LEONARDO MÜLLER ADAIME

APLICAÇÃO DO *VISUALIZATION TOOLKIT* PARA PÓS-PROCESSAMENTO DE ANÁLISES PELO MÉTODO DOS ELEMENTOS FINITOS

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciências, Curso de Pós-Graduação em Métodos Numéricos em Engenharia, Área de Concentração em Mecânica Computacional, Setores de Tecnologia e de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Dr. Sergio Scheer

CURITIBA
2005

A todos que com a imaginação, a mais poderosa forma de visualização, contribuíram para o avanço da ciência e da humanidade.

AGRADECIMENTOS

A Deus, pelas oportunidades.

Ao professor Sérgio Scheer, pela ajuda, atenção e orientação do trabalho.

Aos professores e colegas do PPGMNE.

A CAPES, pelo apoio financeiro.

A todos os amigos do CESEC, pelo companheirismo.

Aos demais amigos.

A família, pelo apoio, desde sempre.

Imaginação ou visualização e, em particular, o uso de diagramas, têm um papel crucial a representar na ciência.

René Descartes, 1637

SUMÁRIO

LISTA DE FIGURAS viii

LISTA DE SIGLAS E ABREVIATURAS x

RESUMO xi

ABSTRACT xii

1 INTRODUÇÃO 1

1.1 OBJETIVOS E ESCOPO 1

1.2 ESTRUTURA 2

2 VISUALIZAÇÃO 3

2.1 CONCEITUAÇÃO 3

2.2 TÉCNICAS DA VISUALIZAÇÃO CIENTÍFICA 5

2.3 *TOOLKITS* DE VISUALIZAÇÃO CIENTÍFICA 7

2.4 ÁREAS DE APLICAÇÃO 8

2.4.1 Visualização e o Método dos Elementos Finitos 8

2.4.1.1 Panorama dos trabalhos publicados 9

2.4.2 Outras Aplicações 11

3 BASES DA VISUALIZAÇÃO CIENTÍFICA E DO PÓS-PROCESSAMENTO DE ANÁLISES PELO MÉTODO DOS ELEMENTOS FINITOS 13

3.1 PRINCÍPIOS E COMPONENTES DA VISUALIZAÇÃO CIENTÍFICA 13

3.1.1 Modelagem Geométrica 13

3.1.1.1 Modelos e estruturas de dados 14

3.1.2 Iluminação e Cor 15

3.1.3 Textura 17

3.1.4 Renderização 17

3.1.5 Componentes de uma Cena Tridimensional 19

3.1.6 Interatividade 21

3.1.7 Animação 21

3.2 CARACTERÍSTICAS DE UM SISTEMA DE VISUALIZAÇÃO 22

3.2.1 Modelo do Processo de Visualização 23

3.2.2 Modelo de Fluxo de Dados 23
3.3 DADOS REPRESENTADOS 25
3.3.1 Classificação dos Tipos de Dados 25
3.3.2 Organização dos Dados 26
3.4 TÉCNICAS DE VISUALIZAÇÃO 29
3.4.1 Técnicas para Entidades Escalares 29
3.4.1.1 Mapeamento por cores 29
3.4.1.2 Mapeamento de imagens 30
3.4.1.3 Isolinhas e isosuperfícies 31
3.4.1.4 Mapeamento sobre superfícies 33
3.4.1.5 Renderização volumétrica 34
3.4.2 Técnicas para Entidades Vetoriais 35
3.4.2.1 Ícones pontuais 35
3.4.2.2 Traço de partículas 37
3.4.2.3 Linhas de fluxo 38
3.4.2.4 Superfícies e fitas de fluxo 39
3.4.2.5 Mapeamento de texturas 40
3.4.2.6 Desenho de deformadas 41
3.4.3 Técnicas para Entidades Tensoriais 42
3.4.3.1 Glifos ou ícones 42
3.4.3.2 *Streamtubes* ou hiper-linhas de corrente 43
3.4.4 Algoritmos de Visualização 43
3.4.4.1 *Marching cubes* e derivados 44
3.4.4.2 *Splatting* 45
3.4.4.3 *Ray tracing* e *ray casting* 46
3.5 CARACTERÍSTICAS DOS PACOTES DE VISUALIZAÇÃO CIENTÍFICA 47
4 O *VISUALIZATION TOOLKIT* - VTK 48
4.1 INSTALAÇÃO 49
4.1.1 CMake 50
4.2 ARQUITETURA 51

4.2.1 Modelo Gráfico 52
4.2.2 Modelo de Visualização 55
4.2.3 *Pipeline* de Execução 56
4.3 REPRESENTAÇÃO DE DADOS 56
4.3.1 Conjuntos de Dados 56
4.3.2 Tipos de células 57
4.3.3 Atributos de Dados 60
4.3.4 Tipos de conjuntos de dados 61
4.4 TÉCNICAS 63
4.4.1 Escalares 64
4.4.2 Vetoriais 64
4.4.3 Tensoriais 65
4.4.4 Outras Técnicas 66
4.5 MANIPULAÇÃO DE ARQUIVOS 68
4.5.1 Formato VTK 68
5 APLICAÇÕES DO VTK PARA VISUALIZAÇÃO DE RESULTADOS DO MÉTODO DOS ELEMENTOS FINITOS 70
5.1 MAPEAMENTO DE CORES EM UMA ESTRUTURA 70
5.2 GERAÇÃO DE ARQUIVOS DO TIPO VTK 74
5.3 ISOSUPERFÍCIES 75
5.4 PLANOS DE CORTE, ANIMAÇÃO E EXTRAÇÃO DE FATIAS 77
5.5 ÍCONES PONTUAIS 80
5.6 GEOMETRIA DE ENCOSTA 82
6 CONCLUSÕES E CONSIDERAÇÕES FINAIS 85
6.1 SUGESTÕES DE TRABALHOS FUTUROS 85
REFERÊNCIAS 87
APÊNDICES 92

LISTA DE FIGURAS

FIGURA 1 – CONE DO MODELO DE COR HSV 16

FIGURA 2 – PROJEÇÃO DE CENA 3D EM IMAGEM 2D 20

FIGURA 3 – ESQUEMA DE MODELO DE PROCESSO DE VISUALIZAÇÃO 24

FIGURA 4 – TIPOS DE MALHAS 26

FIGURA 5 – MAPEAMENTO DE DISTRIBUIÇÃO DE TEMPERATURA EM CORES 30

FIGURA 6 – ISOLINHAS (CURVAS DE NÍVEL) DA ACRÓPOLE EM ATENAS 32

FIGURA 7 – ISOSUPERFÍCIE DA VELOCIDADE DO VENTO (40 M/S) 33

FIGURA 8 – RENDERIZAÇÃO VOLUMÉTRICA 35

FIGURA 9 – PLOTAGEM DE ÍCONES PONTUAIS (SETAS) 36

FIGURA 10 – LINHAS DE FLUXO EM ESCOAMENTO 38

FIGURA 11 – FITAS DE FLUXO EM ESCOAMENTO 39

FIGURA 12 – COMPARAÇÃO ENTRE UMA IMAGEM LIC MAPEADA POR CORES E EM TONS DE CINZA 41

FIGURA 13 – REPRESENTAÇÃO DOS TENSORES ANISOTRÓPICOS DE FORMA EM MOLÉCULAS 43

FIGURA 14 – DIAGRAMA DA CLASSE VTKRENDERWINDOW 49

FIGURA 15 – EXEMPLO DE USO DO CMAKE 51

FIGURA 16 – ESQUEMA DA ARQUITETURA DO VTK 51

FIGURA 17 – EXEMPLO DE RENDERIZAÇÃO COM VTK 53

FIGURA 18 – *PIPELINE* DE VISUALIZAÇÃO DO VTK 55

FIGURA 19 – *PIPELINE* DE EXECUÇÃO DO VTK 56

FIGURA 20 – CÉLULAS LINEARES DO VTK 58

FIGURA 21 – CÉLULAS NÃO-LINEARES DO VTK 59

FIGURA 22 – TIPOS DE CONJUNTOS DE DADOS DO VTK 62

FIGURA 23 – MAPEAMENTO DE CORES EM UMA VIGA 70

FIGURA 24 – MAPEAMENTO DE CORES EM PEÇA 74

FIGURA 25 – ISOSUPERFÍCIES EM VIGA 76
FIGURA 26 – SUPERFÍCIES COM TENSÃO NULA EM VIGA "T" 77
FIGURA 27 – PLANO DE CORTE 78
FIGURA 28 – DEFORMADA DE VIGA "T" COM TRANSPARÊNCIA E PLANO DE CORTE 79
FIGURA 29 – EXTRAÇÃO DE FATIAS 79
FIGURA 30 – DETALHE DA BASE DE ESCADA 80
FIGURA 31 – ÍCONES PONTUAIS DE TAMANHO VARIÁVEL 81
FIGURA 32 – ÍCONES PONTUAIS DE TAMANHO CONSTANTE 81
FIGURA 33 – MAPEAMENTO DE CORES DE ENCOSTA 82
FIGURA 34 – CORTE DE ENCOSTA 83
FIGURA 35 – EXTRAÇÃO DE SEÇÃO DE ENCOSTA 84

LISTA DE SIGLAS E ABREVIATURAS

1D	Unidimensional ou uma dimensão
2D	Bidimensional ou duas dimensões
3D	Tridimensional ou três dimensões
AVS	*Advanced Visual Systems*
CG	Computação Gráfica
DINELPAR	Dinâmica de Encostas Litorâneas do Paraná e Implicações em Obras de Engenharia
DVR	*Direct Volume Rendering* (Renderização Volumétrica Direta)
EF	Elementos Finitos
GUI	*Graphics User Interface* (Interface Gráfica do Usuário)
HSV	Modelo de cor, matiz, saturação e brilho (*Hue, Saturation, Value*)
MEF	Método dos Elementos Finitos
NAG	*Numerical Algorithms Group*
OpenDX	*Open Visualization Data Explorer*
RGB	Modelo de cor, componentes vermelha, verde e azul (*Red, Green, Blue*)
SciVis	Visualização Científica (*Scientifc Visualization*)
VisAD	*Visualization for Algorithm Development*
VRML	*Virtual Reality Modeling Language*
VTK	*Visualization Toolkit*
VV	Visualização volumétrica

RESUMO

A Visualização Científica (SciVis) é um campo da Computação Gráfica que envolve a exploração de dados e informação de modo a haver ganho de compreensão e percepção dos dados. Ela é aplicada em diversas áreas, incluindo engenharia, medicina, bioengenharia, geologia, meteorologia, entre outras. Suas diversas técnicas, que possibilitam a representação de dados escalares, vetoriais e tensoriais, são capazes de produzir pós-processadores para análises numéricas. Comumente, as técnicas da SciVis são reunidas em pacotes ou *toolkits*, como por exemplo, o *Visualization Toolkit* (VTK). Ele é um sistema portável, orientado a objetos e de código aberto para computação gráfica tridimensional, visualização, e processamento de imagem, distribuído gratuitamente. É constituído por um núcleo na linguagem C++ e uma camada interpretada que permite a utilização de Tcl, Java ou Python. Este pacote foi escolhido para a implementação de algumas técnicas para visualização de resultados de análises pelo Método dos Elementos Finitos. Os programas desenvolvidos representam dados obtidos de análises utilizando-se o programa ANSYS 8.0. As listas de resultados são convertidos em arquivos do tipo VTK, que são visualizados com mapeamento por cores, desenho de ícone pontuais, animação, transparência, planos de corte, interatividade, e outras técnicas que fazem parte da Visualização Científica.

Palavras-chave: Visualização Científica (SciVis), *Visualization Toolkit* (VTK), Método dos Elementos Finitos (MEF), Pós-processamento.

ABSTRACT

Scientific Visualization (SciVis) is a Computer Graphics field that is responsible for the graphical representation of data as a means of gaining understanding and insight into them. It is applied in many areas, including engineering, medicine, bioengineering, geology, meteorology, among others. SciVis techniques, which allow representation of scalar, vector and tensor data, are capable of producing numerical analysis post-processors. These techniques are usually embedded in toolkits, such as the Visualization Toolkit (VTK). This is a portable, object-oriented, open source, freely available software system for three-dimensional computer graphics, image processing, and visualization. VTK consists of a C++ kernel and an interpreted layer that allows its use with Tcl, Java or Python. This toolkit was chosen for the implementation of some techniques to be used with Finite Element Method analysis. The applications developed represent data taken from analysis using the computer software ANSYS 8.0. The list of results are converted to VTK format files, which are visualized with color mapping, arrow plot, transparency, cutting planes, interactivity, and other techniques that are part of Scientific Visualization.

Keywords: Scientific Visualization (SciVis), Visualization Toolkit (VTK), Finite Element Method (FEM), Post-processing.

1 INTRODUÇÃO

Visualizar resultados nem sempre é tarefa fácil, especialmente quando se trata de análises numéricas. O Método dos Elementos Finitos, um método numérico largamente usado na engenharia, pode apresentar uma quantidade imensa de resultados, que podem se tornar difíceis de visualizar. A etapa que corresponde à representação dos dados determinados é conhecida como pós-processamento. Essa etapa é realizada com a utilização de diversas técnicas de visualização, reunidas na ciência denominada Visualização Científica. Muitas vezes, devido ao desconhecimento dessas técnicas, análises numéricas complexas ou em três dimensões são descartadas ou simplificadas para que os resultados encontrados sejam em menor número e mais simples de entender. Além disso, essas técnicas utilizam muito a Computação Gráfica, e por esse motivo, podem ser difíceis de serem implementadas por analistas que não tenham formação em computação, e que acabam dependendo de programas comerciais. A maioria dos aplicativos tradicionais de análise pelo Método dos Elementos Finitos possui capacidade de pós-processamento, mas nem todos são eficientes e a maioria deles é financeiramente dispendiosa. Nem sempre compensa a aquisição de um desses programas para a utilização de apenas uma parte pequena de suas funções.

1.1 OBJETIVOS E ESCOPO

A Visualização Científica e suas técnicas para representação de dados numéricos foram estudadas como apoio a análises pelo Método dos Elementos Finitos. Uma série de algoritmos de pós-processamento foi implementado para avaliar a viabilidade de uso e a qualidade dessas técnicas.

Foi avaliada a possibilidade da utilização de uma das diversas bibliotecas de Visualização Científica para as implementações. A escolhida foi o *Visualization Toolkit* (VTK), um sistema de código aberto, orientado a objetos para computação gráfica, visualização e processamento de imagens, devido a suas características de

portabilidade, facilidade de aquisição e utilização.

Este trabalho faz parte do projeto Dinâmica de Encostas Litorâneas do Paraná e Implicações em Obras de Engenharia (DINELPAR), como apoio a mapeamento, modelagens numérico-computacionais e análises.

O trabalho não pretende esgotar a utilização das diversas técnicas de visualização nem desenvolver um aplicativo completo de pós-processamento, mas viabilizar um conjunto de métodos que auxiliem em análises pelo Método dos Elementos Finitos e possibilitem o desenvolvimento de futuras aplicações.

1.2 ESTRUTURA

Além deste capítulo introdutório, o trabalho inclui outros cinco.

No capítulo 2, é apresentado um breve histórico e conceituação da visualização, suas técnicas e aplicações.

Já no capítulo 3, são relatados os princípios que envolvem a Visualização Científica, exibindo alguns conceitos da Computação Gráfica e as características de um sistema de visualização. Além disso, as técnicas comumente utilizadas são listadas, assim como os tipos de dados e suas estruturas.

O capítulo 4 trata da biblioteca de Visualização Científica *Visualization Toolkit* (VTK), escolhida para experimentação das técnicas.

No capítulo 5 são mostradas algumas aplicações para visualização de resultados de análises pelo Método dos Elementos Finitos utilizando recursos do VTK.

O capítulo 6 encerra o trabalho com conclusões e considerações finais.

Nos apêndices podem ser encontrados trechos dos códigos-fonte das aplicações, além de um exemplo de arquivo do tipo VTK.

2 VISUALIZAÇÃO

Este capítulo apresenta conceitos, definições e um breve histórico da visualização. Além disso, alguns pacotes gráficos de Visualização Científica são brevemente descritos e aplicações de visualização de dados são relatadas.

2.1 CONCEITUAÇÃO

De forma genérica, visualização pode ser definida como técnicas para criação de imagens, diagramas, ou animações para comunicar uma mensagem. Através de imagens visuais, ela tem sido uma maneira de comunicação de idéias abstratas e concretas, desde que o homem começou a desenhar. Exemplos de visualização incluem as pinturas pré-históricas em cavernas, os hieróglifos de vários povos ancestrais, como o egípcio, e os mapas utilizados pelos antigos marinheiros e generais.

A visualização como apoio à compreensão e ao estudo da ciência, ou Visualização Científica, não é algo novo. Como BRODLIE (1995) comenta, desde o século X os cientistas já representavam dados por meio de gráficos e imagens. Um exemplo comum eram os antigos planetários, nos quais se mostravam as órbita de diversos planetas e estrelas de forma bastante compreensível. Podem-se citar também os desenhos geométricos dos grandes matemáticos gregos, os métodos revolucionários de desenho técnico de Leonardo da Vinci para fins científicos e de engenharia e os diagramas de funções que ajudaram no entendimento de diversas leis da física durante os séculos passados, e que ainda são largamente utilizados. Outro exemplo bastante interessante são os modelos tridimensionais de argila do século XIX criados por James Clerk Maxwell. Usados para representar funções de duas variáveis como superfícies, eles ajudaram no melhor entendimento do trabalho de Gibbs em termodinâmica. Esses modelos estão hoje no museu do laboratório Cavendish na Universidade de Cambridge, Inglaterra (UNIVERSITY OF CAMBRIDGE, Department of Physics, 2005), e são os predecessores das plotagens de superfícies geradas por computador, muito comuns nos dias de hoje.

Embora a Visualização Científica com o uso de computadores exista praticamente desde o primeiro computador digital (ROSENBLUM, 1994), a partir da década de 1980 quantidades cada vez maiores de dados provenientes de supercomputadores e sensores ocasionou uma mudança tanto na necessidade quanto na complexidade dos dados a serem visualizados. Nesse contexto, diversas técnicas de Visualização Científica foram desenvolvidas.

O interesse pela Visualização Científica aumentou a partir de 1987. Neste ano, destaca-se o trabalho de McCORMICK et al. (1987) que faz uma das primeiras definições formais da Visualização em Computação Científica (*Visualization in Scientific Computing*), ou simplesmente Visualização Científica (*Scientific Visualization* ou SciVis), como hoje é denominada: "é o campo da Computação Gráfica que inclui interface gráfica, representação de dados e processos de algoritmos, representação visual e outras representações sensoriais como som e toque". Embora o termo visualização seja utilizado, não apenas a visão pode ser usada. Mesmo sendo esse o sentido mais comumente aproveitado, pode-se dispor de quaisquer outros estímulos sensoriais que possam representar dados. McCORMICK et al. (1987) dizem, ainda: "Visualização é um método da computação. Ele transforma o simbólico em geométrico, possibilitando pesquisadores observarem suas simulações. Visualização oferece um método para ver o não visto. Ele enriquece o processo de descoberta científica e promove o conhecimento profundo e inesperado".

Outra definição interessante é a de GERSHON (1994): "Visualização é mais que um método computacional. Visualização é o processo de transformar informação em forma visual, possibilitando usuários a observar a informação. O resultado visual capacita cientistas e engenheiros a perceber visualmente características que passam desapercebidas nos dados e, no entanto, são importantes para a exploração e análise dos dados".

Entretanto, talvez a definição mais completa seja a de BRODLIE et al. (1992): "Visualização Científica está relacionada com a exploração de dados e informação de modo a haver ganho de compreensão e percepção dos dados. O objetivo da Visualização Científica é promover um nível mais profundo de entendimento dos

dados sob investigação, confiando na habilidade poderosa dos humanos de visualizar. Em muitos casos, as ferramentas e técnicas de visualização têm sido usadas para analisar e mostrar grandes volumes de dados multidimensionais, freqüentemente variantes no tempo, de modo a permitir ao usuário extrair características e resultados rápida e facilmente".

Existem duas ciências irmãs da SciVis: a Visualização de Software e a Visualização de Informação. A SciVis abrange a visualização de dados medidos e simulados em experimentos e análises científicas e de engenharia, com a função de comparar valores, identificar, distinguir e categorizar objetos. Normalmente, os dados possuem uma geometria implícita ou intuitiva associada e representam fenômenos físicos, biológicos ou químicos. Segundo OLIVEIRA e MINGHIM (1997), a Visualização de Software "é uma área particular da visualização que estuda ferramentas visuais de auxílio às várias tarefas de desenvolvimento de programas e sistemas, incluindo apoio ao desenvolvimento, depuração, análise, e verificação de software. Devido aos requisitos particulares dessa subárea, ela tem sido tratada separadamente das demais aplicações em visualização de dados". Ainda de acordo com OLIVEIRA e MINGHIM (1997), a Visualização de Informação "engloba o desenvolvimento de técnicas de visualização de dados que não possuem geometria intrínseca ou natureza gráfica. Normalmente, refere-se à representação gráfica de dados complexos, multidimensionais e volumosos, como hierarquias muito grandes, bases de dados, dados obtidos em levantamentos estatísticos, dados financeiros, etc".

2.2 TÉCNICAS DA VISUALIZAÇÃO CIENTÍFICA

Existem diversas técnicas aplicadas pela SciVis. Algumas são específicas para tratar dados escalares, como temperatura, outras dados vetoriais, como deslocamento, e outras ainda, tensoriais, como tensor de tensões.

Entre as que operam com dados escalares, há os gráficos de funções, isolinhas e isosuperfícies, mapeamento de cores, renderização volumétrica, desenho de superfícies elevadas, entre outras. Algumas técnicas trabalham com dados

unidimensionais, enquanto outras trabalham com bi ou mesmo tridimensionais. O mapeamento de cores, por exemplo, que consiste em associar uma gama de cores a uma variação do valor do dado, pode ser usado em análises em duas ou três dimensões.

Outras técnicas trabalham com dados vetoriais, como por exemplo, as linhas e fitas de corrente, o uso de setas, e o desenho da posição das partículas. Essas técnicas trabalham geralmente em duas ou três dimensões.

Não existem muitas técnicas eficientes para representar tensores. A maioria delas usa glifos, ou ícones, para tentar representar os componentes dos tensores. Entretanto, o que tem se mostrado mais eficiente, é a representação dos componentes dos tensores de forma separada, utilizando técnicas para dados escalares. O mesmo pode ser feito com dados vetoriais.

Embora essas técnicas sejam utilizadas graças a algoritmos da Computação Gráfica, da Visão Computacional, e do Processamento de Dados, elas não são novidades, e existem desde antes da larga utilização dos computadores (COLLINS, 1993).

COLLINS (1993) comenta, por exemplo, que os gráficos de funções, sejam explícitas ou implícitas, formados a partir da interpolação dos dados de um conjunto discreto, já existem desde o século X. Eles eram usados para descrever a inclinação das órbitas dos planetas em função do tempo. As isolinhas já existem desde o século XVI, quando eram ligados os pontos de mesma profundidade em rios, e, posteriormente, para representar campos magnéticos, distribuição de temperaturas e alturas. Os mapeamentos de cores já eram utilizados desde pelo menos o século XVIII, e os desenhos de superfícies elevadas desde o século XIX. O primeiro uso de linhas de corrente data de 1665, enquanto as primeiras setas para representar dados de vetores são de 1686. Mesmo os ícones, representando dados de múltiplas variáveis, já eram usados em 1782. Até a renderização volumétrica, que consiste em visualizar todo o volume, sendo realizada computacionalmente através de transparências, cores e reflexões adequadas, já existe antes do computador. Em 1896, “Elihu Thompson gerou o primeiro par de fotografias de raios-X estéreo de um rato, apenas alguns meses após

a invenção dos raios-X" (COLLINS, 1993). Desta forma, tinha-se a ilusão de estar vendo o rato em três dimensões e em todo o seu volume, podendo-se identificar ossos e órgãos.

As diversas técnicas modernas de visualização nada mais são do que variações das tradicionais. No entanto, as técnicas usadas precisam de alguns cuidados para sua aplicação, e não apenas o uso simples dos algoritmos. GLOBUS e RAIBLE (1994) descrevem quatorze erros comuns em Visualização Científica, como por exemplo, não incluir legenda e anotações, ou suavizar o modelo sem nenhuma avaliação da qualidade, ou mesmo generalizar um método bidimensional para três dimensões.

2.3 *TOOLKITS* DE VISUALIZAÇÃO CIENTÍFICA

As diversas técnicas da SciVis foram freqüentemente desenvolvidas e reunidas em bibliotecas ou *toolkits* (pacotes) gráficos, como por exemplo, AVS, VTK, OpenDX, VisAD, IRIS Explorer.

Quanto a esses pacotes, WALTON (1993) afirma: "Eles podem ser usados para prototipar e desenvolver programas rapidamente, os quais possuem um forte grau de *feedback* e interação do usuário com os dados. Como resultado, o usuário se sente com mais controle dos dados, começa a compreender sua significância mais rapidamente e então é capaz de tomar melhores decisões. Esse alto grau de relação entre o usuário e os dados é especialmente importante quando o conjunto de dados é grande, multidimensional ou complexo".

O AVS (*Advanced Visual Systems*) é um dos pacotes para visualização de dados mais antigos, existindo desde 1989 (UPSON et al., 1989). Ele permite a utilização tanto por programadores experientes quanto por usuários diretos, pois possui um ambiente gráfico de desenvolvimento. Possui métodos de visualização para problemas em diversos campos, incluindo ciências, administração, engenharia, medicina, telecomunicações e meio ambiente (AVS, 2005).

Outro pacote, ou biblioteca é o VisAD, constituído de uma série de componentes para visualização interativa e colaborativa e análise numérica de dados.

O nome VisAD é um acrônimo para *Visualization for Algorithm Development* (Visualização para Desenvolvimento de Algoritmo) (VISAD HOME PAGE, 2005).

Um pacote bastante utilizado é o *Open Visualization Data Explorer* (OpenDX). Ele é uma versão de código aberto do produto da IBM *Visualization Data Explorer*. Existe há vários anos, e possui um conjunto de ferramentas para manipulação, transformação, processamento, renderização e animação de dados. Possui integrada uma interface gráfica orientada a objetos (IBM, 2005).

O IRIS *Explorer* é uma ferramenta desenvolvida pelo NAG (*Numerical Algorithms Group*) para desenvolvimento de aplicações de visualização. Possui um ambiente de desenvolvimento visual (NUMERICAL ALGORITHMS GROUP, 2005).

O *Visualization ToolKit* (VTK) é um sistema gratuito de código aberto para computação gráfica 3D, processamento de imagem e visualização muito usado. Ele consiste de uma biblioteca de classes na linguagem de programação C++, e algumas camadas de interface incluindo Tcl/Tk, Java e Python. Com diversas técnicas, é influenciado pelo princípio da Orientação a Objetos (KITWARE, 2005).

Existem, ainda, diversos outros pacotes de SciVis. Alguns deles são específicos para determinadas áreas, enquanto outros são bastante genéricos.

2.4 ÁREAS DE APLICAÇÃO

A Visualização Científica é hoje amplamente utilizada em diversas áreas, entre elas medicina, bioengenharia, geologia, meteorologia e engenharia, incluindo representação de dados obtidos pelo Método dos Elementos Finitos (MEF).

2.4.1 Visualização e o Método dos Elementos Finitos

O Método dos Elementos Finitos é um método numérico para solução aproximada de equações diferenciais parciais. Como descreveram GATASS, CELES FILHO E FONSECA (1991), “de uma maneira geral pode-se dizer que a idéia central do Método dos Elementos Finitos é subdividir os domínios da equação que descreve o

fenômeno físico em pequenas regiões (elementos) onde o comportamento do campo possa ser aproximado por um polinômio de grau baixo. Esse polinômio é escrito em função de valores do campo nos vértices (nós) desses elementos e estes valores (incógnitas do problema discreto) são determinados através da minimização de um funcional". Ele é amplamente usado em análises de várias áreas da engenharia. Exemplos comuns de uso são o cálculo de tensões e deslocamentos em estruturas civis e mecânicas, determinação da distribuição de temperatura em peças metálicas e percolação e adensamento em solos.

Pode-se dividir o MEF em três etapas. A primeira, o pré-processamento, consiste na definição do domínio do problema e a geração da malha de elementos. Essa etapa pode ser bastante complexa, principalmente em análises tridimensionais, e pode determinar a qualidade dos resultados obtidos.

A segunda etapa (processamento) corresponde ao método propriamente dito, e envolve a montagem de uma matriz de rigidez do modelo e a solução do sistema de equações. Obviamente, essa é a parte principal do método, e geralmente consome muito tempo de processamento e análise, e pode incluir não-linearidades, grandes deformações e outras considerações complexas.

A última etapa é chamada de pós-processamento, na qual os resultados são obtidos e podem ser visualizados. Esse passo pode ser facilmente realizado em estudos simples, mas é muito complexo em análises tridimensionais ou variantes no tempo. Essa etapa representa papel fundamental na compreensão do comportamento do modelo. As técnicas da Visualização Científica podem ser utilizadas para a execução dessa terceira etapa.

2.4.1.1 Panorama dos trabalhos publicados

Um dos poucos trabalhos que envolve o Método dos Elementos Finitos e SciVis é o de FASTOOK (1994), que desenvolveu uma modelagem pelo MEF da Antártida, incluindo a atmosfera, oceano e o próprio terreno coberto de gelo. Seu objetivo foi avaliar as influências desse continente no clima através dos anos. Nesse

caso, a visualização foi de fundamental importância para que todo o modelo fosse analisado de forma clara e precisa, principalmente em seu interior. Além desse, BENZLEY et al. (1995) mostram, as características de pré e pós-processadores para o Método dos Elementos Finitos.

Trabalho muito interessante é o de SILVA (1996). Ele utilizou o pacote OpenDX para visualizar os resultados gerados pelo aplicativo ANSYS. Com uma interface bastante fácil, o OpenDX representou resultados de maneira mais eficiente que o programa usado para o cálculo, pois além de uma melhor qualidade gráfica, utilizaram-se planos de corte e animação. De maneira semelhante, TÄGNFORS (1999) utilizou uma biblioteca de SciVis, o AVS, para visualização de resultados do Método dos Elementos Finitos.

Apesar do ANSYS e dos diversos softwares de análises de elementos finitos terem melhorado sua capacidade de pós-processamento nos últimos anos, eles não possuem todos os recursos que um pacote como o VTK e o OpenDX possuem. Diversos desses programas comerciais são ótimos para análises de elementos finitos, mas quase nenhum deles tem grandes recursos de pós-processamento. Além disso, normalmente esses softwares têm custo elevado e as aplicações do MEF desenvolvidas por analistas não devem ficar dependentes da utilização de um aplicativo como estes. Justamente tendo chegado a essa conclusão, KUSCHFELDT et al. (1998) aplicaram técnicas de renderização (*rendering*) em dados obtidos pelo MEF em estruturas mecânicas, mais especificamente em colisões automobilísticas. De acordo com esse trabalho, poucos pós-processadores comerciais aproveitavam o potencial das técnicas de SciVis e não possuíam os desenvolvimentos do estado da arte em tecnologia gráfica e técnicas de visualização. Esses programas não conseguiam representar conjuntos de dados muito grandes e variantes no tempo resultantes de simulações de impactos de forma eficaz, exigindo-se utilizar novas ferramentas que permitem manipulação interativa de geometrias complexas e mapeamento de propriedades físicas. No mesmo ano, WILLIAMS, MAX e STEIN (1998) criaram um sistema de renderização volumétrica de conjunto de dados não estruturados, especialmente de elementos finitos. A ênfase desse trabalho estava em representar os resultados da maneira mais

exata possível, sem grandes erros de interpolação e aproximação.

