversão entre coordenadas é:

$$[\boldsymbol{M}] = \begin{bmatrix} \boldsymbol{X}.\boldsymbol{X}' & \boldsymbol{Y}.\boldsymbol{X}' & \boldsymbol{Z}.\boldsymbol{X}' \\ \boldsymbol{X}.\boldsymbol{Y}' & \boldsymbol{Y}.\boldsymbol{Y}' & \boldsymbol{Z}.\boldsymbol{Y}' \\ \boldsymbol{X}.\boldsymbol{Z}' & \boldsymbol{Y}.\boldsymbol{Z}' & \boldsymbol{Z}.\boldsymbol{Z}' \end{bmatrix} \quad (4.1)$$

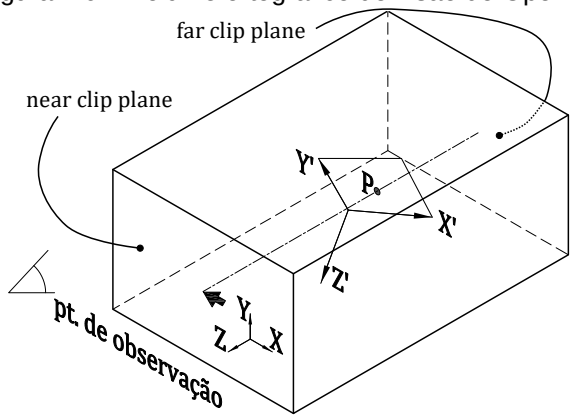
$$\boldsymbol{P}'' = \boldsymbol{P} - (X_0, Y_0, Z_0) \quad (4.2)$$

$$\boldsymbol{P}' = [\boldsymbol{M}]\boldsymbol{P}'' \quad (4.3)$$

$$\boldsymbol{P} = [\boldsymbol{M}]^{-1}\boldsymbol{P}' + (X_0, Y_0, Z_0) \quad (4.4)$$

na qual X_0, Y_0, Z_0 é a posição da origem do eixo de coordenadas auxiliar expressa em coordenadas globais, $\boldsymbol{P}'$ é o ponto $\boldsymbol{P}$ expresso coordenadas locais. A equação (4.1) pertence a funções da classe *auxAxis* e as demais à classe *Vertex*;

Figura 4.6 – Volume ortográfico de visão do OpenGL



Fonte: Elaboração do próprio autor

- Entrada de dados por arquivo externo: basicamente podem ser os arquivos salvos anteriormente pelo próprio CABEMT ou então arquivos com a extensão "*dat"* contendo uma malha gerada por programa externo. No primeiro caso, a classe *loadAndSave* faz a leitura de objetos que foram processados pela interface *serializable* durante o salvamento do arquivo. Essa interface precisa ser estendida a todos os objetos que devem ser guardados na memória quando o arquivo é salvo (linhas, círculo, sólidos, malhas, etc.). A ordem em que os objetos são escritos, deve ser a mesma da leitura (serialização).

  A leitura de arquivos externos foi implementada para importação de sólidos gerados por outros programas. O sólido não é importando diretamente, e sim, a malha que foi aplicada sobre ele. Essa malha deve ser apenas superficial. Um exemplo de programa que gera malhas desse tipo, com a extensão "dat", é o gerador de malhas do pacote Salome-Meca.

4.5.4 Saídas

A interface de saída de dados é responsável por informar *status* do CABEMT ao usuário, salvar arquivos, além de mostrar informações de retorno de métodos e a visualização dos gráficos em tela. Voltando à figura 4.4, observamos que a área de trabalho (*canvas* – item c) também é uma interface de saída, uma vez que os gráficos e praticamente toda a informação de interesse é exibida nessa região. A classe *Drawer* é a principal responsável por levar informações até a área de trabalho, fazendo isso por meio dos métodos de representação gráfica elementares do *OpenGL.*

Abaixo da área de trabalho encontra-se a caixa de texto de histórico (item d) onde são colocadas informações em texto concernentes a retorno de funções, avisos, erros e até de resultados de simulações. As mensagens destinadas a essa caixa de texto passam pelos métodos da classe *textWriter*, responsáveis por conferir formatação ao texto. Ao utilizar um clique duplo nessa área, ela é expandida para melhor visualização.

A barra inferior (item f) exibe a posição corrente do cursor em coordenadas cartesianas referentes ao sistema global de coordenadas e também para o sistema local correntemente selecionado.

Outra interface de saída é a gravação do conteúdo atual do CABEMT em arquivo via comando salvar. As funções da classe *loadAndSave* registram todos os objetos que possuem a implementação *serializable* em uma ordem definida – a mesma que será utilizada no momento de carregar o arquivo.

4.6 REPRESENTAÇÃO DE PRIMITIVAS NO CABEMT

As primitivas são os elementos básicos de qualquer modelo sólido no CABEMT. Grosso modo, são usadas para a criação de faces que por sua vez originarão os sólidos. A maioria dessas primitivas são representadas como segmentos de reta na tela enquanto sua representação matemática é dada pelos atributos/características de sua classe. Alguns autores fazem o discernimento entre entidades geométricas (ponto, reta, superfície) e entidades topológicas (vértice, linha e face). Tal distinção não será feita aqui, porque o CABEMT trata simultaneamente a geometria e a topologia no mesmo objeto, portanto esses conceitos se confundem internamente ao programa.

Nos itens que serão descritos a seguir, os atributos das classes foram divididos em sendo do tipo objeto e não objeto:

- Objeto: característica ou parâmetro representados por um objeto de alguma classe nativa do CABEMT, do sistema Java ou de alguma biblioteca externa;
- Não objeto: característica ou parâmetro representado por um tipo primitivo de variável, tais como os inteiros, números de ponto flutuante de precisão simples ou dupla, booleanos, caracteres, etc.

As classes das entidades primitivas são derivadas da classe abstrata *geometricObject* que define o escopo genérico de atributos e métodos comuns a todas as entidades primitivas. Essas herdam as características de *geometricObject*, além de possuírem as suas próprias. Esse “aproveitamento” das propriedades é chamado de herança, um conceito utilizado em POO.

Para cada tipo de entidade existe uma lista (*arrayList*) em *Modeler* para seu armazenamento (ex.: lineObj, circleObj, ellipseObj, etc.).

Os pontos chave que definem as entidades primitivas podem ser informados pelo usuário através do *prompt* de comando ou através do mouse, cujas ações são tratadas por *textCommEval* ou *EvalCommButton*, respectivamente.

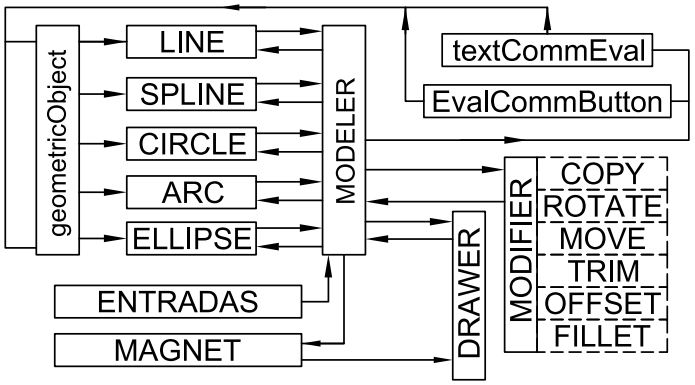
4.6.1 Diagrama de funcionamento

A conexão lógica básica das classes a serem discutidas, relativas à modelagem de primitivas, encontra-se no diagrama da figura 4.7. Observar o papel central de *Modeler* fazendo o gerenciamento do processo.

As funções de *Modifier* operam nas entidades já criadas e, portanto, armazenadas nas listas em *Modeler*. Como elucidado no item 4.5.3, interfaces de *Modeler* fazem a leitura das entradas e o processamento é transferido a *textCommEval* e *EvalCommButton.*

Magnet é invocada constantemente por *Modeler* mediante movimentação do cursor. Ao ativar algum ponto notável envia a *Drawer* os parâmetros necessários para a representação gráfica da entidade auxiliar de traçagem (linhas de construção, ícones de “atração magnética”, etc).

Figura 4.7 – Diagrama básico de funcionamento do CABEMT para representação e modificação de primitivas de modelagem. As caixas com linhas tracejadas são funções da classe M*odifier*



Fonte: Elaboração do próprio autor

4.6.2 Ponto ou vértice

O mais elementar das primitivas, o ponto ou vértice, pode ser representado por objetos das classes *Vertex, Vertice* e *Vertice2D* – mencionadas em ordem decrescente de complexidade. A diferença entre elas está na quantidade de informações carregadas pelo objeto. Por exemplo, certas operações não requerem números de identidade[39] (ID) do elemento, portanto, pode-se utilizar classes cujos objetos possuam menos atributos, a fim de se economizar memória. Analisemos *Vertex* de maneira mais detalhada, uma vez que *Vertice* e *Vertice2D* são apenas classes mais simples baseadas da primeira.

4.6.2.1 Atributos de Vertex: não objetos

- Coordenadas (x,y,z) definindo sua posição;
- Tipo: se é vértice de extremidade, centro, intermediário ou de intersecção. Isso é utilizado por *Magnet* para determinar qual tipo de atrator será exibido na tela, tal como mostrado na figura a seguir.

Figura 4.8 – Tipos de magnetos/atratores do CABEMT. Da esquerda para direita: extremidade, meio, quadrante, centro e intersecção

Fonte: Elaboração do próprio autor

- ID: é um número de identificação unívoco. O último número ID disponível fica em

[39] É um número de identifica univocamente certo objeto.

globalVar, sendo incrementado toda vez de uma entidade é criada;

- ID2: é um identificador auxiliar, utilizado por algumas funções para classificar vértices;
- *S_Vert_ID*: número de identificação exclusivo para identificar vértices quando compõem sólidos. É uma redundância que visa a segurança, segregando essa variável apenas para manipulação de sólidos.

4.6.2.2 Atributos de Vertex: objetos

Não há.

4.6.2.3 Métodos principais de Vertex

- *Add* ou *sub*: faz as operações vetoriais de adicionar ou subtrair vértices, interpretando-os como vetores cujo ponto inicial é sempre a origem;
- *Copy*: copia o vértice atribuindo à cópia um novo ID;
- *Change*: altera as coordenadas do vértice;
- *Scale*: aumenta ou diminui de maneira proporcional as coordenadas do vértice por um fator de escala;
- *calcNorm*: considera Vertex como um vetor e calcula sua normal Euclidiana;
- *dotProd*: calcula o produto escalar contra outro vértice;
- *vecProd*: calcula o produto vetorial contra outro vértice;
- *distVertex*: calcula a distância relativa a outro vértice;
- *evalAngle:* calcula o ângulo formado por dois vetores cujas extremidades são o ponto (0,0,0) e as coordenadas de cada um;
- *localToGlobal:* transformação de coordenadas locais para globais. É necessário especificar um sistema local de coordenadas – entidade da classe *auxAxis*;
- *globalToLocal:* transformação de coordenadas globais para locais. É necessário especificar o sistema local de coordenadas;
- *normalize:* normaliza o vetor formado pelo ponto (0,0,0) e suas coordenadas;
- *rotAroundXYZ:* rotaciona o vetor formado pelo ponto (0,0,0) e suas coordenadas ao redor dos eixos X, ou Y, ou Z;

4.6.3 Linha

É definida em seus pontos extremos por objetos da classe *Vertex*.

4.6.3.1 Atributos de line: não objetos

- *n_pattern*: é um número inteiro que indica o tipo de linha a ser desenhado, tais como linha contínua, tracejada, traço e ponto, etc.;
- ID e ID2: número de identificação unívoco da linha e número de identificação auxiliar;
- *thickness*: número que determina a espessura da linha;

4.6.3.2 Atributos de line: objetos

- V1, V2 e V3 (*vertex)*: vértices das extremidades e do ponto intermediário da linha;
- auxAxis: referência ao eixo de coordenadas associado à linha;

4.6.3.3 Métodos principais de line

- *draw:* envia os comandos *OpenGL* para representação gráfica;
- *drawLineByCommand:* trata comandos do teclado para criação da linha;
- *drawLineByMouse:* trata comando do mouse para criação da linha;
- *copy, move, rotate;* copia, move e rotaciona;
- *offset*: cria uma linha paralela à atual, dados um ponto que define o lado ao qual a cópia será criada e a distância da linha original.

4.6.4 Círculo

A classe *circle* trata apenas de entidades fechadas, se alguma operação abrir o círculo, a entidade é transformada para classe *arc*.

4.6.4.1 Atributos de circle: não objetos

- *radius*: raio do círculo;
- *n:* número de divisões do círculo, ou seja, a quantidade de linhas que aproximarão a forma real do círculo pretendido;
- *ID*: número de identificação;

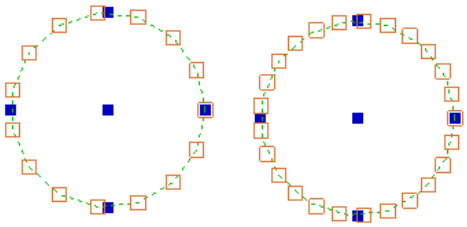
4.6.4.2 Atributos de circle: objetos

- center: objeto da classe *Vertex* que define o centro do círculo;
- *quad_1, quad_2, quad_3, quad_4:* objetos da classe Vertex que definem a posição dos quadrantes;
- *auxAxis:* referência ao eixo de coordenadas associado ao círculo;

4.6.4.3 Métodos principais de circle

- *draw, drawCircleByCommand, drawCircleByMouse, copy, move, rotate, offset: ver funções análogas para linha;*
- *showDiv*: exibe na tela a divisão do círculo, mostrando *n* entidades da classe *box,* de modo que o usuário possa avaliar a razoabilidade da segmentação corrente do círculo como mostrado na figura a seguir;

Figura 4.9 – Ajuste do número de subdivisões de um círculo.



Fonte: Elaboração do próprio autor

- *refreshDiv*: atualiza o número de subdivisões do círculo (n) de acordo com a quantidade especificada pelo usuário;

4.6.5 Arco

No CABEMT, um arco pode ser desenhado por três pontos não coincidentes que pertençam ao mesmo. Outra forma é especificar o centro, o ponto inicial e um ponto final. Este último, entretanto, define apenas o ângulo do arco, logo não precisa coincidir com ele.

4.6.5.1 Atributos de *arc*: não objetos

- *Radius, n* e *ID*: análogos ao círculo;
- *Angle*: ângulo total do arco;
- *StartAngle*: ângulo entre uma linha que vai do centro ao início do arco e o vetor unitário na direção X do sistema de coordenadas associado ao arco;
- *MidAngle* e *endAngle:* análogos a *startAngle*, mas referentes aos pontos intermediário e final do arco, respectivamente;

4.6.5.2 Atributos de *arc*: objetos

- *Center, auxAxis*: análogos ao círculo;
- *Mid*: ponto intermediário do arco. É um objeto da classe *Vertex;*
- End: ponto final do arco – membro de *Vertex*.

4.6.5.3 Métodos principais de *arc*

- *Comp_center_Radius:* calcula a posição do centro e o raio do arco dados o início, fim e meio;
- *EvalAngles:* determina os ângulos inicial, central e final do arco, armazenando-os nos seus atributos correspondentes;
- *ComputeMidPoint:* determina o ponto central do arco, aquele que o divide em dois arcos de comprimentos iguais;
- *DrawArcByCommand, drawArcByMouse:* são análogas aos comandos para o círculo;
- *ShowDiv, refreshDiv, copy, move, rotate, offset*: análogas às funções da classe *circle;*
- *isInsideArc:* verifica se um ponto está dentro da meia-lua formada pelo arco e a união entre suas extremidades. Esse algoritmo é útil para excluir pontos de intersecção que não fazem parte do arco, uma vez que os pontos são calculados para a circunferência e só depois avaliados como pertinentes ou não ao arco.

4.6.6 Elipse

É bastante comum modelar trincas como faces elípticas e de fato, a forma das trincas após se propagarem costuma aproximar-se de uma elipse decorrentes da variação do FIT no contorno de trinca. O CABEMT não dá suporte a elipses parciais (abertas), porém tais formas podem ser alcançadas por meio de operações booleanas. A parametrização das elipses foi realizada conforme a equação (2.5).

4.6.6.1 Atributos de *ellipse*: não objetos

- *Editing*: booleano que indica se a elipse está ou não em modo edição. Se não estiver, um condicionante aciona um algoritmo de desenho rápido;
- *a,b*: comprimento dos semi-eixos;

- *n:* número de subdivisões da elipse;
- *ID:* análogo às outras classes;

4.6.6.2 Atributos de *ellipse*: objetos

- *Center;*
- *Axis_X, axis_Y*: membros de *Vertex* que marcam a posição das extremidades dos semi-eixos da elipse;
- *Axis_XOP, axis_YOP*: membros de *Vertex* que marcam a posição das extremidades opostas aquelas dos atributos do tópico anterior;
- *Vert:* é uma lista (*arrayList*) de pontos utilizados no desenho da elipse;
- *LineTemp:* linha temporária, utilizada para representar os semi-eixos quando a elipse está sendo criada. Tem função primordialmente estética.
- *auxAxis:* análogo às classes anteriores.

4.6.6.3 Métodos principais de *ellipse*

- *Draw, drawEllipseByCommand, drawEllipseByMouse, copy, move, rotate, showDiv, refreshDiv:* análogos às classes anteriores*;*
- *genEllipseLineRepr:* gera os pontos que definem a representação da elipse por meio de uma sucessão de linhas e armazena-os em *Vert para desenho rápido;*

4.6.7 Spline

A maioria dos componentes da indústria e de interesse em análise de tensões possui formas elementares, de tal sorte que as primitivas discutidas nos itens anteriores são capazes de criar o modelo sólido desses componentes. É evidente que isso não é mera obra do acaso, uma vez que se tenta utilizar geometrias mais amenas ao tratamento matemático e mais fáceis de se fabricar. Certos elementos, entretanto, para desempenhar sua função de maneira otimizada, requerem geometrias curvas não tradicionais, tais como pás de ventiladores e compressores, cascos de navios, fuselagens, etc. Outros, possuem geometria complexa para fins estéticos, algo comum nas obras de arquitetura.

Além disso, trincas podem ter uma geometria irregular e pode ser conveniente aproximar o modelo sólido/geométrico a essas características. É por isso que o

CABEMT oferece suporte à geração de *splines* cúbicas, um elemento representável por NURBS, já tratadas no item 2.1.1. Felizmente, o OpenGL possui uma série de funções para representação gráfica de NURBS (JACKIE, DAVIS e DIXON, 1997). Todas as *splines* no CABEMT são geradas via especificação de pontos interpoladores ao invés de pontos de controle.

As *splines* são armazenadas de maneira híbrida: de uma forma representadas como um conjunto de segmentos de reta e da outra com os seus parâmetros caraterísticos (pontos de controle, grau, vetor de *knots*, parâmetros). Para a representação gráfica os parâmetros são passados como argumentos para as funções do OpenGL encarregadas de gerar as NURBS. Por outro lado, quando se efetua uma operação de varredura (extrusão, revolução...) na spline, ela é entendida como um conjunto de segmentos de reta, já que todos os sólidos são poliedros no CABEMT.

4.6.7.1 Atributos de *spline*: não objetos

- *Knot:* lista de *knots* da *spline* – vide item 2.1.1;
- *U_bar*: lista dos parâmetros de discretização da curva;
- *Ctlarray*: lista de pontos de controle da curva que são determinados a partir dos pontos de interpolação fornecido pelo usuário e armazenados em *points*;
- *Degree*: grau da spline – igual a três;
- *Ndiv*: número de subdivisões da *spline* para a representação por segmentos de reta*;*
- ID: identificação;
- *Editing:* análogo ao comando equivalente para a classe *ellipse;*
- *showLineRep:* comuta o modo de exibição da *spline* como NURBS ou como os segmentos de reta. Tem a função de ajudar o usuário a definir o número de divisões da spline antes de usá-la para criação de sólidos;

4.6.7.2 Atributos de *spline*: objetos

- *Points*: pontos de interpolação definidos pelo usuário – entidades da classe *Vertex*;
- *Lines:* o conjunto das linhas que representam a *spline* – entidades da classe *line*;
- *StartTang:* entidade da classe *vertex* que representa o vetor tangente do trecho

inicial da spline;

- *EndTang:* entidade da classe *vertex* que representa o vetor tangente do trecho final da spline;
- *AuxAxis:* conforme já discutido nas outras classes.

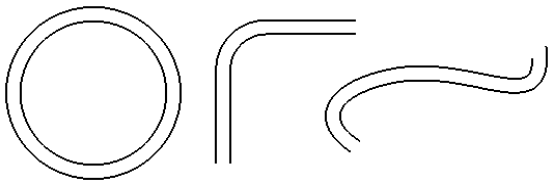
4.6.8 Modificadores de primitivas

Os *softwares* de modelagem ou desenho normalmente apresentam algumas funções auxiliares que modificam, criam cópias, rotacionam ou transladam as primitivas de modelagem. São funções não essenciais, mas que facilitam e agilizam a criação dos modelos. Por isso, implementaram-se algumas delas no CABEMT, cujas descrições encontram-se nos itens subsequentes.

4.6.8.1 *Offset* (cópia paralela)

Copia uma entidade de tal forma que se mapearmos a entidade mãe e os pontos correspondentes na filha, eles estarão a uma distância constante especificada pelo usuário. Na figura a seguir, encontram-se alguns exemplos de *offset*:

Figura 4.10 – Comando offset aplicado a algumas primitivas de modelagem: círculo, linhas + arco e spline.



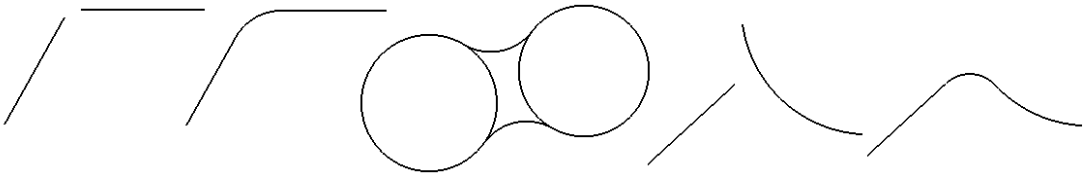
Fonte: Elaboração do próprio autor (CABEMT)

Para cada classe de primitiva existe uma implementação específica do método *offset*, visto que os algoritmos são diferentes entre si.

4.6.8.2 *Fillet* (concordância)

Cria um arco de concordância com raio especificado pelo usuário entre duas retas concorrentes, dois círculos e entre arco e linha. Para cada tipo de concordância há uma função da classe *Modifier* para determinar o arco concordante: *FilletCircle, FilletLines, FilletLineCircle, FilletLineArc.*

Figura 4.11 – Exemplos de concordâncias efetuadas pela classe m*odifier*



Fonte: Elaboração do próprio autor (CABEMT)

4.6.8.3 *Trim* (aparar)

Dadas duas entidades que se interceptam em algum ponto, o método *trim* realiza o corte de uma delas, sob seleção do usuário, no ponto de interseção.

4.6.8.4 *Copy* (copiar)

Simplesmente cria uma cópia da entidade selecionada. O local onde a cópia é posicionada depende de um ponto de referência definido previamente e um novo local para qual esse ponto será movido.

4.6.8.5 *Move* (mover)

Efetua a translação de uma entidade baseando-se num ponto de referência e um novo ponto definido;

4.6.8.6 *Rotate* (rotacionar)

Rotaciona uma entidade tendo como base um ponto de referência e um ângulo definidos pelo usuário.

4.6.9 Auxiliares de traçagem

Não fosse pelas funções auxiliares de traçagem, o usuário precisaria utilizar a entrada manual de coordenadas em demasia, tornando a modelagem muito cansativa. A seguir serão apresentados brevemente os recursos de auxílio à modelagem disponíveis no CABEMT.

- Magnetos: cada entidade possui pontos notáveis onde é comum de se fazer a ligação com outros elementos. Tais pontos ficam armazenados no objeto representativo da entidade como visto nos itens 4.6.3 a 4.6.7. Ao se movimentar o cursor do mouse, a classe *Modeler* fica acompanhando sua posição constantemente comparando-a aos pontos chave calculados por *evalIntersections* da classe *Magnet.* Quando a distância do cursor em relação a um ponto notável torna-se menor ou igual a um valor pré-estabelecido no programa, as coordenadas de clique do mouse serão feitas como idênticas às do ponto notável. Ademais,

gráficos indicativos dessa “magnetização” aparecem na tela, conforme mostrado na figura 4.8;

- Linhas polares: são semi-retas que partem do ponto de trabalho (último ponto ativo), irradiando-se e ocorrendo em ângulos de 90°. Quando o cursor fica próximo à uma linha dessas, sua posição, para todos efeitos, torna-se a do ponto mais próximo pertencente à linha polar;
- Linhas de construção: são linhas polares que sofrem ativação progressiva conforme o cursor “passeia” pelos pontos notáveis das entidades. As interseções dessas linhas também criam pontos notáveis os quais podem ser utilizados na criação do modelo. A figura abaixo mostra a intersecção de duas linhas de construção (tracejadas) permitindo que um retângulo seja criado com rapidez. Esses pontos de intersecção são uma grande ajuda par aa modelagem. No APÊNDICE F encontram-se algumas fórmulas de cálculo para os pontos de intersecção das entidades mais comuns.

Figura 4.12 – Um ponto de intersecção determinado por linhas de construção

Fonte: Elaboração do próprio autor (CABEMT)

Magnetos e linhas de construção são criados pela classe *Magnet,* ao passo que as linhas polares são criadas diretamente por uma função da classe *Drawer.*

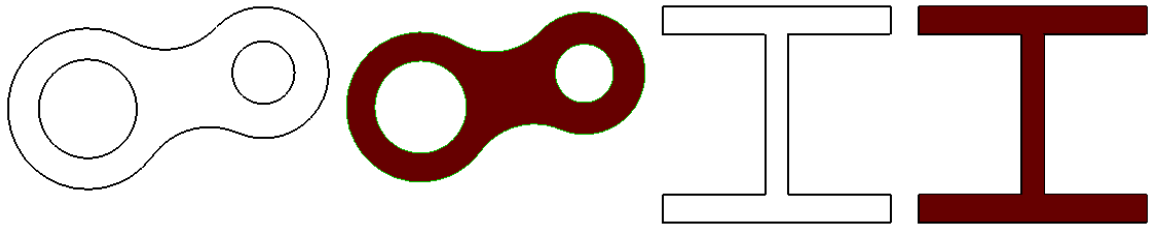
4.7 MODELAGEM DE SÓLIDOS NO CABEMT

Os sólidos são a entidade geométrica final e o objetivo da modelagem no CABEMT, embora as superfícies sejam as entidades de interesse no MEC. O sólido garante que as superfícies estejam ligadas topologicamente, formando um invólucro fechado e que estejam adequadamente orientadas, haja vista que a normal das faces é imprescindível ao MEC.

Em geral, a modelagem começa pelo desenho de um esboço bidimensional através das primitivas de modelagem. Este esboço pode ou não conter furos, mas não deve ter extremidades livres nas entidades que o compõe. O comando *makeFace* quando aplicado ao esboço cria uma face, marcada pela cor vermelha no CABEMT –

denominada “face livre”, por não estar associada a sólidos. Exemplos de esboços e faces podem ser vistos na figura a seguir:

Figura 4.13 – Transformação de esboços constituídos por primitivas de modelagem em faces pelo comando *makeFace*

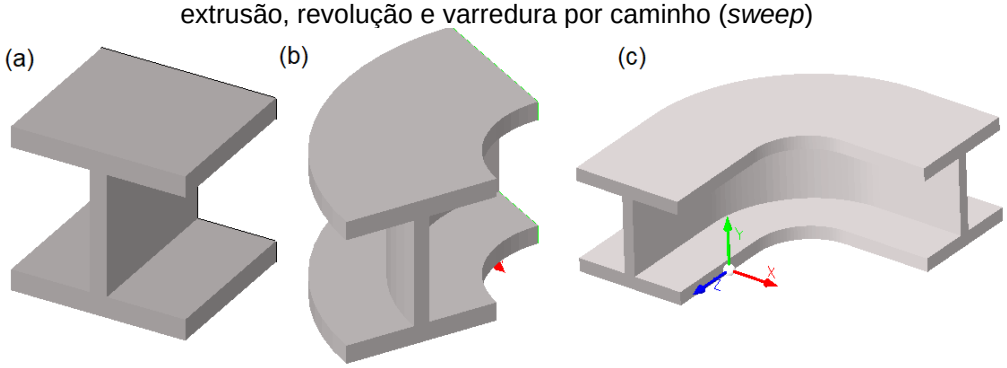


Fonte: Elaboração do próprio autor (CABEMT)

Para representar graficamente as faces, o *OpenGL* internamente faz a sua tesselação, subdividindo-as em triângulos.