Já no trabalho de NIKISHKOV (2003) é utilizada a biblioteca Java 3D para visualização de resultados combinados de elementos finitos e elementos de contorno. O objetivo era criar uma aplicação leve para representar os resultados obtidos, uma vez que a utilização de bibliotecas próprias para esse fim, como VTK, AVS e OpenDX, ou mesmo de programas comercias de análises numéricas com capacidade de pós-processamento é de certa forma muito custosa computacionalmente, principalmente por possuírem muitos recursos, que acabam não sendo usados. Linha semelhante de trabalho é o de WU, WU e YANG (2002), que desenvolveram uma aplicação eficiente para visualização de resultados de métodos numéricos, mas apenas para a superfície exterior do modelo.

2.4.2 Outras Aplicações

Além dos trabalhos relacionados ao MEF, pode-se destacar, no campo da astronomia, o de GELLER e FALCO (1994) que utilizam a SciVis para a representação dos corpos celestes.

QUARESMA, OLIVEIRA e WENDLAND (2003) desenvolveram um visualizador de processos em aqüíferos, utilizando o pacote OpenDX.

Trabalhos interessantes na área médica, uma das primeiras a fazer uso da SciVis, são o de SVAKHINE, EBERT e STREDNEY (2005), que criaram um pacote de renderização de volume, com resultado muito interessante, para visualização do corpo humano, e o de BERKLEY et al. (2004), que utilizam SciVis e realidade virtual para o treinamento de procedimentos cirúrgicos com grande realismo, no caso suturas. Nesse trabalho, utilizou-se o MEF para a representação da geometria complexa da pele. Outro trabalho interessante é o de WENGER et al. (2004), que representa dados obtidos de imagem por ressonância magnética e de escoamento de fluidos em três dimensões, para visualização da distribuição sangüínea no cérebro. DUNCAN, MACKE e OLSON (1995) utilizaram o AVS para desenvolver um sistema para visualização biomolecular. Nessa área pode-se citar também SIMONETTI (2000).

Na área odontológica, há o trabalho de ALVES et al. (2000), que utilizam VTK e Java para Visualização Científica pela Internet. Modelos dentários são facilmente visualizados, utilizando-se diversos recursos da SciVis.

No campo da Meteorologia, há o trabalho de ZANDONÁ (2004), que desenvolveu um ambiente de visualização para modelos numéricos de previsão de tempo e informações ambientais, com o uso do VisAD.

Esses e diversos outros trabalhos, principalmente na área médica, demonstram a eficácia das técnicas de Visualização Científica. Essa é uma ciência em franco desenvolvimento, devido ao grande avanço em hardware (placas gráficas, processadores e memória) e software.

3 BASES DA VISUALIZAÇÃO CIENTÍFICA E DO PÓS-PROCESSAMENTO DE ANÁLISES PELO MÉTODO DOS ELEMENTOS FINITOS

A Visualização, em qualquer de suas aplicações, é fortemente baseada na utilização de algoritmos e bibliotecas da Computação Gráfica (CG), seja para representação e armazenamento de dados quanto para sua manipulação.

Assim, neste capítulo são apresentadas as características da Visualização Científica, incluindo princípios e componentes, conceitos da Computação Gráfica, tipos de dados representados e técnicas.

3.1 PRINCÍPIOS E COMPONENTES DA VISUALIZAÇÃO CIENTÍFICA

O modelo de um pós-processador de Elementos Finitos deve poder armazenar todos os dados relacionados aos elementos e seus resultados, tratá-los de forma adequada, e exibi-los de maneira compreensível.

3.1.1 Modelagem Geométrica

A geometria das formas dos objetos se constitui na base para visualização computacional. Há, para representação de objetos geométricos, diversos modelos de dados. Os sistemas de visualização precisam manipular e modificar essas informações, e a escolha de um modelo de dados é dependente dos requisitos da aplicação. Quanto mais abstratos o modelo e a estrutura de dados, mais poderosos eles poderão ser.

Segundo FOLEY et al. (1990), “modelos geométricos são coleções de componentes com geometria bem definida e, freqüentemente, interconexões entre componentes, incluindo estruturas arquitetônicas e de engenharia, moléculas e outras estruturas químicas, estruturas geográficas e veículos. Esses modelos são usualmente representados por diagramas de blocos”.

3.1.1.1 Modelos e estruturas de dados

Existem diversos modelos geométricos, desde os mais simples, que representam um mínimo de informações até os mais complexos, que detalham os modelos e permitem relação entre seus componentes.

ENCARNAÇÃO (1993) classifica os modelos de representação em gráficos, de decomposição, construtivos e de contorno.

Os modelos gráficos são constituídos de entidades simples como pontos, linhas e polígonos. Eles são úteis para visualizações simples e de baixa qualidade.

Os de decomposição consistem de uma coleção de elementos básicos e operações de junção. Embora também sejam simples, permitem algoritmos bastante velozes.

Os modelos construtivos também são constituídos por uma série de elementos básicos, mas têm operadores de combinação mais poderosos, comparado com os do modelo anterior. Com esse modelo, podem-se também desenvolver algoritmos bastante eficientes.

Já os modelos de contorno (ou superfície) representam um objeto geométrico por sua superfície de contorno. Com essa representação, uma grande gama de operações complexas pode ser introduzida. Todos os modelos, exceto o gráfico, podem ser convertidos nesse.

Dentre os modelos de contorno, existem estruturas baseadas em polígonos, outras em vértices e outras, ainda, em arestas. Exemplo muito interessante e eficiente de estrutura de dados baseada em arestas é a do tipo semi-aresta, apresentada por MÄNTYLÄ (1988). Esta estrutura é robusta o suficiente para representar os modelos geométricos envolvidos no Método dos Elementos Finitos, e possibilita um acesso eficiente a cada componente. Com esta estrutura, é também possível realizar as técnicas de visualização mais comumente utilizadas.

Qualquer combinação dos quatro modelos resulta em um modelo híbrido.

Uma implementação interessante de estrutura de dados é o de SOUZA (2003), que adicionou classes ao VTK para tornar sua estrutura de dados mais robusta. Outros

exemplos importantes são os desenvolvidos no Grupo de Tecnologia em Computação Gráfica (TECGRAF, 2005) da Pontifícia Universidade Católica do Rio de Janeiro. Exemplos são os trabalhos de LIRA (2002), CAVALCANTE NETO (1998), CAMPOS (1991), entre outros.

3.1.2 Iluminação e Cor

A maior parte do processo físico de iluminação pode ser facilmente simulado no computador. Entretanto, sua utilização na obtenção de imagens de alta qualidade implica em altos custos computacionais. Assim, é mais comum, especialmente em aplicações de visualização como em EF, a adoção de um modelo simplificado, que modela apenas os processos rudimentares da reflexão da luz sobre uma superfície. A iluminação, de forma direta, não é muito importante em análises numéricas. Mas, de forma indireta, pelo efeito de sombreamento ou tonalização, ela pode dar a impressão de tridimensionalidade, facilitando a análise de modelos 3D.

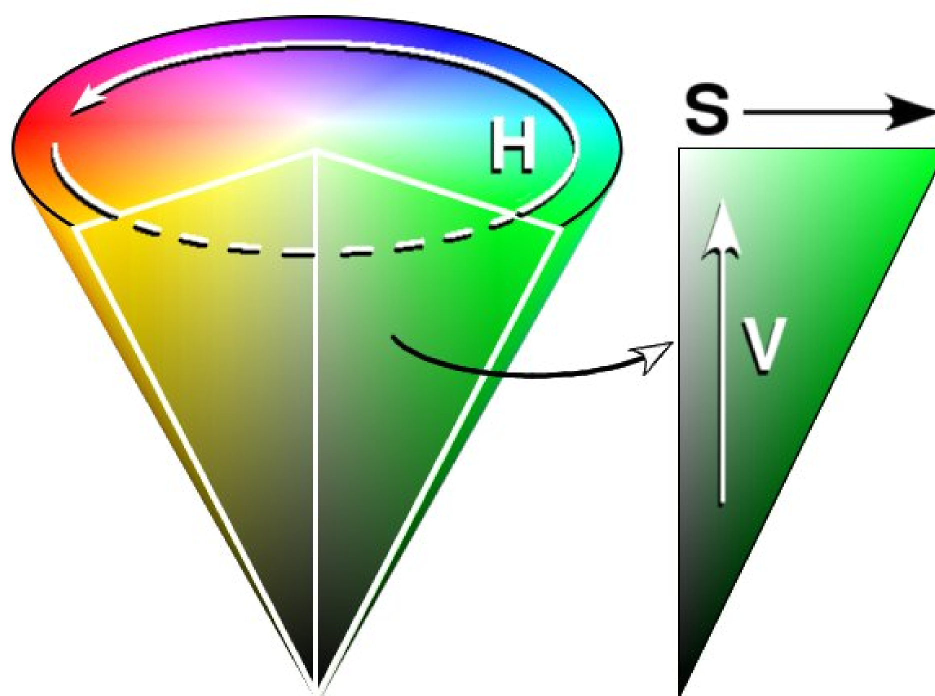
A cor é de fundamental importância no MEF. A aplicação mais comum consiste em associar cores a valores de dados. Por exemplo, associar uma gama de cores, variando do vermelho ao azul, com os valores de tensão em uma direção, da máxima tração à máxima compressão. Além dessa aplicação, a utilização de cores é item obrigatório em aplicações de CG.

O processo de calcular a iluminação para cada ponto de uma superfície é denominado tonalização (*shading*) (OLIVEIRA e MINGHIM, 1997). Para superfícies coloridas, a intensidade de cada componente da luz (vermelha, verde e azul, ou RGB), deve ser calculada individualmente. Isso porque, em CG, a maioria dos sistemas gráficos trabalha com o modelo de cores RGB, que é o mais próximo do hardware gráfico, que trabalha com fósforo destas três cores. Esse modelo assume que a cor de um pixel é determinada pela soma dos componentes vermelho, verde e azul da cor, sendo que a quantidade de cada componente é expressa no intervalo [0,1] ou [0,255].

Existem modelos de cores mais intuitivos, como o HSV (*Hue, Saturation, Value*, ou matiz, saturação e brilho), que também estão disponíveis para a geração de

tabelas de cores em muitos sistemas. No modelo HSV, a definição de cada cor básica se dá sobre um círculo de cores, na qual a borda que corresponde ao grau 0 (3 horas) é vermelha, ao grau 60 é amarelo, ao grau 120 é verde, e assim por diante, conforme representado pelo cone da Figura 1. O parâmetro de saturação define a proporção de quanto daquele matiz é utilizado, com valores mais saturados nas bordas do círculo e menos saturados próximos do centro (valores entre 0 e 1), e o parâmetro brilho define a luminosidade da cor (valores entre 0 e 1).

FIGURA 1 – CONE DO MODELO DE COR HSV



FONTE: HSV COLOR SPACE (2005)

A definição de cores e de tabelas de cores apropriadas para uma cena é um assunto de extrema importância em Visualização. Tais tabelas normalmente são responsáveis pelo mapeamento de informação numérica e, portanto, assumem significado específico, como por exemplo, valor de temperatura. A correta definição de cores e outros atributos visuais tem sido assunto de pesquisas em visualização.

3.1.3 Textura

Objetos reais não são suficientemente caracterizados apenas por sua geometria tridimensional, mas necessitam de informação adicional de sua superfície. No caso mais simples, a superfície do objeto é coberta com uma cor constante. Como regra, as superfícies de objetos não são desestruturadas, mas possuem variação de cor dependente de posição, transparência, rugosidade, e também geometria, que dão a característica de aparência de materiais. Essas propriedades são sumarizadas no termo textura.

No significado original, textura visual descreve uma impressão sensorial. É considerada uma percepção visual quando dois vizinhos, possivelmente estruturados, pertencentes ao campo visual possam ser separados sem esforço. Portanto, regiões com cor homogênea representam um caso especial de texturas visuais (ENCARNAÇÃO, 1993).

O objetivo de simulação de texturas em Computação Gráfica é gerar imagens de textura num dispositivo visual final, que evoca uma sensação de textura similar à causada pela original.

Em CG, além de representar objetos reais, as texturas são usadas para dar suporte a aplicações científicas, para percepção do espaço e forma de objetos tridimensionais, e como primitivas básicas de conjunto de dados de visualização com parâmetros múltiplos. As texturas são tratadas como parâmetros de aparência, como por exemplo, cor, rugosidade, reflexão e brilho.

No entanto, as texturas não são comumente usadas em análise pelo MEF, mas algumas técnicas de pós-processamento utilizam as texturas para representar dados vetoriais de maneira qualitativa.

3.1.4 Renderização

Uma das funcionalidades fundamentais da SciVis, e conseqüentemente, da Computação Gráfica é a renderização (*rendering*). Segundo OLIVEIRA e MINGHIM

(1997), essa é o processo de geração de uma imagem em uma tela a partir de uma descrição computacional. O processo varia bastante, e muitas técnicas estão disponíveis para sua realização, envolvendo desde a geração de desenhos em duas dimensões (2D) até técnicas sofisticadas de geração de imagens tridimensionais (3D).

Uma técnica de renderização é composta de um procedimento de eliminação de superfície escondida e um modelo de *shading* (cor, luz e sombra). A eliminação de superfícies escondidas faz com que apenas as partes do modelo que são visíveis pelo observador sejam mostradas. O modelo de *shading* determina a aparência final de cada superfície na tela, de acordo com um modelo de iluminação, que leva em conta as condições de luz, propriedades ópticas das superfícies (cor, textura, reflexão, etc.) e a posição e orientação das superfícies em relação às fontes de luz, a outras superfícies e ao observador.

A maioria das técnicas 3D, nas quais são baseados diversos algoritmos da SciVis, gera a imagem de um objeto por meio da simulação de sua iluminação ambiente. Em outras palavras, algoritmos de renderização consideram que os objetos em uma cena são atingidos por raios provenientes de fontes de luz. As técnicas de renderização baseadas nesse conceito variam consideravelmente: as mais simples desconsideram as reflexões entre objetos e a atenuação da luz no ambiente, enquanto que as técnicas mais complexas, como o *Ray tracing*, e que produzem imagens bastante realísticas, adotam um modelo para a simulação a interação dos raios de luz com todos os objetos da cena, a um alto custo computacional.

Em SciVis, boa parte das estratégias de renderização é baseada em modelos simples de iluminação, pois se espera das imagens que sejam mais significativas do que realísticas, o que pode ser obtido com técnicas relativamente simples. Os processos de renderização podem ser subdivididos em duas categorias: métodos baseados no espaço da imagem (ou ordem da imagem) e métodos baseados no espaço do objeto (ou ordem do objeto). Nos primeiros, a cena a ser visualizada é renderizada a partir dos pontos de tela (pixels), verificando-se quais elementos no espaço contribuem para a cor de cada pixel da área visível da tela. A segunda classe de métodos trabalha renderizando individualmente cada objeto da cena. Por exemplo, numa cena que

possui um cubo, um cone e o chão, um método baseado na ordem da imagem verificaria quais partes dos três influenciam na cor de um determinado pixel da tela, enquanto que um método baseado no espaço renderizaria primeiro o cubo, depois o cone, depois o chão.

Métodos de renderização podem ser também classificados como de frente-para-trás ou de trás-para-frente, dependendo da forma como o espaço é percorrido, ou seja, primeiro os objetos mais próximos ou primeiro os objetos mais distantes do observador são considerados, respectivamente. Além disso, existem técnicas de renderização por superfícies e volumétricas. No primeiro tipo, uma representação geométrica dos dados (uma superfície) é criada antes do processo de iluminação. Tal objeto é, então, apresentado na tela utilizando técnicas convencionais da CG. Na renderização volumétrica, a intensidade dos pontos de tela é determinada diretamente a partir dos elementos do volume de dados, sem a criação de uma representação intermediária. Elementos de volume unitários são chamados de *voxels*, que são correspondentes tridimensionais dos pixels. O volume representa o trecho do espaço 3D onde se localiza o objeto, dado ou cena a ser renderizado.

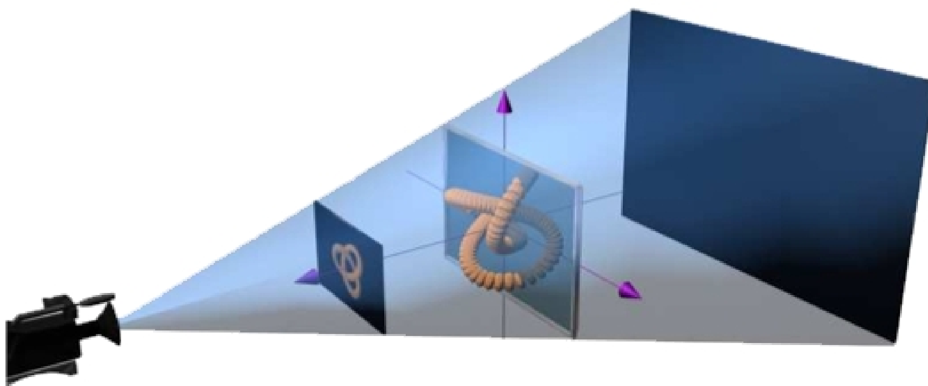
Em análises pelo MEF, a renderização de volume gera alguma confusão, pois como todo o volume é visualizado ao mesmo tempo, com o uso de transparências, o resultado final em 2D é de uma mistura de informações e cores, em que fica difícil de avaliar cada dado separadamente. Entretanto, pode ser útil para uma avaliação prévia e geral de todo o modelo. As técnicas usuais dos pós-processadores de EF utilizam renderização de superfície, por ser mais fácil perceber as características dos dados. Para visualizar o volume como um todo, é necessário utilizar planos de corte e/ou animação para visualização do modelo por partes sucessivas.

3.1.5 Componentes de uma Cena Tridimensional

Um dos objetos mais importantes na construção de uma cena tridimensional é a câmera sintética, que possibilita a visão de qualquer outro objeto. Portanto, pelo menos uma câmera precisa ser definida em cada cena.

Uma câmera sintética pode ser caracterizada de diversas maneiras, como por exemplo, como um ponto de visão e centro de interesse que define o centro da imagem da câmera. Outra idéia é associar o ponto de visão com uma direção. A Figura 2 mostra uma representação da projeção de uma imagem em uma tela 2D, de acordo com a câmera.

FIGURA 2 – PROJEÇÃO DE CENA 3D EM IMAGEM 2D



FONTE: SETRED (2005)

O movimento de câmera é o mais utilizado para representar o observador em sistemas gráficos e de visualização (OLIVEIRA e MINGHIM, 1997). O posicionamento e a animação de objetos se dão tanto pela alteração da posição do objeto em relação a um sistema de coordenadas global da cena, como pela alteração pura e simples da posição da câmera.

As projeções mais comuns, para a criação de uma imagem, são a perspectiva e a paralela. Em análises de EF, a projeção paralela ortogonal é preferida à perspectiva, pois, enquanto esta distorce as dimensões, dando uma impressão tridimensional, aquela mantém proporções e dimensões. A manutenção dessas características é importante para uma compreensão global do comportamento do modelo, mesmo diminuindo a sensação de tridimensionalidade.

Outra classe de objetos importante á a que engloba as fontes de luz. As cores

na imagem da câmera são determinadas por interação entre luz e objetos, de acordo com a distribuição de luminosidade e as propriedades do objeto.

3.1.6 Interatividade

Interação pode ser entendida como o conjunto de ações do usuário e a reação do computador durante o processo de visualização. Ela é importante para uma melhor exploração e compreensão dos dados, seja através de mouse, teclado ou qualquer outro dispositivo de entrada. A interatividade deve permitir a navegação do usuário pelo modelo numérico, possibilitando ver pontos de maior relevância, e escolher ver apenas os dados escolhidos.

Uma resposta gráfica rápida faz com que o usuário sinta como se um objeto real seguisse suas diretivas. Normalmente, a interatividade inclui operações de rotação, translação, zoom, e a mudança da técnica utilizada, de pontos de visão e de parâmetros dos dados.

3.1.7 Animação

Segundo ENCARNAÇÃO (1993), animar significa, literalmente, trazer à vida. Na prática, significa definir mudanças em parâmetros no tempo, ou aplicar operações em objetos no tempo. A animação computacional é interdisciplinar, usando métodos de diferentes campos.

Numa animação comum, os intervalos de tempo entre um passo e outro devem ser tais que possam assegurar a sensação de continuidade, como num filme, produzindo a ilusão de movimento contínuo. Para criar esta ilusão, comumente é necessário o uso de interpolações entre uma etapa e outra, quando os dados estão esparsos no tempo. Entretanto, este efeito de continuidade não é fundamental em EF, pois ao analista é importante a representação sucessiva dos dados simulados ou mensurados, mesmo que sem um efeito de animação contínua.

Aplicar o parâmetro tempo a uma visualização pode ajudar em muitas

situações para aumentar a qualidade da representação visual e mostrar mais informações.

O tempo pode ser usado para mapear uma dimensão de um conjunto de dados, em geral tempo para processos dependentes de tempo, mas também em outros casos, como por exemplo, análises que envolvem incrementos de carga. Neste caso, cada incremento é representado em tempos diferentes, podendo-se ver o comportamento do modelo como se a carga estivesse realmente sendo aplicada em tempo real.

O tempo, em correspondência com navegação, permite uma exploração mais intensiva de correlações espaciais possível. Especialmente em uma cena 3D, profundidade e a forma dos objetos pode ser percebida mais facilmente se o observador puder se mover, mesmo se a liberdade de movimentos for limitada. A animação pode, ainda, ser usada para criar efeito 3D, com o uso de movimento da câmera.

3.2 CARACTERÍSTICAS DE UM SISTEMA DE VISUALIZAÇÃO

OLIVEIRA e MINGHIM (1997) subdividem uma técnica de visualização científica em vários passos: a construção de um modelo empírico a partir dos dados, a seleção de algum mecanismo de mapeamento do modelo em um objeto de visualização abstrato, como uma imagem ou mapa de contornos, e a renderização da imagem no dispositivo de exibição gráfico. A visualização envolve, portanto, a exploração, transformação e mapeamento de dados em objetos gráfico.

Os dados em SciVis podem estar definidos sobre domínios unidimensionais (1D), bidimensionais (2D), tridimensionais (3D) e, genericamente, multidimensionais. Em geral, eles estão organizados em uma malha (regular ou irregular), mas também podem se constituir de um conjunto de pontos esparsos. As informações associadas a cada ponto podem ser constituídas de um ou mais valores escalares, vetoriais ou tensoriais, e os dados podem ser estáticos ou variantes no tempo. Uma grande variedade de técnicas de visualização está disponível atualmente. Existem várias classificações e denominações variadas para elas.

3.2.1 Modelo do Processo de Visualização

Os sistemas de visualização devem ser usados como parte integrante do processo de investigação científica.

Dados podem ser obtidos de duas maneiras: medidas experimentais, através da amostragem de processos empíricos, ou simulações computacionais. A conseqüência é que os dados são conhecidos apenas em pontos discretos. Isso cria um sério problema, pois uma das atividades relevantes em visualização é justamente determinar valores dos pontos arbitrários. A solução óbvia para esse problema é o uso de funções de interpolação, através das quais se infere uma relação entre valores de dados vizinhos. A função de interpolação mais usual é a linear, mas existem funções quadráticas, cúbicas, do vizinho mais próximo, *spline*, entre outras. Estas funções são usadas para gerar dados nos intervalos entre pontos desconhecidos, e isso pode introduzir erros.

Os problemas relacionados à amostragem e à interpolação, que devem ser considerados no contexto do processo científico, refletem-se como limitações a serem tratadas também nas técnicas e sistemas de visualização. Os pós-processadores trabalham com modelos computacionais dos fenômenos em estudo, e não com modelos matemáticos ideais.

Muitas operações em sistemas de visualização são dependentes do núcleo gráfico utilizado e de sua interface com ele. Os sistemas gráficos fornecem mecanismos de apresentação das imagens geradas pelas visualizações. A maioria dos sistemas gráficos trabalha com bibliotecas gráficas básicas já estabelecidas, como a OpenGL (OPENGL, 2005). Também estão incluídos nos sistemas gráficos os componentes de interface final com o usuário, a base de dados gráfica, o gerente de visualização e a interface de programação.

3.2.2 Modelo de Fluxo de Dados

UPSON et al. (1989) definiram um modelo do processo de visualização, que teve profunda influência no desenvolvimento da visualização. Trata-se de um modelo

orientado a dados, no qual os dados são transformados em um número de passos lógicos até ser representado como imagem. O modelo assume que existe uma fonte de dados, que pode ser uma simulação ou um conjunto de observações ou medições. A Figura 3 mostra o esquema geral deste modelo.

FIGURA 3 – ESQUEMA DE MODELO DE PROCESSO DE VISUALIZAÇÃO

FONTE: ADAPTADO DE BRODLIE (1993)

O primeiro passo é o de filtro, no qual os dados de interesse são derivados da fonte original crua. Este passo pode envolver uma variedade de operações, como por exemplo, interpolação a partir dos dados esparsos em uma malha regular, extração de uma seção específica dos dados, ou ainda, realização de alguma operação de suavização.

A segunda etapa, que corresponde à construção da representação dos dados, é denominada de mapeamento. Por exemplo, se um mapeamento de contorno é desenhado a partir de dados de altura 2D, isso envolverá a construção das isolinhas como um conjunto de polilinhas, ou curvas.

O último passo é a renderização, ou seja, a representação geométrica é convertida em imagem, a qual pode ser desenhada.

Este modelo tem sido a base de diversos sistemas de visualização. Em geral, eles fornecem uma série de filtros e mapeadores (*mappers*), que podem ser combinados, gerando o fluxo de dados até a renderização da imagem.

BRODLIE (1993) comenta que, embora este modelo tenha tido sucesso, ele pode dar a falsa visão do mundo porque é centralizado nos dados. Raramente se quer visualizar os dados, mas o campo, ou modelo, do qual os dados foram amostrados. É trabalho da visualização reconstruir este campo a partir dos dados, e então usar diferentes técnicas para mostrá-lo. Os dados só são úteis se for possível conhecer o

comportamento do modelo do qual eles foram extraídos. Uma solução é adicionar aos filtros uma construção do modelo (modelagem). Portanto, primeiro se construiria o modelo, e então, os dados seriam extraídos para posterior representação gráfica. Entretanto, nas análises pelo MEF, os dados estão associados a uma geometria, que existe indiretamente, pela associação de todos os elementos. Portanto, pode-se visualizar os dados juntamente com o modelo completo, de forma fácil.

3.3 DADOS REPRESENTADOS

Existem diversas fontes de dados científicos, seja por análises numéricas computacionais, ou por medição. De uma mesma fonte podem surgir diversos tipos de conjuntos de dados, e cada conjunto de dados pode conter diversas variáveis dependentes e independentes (TREINISH, 1993).

Como geralmente trabalha-se com grande quantidade de dados, é importante a organização e gerenciamento adequados dos dados, além da facilidade de acesso. Existem análises pelo MEF com centenas de milhares de elementos e nós, e esse grande volume de dados precisa ser representado adequadamente.

3.3.1 Classificação dos Tipos de Dados

Segundo BRODLIE et al. (1992), um conjunto de dados pode ser considerado como uma entidade que contém um conjunto de valores em certo intervalo de variação e está definido sobre algum domínio de variáveis independentes. Entretanto, nem sempre o conjunto de dados implica na existência de um mapeamento: pode-se visualizar um conjunto de pontos esparsos sem qualquer função associada.

Pode-se dividir os dados em escalares, vetores e tensores. Esta divisão é a base para a principal classificação das técnicas de visualização. A dimensão do domínio pode ser usada para uma subclassificação, e permite estabelecer distinções adicionais de acordo com a natureza do domínio.

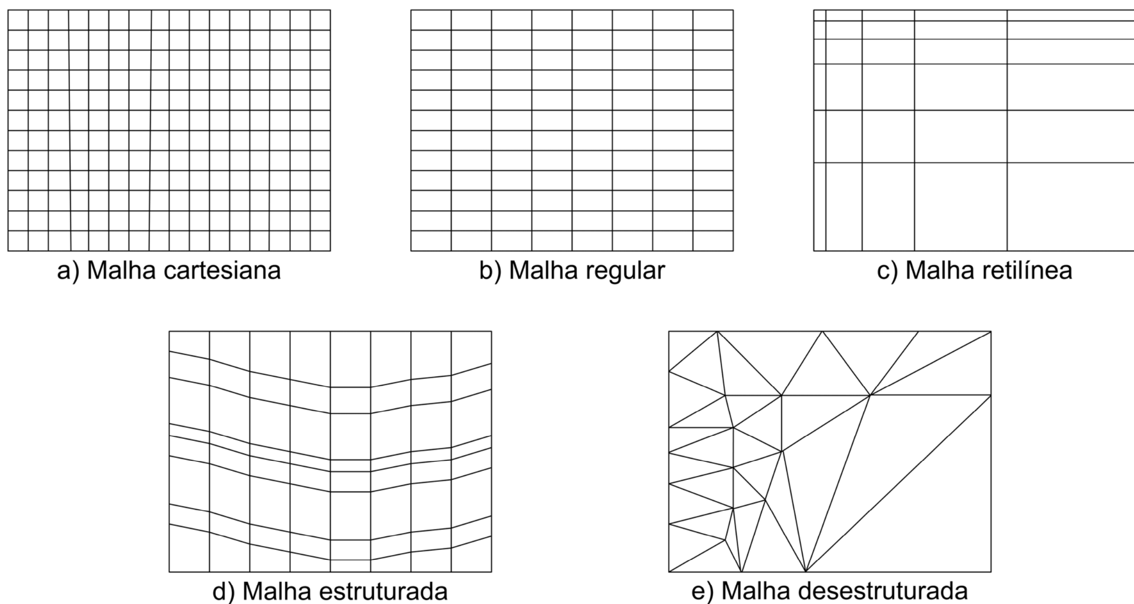
Existem técnicas de visualização específicas para entidades do tipo conjunto

de escalares, vetoriais e tensoriais.

3.3.2 Organização dos Dados

Em geral, um conjunto de dados possui certa organização. Normalmente, consiste de uma malha, ou grade de células, sendo que os dados estão posicionados nos vértices de cada célula, e valores no interior das células podem ser obtidos por interpolação. Dessa forma, a estrutura de um conjunto de dados está implicitamente definida por pontos, que associam informações geométricas aos itens de dados, e por células, que associam propriedades topológicas. Os dados propriamente ditos são atributos associados aos elementos (pontos e células) dessa estrutura. De acordo com a organização definida pela estrutura, pode-se ter vários tipos de malhas, exemplificadas na Figura 4.

FIGURA 4 – TIPOS DE MALHAS



FONTE: ADAPTADO DE OLIVEIRA E MINGHIM (1997)

Em uma malha cartesiana todos os elementos são quadrados, no caso 2D ou cubos idênticos, no caso 3D, alinhados aos eixos principais. Uma malha regular tem todos os seus elementos idênticos e alinhados aos eixos, mas esses elementos são

retângulos (ou paralelepípedos) regulares. Elementos de uma malha retilínea são quadriláteros ou hexaedros alinhados aos eixos, mas não necessariamente idênticos, como ilustrado.

Elementos em uma malha estruturada são quadriláteros ou hexaedros não alinhados aos eixos principais, como os que aparecem em grades esféricas ou curvilíneas. Uma malha estruturada em blocos é um conjunto de malhas estruturadas agrupadas. Uma malha desestruturada (ou não-estruturada) contém polígonos ou poliedros sem qualquer padrão explícito de conectividade. No caso tridimensional, as células podem ser tetraedros, hexaedros, pirâmides, ou outro tipo. Uma malha híbrida é uma combinação de quaisquer tipos anteriores.

Existem, ainda, as malhas sem conectividade, os pontos esparsos. Podem ter posição regular ou irregular, dependendo dos espaçamentos entre pontos (TREINISH, 1993).

Em resumo, a principal distinção é que conjuntos de dados podem ser regulares (estruturados), ou irregulares (desestruturados). Conjuntos regulares apresentam um relacionamento inerente entre os pontos aos quais estão associados os dados. Com isso, não é necessário armazenar explicitamente as posições de todos os pontos. No caso de uma malha cartesiana, por exemplo, basta manter a posição inicial, o espaçamento entre os pontos e o número total de pontos, o que permite uma representação computacional compacta. Esse não é o caso dos conjuntos irregulares que, por outro lado, têm a vantagem de serem mais flexíveis e de permitir representações adaptativas (com mais células nas regiões nas quais a informação é mais densa).