Em seguida, as faces criadas podem sofrer alguma operação de varredura que dará origem ao sólido, tal como as mostradas na figura 4.14.

Figura 4.14 – Diferentes operações de varredura sobre uma mesma face. Da esquerda para direita: extrusão, revolução e varredura por caminho (*sweep*)



Fonte: Elaboração do próprio autor (CABEMT)

Esses sólidos são constituídos por um conjunto de *surfaces,* cada uma contendo pelo menos uma face.

As operações booleanas entre sólidos e a ferramenta *slice* (fatiar) estendem sobremaneira a capacidade de geração de modelos tridimensionais a partir de outros mais simples.

4.7.1 Estrutura de dados

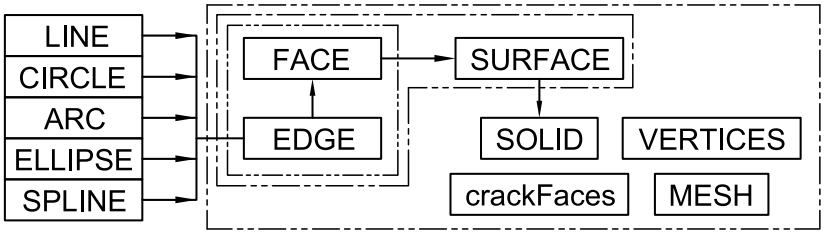
No CABEMT, quando o comando *makeFace* opera num conjunto fechado de primitivas, formando assim uma face – ocorre a transformação dessas primitivas em arestas (edges). Elas correspondem aos segmentos de reta que representavam graficamente as primitivas, ou seja, o comando *makeFace,* ao criar as arestas, gera informações de topologia. Os objetos da classe *face* são um conjunto de arestas

(*edges*) que por sua vez são constituídas por dois pontos, cada qual em uma extremidade.

No momento em que a *face* e suas arestas sofrem alguma operação de varredura, formam-se conjuntos de faces planares, consistindo na aproximação poliédrica do sólido.

Na figura 4.15 observamos um esquema básico dos objetos que se relacionam com a classe *solid.* Por conter os objetos de *surface*, *solid* faz referência indireta às suas faces, que por sua vez referem-se às suas arestas (*edges*). *Solid* também armazena listas de seus vértices, de faces de trinca e também da malha de elementos de contorno. Nos próximos itens, estes elementos serão descritos em detalhes.

Figura 4.15 – Diagrama simplificado da estrutura de dados de um objeto da classe *solid*



Fonte: Elaboração do próprio autor

4.7.1.1 Edge (aresta)

É um elemento de topologia conectando dois vértices subsequentes. As arestas provêm dos segmentos de reta obtidos da representação das primitivas de modelagem, conforme ilustrado na figura 4.15. Cada aresta se relaciona a somente duas faces – uma a seu lado direito e outra ao lado esquerdo.

4.7.1.1.1. Atributos de *edge*: não objetos

- *ElementSize*: tamanho da subdivisão da aresta para a geração de malha;
- *powerLawCoef*: é um coeficiente que deixa as subdivisões da malha não uniformes. Útil para malhas que requeiram refinamento local;
- *NumOfSubDiv*: quantidade de subdivisões da aresta para geração de malha. Esta propriedade e *elementSize* são função uma da outra, desse modo o usuário escolhe uma delas e a outra estará automaticamente definida;
- *Processed*: booleano utilizado para identificar uma aresta que já tenha sido subdividida. Isso é utilizado no algoritmo de geração de malha;

- *Meshed*: booleano responsável por indicar se a aresta já possui malha. Isso será consequência de a malha ter sido gerada em alguma face vizinha ou não;
- *Flip*: variável auxiliar que troca o sentido de redução de espaçamento das subdivisões, caso *powerLawCoef* seja diferente de 1,0;
- *Type*: um caractere responsável por indicar o tipo da aresta, podendo ser n = nenhum, l = linha, c = trinca e outros;
- ID: número de identificação unívoco a aresta;
- *Facet:* quando uma aresta é definida como sendo *facet*, os elementos (da malha) que se conectam a ela serão contínuos. Se *facet* for falso, a aresta marca uma mudança brusca de direção e elementos descontínuos são atribuídos. Exemplos de faces classificadas como *facets* encontram-se na parede lateral do cilindro na figura 4.16 (a). Nesse caso, as faces da parede lateral do cilindro são suaves e, portanto, marcadas como *facets* para as quais não serão associados elementos descontínuos;
- *Select:* indica se a aresta está selecionada pelo usuário ou não;

4.7.1.1.2. Atributos de *edge*: objetos

- *From*: número de identificação (S_Vert_ID) do vértice origem da aresta;
- *To*: número de identificação (S_Vert_ID) do vértice destino/final da aresta;
- *AdjFaces*: lista que armazena as referências para as faces vizinhas à aresta;
- *Vfrom:* referência ao vértice de origem da aresta e que se encontra na lista vertices do objeto *solid;*
- *Vto:* análogo a *Vfrom*, porém referente ao vértice final;
- *Neighbor:* referência à aresta vizinha – idêntica à aresta corrente, porém percorrida no sentido inverso;
- *Surf:* referência ao objeto *surface* que contém a aresta;
- *Face:* referência ao objeto *face* que contém a aresta.

4.7.1.1.3. Métodos de *edge*

- *Copy;*

- *Flip:* inverte a orientação da aresta;
- *ToLine*: cria e retorna um objeto da classe *line* cujas extremidades são os vértices da aresta;
- *GenSubDivPreview:* cria marcações visuais transitórias por meio de objetos da classe *box,* para exibir ao usuário as subdivisões da aresta;
- *IsPointInEdge:* verifica se um ponto coincide com a reta que passa pelos dois extremos da aresta, dada uma tolerância.

4.7.1.2 Face

Formada pela união sequencial de um conjunto de arestas que delimita uma região.

4.7.1.2.1. Atributos de *face*: não objetos

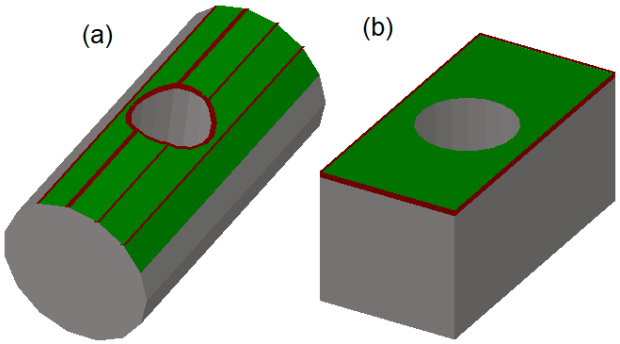
- *Facet:* se verdadeiro, indica que a face está sendo utilizada para representar uma superfície curva. Espera-se então, que as normais entre quaisquer faces nessa condição assumam direções próximas. Esse marcador é uma consequência do CABEMT tratar todas as superfícies como poliedros;
- *IsCrack:* se verdadeiro a face é representa uma trinca;
- *HasCrack:* se verdadeiro, a face contém uma trinca;
- *Area:* área da superfície;
- *ID: número unívoco de identificação;*
- *elementSize: tamanho preferencial que os elementos assumirão na face após a geração de malha;*

4.7.1.2.2. Atributos de *face*: objetos

- *Vertices_face:* entidades da classe *Vertex* que delimitam a face. Estão ligados às arestas;
- *Half_edges:* são as semi-arestas que delimitam a face, armazenadas na ordem "de-para" correta ($from_1 \rightarrow to_1 = from_2 \rightarrow to_2$...). Observar que as *half_edges* são elementos da classe *edge*;
- *Holes:* são furos na face e consistem em agrupamentos de arestas sequenciais delimitando uma região fechada. Existe um agrupamento para cada furo. Na

representação por poliedros, nem toda abertura na superfície de um sólido é classificada como *hole*. Essa denominação ocorre somente para aqueles furos que atravessam a face e não se interceptam com quaisquer de suas arestas conforme mostrado na figura 4.16 (b). No caso do cilindro, entretanto, o furo intersecta as arestas das faces. Nesse caso, as faces são recortadas pelas laterais ou divididas em duas.

Figura 4.16 – Furos e abertura em sólidos - a) Furo em uma superfície composta por facetas; b) em uma superfície composta por apenas uma face. Algumas arestas foram realçadas para melhor visualização



Fonte: Elaboração do próprio autor

- *Nodes*: referências aos nós da malha que pertencem à face;
- *Normal*: objeto da classe *vertex* representando o vetor normal à face. É sempre unitário;
- *Centroid*: posição do centroide da face;
- *AuxAxis*: eixo de coordenadas auxiliar da face. Seu eixo Z aponta na mesma direção e sentido que a normal.

4.7.1.2.3. Métodos de *face*

- *MakeFace:* vasculha as listas de primitivas em *Modeler* procurando por itens selecionados pelo usuário, identifica se tais itens podem formar um conjunto ordenado e fechado de arestas. Caso verdadeiro, cria uma face com essas primitivas;
- *MergeVertices:* une dois vértices muito próximos. Um deles é deletado e todas as referências a ele são substituídas por referência ao vértice que permaneceu;
- *OrderEdges:* tenta ordenar um conjunto e arestas identificando a possibilidade de conexão em suas extremidades;
- *orientFace:* garante que as arestas de uma face estejam orientadas conforme a

sua normal, seguindo a regra da mão direita;

- *Copy;*
- *InvertOrientation:* inverte a orientação da face;
- *CalculateNormalOfOrientedFace:* calcula a normal de uma face baseando-se no sequenciamento de suas arestas, assumindo que o mesmo está correto. Considerando um sistema de coordenadas local na face, a sua normal será o produto vetorial entre as arestas[40] a e $a-1$ que compartilham o vértice inferior esquerdo da face. A normalização desse produto vetorial dá a normal da face;
- *CorrectNormalOfHole:* a normal de um furo deve apontar na mesma direção que a normal da superfície que o contém (convenção do CABEMT), caso contrário, essa função corrige a orientação do furo;
- *InvertHoleOrientation:* é utilizada pela função anterior caso seja necessário inverter a orientação do furo;
- *Draw:* aciona comandos do OpenGL para efetuar a representação gráfica da face;
- *findVert:* encontra um vértice em *vertices_face* através de seu número S_Vert_ID;
- *calcAreaAndCentroid:* calcula a área e o centroide da face através da sua subdivisão em triângulos (tecelagem[41]). Essa função aciona outras da classe face: *nextVertice, groupEdgesSeparateHoles, findOuterLoop, isLoopClosed;*
- *PlaneEquation:* retorna os parâmetros a, b, c e d da equação do plano ($ax+by+cz+d=0$) coincidente com a face.

4.7.1.3 Surface

Surface é uma coleção de faces que representam uma superfície curva ou plana, sendo esta uma convenção do CABEMT.

4.7.1.3.1. Atributos de *surface*: não objetos

- *Select:* análogo ao mesmo atributo de face;

[40] Entende-se que o produto vetorial de duas arestas é na realidade o produto vetorial entre dois vetores unitários, cada qual apontando na mesma direção e sentido que a sua aresta.
[41] A tecelagem é a subdivisão de uma face em elementos mais simples, geralmente triângulos. No CABEMT a tecelagem não cria pontos internos à face, toda a discretização é realizada a partir dos vértices da face

- *ID:* número de identificação;
- *ElementSize:* tamanho do elemento na superfície. Tem prioridade superior ao tamanho dos elementos das faces;
- *Crack:* se verdadeiro, indica que a superfície é uma trinca;
- *Meshed:* se verdadeiro, a superfície tem uma malha associada;

4.7.1.3.2. Atributos de sur*face*: objetos

- *Faces:* é uma lista das faces que compõem a superfície.

4.7.1.3.3. Métodos de *surface*

- *Draw:* faz a representação gráfica;
- *Copy;*
- *Move*.

4.7.1.4 Solid

O CABEMT utiliza o paradigma da representação dos sólidos por fronteiras (B-rep). Estes são um conjunto de objetos da classe *surface* que por sua vez é composta pelas faces e arestas.

4.7.1.4.1. Atributos de *solid*: não objetos

- *Select:* análogo ao atributo homônimo de *surface;*
- *ElementSize:* define o tamanho de elementos nas superfícies de sólido. Predomina em caso de conflito com a propriedade análoga em *face*;
- *Hide:* se verdadeiro, oculta a exibição do sólido na área de trabalho;
- *MaxS_VertID:* armazena o maior valor já atribuído ao identificador S_Vert_ID. É usado apenas em operações de salvar e carregar arquivos. Evita a criação de novas entidades com os mesmos números de *S_Vert_ID* após carregar um arquivo.

4.7.1.4.2. Atributos de *solid*: objetos

- *Surfaces:* uma lista de todas as superfícies que compõem o sólido;
- *CrackFaces:* uma lista de referências às faces de compõem o sólido e que são trincas;

- *Vertices:* uma lista de objetos da classe *vertex* que fazem parte do sólido. São também referenciados pelas superfícies e faces;
- *Mesh:* uma lista de malhas de elementos de contorno associadas ao sólido. Atualmente o CABEMT suporta apenas uma malha por sólido.

4.7.1.4.3. Métodos de *solid*

- *Draw:* itera em todas as faces de cada superfície do sólido, invocando o método *draw de cada uma*;
- *findVertexForSolid*[42]*:* procura um vértice na lista que corresponda a um determinado número de identificação *S_Vert_ID*;
- *verticesCleaner:* itera na lista de *vertices* procurando itens sem associação ao sólido e apagando-os;
- *selectAllSurf:* seleciona todas as superfícies que compõem o sólido;
- *ToFile:* escreve a topologia do sólido e seus componentes em um arquivo de texto com o objetivo de facilitar a elaboração de algoritmos ou resolver problemas de programação. Pode servir, no futuro, para exportação do sólido para outros formatos;
- *mergeVertices:* une dois vértices cuja distância seja menor que o especificado. A função propaga a mudança para todos os elementos de hierarquia inferior: superfícies, faces e arestas;
- *ClassifyEdge:* classifica as arestas como do tipo ‘x’ utilizada em *facets* ou do tipo ‘n’, normal. Essa função é utilizada na importação de malhas. Ela verifica as normais de faces adjacentes, i.e., que compartilham a mesma aresta. Caso o produto escalar das normais dessas faces exceda um limite pré-determinado, pode-se dizer que a transição entre as duas faces é suave e a aresta comum será marcada como ‘x’. Nesse caso, não haverá criação de descontinuidade[43] nos nós dos elementos coincidentes com a aresta;

---

[42] Essa é uma função que tende a ficar obsoleta. Os novos métodos que se relacionam a *solid* guardam referências aos vértices ao invés do número de identificação. Armazenar os números de identificação para depois procurá-los em uma lista quando necessário é ineficiente.

[43] Um elemento descontínuo é aquele em que parte ou todos os nós não são coincidentes com os vértices do elemento. São comumente utilizados no MEC nas regiões em que a superfície não é suave.

- *insertVertex:* insere um vértice na lista *vertices* desde que já não esteja na lista;
- *removeFace:* remove uma face do sólido;
- *removeSurfaceThatHasFace:* remove uma superfície que possua uma face determinada por um código ID fornecido;
- *clearSolid:* itera em todas as listas do sólido, removendo itens sem associação com os demais. Isso pode acontecer após operações booleanas, nas quais vértices que já não mais pertencem ao novo sólido ainda persistem nas listas.

4.7.2 Operações para geração de sólidos

No CABEMT, o único caminho entre as primitivas de modelagem e os sólidos são as operações de varredura, visto que não há funções para inserção direta de sólidos fundamentais (caixas, cilindros, esferas, toro). Essas, entretanto, são dispensáveis, uma vez que as varreduras podem facilmente gerar esses elementos. É bom relembrar que o argumento das funções de varredura são entidades da classe *face*, consistindo de um grupo topologicamente ordenado e fechado de arestas que não se cruzam.

Para cada operação de varredura no CABEMT, existe uma classe correspondente – constituída por métodos estáticos. Essas classes apenas operam sobre as faces, criando objetos da classe sólidos. Nesse sentido elas trabalham de maneira similar à programação estrutural, daí serem constituídas somente por métodos estáticos.

4.7.2.1 Extrusão

Os algoritmos do CABEMT implementam as etapas da instrução 2.2 para extrudar faces. Um exemplo de extrusão realizada pelo CABEMT encontra-se na figura 4.14 (a).

4.7.2.2 Revolução

Implementada conforme a instrução 2.3. Também recebe como argumento um objeto da classe face. Vide figura 4.14 (b) para um exemplo de revolução parcial de um perfil I no CABEMT.

4.7.2.3 Extrusão em caminho (sweep)

A extrusão em caminho é bastante útil para modelagem de tubulações e trilhos.

Os algoritmos foram implementados tomando-se como base a instrução 2.5. Ver figura 4.14 (c) para um perfil extrudado no CABEMT ao longo de um caminho formado por duas retas e um arco.

4.7.2.4 Loft ou varredura ordenada

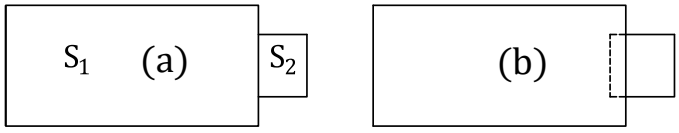
A operação de *loft* ou de varredura ordenada é bastante similar à extrusão, contudo, enquanto nesta a seção transversal do sólido permanece constante, naquela é permitida sua variação em forma e tamanho. Pás de turbomáquinas, asas e certos elementos de caldeiraria apresentam essas características. A figura 2.20 traz exemplos de sólidos gerados pelo operador *loft* no CABEMT.

4.7.3 Operações booleanas

Nem sempre as operações de varredura apresentadas são o suficiente para fazer o modelamento de componentes reais. Por vezes é necessário combinar dois ou mais sólidos para que um modelo satisfatório seja produzido. No contexto da modelagem computacional de sólidos essas combinações são denominadas de operações booleanas, que podem ser de intersecção (AND), adição (OR - AND) ou subtração ($\boldsymbol{S_k}$-AND).

Os algoritmos de operações booleanas, pelo menos do ponto de vista lógico, são complicados, de modo que a realização dessas operações com robustez precisa levar em conta muitas situações e alinhamentos especiais entre os sólidos. Para reduzir um pouco a carga de programação dessa funcionalidade no CABEMT, excluiu-se dos algoritmos a avaliação de tangenciamento entre superfícies e também de operações que resultariam em sólidos com volume igual a zero, ou seja: pontos e arestas. Dessa forma, se o usuário deseja interceptar ou adicionar um sólido a outro, ele deve se assegurar que a intersecção destes sólidos resulte em outro com volume não nulo – conforme mostrado na figura abaixo.

Figura 4.17 – Exemplos de como posicionar dois sólidos para adição ou intersecção no CABEMT. Em (a), vemos um método inadequado. O método preferido (b) vai garantir uma intersecção entre os sólidos



Fonte: Elaboração do próprio autor

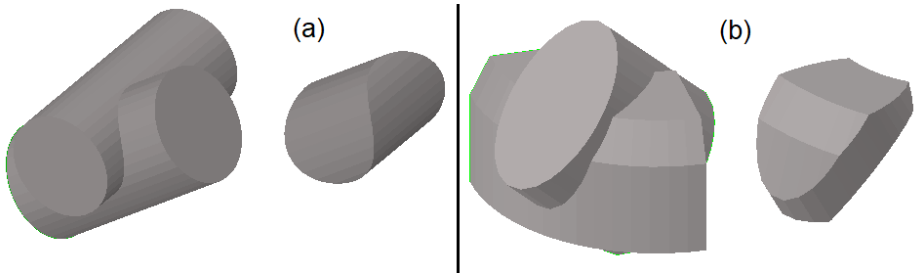
A maioria dos métodos encarregados dessas operações booleanas estão agrupados na classe *boolean_Operators.* Tratam-se de métodos estáticos que

recebem como argumento objetos da classe *solid* e de outras.

4.7.3.1 Intersecção de sólidos

É a operação booleana mais fundamental, já que dela se derivam as outras duas. Os algoritmos que a implementa seguem a instrução 2.6. Vê-se na figura 4.18 que os algoritmos são capazes de calcular intersecções não triviais.

Figura 4.18 – Exemplos de intersecções determinadas pelo CABEMT. Em (a) dá-se a intersecção entre dois cilindros; em (b) entre um elipsóide e uma forma de revolução qualquer, resultando num sólido exótico

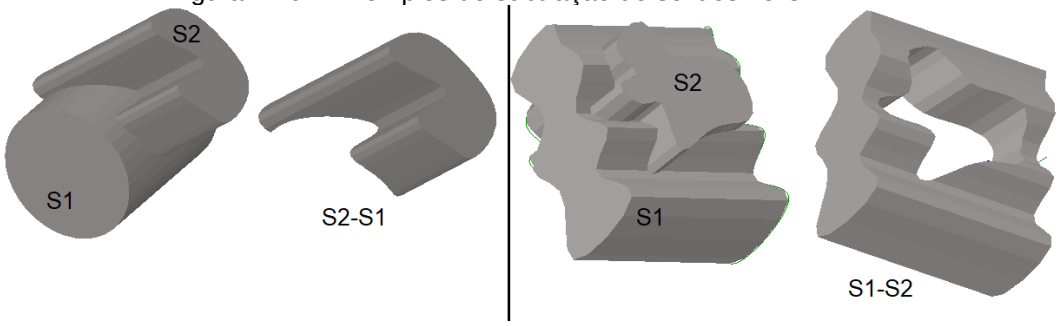


Fonte: Elaboração do próprio autor (CABEMT)

4.7.3.2 Subtração de sólidos

Conforme explicado na instrução 2.7, a intersecção precede a subtração. Em uma operação simbólica de subtração, digamos $\boldsymbol{S_3 = S_2 - S_1}$, as superfícies de $\boldsymbol{S_3}$ serão todas aquelas de $\boldsymbol{S_2}$ que não estiverem completamente no interior de $\boldsymbol{S_1}$ e as superfícies de $\boldsymbol{S_2 \cap S_1}$ que estiverem dentro de $\boldsymbol{S_2}$.

Figura 4.19 – Exemplos de subtração de sólidos no CABEMT



Fonte: Elaboração do próprio autor (CABEMT)

4.7.3.3 Adição ou união de sólidos

Uma vez elaborados os algoritmos de intersecção e subtração; a adição é realizada facilmente. Nessa operação, todas as faces de $\boldsymbol{S_1 - S_2}$ e $\boldsymbol{S_2 - S_1}$ são transportadas para o sólido resultante ($\boldsymbol{S_3}$), exceto aquelas comuns a $\boldsymbol{S_1 \cap S_2}$, conforme estabelecido na instrução 2.8. Não serão mostradas figuras de adição de sólidos, pois graficamente, elas são indistinguíveis da sua justaposição.

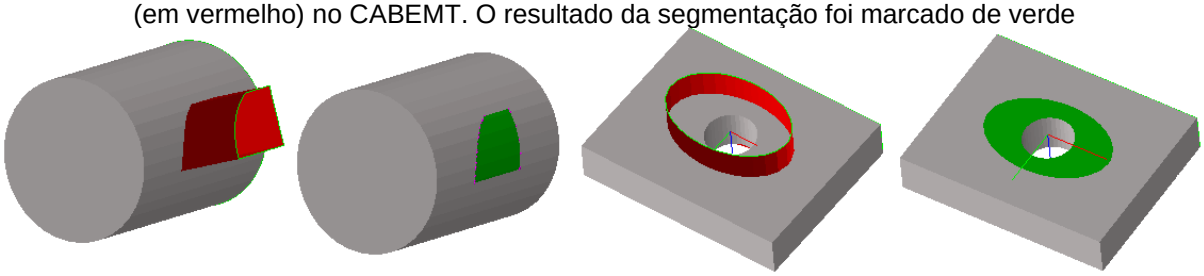
4.7.4 Modificadores de geometria sólida

As operações de varredura e booleanas conferem uma boa flexibilidade na modelagem de sólidos. Todavia, algumas estruturas possuem detalhes difíceis, cuja modelagem apenas com as ferramentas apresentadas, seria – se possível, de grande morosidade. Então é necessário lançar mão de modificadores especiais de geometria e topologia, a serem discutidos a seguir.

4.7.4.1 Divisão/segmentação de uma superfície utilizando outra

Às vezes é desejável segmentar uma superfície para fazer um refinamento local na malha ou definir uma região de carregamento mecânico. No CABEMT é possível efetuar a segmentação de uma superfície de um sólido utilizando como ferramenta de corte uma superfície livre, conforme figura 4.20.

Figura 4.20 – Exemplos de segmentação da superfície de um sólido a partir de uma superfície livre (em vermelho) no CABEMT. O resultado da segmentação foi marcado de verde



Fonte: Elaboração do próprio autor (CABEMT)

O processo de segmentação pode ser realizado por meio da implementação de algoritmos que sigam a instrução a seguir.

Instrução 4.1 – Efetuar segmentação de superfície

Dados: uma superfície $\boldsymbol{Su_1}$ pertencente a um sólido $\boldsymbol{S_1}$ e a superfície de corte $\boldsymbol{Su_c}$ que deverá ser livre (não pertencer a nenhum sólido) e obrigatoriamente interceptar-se com $\boldsymbol{Su_1}$.

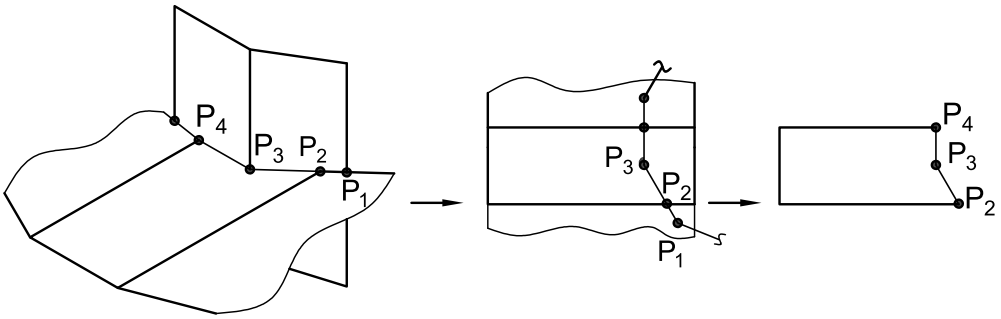
a) Efetuar a intersecção entre todas as faces de $\boldsymbol{Su_1}$ e $\boldsymbol{Su_c}$, obtendo assim as equações das retas de intersecção e as faces associadas a cada reta;

b) Para cada reta do passo a), efetuar a sua intersecção com as arestas das faces associadas, também coletadas no passo a). A cada ponto de intersecção será associado um vértice, conforme mostrado na figura 4.21;

c) Os vértices calculados no passo anterior serão utilizados para subdividir as

arestas. Dessa forma, se antes uma aresta era determinada pelos pontos $P_i \rightarrow P_j$, se agora existir um ponto P_k entre eles – duas arestas serão originadas: $P_i \rightarrow P_k$ e $P_k \rightarrow P_j$. Um vértice pertencerá a uma aresta se a distância entre os dois é inferior a uma tolerância pequena (colinearidade) e a soma das distâncias entre o vértice e cada uma das extremidades é igual (dentro de uma tolerância pequena) ao comprimento da aresta;

Figura 4.21 – Geração de vértices durante a intersecção entre duas superfícies

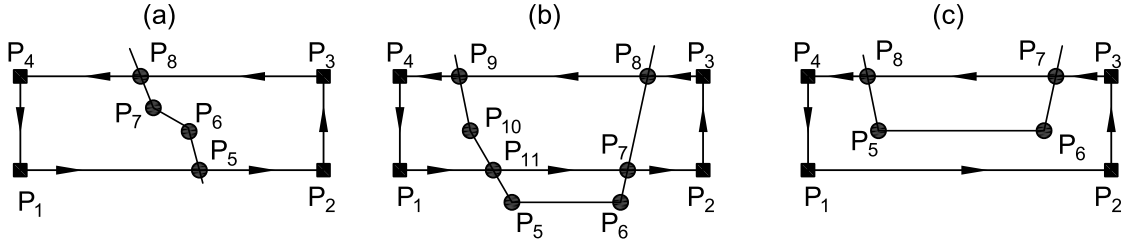


Fonte: Elaboração do próprio autor

d) Estando as arestas segmentadas é necessário avaliar as faces. O algoritmo deve levar em conta diferentes possibilidades de segmentação conforme ilustrado na figura 4.22. Os pontos marcados com um quadrado (ancestrais) correspondem aqueles que já faziam parte da face enquanto os pontos circulares foram obtidos nos passos anteriores, via segmentação das arestas. A iteração pode começar por qualquer um dos pontos quadrados.

No caso da figura 4.22 (a) se iniciássemos pelo ponto P_1 – ao chegar em P_5 será possível seguir para P_2 ou P_6. Deve-se dar prioridade aos novos pontos, segue-se então para P_6. Quando chegar em P_8 haverá novamente duas possiblidades: P_3 e P_4. No entanto, não se pode violar o sentido de orientação de uma aresta já existente na face, logo, segue-se para P_4. Daí, nota-se que o caminho se fecha ao chegar em P_1 – o ponto de partida. Formou-se então uma face.

Figura 4.22 – Segmentações de arestas e faces



Fonte: Elaboração do próprio autor

Se houver mais pontos ancestrais a processar, repetem-se as etapas anteriores. No caso da figura 4.22 (a), ainda restaram P_2 e P_3. Escolhendo qualquer um deles é possível percorrer as arestas até fechar a outra face.

e) Na situação da figura 4.22 (b) os passos anteriores produzirão duas faces, restando ainda criar a face central. Quando não existir mais pontos ancestrais para analisar, o algoritmo deve verificar se ainda restou alguma aresta antiga sem processamento, isto é, sem alocação em alguma face nova. Pela figura, observamos que restariam as arestas ($P_{11} \rightarrow P_7$) e ($P_8 \rightarrow P_9$). Seleciona-se qualquer um dos vértices dessas arestas para início. O algoritmo agora só poderá seguir para vértices novos. Se sairmos de P_8, seguiremos para P_9, depois P_{10}, P_{11}, P_7 e finalmente P_8, fechando a face. Observe que jamais poderemos ir para P_5 ou P_6, visto que esses pontos não estão associados à face correntemente processada. Casos como o da figura 4.22 (c) são processados da mesma forma que em (b). Observe que o CABEMT não aceita a segmentação de uma face em quatro ou mais partes;

f) Toda face que sofreu segmentação deverá ser deletada. E as novas faces devem ser adicionadas à superfície apropriada do sólido;

g) Quando a segmentação ocorre no entorno de um furo (*hole*), como mostrado na figura 4.20 (à direita), serão originadas duas faces: uma fica dentro do invólucro formado pela superfície de corte e a outra é a face antiga recortada pela intersecção com a superfície de corte. É necessário verificar se todos os vértices do furo estão em uma ou outra face para alocá-lo corretamente. O CABEMT não suporta cortes em furos, somente no entorno deles.

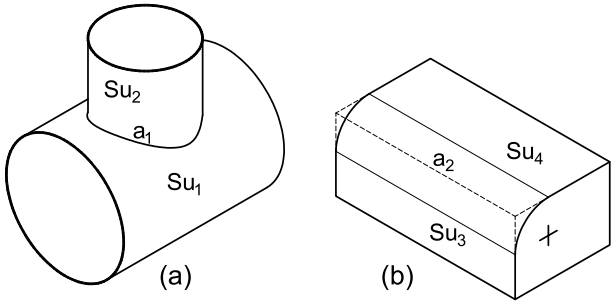
4.7.4.2 Concordância (*fillet*) entre superfícies

Com frequência, as peças produzidas na indústria metal-mecânica possuem arredondamentos em suas bordas de modo a evitar concentradores de tensões – potenciais nucleadores de trincas. É então importante que o CABEMT possua a facilidade de criar uma superfície de concordância na intersecção de duas outras mediante a especificação de um raio. Essa operação, muito similar ao caso unidimensional, recebe o nome de *fillet* (chamemos de concordância) e é um tipo de operação de Minkowski (ROSSIGNAC; REQUICHA, 1999).

A metodologia explicada aqui foi desenvolvida para atender à necessidade

específica no CABEMT, isto é, ser capaz de suavizar contornos na intersecção ente duas superfícies curvas. Os algoritmos elaborados aceitam aplicar a concordância somente entre superfícies cujas arestas em comum (no ponto de intersecção) conectam-se somente às próprias superfícies. Não foi desenvolvido um algoritmo geral de Minkowski conforme mostrado, porém sem muitos detalhes, por Varadhan e Manocha (2006). Essa peculiaridade é ilustrada na figura a seguir:

Figura 4.23 – Situações típicas de aplicação do comando *fillet* no CABEMT: (a) permitido; (b) não permitido

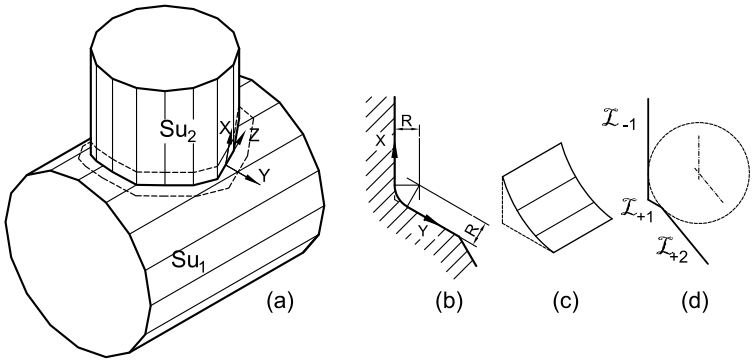


Fonte: Elaboração do próprio autor

Na figura 4.23 (a), a aresta a_1 conecta-se somente às superfícies Su_1 e Su_2 que são, portanto, candidatas ao comando *fillet.* A situação em (b) é diferente, uma vez que a aresta a_2 além de se conectar a Su_3 e Su_4, também é ligada a outras superfícies do sólido. Nessa situação, Su_3 e Su_4 não são candidatas ao comando *fillet.* Note que, entretanto, a superfície curva nesse caso poderia ser facilmente realizada por extrusão.

Na figura 4.24 observamos que o estabelecimento de sistemas de coordenadas locais é necessário para traçar a concordância. À cada extremidade de aresta da interseção entre as superfícies $\boldsymbol{Su_1}$ e $\boldsymbol{Su_2}$ é determinado um sistema auxiliar conforme mostrado na figura.

Figura 4.24 – Concordância entre duas superfícies: a) posicionamento dos sistemas de coordenadas locais; b) detalhe em corte; c) aproximação poliédrica da concordância; d) quando uma linha é muito curta



Fonte: Elaboração do próprio autor

Podemos agora sintetizar o processo de concordância na seguinte instrução:

Instrução 4.2 – Efetuar a concordância entre duas superfícies

Dados: duas superfícies que se interceptam, ambas pertencentes a um único sólido. Necessário também especificar o raio de concordância.

a) Dadas as duas superfícies ($\boldsymbol{Su_1}$ e $\boldsymbol{Su_2}$) para as quais se deseja efetuar a concordância – tomar as arestas comuns a ambas que serão aquelas formadas pela sua intersecção;

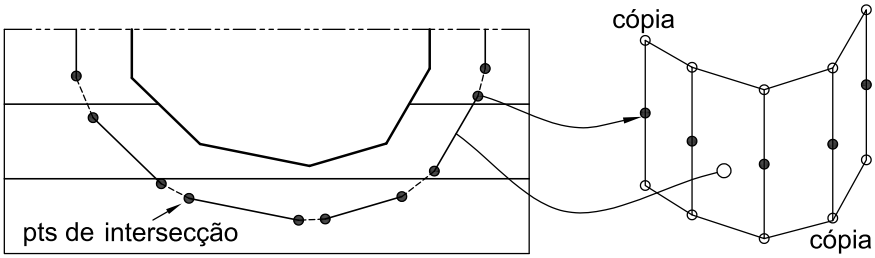
b) Para cada extremidade de aresta do passo a), estabelecer um sistema de coordenada local. O eixo Z deve ser paralelo ao vetor diretor da aresta ao passo que o eixo X deve estar alinhado com a normal da face de $\boldsymbol{Su_1}$ ou $\boldsymbol{Su_2}$ que possua a aresta em comum;

c) Para cada sistema (XYZ), efetuar a intersecção de seu plano XY com $\boldsymbol{Su_1}$ e $\boldsymbol{Su_2}$, obtendo assim as linhas de corte $\mathcal{L}_{-1}, \mathcal{L}_{+1}, \mathcal{L}_{+2}$ ... (uma em cada faceta) tal como ilustrado na figura 4.24 (d). Por segurança, fazer com que o corte passe por pelo menos três facetas de cada superfície se isso for possível (linhas $\mathcal{L}_{-1}, \mathcal{L}_{-2}, \mathcal{L}_{-3}, \mathcal{L}_{+1}, \mathcal{L}_{+2}, \mathcal{L}_{+3}$);

d) Com base nas linhas encontradas no passo anterior e o raio, determinar a posição dos pontos de concordância nas faces de $\boldsymbol{Su_1}$ e $\boldsymbol{Su_2}$, os centros dos arcos e os pontos intermediários dos arcos;

e) Formar as superfícies de corte cuja intersecção com $\boldsymbol{Su_1}$ e $\boldsymbol{Su_2}$ é representada pelas linhas tracejadas da figura 4.24 (a). A construção dessas superfícies pode ser vista na figura 4.25 e figura 4.26 (a). Elas são formadas pela união de vértices originados dos pontos de intersecção. Eles são copiados na direção da normal da face na qual se encontram – perceber esse fato na figura 4.26. A união desses vértices copiados origina as superfícies de corte;

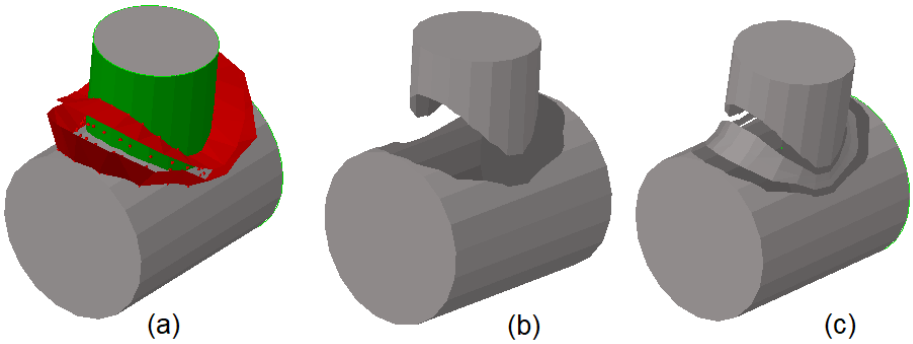
Figura 4.25 – Detalhamento da geração da superfície de corte a partir dos pontos de concordância



Fonte: Elaboração do próprio autor

As faces vermelhas da figura a seguir foram extraídas diretamente do CABEMT para ilustrar como são geradas as superfícies de corte.

Figura 4.26 – Operações intermediárias da função *fillet*: a) superfícies de corte em representação tridimensional – marcadas de vermelho; b) situação após corte e eliminação de faces; c) após criação das faces intermediárias



Fonte: Elaboração do próprio autor (CABEMT)

f) Segmentar as superfícies de $\boldsymbol{Su_1}$ e $\boldsymbol{Su_2}$ utilizando as superfícies de corte, conforme item 4.7.4.1;

g) Apagar todas as faces que possuem as arestas identificadas no passo a). Dessa forma ficará um vazio na região de concordância, como na figura 4.26 (b);

h) Selecionar todos os vértices das arestas solitárias de $\boldsymbol{Su_1}$ e $\boldsymbol{Su_2}$, agrupando-os conforme a sua superfície de origem em listas. Uma aresta é denominada solitária se ela não possui uma adjacente percorrida em direção inversa. Naturalmente as únicas arestas solitárias serão aquelas onde se efetuou o corte, pois ela terá apenas uma face vizinha;

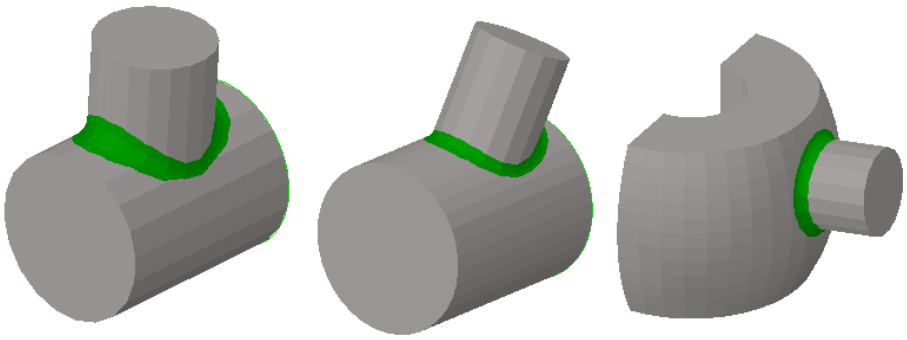
i) Para os vértices das arestas solitárias de $\boldsymbol{Su_1}$, por exemplo, determinar os seus correspondentes em $\boldsymbol{Su_2}$ iterando pelas duas listas estabelecidas no passo anterior. Um vértice V_j da lista L_1 é correspondente a um vértice V_k da lista L_2 se não existir nenhum outro vértice para lista L_2 mais próximo de V_j que V_k;

j) Baseando-se nas informações do passo d) é possível determinar as faces intermediárias da concordância que são mostradas na figura 4.26 (c). A geometria é melhor detalhada na figura 4.24 (c);

k) Por fim, é feita a tecelagem entre as faces intermediárias e as arestas solitárias para o fechamento do sólido. Busca-se novamente minimizar as distâncias entre pontos correspondentes; fazendo-se, em seguida, a conexão entre arestas de modo a não ocorrer cruzamentos.

As figuras abaixo ilustram algumas concordâncias geradas no CABEMT:

Figura 4.27 – Alguns exemplos de concordância efetuados pelo CABEMT. A superfície gerada na concordância foi marcada de verde para destaque



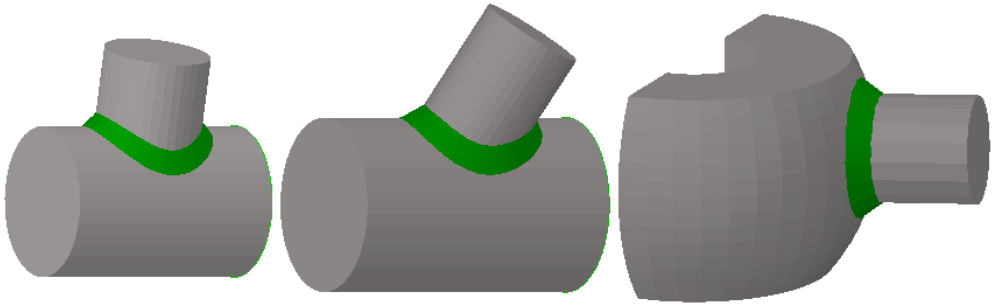
Fonte: Elaboração do próprio autor

4.7.4.3 Aplicação de chanfro entre superfícies

Os chanfros são frequentemente utilizados para modelar a geometria de soldas de juntas em ângulo. Por causa disso, a capacidade de efetuar chanfros entre superfícies é importante para os modelos sólidos de engenharia.

A função chanfro utiliza basicamente as mesmas funções que *fillet*, contudo, ao invés de dispor os pontos seguindo uma circunferência (os arcos), o faz em linhas retas.

Figura 4.28 – Exemplos de geração de chanfros entre duas superfícies no CABEMT

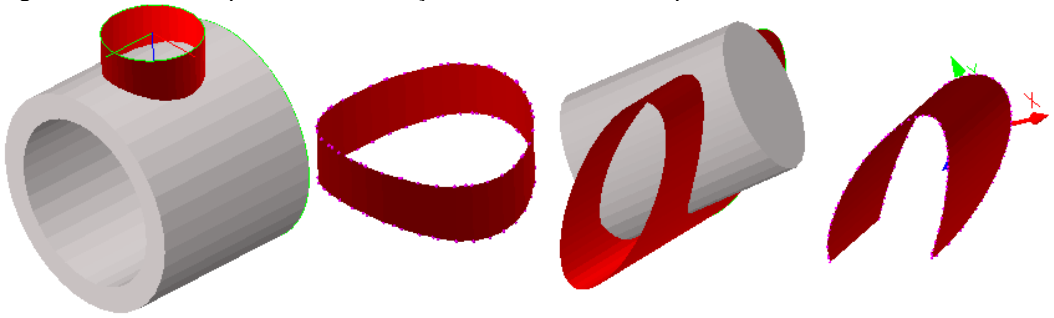


Fonte: Elaboração do próprio autor

4.7.4.5 Intersecção entre sólido e superfície

Algumas superfícies, de formato bastante distinto, só podem ser obtidas por meio da intersecção entre um sólido de trabalho e outra superfície – ou através da criação e conexão manual de cada vértice e aresta. A primeira opção é sem dúvida a mais desejável em termos de agilidade. Entidades desse tipo podem ser utilizadas na modelagem de defeitos planares de fabricação em peças curvas que se conectam por solda. Tais defeitos tendem a acompanhar a geometria do bisel e por isso possuem formatos especiais. Alguns exemplos de superfícies geradas por intersecção superfície-sólido são mostrados na figura 4.29.

Figura 4.29 – Exemplos de intersecção entre sólidos e superfícies realizados no CABEMT



Fonte: Elaboração do próprio autor

O funcionamento desse método é muito similar ao de segmentação de superfícies mostrado no item 4.7.4.1, de tal modo que toma emprestado algumas funções daquele método. A diferença básica é que somente as faces das superfícies livres (em vermelho) e nas quais todos os nós foram gerados na intersecção; restarão no modelo.

4.7.5 Inclusão de trincas no sólido

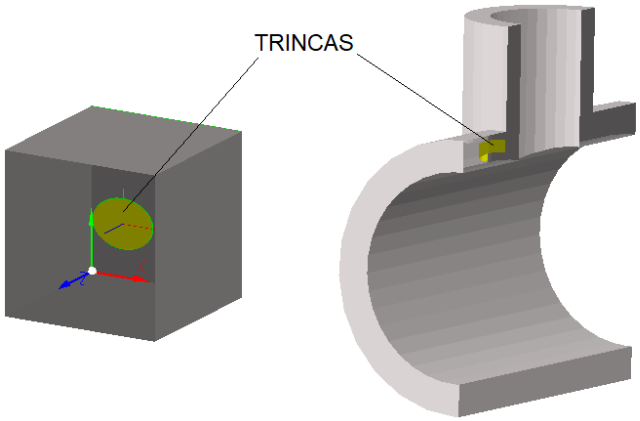
A maneira mais direta de se gerar uma trinca no CABEMT é por meio da face que a representa tal como ilustrado na figura 4.30. Se as trincas estiverem totalmente no interior do sólido, a operação é trivial. Basta transferir a superfície livre para a lista de superfícies do objeto da classe *solid* e fazer referência a essa superfície na lista *crackFaces,* também pertencente a um objeto de *solid.* Ainda será necessário marcar o atributo *isCrack* dessas faces. A figura 4.30 mostra dois exemplos de trinca interna. No CABEMT todas as faces de trinca são identificadas com a cor amarela para facilitar a distinção entre outros tipos de faces. Sua geração é feita pelas classes *crackGeometry* e *crackGeometry2*.

Quando a trinca aflora na superfície do sólido, as intersecções entre as

superfícies exigem um tratamento especial. Foram implementadas duas estratégias para lidar com esse problema: uma para trincas planares e a outras para trincas curvas. Discutiremos primeiramente o caso de trincas planares que se baseia na subtração de um sólido com o formato da trinca. As etapas de criação são as seguintes:

- A face planar geradora da trinca é posicionada no local de interesse – figura 4.31 (a);
- O algoritmo extruda a face a uma pequena distância, gerando um sólido – figura 4.31 (b);

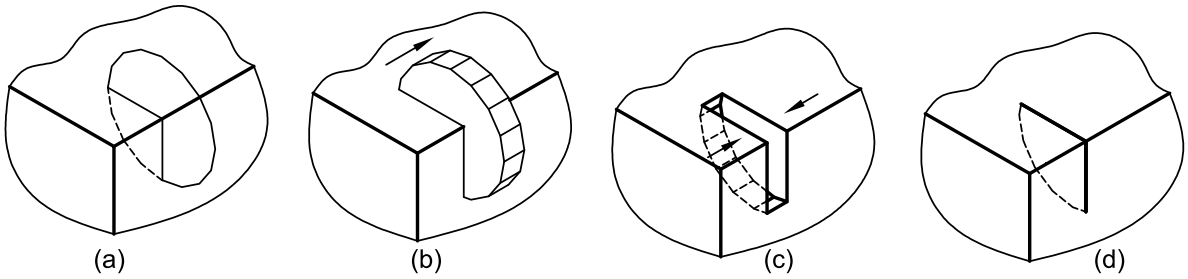
Figura 4.30 – Exemplos de trincas internas criadas no CABEMT. À esquerda, trinca elíptica interna em um tubo. À direita, trinca do tipo falta de fusão na junção entre dois tubos. Algumas faces foram removidas para permitir a visualização.



Fonte: Elaboração do próprio autor

- É feita a subtração entre o sólido principal e o gerador de trinca, formando uma cavidade cuja topologia foi corretamente determinada pelo algoritmo de operações booleanas – figura 4.31 (c);
- As faces opostas da cavidade são aproximadas até se tocarem e uma delas é excluída. As faces laterais, agora com área igual a zero, são também excluídas – figura 4.31 (d);

Figura 4.31 – Etapas de criação de trincas pelo método de subtração



Fonte: Elaboração do próprio autor

De início, o CABEMT suportava apenas a geração de trincas internas e superficiais planares, uma vez que a função de extrudar não opera em faces não-planares. Contudo, é relativamente comum encontrar trincas não planares quando estas começam a assumir comprimentos mais elevados. Por isso, foi implementado um novo método de geração de trinca, ocupando a classe *crackGeometry2* em razão de se desejar manter inalterado o algoritmo da subtração que já havia sido testado e apresentava bom funcionamento.

Esse método, doravante denominado método do corte, pode ser realizado conforme a seguinte instrução:

Instrução 4.3 – Gerar uma trinca a partir de superfície curva

Dados: um sólido $\boldsymbol{S}$ no qual a intersecção com uma superfície curva $\boldsymbol{S_u}$ formará uma trinca.

a) Para cada par de faces (uma de $\boldsymbol{S}$, outra de $\boldsymbol{S_u}$ – vide figura 4.32) determinar os parâmetros que definem a reta de intersecção entre o par. Cada face processada ou criada a seguir deve ser armazenada em uma lista (L_F);

b) De cada reta formada, determinar os pontos de intersecção com as arestas das faces que a originaram. O ponto de intersecção é selecionado se estiver dentro de ambas as faces ou pelo menos coincidente com as arestas delas;

c) A partir do conjunto de pontos selecionados (vide figura 4.32 (b)) e da relação de cada um com suas faces de origem, é possível determinar a conexão entre eles. Desse modo, arestas entre esses pontos de intersecção são determinadas;

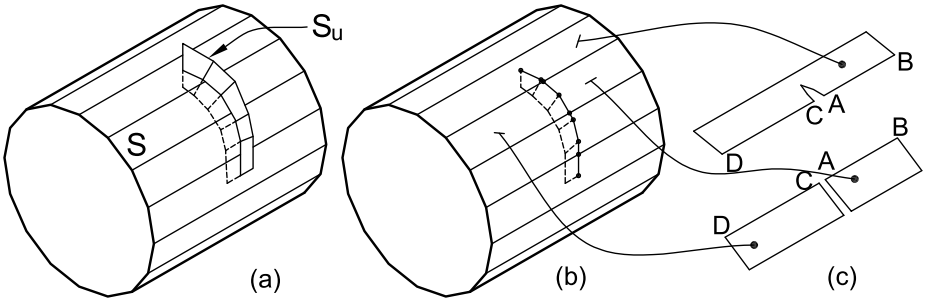
d) As arestas de $\boldsymbol{S}$ e $\boldsymbol{S_u}$ devem incorporar os pontos de intersecção. Por exemplo, se uma aresta era dada por $V_1 \rightarrow V_2$, na ocorrência de um ponto de intersecção (V_i), ela será substituída por duas arestas: $V_1 \rightarrow V_i$ e $V_i \rightarrow V_2$. Em suma, estamos

dividindo a face devido à presença da trinca;

e) Quando uma face de $\boldsymbol{S}$ possui apenas um ponto de intersecção em uma e suas arestas, ela sofrerá um corte – vide face superior da figura 4.32 (c). Quando a face possui dois pontos de intersecção distintos em suas arestas, ela será dividida, como é o caso da face inferior da figura 4.32 (c);

f) Para as faces de $\boldsymbol{S}$ a serem recortadas, conectar as arestas de intersecção (determinadas no passo b) ao vértice de intersecção presente na aresta da face. O algoritmo deve buscar todas as arestas de intersecção relacionadas à face que está sendo processada. Repare que isso pode ser realizado facilmente se no passo b) utilizarmos uma classe, digamos *intersectPoints*, que tem como atributos: faces participantes do processo de intersecção, vértice gerado no processo de intersecção e novo vértice "espelho". Dessa forma é mantido um histórico de quais foram os elementos geradores dos pontos de intersecção;

Figura 4.32 – Geração de uma trinca a partir de uma superfície curva



Fonte: Elaboração do próprio autor

g) Duplicar os vértices de intersecção, conforme os vértices identificados como A e C na figura 4.32 (c). Diz-se que A tem o vértice espelhado C, cada um no seu lado da trinca. Antes de se fazer a duplicação, é necessário verificar se o vértice na face vizinha já passou por esse processo, ou seja, se ele já possui um vértice "espelho". Se assim for, não é necessário gerar um novo;

h) De modo similar às faces recortadas; para as divididas, deve-se primeiramente determinar as arestas formadas pelos pontos de intersecção associados à face. Isso também requer uma procura nas faces vizinhas, de modo a verificar se vértices "espelho" já foram criados anteriormente;

i) Dado o conjunto de arestas (C_A) da face que sofreu corte, selecionar uma qualquer e avançar até topar com um ponto de intersecção ou um espelho,

após isso recuar até ocorrer o mesmo tipo de encontro. Pegar essas arestas percorridas nessa “caminhada” e segregar em um conjunto (C_{A1});

j) Determinar $C_{A2} = C_A - C_{A1}$;

k) Fechar a face formada por C_{A1} com base nas arestas calculadas no passo h) – a nova face será denominada F';

l) Fechar a face formada por C_{A2} com base nas arestas calculadas no passo h) – a nova face será denominada F'';

m) Criar vértices “espelho” para aqueles que não possuem nenhum;

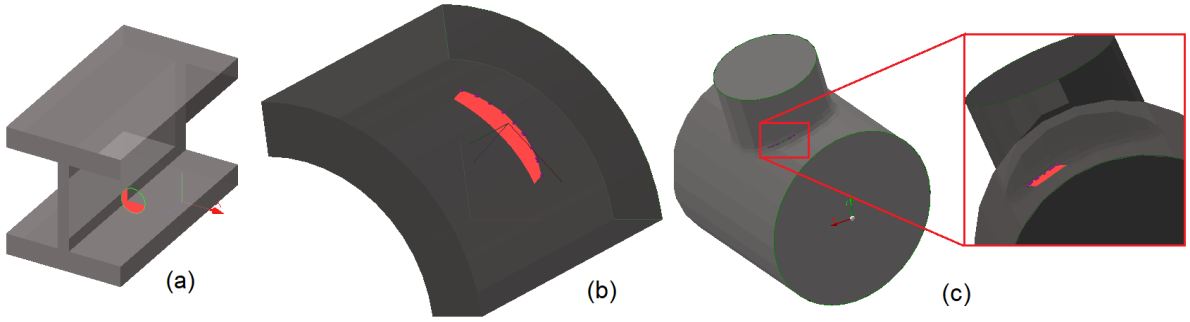
n) Verificar se os pontos de intersecção nas arestas de F' devem realmente pertencer a esta face ou a F'' através da verificação da conectividade com as faces vizinhas que já tenham sido processadas, ou seja, que estão em L_F. Supondo que a face correntemente processada seja a superior da figura 4.32 (c), por exemplo – se a face inferior já tiver sido processada, a superior obrigatoriamente deverá ter os vértices na disposição mostrada. Porém, se a face inferir não tiver sido processada, A e C poderiam estar permutados e a face inferior é que deveria acompanhar isso quando vir a ser tratada;

o) Terminado o processamento das faces de $\boldsymbol{S}$, procede-se às faces de $\boldsymbol{S_u}$. Isso é mais fácil, porque apesar das faces também precisarem de sofrer divisão, é sabido imediatamente que um lado da divisão, aquele que contém vértices no exterior de $\boldsymbol{S}$, não é válido. Aproveita-se então, somente as arestas que não possuem vértices no exterior de $\boldsymbol{S}$ e conectam-se as extremidades soltas baseando-se nas arestas de intersecção do passo b).

É prudente observar que a divisão das faces para a geração de uma trinca é diferente do processo de divisão/segmentação mostrado na figura 4.22. Nesta, as faces compartilham o mesmo nó na divisão, ao passo que para a geração de trinca, cada face ficará associada a um grupo de nós particular. Isso ocorre para que a superfície possa se abrir conforme a trinca é solicitada mecanicamente.