As malhas irregulares são de fundamental importância no MEF, pois, normalmente, as malhas de elementos finitos são desse tipo. Em geral, uma malha descreve a base geométrica para o mapeamento das funções ou variáveis dependentes a algum sistema de coordenadas.

3.3.3 Gerenciamento de Dados

O gerenciamento de dados é uma tarefa importante na Visualização Científica. Como os dados são coletados ou computados em larga escala de forma crua, gerando então geram dados derivados, imagens e informação, seu gerenciamento se torna crucial.

Componentes de gerenciamento de dados em sistemas de visualização são responsáveis pela importação e exportação de arquivos e dados em diferentes formatos, bem como pela comunicação com fontes externas de informação e acesso a bases de dados. Também se inclui nessa classe de procedimento o controle da memória interna, cuja demanda é muito alta.

Um dos fatores críticos em um sistema de visualização é a organização da base de dados, que pode ser seqüencial, hierárquica, relacional ou orientada a objetos. O modelo de dados se refere à organização interna dos dados utilizada pelas técnicas de visualização. O projeto de um modelo correto é fundamental para a eficácia de uma técnica de visualização. Aspectos importantes são eficiência de armazenamento e execução, e suporte à análise. Além disso, é importante a redução de dados, que abrange algoritmos de compressão de dados e de eliminação de elementos redundantes, como vértices, arestas e triângulos duplicados em uma representação poligonal de superfície. Isto reduz o custo computacional, fator crítico para análises em tempo real.

Algumas estruturas de dados resolvem problemas de relacionamento e acesso, pois a simples listagem dos dados não é suficiente. Uma estrutura mais complexa, como por exemplo, a semi-aresta, é necessária.

A compressão de dados é algo também importante, principalmente para o armazenamento em arquivos e transmissão de grandes volumes de dados.

3.4 TÉCNICAS DE VISUALIZAÇÃO

Normalmente, as diversas técnicas de Visualização Científica são aplicadas especificamente a um tipo de dado. Algumas exibem escalares, outras vetores, e outras, ainda, tensores.

3.4.1 Técnicas para Entidades Escalares

Além do gráfico matemático convencional e do desenho de pontos no espaço, existem algumas técnicas para representação de dados escalares.

3.4.1.1 Mapeamento por cores

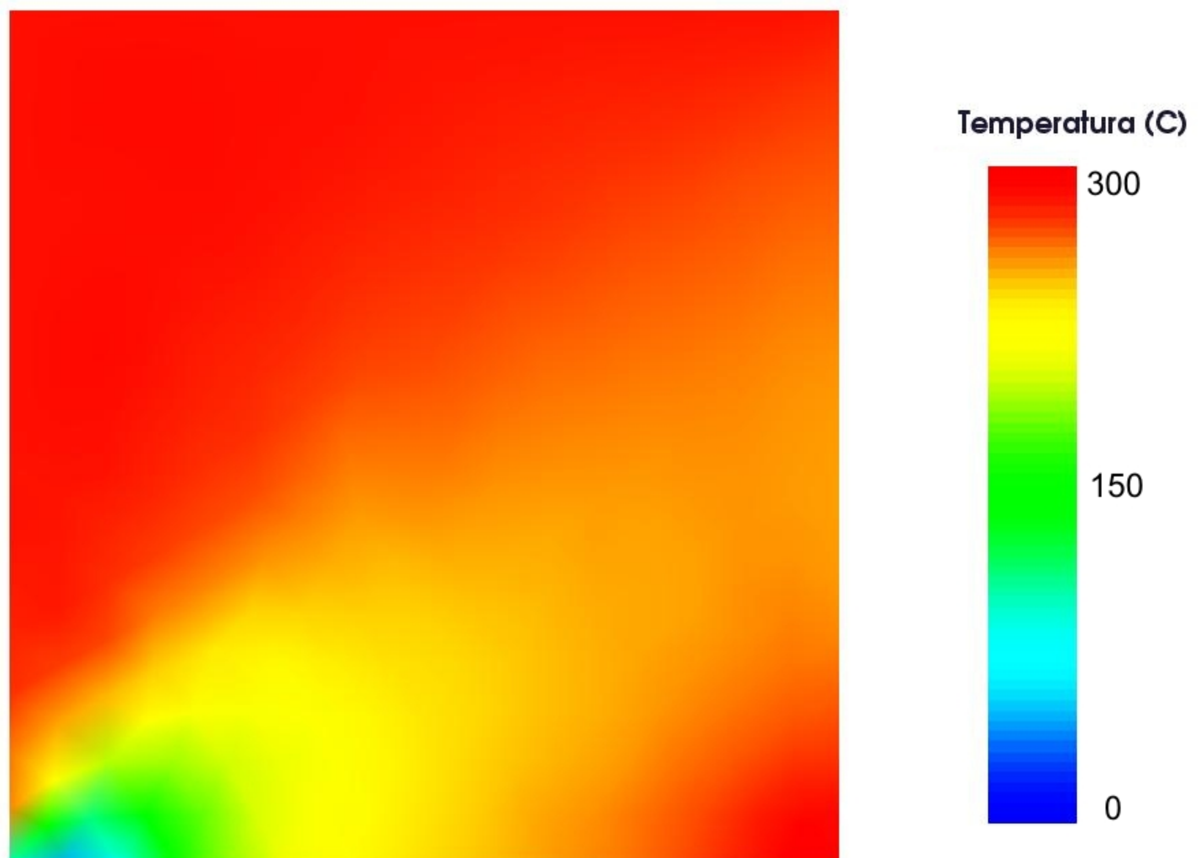
O mapeamento por cores é uma técnica de visualização bastante comum que consiste em associar dados escalares a cores, exibindo-as como indicação dos valores. Este mapeamento é implementado através da indexação de uma tabela de cores (*color lookup table*). Os valores escalares são usados como índices para esta tabela. A tabela é um vetor de cores (descritas normalmente em termos de seus componentes RGB), e está associada a um intervalo de variação definido por um valor mínimo e um máximo. Os valores escalares são mapeados nesse intervalo: para cada escalar, obtém-se um índice para uma posição na tabela. A questão principal, para se gerar boas visualizações, é escolher cuidadosamente as entradas da tabela, uma tarefa nem sempre simples. A técnica é aplicável a dados escalares definidos em domínios 1D, 2D e 3D.

Existe uma forma mais genérica de tabela de cores, denominada função de transferência. Uma função de transferência é qualquer expressão que mapeia um valor escalar em valores RGB especificando uma cor e, eventualmente, um valor de opacidade. A tabela de cores é uma amostragem discreta dessa função de transferência.

BERTON (1990) afirma que a tabela de cores é normalmente escolhida para relacionar alguma percepção humana da propriedade, por exemplo, a seqüência

branco, vermelho, laranja, amarelo... pode representar temperatura decrescente. O método tradicional é o mapeamento com luminescência, o qual relaciona cores brilhantes (claras) com valores altos. Um exemplo bastante comum é o mapeamento de componente de tensão em cores. Associa-se, por exemplo, vermelho com tração, e azul com compressão, enquanto o verde é associado a valores neutros. A Figura 5 mostra um mapeamento por cores da distribuição de temperatura em uma chapa metálica. No lado direito da figura há uma barra de cores, que mostra a relação das cores com valores, no caso de temperatura (^{o}C).

FIGURA 5 – MAPEAMENTO DE DISTRIBUIÇÃO DE TEMPERATURA EM CORES



FONTE: O AUTOR

3.4.1.2 Mapeamento de imagens

OLIVEIRA e MINGHIM (1997) definem outro tipo de mapeamento, o de imagens. A partir de um conjunto de valores definidos sobre pontos em um plano 2D,

pode-se exibir uma imagem da função subjacente. O domínio é dividido em uma grade de células (tipicamente correspondendo aos pixels a serem exibidos), e a cor de cada célula é escolhida para representar o valor da função no ponto correspondente. Esta é a técnica adotada para visualizar dados obtidos por satélites ou scanners. Nesse caso, a malha de dados é densa e gera imagens que podem ser processadas através de técnicas de processamento de imagens. É possível generalizar a técnica para exibir múltiplos campos escalares definidos sobre uma região.

3.4.1.3 Isolinhas e isosuperfícies

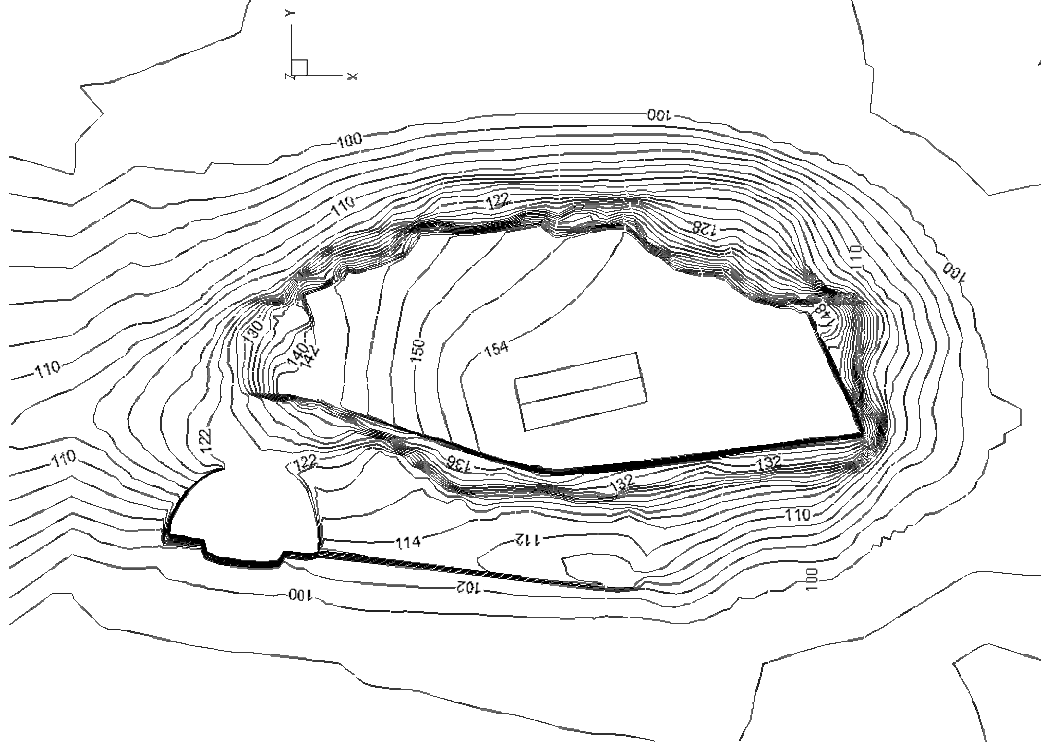
A partir de um conjunto de valores definidos sobre pontos em um plano 2D, podem-se traçar isolinhas de valor constantes. Os parâmetros para essa técnica são os valores das isolinhas, que correspondem aos valores de interesse. Exemplos de visualizações desse tipo são os mapas de previsão do tempo, anotados com linhas de temperatura constante (isotermas), e os mapas topológicos anotados com linhas de elevação constante (curvas de nível). A geração de isolinhas requer o uso de alguma técnica de interpolação, a qual depende da organização dos dados. A Figura 6 é um exemplo de isolinhas de elevação constante do terreno em torno do Parthenon na Grécia, em metros acima do nível do mar.

As isosuperfícies são o equivalente tridimensional das isolinhas. Nesse caso, a abordagem consiste em aplicar um detector de superfícies ao volume de pontos amostrados e, em seguida, ajustar primitivas geométricas sobre as superfícies detectadas, as quais são posteriormente renderizadas por técnicas tradicionais.

Uma vantagem das isosuperfícies é a possibilidade de identificação de estruturas nas medições ou simulações. Elas também permitem análise em tempo real, pois, apesar do processo de extração das superfícies ser lento, a renderização das superfícies resultantes pode ser muito rápida, principalmente se feita por hardware. Entre as desvantagens, tem-se a perda da informação relativa ao conteúdo do conjunto de dados fora dos valores selecionados para o traçado. Além disso, nem sempre os dados podem ser descritos em termos de superfícies, como é o caso de objetos amorfos

(nuvens, fumaça e etc.). Exemplo de isosuperfície é o da Figura 7, que gerada com o Vis5D, o *toolkit* que deu origem ao VisAD, mostra o volume cuja velocidade do vento é de 40 m/s.

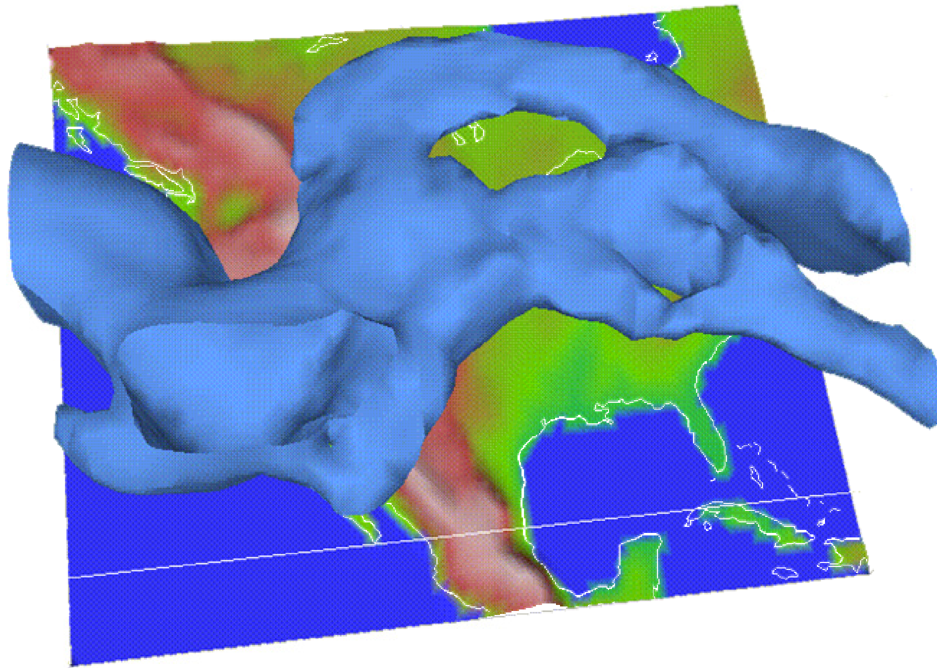
FIGURA 6 – ISOLINHAS (CURVAS DE NÍVEL) DA ACRÓPOLE EM ATENAS



FONTE: CFD NORWAY (2005)

Os métodos para o ajuste de primitivas geométricas incluem métodos de conexão de contornos, muito usados em aplicações médicas, e métodos de intersecção de *voxels*. Os primeiros envolvem a identificação de contornos para um valor de dados. Depois, as curvas em fatias adjacentes são conectadas, em uma abordagem denominada tecelagem (*patching*), ou triangulação. Exemplos típicos de algoritmos pertencentes à segunda categoria, intersecção de *voxels*, são *marching cubes* e *dividing cubes*, descritos mais adiante.

FIGURA 7 – ISOSUPERFÍCIE DA VELOCIDADE DO VENTO (40 m/s)



FONTE: CENTER FOR APPLIED COMPUTER SCIENCE, UNIVERSITY OF COLOGNE (2005)

3.4.1.4 Mapeamento sobre superfícies

Uma alternativa para as isolinhas consiste em traçar uma malha de linhas paralelas ao eixo x e y que estejam na superfície subjacente, projetando a malha em um plano 2D. A superfície pode ser exibida na forma de fio-de-arame ou tonalizada. Uma variação dessa técnica são os *height-fields*, que possibilitam a exibição de dois campos escalares definidos sobre um domínio 2D. Um campo é exibido como uma malha que descreve uma superfície, e o outro como um mapa de contornos tonalizados (ou uma imagem) posicionado sobre a superfície.

Dados em uma malha quadrada bidimensional (x, y) são mapeados em uma coordenada z, dando a aparência de um mapa de terreno, com montanhas e vales, embora a propriedade em questão nem sempre seja altura. [COLLINS, 1993]

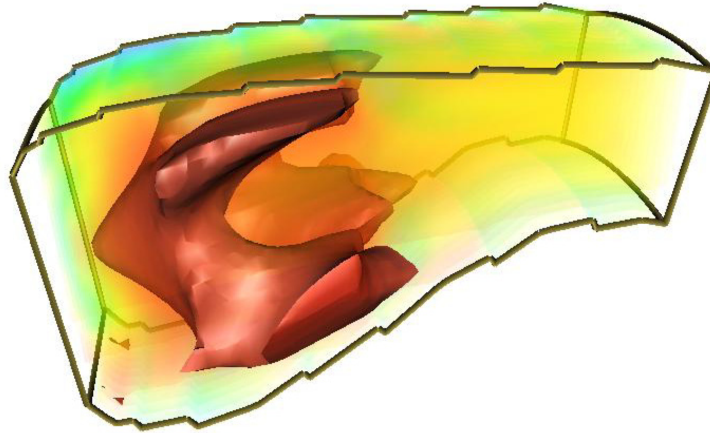
3.4.1.5 Renderização volumétrica

As técnicas de Renderização Volumétrica Direta (*Direct Volume Rendering*, ou DVR) permitem a renderização direta de conjuntos de dados 3D, sem a geração de primitivas geométricas intermediárias. Elas permitem ao usuário enxergar "dentro" (ou "através") do volume de dados, possibilitando a visualização de mais informações do que as técnicas de renderização de superfícies. Ao contrário do mapeamento de texturas, a imagem bidimensional resultante da renderização de volume pode ter a aparência de um gel semitransparente, devido ao fato de os valores dos dados serem mapeados tanto por cor como por transparência e todos os *voxels* contribuírem para a imagem final. Entretanto, pode causar alguma confusão quanto à identificação de determinados valores no meio do volume, o que não é desejado em análises numéricas. A Figura 8 demonstra este problema, com a renderização volumétrica de um campo escalar. O mapeamento de cores no interior do modelo se torna confuso devido ao uso de transparências. Estas técnicas podem ser agrupadas em duas categorias: as que operam no espaço do objeto e as que operam no espaço da imagem. A renderização volumétrica tem por objetivo visualizar a estrutura interna do volume de dados, sendo que para isso é necessário trabalhar com transparência. Portanto, define-se uma tabela que mapeia valores de dados em cores, e uma tabela de opacidades que mapeia valores de interesse em coeficientes que implicam maior grau de opacidade, e valores de menor interesse em coeficientes que implicam maior grau de transparência. As imagens são formadas pela combinação apropriada das cores e opacidades de todos os *voxels* que são projetados no mesmo pixel no plano da imagem.

O processo de definição das tabelas de cores e de opacidades, conhecido como classificação de dados, é crítico para gerar boas visualizações. Em geral, as tabelas são criadas pelo usuário depois de uma exploração de uma fatia do volume com uma sonda que imprime os valores dados, sendo que os intervalos são ajustados interativamente, até gerar um resultado aceitável. Por exemplo, uma tabela para renderizar dados obtidos por Tomografia Computadorizada poderia mapear a densidade associada aos ossos para branco/opaco, densidades de músculos para vermelho/semitransparente,

densidades associadas à gordura para bege/semitransparente.

FIGURA 8 – RENDERIZAÇÃO VOLUMÉTRICA



FONTE: ADAPTADO DE KITWARE (2005)

3.4.2 Técnicas para Entidades Vetoriais

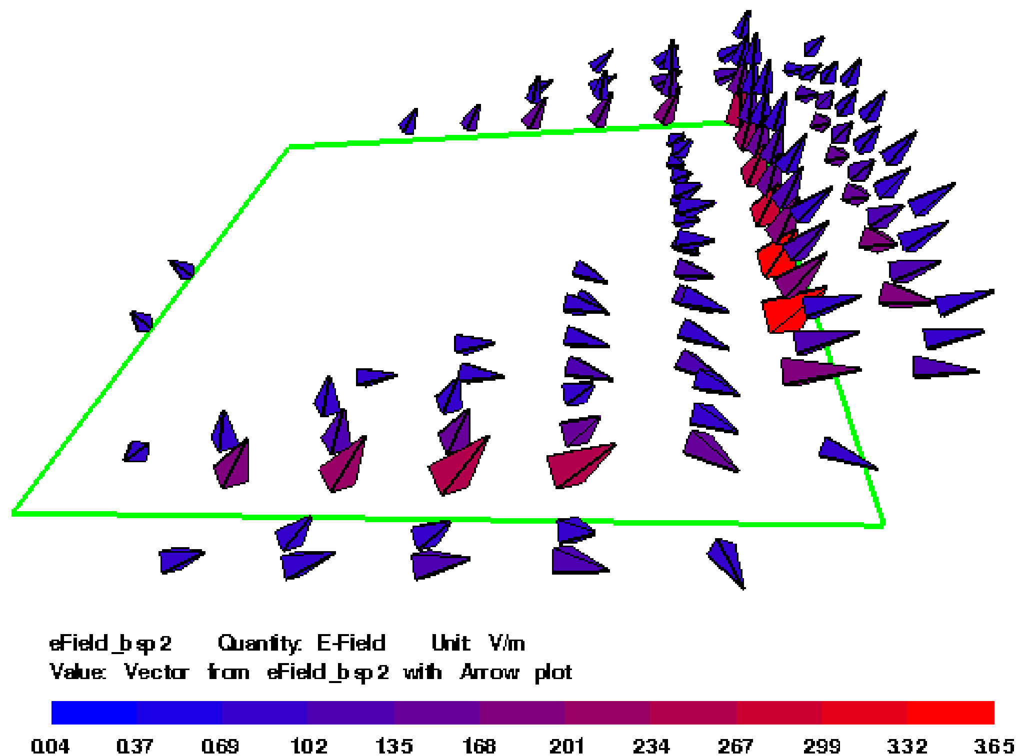
É comum que os dados sejam vetores definidos sobre pontos geométricos. Esses campos vetoriais podem ter duas, três ou mais dimensões, e podem estar definidos sobre regiões planares ou volumétricas. A visualização de campos vetoriais em uma tela 2D não é trivial. Por exemplo, é difícil interpretar uma imagem carregada de pequenas setas. Uma opção consiste em extrair uma quantidade escalar do campo vetorial, e usar alguma técnica aplicável a campos escalares. Mas isso implica em uma perda de informações, o que motiva intensa pesquisa em estratégias adequadas para a visualização de campos vetoriais.

3.4.2.1 Ícones pontuais

No meio científico, uma representação usual e bastante intuitiva para vetores é através de setas, ou seja, segmentos de reta orientados segundo a direção e sentido do vetor e, em geral, de comprimento proporcional à sua magnitude mesmo. Ao invés de se caracterizar a magnitude pelo comprimento da seta, pode-se usar setas de mesmo

tamanho e usar a espessura da seta para representar sua magnitude. Genericamente, a técnica computacional envolve a escolha de uma quantidade de pontos representativos do domínio e a atribuição, a cada um deles, de um ícone orientado, ou glifo, que indique a direção do campo vetorial no ponto (por exemplo, o deslocamento em um ponto). Um glifo é um objeto cuja aparência é afetada por atributos de um dado de entrada ao qual está associado. Esse objeto pode ser uma entidade geométrica, um conjunto de dados, ou uma imagem gráfica. No caso mais comum, o glifo é uma seta, mas outras representações gráficas, como barbelas, poderiam ser adotadas.

FIGURA 9 – PLOTAGEM DE ÍCONES PONTUAIS (SETAS)



FONTE: FRAUNHOFER IZM (2005)

O uso de ícones pontuais também é útil para visualizar fluxo 3D em uma seção transversal planar do volume. Vetores 3D são mostrados em um plano 2D, podendo apontar para dentro ou para fora da superfície de exibição. Pode-se exibir vetores 3D em todo o volume, aumentando o realismo através de *depth-cueing* e de efeitos de iluminação/tonalização no caso de vetores sólidos. Pode-se, também, mapear cor aos

vetores para fornecer informação adicional sobre a magnitude ou uma outra variável. A escolha do tipo do vetor e da forma das setas afeta a percepção da direção dos vetores. Entretanto, esse tipo de representação não se mostra muito eficiente no caso 3D geral, por uma série de motivos. Em particular, ocorre um congestionamento visual quando os ícones se sobrepõem desorganizadamente devido à projeção na imagem 2D final, tornando impossível extrair a estrutura intrínseca do campo vetorial. Isto é ilustrado na Figura 9, que mostra a representação de um campo vetorial com setas, associado a um mapeamento de cores. A utilização de setas de mesmo comprimento, mas variantes na espessura ou cor pode amenizar este fato. Outro problema ocorre devido à ambigüidade da projeção final dos glifos, pois, em geral, um segmento de reta projetado num plano pode corresponder a várias posições da mesma linha no espaço, gerando problemas de percepção de orientação. Além disso, caso se utilize projeção perspectiva, a dimensão das setas será alterada de acordo com a posição de visão.

3.4.2.2 Traço de partículas

As setas podem ser vistas como aproximações para o movimento de um ponto ao longo de um intervalo de tempo cuja duração é determinada pelo fator de escala. Isso sugere uma técnica alternativa: deslocar repetidamente pontos, ou partículas, ao longo de vários intervalos de tempo. As características do campo vetorial são usadas para mover as partículas no espaço e no tempo. Isso requer a reamostragem do campo vetorial em posições nem sempre correspondentes dos valores discretos dos dados. Em uma variação, rastro de partícula, as partículas antigas são mostradas por certo número de passos de tempo.

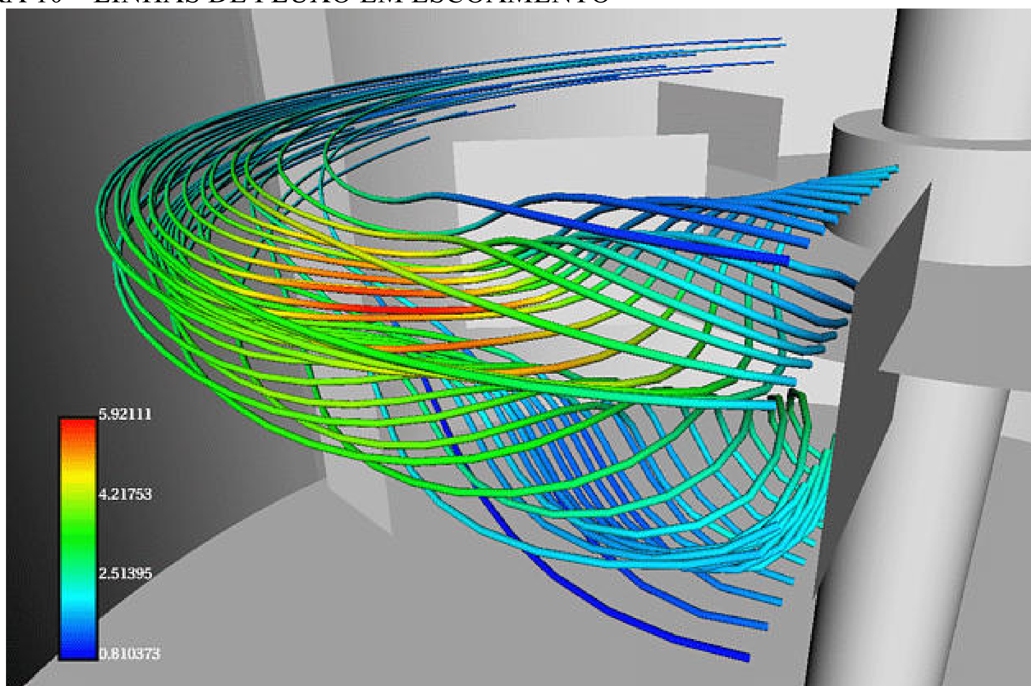
O olho tende a enxergar o caminho percorrido pelas partículas, dando ao observador uma idéia qualitativa do fluxo nessa região. Este caminho pode ser mostrado como uma animação ao longo do tempo (dando a ilusão do movimento), ou em uma única seqüência, dando a aparência de uma trajetória.

3.4.2.3 Linhas de fluxo

Uma extensão natural da técnica de animação de partículas é conectar a posição do ponto em vários instantes de tempo. O resultado é uma aproximação numérica para a trajetória de uma partícula, representada como uma linha. Genericamente, pode-se gerar linhas de escoamento que indicam a direção de escoamento de um fluxo, por exemplo. Uma extensão é a *streakline*, que são todas as linhas de corrente que passam por um ponto, mas se originam de diferentes posições.

Pode-se, também, usar cores para indicar a magnitude da velocidade da partícula a cada ponto da linha. Uma dificuldade comum às setas e linhas de corrente é a densidade de informações apresentadas. A Figura 10 mostra linhas de corrente de um fluxo, associadas ao mapeamento de cores da energia cinética da turbulência.

FIGURA 10 – LINHAS DE FLUXO EM ESCOAMENTO



FONTE: VISUALISIERUNG UND ANALYSE TURBULENTER STRÖMUNGEN (2005)

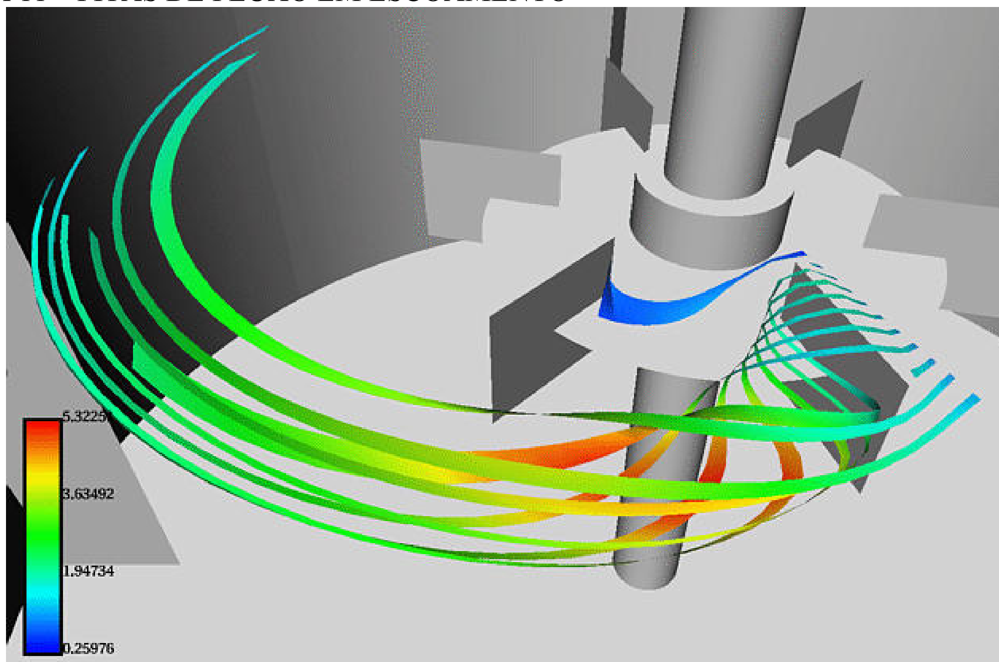
As linhas de escoamento são muito mais eficientes do que ícones pontuais para transmitir características gerais de escoamento, pois geram representações com menor congestionamento visual. Entretanto, os esquemas de interpolação e de

integração numéricos adotados são fontes de erros de visualização. Processos numéricos elevam o custo da visualização de linhas de escoamento consideravelmente em relação a ícones pontuais, principalmente no caso 3D.

3.4.2.4 Superfícies e fitas de fluxo

As linhas de escoamento permitem visualizar trajetórias de partículas em um campo vetorial. Colorindo essas linhas, ou criando glifos locais (como linhas pontilhadas ou cones orientados), pode-se transmitir informação adicional, por exemplo, uma variável escalar ou temporal. Entretanto, essas técnicas transmitem informações elementares sobre o campo vetorial, isto é, elas representam seus dados apenas ao longo da extensão de seu domínio espacial (em geral, pontos). Informações locais, como rotação, e informações globais, como tubo de vórtices, não são representadas.