Os algoritmos implementados conforme essas instruções se mostram bastante flexíveis na geração de trincas no modelo sólido, conforme se observa na figura 4.33.

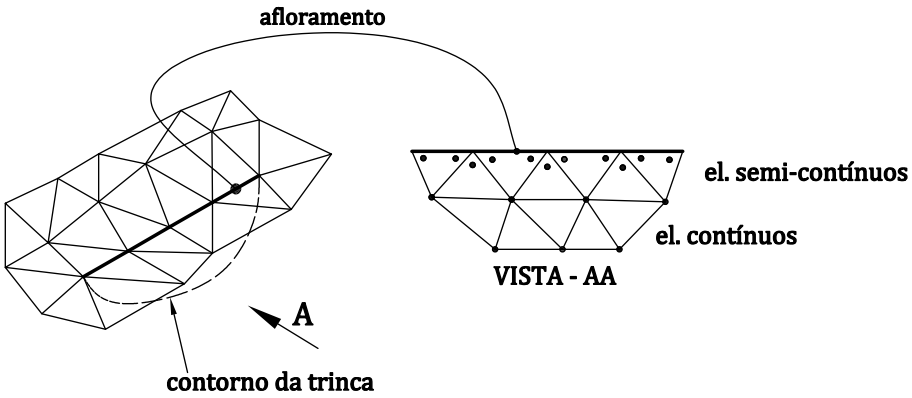
Figura 4.33 – Trincas inseridas no modelo sólido do CABEMT (marcadas em vermelho): a) trinca transversal em um perfil I, b) trinca curva em uma tubulação ou painel, c) trinca na solda de uma derivação T. A cor original da trinca (amarelo no CABEMT) foi modificada para facilitar a visualização.



Fonte: Elaboração do próprio autor

A topologia das arestas da trinca não precisa ser necessariamente consistente com a topologia da superfície na qual ela aflora. Isso ocorre porque os elementos na face da trinca serão obrigatoriamente descontínuos, portanto, nunca se conectam diretamente às arestas no afloramento da trinca. De fato, poderíamos criar um defeito apenas gerando elementos correspondentes à sua face e manipulando os nós dos elementos no ponto de afloramento – de tal forma que esses sejam semi-descontínuos ou descontínuos na conexão com o afloramento da trinca, conforme figura a seguir:

Figura 4.34 – Afloramento de uma trinca coincidindo com as arestas dos elementos na superfície



Fonte: Elaboração do próprio autor

Essa observação permite criar trincas em locais difíceis (em linhas curvas) que poderiam complicar as operações booleanas utilizadas nas instruções anteriores. Esse método foi implementado no CABEMT por um algoritmo baseado na seguinte instrução:

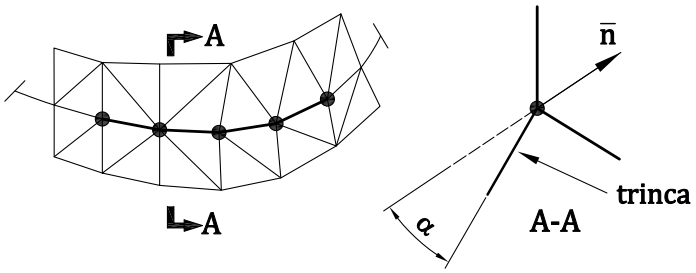
Instrução 4.4 Gerar uma trinca a partir de uma malha existente

Dados de entrada: arestas que marcam o afloramento da trinca, profundidade da trinca, forma da superfície de trinca (retangular ou semielíptica), tamanho dos elementos na face da trinca.

a) A partir de uma seleção contínua de arestas[44] informadas pelo usuário, determinar o seu correto sequenciamento através da conectividade entre elas, explorando os vértices terminais. Se a sequência não for contínua, interromper e avisar o usuário; senão, seguir;

b) Calcular, para cada nó, a média das normais entre os elementos vizinhos. Esse vetor médio ($\bar{\boldsymbol{n}}$) será paralelo ao plano da trinca. Ele poderá ser rotacionado em um ângulo α especificado. Detalhes podem ser vistos na figura a seguir:

Figura 4.35 – Geração de uma trinca a partir de uma superfície curva



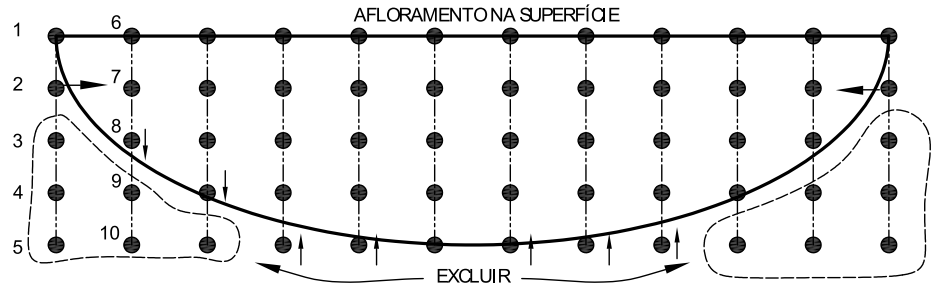
Fonte: Elaboração do próprio autor

c) Para cada nó nas arestas selecionadas, criar novos na direção ($-\bar{n}$), com espaçamento igual ao tamanho especificado dos elementos na superfície de trinca. A criação dos novos nós deve respeitar o formato da trinca (elíptico ou retangular). Dessa forma, o último nó em cada posição deverá obrigatoriamente coincidir com a borda da trinca, nem que para isso seja necessário deslocá-lo. Na figura 4.36 podemos ver os nós que serão excluídos do processo. Aqueles próximos à borda (por exemplo o nó 8) serão deslocados até ela. Experimentalmente verifica-se que o nó na posição 2 (e o análogo no lado oposto) deve ser deslocado no sentido horizontal ao passo que os outros devem ser deslocados no sentido vertical.

d) Formar elementos por meio da adequada união dos novos nós. Isso pode ser feito facilmente porque os nós são criados de maneira ordenada conforme se observa na figura 4.36.

[44] Essas “arestas” não são aquelas do modelo sólido e sim, correspondentes ao lado de um elemento da malha. É a linha mais escura mostrada na Figura 4.34.

Figura 4.36 – Posicionamento de nós para geração de malha em uma trinca semielíptica a partir da seleção de arestas na superfície



Fonte: Elaboração do próprio autor

4.7.6 Importação de modelos sólidos

O modelador de sólidos do CABEMT se mostra robusto e flexível, contudo, certas formas geométricas muito complexas podem não ser possíveis de se gerar no CABEMT. Por isso, foram implementados algoritmos para efetuar a importação de sólidos criados por outros programas.

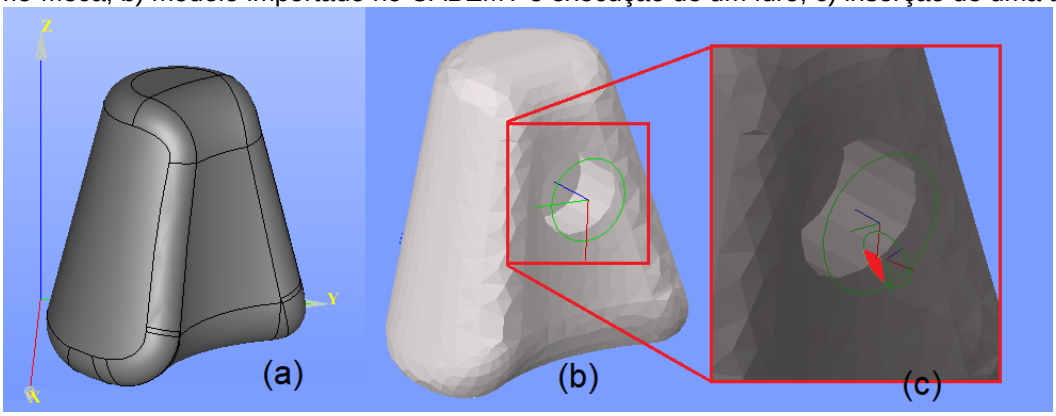
O problema nessa importação é que o CABEMT representa sólidos apenas por poliedros, ao passo que a maioria dos modeladores sólidos trabalham com quádricos e/ou NURBS. Por isso, o método de importação no CABEMT deve ser aplicado a uma malha que discretiza a superfície do sólido. Essas malhas são geralmente armazenadas como um arquivo de texto e, em geral, cada programa dispõe os dados de uma forma diferente.

Na classe *importMeshDialog* encontram-se as funções para ler malhas geradas pelo módulo SMESH do programa Salome-Meca, uma plataforma de código aberto para simulação via MEF. A malha deve estar na forma de arquivo .dat.

Durante a importação, à cada elemento da malha é assimilada uma face. O ângulo entre normais de faces adjacentes é verificado e se ele for menor que um valor especificado pelo usuário, as faces são agrupadas em uma mesma superfície.

Na figura 4.37 (a), um sólido que não poderia ser modelado diretamente no CABEMT foi discretizado no SALOME. A malha foi exportada para o formato .dat. e importada no CABEMT. Para demonstrar que as capacidades de modelagem sólida do CABEMT também se aplicam ao sólido importado, um furo cego foi realizado no objeto, conforme mostrado em (b). Em seguida foi criada uma pequena trinca no furo, conforme ilustrado em (c).

Figura 4.37 – Importação de um sólido por meio da discretização de sua superfície: a) modelo no Salome-Meca, b) modelo importado no CABEMT e execução de um furo, c) inserção de uma trinca



Fonte: Elaboração do próprio autor

Com a capacidade de importar sólidos de terceiros, o CABEMT tem seu poder de representação estendido, cobrindo praticamente qualquer tipo de sólido de interesse na engenharia. No momento, o CABEMT pode importar malhas apenas do Salome/Meca, entretanto, é simples implementar algoritmos para importar malhas de outros programas que gerem arquivo das malhas em texto.

4.8 GERAÇÃO DE MALHA NO CABEMT

4.8.1 Introdução

Algoritmicamente, a geração de malhas bidimensionais é certamente um processo menos complexo que a modelagem de sólidos. No CABEMT, procurou-se limitar as opções de discretização do domínio apenas ao essencial. Isso quer dizer que a geração da malha ocorre de maneira lenta, mas que para a maioria dos sólidos não chega a ser um grande problema.

No CABEMT, os elementos da malha possuem vértices ou pontos, e nós. Os vértices caracterizam geometricamente o elemento, delimitando suas fronteiras. Os nós são os pontos de interpolação de uma determinada grandeza, além de fornecerem informações de conexão entre os elementos (assim como os vértices), crucial na etapa de montagem do sistema linear.

4.8.2 Estrutura de dados

Mesh é a principal classe encarregada da geração de malhas no CABEMT. A classe *modMesh* contém alguns métodos para modificação e ajuste da malha. Além disso existem duas classes parcialmente inativas: *meshDivAndConquer* e *meshQuadTree*. A serem explicadas no item 4.8.5.

Mesh é uma classe constituída basicamente por métodos estáticos e dinâmicos, recebendo objetos da classe *solid* para discretização. Ela cria instâncias da sua própria classe (*mesh*), de *node* e também de *element*, seus constituintes fundamentais.

Verifiquemos a seguir em maiores detalhes os componentes e funções dessas três classes.

4.8.2.1 Node

Representa um nó da malha.

4.8.2.1.1. Atributos de *node*: não-objetos

- x,y,z: as coordenadas globais do nó;
- Number: um número inteiro de identificação unívoca do nó;
- OldNumber: número inteiro auxiliar;
- InEdge: booleano que indica se o nó está ou não em uma aresta do sólido;
- Corner: booleano que indica se o nó está ou não em um vértice do sólido;
- InFacet: booleano que indica se o nó está sobre uma face classificada como *facet;*
- InHole: booleano que indica se o nó está em uma aresta que pertence à borda de um furo;
- select: booleano que indica se o nó foi selecionado pelo usuário;
- EdgeNumber: inteiro contendo a identificação da aresta associada ao nó;
- OppNode: inteiro contendo o número do nó oposto, ou seja, na outra face da trinca;
- Color: vetor indicando a cor do nó em RGB.

4.8.2.1.2. Atributos de *node*: objetos

- FaceID: lista contendo os números das faces associadas ao nó;
- Normal: um vetor normal associado ao nó. Não é a normal da superfície calculada na posição nodal. Trata-se de um vetor normal à linha formada por um afloramento da trinca e paralela à superfície de afloramento. Essa normal será usada para posicionar nós no entorno do afloramento;

4.8.2.1.3. Métodos de *node*

- Change: muda a posição do nó mediante a entrada de coordenadas ou objeto da classe *vertex;*
- ToVertex: cria e retorna um membro da classe *vertex* com as coordenadas do nó;
- DrawNode: envia os comandos OpenGL para o desenho do nó. Basicamente desenha uma esfera utilizando a biblioteca GLU do OpenGL;
- Copy: copia um nó;
- CopyCrack: copia o nó, porém inverte sua normal;
- GlobalToLocal: transforma as coordenadas do nó para um sistema de coordenadas cartesiano especificado;
- LocalToGlobal: transforma as coordenadas do nó de um sistema de coordenadas cartesiano especificado para o sistema global.

4.8.2.2 Element

4.8.2.2.1. Atributos de *element:* não-objetos

- *faceID*: número de identificação da face a qual o elemento pertence;
- *stress*: matriz contendo os valores das tensões calculadas nos vértices do elemento;
- *TempOrPot*: temperatura ou potencial prescritos nos vértices do elemento;
- *Flux*: fluxo prescrito nos vértices do elemento;
- *Area*: área do elemento;
- *ElNumb*: número inteiro que identifica o elemento univocamente;
- *InCrackFace*: booleano que indica se o elemento está em uma superfície de trinca;
- *OppositeCrackFace*: booleano que indica se o elemento está na face oposta da trinca oposta;
- *OppCrackELnumber*: número do elemento na face de trinca oposta, se ela existir. Caso contrário é igual a -1;
- *VertColors*: matriz que armazena as cores dos vértices do triângulo. Isso é utilizado

durante o pós-processamento.

4.8.2.2.2. Atributos de *element:* objetos

- *Nodes*: os nós que constituem o elemento. São armazenados de modo que percorram o elemento em sentido anti-horário;
- *Vert*: os vértices do elemento em ordem anti-horária;
- *Normal*: o vetor normal do elemento – entidade da classe *vertex*;
- Cg: entidade da classe *vertex.* É a posição do centroide do elemento;
- *DescontNodes*: lista com as posições dos nós que caracterizam um elemento descontínuo. Por exemplo: {1,2} indica que os nós na posição 1 e 2 não coincidem com os vértices do elemento. A posição dos nós descontínuos (em termos de coordenadas naturais no elemento) é fixa;

4.8.2.2.3. Métodos de *element*

- *DrawElement*: envia os comandos do *OpenGL* para a representação gráfica do elemento;
- *CopyCrack*: copia o elemento para a face oposta da trinca;
- *LinearToQuad*: transforma um elemento linear em quadrático;
- *LinearToQuadCrack*: análogo ao anterior, mas utilizado em elementos nas faces da trinca. Sempre transforma os elementos em quadráticos descontínuos;
- *DivideElement*: divide um elemento em dois, dados um ponto em sua aresta e a posição do nó sucessor e predecessor ao ponto. Será utilizado no cálculo das integrais quase-singulares;
- *DivideRetElement*: análogo ao método anterior, porém aplica-se a elementos quadrangulares;
- *DivElementPointInside*: divide um elemento triangular ou quadrangular fazendo com que um ponto em seu interior seja o vértice comum a todos os elementos. Será utilizada no cálculo das integrais quase-sigulares;
- *genElbyVert*: gera um elemento tendo como entrada seus vértices, interpolação linear ou quadrática e se descontínuo ou não;

- *normalCalc*: calcula a normal de um elemento triangular.

4.8.2.3 Mesh

A classe *mesh* representa a malha em si com seus nós, elementos e condições de contorno. Relembrando o item 4.7.1.4, objetos da classe *mesh* serão associados a objetos da classe *solid*.

4.8.2.3.1. Atributos de *mesh:* não-objetos

Não há

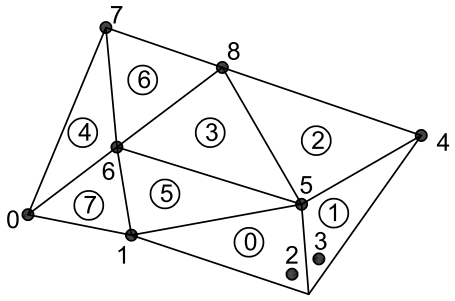
4.8.2.3.2. Atributos de *mesh:* objetos

- *Nodes*: uma lista com os n objetos da classe *nodes* que constituem a malha;
- *Elements*: uma lista com os N_e objetos da classe *element* que constituem a malha
- *Nodes_Elements*: é uma lista[45] $N\ x\ (quantidade\ de\ elementos\ associados)$ onde a posição da linha está associada à posição do nó armazenado em nodes. Por exemplo, supomos que um nó na posição 20 em nodes esteja associado aos elementos das seguintes posições da lista *elements:* 45, 90, 2. Então, na posição 20 de *nodes_elements* haverá os seguintes números {45,90,2}. Isso permite uma busca muito rápida sobre quais elementos estão associados a um determinado nó. Esse tipo de informação é solicitado por muitos algoritmos;
- *Elements_Nodes*: é similar ao item anterior, porém nessa lista ($N_e\ x\ núm.\ de\ nós\ associados$), a linha representa a posição do elemento na lista *elements* e os itens da coluna representam a posição de cada nó do elemento na lista *Nodes;*

A figura a seguir mostra como as listas *Nodes_Elements* e *Element_Nodes* são construídas:

[45] Na verdade, um objeto da classe *ArrayList* – nativa do Java. Tem a vantagem de admitir um número variável de colunas para cada posição de linha da lista.

Figura 4.38 - Construção das listas relacionais de elementos e nós no CABEMT



EL	ELEMENTS_NODES
0	1-2-5
1	3-4-5
2	5-4-8
3	5-8-6
4	6-7-0
5	1-5-6
6	6-8-7
7	0-1-6

NÓ	NODES_ELEMENTS
0	7-4
1	0-5-7
2	0
3	1
4	1-2
5	0-1-2-3-5
6	3-4-5-6-7
7	4-6
8	2-3-6

Fonte: Elaboração do próprio autor

- *Elements*_Vert: análogo a *Elements_Nodes*, mas se referindo aos elementos e seus vértices;
- *Vert_Elements*: análogo a *Nodes_Elements,* mas se referindo aos vértices e seus elementos;
- *CrackFrontVert*: armazena em listas os vértices que definem o(s) contorno(s) de trinca do modelo de modo a agilizar o pós-processamento;
- *ProcessedEdges*: armazena uma lista contendo o número do nó e a identificação da aresta relacionada a ele. Isso possibilita o reconhecimento de arestas já processadas e evita dupla discretização;
- *Prescribed_u*: lista contendo os deslocamentos em x,y e z impostos aos nós como condição de contorno. Quando nenhuma, assume o valor NaN;
- *Prescribed_t*: análogo a *prescribed_u*. Quando os dois não forem informados pelo usuário, assume o valor nulo. Se os deslocamentos forem informados, assume o valor NaN;
- *Prescribed_T*: lista com o valor prescrito de temperatura nos nós – condição de contorno. Quando não especificado pelo usuário, assume o valor NaN;
- *Prescribed_q*: lista com o fluxo térmico nodal prescrito pelo usuário. Comporta-se como *prescribed_t*;
- *Tinf*: lista com as temperaturas “ao infinito” prescrita pelo usuário para o cálculo de convecção;
- *Coef_h*: lista com os coeficientes de convecção para cada nó, definido pelo usuário;

- *Vertices*: lista dos objetos da classe *vertex* representando as extremidades dos elementos.

Todas as listas de *mesh* são membros da classe *ArrayList* do Java. Essas listas organizam praticamente toda a informação necessária para a resolução do problema pelo MEC. Observar que há redundância de algumas informações, por exemplo: um objeto da classe *elements* contém os seus vértices como atributos, mas estes também estão na lista *vertices* do objeto *mesh*. Há de se observar, no entanto, que na memória do sistema não existem duas entidades para o mesmo vértice. O espaço é ocupado para apenas um vértice e o outro é uma referência (o endereço do objeto), ocupando pouca memória.

Essa dupla representação torna a elaboração de alguns algoritmos mais prática. É mais compreensível utilizar as listas ao montar o sistema de equações, uma vez que a posição da entidade na lista permite a determinação imediata de seu lugar no sistema de equações. Por outro lado, quando se necessita de uma propriedade de algum objeto durante um cálculo qualquer, geralmente é mais simples acessar seus atributos diretamente.

4.8.2.3.3. Métodos de *mesh*

- *GenerateSolidMesh*: é a função principal, responsável por chamar as demais de modo a gerar a malha no sólido. Após a criação dos elementos, avalia quais deles se transformarão em semi-contínuos ou descontínuos;
- *GenerateSurfaceMesh*: é a segunda função principal, responsável por acionar as demais funções a fim de se gerar a malha em uma superfície;
- *GenerateEdgesNodes*: esse método atribui nós/pontos às arestas de uma face conforme o espaçamento definido pelo usuário ou o padrão utilizado pelo programa. A função verifica se as arestas vizinhas (que estão em orientação inversa à face corrente) já passaram por esse processo. Em caso positivo, apenas associa os nós daquelas à aresta corrente;
- *GenerateHolesEdgesNodes*: funcionamento similar à anterior. Ela também atribui “vetores normais” aos nós para guiar o processo de expansão de fronteira;
- *CreateFaceGrid*: insere pontos no interior da face pela superposição de uma grade quadriculada. Ver item 4.8.4 para maiores detalhes;

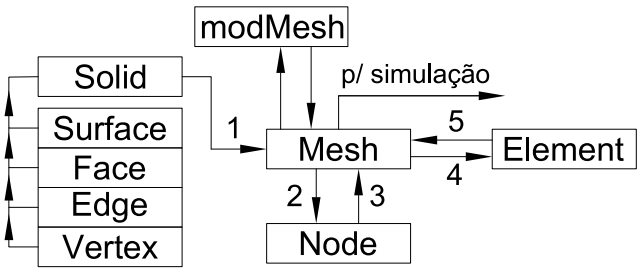
- *CreateFaceGridAdvBoundary*: insere pontos no interior da face pelo método da fronteira em avanço;
- *CreateHoleGridAdvBoundary*: insere pontos em uma face pelo método da expansão de fronteira a partir de furos;
- *SurfCrackAdvBoundary*: procedimento de expansão de fronteira específico para trincas;
- *mergeNodes*: funde nós muito próximos;
- *Delaunay_O4*: método de triangulação de Delaunay direto - $O(N^4)$;
- *InsertNode*: insere um nó na face e refaz a triangulação. Esse método precisa ser revisto para que a triangulação seja refeita somente nos elementos afetados pela inserção;
- *InsertNodeInEdge*: insere um nó em uma aresta e refaz a triangulação. Pelos mesmos motivos acima, o método precisa ser revisado para maior eficiência;
- *delNode*: apaga um nó e refaz a triangulação;
- *TreatCornerNodeOnCrackHole*: cria nós adicionais nas imediações da ponta da trinca de forma a melhorar a qualidade da malha naquela região;
- *InitialFillingPrescribed*_t_and_u: preenche as listas de condições de contorno apenas para habilitá-las. Valores de deslocamentos, temperaturas, coeficientes de troca térmica e temperaturas ao infinito são definidos como NaN. Os valores de força de superfície e fluxo não definidos como nulos – condição padrão;
- *FillPrescribedForGiven_u*: recebe um valor de deslocamento, sua direção e a superfície ao qual será aplicado. Em seguida, atribui esses deslocamentos preenchendo as listas *prescribed_u.* Atribui às forças de superfície correspondentes o valor NaN;
- *FillPrescribedForGiven_t*: análogo ao método anterior, porém não altera o valor dos deslocamentos que por padrão já estarão como NaN;
- *FillPrescribedForGiven_Temp*: análogo aos métodos anteriores – corresponde à aplicação das condições de contorno de temperatura;
- *FillPrescribedForGiven_q*: atribui condição de contorno de fluxo térmico;

- *FillPrescribedForGiven_h*: atribui os valores dos coeficientes de convecção na superfície, quando tal condição de contorno for selecionada;
- *FillPrescribedForGiven_Tinf*: atribui valores de temperaturas do fluido (T_∞) para cada nó;
- *FillPrescrMoment*: o CABEMT aceita imposição direta de momentos em estruturas tridimensionais. Essa função decompõe os momentos em forças equivalentes ao longo da face;
- *EvalCommBC*: apresenta e recolhe informações do prompt de comando para atribuição das condições de contorno estruturais. Tais informações são digitadas pelo usuário;
- *EvalCommTempBC*: análogo ao método anterior, mas dedicado ao tratamento de condições de contorno de temperatura;
- *linearToQuad*: transforma elementos lineares em quadráticos descontínuos. Atualmente sem capacidade de transformação para elementos quadráticos contínuos, pois esse elemento não é utilizado no CABEMT.

4.8.3 Diagrama e funcionamento

O diagrama na figura 4.39 esquematiza o fluxo básico de dados durante a geração de malha no CABEMT. O parâmetro principal de entrada é um objeto da classe *solid* – dele fazem parte as superfícies, faces e arestas. Tais entidades também contém as informações de espaçamento da malha que foram inseridas pelo usuário ou são padrão do programa. Todas essas informações são recebidas e processadas pelas funções da classe *mesh.* Deste primeiro processamento resulta a distribuição de pontos/nós pelas faces do sólido.

Figura 4.39 – Diagrama básico do fluxo de informações para geração de malha no CABEMT. Os números indicam a ordem de transmissão dos dados.



Fonte: Elaboração do próprio autor

Os nós gerados são agrupados nas listas de *mesh*. No fluxo 4, as funções de

mesh realizam a triangulação de Delunay dos vértices/nós[46], criando os elementos – sendo transferidos e armazenados no objeto da classe *mesh,* conforme fluxo 5. Até aí todos os elementos criados são triangulares lineares contínuos. Somente no final do processamento é que são modificados para descontínuos ou não descontínuos conforme as seguintes regras:

- Se o elemento tem apenas um ou dois vértices em arestas ou cantos: transformá-lo para elemento semi-descontínuo no qual os nós em arestas e cantos serão deslocados para o interior do elemento;
- Se os três vértices pertencem a arestas e/ou cantos: transformar o elemento em descontínuo;
- Se o elemento está em uma face de trinca: transformá-lo para descontínuo quadrático. Na versão atual do CABEMT, somente elementos em trincas podem ser quadráticos.

Será apresentada mais adiante a opção de converter os elementos triangulares localizados nas trincas para elementos quadrangulares. Isso é feito mediante intervenção do usuário e o processamento é feito por funções da classe auxiliar *modMesh.*

4.8.4 Discretização do domínio

Conforme apresentado no item 3.3, a discretização do domínio requer pelo menos dois passos fundamentais: o posicionamento de nós e a interligação destes para formação de elementos. Entretanto, para se efetuar o posicionamento e distribuição de nós nas arestas e superfícies do domínio, são necessárias informações geométricas acerca dessa distribuição, de tal forma que a distância entre os nós é a mais importante. Não obstante, informações acerca da topologia à qual o nó está associado também devem ser obtidas pelos algoritmos, tais como se o nó situa-se em uma aresta ou vértice do sólido, a fim de se atribuir elementos descontínuos ou não neste local.

Grosseiramente, a sequência de criação da malha é a seguinte:

[46] Até o final do processo de geração de malha, a noção entre nós e vértices se confunde, porque afinal de contas, eles ocupam o mesmo lugar geométrico. Somente após a geração de elementos descontínuos é que os vértices e nós não ocuparão necessariamente o mesmo lugar.

- Criação e posicionamento de nós nas arestas de cada face: consiste simplesmente em distribuir os nós ao longo das arestas, levando-se em conta o espaçamento desejado pelo usuário. Durante esse processo as arestas de outras faces vizinhas à que está sendo processada são investigadas para avaliar se já possuem nós associados. Caso isso ocorra, a semi-aresta da face corrente que possua uma vizinha processada não precisa de novos nós, apenas englobar os nós desta semi-aresta vizinha;

- Criação e posicionamento de nós no interior das faces: há várias formas de se fazer isso, no entanto, o CABEMT concentra-se na técnica da grade ou da expansão de fronteira. Na técnica da grade, conforme figura 3.3, uma malha segmentada em vários quadrados com nós em seus vértices é superposta à face. Nós que estejam muito próximos às arestas, no exterior da face ou no interior de furos são excluídos.

  Na expansão por fronteira os nós das arestas são copiados em sucessivas frentes, seguindo a perpendicular da aresta, conforme mostrado na figura 3.4. Quando um novo nó está muito próximo de outro existente, ele não é inserido.

  Na versão atual do CABEMT, todas as faces podem ter seus nós distribuídos pela grade ou avanço de fronteira, exceto nas imediações dos furos, onde os nós são sempre distribuídos pelo avanço de fronteira. Isso confere maior qualidade nas imediações desses elementos geométricos (vide figura 3.3) que frequentemente são regiões de interesse em virtude de seu potencial em concentrar tensões.

  Devido à representação dos sólidos por poliedros, todas as faces são planares, fazendo com que o processo de geração da malha, para todos os efeitos seja estritamente bidimensional. A distribuição de nós, por exemplo, é feita em sistemas de coordenadas locais – um para cada face.

- Formação dos elementos: nessa etapa, os nós são sistematicamente conectados de modo a formarem os elementos. O CABEMT sempre gera elementos triangulares. A única exceção é que existe a possibilidade de se utilizar elementos quadrangulares nas faces de trinca – mesmo assim, esses são formados pela união de dois elementos triangulares adjacentes;

4.8.5 Triangulação de Delaunay

Como já explicado ao longo desse trabalho, a triangulação de Delaunay é um

método de união de pontos em triângulos de modo a maximizar a qualidade dos elementos. Os algoritmos do CABEMT implementam a instrução 3.3 para a triangulação. Esse método é mais lento ($O(N^2)$) que a divisão e conquista ($O(NlogN)$), mas se aproveita da representação poliédrica, que às vezes segmenta o domínio em muitas faces. Como a triangulação é feita face por face, o número de nós para cada execução do algoritmo é em geral pequeno, não resultando em tempos elevados de triangulação. Isso remonta, ainda que fracamente, ao processo de divisão e conquista.

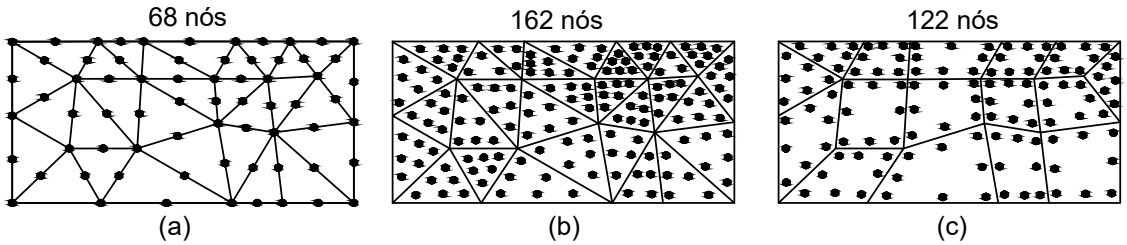
A geração de malha por *quadTree*, ainda está em desenvolvimento. Ainda são necessários ajustes quando a superfície é composta por facetas.

Há um algoritmo de geração de malha ainda incompleto e hibernado: geração de malha por divisão e conquista. Esse método pode tornar-se disponível em versões futuras do CABEMT.

4.8.6 Conversão para malha de elementos quadrangulares

A aplicação das equações de contorno hipersingulares nas faces da trinca exige, pelo menos em teoria, que os elementos nessas regiões sejam descontínuos, a fim de se garantir a continuidade das derivadas do campo de deslocamento. Essa necessidade resulta em uma concentração de nós mais elevada que o emprego de elementos contínuos, conforme se observa na figura a seguir:

Figura 4.40 – Comparativo entre densidades nodais para diferentes tipos de elementos: a) triangular contínuo, b) triangular descontínuo c) predominantemente quadrangular descontínuo



Fonte: Elaboração do próprio autor

Ainda observando a figura, nota-se que para uma malha cuja maioria dos elementos triangulares foram unidos e transformados em quadrangulares (c), o número de nós caiu cerca de 25% para o mesmo tamanho médio de elemento. Portanto, a conversão de malha constituída por elementos triangulares pode ser interessante no MEC. Isso é especialmente importante porque a maior densidade de elementos usualmente ocorre nas trincas, local de maior interesse na estrutura.

A conversão da malha é tarefa simples, bastando determinar todos elementos

vizinhos daquele correntemente processado e procurar o vizinho que forma com o elemento corrente o quadrângulo que minimize o desvio em relação ao ângulo reto, isto é:

$$mín\left(\left|minAng - \frac{\pi}{2}\right| + \left|maxAng - \frac{\pi}{2}\right|\right) \quad (4.5)$$

onde $minAng$ e $maxAng$ são o menor e o maior ângulo do elemento quadrangular testado. Além disso, pode-se determinar um critério de não formação do elemento quadrangular. No CABEMT, se o melhor quadrângulo tiver arestas com ângulo superior a 140º entre si, não ocorrerá a formação do elemento.

4.8.7 Deslocamento de nó conforme especificado

O comando *moveElementVert (mev)* recebe um vértice selecionado pelo usuário e o move até um ponto selecionado por ele desde que esse ponto esteja dentro dos elementos adjacentes ao vértice selecionado.

A seleção do vértice pode ser determinada como aquele que estiver mais próximo da linha (raio) de visada entre os planos de corte próximo e distante – ver figura 4.6. Considerando que a linha imaginária (ou de visada)[47] possa ser descrita pelos pontos $\boldsymbol{P_1}$ e $\boldsymbol{P_2}$, e $\boldsymbol{P_0}$ seja o ponto cuja distância se quer medir, tem-se que a distância d pode ser calculada por:

$$d' = \left|\frac{(\boldsymbol{P_2} - \boldsymbol{P_1})\ \boldsymbol{x}\ (\boldsymbol{P_1} - \boldsymbol{P_0})}{|\boldsymbol{P_2} - \boldsymbol{P_1}|}\right| \quad (4.6)$$

Na sequência o usuário clica no ponto de destino desejado, fazendo com que um laço percorra todos os elementos. Para cada um deles é determinado um ponto de interseção entre a linha imaginária e o plano que contém o elemento. Se o elemento tem normal (a, b, c) e a equação do seu plano é $ax + by + cz + d$, a linha tem vetor diretor (u, v, w) e passa por um ponto (x_0, y_0, z_0), então o parâmetro da intersecção plano-reta é:

$$t = \frac{-ax_0 - by_0 - cz_0 - d}{au + bv + cw} \quad (4.7)$$

Portanto o ponto de intersecção, baseado na equação da reta é:

---

[47] Relembrando: essa é a linha que sai da ponta da seta do mouse e atravessa o plano de corte próximo e o distante.

$$\begin{aligned} x &= x_0 + ut \\ y &= y_0 + vt \\ z &= z_0 + wt \end{aligned} \tag{4.8}$$

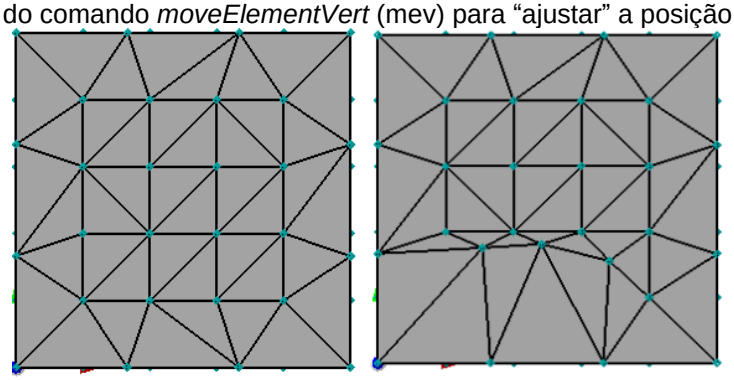
Se o ponto (x, y, z) estiver dentro do elemento, então desloca-se o nó para essa nova posição. A condição de ponto dentro ou fora de um elemento pode ser verificada pelo algoritmo exposto no APÊNDICE C. Alternativamente, para um elemento triangular, pode-se afirmar que o ponto estará fora do triângulo se qualquer uma das desigualdades abaixo for verdadeira:

$$\begin{aligned} \left(\overrightarrow{AB}\ x\ \overrightarrow{AP}\right).\boldsymbol{n} &< 0 \\ \left(\overrightarrow{BC}\ x\ \overrightarrow{BP}\right).\boldsymbol{n} &< 0 \\ \left(\overrightarrow{CA}\ x\ \overrightarrow{CP}\right).\boldsymbol{n} &< 0 \end{aligned} \tag{4.9}$$

A figura 4.41 ilustra uma situação em que os nós de alguns elementos foram movidos manualmente.

O CABEMT também aceita o deslocamento de nós nas arestas, porém apenas quando estes não coincidem com algum vértice da geometria. Nesse caso, é preciso determinar o ponto (x, y, z) na aresta e que fica à menor distância da linha de visada entre o plano de corte distante e o próximo.

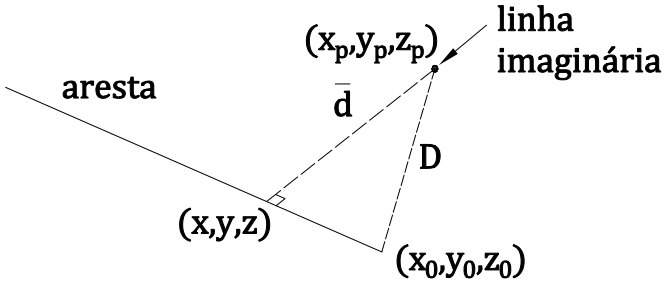
Figura 4.41 – Uso do comando *moveElementVert* (mev) para “ajustar” a posição de alguns vértices.



Fonte: Elaboração do próprio autor

Uma forma satisfatoriamente aproximada de se fazer isso encontra-se na figura a seguir:

Figura 4.42 – Determinação do ponto de uma aresta mais próximo da linha imaginária



Fonte: Elaboração do próprio autor

Na figura quer se determinar a nova posição (x, y, z) do nó, requerida pelo usuário ao clicar o *mouse* em uma região próxima à aresta do sólido. A linha imaginária dispara pelo *mouse* está perpendicular ao plano do papel. A distância entre as duas linhas, $\bar{d}$, pode ser calculada por (SANTOS, 2007):

$$\bar{d} = \frac{|\overrightarrow{\boldsymbol{P_1 P_2}} \cdot (\boldsymbol{V_1} \, x \, \boldsymbol{V_2})|}{|\boldsymbol{V_1} \, x \, \boldsymbol{V_2}|} \tag{4.10}$$

em que $\boldsymbol{P}_1$ e $\boldsymbol{P_2}$ são os pontos iniciais de cada linha (a aresta e a linha imaginária), $\boldsymbol{V_1}$ e $\boldsymbol{V_2}$ são os vetores diretores da aresta e da linha de visada. O ponto cujas coordenadas são x_0, y_o, z_0 é início da aresta cujo vetor diretor é dado por $V_1 = (u, v, w)$. Considerando que a aresta seja parametrizada da mesma forma que na equação (4.8), o parâmetro t que marca a posição (x, y, z), ou seja, a projeção de um ponto sobre a linha, pode ser calculado por:

$$t = \sqrt{\frac{D^2 - \bar{d}^2}{u^2 + v^2 + w^2}} \tag{4.11}$$

Nesse caso D é a distância entre (x_0, y_0, z_0) a (x_p, y_p, z_p), este último é o ponto de projeção da posição inicial do vértice (nó) do elemento até a linha de visada. O processo todo pode ser resumido na seguinte instrução:

Instrução 4.5 Deslocar um nó ao longo de uma aresta

a) Selecionado um nó $\boldsymbol{P}$ na aresta, projetá-lo na linha imaginária e encontrar o ponto (x_p, y_p, z_p) usando as equações (4.10) e (4.11);

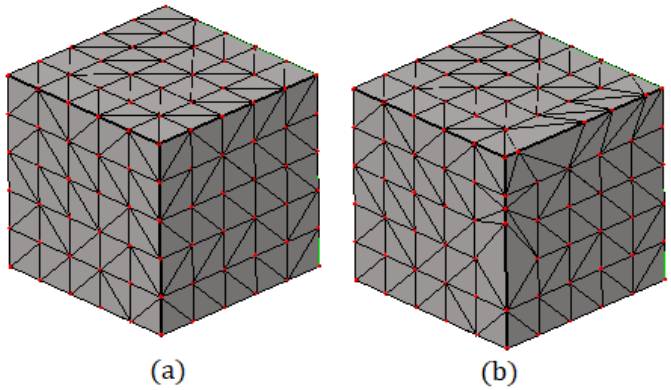
b) Fazer a projeção de (x_p, y_p, z_p) na aresta, também utilizando as equações (4.10) e (4.11), portanto descobrindo o ponto (x, y, z) sobre a aresta;

c) Deslocar o nó $\boldsymbol{P}$ até (x, y, z)

A figura abaixo mostra o deslocamento de alguns nós localizados nas arestas

de um cubo.

Figura 4.43 – Deslocamentos de nós em arestas realizado no CABEMT: a) malha original, b) malha após interferência do usuário

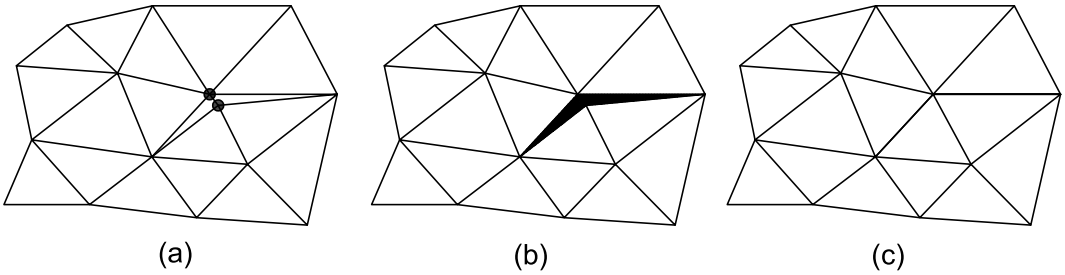


Fonte: Elaboração do próprio autor

4.8.8 Fusão de elementos

Em algumas situações, dois vértices podem ficar muito próximos um do outro e os elementos adjacentes serem de baixa qualidade. O comando *mergeELvert(mrgev)* identifica tal situação (distância < tolerância especificada pelo usuário), seleciona os elementos comuns aos dois nós, remove um dos nós e todos os elementos selecionados. Em seguida, reconecta os elementos que antes estavam associados ao nó excluído. A figura 4.44 ilustra o processo.

Figura 4.44 – Fusão de elementos a) dois nós selecionados pelo critério de distância, b) elementos selecionados, c) resultado final após fusão dos elementos



Fonte: Elaboração do próprio autor

4.8.9 Malha volumétrica

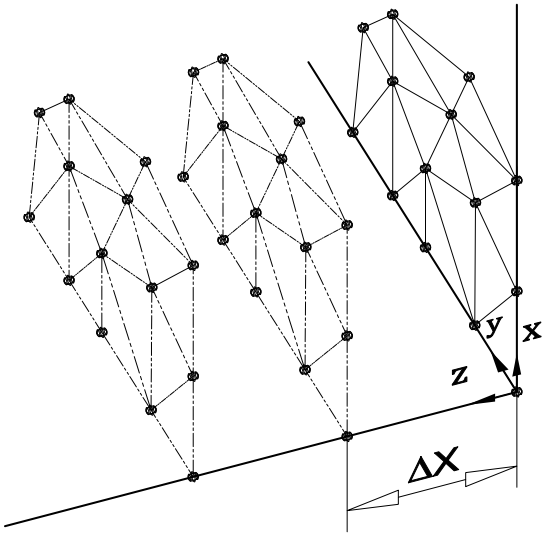
A menos que seja utilizada a técnica da tripla reciprocidade, que é de difícil implementação, a resolução de problemas elasto-plásticos pelo MEC exige a criação de células nas regiões onde ocorrerá deformação plástica. O propósito do CABEMT não será de fazer simulações de deformação plástica generalizada, esta ficará contida em pequenas regiões do domínio. Portanto, a geração de malha tridimensional foi

programada para a discretização local de pequena extensão. Ela poderá ser feita de duas maneiras: a partir da seleção de elementos e sua “expansão” ou por meio da extrusão de elementos.

4.8.9.1 Geração de malha por meio da expansão de elementos

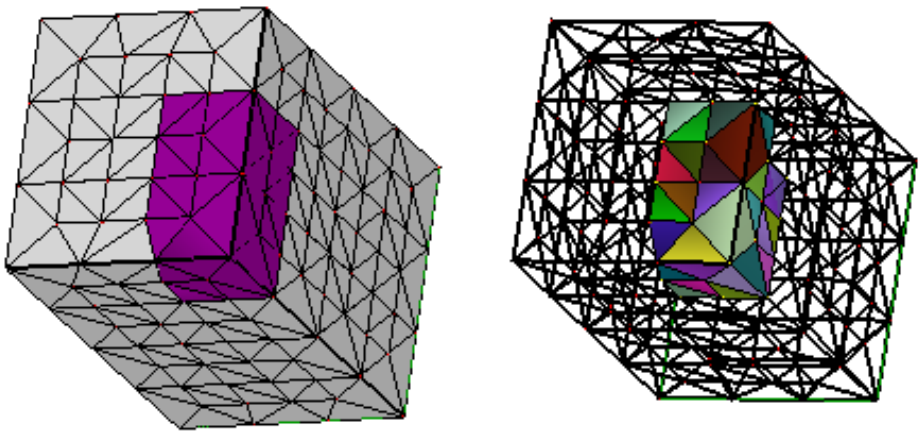
Selecionam-se os elementos cujos nós serão propagados em novas nuvens para o interior do sólido. O passo (ΔX) e o número de nuvens também são dados de entrada. Na figura 4.45 os elementos de uma superfície paralela ao plano XY foram selecionados. Outras superfícies, paralelas aos planos YZ e XZ por exemplo, também poderiam ter sido selecionadas. Um nó da nuvem somente será criado se ele respeitar uma distância mínima – especificada pelo usuário – em relação aos demais existentes. Esse método de geração de malha funciona muito bem em cantos e áreas pequenas, como evidenciam as figuras 4.46 e 4.47.

Figura 4.45 – Geração de malha tridimensional por expansão de elementos selecionados – duas nuvens criadas



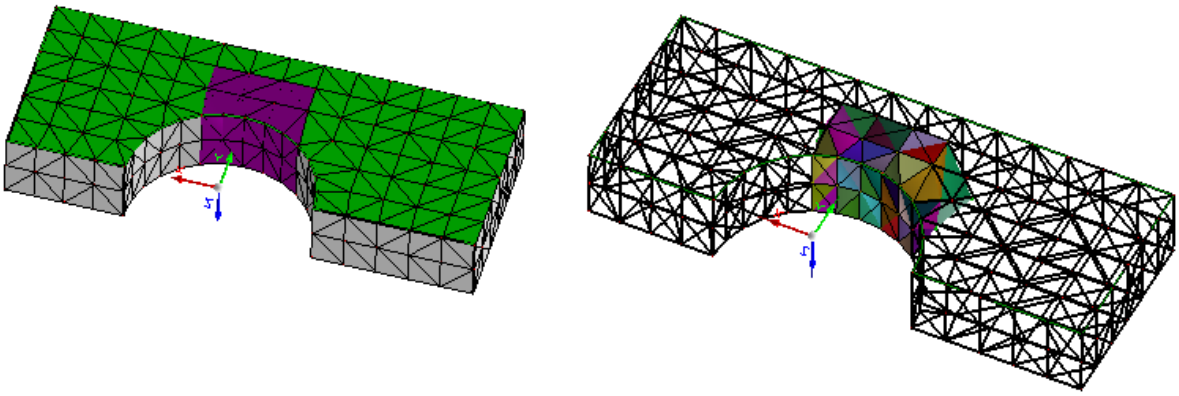
Fonte: Elaboração do próprio autor

Figura 4.46 – 1º Exemplo de geração de malha local tridimensional por expansão no CABEMT



Fonte: Elaboração do próprio autor

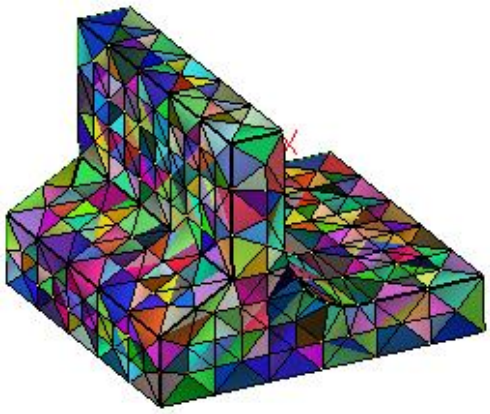
Figura 4.47 – 2º Exemplo de geração de malha local tridimensional por expansão no CABEMT



Fonte: Elaboração do próprio autor

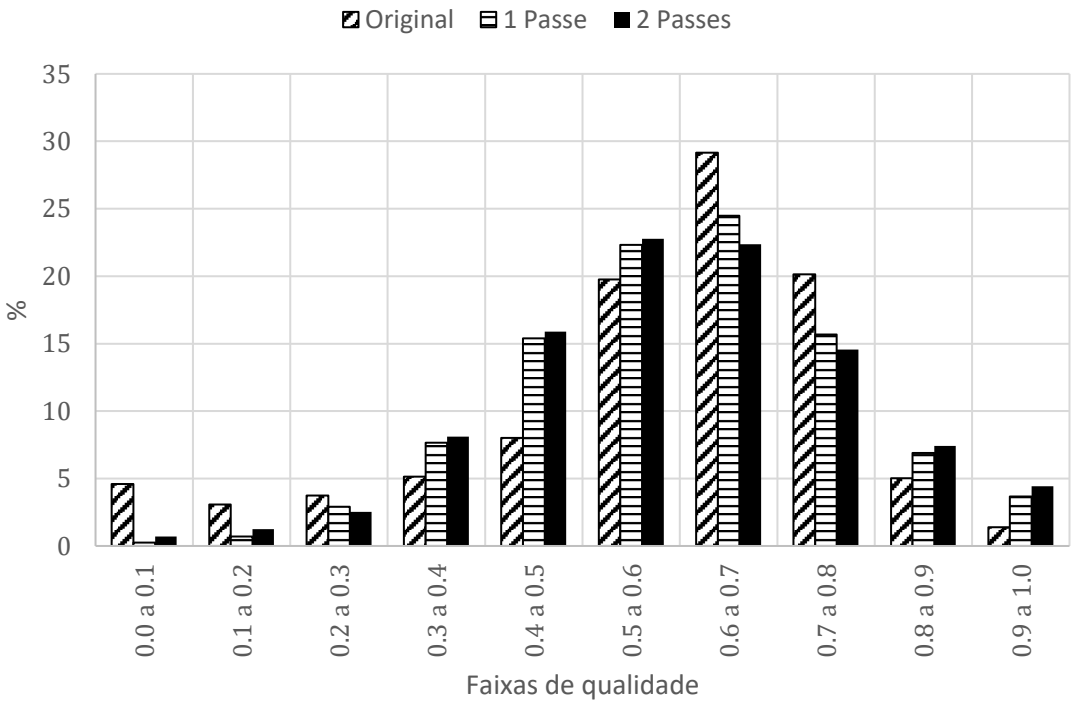
Para verificar o algoritmo de otimização, pode-se fazer um experimento usando o método visto no item 3.3.4 de otimização na malha, aplicando-lhe a discretização da figura 4.48. No histograma de frequências representado na figura 4.49 nota-se uma melhora sensível da qualidade da malha no primeiro passe de otimização – há uma redução expressiva de elementos de baixa qualidade. O uso de dois passes nesse caso já não traz mudanças relevantes.

Figura 4.48 – Uma malha tridimensional tetraédrica para teste do método de otimização da posição interna dos nós



Fonte: Elaboração do próprio autor

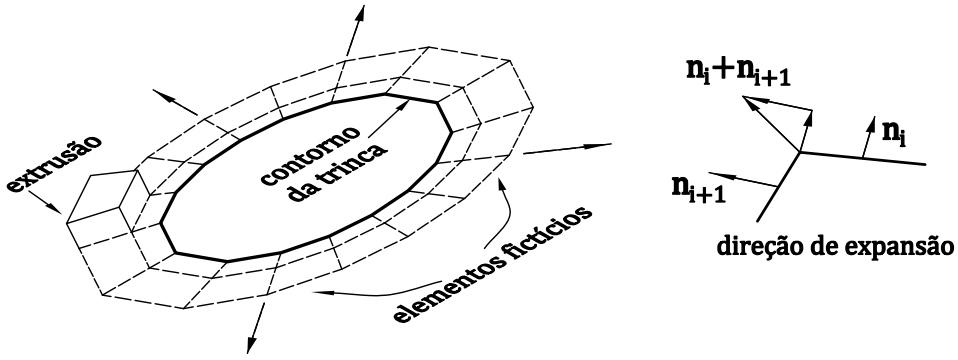
Figura 4.49 – Otimização de posicionamento de um nó interno em malha tetraédrica



Fonte: Elaboração do próprio autor

Para a geração de malhas volumétricas no entorno de uma trinca, o CABEMT extruda diretamente os elementos da sua face. Porém, dessa forma a frente de avanço da trinca ficaria sem elementos no entorno. Pra resolver esse problema, o algoritmo cria elementos bidimensionais fictícios para depois extrudá-los, conforme ilustrado abaixo:

Figura 4.50 – Criação de elementos volumétricos por extrusão avante ao contorno de trinca

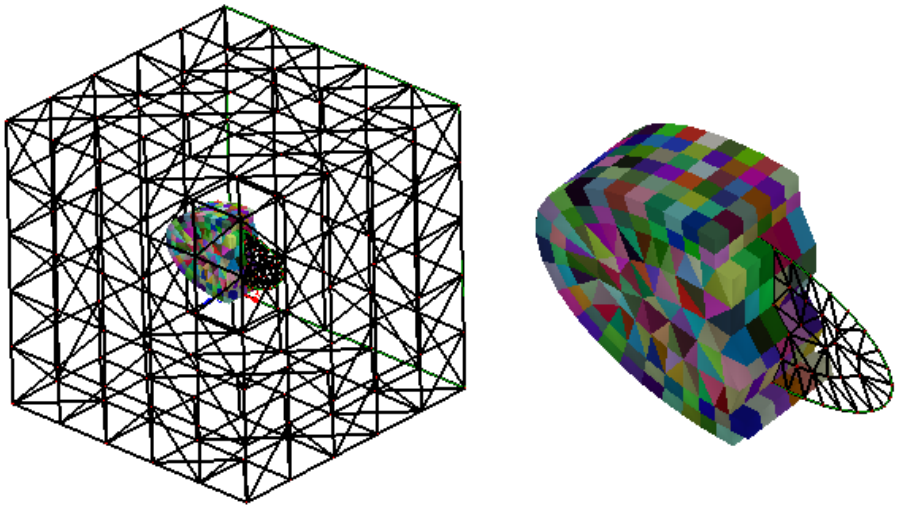


Fonte: Elaboração do próprio autor

A extrusão pode ser feita de maneira trivial, estando ordenados os nós do contorno da trinca. A direção de expansão para criar os elementos fictícios é determinada pela média das normais das arestas que se conectam a um nó, conforme mostrado no lado direito da figura 4.50.

Na figura a seguir observa-se o resultado do método de geração de células no entorno de uma trinca interna elíptica.

Figura 4.51 – Exemplo de malha volumétrica gerada ao redor de uma trinca interna elíptica – visão geral e detalhe. Algumas células foram omitidas para visualização do interior da malha



Fonte: Elaboração do próprio autor (CABEMT)

No CABEMT, a geração desse tipo de malha está limitada a defeitos internos. No caso de afloramento na superfície, a geração ainda não está totalmente automatizada, de tal sorte que o usuário deve efetuar a remoção manual de alguns elementos no afloramento. A geração por extrusão tende a criar elementos de melhor qualidade que a expansão de tetraedros desde que os elementos bidimensionais que dão origem às células sejam de boa qualidade.

4.8.10 Verificação de malha

A capacidade dos métodos numéricos obter resultados verossímeis está relacionada, entre outras coisas, à qualidade e correta conectividade dos elementos da malha. A discretização apenas nas superfícies pode ter sua qualidade verificada facilmente por inspeção visual. Mesmo assim, foi escrita a função *checkMesh* do CABEMT que faz as seguintes verificações:

- Calcula a qualidade de cada elemento utilizando a fórmula (3.1). Elementos ruins, ou seja, cujo parâmetro de qualidade excede a 6,0 – são marcados por linhas de forma que o usuário possa fazer alguma intervenção para melhorar a qualidade;
- Verifica se existe algum nó órfão na malha, ou seja, que não está associado a elementos. Se existirem, informa os números na tela de saída;
- Verifica se existe sobreposição de elementos. Elementos sobrepostos são aqueles em que todos os seus vértices são iguais ao sobreposto. Se existirem, informa os números na tela de saída;
- Verifica se existe algum elemento cuja área seja nula. Exclui se houver;
- Verifica se a orientação dos nós está coerente com a orientação dos vértices do elemento. Informa os números dos elementos desconformes na tela, se existirem;
- Verifica se as normais dos elementos estão coerentes com as normais das faces nas quais eles fazem parte. Mostra os números de elementos que não estiverem.