FIGURA 11 – FITAS DE FLUXO EM ESCOAMENTO



FONTE: VISUALISIERUNG UND ANALYSE TURBULENTER STRÖMUNGEN (2005)

Uma extensão natural da técnica de linhas de corrente consiste em alargar a linha, criando uma fita, como na Figura 11, em que se pode perceber as rotações do

campo vetorial. A fita pode ser construída gerando duas linhas de fluxo adjacentes e conectando-as através de uma malha poligonal. A técnica funciona bem desde que as linhas de corrente originais permaneçam relativamente próximas entre si. Se elas divergem, a fita resultante não é uma boa representação do fluxo, pois a superfície da fita deve ser tangente ao campo vetorial a cada ponto sendo que, em geral, a malha poligonal conectando duas linhas de fluxo muito separadas entre si não satisfaz esse requisito.

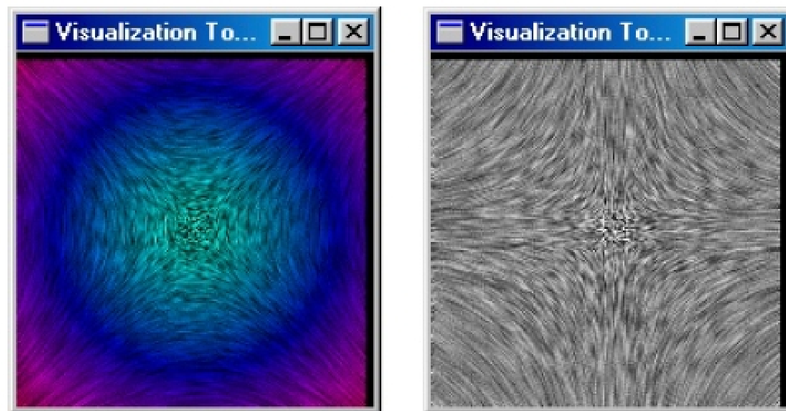
3.4.2.5 Mapeamento de texturas

Texturas são um recurso muito usado em CG para introduzir detalhes na superfície de objetos sem recorrer a primitivas gráficas para modelá-los. O mapeamento de textura pode ser visto como a colagem de uma imagem sobre a superfície de um objeto. O uso dessa técnica requer duas informações: a textura a ser mapeada e as coordenadas de mapeamento, que especificam a posição na qual a imagem será colada. Uma outra abordagem usa definições procedimentais para gerar a textura, ao invés de uma imagem. Nessa abordagem, durante o processo de renderização, para cada pixel é ativado um procedimento que gera a textura a ser associada a ele.

Existem várias aplicações úteis para o mapeamento de texturas em visualização. Padrões de textura podem ser gerados procedimentalmente como uma função dos dados. Um exemplo seria mudar a aparência de uma superfície com base no valor do dado local. Mapeamentos de textura podem ser animados em função do tempo. Escolhendo um mapeamento cuja intensidade varia monotonicamente de escuro para claro, e movendo a textura ao longo do objeto, o objeto parece movimentar-se na direção do movimento do mapa de textura. Essa técnica pode ser usada para simular o movimento de um campo vetorial ilustrando, por exemplo, a magnitude dos vetores ao dar aparência de movimento para objetos como setas. Ela pode, também, ser usada para visualizar campos escalares, variando a textura em relação ao valor do dado, equivalentemente ao mapeamento de cores.

MENDONÇA (2001) demonstra a aplicação de mapeamento de textura para Visualização Científica com o uso do VTK, mostrada na Figura 12. O mapeamento é feito com uma técnica chamada convolução integral de linha (*line integral convolution* - LIC), que faz uma interpolação de campo vetorial com um espaço de textura definido.

FIGURA 12 – COMPARAÇÃO ENTRE UMA IMAGEM LIC MAPEADA POR CORES E EM TONS DE CINZA



FONTE: MENDONÇA (2001)

3.4.2.6 Desenho de deformadas

Para o caso específico de representação de dados de deformação ou deslocamento de sólidos, o simples desenho da geometria deformada pode representar adequadamente os dados. Para isto, basta que se atualize o modelo geométrico. Normalmente, aumenta-se a magnitude dos deslocamentos, mantendo-se suas direções. Isto é feito principalmente quando esses valores são muito menores que as dimensões da geometria. Embora se perca a representação exata do modelo, esta desproporção demonstra o campo vetorial. A animação pode representar papel importante nesta técnica, proporcionando a percepção de que uma estrutura está se deformando em tempo real.

3.4.3 Técnicas para Entidades Tensoriais

A representação de tensores é um desafio na SciVis, pois é difícil representá-los em uma simples imagem bidimensional. Para solucionar essa questão, algumas estratégias podem ser adotadas. COLLINS (1993) sugere que tensores de ordem 3 podem ser simplificados para uma ordem menor. Um tensor de tensões (ordem 2) pode ser contraído com um vetor normal (ordem 1) para originar um vetor de superfície de tração (ordem 1). Nesse caso, as técnicas para escalares ou vetores seriam utilizadas.

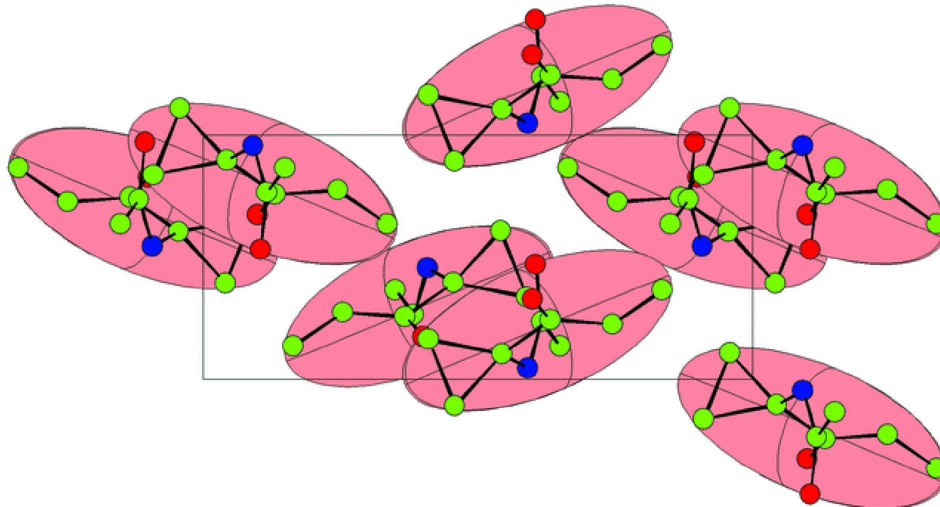
No entanto, com essas simplificações, perde-se a percepção completa do tensor.

3.4.3.1 Glifos ou ícones

Existem alguns glifos que podem ser utilizados para representar tensores. Por exemplo, pode-se representar os elementos de um tensor por três setas com cabeças duplas perpendiculares. Outro exemplo é o elipsóide de Lamé, usado para representar componentes espaciais de movimento térmico de um átomo em uma estrutura cristalina. Neste exemplo, os elementos da diagonal do tensor mapeiam os raios do elipsóide, como visto na Figura 13. Exemplo interessante é *shaft-and-disk* (haste e disco), usado por HABER e MCNABB (1990) para representar tensor de tensão. A direção da haste, sua cor, seu comprimento e seu eixo e a cor do disco representam os elementos do tensor.

Existem glifos dos mais variados tipos, e todos tentam representar os elementos de um tensor da melhor maneira possível. Entretanto, todas as representações com glifos levam a certa confusão por acúmulo de dados e sobreposição de imagens, especialmente em modelos densos com diversos pontos relevantes.

FIGURA 13 – REPRESENTAÇÃO DOS TENSORES ANISOTRÓPICOS DE FORMA EM MOLÉCULAS



FONTE: COOPER, FOXMAN E YANG (2004)

3.4.3.2 *Streamtubes* ou hiper-linhas de corrente

Outra alternativa para a representação de dados tensoriais é o uso de hiper-linhas de corrente (SCHROEDER, MARTIN e LORENSEN, 2004). Esta técnica alia características dos glifos e linhas de corrente, e consiste em criar uma linha de corrente através de uma das três direções do tensor, e então fazendo uma varredura com uma primitiva geométrica ao longo da linha. Normalmente, uma elipse é usada como primitiva, sendo que as duas outras direções do tensor definem as duas diagonais da elipse. Isto resulta num formato tubular. Outra primitiva possível é uma cruz, com seus braços representando direções do tensor.

3.4.4 Algoritmos de Visualização

Além dos algoritmos de interpolação, de tratamento de luz e cores, de mudança de coordenadas, de eliminação de superfícies escondidas, entre outros, existem algoritmos mais complexos, que realizam funções de extração de geometria e visualização volumétrica (VV).

A maioria dos algoritmos de VV adota uma projeção ortográfica, que garante que os analistas não serão confundidos pelos efeitos associados a uma visualização

perspectiva. Por outro lado, recursos como animação e tonalização precisam ser adicionados para transmitir a noção de profundidade perdida na projeção ortográfica.

A tonalização das imagens requer a especificação das posições das fontes de luz, e também o cálculo da normal à superfície em cada um de sues pontos. A maioria dos algoritmos de extração de superfícies e de DVR usam o gradiente de valores da malha para determinar a normal. O gradiente no interior de uma célula pode ser interpolado a partir dos valores nos vértices, que são conhecidos. Um sistema pode pré-calcular e armazenar o produto escalar entre o vetor de iluminação e o gradiente em cada vértice da malha. Isto requer um espaço adicional, mas permite renderização mais rápida, desde que a posição da fonte de luz não mude.

3.4.4.1 *Marching cubes* e derivados

O algoritmo de *marching cubes* é um processo de detecção de superfície. Ele é iniciado pela seleção do valor escalar de interesse. O *marching cubes* trata as células (*voxels*, no caso 3D) independentemente. Alguns *voxels* estão completamente dentro ou completamente fora do volume de interesse delimitado pela isosuperfície definida pelo valor de interesse, enquanto que outros são interceptados por ela. O pressuposto básico da técnica é que uma isosuperfície só pode atravessar uma célula segundo um número finito de maneiras distintas, as quais dependem de quais vértices da célula estão dentro do volume delimitado pela isosuperfície, e quais estão fora.

O algoritmo prossegue (“marcha”) de célula em célula, avaliando se esta está dentro, fora, ou é cortada por uma superfície. Depois que todas as células foram visitadas, a isosuperfície, definida em termos de uma malha poligonal, é criada, e está pronta para ser renderizada.

Um problema do *marching cubes* é que existem casos ambíguos. Esta ambigüidade ocorre na face de um *voxel* quando vértices em arestas adjacentes estão em estados diferentes, mas vértices diagonais estão no mesmo estado (todos dentro ou todos fora).

Várias abordagens foram adotadas para resolver esse problema, e o algoritmo

tetraedros marchantes (*marching tetrahedra*) é uma delas. Ele opera decompondo os cubos em tetraedros. Os testes de intersecção com arestas são feitos com os tetraedros. A decomposição em tetraedros elimina os casos ambíguos, mas gera isosuperfícies compostas por mais triângulos, exigindo mais memória e mais tempo de processamento.

O *dividing cubes* também é um algoritmo para geração de isolinhas ou isosuperfícies. Ele é semelhante ao *marching cubes*, exceto pelo fato de que, depois de determinar que um *voxel* seja interceptado pela isosuperfície, o algoritmo projeta o *voxel* no plano de imagem. Se o *voxel* é projetado em um único pixel (ou uma região menor que um pixel), ele é renderizado como um ponto. Caso contrário, o *voxel* é subdividido em sub-*voxels*, até que, eventualmente, o conteúdo de um sub-*voxel* será projetado em um único pixel. A idéia básica do algoritmo é gerar uma superfície definida por pontos, ao invés de triângulos: dessa forma, não é preciso calcular interseções ou ajustar triângulos.

Uma desvantagem de criar isosuperfícies como nuvens de pontos suficientemente densas é que a magnificação da superfície (através de um *zoom*, por exemplo), revela sua natureza descontínua. Portanto, o conjunto de pontos já deve ser construído antecipando o *zoom* máximo permitido, ou deve ser construído dinamicamente.

3.4.4.2 *Splatting*

Splatting é um algoritmo de DVR que opera na ordem dos objetos, ou seja, os *voxels* são projetados diretamente no plano de imagem. O algoritmo percorre os *voxels* em uma travessia frente-para-trás (a frente do volume é a que está mais próxima do olho do observador, com o plano da imagem posicionado entre o observador e o volume), sendo que a contribuição de cada *voxel* é calculada e combinada com as contribuições dos outros *voxels* que se projetam no mesmo pixel.

A cada *voxel* é atribuída uma cor e uma opacidade, de acordo com as tabelas de mapeamento definidas. A seguir, o *voxel* é projetado no plano da imagem.

A ordem de travessia pode ser alterada. A travessia frente-para-trás apresenta vantagens em alguns casos. Por exemplo, o processo de composição para um pixel pode ser interrompido assim que a sua opacidade atinge o valor máximo (1). Em uma travessia trás-para-frente o valor do pixel pode mudar significativamente durante o processo de renderização. Por outro lado, este tipo de travessia pode ser útil para o usuário ir visualizando as imagens progressivamente, vendo as estruturas que serão posteriormente ocultas por outras.

3.4.4.3 *Ray tracing* e *ray casting*

Técnicas que operam no espaço da imagem determinam como as amostras de dados afetam cada pixel da imagem. *Ray tracing* é um método para renderização de imagens. Ele trabalha traçando o caminho percorrido por um raio de luz através da cena, e calculando reflexão, refração, ou absorção do raio quando ele cruza um objeto. No *ray tracing* tradicional, os raios interagem apenas com a superfície dos objetos, e um raio pode gerar vários outros quando atinge uma superfície. Já no *ray casting* volumétrico, um raio continua o seu caminho pelo interior do volume em linha reta, até que a opacidade acumulada no caminho do raio atinja o valor máximo (1), sendo que não são geradas sombras ou reflexões no processo. Esse é o algoritmo mais usado para a geração de imagens volumétricas de alta qualidade.

O processo básico corresponde a disparar um raio a partir da câmera, passando por um determinado pixel no plano da imagem e penetrando no volume. O raio é amostrado em pontos discretos, separados por uma distância compatível com a resolução da espacial dos dados. O valor associado a um ponto no interior de uma célula é calculado por interpolação constante ou por interpolação linear a partir dos valores nos vértices. Em seguida, as tabelas de mapeamento são usadas para determinar a cor e opacidade associadas a esse valor do dado. Os valores de cor e opacidade são adicionados ao valor do pixel usando uma fórmula ponderada, e o raio prossegue para o próximo ponto de reamostragem. Para cada *voxel* cruzado pelo raio são acumuladas cor e opacidade associadas, até a opacidade atingir o valor 1 ou até

que o raio deixe o volume. O processo é repetido para cada pixel no plano da imagem.

Uma vantagem dessa técnica é que objetos pequenos ou pouco definidos são mostrados. Além disso, o processo não envolve uma classificação binária do tipo dentro ou fora, como nos algoritmos de extração de superfícies. As imagens mostram o conteúdo do volume inteiro, e não apenas uma superfície em particular. Adicionalmente, o traçado de um raio é independente dos demais, o que torna o algoritmo facilmente paralelizável. Uma possível desvantagem é o alto custo computacional. Estratégias podem ser adotadas para acelerar o processo, sendo que algumas envolvem o uso de estruturas de dados sofisticadas para armazenar o volume.

3.5 CARACTERÍSTICAS DOS PACOTES DE VISUALIZAÇÃO CIENTÍFICA

Os diversos pacotes de Visualização Científica reúnem as características expostas. A maioria deles possui um modelo de fluxo de dados igual ou semelhante ao desenvolvido por UPSON et al. (1989). Além disso, eles devem ser eficientes, tanto em relação ao armazenamento, quanto à velocidade de processamento, possuindo uma boa estrutura de dados. Eles devem, ainda, disponibilizar algumas das técnicas expostas, trabalhando com alguma biblioteca gráfica como, por exemplo, a OpenGL (OPENGL, 2005).

Mesmo oferecendo diversos recursos e possuindo uma arquitetura bastante complicada, eles devem oferecer possibilidade de acesso em uma camada compreensível, para que um usuário que não seja especialista em Computação Gráfica possa utilizá-lo sem dificuldade.

Os pacotes devem também trabalhar com diferentes tipos de dados e poder gravar arquivos com as informações relevantes, possibilitando armazenamento e leitura. Muitos deles possuem recursos de importação e exportação de arquivos, fazendo com que diferentes aplicativos possam compartilhar os mesmos dados.

Devido ao fato de serem de uso geral, esses pacotes perdem um pouco de eficiência quando comparados a aplicativos de uso único, mas normalmente são muito mais robustos e podem ser utilizados em muitas áreas diferentes.

4 O *VISUALIZATION TOOLKIT* - VTK

O *Visualization Toolkit* (VTK) é um sistema portável, orientado a objetos e de código aberto para computação gráfica tridimensional, visualização, e processamento de imagem (SCHROEDER, MARTIN e LORENSEN, 2004) distribuído gratuitamente pela Kitware. Implementado em C++, ele suporta também as linguagens Tcl, Python e Java, permitindo aplicações complexas e desenvolvimento rápido. Embora o VTK não possua componentes de interface, ele pode ser integrado com os já existentes, como Tk ou X/Motif (SCHROEDER, AVILA e HOFFMAN, 2000).

Este pacote de ferramentas surgiu em dezembro de 1993 como software que acompanhava a primeira edição do livro de SCHROEDER, MARTIN e LORENSEN (2004). Desde então, diversas pessoas têm participado em seu desenvolvimento, fato possível graças à opção dos autores de tornar o código do sistema aberto.

Por ter sido desenvolvido de acordo com o paradigma de orientação a objetos para programação, os componentes do VTK possuem características tais como herança, encapsulamento e polimorfia.

O VTK é portável, ou seja, trabalha em diversas plataformas (combinação de sistema operacional, hardware e compilador). Ele funciona, por exemplo, nos sistemas operacionais Microsoft Windows e no Linux rodando em máquinas Intel e Apple Macintosh com sistema OSX. Esta característica é importante, pois facilita a distribuição de aplicações desenvolvidas para usuários e outros desenvolvedores variados. Isso só é possível graças às características de implementação, que em sua estrutura orientada a objetos permite ao desenvolvedor ignorar a plataforma utilizada quando cria um objeto. Um exemplo é a classe *vtkRenderWindow*, cujo diagrama de classes está ilustrado na Figura 14. Ao instanciar-se um objeto dessa classe, internamente, uma classe específica, como por exemplo, a *vtkWin32OpenGLRenderWindow*, é instanciada para operar corretamente com a plataforma utilizada. O código permanece idêntico em qualquer sistema.

FIGURA 14 – DIAGRAMA DA CLASSE VtkRenderWindow

FONTE: ADAPTADO DE KITWARE (2004)

Este capítulo trata das principais características do *Visualization Toolkit*, começando por sua instalação, descrevendo sua arquitetura, seus tipos de dados e suas técnicas.

4.1 INSTALAÇÃO

Existem duas maneiras de instalação do VTK: a instalação das bibliotecas pré-compiladas e a compilação do código fonte.

A primeira opção é muito mais simples e rápida, mas não está disponível para todos os sistemas operacionais em que o VTK pode ser utilizado, apenas para o Microsoft Windows.

Por outro lado, a segunda opção pode ser utilizada nas principais plataformas existentes, mas demanda mais tempo e é mais complicada.

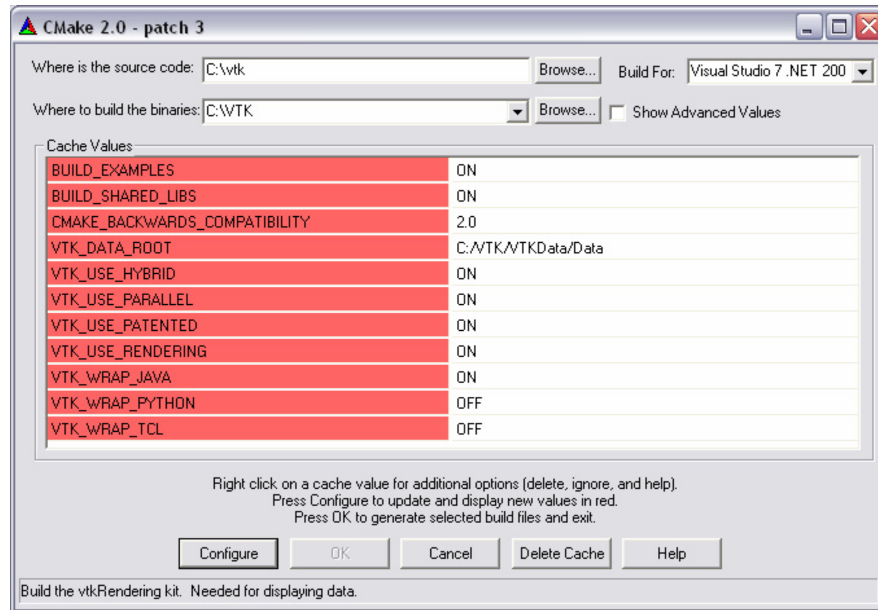
4.1.1 CMake

Para a compilação do VTK e dos códigos desenvolvidos com ele, de forma independente, é útil a ferramenta CMake (MARTIN e HOFFMAN, 2004). Também de código aberto, ela serve para a configuração do processo de criação de códigos multiplataforma.

Arquivos simples, independentes de plataforma (CMakeList.txt) são usados para descrever o processo e listar dependências necessárias na compilação. Quando o CMake é executado, ele gera arquivos nativos para o sistema operacional e o compilador utilizados. Por exemplo, para o Microsoft Visual C++, *workspaces* são criados. Já no Unix, são criados os arquivos *makefiles*. Dessa maneira, é possível compilar o VTK em qualquer computador usando um único código fonte, e trabalhar com as ferramentas de desenvolvimento que são naturais na plataforma em que se está trabalhando. O código fonte e a versão compilada do CMake podem ser encontrados em KITWARE (jul. 2005).

Para se executar o CMake, cujo exemplo de uso está mostrado na Figura 15, são necessárias inicialmente três informações básicas: qual compilador utilizar, onde está o código fonte, e onde gravar os objetos criados, ou seja, bibliotecas e arquivos binários. Depois de fornecidas essas informações, entra-se na etapa de configuração. Nessa etapa, são encontrados os recursos do sistema e determinada a configuração. Também são mostradas algumas opções de controle do processo de criação, em que se pode selecionar itens a serem compilados. Depois de configurado, chega-se a etapa de geração, na qual os arquivos específicos do compilador em uso são criados.

FIGURA 15 – EXEMPLO DE USO DO CMAKE

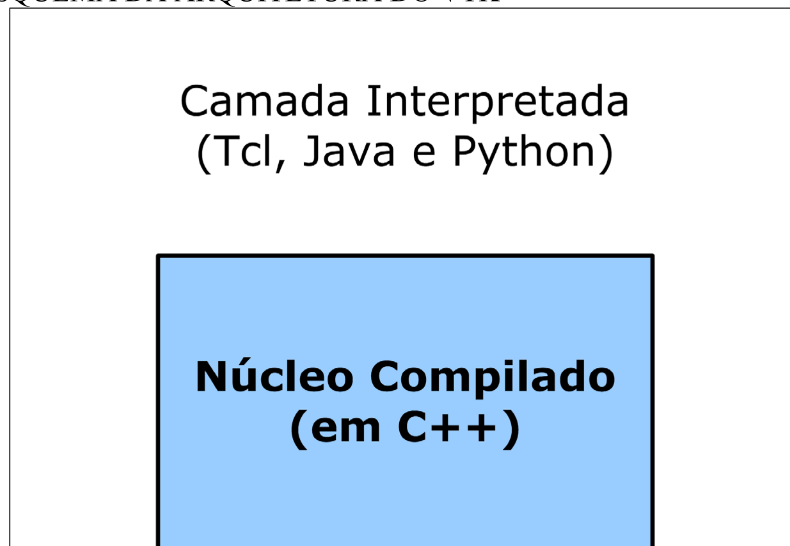


FONTE: O AUTOR

4.2 ARQUITETURA

O *Visualization Toolkit* consiste de duas partes principais: uma biblioteca de mais de 700 classes em C++ compiladas, e uma camada interpretada que permite a utilização das classes pelas linguagens Java, Tcl e Python, como representado na Figura 16.

FIGURA 16 – ESQUEMA DA ARQUITETURA DO VTK



FONTE: ADAPTADO DE SCHROEDER, MARTIN E LORENSEN (2004)

Este tipo de arquitetura permite a criação de algoritmos eficientes (tanto relativamente a memória quanto a processamento) na linguagem C++, e sua utilização em outras linguagens. O núcleo corresponde a estruturas de dados, algoritmos e funções de sistema, e proporciona eficiência e velocidade adequados. A camada interpretada oferece flexibilidade e extensibilidade. Por exemplo, a utilização de ferramentas de GUI (*Graphic User Interface*, ou interface gráfica do usuário), tais como Tcl/Tk, Python/Tk e Java AWT, permite rápido desenvolvimento de aplicações profissionais (SCHROEDER, MARTIN e LORENSEN, 2004).

O VTK possui dois subsistemas principais, o modelo gráfico e o de visualização.

4.2.1 Modelo Gráfico

O Modelo Gráfico do VTK cria uma camada abstrata sobre a linguagem gráfica, como por exemplo, OpenGL, para assegurar a portabilidade entre plataformas (SCHROEDER, AVILA e HOFFMAN, 2000).

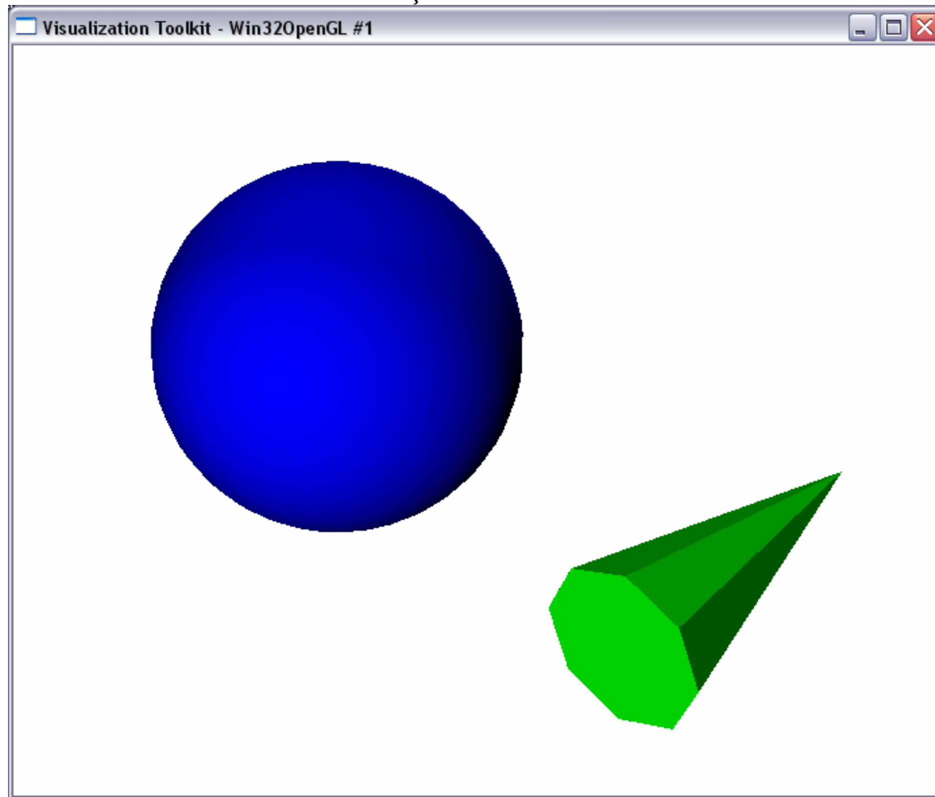
Existem sete objetos básicos utilizados no VTK para a renderização de uma cena, embora existam muitos outros por trás de todo o processo. Estes objetos são, de acordo com SCHROEDER, MARTIN e LORENSEN (2004):

a) *vtkRenderWindow*: É a janela na qual ocorre a renderização;
b) *vtkRenderer*: Coordena o processo de renderização, envolvendo luz, câmeras e atores;
c) *vtkLight*: São as fonte de luz para iluminação da cena;
d) *vtkCamera*: A câmera da cena;
e) *vtkActor*: Representa um objeto renderizado na cena, suas propriedades e posição;
f) *vtkProperty*: Define as propriedades de aparência de um ator, tais como cor e transparência;
g) *vtkMapper*: Define a representação geométrica para um ator.

A Figura 17 exemplifica uma imagem renderizada com o VTK. O código-

fonte do algoritmo que produziu a figura é encontrado nos apêndices, e contém os item básicos listados.

FIGURA 17 – EXEMPLO DE RENDERIZAÇÃO COM VTK



FONTE: O AUTOR

A classe *vtkRenderWindow* junta todo o processo de renderização. Ela é responsável pelo controle de uma janela no dispositivo de vídeo. Nos computadores utilizando Windows, ela será uma janela do Microsoft Windows; nos sistemas Linux e Unix, será uma *X Window*; e no Mac (OSX), uma janela *Quartz*. No VTK, as instâncias de *vtkRenderWindow* são independentes de dispositivo. Independentemente do hardware e do software utilizados, elas se adaptarão automaticamente à configuração do computador quando criadas.

A classe *vtkRenderer* é responsável pela junção de informações sobre luzes, câmera, e atores, utilizados na produção da imagem. Cada instância mantém uma lista ativa dessas entidades em uma cena particular. Pelo menos um ator precisa ser definido. Se luzes e câmera não forem definidos, eles serão criados automaticamente

pelo *vtkRenderer*. É necessário que o *vtkRenderer* esteja associado a um objeto da classe *vtkRenderWindow* no qual haverá o desenho da cena.

As duas classes supramencionadas são usadas para gerenciar a interface entre o hardware gráfico e o sistema operacional do computador.

Instâncias da classe *vtkLight* são responsáveis pela iluminação da cena. As luzes no VTK podem ser posicionais, com um cone de iluminação e fatores de atenuação associados, ou posicionadas no infinito, com raios paralelos entre si.

As câmeras, criadas por *vtkCamera*, controlam como a geometria 3D será projetada em uma imagem 2D durante o processo de renderização. Elas possuem parâmetros como: posição, ponto focal, localização dos planos que determinam o volume de visão, vetor que aponta para cima e campo de visão.

vtkActor representa os objetos na cena. Em particular, ele combina propriedades do objeto, como cor, tipo de sombra, transparência, definição geométrica, e orientação no sistema de coordenadas global. Tudo isso é implementado pela criação automática de diversos outros objetos como, por exemplo, *vtkProperty* e *vtkTransform*. Existem, ainda, diversas outras classes de atores com comportamento especializado, implementados como subclasses de *vtkActor*.

Instâncias de *vtkProperty* afetam a aparência de renderização de um ator. Quando os atores são criados, uma instância de suas propriedades é automaticamente criada com ele. Entretanto, também é possível criar objetos de propriedade diretamente e, então, associá-lo a um ou mais atores. Desta forma, dois atores podem compartilhar das mesmas propriedades.

Existem ainda outras classes muito úteis no processo de renderização, como por exemplo, *vtkRenderWindowInteractor*, que permite a interação do usuário com a cena de maneira simples e com muitos recursos.

4.2.2 Modelo de Visualização

O processo de visualização (*pipeline* de visualização) é responsável por construir a representação geométrica que é então renderizada pelo *pipeline* gráfico.