A figura 4.52 mostra a tela de saída do comando de verificação de malha. A partir das informações levantadas, é possível intervir e ajustar a malha. Em geral, essa modificação pode ser realizada manualmente com facilidade, outra vantagem das malhas de contorno.

Figura 4.52 – Exemplo da tela de saída do comando *checkMesh* do CABEMT

```
---------- CHECK 2D MESH RESULTS ----------
---------- ELEMENTS QUALITY Q ----------
(Q CLOSE TO 1.0 IS GOOD, MORE THAN 6.0 SHOULD BE AVOIDED)
Q < 3.0 = 100.0%
3.0 < Q < 6.0 =  0.0%
6.0 < Q < 10.0 =  0.0% (RED LINES)
Q > 10  =  0.0% (MAGENTA LINES)
No orphan nodes detected: OK
No overlaping elements detected: OK
No zero area elements detected: OK
No mismatch between Element Vertices - Nodes orientation: OK
No mismatch between face normal and element normal: OK
----------x----------|
```

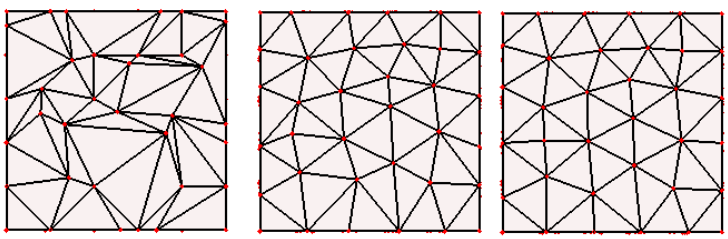
Fonte: Elaboração do próprio autor

4.8.11 Outras funções de MODMESH

Conforme mencionado anteriormente, a classe *modMesh* abriga funções auxiliares para a manipulação de malhas. Além da função de transformação vista no item anterior, vale a pena mencionar outras funções de *modMesh*, tais como:

- *delElement*: deleta um ou mais elementos selecionados por meio do mouse ou pela entrada de seus números no *prompt*;
- *createElement*: cria um elemento triangular desde que três nós estejam previamente selecionados;
- *filterCrackElem:* segrega elementos que fazem parte de uma determinada trinca;
- *generateBC_Arrows:* gera setas indicadoras de carregamentos e restrições, permitindo ao usuário visualizar as condições de contorno aplicadas;
- *laplacian2D (lapl2d):* realiza a suavização laplaciana da malha conforme metodologia descrita no item 3.3.4 e cujo efeito pode ser observado na figura a seguir:

Figura 4.53 – Aplicação do comando de suavização laplaciana em uma malha bidimensional: a) malha original, b) 1º passe, c) 2º passe



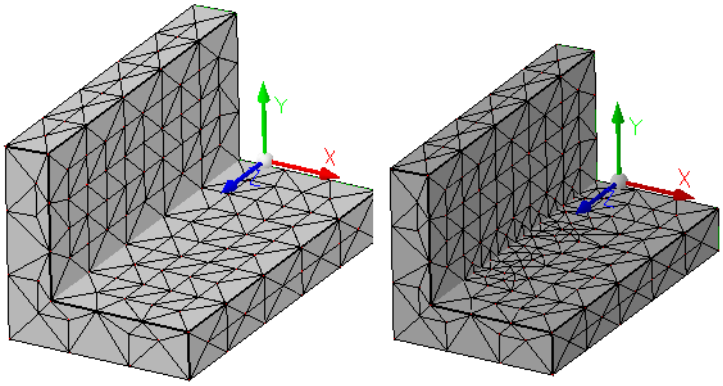
(a) (b) (c)

Fonte: Elaboração do próprio autor (CABEMT)

Nota-se que já no primeiro passe a qualidade da malha melhora substancialmente. Verificar também que os nós localizados nas arestas também são movidos, porém com a restrição de permanecerem na aresta.

- *addElementVert* (aev): permite a inserção de um novo vértice na malha cuja posição é determinada pelo mouse. A inserção pode ocorrer na região interna ou nas arestas das faces. Possibilita um ajuste manual fino. Na figura abaixo, adicionaram-se vértices de modo a refinar a região de transição geométrica da cantoneira:

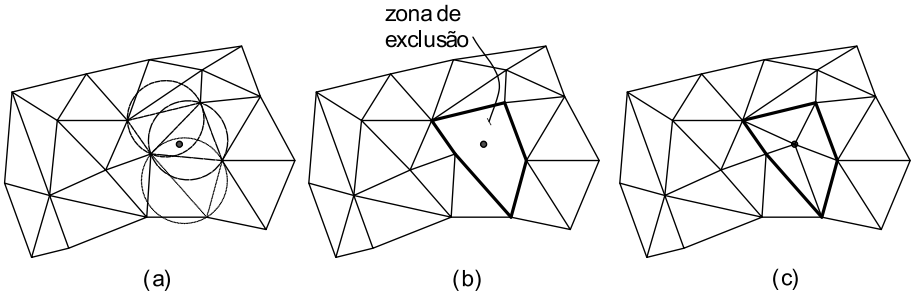
Figura 4.54 – Emprego da função addElementVert



Fonte: Elaboração do próprio autor (CABEMT)

Quando um ponto é inserido, o algoritmo deve determinar todos os elementos cujo circuncírculo engloba o novo ponto. Tais elementos devem ser excluídos e novos criados mediante a ligação do novo ponto com as arestas que envolvem a zona de exclusão, conforme mostrado na figura abaixo:

Figura 4.55 – Etapas para inserção de vértice: a) exclusão de elementos, b) zona de exclusão e c) formação dos novos elementos



Fonte: Elaboração do próprio autor

4.9 FUNÇÕES E MÓDULOS AUXILIARES

Nos itens anteriores, foram apresentadas praticamente todas as funções do CABEMT tanto para modelamento sólido quanto para montagem e resolução dos

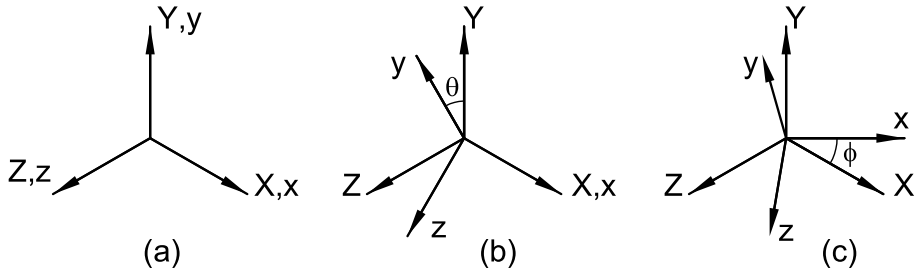
sistemas de equações do MEC. A seguir serão apresentadas algumas funções especiais que facilitam o trabalho de modelagem.

4.9.1 Usando o mouse para visualizar

A dificuldade ou facilidade de se construir um modelo sólido computacional está fortemente ligada à habilidade de interagir com o mesmo e observá-lo sob diferentes ângulos e aproximações. Para essas tarefas, o mouse é sem dúvida o dispositivo mais prático. Podemos eleger três operações básicas de manipulação passiva do modelo via mouse:

- Aproximar ou afastar (zoom): operação trivial que apenas aciona o comando *glScale* do *OpenGL* mediante o movimento da roda do mouse;
- Arraste (*pan*): quando o usuário pressiona e segura a roda do *mouse* e o movimenta, efetua-se a translação dos elementos representados na tela. Isso é facilmente programado através do comando *glTranslate* do *OpenGL*;
- Rotação 3D: a ideia é utilizar os movimentos bidimensionais do *mouse* para produzir rotações tridimensionais na tela enquanto o usuário segurar o botão esquerdo e arrastar *o mouse.* Isso é alcançado ao se considerar dois sistemas de coordenadas: o global que é o sistema enxergado pelo usuário (e ilustrado na tela do CABEMT) e um sistema fixo, transparente ao usuário e imutável conforme ilustrado na figura a seguir.

Figura 4.56 – Rotação de objetos para visualização: a) sistema de coordenadas global (x, y, z) e fixo (X, Y, Z) coincidentes; b) rotação em torno de X; c) rotação em torno de Y



Fonte: Elaboração do próprio autor

Quando o usuário arrasta o *mouse* verticalmente, ele produz uma rotação do sistema global na direção do eixo X. Se por outro lado o *mouse* é arrastado horizontalmente, ele produz uma rotação em torno do eixo Y.

É preciso ter em mente que o sistema de coordenadas fixo é puramente fictício – todas as operações de rotação ocorrem no sistema global que é aquele no qual o

comando *glRotate* do *OpenGL* de fato opera. Por conseguinte, quando desejarmos rotacionar o universo gráfico em torno do eixo Y, devemos saber qual é o vetor (x', y', z') no sistema global que é coincidente a Y.

Esse vetor pode ser obtido por meio da transformação global → local da classe *auxAxis*. Neste caso, é necessário transformar o vetor (0,1,0) para o sistema global corrente se quisermos rotacionar em torno de Y. Como formar a matriz de transformação de *auxAxis*? Simplesmente coletando os termos da matriz de visualização do modelo (*modelview matrix*) disparando o comando *glGetDouble* do *OpenGL*, tendo como argumento *GL_MODELVIEW_MATRIX*.

O resultado torna a visualização dos objetos no CABEMT uma tarefa simples e rápida, fundamental para auxiliar na extração de informações e criação do modelo em si. Também existe a possibilidade de utilizar o teclado para *zoom* e rotação conforme os comandos no APÊNDICE A.

4.9.2 Salvar e carregar um arquivo

Já vimos que todos os componentes no CABEMT são objetos de classes do programa, como por exemplo: linhas, círculos, sólidos, malhas, etc. Intrinsicamente à programação em Java, tais objetos ficam armazenados na memória RAM do computador por todo o período em que o CABEMT estiver sendo utilizado. Desse modo, podem ser rapidamente acessados e modificados.

Quando o programa é fechado, toda a informação é perdida a menos que haja alguma forma de armazenar esses objetos e seus atributos na memória perene, normalmente aquela do disco rígido ou drives SSD.

Felizmente o Java possui uma maneira conveniente de se fazer isso através das suas classes nativas S*erializable, ObjectInputStream, ObjectOutputSteam, fileInputStream* e *fileOutputStream*. Elas permitem que objetos sejam escritos ou lidos, desde que suas classes implementem a interface *serializable*. No CABEMT isso é feito para todas as classes de entidades gráficas. O processo de salvar e carregar é gerenciado pela classe loadAndSave, conforme esquematizado na figura 4.57.

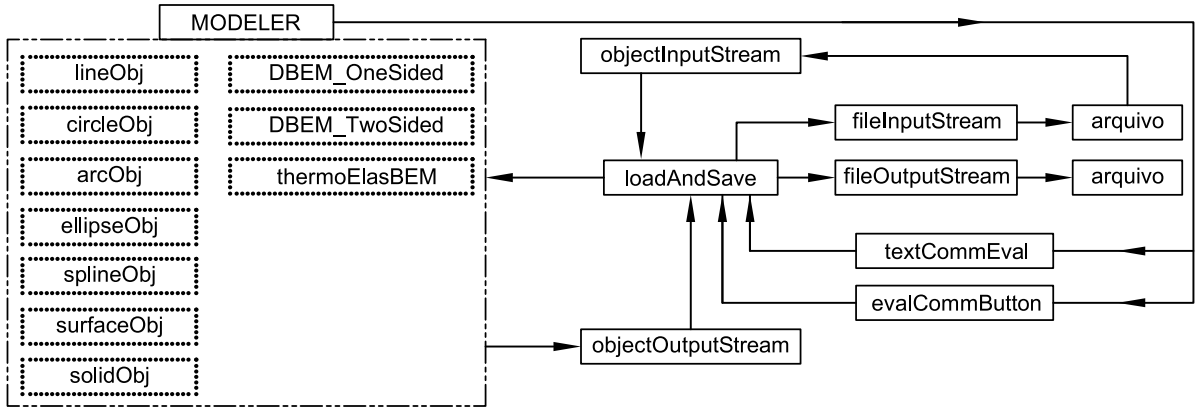
Para salvar, o usuário digita o comando no *prompt* ou clica nos menus. A ação é enviada para a classe *Modeler* que a encaminha para *textCommEval* ou

evalCommButton. O nome do arquivo é então encaminhado para a classe *loadAndSave*. Por sua vez, ela vai invocar o método *writeObject* da classe *objectOutputStream* escrevendo os objetos na lista de *Modeler* em um objeto do tipo *outputStream*. Ele é então transferido para um arquivo por meio de *fileOutputStream*.

Pra carregar um arquivo, os comandos do usuário novamente são lidos por *Modeler* e encaminhados para *textCommEval* ou *evalCommButton*. Daí encaminham-se para loadAndSave. A função *readObject* da classe *fileInputStream* lê as informações no arquivo e as transfere para um objeto da classe *objectInputSteam*. Em seguida são reinterpretados nos diferentes objetos nativos do CABEMT e transferidos às listas em *Modeler,* conforme mostrado na figura 4.57.

A desvantagem do uso de serializable no salvamento de arquivos é que se o escopo da classe for alterado em uma modificação do programa, ocorrerá uma falha ao carregar o arquivo. Em modificações futuras do programa, seria desejável implementar um formato de arquivo em ASCII ou JSON para que pudesse ser aberto em qualquer versão do software.

Figura 4.57 – Diagrama de funcionamento para salvar e carregar arquivos no CABEMT. Caixas pontilhadas são listas.



Fonte: Elaboração do próprio autor

4.9.3 Desfazer e refazer

Ao longo do processo de modelagem, o usuário pode cometer erros ou perceber que seria melhor ter realizado alguma operação anteriormente à atual. O programa, por sua vez, pode entregar resultados errôneos dependendo das entradas fornecidas a seus algoritmos. Por causa desses erros ou necessidade de mudanças/correções em operações passadas, a capacidade de desfazer é muito importante a um *software* de modelagem.

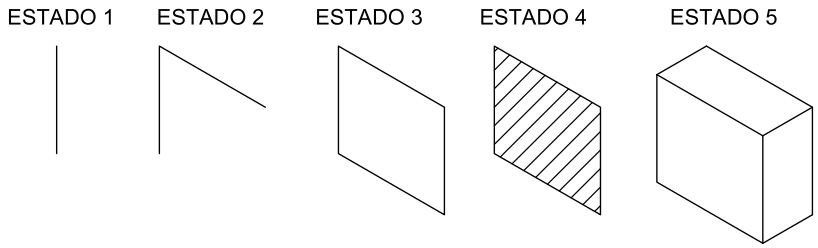
Podemos trabalhar com pelo menos dois métodos de desfazer: por operações reversas ou por resgate de estados:

- Desfazer por operações reversas: consiste em estabelecer funções que retornam uma determinada operação ao seu estado original. Ao adicionar uma linha ao modelo, por exemplo, o CABEMT insere o objeto da classe *line* na lista *lineObj* da classe principal *Modeler.* Desfazer essa operação consiste trivialmente em apagar o elemento adicionado.

  A inversão das operações nem sempre é tão elementar, especialmente quando se trata de modelos sólidos. As operações de topologia geralmente possuem uma inversora. Dividir uma aresta por um vértice, por exemplo, é o inverso de unir duas arestas. Desfazer uma complexa operação booleana utilizando funções inversoras é uma tarefa desafiadora de programação. O *software* GWB criado por Mäntylä (1988) utiliza esse conceito, no entanto, toda a estrutura do programa é baseada em operadores de Euler. O CABEMT utiliza tais operadores somente de maneira indireta, algo que complicaria ainda mais os algoritmos de inversão;

- Desfazer por resgate de estados: essa metodologia trabalha armazenando um número especificado de estados do modelo e invocando-os mediante a solicitação do comando desfazer. Um estado é uma determinada configuração do modelo, ou seja, os objetos que dele fazem parte. Nessa definição pode-se incluir ou não outras características visuais como o ângulo de visão e *zoom* aplicados. A figura abaixo exemplifica os estados de criação de um modelo sólido.

Figura 4.58 – Estados de um modelo no CABEMT



Fonte: Elaboração do próprio autor

Cada estado é salvo em uma lista com objetos da classe *objOutputStream*, a mesma utilizada para salvar arquivos vista no item 4.9.2. Essa lista armazena até vinte estados e se o número a salvar for maior, aqueles mais antigos são perdidos.

Quando o comando desfazer é acionado, todos os objetos atuais são deletados e o último estado na lista (N_s) é carregado. Acionando o comando novamente,

carrega-se o penúltimo ($N_S - 1$), etc. Por outro lado, refazer carrega o estado $N_S + 1$ se ele existir.

No CABEMT os estados são alocados na memória RAM, mas poderiam ser gravados no disco rígido, sob pena de algum sacrifício na fluidez do programa em consequência do elevado número de salvamentos. Estes ocorrem toda vez em que for gerada uma "modificação relevante" na configuração atual. Exemplos de modificações relevantes são: inclusão/exclusão de uma primitiva de modelagem, criação de faces, extrusão, revolução, etc. Uma maneira de se amenizar isso consiste em agrupar várias etapas para então gravar em disco.

A classe dedicada às operações de desfazer e refazer é undoRedo, cujo funcionamento é muito similar à loadAndSave esquematizada na figura 4.57, com a ressalva de que os dados são armazenados em lista na memória RAM ao invés de arquivos. Além disso, visando economia de memória, o recurso de desfazer não se aplica quando o objeto possuir uma malha associada.

4.10 LIMITAÇÕES

Ao se elaborar um programa de complexidade razoável quase sempre haverá elementos que apresentam conflitos entre si (abrangência, custo, esforço dispendido...). Em decorrência desses conflitos, há algumas limitações no *software*, cujas mais relevantes serão listadas a seguir:

- Alinhamentos especiais: a representação de sólidos por meio de poliedros certamente reduziu bastante o custo/tempo necessário à programação do CABEMT. Contudo, essa metodologia introduz algumas dificuldades devido ao número elevado de vértices e arestas gerados. Aumenta-se então a probabilidade de coincidências geométricas que precisam de tratamento específico, especialmente durante as operações booleans. Esses problemas tem sido resolvidos por oportunidade. Além disso, não representam grandes empecilhos, haja vista que uma mudança diminuta no modelo evita essas coincidências e a consequente a falha dos algoritmos, conforme já comentado na figura 4.17;
- Geração de malha tridimensional: a discretização espacial ainda precisa de algumas melhorias, especialmente no que diz respeito à eliminação de elementos de baixa qualidade. Mesmo com a otimização implementada, é comum restar

poucos elementos com qualidade baixa. Somente um algoritmo de eliminação de *slivers* poderá obter melhoria;

- Concordância e chanfro entre superfície: conforme esclarecido nos itens 4.7.4.2 e 4.7.4.3, essas funções apenas trabalham quando a aresta de intersecção entre duas superfícies é comum apenas às faces das mesmas. Porém, na maioria dos casos em que isso não ocorre, é possível inserir essas entidades no modelo por outros meios. Se ainda não for possível, há o recurso de importação de sólidos visto no item 4.7.6. Contudo, é interessante expandir a capacidade de concordância do programa para facilitar a modelagem;
- Aproximação da malha ao sólido: a representação poliédrica impede que o refinamento de malha se ajuste melhor ao sólido “real” porque afinal de contas ele já está sendo aproximado pelo poliedro. A única maneira de contornar isso é ajustando bem as subdivisões dos elementos curvos na fase de traçagem das primitivas.

Aparentemente as limitações mencionadas não comprometem aqueles objetivos iniciais delineados para o CABEMT conforme o item 4.3.

4.11 OUTROS TESTES DO MODELADOR DE SÓLIDOS

Ao longo dessa dissertação já foram mostrados muitos exemplos de sólidos e malhas geradas no CABEMT. Os exemplos a seguir expandem o repertório apresentado e exemplificam algumas situações em que o programa pode ser utilizado.

4.11.1 Modelagem de tubulação

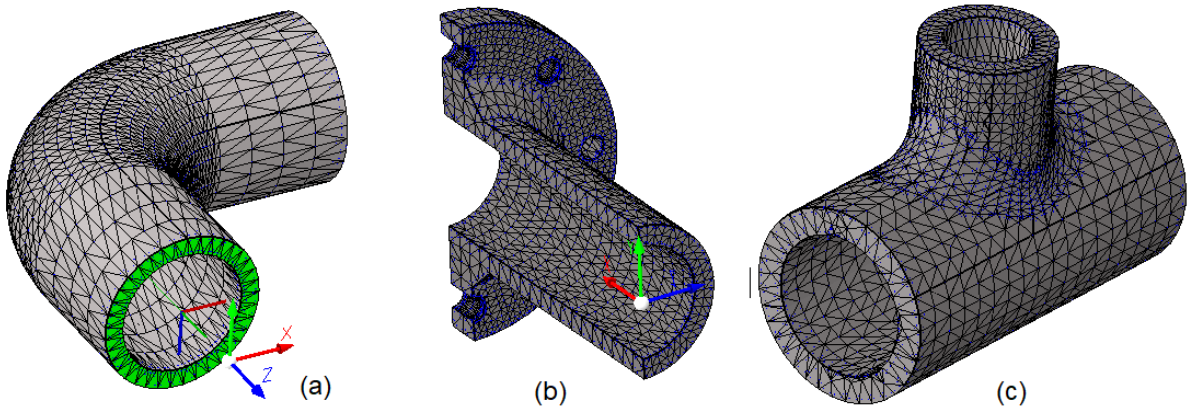
Tubulações são o elemento mais comum em qualquer indústria de processo e estão submetidas a esforços estáticos tais como a pressão interna, peso próprio, tensões induzidas pela expansão térmica e tensões induzidas por gradientes térmicos. Mas também estão sujeitas a esforços dinâmicos devido a vibrações causadas pelo fluxo, por equipamentos às quais se conectam ou ainda a agentes externos como terremotos e colisões diversas.

Por causa dessas solicitações mecânicas, não é incomum encontrar defeitos que requeiram reparos imediatos ou que possam ser planejados. Nesse último, se a avaliação de mecânica da fratura permitir.

Na figura abaixo observam-se alguns elementos típicos de tubulação

modelados no CABEMT.

Figura 4.59 – Alguns elementos de tubulação gerados no CABEMT; a) curva 90°; b) flange e tubo; c) tê de redução



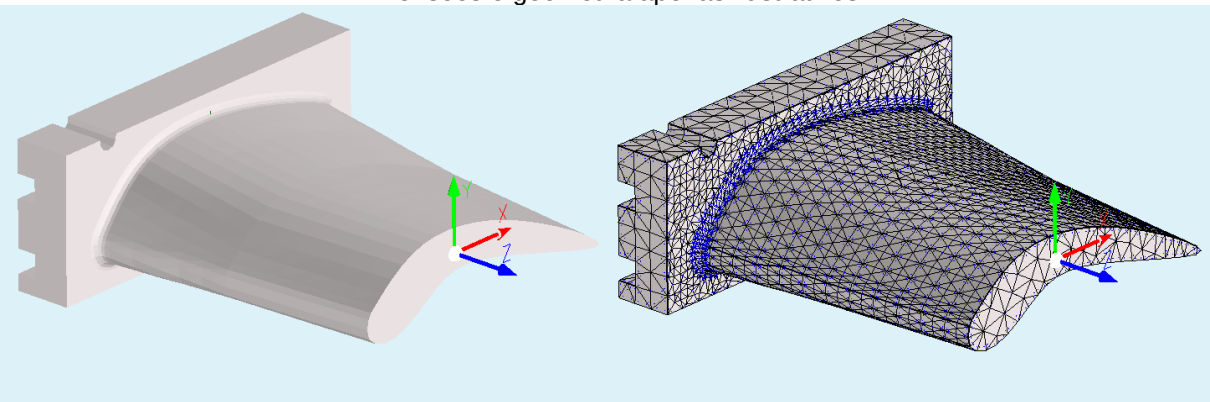
Fonte: Elaboração do próprio autor

Além desses elementos, ver figura 4.65 (c) para um modelo de derivação em "T" recortada em seus planos de simetria.

4.11.2 Modelagem de uma pá de compressor

Os componentes de turbomáquinas sempre estarão sujeitos à fadiga, uma vez que a presença de vibrações é intrínseca a seu funcionamento. Componentes esbeltos como as pás de compressores axiais são um bom exemplo de elemento severamente solicitado nesse tipo de máquina. Ver figura 1.1 para um exemplo de falha provocada pela propagação de trincas de fadiga em uma pá de compressor. O CABEMT tem condições de criar o modelo sólido desse tipo de geometria.

Figura 4.60 - Modelo de uma pá de compressor axial e sua malha. Ambos gerados no CABEMT. Dimensões e geometria apenas ilustrativos



Fonte: Elaboração do próprio autor

Esse modelo foi criado através das funções *loft, spline, extrude* e operações booleanas.

4.11.3 Modelagem de bocais

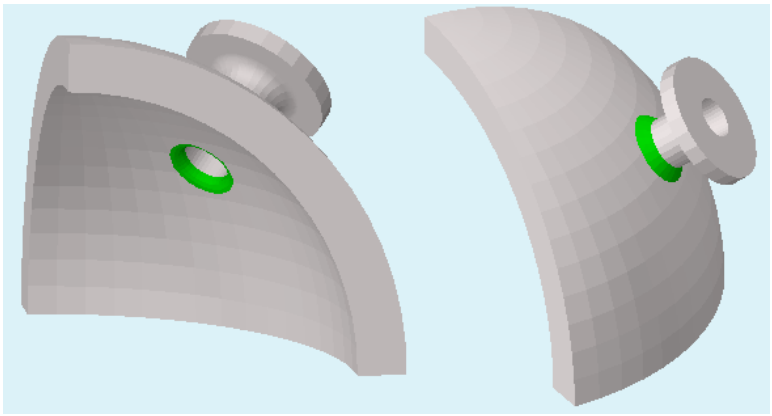
Ao lado das tubulações, os vasos de pressão constituem fração significativa dos equipamentos da indústria de processo. Eles acumulam fluidos sob pressão nas várias etapas do processamento e comunicam-se entre si por meio das tubulações. Quando reações químicas ocorrem em seu interior, são chamados de reatores.

As conexões desses equipamentos são denominadas bocais e podem apresentar várias configurações. Em quase todos os casos, são constituídos por elementos soldados às calotas ou ao casco dos vasos. Esses bocais recebem os esforços de expansão térmica das tubulações, esforços provocados por vibrações e também podem sofrer choques térmicos.

Dado que as soldas dos bocais podem conter defeitos, a atuação dos esforços repetitivos pode fazer com que eles se propaguem e possam vir a representar risco na segurança pessoal e de processo. Por isso, a detecção de trincas através da inspeção é fundamental. No melhor dos casos, os defeitos encontrados não representam ameaça imediata à integridade e podem ser acompanhados nas inspeções periódicas. Essa avaliação é, novamente, realizada através da mecânica da fratura.

Reparos substanciais nas conexões de vasos de pressão, dependendo do tipo de material, podem ser extremamente complexos ou até inviáveis de se realizar em campo.

Figura 4.61 – Duas vistas de um modelo sólido de bocal numa calota hemisférica. Modelos gerados no CABEMT.



Fonte: Elaboração do próprio autor

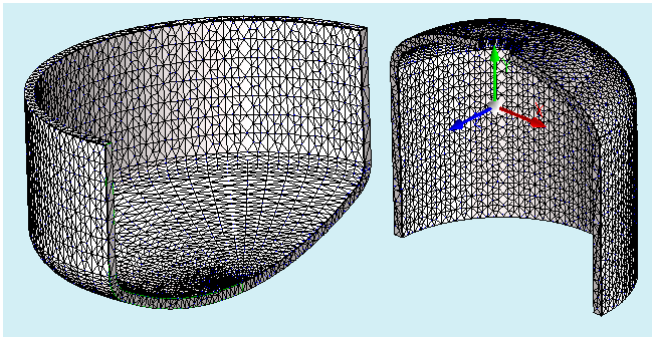
Na figura acima modelou-se um bocal em uma calota esférica, podendo representar um vaso totalmente esférico ou um vaso cilíndrico com calota hemisférica. Internamente, a junção com o bocal sofreu um processo de suavização (concordância)

e externamente foi realizada uma solda de ângulo para reforço.

Na figura 4.62 encontram-se dois modelos parciais de calotas conectadas a seus cascos. Trincas nesses componentes são relativamente comuns em virtude de solicitações termo-mecânicas e/ou mecanismos de danos atuantes.

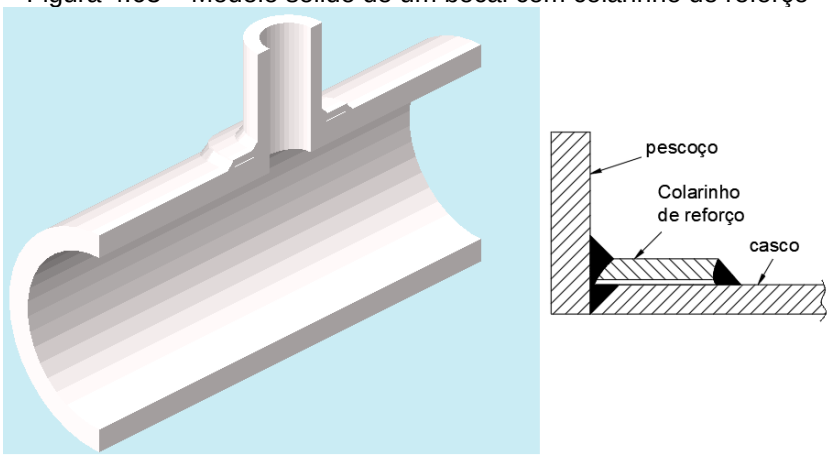
Na figura 4.63 foi modelado um bocal soldado em casco cilíndrico. Esse bocal possui um colarinho de reforço como forma de compensar a área do casco perdida no furo, conforme previsto no ASME VIII. As soldas das juntas em ângulo também foram inclusas no modelo.

Figura 4.62 – Tampo + casco toriesférico e, à direita, tampo + casco semi-elíptico modelados no CABEMT



Fonte: Elaboração do próprio autor

Figura 4.63 – Modelo sólido de um bocal com colarinho de reforço



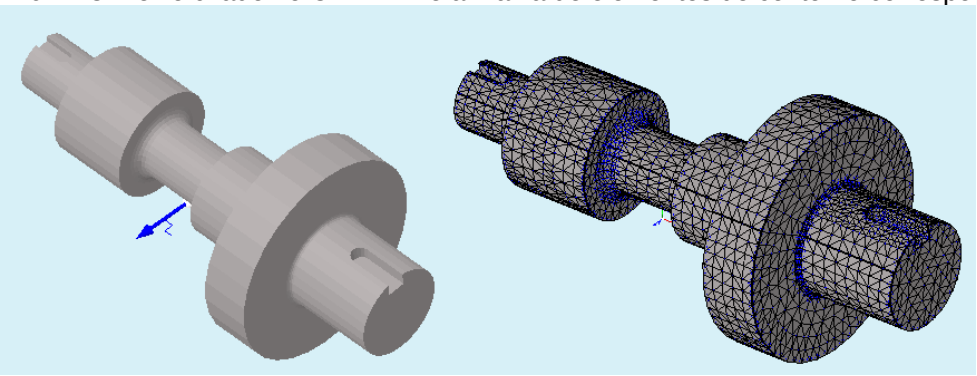
Fonte: Elaboração do próprio autor

4.11.4 Modelagem de um eixo

Eixos são componentes fundamentais de praticamente qualquer máquina. Sofrem principalmente esforços de cisalhamento e de flexão, este último quase sempre de maneira cíclica, propício a nuclear e propagar trincas. A figura a seguir

mostra um eixo genérico criado no CABEMT.

Figura 4.64 – Um eixo criado no CABEMT e a malha de elementos de contorno correspondente



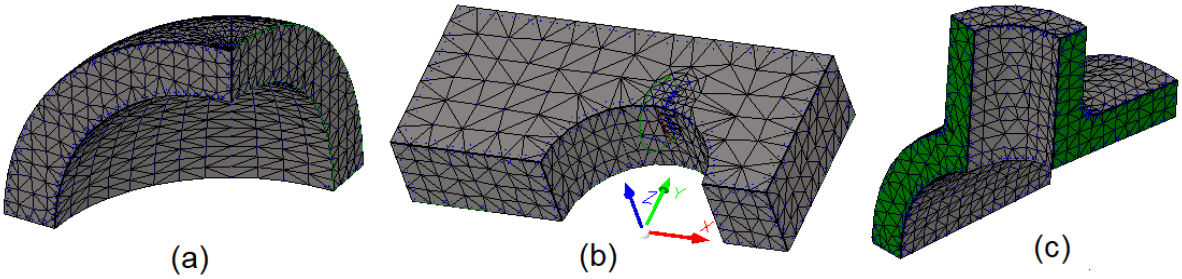
Fonte: Elaboração do próprio autor

4.11.5 Alguns exemplos de malhas geradas no CABEMT

4.11.5.1 Malhas em superfícies

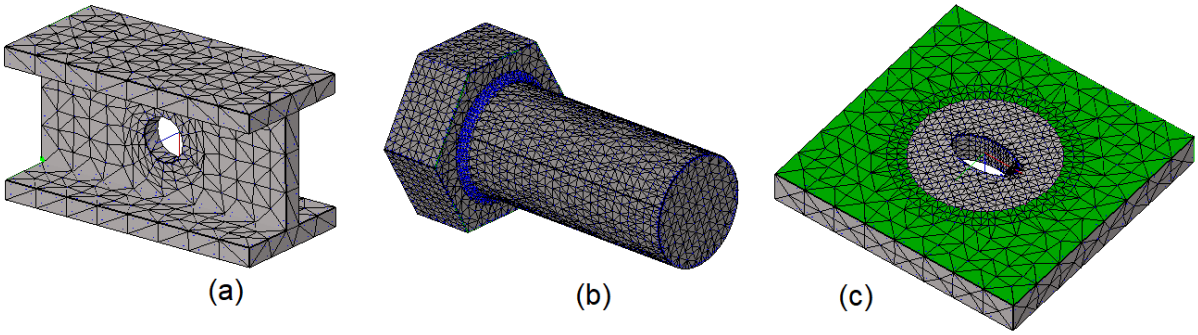
Na figura 4.65 estão alguns sólidos gerados no CABEMT após sua discretização.

Figura 4.65 – Exemplos de malhas de elementos de contorno geradas no CABEMT: a) um oitavo de vaso esférico; b) olhal com uma trinca na furação; c) bocal de casco cilíndrico



Fonte: Elaboração do próprio autor

Figura 4.66 – Modelos sólidos e malhas geradas pelo CABEMT: a) viga com furo na alma; b) um parafuso, notar concordância na junção cabeça-corpo; c) uma placa com furo elíptico - foi utilizada a segmentação de superfícies para refinar a malha na região de interesse



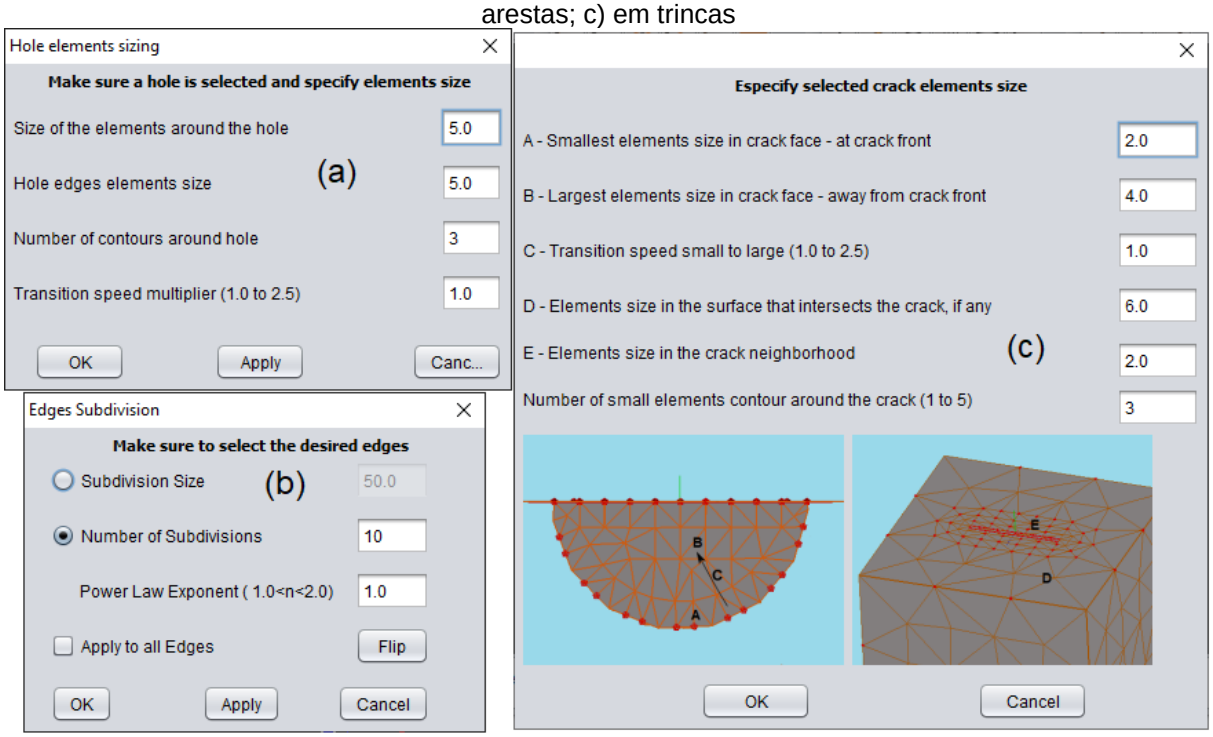
Fonte: Elaboração do próprio autor

Na figura 4.66 (a) o algoritmo de expansão de fronteira gerou elementos

bastante uniformes no entorno do furo. Na figura (b) um parafuso de cabeça sextavada foi discretizado com uma malha bastante fina. Por último, na figura (c), a segmentação de superfícies foi utilizada para gerar uma malha refinada no entorno do furo elíptico, permitindo-a ser mais grosseira nas demais regiões. Isso é bastante útil quando se tem domínios grandes e as regiões de gradientes de tensão, que exigem malhas mais refinadas, são relativamente pequenas.

Alguns exemplos de caixas de diálogo que permitem ao usuário atribuir as características de subdivisão desejadas encontram-se na figura 4.67. Há inclusive figuras auxiliares na caixa de diálogo para ajudar o usuário ajustar os tamanhos de malha nas faces de trinca e ao redor, como se pode ver na figura 4.67 (c). A subdivisão das arestas pode ser feita pela entrada do número ou do tamanho das divisões, pode-se ainda usar um fator para deixar o espaçamento não uniforme – vide figura 4.67 (b).

Figura 4.67 – Caixas de diálogo para seleção de tamanho de malha no CABEMT: a) em furos; b) em arestas; c) em trincas

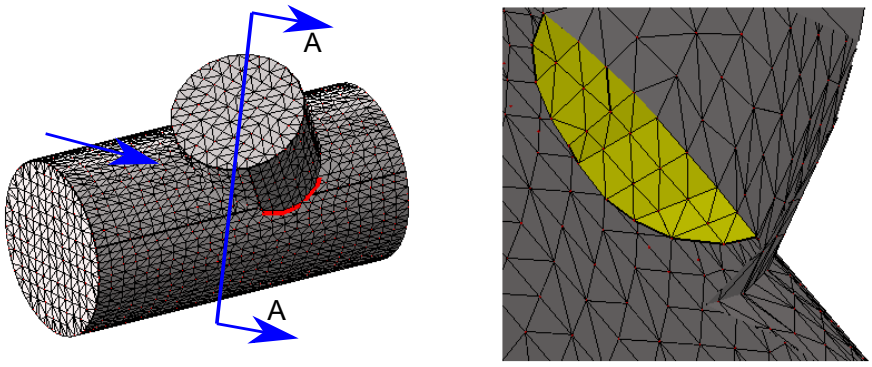


Fonte: Elaboração do próprio autor (CABEMT)

4.11.5.2 Malhas em trincas

A geração de trincas diretamente a partir do modelo sólido (método convencional) já foi apresentada nas figuras do item 4.7.5. Outra forma de gerar o defeito é por meio da seleção de arestas dos elementos da malha que correspondem ao afloramento da trinca, conforme ilustrado na figura a seguir:

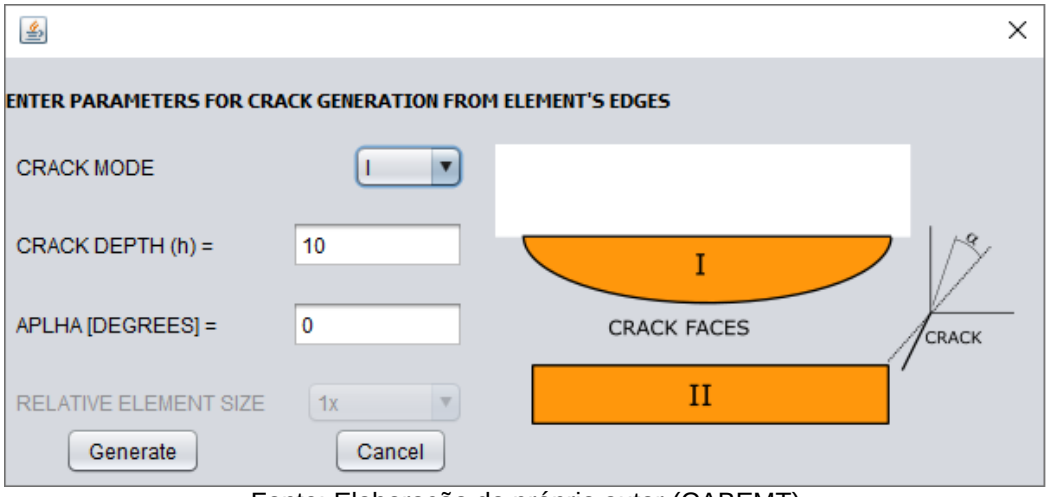
Figura 4.68 – Criação de uma trinca em aresta de entroncamento oblíquo entre dois cilindros. A linha vermelha são as arestas de afloramento selecionadas



Fonte: Elaboração do próprio autor (CABEMT)

A modelagem desse defeito seria muito trabalhosa se fosse utilizado o método convencional, isto é: fazer a intersecção entre o sólido e uma superfície que contém geometricamente a trinca. Por outro lado, leva poucos segundos para a geração do defeito através da seleção de arestas. A definição de parâmetros na tela de comando do CABEMT – acessada pelo menu Build → Crack → From Sel. Elem. Edges – é mostrada abaixo:

Figura 4.69 – Tela do CABEMT: criação de trincas através da seleção de arestas de elementos que definem seu afloramento



Fonte: Elaboração do próprio autor (CABEMT)

5 DISCUSSÃO E COMENTÁRIOS

Ao longo do desenvolvimento do CABEMT, além dos testes mostrados nessa dissertação, foram realizados muitos outros para verificar o correto funcionamento do programa. Não sendo viável apresentar todos esses testes, é porém factível tecer alguns comentários gerais acerca das principais disciplinas envolvidas neste trabalho, realizando uma análise crítica a seu respeito.

A MCS é indispensável em virtualmente qualquer aplicação prática dos métodos numéricos. Sem ela, a geração de malha teria que ser efetuada manualmente – uma tarefa enfadonha, propensa a erros e próxima do impossível para domínios de geometria não elementar.

Conforme comentado anteriormente nesse documento, a literatura brasileira de MCS é relativamente esparsa, provavelmente devido aos seguintes pontos:

- É uma disciplina aparentemente consolidada, com poucas possibilidades de melhoria e desenvolvimento, portanto não trazendo interesse aos pesquisadores;
- O “custo” de implementação de algoritmos robustos de MCS pode ser bastante elevado devido ao fato de ser necessário levar em conta diversos alinhamentos e situações especiais que devem ser tratados individualmente, conforme já mencionado anteriormente.

Apesar dessas questões, ainda existem áreas em aberto na MCS, principalmente quanto a sua potencial associação a técnicas de tratamento de imagem e reconhecimento de padrões, procurando assim criar modelos a partir de reconstrução de conjuntos de fotografias ou leituras sistemáticas de distâncias. Isso pode ter significativa relevância prática.

A modelagem por meio de poliedros no CABEMT, apesar de esteticamente não muito elegante, tem razoável flexibilidade e poder representativo de elementos da prática de engenharia. Essa técnica é conveniente para os métodos numéricos, já que na maioria das vezes, a discretização dos elementos culminará em um tipo de aproximação poliédrica do modelo[48]. De qualquer modo, a forma de representar os

[48] Tem-se notado um esforço na literatura técnica de MEC a tendência do uso de métodos isogeométricos que se aproveita da representação de superfícies via NURBS. O CABEMT poderá

sólidos no CABEMT é apropriada ao CAE e até mesmo para o CAM[49].

Ademais, a base teórica para a MCS no estado da arte, com uso de quádricos e NURBS, compartilha muito com a base para a representação poliédrica, de tal forma que uma extensão de funcionalidades de um programa que lida com modelagem puramente poliédrica é totalmente plausível.

A funcionalidade de MCS no CABEMT é fundamental para a realização do objetivo de o programa ser autossuficiente, não dependendo assim da importação de modelos elaborados por terceiros, por conseguinte, também facilita o uso do programa.

futuramente adotar uma representação híbrida poliedral-NURBS de forma a aproveitar os algoritmos já elaborados ao passo que possibilitará o emprego dessa nova linha de pesquisa no programa.

[49] Um formato bastante utilizado em sistemas CAM é o STL (SZILVÁSI-NAGY e MÁTYÁSI, 2003). Nesse formato, as superfícies são representadas com facetas triangulares, portanto, totalmente compatível com a representação poliédrica.

6 MELHORIAS FUTURAS DO CABEMT – MODELAGEM DE SÓLIDOS

A programação de um *software* parece uma tarefa interminável devido ao número elevado de possiblidades de criação de melhorias ou de algoritmos totalmente novos. Serão assinaladas apenas as linhas de desenvolvimento mais relevantes que poderão ser incorporadas ao programa. Para cada item será estimado qualitativamente o custo benefício de implementar a melhoria.

No que tange ao CABEMT, as linhas de desenvolvimento sugeridas são:

- Implementar a capacidade de representação de sólidos via NURBS e/ou quádricos. Objetivo – conferir representação mais realista aos sólidos. A migração de modelagem poliédrica para NURBS/quádricos pode ser feita aproveitando grande parte dos códigos existentes. Pode-se ainda optar por um caminho intermediário. Nesse caso, as entidades do modelo poliédrico conteriam informações acerca de sua origem e o modelo, após concluído, poderia ser convertido à representação por NURBS/quádricos. Para as operações booleanas poder-se-ia aproveitar o código existente se as superfícies em NURBS/quádricos fossem tesseladas novamente em poliedros. O custo benefício dessas implementações parece baixo;
- Se a modificação acima fosse realizada, seria possível melhorar a distribuição dos pontos antes de se realizar a geração de malha. A parametrização intrínseca aos quádricos e NURBS poderia ser usada nesse sentido;
- Concluir a implementação do algoritmo de geração de malha por divisão e conquista. Objetivo: reduzir o tempo de geração de malha. O custo benefício é baixo porque a geração de malha do programa já tem uma boa velocidade sendo capaz de gerar malhas com milhares de elementos em segundos;
- Ampliar a capacidade de importar malhas parece ter um bom custo benefício, pois são algoritmos fáceis de se criar e o usuário pode querer aproveitar uma malha discretizada em outro programa e manipulá-la no CABEMT;
- Criar funções para aumentar a capacidade de intervenção sobre a geometria e a topologia, pois é algo que os programas disponíveis costumam limitar bastante.

7 CONCLUSÃO

A mecânica da fratura é disciplina que oferece grande potencial de ganhos financeiros e de segurança quando empregada na manutenção e/ou projeto. No entanto, para sua aplicação em problemas práticos de engenharia, muitas vezes é necessário lançar mão de métodos numéricos.

Dependendo do método, a geração do modelo e da sua discretização pode ser acentuadamente difícil. Por outro lado, a incorporação de defeitos planares é bastante trivial no MEC, fazendo-o a técnica ideal para esse tipo de análise. A menos que o objeto de estudo seja muito simples – algo difícil de encontrar na prática de engenharia[50] – a geração de malha só poderá ser construída se antes existir uma representação adequada do objeto. Somente a modelagem computacional de sólidos pode fornecer uma descrição completa e unívoca[51] da geometria e topologia do objeto.

A técnica de modelagem de sólidos por poliedros se mostra bastante compatível com o MEC e tem poucas restrições para modelar estruturas encontradas na prática de engenharia. Além disso, sua implementação é mais simples que as representações mais sofisticadas. Tal simplicidade, no entanto, não indica perda relevante de seu potencial representativo e menos ainda da sua capacidade de fornecer malhas adequadas aos métodos numéricos. Essa metodologia, entretanto, pode não ser adequada quando se é necessário fazer discretizações muito finas de superfícies curvas por causa dos vários recortes nas faces que podem trazer dificuldades numéricas. Há também a exigência de mais memória, exceto se comparada aos métodos de decomposição espacial. Não se espera, entretanto, a necessidade de discretizações tão refinadas para a maioria dos problemas práticos, pelo menos para o MEC.

A MCS por poliedros pode servir de base para implementações mais modernas por NURBS e quádricos já que aquela, no limite, aproxima-se das superfícies curvas e suaves quando se aumenta o número de divisões. Requer-se, porém, refinar os algoritmos para avaliar situações em que muitos pontos podem estar próximos e que

[50] Quando se precisa lançar mão de uma simulação numérica é provável que o problema seja suficientemente complexo.
[51] Assumindo que o objeto é um *manifold*

podem gerar erros devido à precisão finita da representação digital.

Mostrou-se a viabilidade de modificadores locais do tipo Minkowski, tais como concordância e chanfro na representação poliédrica. Mostrou-se também que a inclusão de um defeito planar no modelo sólido pode ser feita aproveitando-se os algoritmos de operações booleanas implementados.

O fato do MEC, na maioria dos casos, necessitar apenas de discretização na superfície do sólido traz imensas vantagens não só na simplicidade do algoritmo de geração de malha em si, mas também na sua otimização. A qualidade da discretização pode ser avaliada até mesmo por inspeção visual, algo praticamente impossível em malhas tridimensionais tetraédricas. De fato, a inclusão de manipuladores locais da malha no CABEMT, permite que o usuário adeque facilmente as regiões que julgar necessário.

A triangulação de Delaunay não é capaz de gerar malhas tridimensionais otimizadas. É necessário empregar algum método de otimização posteriormente. Ainda assim, há risco de que elementos de baixa qualidade (*slivers*) permaneçam na malha, porém em pequena quantidade. Somente a adição estratégica de novos nós tem potencial para eliminar, senão todos, pelo menos a maioria desses elementos desfavoráveis.

Elaborou-se uma maneira de inclusão de trinca no modelo já discretizado. Técnica que confere enorme flexibilidade na modelagem de defeitos. Ressalta-se que esse método só pode ser empregado em malhas superficiais, mais uma vantagem do MEC.

O modelador de sólidos do CABEMT e o gerador de malhas, quando comparado a programas profissionais mostra-se mais lento e apresenta menos recursos. A menor velocidade não é um problema significativo, trata-se no pior dos casos, de minutos de diferença. A segunda desvantagem é mais séria, mas o CABEMT tem por objetivo tratar sólidos de complexidade baixa a moderada. Para elementos muito complicados, pode-se importá-los diretamente para o CABEMT.

Pode-se dizer que o programa desenvolvido confere uma robusta, ainda que simples, base de modelagem sólida e que o projeto pode agora evoluir para o tratamento numérico e simulação via MEC. Melhorias e modificações na geração de malha 3D poderão estender seu uso para o MEF também.

Nos cursos de engenharia em geral os sistemas CAD são tratados como uma caixa preta na qual as etapas lógico-matemáticas envolvidas na geração dos modelos são totalmente omitidas. Espera-se que esse trabalho, em sua dimensão didática, possa tornar mais claras as inúmeras manipulações computacionais implementadas em sistemas CAD. Saber como esses sistemas funcionam pode incentivar a criação de novos softwares em projetos de pesquisa ou de aplicação direta na indústria. Dada a velocidade dos computadores atuais, a modelagem computacional de sólidos e os métodos numéricos, sem dúvida, terão papel cada vez mais relevante na indústria, quer seja no projeto, nas obras ou na manutenção.

REFERÊNCIAS

ALIABADI, M. H. F. **The boundary element method. Volume 2:** Applications in solids and structures. Chichester: John Wiley & Sons, 2002. v. 2.

AL-QAHTANI, S. S. *et al*. Comparing Selected Criteria of Programming Languages Java, PHP, C++, Perl, Haskell, AspectJ, Ruby, COBOL, Bash Scripts and Scheme - Revision 1.0. **CoRR**. [S.l.], p. 149. 2010. Disponível em: http://arxiv.org/abs/1008.3434. Acesso em: 10 jun. 2020.

ANDERSON, T. L. **Fracture Mechanics:** Fundamentals and Applications. 3. ed. Boca Raton: CRC Taylor & Francis, 2005.

ASME & API. **API-579 Fitness-For-Service**. [*S.l.*]: ASME International & American Petroleum Institute, 2016. 1320 p.

AUBRY, J. P. **Beginning with Code_Aster:** A Practical Introduction to Finite Element Method Using Code_Aster, Gmsh and Salome. Paris: FramaBook, 2013.

ARUOBA, S. B.; VILLAVERDE, J. F. A Comparison of Programming Languages in Economics. **Journal of Economic Dynamics and Control**, Amsterdam, v. 58, n. 20263, p. 265-273. 2015.

BATHE, K. J. **Finite Element Procedures**. 2ª. ed. Watertown: Prentice Hall, 2014.
BATISTA, V. H. F. **Geração de Malhas Não-Estruturadas Tetraédricas utilizando um método de avanço de fronteira**. 2005. 85 f. Dissertação (Mestrado em Engenharia Mecânica) – Faculdade de Engenharia, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2005.

BAUMGART, B. G. **Geometric Modeling For Computer Vision**. Palo Alto: Universidade de Stanford, 1974.

BECHT ENGINEERING. **About Becht FFS Program**, 2020. Disponivel em: https://bechtffs.com/about-the-program. Acesso em: 17 jan. 2020.

BENVENUTO, E. **An Introduction to the History of Structural Mechanics**. Nova Iorque: Springer-Verlag, 1991.

BERN, M.; PLASSMANN, P. Mesh Generation. **Handbook of Computational Geometry**, Amsterdam, 2000. p. 291-332.

BREBBIA, C. A. The birth of the boundary element method from conception to application. **Engineering Analysis with Boundary Elements**, Oxford, v. 77, n.1, p. 3-10, 2017.

BRITISH STANDARDS. **BS-7910 Guide to methods for assessing the acceptability of flaws in metallic structures**. [*S. l.*]: BSI Standards Publication. 2013.

BROEK, D. **Elementary Engineering Fracture Mechanics**. 3. ed. Hague: Martinus Nijhoff Publishers, 1984.

BRONSHTEIN, I. N. *et al*. **Handbook of Mathematics**. 6. ed. [*S. l*.]: Springer, 2015.

BRUTTO, M. Lo.; MELI, P. Computer vision tools for 3D modelling in archaeology. **International Journal of Heritage in the Digital Era**, Thousand Oaks, v. 1, n.1, p. 1-6, 2012.

CHANDWANI, R.; TIMBRELL, C. M.; WIEHAHN, M. A. Crack Modelling In Power Plant Components. **Remaining Life Assessment of Aged Components in Thermal Power Plants and Petrochemical Industries**, Jamshedpur, v. 2, n. 1, p. 155-167, 2008.

CHEW, L. P. Guaranteed-quality mesh generation for curved surfaces. *In*: ANNUAL SYMPOSIUM ON COMPUTATIONAL GEOMETRY, 93., 1993, Nova Iorque. **Procedings** […] New York: Association for Computing Machinery, 1993, p. 274-280.

CITARELLA, R. *et al*. Thermo-mechanical crack propagation in aircraft engine vane by coupled FEM-DBEM approach. **Advances in Engineering Software**, Londres, v. 67, n. 1, p. 57-69, 2014.

CORNELL FRACTURE GROUP. **Franc3D: Concepts & Users Guide**. Ithaca: [*s. n*.], 2003. Disponível em: https://bpb-us-w2.wpmucdn.com/ sites.coecis.cornell.edu/ dist/6/47/files/2016/05/F3D_Menu_Dialog_CFG_V2.6-2k9eodp.pdf. Acesso em: 20 jan. 2020.

DAI, C.; LIU, H. L.; DONG, L. A Comparison of Objective Functions of Optimization-Based Smoothing Algorithm for Tetrahedral Mesh Improvement. **Journal of Theoretical and Applied Mechanics**, Varsóvia, v. 52, n. 1,p. 151-163, 2014.

DU, T. *et al*. Inverse CSG: Automatic Conversion of 3D Models to CSG Trees. **ACM Transactions on Graphics**, New York, v. 17, n. 6, p. 1-16, 2018

DWYER, R. A. A Faster Divide-and-Conquer Algorithm for Constructing Delaunay Triangulations. **Algorithmica**, New York, v. 2, n. 1, p. 137-151, 1987.