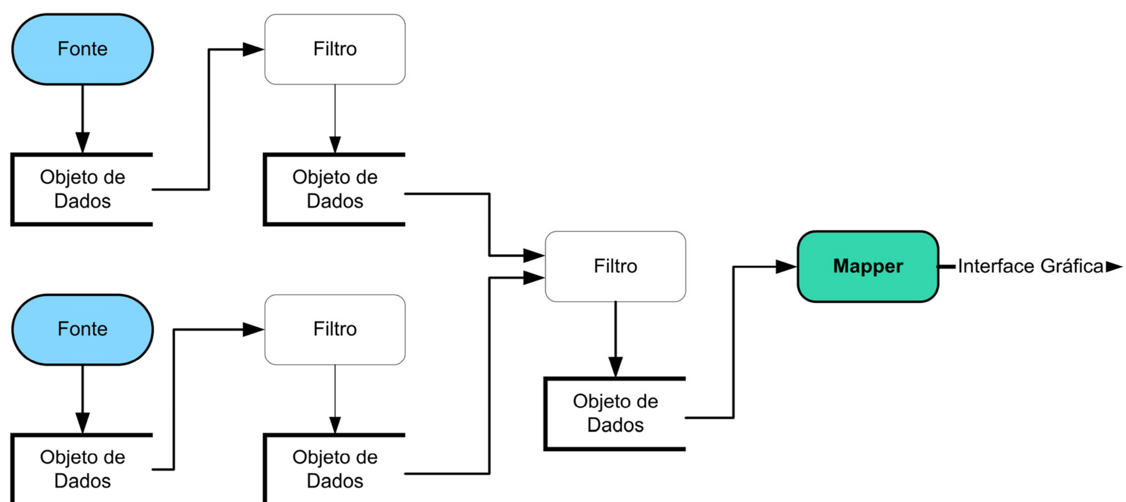
O VTK usa um fluxo de dados para transformar informação em dados gráficos. Existem dois tipos básicos de objetos envolvidos nesse fluxo: *vtkDataObject* e *vtkProcessObject* (SCHROEDER, MARTIN e LORENSEN, 2004).

O primeiro corresponde aos objetos de dados, que representam diversos tipos de dados. Vários conjuntos de dados podem ser representados no VTK com uma estrutura formal. Esses objetos consistem de estrutura geométrica e topológica, e de atributos de dados associados, como por exemplo, valores escalares e vetoriais.

Já o segundo corresponde aos objetos de processo, ou, mais genericamente denominados, filtros. Eles operam nos objetos de dados para produzir novos objetos de dados, e representam ainda os algoritmos do sistema.

Os objetos de processo e de dados são conectados para formar o *pipeline* de visualização, ou seja, a rede de fluxo de dados, como mostrado na Figura 18. Os filtros recebem um ou mais objetos de dados e geram, também, um ou mais objetos de dados, que são então transformados em dados gráficos pelos *mappers* e renderizados.

FIGURA 18 – *PIPELINE* DE VISUALIZAÇÃO DO VTK

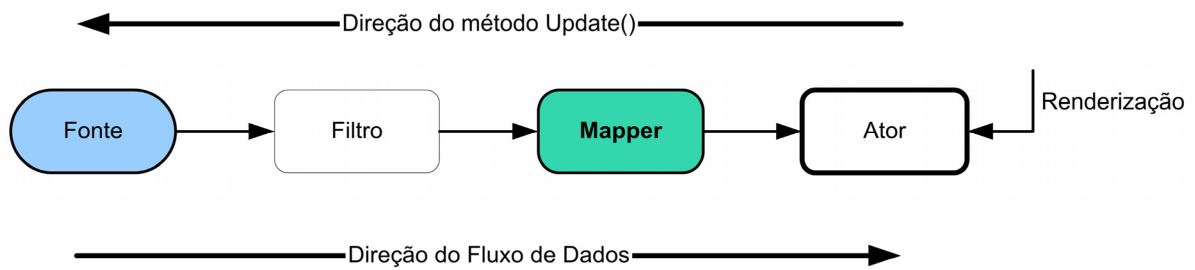


FONTE: ADAPTADO DE SCHROEDER, MARTIN E LORENSEN (2004)

4.2.3 *Pipeline* de Execução

Os *pipelines* de visualização do VTK só são executados quando algum dado relacionado é requisitado. A Figura 19 ilustra em alto nível de abstração o pipeline de execução. Nela, o método *Render()* normalmente inicia a requisição dos dados, que é entregue com o fluxo de dados. Dependendo de quais partes do pipeline estão desatualizados, os filtros podem ser re-executados, para trazer os dados atualizados ao fim da cadeia de fluxo, que podem então ser renderizados por um ator.

FIGURA 19 – *PIPELINE* DE EXECUÇÃO DO VTK



FONTE: ADAPTADO DE SCHROEDER, MARTIN E LORENSEN (2004)

4.3 REPRESENTAÇÃO DE DADOS

A forma mais geral de dado no VTK é o objeto de dados. Ele é uma coleção de dados sem nenhuma forma, e representa os dados que são processados pelo *pipeline* de visualização. Sozinho, ele não tem muita informação útil. Somente quando organizado em uma estrutura, eles fornecem uma forma que pode ser operada pelos algoritmos de visualização.

4.3.1 Conjuntos de Dados

Os objetos de dados com uma estrutura organizacional e atributos de dados associados formam conjuntos de dados (*datasets*). A maioria dos filtros, ou objetos de

processo, no VTK opera com esses conjuntos de dados.

A estrutura desses conjuntos possui duas partes: uma topologia e uma geometria. A topologia é o conjunto de propriedades invariantes sob certas transformações geométricas (rotação, translação e escala não-uniforme). Já a geometria é a instancialização da topologia, ou a especificação da posição no espaço tridimensional. Por exemplo, fazendo com que um volume seja especificado por quatro pontos, define-se a topologia. Determinando-se as coordenadas desses pontos, especifica-se a geometria.

Os atributos dos dados associados aos conjuntos de dados são informações suplementares associadas com a geometria e/ou a topologia.

No modelo de conjuntos de dados do VTK, a estrutura compõe-se de células e pontos. Estes definem a geometria, enquanto aquelas, a topologia. Os atributos podem estar ligados às células ou aos pontos, e podem ser escalares, vetoriais, normais, coordenadas de texturas ou tensores.

4.3.2 Tipos de células

Um conjunto de dados consiste de uma ou mais células, os blocos fundamentais de um sistema de visualização. Elas são definidas especificando-se um tipo em combinação com uma lista ordenada de pontos. Esta lista, denominada lista de conectividade, combinada com o tipo de célula especificado, define implicitamente a topologia da célula. As coordenadas dos pontos definem sua geometria.

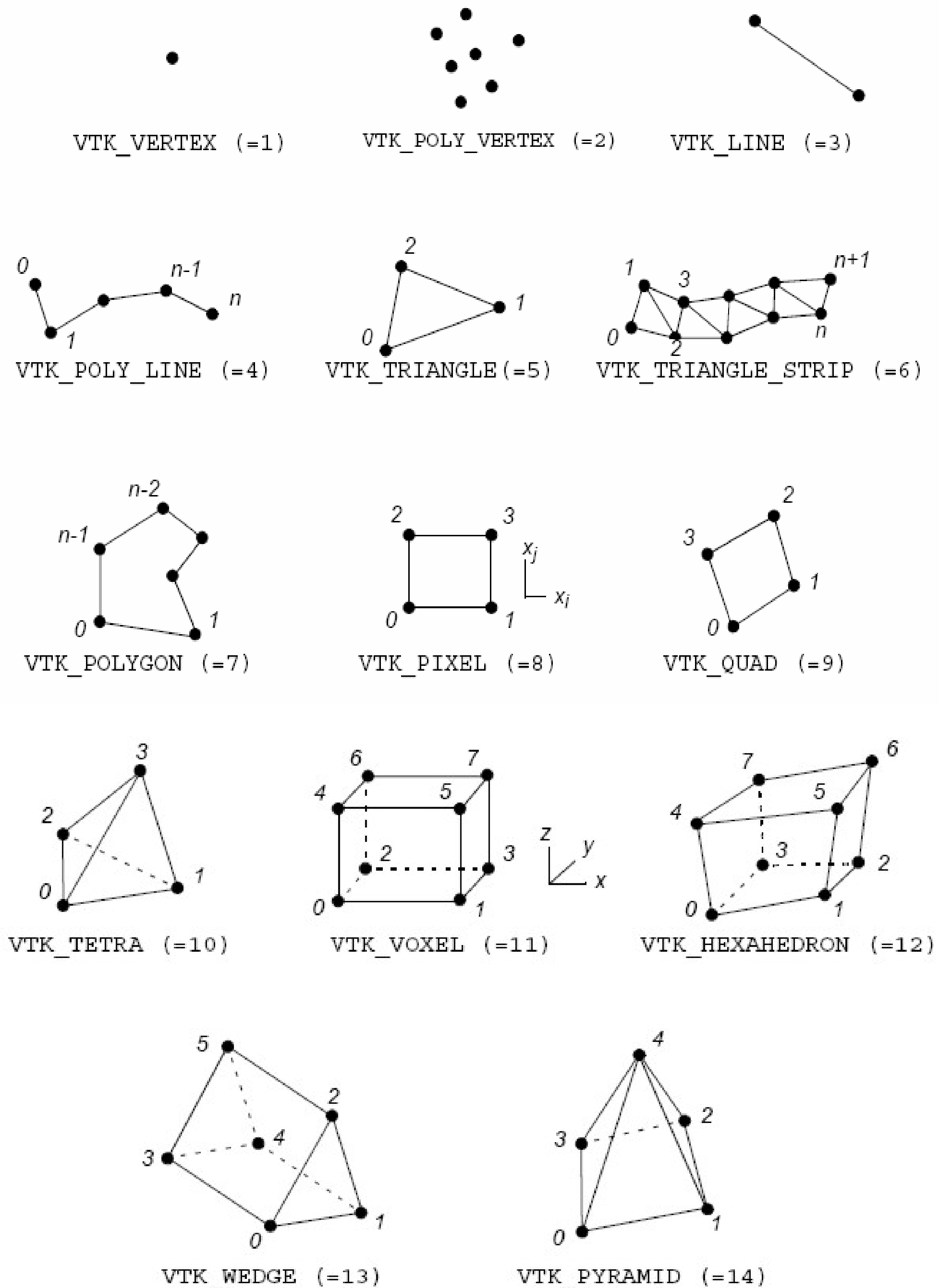
No VTK existem células lineares, ou seja, células que possuem funções de interpolação constantes e conseqüentemente, possuem arestas retas, e células não-lineares, com funções de interpolação quadráticas.

A Figura 20 mostra os 14 tipos de células lineares possíveis com o VTK. Os nomes abaixo das células são os usados pelo *toolkit*, e os números entre parênteses são os respectivos códigos.

O primeiro tipo de célula é o vértice (*vertex*), definido por um único ponto. O polivértice (*polyvertex*) é um conjunto de vértices, definido por um grupo arbitrário de

pontos. As linhas são definidas por dois pontos, sendo sua direção do primeiro para o segundo ponto. A poli linha é uma célula composta por uma ou mais linhas conectadas. Ela é definida por uma lista ordenada de *n+1* pontos, em que *n* é o número de linhas na polilinha.

FIGURA 20 – CÉLULAS LINEARES DO VTK

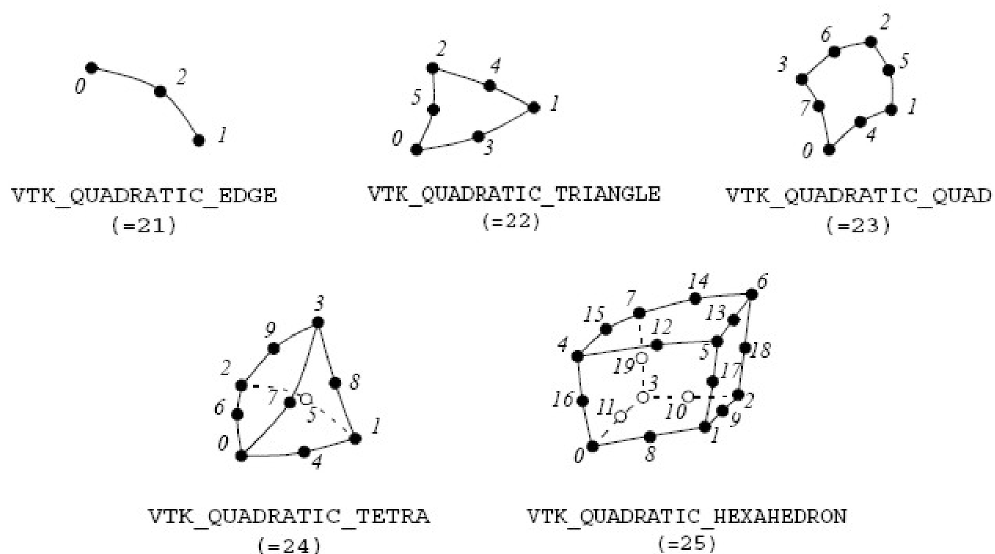


FONTE: KITWARE (2004)

A célula bidimensional mais simples é o triângulo, definido por uma lista de três pontos. A ordem dos pontos especifica a direção da normal da superfície pela regra da mão direita. A faixa de triângulos (*triangle strip*) é um conjunto de triângulos. O quadrilátero é definido por quatro pontos pertencentes a um mesmo plano, sendo sua normal também definida pela regra da mão direita. Ele é convexo e suas arestas não se cruzam. O pixel também é definido por quatro pontos, mas com a adição de constantes geométricas. Cada aresta do pixel é perpendicular a sua aresta adjacente, e paralela a um dos eixo cartesianos. Portanto, a normal também é paralela a um dos eixos. O polígono é definido por uma lista de três ou mais pontos coplanares. Ele pode ser côncavo, mas não deve possuir furos nem intersecções.

O tetraedro é definido por quatro pontos não coplanares. Ele possui seis arestas e quatro faces triangulares. O hexaedro possui seis faces quadrangulares, doze arestas e oito vértices, que o definem. O voxel é também um hexaedro, e é o equivalente tridimensional do pixel. Outra célula existente é o prisma de base triangular, definido por seis pontos. Existe outra célula linear, a pirâmide. De base quadrangular, ela é definida por cinco pontos. Além destas células com arestas retas, existem células não-lineares. Elas são construídas adicionando-se nós no ponto médio de cada aresta. A Figura 21 mostra as células não-lineares existentes.

FIGURA 21 – CÉLULAS NÃO-LINEARES DO VTK



FONTE: KITWARE (2004)

A primeira célula não-linear é a aresta quadrática, definida por três pontos. Os dois primeiros definem os pontos inicial e final, e o terceiro ponto corresponde ao central. Analogamente, o triângulo quadrático é definido por seis pontos. Os três primeiros definem os vértices e os outros três os pontos médios das arestas. O quadrilátero quadrático é definido por oito pontos. O tetraedro quadrático é definido por dez. Os quatro primeiro determinam os vértices e os seis restantes os pontos médios das arestas. Analogamente, o hexaedro quadrático é definido por 20 pontos.

Uma diferença significativa entre células lineares e não lineares é o modo como são renderizadas e operadas pelos diversos algoritmos de visualização. As lineares são facilmente convertidas em primitivas gráficas lineares, as quais são processadas pela biblioteca gráfica. Já as não-lineares normalmente não possuem suporte direto pela biblioteca gráfica. Portanto, elas precisam ser tratadas de forma especial pelo sistema de visualização. O VTK, neste caso, transforma as células não-lineares em lineares, dividindo-as. Este processo é denominado *tesselation*. Por exemplo, uma aresta quadrática é dividida, em seu ponto médio, em duas outras, criando duas arestas lineares que podem ser tratadas como primitivas gráficas. Esse método não é perfeito, pois acaba simplificando a geometria, mas produz resultados aceitáveis em modelos densos, e em análises numéricas.

Dentre todos os tipos de células do VTK, algumas se destacam pelo uso em análise pelo MEF. As células representam os próprios elementos, e os pontos seus nós. Desta forma, não é necessária nenhuma adaptação para representação dos modelos de elementos finitos. No caso bidimensional, existem o triângulo e o quadrilátero. Já no caso tridimensional, existem tetraedros e hexaedros. Esses são os tipos de elementos comumente utilizados, tanto lineares quanto não-lineares, em análises numéricas, pois possuem, normalmente, maior precisão.

4.3.3 Atributos de Dados

Os atributos são informações associadas à estrutura do conjunto de dados. Isso inclui tanto a geometria quanto a topologia. O VTK possibilita a associação de

atributos tanto aos pontos quanto às células. Em EF, normalmente, resultados como deslocamento e tensão são associados aos nós. Entretanto, eles podem ser associados com os próprios elementos.

O primeiro tipo de atributo associado são os escalares. São valores únicos associados a cada parte do conjunto de dados. Exemplos são distribuição de temperatura, componente de tensão em uma direção, densidade, entre outros.

Os vetores são dados com uma magnitude e uma direção. No espaço tridimensional, são representados por três valores componentes, (u, v, w). Exemplos são: deslocamento, velocidade de fluxo, função gradiente e outros. Além de vetores genéricos, é possível associar dados a normais, ou seja, vetores unitários. Eles são normalmente usados para controlar efeitos de sombra em objetos, e também para algumas técnicas de visualização.

Tensores são generalizações matemáticas de vetores e matrizes. Um tensor pode ser considerado como uma tabela, que pode possuir nenhuma dimensão (um escalar), uma dimensão (vetor), duas dimensões (matriz), ou até maiores dimensões. O VTK trata apenas de tensores simétricos reais 3 x 3.

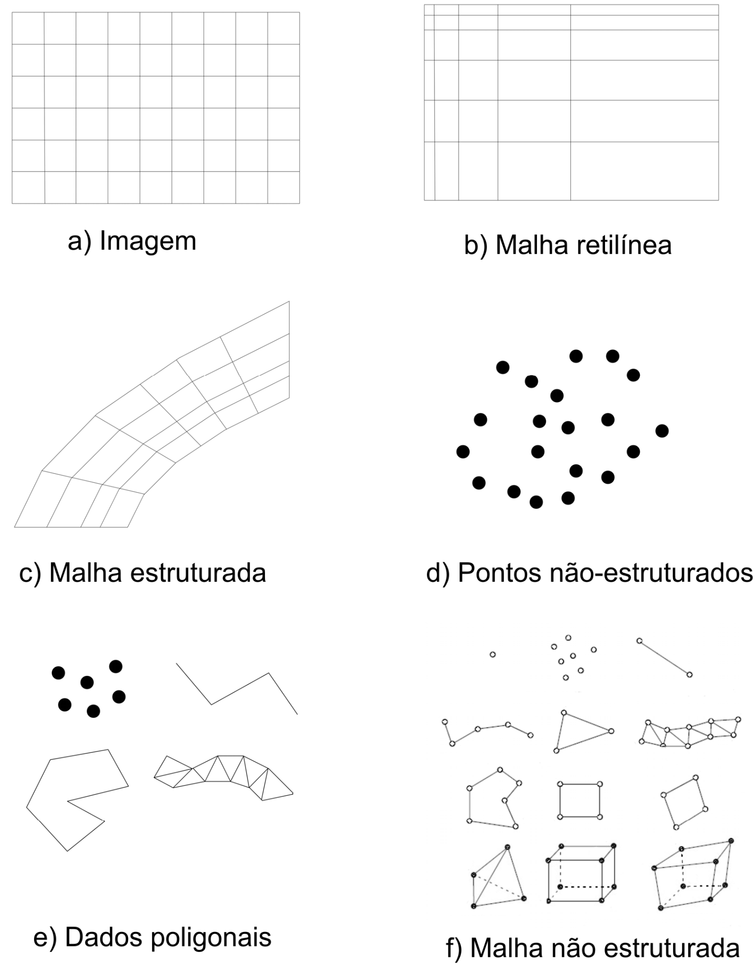
Outro atributo que pode ser associado consiste de coordenadas de textura. Elas são usadas para mapear um ponto no espaço em um espaço de textura. Uma aplicação de textura em duas dimensões pode ser entendida como "colar" uma fotografia em um ou mais polígonos.

4.3.4 Tipos de conjuntos de dados

Um conjunto de dados consiste de uma estrutura associada a um atributo. A estrutura possui propriedades topológicas e geométricas e é composta de um ou mais pontos e células. Os tipos comuns de conjuntos de dados do VTK são mostrados na Figura 22.

Um conjunto de dados é caracterizado de acordo com sua estrutura: se ela é regular ou irregular. Os conjuntos não estruturados tendem a ser mais gerais, mas requerem maiores recursos computacional e de memória.

FIGURA 22 – TIPOS DE CONJUNTOS DE DADOS DO VTK



FONTE: ADAPTADO DE KITWARE (2004)

Exemplo de conjunto estruturado é o conjunto de dados de imagem. Corresponde a um conjunto de pontos igualmente espaçado, em uma, duas ou três dimensões. Ele pode ser representado implicitamente por uma origem, um espaçamento e uma dimensão dos dados. Seu uso mais comum são em imagem, mas pode também representar outros modelos.

Malhas retilíneas possuem espaçamento variável entre seus pontos. Esse conjunto possui topologia estruturada e geometria apenas parcialmente regular.

Semelhantemente, existem as malhas estruturadas, que possuem também topologia regular, mas geometria não estruturada. A geometria precisa ser representada

explicitamente, através das coordenadas dos vértices. Esse tipo de malha é bastante comum em algumas análises numéricas, como por exemplo, por diferenças finitas.

Os pontos não estruturados são um conjunto de pontos aleatórios. Úteis quando não há relação entre esses pontos, mas existe apenas um conjunto de atributos associados a eles.

Um tipo bastante comum de conjunto é o poligonal, que reúne vértices, linhas e polígonos. Embora esse conjunto seja facilmente representado por primitivas gráficas, ele é irregular.

O conjunto mais importante para MEF é a malha não estruturada. Esse é o conjunto mais genérico. Tanto a topologia quanto a geometria são completamente irregulares. Qualquer tipo de célula pode ser combinada nesse conjunto. Embora esse tipo de conjunto seja o que requer maior quantidade de memória e de processamento, pode representar facilmente qualquer modelo de elementos finitos, inclusive associando, por exemplo, tetraedros não-lineares com hexaedros lineares. Conseqüentemente, modelos bastante complexos podem ser representados.

De forma geral, a malha não estruturada (*vtkUnstructuredGrid*) é representada por três listas. A primeira lista contém as coordenadas dos pontos, enquanto a segunda contém a informação de quais pontos formam cada célula. A terceira lista possui informação sobre o tipo de célula, e conseqüentemente, o número de pontos por célula.

4.4 TÉCNICAS

Os algoritmos que transformam os dados constituem a parte principal da visualização de dados. Os algoritmos podem ser classificados de acordo com a estrutura, ou seja, efeitos sobre a geometria e a topologia dos objetos, e com o tipo de transformação, ou seja, tipos de dados sobre os quais os algoritmos operam e também tipos de dados que geram (SCHROEDER, MARTIN e LORENSEN, 2004).

As transformações estruturais podem ser divididas em quatro classes:

a) transformações geométricas, que alteram a geometria, mas não a topologia. Por exemplo, rotação, translação, escalamento;

b) transformações topológicas, que alteram a topologia, mas não a geometria. Um exemplo é converter dados poligonais em malha desestruturada;
c) transformações de atributos, que convertem atributos de dados de uma forma para outra ou criam atributos novos a partir dos dados de entrada. Como exemplo, calcular valores de vetores ou criar escalares baseados em elevações;
d) transformações combinadas, que alteram tanto a estrutura do conjunto de dados como os atributos dos dados. Um exemplo é o cálculo de linhas de contorno ou superfícies.

Os algoritmos podem ser também classificados de acordo com o tipo de dado no qual eles operam ou o tipo de dado que eles geram, ou seja, escalares, vetoriais ou tensoriais.

4.4.1 Escalares

Existem diversos algoritmos que operam com valores escalares, os quais são os mais comumente utilizados em visualização, principalmente por sua facilidade.

O mapeamento de cores é o primeiro deles, e a mais comum das técnicas de visualização científica. Para isso, é necessário indexar uma tabela de cores. Os valores escalares servem de índices na tabela. Essa tabela possui uma lista de cores (suas componentes RGB ou outras representações), e associado a ela existe um campo escalar, definido por um valor máximo e um mínimo. Esta técnica é geralmente a escolhida para visualizar distribuição de temperatura e componentes do tensor de tensão ou de deformação.

4.4.2 Vetoriais

Para visualizar dados vetoriais, é possível, com o VTK, utilizar algumas técnicas.

A primeira delas é a utilização de pequenas linhas, setas ou barras para cada

vetor. Essas linhas começam no ponto ao qual o vetor está associado e é orientado na direção e sentido dos componentes do vetor. Essas linhas dão noção da direção e da intensidade do campo vetorial, além de poderem ser coloridas de acordo com a magnitude, por exemplo. Pode-se, ainda, utilizar um glifo (por exemplo, cone ou triângulo) para esse mesmo fim.

Outra opção é o desenho do modelo deformado, para visualização de deslocamentos. Associada à animação, essa técnica produz resultados muito interessantes na compreensão de deformações. Além disso, pode-se associar mapeamento de cores com algum outro atributo.

Técnica bastante utilizada para análise de fluidos são as linhas de corrente, que consiste na representação da posição de uma ou mais partículas no tempo, originando algumas linhas. De forma mais sofisticada, pode representar até mesmo rotações no fluido, pelas fitas de corrente.

4.4.3 Tensoriais

A visualização direta de tensores é ainda um problema. O VTK fornece três opções para visualizar tensores 3x3 simétricos e compostos de números reais. As duas primeiras, semelhantes, são a representação dos eixos dos tensores e as elipsóides dos tensores. Ambas são, na verdade, glifos, que buscam representar o tensor em um ponto. É sempre possível criar glifos novos para essa representação. Entretanto, a maneira mais fácil ainda é visualizar os componentes dos tensores de forma separada mediante técnicas para escalares ou vetores.

Outra alternativa é o uso de hiper-linhas de corrente, criando uma linha de corrente através de uma das três direções do tensor, e então fazendo uma varredura com uma primitiva geométrica ao longo da linha.

4.4.4 Outras Técnicas

Além das técnicas de visualização científica, o VTK possui algoritmos para resolver alguns problemas.

Pode-se, por exemplo, inserir facilmente planos de corte em qualquer modelo. Esses planos são muito úteis para a análise de porções específicas do modelo.

É possível, também, extrair uma parte específica do conjunto analisado, baseado, por exemplo, em valores dos atributos.

A renderização de volume é uma ferramenta poderosa muito utilizada em diversos estudos. Existem três técnicas principais para renderização volumétrica implementadas no VTK: *ray casting*, mapeamento de textura 2D, e suporte para o hardware de renderização volumétrica *VolumePro*.

É possível, ainda, a visualização de imagens em três dimensões, pelo uso de óculos para este fim. Esta técnica é conhecida como renderização em estéreo. Ela faz uso de duas imagens próximas, e cada qual é vista apenas por um olho, dando a impressão de um objeto realmente tridimensional.

O VTK suporta também interação através de mouse e teclado, tendo implementado diversos recursos básicos, como zoom, arrasto, rotação, seleção de objeto, entre outros. É possível a criação dos próprios eventos interativos.

O VTK pode realizar até mesmo triangulações/tetraedralizações de Delaunay, em duas e três dimensões. Este tipo de triangulação é muito importante, principalmente na geração de malhas de elementos finitos. Embora haja esse recurso, ele não é muito eficiente. Uma opção recomendada é o *Qhull* (BARBER, DOBKIN e HUHDANPAA, 1996). Também disponível gratuitamente, este pacote robusto realiza a triangulação de Delaunay de forma eficiente em espaços até nove dimensões, além de determinar o fecho convexo de um conjunto de pontos e o diagrama de Voronoi (QHULL CODE, 2005). Entretanto, nem o VTK nem o *Qhull* fazem verificações quanto à qualidade da malha gerada, que é indispensável para caracterizar uma malha de elementos finitos.

O VTK pode, ainda, ser utilizado pela Internet. Ele possui dois modos de

visualização através da Web: usando VRML e Java.

VRML (*Virtual Reality Modeling Language*) (WEB3D CONSORTIUM, 2005) é um padrão de arquivo que representa vetores gráficos tridimensionais interativos. Existem diversos *browsers* que suportam esse formato e possibilitam o uso do hardware gráfico do cliente, se existente. Um arquivo do tipo VRML geralmente possui a extensão .wrl, e contém uma cena inteira. O VTK possui uma classe que exporta cenas para o formato VRML e, portanto, um arquivo desse tipo pode ser facilmente criado. Dessa forma, um cliente pode "navegar" pela cena criada. É possível oferecer mais interatividade ao cliente, para que ele escolha, por exemplo, o que ver na cena, cores e etc. Isso pode ser feito criando um servidor de visualização VRML. Por exemplo, em um formulário HTML podem ser definidos parâmetros para visualização, os quais são passados para o servidor. Com os parâmetros, o servidor pode gerar um novo arquivo VRML que pode ser visualizado pelo cliente.

A segunda opção para visualização na Web é a linguagem Java. Embora VRML produza resultados muito bons, Java pode proporcionar maior flexibilidade. Como Java é uma linguagem de programação completa, qualquer aplicativo de visualização escrito em Java funcionará em um sistema que suporte Java. Além disso, Java tem a flexibilidade de visualização no servidor ou no cliente.

Infelizmente, a biblioteca básica do Java não possui suporte direto para uso de hardware gráfico 3D, limitando alguns tipos de visualização. Para fazer uso desses hardwares, é necessário o uso de um *toolkit*, como Java3D ou VTK. O uso dessas bibliotecas sacrifica de alguma maneira a portabilidade do aplicativo, pois para um *toolkit* como o VTK, o cliente deve baixar o suporte nativo para VTK (biblioteca de objetos), antes de poder rodar uma aplicação VTK-Java. Uma vez isso feito, todas as funcionalidades do VTK podem ser incorporadas num código na linguagem Java, como por exemplo um *Applet*, e, dependendo das necessidades, os dados podem ser deixados no servidor e os resultados enviados ao cliente, ou os dados podem ser enviados ao cliente para processamento e visualização. Exemplo de trabalho utilizando Java e VTK pode ser visto em ALVES (2000).

4.5 MANIPULAÇÃO DE ARQUIVOS

O VTK apresenta funcionalidades para leitura, importação, gravação e exportação de arquivos. Ele possui um modelo próprio para o formato de seus arquivos, chamado arquivo VTK. Mas além de poder ler e gravar em arquivos VTK, é possível também fazer o intercâmbio com outros formatos (KITWARE, 2004).

Alguns dos formatos possíveis de leitura e gravação pelo VTK são: BYU, PDB, PLY, STL, XML, BMP, DICOM, GES, JPEG, PNM, PNG, SLC, TIFF, PLOT3D, POP, AVS, CGM e PNG.

Além disso, existem alguns formatos de arquivo que guardam informações de toda a cena, incluindo geometria, câmera, luzes, atores e propriedades. Alguns formatos suportados são: 3D *Studio*, VRML, *PostScript*, *Inventor*, *Wavefront* e *GeomView*.

4.5.1 Formato VTK

Quanto ao modelo próprio do VTK, existem dois formatos. O mais simples é o *legacy*, que consiste de uma seqüência de informações que são facilmente lidas e escritas, tanto manualmente quanto computacionalmente. O outro formato, de maior complexidade e, no entanto, mais flexível, é o baseado em XML, que permite acesso aleatório e paralelo.

O formato mais simples (*legacy*) consiste de cinco partes básicas:

a) versão do arquivo e identificação. Esta parte contem uma única linha: #vtk DataFile Version x.x. O x.x deve ser substituído pelo número da versão, que atualmente é 3.0;
b) cabeçalho, que consiste de um conjunto de no máximo 256 caracteres, terminando com '\n'. Ele pode ser usado para descrever os dados ou qualquer outra informação pertinente;
c) formato do arquivo, que pode ser ASCII ou binário. A linha deve ser ASCII ou BINARY, respectivamente;

d) estrutura do conjunto de dados. Corresponde à geometria e à topologia dos dados. Começa com uma linha contendo a palavra DATASET, seguida por outra que define seu tipo. Pode também conter outras palavras, dependendo do tipo de dados;

e) descrição dos atributos dos dados. Começa com a palavra POINT_DATA ou CELL_DATA, seguido de um número inteiro especificando o número de pontos ou células, respectivamente. As palavras seguintes definem os valores dos atributos, ou seja, escalares, vetores, normais e etc.

Embora arquivos do tipo binário sejam possíveis, podem ocorrer alguns problemas de portabilidade. Isso é devido a diferentes formas de representação em diferentes sistemas. Por exemplo, alguns sistemas usam um número diferente de bytes para representar um inteiro ou outro tipo nativo. O uso do formato ASCII pode resolver esse problema.

Os arquivos no formato VTK suportam cinco tipos de conjunto de dados: pontos estruturados, *grid* estruturado, *grid* retilinear, *grid* não estruturado e dados poligonais. Nos apêndices há um exemplo do tipo de arquivo VTK.

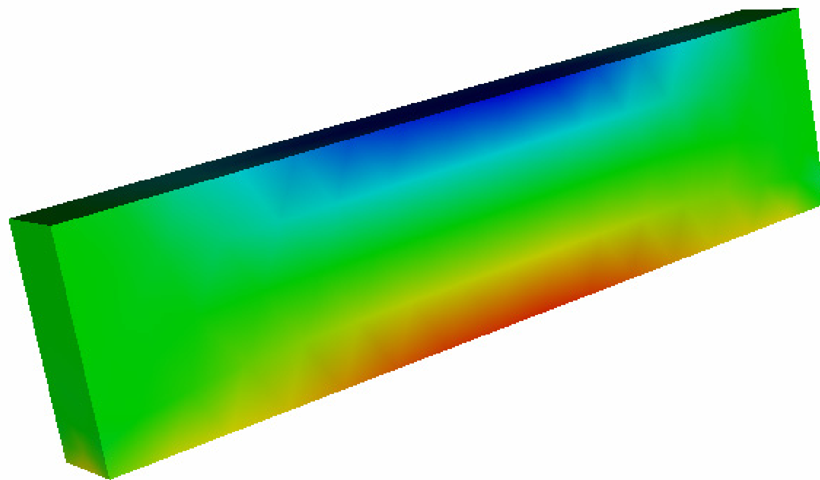
5 APLICAÇÕES DO VTK PARA VISUALIZAÇÃO DE RESULTADOS DO MÉTODO DOS ELEMENTOS FINITOS

Como experimentação das técnicas de SciVis, foram desenvolvidos alguns programas para visualização de resultados obtidos em análises pelo Método dos Elementos Finitos. Para implementação, a biblioteca VTK foi utilizada em modelos tridimensionais, e os dados foram obtidos com o aplicativo de análises pelo MEF, ANSYS versão 8.0 (ANSYS, 2005).

5.1 MAPEAMENTO DE CORES EM UMA ESTRUTURA

O primeiro algoritmo desenvolvido consiste no mapeamento de cores das tensões de uma estrutura. Os componentes são visualizados separadamente, nas direções X, Y e Z, como na Figura 23, que representa a tensão na direção X em uma viga de seção retangular.

FIGURA 23 – MAPEAMENTO DE CORES EM UMA VIGA



FONTE: O AUTOR

Inicialmente, por facilidade, a geometria e os elementos finitos são inseridos diretamente no código fonte, assim como os valores de tensão. Estes dados são listados

como simples arquivos de texto pelo aplicativo ANSYS, e incorporados como matrizes e vetores no código. A listagem das coordenadas dos pontos, que representam os nós dos elementos, é feita por uma matriz de valores de tipo ponto flutuante (*float*). Cada linha possui três valores, correspondendo às coordenadas X, Y e Z do ponto. Os valores dos componentes de tensão em cada direção são representados em vetores, sendo cada linha associada a cada nó. Já os elementos do tipo tetraedro são inseridos como uma matriz de quatro colunas. Cada linha indica a incidência de um elemento, associando-o com o índice de cada nó, da lista de pontos. Para esta última matriz, foi usada a classe *vtkIdType*, que nada mais é do que um tipo inteiro para identificação de pontos e células. Com essa representação, toda a geometria fica definida, de acordo com a malha de elementos e os resultados gerados pelo aplicativo ANSYS. Entretanto, esses dados precisam ser associados a uma única entidade para sua manipulação pelo VTK. Para isso, é utilizada a classe *vtkUnstructuredGrid*, um conjunto de dados que representa combinações arbitrárias dos tipos possíveis de células, ou seja, uma malha não estruturada. Um objeto dessa classe associa pontos com células e atributos como escalares. Então, as coordenadas dos pontos são agrupadas em uma lista de pontos tridimensionais, *vtkPoints*. Essa lista é associada à malha. Os escalares são inseridos em uma lista dinâmica de valores de ponto flutuante, *vtkFloatArray*, que é relacionada aos atributos dos pontos da malha. Os elementos são inseridos na malha, enumerando-se os nós que os definem, e o seu tipo, no caso tetraedro.

Depois, um *mapper* é criado. Ele associa os valores escalares a cores na geometria, e mapeia um conjunto de dados em primitivas gráficas, seguindo o modelo de visualização. A classe responsável por isso é a *vtkDataSetMapper*.

Para que o mapeamento seja feito como desejado, é necessário criar uma tabela de cores (*vtkLookupTable*), que relaciona valores com cores. Algumas propriedades dessa tabela precisam ser fornecidas, tais como a variação de valores, que pode ser a mesma dos atributos do conjunto de dados, variando do menor ao maior. Outro item importante é a variação de cores. Pode-se determinar as cores diretamente, ou de forma automática. Embora a tabela associe cores à especificação RGB, na escolha automática, é preciso fornecer variação de valores na especificação HSV. O

item mais importante é o matiz (*hue*), e o caso mais comum é associar o azul ao mínimo (*hue* 2/3) e o vermelho ao máximo (*hue* 0). Pode-se também definir o número de cores diferentes no intervalo definido.

Posteriormente, o *mapper* é associado a um ator (*vtkActor*), que é adicionado a um renderer (*vtkRenderer*). Ele controla o processo de renderização, criando uma imagem. Pode-se também adicionar uma câmera (*vtkCamera*) e luzes (*vtkLight*) ao renderer.

Finalmente, o renderer é adicionado a uma janela de renderização (*vtkRenderWindow*), que pode ser exibida pelo hardware gráfico. Além disso, pode-se criar um objeto para possibilitar a interação do usuário com a cena. A classe *vtkRenderWindowInteractor* possibilita esta interação, via teclado e mouse, com movimento de câmera, zoom, entre outras opções de eventos.

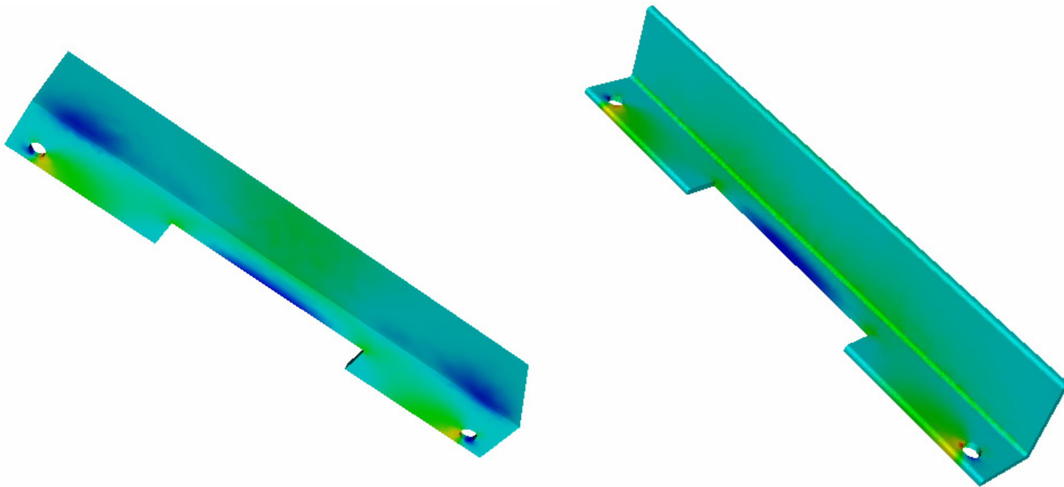
O código a seguir apresenta os principais itens abordados, e sua versão completa encontra-se nos apêndices:

```
//array com 181 pontos (x, y, z) representando os nós
static float coord[181][3];
//array com a incidência de 459 elementos do tipo tetraedro (4 nós cada)
static vtkIdType pts[459][4];
//array com os valores escalares (tensão) relacionados com os nós
static float strX[181];

vtkPoints *pontos = vtkPoints::New(); //lista de pontos
vtkFloatArray *escalares = vtkFloatArray::New();//lista de valores float

for (i=0; i<181; i++) {
  pontos->InsertPoint(i, x[i]); //insere cada ponto na lista
}

//malha não-estruturada
vtkUnstructuredGrid *malha = vtkUnstructuredGrid::New();
malha->SetPoints(points); //define os pontos da malha

for (i = 0;i <181; i++) {
  escalares->InsertValue(i, strX[i]);//define valores escalares
}

//insere os tetraedros na malha
for (i = 0;i <459; i++) malha->InsertNextCell(10, 4, pts[i]);

malha->GetPointData()->SetScalars(escalares);//define os valores escalares da malha

//mapper
vtkDataSetMapper *mapper = vtkDataSetMapper::New();
mapper->SetInput(malha);//define a malha não estruturada como dado de entrada
//tabela de cores
vtkLookupTable *stc = vtkLookupTable::New();
```

```
stc->SetRange(data->GetScalarRange());//valores máx. e mín. da tabela
stc->SetHueRange (.6667, .0);//define variação de cores da tabela
                              //(do azul ao vermelho)
stc->Build();//construção da tabela

mapper->SetLookupTable(stc); //define a tabela criada para o mapper
mapper->SetScalarRange(malha->GetScalarRange());//define a variação dos valores

//ator
vtkActor *ator = vtkActor::New();
ator->SetMapper(mapper);

//câmera
vtkCamera *camera = vtkCamera::New();
camera->SetPosition(1,1,1);
camera->SetFocalPoint(0,0,0);

// renderer
vtkRenderer *renderer = vtkRenderer::New();

// RendereWindow
vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin->AddRenderer(renderer);
renWin->SetSize(800,600); //tamanho da janela


// RenderWindowInteractor, com os eventos padrão
vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
iren->SetRenderWindow(renWin);

//adiciona o ator ao renderer e define outras propriedades
renderer->AddActor(ator);
renderer->SetActiveCamera(camera); //define a câmera
renderer->ResetCamera();
renderer->SetBackground(1,1,1); //define a cor de fundo (branco)

//renderiza a cena
renWin->Render();

iren->Start(); //inicia o interactor
```

A Figura 24 mostra duas vistas de uma peça, cuja tensão na direção Y é mapeada por cores. A geometria do modelo foi retirada dos exemplos do ANSYS.

FIGURA 24 – MAPEAMENTO DE CORES EM PEÇA



FONTE: O AUTOR

5.2 GERAÇÃO DE ARQUIVOS DO TIPO VTK

Uma solução para evitar a inserção das listas de dados do ANSYS diretamente no código fonte é a criação de um arquivo para armazenamento dos elementos, nós e atributos. O formato de arquivo VTK pode ser usado para esta finalidade, uma vez que pode armazenar todos estes campos.

Foi criada uma aplicação para conversão das listas de dados do ANSYS em arquivos do tipo VTK. Essas listas seguem um padrão, que incluem um cabeçalho e a listagem da informação de forma ordenada, linha por linha. Basta que se ignore o cabeçalho e que se armazene as informações pertinentes de cada linha em uma estrutura de dados simples. Para esta finalidade, foi criada uma estrutura de dados contendo listas duplamente encadeadas de nós, triângulos e tetraedros. Ela guarda todas as informações da malha e pode ser utilizada facilmente para a escrita do arquivo VTK. Nos apêndices há o código referente a esta estrutura.

Uma vez lidos os dados, eles devem ser escritos na seqüência adequada do formato VTK. Inicia-se com a identificação e o cabeçalho do arquivo, seguindo-se a listagem do conjunto de dados, finalizando-se com os valores dos atributos dos dados. O código para a leitura das listas do ANSYS e da gravação de arquivo do tipo VTK, além de exemplo de arquivo neste formato, é exposto nos apêndices.

Arquivos desse tipo, que incluem malha não estruturada, podem ser facilmente lidos e incorporados a uma aplicação com um objeto da classe *vtkUnstructuredGridReader*. E com uma malha do VTK já criada, pode-se gravar um arquivo com *vtkUnstructuredGridWriter*.

Uma aplicação para importação de outros tipos de arquivos pode ser facilmente realizada de maneira semelhante, desde que se conheça a formatação do arquivo para a leitura dos dados.

A leitura e gravação de um arquivo do tipo VTK contendo malha não estruturada pode ser feita como no trecho de código abaixo:

```
//malha não estruturada
vtkUnstructuredGrid *dados = vtkUnstructuredGrid::New();

//leitor de arquivo VTK
vtkUnstructuredGridReader *reader = vtkUnstructuredGridReader::New();
reader ->SetFileName("arquivo.vtk");
reader ->OpenVTKFile();
dados = reader->GetOutput();
reader ->CloseVTKFile();

//gravador de arquivo VTK
vtkUnstructuredGridWriter *writer = vtkUnstructuredGridWriter::New();
writer->SetFileTypeToASCII();
writer->SetInput(dados);
writer->SetFileName("teste.vtk");
writer->Write();
```

5.3 ISOSUPERFÍCIES

Utilizando os arquivos do tipo VTK gerados, foram testadas algumas outras técnicas. A primeira delas é a criação de isosuperfícies.

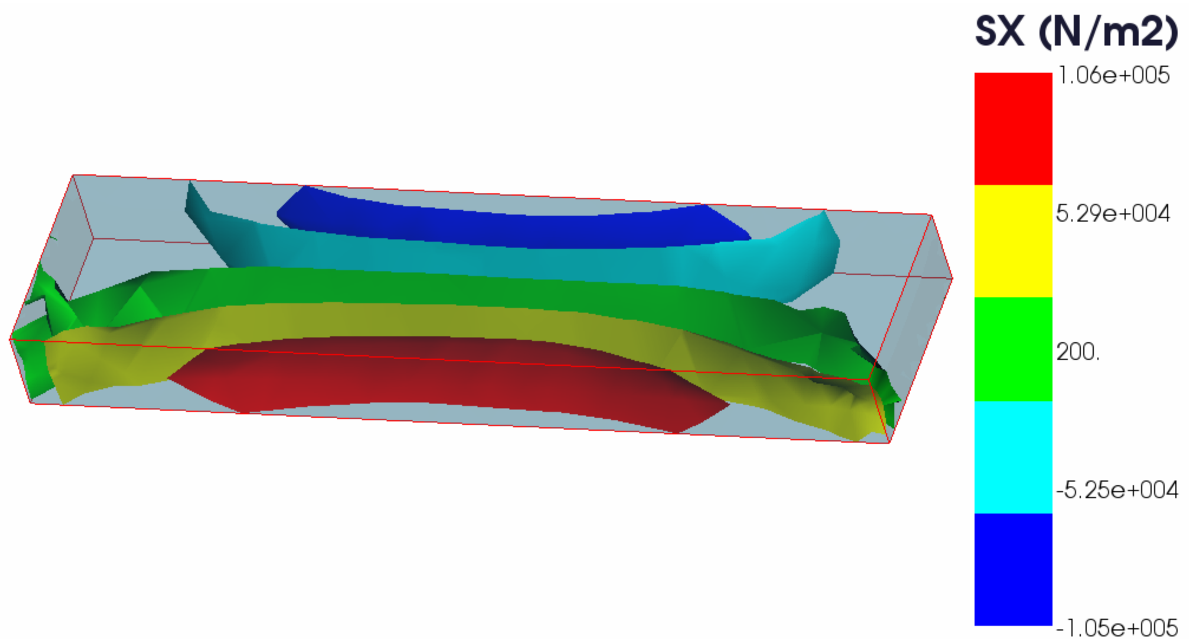
Para se realizar esta técnica, basta adicionar um filtro ao processo, o *vtkContourGrid*. Esta classe, especializada para malhas não estruturadas, gera isolinhas, no caso bidimensional, e isosuperfícies, no caso tridimensional, a partir de valores escalares. Para usar este filtro, é preciso especificar um ou mais valores de contorno. O método mais simples consiste em definir um número de isosuperfícies e um intervalo, e os valores são encontrados automaticamente. Cada isosuperfície é representada com a cor correspondente ao mapeamento do valor específico dela.

A utilização simples desta classe não apresenta bons resultados, pois as

isosuperfícies são extraídas do domínio sem nenhum suavização, dando um aspecto artificial. O uso de *vtkPolyDataNormals* pode resolver este problema, pois além de calcular as normais de uma malha poligonal, pode suavizá-la. A suavização é feita pelo cálculo da média das normais de polígonos adjacentes, sendo necessário definir um ângulo máximo entre essas normais. Outro filtro que pode ser usado em conjunto é o *vtkStripper*, que agrupa os triângulos gerados, facilitando a renderização.

A Figura 25 mostra uma viga com cinco isosuperfícies da tensão na direção X, geradas pelas classes mencionadas.

FIGURA 25 – ISOSUPERFÍCIES EM VIGA



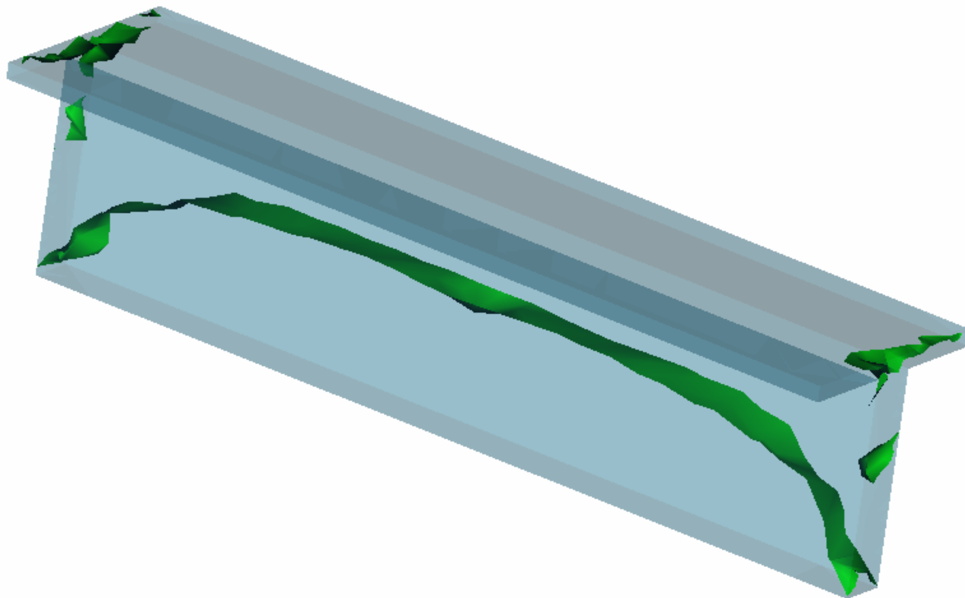
FONTE: O AUTOR

Outro recurso bastante útil é o uso de transparências. Elas permitem dar a idéia geral da geometria do modelo, enquanto as isosuperfícies são exibidas de maneira opaca. Esse recurso é possível pela adição de um *mapper* que contenha toda a geometria em um ator, cuja propriedade de opacidade é definida em um valor baixo, como por exemplo, 0,3. Embora nem sempre se mapeie cores em uma superfície da geometria do modelo semitransparente, isso pode ser útil com o uso de outra técnica, como isosuperfícies. Pode-se visualizar duas grandezas diferentes, ou ainda, reforçar uma única informação com duas técnicas.

Em todas as técnicas que utilizam cores é importante a adição de uma barra de cores como legenda. O VTK possui a classe *vtkScalarBarActor*, que cria uma destas barras com anotações. Uma barra de cor é uma legenda que indica a correspondência entre cor e valor. Esta legenda consiste de uma barra consistindo de pequenos retângulos, cada qual colorido por uma única cor. Este objeto é também um ator bidimensional, que pode ser adicionado sobre uma cena. Ele deve ser associado a uma tabela de cores e pode conter informações tais como título.

A Figura 26 usa as isosuperfícies para determinar os pontos em que a tensão na direção X é nula, mostrando a “linha” neutra na viga com seção “T”.

FIGURA 26 – SUPERFÍCIES COM TENSÃO NULA EM VIGA “T”



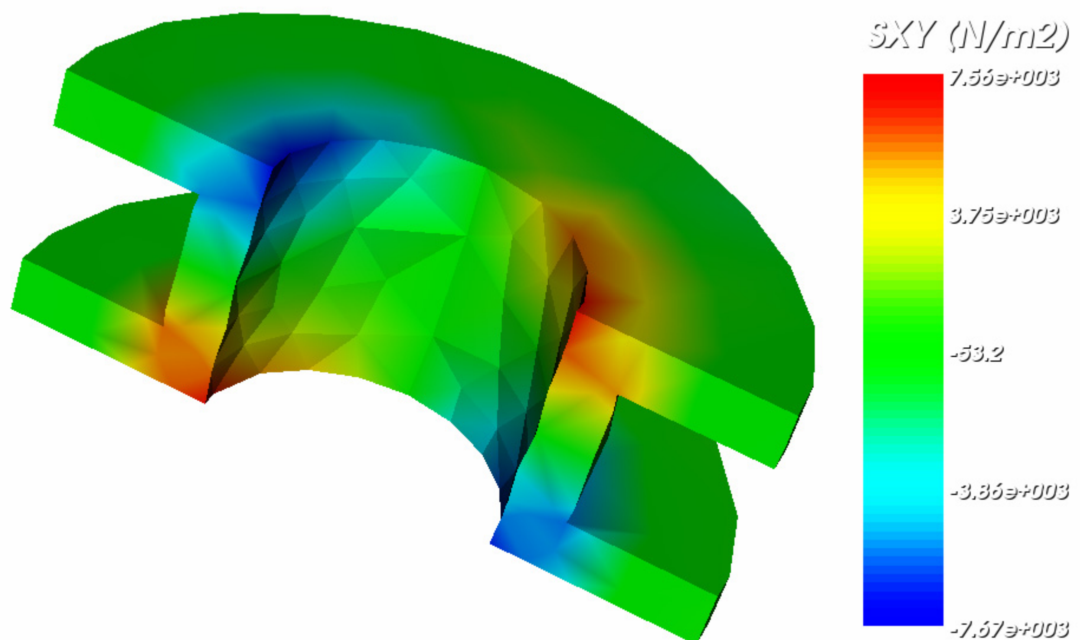
FONTE: O AUTOR

5.4 PLANOS DE CORTE, ANIMAÇÃO E EXTRAÇÃO DE FATIAS

Um plano de corte é bastante útil para se observar uma parte interna de um modelo. O VTK permite a divisão da geometria a partir de um ou mais planos, ou de forma genérica, a partir de uma função qualquer. Para isso, basta utilizar o filtro *vtkClipDataSet*. No caso de planos de corte, cria-se uma instância da classe *vtkPlane*,

que pode ser definida por um ponto de origem e uma direção normal. Este plano é associado ao filtro, que extrairá da geometria apenas o que está do lado para o qual “aponta” a normal do plano. Exemplo de plano de corte é mostrado na Figura 27. A geometria da figura representada foi retirada dos exemplos do ANSYS, e apenas meia peça é visualizada para que se possa observar a tensão de cisalhamento no plano XY em seu interior.

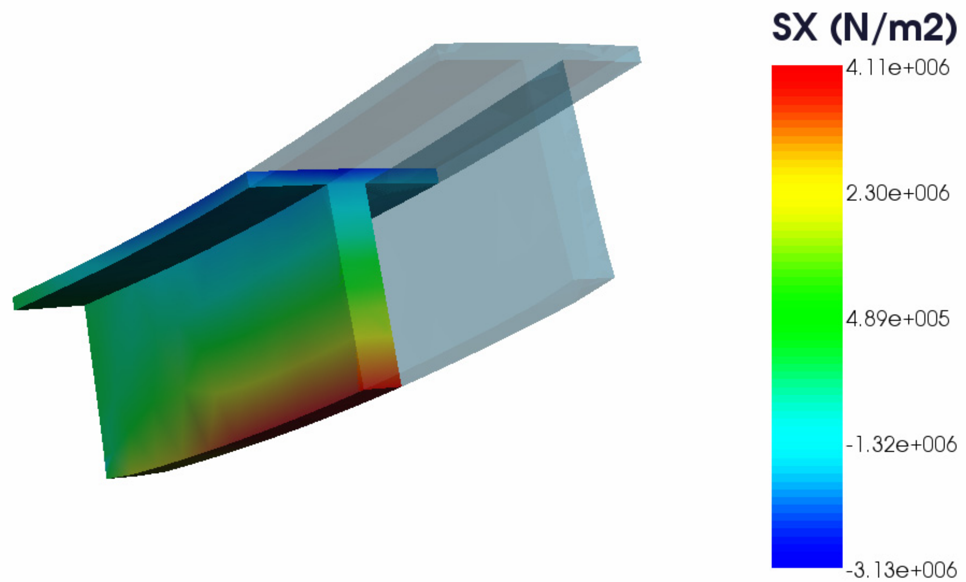
FIGURA 27 – PLANO DE CORTE



FONTE: O AUTOR

Aos planos de corte costuma-se também adicionar transparências, como na Figura 28, além de animações. Estas podem ser um recurso útil, pois fazer o plano de corte se deslocar sucessivamente em uma direção pode dar uma visão geral do modelo, inclusive de seu interior. Para realizar animações, deve-se variar a posição do plano de corte e atualizar o modelo em cada interação, dando a impressão de movimento. O espaçamento entre cada posição do plano deve ser pequena para que se evite saltos no corte, a menos que eles sejam necessários.

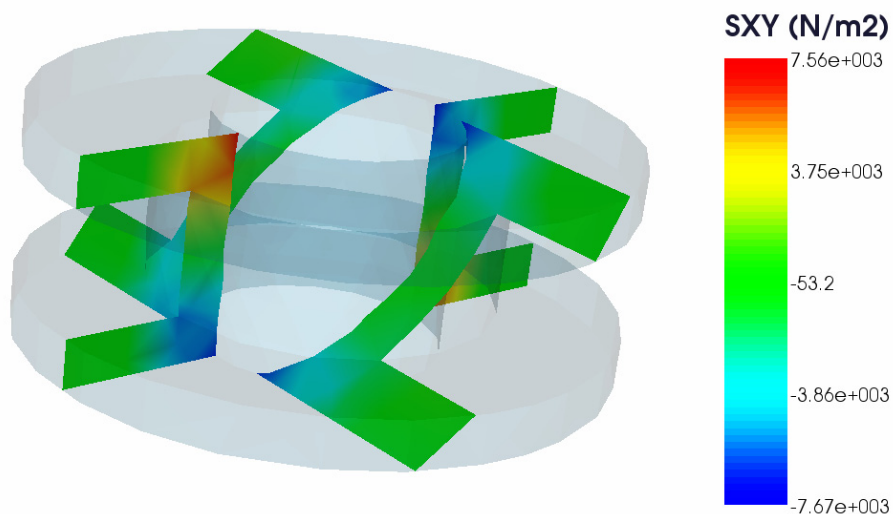
FIGURA 28 – DEFORMADA DE VIGA “T” COM TRANSPARÊNCIA E PLANO DE CORTE



FONTE: O AUTOR

As animações para representar deslocamentos são feitas de forma semelhante. Atualiza-se a renderização do modelo enquanto se varia a posição dos nós, de acordo com valores de deslocamento de coordenadas. No caso de pequenas variações, pode-se acrescentar um multiplicador, que fará com que os deslocamentos sejam mais facilmente perceptíveis.

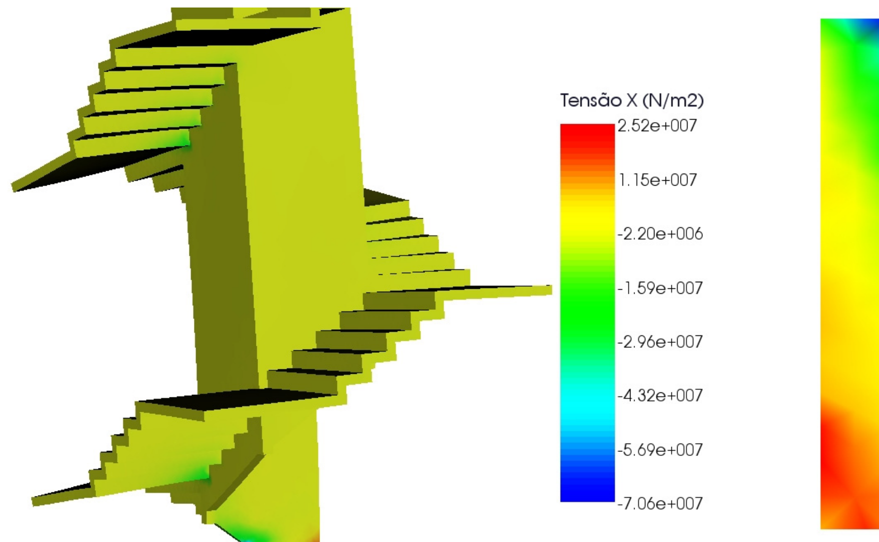
FIGURA 29 – EXTRAÇÃO DE FATIAS



FONTE: O AUTOR

Outra extração de geometria interessante é a de fatias, que pode ser feita com *vtkCutter*. Enquanto *vtkClipDataSet* extrai tudo que está de um lado de uma função, *vtkCutter* extrai tudo que não está na função. Por exemplo, definindo-se um plano como função, o resultado será também um plano. Isto facilita, muitas vezes, a análise de determinados planos importantes como imagens bidimensionais. A Figura 29 mostra duas fatias extraídas de uma estrutura mecânica e a Figura 30 mostra a distribuição de tensões em uma escada, e a seu lado apenas a fatia da base, facilmente obtida.

FIGURA 30 – DETALHE DA BASE DE ESCADA



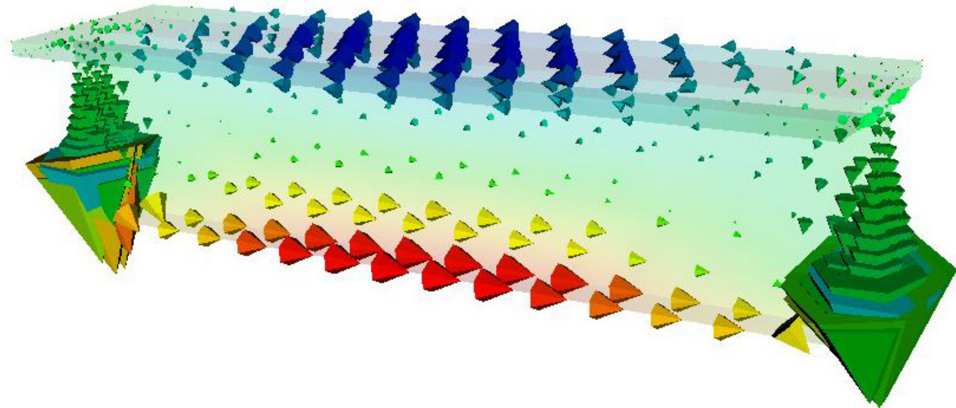
FONTE: O AUTOR

5.5 ÍCONES PONTUAIS

A criação de ícones pontuais para representação de dados vetoriais no VTK pode ser realizada com a utilização da classe *vtkGlyph3D*. É necessário criar um objeto que será o ícone representado. A opção mais simples é a utilização de cones, com *vtkConeSource*. Entretanto, pode-se criar outra geometria e associá-la ao glifo.

As Figuras 31 e 32 mostram a representação de entidades vetoriais (tensões principais) em uma viga de seção "T". Na primeira figura, as setas variam de tamanho de acordo com a magnitude do vetor. Isto causa problemas de escala, pois algumas setas ficam excessivamente grandes, enquanto outras ficam muito pequenas.