EDELSBRUNNER, H. Triangulations and Meshes in Computational Geometry. **Acta Numerica**, Cambridge, v. 9, n. 1, p. 133-213, 2000.

ERTEN, H.; ÜNGÖR, A. Triangulations with Locally Optimal Steiner Points. **SIAM Journal on Scientific Computing**, Philadelphia, v. 31, n. 3, p. 2103-2130, 2009.

FREY, P. J.; MARECHAL, L. Fast Adaptative Quatree Mesh Generation. *In*: Proceedings of the 7th International Meshing Roundtable, [s.n.], 1998, **Proceedings […],** Rocquencourt: US Department of Energy, 1998, p. 211-224.

GEUZAINE, C.; REMACLE, J. F. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. **International Journal for Numerical Methods in Engineering**, Hoboken, v. 79, n. 11, p. 1309-1331, 2009.

GOURAUD, H. **Computer display of curved surfaces**. 1971, 80 f. Tese (Doutorado em Computação) – Faculdade de Computação, Universidade de Utah, Salt Lake City, 1971.

GUIBAS, L.; STOLFI, J. Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams. **ACM Transactions on Graphics (TOG)**, New York, v. 4, n. 2, p. 74 -123, 1985.

HOFFMANN, C. M. **Geometric and Solid Modeling**. [*S.l.: s.n*.], 1992. Disponivel em: <https://www.cs.purdue.edu/homes/cmh/distribution/books/geo.html>. Acesso em: 16 jan. 2020.

HUMER, S. J.; FOSTER, E. C. **A Comparative Analysis of the C++, Java, and Python Languages**. Keene State College. Keene. 2014. Disponível em: https://www.elcfos.com/data/papers-in-cs/CS-Paper05_PPL- ProjectA_HumerFoster_1491251202.pdf. Acesso em: 14 jul. 2021.

JACKIE, N.; DAVIS, T.; DIXON, M. **OpenGL Programming Guide:** The Official Guide to Learning OpenGL: Version 1.1. Boston: Addison-Wesley, 1997.

JANSEN, F. W. **Solid Modelling with Faceted Primitives**. Amsterdã: Delft University Press, 1987.

JONES, D. R. H. (Ed.). **Failure Analysis Case Studies II**. Kidlington: Pergamon, 2001.

KAVOURAS, M.; SMART, J. Solid Modeling in geology and mining. **International journal of surface mining, reclamation and environment**, Abington, v. 3, n. 1, p. 43-47, 1989.

KHOIROM, S.; SONIA, M.; LAIKHURAM, B; LAISHRAM, J.; SINGH, T. D. Comparative Analysis of Python and Java for Beginners. **International Research Journal of Engineering and Technology**, Tamilnadu, v. 7, n. 8, p. 4384-4407, 2020.

LAHAM; AL, S. **Stress Intensity Factor and Limit Load Handbook**. [*S. l*.]: British Energy Generation, 1998.

LEE, D. T.; SCHACHTER, B. J. Two algorithms for constructing a Delaunay triangulation. **International Journal of Computer & Information Sciences**, Amsterdam, v. 9, n. 1, p. 219-242, 1980.

LISCHINSKI, D. Incremental Delaunay Triangulation. *In*: Graphics gems IV, 4., 1994, San Diego. **Conferência[…]** San Diego: Academic Press Professional Inc.. 1994. p. 47-59.

LO, S. H. Optimization of Tethaedral Meshes Based on Element Shape Measures. **Computers & Structures**, Oxford, v. 63, n. 5, p. 951-961, 1997.

LOGAN, D. L. **A First Course in the Finite Element Method**. 4. ed. [*S. l.*]: Thomson, 2007.

LEORDEAN, D.; VILAU, C.; DUDESCU, M. C. Generation of Computational 3D Models of Human Bones Based on STL Data and CAD Software Packages. **Applied Sciences**, Basiléia, v. 11, n. 17, p. 1-15, 2021.

MA, Y.; WANG, M. An Efficient Method to Improve the Quality of Tetrahedron meshes with MFRC. **Scientific Reports**, [*s. l.*], v. 11, n. 1, [S.I.], 2021.

MÄNTYLÄ, M. **An Introduction to Solid Modeling**. Rockville: Computer Science Press, 1988.

MORASSO, P. Solid modeling in robotic vision. **Computers in industry**, Amsterdam, v. 7, n. 3, p. 227-232, 1986.

MOURA, A. L. **Uma Proposta para a Triangulação de Delaunay 2D e Localização Planar de Pontos em Ocaml**. 2006. 115 f. Tese (Doutorado em Engenharia) – Universidade Federal de Uberlândia, Uberlândia, 2006.

NANZ, S.; FURIA, C. A. A Comparative Study of Programming Languages in Rosetta Code. *In*: IEE International Conference on Software Engineering, 2015 Florença. **Procedings** […], Florença: IEE, 2015, p. 778-788.

PÉBAY, P. P.; BAKER, T. J. Analysis of Triangle Quality Measures. **Mathematics of Computation**, Providence, v. 72, n. 244, p. 1817-1839, 2003.

PELLICIONE, A. D. S. *et al*. **Análise de Falhas em Equipamentos de Processo:** Mecanismos de Danos e Casos Práticos. Rio de Janeiro: Interciência, 2012.

PETERSON, S. **Computing Constrained Delaunay Triangulations in the Plane,** 1998. Disponivel em: http://www.geom.uiuc.edu/~samuelp/del_project.html. Acesso em: 6 fev. 2020.

PIEGL, L.; TILLER, W. **The NURBS book**. 2ª. ed. Berlin: Springer, 1997.

REQUICHA; G., A. A.; ROSSIGNAC, J. R. Solid Modeling and Beyond. **Computer Graphics and Applications**, Washington, v. 12, n. 5, p. 31-44, 1992.

RIBES, A.; BRUNETON, A.; GEAY, A. **SALOME: an Open-Source simulation platform integrating ParaView**, 2017. Disponível em: https://doi.org/10.13140 /RG.2.2.12107.08485. Acesso em: 16 ago. 2021.

RICE, J. R. **The Aspect Ratio Significant for Finite Element Problems**. Purdue University. West Lafayette, p. 14, 1985. Disponível em: https://docs.lib.purdue.edu/cgi/viewcontent.cgi?referer=&httpsredir=1&article=1453&context=cstech. Acesso em: 5 nov. 2019.

ROSSIGNAC, J.; REQUICHA, A. A. G. **Solid Modeling**. [*S. l.*]: Georgia Institute of Technology, 1999. Disponível em: https://smartech.gatech.edu/bitstream/handle/1853/3375/99-09.pdf. Acesso em: 20 nov. 2019.

RUPPERT, J. A new and Simple Algorithm for Quality 2-Dimensional Mesh Generation. *In*: Proceedings of the Fourth Annual ACM Symposium on Discrete Algorithms, 1993, Austin. **Proceedings** […], Philadelphia: Society for Industrial and Applied Mathematics, 1993, p. 83-92.

SANTOS, R. J. **Um Curso de Geometria Analítica e Álgebra Linear**. Belo Horizonte: Imprensa Universitária da UFMG, 2007.

SCHIARA, L. D. S.; RIBEIRO, G. O. Finite element mesh generation for fracture mechanics in 3D coupled with ansys®: elliptical cracks and lack of fusion in nozzle welds. **Journal of the Brazilian Society of Mechanical Sciences and Engineering**, Rio de Janeiro, v. 38, n. 1, p. 253-263, 2016.

SHEWCHUK, J. R. Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. **Applied Computational Geometry: Towards Geometric Engineering**, Berlin, v. 1148, p. 203-222, 1996.

SHREINER, D. *et al*. **OpenGL Programming Guide**. 8. ed. Ann Arbor: Addison-Wesley, 2013.

SIBSON, R. Locally equiangular triangulations. **The Computer Journal**, Bath, v. 21, n. 1, p. 243-245, 1978.

STEIN, M.; GEYER-SCHULZ, A. A Comparison of Five Programming Languages in a Graph Clustering Scenario. **Journal of Universal Computer Science**, [*S. l.*], v. 19, n. 3, p. 428-456, 2013.

STEWART, J. **Cálculo**. 5. ed. São Paulo: Thomson Learning, 2006 (Volume II).

STROUD, I.; NAGY, H. **Solid Modeling and CAD Systems:** How to Survive a CAD System. 1. ed. Londres: Springer, 2011.

SZILVÁSI-NAGY, M.; MÁTYÁSI, G. Analysis of STL Files. **Mathematical and Computer Modelling**, Abingdon, v. 38, p. 945-960, 2003.

TWI GLOBAL. **New Version of CrackWise Software**, 2002. Disponivel em: <https://www.twi-global.com/media-and-events/connect/2002/march-april-2002/new-version-of-crackwise-software>. Acesso em: 17 jan. 2020.

VARADHAN, G.; MANOCHA, D. Accurate Minkowski sum approximation of polyhedral models. **Graphical Models**, [S.l.], v. 68, n. 4, p. 343-355, 2006.

VENART, J. E. S. Flixborough: the explosion and its aftermath. **Process safety and environment protection**, Rugby, v. 82, n. 2, p. 105-127, 2004.

WEATHERILL, N. P.; HASSAN, O. Efficient 3-dimensional Delaunay triangulation with automatic point generation and imposed boundary constraints. **Int. J. Numer. Meth. Eng.**, Hoboken, v. 37, n. 12, p. 2005-2039, 1994.

ZIENKIEWICZ, O. C.; TAYLOR, R. **The Finite Element Method Set**. 6. ed. [*S. l.*]: Elsevier, 2005.

APÊNDICE A LISTA E DESCRIÇÃO DE CLASSES DO CABEMT

CLASSE	FUNÇÕES E/OU CARACTERÍSTICAS BÁSICAS
Arc	Criação, manipulação e representação de arcos
Arrow_3D	Criação de setas tridimensionais multipropósitos
Boolean_Operators	Executa operações booleanas entre dois sólidos
Boolean_OperatorsSurf	Executa operações booleanas entre sólidos e superfícies
Box	Criação de caixas para sinalizações diversas
Calculator	Classe com diversas funções de cálculo não especializadas
Calculator2	Classe com diversas funções de cálculo não especializadas
Cells	Célula de um octree
Circle	Criação, manipulação e representação de círculos
Complex	Classe para tratamento de números complexos
CplxMatrix	Classe para tratamento de matrizes de números complexos
CrackGeometry	Cria trincas internas e planares
CrackGeometry2	Cria trincas curvas
CrackGeometry3	Cria trincas a partir de arestas selecionadas
CrackSizing	Caixa de diálogo para dimensionamento da malha em trincas
CubicSpline	Criação, manipulação e representação de splines via NURBS
Dots	Plota pontos na tela – multipropósito
Drawer	Classe que emite os comandos do OpenGL para realizar a exibição gráfica
Edge	Aresta de um sólido
EdgeSizing	Caixa de diálogo de segmentação de arestas para geração de malha
Element	Elemento de contorno e todos os parâmetros relativos a ele
Ellipse	Criação, manipulação e representação de elipses
Face	Estrutura de representação de uma face – entidade geométrica e topológica
Face Sizing	Caixa de diálogo de segmentação de faces para geração de malha
GeometricObject	Classe abstrata que contém as demais relativas a objetos geométricos (linhas, círculos, arcos, etc)
GlobalVar	Armazena as variáveis comuns (globais) do programa
Hole	Estrutura de representação de um furo
HoleSizing	Caixa de diálogo de segmentação de arestas de furos para geração de malha
Legend	Cria e controla exibição de legendas e paleta de cores
Line	Criação, manipulação e representação de linhas
Magnet	Criação de entidades auxiliares de desenho – atratores
MatPropDiag	Caixa de diálogo para atribuição ou modificação de propriedades do material
Mesh	Gera e representa uma malha
meshCell	Gera e representa uma malha volumétrica (composta de células)

MeshDivAndConquer	Geração de malhas por divisão e conquista – em desenvolvimento
meshQuadTree	Contém funções para geração de malha por *quadtree*
modMesh	Contém funções para modificação da malha
modMeshCell	Contém funções para modificação da malha volumétrica
Modeler	Classe principal que controla todas as outras
ModelerAuxFunc	Abriga algumas funções auxiliares da classe principal (Modeler)
Modifier	Abriga e aciona funções modificantes (mover, copiar, *fillet*, etc)
Node	Representa um nó
Octree	Gera e representa um *octree*
PreProcessor1	Classe com algumas funções auxiliares de pré-processamento
Q_cell	Célula de um *quadtree*
QuadTree	Gera e representa um *quadtree*
Refactor	Reconstrutora de associações e referência entre vértices, arestas, faces
Selector	Responsável pela seleção de entidades mediante entradas do usuário
Solid	Estrutura de representação topológica de um sólido
SolidOperators	Funções auxiliares de processamento de sólidos
SolidSizing	Caixa de diálogo de segmentação de sólidos para geração de malha
Surface	Estrutura de representação de superfícies
SurfaceSizing	Caixa de diálogo de segmentação de superfícies para geração de malha
SurfaceSplitter	Contém as funções para dividir uma superfície
UndoRedo	Contém as funções para as operações de desfazer e refazer
Vertex	Representa um vértice
Vertice	Representa um vértice usando menos parâmetros que vertice
Vertice2D	Idem ao anterior, porém apenas para duas dimensões
auxAxis	Eixo de coordenadas auxiliar
chamferSurface	Contém as funções para criação de chanfro entre duas superfícies
chart	Cria e exibe um gráfico simplificado
evalButtonComm	Trata comandos do mouse
extrude	Cria um sólido por extrusão
filletSurf	Cria uma superfície de concordância entre outras duas superfícies
importMeshDialog	Faz a importação de malha de um programa externo
inputFilter	Avalia se deve aceitar ou não uma entrada do teclado
intersectCalculator	Funções para cálculo de intersecção entre duas entidades geométricas
iCell	Representação de uma célula de uma malha tridimensional
iFace	Representação da face de uma célula da malha
iNode	Representação de um nó interno para malhas volumétricas
loadAndSave	Contém funções de salvamento e carregamento de arquivos
loft	Funções para execução de varredura ordenada

revolve	Funções para execução de varredura rotacional
rotateAuxAxis	Caixa de diálogo para manipulação de eixo auxiliar de coordenadas
sliceSolid	Contém funções para efetuar a divisão de um sólido
sweep	Armazena funções para execução de varredura por caminho
textCommEval	Recebe e trata comandos ineridos na barra de comandos
textWriter	Faz registros de texto na janela de histórico
translateAxis	Caixa de diálogo para translação de um eixo auxiliar de coordenadas

APÊNDICE B LISTA DE COMANDOS DO CABEMT

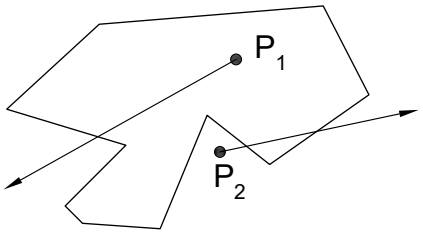
COMANDO	ALIAS	FUNÇÃO
ADD	-	Une dois sólidos
ADDELVERT	AEV	Insere um vértice (nó) no ponto indicado
ARC	AC	Criar um arco
AUXAXIS	AX	Cria um sistema auxiliar de coordenadas
CHAMFERSURF	CHS	Determina uma superfície de chanfro entre duas outras superfícies
CHECKMESH	-	Verifica a qualidade e integridade de malha de contorno
CIRCLE	CI	Criar um círculo
CLEARMESH	-	Exclui a malha do sólido selecionado
CLIP	-	Ativa ou desativa o clipping plane do OpenGL
COPY	CO	Copiar uma entidade geométrica
ELINF	-	Mostra informações de um elemento
ELLIPSE	ELL	Criar uma elipse
ERASE	E	Apaga uma entidade selecionada, se possível
EXTRUDE	EX	Criar um sólido por extrusão
EXTRUDELINES	EXTLINES	Faz a extrusão de primitivas, dando origem a superfícies
FILLET	FF	Criar arco de concordância entre primitivas
FILLETSURF	FFS	Determina uma superfície de concordância entre duas outras superfícies
INTERSECT	INT	Determina a intersecção entre dois sólidos
INTERSECTSURFACE	INTS	Divide a superfície de um sólido conforme a intersecção com uma superfície livre
LAPL2D	-	Realiza a suavização laplaciana
LIGHTON	-	Liga a iluminação
LIGTHOFF	-	Desliga a iluminação
LINE	L	Criar uma linha
LOAD	-	Carrega um arquivo com o nome padrão - saved.l2s
LOFT	-	Cria um sólido por varredura ordenada
MAKECRACK	MC	Cria uma trinca a partir de uma face pré-selecionada pelo usuário
MAKEFACE	MF	Criar uma face a partir de um conjunto de primitivas selecionadas
MERGEELVERT	MRGEV	Une elementos cujos vértices estejam muito próximos
MESHSOL	MESH	Inicia a geração de malha de elementos de contorno
MOVE	M	Mover uma entidade geométrica
MOVEELVERT	MEV	Permite que o usuário movimente manualmente um nó
NODEINF	-	Dá informações acerca de um nó
OFFSET	O	Criar cópia paralela ou concêntrica de entidade

		geométrica
REDO	R	Refaz a última operação, se possível
REVOLVE	REV	Cria um sólido por varredura rotacional
ROTATE	RO	Rotacionar uma entidade geométrica
SAVE	-	Salva o arquivo com um nome padrão - saved.l2s
SLICE	-	Divide um sólido através de um plano de corte
SPLINE	SPL	Criar uma spline
SPLITSURFACE	SPLITSURF	Realiza a interseção entre um sólido e uma superfície livre, dando origem a uma superfície livre
SUBTRACT	SUB	Subtrai dois sólidos
SWEEP	SW	Cria um sólido por varredura em caminho
TRIM	TR	Aparar uma entidade geométrica
UNDO	U	Desfaz a última operação, se possível
-	SHIFT + →	Rotaciona a visualização no sentido Y+
-	SHIFT + ←	Rotaciona a visualização no sentido Y-
-	SHIFT + ↑	Rotaciona a visualização no sentido X+
-	SHIFT + ↓	Rotaciona a visualização no sentido X-
-	CTRL + ↑	Zoom +
-	CTRL + ↓	Zoom -
-	F2	Habilita seleção de apenas linhas ou curvas
-	F3	Habilita seleção de apenas faces
-	F4	Habilita seleção de apenas superfícies
-	F5	Habilita seleção de apenas sólidos
-	F6	Habilita seleção de qualquer entidade
-	F7	Habilita seleção de sistema de coordenadas
-	F8	Habilita seleção de apenas arestas
-	F9	Habilita seleção de apenas trincas

APÊNDICE C CONDIÇÕES PARA UM PONTO ESTAR NO INTERIOR DE UM DOMÍNIO

Em muitos algoritmos da modelagem de sólidos e em alguns do MEC, é necessário determinar se um ponto está no interior de uma face ou de um sólido. Um método bem conhecido para avaliar essa condição num polígono é mostrado na figura abaixo:

Figura C.1 – Raios para determinar se um ponto está no interior de um polígono



Fonte: Elaboração do próprio autor

Dado um determinado ponto ($\boldsymbol{P_k}$), dispara-se um “raio de prova” em uma direção qualquer. Conta-se quantas vezes esse raio atravessou alguma aresta do polígono. Se essa quantidade for par, $\boldsymbol{P_k}$ está fora do polígono; se ímpar, em seu interior.

Se o raio passar por um vértice o algoritmo pode falhar, portanto, ao detectar essa condição, é indicado rotacionar ligeiramente o raio tendo como pivô o ponto de origem. Após a rotação, avaliam-se as condições como da forma descrita anteriormente.

No CABEMT essa lógica foi estendia aos domínios tridimensionais. Para um ponto $\boldsymbol{P_s}$, utilizam-se três raios, cada um orientado conforme os eixos do sistema de coordenadas global. Se o eixo tocar alguma aresta ou vértice do sólido, efetua-se uma ligeira rotação para escapar desse conflito. Em seguida, determinam-se as intersecções dos raios com as faces do sólido. Nesse processo, será necessário utilizar o algoritmo nos polígonos que constituem as faces.

Se qualquer um dos raios apresentarem um número par de intersecções, $\boldsymbol{P_s}$ estará no exterior do sólido. Se a face for triangular, pode-se utilizar as inequações (4.9) para avaliar a condição de ponto dentro ou fora, pois são mais rápidas.

APÊNDICE D RETA DE INTERSECÇÃO ENTRE DOIS PLANOS

Dado dois planos não paralelos:

$$\wp_1 \equiv a_0x + b_0y + c_0z + d_0 = 0$$
$$\wp_2 \equiv a_1x + b_1y + c_1z + d_1 = 0$$

É de vital importância determinar a equação da reta de intersecção entre os dois. Da geometria analítica, sabemos que seu vetor diretor corresponderá ao produto vetorial entre os diretores dos planos, ou seja, $(a_0, b_0, c_0)\ X\ (a_1, b_1, c_1)$. Resta determinar as coordenadas de um ponto qualquer (x_0, y_0, z_0) sob essa reta de intersecção para ter finalmente a descrição completa da mesma.

Igualando cada par de coordenadas, temos:

Se fizermos $z = 0$

$$\left[y_0 = \frac{\frac{a_1d_0}{a_0} - d_1}{b_1 - \frac{a_1b_0}{a_0}} \quad e \quad x_0 = -\frac{b_0y_0 + d_0}{a_0} \right] \quad OU \quad \left[y_0 = \frac{\frac{a_0d_1}{a_1} - d_0}{b_0 - \frac{a_0b_1}{a_1}} \quad e \quad x_0 = -\frac{b_1y_0 + d_1}{a_1} \right]$$

Se fizermos $y = 0$

$$\left[z_0 = \frac{\frac{a_1d_0}{a_0} - d_1}{c_1 - \frac{a_1c_0}{a_0}} \quad e \quad x_0 = -\frac{c_0z_0 + d_0}{a_0} \right] \quad OU \quad \left[z_0 = \frac{\frac{a_0d_1}{a_1} - d_0}{c_0 - \frac{a_0c_1}{a_1}} \quad e \quad x_0 = -\frac{c_1z_0 + d_1}{a_1} \right]$$

Se fizermos $x = 0$

$$\left[z_0 = \frac{\frac{b_1d_0}{b_0} - d_1}{c_1 - \frac{b_1c_0}{b_0}} \quad e \quad y_0 = -\frac{c_0z_0 + d_0}{b_0} \right] \quad OU \quad \left[z_0 = \frac{\frac{b_0d_1}{b_1} - d_0}{c_0 - \frac{b_0c_1}{b_1}} \quad e \quad y_0 = -\frac{c_1z_0 + d_1}{b_1} \right]$$

Qualquer uma das equações acima pode ser utilizada, basta ter o cuidado de selecionar um par que não tenha denominador nulo. Notar que os valores de d_0 e d_1 podem ser prontamente calculados após determinar (x_0, y_0, z_0).

APÊNDICE E PONTO DE INTERSECÇÃO ENTRE DUAS RETAS NO ESPAÇO

Embora as retas estejam no espaço $\mathbb{R}^3$, assume-se que sejam coplanares. Isso sempre acontecerá quando se buscam os pontos de intersecção entre uma reta $\mathcal{L}$, advinda de uma face cujo plano é $\wp$, e a reta de intersecção entre $\wp$ e outro plano qualquer. As retas podem ser descritas pela equação abaixo:

$$\mathcal{L}_1 \equiv \begin{cases} x = x_0 + at \\ y = y_0 + bt \\ z = z_0 + ct \end{cases} \quad e \quad \mathcal{L}_2 \equiv \begin{cases} x = u_0 + ds \\ y = v_0 + es \\ z = w_0 + fs \end{cases}$$

em que (x, y, z) são coordenadas quaisquer nas retas, (a, b, c) e (d, e, f) são os vetores diretores e (x_0, y_0, z_0) e (u_0, v_0, w_0) são os pontos iniciais das retas

Igualando coordenadas e fazendo as operações algébricas apropriadas resulta nos parâmetros s e t do ponto de intersecção:

$$s = \frac{av_0 + aw_0 - ay_0 - bu_0 + bx_0 - az_0 - cu_0 + cx_0}{bd + cd - ae - af} \quad OU$$

$$s = \frac{bu_0 + bw_0 - bx_0 - av_0 + ay_0 - bz_0 - cv_0 + cy_0}{ae + ce - bd - bf} \quad OU$$

$$s = \frac{cu_0 + cv_0 - cx_0 - aw_0 + az_0 - cy_0 - bw_0 + bz_0}{af + bf - cd - ce}$$

$$t = \frac{u_0 + ds - x_0}{a} \quad OU \quad t = \frac{v_0 + es - y_0}{b} \quad OU \quad t = \frac{w_0 + fs - z_0}{c}$$

Com mais um pouco de manipulação algébrica, é possível condensar as equações acima em:

$$s = \frac{-F_2(a^2 + b^2 + c^2) + F_1}{(d^2 + e^2 + f^2)\dfrac{a^2 + b^2 + c^2}{ad + eb + fc} - ad - be - cf}$$

$$t = \frac{F_1 + s(ad + be + cf)}{a^2 + b^2 + c^2}$$

na qual:

$$F_1 = cw_0 - cz_0 + bv_0 - by_0 + au_0 - ax_0$$

$$F_2 = fw_0 - fz_0 + ev_0 - ey_0 + du_0 - dx_0$$

APÊNDICE F PONTOS DE INTERSECÇÃO ENTRE ENTIDADES UNIDIMENSIONAIS COPLANARES

As equações mostradas aqui foram simplesmente obtidas pela igualdade de coordenadas das diferentes entidades. Assume-se que todas estejam no plano XY, nem que seja no sistema de coordenadas local.

- Intersecção de linha com linha: já mostrado no APÊNDICE E
- Intersecção de linha com círculo ou arco

Assume-se que a reta é descrita por:

$$y = mx + n$$

e o círculo tem centro (x_c, y_c) e raio r_c. Normalmente tem-se dois pontos de intersecção, dados por:

$$x_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \quad e \quad x_2 = \frac{-B - \sqrt{B^2 - 4AC}}{2A}$$

$$y_1 = mx_1 + n \qquad e \qquad y_2 = mx_2 + n$$

em que:

$$A = 1 + m^2$$
$$B = 2mn - 2y_c m - 2x_c$$
$$C = x_c^2 + n^2 - 2y_c n + y_c^2 - r_c^2$$

Deve-se programar com cuidado e tratar casos especiais, tais como $m \to \infty$. É necessário avaliar se os pontos de intersecção caem fora das linhas e arcos, excluindo-os.

- Intersecção entre dois círculos, dois arcos ou arco e círculo

São especificados através dos centros (x_c, y_c) e (x_c', y_c') , além dos raios r_c e r_c'. Os pontos de intersecção, se existirem, serão:

$$y_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \quad e \quad y_2 = \frac{-B - \sqrt{B^2 - 4AC}}{2A}$$

$$x_1 = \frac{K - 2(y_c' - y_c)y_1}{2(x_c' - x_c)} \qquad e \qquad x_2 = \frac{K - 2(y_c' - y_c)y_2}{2(x_c' - x_c)}$$

em que:

$$K = -x_c^2 + x_c'^2 - y_c^2 + y_c'^2 - r_c'^2 + r_c^2$$

$$R = 4(x_c'^2 - 2x_c x_c' + x_c^2)$$
$$A = 4(y_c'^2 - 2y_c y_c' + y_c^2) + R$$
$$B = -\frac{[4K(y_c' - y_c) - 4Rx_c(y_c' - y_c)]}{2(x_c' - x_c) + 2Ry_c}$$
$$C = K^2 - \frac{Rx_c K}{x_c' - x_c} + x_c^2 R + Ry_c^2 - Rr_c^2$$

- Intersecção linha e elipse

$$x_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A} \quad e \quad x_2 = \frac{-B - \sqrt{B^2 - 4AC}}{2A}$$

$$y_1 = mx_1 + n \quad e \quad y_2 = mx_2 + n$$

em que:

$$A = K + Lm + Mm^2$$
$$B = -2Kx_e - Ly_e - Lx_e m - 2My_e m + Ln + 2Mmn$$
$$C = (-Lx_e - 2My_e)n + Kx_e^2 + Lx_e y_e + My_e^2 + Mn^2 - 1$$

Os parâmetros K, L, M para a elipse são calculados pelas fórmulas a seguir:

$$K = \frac{\cos^2\theta_0}{a_e^2} + \frac{sen^2\theta_0}{b_e^2}$$
$$L = 2\, sen\theta_0 \cos\theta_0 \left(\frac{1}{a_e^2} - \frac{1}{b_e^2}\right)$$
$$M = \frac{\cos^2\theta_0}{b_e^2} + \frac{sen^2\theta_0}{a_e^2}$$

na qual θ_0, a_e, b_e, x_e e y_e podem ser vistos na figura 2.3.