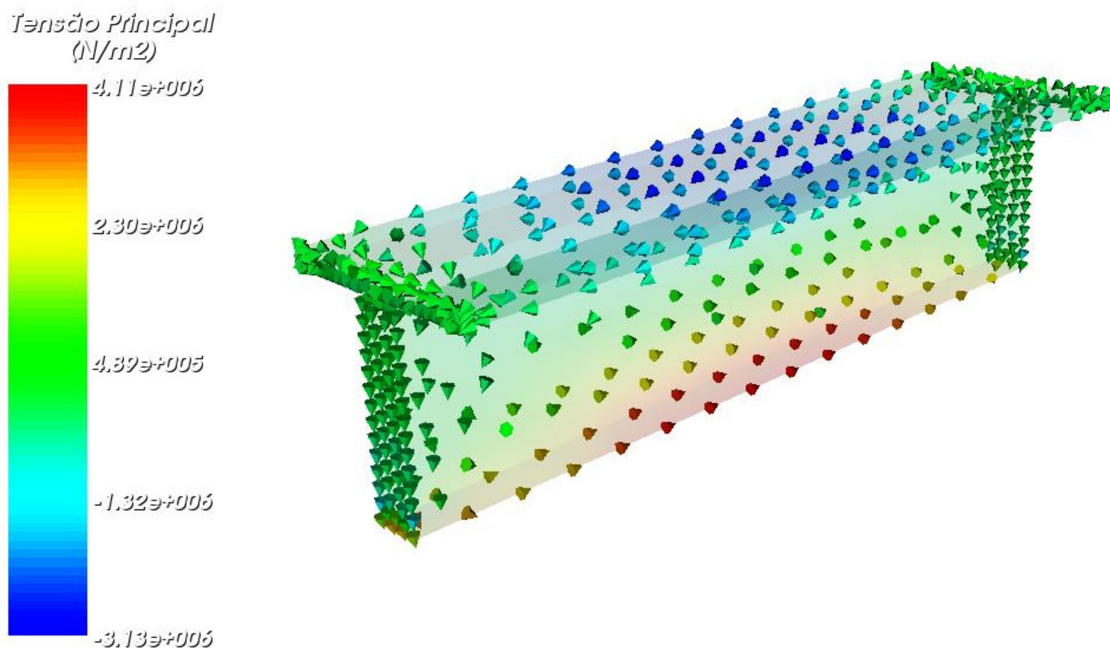
FIGURA 31 – ÍCONES PONTUAIS DE TAMANHO VARIÁVEL



FONTE: O AUTOR

Com a utilização de ícones de mesmo tamanho, não ocorre sobreposição das setas, mas perde-se um pouco a noção de sua magnitude. Em ambos os casos, ocorre certa confusão, principalmente nas áreas mais densas, pois não se tem a noção exata da posição de todos os vetores, devido à projeção tridimensional em uma simples imagem.

FIGURA 32 – ÍCONES PONTUAIS DE TAMANHO CONSTANTE



FONTE: O AUTOR

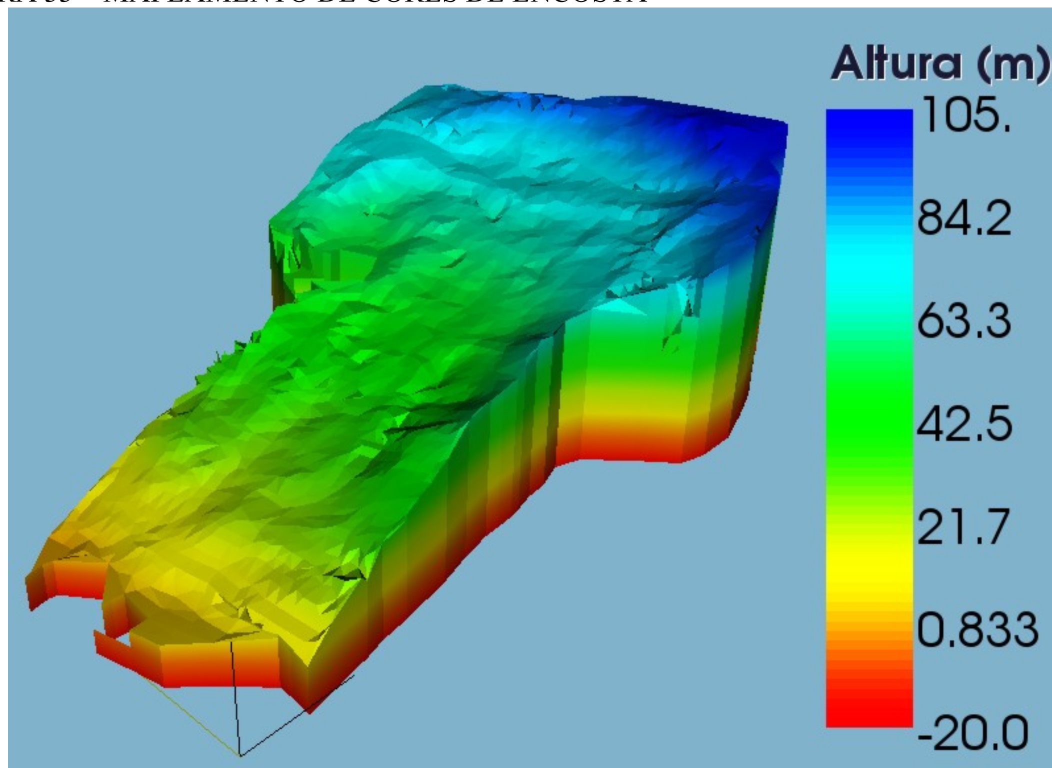
5.6 GEOMETRIA DE ENCOSTA

Para se experimentar as técnicas em geometria complexa, elas foram aplicadas em dados referentes a uma encosta real.

Foram fornecidos os pontos tridimensionais das curvas de nível de um terreno. Estes pontos foram triangulados, pelo método de Delaunay, após uma limpeza de pontos próximos entre si. Embora o VTK possua métodos para este tipo de triangulação, ela não se mostrou muito eficiente. Portanto, foi utilizado o Qhull (QHULL CODE, 2005), que realizou a triangulação dos pontos com velocidade bastante aceitável, e com qualidade.

A partir da superfície gerada, foram criados pontos abaixo dela e, posteriormente, realizada uma tetraedralização de todos estes pontos. Os tetraedros acima da superfície foram removidos por meio de algoritmos de interseção de raio e triângulo, muito utilizadas em *ray tracing*, criando um modelo 3D representativo da encosta.

FIGURA 33 – MAPEAMENTO DE CORES DE ENCOSTA

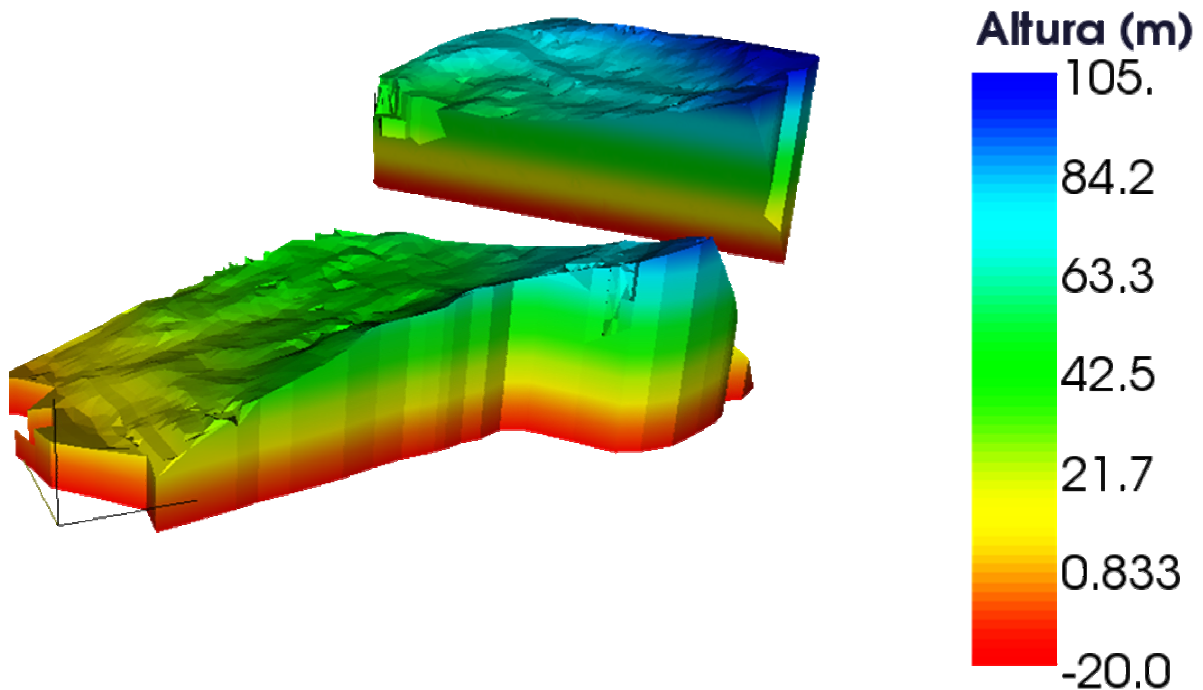


FONTE: O AUTOR

O resultado foi representado com mapeamento das cores de acordo com sua altura, como mostrado na Figura 33. Planos de corte, mostrada na Figura 34, animação, transparência e extração de seção, exemplificada na Figura 35, foram também utilizados.

Embora o resultado tenha sido bom, a malha gerada pela tetraedralização não é uma malha de elementos finitos. Para que ela se torne uma, é necessária uma avaliação de sua qualidade, melhoria e refinamento, se necessário. Entretanto, sua representação no VTK seria idêntica.

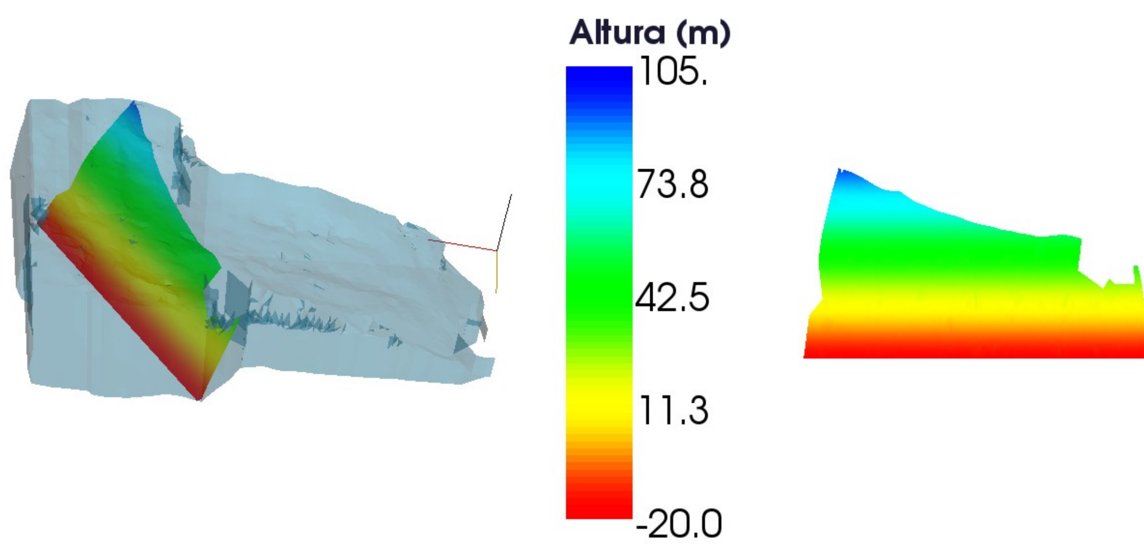
FIGURA 34 – CORTE DE ENCOSTA



FONTE: O AUTOR

Todos os programas foram desenvolvidos com o Microsoft Visual Studio .NET, na linguagem de programação C++ e com o VTK. Nos apêndices encontram-se os códigos-fonte destes programas.

FIGURA 35 – EXTRAÇÃO DE SEÇÃO DE ENCOSTA



FONTE: O AUTOR

6 CONCLUSÕES E CONSIDERAÇÕES FINAIS

As aplicações desenvolvidas demonstram que é possível, com a utilização de um pacote de Visualização Cientifica, a implementação de técnicas eficientes de representação de dados numéricos para auxílio dos analistas que usam o Método dos Elementos Finitos. Além disso, essas técnicas possuem potencial para serem utilizadas em outros tipos de análises numéricas, como por exemplo, os métodos dos Volumes Finitos, das Diferenças Finitas e dos Elementos de Contorno.

Embora os algoritmos utilizados sejam eficientes para visualização de dados escalares, necessita-se de uma boa técnica para a representação de vetores e tensores em três dimensões, pois as técnicas utilizadas normalmente não são capazes de representá-los sem produzir ambigüidades e deixar de mostrar pontos relevantes. Estes tipos de dados podem ser facilmente analisados através de seus componentes escalares, mas perde-se sua noção geral. Vetores e tensores são dados muito importantes em análises pelo MEF, principalmente em estruturas.

O pacote utilizado nas aplicações, o *Visualization Toolkit*, demonstrou ser uma boa biblioteca para se utilizar em um aplicativo de pós-processamento nas análises pelo MEF. Além de ser gratuito, é independente de sistema operacional e está disponível em diversas linguagens de programação. Embora ainda sejam necessárias mais implementações, os resultados foram representados de forma satisfatória, e podem ser reproduzidos de maneira simples.

6.1 SUGESTÕES DE TRABALHOS FUTUROS

Neste trabalho, foram utilizadas algumas técnicas de SciVis de forma independente, como estudo das mesmas. É importante que seja desenvolvido um aplicativo completo de pós-processamento, que reúna essas diversas técnicas, possua grande interatividade e facilidade de importação e exportação de dados. Um aplicativo desse tipo seria de grande utilidade para analistas que utilizam métodos numéricos, em especial para auxílio ao ensino e aprendizagem.

Seria também interessante o estudo mais aprofundado do VTK e de outras bibliotecas de visualização, para posterior comparação. Dessa maneira, a melhor delas pode ser utilizada de forma ainda mais completa e efetiva.

Um ponto fundamental em aplicações que utilizam Computação Gráfica e Métodos Numéricos é a eficiência, tanto de processamento quanto de armazenamento de dados. Neste trabalho, o objetivo não foi buscar desempenho, a intenção foi avaliar a possibilidade da utilização de uma biblioteca de visualização e suas técnicas. Um estudo mais aprofundado sobre eficiência é necessário para que resultados melhores sejam alcançados.

Não foi estudada a possibilidade da utilização da imersão em realidade virtual em pós-processamento do Método dos Elementos Finitos, mas recursos sofisticados como esse podem representar avanços no campo da visualização, inclusive de dados vetoriais e tensoriais.

Outro ponto importante não explorado neste trabalho é a SciVis pela Internet, que possibilita o fácil acesso a aplicações de qualquer parte do planeta.

REFERÊNCIAS

ALVES, A. D. et al. Interactive Visualization over the WWW. In: BRAZILIAN SYMPOSIUM ON COMPUTER GRAPHICS AND IMAGE PROCESSING (SIBGRAPI), 13., 2000. **Anais**… Gramado, 2000.

ANSYS. Welcome to ANSYS, Inc. – Corporate Homepage. Disponível em: < http://www.ansys.com>. Acesso em: 11 fev. 2005.

AVS. AVS: Advanced Visual Systems: data visualization. Disponível em: <http://www.avs.com>. Acesso em: 03 jun. 2005.

BARBER, C. B.; DOBKIN, D. P.; HUHDANPAA, H. The Quickhull algorithm for convex hulls. ***ACM Trans. on Mathematical Software***, v. 22, n. 4, p.469-483, 1996.

BENZLEY, S. E. et al. Pre- and post-processing for the finite element method. **Finite Elements in Analysis and Design**, v. 19, p. 243-260. Elsevier Science, 1995.

BERKLEY, J. et al. Real-time finite element modeling for surgery simulation: an application to virtual suturing. **IEEE Transactions on Visualization and Computer Graphics**, v. 10, p. 314-325. IEEE, 2004.

BERTON, J. A. Strategies for scientific visualization: analysis and comparison of current techniques. In: SPIE, 1990, Santa Clara, **Anais**... Bellingham: SPIE, 1990, p. 110-121.

BRODLIE, K. et al. (Eds.) **Scientific Visualization, techniques and applications**. Springer-Verlag, 1992.

BRODLIE, K. A classification scheme for scientific visualization. In: EARNSHAW, R. A.; WATSON, D. (Eds.) **Animation and Scientific Visualization:** tools & applications. Academic Press, 1993. p. 125-140.

BRODLIE, K. Scientific Visualization: past, present and future. **Nuclear Instruments & Methods in Physics Research**, p. 104-111. Elsevier Science, 1995.

CAMPOS, J. A. P. **Geração de malhas de elementos finitos bidimensionais baseada em uma estrutura de dados topológica.** Dissertação (Mestrado em Engenharia Civil) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 1991.

CAVALCANTE NETO, J. B. **Geração de malha e estimativa de erro para modelos tridimensionais de elementos finitos com trincas.** Dissertação (Mestrado em Engenharia Civil) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 1998.

CENTER FOR APPLIED COMPUTER SCIENCE, UNIVERSITY OF COLOGNE. Vis5D. Disponível em: <http://www.uni-koeln.de/rrzk/software/grafik/visualization/vis5d/zaik.html>. Acesso em: 11 set. 2005.

CFD NORWAY. Rapport: wind simulations on acropolis. Disponível em: <http://www.cfdnorway.no/projects/acropolis/CFDn_223_1999.htm>. Acesso em: 11 set. 2005.

COLLINS, B. M. Data visualization: has it all been seen before? In: EARNSHAW, R. A.; WATSON, D. (Eds.) **Animation and Scientific Visualization:** tools & applications. Academic Press, 1993. p. 3-28.

COOPER, R. I.; FOXMAN, B. M.; YANG, L. CIF applications. XVI. CIF2CRY: for CIF input into the CRYSTALS program. **Applied Crystallography**, v. 37, p. 669-671. International Union of Crystallography, 2004.

DUNCAN, B. S.; MACKE, T. J.; OLSON, A. J. Biomolecular visualization using AVS. **Journal of Molecular Science**, New York: Elsevier Science, 1995.

ENCARNAÇÃO, J. L. Advanced research and development topics in animation and scientific visualization. In: EARNSHAW, R. A.; WATSON, D. (Eds.) **Animation and Scientific Visualization:** tools & applications. Academic Press, 1993. p. 37-73.

FASTOOK, J. L. Modeling the Ice Age: the finite element method in glaciology. **IEEE Computational Science & Engineering**, p. 55-67. IEEE, 1994.

FOLEY, J. D. et al. **Computer graphics**: principles and practice, 2nd. ed. Reading: Addison Wesley, 1990.

FRAUNHOFER IZM. Gradian. Disponível em: <http://www.pb.izm.fhg.de/ase/010_research_fields/040_kmsd/gradian,IMG_DEFAULT.html>. Acesso em: 11 set. 2005.

GATASS, M.; CELES FILHO, W.; FONSECA, G. L. Computação gráfica aplicada ao Método dos Elementos Finitos. CONGRESSO NACIONAL DE MATEMÁTICA APLICADA E COMPUTACIONAL (CNMAC), 14., Nova Friburgo. **Minicurso do CNMAC**. Nova Friburgo, SBMAC, 1991.

GELLER, M. J.; FALCO, E. E. Graphic voyages through the Universe. **IEEE Computer Graphics and Applications**, p. 7-11. IEEE, 1994.

GERSHON N. Advances and Challenges. In: ROSENBLUM, L. el al. (Ed.) **Scientific Visualization:** from perception to visualization. Academic Press, 1994.

GLOBUS, A.; RAIBLE, E. Fourteen ways to say nothing with Scientifc Visualization. **Computer**, p. 86-88. IEEE, 1994.

HABER, R. B.; MCNABB, D. A. Visualization idioms: a conceptual model for scientific visualization systems. In: NIELSON, G. M.; SHRIVER, B. D.; ROSENBLUM, L. J. (Eds.), **Visualization in Scientific Computing**. Los Alamos: IEEE, 1990, p. 74-93.

HSV color space - Wikipedia, the free encyclopedia. Disponível em: <http://en.wikipedia.org/wiki/HSV_color_space>. Acesso em: 16 ago. 2005.

IBM. IBM Research Visualization Data Explorer. Disponível em: <http://www.research.ibm.com/dx>. Acesso em: 02 fev. 2005.

KITWARE. **The VTK user's guide**: install, use and extend The Visualization Toolkit. 2004.

KITWARE. CMake: cross platform make. Disponível em: <www.cmake.org>. Acesso em: 10 jul. 2005.

KITWARE. VTK home page. Disponível em: <http://www.vtk.org>. Acesso em: 19 jan. 2005.

KUSCHFELDT, S. et al. Efficient visualization of crash-worthiness simulations. **IEEE Computer Graphics and Applications**, p. 60-65. IEEE, 1998.

LIRA, W. W. M. **Modelagem geométrica para elementos finitos usando multi-regiões e superfícies paramétricas.** Tese (Doutorado Engenharia Civil) Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2002.

MÄNTYLÄ, M. **An Introduction to Solid Modeling**. Rockville: Computer Science Press, 1988.

MARTIN, K.; HOFFMAN, B. **Mastering CMake:** a cross-platform build system. Kitware, Inc., 2004

MCCORMICK, B. H.; DEFANTI, T. A.; BROWN, M. D. Visualization in Scientific Computing. **Computer Graphics** (edição especial), v. 21, n. 6, nov. 1987.

MENDONÇA, M. de B. **Aplicação de texturas em Visualização Científica**. 91 f. Dissertação (Mestrado Ciências da Computação e Matemática Computacional) – Universidade de São Paulo, São Carlos, 2001.

NIKISHKOV, G. P. Generating contours on FEM/BEM higher-order surfaces using Java 3D textures. **Advances in Engineering Software**. v. 34, p. 469–476. Elsevier Science, 2003.

NUMERICAL ALGORITHMS GROUP. IRIS explorer center. Disponível em: <http://www.nag.co.uk>. Acesso em: 02 jun. 2005.

OLIVEIRA, M. C. F. de; MINGHIM, R. Uma introdução à Visualização Computacional. In: JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA (JAI), 16., 1997, Brasília. **Anais**... Brasília: SBC, 1997. p. 85-131.

OPENGL. OpenGL – the industry standard for high performance graphics. Disponível em: <http://www.opengl.org>. Acesso em: 11 set. 2005.

QHULL code for convex hull, delaunay triangulation, voronoi diagram, and halfspace intersection. Disponível em: <http://www.qhull.org>. Acesso em: 08 abr. 2005.

QUARESMA, J. E.; OLIVEIRA, J. N.; WENDLAND, E. C. VPA: um visualizador de processos em aqüíferos. In: IBERIAN LATIN-AMERICAN CONGRESS ON COMPUTATIONAL METHODS IN ENGINEERING (CILAMCE), 14., 2003, Ouro Preto. **Anais**...

ROSENBLUM, L. J. Research issues in Scientific Visualization. **IEEE Computer Graphics and Applications**. IEEE, 1994.

SCHROEDER, W. J.; MARTIN, K. M.; LORENSEN, W. E. The design and implementation of an object-oriented toolkit for 3D graphics and visualization. In: IEEE VISUALIZATION, 7., 1996, Los Alamos. **Anais**... Los Alamos, IEEE, 1996.

SCHROEDER W. J.; AVILA, L. S.; HOFFMAN, W. Visualizing with VTK: a tutorial. **IEEE Computer Graphics and Applications**. IEEE, 2000.

SCHROEDER, W.; MARTIN, K.; LORENSEN, B. **The Visualization Toolkit**: an object-oriented approach to 3D graphics. 3rd. ed. Kitware, Inc, 2004.

SETRED. Disponível em: <http://www.setred.com/technology>. Acesso em: 12 jun. 2005.

SILVA, E. C. Using Data Explorer to visualize data generated by ANSYS. In: IEEE VISUALIZATION, 7., 1996, San Francisco. **Anais**…

SIMONETTI, D. G. **Interactive visualization of biomedical data**. 58 f. Dissertação (Mestrado em Ciência da Computação) - Universiteit van Amsterdam, Amsterdam, 2003.

SOUZA, C. A. S. de. **Implementação de uma estrutura de dados para visualização científica**. 92 f. (Mestrado Ciências da Computação e Matemática Computacional) – Universidade de São Paulo, São Carlos, 2003.

SVAKHINE, N.; EBERT, D. S.; STREDNEY, D. Illustration Motifs for effective medical volume illustration. **IEEE Computer Graphics and Applications**, p. 31-39. IEEE, 2005.

TÄGNFORS, H. Visualization of FE-data, interactively and on video. **Advances in Engineering Software**, v. 30, p. 753–762. Elsevier Science Ltd e Civil-Comp Ltd, 1999.

TECGRAF. Tecgraf – Tecnologia em Computação Gráfica. Disponível em: < http://www.tecgraf.puc-rio.br>. Acesso em: 02 set. 2005.

TREINISH, L. A. Unifying principles of data management for scientific visualization. In: EARNSHAW, R. A.; WATSON, D. (Eds.) **Animation and Scientific Visualization:** tools & applications. Academic Press, 1993. p. 141-170.

UNIVERSITY OF CAMBRIDGE. Department of Physics. Cambridgephysics.com. Disponível em: <http://www.cambridgephysics.com> Acesso em: 26 jun. 2005.

UPSON, C. et al. The Advanced Visualization System: a computational environment for Scientific Visualization. **IEEE Computer Graphics & Applications**. IEEE, 1989.

VISAD Home Page. Disponível em: <http://www.ssec.wisc.edu/~billh/visad.html>. Acesso em: 12 jun. 2005.

VISUALISIERUNG UND ANALYSE TURBULENTER STRÖMUNGEN. SFB-Bunch. Disponível em: <http://www.lstm.uni-erlangen.de/SFB603_C3/sfbbook. Acesso em: 11 set. 2005.

WALTON, J. Get the picture: new directions in data visualization. In: EARNSHAW, R. A.; WATSON, D. (Eds.) **Animation and Scientific Visualization:** tools & applications. Academic Press, 1993. p. 29-36.

WEB3D CONSORTIUM. Web3D Consortium - Open Standards for Real-Time 3D Communication. Disponível em: < http://www.web3d.org>. Acesso em: 11 set. 2005.

WENGER, A. et al., Interactive volume rendering of thin thread structures within multivalued scientific data sets. **IEEE Transactions on Visualization and Computer Graphic**s, v. 10, n. 6, p. 664-672. IEEE, 2004.

WILLIAMS, P. L.; MAX, N. L.; STEIN, C. M. A high accuracy volume renderer for unstructured data. **IEEE Transactions on Visualization and Computer Graphics**, v. 4, n. 1, p. 37-54. IEEE, 1998.

WU, J.; WU, G.; YANG, P. A fast algorithm for visualizaing 3D unstructured grid data on outer surfaces. **Advances in Engineering Software**, v. 33, p. 291-295. Elsevier, 2002.

ZANDONÁ, C. A. W. **Ambiente de visualização integrado para modelos numéricos de previsão de tempo e informações ambientais**. 67 f. Dissertação (Mestrado em Métodos Numéricos em Engenharia) - Universidade Federal do Paraná, Curitiba, 2004.

APÊNDICES

APÊNDICE 1 – ALGORITMO BÁSICO PARA VISUALIZAÇÃO COM O VTK.......93

APÊNDICE 2 – EXEMPLO DE ARQUIVO DO TIPO VTK (EXTRATO)...................96

APÊNDICE 3 – APLICAÇÃO DE MAPEAMENTO DE CORES.................................99

APÊNDICE 4 – CRIAÇÃO DE ARQUIVOS DO TIPO VTK 103

APÊNDICE 5 – ALGORITMO PARA CRIAÇÃO DE ISOSUPERFÍCIES................111

APÊNDICE 6 – ALGORITMO COM PLANOS DE CORTE, ANIMAÇÃO E EXTRAÇÃO DE FATIAS 115

APÊNDICE 7 – ALGORITMO DE ÍCONES PONTUAIS 119

APÊNDICE 1 – ALGORITMO BÁSICO PARA VISUALIZAÇÃO COM O VTK

```
/*==========================================================================
exemploVTK.cxx
Exemplo de algoritmo básico de visualização utilizando o VTK. Desenha uma
esfera e um cone em uma janela padrão.
==========================================================================*/

#include "vtkSphereSource.h"
#include "vtkConeSource.h"
#include "vtkPolyDataMapper.h"
#include "vtkActor.h"
#include "vtkRenderWindow.h"
#include "vtkRenderer.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkProperty.h"
#include "vtkCamera.h"
#include "vtkLight.h"

void main(){

  //geometria da esfera
  vtkSphereSource *esfera = vtkSphereSource::New();
  esfera->SetRadius(1.0);
  esfera->SetThetaResolution(32);
  esfera->SetPhiResolution(32);

  //geometria do cone
  vtkConeSource *cone = vtkConeSource::New();
  cone->SetCenter(1, -2, -1);
  cone->SetHeight(2);
  cone->SetResolution(8);

  //mapper da esfera
  vtkPolyDataMapper *mapperEsfera = vtkPolyDataMapper::New();
  mapperEsfera->SetInput(esfera->GetOutput());

  //mapper do cone
  vtkPolyDataMapper *mapperCone = vtkPolyDataMapper::New();
  mapperCone->SetInput(cone->GetOutput());

  //ator da esfera
  vtkActor *atorEsfera = vtkActor::New();
  atorEsfera->SetMapper(mapperEsfera);
  atorEsfera->GetProperty()->SetColor(0,0,1);

  //ator do cone
  vtkActor *atorCone = vtkActor::New();
  atorCone->SetMapper(mapperCone);
  atorCone->GetProperty()->SetColor(0,1,0);

  //renderer e janela de renderização
  vtkRenderer *renderer = vtkRenderer::New();
  vtkRenderWindow *renWin = vtkRenderWindow::New();
  renWin->AddRenderer(renderer);

  //câmera
  vtkCamera *camera = vtkCamera::New();
  camera->SetPosition(-8.0, -8.0, 8.0);
  camera->SetViewUp(.0, .0, 1.0);
  camera->SetFocalPoint(.0,-1.0,.0);
```

```
  camera->Zoom(1.8);

  //iluminação
  vtkLight *iluminacao1 = vtkLight::New();
  iluminacao1->SetColor(1.0, .6, .7);
  iluminacao1->SetPosition(2.0, 1.0, 1.0);
  iluminacao1->SetFocalPoint(.0, .0, .0);
  vtkLight *iluminacao2 = vtkLight::New();
  iluminacao2->SetColor(1.0, 1.0, 1.0);
  iluminacao2->SetPosition(-2.0, -1.0, 1.0);

  //interação
  vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
  iren->SetRenderWindow(renWin);

  //adição dos atores, câmera e luzes à cena
  renderer->AddActor(atorEsfera);
  renderer->AddActor(atorCone);
  renderer->SetBackground(1,1,1);
  renderer->SetActiveCamera(camera);
  renderer->AddLight(iluminacao1);
  renderer->AddLight(iluminacao2);

  //renderização da cena
  renWin->Render();

  //inicio da interação
  iren->Start();
}
```

APÊNDICE 2 – EXEMPLO DE ARQUIVO DO TIPO VTK (EXTRATO)

```
# vtk DataFile Version 3.0
Arquivo de Exemplo
ASCII

DATASET UNSTRUCTURED_GRID
POINTS 564 float
0 93 20
0 0 20
0 86.3571 20
0 79.7143 20
0 73.0714 20
0 66.4286 20
0 59.7857 20
0 53.1429 20
0 26.5714 20
0 19.9286 20
0 13.2857 20
0 6.64286 20
0 93 0
0 93 13.3333
0 93 6.66667
(...)

CELLS 1396 6980
4 234 190 464 137
4 184 277 183 206
4 277 183 206 551
4 551 286 276 275
4 551 205 286 275
4 551 275 182 205
4 552 274 203 181
4 552 180 181 203
4 526 399 549 419
4 273 151 218 552
4 219 273 151 218
4 552 274 181 150
4 81 156 112 553
4 132 266 470 229
4 553 83 554 98
(...)

CELL_TYPES 1396
10
10
10
10
10
10
10
10
10
10
10
10
10
10
10
(...)

POINT_DATA 564
SCALARS escalares float
LOOKUP_TABLE default
-222580
1.5484e+006
-64374
-95618
```

```
-44542
-59880
-21722
1512.6
-38132
-115630
-312730
-397040
-187660
-283700
-1.2382e+006
(...)

VECTORS vetores float
-222580 -291990 -826730
1.5484e+006 -1.2471e+007 554670
-64374 -452150 -128190
-95618 -848430 -23735
-44542 -1.3499e+006 9143.5
-59880 -1.9133e+006 -39294
-21722 -2.4497e+006 -17481
1512.6 -3.0835e+006 448.85
-38132 -3.6682e+006 -43745
-115630 -4.4888e+006 -179530
-312730 -5.2982e+006 -174930
-397040 -7.0363e+006 -275460
-187660 -7.1284e+006 -525680
-283700 -9.8113e+006 -716230
-1.2382e+006 -1.2803e+007 -621760
(...)
```

APÊNDICE 3 – APLICAÇÃO DE MAPEAMENTO DE CORES

```
/*=========================================================================
viga.cxx
Desenha uma viga em função de seus nós e elementos. Representa as tensões
na direção do eixo X por mapeamento de cores.
Os dados utilizados foram retirados de uma análise pelo Método dos
Elementos Finitos utilizando o aplicativo Ansys 8.0.
=========================================================================*/

//adição dos cabeçalhos das classes utilizadas
#include "vtkActor.h"
#include "vtkCamera.h"
#include "vtkFloatArray.h"
#include "vtkPoints.h"
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkRenderer.h"
#include "vtkUnstructuredGrid.h"
#include "vtkDataSetMapper.h"
#include "vtkLookupTable.h"
#include "vtkLogLookupTable.h"

int main(int argc, char *argv[]){

   int i;

   //array com os 181 pontos (x, y, z) representados os nós
   static float coord[181][3]={
        {0, 0.6, 0},
        {0, 0, 0},
        {0, 0.4, 0},
        ...
        {1.999935269, 0.299935222, 0.299935222},
        {2.400019407, 0.300064683, 0.302783698},
        {0.16938819, 0.363896906, 0.440417856}};

   //array com a incidência dos 459 elementos do tipo tetraedro (4 nós)
   static vtkIdType pts[459][4]={
        {165,3,153,52},
        {165,153,3,152},
        {165,49,166,167},
        ...
        {169,113,35,37},
        {170,162,158,111},
        {174,114,115,34},
        {143,10,9,40}};

   //array com os valores escalares (tensão) relacionados com os nós
   static float strX[181]={
        -158.75,
        202.25,
        16.994,
        ...
        -37.392,
        -77.227,
        95.541,
        107.84};

   vtkPoints *pontos = vtkPoints::New(); //lista de pontos
   vtkFloatArray *escalares = vtkFloatArray::New();//lista de valores float
```

```
for (i=0; i<181; i++) {
   pontos->InsertPoint(i, x[i]); //insere cada ponto na lista
}

//malha não-estruturada
vtkUnstructuredGrid *malha = vtkUnstructuredGrid::New();
malha->SetPoints(points); //define os pontos da malha

for (i = 0;i <181; i++) {
   escalares->InsertValue(i, strX[i]);//define os valores escalares
}

//insere os tetraedros na malha. O 10 é o tipo tetraedro, 4 é o
//número de pontos de cada célula
for (i = 0;i <459; i++) malha->InsertNextCell(10, 4, pts[i]);

malha->GetPointData()->SetScalars(escalares);//define os velores
                                               escalares da malha

pontos->Delete();//apaga a lista de pontos (já inserida na malha)
escalares->Delete();//apaga a lista de escalares (já inserida na malha)

//mapper
vtkDataSetMapper *mapper = vtkDataSetMapper::New();
mapper->SetInput(malha);

//tabela de cores
vtkLookupTable *stc = vtkLookupTable::New();
stc->SetRange(data->GetScalarRange()); //valores máx. e mín. da tabela
stc->SetHueRange (.6667, .0);//variação de cores da tabela (do azul ao vermelho)
stc->SetNumberOfTableValues(1024);//número de cores da tabela
stc->Build();//construção da tabela

mapper->SetLookupTable(stc); //define a tabela criada para o mapper
mapper->SetScalarRange(malha->GetScalarRange());//define a variação
               //dos valores do mapper de acordo com os valores de entrada

//atores para cada mapper
vtkActor *ator = vtkActor::New();
ator->SetMapper(mapper);

//câmera
vtkCamera *camera = vtkCamera::New();
camera->SetPosition(1,1,1);
camera->SetFocalPoint(0,0,0);

// renderer
vtkRenderer *renderer = vtkRenderer::New();

// RendereWindow
vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin->AddRenderer(renderer);
renWin->SetSize(800,600); //tamanho da janela

// RenderWindowInteractor com os eventos padrão
vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
iren->SetRenderWindow(renWin);
```

```
    //adiciona o ator ao renderer e define outras propriedades
    renderer->AddActor(ator);
    renderer->SetActiveCamera(camera);
    renderer->ResetCamera();
    renderer->SetBackground(1,1,1);

    //Renderiza
    renWin->Render();

    iren->Start(); //inicia o interactor

    //apaga objetos criados
    malha->Delete();
    stc->Delete();
    ator->Delete();
    camera->Delete();
    renderer->Delete();
    renWin->Delete();
    iren->Delete();

    return 0;
}
```

APÊNDICE 4 – CRIAÇÃO DE ARQUIVOS DO TIPO VTK

```
/*============================================================================
ANSYS2VTK.cxx
Programa de geração de arquivos do tipo VTK, incluindo estrura de dados.
============================================================================*/
#include <iostream>
#include <tchar.h>
#include <fstream>
#include <conio.h>
#include "No.h"
#include "Tetra.h"
#include <stdio.h>
#include <stdlib.h>
using namespace std;

No *no, *nInsere, *noSuperficie, *noSInsere;
Tetra *tetra, *tInsere;
double x, y, z;
int i, j, n, np = 0, nc = 0, n1, n2, n3, n4, n5, n6, n7, n8, nada;
int nNos = 0, nTetra = 0;
char *cab1, *titulo, *unst, *fileNos, *fileElementos, *fileTensao, *fileDesloc,
*fileVTK, *hdr, *hdr2, *hdr3, *hdr4;

void criaArquivoVTK();
int leNosAnsys();
int leElementosAnsys();
int leDeslocAnsys();
int leTensaoAnsys();

int main(int argc, _TCHAR* argv[]){

      cab1 = "# vtk DataFile Version 3.0\n";
      titulo = "Teste de geração de arquivo\n";
      unst = "DATASET UNSTRUCTURED_GRID\n";
      fileNos = "nlist.lis";
      fileElementos = "elist.lis";
      fileTensao = "tensao.lis";
      fileDesloc = "desloc.lis";
      fileVTK = "vtk.vtk";
      hdr = "   NODE        X                   Y                   Z";
      hdr2 = "    ELEM MAT TYP REL ESY SEC        NODES";
      hdr3 = "    NODE     SX          SY          SZ          SXY         SYZ
SXZ";
      hdr4 = "    NODE      UX          UY          UZ";


      leNosAnsys();
      leElementosAnsys();
      leDeslocAnsys();
      leTensaoAnsys();

      criaArquivoVTK();

      return 0;

}

void criaArquivoVTK(){

      ofstream saida("vtk.vtk");

      saida << cab1 << titulo << "ASCII\n\n" << unst << "POINTS" << ' ' << nNos <<
            ' ' << "float" << '\n';

      for (i = 1; i <= nNos; i++){
            saida << no->x << ' ' << no->y << ' ' << no->z << '\n';
```

```
            no = no->proximo; }

        saida << "\nCELLS " << nTetra << ' ' << 4*nTetra + nTetra << '\n';

        for (i = 1; i <= nTetra; i++){
            saida << '4' << ' ' << tetra->n1 - 1 << ' ' << tetra->n2 - 1 << ' ' <<
                tetra->n3 - 1 << ' ' << tetra->n4 - 1 << '\n';
            tetra = tetra->proximo; }

        saida << "\nCELL_TYPES " << nTetra;

        for (i = 1; i <= nTetra; i++)saida << "\n10";

        saida << "\n\nPOINT_DATA " << nNos << '\n';
        saida << "SCALARS escalares float\n";
        saida << "LOOKUP_TABLE default";

        for (i = 1; i <= nNos; i++){
            saida << '\n' << no->tensaoX;
            no = no->proximo; }

        saida << "\nVECTORS vetores float";
        for (i = 1; i <= nNos; i++){
            saída <<'\n'<< no->tensaoX << ' ' << no->tensaoY << ' '<< no->tensaoZ;
            no = no->proximo; }

        cout << "Arquivo .vtk gerado\n";

        saida.close();
}

int leNosAnsys(){

        ifstream entrada(fileNos);
        if(!entrada) {
            cout << "\nO arquivo com os nós não existe ou foi danificado\n";
            return 1; }

        char linha[256] = "";

        while(strncmp(linha, hdr, strlen(hdr)) && !entrada.eof())
            entrada.getline(linha, 255, (char) '\n');

        if(entrada.eof()) return 0;

        no = NULL;

        while(!entrada.eof()){

            while(!entrada.eof() && !entrada.fail()){
                streampos loc = entrada.tellg();
                entrada >> n >> x >> y >> z;
                if(entrada.fail()){
                    entrada.seekg(loc);
                    entrada.clear();
                    entrada.getline(linha, 255, (char) '\n');
                    break; }
                if (no == NULL) {
                    no = inicializaNo();
                    no = preencheNo(no, n, n, x, y, z);
                    nNos = 1; }
                else{
                    nInsere = inicializaNo();
                    nInsere = preencheNo(nInsere, n, n, x, y, z);
                    no = insereFimNo(no, nInsere);
                    nNos++;     }
```

```
                }
        }

        entrada.close();
        return 0;
}

int leElementosAnsys(){

        ifstream entrada2(fileElementos);
        if(!entrada2) {
                cout << "\nO arquivo com os elementos não existe ou foi danificado\n";
                getchar();
                return 1; }

        char linha[256] = "";

        while(strncmp(linha, hdr2, strlen(hdr2)) && !entrada2.eof())
                entrada2.getline(linha, 255, (char) '\n');

        if(entrada2.eof()) return 0;

        tetra = NULL;

        while(!entrada2.eof()){

                while(!entrada2.eof() && !entrada2.fail()){
                        streampos loc = entrada2.tellg();
                        entrada2 >> n >> nada >> nada >> nada >> nada >> nada >> n1 >>
                                  n2 >> n3 >> n4 >> n5 >> n6 >> n7 >> n8;
                        if(entrada2.fail()){
                                entrada2.seekg(loc);
                                entrada2.clear();
                                entrada2.getline(linha, 255, (char) '\n');
                                break; }
                        if (tetra == NULL) {
                                tetra = inicializaTetra();
                                tetra = preenche(tetra, n, n, n1, n2, n3, n5);
                                nTetra = 1; }
                        else{
                                tInsere = inicializaTetra();
                                tInsere = preenche(tInsere, n, n, n1, n2, n3, n5);
                                tetra = insereFim(tetra, tInsere);
                                nTetra++; }
                }
        }
        entrada2.close();
        return 0;
}

int leDeslocAnsys(){

        ifstream entrada4(fileDesloc);
        if(!entrada4) {
                cout << "\nO arquivo com os desloc. não existe ou foi danificado\n";
                return 1; }

        char linha[256] = "";

        while(strncmp(linha, hdr4, strlen(hdr4)) && !entrada4.eof())
                entrada4.getline(linha, 255, (char) '\n');

        if(entrada4.eof()) return 0;

        double dX = .0, dY = .0, dZ = .0;
        No *noMe = no;
```

```
	while(!entrada4.eof()){

		while(!entrada4.eof() && !entrada4.fail()){
			streampos loc = entrada4.tellg();
			entrada4 >> n >> dX >> dY >> dZ;
			if(entrada4.fail()){
				entrada4.seekg(loc);
				entrada4.clear();
				entrada4.getline(linha, 255, (char) '\n');
				break; }
			while(n != noMe->nANSYS) noMe = noMe->proximo;
			noMe->deslocX = dX;
			noMe->deslocY = dY;
			noMe->deslocZ = dZ;
			noMe = noMe->proximo;	}

	}

	entrada4.close();
	return 0;
}

int leTensaoAnsys(){

	ifstream entrada3(fileTensao);
	if(!entrada3) {
		cout << "\nO arquivo com as tensões não existe ou foi danificado\n";
		return 1;
	}

	char linha[256] = "";

	while(strncmp(linha, hdr3, strlen(hdr3)) && !entrada3.eof())
		entrada3.getline(linha, 255, (char) '\n');

	if(entrada3.eof()) return 0;

	double tX = .0, tY=.0, tZ=.0, tXY=.0, tYZ=.0, tXZ=.0;
	No *noMe = no;

	while(!entrada3.eof()){

		while(!entrada3.eof() && !entrada3.fail()){
			streampos loc = entrada3.tellg();
			entrada3 >> n >> tX >> tY >> tZ >> tXY >> tYZ >> tXZ;
			if(entrada3.fail()){
				entrada3.seekg(loc);
				entrada3.clear();
				entrada3.getline(linha, 255, (char) '\n');
				break; }
			while(n != noMe->nANSYS)
				noMe = noMe->proximo;
			noMe->tensaoX = tX;
			noMe->tensaoY = tY;
			noMe->tensaoZ = tZ;
			noMe->tensaoXY = tXY;
			noMe->tensaoYZ = tYZ;
			noMe->tensaoXZ = tXZ;
			noMe = noMe->proximo;	}
	}
	entrada3.close();
	return 0;
}
```

```
//ESTRUTURA DE DADOS

//No.h
struct No{
	static unsigned short nNo;
	unsigned short	nANSYS;		//identificador do vértice
	unsigned short	nVTK;
	double		x, y, z;		//coordenadas do vértice
	No		*proximo;		//ponteiro para o próximo vertice
	No		*anterior;		//ponteiro para o vertice seguinte
	bool		usado;
	double		tensaoX, tensaoY, tensaoZ, tensaoXY, tensaoYZ, tensaoXZ;
	double		deslocX, deslocY, deslocZ;
	bool		superficie;

};
No* inicializaNo(void);
No* insereNo(No*, No* nInserido);
No* insereFimNo(No*, No* nInserido);
No* preencheNo(No* no, unsigned short nA, unsigned short nV, double X, double Y,
		double Z);
No* retornaNo(No*, unsigned short);
No* copiaDados(No* para, No* de);
void numeraNo(No*, unsigned short);

//No.cpp
#include ".\no.h"
No* inicializaNo(void){
	No* no = (No*) malloc(sizeof(No));
	no->anterior = no;
	no->proximo = no;
	no->nANSYS = NULL;
	no->nVTK = NULL;
	no->x = NULL;
	no->y = NULL;
	no->z = NULL;
	no->usado = false;
	no->tensaoX = .0;
	no->tensaoY = .0;
	no->tensaoZ = .0;
	no->tensaoXY = .0;
	no->tensaoYZ = .0;
	no->tensaoXZ = .0;
	no->deslocX = .0;
	no->deslocY = .0;
	no->deslocZ = .0;
	no->superficie = false;
	return no;
}
No* insereFimNo(No* no, No* nInserido){
	No *nAnterior;
	nAnterior = no->anterior;
	no->anterior = nInserido;
	nAnterior->proximo = nInserido;
	nInserido->anterior = nAnterior;
	nInserido->proximo = no;
	return no;}
No* insereNo(No* no, No* nInserido){
	no = no->proximo;
	return no;
}
No* preencheNo(No* no, unsigned short nA, unsigned short nV, double dX, double dY,
		double dZ){
	no->nANSYS = nA;
	no->nVTK = nV;
	no->x = dX;
```

```
        no->y = dY;
        no->z = dZ;
        return no;
}
No* retornaNo(No* no, unsigned short n){
        if (no->nANSYS == n) return no;
        No *retorna;
        No* primeiro = no;
        no = no->proximo;
        while(no != primeiro){
                if (no->nANSYS == n){
                        retorna = no;
                        no = primeiro;
                        return retorna;}
                no = no->proximo;
        }
        return NULL;
}
No* copiaDados(No* para, No* de){
        para->x = de->x;
        para->y = de->y;
        para->z = de->z;
        para->nANSYS = de->nANSYS;
        para->nVTK = de->nVTK;
        para->deslocX = de->deslocX;
        para->deslocY = de->deslocY;
        para->deslocZ = de->deslocZ;
        para->tensaoX = de->tensaoX;
        para->tensaoY = de->tensaoY;
        para->tensaoZ = de->tensaoZ;
        para->tensaoXY = de->tensaoXY;
        para->tensaoYZ = de->tensaoYZ;
        para->tensaoXZ = de->tensaoXZ;
        return para;
}

void numeraNo(No* no, unsigned short n){
        no->nANSYS = n;
}


//Tetra.h
#include "No.h"
struct Tetra{
        static unsigned short nTetra;
        unsigned short      nANSYS;             //identificador do vértice
        unsigned short      nVTK;
        unsigned short      n1, n2, n3, n4;     //coordenadas do vértice
        Tetra         *proximo;                 //ponteiro para o próximo vertice
        Tetra         *anterior;                //ponteiro para o vertice seguinte
};

Tetra* inicializaTetra(void);
Tetra* insere(Tetra*, Tetra* nInserido);
Tetra* insereFim(Tetra*, Tetra* nInserido);
Tetra* preenche(Tetra*, unsigned short, unsigned short, unsigned short, unsigned
                short, unsigned short, unsigned short);

//Tetra.cpp
#include ".\tetra.h"
Tetra* inicializaTetra(void){
        Tetra* tetra = (Tetra*) malloc(sizeof(Tetra));
        tetra->anterior = tetra;
        tetra->proximo = tetra;
        tetra->nANSYS = NULL;
        tetra->nVTK = NULL;
```

```
        tetra->n1 = NULL;
        tetra->n2 = NULL;
        tetra->n3 = NULL;
        tetra->n4 = NULL;
        return tetra;
}
Tetra* insereFim(Tetra* tetra, Tetra* nInserido){
        Tetra *nAnterior;
        nAnterior = tetra->anterior;
        tetra->anterior = nInserido;
        nAnterior->proximo = nInserido;
        nInserido->anterior = nAnterior;
        nInserido->proximo = tetra;
        return tetra;
}
Tetra* insere(Tetra* tetra, Tetra* nInserido){
        return NULL;
}
Tetra* preenche(Tetra* tetra, unsigned short nA, unsigned short nV, unsigned short
                N1, unsigned short N2, unsigned short N3, unsigned short N4){
        tetra->nANSYS = nA;
        tetra->nVTK = nV;
        tetra->n1 = N1;
        tetra->n2 = N2;
        tetra->n3 = N3;
        tetra->n4 = N4;
        return tetra;
}
```

APÊNDICE 5 – ALGORITMO PARA CRIAÇÃO DE ISOSUPERFÍCIES

```
/*=============================================================================
isosuperf.cxx
Programa para criação de isosuperfícies. Utiliza transparências e barra de
cores.
=============================================================================*/

#include "vtkActor.h"
#include "vtkCamera.h"
#include "vtkPolyData.h"
#include "vtkPolyDataMapper.h"
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkRenderer.h"
#include "vtkUnstructuredGrid.h"
#include "vtkContourGrid.h"
#include "vtkDataSetMapper.h"
#include "vtkLookupTable.h"
#include "vtkProperty.h"
#include "vtkScalarBarActor.h"
#include "vtkTextProperty.h"
#include "vtkUnstructuredGridWriter.h"
#include "vtkUnstructuredGridReader.h"
#include "vtkPolyDataNormals.h"
#include "vtkStripper.h"

int main( int argc, char *argv[] ){

  int i;

  //Malha não estruturada
  vtkUnstructuredGrid *dados = vtkUnstructuredGrid::New();

  //Leitor de arquivo VTK
  vtkUnstructuredGridReader *reader = vtkUnstructuredGridReader::New();
  reader ->SetFileName("arquivo.vtk");
  reader ->OpenVTKFile();
  dados = reader->GetOutput();
  reader ->CloseVTKFile();

  //Gravador de arquivo VTK
  vtkUnstructuredGridWriter *writer = vtkUnstructuredGridWriter::New();
  writer->SetFileTypeToASCII();
  writer->SetInput(dados);
  writer->SetFileName("teste.vtk");
  writer->Write();

  vtkDataSetMapper *mapper = vtkDataSetMapper::New();
  vtkDataSetMapper *mapperTransparente = vtkDataSetMapper::New();

  //Filtro de isosuperfícies
  vtkContourGrid *cGrid = vtkContourGrid::New();
  cGrid->SetInput(dados);
  double *range;
  range = dados->GetScalarRange();
  cGrid->GenerateValues(7, range[0]+2, range[1]-2);
  cGrid->ComputeNormalsOn();
  cGrid->ComputeGradientsOn();
```

```
//Suavização das superfícies
vtkPolyDataNormals *suav = vtkPolyDataNormals::New();
suav->SetInput(cGrid->GetOutput());
suav->SetFeatureAngle(172.0);

vtkStripper *striper = vtkStripper::New();
striper->SetInput(suav->GetOutput());

mapper->SetInput(striper->GetOutput());//Poderia usar vtkPolyDataMapper
mapper->SetScalarRange(dados->GetScalarRange());
mapperTransparente->SetInput(dados);
mapperTransparente->ScalarVisibilityOff();

vtkLookupTable *stc = vtkLookupTable::New();
stc->SetRange(dados->GetScalarRange());
stc->SetHueRange (.6667, .0);
stc->SetNumberOfTableValues(5);
stc->Build();

mapper->SetLookupTable(stc);

vtkScalarBarActor *barra = vtkScalarBarActor::New();
barra->vtkScalarBarActor::SetLookupTable(mapper->GetLookupTable());
barra->SetNumberOfLabels(5);
barra->SetOrientationToVertical();
barra->SetTitle("SX (N/m2)");

vtkTextProperty *texto = vtkTextProperty::New();
texto->SetFontFamilyToArial();
texto->SetColor(.0,.0,.0);
texto->AntiAliasingOn();

vtkTextProperty *textoTitulo = vtkTextProperty::New();
texto->SetFontFamily(VTK_ARIAL);
textoTitulo->SetColor(.1,.1,.2);
textoTitulo->BoldOn();
textoTitulo->ShadowOn();
textoTitulo->AntiAliasingOn();

barra->SetLabelTextProperty(texto);
barra->SetTitleTextProperty(textoTitulo);

vtkActor *ator = vtkActor::New();
ator->SetMapper(mapper);
ator->GetProperty()->SetInterpolationToPhong();

vtkActor *atorTransparente = vtkActor::New();
atorTransparente->SetMapper(mapperTransparente);
atorTransparente->GetProperty()->SetInterpolationToPhong();
atorTransparente->GetProperty()->SetOpacity(0.28);
atorTransparente->GetProperty()->SetColor(.2,.6,.74);

vtkCamera *camera = vtkCamera::New();
camera->SetPosition(-8,-8,8);
camera->SetViewUp(0, 0, 1);
camera->SetFocalPoint(0,0,0);
camera->ParallelProjectionOn();

vtkRenderer *renderer = vtkRenderer::New();
```

```
  vtkRenderWindow *renWin = vtkRenderWindow::New();
  renWin->AddRenderer(renderer);

  vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
  iren->SetRenderWindow(renWin);

  renderer->AddActor(ator);
  renderer->AddActor(atorTransparente);
  renderer->AddActor(barra);
  renderer->SetBackground(1.,1.,1.);
  renderer->SetActiveCamera(camera);
  renderer->ResetCamera();
  camera->Zoom(1.8);

  renWin->SetSize(800,600);
  renWin->Render();

  iren->Initialize();
  iren->Start();

  ator->Delete();
  atorTransparente->Delete();
  dados->Delete();
  mapper->Delete();
  mapperTransparente->Delete();
  stc->Delete();
  cGrid->Delete();
  barra->Delete();
  texto->Delete();
  textoTitulo->Delete();
  reader->Delete();
  writer->Delete();
  camera->Delete();
  renderer->Delete();
  renWin->Delete();
  iren->Delete();

  return 0;
}
```

APÊNDICE 6 – ALGORITMO COM PLANOS DE CORTE, ANIMAÇÃO E EXTRAÇÃO DE FATIAS

```
/*==========================================================================
anim.cxx
Programa que cria planos de corte e extração de fatias, e realiza animação
dos planos.
==========================================================================*/

#include "vtkActor.h"
#include "vtkCamera.h"
#include "vtkPolyDataMapper.h"
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkRenderer.h"
#include "vtkUnstructuredGrid.h"
#include "vtkDataSetMapper.h"
#include "vtkPlane.h"
#include "vtkClipDataSet.h"
#include <math.h>
#include "vtkLookupTable.h"
#include "vtkAxes.h"
#include "vtkLogLookupTable.h"
#include "vtkScalarBarActor.h"
#include "vtkUnstructuredGridWriter.h"
#include "vtkUnstructuredGridReader.h"
#include "vtkCutter.h"
#include "vtkProperty.h"


int main( int argc, char *argv[] ){

  int i;

  vtkUnstructuredGrid *dados = vtkUnstructuredGrid::New();

  vtkUnstructuredGridReader *reader = vtkUnstructuredGridReader::New();
  reader->SetFileName("arquivo.vtk");
  reader->OpenVTKFile();
  dados = reader->GetOutput();
  reader->CloseVTKFile();

  vtkDataSetMapper *mapper = vtkDataSetMapper::New();
  vtkDataSetMapper *mapperTransparente = vtkDataSetMapper::New();

  vtkPolyDataMapper *mapperFatia = vtkPolyDataMapper::New();

  vtkClipDataSet *corte = vtkClipDataSet::New();

  vtkCutter *fatia = vtkCutter::New();

  vtkPlane *plano = vtkPlane::New();
  plano->SetNormal(1,0,0);
  plano->SetOrigin(1.5,0,0);

  corte->SetInput(dados);
  corte->SetClipFunction(plano);

  fatia->SetInput(dados);
  fatia->SetCutFunction(plano);

  mapper->SetInput(corte->GetOutput());
```

```
mapperFatia->SetInput(fatia->GetOutput());
mapperTransparente->SetInput(dados);
mapperTransparente->ScalarVisibilityOff();

vtkLookupTable *stc = vtkLookupTable::New();
stc->SetRange(dados->GetScalarRange());
stc->SetHueRange (.6667, .0);
stc->SetNumberOfTableValues(1024);
stc->Build();

mapper->SetLookupTable(stc);
mapper->SetScalarRange(dados->GetScalarRange());
mapperFatia->SetLookupTable(stc);
mapperFatia->SetScalarRange(dados->GetScalarRange());

vtkScalarBarActor *barra = vtkScalarBarActor::New();
barra->vtkScalarBarActor::SetLookupTable(mapper->GetLookupTable());
barra->SetNumberOfLabels(5);
barra->SetOrientationToVertical();
barra->SetTitle("SX (N/m2)");

vtkActor *ator = vtkActor::New();
ator->SetMapper(mapper);
ator->GetProperty()->SetInterpolationToPhong();

vtkActor *atorTransparente = vtkActor::New();
atorTransparente->SetMapper(mapperTransparente);
atorTransparente->GetProperty()->SetOpacity(0.3);
atorTransparente->GetProperty()->SetColor(.2,.6,.75);

vtkCamera *camera = vtkCamera::New();
camera->SetPosition(-8,-8,8);
camera->SetViewUp(0, 0, 1);
camera->SetFocalPoint(0,0,0);

vtkRenderer *renderer = vtkRenderer::New();

vtkRenderWindow *renWin = vtkRenderWindow::New();
renWin->AddRenderer(renderer);
renWin->SetSize(800,600);

vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
iren->SetRenderWindow(renWin);

renderer->AddActor(ator);
renderer->AddActor(atorTransparente);
renderer->SetActiveCamera(camera);
renderer->ResetCamera();
camera->Zoom(1.8);
renderer->SetBackground(1.,1.,1.);
renderer->AddActor(atorEixo);
renderer->AddActor2D(barra);

renWin->Render();

for(double di=0; di<4; di+=.01){
    plano->SetOrigin(di,0,0);
    renWin->Render();
    for(int j=0; j<= 1000000; j++);
}
```

```
  ator->SetMapper(mapperFatia);
  plano->SetNormal(0,1,0);
  for(double di=0; di<2; di+=.01){
      plano->SetOrigin(0,di,0);
      renWin->Render();
      for(int j=0; j<= 10000000; j++);
  }

  renWin->Render();

  iren->Start();

  ator->Delete();
  atorTransparente->Delete();
  barra->Delete();
  atorEixo->Delete();
  camera->Delete();
  renderer->Delete();
  renWin->Delete();
  iren->Delete();

  return 0;
}
```

APÊNDICE 7 – ALGORITMO DE ÍCONES PONTUAIS

```
/*===========================================================================
glifos.cxx
Programa para desenho de ícones pontuais.
===========================================================================*/
#include "vtkUnstructuredGrid.h"
#include "vtkUnstructuredGridReader.h"
#include "vtkUnstructuredGridWriter.h"
#include "vtkConeSource.h"
#include "vtkGlyph3D.h"
#include "vtkPolyDataMapper.h"
#include "vtkActor.h"
#include "vtkRenderer.h"
#include "vtkRenderWindow.h"
#include "vtkRenderWindowInteractor.h"
#include "vtkLookupTable.h"
#include "vtkDataSetMapper.h"
#include "vtkProperty.h"
#include "vtkScalarBarActor.h"
#include "vtkTextProperty.h"

int main( int argc, char *argv[] ){

  int i;

  vtkUnstructuredGrid *dados = vtkUnstructuredGrid::New();

  vtkUnstructuredGridReader *reader = vtkUnstructuredGridReader::New();
  reader->SetFileName("arquivo.vtk");
  reader->OpenVTKFile();
  dados = reader->GetOutput();
  reader->CloseVTKFile();

  vtkUnstructuredGridWriter *writer = vtkUnstructuredGridWriter::New();
  writer->SetFileTypeToASCII();
  writer->SetInput(dados);
  writer->SetFileName("teste.vtk");
  writer->Write();

  vtkLookupTable *stc = vtkLookupTable::New();
  stc->SetRange(dados->GetScalarRange());
  stc->SetHueRange (.6667, .0);
  stc->SetNumberOfTableValues(1024);
  stc->Build();

  vtkConeSource *cone = vtkConeSource::New();
  cone->SetResolution(6);

  vtkGlyph3D *glyph = vtkGlyph3D::New();
  glyph->SetInput(dados);
  glyph->SetSource(cone->GetOutput());
  glyph->SetScaleModeToScaleByVector();
  //glyph->SetScaleModeToDataScalingOff();
  glyph->SetScaleFactor(6);
  glyph->SetColorModeToColorByVector();

  vtkPolyDataMapper *spikeMapper = vtkPolyDataMapper::New();
  spikeMapper->SetInput(glyph->GetOutput());
  spikeMapper->vtkPolyDataMapper::SetLookupTable(stc);
  spikeMapper->SetScalarRange(dados->GetScalarRange());
```

```
  vtkDataSetMapper *mapper = vtkDataSetMapper::New();
  mapper->SetInput(dados);
  mapper->SetLookupTable(stc);
  mapper->SetScalarRange(dados->GetScalarRange());

  vtkDataSetMapper *mapperTransparente = vtkDataSetMapper::New();
  mapperTransparente->SetLookupTable(stc);
  mapperTransparente->SetInput(dados);
  mapperTransparente->SetScalarRange(dados->GetScalarRange());

  vtkActor *spikeActor = vtkActor::New();
  spikeActor->SetMapper(spikeMapper);

  vtkActor *ator = vtkActor::New();
  ator->SetMapper(mapper);

  vtkActor *atorTransparente = vtkActor::New();
  atorTransparente->SetMapper(mapperTransparente);
  atorTransparente->GetProperty()->SetOpacity(0.14);

  vtkRenderer *renderer = vtkRenderer::New();
  renderer->AddActor(spikeActor);
  renderer->AddActor(atorTransparente);
  renderer->AddActor(ator);
  renderer->SetBackground(1, 1, 1);

  vtkRenderWindow *renWin = vtkRenderWindow::New();
  renWin->AddRenderer(renderer);
  renWin->SetSize(600, 600);

  vtkRenderWindowInteractor *iren = vtkRenderWindowInteractor::New();
  iren->SetRenderWindow(renWin);

  renWin->Render();
  iren->Start();

}
```